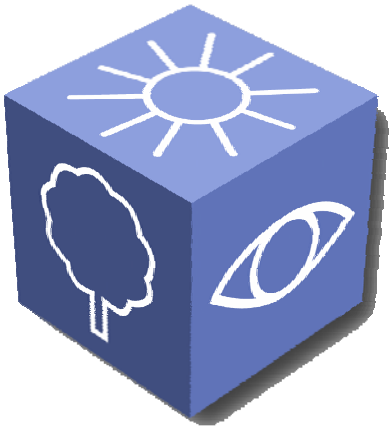


GAMETOOLS

ADVANCED TOOLS FOR DEVELOPING HIGHLY REALISTIC COMPUTER GAMES

REPORT ON ILLUMINATION ALGORITHMS

Document identifier:	**GameTools-5-D5.2-02-1-1-Report on Illumination Algorithms**
Date: (use "update field" Word function, right mouse button)	**15/09/2005**
Work package:	**WP05: Illumination**
Partner(s):	**BUTE, UdG, Unilim**
Leading Partner:	**BUTE**
Document status:	**APPROVED**
Deliverable identifier:	**D5.2**

Abstract: This technical report describes the different algorithms used on the implementation of the Illumination module.

Delivery Slip

	Name	Partner	Date	Signature
From	László Szirmay-Kalos	BUTE	09-09-2005	
Reviewed by	Moderator and reviewers	ALL		
Approved by	Moderator and reviewers	ALL		

Document Log

Issue	Date	Comment	Author
1-0	24-08-2005	First draft	László Szirmay-Kalos
1-1	09-09-2005	Final version	László Szirmay-Kalos

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files / URL
Word	gametools-ist-2-004363-5-d5.2-02-1-1-report on illumination algorithms.doc (use “update field” Word function)

Contents

Chapter 1

Introduction

The ultimate objective of *rendering* is to provide the user with the illusion of watching real objects on the computer screen. To provide the illusion of watching the real world, the color sensation of an observer looking at the artificial image generated by the graphics system must be approximately equivalent to the color perception which would be obtained in the real world. The color perception of humans depends on the *light power* reaching the eye from a given direction and on the operation of the eye. The power is determined from the *radiance* of the visible points. The radiance depends on the shape and optical properties of the objects and on the intensity of the light sources. To find these radiance values, rendering algorithms should identify the visible points, and then compute the radiance of these points. Radiance or color computation of points is the main task of the *illumination workpackage*.

Illumination calculation determines the light reflected off the surface points at different directions. Since light is an electromagnetic wave, light distribution in a point and at a given direction can be represented by a wavelength-dependent function. Rendering algorithms usually evaluate this functions at a few representative wavelengths. On a given wavelength the intensity of the light is described by the *radiance*. In scenes not incorporating *participating media* it is enough to calculate the radiance at surface points. The radiance reflected off a surface point is affected by the emission of this point (*lighting*), the illumination provided by other surface points and the optical properties of the material at this point (*material properties*). Formally this dependence is characterized by the *rendering equation*. The *rendering equation*[59] expresses the *radiance* $L(\vec{x},\vec{\omega})$ $[W \cdot m^{-2} \cdot sr^{-1}]$ of a surface point $\vec{x}$ in direction $\vec{\omega}$, and has the following form:

$$L = L^e + \mathcal{T}L. \tag{1.1}$$

If only direct contribution is considered, then $L = L^e$. The light-surface interaction is described by integral operator $\mathcal{T}$, which has the following form

$$(\mathcal{T}L)(\vec{x},\vec{\omega}) = \int_{\Omega} L(h(\vec{x},-\vec{\omega}'),\vec{\omega}') \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \; d\omega' \tag{1.2}$$

where $L(\vec{x},\vec{\omega})$ and $L^e(\vec{x},\vec{\omega})$ are the radiance and emission of the surface in point $\vec{x}$ at direction $\vec{\omega}$, Ω is the directional sphere, $h(\vec{x},\vec{\omega}')$ is the visibility function defining the point that is visible from point $\vec{x}$ at direction $\vec{\omega}'$, $f_r(\vec{\omega}',\vec{x},\vec{\omega})$ is the bi-directional reflection/refraction function, and θ' is the angle between the surface normal and the incoming direction $-\vec{\omega}'$ (figure 1.1).

Rendering computes the radiance of points visible from the eye by integrating their reflected (or refracted) illumination. The evaluation of these integrals is rather time consuming and requires incoherent visibility queries that do not allow the optimal exploitation of the current graphics processing units (GPU). On the other hand, games should render dynamic characters moving in very complex environments at least 20 times per second. To meet this performance requirement, a part of the computation should be executed before the actual game is started, and we should apply simplifications that significantly reduce the rendering time without destroying the image quality.

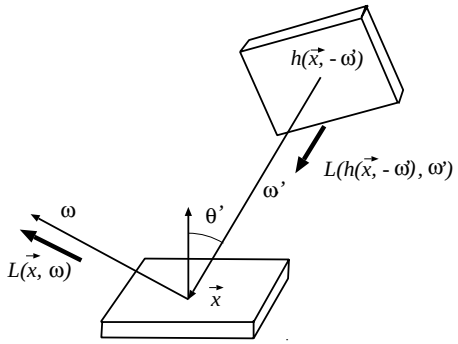


Figure 1.1: Geometry of the rendering equation

Taking into account that pre-computation works well for static geometry (and especially for diffuse surfaces), static and dynamic objects are usually separated (figure 1.2). The self illumination of the static scene is usually rendered in a pre-processing phase by a non real-time global illumination algorithm (e.g. radiosity). On the other hand, the illumination onto dynamic objects, as well as the illumination reflected by these dynamic objects onto static ones are determined run time. The available rendering time makes simplifications necessary when the light transport is computed between dynamic and static objects. The effect of dynamic objects onto static ones are handled by shadow mapping and caustic computations, that is, only the direct lighting and high-frequency indirect lighting of static objects is modulated by the presence of dynamic ones. Since dynamic objects are usually much smaller than the static environment, this is usually acceptable. However, because of these typical size differences, the indirect illumination from static objects to dynamic ones must be fully computed on sufficient accuracy.

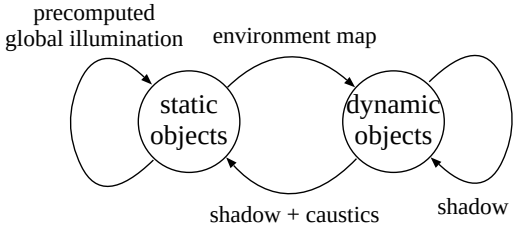


Figure 1.2: Simplified light transport

1.1 Structure of illumination tools

The illumination workpackage is organized around the simplified light transport model (figure 1.2) and include different algorithms, called tools of different properties and application areas. These tools complement each other. In a particular game, a subset of these tools can be incorporated depending on the properties and the required illumination properties.

In order to meet the very high speed requirements of games (at least 20-30 frames per second), all tools have two parts, an *illumination info generation* part and a *final gathering* part (figure 1.3). The illumination info generation part is responsible for precomputing illumination information and storing it in a data structure called *illumination info*, from which the final gathering part produces the image for the particular camera. To produce continuous animation, the final gathering part should run on high frame rates. Since the scene may change, this illumination info generation should also be repeated at a sufficient frequency, but this frequency can be significantly lower

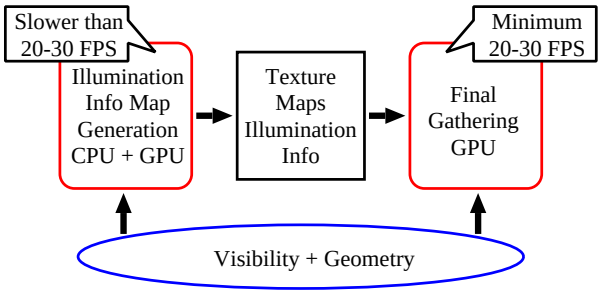


Figure 1.3: Structure of illumination tools

than the frame rate.

The very high performance requirements can only be met if the final gathering part is executed by the GPU, which requires the illumination info to be stored in the texture memory. Note, however, that the illumination info is not necessarily a conventional texture, but can also mean geometric or shadowing data, compressed light paths, etc. On the other hand, the illumination info generation part can be executed both the CPU and the GPU depending on its features.

The subsequent chapters discuss illumination tools one by one. The tools belong to five main categories, the *GPU ray-tracing*, the *global illumination*, the *image based rendering*, the *participating media*, and to *material models*. The reason of presenting more than one tools per category is to allow a later selection based on the results of the implementation.

The first category of tools solve the ray-tracing problem on the GPU. The *approximate ray-tracing tool* of chapter 2 proposes a general technique to trace incoherent rays, i.e. rays not sharing the same origin, on the GPU very efficiently. Then, this general technique is applied in different tools providing *localized reflections*, *localized multiple refractions*, *caustics*, and *soft shadows*. In chapter 3 this concept is used for diffuse and specular reflections as well. Chapter 4 presents a GPU based brute force ray tracer. The *image based lighting tool* of chapter 5 computes the reflection and self-shadowing due to a high-dynamic range environment map.

Chapters 6, 7, 8, and 9 discuss global illumination algorithms made suitable for real-time applications. Since these algorithms are not fast enough to solve the global illumination problem from scratch in real-time, they obtain slowly changing textures, that can be mapped onto objects by the final gathering on high frame rates. The *photon map filtering tool* is a GPU version of the photon mapping algorithm. The *stochastic iteration tool* runs a stochastic iteration algorithm to find the global radiance distribution. The *obscurances tool* applies the obscurances model to simplify the radiance transport problem. Finally, the *precomputed light paths tool* precomputes light paths, which are modulated by the actual lighting and camera properties.

The *Fresnel reflection tool* of chapter 10 presents a simple Fresnel reflection approximation and a physically plausible BRDF model that can characterize both dielectric materials and metals. These material models can be incorporated into the final gathering operation.

Chapters 11, 12, and 13 address the rendering of *participating media*, a very important class of natural phenomena. The *spherical billboard tool* of chapter 11 solves the annoying billboard clipping and popping artifacts showing up when participating media is modeled by a particle system. The *cloudy natural phenomena tool* of chapter 12 introduces a hierarchical particle representation, which enables us to compute multiple forward scattering effects in dynamically evolving clouds. The *participating media illumination network tool* of chapter 13, on the other hand, can render arbitrary scattering in real-time, but assumes that the volume is static.

Finally, the *image based rendering tool* of chapter 14 uses images, i.e. billboard clouds to display trees and forests keeping parallax effects.

Chapter 2

Approximate Ray-Tracing Tool

This tool presents a fast approximation method to obtain the point hit by a reflection or refraction ray. The calculation is based on the distance values stored in environment map texels. This approximation is used to localize environment mapped reflections and refractions, that is, to make them depend on where they occur. On the other hand, placing the eye into the light source, the method is also good to generate real-time caustics. Computing a map for each refractor surface, we can even evaluate multiple refractions without tracing rays. The method is fast and accurate if the scene consists of larger planar faces, when the results are similar to that of ray-tracing. On the other hand, the method suits the GPU architecture very well, and can render ray-tracing and global illumination effects with few hundred frames per second.

2.1 Introduction

When computing the light transfer, the basic operation is tracing a ray from its origin point at a direction to find that point which is the source of illumination. Current graphics processing units (GPU) trace rays of the same origin very efficiently. However, in reflection, refraction and caustic computations the rays are not so coherent, but we need to trace just a single ray from each of many origins. Although it is possible to implement such a general ray tracer on the GPU [99, 100], its performance is much poorer than that of tracing a bundle of rays sharing the same origin.

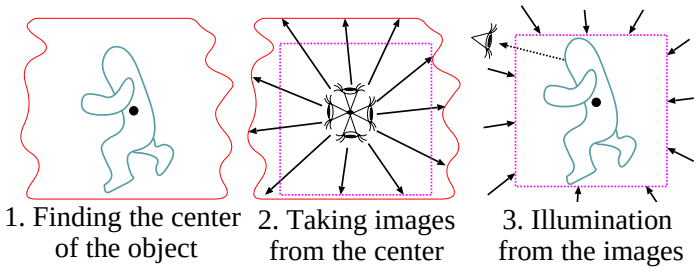


Figure 2.1: Steps of environment mapping

A GPU friendly approximation technique is *environment mapping*, which assumes that the hit point is very (infinitely) far, and thus it becomes independent of the ray origin. In this case rays can be translated to the same *reference point*, so we get that case back for which the GPU is an optimal tool. When rays originate at a given object, environment mapping takes images about the environment from the center of the object, then the environment of the object is replaced by a cube, or by a sphere, which is textured by these images (figure 2.1). When the incoming

illumination from a direction is needed, instead of sending a ray we can look up the result from the images constituting the *environment map*. Environment mapping [14] has been originally proposed to render ideal mirrors in local illumination frameworks, then extended to approximate general secondary rays without expensive ray-tracing [42, 103, 137]. Environment mapping has also become a standard technique of *image based lighting* [87, 29].

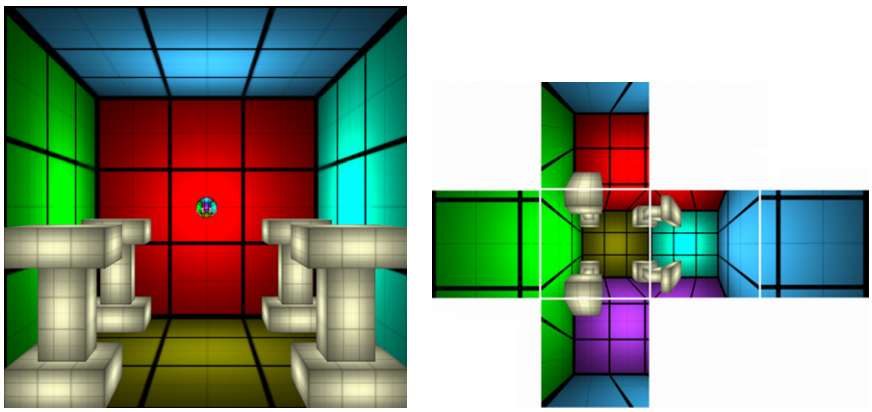


Figure 2.2: A reflective sphere and the used environment map

A fundamental problem of environment mapping is that the environment map is the correct representation of the direction dependent illumination only at a single point, the reference point of the object. For other points, accurate results can only be expected if the distance of the point of interest from the reference point is negligible compared to the distance from the surrounding geometry (figure 2.2). However, when the object size and the scale of its movements are comparable with the distance from the surrounding surface, errors occur, which create the impression that the object is independent of its illuminating environment (figure 2.3).

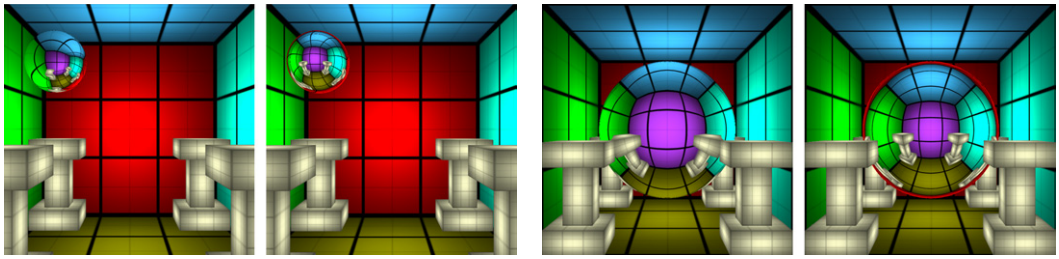


Figure 2.3: Errors of environment mapping compared to ray tracing

A compromise is needed that uses just a single environment map for the whole object, which is recomputed only if the object has moved far from the reference point. Thus the computational cost is close to that of the original environment mapping method. On the other hand, the single environment map is used "intelligently" to provide different local illumination information for every point, based on the relative location from the reference point. At a given time, we associate just a single environment map with each dynamic, reflective object, but make "localized" illumination lookups when its different points are shaded. Making texture map lookups depend on the viewing direction has been suggested in the context of bump mapping [136], displacement mapping [133], and in image based rendering [73]. Localized image based lighting has been proposed by Bjorke [12], where a proxy geometry (e.g. a sphere or a cube) of the environment is intersected by the

reflection ray to obtain the visible point. Due to the fixed and simple proxy geometry, it is possible to implement ray-tracing on the pixel shader of the GPU. However, the assumption of a simple and constant environment geometry creates visible artifacts that make the proxy geometry apparent during animation.

Unlike previous environment mapping methods, we do not ray-trace the neighborhood or the proxy geometry, but rely solely on environment map lookups. The geometric information required by the localization process is also stored in the environment map, similarly to nailboards [111] or layered depth impostors [32]. This information is the distance of the source of the illumination from the reference point where the environment map was taken. The proposed method picks environment map texels for a ray based on geometric information. Of course, a single map cannot always guarantee correct results if view dependent occlusions occur. As the object moves in the environment, the probability of such occlusions increases. Thus when the movement exceeds a threshold, the environment map is regenerated from the translated reference point. Since the environment map is not refreshed in every frame, the amortized cost of its generation becomes negligible.

The new environment mapping process provides exact results if the surface of the environment is a plane between the sampled points, and the approximation error is small even for non-planar surfaces as well. The algorithm is simple and can be executed by current vertex and pixel shaders at very high frame rates.

2.2 Localization of the environment map

The basic idea of this chapter to localize environment maps is discussed using the notations of figure 2.4. Let us assume that center $\vec{o}$ of our coordinate system is the reference point of the environment map and we are interested in the illumination of point $\vec{x}$ from direction $\vec{R}$. We suppose that direction vector $\vec{R}$ has unit length.

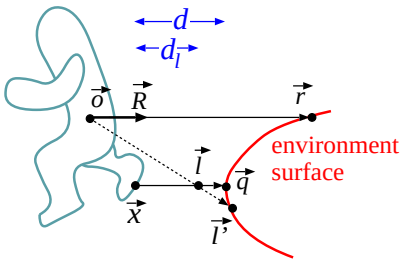


Figure 2.4: Localization of the environment map having reference point $\vec{o}$. When computing the incoming radiance at point $\vec{x}$ from direction $\vec{R}$, ray tracing would select point $\vec{q}$, classical environment mapping would read the radiance of point $\vec{r}$, while the proposed method calculates $\vec{l}$ approximating the hit point on the ray and looks up the environment map in this direction obtaining the radiance of point $\vec{l}'$.

Classical environment mapping would look up the illumination selected by direction $\vec{R}$, that is, it would use the radiance of point $\vec{r}$. However, $\vec{r}$ is usually not equal to point $\vec{q}$, which is in direction $\vec{R}$ from $\vec{x}$, and thus satisfies the following ray equation for some distance d:

$$\vec{q} = \vec{x} + \vec{R} \cdot d. \tag{2.1}$$

Our localization method finds an approximation of d using an iterative process working with distances between the environment and reference point $\vec{o}$. The required distance information can be computed during the generation of the environment map. While a normal environment map stores the illumination for each direction in R,G,B channels, now we also obtain the distance of

the visible point for these directions and store it, for example, in the alpha channel. We call these extended environment maps as *distance impostors*.

2.2.1 Finding ray hit approximations

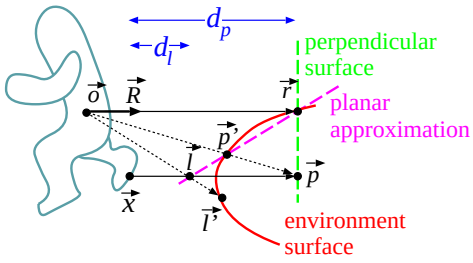


Figure 2.5: Identifying first approximation point $\vec{p}$ assuming that the surface is perpendicular to $\vec{R}$, and second approximation point $\vec{l}$ supposing that the surface is planar between points $\vec{r}$ and $\vec{p}'$.

To find an initial guess for the ray hit, we first assume that the environment surface at $\vec{r}$ is perpendicular to ray direction $\vec{R}$ (figure 2.5). In case of perpendicular surface, the ray would hit point $\vec{p}$. Points $\vec{r}$, $\vec{x}$ and origin $\vec{o}$ define a plane, which is the base plane of figure 2.5. This plane also contains visible point approximation $\vec{p}$ and unit direction vector $\vec{R}$. Multiplying ray equation

$$\vec{x} + \vec{R} \cdot d_p = \vec{p} \tag{2.2}$$

by direction vector $\vec{R}$ and substituting $\vec{R} \cdot \vec{p} = |\vec{r}|$, which is the consequence of the perpendicular surface assumption, we can express ray parameter d_p:

$$d_p = |\vec{r}| - \vec{R} \cdot \vec{x}.$$

If we used the direction of point $\vec{p}$ to lookup the environment map, we would obtain the color of point $\vec{p}'$, which is in the direction of $\vec{p}$ but is on the surface.

If the surface were a plane between $\vec{p}'$ and $\vec{r}$, then the intersection of the planar surface and the plane defined by points $\vec{r}$, $\vec{x}$ and $\vec{o}$ would be a line, and the point visible from $\vec{x}$ at direction $\vec{R}$ would be $\vec{l}$. Let us assume that the surface can be well approximated by a plane between points $\vec{r}$ and $\vec{p}'$, and find intersection $\vec{l}$ of the plane with the ray. The intersection is on the ray, thus it satisfies the following ray equation:

$$\vec{l} = \vec{x} + \vec{R} \cdot d_l. \tag{2.3}$$

On the other hand, point $\vec{l}$ is also on the line of $\vec{r}$ and $\vec{p}'$, thus it can be expressed as a combination of these two points with unknown weight α:

$$\vec{l} = \vec{r} \cdot \alpha + \vec{p}' \cdot (1 - \alpha).$$

Note that $\vec{l}$ is not necessarily between $\vec{r}$ and $\vec{p}'$ thus α is not restricted to $[0, 1]$ but can have an arbitrary value. Substituting identities $\vec{p}' = \vec{p} \cdot |\vec{p}'|/|\vec{p}|$, $\vec{r} = \vec{R} \cdot |\vec{r}|$, and equation 2.2, we obtain

$$\vec{l} = \vec{r} \cdot \alpha + (\vec{x} + \vec{R} \cdot d_p) \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha) = \vec{x} \cdot \left((1 - \alpha) \cdot \frac{|\vec{p}'|}{|\vec{p}|} \right) + \vec{R} \cdot \left(d_p \cdot (1 - \alpha) \cdot \frac{|\vec{p}'|}{|\vec{p}|} + \alpha \cdot |\vec{r}| \right).$$

Comparing this expression with equation 2.3, we obtain the following requirements for unknowns α and d_l:

$$(1 - \alpha) \cdot \frac{|\vec{p}'|}{|\vec{p}|} = 1, \qquad d_p \cdot (1 - \alpha) \cdot \frac{|\vec{p}'|}{|\vec{p}|} + \alpha \cdot |\vec{r}| = d_l.$$

Solving this equation, we get:

$$d_l = d_p + |\vec{r}| \cdot \left(1 - \frac{|\vec{p}|}{|\vec{p}'|}\right). \tag{2.4}$$

Substituting this distance into the ray equation, we obtain hit point approximation $\vec{l}$, which can be used to look up the environment map. In this way, we would obtain the radiance of point $\vec{l}'$ of figure 2.5.

2.2.2 Refinement by iteration

So far we obtained two initial guesses of the ray parameter d_p and d_l, and consequently two points $\vec{p}$ and $\vec{l}$ that are on the ray, but are not necessarily on the surface, and two other points $\vec{p}'$ and $\vec{l}'$ that are on the surface, but are not necessarily on the ray (figure 2.6). These initial results can be refined by an iteration process, which computes new hit point approximation $\vec{l}_{new}$ assuming that the surface is planar between sample points $\vec{p}'$ and $\vec{l}'$, and then replaces $\vec{p}$ by $\vec{l}$ and $\vec{l}$ by $\vec{l}_{new}$.

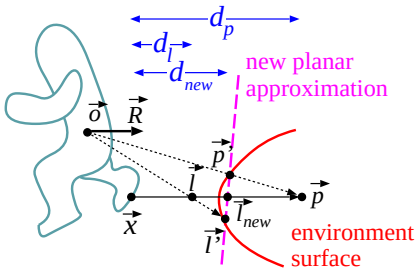


Figure 2.6: Refinement by iteration.

Since new approximation $\vec{l}_{new}$ is on the ray, it satisfies the following ray equation:

$$\vec{l}_{new} = \vec{x} + \vec{R} \cdot d_{new}. \tag{2.5}$$

Point $\vec{l}_{new}$ is also on the line of $\vec{l}'$ and $\vec{p}'$, thus it can be expressed as their combination with unknown weight α:

$$\vec{l}_{new} = \vec{l}' \cdot \alpha + \vec{p}' \cdot (1 - \alpha).$$

Substituting identities $\vec{p}' = \vec{p} \cdot |\vec{p}'|/|\vec{p}|$ and $\vec{l}' = \vec{l} \cdot |\vec{l}'|/|\vec{l}|$, as well as the ray equation for d_p and d_l, we get:

$$\vec{l}_{new} = (\vec{x} + \vec{R} \cdot d_l) \cdot \frac{|\vec{l}'|}{|\vec{l}|} \cdot \alpha + (\vec{x} + \vec{R} \cdot d_p) \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha).$$

Comparing this expression with equation 2.5, we obtain the following requirements for unknowns α and d_{new}:

$$\frac{|\vec{l}'|}{|\vec{l}|} \cdot \alpha + \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha) = 1,$$

$$d_l \cdot \frac{|\vec{l}'|}{|\vec{l}|} \cdot \alpha + d_p \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha) = d_{new}.$$

Solving this equation, we get:

$$d_{new} = d_l + (d_l - d_p) \cdot \frac{1 - |\vec{l}|/|\vec{l}'|}{|\vec{l}|/|\vec{l}'| - |\vec{p}|/|\vec{p}'|}. \tag{2.6}$$

A step of the iteration evaluates this formula and replaces d_p by d_l and d_l by d_{new} together with their associated points on the ray and on the surface.

From mathematical point of view, the proposed iteration method solves ray equation $f(\vec{x}+\vec{R}\cdot d)=0$ with the *secant method* [134], where $f(\vec{r})=0$ is the implicit equation of the environment surface, which is represented by discrete distance samples of the environment map. The secant method usually converges very quickly [134], but it may not converge for high variation functions. Note that in equation 2.6 the absolute value of denominator $|\vec{l}|/|\vec{l'}|-|\vec{p}|/|\vec{p'}|$ can be smaller than the absolute value of enumerator $1-|\vec{l}|/|\vec{l'}|$, which means that step size $|d_l-d_p|$ may also increase. While increasing the step size where necessary improves convergence, it is also the cause of divergence.

To address the convergence issue, the basic iteration should be slightly modified. Let us first recognize that ratios $|\vec{l}|/|\vec{l'}|$ and $|\vec{p}|/|\vec{p'}|$ showing up in equation 2.6 express the accuracy of approximation points $\vec{l}$ and $\vec{p}$. If the point were the real ray hit, then the ratio would be 1. Values smaller than 1 indicate that the approximation point is an *undershooting*, i.e. it is in front of the surface. On the other hand, when the ratio is greater than one, the approximation is an *overshooting* since the approximation point is behind the surface. For example, in figure 2.6 $\vec{p}$ and $\vec{l}_{new}$ are overshooting points, while $\vec{l}$ is an undershooting.

Note that when $\vec{l}$ and $\vec{p}$ are of different types, the absolute value of denominator $|\vec{l}|/|\vec{l'}|-|\vec{p}|/|\vec{p'}|$ cannot be smaller than the absolute value of enumerator $1-|\vec{l}|/|\vec{l'}|$, which results in a step size reduction. This condition can be enforced if last approximation $\vec{l}$ is paired not necessarily with the last but one approximation, but the last approximation of opposite type. This method is equivalent to the *false position root finding method* [134]. We can switch from the secant method to the false position method if we have at least one overshooting point and one undershooting approximation. Since default point $\vec{r}$ corresponds to an infinite ray parameter, which is a sure overshooting, we can use this ideal point at infinity if there are no other overshooting results. On the other hand, if there is no undershooting point, we can either follow the secant rule, use reflection point $\vec{x}$, or default point $\vec{r}$ again to substitute the undershooting point. We have found that the last option is safe and works well.

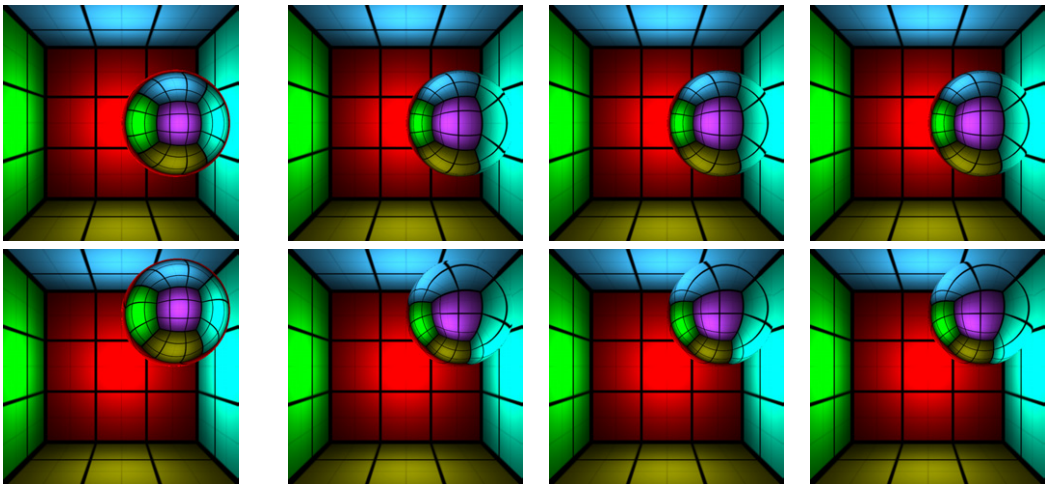
Note that even with guaranteed convergence, the proposed method is not necessarily equivalent to exact ray tracing in the limiting case. Small errors may be due to the discrete surface approximation, or to view dependent occlusions. For example, should the ray hit a point that is not visible from the reference point of the environment map, then the presented approximation scheme would obviously be unable to find that. However, when the object is curved and moving, these errors can hardly be recognized visually.

2.3 Environment mapping with distance impostors

The computation of distance impostors is very similar to that of classical environment maps. The only difference is that the distance from the reference point is also calculated, which can be stored in a separate texture or in the alpha channel of the environment map. Since the distance is a non linear function of the homogeneous coordinates of the points, correct results can be obtained only by letting the pixel shader compute the distance values. Environment maps are usually parameterized by *cube mapping*, which projects onto the six faces of a cube. Having placed the camera at the reference point and set its viewing direction to the directions of coordinate axes, the scene is rendered six times.

Having the distance impostor, we can place an arbitrary object in the scene and illuminate it with its environment map using custom vertex and pixel shader programs. The vertex shader transforms objects to normalized screen space, and also to the coordinate system of the environment map first applying the modeling transform, then translating to the reference point. View vector $\vec{V}$ and normal $\vec{N}$ are also obtained in world coordinates.

Having the graphics hardware computed the homogeneous division and filled the triangle with linearly interpolating all vertex data, the pixel shader is called to find ray hit $\vec{l}$ and to look up the cube map in this direction.



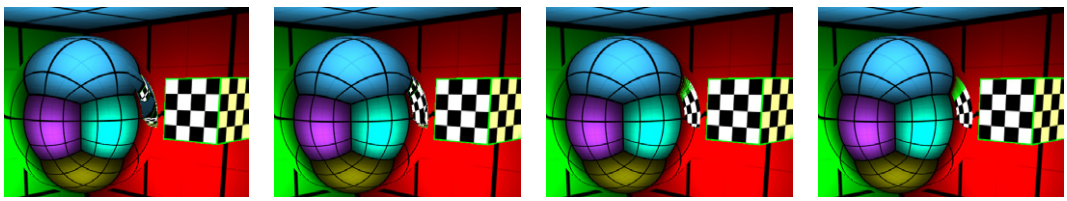
classical environment map
642 FPS

distance impostor
447 FPS

+1 iteration
323 FPS

ray traced reference

Figure 2.7: Comparison of classical and localized environment map reflections with ray traced reflections placing the reference point at the center of the room and moving a reflective sphere to different locations. Note that even the initial guess made with the distance impostor is accurate almost everywhere but the corners where one iteration step is enough. The FPS values are measured with 700×700 resolution on an NV6800GT.



distance impostor
430 FPS

+1 iteration
323 FPS

+4 iterations
169 FPS

+8 iterations
108 FPS

Figure 2.8: A more difficult case when the room contains a box that makes the scene strongly concave and is responsible for view dependent occlusions.

The HLSL code of function `Hit` computing hit point approximation $\vec{l}$ with the false position method is shown below:

```
float3 Hit(float3 x, float3 R, samplerCUBE mp) {
   float rl = texCUBE(mp, R).a; // |r|
   float dp = rl - dot(x, R);
   float3 p = x + R * dp;
   float ppp = length(p)/texCUBE(mp,p).a;
   float dun =0, dov =0, pun = ppp, pov = ppp;
   if (ppp < 1) dun = dp; else dov = dp;
   float dl = max(dp + rl * (1 - ppp), 0);
   float3 l = x + R * dl;
                                 // iteration
   for(int i = 0; i < NITER; i++) {
      float ddl;
      float llp = length(l)/texCUBE(mp,l).a;
      if (llp < 1) { // undershooting
         dun = dl; pun = llp;
         ddl = (dov == 0) ? rl * (1 - llp) :
               (dl-dov) * (1-llp)/(llp-pov);
      } else {       // overshooting
         dov = dl; pov = llp;
         ddl = (dun == 0) ? rl * (1 - llp) :
               (dl-dun) * (1-llp)/(llp-pun);
      }
      dl = max(dl + ddl, 0);  // avoid flip
      l = x + R * dl;
   }
   return l;
}
```

This function gets ray origin `x` and direction `R`, as well as cube map `mp` of the environment, and returns hit approximation `l`. We suppose that the distance values are stored in the alpha channel of the environment map. Ratios $|\vec{l}|/|\vec{l'}|$ and $|\vec{p}|/|\vec{p'}|$ are represented by variables `llp` and `ppp`, respectively. Note that variables `dun` and `dov` store the last undershooting and overshooting ray parameters. If there has been no such approximation, the ray parameters are zero. In this case default point $\vec{r}$ takes their roles. In order to avoid ray flipping, the algorithm limits ray parameters for the non-negative domain.

The pixel shader calls function `Hit` and looks up cube map `envmap` again to find illumination `I` of the hit point:

```
N = normalize(N); V = normalize(V);
R = reflect(V, N);       // reflection dir.
float3 l = Hit(x, R, envmap);   // ray hit
float3 I = texCUBE(envmap, l).rgb;
```

The next step is the computation of the reflection of incoming radiance `I`. If the surface is an ideal mirror, the incoming radiance should be multiplied by the Fresnel term evaluated for the angle between surface normal $\vec{N}$ and reflection direction $\vec{R}$. We applied an approximation of the Fresnel function, which is similar to Schlick's approximation [112] in terms of computational cost, but can take into account not only *refraction index* n but also *extinction coefficient* k, which is essential for realistic metals (see chapter 10):

$$F(\vec{N}, \vec{R}) = \frac{(n-1)^2 + k^2 + 4n(1 - \vec{N} \cdot \vec{R})^5}{(n+1)^2 + k^2}.$$

Figure 2.7 compares images rendered by the proposed method with standard environment mapping and ray tracing. Note that for such scenes where the environment is convex from the reference point of the environment map, and there are larger planar surfaces, the new algorithm converges very quickly. In fact, even the initial guesses are usually accurate, and iteration is needed only close to edges and corners.

Figure 2.8 shows a difficult case where the box makes the environment surface concave and of high variation. Note that the convergence is still pretty fast, but the converged image is not exactly what we expect. We can observe that the green edge of the box is visible in a larger portion of the reflection image. This phenomenon is due to the fact that a part of the wall is not visible from the reference point of the environment map, but are expected to show up in the reflection. In such cases the algorithm can go only to the edge of the box and substitutes the reflection of the occluded points by the blurred image of the edge.

2.4 Multiple refractions

The proposed method can be used to simulate not only reflected but also refracted rays, just the direction computation should be changed from the law of reflection to the Snellius-Descartes law of refraction, that is, the `reflect` operation should be replaced by the `refract` operation in the pixel shader. However, tracing a refraction ray on a single level is usually not enough since the light is refracted at least twice to go through a refractor. The location of the second refraction as well as the normal vector at this point depend on the geometry of the object, which can only be analyzed by ray-tracing unless the refractor is very special (e.g. a sphere, a cylinder, etc.).

Applying distance impostors, however, we can solve this problem as well if the refractor is not strongly concave, i.e. all surface points can be seen from its center point. We create a distance impostor for each refractor, which stores the distance of the refractor surface from its center and the normal vector of the surface. If the refractor has static geometry, these impostors can be obtained during preprocessing. We call this distance impostor as the *refractor map*.

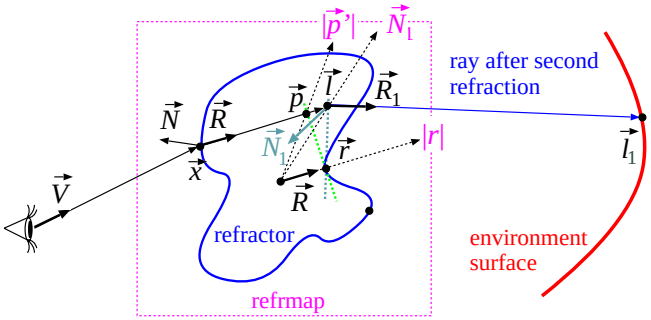
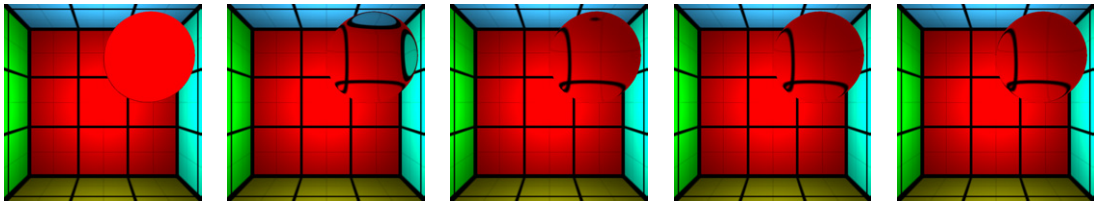


Figure 2.9: Multiple refractions without iterative refinement. The ray refracts at $\vec{x}$ to direction $\vec{R}$. The refractor map is looked up in direction $\vec{R}$ to obtain $\vec{r}$. Using the perpendicular surface and planar surface assumptions we get $\vec{p}$ and $\vec{l}$, respectively. The refractor map is looked up in direction $\vec{l}$ to find normal $\vec{N}_1$ of the second refraction. Then the ray is refracted again at $\vec{l}$ using normal $\vec{N}_1$, and the process is continued with the environment map localization.

Now let us consider point $\vec{x}$ on the surface of the refractor visible from the camera (figure 2.9). Using the proposed method for the refractor map, we obtain that point which is hit by the refraction ray. The normal vector at this point can be read from the distance impostor. Computing another refraction at this point and setting the origin of the refraction ray to the previously identified point, we can continue with the real environment map and find that point and color, which is visible after two refractions.

The pixel shader of double refractions uses the refractor map (`refrmap`) with reference point stored in variable called `objcenter`:

```
N = normalize(N); V = normalize(V);
R = refract(V, N, 1/n); // first refraction
float3 l = Hit(x-objcenter, R, refrmap);
float3 N1 = texCUBE(refrmap, l).xyz;
R1 = refract(R, N1, n); // second refraction
   // envmap lookup
float3 l1 = Hit(l+objcenter, R1, envmap);
float3 I = texCUBE(envmap, l1); // in rad.
```



classical environment map 804 FPS | distance impostor single refraction 695 FPS | distance impostor double refraction 508 FPS | +1 iteration double refraction 249 FPS | ray traced reference

Figure 2.10: Refractions of a sphere having refraction index $n = 1.1$.

Figure 2.10 shows a refracting sphere rendered with classical environment map and also by the new method with single and double refractions. Note that multiple refraction has a significant effect even when the refraction index is close to 1, and also that the proposed method is quite accurate even with only one additional iteration step.

2.5 Application to caustics generation

The method presented so far can compute the hit point after the first (or higher order) reflection or refraction of the visibility ray. If we replace the eye by a light source, the same method can also be used to determine the first (or higher order) bounce of the light ray, thus we can compute the indirect illumination bounced off dynamic objects onto static ones.

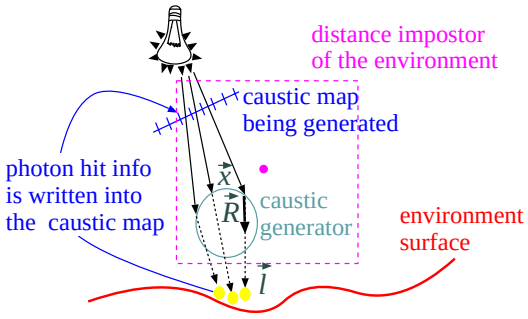


Figure 2.11: Caustics generation with environment maps

These indirect effects have a significant impact on the final image if the dynamic object is close to be an ideal reflector or refractor, when these effects show up in forms of caustic spots [55, 129, 132]. The proposed distance impostors can thus be used to compute caustics.

When rendering the scene from the point of view of the light source, the view plane is placed between the light and the refractor (figure 2.11). The image on this view plane is called *caustic map*. Note that this step is very similar to the generation of depth images for shadow maps. In fact, if we combine the method with shadow mapping, we obtain caustics almost for free. However, implementing the caustic map generation separately allows us to optimally set the position and the resolution of the view plane of the caustic map.

Supposing that the surface is an ideal reflector or refractor, point $\vec{l}$ that receives the illumination of a light source after a single or multiple reflection or refraction can be obtained by the proposed approximate ray tracing, and particularly by calling the `Hit` function, after making the following substitutions: point $\vec{x}$ is visible from the light source through a caustic map pixel, $\vec{R}$ is the refraction (or reflection) of the direction from the light source onto the surface normal at $\vec{x}$. The localized environment map lookups provide an approximation of that point $\vec{l}$, which is hit by a photon after a single reflection, or alternatively, $\vec{l}$ is an approximation of the direction of the photon hit from the reference point of the environment map.

The photon hit parameters are stored in that caustic map pixel through which the primary light ray arrived at the caustic generator object. There are several alternatives to represent a photon hit. Considering that the reflected radiance caused by a photon hit is the product of the BRDF and the power of the photon, the representation of the photon hit should identify the surface point and its BRDF. A natural identification is the texture coordinates of that surface point, which is hit by the ray. A caustic map pixel stores the identification of the texture map, the u and v texture coordinates, and finally the luminance of the power of the photon. The photon power is computed from the power of the light source and the solid angle subtended by the caustic map pixel.

The identification of the u and v texture coordinates from the direction of the photon hit requires another texture lookup. Suppose that together with the environment map, we also render another map, called `uvmap`, which has the same structure, but stores the u, v coordinates and the texture id in its pixels. Having found the direction of the photon hit, this map is read to obtain the texture coordinates, which are finally written into the caustic map.

The vertex shader of caustic map generation transforms the points and illumination direction $\vec{L}$ to the coordinate system of the environment map. Then the pixel shader computes the location of the photon hit and puts it into the target pixel:

```
N = normalize(N); L = normalize(L);
R = refract(L, N, 1/n);    // or reflect ...
float3 l = Hit(x, R, envmap); // photon hit
float3 hituv = texCUBE(uvmap, l).xyz;
return float4(hituv, power); // to caustmap
```

In order to recognize those texels of the caustic map where the refractor is not visible, we initialize the caustic map with -1 alpha values. Checking the sign of the alpha later, we can decide whether or not it is a photon hit.

The generated caustic map is used to project caustic textures onto surfaces [41], or to modify their light map in the next rendering pass. Every photon hit should be multiplied by the BRDF, and the product is used to modulate a small filter texture, which is added to the texture of the surface. The filter texture corresponds to Gaussian filtering in texture space. In this pass we render as many small quadrilaterals (two adjacent triangles in DirectX) or point sprites as texels the caustic map has.(figure 2.12)

The caustic map texels are addressed one by one with variable `caustcoord` in the vertex shader shown below. The center of these quadrilaterals is the origo, and their size depends on the support of the Gaussian filter. The vertex shader changes the coordinates of the quadrilateral vertices and centers the quadrilateral at the u, v coordinates of the photon hit in texture space if the alpha value of the caustic map texel addressed by `caustcoord` is positive, and moves the quadrilateral out of the clipping region if the alpha is negative. This approach requires the texture memory storing the caustic map to be fed back to the vertex shader, which is possible on 3.0 compatible vertex shaders.

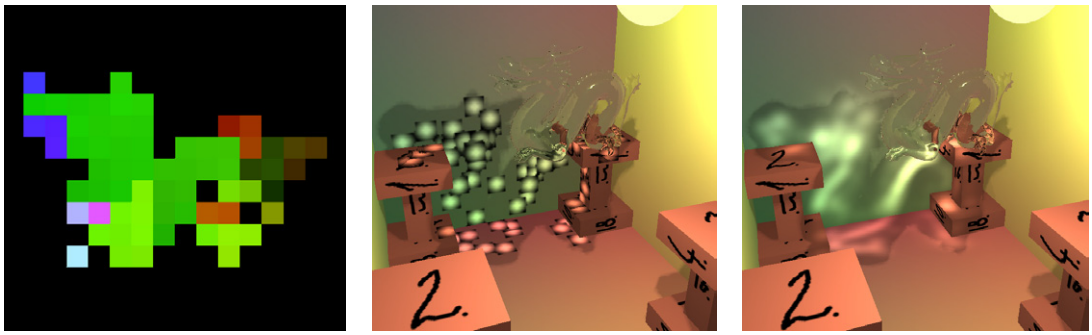


Figure 2.12: A photon map, a room without blending and a room with blending enabled.

The vertex shader of projecting caustic textures onto surfaces is as follows:

```
float4 ph = tex2Dlod(caustmap, IN.caustcoord);
OUT.filtcoord = IN.pos.xy; // filter coords
OUT.texcoord.x = ph.x + IN.pos.x / 2;
OUT.texcoord.y = ph.y - IN.pos.y / 2;
OUT.hpos.x = ph.x * 2 - 1 + IN.pos.x;
OUT.hpos.y = 1 - ph.y * 2 + IN.pos.y;
OUT.hpos.w = 1;
if (ph.a < 0) OUT.hpos.z = 2; // ignore
else          OUT.hpos.z = 0; // valid
OUT.power = ph.a;         // photon power
```

Note that the original x, y coordinates of quadrilateral vertices are copied as filter texture coordinates, and are also moved to the position of the photon hit in the texture space of the surface. The output position register (`hpos`) also stores the texture coordinates converted from $[0, 1]^2$ to $[-1, 1]^2$ which corresponds to rendering to this space. The `w` and `z` coordinates of the position register are used to ignore those caustic map elements which have no associated photon hit.

The pixel shader computes the color contribution as the product of the photon power, filter value and the BRDF:

```
float3 brdf = tex2d(textureid, texcoord);
float  w    = tex2d(filter, filtcoord);
return power * w * brdf;
```

The target of this rendering is the light map or the modified texture map. Note that the contribution of different photons should be added, thus we should set the blending mode to "add" before executing this phase.

Figure 2.13 shows the implementation of the caustics generation, when a 64×64 resolution caustic map is obtained in each frame, which is fed back to the vertex shader. Note that even with shadow, reflection, and refraction computation, the method runs with 182 FPS.

2.6 Application to soft shadow calculation

Soft shadow calculation has similarities with caustic generation. In case of caustic generation we shoot rays from the light source and calculate refraction directions at the hit point with the refractor object and use these directions to look up localized from the environment map. The last step is to store these texture coordinates which represent the photon hit points in the texture atlas. In case of soft shadow generation, we also shoot rays from the light source and calculate refraction with $n = 1$ refraction index. It means that the ray gets through the object in a straight line. With these directions we also apply the localized look up from the environment map and

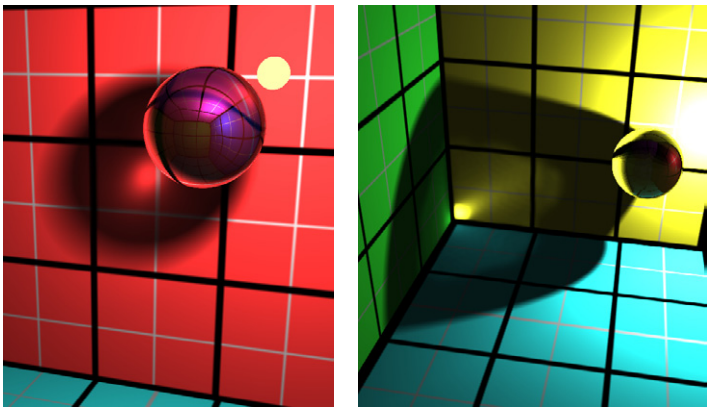


Figure 2.13: Real-time caustics caused by a glass sphere ($n = 1.3$), rendered by the proposed method on 182 FPS

store the texture coordinates in a *shadow map*. This map contains the texture coordinates behind the object seeing from the light source. Points of these texture coordinates will be in shadow.

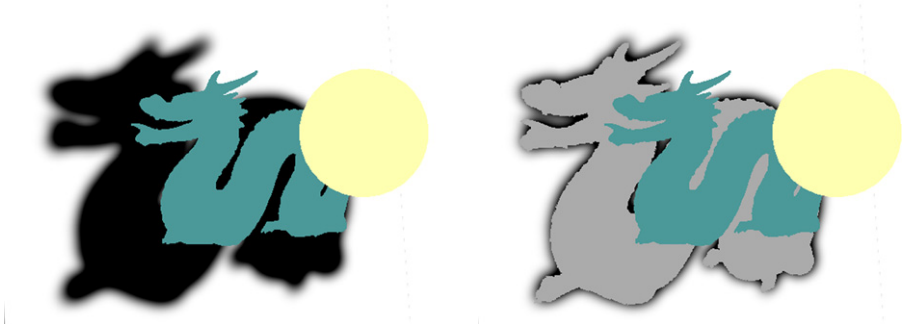


Figure 2.14: Soft shadow, and soft shadow with hard shadow inside.

For caustic generation we lighten the atlas where the photon hits located. For soft shadow generation we darken the atlas at the photon hits position (figure 2.14).

The smoothness of shadow depends on the resolution of the environment map and shadow map. The penumbra region is controlled by the size, intensity and filter of a blended photon hit. The bigger the photon hit, the smoother the transition is. The size cannot be constant for every hits, but depends on the distance proportion between the the light source and the shadow caster, and between the shadow caster and shadow receiver. The intensity of a photon hit depends on the angle of incidence on the shadow receiver.

This soft shadow generation runs about a 200 FPS with 128×128 photon map size.

2.7 The complete rendering algorithm

The different techniques proposed by this chapter can be combined in a complete real-time rendering algorithm. The input of this algorithm include

- the definition of the static environment in form of triangle meshes, material data, textures, and light maps having been obtained with a global illumination algorithm,

- refractor maps of those dynamic objects, which are expected to refract (or reflect) the light multiple times,
- the definition of dynamic objects set in the actual frame,
- the current position of light sources and the eye.

The image generation requires a preparation phase, and a rendering phase from the eye. The preparation phase computes the environment maps. Depending on the distribution of the dynamic objects, we may generate only a single environment map for all of them, or we may maintain a separate map for each of them. Note that this preparation phase is usually not executed in each frame, only if the object movements are large enough. If we update these maps after every 100 frames, then the cost amortizes and the slow down becomes negligible. If the scene has caustic generators, then a caustic map is obtained for each of them, and caustic maps are converted to light maps during the preparation phase.

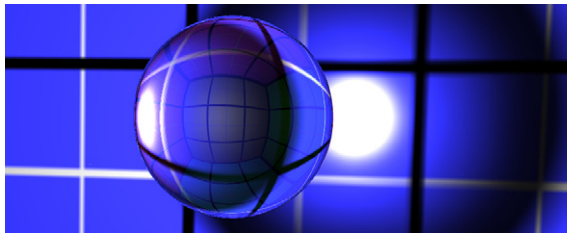


Figure 2.15: Caustics seen through the refractor object.

The final rendering phase from the eye position consists of three steps. First the static environment is rendered with their light maps making also caustics visible. Then dynamic objects are sent to the GPU, having enabled the proposed localized environment mapping and also multiple refraction computation. Note that in this way the reflection or refraction of caustics can also be generated (figure 2.15).

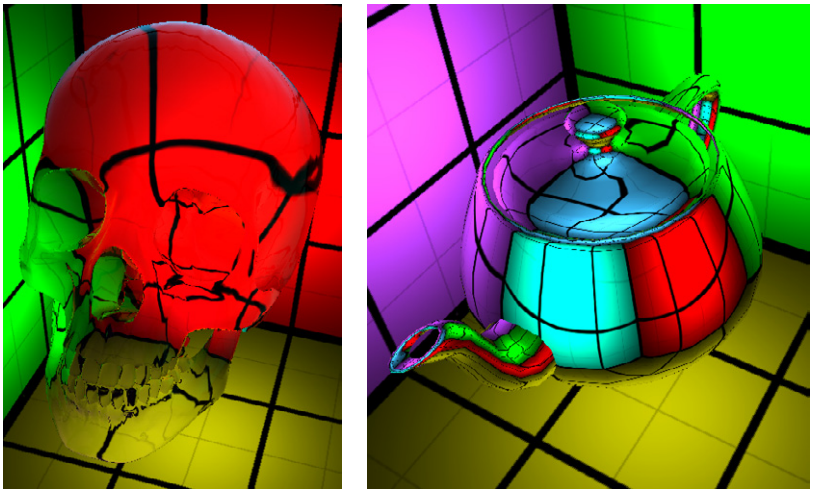


Figure 2.16: Left: a glass skull ($n = 1.3$, $k = 0$) of 61000 triangles rendered on 130 FPS. Right: an alu teapot ($n = 0.5..2.3$, $k = 5..9$) of 2300 triangles rendered on 440 FPS.

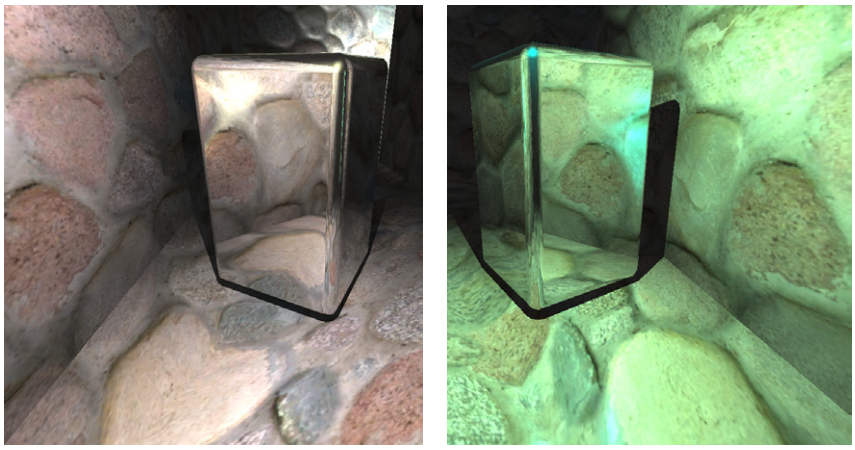


Figure 2.17: A reflective box with shadows in a stone environment illuminated by dynamic lights (160 FPS)

The presented algorithm has been implemented in Direct3D environment, first as a test application, and then in a game. The images of the test application are shown by figures 2.7, 2.8, 2.10, 2.13, 2.15, and 2.16. The test application computes the environment map only once to show that the proposed localization gives good results even if the objects moved significantly from their original positions. Note that this application runs typically with few hundred frames per second even in full screen mode on an NV6800GT graphics card and P4/3GHz CPU, and can maintain this speed for tens of thousand of triangles. For comparison, the peak performance is 1095 FPS for the scene of figure 2.7 when the pixel shader executes a return instruction for the sphere.

We also included the proposed method in a game executing shadow computation, collision detection, etc. (figures 2.17, 2.18, and 2.19) that can run with about a hundred FPS. In this game we used $6 \times 256 \times 256$ resolution cube maps that are recomputed in every 150 msec. We have realized that the speed improves by an additional 20 percent if the distance values are separated from the color data and stored in another texture map. The reason of this behavior is that the pixel shader reads the distance values several times from different texels before the color value is fetched, and separating the distance values increases texture cache utilization.

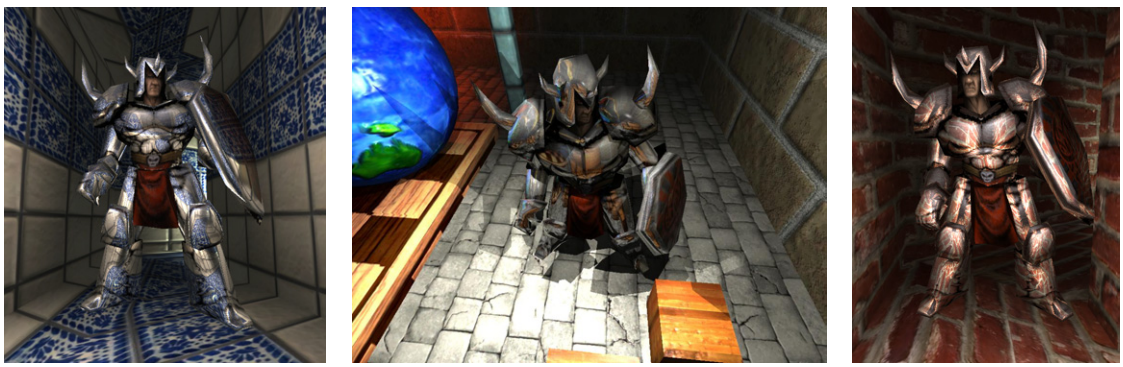


Figure 2.18: A knight in reflective armor in textured environment illuminated by dynamic lights (105..130 FPS).

Figure 2.19: Reflective and refractive spheres in a game environment running with 105 and 85 FPS, respectively.

2.8 Conclusions

This chapter presented a localization method for environment maps, which uses the distance values stored in environment map texels. The accuracy of the initial guess can be improved by iteration. In this sense, the localization method is equivalent to approximate ray-tracing, which solves the ray equation by numerical root finding.

The proposed solution can introduce effects in games that are usually simulated by ray tracing, such as single or multiple reflections and refractions on curved surfaces, and caustics. Unlike ray tracing the proposed algorithm works with a predefined set of rays obtained by rendering the scene from the reference point, which makes a bridge between tracing rays and the incremental rendering concept of the GPU.

Chapter 3

Indirect Illumination Gathering Tool

This tool presents a fast approximation method to obtain the indirect diffuse and glossy reflection on a dynamic object, caused by diffuse or moderately glossy environment. Instead of tracing new rays to find the incoming illumination, we look up this information from a cube map rendered from a reference point. However, to cope with the difference between the incoming illumination of the reference point and of the shaded point, we apply a correction that uses geometric information also stored in cube map texels. This geometric information is the distance between the reference point and the surface visible from a cube map texel. The method computes indirect illumination although approximately, but providing a very pleasing visual quality. The method suits very well to the GPU architecture, and can render these effects interactively.

3.1 Introduction

Final gathering, i.e. the computation of the reflection of the indirect illumination toward the eye, is one of the most time consuming steps of realistic rendering. This step usually requires many sampling rays from each point visible in some pixel. Ray casting finds *illuminating points* for each sampling point and at different directions, whose radiance is inserted into the rendering equation. In practical cases, number P of sample points visible from the camera is over hundred thousands or millions, while number D of sample directions are at most a hundred or a thousand to eliminate annoying sampling artifacts. In games and in real-time systems rendering, which includes final gathering, cannot take more than a few tens of milliseconds. This time does not allow tracing $P \cdot D$, i.e. a large number of, rays.

To solve this problem, we can exploit the fact that in games the dynamic objects are usually significantly smaller than their environment. Thus the global indirect illumination of the environment can be computed in a preprocessing phase, since it is not really affected by the smaller dynamic objects. On the other hand, when the indirect illumination of the dynamic objects is evaluated, their small size makes it possible to reuse illumination points selected for other sample points. The first idea of this paper is to trace rays with the graphics hardware just from a single reference point representing the dynamic object. Then the illumination points selected by these rays and their radiance are used not only for the reference point, but for all visible points of the dynamic object. This approach has two advantages. On the one hand, instead of tracing $P \cdot D$ rays, we solve the rendering problem by tracing only D rays. On the other hand, these rays form a bundle meeting in the reference point and are regularly spaced. Such ray bundles can be very efficiently traced by a graphics hardware. This method has the limitation that assuming the same set of illuminating points visible from each sample point, shadowing effects are ignored. However, while shadows are crucial for direct lighting, shadows from indirect lighting are not so visually important. Thus the user or the gamer finds this simplification acceptable.

Unfortunately, this simplification alone cannot allow real time frame rates. The evaluation of the reflected radiance at a sample point still requires the evaluation of the BRDF and the orientation angle, and the multiplication with the radiance of the illumination point by D times. These computations would need to much time.

In order to further increase the speed, we propose to carry out as much computation globally for all sample points, as possible. Clearly, this is again an approximation, since the weighting of the radiance of each illumination point at each sample point is different. From mathematical point of view, we need to evaluate an integral of the product of the illumination and the local reflection for every sample point. To allow global computation, the integrals of these products are approximated by the product of the integrals of the illumination and the local reflection, thus the illumination can be handled globally for all sample points.

The organization of this chapter is as follows. First we discuss when factoring of integrals is acceptable to simplify computations and to allow the incorporation of a single data structure, and how this data structure can be used to provide illumination information for many sample points in section 3.2.1. Section 3.3 discusses implementation details, and finally we present results and conclusions.

3.2 The new algorithm

3.2.1 Factoring the reflected radiance

According to the rendering equation, the reflected radiance of point $\vec{x}$ in viewing direction $\vec{\omega}$ can be expressed by the following integral

$$L^r(\vec{x} \to \vec{\omega}) = \int_{\Omega'} L^{in}(\vec{x} \leftarrow \vec{\omega}') \cdot f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos\theta_{\vec{x}} \; d\omega',$$

where Ω' is the set of possible illumination directions, $L^{in}(\vec{x} \leftarrow \vec{\omega}')$ is the incoming radiance arriving at point $\vec{x}$ from illumination direction $\vec{\omega}'$, f_r is the BRDF, and $\theta_{\vec{x}}$ is the angle between the surface normal at point $\vec{x}$ and the illumination direction.

Our goal is to reuse the incoming radiance information obtained for a reference point $\vec{o}$ for other nearby points as well. To do so, we use approximations and factor out those components from the rendering equation, which strongly depend on actual point $\vec{x}$.

In order to estimate the integral of the rendering equation, directional domain Ω' is decomposed to solid angles $\Delta\omega'_i, i = 1, \ldots, N$, where the radiance is roughly uniform in each domain. After decomposing the directional domain, the reflected radiance is expressed as the following sum:

$$L^r(\vec{x} \to \vec{\omega}) = \sum_{i=1}^{N} \int_{\Delta\omega'_i} L^{in}(\vec{x} \leftarrow \vec{\omega}') \cdot f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos\theta_{\vec{x}} \; d\omega'.$$

Considering a single term of this sum, representing the radiance reflected from $\Delta\omega'_i$, we use an approximation and factor out the average incoming radiance:

$$\int_{\Delta\omega'_i} L^{in}(\vec{x} \leftarrow \vec{\omega}') \cdot f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos\theta_{\vec{x}} \; d\omega' \approx \tilde{L}_{in}(\vec{x} \leftarrow \Delta\omega'_i) \cdot \int_{\Delta\omega'_i} f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos\theta_{\vec{x}} \; d\omega'$$

where $\tilde{L}_{in}$ is the *average incoming radiance* coming from $\Delta\omega'_i$:

$$\tilde{L}_{in}(\vec{x} \leftarrow \Delta\omega'_i) = \frac{1}{\Delta\omega'_i} \cdot \int_{\Delta\omega'_i} L^{in}(\vec{x} \leftarrow \vec{\omega}') \; d\omega'.$$

This factoring is acceptable because if $\Delta\omega'_i$ is small, then L^{in} has small variation and depends on the *illuminating* point, while $f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos\theta_{\vec{x}}$ depends on the *receiver* point.

The second factor is the reflectivity integral, which is also expressed as product of the average integrand and the size of the integration domain:

$$\int_{\Delta\omega_i'} f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos\theta_{\vec{x}} \, d\omega' = a(\Delta\omega_i' \to \vec{x} \to \vec{\omega}) \cdot \Delta\omega_i'$$

where

$$a(\Delta\omega_i' \to \vec{x} \to \vec{\omega}) = \frac{1}{\Delta\omega_i'} \cdot \int_{\Delta\omega_i'} f_r(\vec{\omega}' \to \vec{x} \to \vec{\omega}) \cdot \cos\theta_{\vec{x}} \, d\omega'$$

is the *average reflectivity* from solid angle $\Delta\omega_i'$.

3.2.2 Evaluating the factors

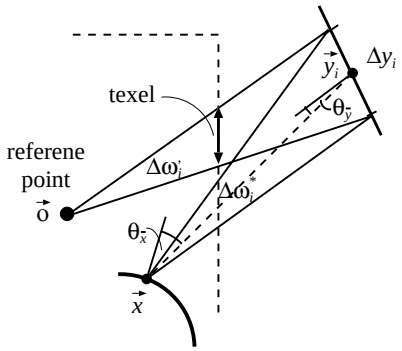


Figure 3.1: The notations of the evaluation of factors

Now let us consider the average incoming radiance and express it with the properties of the surface visible from $\vec{x}$ at the given solid angle. Let us denote the point visible in direction $\vec{\omega}'$ by $\vec{y}$, the area visible in solid angle $\Delta\omega_i'$ by Δy_i, and the center of this area by $\vec{y}_i$. The visible area and the solid angle have the following relationship:

$$\Delta\omega_i' = \frac{\Delta y_i \cdot \cos\theta_{\vec{y}_i,\vec{x}}}{|\vec{x} - \vec{y}_i|^2}$$

where $\theta_{\vec{y}_i,\vec{x}}$ is the angle between the normal vector at $\vec{y}_i$ and the direction from $\vec{y}_i$ to $\vec{x}$.

The average incoming radiance $\tilde{L}_{in}$ can also be approximated as an average in surface area Δy_i:

$$\tilde{L}_{in}(\vec{x} \leftarrow \Delta\omega_i') \approx \tilde{L}_{in}(\vec{x} \leftarrow \Delta y_i) = \frac{1}{\Delta y_i} \cdot \int_{\Delta y_i} L(\vec{y} \to \vec{\omega}_{\vec{y}\to\vec{x}}) \, dy.$$

In diffuse case, the average reflectivity is evaluated using only one directional sample $\omega_{\vec{y}_i\to\vec{x}}$ pointing from $\vec{y}_i$ toward $\vec{x}$, that is, the integrand is assumed to be constant through the domain $\Delta\omega_i'$:

$$a(\Delta y_i \to \vec{x} \to \vec{\omega}) \approx f_r(\omega_{\vec{y}_i\to\vec{x}} \to \vec{x} \to \vec{\omega}) \cdot \cos\theta_{\vec{x}}.$$

In specular case, the quadrature is evaluated using many samples and it is stored in a lookup table. In both cases the average depends on the incoming direction, which is defined by point $\vec{y}_i$. Thus in both cases we shall express the average from $\vec{y}_i$, which is emphasized by the following change in notation:

$$a(\Delta\omega_i' \to \vec{x} \to \vec{\omega}) \approx a(\vec{y}_i \to \vec{x} \to \vec{\omega}).$$

Thus, the reflected radiance can be expressed as

$$L^r(\vec{x} \to \vec{\omega}) \approx \sum_{i=1}^{N} \tilde{L}_{in}(\vec{x} \leftarrow \Delta y_i) \cdot a(\vec{y}_i \to \vec{x} \to \vec{\omega}) \cdot \Delta\omega_i'.$$

3.2.3 Reusing illumination information

Suppose now that for reference point $\vec{o}$, we identify elementary surfaces Δy_i, and evaluate incoming radiance $\tilde{L}_{in}(\vec{o} \leftarrow \Delta y_i)$. To do this, we render the scene from reference point $\vec{o}$ onto the six sides of a cube. In each pixel of these images we store the radiance of the visible point and the distance from the reference point.

The reflected radiance at the reference point is:

$$L^r(\vec{o} \to \vec{\omega}) \approx \sum_{i=1}^{N} \tilde{L}_{in}(\vec{o} \leftarrow \Delta y_i) \cdot a(\vec{y}_i \to \vec{o} \to \vec{\omega}) \cdot \Delta\omega_i'.$$

Let us now consider another point $\vec{x}$ close to the reference point $\vec{o}$ and evaluate a similar integral for point $\vec{x}$ while making exactly the same assumption on the surface radiance, i.e. it is constant in areas Δy_i:

$$L^r(\vec{x} \to \vec{\omega}) \approx \sum_{i=1}^{N} \tilde{L}_{in}(\vec{x} \leftarrow \Delta y_i) \cdot a(\vec{y}_i \to \vec{x} \to \vec{\omega}) \cdot \Delta\omega_i^*.$$

where

$$\Delta\omega_i^* = \frac{\Delta y_i \cdot \cos\theta_{\vec{y}_i,\vec{x}}}{|\vec{x} - \vec{y}_i|^2}$$

is the solid angle in which Δy_i is seen from $\vec{x}$.

Recall that according to the correspondence of $\Delta\omega_i'$ and Δy_i, we can express Δy_i as

$$\Delta y_i = \frac{\Delta\omega_i' \cdot |\vec{o} - \vec{y}_i|^2}{\cos\theta_{\vec{y}_i,\vec{o}}}.$$

Substituting this into the formula of $\Delta\omega_i^*$, we obtain

$$\Delta\omega_i^* = \Delta\omega_i' \cdot \frac{|\vec{o} - \vec{y}_i|^2 \cdot \cos\theta_{\vec{y}_i,\vec{x}}}{|\vec{x} - \vec{y}_i|^2 \cdot \cos\theta_{\vec{y}_i,\vec{o}}}.$$

Assume that the environment surface is not very close, thus the angles between the normal vector at $\vec{y}_i$ and reflection vectors from $\vec{o}$ and from $\vec{x}$ are similar ($\cos\theta_{\vec{y}_i,\vec{x}} \approx \cos\theta_{\vec{y}_i,\vec{o}}$).

On the other hand, supposing diffuse or moderately specular environment, receiving point $\vec{x}$ can be translated a little while $\tilde{L}_{in}(\vec{x} \leftarrow \Delta y_i)$ remains approximately the same, making $\tilde{L}_{in}$ independent of the receiving point $\vec{x}$ and allowing its global computation, that is

$$\tilde{L}_{in}(\vec{x} \leftarrow \Delta y_i) \approx \tilde{L}_{in}(\vec{o} \leftarrow \Delta y_i).$$

Using these approximations – although Δy_i and $\cos\theta_{\vec{y}_i,\vec{x}}$ are not known – the reflected radiance at point $\vec{x}$ can be expressed as

$$L^r(\vec{x} \to \vec{\omega}) \approx \sum_{i=1}^{N} \tilde{L}_{in}(\vec{x} \leftarrow \Delta y_i) \cdot a(\vec{y}_i \to \vec{x} \to \vec{\omega}) \cdot \Delta\omega_i' \cdot \frac{|\vec{o} - \vec{y}_i|^2}{|\vec{x} - \vec{y}_i|^2}.$$

Those factors that are independent of point $\vec{x}$ can be pre-computed in

$$L_i^* = \tilde{L}_{in}(\vec{o} \leftarrow \Delta y_i) \cdot \Delta\omega_i \cdot |\vec{o} - \vec{y}_i|^2,$$

and stored in the texels of the environment map. Then the summation of

$$L^r(\vec{x} \to \vec{\omega}) \approx \sum_{i=1}^{N} \frac{L_i^* \cdot a(\vec{y}_i \to \vec{x} \to \vec{\omega})}{|\vec{x} - \vec{y}_i|^2}$$

is executed on-the-fly for each visible point $\vec{x}$.

The resolution of this environment map depends on how quickly average reflectivity a changes. For diffuse materials a changes slowly, thus we can obtain good results using a low-resolution map, which requires a few terms to be added. On the one hand, for a highly specular material a has changes quickly but is non-zero only for a few texels, thus the sum has just a few non-zero terms again even if the resolution of the environment map is higher than used for the diffuse case. We set the resolution of the map according to the specularity of the material and evaluate just a few (say 5) terms.

3.3 Implementation

The environment map is sampled down with the following pixel shader:

```
float4 ReduceCubeMapPS( float2 Tex : TEXCOORD0,
                        float4 pos : TEXCOORD1) : COLOR0 {
   float4 color = 0;
   float2 tpos = (texture coordinate of the first texel to be sampled)
   for (int i = 0; i < 64; i++)    // averaging values of 64x64 texels
   for (int j = 0; j < 64; j++) {
      t.x = tpos.x + i/256.0f;
      t.y = tpos.y + j/256.0f;
      color += tex2D(CubeMapSampler, t);
   }
   color /= 64 * 64;
   return color;
```

The computation of the stored L_i^* texel values is done as follows:

```
float4 PreprocCubeMapPS( float2 Tex : TEXCOORD0,
                         float4 pos : TEXCOORD1) : COLOR0 {
   float2 tpos = pos.xy/2+0.5;   // position (-1..1) to texture coordinate (0..1)
   tpos.y = 1-tpos.y;
   float r2 = 1 + pos.x*pos.x + pos.y*pos.y;
   return tex2D(SmallCubeMapSampler, tpos) / (r2*r2); (*)
}
```

The solid angle subtended by the given texel is: $\Delta\omega_i^* = \Delta y_i \cdot \cos\theta / r^2 = \Delta y_i / r^3$, where

$$r = \sqrt{1 + (pos.x)^2 + (pos.y)^2}.$$

Note that — instead of r^3 — a factor of r^4 appears in line (*) to avoid an expensive square root operation. The extra r factor will be compensated later. Multiplying with factor $\Delta y_i = N^2/4$, where N is the resolution of the initial cube map is also delayed in order to avoid small texture values that may lead to reduced precision.

The naive implementation of the calculation of the diffuse reflection of the environment map is:

```
float4 DiffuseScenePS( float2 Tex    : TEXCOORD0,
                       float3 Normal : TEXCOORD1,
                       float3 V      : TEXCOORD2,
                       float3 position : TEXCOORD3 ) : COLOR0 {
   V = normalize( V ); Normal = normalize( Normal );
   float4 intens = 0;

   for (int x = 0; x < 4; x++)           // for each texel
      for (int y = 0; y < 4; y++) {
         float2 tpos = float2(x/4.0f, y/4.0f);
         float2 pos  = 2 * float2(tpos.x, 1-tpos.y) - 1;

         intens += GetContibution( float3(pos.x, pos.y,  1), position, Normal);
         intens += GetContibution( float3(pos.x, pos.y, -1), position, Normal);
          // other 4 directions...
   }
   return intens;
}
```

The implementation of the `GetContribution` function is:

```
float4 GetContibution(float3 IllumDir, float3 position, float3 normal) {
   float4 L_preproc = 4.0 / (4*4) * texCUBE(PreprocMapSampler, IllumDir);  (**)

   float r = texCUBE(CubeMapSampler, IllumDir).a;
   float r_= length(position - IllumDir/length(IllumDir) * r);

   float r_Cos = max(dot(normal,IllumDir),0);                              (***)
   return L_preproc * k_d * r_Cos * (r*r) / (r_*r_);
}
```

Line (**) contains the "missing" multiplication with $\Delta y_i = N^2/4$. The extra r factor is compensated in line (***) since vector `IllumDir` is not normalized and its length equals r.

3.4 Results

Let us consider a simple environment consisting of a cubic room with a divider face (Figure 3.2). The object to be illuminated is located in the center of the room.

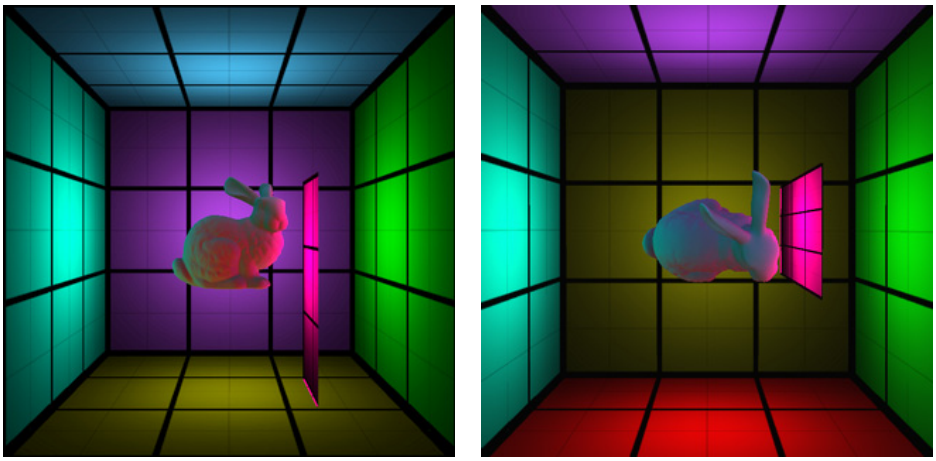


Figure 3.2: A bunny in a simple environment (side and top view).

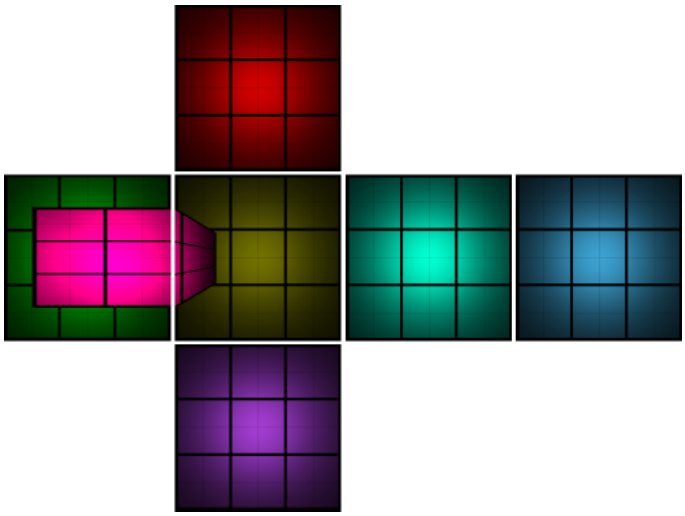


Figure 3.3: Large-resolution cube map taken from the reference point.

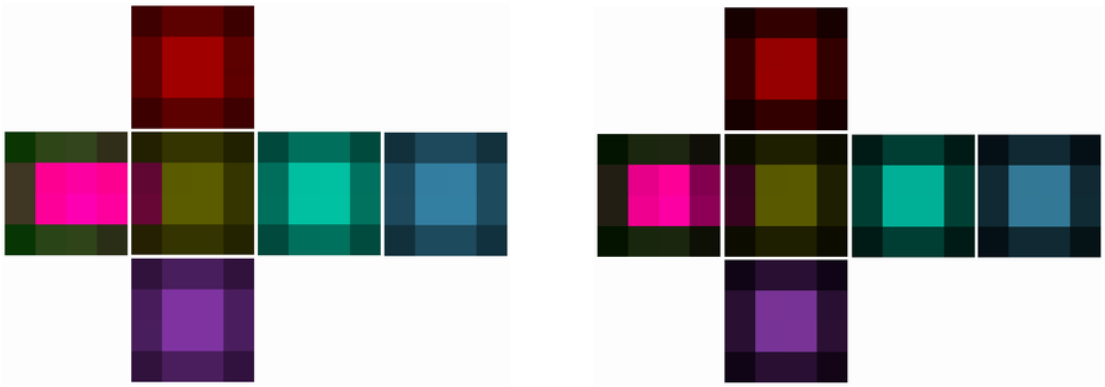


Figure 3.4: Low-resolution cube maps containing the average incoming radiance *(left)* and the average incoming radiance weighted with the solid angle subtended by the given texel *(right)*, respectively.

To record the incoming radiance, six images are taken from the middle of the room (called reference point) and they are stored in a cube map of resolution 256×256 (Figure 3.3). Then, these images are downsampled by averaging 64×64 neighboring values. The resulting average incoming radiance values are stored in another cube map of resolution 4×4. Then, the value of each texel is weighted with the solid angle subtended by that texel (Figure 3.4).

The resulting cube map contains values that can be directly used when calculating the illumination of point $\vec{x}$. See Figure 3.5 for the results.

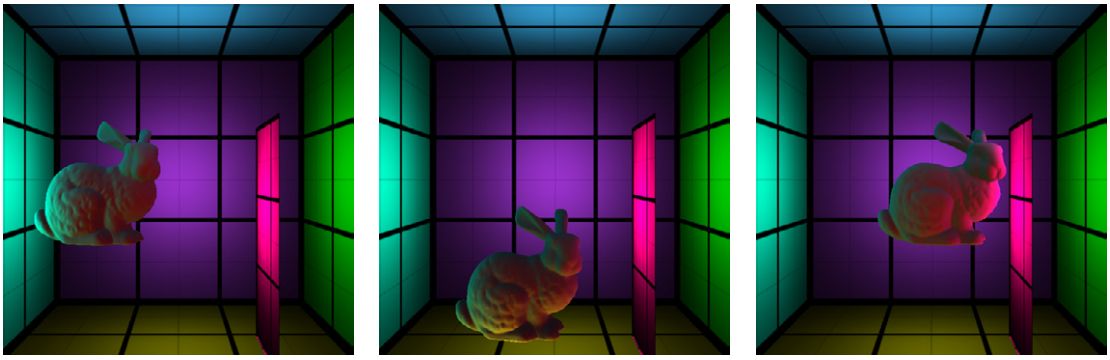


Figure 3.5: The illumination of the bunny at different points of the environment.

3.5 Conclusions

This tool presented a localization method for computing diffuse and glossy reflections of the incoming radiance stored in environment maps, which uses the distance values stored in the environment map texels.

Chapter 4

Hierarchical Ray Engine Tool

We present an improved algorithm based on the Ray Engine approach, which builds a hierarchy of rays instead of objects, completely on the graphics card. Exploiting the coherence between rays when displaying refractive objects or computing caustics, realtime frame rates are achieved without preprocessing. Thus, the method fills a gap in the realtime rendering repertoire.

4.1 Introduction

With high performance hardware designed to support scan conversion image synthesis, most research aims to eliminate time consuming ray casting from illumination algorithms, or to move it to a preprocessing step computing texture maps. However, there are some light transport effects that exhibit inherently recursive behavior, most prominently visible refractive objects, or caustics via multiple reflections or refractions. Accurate maps or transport factor matrices cannot be constructed with a feasible storage requirement. On the other hand, these problems are effectively handled by recursive raytracing or photon tracing, both based on ray casting.

In order to make ray casting feasible in realtime applications, it is imperative to make use of the immense computing power of the GPU. One delivering research direction has spawned from the approach of Purcell et al.[99], the impact of which we will briefly evaluate in Section 4.3.1. If we are looking for a solution which does not rely on a pre-built acceleration structure, the most important milestone we find is the Ray Engine[18]. Based on the recognition that ray casting is a crossbar on rays and primitives, while scan conversion is a crossbar on pixels and primitives, they have devised a method for computing all possible ray-primitive intersections on the GPU. On contemporary hardware they could achieve processing power similar to the CPU's.

As the ray engine serves as the basis of our improved approach, let us reiterate its working mechanism in current GPU terminology. Every pixel of the render target is associated with a ray. The origin and direction of rays to be traced are stored in textures that have the same dimensions as the render target. In every pass, a single ray casting primitive is taken, and it is rendered as a full-screen quad, with the primitive data attached to the quad vertices. Thus, pixel shaders for every pixel will receive the primitive data, and can also access the ray data via texture reads. The ray-primitive intersection calculation can be performed in the shader. Then, using the distance of the intersection as a depth value, a depth test is performed to verify that no closer intersection has been found yet. If the result passes the test, it is written to the render target and the depth buffer is updated. This way every pixel will hold the information about the nearest intersection between the scene primitives and the ray associated with the pixel. The pitfall of the ray engine is that it implements the naive ray casting algorithm of testing every ray against every primitive. On the other hand, ray casting research has been directed on building effective acceleration hierarchies to minimize the number of actual intersection tests performed [3]. Unfortunately, these results cannot be easily ported to the graphics hardware.

4.2 Acceleration hierarchy for the GPU

CPU-based acceleration schemes are spatial object hierarchies. Although considerable research has dealt with exploiting the coherence between neighboring rays, including longest common traversal sequences [48] and image space interpolation [131], these have not altered the basic approach. That is, for a ray, we try to exclude as many objects as possible from intersection testing. This cannot be done in the ray engine architecture, as it follows a per primitive processing scheme instead of the per ray philosophy. Therefore, we also have to apply an acceleration hierarchy the other way round, not on the objects, but on the rays.

In typical applications, realtime ray casting augments scan conversion image synthesis where recursive ray tracing from the eye point or from a light sample point is necessary. In both scenarios, the primary ray impact points are determined by rendering the scene from either the eye or the light. As nearby rays hit similar surfaces, it can be assumed that reflected or refracted rays may also travel in similar directions, albeit with more and more deviation on multiple iterations. If we are able to compute enclosing objects for groups of nearby rays, it may be possible to exclude all rays within a group based on a single test against the primitive being processed. This approach fits well with the ray engine. Whenever the data of a primitive is processed, we should find a way not to render it on the entire screen as a quad, but invoke the pixel shaders only where an intersection is possible. An obvious solution is to split the render target into tiles, render a set of tile quads instead of a full screen one, but make a decision for every tile beforehand whether it should be rendered at all. At a first glimpse, this may appear counterproductive, as, apparently, far more quads will be rendered. However, there is a set of issues that disprove concerns.

- The ray engine is pixel shader intensive, and makes practically no use of the vertex processing unit. The number of pixel shader runs, which remains crucial, is by no means increased.

- The high level test of whether a tile may include valid intersections can be performed in the vertex shader. If the intersection test fails, the vertices of the quad are transformed out of view, and discarded by clipping.

- Instead of small quads, one can use point primitives, described by a single vertex. This eliminates the fourfold overhead of processing the same vertex data for all quad vertices, and needlessly interpolating values.

In order to implement this idea, we have to solve two problems. First, for rays grouped in the same tile, an enclosing object should be computed, for which an intersection test is fast. This computation should be performed on the GPU. Second, the data describing these enclosing objects should be accessible to the vertex shader.

The latter problem can be resolved by texture reads in the vertex shader. If data is rendered to a texture, where every texel corresponds to an enclosing object, it can be accessed when processing the appropriate tile. If we reorganize the rendering process, even this vertex texture fetch can be eliminated. Remember that we are rendering a tile for every ray casting primitive, for every possible tile position. Now if we take a tile position, and render all the primitives there at once, the enclosing object information is static. It can be passed to the vertex shader in uniform parameters. In order to do this, we have to read back the texture holding the enclosing object data from the graphics card. However, as it contains only as many texels as many tiles are used (16×16 is typical), this is not an expensive operation.

Figure 4.1 depicts the data flow in an application for tracing refracted rays, using the proposed method. Ray casting primitives are encoded as single vertices, and can be channelled to the shaders as a vertex buffer of point primitives. As a result of the intersection tests, the ray defining the next segment of the refraction path is written to the render target. The process is repeated for pixels, in which the path has not yet terminated. In the beginning of every iteration, an enclosing cone for rays is built for every tile, and stored in a texture. This texture is used in consequent vertex shader runs to carry out preliminary intersection tests.

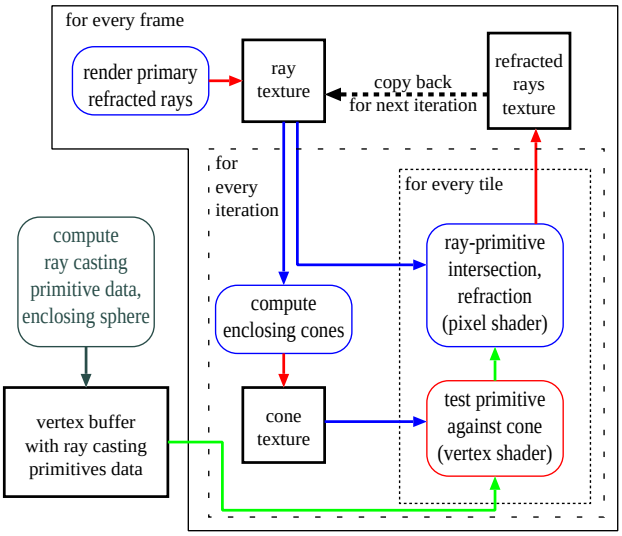


Figure 4.1: Block diagram of the hierarchical ray engine.

4.3 Construction of an enclosing cone

We need to be able to perform the intersection test between the enclosing object and the ray casting primitive as fast as possible. At the same time, representation of both the primitives and the ray-enclosing objects must be compact, because primitive data has to be passed in a very limited number of vertex registers, and enclosing objects must be described by a few texels. One rapid test is the intersection test between an infinite cone and a sphere. Enclosing spheres for all ray casting primitives can easily be computed, and described by a 3D position and a radius. Enclosing infinite cones of rays are described by an origin, a direction and an opening angle.

The infinite enclosing cones must be constructed in a pixel shader, in a pass before rendering the intersection records themselves. Note that in a practical application, the rays to be traced will be different for every frame, and for every level of refraction, so the reconstruction of the cones is also time critical. Therefore, a fast incremental approach is preferred over a tedious one, which could possibly produce more compact results, via, for instance, linear programming. The algorithm goes as follows:

1. Start with the zero angle enclosing cone of the first ray.

2. For each ray

   (a) Check if the direction of the ray lies within the solid angle covered by the cone, as seen from its apex. If it does not, extend the cone to include both the original solid angle and the new direction.

   (b) Check if the origin of the ray is within the area enclosed by the cone. If it is not, translate the cone so that it includes both the original cone and the origin of the ray. The new cone should touch both the origin of the ray and the original cone, along one of its generator lines.

Both steps of modifying the cone require some mathematics. Let $\vec{x}$ be the axis direction of the cone, $\vec{a}$ its apex, φ the half of the opening angle, $\vec{r}$ the direction of the ray, and $\vec{p}$ its origin.

First, if the solid angle defined by the cone does not include the direction of the ray, the cone has to be extended (See Figure 4.2). This is the case if $\vec{x} \cdot \vec{r} < \cos\varphi$. Then, the generator direction $\vec{e}$, opposite to the ray direction, has to be found. If $\vec{r}$ is projected onto $\vec{x}$, the direction from $\vec{r}$ to

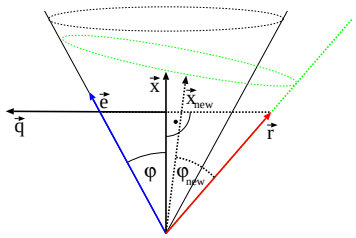


Figure 4.2: Extending the cone.

the projected point defines $\vec{q}$:

$$\vec{q} = \frac{(\vec{x}\cdot\vec{r})\cdot\vec{x} - \vec{r}}{|(\vec{x}\cdot\vec{r})\cdot\vec{x} - \vec{r}|}.$$

Then $\vec{e}$ is found as a combination of $\vec{x}$ and $\vec{q}$:

$$\vec{e} = \vec{x}\cdot\cos\varphi + \vec{q}\cdot\sin\varphi.$$

The new axis direction should be the average of $\vec{e}$ and $\vec{r}$, and the opening angle should also be adjusted:

$$\vec{x}_{\text{new}} = \frac{\vec{e}+\vec{r}}{|\vec{e}+\vec{r}|}, \qquad \cos\varphi_{\text{new}} = \vec{x}_{\text{new}}\cdot\vec{r}.$$

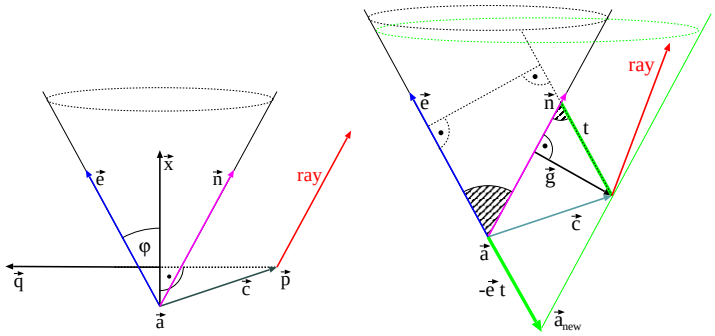


Figure 4.3: Finding the near and far generators (left), and translating a cone (right).

Translating the possibly extended cone to include the origin is somewhat more complicated, but follows the same trail (See Figure 4.3). First the *nearest* generator direction $\vec{n}$ and the *farthest* generator direction $\vec{e}$ are found just like before. The vector $\vec{c} = \vec{p} - \vec{a}$ plays the role what $\vec{r}$ had in the previous computation;

$$\vec{q} = \frac{(\vec{x}\cdot\vec{c})\cdot\vec{x} - \vec{c}}{|(\vec{x}\cdot\vec{c})\cdot\vec{x} - \vec{c}|}.$$

Like before,

$$\vec{e} = \vec{x}\cdot\cos\varphi + \vec{q}\cdot\sin\varphi, \qquad \vec{n} = \vec{x}\cdot\cos\varphi - \vec{q}\cdot\sin\varphi.$$

We want to translate the cone along generator $\vec{e}$ so that the generator $\vec{n}$ moves to cover the ray origin $\vec{p}$ (Figure 4.3, on the right). The distance vector $\vec{g}$ between the origin and the nearest generator is found as:

$$\vec{g} = \vec{c} - (\vec{n}\cdot\vec{c})\cdot\vec{n}.$$

The translation distance t and the new apex position are:

$$t = \frac{\vec{g}^2}{\vec{e}\cdot\vec{g}}, \qquad \vec{a}_{\text{new}} = \vec{a} - \vec{e}\cdot t.$$

Using the two steps together, we find a new cone that includes both the previous cone and the new ray, has a minimum opening as a priority, and was translated by a minimum amount as a secondary objective. Of course, knowing nothing about the rays already included in the cone, we cannot state that the computation achieves an optimal result in any way. However, it is conservative and mostly needs vector operations, fitting well in a pixel shader. This, actually, leaves the texture fetches required to get ray data to be the bottleneck of the execution. Furthermore, cone construction is only performed once for every tile of rays.

The computation of enclosing spheres of ray casting primitives is done on the CPU, at the same time when all the data required for the intersection test is assembled. The result is a vertex buffer, in which all ray casting primitives are encoded as vertices. Note that the position value slot can be used for passing the enclosing sphere data, as it will simply be exchanged with the tile position in the vertex shader. The cone intersects the sphere if

$$\varphi > \arccos[(\vec{v} - \vec{a}) \cdot \vec{x}] - \arcsin[r/|\vec{v} - \vec{a}|],$$

where $\vec{a}$, $\vec{x}$ and φ describe the cone as before, $\vec{v}$ is the center of the sphere and r is its radius.

In a typical application, given an array of eye or light rays, we need to find those rays, which, after multiple reflections and refractions, do not hit an ideal surface any more. These results can either be used to trace them against the non-ideal geometry of the scene, or in any other method like environment mapping or caustics generation.

Firstly, there may be pixels in which no refractive or reflective surface is visible. Furthermore, in every iteration replacing rays with their reflected or refracted successors, there will be rays not arriving on any ideal surface, producing no output. It is desirable that for those pixels where there is no ray to trace, the pixel shader is not invoked at all. This is achieved using the stencil buffer and early stencil testing. When rendering intersections, every bit of the stencil serves as a flag for a specific iteration. The stencil read and write masks select the flag bit of the previous and current iterations, respectively. Should ray casting fail to hit any object, the stencil bit will not be set, and in the next iteration the pixel will be skipped. With an eight bits deep stencil buffer, this allows for eightfold reflection or refraction to be traced.

4.3.1 Comparison with the ray engine

The ray engine was designed to be a general purpose raytracer, with no preconceptions on what rays there are to be traced. While this generality could be considered a great advantage, current advancements in technology and research make the ray engine approach obsolete in all but few areas of application. Firstly, real time global illumination is better supported by texture-based methods making use of precomputed maps or rendered environments. Secondly, in offline computations, where ray casting is still essential, ray casting acceleration schemes well known on the CPU will very soon be adopted for the graphics hardware [99]. The acceleration structure must still be built on the CPU, rendering the approach incapable of realtime rendering of dynamic scenes. That practically leaves those two areas for realtime ray casting which our hierarchical ray engine specializes in: visible refractive objects and caustics. Our algorithm can outperform the ray engine by making use of the coherence of rays in these particular problems.

4.4 Results

We have implemented the ray engine for tracing refracted eye rays, without the CPU-GPU load sharing scheme, and our algorithm (Figure 4.4), to compare their performance. Both algorithms also performed a refracted direction calculation for every intersection test.

For the hierarchical ray engine implementation, we divided the 256×256 ray array into 16×16 tiles. As depicted in Figure 4.1, the enclosing cones for the tiles were computed in a shader. The results shown in Figure 4.5 were obtained for different geometries, with the maximum path length set to 5, on an NV6800GT. A refractive surface was visible in every pixel. Considering that it takes at least two iterations to discard a path that has entered an object, a minimum estimate for the

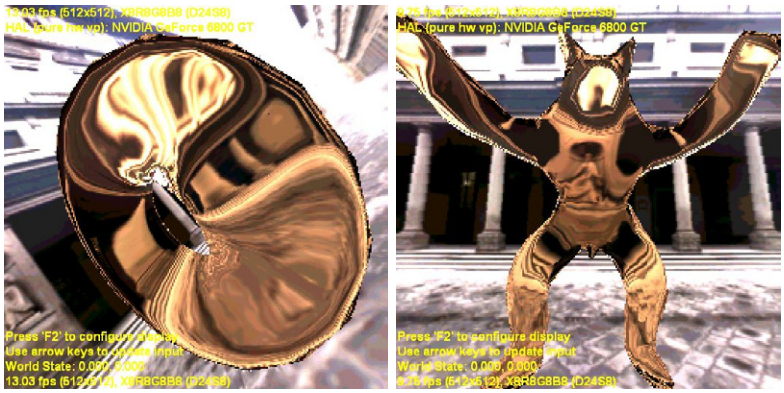


Figure 4.4: Images rendered using the hierarchical algorithm.

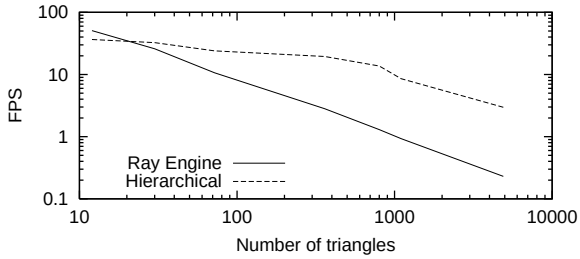


Figure 4.5: Frames per second rates achieved for rendering refractive objects.

throughput can be given. With 256×256 pixels, 5000 triangles, 2 iterations, and an FPS of 0.23 our ray engine implementation computed $150M$ intersections per second. With the hierarchical approach achieving 3 FPS, this figure reaches $2G$. The improvement over the naive approach is very much dependent on the complexity of the geometry. If there are few primitives, the ray casting does not take much time, and the constant overhead of constructing the enclosing cones does not pay off. However, as the number of primitives increases, the situation is reversed. Already at a triangle count of 100, the new algorithm runs twice as fast. Furthermore, for $256x256$ eye rays, all hitting a refractive surface, frame rates enough for real time rendering have been achieved. This makes it possible to interactively and accurately render visible refractive objects, or, when using the technique to trace photon paths, correct caustic patterns.

4.5 Cooperation with other tools

The ray engine tool performs accurate ray casting, while the localized environment map based methods perform approximate ray casting. They are obviously interchangeable in some tasks, like caustic generation or reflections. However, they are also complementing each other. While localized environment maps can be best used for detailed geometries with a convex topology, the ray engine can handle any kind of topology, but only a few hundreds of primitives in real time. Furthermore, the ray engine is not necessarily limited to the triangle primitives. While the localized environment maps will still be the primary tool to simulate the interactions between the moving actors and the static level geometry, the ray engine will make it possible to apply accurate ray casting techniques to light transport phenomena on the dynamic actors themselves, like accurate multiple refraction caustics or self-reflection.

Chapter 5

Image Based Lighting Tool

In many computer graphics applications it is desirable to augment the virtual objects with high dynamic range images representing a real environment. In order to provide the illusion that the virtual objects are parts of the real scene, the illumination of the environment should be taken into account when the virtual objects are rendered. Proper calculation of the illumination from an environment map may be extremely expensive if we wish to account for occlusion, self shadowing and specular materials. In this chapter we present a method that does all the calculations on a per-frame basis, and is already real-time on non-cutting-edge hardware.

5.1 Introduction

When real and virtual objects have to be shown together, we have to illuminate the virtual objects by the light present in the real environment. The information of the environment lighting is usually available in form of high dynamic range images taken of the real world, called *environment maps* [14, 42, 103, 137]. This chapter proposes a method to compute accurate environment lighting with shadows for dynamic scenes. Unlike previous image based lighting and shadow algorithms, our aim is to compute these methods in real-time without preprocessing of the scene, exploiting the features of the graphics hardware.

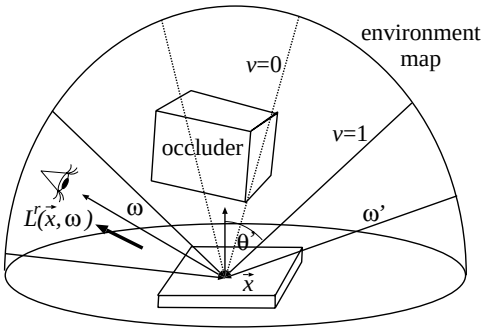


Figure 5.1: Environment mapping

From the point of view of rendering, the environment map is a large, heterogeneous area light source. Alternatively, the environment map can also be imagined as a large sky dome or a sky cube (figure 5.1). The illumination of this source on the virtual objects can be obtained by tracing rays from the points of the virtual object in the directions of the environment map, and checking whether or not occlusions occur [29, 68, 121]. However, the software ray-casting approach fails to

deliver real-time frame rates, and a technique that allows for an effective hardware implementation should take a different, texture-based approach [83].

Using the reflection or the refraction direction to address the environment texture in order to render an ideally reflective or refractive surface is a standard technique in computer graphics [42]. High dynamic range maps [30] have been added to increase realism. However, shadows are not rendered in these effects. Self-shadows are also missing, excluding objects with the slightest concavities.

More recently, Mei et al [83] have proposed a solution to render both shadows and reflective caustics under environment lighting. Their method is based on *Spherical Radiance Transfer Maps* including Spherical Shadow Maps for both mutual and self-shadow rendering. These maps have to be pre-computed for every one of the several thousand mesh vertices, and their resolution should be large enough to cover hundreds of sampled directions. Thus the time taken by preprocessing is out of proportions, and large amounts of data have to be maintained. Moreover, there are limitations, the most severe being that the objects are not deformable, not allowing for widely used techniques like skeletal animation. It is also assumed that lights are distant and minor artifacts may appear on objects close to each other because of the spherical approximation. The main caveat of the Spherical Shadow Map method is that shadow maps are rendered for the vertices. Then, when the illumination of a vertex is computed, sampled directions are tested against this map to see whether the environment map is occluded in that direction. This setup requires a large number of shadow maps, and could result in inaccurate per-pixel results, depending on tessellation.

The key idea of our approach is to decompose the environment map to finite number of directional domains. While visibility is checked with one sample in each domain with a depth map, the reflected radiance is computed accurately. The number of directional domains can be considered fairly constant, so the corresponding depth maps can be calculated run-time, allowing for dynamic scenes. Furthermore, it is possible either to render to a texture atlas, allowing for further processing, or to evaluate shadowing in a view-dependent manner, for every pixel, avoiding some texturing artifacts.

5.2 The proposed method

In order to compute the image of a virtual scene under environment illumination, we should evaluate the reflected radiance L^r of every visible point $\vec{x}$ at view direction $\vec{\omega}$:

$$L^r(\vec{x},\vec{\omega}) = \int\limits_{\Omega'} L^{env}(\vec{\omega}') \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \cdot v(\vec{x},\vec{\omega}')\ d\omega',$$

where Ω' is the set of all directions, $L^{env}(\vec{\omega}')$ is the radiance of the environment map at direction $\vec{\omega}'$, f_r is the Bi-directional Reflectance Distribution Function (BRDF), θ' is the angle between the surface normal and illumination direction $\vec{\omega}'$, and $v(\vec{x},\vec{\omega}')$ is the *visibility function* checking whether no virtual object is seen from $\vec{x}$ at direction $\vec{\omega}'$ (that is, the environment map can illuminate this point from the given direction). In order to estimate this integral, directional domain Ω' is decomposed to solid angles $\Delta\omega'_i, i = 1,\ldots,N$ meeting the following criteria:

- the radiance is roughly uniform in each domain,
- the solid angles are small and are inversely proportional to the average radiance, thus it is enough to test the visibility of the environment map with a single sample in each solid angle:

$$L^r(\vec{x},\vec{\omega}) = \sum_{i=1}^{N} \int\limits_{\Delta\omega'_i} L^{env}(\vec{\omega}') \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos\theta' \cdot v(\vec{x},\vec{\omega}')\ d\omega' \approx v(\vec{x},\vec{\omega}'_i) \cdot \tilde{L}_i^{env} \cdot a(\Delta\omega'_i,\vec{\omega}),$$

where

$$\tilde{L}_i^{env} = \int\limits_{\Delta\omega'_i} L^{env}(\vec{\omega}')\ d\omega'$$

is the *total incoming radiance* from solid angle $\Delta\omega'_i$, and

$$a(\Delta\omega'_i, \vec{\omega}) = \frac{1}{\Delta\omega'_i} \cdot \int_{\Delta\omega'_i} f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos\theta' \; d\omega'.$$

is the *average reflectivity* from solid angle $\Delta\omega'_i$ to viewing direction $\vec{\omega}$. In order to use this approximation, the following tasks need to be solved:

1. The directional domain should be decomposed meeting the prescribed requirements, and the total radiance should be obtained in them. Since these computations are independent of the objects and of the viewing direction, we can execute them in the preprocessing phase.

2. The visibility of the environment map in the directions of the centers of the solid angles needs to be determined. Since objects are moving, this calculation is done on the fly. Note that this step is equivalent to shadow computation assuming directional light sources.

3. The average reflectivity values need to be computed and multiplied with the total radiance and the visibility for each solid angle. Since the average reflectivity also depends on the normal vector and viewing direction, we should execute this step as well on the fly.

5.2.1 Decomposition of the environment map

For a static environment map, the generation of sample directions should be performed at loading time, making it a non time-critical task. The goal is to make the integral quadrature accurate while keeping the number of samples low. This requirement is met if solid angles $\Delta\omega'_i$ are selected in a way that the environment map radiance is roughly homogeneous in them, and their size is inversely proportional to this radiance. The task is usually completed by generating random samples with a probability proportional to the power of environment map texels, and applying *Lloyd's relaxation* [74] to spread the samples more evenly. A weighted version of the relaxation method may be used to preserve the density distribution. As the basic idea of Lloyd's relaxation is to move the sample points to the center of their respective Voronoi areas, the relatively expensive computation of the Voronoi mesh is necessary.

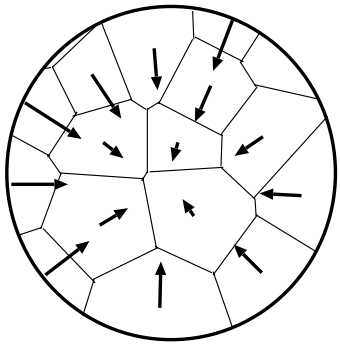


Figure 5.2: Radiance values of texels within the Voronoi areas are summed to compute their total radiance

Recently an extremely fast algorithm using *Penrose tiling* has been published to solve this problem [93]. This method should be considered if non-static environment maps are to be used. However, if we wish to assign the most accurate light power values to the sampled directions, we have to add up the contributions of those texels, which fall into the Voronoi region of the direction. In this case, Lloyd's relaxation means no significant overhead, as rigorously summing the texel

contributions takes more time. If this operation were to be done on a per-frame basis, then we should resort to random sampling of a smoothed map. As for the test application, we considered static environments only, and opted for a sample set generated by a two-dimensional Halton series, decimated according to the importance function, and relaxed using Lloyd's algorithm.

Figure 5.3: Final Delaunay grid on high dynamic range image. The centers of Voronoi cells are the sample points.

Centers of these cells define sample directions $\vec{\omega}_i'$ for which the visibility will be tested. Having identified the solid angles, the radiance is summed in their texels to obtain total incoming radiance $\tilde{L}_i^{env}$. In order to prepare for the fast calculation of the reflectivity, the approximate radius r_i of these cells are also obtained, which corresponds to the angle of view in which this subdomain is visible from the center of the environment map.

5.2.2 Visibility determination

Having decomposed the environment map to solid angles and deciding that the visibility is tested with a single sample in each subdomain, the problem is traced back to calculation the shadow caused by directional light sources. In order to render shadows effectively, a hardware-supported shadow technique has to be applied. *Depth map shadows* [138] generated for every discrete direction are well suited for the purpose. As we are unwilling to keep all depth maps in memory at once, or read all of them in a single pass, the rendering is decomposed into runs. When the resolution of the depth map is 256×256 and each element stores 16 bit depth values, one depth map requires 0.13 Mb storage space. For a typical scenario of 100 sampling points on the environment map, we use of 13 Mb of memory and we need to perform one hundred texture reads in the fragment shader.

5.2.3 Computation of the average reflectivities

When average reflectivity $a(\Delta\omega_i', \vec{\omega})$ from solid angle $\Delta\omega_i'$ to viewing direction $\vec{\omega}$ is computed, we have to make simplifications to make the method real time. We consider a standard diffuse + specular BRDF, where the specular part is defined by the Phong model:

$$f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos\theta' = k_d \cdot \cos\theta' + k_s \cdot \cos^n \psi,$$

where k_d is the diffuse reflection parameter, k_s is the specular reflectivity, n is the shininess of the surface, and ψ is the angle between the ideal reflection direction of view vector $\vec{\omega}$ and the illumination direction $\vec{\omega}'$. Let us first examine the diffuse reflection. Since in this case the

integrand is a cosine function, which has low variation, the integral is estimated from a single sample associated with the center of the Voronoi region:

$$a_d(\Delta\omega_i', \vec{\omega}) \approx k_d \cdot \cos\theta_c$$

where θ_c is the angle between the surface normal and direction $\vec{\omega}_i'$ corresponding to the center of this Voronoi region.

The case of specular reflection is more complicated, since it may have high variation when the Voronoi region contains the ideal reflection direction. Suppose that Voronoi cell $\Delta\omega_i'$ can be approximated by a spherical circle of angle r_i, and let us denote the angle between the ideal reflection direction and the center of the Voronoi cell by ψ_c. If the ideal reflection direction is not contained by $\Delta\omega_i'$, i.e. $\psi_c > r_i$, then a single sample is usually enough:

$$a_s(\Delta\omega_i', \vec{\omega}) \approx k_s \cdot \cos^n \psi_c.$$

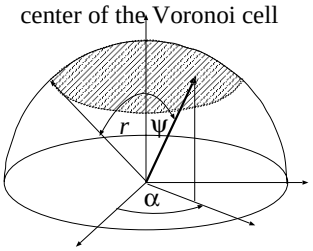


Figure 5.4: Average reflectance computation when $\psi_c = 0$

However, when the ideal reflection direction is contained by the Voronoi cell, the high variation of the function would result in a large error if only a single sample were considered. Let us first examine a special case, when the ideal reflection direction is equal to the center of the Voronoi cell, that is $\psi_c = 0$. In this special case, the average reflectivity can be analytically determined. We work with the angle ψ between the ideal reflection direction and angle α that is between the projection of $\vec{\omega}'$, and a reference direction on the plane perpendicular to the the ideal reflection direction (figure 5.4). Thus, when $\psi_c = 0$ the average specular reflectivity $a_s(\Delta\omega_i', \vec{\omega})$ is:

$$\frac{\int_{\alpha=0}^{2\pi}\int_{\psi=0}^{r_i} k_s \cdot \cos^n\psi \cdot \sin\psi \; d\psi d\alpha}{\int_{\alpha=0}^{2\pi}\int_{\psi=0}^{r_i} \sin\psi \; d\psi d\alpha} = k_s \cdot \frac{(1-\cos^{n+1} r_i)}{(n+1)\cdot(1-\cos r_i)}.$$

Note that if we took just a single sample selected by $\psi_c = 0$, then the approximation of the average albedo would be k_s, thus the ratio of the approximated and the real radiance would be

$$\frac{(n+1)\cdot(1-\cos r_i)}{(1-\cos^{n+1} r_i)}.$$

Figure 5.5 shows this ratio as a function of Voronoi cell size r_i for different shininess values. Note that the ratio can be very far from 1, which means a large error. This error is also visible in figure 5.6 that is rendered with using a single sample in each cell.

Note that we obtained an exact formula for the case of $\psi_c = 0$ and an approximation for $\psi_c > r_i$. In order to propose a good approximation for angles $0 < \psi_c < r_i$, we blend these results with weights $(1-\cos\psi_c)/(1-\cos r_i)$ and $1-(1-\cos\psi_c)/(1-\cos r_i)$, respectively, thus we obtain the following approximation for $a_s(\Delta\omega_i', \vec{\omega})$:

$$k_s \cdot \frac{(1-\cos^{n+1} r_i)\cdot(\cos\psi_c - \cos r_i)}{(n+1)\cdot(1-\cos r_i)^2} + k_s \cdot \cos^n\psi_c \cdot \frac{1-\cos\psi_c}{1-\cos r_i}.$$

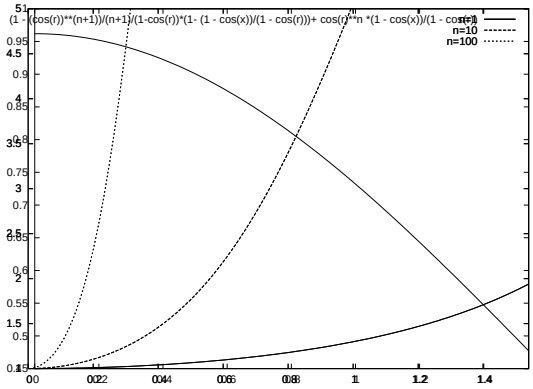


Figure 5.5: Ratio of the approximated and exact average reflectivities when $\psi_c = 0$ as a function of $r_i = 0 \ldots \pi/2$.

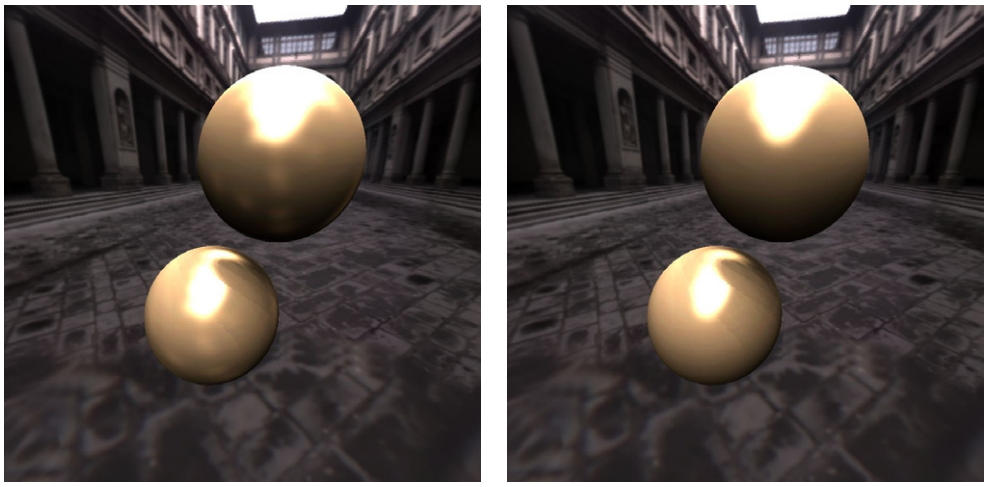


Figure 5.6: Spheres in Florence rendered with taking a single sample in each Voronoi cell, assuming directional light sources (left) and using our new formula for average reflectance over Voronoi cells (right). The shininess of the spheres is 100.

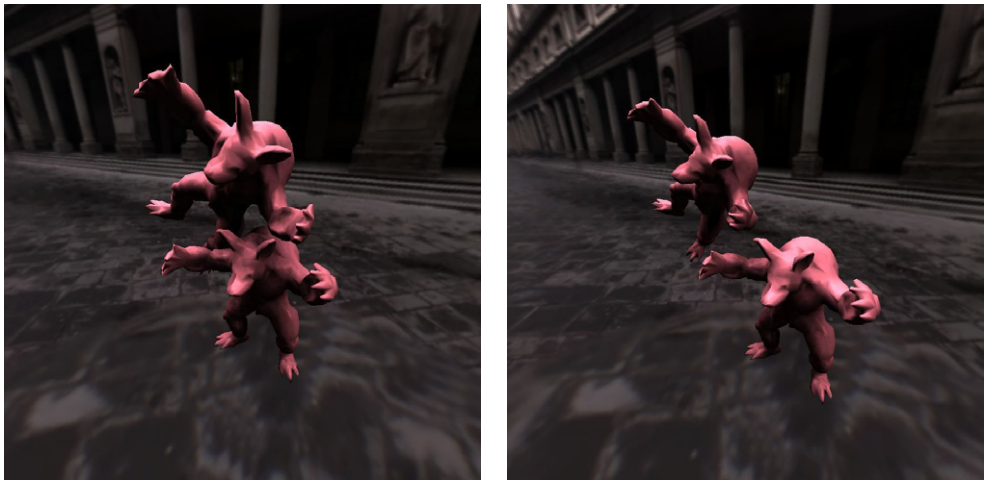
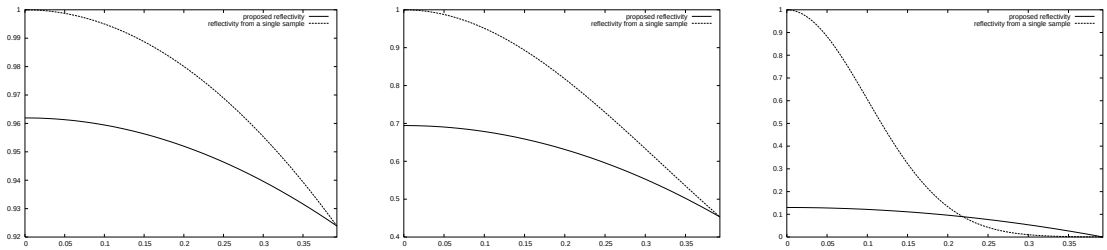


Figure 5.7: Two images from a video rendered on 18 FPS. Observe how the illumination of the standing Armadillo changes when the other Armadillo flies over.



$n = 1$ $n = 10$ $n = 100$

Figure 5.8: Comparison of the proposed average reflectance approximation with that of obtained with a single sample as functions of $\psi_c \in [0, r]$ for materials of different shininess. The size of the Voronoi cell is $r = \pi/8$.

Note that this formula is quite cheap computationally. During preprocessing we associate a spherical circle with each Voronoi cell and store the cosine of its angle ($\cos r_i$) together with center direction $\vec{\omega}_i$. When the given cell is processed for mirror direction $\vec{\omega}_R$, the scalar product $\cos \psi_c = \vec{\omega}_i \cdot \vec{\omega}_R$ is obtained and $\cos r_i$ is taken from the stored parameters.

5.3 Results

The algorithm has been implemented on an NV6800GT graphics card. With Shader Model 3.0, multiple directional samples could be handled in a single pass. This also implies more texture reads, but with tiling depth maps into bigger textures, the limitation on active samplers can also be avoided. The rendering times and the rendered images are shown in table 5.1 and in figure 5.7, respectively. Note that we could achieve interactive rates with a high number of samples, delivering artifact-free animation.

Scene	Sample points	Frames / sec
Armadillo	64	33 Fps
Armadillo	128	16.8 Fps
Two Armadillos	64	17.8 Fps
Two Armadillos	128	7.5 Fps

Table 5.1: Rendering times for the scene of Armadillos visiting Florence. A single Armadillo character consists of 15000 triangles.

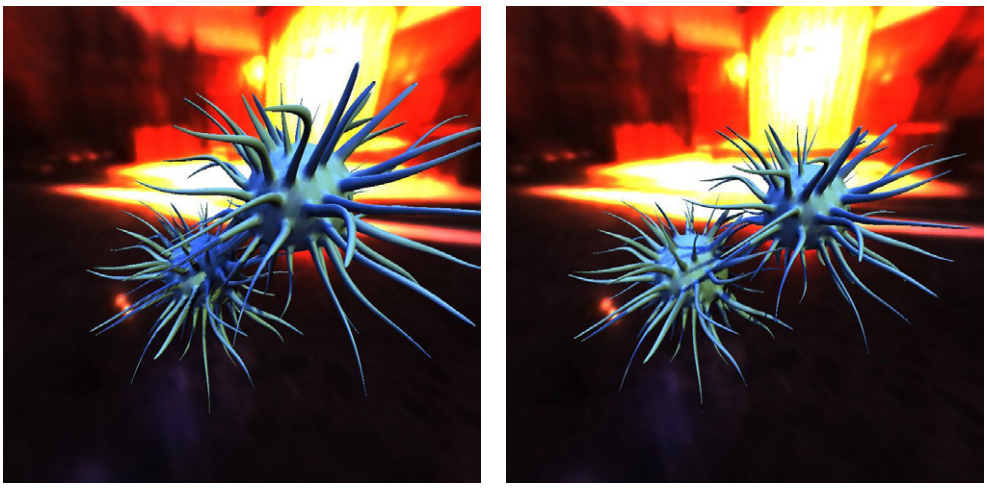


Figure 5.9: Viruses in the Irish pub scene (30 FPS)

5.4 Conclusions

This chapter proposed a real-time environment lighting algorithm that generates shadows and works well even for specular surfaces. With a low number of directional samples (32), slight shadow patterns were observable when the objects were moving, but the frame rates were more than enough for interactive rendering. A higher number of samples, producing fine results for animations, are more demanding, but still can provide real-time frame rates.

Chapter 6

Photon Map Filtering tool

Photon mapping methods obtain the indirect illumination of a point by finding those photon hits that arrived at the neighborhood of the point on the object surface. In this report we propose a method that stores the photon hits in a texture of the graphics hardware and replaces the traditional kd-tree based neighborhood searches by the filtering of this texture. This step finds the irradiance of all points (i.e. all texels) simultaneously in a single step, thus the average irradiance of a point can be obtained by a single texture lookup. Using this approach we can port the final gathering step of photon mapping to the graphics hardware (GPU). The CPU is only responsible for generating new light paths and updating the unfiltered photon map. Thanks to the optimal subdivision of the computation work between the the CPU and the GPU, the proposed algorithm can render globally illuminated scenes interactively.

6.1 Photon mapping

Photon mapping is a two-phase global illumination algorithm [56, 55, 57]. In the first phase a lot of light paths are generated originating at the light sources and bouncing at the surfaces randomly. The random generation of the paths is usually governed by BRDF sampling and Russian roulette. At each surface hit the Monte Carlo estimation of the incoming power is computed and stored. The data structure representing these hits is called the *photon-map*. The photon-map is usually organized in a *kd-tree* to support efficient photon retrieval. During this retrieval process we need those photons that are in the neighborhood of the point of interest. A photon hit is stored with the power of the photon on different wavelengths, position, direction of arrival, and with the surface normal.

The photon map represents the indirect illumination, which can be taken into account when the reflected radiance of a point is obtained. This calculation is called *final gathering*. Suppose we need to determine reflected radiance L of point $\vec{x}$ in direction ω. The gathering phase is based on the following approximation of the light transport operator:

$$L(\vec{x},\omega) = \int_{\Omega'} L^{in}(\vec{x},\omega') \cdot f_r(\omega',\vec{x},\omega) \cdot \cos\theta' \; d\omega' = \int_{\Omega} \frac{d\Phi(\vec{x},\omega')}{dA \cos\theta' d\omega'} \cdot f_r(\omega',\vec{x},\omega) \cdot \cos\theta' \; d\omega' \approx$$

$$L(\vec{x},\omega) = \int_{\Omega'} L^{in}(\vec{x},\omega') \cdot f_r(\omega',\vec{x},\omega) \cdot \cos\theta' \; d\omega' \approx \sum_{i=1}^{n} \frac{\Delta\Phi(\vec{x}_i,\omega_i')}{\Delta A} \cdot f_r(\omega_i',\vec{x},\omega), \qquad (6.1)$$

where L^{in} is the incoming radiance, f_r is the BRDF, θ' is the angle between the surface normal and the incoming direction, and $\Delta\Phi(\vec{x}_i,\omega_i')$ is the power of a photon landing at $\vec{x}_i$ of surface ΔA from direction ω_i'. The $\Delta\Phi$ and ΔA quantities are approximated from the photons in the neighborhood of $\vec{x}$ in the following way. A sphere centered around $\vec{x}$ is extended until it contains

n photons. If at this point the radius of the sphere is r, then the intersected surface area is $\Delta A = \pi r^2$ (see Figure 6.1).

This original version of the photon-map algorithm has several drawbacks, including the large storage space needed for the photon hits, the time consuming final gathering and the filtering artifacts caused by using the n photons nearby. In order to reduce these artifacts, we should take into account only those photon hits that arrived at the same surface, or more precisely, when the surface normal associated with the photon hit is similar to that of the shaded point. On the other hand, the approximation of the area from where the photons are gathered by an intersection circle can be very inaccurate at surface boundaries. Here a more precise area calculation is needed as suggested by [52]. Christensen [19] proposed another improvement to compute diffuse interreflections. In the preprocessing phase, the irradiance is estimated at each photon hit from the other photon hits that are nearby. Thus during final gathering, we do not have to find the n nearest photons, but only the closest one where the normal vector is similar to the normal vector of the given point.

6.2 Photon tracing with improved density estimation

Algorithms like photon mapping can also be discussed as tools to solve a density estimation problem [114]. Photon hits represent a sampling of the irradiance, from which a smooth reconstruction of the reflected radiance should be generated. To achieve this, a convolution operation executing low pass filtering is defined:

$$L(\vec{x}, \omega) \approx \sum_{i=1}^{n} \Delta\Phi(\vec{x}_i, \omega_i') \cdot f_r(\omega_i', \vec{x}, \omega) \cdot k(\vec{x}_i - \vec{x}), \tag{6.2}$$

where $\vec{x}_i$ is the location of the ith photon hit and k is the filter kernel, which can be integrated to 1. The density estimation can be implemented both with a gathering [56] or a shooting approach [119]. In case of shooting, the photon hits are splat as small textured quads onto the object surfaces and alpha blending is used to add up their contribution.

6.3 Computation of the reflected radiance by texture filtering

In our approach, the photon hits are stored in texture space [6]. Since neighborhood searches are also executed in the texture space, we look for a neighborhood of the respective texel. Usually if two points belong to the same surface and are close, their texels will also be close. Thus when we gather photon hits from the neighboring texels, we can assume that these hits would be close to the shaded point in object space as well. However, it can happen that two 3D points are far while their respective texels are close. Obviously in this case we cannot use the photon hits of the other point for the radiance calculation. To recognize these cases, the surface identities (id) should also be stored in the texture. When the neighborhood is built and if the ids of neighboring texels are different, the neighborhood must not be extended with those texels. On the other hand, we should also store the surface normals to avoid considering those photon hits that have significantly different orientation.

Having established an appropriate neighborhood in texture space, the surface area corresponding to this neighborhood is computed as required by equation 6.1. This computation can again be supported by information stored in texture. During preprocessing the value of the surface area corresponding to a given texel is computed and stored. The texel areas are calculated as the ratio of the area of the surface elements (triangles) in world space and in texture space. When the neighborhood of a texel is built, these area values are summed.

Summarizing, our method handles the steps of photon mapping as follows:

1. Photon tracing on CPU.

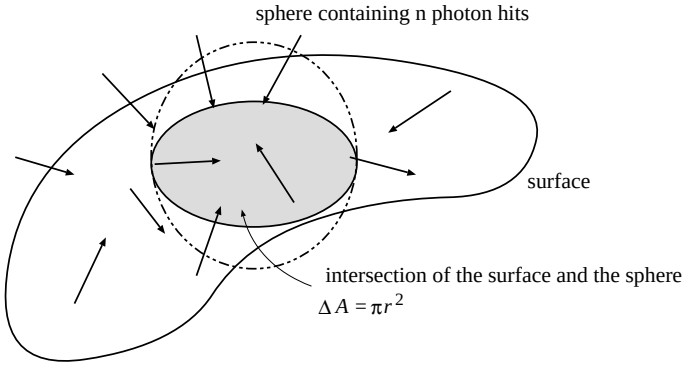


Figure 6.1: Gathering photons by extending the neighborhood of the actual point for reflected radiance calculation

2. Photon hit conversion into textures.
3. Final gathering step in two passes:
   (a) filtering the view dependent reflected radiance at those points which correspond to texel centers (see section 6.4 for details), and
   (b) mapping this texture on the object surface during a normal, local illumination rendering.

6.4 Computation of the reflected radiance

The reflected radiance computation (the photon map filtering method) is implemented as a pixel shader program on the GPU. Our proposed algorithm works with the following textures:

- `diffusemap` (the diffuse reflectivities),
- `photonhit` (the original photon hit powers),
- `photondir` (the incoming direction of the hits),
- `surface` (the surface id and the area of the surface in channel r and g, respectively),
- `normmap` (the x, y and z coordinates of surface normals in the channels r, g and b, respectively).

The pixel shader of the radiance computation checks texels in the $(2N+1) \times (2N+1)$ square neighborhood of sample point of texture coordinate `pIN.uv`, and decides whether they could be added to the neighborhood. Using variable filter kernels, we need a texture (`filter`) storing the filter values in the $(2N+1) \times (2N+1)$ texel neighborhood. We cannot assume that the kernel integrates to one, because not necessarily all texels will be taken into account due to different surfaces and normal vectors. To handle this problem, a normalization constant is calculated similarly to the `area`. For texels belonging to the neighborhood, their hit power is multiplied with the BRDF of the point of interest and the resulting reflected radiance is added to the reflected radiance of the point. The BRDF function takes incoming direction of the photon `indir` and eye direction `pIN.eye`, which is computed by the vertex shader and is interpolated by the graphics hardware before calling this pixel shader code.

The presented program works with textures storing the normal vector, the surface id, and surface area of those points they are associated with. These textures can be generated using

render-to-texture technology. Figure 6.2 shows the normal and area maps of one of our test scenes.

The pixel shader program of our algorithm is the following:

```
float3 surf0 = tex2d(surface, pIN.uv);
float3 norm0 = tex2d(normmap, pIN.uv);
float area = 0;
for(int dy = -N; dy <= N; dy += 1)
  for(int dx = -N; dx <= N; dx += 1) {
    float2 uv1 = pIN.uv + float2(dx, dy);
    float  k = tex2d(filter, uv1);
    float2 surf = tex2d(surface, uv1);
    float3 norm = tex2d(normmap, uv1);
    if(surf.r == surf0.r && dot(norm, norm0) > threshold ) {
      area += surf.g * k;
      float3 pow = tex2d(photonhit, uv1) * k;
      float3 indir = tex2d(photondir, uv1);
      rad += pow * BRDF(indir,pIN.uv,pIN.eye);
    }
  }
}
return rad/area;
```

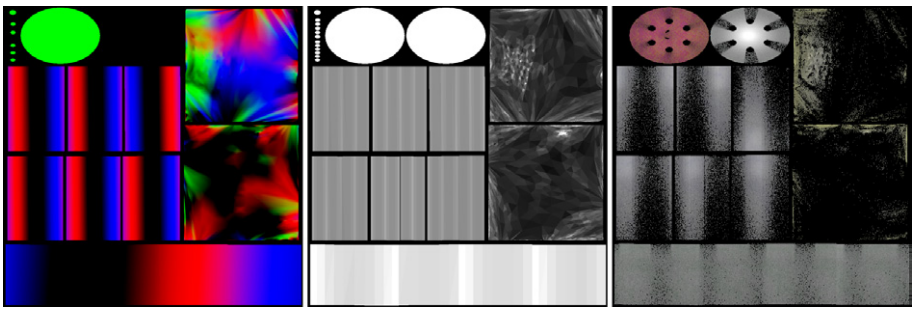


Figure 6.2: The normal (left), the area (middle) and the photon map of the armadillo scene

To speed up and optimize the algorithm we can take another approach to implement the filtering mechanism. We do not calculate filtered photon map texels in a pixel shader program using a lot of texture look-ups in loops, but we draw the unwrapped photon map many times on itself with predefined offsets (corresponding to the kernel elements) using alpha blending. See Figure 6.3.

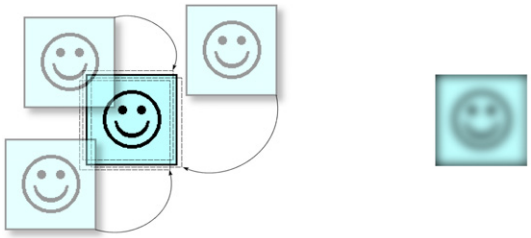


Figure 6.3: Filtering by alpha blending. On the left the original texture is blended many times onto itself offseted and weighted corresponding to the position and value of the kernel elements. On the right the resulting map can be seen.

6.5 Generating the photon textures

Having generated all photon hits by simulating random walks starting at the light sources on CPU and computed the texture coordinates of these hits, they are written into the maps of photon powers and incoming directions and uploaded to the GPU. Photon directions are not needed if the surfaces are diffuse, but play a significant role in case of specular materials. Note that it can happen that two or more photon hits fall on the same texel, when the photon hits are very dense at this surface region. The probability of this event grows as we decrease the resolution of the photon map texture. This is not a problem for diffuse reflections, since their BRDF is independent of the incoming direction, thus powers can be added. However, the direction dependent specular reflections cannot be handled in this way, but we should purge photon hits and reduce their density where they are highly concentrated. This operation should maintain the total power arriving at this surface region, i.e. when photons are ignored, the power of the kept photons must be scaled up.

In order to investigate this problem mathematically, let us revisit equation 6.2 expressing the reflected radiance, and group the terms according to the texels:

$$L(\vec{x},\omega) \approx \sum_{i=1}^{n} \Delta\Phi(\vec{x}_i,\omega_i') \cdot f_r(\omega_i',\vec{x},\omega) \cdot k(\vec{x}_i-\vec{x}) =$$

$$\sum_{j=1}^{N}\sum_{i=1}^{n_j} \Delta\Phi(\vec{x}_j,\omega_{ji}') \cdot f_r(\omega_{ji}',\vec{x},\omega) \cdot k(\vec{x}_j-\vec{x}),$$

where texel j is expected to store photon hits $\Delta\Phi(\vec{x}_j,\omega_{ji}')$ for all $i=1,\ldots,n_j$.

In case of diffuse reflections, the sum belonging to a single texel is:

$$\sum_{i=1}^{n_j} \Delta\Phi(\vec{x}_j,\omega_{ji}') \cdot f_r(\vec{x}) \cdot k(\vec{x}_j-\vec{x}) = f_r(\vec{x}) \cdot k(\vec{x}_j-\vec{x}) \cdot \sum_{i=1}^{n_j} \Delta\Phi(\vec{x}_j,\omega_{ji}'),$$

thus the powers of the photon hits can be simply added.

For specular reflections, let us approximate the sum belonging to a single texel applying Monte Carlo methods. We find a probability density p_{ji} so that $\sum_{i=1}^{n_j} p_{ji} = 1$, sample an integer i^* with this density, and approximate the sum as:

$$\sum_{i=1}^{n_j} \Delta\Phi(\vec{x}_j,\omega_{ji}') \cdot f_r(\omega_{ji}',\vec{x},\omega) \cdot k(\vec{x}_j-\vec{x}) \approx k(\vec{x}_j-\vec{x}) \cdot \frac{\Delta\Phi(\vec{x}_j,\omega_{ji^*}') \cdot f_r(\omega_{ji^*}',\vec{x},\omega)}{p_{ji^*}}.$$

In order to make the variance of this random estimation small, selection probabilities are set proportional to the luminance of the photon hits. Let us denote the luminance of wavelength dependent power Φ by $\mathcal{L}(\Phi)$. The probability of selecting photon hit i is:

$$p_{ji} = \frac{\mathcal{L}(\Phi(\vec{x}_j,\omega_{ji}'))}{\sum_{l=1}^{n_j}\mathcal{L}(\Phi(\vec{x}_j,\omega_{jl}'))}.$$

Thus, having sampled hit i^*, the Monte Carlo estimate of the reflected radiance is:

$$k(\vec{x}_j-\vec{x}) \cdot \frac{\Delta\Phi(\vec{x}_j,\omega_{ji^*}')}{\mathcal{L}(\Phi(\vec{x}_j,\omega_{ji}'))} \cdot f_r(\omega_{ji^*}',\vec{x},\omega) \cdot \sum_{l=1}^{n_j}\mathcal{L}(\Phi(\vec{x}_j,\omega_{jl}')).$$

The Monte Carlo estimate introduces a small variance in the estimation. However, filtering reduces this variance and makes the variance of neighboring pixels correlated, thus we can still avoid noise artifacts in the image.

6.6 Discussion and Conclusions

We used two test scenes (Figure 6.4). Initially two million photons were emitted. The armadillo scene$^{(1)}$ contains 3335, the Cornell box scene$^{(2)}$ contains 144 facets. The computer configuration was: WinXP, AMD AthlonXP 1492 MHz CPU, NVIDIA GeForce 6800 GT graphics card. The photon maps and the area map are generated off-line on CPU, the normal map is created on GPU only once before the normal work-flow. For details see Table 6.1.

The proposed approach can be used for interactive walk-through animations and even in interactive global illumination. In the latter case even photon maps should be regenerated between frames. In order to cope with the speed requirements, we do not rebuild all photon paths in each frame, just those that have been changed with high probability. These photon paths are identified by selective photon tracing [34].

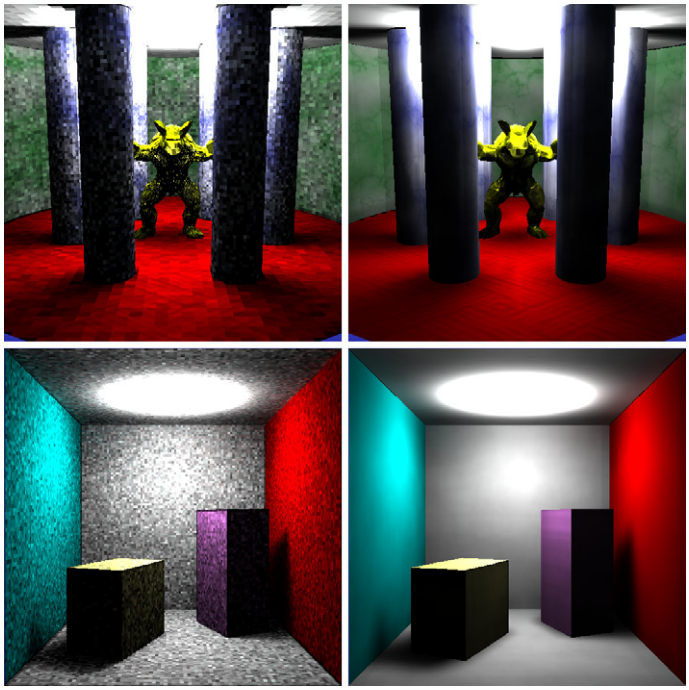


Figure 6.4: The armadillo and Cornell box scene (with unfiltered and filtered photon map). Photon map resolution: 256×256, kernel size: $7 \times 7^{(1)}$ and $11 \times 11^{(2)}$

	Photon map	**Filter kernel**		
resolution	512×512	11×11	7×7	3×3
FPS$^{(1)/(2)}$		**0.8 / 1.6**	**2.0 / 3.8**	**11 / 20**
resolution	256×256	11×11	7×7	3×3
FPS$^{(1)/(2)}$		**1.8 / 3.6**	**4.4 / 8.9**	**22 / 45**

Table 6.1: Speed and performance results.

Chapter 7

Stochastic Iteration Tool

This chapter presents an algorithm for the glossy global illumination problem, which runs on the Graphics Processing Unit (GPU). In order to meet the architectural limitations of the GPU, we apply randomization in the iteration scheme. Randomization allows to use that set of the possible light interactions, which can be efficiently computed by the GPU, and makes it unnecessary to read back the result to the CPU. Instead of tessellating the surface geometry, the radiance is stored in texture space, and is updated in each iteration. The visibility problem is solved by hardware shadow mapping after hemicube projection. The shooter of the iteration step is selected by a custom mipmapping scheme, realizing approximate importance sampling. The variance is further reduced by partial analytic integration. The result is a fast program that can render moderately complex scenes interactively.

7.1 Introduction

This chapter presents the GPU implementation of a stochastic global illumination algorithm. Unlike previous approaches, we emphasize here the application of randomization as a tool to solve general problems on the GPU. Randomization, also called Monte Carlo method, has proven to be very successful to solve high dimensional integration problems. On the other hand, randomization also allows to replace a computation by another simpler calculation which gives back the result of the original computation just in an average case. In this sense, randomization provides us enormous freedom to simplify or to restructure algorithms. In case of special hardware the goal of randomization is to trace back the computation to random steps, which can efficiently be carried out by the GPU hardware.

The graphics hardware uses the z-buffer algorithm for visibility checks. This algorithm processes the geometry (triangles) in an arbitrary order, but the valid visibility information is available only if the complete geometry has gone through the pipeline. When a point of a triangle is processed, it is not yet known whether or not this point is visible. Thus computations involving the visibility function require two passes. In the first pass the visibility function is computed, then in the second pass the computation can depend on this valid visibility information. This two pass approach resembles to shadow map methods [25].

GPU algorithms are data driven, which means that the result can only be written into the pixel (or texel), which is at the end of the current processing pipeline. Thus we should know the target pixel even at the vertex shader where the pixel data will be written to. Consider the example of ray-shooting, which selects an origin and shoots a ray into the scene, and finally adds the carried power to that surface area, which is hit by the ray. Such shooting type algorithms cannot be implemented by the pixel shader since the location of the result (i.e. the receiver of the transfer) becomes known after the calculation, while a pixel shader is allowed to write only to its own pixel. However in gathering, a ray is shot from the ray origin, and the radiance of the hit point is transferred to the origin of the ray. Thus the location where the result should be written

is known in advance, which meets the requirements of the GPU pipeline.

Considering this fact, gathering type global illumination algorithms may seem to be better for GPU implementation. However, it is known that shooting type approaches have potentially faster convergence [107, 8], thus — despite to the difficulties — shooting is still worth implementing on the GPU. We concluded that due to the temporarily invalid visibility information, computations based on visibility consist of at least two passes. The problem of shooting can be solved if in the second pass we set a transformation that visits potentially all possible result locations, and based on the visibility information of the previous pass, these result locations are either updated or left unchanged.

Because of the fixed register set, the amount of data arriving at a pixel shader is limited, thus many partial results cannot be computed and summed in a single pass. A usual trick to attack this problem is computing a single term of the sum in each pass, which is added with a copy image of the target buffer. This copy is mirrored at the end of each pass. This technique, which is called *ping-pong buffering*, can be used only in special circumstances where the number of added terms is rather limited, since the number of the required passes is equal to the number of terms to be summed.

This problem can be solved by Monte Carlo summation. The sum is approximated by a randomly selected term, which is divided by the selection probability. Thus the approximation can be obtained in a single pass no matter how many terms the sum has. The randomization of the summation introduces some random noise in the result, which can be eliminated by averaging the temporary results. This averaging, on the other hand, can be very efficiently implemented by the ping-pong technique. The previous result is read from one of the ping-pong buffer, is averaged with the actual result, and the result is written to the other ping-pong buffer.

Monte Carlo methods have been successfully applied to solve global illumination problems since they can efficiently approximate high dimensional integrals inherent in global illumination [8, 66]. Note, however, that now randomization has a different objective. Randomization is regarded as a technique that can substitute a complex operation (summation or integration) by a much simpler technique, which can be executed by the available hardware.

The chapter is organized as follows. Section 7.2 discusses previous work on the hardware support for global illumination, and the new algorithm is placed among the existing methods. Section 7.3 presents the new method and its implementation details. Section 7.5 analyzes its performance, and finally the chapter draws some conclusions.

7.2 Global illumination with hardware support

Global illumination algorithms aim at the solution of the rendering equation

$$L(\vec{x},\omega) = E(\vec{x},\omega) + R(\vec{x},\omega),$$

which expresses the radiance $L(\vec{x},\omega)$ of point $\vec{x}$ at direction ω as a sum of emission E and reflection R of the radiances of all points that are visible from point $\vec{x}$. The reflection of the radiance of visible points is expressed by an integral operator

$$(\mathcal{T}_{f_r} L) = \int_S L(\vec{y},\omega_{\vec{y}\to\vec{x}}) \cdot f_r(\omega_{\vec{y}\to\vec{x}},\vec{x},\omega) \cdot G(\vec{x},\vec{y})\ dy, \tag{7.1}$$

which is also called the *light transport operator* [59]. In this equation S is the set of surface points, f_r is the BRDF and

$$G(\vec{x},\vec{y}) = v(\vec{x},\vec{y}) \cdot \frac{\cos\theta_{\vec{x}} \cdot \cos\theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2}$$

is the *geometric factor*, where $v(\vec{x},\vec{y})$ is the mutual visibility indicator, which is 1 if points $\vec{x}$ and $\vec{y}$ are visible from each other and zero otherwise, $\theta_{\vec{x}}$ and $\theta_{\vec{y}}$ are the angles between the surface normals and direction $\omega_{\vec{y}\to\vec{x}}$ that is between $\vec{x}$ and $\vec{y}$.

The solution of the rendering equation requires general purpose instructions and is thus usually computed on the CPU [35]. Our goal is to take advantage of the huge computation power of the GPU for the solution of the rendering equation. In order to do so, we should transform the algorithm according to the capabilities of the GPU.

The CPU-based solution algorithms can be classified as random walk [59] and iteration techniques. The GPU support of random walk algorithms has been examined in [100]. Since iteration algorithms are conceptually closer to local illumination, which is originally supported by GPUs, we believe that iteration algorithms are better candidates for GPU implementation.

Iteration techniques are based on the fact that the solution of the rendering equation is the fixed point of the following iteration scheme:

$$R_m = \mathcal{T}_{f_r} L_{m-1} = \mathcal{T}_{f_r}(E + R_{m-1}).$$

If this scheme is convergent, then the solution can be obtained as a limiting value:

$$R(\vec{x}, \omega) = \lim_{m \to \infty} R_m(\vec{x}, \omega).$$

Iteration works with the complete radiance function, whose temporary version should be represented somehow. The classical approach is the *finite-element method*, which approximates the radiance function in a function series form. In the simplest diffuse case we decompose the surface to small elementary surfaces $A^{(1)}, \ldots, A^{(n)}$ and apply a piece-wise constant approximation, thus the reflected radiance function is represented by the reflected radiance of these patches, that is by $R^{(1)}, \ldots, R^{(n)}$. In the glossy case the radiance is also a function of the direction, which may be handled by applying a similar finite element decomposition in the directional domain [115], but this approach would increase the storage requirements considerably. Fortunately, applying randomization, we can solve the glossy global illumination problem without introducing any finite elements in the directional domain [123].

If these elementary surfaces are small, we can consider just a single point of them in the algorithms, while assuming that the properties of other surface points are similar. Surface properties, such as the BRDF and the emission can be given by directional functions $f_r^{(1)}(\omega^{in}, \omega^{out}), \ldots, f_r^{(n)}(\omega^{in}, \omega^{out})$, and $E^{(1)}(\omega^{out}), \ldots, E^{(n)}(\omega^{out})$, in each positional finite element. These directional functions can be defined by material properties, such as the diffuse and specular albedos and the shininess. We shall use a physically plausible diffuse-specular model [89] defined by the diffuse and specular albedos and the shininess, and assume that the lights are diffuse, but other material and light models could also be used in the proposed method.

Iteration simultaneously computes the interaction between all surface elements, which has quadratic complexity in terms of the number of finite elements, and is hard to implement on the GPU. This problem can be attacked by special iteration techniques, such as Southwell iteration (also called progressive radiosity), hierarchical radiosity, or by randomization. Southwell iteration computes the interaction of the element having the highest unshot radiosity and all other surface elements [23]. It is quite simple to implement but has also quadratic complexity [126]. Methods supporting progressive radiosity with the graphics hardware is as old as the method itself. The form factors between the shooter and all other patches can be computed with the hemicube method that identifies the visible patches by the z-buffer hardware [22].

The quadratic complexity can be reduced by hierarchical [5] and Monte Carlo [113, 88] approaches. The limited capabilities of the hardware seem not to be appropriate for the implementation of demanding hierarchical approaches. Monte Carlo techniques, on the other hand, can greatly benefit from the hardware. Stochastic perspective ray bundles use the same elementary step as progressive radiosity [124], while parallel ray-bundles [88] can be traced by depth peeling [127] or can be made appropriate for normal z-buffer rendering by applying further randomization [79]. All of these early attempts to use the hardware suffered from the limitation of the fixed pipeline, which did not allow the complete algorithm to be executed on the graphics card. Read-backs transferring data to the CPU, however, significantly reduced the speed. An exception is the instant radiosity [65], which computed the last eye-step of a random-walk algorithm using the standard graphics hardware, thus it could eliminate the read-backs.

The emergence of programmable graphics hardware has made it possible to implement the complete algorithm without the performance penalty of read-backs. CPU algorithms usually decompose general surfaces to triangular patches. However, in GPU approaches this is not feasible since GPU processes patches independently thus the computation of the interdependence of patch data is difficult. Instead, the radiance function can be stored in a texture [6, 91]. An elementary surface area $A^{(i)}$ is the surface which is mapped onto texel i. Full matrix radiosity [17], progressive refinement [91], substructuring [25], and final gathering with parallel ray bundles [43] have already been successfully implemented on the graphics hardware. However, as these algorithms have quadratic complexity they could achieve interactive frame rates on only simple models represented by at most 100×100 texture resolution. Due to the complicated operations, the CPU implementation has turned out to be not slower than the GPU algorithm in many applications [17].

Unlike these previous methods, we implemented the implementation of a Monte Carlo global illumination algorithm on the graphics hardware. The formal basis of such approaches is the stochastic iteration, which was originally proposed for the solution of the linear equations [88, 107, 8], then extended for the solution of integral equations [123]. Stochastic iteration means that in the iteration scheme a random transport operator $\mathcal{T}^*_{f_r}$ is used instead of the light-transport operator $\mathcal{T}_{f_r}$. The random transport operator has to give back the light-transport operator in the expected case:

$$R_m = \mathcal{T}^*_{f_r}(E + R_{m-1}), \qquad E[\mathcal{T}^*_{f_r} L] = \mathcal{T}_{f_r} L.$$

Note that such an iteration scheme does not converge, but the iterated values will fluctuate around the real solution. To make the sequence converge, we obtain an image estimate at each iteration step, and compute the final result as the average of these image estimates:

$$C = \mathcal{M}E + \frac{1}{m} \cdot \sum_{k=1}^{m} \mathcal{M}R_k,$$

where $\mathcal{M}$ is the measuring operator computing the image seen from the camera from the actual radiance distribution. It means that the implementation of a stochastic iteration algorithm requires the storage of the evolving image (a value C for each pixel), and the temporary version of the reflected radiance function $R_k(\omega)$. The complexity of the representation of this function depends on the properties of the random transport operator, and its simplification is an important criterion to find a good randomization.

The core of all stochastic iteration algorithms is the definition of the random transport operator. We prefer those random operators, that can be efficiently computed on the GPU, introduce small variance, and does not require the complete storage of the radiance function for all points and directions.

To make our algorithm efficient we implemented the random hemicube shooting (perspective ray bundles) [124] on the GPU. Unlike the CPU based method, we store both the actual irradiance and the accumulated radiance for the camera in textures, and minimize the data storage requirements. During the porting of the algorithm, we had to solve the problems of texture based non-diffuse radiance representation and importance sampling. We significantly simplified the data structures as needed by the GPU and stored the temporary reflected radiance and the evolving image in floating point textures. Note that although the new method solves the non-diffuse global illumination problem, thanks to randomization, it still uses two small RGBA textures. The GPU algorithm and the radiance representation are practically independent of the patch decomposition, surfaces describe only the geometry. We may call this approach as *patchless rendering*, since having detected the visibility between two points, all computations are done on texels independently of the surfaces. It allows us to work with the original geometry, no patch subdivision is necessary, but the finite element representation of the radiance can be high. We consider this as one of the main novelties of the method comparing to previous CPU based stochastic iteration algorithms.

On the other hand, with respect to previous GPU based radiosity algorithms, the main novelty is the introduction of randomization. Randomization allowed the rendering of non-diffuse scenes

without any increase in required texture space. On the other hand, we can benefit the sub-quadratic complexity of Monte Carlo methods, which makes randomized approaches a definite winner for more complex scenes. Moreover, randomization can be regarded as a general tool to trace back the steps of general algorithms to those, which can be executed by the GPU.

The result of stochastic approaches is a random variable, that is, the error is characterized by its variance. The variance can generally be reduced by applying a sampling scheme, which mimics the integrand, as suggested by *importance sampling*. Since in GPU approaches, the elementary operations are limited, we usually cannot apply very good sampling strategies. What we can do, however, is to apply partial analytic integration or to compensate the error knowing the desired and actual sampling densities. This compensation strategy is called *weighted importance sampling* [96]. Weighted importance sampling has already been used in computer graphics [11, 125], but now we apply the basic idea in a different context.

7.3 The new GPU based global illumination algorithm

In order to obtain a random approximation of equation 7.3, two points $\vec{x}$ and $\vec{y}$ need to be obtained with probability density $p(\vec{x},\vec{y})$, and use the following primary estimate:

$$L_m^{(i)} = \left(\frac{1}{A_i} \cdot v(\vec{x},\vec{y}) \cdot L(\vec{y}) \cdot f_r(\vec{x}) \cdot \frac{\cos\theta'_{\vec{x}} \cdot \cos\theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2} \right) / p(\vec{x},\vec{y}). \tag{7.2}$$

According to the rendering equation, the reflected radiance can be obtained by evaluating a surface integral. In a GPU algorithm, the surface is decomposed to small patches corresponding texels of the texture map. Let us first assume that the resolution of this texture map is high enough to make all elementary surfaces small, and thus we can check the mutual visibility of two elementary surfaces by inspecting only their centers. In this case the update of the reflected radiance representation in a single iteration can be approximated in the following way:

$$R_m^{(i)}(\omega) = \frac{1}{A^{(i)}} \cdot \int_{A^{(i)}} \int_S L_{m-1}(\vec{y}, \omega_{\vec{y}\to\vec{x}}) \cdot f_r^{(i)}(\omega_{\vec{y}\to\vec{x}}, \omega) \cdot G(\vec{x},\vec{y})\ dydx \approx$$

$$\sum_{j=1}^{n} L_{m-1}^{(j)}(\omega_{\vec{y}_j\to\vec{x}_i}) \cdot f_r^{(i)}(\omega_{\vec{y}_j\to\vec{x}_i}, \omega) \cdot G(\vec{y}_j,\vec{x}_i) \cdot A^{(j)}, \tag{7.3}$$

where $\vec{y}_j$ and $\vec{x}_i$ are the centers of elementary surfaces $A^{(j)}$ and $A^{(i)}$, respectively.

$$\frac{A^{(j)}}{K} \cdot \sum_{k=1}^{K} \sum_{j=1}^{n} L_{m-1}^{(j)} \cdot f_r^{(i)}(\omega_{\vec{y}_j\to\vec{x}_i}, \omega) \cdot G(\vec{y}_j, \vec{x}_{i,k}), \tag{7.4}$$

where $\vec{y}_j$ is the center of elementary surface $A^{(j)}$, and we placed K number of $\vec{x}_{i,k}$ samples on elementary surface $A^{(i)}$. Note that we applied the *disc-to-point form factor* approximation to reduce the double integral over $\vec{y}$ to a single integral fixing $\vec{y}$ to center $\vec{y}_j$ of elementary surface $A^{(j)}$, then approximated the single integral by a discrete sum.

$$\frac{A^{(j)}}{K} \cdot \sum_{k=1}^{K} \sum_{j=1}^{n} L_{m-1}^{(j)} \cdot f_r^{(i)}(\omega_{\vec{y}_j\to\vec{x}_i}, \omega) \cdot v(\vec{x}_{i,k}, \vec{y}_j) \cdot \frac{\cos\theta_{\vec{x}_{i,k}} \cdot \cos\theta_{\vec{y}_j}}{|\vec{x}_{i,k}-\vec{y}_j|^2 + A^{(j)}/pi}, \tag{7.5}$$

and $\vec{x}_i$ are the centers of elementary surfaces A_j and A_i, respectively, and

$$F_{ij} = \frac{1}{A^{(i)}} \cdot \int_{A^{(i)}} \int_{A^{(j)}} v(\vec{x},\vec{y}) \cdot \frac{\cos\theta_{\vec{x}} \cdot \cos\theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2 \cdot \pi}\ dxdy \approx \frac{A^{(j)}}{A^{(i)}} \cdot \int_{A^{(i)}} v(\vec{x},\vec{y}_j) \cdot \frac{\cos\theta_{\vec{x}} \cdot \cos\theta_{\vec{y}_j}}{|\vec{x}-\vec{y}_j|^2 \cdot \pi + A^{(j)}}\ dx \approx$$

$$\frac{A^{(j)}}{K} \cdot \sum_{k=1}^{K} v(\vec{x}_{i,k}, \vec{y}_j) \cdot \frac{\cos\theta_{\vec{x}_{i,k}} \cdot \cos\theta_{\vec{y}_j}}{|\vec{x}_{i,k} - \vec{y}_j|^2 \cdot \pi + A^{(j)}}.$$

is the *form factor.* placing K number of $\vec{x}_{i,k}$ samples on elementary surface i.

In order to efficiently detect visibility between points $\vec{y}_j$ and $\vec{x}_i$, we run a GPU algorithm that is quite similar to depth buffer shadow methods. First a depth image is computed placing the camera at $\vec{y}_j$ and associating the pixels with the cells of a discretized hemicube. Then in the second pass, the center of every surface element $A^{(i)}$ is checked whether or not this surface element is farther from $\vec{y}_j$ than the closest surface projecting onto the same pixel as the center of $A^{(i)}$. If a surface element (i.e. a texel) turns out to be visible, then the contribution from the shooter is added, otherwise this contribution is zero.

Equation 7.3 — which has to be computed in every iteration — contains a sum for every finite element i and for each iteration. This sum is estimated randomly since the random estimator completely eliminates summation that would pose problems to the GPU. A surface element $A^{(j)}$ is selected with probability p_j, and $\vec{y}_j$ is set to its center. This randomization allows to compute the interaction between shooter surface element $A^{(j)}$ and all other receiver surface elements $A^{(i)}$, instead of considering all shooters and receivers simultaneously. Having selected shooter $A^{(j)}$ and its center point $\vec{y}_j$, the radiance of this point is sent to all those surface elements that are visible from here. Since the surface elements are small, the visibility is checked by inspecting only the center $\vec{x}_i$ of the receiver surface element $A^{(i)}$. The Monte Carlo estimate of the reflected radiance of finite element $A^{(i)}$ after this transfer is

$$R_m^{(i)}(\omega) = \frac{L_{m-1}^{(j)}(\omega_m^{(i)}) \cdot f_r^{(i)}(\omega_m^{(i)}, \omega) \cdot G_m^{(i)}}{p_j}, \tag{7.6}$$

where direction $\omega_m^{(i)}$ points from the randomly selected shooter $\vec{y}_j$ of iteration m to point $\vec{x}_i$, and G_m is the geometry factor between them.

Due to the possible glossy reflections, the reflected radiance depends on viewing direction ω, thus its representation would require finite element decomposition in the directional domain as well. Note, however, that this dependence is caused by the BRDF, which makes it possible to store the irradiance by a single value. The iteration algorithm stores the *actual irradiance texture* caused by the last iteration step:

$$I_m^{(i)} = \frac{L_{m-1}^{(j)}(\omega_m^{(i)}) \cdot G_m}{p_j}. \tag{7.7}$$

Storing the shooter location of the previous iteration $\vec{y}_j$ in a global (uniform) parameter, the reflected radiance in an arbitrary direction ω can be obtained as

$$R_m^{(i)}(\omega) = I_m^{(i)} \cdot f_r^{(i)}(\omega_m^{(i)}, \omega). \tag{7.8}$$

The final image will be the average of the estimates computed for the eye direction. To obtain this average we compute the reflected radiance for the eye direction and add this result to *accumulation texture* value C:

$$C_m^{(i)} = C_{m-1}^{(i)} + I_m^{(i)} \cdot f_r^{(i)}(\omega_m^{(i)}, \omega_{eye}). \tag{7.9}$$

When the final result is displayed, we render the scene with the texture of values C divided by the number of iterations.

In order to realize this random transport operator, two tasks need to be solved, including the random selection of a texel identifying point $\vec{y}_j$, and the update of the irradiance at those texels which correspond to point $\vec{x}_i$ visible from $\vec{y}_j$ while also computing the contribution of this transfer onto the image.

7.3.1 Random texel selection

The randomization of the iteration introduces some variance in each step, which should be minimized in order to get an accurate result quickly. According to importance sampling, the introduced variance can be reduced with a selection probability that is proportional to the integrand. Unfortunately, this is just approximately possible, and the selection probability is set proportional to the current power of the selected texel. Since the reflected radiance can be due to only a transfer from the previous shooter, the power can be expressed from the irradiance:

$$\Phi_m^{(j)} = \int_\Omega \int_{A^{(j)}} L_m(\vec{x}, \omega) \cdot \cos\theta \; dx d\omega = (E^{(j)} \cdot \pi + I_m^{(j)} \cdot a^{(j)}(\omega_m^{(j)})) \cdot A^{(j)},$$

where $a^{(j)}(\omega^{in}) = \int_\Omega f_r^{(j)}(\omega^{in}, \omega) \cdot \cos\theta \; d\omega$ is the *albedo* of surface element $A^{(j)}$.

If the light is transferred on several wavelengths simultaneously, the luminance of the radiated power should be used. Thus the selection probability of elementary surface j is:

$$p_j = \frac{\mathcal{L}(\Phi^{(j)})}{\Phi}, \qquad \Phi = \sum_k \mathcal{L}(\Phi_k),$$

where $\mathcal{L}$ is the luminance of a spectrum represented by red, green and blue components. Substituting this selection probability into equation 7.7, we obtain:

$$I_m^{(i)} = \Phi \cdot \frac{E^{(j)} + I_{m-1}^{(j)} \cdot f^{(j)}(\omega_{m-1}^{(j)}, \omega_m^{(i)})}{\mathcal{L}(E^{(j)} \cdot \pi + I_{m-1}^{(j)} \cdot a^{(j)}(\omega_{m-1}^{(j)}))} \cdot G_m. \tag{7.10}$$

Note that if uniform parameterization is used, then $A^{(i)}$ is similar to all texels.

The random selection according to p_j can be supported by a *mipmapping* scheme. Mipmapping has originally been proposed for texture filtering. Later it was also used to find the maximum value in an image [25]. In our approach, however, we use mipmapping to sample randomly, proportional to stored value $\mathcal{L}(\Phi^{(j)})$. A mipmap can be imagined as a quadtree, which allows the selection of a texel in $\log_2 \mathcal{R}$ steps, where $\mathcal{R}$ is the resolution of the texture. Each texel is the sum of the luminance of four corresponding texels on a lower level. The top level of this hierarchy has only one texel, which contains the average of the luminance of all elementary texels. Unfortunately, current graphics cards do not provide automatic mipmapping for floating point textures. Thus the mipmap generation and sampling based on mipmaps are implemented by custom vertex and pixel shaders. Both generation and sampling require the rendering of a textured rectangle (also called a full screen quad) by $\log_2 R$ times.

The pixel shader of a pass of the mipmap generation, which builds a single mipmap level, is as follows:

```
float2 uv = texcoord;
float2 uv1 = float2(uv.x, uv-recres);
float2 uv2 = (uv.x-recres, uv.y-recres);
float2 uv3 = float2(uv.x-recres, uv.y);
return tex2D(mipmap, uv1) + tex2D(mipmap, uv2) + tex2D(mipmap, uv3) + tex2D(mipmap, uv);
```

In order to speed up rendering, we do not generate a new mipmap after each iteration step. Note that this does not introduce any bias, and only makes the importance sampling less accurate.

The generated mipmap is used to sample a texel with a probability that is proportional to its luminance. First the luminance of the top level texel is retrieved from a texture and is multiplied by a random number uniformly distributed in the unit interval. Then the next mipmap level is retrieved, and the four texels corresponding to the upper level texel is obtained. The luminance of the four pixels are summed, the running sum (denoted by `cmax` in the program below) is compared to value `r` obtained on the higher level. When the running sum gets larger than the selection value from the higher level, the summing is stopped and the actual value is selected. A new selection value is obtained as the difference of the previous value and the luminance of all texels before

the found texel (`r-cmin`). Then the same procedure is repeated in the next pass on the lower mipmap levels. This procedure terminates at a leaf texel with a probability that is proportional to its luminance.

The pixel shader of a pass of the mipmap based sampling, which selects according to random value `r` passed from the upper level and originally set randomly with uniform distribution in $[0, \Phi]$:

```
float cmin = 0, cmax = tex2D(texture, uv);
if(cmax >= r) {
   c = float3(r-cmin, uv.x, uv.y);
} else {
   cmin = cmax;
   float2 uv1 = float2(uv.x-rr, uv.y);
   cmax += tex2D(texture, uv1);
   if(cmax >= r) {
      c = float3(r-cmin, uv1.x, uv1.y);
   } else {
      cmin = cmax;
      uv1 = float2(uv.x, uv.y-rr);
      cmax += tex2D(texture, uv1);
      if(cmax >= r) {
         c = float3(r-cmin, uv1.x, uv1.y);
      } else {
         cmin = cmax;
         uv1 = float2(uv.x-rr, uv.y-rr);
         c = float3(r-cmin, uv1.x, uv1.y);
      }
   }
}
return c;
```

The `rr` parameter is the distance between two neighboring pixels in texture address space.

7.3.2 Update of the radiance texture

The points visible from $\vec{y}$ can be found by placing a hemicube around $\vec{y}$, and then using the z-buffer algorithm to identify the visible patches. Since it turns out just at the end, i.e. having processed all patches by the z-buffer algorithm, which points are really visible, the application of the random transfer operator requires two passes.

First pass: construction of the depth map

In the first pass the center and the base of the hemicube are set to $\vec{y}$ and to the surface at $\vec{y}$, respectively, then the scene is rendered computing the z coordinate of the visible points. These values are written into a texture, called the *depth map*. We also compute a tolerance value as the absolute value of the dot product between the surface normal and the view direction, so we can identify the faces that are close to being viewed at grazing angles from the shooter's point of view.

Note that this approach differs from earlier methods [91, 25] creating a map of patch indices and checking whether a patch associated with the pixel is the same as the id stored in the map. Working with depth values instead of patch indices allows tolerance to be incorporated, which can eliminate dot artifacts of previous methods.

We also tried the application of hemispherical mapping [25], but we gave it up because the distortion of hemispherical projection introduced many artifacts. The problem is that hemispherical projection is non-linear, and maps triangles to regions bounded by ellipses. However, the current graphics hardware always assumes that the points leaving the vertex shader are triangle vertices, and generates those pixels that are inside these triangles. This is acceptable only if the triangles are very small, and thus the distortion caused by the hemispherical mapping is negligible. Recall that in our "patchless" rendering approach, no tessellation of the geometry is required, thus hemispherical projection is not feasible.

Second pass: irradiance update

In the second pass we render into the rectangle of the irradiance texture. It means that the pixel shader visits each texel, and updates the stored actual irradiance (`I`) and the accumulating radiance (`C`) according to equations 7.7 and 7.9.

The *vertex shader* is set to map a point onto the corresponding texel having coordinates `texx`:

```
OUT.hpos.x = 2 * IN.texx.x - 1;
OUT.hpos.y = 1 - 2 * IN.texx.y;
OUT.hpos.z   = 0;
OUT.hpos.w   = 1;
OUT.texx = IN.texx;
```

Note that the first instruction not only makes the vertex coordinate equal to the texture coordinate, but also applies a transformation. This transformation is necessary because the vertex screen coordinates must be in [-1, 1], while the texture coordinates are expected in [0,1]. Direct3D has a texture space which defines the upper left corner as $(0, 0)$ and the lower right corner as $(1, 1)$, so we need to flip y coordinates. Also notice the `offset`, which moves to the center of the texels, and has to be equal $1/(2\mathcal{R})$, where $\mathcal{R}$ is the resolution.

The vertex shader also transforms the input vertex to camera space (`x`), as well as its normal vector `xnorm` to compute radiance transfer, determines homogeneous coordinates `vch` for the location of the point in the depth map.

```
x        = mul(position, modelview).xyz;
xnorm    = mul(xnorm, modelviewIT).xyz;
viscoord = mul(position, modelviewproj);
```

In equation 7.7, the geometric factor depends on the receiver point, thus its accurate evaluation could be implemented by the *pixel shader*:

```
float3 ytox = normalize(x); // dir y to x
float  xydist2 = dot(x, x); // |x - y|^2
float  cthetax = dot(xnorm, -ytox);
if (cthetax < 0) costhetax = 0;
float3 ynorm(0, 0, 1);
float  cthetay = ytox.z;
if (cthetay < 0) costhetay = 0;
float G = cthetax * cthetay / xydist2;
```

Note that we took advantage of the fact that $\vec{y}$ is the eye position of the camera, which is transformed to the origo by the `modelview` transform, and the normal vector at this point is transformed to axis z.

When a texel of the current radiance map and of the accumulating radiance map is shaded, it is checked whether or not the center of the surface corresponding this texel is visible from the shooter by comparing the depth values stored in the visibility map. We have to apply a tolerance based on the cosine of the viewing angle in order to avoid point artifacts. This tolerance is stored in the green channel of the visibility map. The pixel shader code responsible for converting homogeneous coordinates (`vch`) to Cartesian coordinates (`vcc`) and computing the visibility indicator is:

```
float3 vcc = vch.xyz / vch.w; // Cartesian
vcc.x = (vcc.x + 1) / 2;      // Texture space
vcc.y = (1 - vcc.y) / 2;
float2 depth = tex2D(depthmap, vcc).rg;
float vis = (abs(depth.r - vcc.z) < (eps1 - eps2 * depth.g));
```

To obtain the radiance transfer from shooter $\vec{y}$ to the processed point $\vec{x}$, first the radiance of shooter $\vec{y}$ is calculated from its irradiance stored in irradiance map `irradmap` according to its BRDF, and its emission stored in `emissmap`. Shooter's texture coordinates `texy` and previous shooter `yprev` are passed as uniform parameters:

```
float3 Iy = tex2D(irradmap, texy);
float3 Ey = tex2D(emissmap, texy);
float3 yin = normalize(yprev);  // yprev - y
float3 Ly = Ey + Iy * BRDF(texy, yin, ynorm, ytox);
```

The `BRDF` function reads the BRDF parameters from texture map `brdfmap` according to texture coordinates `texy` and computes a simple streched-Phong BRDF [89].

```
float3(float4 f, float3 din, float3 norm, float3 dout, float shine) {
   float3 fd = float3(f.r, f.g, f.b)/pi;
   float3 fs = float3(1, 1, 1) * f.a * (shine+1)/2/pi;
   float3 cin = dot(din, norm);
   float3 cout = dot(dout, norm);
   if (cout < 0 || cin < 0) return float(0, 0, 0);
   float cmax = (cin > cthetay) ? cin : cthetay;
   float cfi = dot(reflect(din, norm), dout);
   return (fd + fs * pow(cfi, shine) / cmax);
}
```

The new irradiance at $\vec{x}$ is obtained from the radiance at $\vec{y}$ multiplying it with visibility `vis` and geometric factor `G` computed before, and divided by probability `p` passed as a uniform parameter. The emission and surface area of this texel are read from texture map `emissmap`. The luminance of the reflected power (`lumPowx`) at $\vec{x}$ is also computed to allow importance sampling in the subsequent iteration step, and stored in the alpha channel of the irradiance. Additionally, the contribution to the eye is also determined and the increase value is output in `C`.

```
float3 Ix = Ly * G * vis / p;
float4 Ex = tex2D(emissmap, texx);
float  Ax = Ex.a;           // surface area
float3 alb = Albedo(texx);  // albedo of x
float3 em  = float(1, 1, 1) * pi;
float  lumPowx = (dot(E, em) + dot(I, alb))/3;
OUT.I = float4(Ix, lumPowx);
float3 xtoe = normalize(eye - x);
float3 Lx = Ex + Ix * BRDF(texx, ytox, xnorm, xtoe);
OUT.C = C + Lx;
```

This pixel shader outputs two values, including the irradiance `I` and accumulating radiance toward the eye `C`, thus it requires the multiple render target option.

7.4 Variance reduction

When we applied Monte Carlo estimates for the sum (or integral) of equation 7.3, the sampling probability could not precisely mimic the terms. Due to the requirement of GPU implementation, we could take into account only the power of the shooter, but neither the geometric factor between the shooter and the receiver, nor the BRDF of the receiver. Unfortunately, the geometric factor may have a large variation, especially around the corners of the scene where the source and the receiver are very close, which results in larger error at these regions. This means that close to the corners the light is transported seldomly but in great amounts, which causes bright spikes (figure 7.1). We propose two techniques to reduce this error without modifying the underlying sampling scheme.

7.4.1 Partial analytic integration

Recall that we applied a finite element decomposition over the positional variation of the irradiance function. The decomposition was so fine that we could test the visibility for a single point of each finite element, thus we could eliminate all summations that would pose problems to the GPU. However, using the same level of surface tessellation for both visibility calculations and irradiance representation is not very effective since visibility information changes much more quickly than the

relatively smoother irradiance function. This approach requires large textures, and small finite elements make it possible that two finite element centers get too close to each other, which is responsible for corner spikes. This problem can be solved if we use denser samples for visibility computations than for irradiance representation. It corresponds to merging different terms of equation 7.3 that correspond to elementary surfaces close to each other and therefore have similar radiance (a term is in fact a texel that represents a small surface area). Merging neighboring texels is a texture filtering, which replaces the texture by a higher mipmap level.

Note that unlike in classical radiosity algorithms and in [25], we do not rely on the patch structure when the correspondence between the two levels of details are defined. The lower resolution texture can be imagined as a filtered version on a higher level in a mipmap structure. When this filtering is implemented, we have to be careful not to combine two texels that would correspond to two different surfaces. This problem is solved by looking up another texture generated during the preprocessing phase. This texture stores a patch id for each texels, and can be used to detect whether or not two texel values can be combined.

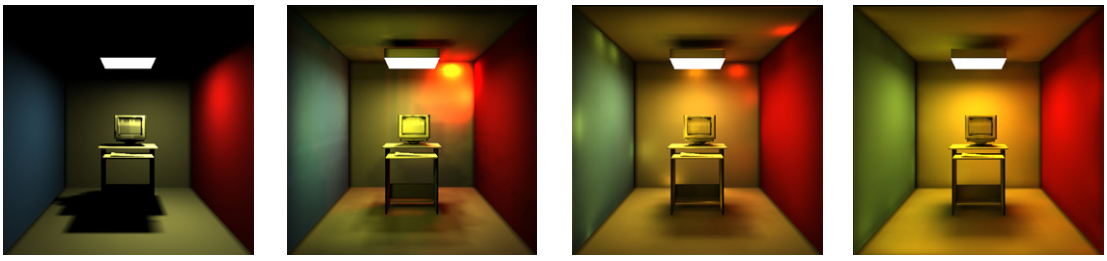
When combining several texels, the resulting radiance is the average of the radiance of individual texels, and the resulting geometry factor is the sum of the individual geometry factors. The sum (or the integral) of the geometry factors can be obtained using the point to disc approximation. Let us suppose that the merged texels correspond to surface area A with center $\vec{y}_j$. If the radiance of the texels are similar and their average is $\tilde{L}$, then

$$\int_A L(\vec{y}, \omega_{\vec{y}\to\vec{x}_i}) \cdot f_r^{(i)}(\omega_{\vec{y}\to\vec{x}_i}, \omega) \cdot G(\vec{x}_i, \vec{y})\ dy \approx$$

$$\tilde{L}(\omega_{\vec{y}_j\to\vec{x}_i}) \cdot f_r^{(i)}(\omega_{\vec{y}_j\to\vec{x}_i}, \omega) \cdot v(\vec{y}_j, \vec{x}_i) \cdot \frac{\cos\theta_{\vec{y}_j} \cdot \cos\theta_{\vec{x}_i}}{|\vec{y}_j - \vec{x}_i|^2 + A/\pi} \cdot A. \tag{7.11}$$

Note that this merging can significantly reduce the variation of the terms since replacing elementary surfaces $A^{(j)}$ by larger surfaces A, the center $\vec{y}_j$ of the larger surface will be farther from receivers $\vec{x}_i$. On the other hand, the disc to point form factor approximation adds an A/π term to the denumerator, which cannot be as small as in the original approach.

7.5 Implementation results and further improvements



1 iteration, 15 msec | 10 iterations, 130 msec | 100 iterations, 1.2 sec | 1000 iterations, 10 sec

Figure 7.1: Image rendered with partial analytic integration and multiple importance sampling using 100 iteration. All objects are mapped to a single texture map of resolution 128×128. The rendering times are measured on an Nvidia 6800GT graphics card.

The proposed method has been implemented on an NV6800GT graphics card in DirectX/HLSL environment. The data associated with surface elements are stored in textures, including

- *BRDF texture* (`brdfmap`) $[a_r^d, a_g^d, a_b^d, a^s]$ storing the red, green and the blue diffuse albedos and the specular reflection parameters in the alpha channel. The integer part of the alpha

channel defines the shininess while the fractional part the specular albedo. We assume that specular reflections are independent of the wavelength, i.e. we can model dielectric materials.

- *Emission texture* (`emissmap`) $[E_r, E_g, E_b, A]$, which stores the emission intensities and the surface area corresponding to this texture element. Note that in this way we can model diffuse light sources.
- *Actual irradiance* (`irradmap`) $[I_r, I_g, I_b, \mathcal{L}\Phi]$ representing the irradiance values as well as the luminance of the reflected power.
- *Accumulated radiance* (`radmap`) $[C_r, C_g, C_b]$.

The implementation has been tested with the Cornell box scene and we concluded that a single iteration requires less than 20 msec for a few hundred vertices and for 128×128 resolution radiance maps, while keeping the depth maps at 256×256 (the algorithm is pixel shader limited). Using 64×64 resolution radiance maps introduced a minor amount of shadow bleeding, but increased iteration speed by approximately 40%. Since we can expect converged images after $40 - 80$ iterations for normal scenes, this corresponds to $0.5 - 1$ frames per second, without exploiting frame-to-frame coherence. The algorithm (depending on the resolution) uses several render-to-surface objects and does one iteration in $22 - 30$ passes. With faster, hardware optimized shadow calculation we should be able reduce the number of passes by 4 as well. In order to eliminate flickering, we should use the same random number generator in all frames. On the other hand, as in all iterative approaches, frame to frame coherence can be easily exploited. In case of moving objects, on the other hand, we can take the previous solution as a good guess to start the iteration. This trick not only improves accuracy, but also makes the error of subsequent steps highly correlated, which also helps eliminating flickering.

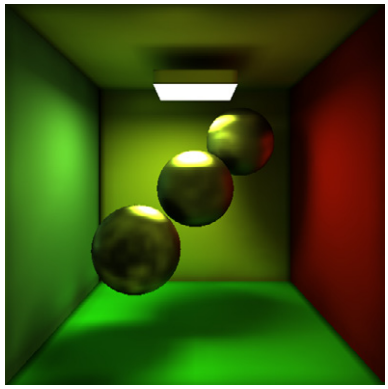


Figure 7.2: Specular objects in a diffuse room rendered with the proposed method in 9 sec on an Nvidia 6800GT graphics card.

7.6 Conclusions

This chapter presented a stochastic glossy global illumination algorithm running entirely on the GPU. In order to port a global illumination algorithm — which should follow the interdependence of the patch radiances — onto the graphics hardware — which usually assumes patch independence — we used randomization. Randomization has turned out to be an efficient tool to modify algorithms to meet the capabilities of the underlying hardware. The final algorithm is fast, it can render moderately complex scenes interactively, and thus is an appropriate candidate to include global illumination effects in games. On the other hand, the algorithm is relatively simple and easy to implement.

Chapter 8

Obscurances Tool

The obscurance method is a powerful technique that simulates the effect of diffuse interreflections, i.e. radiosity, on an object at a much lower performance cost. Its main advantage lies in the fact that this technique considers only neighboring interactions instead of attempting to solve all the global ones. Another advantage of this technique is that it is decoupled from direct illumination computation. This tool applies obscurances to be used in a game environment, allowing realistic and fast illumination of the scene.

8.1 Introduction

Radiosity techniques [40] are commonly used to simulate diffuse global illumination – although very powerful and increasingly faster [10], they do not yet fulfil the requirements for fast and efficient real-time scene editing or rendering. With radiosity, the interaction of each surface in the scene with each other surface has to be (at least potentially) considered; the obscurance method [139, 54] is designed to offer a fast alternative to radiosity.

This technique can deal with any number of moving light sources with no added cost since the indirect illumination and direct illumination are effectively decoupled. The obscurance technique also allows the addition of color bleeding [85] to your lighting.

Obscurances have already been used in 3D computer games and animations in a simplified form commonly called ambient occlusions [20, 95, 16]. This tool presents a new algorithm to compute obscurances using depth peeling [36]. In addition we will discuss the real-time update of obscurances for moving objects within a scene.

8.2 Obscurances

In [139, 54] the obscurances illumination model was defined. Obscurances take account of secondary diffuse illumination, being totally decoupled from direct illumination. Indirect illumination for point P is defined as:

$$I(P) = \frac{1}{\pi} \cdot R(P) \cdot I_A \cdot \int_{\vec{\omega} \in \Omega} \rho(d(P, \vec{\omega})) \cdot \cos\theta \; d\omega \tag{8.1}$$

where

- $d(P, \vec{\omega})$ is the distance between P and the next surface at direction $\vec{\omega}$,
- $\rho(d)$ is a function that determines the magnitude of ambient light incoming from neighborhood d, and taking values between 0 and 1,

- θ is angle between direction $\vec{\omega}$ and the normal at P,
- I_A is the ambient light intensity,
- $R(P)$ is the reflectivity at P,
- $1/\pi$ is the normalization factor such that if $\rho() = 1$ over the whole hemisphere Ω then $I(P)$ is $R \times I_A$.

Direct illumination can be added to (8.1) to obtain the final illumination at the point. Function $\rho()$ increases with d. Its shape is given in figure 8.1.

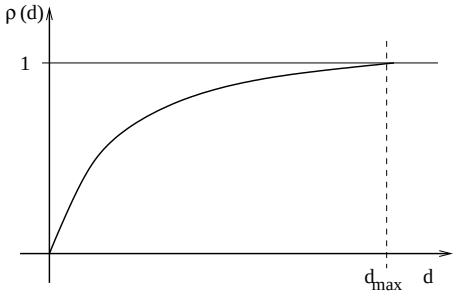


Figure 8.1: Shape of $\rho(d)$ function.

A maximum distance for interaction, d_{max} is defined, so that when $d \geq d_{max}$ then $\rho(d) = 1$. This means that we only take into account a d_{max}-neighborhood of P. In other words, we are not taking into account occlusions farther than d_{max}. The function used for this tool is $\sqrt{d/d_{max}}$ if $d < d_{max}$ and 1 otherwise.

Obscurance of the point P is then defined as:

$$W(P) = \frac{1}{\pi} \cdot \int_{\vec{\omega} \in \Omega} \rho(d(P, \vec{\omega})) \cdot \cos\theta \; d\omega \tag{8.2}$$

Since $0 \leq \rho(d) \leq 1$, we can assume that $0 \leq W(P) \leq 1$. The obscurance for a patch (in radiosity terms, a polygon from a subdivided mesh) is the average of the obscurances for all points within the patch. An obscurance value of 1 means that the patch is totally open (or not occluded by neighboring polygons), while a value of 0 means that it is totally closed (or occluded by neighboring polygons).

For a closed environment, the ambient light in (8.1) can be computed as the average light intensity in the scene using the following formula:

$$I_A = \frac{R_{ave}}{1 - R_{ave}} \cdot \frac{\sum_{i=1}^{n} A_i E_i}{A_T} \tag{8.3}$$

where

$$R_{ave} = \frac{\sum_{i=1}^{n} A_i R_i}{A_T} \tag{8.4}$$

where A_i, E_i and R_i are the area, emissivity and reflectivity of patch i, respectively, A_T is the sum of the areas, and n is the number of patches in the scene. The ambient term considered here corresponds to the indirect illumination only, as direct illumination is computed separately. Since the obscurance computation only takes into account the neighborhood of a patch (i.e. near than d_{max}), the computation can be done very efficiently. For further details see [139] and [54].

8.3 Color bleeding

The obscurance approach as presented in previous section lacks one of the features which comes from radiosity lighting, color bleeding. Since the light reflected from a patch acquires some of its color, the surrounding patches receive colored indirect lighting. Here, we present a straightforward technique to account for this color bleeding, with no added computational cost.

The obscurances formula (8.2) is modified slightly to include the reflectivity term of the patches:

$$W(P) = \frac{1}{\pi} \cdot \int_{\vec{\omega} \in \Omega} R(Q) \cdot \rho(d(P, \vec{\omega})) \cdot \cos\theta \; d\omega \tag{8.5}$$

where $R(Q)$ is the reflectivity of point Q as seen from P in direction $\vec{\omega}$. When no surface is seen at a distance less than d_{max} in direction $\vec{\omega}$, the obscurance takes the value of R_{ave}.

For coherency, the ambient light equation (8.3) also has to be modified, yielding the following value:

$$I_A = \frac{1}{1 - R_{ave}} \cdot \frac{\sum_{i=1}^{n} A_i E_i}{A_T}. \tag{8.6}$$

The improved obscurance model can be computed in several ways. The usual option is to compute the obscurance equation (8.5) using Monte Carlo (or quasi Monte Carlo) ray casting technique, which casts several rays from a patch distributed along $\cos\theta$. The obscurance for the patch i will then be the average of the values gathered by the rays cast from this patch:

$$W(i) = \frac{1}{N_i} \cdot \sum_{j=1}^{N_i} \rho_j R_{int} \tag{8.7}$$

where N_i is the number of rays cast from patch i, R_{int} is the reflectance at the intersected patch (if no patch is intersected, then we take R_{ave}) and j is the value of the function $\rho()$ for ray j. Several Monte Carlo and quasi-Monte Carlo sampling techniques have been tested for obscurance computation in [84], and the Halton quasi-Monte Carlo sequence gave the best performance.

But this quasi-Monte Carlo approach is not our only option. Sbert [109] demonstrated that casting cosine distributed rays from all patches in the scene is equivalent to casting global lines joining random points of the bounding sphere of the scene. Furthermore, it is also equivalent to casting bundles or parallel rays of random directions (see Figure 8.2). Bundles of parallel rays can be efficiently cast on the graphics hardware using the depth peeling algorithm as shown in the next section.

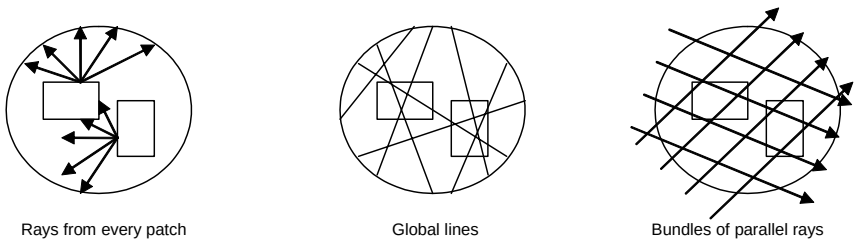


Figure 8.2: Different ray-tracing techniques for computing obscurances.

In figure 8.3a we show the Cornell box scene computed with the obscurances but without color bleeding, while in Fig. 8.3b we have used our improved algorithm. Color bleeding is clearly visible, adding a lot of realism to the image, with no added cost. We can compare these images with the one obtained with a more complete radiosity algorithm (Fig. 8.3c) and see that the improved obscurance method represents a step forward towards simulating a radiosity image.

(a) No color bleeding. (b) With color bleeding. (c) A radiosity solution.

Figure 8.3: Comparison of the obscurances solution, without and with color bleeding, with a radiosity solution.

8.4 GPU obscurances using depth peeling

The basic idea behind the depth-peeling technique is to extract visibility layers from the scene in order to do some computation between them. In [36], the technique is used to achieve order-independent transparency. Global illumination [128, 44] has been done also with depth-peeling.

We can see the pixel image resulting from depth-peeling as being equivalent to tracing a bundle of parallel rays through the scene where each pixel corresponds to a ray in the bundle. Each of these rays may intersect several surfaces in the scene, and through depth-peeling we can discover all of the intersections in the form of image layers and not only the closest one obtained by the z-buffer algorithm.

Once we have chosen a random direction for the bundle, the computation of obscurances with depth peeling is divided into two phases. In the first phase, layers are obtained using depth-peeling. In the second phase, the obscurances between each pair of layers are computed and the result is added and averaged in the corresponding obscurance map position.

8.4.1 Depth Peeling

We assume that, in a pre-processing step, the scene is completely mapped to a single texture atlas. A texel of the texture atlas corresponds to a small surface area, that corresponds to obscurance patches. When a patch is referenced, we can simply use the texture address of the corresponding texel. The obscurance computation picks a random direction and carries out depth-peeling process in this direction. When we let the GPU to do it for us, we use an orthogonal projection, and from the sampled direction we render the scene setting the model-view transform to rotate the sample direction to the z axis.

We use the *pixel* (RGBA) of an image layer to store the patch identification, a flag indicates whether the patch is front-facing or back-facing to the camera and the camera to patch distance. Our pixel buffer is initialized with (-1.0,-1.0,1.0,1.0), giving us reasonable default values.

The facing direction of a pixel can be determined by using the cosine of the angle between the camera's -z vector and the normal vector of the patch. If the result is greater than 0, it is front-facing, otherwise it is back-facing. The cosine can be determined by using the z component of the dot product between the inverse transpose model-view matrix and the normalized normal vector of the patch.

As we store the pixels in a four-component float array (or the RGBA color), we use the first two components to store the patch ID ($RG \leftarrow (u, v)$), the third to store the cosine ($B \leftarrow \cos\alpha$), and the fourth component to store the distance between the camera and the patch ($A \leftarrow z$).

The vertex shader receives the vertex coordinates, the texture coordinates (in (u,v), identifying the texel), and the normal, and it generates the cosine and the transformed vertex position:

```
void main( float4 position      :  POSITION,
           float2 texCoord      :  TEXCOORD0, //Patch ID
           float4 Norm          :  NORMAL, //Patch Normal

           out float4 oposition :  POSITION,
           out float2 otexCoord :  TEXCOORD0,
           out float cosine     :  TEXCOORD1,

           uniform float4x4 modelView,
           uniform float4x4 modelViewInvTrans)
{
           oposition = mul(modelView,position);
           otexCoord = texCoord;

           cosine = mul(modelViewInvTrans,Norm).z; //sample direction is rotated to (0,0,1)
}
```

The fragment shader receives the interpolated texture coordinates of the fragment, the position (where z is the depth), the cosine and the interpolated texture coordinates of the patch. For the first layer, the depth does not need to be compared with the previous one. However, for all subsequent layers, we sample the previous layer using the texture coordinates and discard it if the depth of the previous layer (the fourth component of the sample) is closer to the camera than the actual fragment thus getting the peeling effect (Figure 8.4).

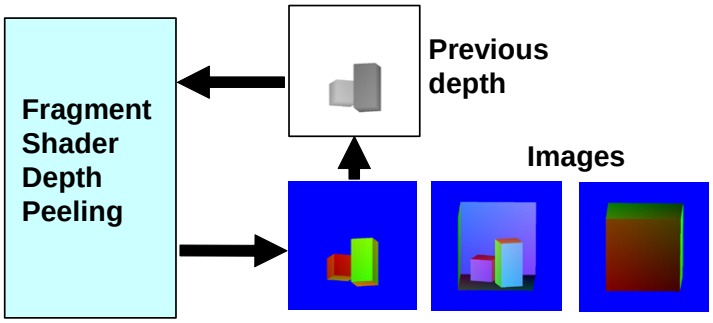


Figure 8.4: Schema of the depth peeling with GPU.

This rendering step is repeated until all pixels are discarded. The images of all the rendered layers define all ray-surface intersections (Figure 8.5).

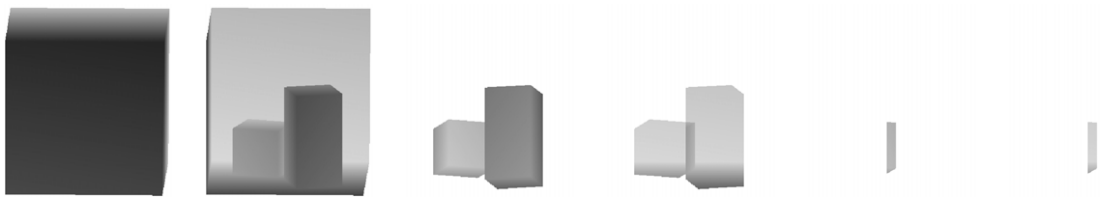


Figure 8.5: Six different image layers showing depth information for each pixel for the Cornell Box scene.

The fragment shader code is:

```
void main(float4            position :  WPOS,
          float2            texCoord :  TEXCOORD0, //Patch ID
          float             cosine   :  TEXCOORD1, //Patch Orientation

          out float4        color    :  COLOR, //Patch ID + Orientation + Depth

          uniform sampler2D ztex, //previous depth image
          uniform float     res, //resolution of projection window
          uniform float     first) //is first layer?
{
   if( first == 0.0 ) // not first -> peel
   {
      float depth = tex2D(ztex,position.xy/res).a; //last depth
      if (position.z < (depth + 0.000001)) discard; //ignore previous layers
   }
   color.rg = texCoord;
   color.b = cosine;
   color.a = position.z; //new depth
}
```

8.4.2 Obscurances

For each pair of consecutive layers, the obscurance formula is computed.

We configure the camera to obtain a one-to-one mapping between pixels and texels. The size of the viewport is set to the same resolution as the obscurance map, starting from (0,0), with an orthogonal projection from -1 to +1 in both dimensions.

Now each pair of consecutive images is taken from the texture memory and sent to the graphic pipeline as a stream of points of size 1.0 (render to vertex array). This way we can update a single position in the target buffer for each element of the image. This will generate a pair of point streams A and B that are merged together and sent to the Vertex Shader. Stream A is sent as vertex positions and stream B as texture coordinates. As we generate the streams in both images in the same way, points at the same position in streams A and B are at the same position in consecutive images, thus may see each other in the sampling direction and transfer energy consequently (Figure 8.6).

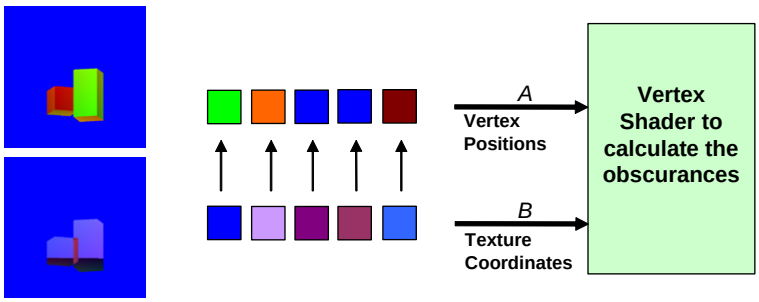


Figure 8.6: Two consecutive layers (left) generate two streams of points carrying patch ID's (middle) that are merged together and processed by the vertex shader (right).

The obscurance computation needs to be done bidirectionally but we cannot generate two values in different positions of the target buffer in a single pass and thus we have to do a two-pass transfer. In the first pass, we update the patches in the projection that generated stream A using the information in pixels of stream B (Figure 8.6). In the second pass the streams are exchanged, thus the same set of shaders are used in both directions.

If a patch in stream A cannot see the corresponding patch in stream B, the vertex carrying this patch is eliminated by moving it out of the view frustum. If patches see each other and the difference between their distances to the camera is less than d_{max} , then the transfer is done. If the distance is greater or transfers with the background, then the patch gets the ambient reflectivity. When the transfer process is done, we generate vertex coordinates to update the position in the obscurance map that corresponds to the patch identified by the two first components of the current pixel element in stream A.

The vertex shader needs to generate vertex coordinates in homogeneous clip space. The desired position is encoded as the patch ID but is in a normalized form (as 2D texture coordinates are in the range [0..1]). The following formula computes which homogeneous clip coordinates we need to generate to obtain the desired normalized window coordinates given a camera and a viewport set as explained earlier:

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} 2x_w - 1 \\ 2y_w - 1 \\ 2z_w - 1 \\ 1 \end{pmatrix} \tag{8.8}$$

Note that the clipping process only keeps the fragment if $-w_c \leq z_c \leq w_c$. As we define a w_c as 1, the clipping process will only keep the fragment when $-1 \leq z_c \leq 1$. If $z_c = 2.0$ then $z_w = 1.5$ and the fragment is out of the view frustum and is discarded. If we set $z_c = 0.0$ then $z_w = 0.5$, thus the vertex is kept by the clipping process. So we can use the z_c value as a way to accept or discard vertices. The vertex shader for the obscurance transfer process is:

```
void main( float4 A   :  POSITION, //x,y = ID; z = Orientation; w = Depth;
           float4 B   :  TEXCOORD0, //x,y = ID; z = Orientation; w = Depth;

           out float4 oposition :  POSITION,
           out float2 pA        :  TEXCOORD0, //Texture coordinates of A.
           out float2 pB        :  TEXCOORD1, //Texture coordinates of B.
           out float  distance  :  TEXCOORD2, //Distance.

           uniform float        direction) //Switch of direction
{
   //If patches see each other, i.e.  both exist and one is front facing and
   //the other is back facing.
   //Direction tells if we are transferring in the camera -z direction or +z.
   //A.r contains the patch ID. If it contains -1.0 means that it does not
   //belong to the scene.
   //A.b contains the cosine.
   if(((direction == 0) && (A.r != -1.0) && (A.b < 0.0)
         && ((B.b > 0.0) || (B.r == -1.0)))
      || ((direction == 1) && (A.r != -1.0) && (A.b > 0.0)
         && ((B.b < 0.0) || (B.r == -1.0))))
   {
      pA = float2(A.r, A.g); //Set the texture coordinates for A.
      pB = float2(B.r, B.g); //Set the texture coordinates for B.
      //Create vertex to update desired position.  z = 0.0 => kept by clipping
      oposition = float4((pA * 2.0) - float2(1.0, 1.0), 0.0, 1.0);
      //Calculate distance.  If patch in stream B not in scene => distance = 1.0
      distance = (texCoord.b != 1.0)?  abs(B.a - A.a) :  1.0;
   }
   else //If there's no transfer move out from the view frustum.
   {
      // z = 2.0 to get the id ignored by clipping.
      oposition = float4( 0.0, 0.0, 2.0, 1.0 );
      p1 = p2 = float2(1.0, 1.0);
      distance = 0.5;
   }
}
```

The fragment shader just applies the obscurance formula $\rho = \sqrt{d/d_{max}}$ if $d < d_{max}$ and 1 otherwise.

```
void main( float2 pA : TEXCOORD0, //Texture coordinates of A.
           float2 pB : TEXCOORD1, //Texture coordinates of B.
           float  distance :  TEXCOORD2,

           out float4 ocolor :  COLOR,

           uniform sampler2D reflectivity,
           uniform float     dmax,
           uniform float3    ambient)
{
   if(d>=dmax) ocolor.rgb = ambient; //If distance > dmax, add ambient
   //else we apply the obscurances formula
   else ocolor.rgb = tex2D(reflectivity,pB).rgb * sqrt(distance/dmax);
   ocolor.a = 1.0;
}
```

Figures 8.7, 8.8, and 8.9 show the results of applying our algorithm to models of the *De Espona* library. We show respectively the obscurances map, obscurances with direct illumination, and direct illumination with constant ambient term. Observe the quality of the illumination obtained with obscurances.

8.5 Real-time update for moving objects

Although the computation of the initial obscurance value is not done in real-time, we can update them in real-time efficiently for a moderate number of polygons.

The depth-peeling algorithm, as shown in section 8.4, is not the ideal algorithm to recompute the obscurances for a small part of the scene or moving objects, since by its nature it processes the whole scene at a time. More practical candidates are the computation by ray-tracing or hemi-cube projection.

For this article, we will consider the ray-tracing approach. The algorithm starts by creating a list of influenced patches. For each patch we compute the initial obscurance. If a ray from a patch hits a moving object we store the patch in the list of influenced patches. The patches of the moving objects are also stored in this list. When an object moves to a new position, the obscurances are recomputed for each patch in the list of influenced patches, creating a new list of influenced patches.

In Figure 8.10 we show the result of recomputing in real-time the obscurances for a moving object, using the ray-tracing technique.

8.6 Conclusions

This tool uses the obscurances method, a simple way to simulate diffuse global illumination, implemented in an efficient way in the GPU using the depth peeling technique. We have also seen that, as we query just a limited space around an object, obscurances can be updated in real-time for moving objects in the scene and the surfaces around, using a ray-casting technique.

As future work we plan to improve the efficiency of the depth peeling technique, by reading back textures to be processed on the CPU using the symmetric PCI Express buffer. We will also study the efficiency of the hemicube projection method for the real-time update for moving objects. The extension of obscurances to non-diffuse environments is also considered.

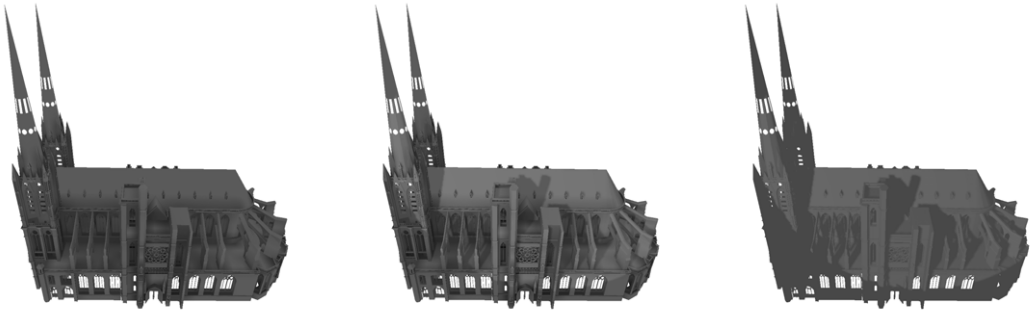


Figure 8.7: Cathedral model, 193180 polygons, obscurances computed in 38 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.

Figure 8.8: Tank model, 225280 polygons, obscurances computed in 38 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.

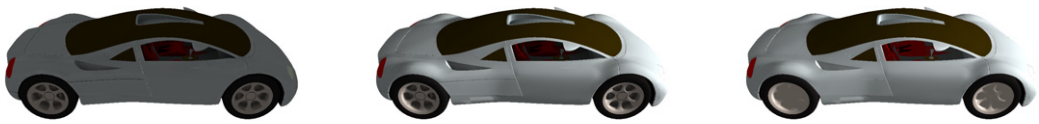


Figure 8.9: Car model, 97473 polygons, obscurances computed in 32 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.

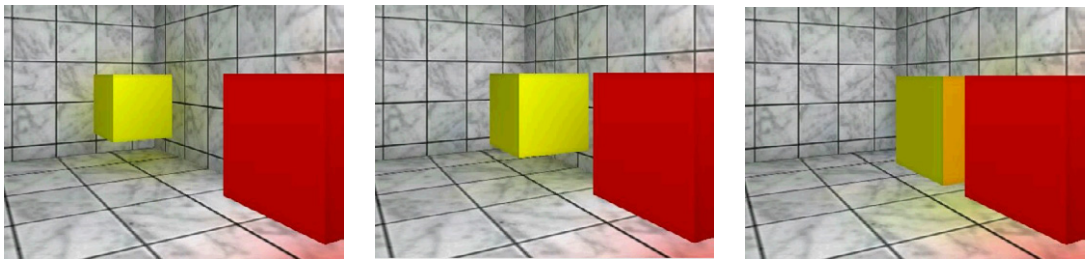


Figure 8.10: Three positions of a moving object showing the dynamic recomputation of obscurances.

Chapter 9

Precomputed Light Paths Tool

This tool presents a real-time global illumination method for static scenes illuminated by arbitrary, dynamic light sources. The algorithm obtains the indirect illumination caused by the multiple scattering of the light from precomputed light paths. The illumination due to the precomputed light paths is stored in texture maps. Texture based representations allow the GPU to render the scene with global illumination effects on very high frame rates even when the camera or the lights move. The proposed method requires moderate preprocessing time and can also work well for small light sources that are close to the surface. The implemented version considers only diffuse reflections. The method scales up very well for complex scenes and there is the possibility for trading storage space for high frequency details in the indirect illumination.

9.1 Introduction

Rendering requires the identification of those light paths that connect light sources to the eye via reflections and refractions, and then the computation of the sum of their contributions. This summation becomes a high-dimensional integral, which is usually impossible to evaluate in real-time. In order to cope with this problem, we assume that the scene is static, when the light paths visiting the points of the scene do not change. This recognition allows us to precompute those integrals that are responsible for the self illumination of the scene, and combine the prepared data with the actual lighting conditions during rendering. This means that during rendering just a low dimensional integral needs to be evaluated, which is possible with very high frame rates.

The chapter is organized as follows. Section 9.2 reviews previous work and places the new approach among the related techniques. Section 9.3 presents the new method both intuitively and providing the mathematical discussion. Section 9.4 describes real-time global illumination applications exploiting the presented idea, and evaluates their performance.

9.2 Previous work

Several approaches have emerged that limit the freedom of object changes in order to make global illumination computations fast. If the scene is static, then radiance transfer coefficients between surface elements do not change. This has been recognized even in the context of finite-element based radiosity algorithms, where form factors could be computed only once. The price is that the required storage is a quadratic function of the number of patches. The idea has been further generalized to include arbitrary number of specular reflections with the introduction of *extended form factors* [115].

Precomputed radiance transfer (PRT) [118, 117] is a method to realistically render rigid bodies assuming that a single object is illuminated by low-frequency hemispherical image based lighting. Its basic idea is that the incident illumination and reflected lighting functions are approximated by a finite element series, and thus lighting is expressed by vectors of coefficients multiplying

the predefined basis functions (usually by spherical harmonics). The correspondence between the incident and reflected light vectors is linear and can be defined by a matrix, which requires a lot of time to obtain, but can be computed during preprocessing. Then, during rendering, whenever the environment lighting changes, the reflected radiance can be quickly updated by a matrix-vector multiplication. The original idea has been improved in many different ways, including arbitrary BRDFs [63], speeding up both preprocessing and rendering phases [71], and principal component analysis to compress data [117]. The fundamental limitation of the PRT methods of assuming infinitely distant lighting has been addressed in [2], where a first-order Taylor approximation has been used to consider midrange illumination. Ng [90] replaced spherical harmonics by wavelets to allow high frequency environment maps.

Mei et al [83] have proposed a solution to render both shadows and self illumination. Their method is based on *spherical radiance transfer maps* including spherical shadow maps for both mutual and self-shadow rendering. These maps have to be pre-computed for every one of the several thousand mesh vertices, and their resolution should be large enough to cover hundreds of sampled directions. It is also assumed that lights are distant and minor artifacts may appear on objects close to each other because of the spherical approximation. The main caveat of the spherical shadow map method is that shadow maps are rendered for the vertices. Then, when the illumination of a vertex is computed, sampled directions are tested against this map to see whether the environment map is occluded in that direction. This setup requires a large number of shadow maps, and could result in inaccurate per-pixel results, depending on the tessellation.

Radiance transfer precomputation has also been applied to translucent objects [72], where the vertex-to-vertex radiosity transfer factors are stored and the response to an illumination is obtained by a finite-element series using these factors as weights. Connecting the transfer factors to the vertices poses similar problems as in spherical transfer maps. The storage requirements is a quadratic function of the number of vertices, which prohibits complex or highly tessellated objects and requires well-shaped patches.

Light path reuse [108] is a Monte Carlo technique to speed up animated light sequences assuming that the objects and the camera are still. Under these circumstances, a light path obtained in a frame may also be used in other frames if the ray between the light point and the first hit point is not invalidated by new occlusions. The combination of all paths in a single frame is governed by multiple importance sampling guaranteeing that the variance of the solution in a frame is close to what could be obtained if we dedicated all paths solely to this particular frame.

Research efforts aiming at including meso-structure properties into material data are also related to this tool since these methods also store self-illumination capabilities compactly and combine them with the actual lighting. These methods are different extensions of texture maps, such as *bi-directional texture functions* [28, 120], or *polynomial texture maps* [78], that are usually obtained by a measuring process. The self-shadowing of meso-structure features can also be precomputed and used in interactive rendering [50].

The method presented in this chapter is related to the mentioned previous approaches in the sense that it also uses a preprocessing step that computes the radiance of certain reference points on the surface assuming a standard illumination. From a different point of view, the preprocessing computes radiance transfer factors between so called entry and reference points. Then, in the rendering phase the precomputed data is combined with the information of the actual lighting conditions. However, the new approach is not limited to environment lighting, does not require finite-elements, such as spherical harmonics or wavelets, but represents the light transport by a set of pre-generated random light paths stored in a compact way. Representing the lighting information in path space rather than in the space of directional finite elements allows very high frequency light sources such as point lights that can be close to the surfaces of the scene, and makes preprocessing times affordable. In this sense, our method is strong where PRT is weak. Similarly to the PRT method, in our case it is also possible to apply compression that trades accuracy for storage space. However, due to the path space representation, this compression scheme is significantly simpler to implement and results in visually pleasing images even in highly compressed cases.

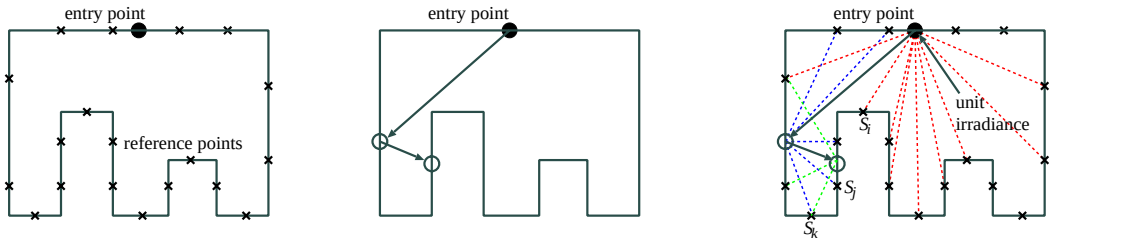
In our method, the light transfer representation is made independent of the vertices of the

mesh, which has the advantage that the proposed method does not require carefully prepared meshes consisting of well-shaped triangles of similar size, but perform well on irregular meshes as well. Based on the recognition made by previous Monte Carlo global illumination research [35], the number of light paths needed to render a scene is roughly independent of the complexity of the scene. It means that the number of entry points serving as the origins of these light paths can be set independently of the vertices. Similarly, the reference points that gather the illumination can also be separated from the vertices allowing a more uniform representation. Thus our initial storage complexity, that is the product of the entry and reference points, can be made independent of the vertices, which makes the method scale well for more complex scenes.

Due to its moderate preprocessing times, the method can also be integrated into a real-time system to cope with slowly changing scenes. It also means that the new method does not necessitate any sophisticated compression and decompression algorithm.

9.3 Method overview

The method consists of a preprocessing step and a fast rendering step.

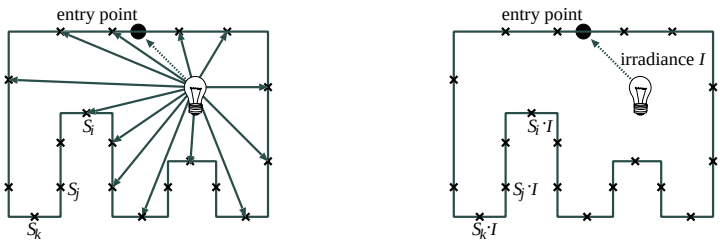


1. Reference points are defined and entry points are sampled

2. Path generation from the entry points

3. S is the illumination of reference points
PRM = $\{(\bullet, \times, S)\}$

Figure 9.1: Overview of the preprocessing phase. Entry points are depicted by $\bullet$, and reference points by $\times$. The PRM is a collection of (entry point $\bullet$, reference point $\times$, illumination S_k) triplets, called items.



1. Direct illumination + entry point visibility

2. Weighting irradiance I with items S

Figure 9.2: Overview of the rendering phase. The illumination of the entry points are computed, from which the illumination of the reference points is obtained by weighting according to the PRM.

9.3.1 Preprocessing

The preprocessing step determines the self illumination capabilities of the static scene. This information is computed for finite number of *reference points* on the surface, and we use interpolation for other points. The reference points are depicted by symbol $\times$ in figure 9.1. The reference points can be defined as points corresponding to the texel centers of the texture map of the surface.

The first step of preprocessing is the generation of certain number of *entry points* on the surface. These entry points are samples of first hits of the light emitted by moving light sources. During preprocessing we usually have no specific information about the position and the intensity of the animated light sources, thus entry points should cover the surfaces densely for all possible light source positions, and unit incoming radiance is assumed at these sample points. Entry points are depicted by symbol $\bullet$ in figure 9.1. Entry points are used as the start of a given number of light paths. A light path is a random or a quasi-random walk [65] along the surface. If the ray of a path leaves the scene, then the path is considered as terminated. In order to limit the length of paths, we can use random termination (Russian-roulette), or some deterministic decimation scheme [65] to determine the maximum length of each path.

The visited points of the generated paths are connected to all those reference points that are visible from them. In this way we obtain a lot of paths originating at an entry point and arriving at one of the reference points. The contribution of a path divided by the probability of the path generation is a Monte Carlo estimate of the indirect illumination caused by the given reference lighting environment from which the rays are sampled. The sum of the Monte Carlo estimates of paths associated with the same entry and reference point pair is stored. We call this data structure the *precomputed radiance map*, or *PRM* for short. Thus a PRM contains *items* corresponding to groups of paths sharing the same entry and reference points. Items that belong to the same entry point constitute a PRM *pane*.

9.3.2 Rendering

During real-time rendering, PRM is taken advantage of to speed up the global illumination calculation. The lights and the camera are placed in the virtual world (figure 9.2). The direct illumination effects are computed by standard techniques, which usually include some shadow algorithm to identify those points that are visible from the light source. PRM can be used to add the indirect self illumination. This step requires visibility calculations, which is for free, since this visibility information was already obtained during direct illumination computation when shadows were generated.

A PRM pane stores the self illumination computed for a light ray coming from the sampled direction and causing unit irradiance. During the rendering phase, however, we have to adapt to a different lighting environment, that is, to consider other light rays of different radiance and direction arriving at the same entry point. Taking into account the differences of carried radiance and incoming direction, the PRM pane associated with this entry point should be weighted in order to make it reflect the actual lighting situation. Doing this for every entry point hit by a light ray and adding up the results, we can obtain the visible color for each reference point. Then the object is rendered in a standard way with linear interpolation between the reference points.

In order to compute the weights, we have to take into account the contribution and the density of the light paths obtained during preprocessing. The mathematics needed for this computation is discussed in the following section.

9.3.3 Formal discussion of the method

A global illumination approach should evaluate the infinite Neumann series containing high-dimensional integrals. In the proposed approach these high-dimensional integrals are partly precomputed, i.e. the integral along all but one variable is calculated in the preprocessing phase. Then the remaining one-variate quadrature is evaluated during rendering.

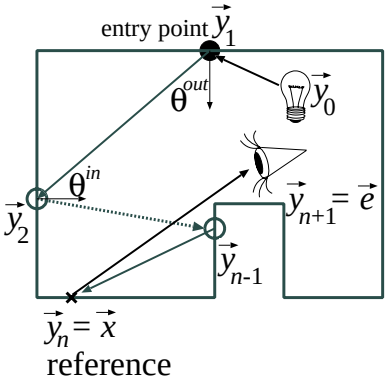


Figure 9.3: Notations used in the formal discussion

Global illumination computes the radiance (power density) of point $\vec{x}$ in the direction of eye $\vec{e}$ by summing the contribution of all light paths that originate at the light sources and arrive at the eye from point $\vec{x}$. Let us denote the visited points of a path by $\vec{y}_0$ (light source), $\vec{y}_1$ (entry point on the surface), $\vec{y}_2, \ldots, \vec{y}_{n-1}$ (internal path points), $\vec{y}_n = \vec{x}$ (visible point of the surface), $\vec{y}_{n+1} = \vec{e}$ (eye) (figure 9.3). The contribution of this path is

$$L^e(\vec{y}_0 \to \vec{y}_1) \cdot G_0 \cdot f_1 \cdot G_1 \cdot \ldots \cdot f_{n-1} \cdot G_{n-1} \cdot f_n,$$

where $L^e(\vec{y}_0 \to \vec{y}_1)$ is the emitted radiance from $\vec{y}_0$ toward $\vec{y}_1$,

$$G_k = G(\vec{y}_k, \vec{y}_{k+1}) = \frac{\cos\theta^{out}_{\vec{y}_k} \cdot \cos\theta^{in}_{\vec{y}_{k+1}}}{|\vec{y}_k - \vec{y}_{k+1}|^2} \cdot v(\vec{y}_k, \vec{y}_{k+1})$$

is the *geometry factor* where $\theta^{out}_{\vec{y}_k}$ is the angle between the surface normal at $\vec{y}_k$ and outgoing ray direction $\vec{y}_k \to \vec{y}_{k+1}$, $\theta^{in}_{\vec{y}_{k+1}}$ is the angle between the surface normal at $\vec{y}_{k+1}$ and incoming ray direction $\vec{y}_{k+1} \to \vec{y}_k$, *visibility function* $v(\vec{y}_k, \vec{y}_{k+1})$ indicates if the two points are not occluded from each other, and

$$f_k = f(\vec{y}_{k-1} \to \vec{y}_k \to \vec{y}_{k+1})$$

is the BRDF of point $\vec{y}_k$ for directions $\vec{y}_{k-1} \to \vec{y}_k$ and $\vec{y}_k \to \vec{y}_{k+1}$. In case of diffuse materials, the BRDF depends only on reflection point $\vec{y}_k$.

In order to calculate the radiance of $\vec{x}$ at the direction of the eye, all light paths ending at $\vec{x}$ should be considered and their contribution added, which leads to an infinite sum of high-dimensional integrals:

$$L(\vec{x} \to \vec{e}) = \sum_{n=2}^{\infty} \int_{\vec{y}_0} \ldots \int_{\vec{y}_{n-1}} L^e(\vec{y}_0 \to \vec{y}_1) \cdot G_0 \cdot f_1 \cdot \ldots \cdot G_{n-1} \cdot f_n \; dy_{n-1} \ldots dy_0. \tag{9.1}$$

Note that the length of the light paths starts at 2, because now we are interested in the indirect illumination only.

In order to speed up the evaluation of these high-dimensional integrals during rendering, the inner integrals along $\vec{y}_1, \ldots, \vec{y}_{n-1}$ are estimated in a preprocessing step. Suppose that we have a sampling scheme that generates partial light path $\vec{y}_1, \ldots, \vec{y}_{n-1}$ with density $d(\vec{y}_1, \ldots, \vec{y}_{n-1})$, where path length n is also a subject of sampling. If we used random sampling, then $d(\vec{y}_1, \ldots, \vec{y}_{n-1})$ would be equal to the product of probability density $p(\vec{y}_1, \ldots, \vec{y}_{n-1})$ and number of samples N, that is:

$$d(\vec{y}_1, \ldots, \vec{y}_{n-1}) = p(\vec{y}_1, \ldots, \vec{y}_{n-1}) \cdot N.$$

Instead of random sampling we can also apply quasi-random points transformed from low-discrepancy series [65]. Having obtained N samples $(\vec{y}_1^i, \ldots, \vec{y}_{n_i-1}^i)$, $(i = 1, \ldots, N)$ with the sampling scheme, we can replace the inner integrals of equation 9.1 by their Monte Carlo estimate:

$$\int_{\vec{y}_0} \ldots \int_{\vec{y}_{n-1}} L^e(\vec{y}_0 \to \vec{y}_1) \cdot G_0 \cdot f_1 \cdot \ldots \cdot G_{n-1} \cdot f_n \; dy_{n-1} \ldots dy_0 \approx$$

$$\sum_{i=1}^{N} \int_{\vec{y}_0} L^e(\vec{y}_0 \to \vec{y}_1^i) \cdot G_0^i \cdot \frac{f_1^i \cdot G_1^i \cdot f_2^i \cdot \ldots \cdot G_{n_i-1}^i \cdot f_{n_i}^i}{d(\vec{y}_1^i, \ldots, \vec{y}_{n_i-1}^i)} \; dy_0.$$

Note that factor

$$R^i = \frac{f_1^i \cdot G_1^i \cdot f_2^i \cdot \ldots \cdot G_{n_i-1}^i \cdot f_{n_i}^i}{d(\vec{y}_1^i, \ldots, \vec{y}_{n_i-1}^i)} \tag{9.2}$$

depends only on $(\vec{y}_1, \ldots, \vec{y}_n)$ and is independent of the illumination and viewing directions. This term can be precomputed for each reference point $\vec{x} = \vec{y}_n$, and stored together with pair $(\vec{x}, \vec{y}_1^i)$.

From stored values R^i, the reflected radiance can be computed as a low dimensional integral:

$$L(\vec{x} \to \vec{e}) = \sum_{i=1}^{N} \int_{\vec{y}_0} L^e(\vec{y}_0 \to \vec{y}_1^i) \cdot G_0^i \cdot R^i \; dy_0,$$

where the integrand is precomputed value R^i multiplied by *weight* G_0^i and emission $L^e(\vec{y}_0 \to \vec{y}_1^i)$, both representing the actual lighting and depend only on $\vec{y}_1^i$. For those samples that share this point, the order of weighting and summation can be changed, thus different precomputed factors R^i can be summed in the preprocessing phase. Let us consider all visible points and re-index the samples in a way that samples $1, \ldots, N_1$ share entry point $\vec{y}_1^1$, samples $N_1 + 1, \ldots, N_1 + N_2$ share entry point $\vec{y}_1^2$, etc. Restructuring the sums and the weighting we obtain:

$$L(\vec{x} \to \vec{e}) = \sum_{k=1}^{K} \int_{\vec{y}_0} L^e(\vec{y}_0 \to \vec{y}_1^k) \cdot G_0^k \cdot \left(\sum_{i=N_{k-1}+1}^{N_{k-1}+N_k} R^i \right) \; dy_0, \tag{9.3}$$

where K is the number of different entry points ($K < N$), and $N_0 = 0$. The following sums are the *items* of the *precomputed radiance map* (PRM) associated with reference point $\vec{x}$:

$$S_k = \sum_{i=N_{k-1}+1}^{N_{k-1}+N_k} R^i.$$

An item of the PRM is selected by entry point $\vec{y}_1$ and reference point $\vec{x}$, and represents the Monte Carlo estimate of the indirect illumination of point $\vec{x}$ when the light ray causing unit irradiance arrives at the surface at $\vec{y}_1$. The objective of preprocessing is the computation of these items for the reference points and entry points.

Having obtained the items of the PRM, the computation of the indirect illumination caused by a small light source is straightforward. Let us denote the origin and the area of the source by $\vec{y}_0$ and Δy_0, respectively. Substituting these into equation 9.3, the indirect reflected illumination of reference point $\vec{x}$ is:

$$L(\vec{x} \to \vec{e}) \approx \sum_{k=1}^{K} L^e(\vec{y}_0 \to \vec{y}_1^k) \cdot v(\vec{y}_0, \vec{y}_1^k) \cdot S^k \cdot \int_{\vec{y}_0} G_0^k \; dy_0 \approx$$

$$\sum_{k=1}^{K} L^e(\vec{y}_0 \to \vec{y}_1^k) \cdot v(\vec{y}_0, \vec{y}_1^k) \cdot \frac{\cos\theta_{\vec{y}_0}^{out} \cdot \cos\theta_{\vec{y}_1^k}^{in}}{|\vec{y}_0 - \vec{y}_1^k|^2 + \Delta y_0/\pi} \cdot S^k, \tag{9.4}$$

where $L^e(\vec{y}_0 \to \vec{y}_1^k)$ is the emission of the source into the direction of entry point $\vec{y}_1^k$, and $v(\vec{y}_0, \vec{y}_1^k)$ is the *visibility function* between the two points. Note that we applied the point to disc form factor approximation [39]. In the special case when the source is a point light with total emission power Φ^e, the formula is written as

$$L(\vec{x} \to \vec{e}) = \sum_{k=1}^{K} \frac{\Phi^e}{4|\vec{y}_0 - \vec{y}_1^k|^2 \pi} \cdot v(\vec{y}_0, \vec{y}_1^k) \cdot \cos\theta_{\vec{y}_1^k}^{in} \cdot S^k.$$

In order to use these formulae, we have to check whether or not the entry points are visible from the light source and carry out the summation only for the visible entry points. This visibility check, as well as the computation of the direct illumination can be implemented both by ray-tracing or on the GPU. In the latter case, we generate shadow maps for the direct illumination computations, which can readily be used for detecting the visibility of the entry points as well. In ray tracing based approaches, the visibility of the entry points from the light sources needs extra rays to be cast. However, the number of entry points is in the order of thousands, which is significantly smaller than the number of pixels of the image, thus the cost of indirect illumination is much smaller than that of the direct illumination.

9.3.4 Definition of the sampling scheme

The core of the presented method is the sampling scheme which obtains light paths $(\vec{y}_1, \ldots, \vec{y}_{n-1})$ with density $d(\vec{y}_1, \ldots, \vec{y}_{n-1})$. All those schemes where d is not zero for paths carrying nonzero radiance are unbiased in Monte Carlo sense, i.e. the expected value of the estimator gives back the correct result. We should prefer those sampling schemes that meet this criterion and have small variance, and consequently result in small error. The following subsections present methods meeting this requirement.

Entry point generation

The first step of sampling is the generation of N_e entry points on the surfaces with some density $d(\vec{y}_1)$. The simplest approach would obtain these points uniformly, first selecting patches proportionally to their area, then a random point on the patch with uniform distribution, resulting in $d(\vec{y}_1) = N_e/\mathcal{S}$ where $\mathcal{S}$ is the area of the surface.

On the other hand, if we have some a-priori knowledge of the possible future lighting, then we can built this information into this probability. Initially we place M *test points* $\vec{Y}_1, \ldots, \vec{Y}_M$ in the 3D space to sample the domain of possible light positions. From each test point $\vec{Y}_m$ we sample N_t number of random or quasi-random directions from a uniform $p_m(\omega)$ which mimics the light source radiance. The test point with the sampled direction define a ray, which is traced to look for a surface intersection. The found surface point will be entry point $\vec{y}_1$. The probability that we hit differential area dy_1 by a ray originating at test point $\vec{Y}_m$ is

$$p_m(\omega_{\vec{Y}_m \to \vec{y}_1}) \cdot \frac{dy_1 \cdot \cos\theta_{\vec{y}_1}^{in}}{|\vec{Y}_m - \vec{y}_1|^2} \cdot v(\vec{Y}_m, \vec{y}_1),$$

where $\theta_{\vec{y}_1}^{in}$ is the angle between the surface normal at $\vec{y}_1$ and the direction of $\vec{Y}_m$ from $\vec{y}_1$. If we shoot N_t rays from each test point $\vec{Y}_m$, then the density of entry points $\vec{y}_1$ is

$$d(\vec{y}_1) = N_t \cdot \sum_{m=1}^{M} p_m(\omega_{\vec{Y}_m \to \vec{y}_1}) \frac{\cos\theta_{\vec{y}_1}^{in}}{|\vec{Y}_m - \vec{y}_1|^2} \cdot v(\vec{Y}_m, \vec{y}_1).$$

Path generation with splitting

Taking the entry point as the origin $N_s \cdot a(\vec{y}_1)$ number of paths are tried to be generated where $a(\vec{y}_1)$ is the albedo of the surface, and *splitting factor* N_s is a global constant of the method. The

reason behind splitting is that in this way the number of random paths can be increased without increasing the number of entry points, i.e. the size of the PRM. The direction of the ray originating in a particular entry point is obtained with cosine distribution.

If the ray hits the surface again, then at the hit point a new direction is sampled with BRDF sampling and this step is continued until the path either leaves the surface, or we decide to terminate it according to Russian roulette.

If the path is terminated, then the hit points are assumed to be *virtual light sources* [65, 130] that illuminate the reference points visible from them. In order to compute this, all reference points are tried to be connected with the points of the paths by shadow rays.

Sampling density

The proposed instructions establish a sampling scheme that obtain point sequences $(\vec{y}_1, \ldots, \vec{y}_{n-1})$ of random length n on the surface of the object. Let us now consider density d of these sequences.

From entry point $\vec{y}_1$ we initiate $N_s \cdot a(\vec{y}_1)$ random paths sampling the direction from cosine distribution, thus the density of selecting $\vec{y}_2$ as the second point of the path, given entry point $\vec{y}_1$, is:

$$\frac{a(\vec{y}_1)}{\pi} \cdot N_s \cdot G(\vec{y}_1, \vec{y}_2) \cdot v(\vec{y}_1, \vec{y}_2).$$

In the following steps, we apply Russian roulette and BRDF sampling again to obtain a new direction. Note that in case of diffuse materials BRDF sampling results in the application of cosine distribution similarly to the entry point. The density of a complete path $(\vec{y}_1, \ldots, \vec{y}_{n-1})$ given that it originates at $\vec{y}_1$ is then

$$d_{\vec{y}_1}(\vec{y}_1, \ldots, \vec{y}_{n-1}) = \frac{a(\vec{y}_1)}{\pi} \cdot N_s \cdot G(\vec{y}_1, \vec{y}_2) \cdot \ldots \cdot \frac{a(\vec{y}_{n-2})}{\pi} \cdot G(\vec{y}_{n-2}, \vec{y}_{n-1}).$$

The unconditional density of sequences $(\vec{y}_1, \ldots, \vec{y}_{n-1})$ is

$$d(\vec{y}_1, \ldots, \vec{y}_{n-1}) = d(\vec{y}_1) \cdot d_{\vec{y}_1}(\vec{y}_1, \ldots, \vec{y}_{n-1}).$$

It is interesting to substitute this density into the formula of R_i factors (equation 9.2) assuming that the light transport is computed just on a single wavelength, i.e. when the continuation probability a equals to $f_r \cdot \pi$. Note that most of the geometric factors and BRDFs can be cancelled from the numerator and denominator, and we obtain

$$R_i = \frac{G^i_{n_i-1} \cdot f^i_{n_i}}{N_s \cdot d(\vec{y}_1)}.$$

In case of spectral transfers, the geometry factors can still be eliminated, but the BRDFs cannot. Instead, they are replaced by their spectral albedo divided by the luminance of this albedo.

9.4 Implementation

The new algorithm consists of a preprocessing step, which builds the PRM, and a rendering step, which takes advantage of the PRM to evaluate self illumination. For the preprocessing step we implemented a combined CPU ray-tracing and GPU method. On the other hand, the rendering step was realized completely on the GPU.

9.4.1 Preprocessing

The first step of preprocessing is the generation of entry points, then the paths, which is executed by the CPU. Having completed a path, we compute the visibility between each path point and each reference point. Assuming that the entry point has unit irradiance, the contribution of the reference points are evaluated. For each reference point, the sum of the contributions is stored together with the index of the entry point, which constitutes the PRM.

While generating random walks is best done on the CPU, the GPU is better in computing the effect of path points on reference points. In fact, we should execute a virtual light source algorithm [65, 130]. Since the reference points are the texels of a texture map, the virtual light source algorithm should be implemented in a way that it renders into a texture map. Items are computed by rendering into the texture with the random walk nodes as light sources. Visibility can be determined using the shadow map technique.

9.4.2 PRM representation as textures

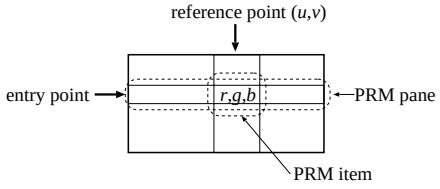


Figure 9.4: Representation of a PRM as an array indexed by entry points and reference points. A single element of this map is the PRM item, a single row is the PRM pane.

A single texel stores a PRM item that represents the contribution of all paths connecting the same entry point and reference point. A PRM can thus be imagined as an array indexed by entry points and reference points and storing the radiance on the wavelengths of red, green, and blue (figure 9.4). Since a reference point itself is identified by two texture coordinates (u, v), a PRM can be stored either in a 3D texture or in a set of 2D textures, where each represents a single PRM pane (i.e. a row of the table in figure 9.4, which includes the PRM items belonging to a single entry point).

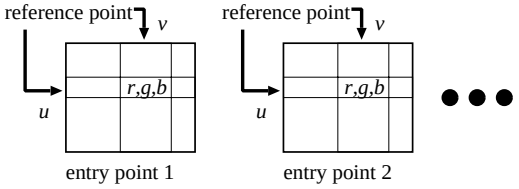


Figure 9.5: Uncompressed PRM structure

The set of 2D textures is called *uncompressed PRM* (figure 9.5). The number of the textures is equal to the number of entry points. However, the graphics hardware has just a few texture units. Fortunately, this can be sidestepped by tiling the PRM panes into one or more larger textures.

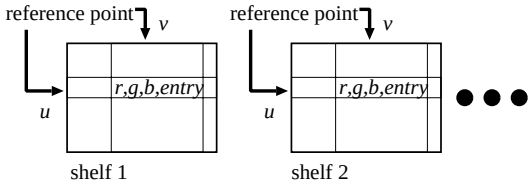


Figure 9.6: Compressed PRM structure

Furthermore, if we consider objects seen from outside, the contribution corresponding to a single entry point may effect only a small fraction of reference points, and thus, texels. Therefore, a large portion of a PRM pane contains texels with zero contribution. This situation lends itself to an effective compression technique, which gathers the valuable texels into just a few texture

maps, called *shelves*. A shelf is a mixture of the items of different PRM panes, thus we extend the R,G,B channels used for the radiance representation with a fourth value, which will store the identification of the entry point. However, this fourth value should be accessed through separate texture sampler, since the contribution is filtered, but the identification of the entry point must not be interpolated. The collection of these shelves is called *compressed PRM* (figure 9.6).

Obviously, ignoring the zero or small contributions this method could reduce the otherwise affordable memory consumption of 0.5 Mbyte per entry point to a fraction. However, care must be taken that no significant items are discarded. That is, we have to use enough shelves to accomodate for the reference point that is influenced by the most entry points. Unfortunately, ordering the items of a reference point according to the contribution value and discarding the smallest values is not a fail-safe approach, as entry points may get more significant as the actual position of the light changes in the final rendering step. Considering this, compressed PRMs are only feasible for objects with a high number of small, unconnected cavities, where a large number of entry points would be required, but the contribution of a single entry point is spatially delimited. Obviously, the exact opposite applies to spaces like caves or room interiors (figures 9.8, 9.9, and 9.10), where an uncompressed map is the better choice, mostly because of the fact that there are very few zero texels that should be discarded.

9.4.3 Entry point clusters

Using the method as described above allows us to render indirect illumination interactively with a typical number of 256 entry points. While this figure is generally considered sufficient for a medium complexity scene, difficult geometries and animation may emphasize virtual light source artifacts as spikes or flickering, thus requiring even more samples. Simply increasing the number of entry points and adding corresponding PRM panes would quickly challenge even the latest hardware in terms of texture memory and texture access performance. To cope with this problem, we can apply an approximation (a kind of lossy compression scheme), which keeps the number of panes under control when the number of entry points increase.

The key recognition is that if two entry points are close and lay on similarly aligned surfaces, then their direct illumination will be probably very similar during the light animation. Of course this is not true when a hard shadow boundary separates the two entry points, but due to the fact that a single entry point is responsible just for a small fraction of the indirect illumination, these approximation errors can be tolerated and do not cause noticeable artifacts. This property can also be understood if we examine how clustering affects the represented indirect illumination. Clustering entry points corresponds to a low-pass filtering of the indirect illumination, which is usually already low-frequency by itself, thus the filtering does not cause significant error. Furthermore, errors in the low frequency domain are not disturbing for the human eye. Clustering also helps to eliminate animation artifacts. When a small light source moves, the illumination of an entry point may change abruptly, possibly causing flickering. If multiple entry points are clustered together, their average illumination will change smoothly. This way clustering also trades high-frequency error in the temporal domain for low-frequency error in the spatial domain.

Therefore, to reduce the number of panes, contributions of a cluster of nearby entry points are added and stored in a single PRM pane. As these *clustered entry points* cannot be separated during rendering, they will all share the same weight when the entry point contributions are combined. This common weight is obtained as the average of the individual weights of the entry points. Clusters of entry points can be identified by the K-means algorithm [62] or, most effectively, by a simple object median splitting kd-tree. It is notable that increasing the number of samples via increasing cluster size N_c has only a negligible overhead during rendering, namely the computation of more weighting factors. The expensive access and combination of PRM items is not affected. This way the method can be scaled up to problems of arbitrary complexity at the cost of longer preprocessing only.

9.4.4 Rendering

While rendering the final image, the values stored in the PRM should be summed according to equation 9.4. Computing *weighting factor*

$$v(\vec{y}_0, \vec{y}_1^k) \cdot \frac{\cos\theta_{\vec{y}_0}^{out} \cdot \cos\theta_{\vec{y}_1^k}^{in}}{|\vec{y}_0 - \vec{y}_1^k|^2 + \Delta y_0/\pi}$$

involves a visibility check that could be done using ray casting, but, as rendering direct illumination shadows would require a shadow map anyway, it can effectively be done in a shader, rendering to a one-dimensional texture of weights. Although these values would later be accessible via texture reads, they can be read back and uploaded into constant registers for efficiency. Furthermore, for an uncompressed PRM, zero weight textures can be excluded, sparing superfluous texture accesses.

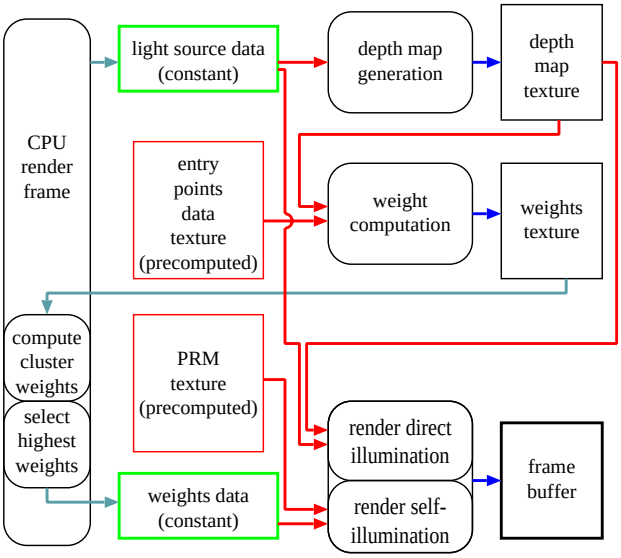


Figure 9.7: Dataflow in the rendering step

In order to find the indirect illumination at a reference point, the corresponding PRM items should be read from the textures and their values summed having multiplied them by the weighting factors and the light intensity. In the uncompressed case we can limit the number of entry points to those having the highest weights. Selection of the currently most significant texture panes can be done on the CPU before uploading the weighting factors as constants.

9.5 Results

The proposed method has been implemented on a P4/2GHz computer with NV6800GT graphics card. Figures 9.8 and 9.9 show a marble chamber test scene consisting of 3335 triangles, rendered on 1024×768 resolution. We used 4096 entry points. Entry points were organized into 256 clusters ($N_c = 4096/256$). Splitting factor N_s was 2. We set the uncompressed PRM pane resolution to 256×256, and used the 32 highest weighted entry clusters. In this case the peak texture memory requirement was 128 Mbytes. The constant parameters of the implementation were chosen to fit easily with hardware capabilities, most notably the maximum texture size and the number of temporary registers for optimized texture queries, but these limitations can be sidestepped easily. As shown in figure 9.8, an impressive entry point density was achieved. Assuming an average albedo of 0.66 and a splitting factor of 2, the 4096 entry points displayed in figure 9.8 translate to approximately 24000 virtual light sources.

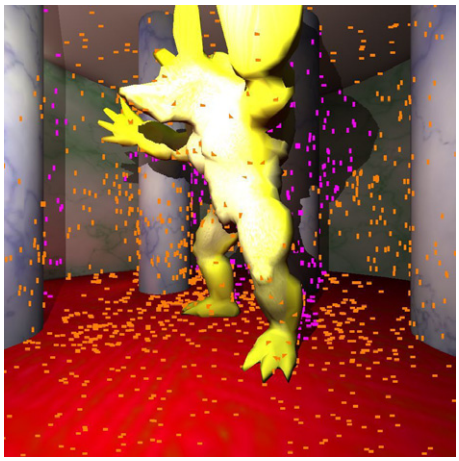


Figure 9.8: Entry points

For this scene, the preprocessing took 8.5 sec, which can further be decomposed as building the kd-tree for ray casting (0.12 sec), light tracing with ray casting (0.17 sec), and PRM generation (8.21 sec). Having obtained the PRM, we could run the global illumination rendering on 40 frames per second interactively changing the camera and light positions.

Figure 9.9 shows screen shots where half of the image was rendered with the new algorithm, and the other half with local illumination to allow comparisons. The effects are most obvious in shadows, but also notice color bleeding and finer details in self-illumination that could not be achieved by fake methods like using an ambient lighting term.

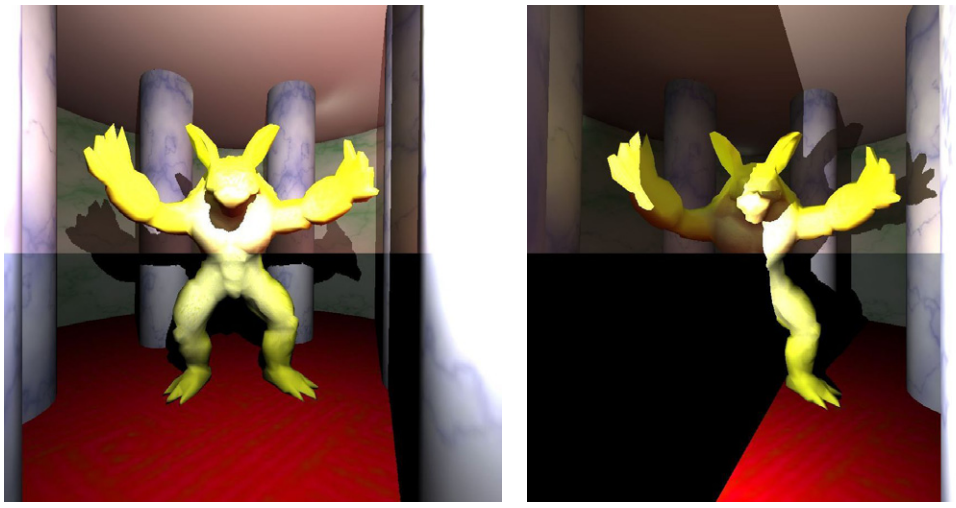
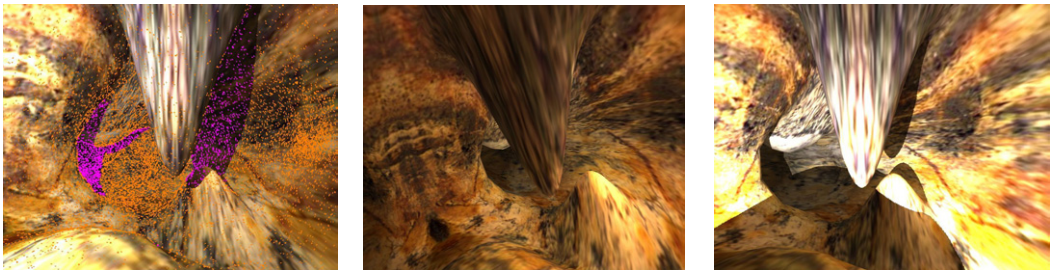


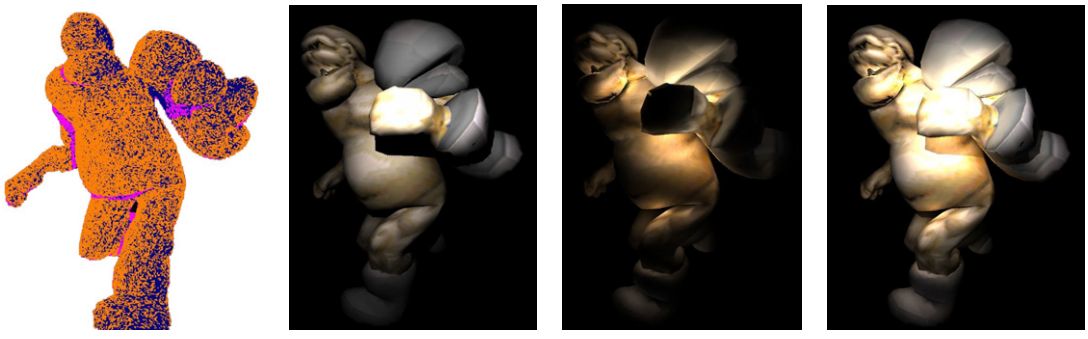
Figure 9.9: Comparison of local illumination and the proposed global illumination rendering methods. The lower half of these images has been rendered with local illumination, while the upper half with the proposed global illumination method on 40 FPS.

Figure 9.10 shows a cave rendered with similar parameters, but increasing the number of entry points to 65536 and simultaneously adapting the cluster size to 256.



entry points | indirect illumination | direct + indirect illumination

Figure 9.10: The Cave defined by 6148 faces rendered on 35 FPS



entry points | direct illumination | indirect illumination | complete illumination

Figure 9.11: The Ogre defined by 4020 faces, and rendered on 43 FPS

9.6 Conclusions

This tool proposed a global illumination algorithm for rigid bodies and for static scenes, but allowing fast moving light sources and camera. Since the geometry is static, we could pre-compute most of the integrals of the infinite Neumann series of a full global illumination solution. This means that in the rendering phase, when light sources are introduced, only a one-variate integral needs to be evaluated, which is possible in real-time. The method can be used in walk-through animations when the avatar takes his own light sources with him, in lighting design when the light sources are placed interactively, and also in games and real-time animations. Of course, the geometry is not constant in games, but in a typical gaming environment, the static environment is usually much larger than the moving dynamic objects. In this case we can suppose that the dynamic objects alter only the direct illumination of the large static environment, but its indirect illumination is not affected. Thus we can still use the proposed method for rendering the environment under dynamic lighting, and apply some fast final gathering technique (e.g. environment mapping) to shade dynamic objects.

Chapter 10

Fresnel Reflection Tool

We propose a physically plausible BRDF model, which takes into account the features of the current graphics processors. This BRDF model is a product of the microfacet distribution, the geometric term and the Fresnel function. For non-metals, the Fresnel term can be rather simple if we apply the approximation proposed by Schlick. In general case, especially for metals, the refraction index becomes a complex number, which makes the evaluation rather computation intensive. We present an accurate simplification, which can also cope with complex refraction indices unlike Schlick's model. In order to establish the approximation, Schlick's formula is rescaled and the residual error is compensated by a simple rational approximation. The resulting formula can present realistic metals and is simple enough to be implemented on the vertex or pixel shader, and used in animations and games.

10.1 Introduction

10.1.1 Physical background

Material models are usually defined by Bidirectional Reflectance Distribution Functions (BRDFs) that describe the chance of reflection for different pairs of incoming and outgoing light directions.

Nowadays with the emergence of programmable vertex and pixel shaders, we can use more sophisticated material models instead of the simple Phong-Blinn reflection model. Unlike the Phong-Blinn model, these sophisticated models can be physically plausible, that is, they do not violate basic rules of optics, including the Helmholtz symmetry and energy conservation.

10.1.2 GPU friendly BRDFs

When an image is rendered, the minimal number of BRDF evaluations is equal to the product of the numbers of pixels and light sources. Usually, the number of evaluations is even larger since we also compute the illumination of those points that turn out to be invisible later. So we can conclude that the efficiency of the BRDF evaluations is crucial to achieve high frame rates, even on today's high performance GPUs.

The BRDF model can be evaluated on the vertex shader (Gouraud shading) or on the pixel shader (Phong shading). It is also possible to combine the two approaches and evaluate the BRDF only partially at the vertices. These partial evaluations are linearly interpolated and are finalized in every pixel by the pixel shader.

BRDF models can be evaluated algebraically, or we can rely on precomputed values stored in textures at least partially. Though vertex shader texture fetches are now possible, textures are accessed far more effectively from the pixel shader, thus this approach is less feasible in vertex shader BRDF evaluations. The other problem of precomputed textures used for BRDF data is that a BRDF model has many variables. It depends on the incoming direction, normal vector and viewing direction. Thus in tangent space, the incoming and viewing directions can be defined

by 4 scalars, which can be reduced to 3 scalars in case of isotropic materials. The storage of a three-variate function, however, requires volumetric textures and is quite expensive.

Considering these aspects, we can establish several requirements towards a GPU friendly BRDF model. First of all, it should be algebraically simple. To ease lookup table realizations, it should be decomposed to several factors, which depend on one or two scalar variables. Finally, from the point of view of vertex shader implementations, it is favorable if the BRDF can be decomposed to factors, where some factors are approximately linearly varying inside a triangle, enabling us to evaluate them in the vertex shader.

10.2 Physically plausible BRDF models

A microfacet based specular BRDF model usually has the following product form [24, 49]:

$$P(\vec{N}\cdot\vec{H})\cdot G(\vec{L},\vec{N},\vec{V})\cdot F(\vec{N}\cdot\vec{H},\lambda),$$

where $\vec{N}$ is the surface normal, $\vec{L}$ is the illumination direction, $\vec{V}$ is the viewing direction, $\vec{H}$ is the halfway vector between the illumination and viewing directions, and λ is the wavelength of light. The meaning of the three factors is as follows.

Microfacet distribution $P(\vec{N}\cdot\vec{H})$ defines the roughness of the surface by describing the density of microfacets that can ideally reflect from the illumination to the viewing direction.

Geometric term $G(\vec{L},\vec{N},\vec{V})$ shows how much of the ideal reflections can actually occur, and is not blocked by another microfacet (called masking and self-shadowing). Geometric term G is independent of the material properties, and it causes a general reduction of the specular term for certain illumination and viewing directions. Such reduction should be compensated in the diffuse reflection, since we can assume that photons reflected on the microfacets by multiple times contribute to the diffuse (also called matte) part. It also means that the matte and specular parts are not independent, as assumed by most of the BRDF models, but are coupled by an appropriate weighting, which depends on the viewing direction [4, 64].

Finally, Fresnel term $F(\vec{N}\cdot\vec{H},\lambda)$ equals to the probability that a photon is reflected from the microfacet considered as an ideal mirror. The normal of the microfacet that can reflect from illumination direction $\vec{L}$ to viewing direction $\vec{V}$ is the halfway vector $\vec{H}=(\vec{V}+\vec{L})/|\vec{V}+\vec{L}|$ (according to the law of ideal reflection). The Fresnel term is the only factor that is wavelength dependent, thus it is the primary source of coloring. That is why the accurate computation of the Fresnel term is so important to present realistic look for materials.

10.2.1 The origin of the refraction index and the Fresnel factor

In addition to the photon concept, another way of understanding light is the wave model. As light travels in a material, the electromagnetic disturbance of the wave forces the electrons of the material to oscillate as well, which in turn radiate electromagnetic waves [37]. This new wave can be observed as a reflected wave on the boundary of different materials, and also combines with the original wave inside the material. This combination makes the original wave apparently change its speed, i.e. the speed will be different from the speed in vacuum. On the other hand, maintaining forced oscillation requires energy, thus the wave will die away in the material, and the wave amplitude diminishes exponentially. Let us consider a one dimensional wave of the electric field in point x and time t, for example:

$$\psi(x,f)=A\cdot e^{-kx}\cdot e^{j\omega(t-\frac{x}{v})},$$

where A is the initial amplitude, k is the extinction coefficient, j is the imaginary unit on the complex plane, $\omega=2\pi/\lambda$ is the phase frequency of the wave (which is inversely proportional to wavelength λ), and v is the speed of the wave inside the material. Let us express the speed of the electromagnetic wave relative to the speed in vacuum, i.e. $v=v_0/n$. With this notation, we can write:

$$\psi(x,f)=A\cdot e^{j\omega\left(t-\frac{x(n-kj)}{v_0}\right)}.$$

The $n - kj$ term is called the complex refraction index of the material. According to the definition above, its real part describes how fast the light traverses inside the material, while the imaginary part defines how quickly the wave dies away (Figure 10.1).

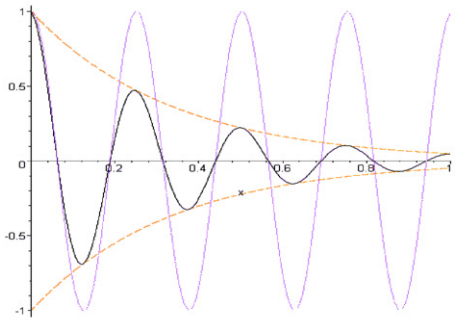


Figure 10.1: Plots for $k = 0$ and $k = 3$ ($n = 2.5$)

Based on the explained model of the forced oscillation of electrons, it is possible to derive a similar wave equation, thus the complex refraction index can also be expressed by material constants. A simple and practical formula is [37]:

$$n(\omega) - jk(\omega) = 1 + \frac{q^2}{2\varepsilon_0 m} \cdot \sum_k \frac{N f_k}{\omega_k^2 - \omega^2 + j\delta_k \omega}$$

where q and m are the charge and mass of the electron, ε_0 is the permittivity constant, k runs over the natural angular frequencies ω_k of the electrons in the atoms of the material, δ_k is the dumping coefficient of the oscillation due to enery loss, N is the number of atoms in a unit volume, and f_k is the weight belonging to this resonance frequency.

We can make the following observations by examining this formula. If the natural frequencies are much greater than the frequency of the light, then the imaginary part is negligible and the real part is greater than 1. This is the case for most of the dielectric materials. However, if the natural frequency is close to the frequency of the visible light, then the imaginary part can be large, and the real part can be either smaller or greater than 1 (this also means that the phase speed inside the material can be larger than the speed of vacuum). Metals do not let the electromagnetic wave travel inside the material (just think of the Faraday cage experiment). The wave dies away quickly due to the larger extinction coefficient.

Applying the Maxwell equations, it is possible to compute the ratio of the amplitude of the electric field reflected off a plane boundary and the amplitude of the incident wave. In rendering we work with power (or radiance, which is the density of power). Since the power of the wave is proportional to the square of the amplitude, the reflectance of a surface is proportional to the square of the amplitude ratio. This ratio also depends on the polarization of the light. The formula of an arbitrary polarization can be expressed from two basic solutions, when the oscillation is parallel or perpendicular to the surface. In the final result for these cases, the complex refraction index also plays a crucial role:

$$F_p = \left| \frac{\cos\theta_{in} - (\eta + \kappa j)\ \cos\theta_{out}}{\cos\theta_{in} + (\eta + \kappa j)\ \cos\theta_{out}} \right|^2, \qquad F_s = \left| \frac{\cos\theta_{out} - (\eta + \kappa j)\ \cos\theta_{in}}{\cos\theta_{out} + (\eta + \kappa j)\ \cos\theta_{in}} \right|^2.$$

Assuming that the light is not polarized, the ratio of the reflected and incident radiance can be expressed by the Pythagoras theorem:

$$F = \frac{F_s + F_p}{2}.$$

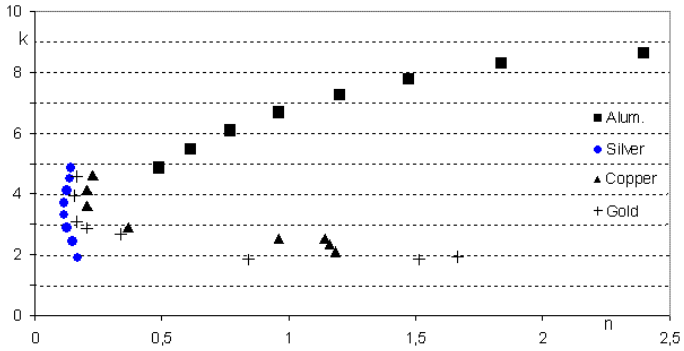


Figure 10.2: Complex refraction indices for different metals, for different wavelengths ($\lambda = 400..800nm$)

Turning back to the photon model of the light, on a given wavelength, the power is proportional to the number of photons, thus the Fresnel term can also be interpreted as the probability that a particular photon is reflected from the boundary of the material.

The Fresnel term depends on three arguments: n, k and θ, where n and k are the real and imaginary parts of the refraction index, respectively, and θ is the angle of incidence for the given microfacet. To simplify the notation let us introduce a new variable x for the term $\cos\theta$. The following form of the Fresnel formula is cited from [39].

$$F_s = \frac{a^2 + b^2 - 2a\cos\theta + \cos^2\theta}{a^2 + b^2 + 2a\cos\theta + \cos^2\theta},$$

$$F_p = F_s \cdot \frac{a^2 + b^2 - 2a\sin\theta\tan\theta + \sin^2\theta\tan^2\theta}{a^2 + b^2 + 2a\sin\theta\tan\theta + \sin^2\theta\tan^2\theta},$$

where a and b are defined by the following equations:

$$2a^2 = \sqrt{(n^2 - k^2 - \sin^2\theta)^2 + 4n^2k^2} + (n^2 - k^2 - \sin^2\theta)$$

$$2b^2 = \sqrt{(n^2 - k^2 - \sin^2\theta)^2 + 4n^2k^2} - (n^2 - k^2 - \sin^2\theta)$$

An optimized shader code of this formula is:

```
// microfacet normal
float3 H = normalize( L+V );
float ctheta = dot(H,L);
float ctheta2 = ctheta * ctheta;
float stheta2 = 1 - ctheta2;
float stheta = sqrt(stheta2);
float n2 = n*n;
float k2 = k*k;
float nk_stheta = n2 - k2 - stheta2;
float nk_stheta2 = nk_stheta*nk_stheta;
float rhs = sqrt(nk_stheta2 + 4*n2*k2);
float rhs_b = rhs - nk_stheta;
float a2 = ( rhs + nk_stheta )/2;
float a = sqrt(a2);
float b2 = ( rhs - nk_stheta )/2;
float Fs = (a2 + b2 - 2*a*ctheta + ctheta2) /
(a2 + b2 + 2*a*ctheta + ctheta2);
float c = a2 + b2 + stheta2*stheta2/ctheta2;
float d = 2*a*stheta2/ctheta;

// Fresnel term
float F = Fs * ( c / ( c+d ));
```

The computation of the exact Fresnel term requires a lot of operations, which is quite expensive even on the graphics hardware. In real time applications we need its approximations, which are much cheaper to evaluate, but are accurate enough not to destroy image quality. The main objective of this paper is to propose such Fresnel approximations, and to show how a simple, but physically plausible BRDF model can be realized.

10.3 Schlick's Approximation

For many non-metallic materials, the extinction coefficient is quite small, which allows us to ignore the imaginary part altogether. The assumption of the extinction coefficient being zero has also been made by Schlick, who has found a simple rational approximation for the Fresnel term in this case [112]:

$$F_{Schlick} = \frac{(n-1)^2 + 4n(1-\cos\theta)^5}{(n+1)^2}$$

As Figure 10.4 shows, the formula provides a fairly good approximation if the extinction coefficient is really zero (k=0). Apparently, the rational approximation was optimized for n=1.5 since for this value, the error is minimal (Figure 10.3). This formula is even further simplified in an NVidia technical report [53].

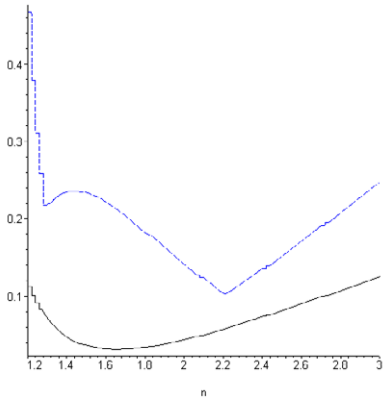


Figure 10.3: Absolute (solid) and relative error (dashed) of Schlick's formula

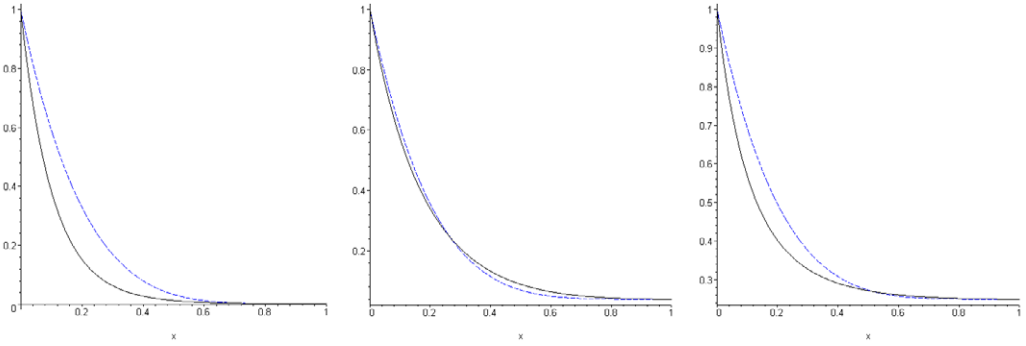


Figure 10.4: Comparing the original Fresnel function (solid line) and Schlick's formula (dashed line) for $n = 1.1$, $n = 1.5$ and $n = 3.0$ when $x = \cos\theta$ is in [0,1].

Figure 10.4 compares the shape of the Schlick's approximation with the original Fresnel function. Note that the shapes of the Schlick's approximations is the same for all of the plots, different n values cause just a scaling and a bias.

10.4 The new approximation handling metallic materials as well

Now let us concentrate on metals or other materials having complex refraction index ($k > 0$) and have a look at the Fresnel values for a fixed n value. We can notice that for smaller extinction coefficients the Fresnel term is a monotonously increasing function of $x = \cos\theta$, but does not reduce to small values at larger $x = \cos\theta$ values unlike Schlick's formula. On the other hand, for larger k values there exists a local minimum. (See the 3D plot and its plane sections on Figure 10.5 and Figure 10.5, respectively).

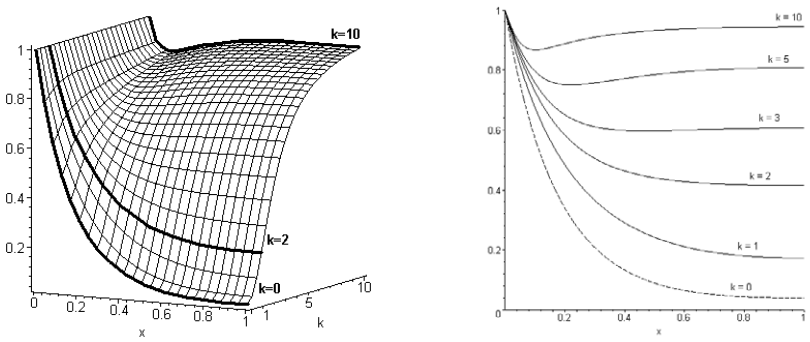


Figure 10.5: Fresnel term for different k and $x = \cos\theta$ values ($n = 1.5$, $k = 0..10$)

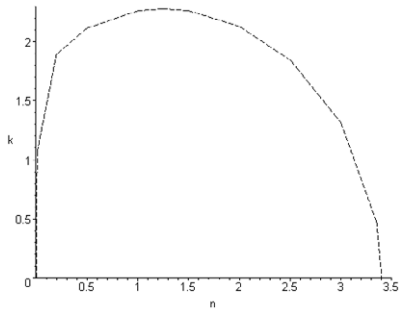


Figure 10.6: For (n, k) values inside the plotted curve, the Fresnel function is monotonous. For larger n and/or k values there exists a local minimum.

However, the minimal k value where this local minimum appears varies for different n values (see Figure 10.6), thus it seems to be rather difficult to create a rational approximation that universally catches the main characteristics of the original function.

If we apply Schlick's formula for metals, the result will be erroneous since the approximation does not obey the value of the original function at $\cos\theta = 1$. Furthermore, the Fresnel function may have a local minimum that the approximation is unable to follow.

In the followings we propose two improvements to significantly reduce the error of the original Schlick's approximation.

10.4.1 Rescaling the Schlick's function

To reduce the error of the approximation we shall rescale Schlick's formula so that it will obey the value of the original function not only at $\cos\theta = 0$, but at $\cos\theta = 1$ as well. To achieve this, we

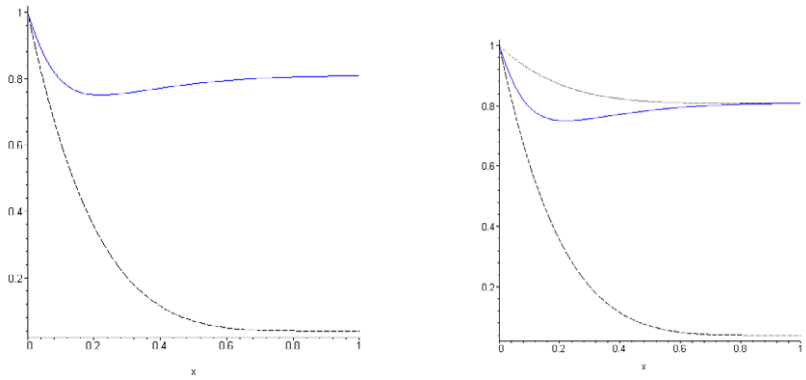


Figure 10.7: Comparing the Fresnel term (solid) and Schlick's approximation (dashed) for $n = 1.5, k = 5$. On the right, our proposal is also shown (dotted).

Figure 10.8: A copper ring rendered with the original Fresnel term and with Schlick's approximation

compute the exact and approximated function values at $\cos\theta = 1$, as follows:

$$F_1 = 1 - Fresnel(n, k, \cos\theta = 1) = 1 - \frac{n^2 - 2n + k^2 + 1}{n^2 + 2n + k^2 + 1} = \frac{4n}{(n+1)^2 + k^2}$$

$$S_1 = 1 - F_{Schlick}(n, \cos\theta = 1) = 1 - \frac{(n-1)^2}{(n+1)^2} = \frac{4n}{(n+1)^2}$$

Thus the scaling factor is:

$$\frac{F_1}{S_1} = \frac{(n+1)^2}{(n+1)^2 + k^2}.$$

Now the modified (rescaled) Schlick's formula can be expressed as follows:

$$F^*(n, k, \cos\theta) = 1 - \frac{F_1}{S_1}\left(1 - F_{Schlick}(n, \cos\theta)\right) =$$

$$= 1 - \frac{(n+1)^2}{(n+1)^2 + k^2} \cdot \frac{4n(1 - (1 - \cos\theta)^5)}{(n+1)^2} = 1 - 4n \cdot \frac{(1 - (1 - \cos\theta)^5)}{(n+1)^2 + k^2}$$

The resulting formula is able to deal with complex refraction indices and is simple enough for practical applications:

$$F^*(n, k, \cos\theta) = 1 - 4n \cdot \frac{(1 - (1 - \cos\theta)^5)}{(n+1)^2 + k^2} = \frac{(n-1)^2 + k^2 + 4n(1 - \cos\theta)^5}{(n+1)^2 + k^2}.$$

After examining the relative error of the modified approximation (see Figure 10.9) we can conclude that this simple modification enables us to extend the original Schlick's formula to complex refraction indices without significant increase of the relative error (let us compare Figure 10.3 and Figure 10.9).

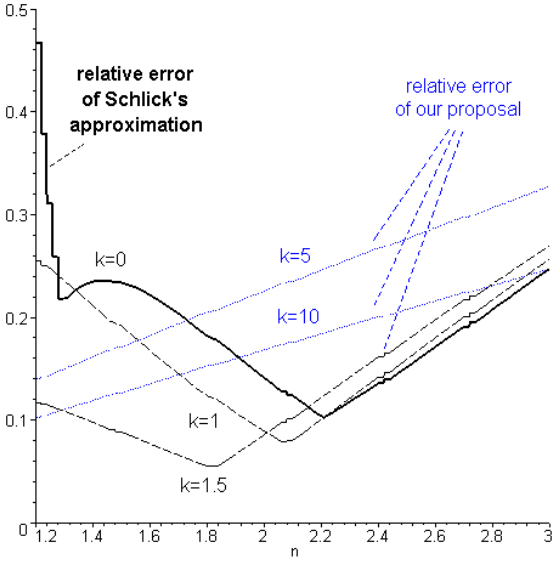


Figure 10.9: Comparing the relative error of the original Schlick's formula and our proposal for different n and k values

10.5 A simple, physically plausible BRDF model

In order to obtain a complete microfacet based BRDF model, we have to define the geometric term and the microfacet distribution as well, in addition to the Fresnel function.

10.5.1 Geometric term

The geometric term defines the probability that a photon aiming at an appropriately oriented microfacet does not collide with other microfacet before (shadowing) or after (masking) reaching the appropriately oriented microfacet. For example, the Cook-Torrance BRDF model uses the following geometric term:

$$g(\vec{N},\vec{L},\vec{V}) = \min\left\{1, 2\frac{(\vec{N}\cdot\vec{H})(\vec{N}\cdot\vec{V})}{(\vec{V}\cdot\vec{H})}, 2\frac{(\vec{N}\cdot\vec{H})(\vec{N}\cdot\vec{V})}{(\vec{V}\cdot\vec{H})}\right\}$$

$$G(\vec{N},\vec{L},\vec{V}) = \frac{g(\vec{N},\vec{L},\vec{V})}{2(\vec{N}\cdot\vec{L})(\vec{N}\cdot\vec{V})}$$

The shader implementation of this term is:

```
float NL = dot(N,L);
float NV = dot(N,V);
float NH = dot(N,H);
float VH = dot(H,V);

float f1 = 2 * NH * NV / VH;
float f2 = 2 * NH * NL / VH;

float g = min(1, min( f1, f2 )) / 4;
float G = g / NL / NV;
```

Note that the geometric term is rather complicated, and usually depends on all input vectors, thus its tabulation is out of question. To cope with this problem, Kelemen et al. [64] suggested

a simplification to the geometric term that factors out the dependence of the light, normal and viewing vectors, and proposed a model that depends only on a single scalar.

$$G(\vec{N},\vec{L},\vec{V}) \approx \frac{1}{2\cdot(1+(\vec{L}\cdot\vec{V}))} = \frac{1}{4(\vec{H}\cdot\vec{L})^2}.$$

Figure 10.10, Figure 10.11 and Figure 10.12 compare the original geometry term of the Cook-Torrance model and the simplified version. Incoming light direction and ideal reflection direction are also shown.

Examining the figures we can conclude that the main characteristics of the modified term are similar to the original one. Slight differences occur at the sharp edges of the original function plot.

Figure 10.10: The original and the simplified geometric term for 0 ° incident angle

Figure 10.11: The original and the simplified geometric term for 30 ° incident angle

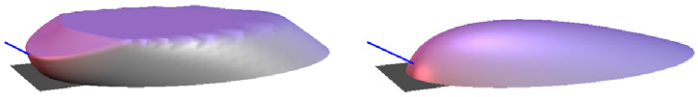


Figure 10.12: The original and the simplified geometric term for 60 ° incident angle

On the other hand, the simplified geometric term is much easier to compute. More importantly, it depends on only the cosine of a single angle that is between the halfway vector and the light vector. Note that the same cosine angle shows up in the Fresnel term as well, thus the two factors can be combined, and we can store their product in a look up table.

10.5.2 Geometric term

The microfacet distribution function defines the probability density that microfacet normals enclose angle δ with the mean surface normal. The theory of scattering of electromagnetic waves lead to the application of the Beckmann distribution:

$$P(\delta) = \frac{1}{m^2\cdot\cos^4\delta}\cdot e^{-\frac{\tan^2\delta}{m^2}}$$

where δ is the angle between halfway vector $\vec{H}$ and normal vector $\vec{N}$. In the Beckmann distribution parameter m describes how rough the surface is.

If it were too time consuming to evaluate this formula, we can select other probability densities as well which are maximal at $\delta = 0$. A simple cosine lobe is usually sufficient:

$$P(\delta) = \frac{s+2}{2\pi}\cdot\cos^s\delta$$

In this cosine distribution the roughness parameter is exponent s.

10.6 Conclusions

This chapter proposed a new approximation for the Fresnel function. Unlike previous approaches, we did not assume that the imaginary part of the refraction index is negligible, thus our model can be applied for a wider range of materials. A particularly important material class contains metals, where the imaginary part of the refraction index is significant.

In addition to the Fresnel term, we also investigated the possibilities of the simplification of the geometric term and the microfacet distribution function. Putting all these simplification together, we can obtain a material model having the realism and physical plausibility (Helmholtz symmetry, energy conservation) of the Cook-Torrance BRDF, and the simplicity and the evaluation speed of Phong-Blinn type BRDFs.

Figure 10.13: Copper, golden and silver skulls rendered with the proposed simple BRDF.

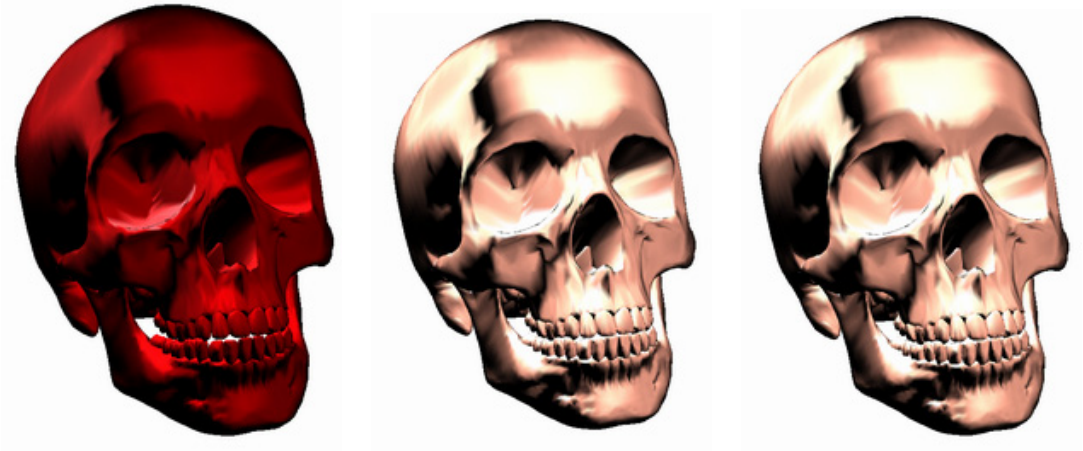


Figure 10.14: A copper skull rendered with the Schlick's formula (left), by the new model (middle), and by the exact Fresnel function.

Chapter 11

Spherical Billboards Tool

This tool encompasses an improved billboard rendering method, which renders particles as aligned quadrilaterals similarly to previous techniques, but takes into account the spherical geometry of the particles during fragment processing. The new method can eliminate billboard clipping and popping artifacts happening when previous methods render objects contained by the participating media or the camera flies into the participating media. The new method can be implemented by a simple fragment program and runs on high framerates.

11.1 Introduction

Participating media [13] are often represented by particle systems [102]. The particle system model assumes that scattering may happen only at discrete points of particle centers and a particle point represents its spherical neighborhood where the volume is locally homogeneous. Particle system rendering methods usually splat particles onto the screen, which substitute them with a semi-transparent, camera-aligned rectangles, called *billboards* [110]. Ignoring the extension along the third dimension simplifies rendering, but causes visual artifacts when the billboard rectangle intersects an object (figure 11.1).

A particle system is a discretization of a continuous volume, which allows us to replace the differentials of the volumetric rendering equation by finite differences. Denoting the length of the ray segment intersecting the sphere of particle j by Δs_j, and the *density*, *albedo* and *phase function* of this particle by τ_j, a_j, P_j, respectively, we obtain the following equation expressing *outgoing radiance* $L(j, \vec{\omega})$ of particle j at direction $\vec{\omega}$:

$$L(j, \vec{\omega}) = I(j, \vec{\omega}) \cdot (1 - \alpha_j) + \alpha_j \cdot C_j, \tag{11.1}$$

where $I(j, \vec{\omega})$ is the *incoming radiance*, $\alpha_j = 1 - e^{-\tau_j \Delta s_j}$ is the *opacity* that expresses the decrease of radiance caused by this particle due to *extinction*, and

$$C_j = a_j \cdot \int_{\Omega'} I(j, \vec{\omega}') \cdot P_j(\vec{\omega}', \vec{\omega}) \; d\omega'$$

is the contribution from *in-scattering*. The opacity also depends on the distance of the ray and the particle center, since rays that are close to the center travel longer inside the particle sphere and thus the attenuation is more significant. This dependence is usually represented by a *opacity billboard*, which is also used to display a particle as a semi-transparent rectangle perpendicular to the ray.

If we know the in-scattering term, the volume can efficiently be rendered from the camera using alpha blending. The in-scattering term is attenuated according to the total opacity of the particles that are between the camera and this particle. This requires the sorting of particles in the view direction before sending them to the frame buffer in back to front order. At a given

particle, the evolving image is decreased according to the opacity of the particle and increased by its in-scattering term (equation 11.1).

11.2 Billboard clipping and popping artifacts

The main problem with billboard type particle systems is that billboards are planes, thus they have no extension along one dimension.

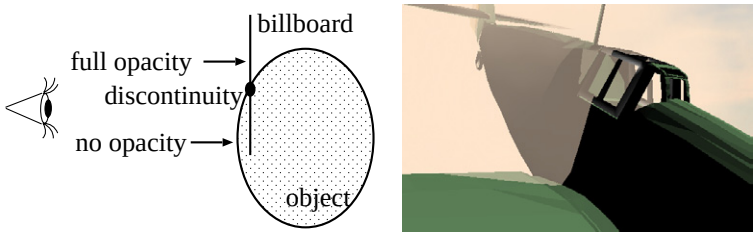


Figure 11.1: Billboard clipping artifact. When the billboard plane intersects the object, transparency becomes spatially discontinuous.

This can cause artifacts when billboards intersect objects making the intersection of the billboard plane and the object clearly noticeable (figure 11.1). The core of this problem is that a billboard fades those objects that are behind it according to its transparency as if the object were fully behind the sphere of the particle. However, those objects that are in front of the billboard plane are not faded at all, thus transparency changes abruptly at the object billboard intersection.

Figure 11.2: Billboard popping artifact. Where the billboard gets to the other side of the front clipping plane, the transparency is discontinuous in time (the figure shows two adjacent frames in an animation).

When the camera moves into the participating media, billboards also cause popping artifacts (figure 11.2). In this case, the billboard is either behind of or in front of the front clipping plane and the transition between the two stages is instantaneous. The former case corresponds to a fully visible, while the latter to a fully invisible particle, which results in an abrupt change during animation.

In this tool we propose a novel solution for including objects into the volume without billboard clipping artifacts, and also to eliminate billboard popping during animation.

11.3 Spherical billboards

Billboard clipping artifacts are solved by dealing with the spherical geometry of particles instead of assuming that a particle can be represented by a planar billboard. However, in order to keep the implementation simple and fast, we still send the particles through the rendering pipeline as quadrilateral primitives, and take into account the spherical shape only during fragment processing.

There are two reasonable alternatives for replacing a sphere by a quadrilateral during geometry processing. The quadrilateral may be either perpendicular to axis z as happens in classical billboard processing, or it may be a quadrilateral that is perpendicular to the view direction at the quadrilateral center in camera space. The first option increases the complexity of the fragment program, while the second requires the vertex shader to rotate billboard. The two techniques need slightly different approaches, which are described in the next two subsections.

11.3.1 Billboards perpendicular to axis z

The algorithm first renders all objects of the scene and saves the depth values in a texture. Note that these depth values are camera space z coordinates.

Then the particles are rendered as quads perpendicular to axis z, placed at the farthest point of the particle sphere from the camera to avoid unwanted front plane clipping. Particle quads are rasterized one by one in descending order of their distance from the camera, disabling depth test. Disabling depth test is needed to eliminate incorrect object-quad clipping.

When rendering a fragment of the particle, we compute the length of the ray segment that is inside the particle sphere in camera space. During this calculation we also take into account the depth of objects and the front clipping plane distance. The opacity is then computed using the effective depth of the particle sphere, assuming that the density is uniform inside a particle.

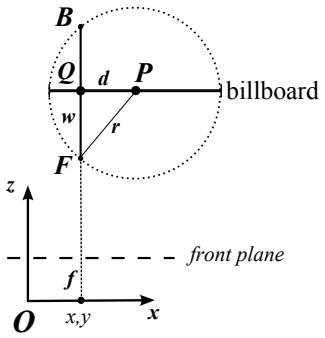


Figure 11.3: Computation of the length of the ray segment inside a particle sphere in camera space.

Let us use the notations of figure 11.3 where a particle of center $\vec{P} = (x_p, y_p, z_p)$ is rendered as a quad perpendicular to axis z, and a ray is cast through point $\vec{Q} = (x, y, z_q)$ of the quadrilateral. The radius of the particle sphere is r, the distance between the ray and the particle center is $d = \sqrt{(x - x_p)^2 + (y - y_p)^2}$. The closest and furthest points of the particle sphere on the ray from the camera are $\vec{F}$ and $\vec{B}$. The ray travels inside the particle in interval $[|\vec{F}|, |\vec{B}|]$, where $|\vec{F}| = z_p - \sqrt{r^2 - d^2}$ and $|\vec{B}| = z_p + \sqrt{r^2 - d^2}$.

In order to take into account the front clipping plane and the object depths, these distances must be altered. In order to eliminate popping artifacts we should ensure that $|\vec{F}|$ is not smaller than the front clipping plane distance f, thus the distance the ray travels in the particle before reaching the front plane is not included. Secondly we should also ensure that $|\vec{F}|$ is not greater

than Z_s, which is the stored scene depth at the given pixel, thus the distance traveled inside the objects is not considered.

With these altered distances we can obtain the real length of the ray segment, and also the corresponding opacity value:

$$\Delta s = max(f, |\vec{F}|) - min(Z_s, |\vec{B}|),$$

$$\alpha = 1 - e^{-\tau \Delta s}.$$

Implementation

In this case the vertex program is similar to that of classical billboard rendering. The billboard is placed at the distance between the camera and the particle and turned perpendicular to axis z.

The fragment program gets some of its inputs from the vertex shader, including the particle position in camera space (`In.P`), the shaded point in camera space (`In.Q`), the particle radius (`In.r`), the screen coordinates of the shaded point (`In.screenCoord`), and it also gets some uniform parameters such as the texture containing the scene depth values (`sceneDepth`), the density (`tau`) and the camera front clipping plane distance (`frontPlane`).

The fragment program executes the following operations:

```
float d = length(In.P.xy - In.Q.xy);
if(d > In.r) alpha = 0;              // sphere is not intersected
else {
    float w = sqrt(In.r*In.r - d*d);
    float Zs = tex2D(sceneDepth, In.screenCoord).r;
    Zf = max(frontPlane, Zp-w);
    Zb = min(Zs, zp+w);
    float ds = Zb - Zf;              // effective length inside the sphere
    alpha = 1 - exp(-tau * ds);      // opacity
}
```

This method uses rays parallel to axis z in camera space, which is only correct in case of orthographic projection. If perspective projection is used (which is common in interactive applications), errors can occur if angle $\vec{O}\vec{Q}\vec{F}$ is large, i.e. on the edges of the screen. However this is usually acceptable, as accuracy is not required in case of such a blurred foggy phenomena, especially in realtime applications. However, if more precise results are needed we can use the method discussed in the next section.

11.3.2 Billboards perpendicular to viewray

The goal of this algorithm is to find the real length of the ray segment inside the particle sphere. The algorithm should be still fast enough but with less error than the method described above. This method needs a different map of the scene, which stores the Euclidean distances of the objects rather than the depth values. The texture storing the distance values is called *distance map*.

The particles are rendered as quads placed at the farthest point of the particle sphere from the camera and rotated towards the camera by the vertex shader. Particle quads are rasterized one by one in descending order of their distance from the camera.

Let us use the notations of figure 11.4 where a particle of center $\vec{P} = (x_p, y_p, z_p)$ is rendered as a quad perpendicular to axis z, and a ray is cast through point $\vec{Q} = (x_q, y_q, z_q)$ of the quadrilateral. All coordinates are in camera space where the eye is in origin $\vec{O}$. The radius of the particle sphere is r. The closest point on the ray from the particle center is $\vec{C}$, which can be approximated by $\vec{Q}$ if the camera is not very close to the particle compared to its size. It means that we shall assume that angle $\vec{O}\vec{Q}\vec{P}$ is close to 90 degrees.

Distance d between the ray and the particle center, i.e. approximately between $\vec{P}$ and $\vec{Q}$ is

$$d \approx |\vec{P} - \vec{Q}|.$$

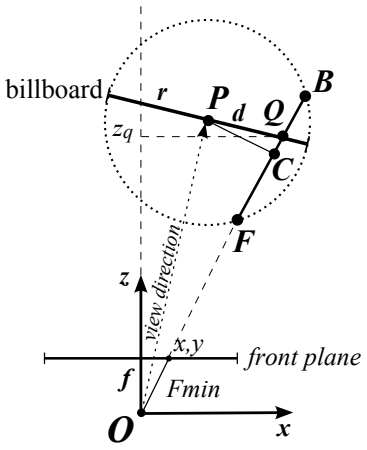


Figure 11.4: Computation of the length of the ray segment traveling inside a particle sphere in camera space.

The ray travels inside the particle approximately in distance interval $[|\vec{F}|, |\vec{B}|]$, where $|\vec{F}| = |\vec{Q}| - \sqrt{r^2 - d^2}$, and $|\vec{B}| = |\vec{Q}| + \sqrt{r^2 - d^2}$, if neither the front clipping plane or the scene objects cut the particle sphere. In order to take into account front clipping distance f, we should ensure that $|\vec{F}|$ is not smaller than

$$F_{\min} = \frac{f}{z_p} \cdot |\vec{Q}|.$$

For the ray passing through point $\vec{Q}$, distance D_s of the visible surface can be looked up from the distance map. To take into account the scene objects, we should ensure that $|\vec{B}|$ is not greater than D_s. With the use of these distances we can compute the real length of the ray segment inside the particle sphere, and the corresponding opacity:

$$\Delta s = min(D_s, |\vec{B}|) - max(F_{\min}, |\vec{F}|),$$
$$\alpha = 1 - e^{-\tau \Delta s}.$$

Implementation

A particle is rendered as a billboard rectangle. The vertex program gets the particle position in world space (`pPos`), the particle radius (`r`) and the texture coordinates with values $[-1, -1, 0]$, $[-1, 1, 0]$, $[1, 1, 0]$, and $[1, -1, 0]$ at the rectangle vertices. One of its main task is to compute the shaded point's clipping space coordinates (`Out.hpos`).

The vertex shader program is:

```
float3 P = mul(pPos, modelview).xyz;
float3 right = normalize(float3(P.y, 0, -P.x));
float3 up = normalize(cross(P, right));
float3 Q = P + texcoord.x * right + texcoord.y * up;
OUT.screenCoord = mul((Q.xy / Q.w + 1) * 0.5, modelviewproj);
OUT.hpos = mul(Q, modelviewproj).xyz;
OUT.P    = P;
OUT.Q    = Q;
OUT.r    = r;
```

The fragment program uses parameters computed by the vertex program, including the particle position in camera space (`In.P`), the shaded point in camera space (`In.Q`) and the screen coordinates of the shaded point (`In.screenCoord`). It also needs the texture containing the scene

distance values (`distMap`), the density (`tau`) and the front plane distance (`frontPlane`). The pixel shader is:

```
float  Ql = length(IN.Q);
float  d = length(P - Q);
if(d > In.r) alpha = 0;
else {
    float w = sqrt(In.r * In.r - d * d);
    float Fmin = frontPlane * Ql / IN.P.z;
    float Ds = tex2d(distMap, In.screenCoord);
    float Df = max(Fmin, Ql - w);
    float Db = min(Ds, Ql + w);
    float ds = Db-Df;
    alpha = 1 - exp(-tau * ds);
}
```

11.4 Results

The presented algorithm has been implemented in OpenGL/Cg environment on an NV6800GT graphics card. The smoke on figure 11.5 consists of 400 large particles. The screenshots were made with the method described first, with billboards perpendicular to the camera's z axis. The frame rate strongly depends on the count of overridden pixels, it varies from 50-90 FPS. The classic method (with artifacts) also has this dependency and runs at 80-120 FPS (figure 11.6). The algorithm has also been implemented in Renderman, results are shown in figure 11.7.

Figure 11.5: Smoke rendered at around 70 FPS with the new method.

11.5 Conclusions

This tool proposed to consider particles as spheres rather than planar billboards during rendering while still rendering them as billboards, which eliminated billboard clipping and popping artifacts. Two methods were discussed for the two common way of billboard rendering, both of them offer high frame rates for displaying volume media in realtime applications, taking advantage of the GPU.

Figure 11.6: Smoke rendered at around 100 FPS with the classic method.

Figure 11.7: Pictures rendered with Pixar's Photorealistic Renderman.

Chapter 12

Cloudy Natural Phenomena Tool

This tool presents a real-time method to realistically render dynamic participating media under changing lighting conditions. In order to cope with performance requirements, the volume is built of instances of particle blocks. The simulation and rendering happen on two levels, on the block level and on the volume level. On the volume level blocks are replaced by depth impostors, which allows for very fast recalculation of the cloud illumination. Including depth information into block impostors our technique also eliminates billboard clipping artifacts when the participating medium contains objects. The proposed method can render swirling clouds and smoke on high frame rates, and can be used in real-time applications.

12.1 Introduction

Particle system rendering methods usually execute a pass for each light source to calculate shadows and lighting in a view independent way, and a final gathering pass to compute the image from the camera [45].

Using the same denoting as in chapter seven (equation 11.1) the radiance equation will look like:

$$L(j, \vec{\omega}) = I(j, \vec{\omega}) \cdot (1 - \alpha_j) + \alpha_j \cdot C_j, \tag{12.1}$$

The in-scattering term C_j requires the evaluation of a directional integral at each particle j. Real-time methods usually simplify this integral and consider only the directions of the light sources in these integrals, and thus allowing only attenuation and forward scattering [45]. The incoming radiance for all particles can be effectively evaluated executing a light pass for each light source. In a light pass, particles are sorted along the light direction, and their opacity billboards are rendered one by one in this order. Before rendering a particle, the color buffer is read back to obtain the light attenuation at this particle, then the opacity texture of the particle is combined with the image to prepare the light attenuation for the subsequent particle.

The incoming radiance of a particle due to a given light source is computed from the light source intensity and the opacity accumulated so far.

If we know the incoming radiance of particles, the volume can efficiently be rendered from the camera using alpha blending. The in-scattering term of a particle is obtained by multiplying the albedo and the phase function with the incoming radiance values due to the different light sources, which is attenuated according to the total opacity of the particles that are between the camera and this particle. This requires the sorting of particles in the view direction before sending them to the frame buffer in back to front order. At a given particle, the evolving image is decreased according to the opacity of the particle and increased by its in-scattering term (equation 12.1).

To create a realistic effect, and to model media with different scattering properties we used the Mie scattering model. This model requires a fairly complex phase function. To avoid long computations in the pixel shader we stored the function values in a pre-computed map.

Particle hierarchies are used to reduce the computational burden of rendering. On the lower level, particles are grouped into blocks, that are rendered once for the current viewing or lighting direction, then the role of the individual particles are taken by these blocks. To eliminate the read-backs of the color buffer, we compute the attenuation at discrete depth samples during light passes. We also propose a novel solution for including objects into the volume without billboard clipping artifacts, which uses *depth impostors*, i.e. billboards augmented with depth information. Unlike *nailboards* [111, 122], our depth impostors store both the front and back depths of a block.

12.2 The new method using particle hierarchies

To reduce the computational burden of rendering, particle hierarchies are formed, and the full system is built of smaller similar blocks. As most natural phenomena shows self-similarity, we can use this approximation in most cases.

A single *block* represents particles that are close to each other. Before rendering for a given direction, the image of the particles of a block is determined from this direction, and then we use these images instead of individual particles. The image is called *depth impostor*. A pixel of the depth impostor stores information that is needed about the particles which project onto this pixel, particularly their total opacity, their minimum (front) and maximum (back) depths. During rendering an impostor pixel acts as a "super-particle" that concentrates all those particles of the block, which are projected onto it. The total opacity is used in the radiance transfer, while the depths are taken into account to eliminate artifacts caused by objects included in the volume.

Replacing particles by blocks, the computation burden can be reduced significantly. If we want a system with N particles, we can build a block of b particles and instance it N/b times. The hierarchical approach needs only $b + N/b$ calculations during simulation and color computation, in contrast to N calculations of the non-hierarchical method.

12.2.1 Generating a depth impostor

To generate a depth impostor representing a block for a particular direction (either light or camera), the particles of the block are rendered and the total opacity, and front and back depths are computed for each pixel on the GPU. These depth impostors are first generated for particle spheres and then for volume blocks. The opacity texture of a particle is pre-defined, and the depth textures of a particle sphere can be created analytically evaluating the depths of a sphere in the preprocessing phase.

The total opacity of a block could be determined using alpha blending of the opacity textures of the particles. The depth values of the block, however, require a different operation. A simple approach would generate the front and back depth textures of a block by rendering the front and back textures of the particles, overwriting the fragment depth value in the fragment shader, and letting the z-buffer to find the minimum and the maximum in two different passes.

However, this simple approach needs three rendering passes for each particle block. Fortunately, it is also possible to execute the three different calculations in a single rendering pass if depth testing is replaced by alpha testing. With the GL_EXT_blend_minmax extension we can set a blend function which computes minimum or maximum. However, this blending is appropriate only for the depth values, but not for the total opacity, which can be solved by the GL_EXT_blend_equation_separate extension, allowing a different blending type for the alpha channel. The layers of a depth impostor are shown in figure 12.1.

12.2.2 Using depth impostors during light passes

Rendering participating media consists of a separate light pass for each light source determining the approximation of the in-scattering term caused by this particular light source, and a final gathering step. Handling light volume interaction on the particle level would be too computation intensive since the light pass requires the incremental rendering of all particles and the read back

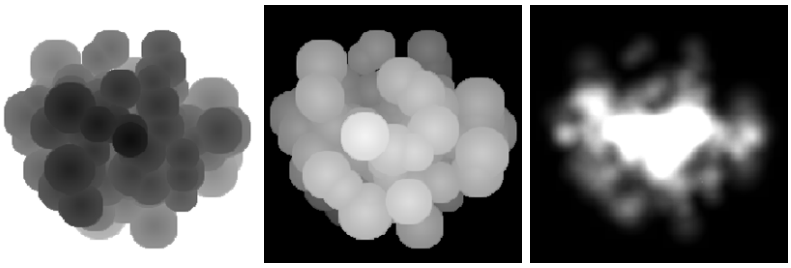


Figure 12.1: Depth impostor layers of a block: front depth, back depth, and accumulated opacity

of the actual result for each of them. To speed up the process, we render particle blocks one by one, and separate light–volume interaction calculation from the particles.

During a light pass we classify particle blocks into groups according to their distances from the light source, and store the evolving image in textures at given sample distances. These textures are called *slices*. The first texture will display the accumulated opacity of the first group of particle blocks, the second will show the opacity of the first and second groups of particle blocks and so on. The required number of depth samples depends on the particle count and the cloud shape. For a roughly spherical shape and relatively few particles (where overlapping is not dominant), even four depths can be enough. Figure 12.2 shows this technique with five depth slices. Four slices can be computed simultaneously if we store the slices in the color channels of one RGBA texture. For a given particle, the vertex shader will decide in which slice (or color channel) this particle should be rendered. The vertex shader sets the color channels corresponding to other slices to zero, and the pixel is updated with alpha blending.

12.2.3 Using depth impostors during final gathering

During final rendering we obtain the depth impostors of the blocks for the viewing direction, sort them and render them one after the other in back to front order. The in-scattering term is obtained from the sampled textures of the slices that enclose the pixel of the block (figure 12.2).

The accumulated opacity of the slices can be used to determine the radiance at the pixels of these slices and finally the reflected radiance of a particle between the slices. Knowing the position of the particles we can decide which two slices enclose it. By linear interpolation between the values read from these two textures we can approximate the attenuation of the light source color. Harris used a similar technique in [46], where he stored these slices in a 3D texture called oriented light volume. In order to obtain a better multiple scattering approximation, the radiance of those pixels of both enclosing slices are taken into account, for which the phase function is not negligible.

12.2.4 Objects in clouds

As we render the particles as billboards the same clipping and popping artifacts occur as described in chapter seven, and the solution to this problem is similar to the one depicted there. The main difference is that while working with spherical particles we can compute the distance light travels in the medium analytically. Contrarily blocks of particles can not be treated as spheres. This problem is solved using the extension of the particle block, i.e. the interval of the block in the depth direction, which is stored in impostor texels.

The algorithm works like in chapter seven. First we render all objects of the scene and save the depth buffer in a texture. Then the particle blocks are rendered one by one, enabling depth test but disabling depth write. When rendering a particle block, we compute the interval the ray travels inside the block adding the depth value of the block center to the front and back depth values of the depth impostor. This interval is compared with the value storing the depth of the

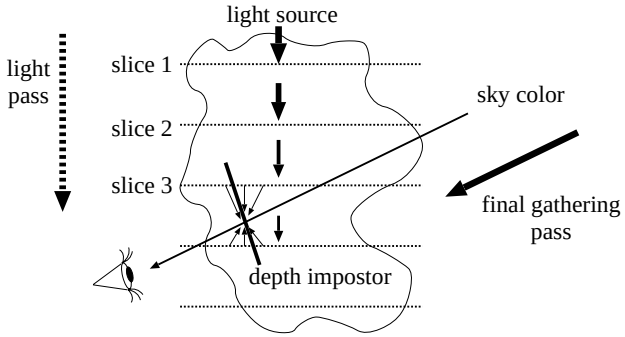


Figure 12.2: Final gathering for a block

visible object.When the interval of the block encloses the depth of the object, the opacity of the particle block is scaled according to the relative distance between the front depth of the particle block and the object, and the depth interval of the particle block. This scaling corresponds to the assumption that the density is uniform inside a block. The results are shown in figure 12.3.

Figure 12.3: Volume rendered with depth impostors eliminating billboard clipping artifacts

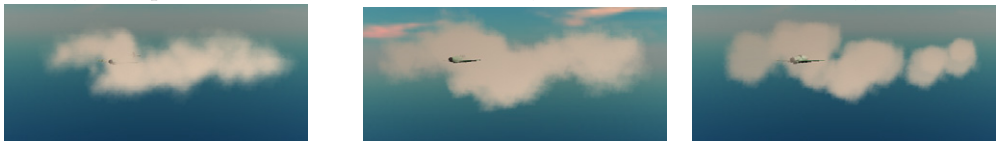
12.3 Results

The presented algorithm has been implemented in OpenGL/Cg environment on an NV6800GT graphics card. The animated cloud of figure 12.4 consists of 8000 particles grouped to different number of blocks, and is illuminated by a directional light. The rendering speed increases as we put more particles in a single block until 40 blocks. For higher number of particles per block, rendering gets slower, because of the block computation overhead.

12.4 Conclusions

The main feature of this tool is to build particle clouds of blocks. A block itself represents many particles, and is defined by a depth impostor. We also applied depth sampling to compute self-shadowing and multiple forward scattering quickly. As a combined effect of these improvements, the presented method renders realistically shaded dynamic smoke or cloud formations under changing lighting conditions at high frame rates, taking advantage of the GPU. On the other hand, the inclusion of front and depth information into the depth impostors eliminated billboard artifacts when the volume contains 3D objects.

no block, 8000 particles, 8 FPS | 1600 blocks, 20 FPS | 800 blocks, 42 FPS



400 blocks, 60 FPS | 40 blocks, 200 FPS | 20 blocks, 180 FPS

Figure 12.4: Images of animated volumes of 8000 particles organized in different number of blocks

Figure 12.5: Swirling cloud rendered at 180 FPS

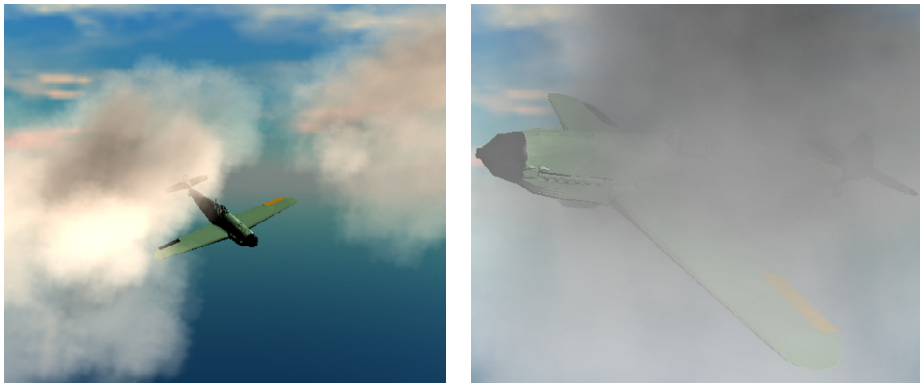


Figure 12.6: Swirling cloud crossed by a plane rendered at 180 FPS

Chapter 13

Participating Media Illumination Networks Tool

This tool constitutes a real-time method to compute multiple scattering in non-homogeneous participating media having general phase functions. The volume represented by a particle system is supposed to be static, but the lights and the camera may move. Lights can be arbitrarily close to the volume and can even be inside. Real-time performance is achieved by reusing light scattering paths that are generated with global line bundles traced in sample directions in a preprocessing phase. For each particle we obtain those other particles which can be seen in one of the sample directions, and their radiances toward the given particle. This information is stored in an illumination network that allows the fast iteration of the volumetric rendering equation. The illumination network can be stored in two-dimensional arrays indexed by the particles and the directions, respectively. Interpreting these two-dimensional arrays as texture maps, the iteration of the scattering steps can be efficiently executed by the graphics hardware, and the illumination can spread over the media in real-time.

13.1 Introduction

Physically plausible rendering of participating media simulates multiple scattering effects [82, 92, 70]. Multiple scattering algorithms can be considered as particular ways to generate light paths connecting the light sources to the eye. The computation time can be reduced if the steps of paths are reused and are not generated again when they are needed.

Path reuse is a common technique of advanced global illumination methods developed for surface and volume rendering. Steps of the path can be reused in bi-directional and even in classical *path tracing* [69, 9]. *Metropolis light transport* reuses path parts that are left unchanged by the current perturbation [94]. *Instant radiosity* [65] and *photon mapping* [58] reuse a shooting path for all gathering paths. *Precomputed radiance transfer* [118] and finite-element based iteration methods [88, 123, 27] can also be interpreted as path reuse techniques. Whenever the radiance of a patch or a voxel is shot or gathered, all paths ending here are simultaneously extended. Path reuse is more explicit in *global line* global illumination methods that exchange radiance along globally sampled lines [15, 107, 98]. Global lines are worth organizing in parallel or perspective bundles since this allows the exploitation of the rendering hardware and the application of incremental scan conversion algorithms [15, 88, 123, 27]. Global lines form an *illumination network*, which can replace the geometry of the surfaces or the density of the volume in illumination computations [50, 116].

In this tool chapter we extend the global line bundle illumination network concept to participating media represented by a particle system, and obtain the particle radiance with iteration. Since the iteration works on the illumination network, it does not require ray casting or queries of the particle system. Encoding the illumination networks by textures, the GPU can calculate a

scattering step on all particles and in all sampled directions rendering a single textured quadrilateral.

13.2 Multiple scattering in volumes

Let us consider how the light goes through participating media. The change of radiance L on path of length ds and of direction $\vec{\omega}$ depends on different phenomena:

- *Absorption* and *outscattering*: the light is absorbed or scattered out from its path when photons collide with the particles. If the probability of collision in a unit distance is τ, then the radiance changes by $-\tau \cdot L \cdot ds$ due to the collisions. After collision a particle may be either absorbed or reflected with the probability of *albedo* a.

- *Emission*: the radiance may be increased by the photons emitted by the participating media (e.g. fire). This increase is $L^e \cdot ds$ where L^e is the emission density.

- *Inscattering*: photons originally flying in a different direction may be scattered into the considered direction. The expected number of scattered photons from differential solid angle $d\omega'$ equals to the product of the number of incoming photons and the probability that the photon is scattered from $d\omega'$ to $\vec{\omega}$. The scattering probability is the product of the collision probability (τ), the probability of not absorbing the photon (a), and the probability density of the reflection direction, called *phase function* P. We use the Henyey-Greenstein phase function [51, 26]:

  $$P(\vec{\omega}', \vec{\omega}) = \frac{1}{4\pi} \cdot \frac{3(1-g^2) \cdot (1 + (\vec{\omega}' \cdot \vec{\omega})^2)}{2(2+g^2) \cdot (1 + g^2 - 2g(\vec{\omega}' \cdot \vec{\omega}))^{3/2}},$$

  where $g \in (-1, 1)$ is a material property describing how strongly the material scatters forward or backward.

  Taking into account all incoming directions Ω', we obtain the following radiance increase due to inscattering:

  $$ds \cdot \tau(s) \cdot a(s) \cdot \left(\int_{\Omega'} L(s, \vec{\omega}') \cdot P(\vec{\omega}', \vec{\omega}) \; d\omega' \right).$$

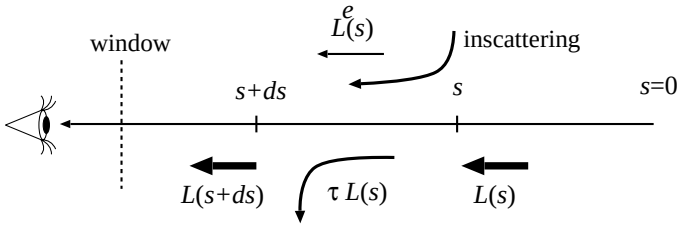


Figure 13.1: Modification of the radiance of a ray in participating media.

Adding the discussed changes, we obtain the following *volumetric rendering equation* for radiance L of the ray at $s + ds$:

$$L(s+ds, \vec{\omega}) = (1 - \tau(s) \cdot ds) \cdot L(s, \vec{\omega}) + ds \cdot L^e(s, \vec{\omega}) + ds \cdot \tau(s) \cdot a(s) \cdot \int_{\Omega'} L(s, \vec{\omega}') \cdot P(\vec{\omega}', \vec{\omega}) \; d\omega'. \quad (13.1)$$

13.2.1 Particle system model

The particle system model of the volume corresponds to a discretization, when we assume that scattering can happen only at N discrete points called particles. We assume that particles are sampled randomly, preferably from a distribution proportional to collision density τ, and we do not require them to be placed at grid points [38]. As demonstrated in [47] such particle systems can generate acceptable clouds with a few hundred particles. Let us assume that particle p represents its spherical neighborhood of diameter Δs_p, and introduce its *opacity* as $\alpha_p = 1 - e^{-\tau_p \cdot \Delta s_p}$, its *emission* as $E_p = L^e \cdot \Delta s_p$, its *incoming radiance* by I_p, and ist *outgoing radiance* by L_p. The *discretized volumetric rendering equation* at particle p is then:

$$L_p(\vec{\omega}) = (1 - \alpha_p) \cdot I_p(\vec{\omega}) + E_p(\vec{\omega}) + \alpha_p \cdot a_p \cdot \int_{\Omega'} I_p(\vec{\omega}') \cdot P_p(\vec{\omega}', \vec{\omega}) \; d\omega'.$$

In homogeneous media, albedo a and phase function P are the same for all particles. In non-homogeneous media, these parameters are particle attributes [104].

13.3 The proposed solution method

We solve the discretized volumetric rendering equation by iteration. The volume is represented by a set of randomly sampled particle positions. Suppose that we have an estimate of particle radiance values (and consequently, of incoming radiance values) at iteration step $n-1$. The new particle radiance in iteration step n is obtained by substituting these values to the right side of the discretized volumetric rendering equation:

$$L_p^n(\vec{\omega}) = (1 - \alpha_p) \cdot I_p^{n-1}(\vec{\omega}) + E_p(\vec{\omega}) +$$

$$\alpha_p \cdot a_p \cdot \int_{\Omega'} I_p^{n-1}(\vec{\omega}') \cdot P_p(\vec{\omega}', \vec{\omega}) \; d\omega'. \tag{13.2}$$

This iteration is convergent if the opacity is in $[0, 1]$ and the albedo is positive and less than 1, which is always the case for physically plausible materials.

In order to calculate the directional integral representing the inscattering term of equation 13.2, we suppose that D random directions $\vec{\omega}_1, \ldots, \vec{\omega}_D$ are obtained from uniform distribution of density $1/(4\pi)$, and the integral is estimated by Monte Carlo quadrature:

$$\int_{\Omega'} I_p(\vec{\omega}') \cdot P_p(\vec{\omega}', \vec{\omega}) \; d\omega' \approx \frac{1}{D} \cdot \sum_{d=1}^{D} I_p(\vec{\omega}'_d) \cdot P_p(\vec{\omega}'_d, \vec{\omega}) \cdot 4\pi.$$

Note that replacing the original integral by its approximating quadrature introduces some error in each iteration step, which accumulates during the iteration. This error can be controlled by setting D according to the albedo of the participating media since if the error of a single step is ϵ and the albedo is a, then the error is bound by $\epsilon/(1-a)$.

13.3.1 Building the illumination network

If we use the same set of sample directions for all particles, then the incoming radiance and therefore the outgoing radiance are needed only at these directions during iteration. For a single particle p, we need D incoming radiance values $I_p(\vec{\omega}_d)$ in $\vec{\omega}_1, \ldots, \vec{\omega}_D$, and the reflected radiance needs to be computed exactly in these directions. In order to update the radiance of a particle, we should know the indices of the particles visible in sample directions, and also the distances of these particles to compute the opacity. This information can be stored in two-dimensional arrays $\mathbf{I}$ and $\mathbf{V}$ of size $N \times D$, indexed by particles and directions respectively (figure 13.2). Array $\mathbf{I}$ is called the *illumination network* and stores the incoming radiance values of the particles on the

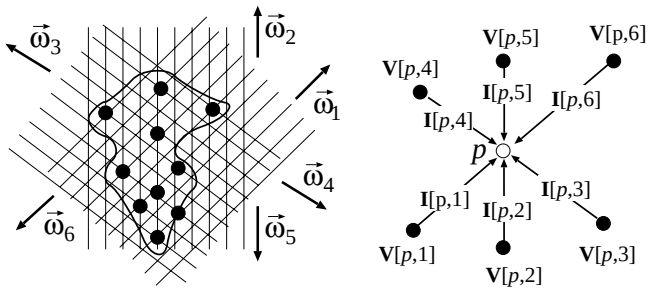


Figure 13.2: Illumination and visibility networks

wavelengths of red, green, and blue. Array **V** is the *visibility network* and stores index of visible particle *vp* and opacity α for each particle and incoming direction, that is, it identifies from where the given particle can receive illumination (figure 13.3).

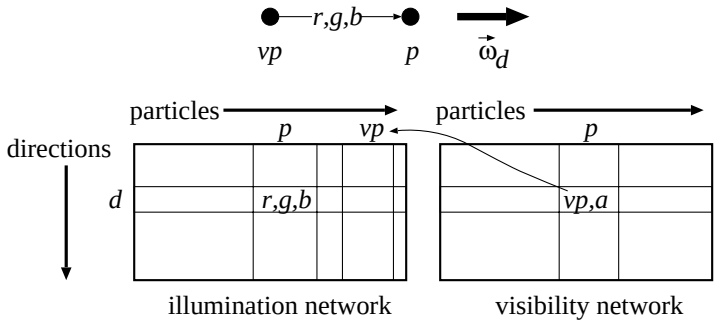


Figure 13.3: Storing the networks in arrays

In order to handle emissions and the direct illumination of light sources, we use a third array **E** that stores the sum of the emission and the reflection of the direct illumination for each particle and discrete direction. This array can be initialized by rendering the volume from the point of view of the light source and identifying those particles that are directly visible. At a particular particle, the discrete direction closest to the illumination direction is found, and the reflection of the light source is computed from the incoming discrete direction for each outgoing discrete direction.

Visibility network **V** expressing the visibility between particles is constructed during a preprocessing phase (figure 13.4). The bounding sphere of the volume is constructed and then D uniformly distributed points are sampled on its surface. Each point on the sphere defines a direction aiming at the center of the sphere, and also a plane perpendicular to the direction. A square window is set on this plane to include the projection of the volume, and the window is discretized to $M \times M$ pixels.

When a particular direction is processed, particles are orthographically projected onto the window, and rendered using a standard z-buffer algorithm, having set the color of particle p equal to its index p (figure 13.4). The contents of the image and depth buffers are read back to the CPU memory, and the indices and depths of the visible particles are stored together with the pixel coordinates. The particles that were visible in the preceding rendering step are ignored in the subsequent rendering steps. Repeating the rendering for the remaining particles and reading back the image and depth buffers again, we can obtain the indices of those particles which were occluded in the previous rendering step. Pairing these indices to those previously obtained ones which have the same pixel coordinates, we can get the pairs of particles that occlude each other

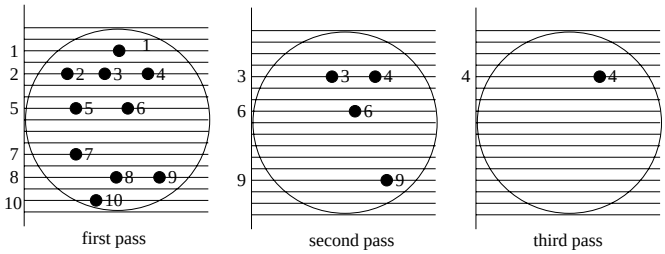


Figure 13.4: Constructing the visibility network

in the given direction. On the other hand, the difference of the depth values is the distance of the particles, from which the opacity can be computed. Repeating the same step until the image is empty, we can build lists of particles that are projected onto the same pixel. Subsequent pairs of these lists define a single row of array $\mathbf{V}$ corresponding to this direction (figure 13.3). Executing this algorithm for all predefined directions, the complete array can be filled up.

Note that multiple z-buffer steps carry out a *depth-peeling* procedure. Since this happens in the preprocessing step, its performance is not critical. However, if we intend to modify the illumination network during rendering in order to cope with animated volumes, then the performance should be improved. Fortunately, the depth peeling process can also be realized on the GPU as suggested by [36, 43].

13.3.2 Setting the parameters of the illumination network

The illumination network depends on radius R of the bounding sphere, number of discrete directions D, and on resolution of the windows $M \times M$. These parameters are not independent, but must be set appropriately taking into account the density of the medium as well. Since point rendering is used during the projection onto the window of size $2R \times 2R$ and of resolution $M \times M$, the projected area of a particle is implicitly set to $A = 4R^2/M^2$. The solid angle in which a particle at point $\vec{x}$ is seen from another particle at $\vec{y}$ is $\Delta\omega = A/|\vec{x} - \vec{y}|^2$. Approximating the maximum distance by the expected free run length $1/\tau$, we get $\Delta\omega \geq A\tau^2$.

If we do not want to miss particle interactions due to the insufficient number of sample directions, solid angle $\Delta\omega$ should not be smaller than the solid angle assigned to a single directional sample, which is $4\pi/D$. Substituting the implicit projected area of a particle, we obtain that the number of sample directions and the resolution of the window should meet $D \cdot (R\tau)^2/\pi \geq M^2$. The number of sample directions is typically 128, factor $R\tau$ is the expected number of collisions while the light travels through half of the volume, which usually ranges in 10–100, thus maximum resolution M is in 60–600, i.e. it is usually quite small.

13.3.3 Iterating the illumination network

The solution of the global illumination problem requires the iteration of the illumination network. A single step of the iteration evaluates the following formula for each particle $p = 1, \ldots, N$ and for each incoming direction $i = 1, \ldots, D$:

$$\mathbf{I}[p,i] = (1 - \alpha_{\mathbf{V}[p,i]}) \cdot \mathbf{I}[\mathbf{V}[p,i],i] + \mathbf{E}[\mathbf{V}[p,i],i] + \frac{4\pi \cdot \alpha_{\mathbf{V}[p,i]} \cdot a_{\mathbf{V}[p,i]}}{D} \cdot \sum_{d=1}^{D} \mathbf{I}[\mathbf{V}[p,i],d] \cdot P_{\mathbf{V}[p,i]}(\vec{\omega}'_d, \vec{\omega}_i).$$

Interpreting the two-dimensional arrays of the emission, visibility and illumination maps as textures, the graphics hardware can also be exploited to update the illumination network. The first texture is visibility network $\mathbf{V}$ storing the visible particle in red and the opacity in green channels, the second stores emission array $\mathbf{E}$ in the red, green, and blue channels, and the third texture is the illumination map, which also has red, green and blue channels. Note that in practical

cases number of particles N is about a thousand, while number of sample direction D is typically 128, and radiance values are half precision floating point numbers, thus the total size of these textures is quite small ($1024 \times 128 \times 8 \times 2$ bytes $= 2$ Mbyte).

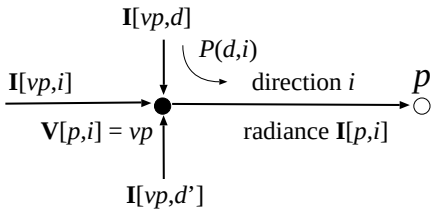


Figure 13.5: Notations in the pixel shader code

In the GPU implementation a single iteration step is the rendering of a viewport sized, textured rectangle, having set the viewpoint resolution to $N \times D$ and the render target to the texture map representing the updated illumination network. A pixel corresponds to a single particle and single direction, which are also identified by input variable `texcoord`. The pixel shader obtains the visibility network from texture `Vmap`, the emission array from texture `Emap`, and the illumination map from texture `Imap`. Function `P` is responsible for the phase function evaluation, which is implemented by a texture lookup of prepared values allowing other phase functions to be also easily incorporated [104]. In this simple implementation we assume that opacity `alpha` is precomputed and stored in the visibility texture, but albedo `alb` are constant for all particles. Should it not be the case, the albedo could also be looked up in a texture.

When no other particle is seen in the input direction, then the incoming illumination is taken from the sky radiance (`sky`). In this way not only a single sky color, but sky illumination textures can also be used [97, 104].

For particle `p` and direction `i`, the pixel shader finds opacity `alpha` and visible particle `vp` in direction `i`, takes its emission or direct illumination `Evp`, and computes and radiance `Ip` as the sum of the direct illumination and the reflected radiance values for its input directions `d = 1...D` (figure 13.5):

```
float  p  = texcoord.x; // particle
float  i = texcoord.y;  // input direction
float  vp = tex2d(Vmap, float2(p,i)).r;
if (vp >= 0) { // another particle is seen
   float alpha = tex2d(Vmap, float2(p,i)).g;
   float3 Evp = tex2d(Emap, float2(vp,i)).rgb;
   float3 Ivp = tex2d(Imap, float2(vp,i));
   float3 Ip = (1 - alpha) * Ivp + Evp;
   for(int d = 0; d < 1; d += 1.0/D) {
      Ivp = tex2d(Imap, float2(vp, d));
      float3 BRDF = alb * alpha * P(d,i);
      Ip +=  BRDF * Ivp * 4 * PI / D;
   }
   return Ip;
} else return sky; // no particle is seen
```

The illumination network provides a view independent radiance representation. When the final image is needed, we can use a traditional participating media rendering method, which sorts the particles according to their distance from the camera, splats them, and adds their contributions with alpha blending.

When the outgoing reflected radiance of a particle is needed, we compute the reflection from the sampled incoming directions to the viewing direction. Finally the sum of particle emission and direct illumination of the external lights is interpolated from the sample directions, and is added to the reflected radiance.

Figure 13.6: Cloud illuminated by two dynamic directional light sources (the first is left-up, the second is bottom-right) and sky illumination, and rendered by an animated camera at 26 FPS.

20 iterations 40 iterations 60 iterations 80 iterations

Figure 13.7: A cloud illuminated by two directional lights rendered with different iteration steps

13.4 Results

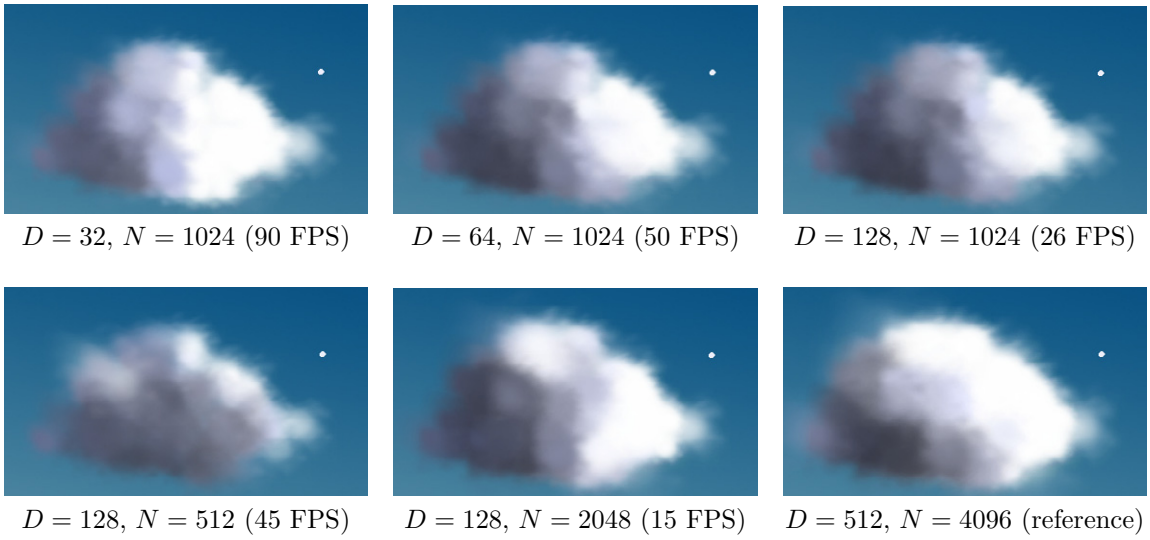
The proposed algorithm has been implemented in OpenGL/Cg environment and run on an NV6800GT graphics card. We compute one iteration in each frame, and when the light sources move, we take the solution of the previous light position as the initial value of the iteration, which results in fast convergence.

The results are shown in figures 13.6, 13.7, and 13.8. Where it is not explicitly stated, the cloud model consists of 1024 particles, and 128 discrete directions are sampled. With these settings the typical rendering speed is about 26 frames per second, and is almost independent of the number of light sources and of the existence of sky illumination. The albedo is 0.9, and the expected number of photon–particle collisions ($2R\tau$) is 20, and material parameter g is 0. Figures 13.6 shows how two external light sources illuminate the cloud. In figure 13.7 we can follow the evolution of the image of the same cloud after different iteration steps, where we can observe the speed of convergence. Figure 13.8 describes how the number of sample directions and the number of particles affect the image quality and the rendering speed. Note that using 128 directions and 512 particles we can obtain believable clouds at interactive frame rates, and the method is not too sensitive to the number of discrete directions. Changing the number of particles, however, has more significant impact on the image.

13.5 Conclusions

This chapter presented a global illumination algorithm for participating media, which works with a prepared visibility network, and maintains a similar illumination network. These networks are two-dimensional arrays that can be stored as textures and managed by the graphics hardware. The resulting algorithm can render multiple scattering effects at high frame rates.

The current implementation assumes that the volume is static. To cope with evolving volumes, such as clouds in wind, fire, or smoke, we plan to gradually update the illumination network, re-evaluating the visibility just in a single direction at a time, and thus amortizing the cost of the building the data structure during animation. We also plan to extend the method for hierarchical particle systems to handle complex phenomena.



$D = 32$, $N = 1024$ (90 FPS) $D = 64$, $N = 1024$ (50 FPS) $D = 128$, $N = 1024$ (26 FPS)

$D = 128$, $N = 512$ (45 FPS) $D = 128$, $N = 2048$ (15 FPS) $D = 512$, $N = 4096$ (reference)

Figure 13.8: Effect of number of discrete directions D and number of particles N on the image quality and on rendering speed. The light source is in the direction of the white point in the upper right part of the image.

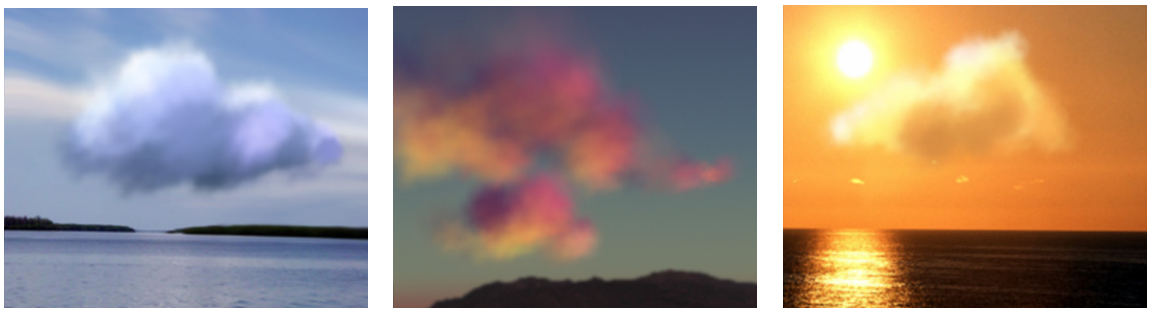


Figure 13.9: Globally illuminated clouds of 512 particles rendered with 128 directions at 45 FPS.

Chapter 14

Image Based Rendering Tool

This tool is an effective method to render complex trees on high frame rates while maintaining parallax effects. Based on the recognition that a planar impostor is accurate if the represented polygon is in its plane, we find an impostor for each of those groups of tree leaves that lie approximately in the same plane. The groups are built automatically by a clustering algorithm. Unlike billboards, these impostors are not rotated when the camera moves, thus the expected parallax effects are provided. On the other hand, clustering allows the replacement of a large number of leaves by a single semi-transparent quadrilateral, which improves rendering time considerably. Our impostors represent the tree from any direction well and provide accurate depth values, thus the method is also good for shadow computation.

14.1 Introduction

One of the major challenges in rendering of vegetation is that the large number of polygons needed to model a tree or a forest exceeds the limits posed by the current rendering hardware [33]. Rendering methods should therefore apply simplifications. To classify these simplification approaches, we can define specific scales of simulation at which rendering should provide the required level of realism:

- *Insect scale*: A consistent, realistic depiction of individual branches and leaves is expected. The images of individual leaves should exhibit parallax effects when the viewer moves, including the perspective shortening of the leaf shape and its texture, and view dependent occlusions.
- *Human scale*: Scenes must look realistic through distances ranging from an arm's reach to some tens of meters. Consistency is desired but not required.
- *Vehicle scale*: Individual trees are almost never focused upon and consistency is not required. Viewing distance may exceed several hundred meters.

There are two general approaches for realistic tree rendering: geometry-, and image-based methods. As it takes roughly hundred thousand triangles to build a convincing model of a single tree, geometry-based approaches should apply some form of Level of Detail technique [75].

Image-based methods [101, 1] represent a trade-off of consistency and physical precision in favor of more photorealistic visuals (figure 14.1). *Billboard rendering* is analogous to using cardboard cutouts. In order to avoid the shortening of the visible image when the user looks at it from grazing angles, the billboard plane is always turned towards the camera. Although this simple trick solves the shortening problem, it is also responsible for the main drawback of the method. The tree always looks the same no matter from where we look at it. This missing parallax effect makes the replacement too easy to recognize. In order to handle this problem, the *view-dependent sprite* method pre-renders a finite set from a few views, and presents the one closest in alignment with

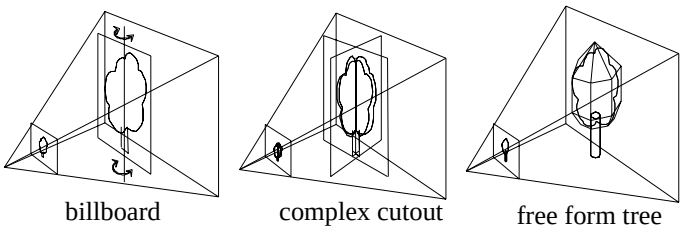


Figure 14.1: A tree representation with images

the actual viewing direction. A popping artifact is visible when there is an alignment change. The *complex cutout* approach uses texture transparency and blending to render more than one views at the same time onto properly aligned surfaces. In order to handle occlusions correctly, billboards can also be augmented with depth information [81, 122]. One of the most advanced methods actually implemented in commercial entertainment software is the basic *free-form textured tree model*, where the images are imposed on the approximated geometry.

The method used by this tool falls into the class of complex cutouts, but it prepares them so carefully that the results are satisfactory not only on vehicle scale, but also on human scale, and with some compromises on insect scale. The method is also good to generate shadows [86] and can be extended to a hierarchical approach [80].

14.2 The new method

In order to reduce the geometric complexity of a tree, we use impostors having transparent textures to represent leaf groups. Since impostors are not rotated when the camera changes, the expected parallax effects are provided [31]. The key of the method is then the generation of these impostors. We should first observe that an impostor is a perfectly accurate representation of a polygon (i.e. leaf) if the plane of impostor is identical to the plane of the polygon. On the other hand, even if the leaf is not on the impostor plane but is not far from that and is roughly parallel to the plane, then the impostor representation results in acceptable errors. Based on this recognition, we form clusters of the leaves in a way that all leaves belonging to a cluster lie approximately on the same impostor plane and replace the whole group by a single impostor. In order to control clustering, we shall introduce an error measure for a plane and a leaf, which includes both the distance of the leaf from the plane and the angle of the plane and leaf normals. The applied *K-means clustering algorithm* [77] starts with a predefined number of random planes, and iterates the following steps:

1. For each leaf we find that plane that minimizes the error. This step clusters the leaves into groups defined by the planes.

2. In each leaf cluster, the plane is recomputed in order to minimize the total error for the leaves of this cluster with respect to this plane.

Let us examine the definition of the error measure and these steps in details.

14.2.1 Error measure for a leaf and a plane

Those $\mathbf{p} = [x, y, z]^T$ points are on a plane that satisfy the following plane equation (T denotes matrix transpose):

$$D_{[\mathbf{n},d]}(\mathbf{p}) = \mathbf{p}^T \cdot \mathbf{n} + d = 0,$$

where $\mathbf{n} = [n_x, n_y, n_z]^T$ is the normal vector of the plane, and d is a distance parameter. We assume that $\mathbf{n}$ is a unit vector and is oriented such that d is non-negative. In this case $D_{[\mathbf{n},d]}(\mathbf{p})$ returns the signed distance of point ($\mathbf{p}$) from the plane.

From the point of view of clustering, leaf i is defined by its unit length average normal $\mathbf{n}_i$ and its center $\mathbf{p}_i$, that is by pair $(\mathbf{n}_i, \mathbf{p}_i)$.

Leaf i approximately lies in impostor plane $[\mathbf{n}, d]$ if $D^2_{[\mathbf{n},d]}(\mathbf{p}_i)$ is small, and its normal is approximately parallel with the normal of the plane, which is the case when $1 - (\mathbf{n}_i^T \cdot \mathbf{n})^2$ is also small. Thus an appropriate error measure for plane $[\mathbf{n}, d]$ and leaf $(\mathbf{n}_i, \mathbf{p}_i)$ is:

$$E([\mathbf{n}, d] \leftrightarrow (\mathbf{n}_i, \mathbf{p}_i)) = D^2_{[\mathbf{n},d]}(\mathbf{p}_i) + \alpha \cdot (1 - (\mathbf{n}_i^T \cdot \mathbf{n})^2), \tag{14.1}$$

where α expresses the relative importance of the orientation similarity with respect to the distance between the leaf center and the plane.

14.2.2 Finding a good impostor plane for a leaf cluster

To find an optimal plane for a leaf group, we have to compute plane parameters $[\mathbf{n}, d]$ in a way that they minimize total error $\sum_{i=1}^{N} E([\mathbf{n}, d] \leftrightarrow (\mathbf{n}_i, \mathbf{p}_i))$ with respect to the constraint that the plane normal is a unit vector, i.e. $\mathbf{n}^T \cdot \mathbf{n} = 1$. Using the Lagrange-multiplier method to satisfy the constraint, we have to compute the partial derivatives of

$$\sum_{i=1}^{N} E([\mathbf{n}, d] \leftrightarrow (\mathbf{n}_i, \mathbf{p}_i)) - \lambda \cdot (\mathbf{n}^T \cdot \mathbf{n} - 1) =$$

$$\sum_{i=1}^{N} (\mathbf{p}_i^T \cdot \mathbf{n} + d)^2 + \alpha \cdot \sum_{i=1}^{N} (1 - (\mathbf{n}_i^T \cdot \mathbf{n})^2) - \lambda \cdot (\mathbf{n}^T \cdot \mathbf{n} - 1) \tag{14.2}$$

according to $n_x, n_y, n_z, d, \lambda$, and have to make these derivatives equal to zero. Computing first the derivative according to d, we obtain:

$$d = -\mathbf{c}^T \cdot \mathbf{n}. \tag{14.3}$$

where

$$\mathbf{c} = \frac{1}{N} \cdot \sum_{i=1}^{N} \mathbf{p}_i$$

is the *centroid* of leaf points. Computing the derivatives according to n_x, n_y, n_z, and substituting formula 14.3 into the resulting equation, we get:

$$\mathbf{A} \cdot \mathbf{n} = \lambda \cdot \mathbf{n}, \tag{14.4}$$

where matrix $\mathbf{A}$ is

$$\mathbf{A} = \sum_{i=1}^{N} (\mathbf{p}_i - \mathbf{c}) \cdot (\mathbf{p}_i - \mathbf{c})^T - \alpha \cdot \sum_{i=1}^{N} \mathbf{n}_i \cdot \mathbf{n}_i^T. \tag{14.5}$$

Note that the extremal normal vector is the eigenvector of symmetric matrix $\mathbf{A}$. In order to guarantee that this extremum is a minimum, we have to select that eigenvector which corresponds to the smallest eigenvalue. We could calculate the eigenvalues and eigenvectors, which requires the solution of the third order characteristic equation. On the other hand, we can take advantage that if $\mathbf{B}$ is a symmetric 3×3 matrix, then iteration $\mathbf{B}^n \cdot \mathbf{x}_0$ converges to a vector corresponding to the largest eigenvalue from an arbitrary, non-zero vector $\mathbf{x}_0$ [135]. Thus if we select $\mathbf{B} = \mathbf{A}^{-1}$, then the iteration will result in the eigenvector of the smallest eigenvalue.

14.2.3 Indirect texturing

The proposed method uses a single texture for the impostor that includes many leaves. It means that the impostor texture should have high resolution in order to accurately represent the textures of individual leaves. The resolution of the impostor texture can be reduced without sampling

artifacts if the impostor stores only the position and orientation of the leaves, while the leaves rotated at different angles are put in a separate texture (figure 14.2). Let us identify the rectangles of the projections of the leaves on the impostor. Each rectangle is defined by its lower left coordinate, the orientation angle of the leaf inside this rectangle, and a global scale parameter (due to clustering the impostor planes are roughly parallel to the leaves thus leaf sizes are approximately kept constant by the projections). If we put the lower left coordinate of the enclosing rectangle

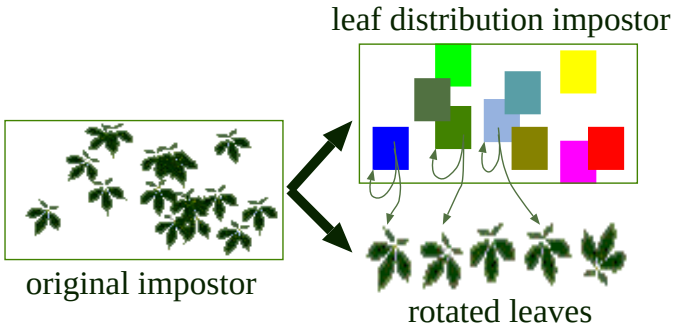


Figure 14.2: Indirect texturing. The impostor is replaced by a leaf distribution impostor and the rotated images of the leaves.

and the orientation angle into the impostor texture, and fill up the texels that are not covered by leaves by a constant value outside [0,1], then this texture has constant color rectangles, which can be down-sampled. In fact, this down-sampling could replace a rectangle of the leaf projection by a single texel. We call this low resolution texture as the *leaf distribution impostor*. During rendering, we address this leaf distribution impostor with texture coordinates u, v. If the first color coordinate of the looked up texel value is outside the [0,1] interval, then this texture coordinate does not address a leaf, and thus should be regarded as transparent. However, if the first coordinate is in [0,1], then first two color coordinates r, g are considered as the texture coordinates of the lower left corner of the leaf rectangle, and color coordinate b is regarded as the rotation angle of the leaf. The rotation angle selects the texture that represents this rotation, and u, v texture coordinates are translated by r, g and scaled by size s of the leaf rectangle, i.e. $u' = (u - r)/s, v' = (v - g)/s$ will be the texture coordinates of the single leaf texture selected by component b.

14.2.4 Smooth level of detail

The accuracy of the impostor representation can be controlled by the number clusters. If clusters are organized hierarchically, then the accuracy can be dynamically set by specifying the used level of the hierarchy. The level is selected according to the viewing distance, which results in a level of detail approach. To make the changes smooth, we can interpolate the plane parameters of the two enclosing levels.

14.3 Results

The proposed algorithm has been implemented in OpenGL/Cg environment, integrated into the Ogre3D game engine, and run on an NV6800GT graphics card. The tree is a European chestnut (Castanae Sativa) having 11291 leaves defined by 112910 faces and 338731 vertices. The trunk of the tree has 46174 faces. The leaves have been converted to 32 impostors using the proposed algorithm. When we did not apply indirect texturing, the impostor resolution was 512×512. Setting the screen resolution to 1024×768, leaf rendering is speeded up from 40 FPS to 278 FPS when the leaf polygons are replaced by the leaf cluster impostors. The comparison of the images of the original polygonal tree and the impostor tree is shown in figure 14.4. Note that this level

polygonal tree 107 impostors, 163 FPS 54 impostors, 256 FPS 29 impostors, 292 FPS

Figure 14.3: Visual and speed comparisons of the original polygonal tree and the tree rendered with different number of impostors.

of similarity is maintained for all directions, and the impostor tree also provides realistic parallax effects.

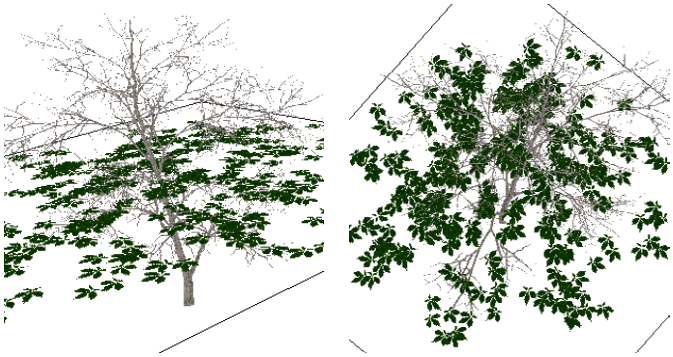


Figure 14.4: Two from the 32 leaf clusters representing the tree

Figure 14.5 shows a terrain of 4960 polygons with 2000 trees rendered on 30 FPS without instancing (the rendering is CPU limited). We used the proposed level of detail techniques to dynamically reduce the number of leaf cluster impostors per tree from 32 to 16 and even to 8.

14.4 Conclusions

This tool encompasses a tree rendering algorithm where clusters of leaves are represented by semi-transparent impostors. The automatic clustering algorithm finds an impostor in a way that the represented leaves lie approximately in its plane. This approach is equivalent to moving and rotating the leaves a little toward their common planes, thus this representation keeps much of the geometry information. Since the impostors are not rotated with the camera, the proposed representation can provide parallax effects and view dependent occlusions for any direction. We also discussed how leaf textures can be visualized without posing high resolution requirements for the impostors, and the possibility of including smooth level of detail techniques.

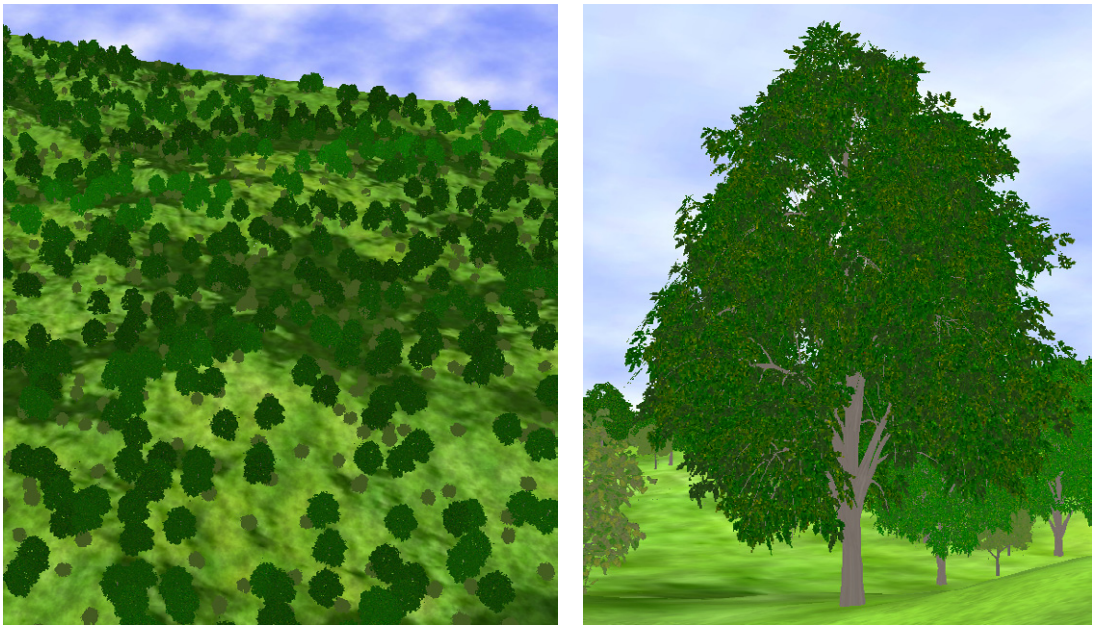


Figure 14.5: Two snapshots from a 30 FPS animation showing 2000 trees

Chapter 15

Rain Rendering Tool

Real-time rendering of virtual weather conditions has been investigated for a long time. Inserting fog or snow in a scene is rather straightforward. Rain is one of the most encountered natural phenomena, but its rendering often lacks realism.

In this tool we propose a realistic real-time rain rendering method using programmable graphics hardware. In order to simulate the refraction of the scene inside a raindrop, the scene is captured to a texture which is distorted according to optical properties of raindrops. This texture is mapped onto each raindrop. Our method also takes into account retinal persistence.

15.1 Introduction

Until a few years ago, speed had usually a higher priority than realism for real-time applications. Nowadays, with the tremendous possibilities of current graphics hardware, these two points become less and less antagonist. Real-time applications begin to have new goals: photo-realism, following physics laws, handling a large number of natural phenomena.

To achieve a high degree of realism in order to immerse the user in a visually convincing environment, developers introduce weather conditions in their applications. Fog rendering reduces the observable depth in the scene, speeding up the rendering process. It has already been introduced in computer graphics, even by a full hardware acceleration. Falling snow can be approximated as an opaque and diffuse material. Consequently, it can be realistically represented using simple particle systems. But falling rain still lacks realism, although it is one of the most encountered weather conditions in real scenes.

Rain rendering methods can be divided into two main categories. Most video-games use particle systems and static textures, leading to a lack of realism. Physically-based methods ([106], [60], [61]) intend to simulate low-motion raindrops on a surface. They generate accurate results, at a high computation cost. The technique we present here has the advantages of both kinds of methods, without their drawbacks.

This tool introduces a method for a realistic rain rendering at a high frame-rate, making use of programmable graphics hardware. In addition, this method is based on physical properties (geometrical, dynamic and optical) of raindrops. An image of the background scene is captured to a texture. This texture is mapped onto the raindrops according to optical laws by a fragment shader. We extend this method to take into account retinal persistence: quasi spherical raindrops appear like streaks. With this motion blur extension, we generate more visually realistic rain rendering.

After quickly presenting the physical (geometrical, dynamic and optical) properties of raindrops,we describe our method to render realistic raindrops in real-time, and explain how we handle retinal persistence. We also propose an extension to handle illumination of raindrops from light sources.

15.2 Physical properties of raindrops

15.2.1 Shape, size and dynamics

The widely spread idea according to which raindrops are tear-shaped, or streak-shaped, is inaccurate. This impression is caused, as we will see in section 15.3.3, by the phenomenon of retinal persistence. Many papers (referenced in [105]), prove that falling raindrops look more like ellipsoids. Small raindrops are almost spherical, and bigger raindrops get flattened at the bottom. This shape is the result of an equilibrium between antagonist forces. Surface tension tries to minimize the contact surface between air and raindrop, which results in a spherical shape. Aerodynamic pressure tries to stretch the drop horizontally, and gives it an ellipsoidal shape.

In this tool we use the equation proposed by Beard and Chuang ([7], [21]) to compute the shapes of raindrops for the desired radius.

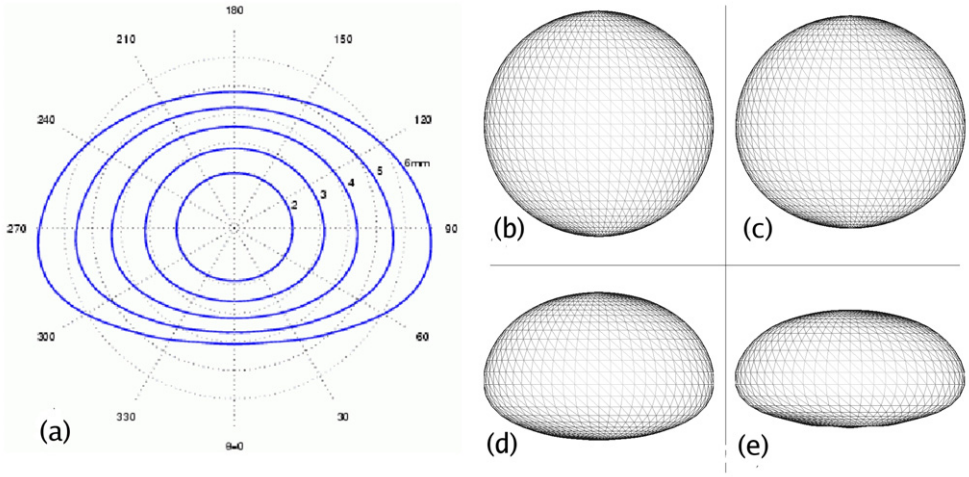


Figure 15.1: Shape of drops. (a) Compared shapes of raindrops of radii R = 1mm, 1.5mm, 2mm, 2.5mm and 3mm [105]. (b) Shape of a droplet of undistorted radius 0.5. (c) Shape of a droplet of undistorted radius 1.0. (d) Shape of a droplet of undistorted radius 3.0. (e) Shape of a droplet of undistorted radius 4.5.

Figure 15.1 shows typical raindrop shapes for common undistorted radii, computed from Beard and Chuang's equation.

Spherical drops		**Ellipsoidal drops**			
radius (mm)	speed (m/s)	radius (mm)	speed (m/s)	radius (mm)	speed (m/s)
0.1	0.72	0.5	4.0	2.5	9.2
0.2	1.62	1.0	6.59	3.0	9.23
0.3	2.47	1.5	8.1	3.5	9.23
0.4	3.27	2.0	8.91	4.0	9.23

Table 15.1: Speed of raindrops depending on their radii [105].

The falling speed of a raindrop depends on its radius. Values presented in table 15.1 are speeds of raindrops which have reached their terminal velocities, when gravity and friction forces compensate. This velocity is quickly reached, and is the speed at which the drops are seen at ground level.

15.2.2 Optical properties

Since we do not intend to render rainbows (diffraction of light) we do not need to take into account the wave character of light. It is physically correct to neglect the wave properties of light for drops much larger than the wavelength of light, which is the case here. We can instead focus on the properties defined by geometrical optics. In this approximation, light is considered as a set of monochromatic rays, which refract and reflect at interfaces between different propagation media.

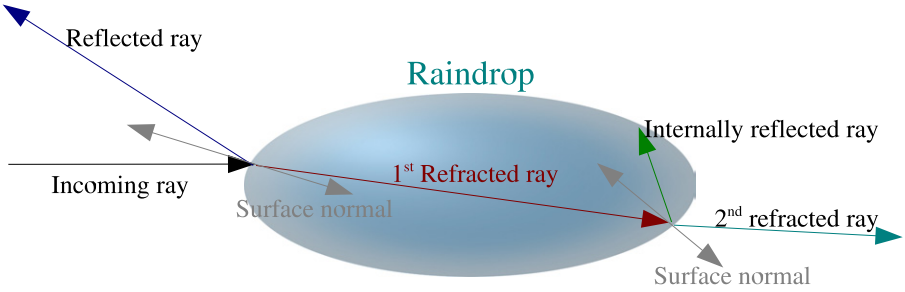


Figure 15.2: Reflection / refraction of a ray in a raindrop.

At an interface, the law of reflection describes the directions of the reflected ray, and Snell's law describes the direction of the refracted ray. Directions of reflected/refracted rays are illustrated in Figure 15.2.

For a specific ray, given its angle of incidence onto the interface and its polarization, the ratio between reflection and refraction is given by the Fresnel factor.

Figure 15.3: A photograph of a real drop refracting the background scene.

In Figure 15.3, an example of refraction can be observed on a photograph (taken with a 1/1000 s shutter speed). The white dots on the photographed water-drop are due to the camera flash.

15.3 Real-time raindrop rendering

15.3.1 Hypotheses

The Fresnel factor computation demonstrates that reflection has only a significant participation to the color of a surface at grazing angles. For a raindrop, this means that reflection is only visible on the border of the drop. In our application, since a raindrop appears rather small on the screen, and reflection is visible only in a small part of each raindrop, it is reasonable to neglect the reflection participation to the appearance of the raindrop, and focus on correct refraction.

Raindrops are rendered as billboards [76] (small quads always facing the camera); the outline shape of the raindrops is given by a mask pre-computed from Beard and Chuang's equation, for the desired raindrop radius. The computation of the mask is explained further in section 15.3.2. The appearance of each raindrop is computed inside a fragment shader.

15.3.2 Description of the method

The image perceived through a water-drop is a rotated and distorted wide angle image of the background scene, as illustrated in Figure 15.3. To simulate this effect, we use the render-to-texture facilities of graphics hardware to capture an image of the scene to a texture, and then map this texture onto each raindrop using vertex and fragment shaders taking into account the refraction inside the water-drops.

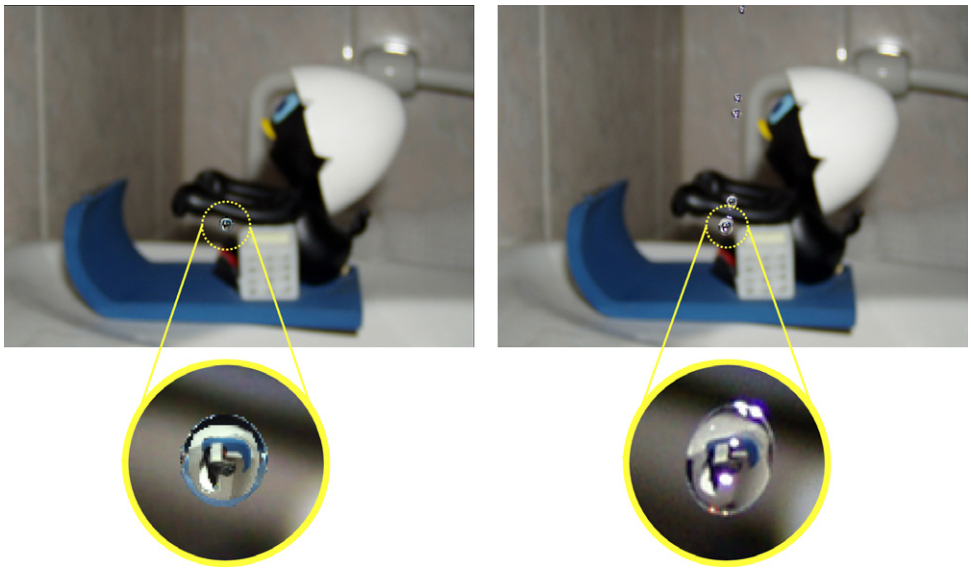


Figure 15.4: Left: An image simulated with our method. Right: A photograph of a real raindrop.

In Figure 15.4, we compare a water-drop simulated using our method (left) and an image of a real falling droplet (right). A photograph of the original scene was used as a background image for the simulated drop. The bottom images show a close view of the original and simulated drops. In the photograph, the real drop just left the tap, so its shape is not yet stabilized and perfectly spherical; this explains why it does not behave exactly as the simulated one.

Physical approximations

To obtain physically accurate results, we should perform a ray-tracing with all the objects in the scene, but this can hardly be done in real-time. The fact that we use only one texture for all the raindrops introduces a small approximation to physics laws, which implies a tremendous increase

in the rendering speed. As it is not generated at the exact location of the drops, the texture does not contain what the drop really "sees". This can result in seldom cases in undetected occlusions, or in additional distortion in the texture mapping. In rain simulation, drops are very small and move very fast, and this approximation is not a major drawback.

15.3.3 First extension: retinal persistence

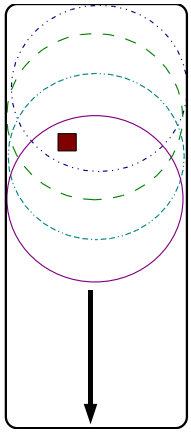


Figure 15.5: The pixel indicated in red receives a contribution from all the successive positions of the raindrop.

In a photography or a motion picture, raindrops appear like streaks due to the shutter speed of the camera. While the shutter is opened, a drop falls down a few centimeters, and impresses the film on a short vertical distance. This effect is usually called "motion blur". It would not be visible for an ideal camera using an infinitesimal shutter speed.

The eye observing real rain behaves differently, for the same result. An eye does not have a shutter, but when an image forms on the retina, it takes 60 to 90 milliseconds to fade away. During this time lapse, the drop keeps falling, and all its different positions compose a continuous sequence of images on the retina, producing this persistence effect.

Human eye is not used to seeing almost spherical drops; our model, although it is physically correct, seems to lack realism. We extended our model to take into account retinal persistence, and generate streaks based upon our accurate raindrop model.

To simulate this effect, our rain particles are reshaped into vertical streaks. We modify the pixel shader we use so that each pixel of a streak receives the contribution of a few successive sample positions of the drop, as illustrated in Figure 15.5.

15.3.4 Second extension: light/raindrop interaction

When rain falls near street lights or head lights, real raindrops present reflects of the color of the light sources.

The optical laws presented in section 15.2.2 still apply when a light source is positioned in the scene. When the observer is close to a light source, the rays coming from this source have a far greater intensity than rays coming from anywhere else in the scene. Using the method described above, a light source positioned behind the observer would not have any influence on the generated raindrops, because our model does not handle reflection (which most of the time, is negligible, see section 15.3.1). In the case of a close light source, reflection and internal reflection or refraction cannot be ignored, since they have an important participation to the appearance of the drop (considering the intensity of rays coming from the light source.)

Computing all the internal reflections of light rays would be the best way to generate physically satisfying images, but it cannot be done in real-time. We simulate this effect by modifying the color of the raindrops pixels, based on the distance between the raindrop and the light source.

This gives visually satisfying results, and can be used with our basic raindrop model as well as with our retinal persistence extension. In our implementation of this technique, we can handle two point light sources at the same time, without any significant performance loss.

15.4 Results

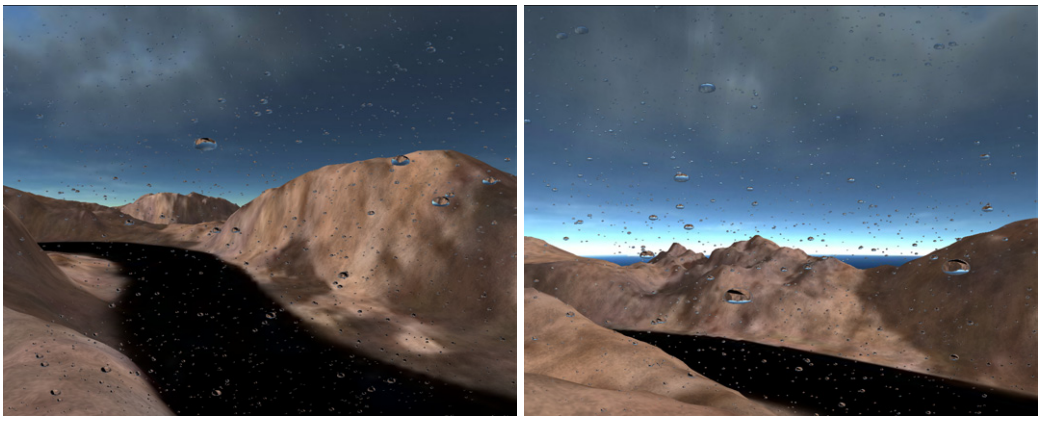


Figure 15.6: Two images generated with our method.

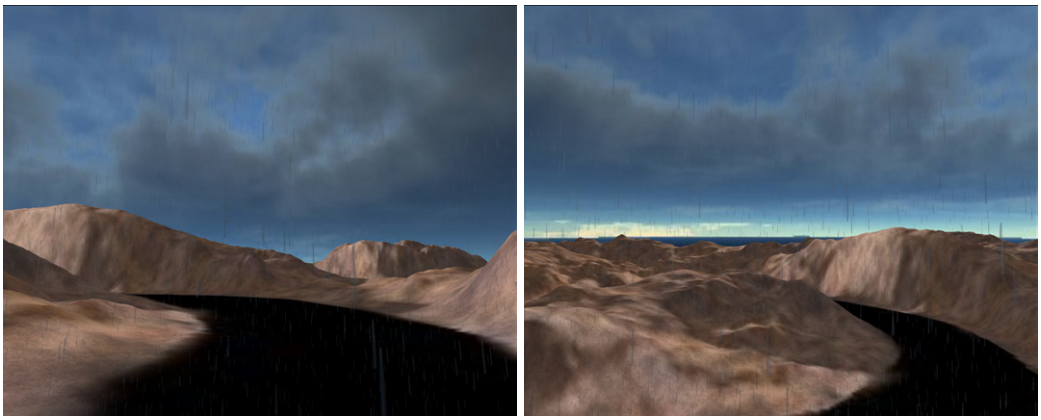


Figure 15.7: Two images generated using our retinal persistence extension.

The main bottleneck in our application is the handling of the particle system. On a PC with a 2600+ AMD CPU and an nVidia Geforce 6800 GT video card, our method generates more than 100 frames per second, when using 5000 particles. Increasing the particle count to 20 000, reduces this frame-rate to 25 images per second. This bottleneck should be removed using a hardware implementation of the particle system, as proposed by Kolb *et al.* [67].

5000 particles are sufficient to provide a realistic rain impression for large raindrops or streaks. When using very small raindrops (below a radius of 1mm), 10000 particles are required for a realistic rain impression.

Figure 15.6 shows a scene including 5000 raindrops of radius 4.5 mm, animated at a frame-rate of 100 Hz.

Figure 15.7 shows results obtained with our retinal persistence extension. 5000 raindrops of radius 1.5 mm are animated at 70 Hz. This extension implies a higher computing cost (depending on the number of samples used), but needs less particles to produce a realistic effect.

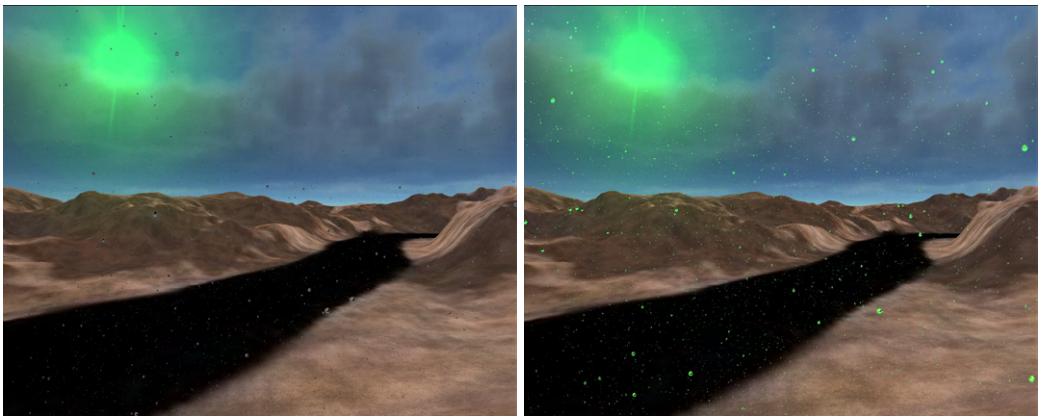


Figure 15.8: A light source modifies the color of the raindrops. Left: Without the light/raindrop extension. Right: With the extension activated.

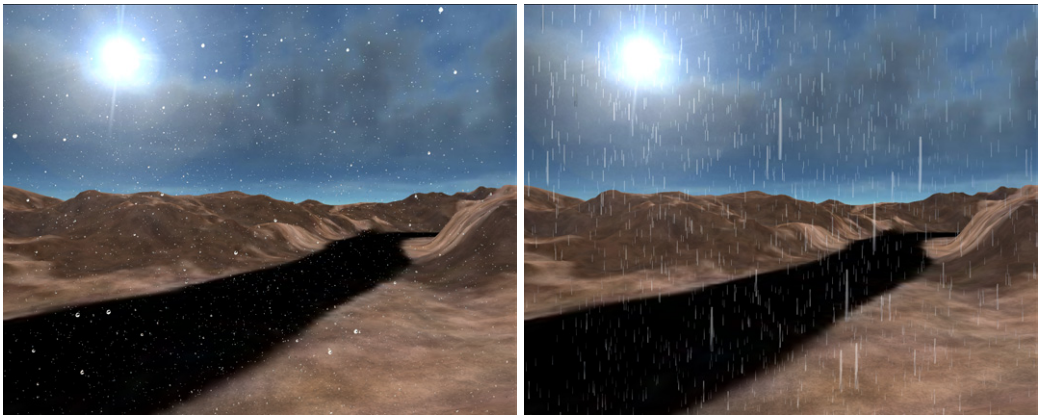


Figure 15.9: Left: Light/raindrop extension, with a white colored light. Right: Using the retinal persistence extension.

Figure 15.8 and Figure 15.9 show images generated with a white or coloured light source, using our retinal persistence extension in the right image.

15.5 Conclusion

We have developed a physically based real-time model for rendering raindrops. We extended this model to handle retinal persistence and light sources. Our model produces better results than the usual particle systems using static textures which are often used in video-games. It achieves a much faster rendering speed than the existing physical based models.

We believe that our model can be widely used in video-games or driving simulators as it generates visually convincing results at a high frame-rate.

For perfectly accurate results, the two possible techniques are either a complete ray-tracing on each object of the scene, or a dynamic generation of a cubic environment map for every single raindrop. Both of these methods cannot run in real-time, at least with current graphics hardware. Our model introduces some approximations to these methods. Consequently, it is not physically completely accurate but allows a real-time high frame-rate execution, and is a good approximation of the images which would be obtained by other methods.

Bibliography

[1] C. Andujar, P. Brunet, A. Chica, I. Navazo, J. Rossignac, and J. Vinacua. Computing maximal tiles and application to impostor-based simplification. *Computer Graphics Forum*, 23(3):401–410, 2004.

[2] T. Annen, J. Kautz, F. Durand, and H-P. Seidel. Spherical harmonic gradients for mid-range illumination. In *Eurographics Symposium on Rendering*, 2004.

[3] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*, pages 201–262. Academic Press, London, 1989.

[4] M. Ashikhmin, S. Premoze, and P. Shirley. A microfacet-based BRDF generator. In *SIGGRAPH'00*, 2000.

[5] L. Aupperle and P. Hanrahan. A hierarchical illumination algorithms for surfaces with glossy reflection. *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 155–162, 1993.

[6] R. Bastos, M. Goslin, and H. Zhang. Efficient radiosity rendering using textures and bicubic reconstruction. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics*, 1997.

[7] K. V. Beard and C. Chuang. A new model for the equilibrium shape of raindrops. *J. Atmos. Sci.*, 44(11):1509–1524, 1987.

[8] P. Bekaert. *Hierarchical and stochastic algorithms for radiosity.* PhD thesis, University of Leuven, 1999.

[9] P. Bekaert, M. Sbert, and J. Halton. Accelerating path tracing by re-using paths. In *Proceedings of Workshop on Rendering*, pages 125–134, 2002.

[10] Philippe Bekaert, Laszlo Neumann, Attila Neumann, Mateu Sbert, and Yves D. Willems. Hierarchical monte carlo radiosity. In G. Drettakis and N. Max, editors, *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop '98)*, pages 259–268, New York, NY, 1998. Springer Wien.

[11] Philippe Bekaert, Mateu Sbert, and Yves D. Willems. Weighted importance sampling techniques for monte carlo radiosity. In B. Peroche and H. Rushmeier, editors, *Rendering Techniques 2000 (Proceedings of the Eleventh Eurographics Workshop on Rendering)*, pages 35–46, New York, NY, 2000. Springer Wien.

[12] K. Bjorke. Image-based lighting. In R. Fernando, editor, *GPU Gems*, pages 307–322. NVidia, 2004.

[13] J. F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *SIGGRAPH '82 Proceedings*, pages 21–29, 1982.

[14] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.

[15] C. Buckalew and D. Fussell. Illumination networks: Fast realistic rendering with general reflectance functions. *SIGGRAPH '89 Proceedings*, 23(3):89–98, July 1989.

[16] Michael Bunnel. *GPU Gems 2*, chapter Dynamic Ambient Occlusion and Indirect Lighting. Addison-Wesley Professional, 2005.

[17] N. Carr, J. Hall, and Hart J. GPU algorithms for radiosity and subsurface scattering. In *Proc. of Workshop on Graphics Hardware*, pages 51–59, 2003.

[18] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proc. Graph. Hw. 2002*, pages 1–10, 2002.

[19] P. Christensen. Faster photon map global illumination. *Journal of Graphics Tools*, 4(3):1–10, 2000.

[20] Per H. Christensen. Global illumination and all that. SIGGRAPH 2003 course notes #9, RenderMan: Theory and Practice, July 2003.

[21] C. Chuang and K.V. Beard. A numerical model for the equilibrium shape of electrified raindrops. *J. Atmos. Sci.*, 47(11):1374–1389, 1990.

[22] M. Cohen and D. Greenberg. The hemi-cube, a radiosity solution for complex environments. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 31–40, 1985.

[23] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, pages 75–84, 1988.

[24] R. Cook and K. Torrance. A reflectance model for computer graphics. *Computer Graphics*, 15(3), 1981.

[25] G. Coombe, M. J. Harris, and A. Lastra. Radiosity on graphics hardware. In *Graphics Interface*, 2004.

[26] W. Cornette and J. Shanks. Physical reasonable analytic expression for single-scattering phase function. *Applied Optics*, 31(16):31–52, 1992.

[27] F. Dachille, K. Mueller, and A. Kaufman. Volumetric global illumination and reconstruction via energy backprojection. In *Symposium on Volume Rendering*, 2000.

[28] K. Dana, B. Ginneken, S. Nayar, and J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics*, 18(1):1–34, 1997.

[29] P. Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *SIGGRAPH '98*, pages 189–198, 1998.

[30] P. Debevec, G. Borshukov, and Y. Yu. Efficient view-dependent image-based rendering with projective texture-mapping. In *9th Eurographics Rendering Workshop*, 1998.

[31] X. Décoret, F. Durand, F. Sillion, and J. Dorsey. Billboard clouds for extreme model simplification. In *SIGGRAPH '2003 Proceedings*, pages 689–696, 2003.

[32] X. Decoret, F. Sillion, G. Schaufler, and J. Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, 18(3), 1999.

[33] O. Deussen, P. Hanrahan, R. Lintermann, R. Mech, M. Pharr, and P. Prusinkiewicz. Interactive modeling and rendering of plant ecosystems. In *SIGGRAPH '98 Proceedings*, pages 275–286, 1998.

[34] K. Dmitriev, S. Brabec, K. Myszkowski, and H-P. Seidel. Interactive global illumination using selective photon tracing. In *Rendering Techniques 2002 (Proceedings of the Thirteenth Eurographics Workshop on Rendering)*, June 2002.

[35] P. Dutre, P. Bekaert, and K. Bala. *Advanced Global Illumination*. A K Peters, 2003.

[36] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 2001.

[37] R.P. Feynman, R.B. Leighton, and Sands. M. *The Feynman Lectures on Physics*. Addison-Wesley, 1969.

[38] R. Geist, K. Rasche, J. Westall, and R. Schalkoff. Lattice-boltzmann lighting. In *Eurographics Symposium on Rendering*, 2004.

[39] A. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers, Inc., San Francisco, 1995.

[40] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the Interaction of Light Between Diffuse Surfaces. In *Computer Graphics (ACM SIGGRAPH '84 Proceedings)*, volume 18, pages 212–222, July 1984.

[41] X. Granier and G. Drettakis. Inremental updates for rapid glossy global illumination. *Computer Graphics Forum*, 20(3):268–277, 2001.

[42] N. Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1984.

[43] T. Hachisuka. Final gathering on GPU. In *ACM Workshop on General Purpose Computing on Graphics Processors*, 2004.

[44] Toshiya Hachisuka. *GPU Gems 2*, chapter High-Quality Global Illumination Rendering Using Rasterization. Addison-Wesley Professional, 2005.

[45] Mark J. Harris. Real-time cloud rendering for games. In *Game Developers Conference*, 2002.

[46] Mark J. Harris, William V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Eurographics Graphics Hardware'2003*, 2003.

[47] Mark J. Harris and Anselmo Lastra. Real-time cloud rendering. *Computer Graphics Forum*, 20(3), 2001.

[48] V. Havran. *Heuristic Ray Shooting Algorithms*. Czech Technical University, Ph.D. dissertation, 2001.

[49] X. He, K. Torrance, F. Sillion, and D. Greenberg. A comprehensive physical model for light reflection. *Computer Graphics*, 25(4):175–186, 1991.

[50] W. Heidrich, K. Daubert, J. Kautz, and H.-P. Seidel. Illuminating micro geometry based on precomputed visibility. In *SIGGRAPH 2000 Proceedings*, pages 455–464, 2000.

[51] G. Henyey and J. Greenstein. Diffuse radiation in the galaxy. *Astrophysical Journal*, 88:70–73, 1940.

[52] Heinrich Hey and Werner Purgathofer. Advanced radiance estimation for photon map global illumination. *Computer Graphics Forum (Proceedings of Eurographics 2002)*, 21(3), September 2002.

[53] American National Standard Institute. Fresnel reflection. Technical report, NVidia, 2002. http://developer.nvidia.com/attach/6664.

[54] A. Iones, A. Krupkin, M. Sbert, and S. Zhukov. Fast realistic lighting for video games. *IEEE Computer Graphics and Applications*, 23(3):54–64, 2003.

[55] H. W. Jensen. Global illumination using photon maps. In *Rendering Techniques '96*, pages 21–30, 1996.

[56] H. W. Jensen and N. J. Christensen. Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *Computers and Graphics*, 19(2):215–224, 1995.

[57] H. W. Jensen and P. H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. *SIGGRAPH '98 Proceedings*, pages 311–320, 1998.

[58] Henrik Wann Jensen and Per H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. In *SIGGRAPH '98 Proceedings*, pages 311–320, 1998.

[59] J. T. Kajiya. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, pages 143–150, 1986.

[60] K. Kaneda, T. Kagawa, and H. Yamashita. Animation of water droplets on a glass plate. In *Computer Animation*, pages 177–189, 1993.

[61] Kazufumi Kaneda, Shinya Ikeda, and Hideo Yamashita. Animation of water droplets moving down a surface. *Journal of Visualization and Computer Animation*, 10(1):15–26, 1999.

[62] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Analysis and Mach. Int.*, 24(7):881–892, 2002.

[63] J. Kautz, P. Sloan, and J. Snyder. Fast, arbitrary BRDF shading for low-frequency lighting using spherical harmonics. In *12th EG Workshop on Rendering*, pages 301–308, 2002.

[64] Cs. Kelemen and L. Szirmay-Kalos. A microfacet based coupled specular-matte BRDF model with importance sampling. In *Eurographics 2001, Short papers, Manchester*, 2001.

[65] A. Keller. Instant radiosity. In *SIGGRAPH '97 Proceedings*, pages 49–55, 1997.

[66] A. Keller. *Quasi-Monte Carlo Methods for Photorealistic Image Synthesis*. Shaker-Verlag, 1998.

[67] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 123–131, New York, NY, USA, 2004. ACM Press.

[68] T. Kollig and A. Keller. Efficient illumination by high dynamic range images. In *Eurographics Symposium on Rendering*, pages 45–51, 2003.

[69] Eric P. Lafortune and Yves D. Willems. Rendering participating media with bidirectional path tracing. In *Rendering Techniques '96*, pages 91–100, 1996.

[70] E. Languenou, K. Bouatouch, and M. Chelle. Global illumination in presence of participating media with general properties. In *Eurographics Workshop on Rendering*, pages 69–85, 1995.

[71] J. Lehtinen and J. Kautz. Matrix radiance transfer. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 59–64, 2003.

[72] Hendrik P.A. Lensch, Michael Goesele, Philippe Bekaert, Jan Kautz, Marcus A. Magnor, Jochen Lang, and Hans-Peter Seidel. Interactive rendering of translucent objects. *Computer Graphics Forum*, 22(2):195–195, 2003.

[73] M. Levoy and P. Hanrahan. Light field rendering. In *SIGGRAPH 96*, pages 31–42, 1996.

[74] S. Lloyd. Least square quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[75] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, 2002.

[76] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *Symposium on Interactive 3D Graphics*, pages 95–102, 211, 1995.

[77] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[78] T. Malzbender, D. Gelb, and H. Wolters. Polynomial texture maps. In *SIGGRAPH 2001*, pages 519–528, 2001.

[79] R. Martinez, L. Szirmay-Kalos, and M. Sbert. A hardware based implementation of the multipath method. In *Computer Graphics International*, Bradford, UK, 2002.

[80] N. Max, O. Deussen, and B. Keating. Hierarchical image-based rendering using texture mapping. In *Eurographics Workshop on Rendering*, pages 57–62, 1999.

[81] N. Max and K. Ohsaki. Rendering trees from pre-computed z-buffer views. In *Eurographics Workshop on Rendering*, pages 74–81, 1995.

[82] Nelson L. Max. Efficient light propagation for multiple anisotropic volume scattering. In *Eurographics Workshop on Rendering*, pages 87–104, 1994.

[83] Ch. Mei, J. Shi, and F. Wu. Rendering with spherical radiance transport maps. *Computer Graphics Forum (Eurographics 04)*, 23(3):281–290, 2004.

[84] Àlex Méndez-Feliu and Mateu Sbert. Comparing hemisphere sampling techniques for obscurance computation. In *Proccedings of the International Conference on Computer Graphics and Artificial Intelligence (3IA 2004)*, Limoges, France, May 2004.

[85] Àlex Méndez-Feliu, Mateu Sbert, and Jordi Catà. Real-time obscurances with color bleeding. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, pages 171–176. ACM Press, april 2003.

[86] A. Meyer, F. Neyeret, and Poulin. Interactive rendering of trees with shading and shadows. In *Eurographics Workshop on Rendering*, 2001.

[87] G. S. Miller and C. R. Hoffman. Illumination and reflection maps: Simulated objects in simulated and real environment. In *SIGGRAPH '84*, 1984.

[88] L. Neumann. Monte Carlo radiosity. *Computing*, 55:23–42, 1995.

[89] L. Neumann, A. Neumann, and L. Szirmay-Kalos. Compact metallic reflectance models. *Computer Graphics Forum (Eurographics'99)*, 18(3):161–172, 1999.

[90] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. All-frequency shadows using non-linear wavelet lighting approximation. *ACM Trans. Graph.*, 22(3):376–381, 2003.

[91] K. Nielsen and N. Christensen. Fast texture based form factor calculations for radiosity using graphics hardware. *Journal of Graphics Tools*, 6(2):1–12, 2002.

[92] T. Nishita, Y. Dobashi, and E. Nakamae. Displaying of clouds taking into account multiple anisotropic scattering and sky light. In *SIGGRAPH '96 Proceedings*, pages 379–386, 1996.

[93] V. Ostromoukhov, C. Donohue, and P-M. Jodoin. Fast hierarchical importance sampling with blue noise properties. In *Proc. SIGGRAPH 2004*, 2004.

[94] M. Pauly, T. Kollig, and A. Keller. Metropolis light transport for participating media. In *Rendering Techniques*, pages 11–22, 2000.

[95] Matt Pharr and Simon Green. *GPU Gems*. Addison-Wesley Professional, 2004.

[96] M. Powell and J. Swann. Weighted importance sampling — a Monte-Carlo technique for reducing variance. *Inst. Maths. Applics.*, 2:228–236, 1966.

[97] A. Preetham, P. Shirley, and B. Smits. A practical analytic model for daylight. In *SIGGRAPH '99 Proccedings*, pages 91–100, 1999.

[98] R. W. Preisendorfer. *Radiative Transfer on Discrete Spaces*. Pergamon Press, 1965.

[99] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.

[100] T. Purcell, T. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50, 2003.

[101] A. Reche, I. Martin, and G. Drettakis. Volumetric reconstruction and interactive rendering of trees from photographs. *ACM Transactions on Graphics*, 23(3), 2004.

[102] W. T. Reeves. Particle systems - techniques for modelling a class of fuzzy objects. In *SIGGRAPH '83 Proceedings*, pages 359–376, 1983.

[103] E. Reinhard, L. U. Tijssen, and W. Jansen. Environment mapping for efficient sampling of the diffuse interreflection. In *Photorealistic Rendering Techniques*, pages 410–422. Springer, 1994.

[104] K. Riley, D. Ebert, M. Kraus, J. Tessendorf, and C. Hansen. Efficient rendering of atmospheric phenomena. In *Eurographics Symposium on Rendering*, pages 374–386, 2004.

[105] Oliver N. Ross. Optical remote sensing of rainfall micro-structures. Master's thesis, Freie Universitt Berlin, 2000. in partnership with University of Auckland.

[106] Tomoya Sato, Yoshinori Dobashi, and Tsuyoshi Yamamoto. A method for real-time rendering of water droplets taking into account interactive depth of field effects. In *Entertainment Computing: Technologies and Applications, IFIP First International Workshop on Entertainment Computing (IWEC 2002)*, volume 240 of *IFIP Conference Proceedings*, pages 125–132. Kluwer, 2002.

[107] M. Sbert. *The Use of Global Directions to Compute Radiosity*. PhD thesis, Catalan Technical University, Barcelona, 1996.

[108] M. Sbert, L. Szecsi, and L. Szirmay-Kalos. Real-time light animation. *Computer Graphics Forum (Eurographics 04)*, 23(3):291–300, 2004.

[109] Mateu Sbert. *The Use of Global Random Directions to Compute Radiosity: Global Monte Carlo Techniques*. PhD thesis, Universitat Politecnica de Catalunya, Barcelona, Spain, 1997. Available from http://ima.udg.es/ mateu.

[110] G. Schaufler. Dynamically generated impostors. In *I Workshop - Virtual Worlds - Distributed Graphics*, pages 129–136, 1995.

[111] G. Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In *Eurographics Workshop on Rendering*, pages 151–162, 1997.

[112] Ch. Schlick. A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering*, pages 73–83, 1993.

[113] P. Shirley. Time complexity of Monte-Carlo radiosity. In *Eurographics '91*, pages 459–466. Elsevier Science Publishers, 1991.

[114] P. Shirley, B. Wade, P. Hubbard, and D Zareski. Global illumination via density-estimation radiosity. In *Eurographics Rendering Workshop '95*, 1995.

[115] F. Sillion and C. Puech. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers, Inc., San Francisco, 1994.

[116] M. Slater, J. Mortensen, P. Khanna, and I. Yu. A virtual light field approach to global illumination. In *Computer Graphics International*, pages 102–109, 2004.

[117] P. Sloan, J. Hall, J. Hart, and J. Snyder. Clustered principal components for precomputed radiance transfer. In *SIGGRAPH 2003*, 2003.

[118] P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH 2002 Proceedings*, pages 527–536, 2002.

[119] Wolfgang Sturzlinger and Rui Bastos. Interactive rendering of globally illuminated glossy scenes. In Julie Dorsey and Philipp Slusallek, editors, *Rendering Techniques '97 (Proceedings of the Eighth Eurographics Workshop on Rendering)*, pages 93–102, New York, NY, 1997. Springer Wien. ISBN 3-211-83001-4.

[120] F. Suykens, K. Berge, A. Lagae, and Dutré Ph. Interactive rendering with bidirectional texture functions. *Computer Graphics Forum (Eurographics 03)*, 22(3):464–472, 2003.

[121] L. Szecsi, M. Sbert, and L. Szirmay-Kalos. Combined correlated and importance sampling in direct light source computation and environment mapping. *Computer Graphics Forum (Eurographics 04)*, 23(3):585–604, 2004.

[122] G. Szijártó. 2.5 dimensional impostors for realistic trees and forests. In Kim Pallister, editor, *Game Programming Gems 5*, pages 527–538. Charles River Media, 2005.

[123] L. Szirmay-Kalos. Stochastic iteration for non-diffuse global illumination. *Computer Graphics Forum*, 18(3):233–244, 1999.

[124] L. Szirmay-Kalos, Gy. Antal, and Benedek B. Global illumination animation with random radiance representation. In *Rendering Symposium*, 2003.

[125] L. Szirmay-Kalos, Gy. Antal, and B. Benedek. Weighted importance sampling in shooting algorithms. In *Spring Conference on Computer Graphics 2003*, pages 192–198, 2003.

[126] L. Szirmay-Kalos and G. Márton. On convergence and complexity of radiosity algorithms. In *Winter School of Computer Graphics '95*, pages 313–322, Plzen, Czech Republic, 14–18 February 1995. http//www.iit.bme.hu/~szirmay.

[127] L. Szirmay-Kalos and W. Purgathofer. Global ray-bundle tracing with hardware acceleration. In *Rendering Techniques '98*, pages 247–258, 1998.

[128] Laszlo Szirmay-Kalos and Werner Purgathofer. Global ray-bundle tracing with hardware acceleration. In G. Drettakis and N. Max, editors, *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop '98)*, pages 247–258, New York, NY, 1998. Springer Wien.

[129] C. Trendall and A. Stewart. General calculations using graphics hardware, with application to interactive caustics. In *Rendering Techniques 2000*, pages 287–298, 2000.

[130] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slussalek. Interactive global illumination using fast ray tracing. In *13th Eurographics Workshop on Rendering*, 2002.

[131] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. In *Rendering techniques '99*, volume 10, pages 235–246, Jun 1999.

[132] M. Wand and W. Strasser. Real-time caustics. *Computer Graphics Forum*, 22(3):611–620, 2003.

[133] L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, and B. Guo. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22(3):334–339, 2004.

[134] E. Weisstein. World of mathematics. Technical report, 2003. http://mathworld.wolfram.com/Curvature.html.

[135] E. Weisstein. World of mathematics. Technical report, 2003. http://mathworld.wolfram.com/Eigenvector.html.

[136] T. Welsh. Parallax mapping with offset limiting: A perpixel approximation of uneven surfaces. Technical report, Infiscape Corporation, 2004.

[137] A. Wilkie. *Photon Tracing for Complex Environments*. PhD thesis, Institute of Computer Graphics, Vienna University of Technology, 2001.

[138] L. Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, 1978.

[139] Sergey Zhukov, Andrei Iones, and Grigorij Kronin. An ambient light illumination model. In George Drettakis and Nelson L. Max, editors, *Rendering Techniques*, pages 45–56. Springer, 1998.