High Quality Rendering using Ray Tracing and Photon Mapping

Siggraph 2007 Course 8

Monday, August 5, 2007

Lecturers

Henrik Wann Jensen
University of California, San Diego

and

Per Christensen
Pixar Animation Studios

Abstract

Ray tracing and photon mapping provide a practical way of efficiently simulating global illumination including interreflections, caustics, color bleeding, participating media and subsurface scattering in scenes with complicated geometry and advanced material models.

This halfday course will provide the insight necessary to efficiently implement and use ray tracing and photon mapping to simulate global illumination in complex scenes. The presentation will cover the fundamentals of ray tracing and photon mapping including efficient techniques and data-structures for managing large numbers of rays and photons. In addition, we will describe how to integrate the information from the photon maps in shading algorithms to render global illumination effects such as caustics, color bleeding, participating media, subsurface scattering, and motion blur. Finally, we will describe recent advances for dealing with highly complex movie scenes as well as recent work on realtime ray tracing and photon mapping.

About the Lecturers

Per H. Christensen

Pixar Animation Studios
per@pixar.com

Per Christensen is a senior software developer in Pixar's RenderMan group. His main research interest is efficient ray tracing and global illumination in very complex scenes. He received an M.Sc. degree in electrical engineering from the Technical University of Denmark and a Ph.D. in computer science from the University of Washington in Seattle. His movie credits include "Final Fantasy", "Finding Nemo", "The Incredibles", and "Cars".

Henrik Wann Jensen

University of California, San Diego
henrik@graphics.ucsd.edu
http://graphics.ucsd.edu/~henrik

Henrik Wann Jensen is an Associate Professor at UC San Diego where he teaches computer graphics. His main research interest is in the area of global illumination and appearance modeling. He received his M.Sc. degree and his PhD degree from the Technical University of Denmark. He is the author of "Realistic Image Synthesis using Photon Mapping," AK Peters 2001.

Contents

Chapter 1

Introduction

This course material describes in detail the practical aspects of the ray tracing and photon map algorithms. The text is based on published papers as well as industry experience. After reading this course material, it should be relatively straightforward to add an efficient implementation of the photon map algorithm to any ray tracer and to understand the details necessary to make ray tracing and photon mapping robust in complex scenes.

1.1 Motivation

The photon mapping method is an extension of ray tracing. In 1989, Andrew Glassner wrote about ray tracing [25]:

> "Today ray tracing is one of the most popular and powerful techniques in the image synthesis repertoire: it is simple, elegant, and easily implemented. [However] there are some aspects of the real world that ray tracing doesn't handle very well (or at all!) as of this writing. Perhaps the most important omissions are diffuse inter-reflections (e.g. the 'bleeding' of colored light from a dull red file cabinet onto a white carpet, giving the carpet a pink tint), and caustics (focused light, like the shimmering waves at the bottom of a swimming pool)."

At the time of the development of the photon map algorithm in 1993, these problems were still not addressed efficiently by any ray tracing algorithm. The photon map method offers a solution to both problems. Diffuse interreflections and caustics are both indirect illumination of diffuse surfaces; with the photon map

method, such illumination is estimated using precomputed photon maps. Extending ray tracing with photon maps yields a method capable of efficiently simulating all types of direct and indirect illumination. Furthermore, the photon map method can handle participating media and it is fairly simple to parallelize.

1.2 What is photon mapping?

The photon map algorithm was developed in 1993–1994 and the first papers on the method were published in 1995. It is a versatile algorithm capable of simulating global illumination including caustics, diffuse interreflections, and participating media in complex scenes. It provides the same flexibility as general Monte Carlo ray tracing methods using only a fraction of the computation time.

The global illumination algorithm based on photon maps is a two-pass method. The first pass builds the photon map by emitting photons from the light sources into the scene and storing them in a *photon map* when they hit non-specular objects. The second pass, the rendering pass, uses statistical techniques on the photon map to extract information about incoming flux and reflected radiance at any point in the scene. The photon map is decoupled from the geometric representation of the scene. This is a key feature of the algorithm, making it capable of simulating global illumination in complex scenes containing millions of triangles, instanced geometry, and complex procedurally defined objects.

Compared with finite element radiosity, photon maps have the advantage that no meshing is required. The radiosity algorithm is faster for simple diffuse scenes but as the complexity of the scene increases, photon maps tend to scale better. Also the photon map method handles non-diffuse surfaces and caustics.

Monte Carlo ray tracing methods such as path tracing, bidirectional path tracing, and Metropolis can simulate all global illumination effects in complex scenes with very little memory overhead. The main benefit of the photon map compared with these methods is efficiency, and the price paid is the extra memory used to store the photons. For most scenes the photon map algorithm is significantly faster, and the result looks better since the error in the photon map method is of low frequency which is less noticeable than the high frequency noise of general Monte Carlo methods.

Another big advantage of photon maps (from a commercial point of view) is that there is no patent on the method; anyone can add photon maps to their renderer. As a result several commercial systems use photon maps for rendering caustics and

global illumination.

1.3 More information

For more information about photon mapping, all the practical details, the theory and the insight for understanding the technique see:

> Henrik Wann Jensen
> *Realistic Image Synthesis using Photon Mapping*
> AK Peters, 2001

This book also contains additional information about participating media and sub-surface scattering. Finally, it contains an implementation with source code in C++ of the photon map data structure.

1.4 Acknowledgements

Henrik would like to thank all the people he has worked with over the years on the research that is contained within the course notes.

Per would like to thank his colleagues at Pixar and Pixar's RenderMan team for providing an inspiring environment and for their help and support. David Laur and Julian Fong implemented large parts of the ray-tracing functionality in Pixar's RenderMan, which was used to generate many of the images in these course notes. Thanks to Wayne Wooten for digging out the data for the Pixar test scenes.

Chapter 2

Ray-tracing fundamentals

This chapter provides an overview of the fundamental aspects of the ray tracing algorithm. Ray tracing has its roots in the renaissance (or perhaps even earlier), and was used for drawing images with correct perspective foreshortening and for design of mirrors and lenses for telescopes and other optical instruments.

Ray tracing can be used for visibility testing (projecting the 3D scene to a 2D image) and shadows, but it really comes into its right element when used to compute specular reflections and refractions. The basic ray tracing algorithm is very simple and elegant; nevertheless it can compute visual phenomena that are very hard to compute with any other method. Monte Carlo ray tracing adds further effects such as soft shadows and ambient occlusion, glossy and diffuse reflections, depth-of-field, and motion blur.

This chapter starts with a brief history of ray tracing, and then shows how to use ray tracing for direct rendering. It then provides an overview of ray intersection calculation methods for various types of surfaces, a key part of ray tracing. Then follows a brief overview of reflection models. With these basics covered, the use of ray tracing to render shadows, reflection and refraction, soft shadows, glossy and diffuse reflections, depth-of-field, and motion blur is described. Various spatial acceleration data structures are used to speed up ray tracing in scenes with many objects; those are only described very superficially. At the end there is a description of ray differentials, a relatively new concept that is useful for many applications including texture filtering and tessellation. The last section contains recommendations for further reading.

2.1 Historical background: ray tracing in the pre-computer age

Ray tracing was used long before the electronic computer was invented. Figure 2.1 shows ray tracing in the year 1525 — back then, the "computer" was the artist's assistant and rays were strands of thread. Albrecht Dürer (1471–1528), a German renaissance painter and engraver, used this device to render images with correct perspective projection [22]. Points on the object (a lute) are projected onto the image. Nowadays we would call this process a "projection of 3D points onto a 2D image".

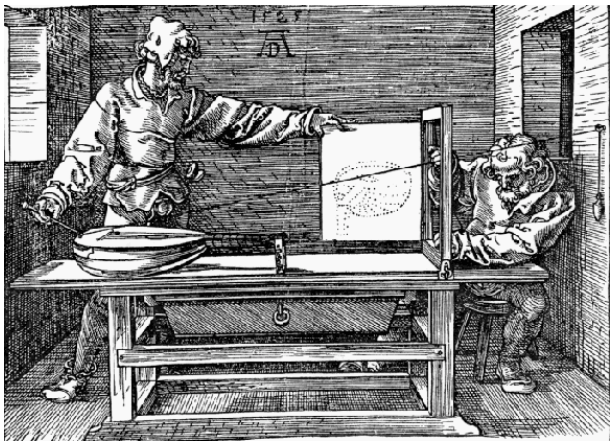


Figure 2.1: Mechanical creation of a perspective image (Dürer, 1525).

Ray tracing was also used for lens design for microscopes, telescopes, binoculars, and cameras. Sir Isaac Newton (1642–1727) showed reflection and refraction of rays in his famous 1704 book *Opticks* [50]. One of his illustrations is shown in figure 2.2.

2.2 Visibility ray tracing

Rendering consists of computing the color of each point in an image. This is done by projecting the 3D scene onto a 2D image. Ray tracing is one of the most popular rendering methods. The basic ray tracing algorithm consists of two main calculations per pixel: find the nearest surface point and compute the color at that point. It determines the visibility of object surfaces by following imaginary rays from the

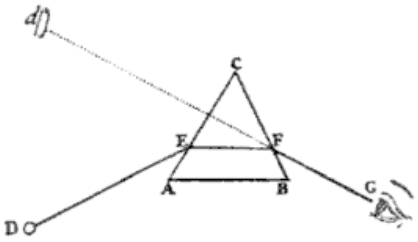


Figure 2.2: Prism with refracted rays (Newton, 1704).

viewer's eye to the objects in the scene. This limited type of ray tracing is sometimes referred to as *ray casting*. Appel [3] was the first to use a computer to render ray-traced images.

The simplest ray tracing algorithm is as follows:

```
for each pixel do
  compute ray for that pixel
  for each object in scene do
    if ray intersects object and intersection is nearest so far then
      record intersection distance and object color
  set pixel color to nearest object color (if any)
```

Computing the ray corresponding to a pixel is very simple: the ray origin is at the viewpoint, and the ray direction is from the viewpoint to the pixel position in the image plane. The computation of the intersection of a ray with an object is fairly simple; we cover the details of this in the following section (2.3).

A simple way to improve the image quality is to shoot several rays per pixel. This reduces aliasing effects such as "jaggies" and staircase effects along object silhouettes.

Figure 2.3 shows a simple ray-traced image of two teapots. The image shows the shape of the teapots, as seen from the viewpoint. However, the image is very simplistic since the teapots are simply rendered black. To improve on this, we need shading and reflection models which are described in section 2.4.

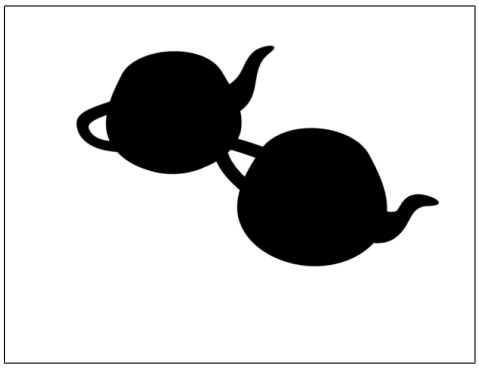


Figure 2.3: Simple image of two teapots.

2.3 Ray intersection calculations

At the heart of all ray tracing algorithms is the computation of ray-object intersections. If there is more than one intersection we usually want the nearest. (However, for shadow rays we often just need to know whether there is *any* intersection; it doesn't matter if it's the nearest one.) This section provides an overview of ray-object intersection calculation methods.

First a definition: a ray is a semi-infinite line. It is defined by an origin $\vec{o}$ and a direction $\vec{d}$: $\vec{p}(t) = \vec{o} + t\vec{d}$ for all $t \geq 0$.

2.3.1 Triangles

A triangle $\mathbf{abc}$ is defined by its three vertices $\vec{a}$, $\vec{b}$, $\vec{c}$. The normal of the triangle can be computed (on-the-fly or just once and stored with the triangle data) using the cross product of two of the triangle edges, for example $(\vec{b} - \vec{a})$ and $(\vec{c} - \vec{a})$:

$$\vec{n} = (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a}) \,.$$

The ray-triangle intersection point $\vec{p}$ must be along the ray (i.e. $\vec{p} = \vec{o} + t\vec{d}$) and must be in the plane of the triangle (i.e. the vector from a triangle vertex to $\vec{p}$ must be perpendicular to the triangle normal), $(\vec{p} - \vec{a}) \cdot \vec{n} = 0$. From these two equations we get:

$$\begin{aligned} 0 &= (\vec{p} - \vec{a}) \cdot \vec{n} \\ &= (\vec{o} + t\vec{d} - \vec{a}) \cdot \vec{n} \\ &= (\vec{o} - \vec{a}) \cdot \vec{n} + t\vec{d} \cdot \vec{n} \end{aligned}$$

Solving for t we get:

$$t = \frac{(\vec{a} - \vec{o}) \cdot \vec{n}}{\vec{d} \cdot \vec{n}} .$$

If the dot product $\vec{d} \cdot \vec{n}$ is 0 the ray is parallel to the plane and there is no intersection. If the computed t is negative there is an intersection but it is behind the ray origin, so we reject it. We also reject the intersection if it is further away than a previously found intersection for that ray.

Given the distance t, we can compute the ray-plane intersection point: $\vec{p} = \vec{o} + t\vec{d}$. Next we must check whether this point is inside or outside the triangle. We do this by computing the barycentric coordinates (u, v) of the hit point. The barycentric coordinates of $\vec{p}$ are defined by

$$\vec{a} + u(\vec{b} - \vec{a}) + v(\vec{c} - \vec{a}) = \vec{p}$$

— as illustrated in figure 2.4.

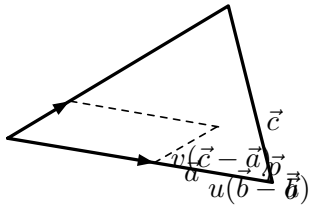


Figure 2.4: Triangle $\mathbf{abc}$ and barycentric coordinates (u, v) of point $\vec{p}$.

Since the vectors in the equation above have three components (x, y, and z) we have three equations to determine the two unknowns u and v, and we are free to choose two of the three equations to solve. In practice, one should pick the two equations where the floating point number precision is highest. Finally, when the barycentric coordinates have been computed, we can determine whether the intersection point is inside the triangle or not. If $u \geq 0$ and $v \geq 0$ and $u + v \leq 1$ then the point is inside (or on the edge of) the triangle and we finally have an actual intersection!

Further implementation details, optimizations, and pseudo-code can be found in e.g. Glassner [25] and Möller and Trumbore [48].

2.3.2 Quadrilaterals (bilinear patches)

A quadrilateral (or bilinear patch) **abcd** is defined by four vertices $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{d}$. We can express all points $\vec{p}$ on the surface of the quadrilateral as a bilinear combination of the four vertices:

$$(1-u)(1-v)\vec{a} + u(1-v)\vec{b} + (1-u)v\vec{c} + uv\vec{d} = \vec{p}.$$

Figure 2.5 shows a quadrilateral with iso-lines for u and v. The point $\vec{p}$ is on the quadrilateral if both u and v are in the range $[0, 1]$.

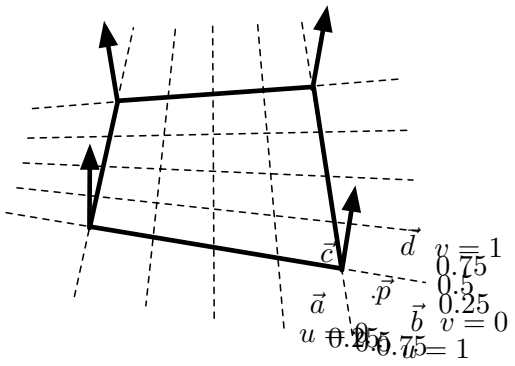


Figure 2.5: Quadrilateral **abcd** and bilinear coordinates (u, v) of point $\vec{p}$.

Quadrilaterals are not flat in general, but for flat quadrilaterals we can compute the intersection point more efficiently than for the general case. We will look at the general, non-planar case first.

Non-planar quadrilaterals

A ray can intersect a non-planar quadrilateral at 0, 1, or 2 points.

Plugging the ray equation $\vec{p} = \vec{o} + t\vec{d}$ into the quadrilateral equation above gives 3 equations (one for each of the x, y, and z coordinates) with 3 unknowns t, u, and v. Unfortunately the equations are non-linear. However, a fairly efficient method is to first solve a quadratic equation for u. If u is in the $[0, 1]$ range then compute the corresponding v. If v is also in $[0, 1]$ then compute t. Reject the intersection if t is negative or larger than the previous nearest hit distance. Implementation details for this method can be found in Stephenson [70]. The normal at the hit point can be computed e.g. by bilinear combination of the four vertex normals.

A more efficient method can be used if the quadrilateral is small. In that case it is often sufficient to approximate the quadrilateral as two triangles. The intersection points might be slightly incorrect, and the u and v iso-lines get a "kink" along the diagonal as shown in figure 2.6. However, if the quadrilateral is smaller than e.g. a pixel this is a very useful optimization, and it is used a lot in practice for calculating ray intersections with finely tessellated geometry.

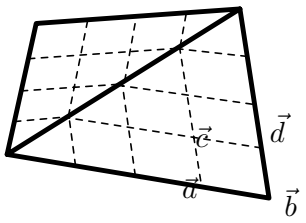


Figure 2.6: Quadrilateral $\mathbf{abcd}$ approximated by two triangles.

Planar quadrilaterals

A ray can intersect a planar quadrilateral at 0 or 1 points. One method computes the ray intersection with the plane the quadrilateral is in (using the same calculation as for ray-triangle tests) and then determines whether the intersection point is inside or outside the quadrilateral. An even more efficient method for convex planar quadrilaterals has been presented by Lagae and Dutré [43].

If the quadrilateral is a rectangle or square it is even simpler to determine the bilinear coordinates u and v — it can be done with just dot products (projecting $\vec{p}$ onto two of the edges).

2.3.3 Quadrics

The class of quadric surfaces consists of disks, spheres, cylinders, cones, ellipsoids, paraboloids, and hyperboloids.

Disks

A disk is defined by its center $\vec{c}$, normal $\vec{n}$, and radius r. The center and normal of the disk define a plane. Finding a ray-disk intersection is very similar to ray-triangle intersection testing. We first compute the ray-plane intersection point $\vec{p}$,

and check that the distance $\vec{t}$ is positive and smaller than the previous nearest intersection. The intersection point is on the disk if $(\vec{p} - \vec{c})^2 \leq r^2$.

Disks are used a lot in practical rendering, for example for rendering of particle systems.

Spheres

A sphere is defined by its center $\vec{c}$ and radius r. If there is an intersection, the intersection point must be somewhere along the ray, and must be on the surface of the sphere. To find the intersection point we plug the ray equation $\vec{p} = \vec{o} + t\vec{d}$ into the sphere equation $(\vec{p} - \vec{c})^2 = r^2$:

$$\begin{aligned} 0 &= (\vec{p} - \vec{c})^2 - r^2 \\ &= \vec{p}^2 - 2(\vec{p} \cdot \vec{c}) + \vec{c}^2 - r^2 \\ &= (\vec{o} + t\vec{d})^2 - 2(\vec{o} + t\vec{d}) \cdot \vec{c} + \vec{c}^2 - r^2 \\ &= \vec{o}^2 + 2t(\vec{o} \cdot \vec{d}) + t^2\vec{d}^2 - 2(\vec{o} \cdot \vec{c}) - 2t(\vec{d} \cdot \vec{c}) + \vec{c}^2 - r^2 \\ &= \vec{d}^2 t^2 + 2\vec{d} \cdot (\vec{o} - \vec{c})t + (\vec{o} - \vec{c})^2 - r^2 \end{aligned}$$

This is a quadratic equation in t. The two solutions are

$$t_1 = \frac{-B + D}{2A} \text{ and } t_2 = \frac{-B - D}{2A}$$

— with $A = \vec{d}^2$, $B = 2\vec{d} \cdot (\vec{o} - \vec{c})$, $C = (\vec{o} - \vec{c})^2 - r^2$, and the discriminant D is $D = \sqrt{B^2 - 4AC}$. (If we know in advance that the ray direction is normalized then $A = 1$.) If the discriminant D is negative there is no (real) solution and the ray does not hit the sphere. If the discriminant is zero the ray is tangent to the sphere, and there is only one intersection point. If the discriminant is positive there are two intersection points; the nearest intersection point is the one with the smallest non-negative value of t. Given the intersection distance t we can then compute the intersection point $\vec{p}$.

The normal at the intersection point is $\vec{n} = \vec{p} - \vec{c}$.

Other quadrics

Ray intersections with cylinders, cones, ellipsoids, paraboloids, and hyperboloids can be computed by solving a quadratic equation in a similar fashion, see e.g. Glassner [25].

Alternatively, ellipsoids can also be tested by transforming the ray by the same transformation that transforms the ellipsoid to a sphere; then the intersection can

be computed as a ray-sphere intersection. (If there is a hit, the hit point and normal must be transformed back.)

2.3.4 Implicit surfaces

An implicit surface is defined by a function f: the surface is the set of points $\vec{p}$ where the value of the function is 0, $f(\vec{p}) = 0$. So to find the ray-surface intersection we have to determine the (nearest) point $\vec{p}$ along the ray where $f(\vec{p})$ is 0:

$$f(\vec{o} + t\vec{d}) = 0$$

This can be done using e.g. Newton-Raphson iteration or other iterative methods. An efficient algorithm is described by Sherstyuk [62]. The surface normal at the intersection point is given by the gradient of the function at that point:

$$\vec{n} = \nabla f(\vec{p}) = \left(\frac{\partial f(\vec{p})}{\partial x}, \frac{\partial f(\vec{p})}{\partial y}, \frac{\partial f(\vec{p})}{\partial z} \right) .$$

2.3.5 NURBS surfaces

NURBS surfaces [23, 58] are widely used in the CAD industry for modeling cars, airplanes, etc. NURBS surfaces can be intersection tested directly (see Kajiya [36], Martin et al. [47], and Abert et al. [1]) or tessellated into polygon (triangle or quadrilateral) meshes and then intersection tested.

NURBS surfaces often have trimming curves, indicating parts of the surface that are "cut away". In this case, the ray-NURBS intersection point must be checked against the trimming curve to determine whether the intersection point is inside or outside the trim curves. If the hit point is outside the trimming curve it should be rejected.

2.3.6 Subdivision surfaces

Subdivision surfaces [10, 21] are widely used in the movie industry to model smooth surfaces with complex topology, for example humans, animals, etc. [20]. The most common subdivision surface types are Catmull-Clark [10] and Loop [46]. Direct ray tracing of subdivision surfaces is somewhat tricky, particularly near extraordinary vertices; see Kobbelt et al. [41] and Pharr and Humpheys [56] for algorithms. Another strategy is to tessellate the subdivision surface into a polygon mesh and then ray trace the mesh.

2.3.7 Displacement-mapped surfaces

Displacement maps are used to alter the shape of surfaces, for example to add details such as wrinkles, dents, large bumps, scratches, reptile scales, etc.

A method by Smits et al. [68] computes the displacements "on the fly" to determine the ray intersection points. The advantage of their method is that no extra memory is required. The disadvantage is that the displacements have to be along the normals and that the displacement function is evaluated repeatedly (which can be very time-consiming). It is faster and more general (but also more memory consuming) to tessellate larger patches of the surfaces, displace the tessellated points, and store the displaced points in a cache [55, 17].

2.3.8 Boxes

As we shall see later, bounding boxes are extremely useful for speeding up ray tracing of complex scenes.

A general box is defined by a vertex and three vectors, see figure 2.7(a). A straightforward intersection test would test each of the six faces for intersection. A faster test can be implemented by utilizing the fact that only the three faces that face toward the ray origin need to be tested.

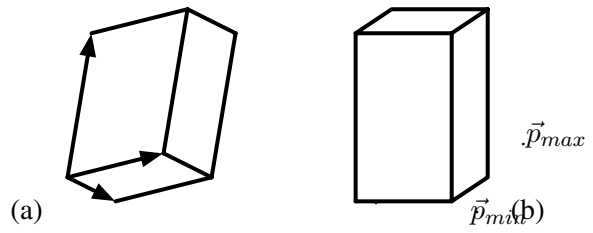


Figure 2.7: Boxes: (a) box with general orientation; (b) an axis-aligned box.

Axis-aligned boxes can be intersection-tested even more efficiently. An axis-aligned box consists of two rectangles in each of the xy, xz, and yz planes. An axis-aligned box is defined by its minimum and maximum vertices $\vec{p}_{min}$ and $\vec{p}_{max}$ — as illustrated in figure 2.7(b). We can consider the box the intersection of three infinite slabs of space. Smits [67] describes a very efficient ray intersection test that utilizes IEEE floating-point conventions to deal gracefully and efficiently with divisions by 0, thereby streamlining the code.

If the boxes are used as bounding boxes, we don't need to know the nearest

intersection point and normal, we just need to know whether the ray intersects the box or not.

2.4 Reflection models

There are many reflection models that can be used with ray tracing, but since reflection models are not a priority for this course, we will only give a very brief overview here.

Diffuse reflection can be modeled with Lambert's cosine law: the reflected light is proportional to the cosine of the angle between the incident light and the surface normal [44]. A more accurate model of diffuse reflection was developed by Oren and Nayar [51]. Glossy and specular surfaces have highlights. The highlights can be computed with the Phong cosine-power formula [8, 6], with Ward's isotropic and anisotropic reflection models [81], or with a number of other reflection models [24]. Surfaces can also have textures assigned to them to modulate the reflection parameters [9, 32]. The textures can be 2D images or 3D tables, or can be computed procedurally.

The illumination of the surfaces is provided by light sources. The simplest light source types are point lights, spot lights, and directional lights. In contrast, very complex light sources [5] are used in movie production. Such light sources can project images like a slide projector and can have complex intensity fall-off, barn doors, cucoloris ("cookies"), etc. In addition, it is very common to add an "ambient" term to the illumination, i.e. a constant amount of illumination — independent of surface position and orientation. The ambient light "fills in" the color on surfaces facing away from the light sources (otherwise those surfaces would be completely black). The ambient term is a cheap hack to approximate the light that bounces around in a real scene.

In this chapter we will only use very simple reflection models: Lambert for diffuse reflection and Phong for specular reflection. We'll use a point light source and ambient for illumination. Figure 2.8 shows two teapots on a square. The surfaces are Lambert diffuse with Phong specular highlights. In addition, the square has a checkerboard texture map. The scene is illuminated by a point light and a dim ambient light.

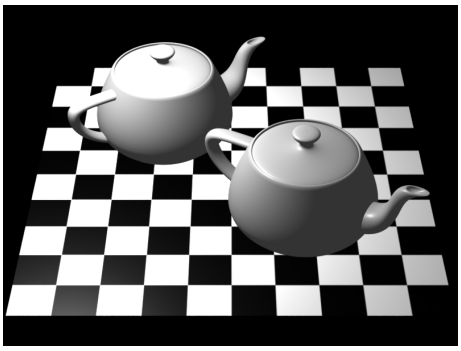


Figure 2.8: Teapots and a square showing Lambert diffuse reflection, Phong specular highlights, a checkerboard texture map, and simple illumination.

2.5 Shadow ray tracing

So far, we have only used ray tracing to determine which objects are visible in which image pixels. The first additional use of ray tracing is for shadow computation: we can determine whether a point is in shadow by tracing a ray from the point to the light source [3]. If the ray hits an opaque object along the way, the object is in shadow; if not, it is illuminated. When computing ray-object intersections for opaque shadows, we only care about hit or no hit; not the intersection point and normal. Figure 2.9(a) shows a few examples of shadow rays.

For point lights and spot lights we trace rays between the surface points and the light source position. For directional light sources, we trace parallel rays from the surface points in the direction of the light. Figure 2.9(b) shows shadows from a point light in the familiar teapot scene.

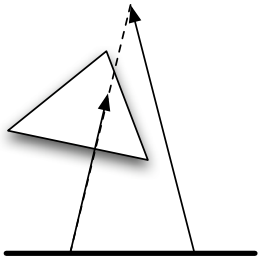
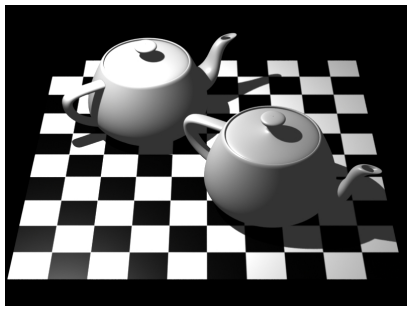
(a)   (b)

Figure 2.9: (a) Shadow rays. (b) Teapots with ray-traced shadows.

If the objects are opaque, any hit will suffice to determine shadow. But if the objects are semitransparent (as e.g. stained glass), we need to get the transmission color of all the intersected surfaces between the point and light source, and then composite the transmission colors by multiplying each color component.

2.6 Recursive ray tracing

When a ray hits a surface with specular reflection or refraction, computing the color there may require tracing more rays — called reflection rays and refraction rays, respectively. Those rays may hit other specular surfaces, causing more rays to be traced, and so on. Hence the term *recursive ray tracing*. Figure 2.10 shows a recursive "tree" of reflection rays. This technique is also known as *classical ray tracing* or *Whitted-style ray tracing* since it was introduced by Turner Whitted in 1980 [83].

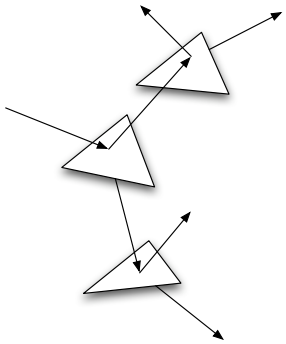


Figure 2.10: Reflection and refraction ray tree.

2.6.1 Reflection

We shoot reflection rays to compute reflection from specular surfaces such as shiny metals. To compute the reflection direction we use the *law of reflection*: the angle of reflection equals the angle of incidence. Furthermore, the reflection direction is constrained to be in the plane spanned by the incident direction and the surface normal. Figure 2.11 shows the incident direction $\vec{i}$, the surface normal $\vec{n}$, the reflection direction $\vec{r}$, as well as the angles of incidence and reflection (θ_i and θ_r).

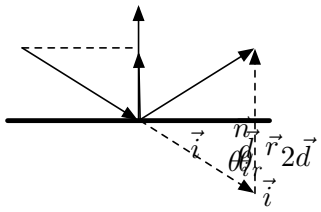


Figure 2.11: Reflection direction geometry.

As can be seen in the figure, $\vec{r} = \vec{i} + 2\vec{d}$, where $\vec{d}$ is the projection of $-\vec{i}$ onto the normal $\vec{n}$ and can be computed as

$$\vec{d} = \frac{(-\vec{i} \cdot \vec{n})\,\vec{n}}{\vec{n}^2}\,.$$

With this, the reflection direction $\vec{r}$ is

$$\vec{r} = \vec{i} + 2\vec{d} = \vec{i} - 2\frac{(\vec{i} \cdot \vec{n})\,\vec{n}}{\vec{n}^2}\,.$$

This formula does not require the vectors to be normalized. However, if we know that the normal $\vec{n}$ is normalized we can avoid the division by $\vec{n}^2$.

Figure 2.12 shows two chrome teapots with ray-traced reflections. The close-up shows how beautifully distorted such reflections can be, even on relatively simple geometry like this.

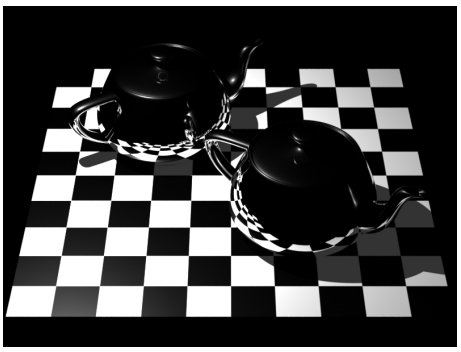
(a)   (b)

Figure 2.12: (a) Chrome teapots with reflections. (b) Close-up showing the reflections more clearly.

In this image, the reflection amount was set to 100%. To be more physically correct, one can scale the reflection amount depending on the reflection angle. The

amount of reflection can be computed using the Fresnel formulas [24, 26] or using a fast approximation by Schlick [61].

2.6.2 Refraction

Dielectric materials such as water and glass exhibit both reflection and refraction.

The refraction direction depends on the two materials' index of refraction η. The index of refraction of vacuum is 1, for air it is 1.0003, for water it is 1.33, for glass it is in the range 1.5–1.75, and for diamond it is 2.42.

Figure 2.13 shows the incident direction $\vec{i}$, surface normal $\vec{n}$, the refraction (transmission) direction $\vec{t}$, and the angles of incidence and refraction (θ_i and θ_t). η_i is the index of refraction of the material of the incident ray, and η_t is the index of refraction of the transmitting material.

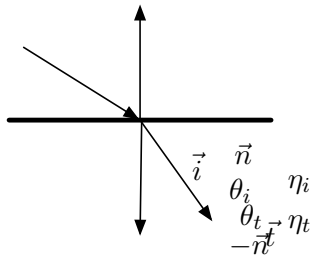


Figure 2.13: Refraction direction geometry.

The direction of the refraction ray can be computed using the *law of refraction* (often called Snell's law or Descartes' law although it was possibly known much earlier by Ibn Sahl [33]). The relationship between the incident and transmitted direction is

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t \,.$$

From this we can derive that the transmitted direction is

$$\vec{t} = -\eta\vec{i} + \left(\eta\cos\theta_i - \sqrt{1-\eta^2(1-\cos^2\theta_i)}\right)\vec{n}$$

— for $\eta = \eta_i/\eta_t$ and normalized $\vec{i}$ and $\vec{n}$. (For a derivation please see e.g. Glassner [25] or Shirley and Morley [64].)

If the quantity under the square root is negative there is no refraction. This is called *total internal reflection* and can only occur when the light transfers from a material with high index of refraction to a material with lower index of refraction.

The angle where the quantity under the square root is zero is called the *critical angle*.

The amount of reflection and refraction depends on the incident angle and index of refraction. The amount can be computed with the Fresnel formulas; these formulas are omitted here, but can be found in e.g. the books by Foley et al. [24] and Glassner [26]. A fast and very useful approximation to the Fresnel formulas was introduced by Schlick [61].

Figure 2.14 shows two glass teapots with ray-traced reflections and refractions. Notice how the glass bends the light that is refracted through it. The amount of reflection and refraction in this image is determined by Fresnel's formulas.

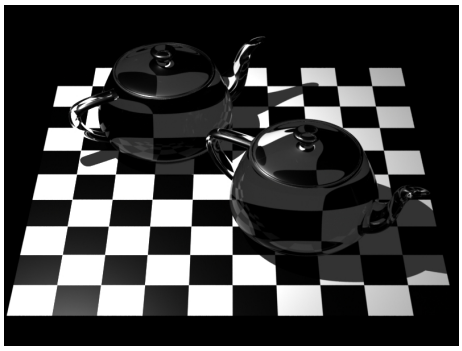


Figure 2.14: Glass teapots with reflection and refraction.

An effect that is often added to render glass and water is distance attenuation (exponential intensity fall-off according to Beer's law). This is simple to implement and is described in e.g. Shirley and Morley's book [64].

2.7 Monte Carlo ray tracing

In the previous section, all ray directions were determined deterministically. In this section we'll look at effects that are computed with *Monte Carlo ray tracing* (also known as stochastic ray tracing), i.e. ray tracing where the ray origins, directions, and/or times are computed using random numbers. Monte Carlo ray tracing is often divided into two categories: distribution ray tracing and path tracing.

2.7.1 Distribution ray tracing

Distribution ray tracing [19] shoots multiple rays from each surface point to sample area lights, glossy and diffuse reflection, and many other effects. Figure 2.15 shows a tree of reflection and refraction rays for distribution ray tracing. As the figure shows, distribution ray tracing is prone to an explosion in the number of rays after a few levels of reflection; to avoid this it is common to reduce the number of rays after a few levels of reflection. With distribution ray tracing, it is quite easy to ensure a good distribution of ray directions at reflection points, for example by stratifying the directions.

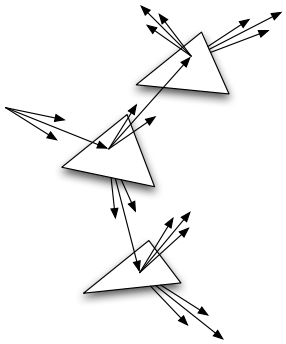


Figure 2.15: Reflection and refraction tree for distribution ray tracing.

2.7.2 Path tracing

Path tracing [37] is a variation of distribution ray tracing where only a single reflection and refraction ray is shot for each point. This avoids the explosion in the number of rays, but a simple implementation would lead to very noisy images. To compensate for that, many visibility rays are traced through each pixel. An advantage of path tracing is that since many visibility rays are shot per pixel, camera effects like depth-of-field and motion blur can be incorporated at little extra cost. On the other hand, it is harder to ensure a good distribution of reflection rays (for example through stratification) than for distribution ray tracing.

Put succinctly, distribution ray tracing shoots most rays deeper in the ray tree, while path tracing shoots most visibility rays.

2.7.3 Soft shadows

Area light sources cause soft shadows. (The region in between complete shadow and complete illumination is called the *penumbra*.) Soft shadows can be computed by shooting shadow rays to random points on the surface of the area light source.

Figure 2.16(a) shows shadow rays from three surface points to a triangular area light source; some of the rays hit an object. Figure 2.16(b) shows soft shadows in the familiar teapot scene. In this image, the light source is spherical and the soft shadow is computed with distribution ray tracing.

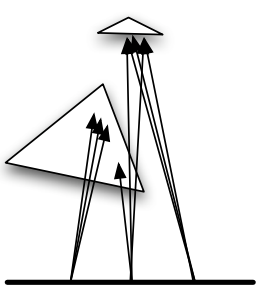
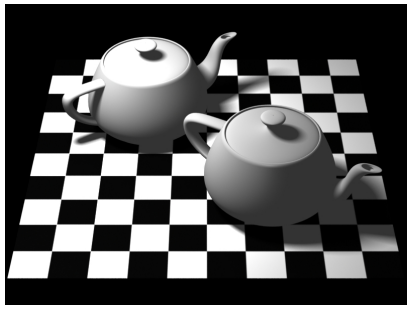
(a)   (b)

Figure 2.16: (a) Shadow rays to an area light source. (b) Teapots with soft shadows.

2.7.4 Ambient occlusion

Ambient occlusion [85, 45] can be thought of as illumination by an extremely large area light source, namely the entire hemisphere above each point. this is similar to the illumination outside on an overcast day.

Figure 2.17(a) shows ambient occlusion rays from two surface points. At the left point most of the rays hit an object, so the occlusion is high; at the right point few rays hit an object, so there is little occlusion. Figure 2.17(b) shows ambient occlusion in the teapot scene. This figure shows pure ambient occlusion; this can of course be combined with surface colors, textures, etc.

2.7.5 Glossy reflections

Glossy reflection of indirect light can be computed by shooting rays within the directions of the glossy reflection distribution. For a given incident direction and a pair of random numbers, the reflection model provides a reflection direction.

(a)  (b)

Figure 2.17: (a) Ambient occlusion rays. (b) Teapots with ambient occlusion.

Figure 2.18 shows glossy reflections in the two teapots; reflections are computed using Ward's (isotropic) glossy reflection model [81]. (There is also an anisotropic version of this shading model — it can be used to render glossy reflections from e.g. brushed metal surfaces.)

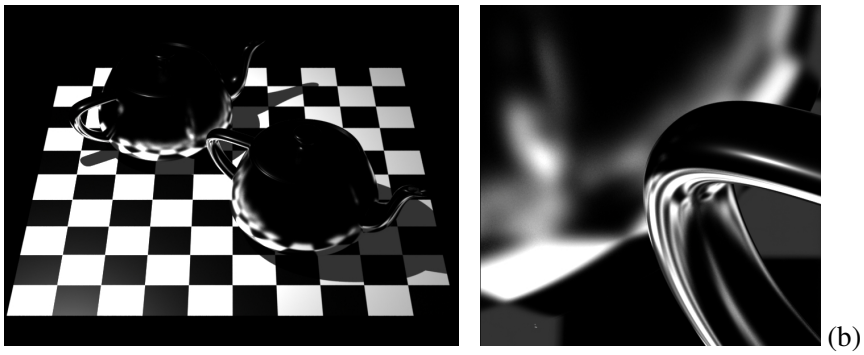
(a)  (b)

Figure 2.18: (a) Teapots with glossy reflections of direct and indirect light. (b) Close-up showing the glossy reflections more clearly.

Similarly, glossy refraction can be computed by distributing the rays around the refraction direction. This gives the appearance of slightly frosted glass.

2.7.6 Diffuse reflections

Ward et al. [82] used wide distribution ray tracing to compute indirect diffuse light. The distribution of reflection rays covers the entire hemisphere above each point, with a cosine-weighted distribution such that more rays are traced in directions toward the pole than near the equator. An example of this can be seen in figure 2.19.

In this image there is no ambient light source; any light in shadow regions is due to diffuse reflection of indirect light. Note in particular how the white checkers are reflected in the bottom of the teapots, and how the spout on the right teapot casts light onto the nearby part of the teapot body. This effect is often called *color bleeding* (although in this case the "color" is white) and can tint surfaces with the colors of nearby objects.

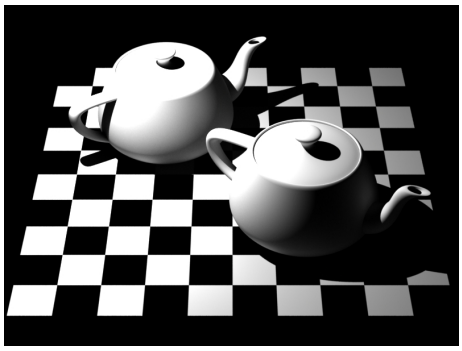


Figure 2.19: Teapots and square with diffuse reflection of direct and indirect light.

2.7.7 Depth of field

Real cameras built with lenses have a finite aperture opening. As a result, they can only focus at a particular distance, and objects that are far from that distance are blurry. (An exception is pin-hole cameras that have a nearly infinitely-small aperture opening so all distances are in focus.) In computer graphics we can simulate the finite aperture opening by tracing rays with slightly varying origins and directions as shown in figure 2.20(a). Figure 2.20(b) illustrates the depth-of-field effect: the front teapot is (mostly) in focus while the rear teapot is out of focus. Distribution ray tracing for more realistic camera models is described by Kolb et al. [42].

2.7.8 Motion blur

In real cameras the shutter has to be open a finite amount of time to capture enough light on the film or CCD chip. If an object is moving within the shutter opening time, it will be blurry in the image. To render this effect, we can shoot the rays at different times within the shutter interval. When intersection testing we move the

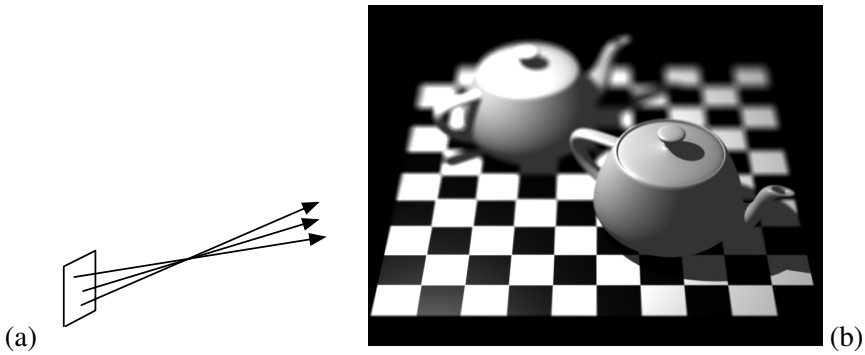


Figure 2.20: (a) Finite aperture. (b) Teapots with depth of field.

objects to the appropriate time for the ray.

Figure 2.21 shows two motion blurred teapots. The chrome teapot is moving while the diffuse teapot is both moving and rotating around its own axis. The teapots themselves are blurred, and their reflections and shadows are also blurred.

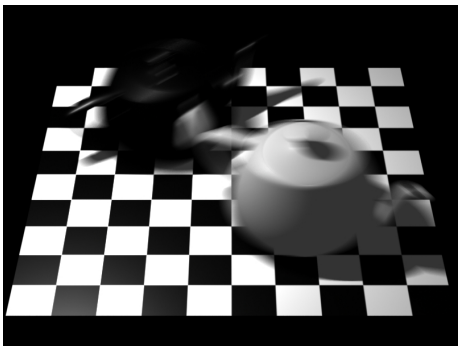


Figure 2.21: Motion-blurred teapots.

2.8 Spatial acceleration data structures

For complex scenes, it would be hopelessly inefficient to test every object for intersection with every ray. We therefore organize the objects in a hierarchy so that a large fraction of the objects can be rejected quickly.

The most important characteristics of an acceleration data structure are construction time, memory use, and ray traversal time. Depending on the application, different emphasis may be put on each of these characteristics. For rendering of

sequences of images (for example for interactive visialization or for "shot" rendering for movies) it is also desirable to choose an acceleration data structure that can be efficiently updated with incremental geometry changes, see for example Reinhard et al. [60] and Wald et al. [76].

There is a bewildering array of acceleration data structures: bounding volume hierarchies, uniform grids, hierarchical grids, BSP-trees, kd-trees, octrees, 5D origin-direction trees, bounding interval hierarchies, and so on. Here we will only describe one acceleration data structure, the bounding volume hierarchy, in detail.

2.8.1 Bounding volume hierarchy

A bounding volume hierarchy (BVH) organizes the objects and their bounding volumes into a tree [39]. The root of the tree is a bounding volume containing the entire scene. The most commonly used bounding volume is the axis-aligned box since such boxes are easy to compute and combine.

For example, the BVH for the teapot scene has five levels of bounding boxes. The top level consists of a single bounding box for the entire scene. The next levels contains the bounding boxes of the two teapots and the square. Each teapot consists of four parts: body, lid, handle, and spout. Each part has a bounding box. The teapot body consists of eight Bezier patches, each with its own bounding box. For a tessellated Bezier patch, each group of quadrilaterals can have a bounding box for efficient ray intersection testing.

The scene modeling hierarchy can be used directly, as in the teapot scene example. Another strategy is to split the geometry such that the surface areas are approximately equal in each part [27]. Smits' article [67] contains much good advice on efficient construction and traversal of bounding volume hierarchies.

When a ray needs to be intersection-tested with the objects in the scene, the first step is to check for intersection with the bounding box of the entire scene. If the ray hits the bounding box, the bounding boxes of the children are tested, and so on. When a leaf of the hierarchy is reached, the object represented by the leaf must be intersection tested.

2.8.2 Which acceleration data structure is best?

None of these acceleration data structures is consistently faster than the other. Which one is optimal for a given scene depends on the scene characteristics, and

whether the emphasis is on fast construction, fast updates, fast ray traversal, or compact memory use. For a detailed analysis please refer to Havran's Ph.D. thesis [30] and the discussions on *Ray Tracing News* [29].

2.9 Ray differentials

Even though ray differentials are a fundamental property of rays, their use for ray tracing is relatively new. They are useful for many applications including texture filtering and tessellation, as will become clear from the following chapters. A ray differential describes the differences between a ray and its — real or imaginary — "neighbor" rays. The differentials give an indication of the beam size that each ray represents, as illustrated in figure 2.22.

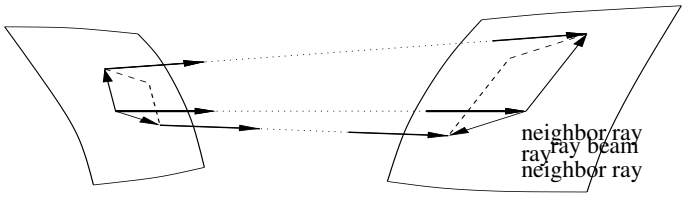


Figure 2.22: Rays and ray beam.

2.9.1 Ray propagation and specular reflection

Igehy's ray differential method [34] keeps track of ray differentials as rays are propagated and specularly reflected and refracted. The curvature at surface intersection points determines how the ray differentials and their associated beams change after specular reflection and refraction. For example, if a ray hits a highly curved, convex surface, the specularly reflected ray will have a large differential (representing highly diverging neighbor rays).

Figure 2.23 shows ray-traced specular reflections. In the left image no ray differentials are computed and the texture filter width is zero; hence the aliasing artifacts. In the right image, ray differentials are used to determine the proper texture filter size. To show the differences clearly, the resolution of the images is very low (200×200 pixels), only a single reflection ray was shot per pixel, and pixel filtering was turned off.

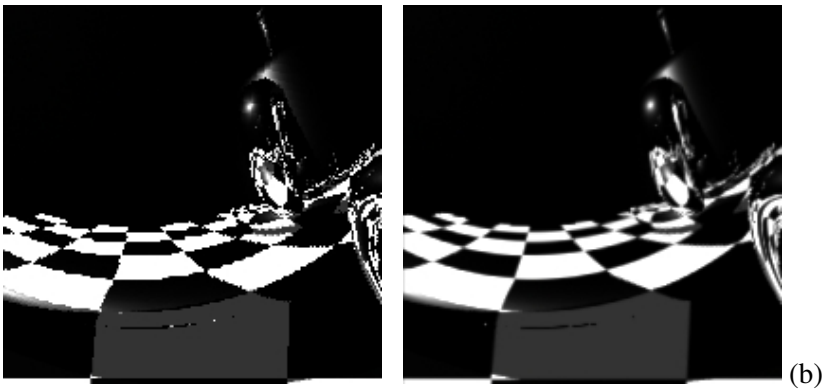
(a)  (b)

Figure 2.23: Reflections: (a) without use of ray differentials; (b) with.

2.9.2 Glossy and diffuse reflection

Suykens and Willems [73] generalized ray differentials to glossy and diffuse reflections. For distribution ray tracing of diffuse reflection or ambient occlusion, the ray differential corresponds to a fraction of the hemisphere. The more rays are traced from the same point, the smaller the subtended hemisphere fraction becomes. If the hemisphere fraction is very small, a curvature-dependent differential (as for specular reflection) becomes dominant.

2.10 Further reading

For further information about ray tracing, the best starting point is one of the excellent books dedicated to ray tracing: *An Introduction to Ray Tracing* edited by Glassner [25], and *Realistic Ray Tracing* by Shirley and Morley [64]. There are also several good books about rendering in general that include ray tracing [24, 26, 63, 56].

Eric Haines has compiled the on-line *Ray Tracing News* [29] since 1987. It contains lots of discussions about the finer points of ray tracing, new developments and insights over the years, etc.

Furthermore, there have been several SIGGRAPH courses on different aspects of ray tracing, for example the *Monte Carlo Ray Tracing* course in 2003 [35] and the *Interactive Ray Tracing* course in 2006 [65].

A series of symposia dedicated to ray tracing was recently started. The first Ray Tracing Symposium was in Salt Lake City in September 2006. It was a great

event, and the proceedings [77] are full of the latest ray tracing research results. The next symposium will be in Ulm, Germany in September 2007. We expect it and the following symposia to be as inspiring as the first.

Chapter 3

Ray tracing in complex scenes

In the previous chapters we have mainly looked at simple scenes with few light sources, relatively few simple objects, and very simplistic shading. However, in practical applications such as movie production, the scenes are much more complex:

- Thousands of light sources.
- More textures than can fit in memory.
- More geometry than can fit in memory (in tessellated form).
- Very complex, programmable shaders for displacement, illumination, and reflection. (10,000s lines of code.)

In addition, the images have to be of very high quality:

- High resolution.
- Motion blur.
- Depth of field.
- No spatial or temporal aliasing (no staircase effects, "crawlies", popping, etc.)

In this chapter we will describe various techniques to render high-quality images of such complex scenes.

3.1 Many light sources

The main expense in computing the direct illumination from a light source is typically computing the shadows. If shadow maps [59] are used, one has to render and manage a shadow map for each light source. If ray tracing is used and if we had to trace at least one shadow ray for each light source, the render times would be unacceptably long.

Fortunately, shadows from many light sources can be dealt with by sorting the light sources based on their potential illumination. Some light sources are so distant and their illumination so dim that they can be approximated very coarsely. At each surface point, the direct illumination of each light source is computed, then the lights are sorted according to illumination strength, and finally a probabilistic selection is done of which lights to compute shadows for, which ones to compute without shadows, and which ones to skip. Details can be found in the articles by Ward [80] and Shirley et al. [66] and in Shirley and Morley's book [64].

3.2 Too many textures

When the textures required to render an image exceeds the available memory, it becomes essential to read the textures from disk on demand, to read the textures only at the required resolution, and to cache the textures in memory.

3.2.1 Multiresolution textures

A texture MIP map [84] is a hierarchy of representations of a texture. Each level is a down-sampled version of the next finer level; typically a pixel at a given level is the average of four pixels at the next finer level. Figure 3.1 shows the six coarsest levels of a texture MIP map. The coarsest level consists of a single pixel, the next level consists of four pixels, and so on.

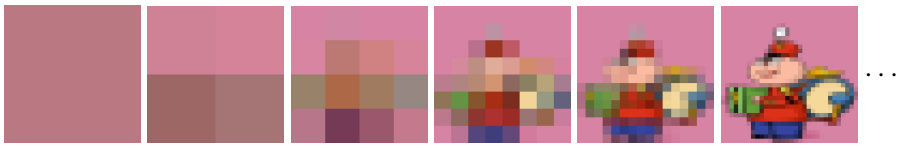


Figure 3.1: Texture MIP map.

According to Peachey [53], Hanrahan was the first to make the observation

that for directly visible geometry and fixed image resolution, the required number of texture pixels is roughly constant: A wide-angle view of a given scene contains many objects, but little texture detail is visible on each object. On the other hand, a close-up view of one of the objects shows the texture of that object at a much higher resolution, but does not show the other objects. All in all, the total number of texture pixels seen in the two images is roughly the same if the appropriate MIP map levels are chosen.

Fortunately this constant nature is true for recursive ray tracing as well — but only if we use ray differentials to determine the appropriate texture resolution at ray hit points.

3.2.2 Texture tiling

It is advantageous to tile the textures so that groups of nearby pixels are read from disk to memory together. Figure 3.2 shows three levels of a tiled texture MIP map. In this example, each tile contains 16×16 pixels. The coarsest MIP map levels (levels 0–3) can be squeezed into a single tile (not shown here). MIP map level 4 consists of a single tile. The next level has 2×2 tiles (still with 16×16 pixels in each tile). The next level has 4×4 tiles, and so on.

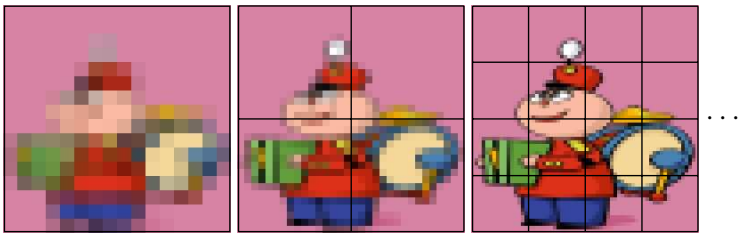


Figure 3.2: Tiled texture MIP map.

3.2.3 Multiresolution texture tile cache

Peachey [53] introduced a multiresolution texture tile caching scheme. He found that texture accesses are highly coherent for rendering of directly visible geometry, and that a cache size of 1% of the total texture size is sufficient. We have observed a similar result for ray tracing when ray differentials are used to select the appropriate MIP map level for texture lookups [17]. We choose the level where the texture pixels are approximately the same size as the ray beam cross-section. Incoherent

rays have wide ray beams, so coarse MIP map levels will be chosen. The finer MIP map levels will only be accessed by rays with narrow ray beams; fortunately those rays are coherent so the resulting texture cache lookups will be coherent as well.

3.3 Geometric complexity

This section discusses methods for rendering large, complex scenes on a single PC. Utilizing clusters of PCs for parallel speed-ups and even larger scenes is described in section 3.4.

3.3.1 Instancing

Instancing can sometimes be a great way to generate geometric complexity in a scene. For example, it is simple to render a field full of sunflowers: only a few unique sunflower shapes need to be modeled, and each sunflower instance is just represented by its model ID and transformation matrix. This saves a lot of memory compared to having to explicitly copy the sunflower geometry millions of times.

It is very simple to ray-trace instanced geometry: simply transform the ray using the inverse of the instance transformation matrix, and test for intersection with the original, untransformed object. If there is a hit, transform the hit point and normal back using the instance transformation.

3.3.2 Ray reordering and shading caching

The Toro renderer [57] reordered the rays to increase the geometric coherency. This made it possible to ray-trace scenes that are larger than the main memory of the computer. Reordering the rays requires that the image contribution of each ray is linear; this is true for real physical reflections but not generally true for the very artistic programmable shaders used in movie production.

The Razor project by Stoll et al. [71] is inspired by the REYES algorithm used for scanline rendering [18]. It shades entire grids of surface points at a time and stores the view-independent parts of the shading results. If some of the following rays hit the same surface patch, the shading results can be reused.

3.3.3 Geometric stand-ins

Wald et al. [75] demonstrated interactive rendering of complex scenes (for example a Boeing 777 airplane with 350 million triangles) using approximate stand-ins for

geometry that has not been loaded yet. The precomputation of the stand-ins is a rather lengthy process, but once it is completed, the scene can be ray-traced interactively.

3.3.4 Multiresolution tessellation

In practical applications, we have found it advantageous to tessellate curves, Bezier patches, NURBS surfaces, subdivision surfaces, and any surface with displacement instead of computing ray-surface intersections with numeric methods.

Patchification and tessellation

The surfaces are split into smaller surface patches of a manageable size — corresponding to tiling of textures. The tessellation rate for a directly visible surface patch should depend on viewing distance and surface curvature, and optionally also viewing angle. For reflections or shadows we can often use coarser tessellations. Figure 3.3 shows an example of five tessellations of a surface patch; in this example the finest tessellation rate is 14×11. The coarser levels consist of subsets of the vertices of the finest tessellation. The coarsest tessellation is simply the four corners of the patch. One can think of the various levels of tessellation as a MIP map of tessellated geometry.

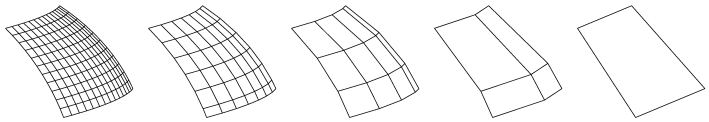


Figure 3.3: Multiresolution tessellation example for a surface patch: 14×11 quads, 7×6 quads, 4×3 quads, 2×2 quads, and 1 quad.

Multiresolution tessellation cache

Pharr and Hanrahan [55] cached tessellated geometry for displaced surfaces but did not exploit multiresolution tessellation. We tessellate surface patches on demand at the required resolution (then displace the vertices if appropriate) and store the tessellations in a cache. Since the size of the tessellations differ so much, the cache can store many more coarse tessellations than fine tessellations.

For ray intersection tests, we choose the tessellation where the quadrilaterals are approximately the same size as the ray beam cross-section. We have observed that accesses to the fine and medium tessellations are usually very coherent. The accesses to the coarse tessellations are rather incoherent, but the cache capacity for coarse tessellations is large and those tessellations are fast to recompute anyway. The fine tessellations are only needed for directly visible geometry, for specular reflections and refractions from flat surfaces, and for diffuse and ambient occlusion rays near the ray origins. For all other rays, the ray beams are wide and the medium and coarse tessellations are used.

Implementation details for our multiresultion tessellation cache can be found in Christensen et al. [17, 16].

3.4 Parallel execution

Ray tracing seems very suitable for parallel speedups: the computations for each pixel is independent of all other pixels. This has lead to the common belief that ray tracing is "embarassingly parallel". However, this is only true if the scene data fit in main memory! If the scene is larger, great care must be taken to maintain and exploit data access coherency. It pays off to arrange the execution order such that subsequent rays tend to traverse the same geometry and access the same textures, thus ensuring good cache behavior.

3.4.1 SIMD instructions

Modern CPUs have SIMD instructions (SSE on Intel, AltiVec on IBM/Motorola, 3dNow on AMD) that perform four operations in parallel. Wald et al. [79] utilized these instructions to intersection-test four rays in parallel against one triangle. This provides good speedups if the rays are coherent, and for visibility rays they reported typical speedups around 3.5.

Another way to utilize the SIMD instructions is to intersection-test one ray against four triangles in parallel. This gives good speedups if the triangles are coherent — as they are if they come from adjacent positions on a tessellated surface — and does not require the rays to be coherent. This was used by Christensen et al. [16]. Another use of the SIMD instructions is to intersection-test all three slabs of an axis-aligned bounding box in parallel.

3.4.2 Multiprocessors

Muuss [49] implemented parallel ray tracers both on multiprocessor machines with shared memory and on distributed computers on a network.

Parker et al. [52] implemented a parallel ray tracer on an SGI Origin 2000 supercomputer with 64 processors. For maximum efficiency, the scene had to fit within the local cache of each processor (4MB on the Origin), the shaders had to be very simple, and the illumination could consist of only one light source. For such relatively simple scenes they obtained interactive speeds, and despite the restrictions the interactivity was quite an accomplishment.

Multiprocessors and multi-core architectures seem highly relevant for the future of ray tracing.

3.4.3 Clusters of PCs

Wald et al. [78] and Kato [38] implemented ray tracers on clusters of standard PCs. Wald's renderer was real-time, while Kato's Kilauea was a (non-interactive) film-quality renderer. Both renderers used a brute-force approach: copy the scene to a dozen or more PCs and send ray packets to each PC for intersection testing. Kilauea could handle scenes larger than the memory of a single PC by dividing the scene geometry up into as many parts as it took to store it. Each PC computed ray-packet intersections for the parts of the scene that it contained.

3.5 Ray tracing in Pixar movies

As an example of the practical use of ray tracing in complex scenes, this section describes how ray tracing has been used in the production of recent Pixar movies.

Pixar's RenderMan renderer (PRMan) [74, 2] is based on the REYES scanline rendering algorithm [18]. The REYES algorithm renders a small image tile at a time; while rendering a tile it can ignore most of the data outside that tile — hence it can deal with very complex scenes. Traditionally, shadows have been computed with shadow maps [59] and reflections have been approximated with reflection maps [7, 28].

Although still based on the REYES algorithm, we have extended PRMan with with on-demand ray tracing [17, 16]. With PRMan's hybrid rendering algorithm there are no visibility rays, but ray tracing can be used to compute e.g. reflections,

shadows, and ambient occlusion. Thanks to the use of ray differentials and multiresolution texture and tessellation caches, very complex scenes can be ray-traced without extraneous restrictions on the shaders, displacements, etc.

The first use of ray tracing in a Pixar movie (as far as I know) was for reflections and refractions in a glass bottle in the movie *A Bug's Life*. The ray tracing was done with an external plug-in [2]. Since no algorithms were in place to deal with ray tracing of very complex scenes, only a subset of the scene geometry was ray traced.

The first wide-spread use of ray tracing at Pixar was for ambient occlusion in the movie *The Incredibles*. It is interesting to note that the shaders at Pixar are so complex and time-consuming that the time spent tracing rays is less than the time spent evaluating shaders at the ray hit points. This is perhaps the main reason for the popularity of ambient occlusion, both at Pixar and elsewhere: it is much faster to compute ambient occlusion than e.g. ray-traced reflections. This despite the fact that ambient occlusion usually requires tracing many more rays than reflections.

Ray tracing was also used in the movie *Cars*. In addition to ambient occlusion, ray tracing was used to compute realistic reflections and to compute shadows in large outdoor scenes with tiny shadow details. Figure 3.4 shows "beauty shots" of two of the characters from *Cars*. Note in particular the reflection of the eyes in the hoods.

(a)   (b)

Figure 3.4: Cars with ray-traced reflections, shadows, and ambient occlusion: (a) *Luigi*, a Fiat 500 "Topolino"; (b) *Doc Hudson*, a Hudson Hornet. (Copyright © 2006 Disney/Pixar.)

For efficiency, the reflections in the *Cars* movie were usually limited to a single level of reflection. There were only a few shots with two levels of reflection; they are close-ups of chrome parts that needed to reflect themselves multiple times to

get the right look. Figure 3.5 shows an example, the rear chrome bumper on Doc Hudson.

Figure 3.5: *Doc Hudson*'s chrome bumper with two levels of ray-traced reflection. (Copyright © 2006 Disney/Pixar.)

Figure 3.6 shows all the main characters in *Cars*. (This is a section of a poster that was originally rendered 3400 pixels wide.) This is a very complex scene with many shiny cars. The shiny cars reflect other cars, as shown in the close-ups. The image also shows ray-traced shadows and ambient occlusion.

Ray tracing is also being used in Pixar's latest movie, *Ratatouille* — mostly for ambient occlusion, reflections in pots and pans, and reflections and refractions in glasses. Figure 3.7 shows an example of ray-traced reflections and refractions in wine glasses.

Figure 3.6: The cast of *Cars* with two close-ups showing ray-traced reflections. (Copyright © 2006 Disney/Pixar.)

Figure 3.7: Ray-traced wine glasses from *Ratatouille*. (Copyright © 2007 Disney/Pixar.)

Chapter 4

A Practical Guide to Global Illumination using Photon Mapping

4.1 Photon tracing

The purpose of the photon tracing pass is to compute indirect illumination on diffuse surfaces. This is done by emitting photons from the light sources, tracing them through the scene, and storing them at diffuse surfaces.

4.1.1 Photon emission

This section describes how photons are emitted from a single light source and from multiple light sources, and describes the use of projection maps which can increase the emission efficiency considerably.

Emission from a single light source

The photons emitted from a light source should have a distribution corresponding to the distribution of emissive power of the light source. This ensures that the emitted photons carry the same flux — ie. we do not waste computational resources on photons with low power.

Photons from a diffuse point light source are emitted in uniformly distributed random directions from the point. Photons from a directional light are all emitted in the same direction, but from origins outside the scene. Photons from a diffuse

square light source are emitted from random positions on the square, with directions limited to a hemisphere. The emission directions are chosen from a cosine distribution: there is zero probability of a photon being emitted in the direction parallel to the plane of the square, and highest probability of emission is in the direction perpendicular to the square.

In general, the light source can have any shape and emission characteristics — the intensity of the emitted light varies with both origin and direction. For example, a (matte) light bulb has a nontrivial shape and the intensity of the light emitted from it varies with both position and direction. The photon emission should follow this variation, so in general, the probability of emission varies depending on the position on the surface of the light source and the direction.

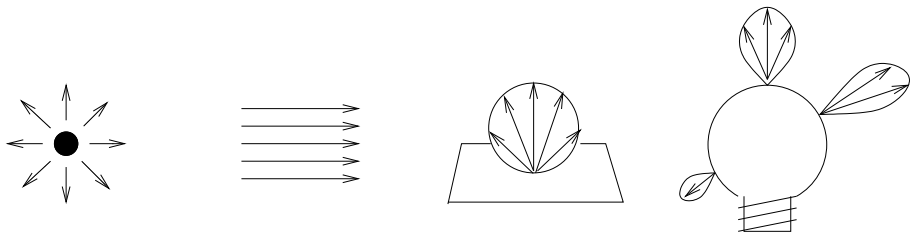


Figure 4.1: Emission from light sources: (a) point light, (b) directional light, (c) square light, (d) general light.

Figure 4.1 shows the emission from these different types of light sources.

The power ("wattage") of the light source has to be distributed among the photons emitted from it. If the power of the light is P_{light} and the number of emitted photons is n_e, the power of each emitted photon is

$$P_{photon} = \frac{P_{light}}{n_e} \,. \tag{4.1}$$

Pseudocode for a simple example of photon emission from a diffuse point light source is given in Figure 4.2.

To further reduce variation in the computed indirect illumination (during rendering), it is desirable that the photons are emitted as evenly as possible. This can for example be done with stratification [**?**] or by using low-discrepancy quasi-random sampling [**?**].

```
emit_photons_from_diffuse_point_light() {
  n_e = 0                    number of emitted photons
  while (not enough photons) {
    do {                     use simple rejection sampling to find diffuse photon direction
      x = random number between -1 and 1
      y = random number between -1 and 1
      z = random number between -1 and 1
    } while ( x^2 + y^2 + z^2 > 1 )

    d⃗ = < x, y, z >
    p⃗ = light source position
    trace photon from p⃗ in direction d⃗
    n_e = n_e + 1
  }
  scale power of stored photons with 1/n_e
}
```

Figure 4.2: Pseudocode for emission of photons from a diffuse point light

Multiple lights

If the scene contains multiple light sources, photons should be emitted from each light source. More photons should be emitted from brighter lights than from dim lights, to make the power of all emitted photons approximately even. (The information in the photon map is best utilized if the power of the stored photons is approximately even). One might worry that scenes with many light sources would require many more photons to be emitted than scenes with a single light source. Fortunately, it is not so. In a scene with many light sources, each light contributes less to the overall illumination, and typically fewer photons can be emitted from each light. If, however, only a few light sources are important one might use an importance sampling map [54] to concentrate the photons in the areas that are of interest to the observer. The tricky part about using an importance map is that we do not want to generate photons with energy levels that are too different since this will require a larger number of photons in the radiance estimate (see section 4.3) to ensure good quality.

Projection maps

In scenes with sparse geometry, many emitted photons will not hit any objects. Emitting these photons is a waste of time. To optimize the emission, *projection map*s can be used [**?**, **?**]. A projection map is simply a map of the geometry as seen from the light source. This map consists of many little cells. A cell is "on" if there is geometry in that direction, and "off" if not. For example, a projection map is a spherical projection of the scene for a point light, and it is a planar projection of the scene for a directional light. To simplify the projection it is convenient to project the bounding sphere around each object or around a cluster of objects [**?**]. This also significantly speeds up the computation of the projection map since we do not have to examine every geometric element in the scene. The most important aspect about the projection map is that it gives a conservative estimate of the directions in which it is necessary to emit photons from the light source. Had the estimate not been conservative (e.g. we could have sampled the scene with a few photons first), we could risk missing important effects, such as caustics.

The emission of photons using a projection map is very simple. One can either loop over the cells that contain objects and emit a random photon into the directions represented by the cell. This method can, however, lead to slightly biased results since the photon map can be "full" before all the cells have been visited. An alternative approach is to generate random directions and check if the cell corresponding to that direction has any objects (if not a new random direction should be tried). This approach usually works well, but it can be costly in sparse scenes. For sparse scenes it is better to generate photons randomly for the cells which have objects. A simple approach is to pick a random cell with objects and then pick a random direction for the emitted photon for that cell [**?**]. In all circumstances it is necessary to scale the energy of the stored photons based on the number of active cells in the projection map and the number of photons emitted [**?**]. This leads to a slight modification of formula 4.1:

$$P_{photon} = \frac{P_{light}}{n_e} \frac{\text{cells with objects}}{\text{total number of cells}} . \tag{4.2}$$

Another important optimization for the projection map is to identify objects with specular properties (i.e. objects that can generate caustics) [**?**]. As it will be described later, caustics are generated separately, and since specular objects often are distributed sparsely it is very beneficial to use the projection map for caustics.

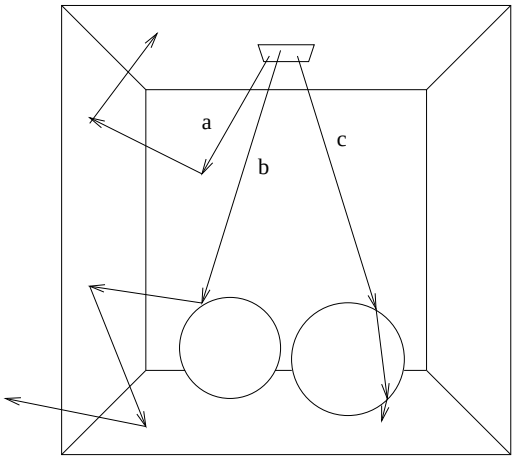


Figure 4.3: Photon paths in a scene (a "Cornell box" with a chrome sphere on left and a glass sphere on right): (a) two diffuse reflections followed by absorption, (b) a specular reflection followed by two diffuse reflections, (c) two specular transmissions followed by absorption.

4.1.2 Photon tracing

Once a photon has been emitted, it is traced through the scene using photon tracing (also known as "light ray tracing", "backward ray tracing", "forward ray tracing", and "backward path tracing"). Photon tracing works in exactly the same way as ray tracing except for the fact that photons propagate flux whereas rays gather radiance. This is an important distinction since the interaction of a photon with a material can be different than the interaction of a ray. A notable example is refraction where radiance is changed based on the relative index of refraction[**?**] — this does not happen to photons.

When a photon hits an object, it can either be reflected, transmitted, or absorbed. Whether it is reflected, transmitted, or absorbed is decided probabilistically based on the material parameters of the surface. The technique used to decide the type of interaction is known as Russian roulette [**?**] — basically we roll a dice and decide whether the photon should survive and be allowed to perform another photon tracing step.

Examples of photon paths are shown in Figure 4.3.

Reflection, transmission, or absorption?

For a simple example, we first consider a monochromatic simulation. For a reflective surface with a diffuse reflection coefficient d and specular reflection coefficient s (with $d + s \leq 1$) we use a uniformly distributed random variable $\xi \in [0, 1]$ (computed with for example `drand48()`) and make the following decision:

$$\begin{array}{lcl} \xi \in [0, d] & \longrightarrow & \text{diffuse reflection} \\ \xi \in]d, s + d] & \longrightarrow & \text{specular reflection} \\ \xi \in]s + d, 1] & \longrightarrow & \text{absorption} \end{array}$$

In this simple example, the use of Russian roulette means that we do not have to modify the power of the reflected photon — the correctness is ensured by averaging several photon interactions over time. Consider for example a surface that reflects 50% of the incoming light. With Russian roulette only half of the incoming photons will be reflected, but with full energy. For example, if you shoot 1000 photons at the surface, you can either reflect 1000 photons with half the energy or 500 photons with full energy. It can be seen that Russian roulette is a powerful technique for reducing the computational requirements for photon tracing.

With more color bands (for example RGB colors), the decision gets slightly more complicated. Consider again a surface with some diffuse reflection and some specular reflection, but this time with different reflection coefficients in the three color bands. The probabilities for specular and diffuse reflection can be based on the total energy reflected by each type of reflection or on the maximum energy reflected in any color band. If we base the decision on maximum energy, we can for example compute the probability P_d for diffuse reflection as

$$P_d = \frac{\max(d_r P_r, d_g P_g, d_b P_b)}{\max(P_r, P_g, P_b)}$$

where (d_r, d_g, d_b) are the diffuse reflection coefficients in the red, green, and blue color bands, and (P_r, P_g, P_b) are the powers of the incident photon in the same three color bands.

Similarly, the probability P_s for specular reflection is

$$P_s = \frac{\max(s_r P_r, s_g P_g, s_b P_b)}{\max(P_r, P_g, P_b)}$$

where (s_r, s_g, s_b) are the specular reflection coefficients.

The probability of absorbtion is $P_a = 1 - P_d - P_s$. With these probabilities, the decision of which type of reflection or absorption should be chosen takes the following form:

$$\begin{array}{lll} \xi \in [0, P_d] & \longrightarrow & \text{diffuse reflection} \\ \xi \in]P_d, P_s + P_d] & \longrightarrow & \text{specular reflection} \\ \xi \in]P_s + P_d, 1] & \longrightarrow & \text{absorption} \end{array}$$

The power of the reflected photon needs to be adjusted to account for the probability of survival. If, for example, specular reflection was chosen in the example above, the power P_{refl} of the reflected photon is:

$$\begin{aligned} P_{refl,r} &= P_{inc,r}\, s_r / P_s \\ P_{refl,g} &= P_{inc,g}\, s_g / P_s \\ P_{refl,b} &= P_{inc,b}\, s_b / P_s \end{aligned}$$

where P_{inc} is the power of the incident photon.

The computed probabilities again ensure us that we do not waste time emitting photons with very low power.

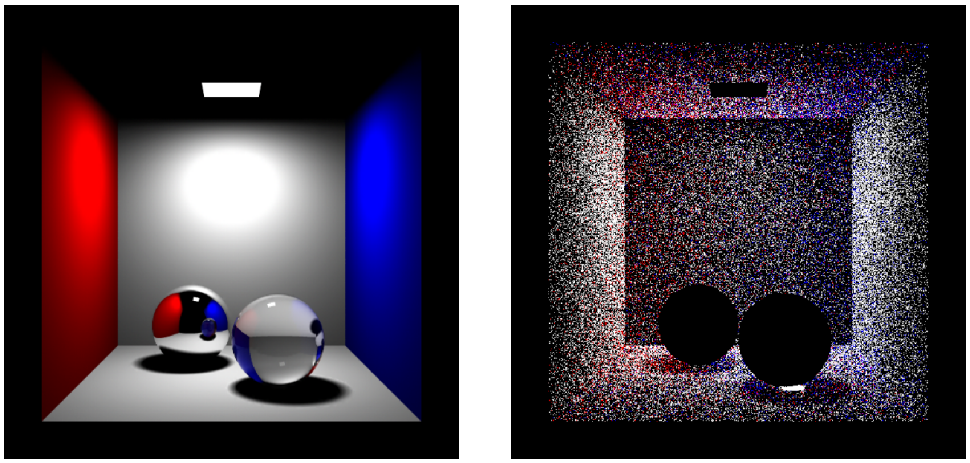
It is simple to extend the selection scheme to also handle transmission, to handle more than three color bands, and to handle other reflection types (for example glossy and directional diffuse).

Why Russian roulette?

Why do we go through this effort to decide what to do with a photon? Why not just trace new photons in the diffuse and specular directions and scale the photon energy accordingly? There are two main reasons why the use of Russian roulette is a very good idea. Firstly, we prefer photons with similar power in the photon map. This makes the radiance estimate much better using only a few photons. Secondly, if we generate, say, two photons per surface interaction then we will have 2^8 photons after 8 interactions. This means 256 photons after 8 interactions compared to 1 photon coming directly from the light source! Clearly this is not good. We want at least as many photons that have only 1–2 bounces as photons that have made 5–8 bounces. The use of Russian roulette is therefore very important in photon tracing.

There is one caveat with Russian roulette. It increases variance on the solution. Instead of using the exact values for reflection and transmission to scale the photon energy we now rely on a sampling of these values that will converge to the correct result as enough photons are used.

Details on photon tracing and Russian roulette can be found in [**?**, **?**, 26].



(a) (b)

Figure 4.4: "Cornell box" with glass and chrome spheres: (a) ray traced image (direct illumination and specular reflection and transmission), (b) the photons in the corresponding photon map.

4.1.3 Photon storing

This section describes which photon-surface interactions are stored in the photon map. It also describes in more detail the photon map data structure.

Which photon-surface interactions are stored?

Photons are only stored where they hit diffuse surfaces (or, more precisely, non-specular surfaces). The reason is that storing photons on specular surfaces does not give any useful information: the probability of having a matching incoming photon from the specular direction is zero, so if we want to render accurate specular reflections the best way is to trace a ray in the mirror direction using standard ray tracing. For all other photon-surface interactions, data is stored in a global data structure, the *photon map*. Note that each emitted photon can be stored several times along its path. Also, information about a photon is stored at the surface where it is absorbed if that surface is diffuse.

For each photon-surface interaction, the position, incoming photon power, and incident direction are stored. (For practical reasons, there is also space reserved for a flag with each set of photon data. The flag is used during sorting and look-up in

the photon map. More on this in the following.)

As an example, consider again the simple scene from Figure 4.3, a "Cornell box" with two spheres. Figure 4.4(a) shows a traditional ray traced image (direct illumination and specular reflection and transmission) of this scene. Figure 4.4(b) shows the photons in the photon map generated for this scene. The high concentration of photons under the glass sphere is caused by focusing of the photons by the glass sphere.

Data structure

Expressed in `C` the following structure is used for each photon [**?**]:

```
struct photon {
  float x,y,z;      // position
  char p[4];        // power packed as 4 chars
  char phi, theta;  // compressed incident direction
  short flag;       // flag used in kdtree
}
```

The power of the photon is represented compactly as 4 bytes using Ward's packed rgb-format [80]. If memory is not of concern one can use 3 floats to store the power in the red, green, and blue color band (or, in general, one float per color band if a spectral simulation is performed).

The incident direction is a mapping of the spherical coordinates of the photon direction to 65536 possible directions. They are computed as:

```
phi = 255 * (atan2(dy,dx)+PI) / (2*PI)
theta = 255 * acos(dx) / PI
```

where `atan2` is from the standard C library. The direction is used to compute the contribution for non-Lambertian surfaces [**?**], and for Lambertian surfaces it is used to check if a photon arrived at the front of the surface. Since the photon direction is used often during rendering it pays to have a lookup table that maps the theta, phi direction to three floats directly instead of using the formula for spherical coordinates which involves the use of the costly `cos()` and `sin()` functions.

A minor note is that the flag in the structure is a short. Only 2 bits of this flag are used (this is for the splitting plane axis in the kd-tree), and it would be possible to use just one byte for the flag. However for alignment reasons it is preferable to

have a 20 byte photon rather than a 19 byte photon — on some architectures it is even a necessity since the float-value in subsequent photons must be aligned on a 4 byte address.

We might be able to compress the information more by using the fact that we know the cube in which the photon is located. The position is, however, used very often when the photons are processed and by using standard float we avoid the overhead involved in extracting the true position from a specialized format.

During the photon tracing pass the photon map is arranged as a flat array of photons. For efficiency reasons this array is re-organized into a balanced kdtree before rendering as explained in section 4.2.

4.1.4 Extension to participating media

Up to this point, all photon interactions have been assumed to happen at object surfaces; all volumes were implicitly assumed to not affect the photons. However, it is simple to extend the photon map method to handle *participating media*, i.e. volumes that participate in the light transport. In scenes with participating media, the photons are stored within the media in a seperate *volume photon map* [**?**].

Photon emission, tracing, and storage

Photons can be emitted from volumes as well as from surfaces and points. For example, the light from a candle flame can be simulated by emitting photons from a flame-shaped volume.

When a photon travels through a participating medium, it has a certain probability of being scattered or absorped in the medium. The probability depends on the density of the medium and on the distance the photon travels through the medium: the denser the medium, the shorter the average distance before a photon interaction happens. Photons are stored at the positions where a scattering event happens. The exception is photons that come directly from the light source since direct illumination is evaluated using ray tracing. This separation was introduced in [**?**] and it allows us to compute the in-scattered radiance in a medium simply by a lookup in the photon map.

As an example, consider a glass sphere in fog illuminated by directional light. Figure 4.5(a) shows a schematic diagram of the photon paths as photons are being focused by refraction in the glass sphere. Figure 4.5(b) shows the caustic photons stored in the photon map.

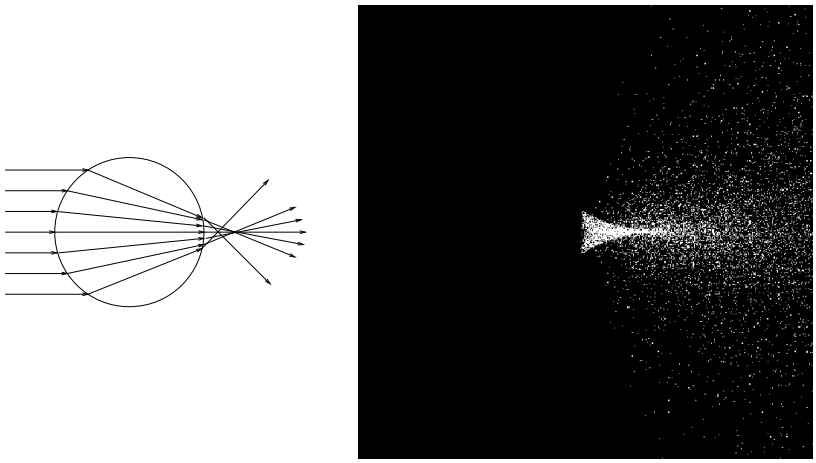


Figure 4.5: Sphere in fog: (a) schematic diagram of light paths, (b) the caustic photons in the photon map.

Multiple scattering, anisotropic scattering, and non-homogeneous media

The simple example above only shows the photon interaction in the fog after refraction by the glass sphere, and the photon paths are terminated at the first scattering event. General multiple scattering is simulated simply by letting the photons scatter everywhere and continuously after the first interaction. The path can be terminated using Russian roulette.

The fog in the example has uniform density, but it is not difficult to handle media with nonuniform density (aka. nonhomogeneous media), since we use ray marching to integrate the properties of the medium. A simple ray marcher works by dividing the medium into little steps [**?**]. The accumulated density (integrated extinction coefficient) is updated at each step, and based on a precomputed probability it is determined whether the photon should be absorbed, scattered, or whether another step is necessary.

For more complicated examples of scattering in participating media, including anisotropic and nonhomogeneous media and complex geometry, see [**?**].

4.1.5 Three photon maps

For efficiency reasons, it pays off to divide the stored photons into three photon maps:

Caustic photon map: contains photons that have been through at least one specular reflection before hitting a diffuse surface: LS^+D.

Global photon map: an approximate representation of the global illumination solution for the scene for all diffuse surfaces: $L\{S|D|V\}^*D$

Volume photon map: indirect illumination of a participating medium: $L\{S|D|V\}^+V$.

Here, we used the grammar from [**?**] to describe the photon paths: L means emission from the light source, S is specular reflection or transmission, D is diffuse (ie. non-specular) reflection or transmission, and V is volume scattering. The notation $\{x|y|z\}$ means "one of x, y, or z", x^+ means one or several repeats of x, and x^* means zero or several repeats of x.

The reason for keeping three separate photon maps will become clear in section 4.4. A separate photon tracing pass is performed for the caustic photon map since it should be of high quality and therefore often needs more photons than the global photon map and the volume photon map.

The construction of the photon maps is most easily achieved by using two separate photon tracing steps in order to build the caustics photon map and the global photon map (including the volume photon map). This is illustrated in Figure 4.6 for a simple test scene with a glass sphere and 2 diffuse walls. Figure 4.6(a) shows the construction of the caustics photon map with a dense distribution of photons, and Figure 4.6(b) shows the construction of the global photon map with a more coarse distribution of photons.

4.2 Preparing the photon map for rendering

Photons are only generated during the photon tracing pass — in the rendering pass the photon map is a static data structure that is used to compute estimates of the incoming flux and the reflected radiance at many points in the scene. To do this it is necessary to locate the nearest photons in the photon map. This is an operation

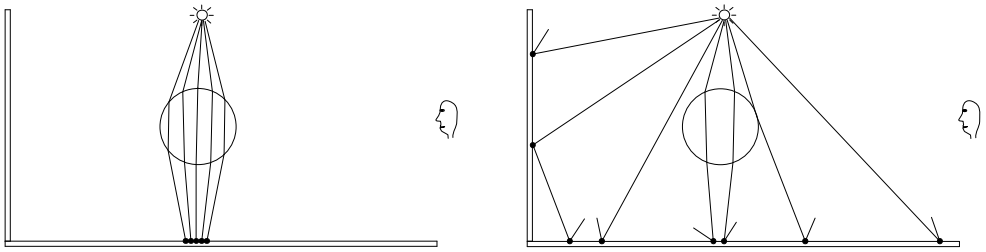


Figure 4.6: Building (a) the caustics photon map and (b) the global photon map.

that is done extremely often, and it is therefore a good idea to optimize the representation of the photon map before the rendering pass such that finding the nearest photons is as fast as possible.

First, we need to select a good data structure for representing the photon map. The data structure should be compact and at the same time allow for fast nearest neighbor searching. It should also be able to handle highly non-uniform distributions — this is very often the case in the caustics photon map. A natural candidate that handles these requirements is *a balanced kd-tree* [**?**]. Examples of using a balanced versus an unbalanced kd-tree can be found in [**?**].

4.2.1 The balanced kd-tree

The time it takes to locate one photon in a balanced kd-tree has a worst time performance of $O(\log N)$, where N is the number of photons in the tree. Since the photon map is created by tracing photons randomly through a model one might think that a dynamically built kd-tree would be quite well balanced already. However, the fact that the generation of the photons at the light source is based on the projection map combined with the fact that models often contain highly directional reflectance models easily results in a skewed tree. Since the tree is created only once and used many times during rendering it is quite natural to consider balancing the tree. Another argument that is perhaps even more important is the fact that a balanced kd-tree can be represented using a heap-like data-structure [**?**] which means that explicitly storing the pointers to the sub-trees at each node is no longer necessary. (Array element 1 is the tree root, and element i has element $2i$ as left child and element $2i + 1$ as right child.) This can lead to considerable savings in

```
kdtree *balance( points ) {
  Find the cube surrounding the points
  Select dimension dim in which the cube is largest
  Find median of the points in dim
  s1 = all points below median
  s2 = all points above median
  node = median
  node.left = balance( s1 )
  node.right = balance( s2 )
  return node
}
```

Figure 4.7: Pseudocode for balancing the photon map

memory when a large number of photons is used.

4.2.2 Balancing

Balancing a kd-tree is similar to balancing a binary tree. The main difference is the choice at each node of a splitting dimension. When a splitting dimension of a set is selected, the median of the points in that dimension is chosen as the root node of the tree representing the set and the left and right subtrees are constructed from the two sets separated by the median point. The choice of a splitting dimension is based on the distribution of points within the set. One might use either the variance or the maximum distance between the points as a criterion. We prefer a choice based upon maximum distance since it can be computed very efficiently (even though a choice based upon variance might be slightly better). The splitting dimension is thus chosen as the one which has the largest maximum distance between the points.

Figure 4.7 contains a pseudocode outline for the balancing algorithm [**?**].

To speed up the balancing process it is convenient to use an array of pointers to the photons. In this way only pointers needs to be shuffled during the median search. An efficient median search algorithm can be found in most textbooks on algorithms — see for example [**?**] or [**?**].

The complexity of the balancing algorithm is $O(N \log N)$ where N is the number of photons in the photon map. In practice, this step only takes a few seconds even for several million photons.

4.3 The radiance estimate

A fundamental component of the photon map method is the ability to compute radiance estimates at any non-specular surface point in any given direction.

4.3.1 Radiance estimate at a surface

The photon map can be seen as a representation of the incoming flux; to compute radiance we need to integrate this information. This can be done using the expression for reflected radiance:

$$L_r(x,\vec{\omega}) = \int_{\Omega_x} f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}') |\vec{n}_x \cdot \vec{\omega}'| \, d\omega_i' \,, \tag{4.3}$$

where L_r is the reflected radiance at x in direction $\vec{\omega}$. Ω_x is the (hemi)sphere of incoming directions, f_r is the BRDF (bidirectional reflectance distribution function) [**?**] at x and L_i is the incoming radiance. To evaluate this integral we need information about the incoming radiance. Since the photon map provides information about the incoming flux we need to rewrite this term. This can be done using the relationship between radiance and flux:

$$L_i(x,\vec{\omega}') = \frac{d^2\Phi_i(x,\vec{\omega}')}{\cos\theta_i \, d\omega_i' \, dA_i} \,, \tag{4.4}$$

and we can rewrite the integral as

$$\begin{aligned} L_r(x,\vec{\omega}) &= \int_{\Omega_x} f_r(x,\vec{\omega}',\vec{\omega}) \frac{d^2\Phi_i(x,\vec{\omega}')}{\cos\theta_i \, d\omega_i' \, dA_i} |\vec{n}_x \cdot \vec{\omega}'| \, d\omega_i' \\ &= \int_{\Omega_x} f_r(x,\vec{\omega}',\vec{\omega}) \frac{d^2\Phi_i(x,\vec{\omega}')}{dA_i} \,. \end{aligned} \tag{4.5}$$

The incoming flux Φ_i is approximated using the photon map by locating the n photons that has the shortest distance to x. Each photon p has the power $\Delta\Phi_p(\vec{\omega}_p)$ and by assuming that the photons intersects the surface at x we obtain

$$L_r(x,\vec{\omega}) \approx \sum_{p=1}^{n} f_r(x,\vec{\omega}_p,\vec{\omega}) \frac{\Delta\Phi_p(x,\vec{\omega}_p)}{\Delta A} \,. \tag{4.6}$$

The procedure can be imagined as expanding a sphere around x until it contains n photons (see Figure 4.8) and then using these n photons to estimate the radiance.

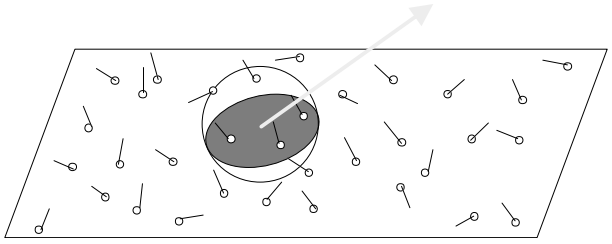


Figure 4.8: Radiance is estimated using the nearest photons in the photon map.

Equation 4.6 still contains ΔA which is related to the density of the photons around x. By assuming that the surface is locally flat around x we can compute this area by projecting the sphere onto the surface and use the area of the resulting circle. This is indicated by the hatched area in Figure 4.8 and equals:

$$\Delta A = \pi r^2 \,, \tag{4.7}$$

where r is the radius of the sphere – ie. the largest distance between x and each of the photons.

This results in the following equation for computing reflected radiance at a surface using the photon map:

$$L_r(x, \vec{\omega}) \approx \frac{1}{\pi r^2} \sum_{p=1}^{N} f_r(x, \vec{\omega}_p, \vec{\omega}) \Delta\Phi_p(x, \vec{\omega}_p) \,. \tag{4.8}$$

This estimate is based on many assumptions and the accuracy depends on the number of photons used in the photon map and in the formula. Since a sphere is used to locate the photons one might easily include wrong photons in the estimate in particular in corners and at sharp edges of objects. Edges and corners also causes the area estimate to be wrong. The size of those regions in which these errors occur depends largely on the number of photons in the photon map and in the estimate. As more photons are used in the estimate and in the photon map, formula 4.8 becomes more accurate. If we ignore the error due to limited accuracy of the representation of the position, direction and flux, then we can go to the limit and increase the number of photons to infinity. This gives the following interesting

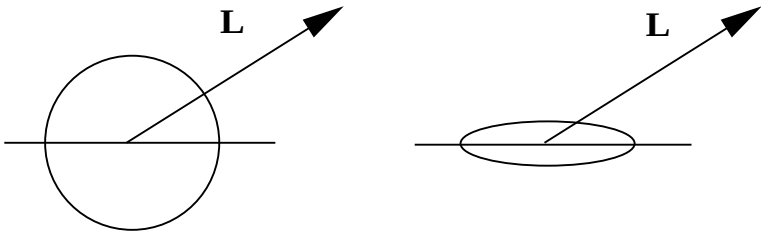


Figure 4.9: Using a sphere (left) and using a disc (right) to locate the photons.

result where N is the number of photons in the photon map:

$$\lim_{N\to\infty} \frac{1}{\pi r^2} \sum_{p=1}^{\lfloor N^\alpha \rfloor} f_r(x, \vec{\omega}_p, \vec{\omega}) \Delta\Phi_p(x, \vec{\omega}_p) = L_r(x, \vec{\omega}) \text{ for } \alpha \in]0, 1[\,. \tag{4.9}$$

This formulation applies to all points x located on a locally flat part of a surface for which the BRDF, does not contain the Dirac delta function (this excludes perfect specular reflection). The principle in equation 4.9 is that not only will an infinite amount of photons be used to represent the flux within the model but an infinite amount of photons will also be used to estimate radiance and the photons in the estimate will be located within an infinitesimal sphere. The different degrees of infinity are controlled by the term N^α where $\alpha \in]0, 1[$. This ensures that the number of photons in the estimate will be infinitely fewer than the number of photons in the photon map.

Equation 4.9 means that we can obtain arbitrarily good radiance estimates by just using enough photons! In finite element based approaches it is more complicated to obtain arbitrary accuracy since the error depends on the resolution of the mesh, the resolution of the directional representation of radiance and the accuracy of the light simulation.

Figure 4.8 shows how locating the nearest photons is similar to expanding a sphere around x and using the photons within this sphere. It is possible to use other volumes than the sphere in this process. One might use a cube instead, a cylinder or perhaps a disc. This could be useful to either obtain an algorithm that is faster at locating the nearest photons or perhaps more accurate in the selection of photons. If a different volume is used then ΔA in equation 4.6 should be replaced by the area of the intersection between the volume and the tangent plane touching

the surface at x. The sphere has the obvious advantage that the projected area and the distance computations are very simple and thus efficiently computed. A more accurate volume can be obtained by modifying the sphere into a disc (ellipsoid) by compressing the sphere in the direction of the surface normal at x (shown in Figure 4.9) [**?**]. The advantage of using a disc would be that fewer "false photons" are used in the estimate at edges and in corners. This modification works pretty well at the edges in a room, for instance, since it prevents photons on the walls to leak down to the floor. One issue that still occurs, however, is that the area estimate might be wrong or photons may leak into areas where they do not belong. This problem is handled primarily by the use of filtering.

4.3.2 Filtering

If the number of photons in the photon map is too low, the radiance estimates becomes blurry at the edges. This artifact can be pleasing when the photon map is used to estimate indirect illumination for a distribution ray tracer (see section 4.4 and Figure 4.15) but it is unwanted in situations where the radiance estimate represents caustics. Caustics often have sharp edges and it would be nice to preserve these edges without requiring too many photons.

To reduce the amount of blur at edges, the radiance estimate is filtered. The idea behind filtering is to increase the weight of photons that are close to the point of interest, x. Since we use a sphere to locate the photons it would be natural to assume that the filters should be three-dimensional. However, photons are stored at surfaces which are two-dimensional. The area estimate is also based on the assumption that photons are located on a surface. We therefore need a 2d-filter (similar to image filters) which is normalized over the region defined by the photons.

The idea of filtering caustics is not new. Collins [**?**] has examined several filters in combination with illumination maps. The filters we have examined are two radially symmetric filters: the cone filter and the Gaussian filter [**?**], and the specialized differential filter introduced in [**?**]. For examples of more advanced filters see Myszkowski et al. [**?**].

The cone filter

The cone-filter [**?**] assigns a weight, w_{pc}, to each photon based on the distance, d_p, between x and the photon p. This weight is:

$$w_{pc} = 1 - \frac{d_p}{k\,r}\,, \tag{4.10}$$

where $k \geq 1$ is a filter constant characterizing the filter and r is the maximum distance. The normalization of the filter based on a 2d-distribution of the photons is $1 - \frac{2}{3k}$ and the filtered radiance estimate becomes:

$$L_r(x,\vec{\omega}) \approx \frac{\sum_{p=1}^{N} f_r(x,\vec{\omega}_p,\vec{\omega})\Delta\Phi_p(x,\vec{\omega}_p)w_{pc}}{(1-\frac{2}{3k})\pi r^2}\,. \tag{4.11}$$

The Gaussian filter

The Gaussian filter [**?**] has previously been reported to give good results when filtering caustics in illumination maps [**?**]. It is easy to use the Gaussian filter with the photon map since we do not need to warp the filter to some surface function. Instead we use the assumption about the locally flat surfaces and we can use a simple image based Gaussian filter [**?**] and the weight w_{pg} of each photon becomes

$$w_{pg} = \alpha\left[1 - \frac{1 - e^{-\beta\frac{d_p^2}{2r^2}}}{1-e^{-\beta}}\right]\,, \tag{4.12}$$

where d_p is the distance between the photon p and x and $\alpha = 0.918$ and $\beta = 1.953$ (see [**?**] for details). This filter is normalized and the only change to equation 4.8 is that each photon contribution is multiplied by w_{pg}:

$$L_r(x,\vec{\omega}) \approx \sum_{p=1}^{N} f_r(x,\vec{\omega}_p,\vec{\omega})\Delta\Phi_p(x,\vec{\omega}_p)w_{pg}\,. \tag{4.13}$$

Differential checking

In [**?**] it was suggested to use a filter based on differential checking. The idea is to detect regions near edges in the estimation process and use less photons in these

regions. In this way we might get some noise in the estimate but that is often preferable to blurry edges.

The radiance estimate is modified based on the following observation: when adding photons to the estimate, near an edge the changes of the estimate will be monotonic. That is, if we are just outside a caustic and we begin to add photons to the estimate (by increasing the size of the sphere centered at x that contains the photons), then it can be observed that the value of the estimate is increasing as we add more photons; and vice versa when we are inside the caustic. Based on this observation, differential checking can be added to the estimate — we stop adding photons and use the estimate available if we observe that the estimate is either constantly increasing or decreasing as more photons are added.

4.3.3 The radiance estimate in a participating medium

For the radiance estimate presented so far we have assumed that the photons are located on a surface. For photons in a participating medium the formula changes to [**?**]:

$$\begin{aligned} L_i(x,\vec{\omega}) &= \int_\Omega f(x,\vec{\omega}',\vec{\omega})\, L(x,\vec{\omega}')\, d\omega' \\ &= \int_\Omega f(x,\vec{\omega}',\vec{\omega})\, \frac{d^2\Phi(x,\vec{\omega}')}{\sigma_s(x)\, d\omega'\, dV}\, d\omega' \\ &= \frac{1}{\sigma_s(x)} \int_\Omega f(x,\vec{\omega}',\vec{\omega})\, \frac{d^2\Phi(x,\vec{\omega}')}{dV} \\ &\approx \frac{1}{\sigma_s(x)} \sum_{p=1}^{n} f(x,\vec{\omega}_p',\vec{\omega})\, \frac{\Delta\Phi_p(x,\vec{\omega}_p')}{\frac{4}{3}\pi r^3}\,, \end{aligned} \tag{4.14}$$

where L_i is the in-scattered radiance, and the volume $dV = \frac{4}{3}\pi r^3$ is the volume of the sphere containing the photons. $\sigma_s(x)$ is the scattering coefficient at x and f is the phase-function.

4.3.4 Locating the nearest photons

Efficiently locating the nearest photons is critical for good performance of the photon map algorithm. In scenes with caustics, multiple diffuse reflections, and/or participating media there can be a large number of photon map queries.

Fortunately the simplicity of the kd-tree permits us to implement a simple but quite efficient search algorithm. This search algorithm is a straight forward exten-

sion of standard search algorithms for binary trees [**?**, **?**, **?**]. It is also related to range searching where kd-trees are commonly used as they have optimal storage and good performance [**?**]. The nearest neighbors query for kd-trees has been described extensively in several publications by Bentley et al. including [**?**, **?**, **?**, **?**]. More recent publications include [**?**, **?**]. Some of these papers go beyond our description of a nearest neighbors query by adding modifications and extensions to the kd-tree to further reduce the cost of searching. We do not implement these extensions because we want to maintain the low storage overhead of the kd-tree as this is an important aspect of the photon map.

Locating the nearest neighbors in a kd-tree is similar to range searching [**?**] in the sense that we want to locate photons within a given volume. For the photon map it makes sense to restrict the size of the initial search range since the contribution from a fixed number of photons becomes small for large regions. This simple observation is particularly important for caustics since they often are concentrated in a small region. A search algorithm that does not limit the search range will be slow in such situations since a large part of the kd-tree will be visited for regions with a sparse number of photons.

A generic nearest neighbors search algorithm begins at the root of the kd-tree, and adds photons to a list if they are within a certain distance. For the n nearest neighbors the list is sorted such that the photon that is furthest away can be deleted if the list contains n photons and a new closer photon is found. Instead of naive sorting of the full list it is better to use a max-heap [**?**, **?**, **?**]. A max-heap (also known as a priority queue) is a very efficient way of keeping track of the element that is furthest away from the point of interest. When the max-heap is full, we can use the distance d to the root element (ie. the photon that is furthest away) to adjust the range of the query. Thus we skip parts of the kd-tree that are further away than d.

Another simple observation is that we can use squared distances — we do not need the real distance. This removes the need of a square root calculation per distance check.

The pseudo-code for the search algorithm is given in Figure 4.10. A simple implementation of this routine is available with source code at [**?**].

For this search algorithm it is necessary to provide an initial maximum search radius. A well-chosen radius allows for good pruning of the search reducing the number of photons tested. A maximum radius that is too low will on the other hand introduce noise in the photon map estimates. The radius can be chosen based on an

```
given the photon map, a position x and a max search distance d^2
this recursive function returns a heap h with the nearest photons.
Call with locate_photons(1) to initiate search at the root of the kd-tree

locate_photons( p ) {
  if ( 2p + 1 < number of photons ) {
    examine child nodes
                Compute distance to plane (just a subtract)
    δ = signed distance to splitting plane of node n
    if (δ < 0) {
                We are left of the plane - search left subtree first
      locate_photons( 2p )
      if ( δ^2 < d^2 )
        locate_photons( 2p + 1 )      check right subtree
    } else {
                We are right of the plane - search right subtree first
      locate_photons( 2p + 1 )
      if ( δ^2 < d^2 )
        locate_photons( 2p )      check left subtree
    }
  }
            Compute true squared distance to photon
  δ^2 = squared distance from photon p to x
  if ( δ^2 < d^2 ) {          Check if the photon is close enough?
    insert photon into max heap h
              Adjust maximum distance to prune the search
    d^2 = squared distance to photon in root node of h
  }
}
```

Figure 4.10: Pseudocode for locating the nearest photons in the photon map

error metric or the size of the scene. The error metric could for example take the average energy of the stored photons into account and compute a maximum radius from that assuming some allowed error in the radiance estimate.

A few extra optimizations can be added to this routine, for example a delayed construction of the max heap to the time when the number of photons needed has been found. This is particularly useful when the requested number of photons is large.

Nathan Kopp has implemented a slightly different optimization in an extended version of the Persistence Of Vision Ray Tracer (POV) called `MegaPov` (avail-

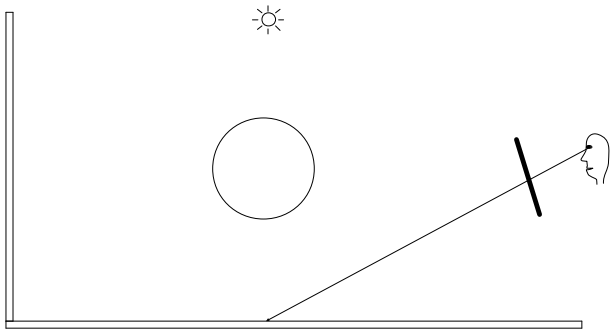


Figure 4.11: Tracing a ray through a pixel.

able at [**?**]). In his implementation the initial maximum search radius is set to a very low value. If this value turns out to be too low, another search is performed with a higher maximum radius. He reports good timings and results from this technique [**?**].

Another change to the search routine is to use the disc check as described earlier. This is useful to avoid incorrect color bleeding and particularly helpful if the gathering step is not used and the photons are visualized directly.

4.4 Rendering

Given the photon map and the ability to compute a radiance estimate from it, we can proceed with the rendering pass. The photon map is view independent, and therefore a single photon map constructured for an environment can be utilized to render the scene from any desired view. There are several different ways in which the photon map can be visualized. A very fast visualization technique has been presented by Myszkowski et al. [**?**, **?**] where photons are used to compute radiosity values at the vertices of a mesh.

In this note we will focus on the full global illumination approach as presented in [**?**]. Initially we will ignore the presence of participating media; at the end of the note we have added some notes for this case.

The final image is rendered using distribution ray tracing in which the pixel radiance is computed by averaging a number of sample estimates. Each sample consists of tracing a ray from the eye through a pixel into the scene (see Figure 4.11).

The radiance returned by each ray equals the outgoing radiance in the direction of the ray leaving the point of intersection at the first surface intersected by the ray. The outgoing radiance, L_o, is the sum of the emitted, L_e, and the reflected radiance

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + L_r(x, \vec{\omega}) \,, \tag{4.15}$$

where the reflected radiance, L_r, is computed by integrating the contribution from the incoming radiance, L_i,

$$L_r(x, \vec{\omega}) = \int_{\Omega_x} f_r(x, \vec{\omega}', \vec{\omega}) L_i(x, \vec{\omega}') \cos\theta_i \, d\omega_i' \,, \tag{4.16}$$

where f_r is the bidirectional reflectance distribution function (BRDF), and Ω_x is the set of incoming directions around x.

L_r can be computed using Monte Carlo integration techniques like path tracing and distribution ray tracing. These methods are very costly in terms of rendering time and a more efficient approach can be obtained by using the photon map in combination with our knowledge of the BRDF and the incoming radiance.

The BRDF is separated into a sum of two components: A specular/glossy, $f_{r,s}$, and a diffuse, $f_{r,d}$

$$f_r(x, \vec{\omega}', \vec{\omega}) = f_{r,s}(x, \vec{\omega}', \vec{\omega}) + f_{r,d}(x, \vec{\omega}', \vec{\omega}) \,. \tag{4.17}$$

The incoming radiance is classified using three components:

- $L_{i,l}(x, \vec{\omega}')$ is direct illumination by light coming from the light sources.
- $L_{i,c}(x, \vec{\omega}')$ is caustics — indirect illumination from the light sources via specular reflection or transmission.
- $L_{i,d}(x, \vec{\omega}')$ is indirect illumination from the light sources which has been reflected diffusely at least once.

The incoming radiance is the sum of these three components:

$$L_i(x, \vec{\omega}') = L_{i,l}(x, \vec{\omega}') + L_{i,c}(x, \vec{\omega}') + L_{i,d}(x, \vec{\omega}') \,. \tag{4.18}$$

By using the classifications of the BRDF and the incoming radiance we can

split the expression for reflected radiance into a sum of four integrals:

$$\begin{aligned} L_r(x,\vec{\omega}) &= \int_{\Omega_x} f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}') \cos\theta_i \, d\omega_i' \\ &= \int_{\Omega_x} f_r(x,\vec{\omega}',\vec{\omega}) L_{i,l}(x,\vec{\omega}') \cos\theta_i \, d\omega_i' + \\ &\quad \int_{\Omega_x} f_{r,s}(x,\vec{\omega}',\vec{\omega}) (L_{i,c}(x,\vec{\omega}') + L_{i,d}(x,\vec{\omega}')) \cos\theta_i \, d\omega_i' + \\ &\quad \int_{\Omega_x} f_{r,d}(x,\vec{\omega}',\vec{\omega}) L_{i,c}(x,\vec{\omega}') \cos\theta_i \, d\omega_i' + \\ &\quad \int_{\Omega_x} f_{r,d}(x,\vec{\omega}',\vec{\omega}) L_{i,d}(x,\vec{\omega}') \cos\theta_i \, d\omega_i' \,. \end{aligned} \tag{4.19}$$

This is the equation used whenever we need to compute the reflected radiance from a surface. In the following sections we discuss the evaluation of each of the integrals in the equation in more detail. We distinguish between two different situations: an accurate and an approximate.

The accurate computation is used if the surface is seen directly by the eye or perhaps via a few specular reflections. It is also used if the distance between the ray origin and the intersection point is below a small threshold value — to eliminate potential inaccurate color bleeding effects in corners. The approximate evaluation is used if the ray intersecting the surface has been reflected diffusely since it left the eye or if the ray contributes only little to the pixel radiance.

4.4.1 Direct illumination

Direct illumination is given by the term

$$\int_{\Omega_x} f_r(x,\vec{\omega}',\vec{\omega}) L_{i,l}(x,\vec{\omega}') \cos\theta_i \, d\omega_i' \,,$$

and it represents the contribution to the reflected radiance due to direct illumination. This term is often the most important part of the reflected radiance and it has to be computed accurately since it determines light effects to which the eye is highly sensitive such as shadow edges.

Computing the contribution from the light sources is quite simple in ray tracing based methods. At the point of interest shadow rays are sent towards the light sources to test for possible occlusion by objects. This is illustrated in Figure 4.12. If a shadow ray does not hit an object the contribution from the light source is

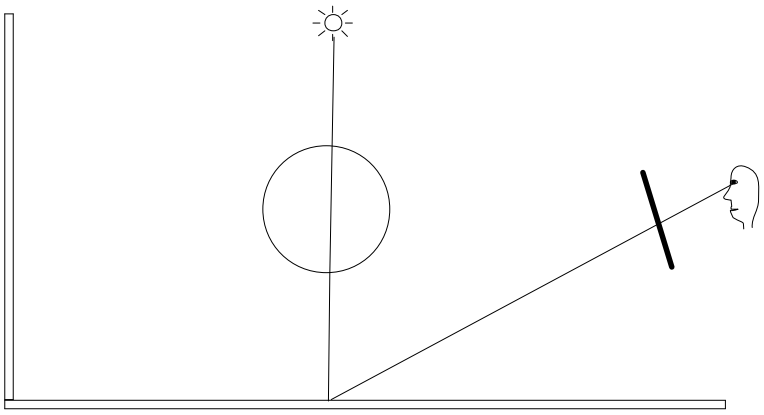


Figure 4.12: Accurate evaluation of the direct illumination.

included in the integral otherwise it is neglected. For large area light sources several shadow rays are used to properly integrate the contribution and correctly render penumbra regions. This strategy can however be very costly since a large number of shadow rays is needed to properly integrate the direct illumination.

Using a derivative of the photon map method we can compute shadows more efficiently using shadow photons [**?**]. This approach can lead to considerable speedups in scenes with large penumbra-regions that are normally very costly to render using standard ray tracing. The approach is stochastic though, so it might miss shadows from small objects in case these aren't intersected by any photons. This is a problem with all techniques that use stochastic evaluation of visibility.

The approximate evaluation is simply the radiance estimate obtained from the global photon map (no shadow rays or light source evaluations are used). This is seen in Figure 4.15 where the global photon map is used in the evaluation of the incoming light for the secondary diffuse reflection.

4.4.2 Specular and glossy reflection

Specular and glossy reflection is computed by evaluation of the term

$$\int_{\Omega_x} f_{r,s}(x, \vec{\omega}', \vec{\omega})(L_{i,c}(x, \vec{\omega}') + L_{i,d}(x, \vec{\omega}')) \cos\theta_i \, d\omega_i' \, .$$

The photon map is not used in the evaluation of this integral since it is strongly

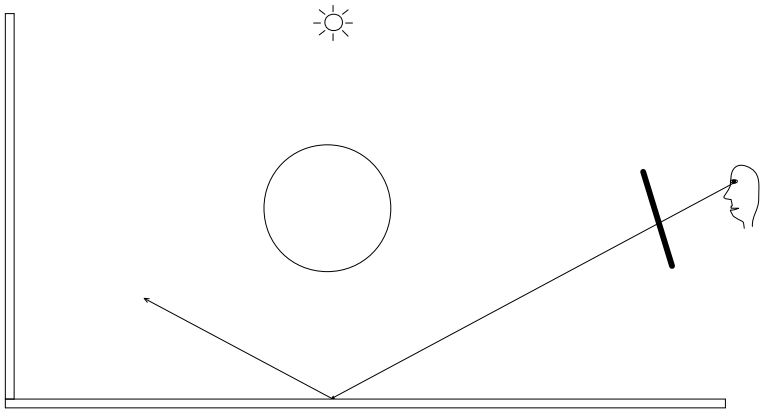


Figure 4.13: Rendering specular and glossy reflections.

dominated by $f_{r,s}$ which has a narrow peak around the mirror direction. Using the photon map to optimize the integral would require a huge number of photons in order to make a useful classification of the different directions within the narrow peak of $f_{r,s}$. To save memory this strategy is not used and the integral is evaluated using standard Monte Carlo ray tracing optimized with importance sampling based on $f_{r,s}$. This is still quite efficient for glossy surfaces and the integral can in most situations be computed using only a small number of sample rays.

This is illustrated in Figure 4.13.

4.4.3 Caustics

Caustics are represented by the integral

$$\int_{\Omega_x} f_{r,d}(x, \vec{\omega}', \vec{\omega}) L_{i,c}(x, \vec{\omega}') \cos\theta_i \, d\omega_i' \, .$$

The evaluation of this term is dependent on whether an accurate or an approximate computation is required. In the accurate computation, the term is solved by using a radiance estimate from the caustics photon map. The number of photons in the caustics photon map is high and we can expect good quality of the estimate. Caustics are never computed using Monte Carlo ray tracing since this is a very inefficient method when it comes to rendering caustics. The approximate evaluation of the integral is included in the radiance estimate from the global photon map.

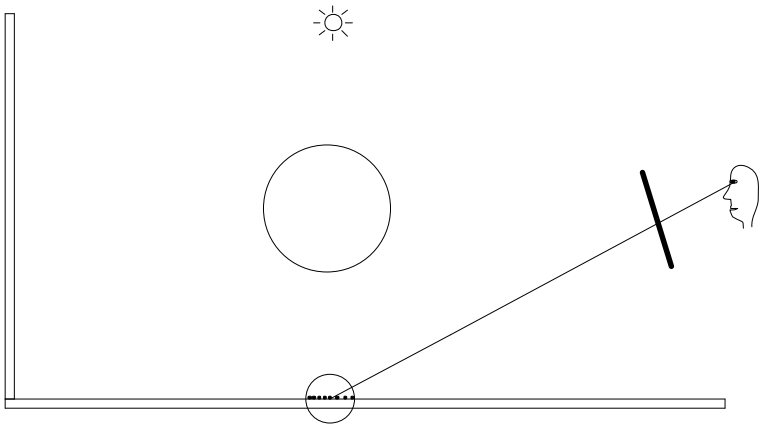


Figure 4.14: Rendering caustics.

This is illustrated in Figure 4.14.

4.4.4 Multiple diffuse reflections

The last term in equation 4.19 is

$$\int_{\Omega_x} f_{r,d}(x, \vec{\omega}', \vec{\omega}) L_{i,d}(x, \vec{\omega}') \cos\theta_i \, d\omega_i' \, .$$

This term represents incoming light that has been reflected diffusely at least once since it left the light source. The light is then reflected diffusely by the surface (using $f_{r,d}$). Consequently the resulting illumination is very "soft".

The approximate evaluation of this integral is a part of the radiance estimate based on the global photon map.

The accurate evaluation of the integral is calculated using Monte Carlo ray tracing optimized using the BRDF with an estimate of the flux as described in [**?**]. An important optimization at Lambertian surfaces is the use of Ward's irradiance gradient caching scheme [82, **?**]. This means that we only compute indirect illumination on Lambertian surfaces if we cannot interpolate with sufficient accuracy from previously computed values. The advantage of using the photon map compared to just using the irradiance gradient caching method is that we avoid having to trace multiple bounces of indirect illumination and we can use the information in the photon map to concentrate our samples into the important directions.

This is illustrated in Figure 4.15.

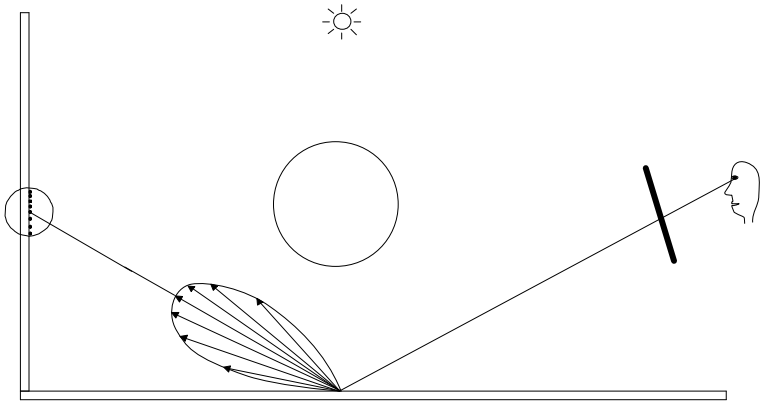


Figure 4.15: Computing indirect diffuse illumination with importance sampling.

4.4.5 Participating media

In the presence of participating media we can still use the framework as presented so far. The main difference is that we need to take the media into account as we trace rays through the scene. This can be done quite efficiently using ray marching and the volume radiance estimate as described in [**?**].

4.4.6 Why distribution ray tracing?

The rendering method presented here is a combination of many algorithms. In order to render accurate images without using too many photons a distribution ray tracer is used to compute illumination seen directly by the eye. One might consider visualizing the global photon map directly, and this would indeed be a full global illumination solution (it would be similar to the density estimation approach presented in [**?**]). The problem with this approach is that an accurate solution requires a large number of photons. Significantly fewer photons are necessary when a distribution ray tracer is used to evaluate the first diffuse reflection. If a blurry solution is not a problem (for example for previewing) then a direct visualization of the photon map can be used. For more accurate results it is often necessary to

use more than 1000 photons in the radiance estimate (see the results section for some examples).

4.5 Examples

In this section we present some examples of scenes rendered using photon maps. Please see the photon map web-page at `http://www.gk.dtu.dk/photonmap` for the latest results. Also refer to the papers included in these notes for more examples.

All the images have been rendered using the `Dali` rendering program. `Dali` is an extremely flexible renderer that supports ray tracing with global illumination and participating media. The global illumination simulation code based on photon maps is a module in `Dali` that is loaded at runtime. All material and geometry code is also represented via modules that are loaded at runtime. `Dali` is multi-threaded and all images have been rendered on a dual PentiumII-400 PC running Linux. The width of each image is 1024 pixels and 4 samples per pixel have been used.

4.5.1 The Cornell box

Most global illumination papers feature a simulation of the Cornell box, and so does this note. Since we are not limited to radiosity our version of the Cornell box is slightly different. It has a mirror sphere and a glass sphere instead of the two cubes featured in the original Cornell box (the original Cornell box can be found at `http://www.graphics.cornell.edu/online/box/`). Classic radiosity methods have difficulties handling curved specular objects, but ray tracing methods (including the photon map method) have no problems with these.

Ray tracing

The image in Figure 4.16 shows the *ray traced* version of the Cornell box. Notice the sharp shadows and the black ceiling of the box due to lack of area lights and global illumination. Rendering time was 3.5 seconds.

Ray tracing with soft shadows

In Figure 4.17 soft shadows have been added. It has been reported that some people associate soft shadows with global illumination, but in the Cornell box example it is still obvious that something is missing. The ceiling is still black. Rendering time was 21 seconds.

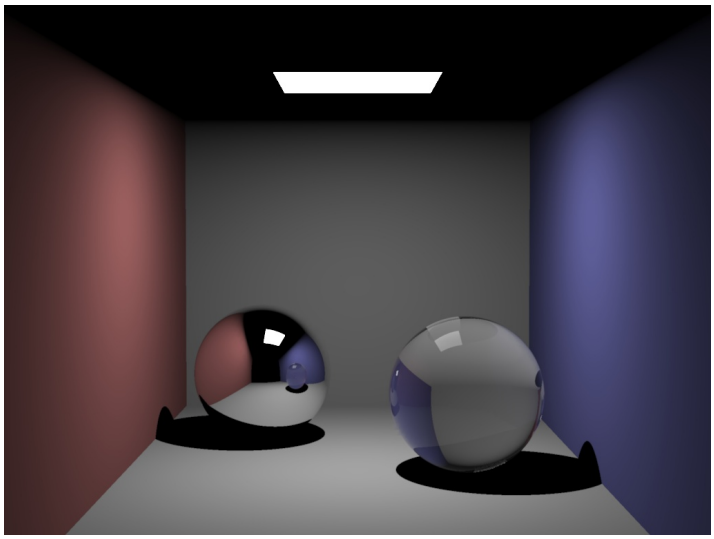


Figure 4.16: Ray traced Cornell box with sharp shadows.

Adding caustics

The image in Figure 4.18 includes the caustics photon map. Notice the bright spot below the glass sphere and on the right wall (due to light reflected of the mirror sphere and transmitted through the glass sphere). Also notice the faint illumination of the ceiling. The caustics photon map has 50000 photons and the estimate uses up to 60 photons. Photon tracing took 2 seconds. Rendering time was 34 seconds. We did not use any filtering of the caustics photons. A maximum search distance of 0.15 was used for the caustics photon map (the depth of the Cornell box is 5 units). Using a search distance of 0.5 increased the rendering time to 42 seconds. For an unlimited initial search radius the rendering time was 43 seconds. The computed images looked very similar. The faint illumination of the ceiling is a caustic (created by the bright caustic on the floor) — it becomes a little softer with the increased search radius. For a search radius of 0.01 the caustics became more noisy, and the rendering time was 25 seconds. For other scenes where the caustics are more localized the influence of the maximum search radius on the rendering time can be more dramatic than for the Cornell box.

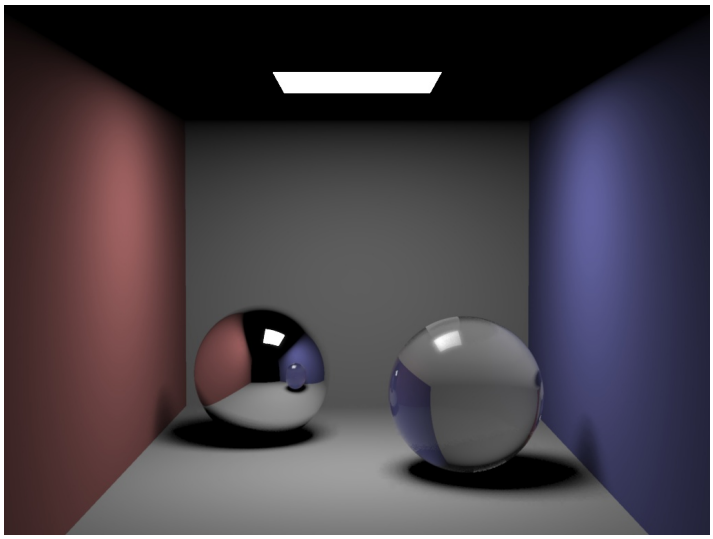


Figure 4.17: Ray traced Cornell box with soft shadows.

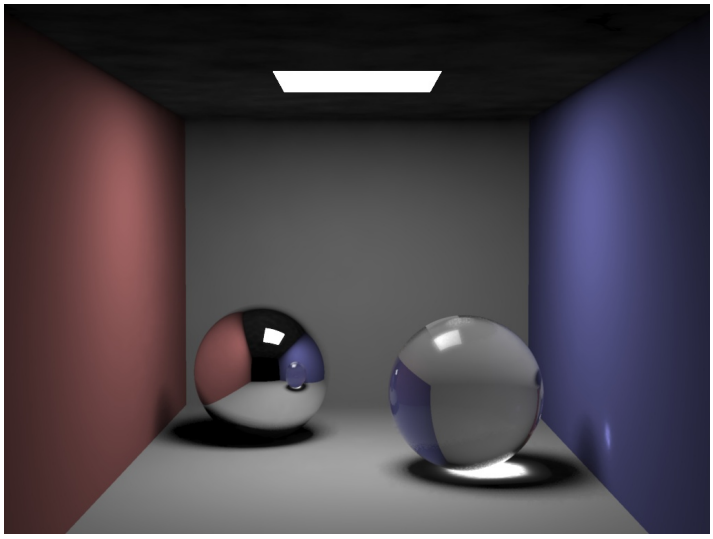


Figure 4.18: Cornell box with caustics.

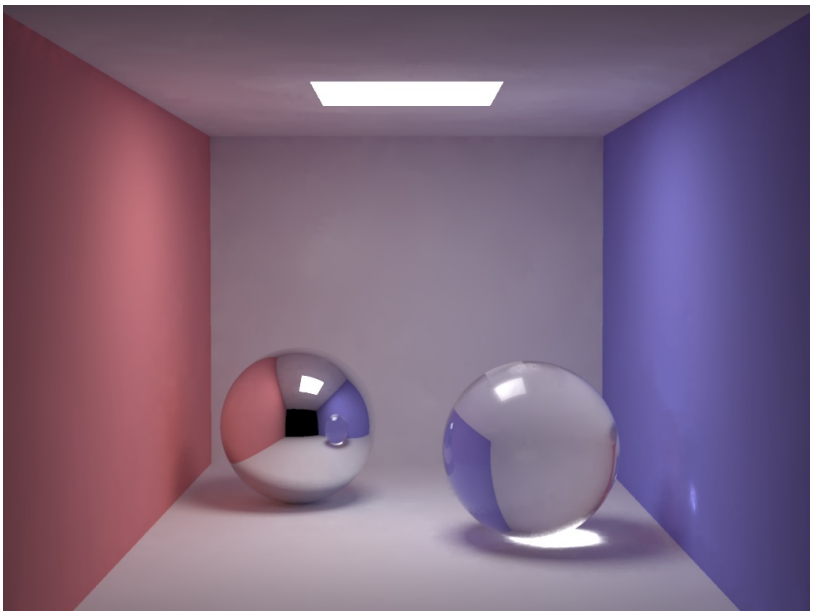


Figure 4.19: Cornell box with global illumination.

Global illumination

In Figure 4.19 global illumination has been computed. The image is much brighter and the ceiling is illuminated. 200000 photons were used in the global photon map and 100 photons in the estimate. The caustic photon map parameters are the same. Photon tracing took 4 seconds. Rendering time was 66 seconds.

The radiance estimate from the global photon map

Finally in Figure 4.20 we have visualized the radiance estimates from the global photon map directly. We have shown images with 100 and 500 photons in the estimate. Notice how the illumination becomes softer and more pleasing with more photons, but also more blurry and with more false color bleeding at the edges. The edge problem can be solved partially by using an ellipsoid or disc to locate the photons (see section 4.3) — with 500 photons in the estimate and the ellipsoid search activated we get the image in Figure 4.21 These images took 30–35 seconds to render. Notice how the quality of the direct visualization gives a reasonable

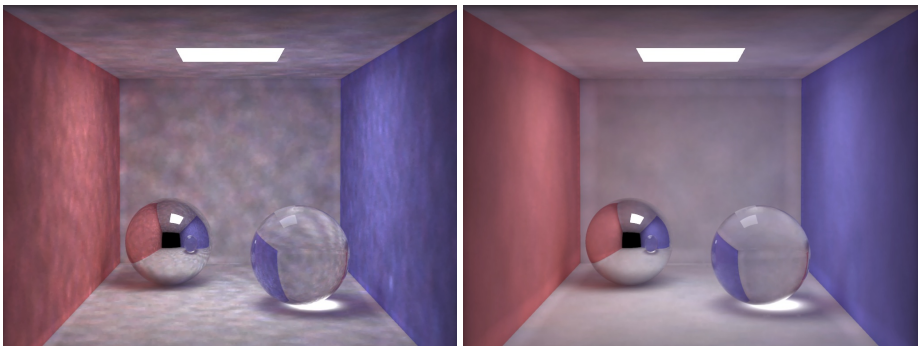


Figure 4.20: Global photon map radiance estimates visualized directly using 100 photons (left) and 500 photons (right) in the radiance estimate.

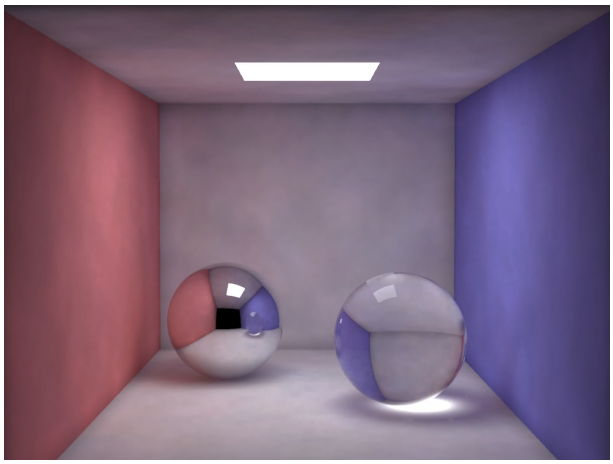


Figure 4.21: Global photon map radiance estimates visualized directly using 500 photons and a disc to locate the photons. Notice the reduced false color bleeding at the edges.

estimate of the overall illumination in the scene. This is the information we benefit from in the full rendering step since we do not have to sample the incoming light recursively.

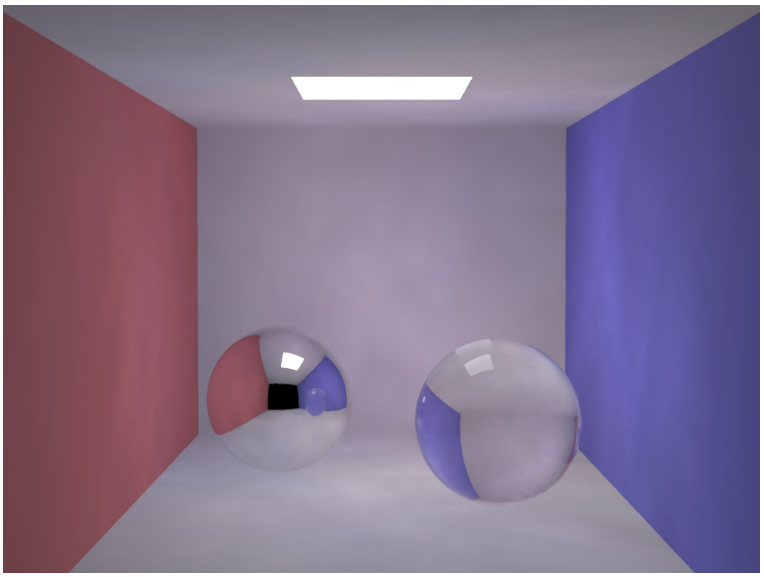


Figure 4.22: Fast visualization of the radiance estimate based on 50 photons and a global photon map with just 200 photons. Rendering time was 4 seconds.

Fast global illumination estimate

For fast visualization of global illumination one can use very few photons in the global photon map. In Figure 4.22 we have visualized the radiance estimate from a global photon map with just 200 photons! We used up to 50 photons in the radiance estimate. The illumination is very blurry and as a consequence the shadows and the caustics are missing, but the overall illumination is approximately correct, and this visualization is representative of the final rendering as shown in Figure 4.19. Photon tracing took 0.03 seconds and the rendering time for the image was 4 seconds. This is almost as fast as the simple ray tracing version, and the main reason is that we only used ray tracing to compute the first intersection and the mirror reflections and transmissions. The global photon map was used to estimate both indirect and direct light.

Figure 4.23: Cornell box with water.

4.5.2 Cornell box with water

In the Cornell box in Figure 4.23 we have inserted a displacement-mapped water surface. To render this scene we used 500000 photons in both the caustics and the global photon map, and up to 100 photons in the radiance estimate. We used a higher number of caustic photons due to the water surface which causes the entire floor to be illuminated by the photons in the caustics photon map. Also the number of photons in the global photon map have been increased to account for the more complex indirect illumination in the scene. The water surface is made of 20000 triangles. The rendering time for the image was 11 minutes.

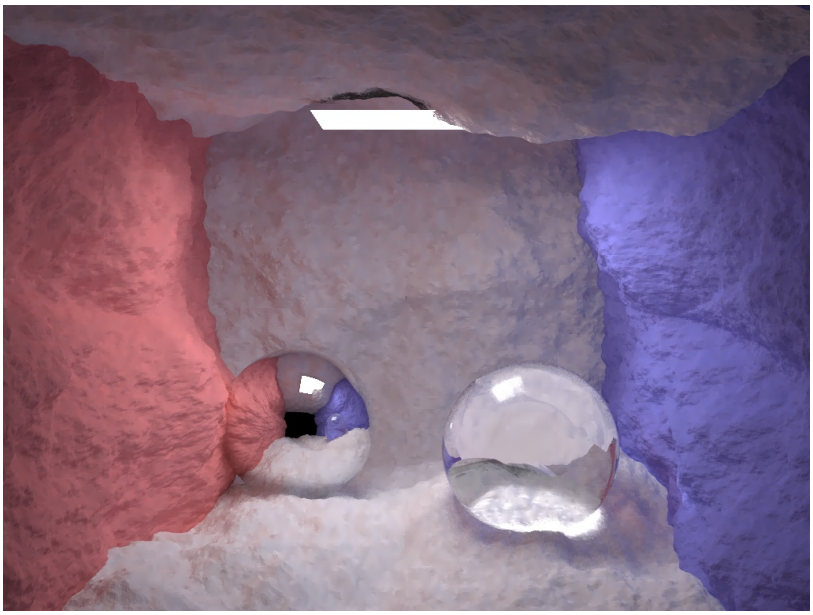


Figure 4.24: Fractal Cornell box.

4.5.3 Fractal Cornell box

An example of a more complex scene is shown in Figure 4.24. The walls have been replaced with displacement mapped surfaces (generated using a fractal midpoint subdivision algorithm) and the model contains a little more than 1.6 million elements. Notice that each wall segment is an instanced copy of the same fractal surface. With photon maps it is easy to take advantage of instancing and the geometry does not have to be explicitly represented. We used 200000 photons in the global photon map and 50000 in the caustics photon map. This is the same number of photons as in the simple Cornell box and our reasoning for choosing the same values are that the complexity of the illumination is more or less the same as in the simple Cornell box. We want to capture the color bleeding from the colored walls and the indirect illumination of the ceiling. All in all we used the same parameters for the photon map as in the simple Cornell box. We only changed the parameters for the acceleration structure to handle the larger amount of geometry. The rendering time for the scene was 14 minutes.

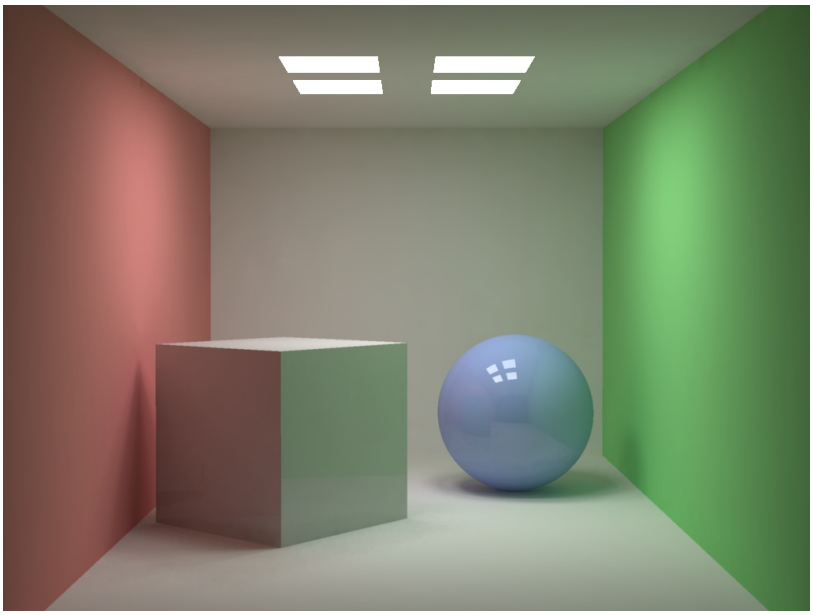


Figure 4.25: Cornell box variation with 4 light sources.

4.5.4 Cornell box with multiple lights

A simple example of a scene with multiple light sources is the variation of the Cornell box scene shown in Figure 4.25. We generated 100000 photons from each light source and the resulting global photon map has 400000 photons. Other than that the rendering parameters were the same as for the other Cornell box with 1 light source. The rendering time for this scene was 90 seconds.

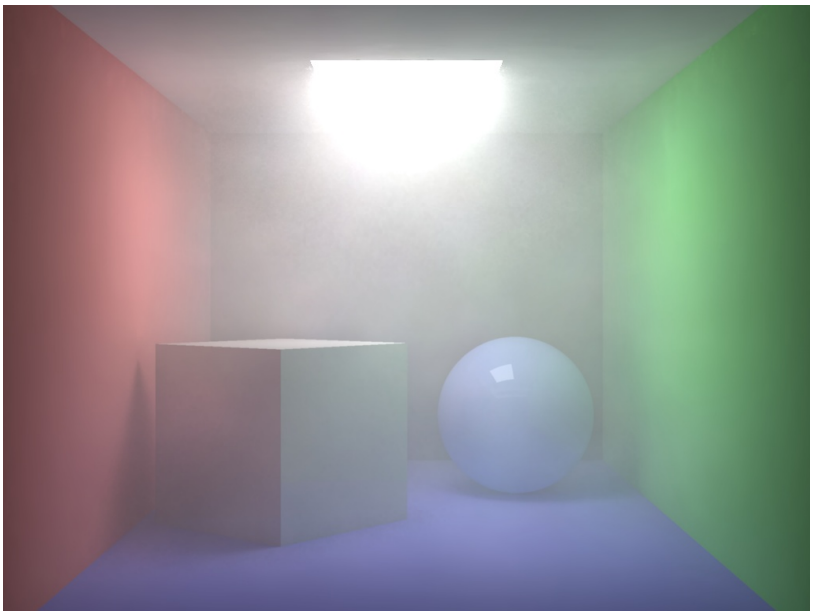


Figure 4.26: Cornell box with a participating medium.

4.5.5 Cornell box with smoke

The Cornell box scene shown in Figure 4.26 is an example of a scene with a uniform participating medium. To simulate this scene we used 100000 photons in the global photon map and 150000 photons in the volume photon map. A simple non-adaptive ray marcher has been implemented so the step size had to be set to a low value which is extra costly. The rendering time for the scene was 44 minutes.

Figure 4.27: A cognac glass with a caustic.

4.5.6 Cognac glass

Figure 4.27 shows an example of a caustic from a cognac glass. The glass is an object of revolution approximated with 12000 triangles. To generate the caustic we used 200000 photons. The radiance estimates for the caustic were computed using up to 40 photons. The rendering time for the image was 8 minutes and 10 seconds — part of this rendering time is due to the ray traced depth of field simulation.

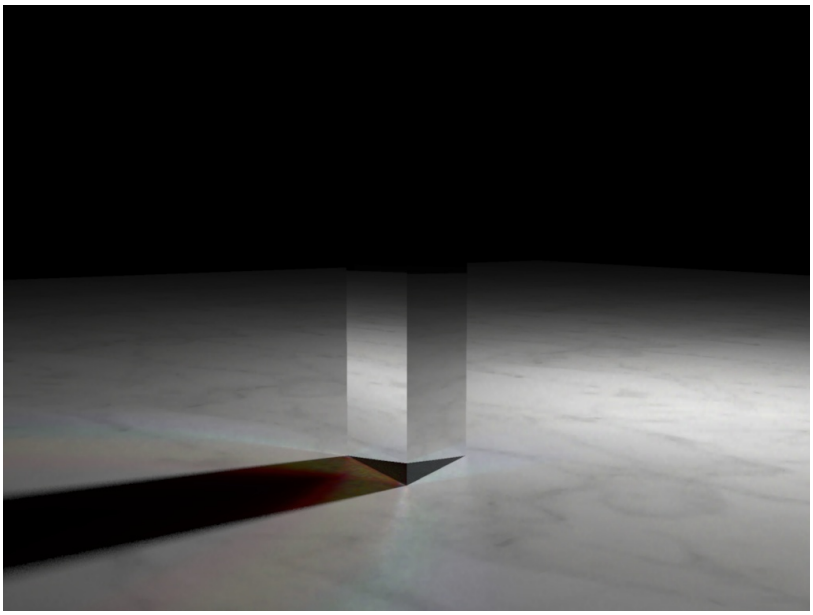


Figure 4.28: Caustics through a prism with dispersion.

4.5.7 Prism with dispersion

The classic example of dispersion with glass prism is shown in Figure 4.28. Even though only three separate wavelengths have been sampled, the color variations in the caustics are smooth. An accurate color representation would require more wavelength samples; such an extension to the photon map is easy to implement. 500000 photons were used in the caustics and 80 photons were used in the radiance estimate. The rendering time for the image was 32 seconds.

Figure 4.29: Granite bunny next to a marble bunny — both models are rendered using subsurface scattering. The photon map is used to compute multiple scattering inside the stone material.

4.5.8 Subsurface scattering

A recent addition to the photon map is the simulation of subsurface scattering [**?**, **?**]. For subsurface scattering we use the photon map to compute the effect of multiple scattering within a given material. This is often very costly to compute and therefore mostly omitted from approaches dealing with subsurface scattering. This is unfortunate since multiple scattering leads to very nice and subtle color bleeding effects inside the material which improves the quality of the rendering.

Figure 4.29 shows a granite bunny next to a marble bunny. Both of these stone models are rendered using subsurface scattering with 100000 photons used to simulate multiple scattering. The rendering time for the picture was 21 minutes. This bunny is the original Stanford bunny and the scene contains 140000 triangles, and it is rendered with full global illumination and depth of field.

Figure 4.30 shows a bust of Diana the Huntress made of translucent marble. For this scene the light source was behind the bust to emphasize the effect of subsurface scattering. Notice the translucency of the hair and the nose region. This image was rendered in 21 minutes using 200000 photons.

Figure 4.30: Translucent marble bust illuminated from behind

4.6 Where to get programs with photon maps

Photon maps are already available on the Internet for downloading. We have collected the following links as of the writing of these notes.

RenderPark (a photorealistic rendering tool) has photon maps (as well as many

other global illumination algorithms). See
`http://www.cs.kuleuven.ac.be/cwis/research/graphics/RENDERPARK/` for more information.

Most commercial renderers now supports photon mapping for global illumination.

Chapter 5

Photon mapping for complex scenes

In this chapter, we present methods for photon emission for complex light sources and photon scattering for complex surface shaders (including displacement shaders), show how to precompute radiosity estimates for efficient final gathering, describe the radiosity atlas method for photon mapping in very large scenes, and finally discuss the use of importance to guide the storage of photons.

In this chapter we will use two scenes from the Pixar movies *Ratatouille* and *Monsters, Inc.* as examples. However, it should be emphasized that photon mapping was not used in the production of those movies; we're just using these scenes here as examples of how photon mapping could be used in a movie production setting with very complex illumination, shaders, and geometry.

5.1 Photon emission from complex light sources

Emitting photons corresponding to simple light sources such as point lights, spot lights, and directional lights is fairly straightforward. However, movie production light source shaders are very complex: they can project images like a slide projector and they can have barn doors, cucoloris ("cookies"), unnatural distance fall-off, fake shadows, artistic positioning of highlights, etc. [5]. For photon emission corresponding to such a light source, we would need to compute a photon emission probability distribution function that corresponds to the light source shader. This can be difficult since the programmable fall-off requires evaluation of the shader not only at different angles but also at different distances.

However, there is a simple method to create photon emission distributions that exactly match very general light sources. The method starts by evaluating the light source shader on the surface points. This is done by rendering the scene with direct illumination from the light source(s) and storing a point cloud of the direct illumination. The point cloud contains an illumination point for each surface shading point (roughly one point per pixel). This is a sampling of the light source shader exactly at the positions where its values matter (namely at the surfaces in the scene), and ensures that the photon distribution exactly matches the illumination — no matter how complex and unpredictable the light source shader is. We basically treat the light source shader as a "black box" that is characterized only by its illumination on the surfaces in the scene.

On specular or partially specular surfaces, the illumination has to be computed for each light source separately, and the illumination at each point must be stored with a vector indicating the incident light direction. For purely diffuse surfaces, the illumination from all light sources can simply be added up (and stored without the light direction vector).

Figure 5.1 shows the direct illumination on the diffuse surfaces of the *Ratatouille* kitchen. The illumination is specified with more than 200 light sources, each with rather complex illumination characteristics. (Rendering this image at resolution 720×405 takes around 42 minutes on 1 processor — this is an indication of just how complex the light source shaders are.) The generated point cloud contains 4.8 million points.

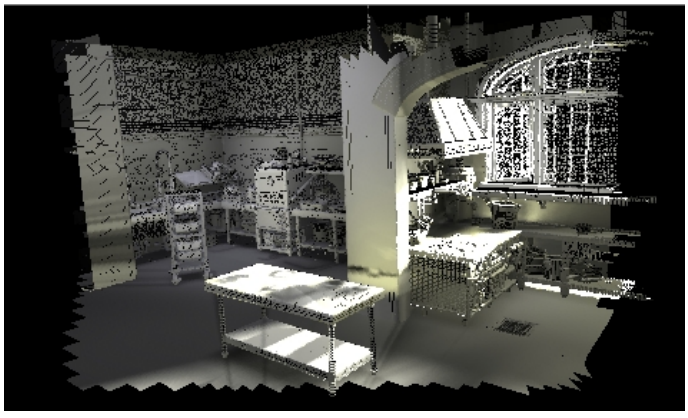


Figure 5.1: Point cloud showing the illumination from over 200 light sources in the *Ratatouille* kitchen.

The next step is to emit photons with probability proportional to the stored illumination power. As always with photon mapping, we must ensure that all emitted (and all stored) photons have the same power. This is necessary in order to minimize noise in the irradiance estimates.

What follows is a more detailed overview of the photon emission algorithm: First we compute the total power of all illumination in the direct illumination point cloud. (The incident power at each point is the point's irradiance times its area.) The target power per photon is the total power in the point cloud divided by the specified number of photons to emit. Then, for each point in the illumination point cloud, we compute the number of photons to emit for that point as the power of the point divided by the target power per photon. This will usually result in a fractional number of photons; we use Russian roulette [4] to round up or down to an integer number and adjust the power of the emitted photon(s) accordingly. The photon(s) are emitted by shooting them from a point just above the position of the point cloud point (jittered within the radius of the point). Since the photons are emitted probabilistically for each point cloud point, the final number of emitted photons may end up being slightly different from the specified target number.

(A possible improvement of the method would stratify the random numbers such that no matter what sequence of pseudorandom numbers is encountered, the Russian roulette will not result in entire regions where the number of emitted photons is rounded down and other regions where it is rounded up. Quasi-random numbers could be used as well.)

The idea of emitting photons from a point cloud of direct illumination values has also been utilized by Hašan et al. [31]. Their photon emission was from and to a sparse point cloud, and was used to compute an estimate of the multi-bounce global illumination in a very coarse point-cloud representation of a scene.

5.2 Photon scattering from complex surfaces

In this section we'll look at photon scattering from surfaces with complex scattering characteristics.

For simple surface shaders, it is straightforward to write a corresponding photon shader. Given an incident photon, the photon shader must stochastically determine whether the photon should be scattered — and if so, the power and direction of the scattered photon. However, just as for the photon emission case discussed above, the surface shaders used in movie production are very, very complex. They

have hundreds or thousands of parameters, and consists of tens of thousands lines of code. In the worst case, some shaders might be "legacy code" that very few people, if any, have a complete understanding of. Writing photon shaders that correspond to such surface shaders would be quite a nightmare.

Instead, we can pry the surface reflection coefficients out of the shader. In the simplest case, the reflection parameter is just a diffuse color; we can obtain that color by illuminating the scene with single ambient light source with intensity 1. Figure 5.2 shows the diffuse colors (diffuse reflection coefficients) in the *Ratatouille* kitchen scene. The scene has around 150 different surface shaders and more than 1000 textures (including a few shadow maps).

(a)

(b)

Figure 5.2: Diffuse colors in kitchen: (a) rendered image; (b) point cloud.

For more complex shaders, there might be coefficients for diffuse reflection, specular reflection, diffuse transmission, and specular transmission, as well as one or more values specifying the narrowness of the specular (glossy) scattering. The reflection coefficients may depend on dozens of textures combined in strange ways. But no matter how complex the shader is, we assume that the result of it can be expressed by a few reflection coefficients that are stored in a point cloud.

The stored colors are used as scattering coefficients during the photon scattering step. They are used to calculate the scattering probability and the power of scattered photons. The point cloud of scattering coefficients is organized into a kd-tree for fast lookups. When a photon hits a surface, we do a kd-tree lookup in the point cloud to determine the coefficients at the hit point. The photon is stored if appropriate and scattered if appropriate (using the usual Russian roulette strategy for computing photon scattering probilities and new photon power).

This method also handles displacement-mapped surfaces gracefully. If a surface is displaced, that displacement will be present both in the point positions in

the stored point cloud of scattering coefficients and in the photon hit positions used for lookups in the kd-tree.

5.3 The radiosity map

In the standard photon mapping method, when a final gather ray hits a point, the nearest n photons are looked up in the photon map kd-tree in order to estimate the irradiance at the hit point.

However, it is more efficient to precompute the irradiance at some or all the photon positions. Then only the nearest photon needs to be looked up at final gather ray hit points [11]. The irradiance precomputation is quick, and makes the rendering 5–7 times faster for typical scenes without degrading the image quality. A further optimization is to store the local diffuse surface color with each photon, and estimating radiosity instead of irradiance at the photon positions [12]. Estimating radiosity takes only a tiny bit longer than estimating irradiance (only a few extra multiplications per photon), but makes it faster to compute the indirect illumination — especially if the scene has complex shaders and many textures. Each radiosity estimate is computed by locating the nearest photons, adding up their power, dividing by the area they cover, and multiplying by the diffuse surface color.

The result of this step is a new point cloud with position, normal, radius and radiosity data. We call this point cloud a *radiosity map*. (The "radiosity" name here denotes that the data in the point cloud are radiosities; it does not imply that they have been computed with a finite-element "radiosity method".) The point positions divide the surfaces into a Voronoi diagram with constant radiosity inside each Voronoi cell. These radiosity values can be visualized directly for a rough estimate of the global illumination in the scene. Figure 5.3 shows the kitchen scene rendered with the radiosity map values.

Figure 5.4(a) shows the Ratatouille kitchen rendered with only direct illumination. Figure 5.4(b) shows the same scene with added indirect illumination computed with final gathering using the radiosity map shown above.

Figure 5.5 shows another example of photon map global illumination. This scene has simple base geometry but texture maps and displaced surfaces: the floor, left and right wall are textured, and the left and back wall and front teapot are displacement mapped. Figure 5.5(a) shows the direct illumination in the scene. Figure 5.5(b) clearly shows the tinting typically associated with color bleeding.

Figure 5.3: Radiosity map for kitchen.

(a) (b)

Figure 5.4: *Ratatouille* kitchen: (a) direct illumination; (b) global illumination. (Copyright © 2007 Disney/Pixar.)

Furthermore, notice that all illumination on the ceiling is indirect light.

The same precomputation method can also be used for precomputing radiances in a volume photon map. This can speed up ray marching through an isotropically reflecting participating medium quite significantly.

5.4 The radiosity atlas for large scenes

Here we extend the photon mapping method to enable it to handle scenes with a lot of geometry. A limitation of the original photon map method, and also the radiosity map method presented in the previous section, is that they assume that the entire photon map is stored in memory. If the scene is so complex (contains so much geometry) that the number of photons required to represent the illumination in it

(a) (b)

Figure 5.5: A box with textures and displacements: (a) direct illumination; (b) global illumination.

exceeds the memory capacity of the computer, we need to divide the photon map into "chunks" that can be read, processed, and cached independently.

The scene used as example in this section is from the Pixar movie *Monsters, Inc.*: a city block of Monstropolis with many individually modeled buildings, trees, cars, etc. The scene geometry consists of 36,000 high-level primitives, mostly NURBS patches and subdivision surfaces. The shaders have been simplified for these tests, and only a single light source is used. Figure 5.6 shows the Monstropolis city block with direct illumination from a single point light source and purely diffuse reflection. Large parts of the scene are completely black since no direct light reaches them.

Figure 5.6: Direct illumination in the Monstropolis scene.

5.4.1 Photon emission and photon tracing

In the first step, we divide the objects into groups. For the Monstropolis scene we use the groups created during modeling, so e.g. each house and each car is a group of objects. Each group gets a separate photon map. Figure 5.7 shows two of the photon maps for the Monstropolis scene. Both the photon powers and the diffuse surface colors are shown. The photon map for the car contains 76,000 photons, while the photon map for the building contains 2.2 million photons. The photons powers get a green tint when refracted through the windshield. Also notice the red and blue diffusely reflected photons on the building.

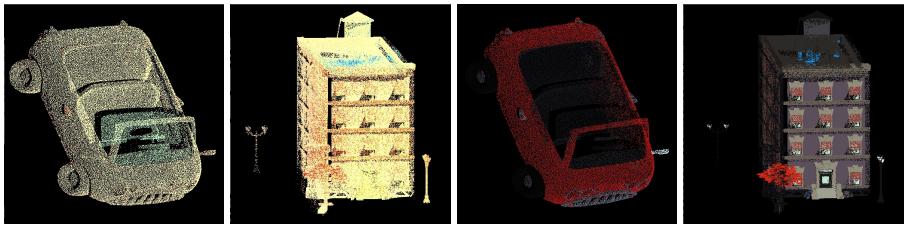


Figure 5.7: Photon maps for the car and building. The two left images show the photon powers, the two right images show the diffuse surface colors.

We call a collection of photon maps a *photon atlas*. Figure 5.8 shows the photon atlas for the Monstropolis scene. For clarity, the figure only shows 0.1% of the 52 million photons in the photon atlas.

5.4.2 Radiosity estimation

As in section 5.3, we sort the photons in a photon map into a kd-tree and precompute the radiosity at each photon location. We do this independently for each photon map. Figure 5.9 shows two of the resulting radiosity maps for the Monstropolis scene.

5.4.3 Generating radiosity brick maps

The next step is to compute a brick map representation of each of the radiosity point clouds.

A *brick map* is a tiled 3D MIP map data structure [15]. It is a useful data structure for representing general 3D textures — both textures on surfaces and in volumes. (Brick maps can also be rendered and ray traced directly as geometric

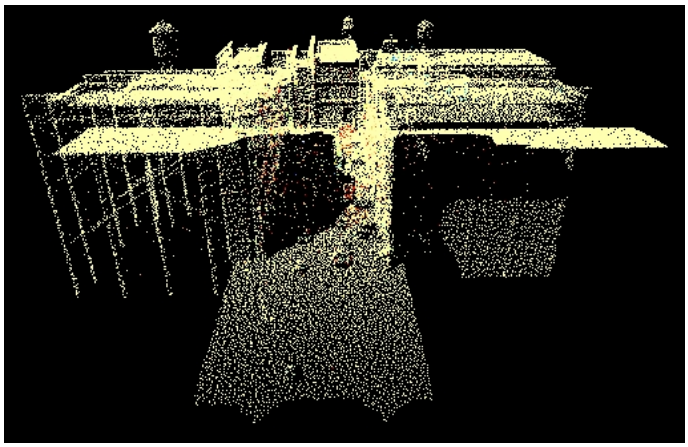


Figure 5.8: Coarse photon atlas (collection of photon maps) for the Monstropolis scene. The photon powers are shown.

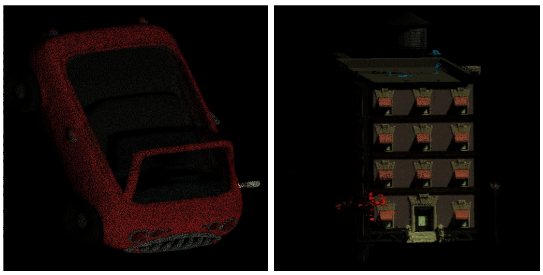


Figure 5.9: Radiosity maps for the car and building.

primitives [14].) A brick consists of 8×8×8 voxel positions. The coarsest representation in a brick map consists of a single brick. The next level consists of (up to) 2×2×2 bricks. The next level consists of (up to) 4×4×4 bricks, etc. The advantage of a brick map representation over a point cloud is that the brick map is hierarchical and tiled, and that the bricks can be cached efficiently.

For efficient global illumination, we use brick maps to represent the radiosity data in each radiosity map. Two of the radiosity brick maps for the Monstropolis scene are shown in figure 5.10. We call the collection of radiosity brick maps a *radiosity brick atlas* or simply a *radiosity atlas*.

Figure 5.11 shows the entire Monstropolis scene rendered with radiosity directly from the radiosity atlas. This image gives a rough indication of the global illumination in the scene, but it is far too noisy and blurry to be used in a movie.

Figure 5.10: Radiosity brick maps for the car and building.

Figure 5.11: The Monstropolis scene rendered with radiosity from the radiosity atlas.

5.4.4 Rendering

When a final gather ray hits a surface point, the ray differential determines the appropriate brick map level to look up in: we choose the brick map level where the brick voxels are approximately the size of the ray beam cross-section (similar to how we select 2D texture levels and tessellation levels).

Figure 5.12 shows a final gather rendering of the scene. At the final gather ray hit points, the radiosity is determined from the radiosity atlas shown in figure 5.11. (Rendering this image took 5.7 hours on a 2 GHz Apple G5. It required the shooting of 413 million final gather rays and 3.8 million shadow rays. There were 3.4 billion brick cache lookups with a hit rate of 99.9%.)

Figure 5.12: Monstropolis city block with global illumination: direct illumination from the sun and sky, and indirect illumination computed using final gathering and the radiosity atlas. (Copyright © Disney/Pixar.)

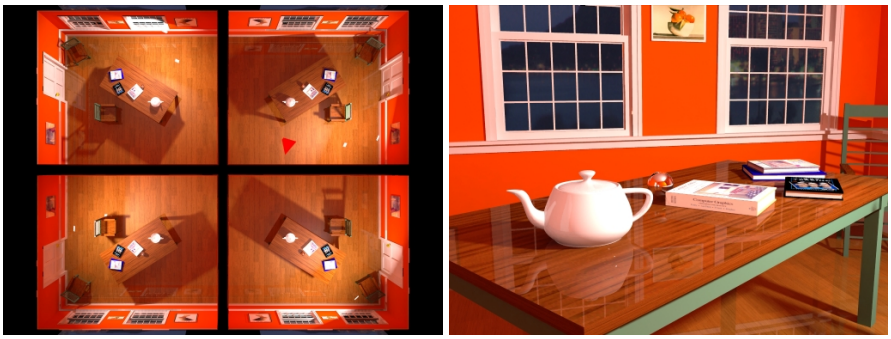
5.5 Importance for photon tracing

This section describes the use of importance to concentrate the photons in those parts of the scene where they contribute most to the final image.

5.5.1 Importance

The first use of importance was for computer simulations of neutron transport for the development of the hydrogen bomb (right after World War II). Fortunately importance can also be used for more peaceful purposes. Smits et al. [69] formally introduced importance to computer graphics. They defined importance as the adjoint of radiosity that has a source term at the viewpoint, and used importance to reduce the number of links in a hierarchical radiosity solution. More generally, importance can be defined as the adjoint of any representation of light with the source term at the viewpoint or at the directly visible points. An overview of importance and its use in computer graphics can be found in Christensen [13]. The savings obtained using importance can be arbitrarily high: just choose a sufficiently large scene with a sufficiently small visible part.

Figure 5.13(a) shows a modest example scene, four rooms with closed doors between them. Figure 5.13(b) is the image we are interested in computing, a close-up of the tabletop in the upper right room. The red pyramid in figure 5.13(a) indicates the viewpoint and viewing frustum for figure 5.13(b).



(a) (b)

Figure 5.13: Orange interior: (a) entire scene seen from above; (b) seen from the intended viewpoint.

5.5.2 Importance emission and estimation

Importance can be used to focus the photon storage in those parts of the scene where the photons will contribute most to the final image.

The first step is to emit importance particles ("importons") [54, 40, 72] from the intended viewpoint in directions within the viewing frustum. These particles are traced around the scene as if they were photons and stored every time they hit a diffuse surface. The stored particles for the example scene are shown in figure 5.14(a). (The emitted importance particles are white, but they change color at each bounce according to the color of the surfaces they hit. In this scene most particles turn orange.) After the importance particle tracing, the importance is estimated at each importance particle location using the local density of importance particles. These importance estimates are stored with the importance particles; the estimates are shown in figure 5.14(b). Note that no importance reaches adjacent rooms since the doors are closed; this reflects the fact that no light from adjacent rooms reaches the visible parts of the scene.

These precomputed importances make it much faster to determine the importance at various locations during photon tracing, as described in the following.

5.5.3 Photon tracing

In the photon tracing phase, the importance estimates are used to determine photon storage probabilities. At locations with low importance, we use Russian roulette to decide whether to store the photon or not; if the photon is stored, its power is

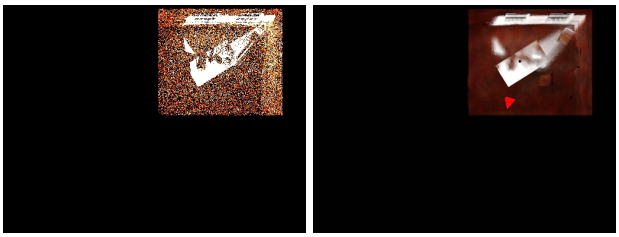
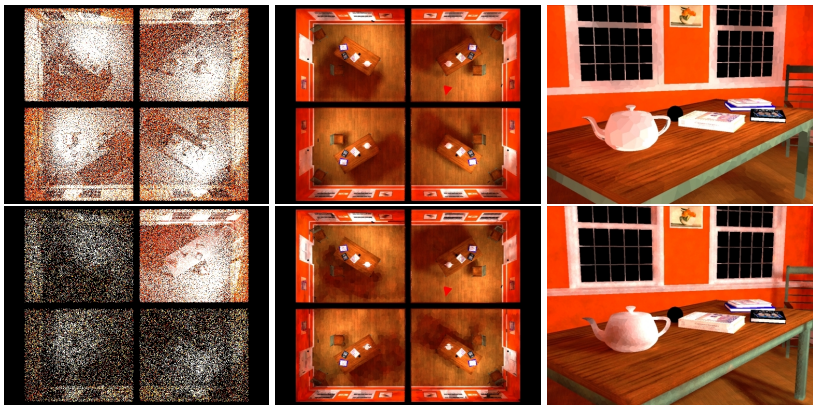
(a)  (b)

Figure 5.14: Importance in interior scene, seen from above: (a) 100,000 importance particles; (b) importance estimates.

increased to compensate for the low storage probability. Compare figures 5.15(a–c) with 5.15(d–f): Importance was not used in figures 5.15(a–c), so most photons are stored in bright regions, no matter how unimportant those regions are. In contrast, figures 5.15(d–f) show the gain from using importance — most photons are stored in areas that are either directly visible from the intended viewpoint, or are significantly influencing the illumination there. Due to the higher concentration of photons, the illumination is approximated much more accurately in figure 5.15(f) than in figure 5.15(c). The most visible difference is the sharper shadows.

(a) (b) (c)



(d) (e) (f)

Figure 5.15: Light in interior scene: (a) 500,000 photons stored without importance; (b)-(c) radiance estimates based on the photons in (a); (d) 500,000 photons stored using importance; (e)-(f) radiance estimates based on the photons in (d).

In this example, importance was only used to determine photon storage. It is also possible to use importance to guide photon emission and scattering [54, 12],

but that is beyond the scope of this course note.

Bibliography

[1] Oliver Abert, Markus Geimer, and Stefan Müller. Direct and fast ray tracing of NURBS surfaces. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2006*, pages 161–168. IEEE, 2006.

[2] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann Publishers, 2000.

[3] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 32, pages 37–45, 1968.

[4] James R. Arvo and David B. Kirk. Particle transport and image synthesis. *Computer Graphics (Proceedings of SIGGRAPH '90)*, 24(4):63–66, 1990.

[5] Ronen Barzel. Lighting controls for computer cinematography. *Journal of Graphics Tools*, 2(1):1–20, 1997.

[6] James F. Blinn. Models of light reflection for computer synthesized pictures. *Computer Graphics (Proceedings of SIGGRAPH '77)*, 11(2):192–198, 1977.

[7] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, 1976.

[8] Phong Bui Tuong. Illumination for computer generated pictures. *Communications of the ACM*, 18(3):311–317, 1975.

[9] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, Salt Lake City, 1974.

[10] Edwin E. Catmull and James H. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, 1978.

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

[12] Per H. Christensen. Photon mapping tricks. In *SIGGRAPH 2002 Course Note #43: A Practical Guide to Global Illumination using Photon Mapping*, pages 93–121, 2002.

[13] Per H. Christensen. Adjoints and importance in rendering: an overview. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):329–340, 2003.

[14] Per H. Christensen. Point clouds and brick maps for movie production. In Markus Gross and Hanspeter Pfister, editors, *Point-Based Graphics*, chapter 8.4. Morgan Kaufmann Publishers, 2007.

[15] Per H. Christensen and Dana Batali. An irradiance atlas for global illumination in complex production scenes. In *Rendering Techniques 2004 (Proceedings of the Eurographics Symposium on Rendering 2004)*, pages 133–141. Eurographics, 2004.

[16] Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. Ray tracing for the movie 'Cars'. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2006*, pages 1–6. IEEE, 2006.

[17] Per H. Christensen, David M. Laur, Julian Fong, Wayne L. Wooten, and Dana Batali. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum (Proceedings of Eurographics 2003)*, 22(3):543–552, 2003.

[18] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. *Computer Graphics (Proceedings of SIGGRAPH '87)*, 21(4):95–102, 1987.

[19] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics (Proceedings of SIGGRAPH '84)*, 18(3):137–145, 1984.

[20] Tony D. DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. *Computer Graphics (Proceedings of SIGGRAPH '98)*, pages 85–94, 1998.

[21] Daniel Doo and Malcolm A. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10(6):356–360, 1978.

[22] Albrecht Dürer. *Treatise on measurement with compasses and straightedge (Underweysung der Messung mit dem Zirkel und Richtscheyt)*. Nuremberg, 1525.

[23] Gerald Farin. *Curves and Surfaces for CAGD: A Practical Guide*. Academic Press, 3rd edition, 1993.

[24] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, 2nd edition, 1990.

[25] Andrew S. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989.

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

[27] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[28] Ned Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1986.

[29] Eric Haines. *Ray Tracing News*. 1987–present. (Web page: www.acm.org/-tog/resources/RTNews/html).

[30] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, Prague, 2001.

[31] Miloš Hašan, Fabio Pellacini, and Kavita Bala. Direct-to-indirect transfer for cinematic relighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, 25(3):1089–1097, 2006.

[32] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, 1986.

[33] Abu Sad al-Ala ibn Sahl. *On Burning Mirrors and Lenses*. Baghdad, 984. (Translated by Roshdi Rashed, 1990).

[34] Homan Igehy. Tracing ray differentials. *Computer Graphics (Proceedings of SIGGRAPH '99)*, pages 179–186, 1999.

[35] Henrik Wann Jensen, James Arvo, Philip Dutré, Alexander Keller, Art Oven, Matt Pharr, and Peter Shirley. *SIGGRAPH 2003 Course Note #44: Monte Carlo Ray Tracing*. 2003.

[36] James T. Kajiya. Ray tracing parametric patches. *Computer Graphics (Proceedings of SIGGRAPH '82)*, 16(3):245–254, 1982.

[37] James T. Kajiya. The rendering equation. *Computer Graphics (Proceedings of SIGGRAPH '86)*, 20(4):143–150, 1986.

[38] Toshiaki Kato. The Kilauea massively parallel ray tracer. In Alan Chalmers, Timothy Davis, and Erik Reinhard, editors, *Practical Parallel Rendering*, chapter 8. A K Peters, 2002.

[39] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics (Proceedings of SIGGRAPH '86)*, 20(4):269–278, 1986.

[40] Alexander Keller and Ingo Wald. Efficient importance sampling techniques for the photon map. In *Proceedings of the 5th Fall Workshop on Vision, Modeling, and Visualization*, pages 271–279, 2000.

[41] Leif Kobbelt, Katja Daubert, and Hans-Peter Seidel. Ray tracing of subdivision surfaces. In *Rendering Techniques '98 (Proceedings of the 9th Eurographics Workshop on Rendering)*, pages 69–80, 1998.

[42] Craig Kolb, Pat Hanrahan, and Don Mitchell. A realistic camera model for computer graphics. *Computer Graphics (Proceedings of SIGGRAPH '95)*, pages 317–324, 1995.

[43] Ares Lagae and Philip Dutré. An efficient ray-quadrilateral intersection test. *Journal of Graphics Tools*, 10(4):23–32, 2005.

[44] Johann H. Lambert. *Photometry: or on the Measure and Gradations of Light, Colors, and Shade*. 1760. (Translated from the Latin by David L. DiLaura, 2001).

[45] Hayden Landis. Production-ready global illumination. In *SIGGRAPH 2002 Course Note #16: RenderMan in Production*, pages 87–102, 2002.

[46] Charles Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, Salt Lake City, 1987.

[47] William Martin, Elaine Cohen, Russel Fish, and Peter Shirley. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools*, 5(1):27–52, 2000.

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

[49] Michael J. Muuss. Rt and remrt — shared memory parallel and network distributed ray-tracing programs. In *USENIX: Proceedings of the Fourth Computer Graphics Workshop*, 1987.

[50] Isaac Newton. *Opticks: A Treatise on the Reflections, Refractions, Inflections and Colours of Light*. London, 1704.

[51] Michael Oren and Shree K. Nayar. Generalization of Lambert's reflectance model. *Computer Graphics (Proceedings of SIGGRAPH '94)*, pages 239–246, 1994.

[52] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. In *Symposium on Interactive 3D Graphics*, pages 119–126, 1999.

[53] Darwyn Peachey. Texture on demand. Technical Report #217, Pixar, 1990. (Available at graphics.pixar.com).

[54] Ingmar Peter and Georg Pietrek. Importance driven construction of photon maps. In *Rendering Techniques '98 (Proceedings of the 9th Eurographics Workshop on Rendering)*, pages 269–280, 1998.

[55] Matt Pharr and Pat Hanrahan. Geometry caching for ray-tracing displacement maps. In *Rendering Techniques '96 (Proceedings of the 7th Eurographics Workshop on Rendering)*, pages 31–40, 1996.

[56] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers, 2004.

[57] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. *Computer Graphics (Proceedings of SIGGRAPH '97)*, pages 101–108, 1997.

[58] Les Piegl and Wayne Tiller. *The NURBS Book*. Springer-Verlag, 1997.

[59] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *Computer Graphics (Proceedings of SIGGRAPH '87)*, 21(4):283–291, 1987.

[60] Erik Reinhard, Brian Smits, and Chuck Hansen. Dynamic acceleration structures for interactive ray tracing. In *Rendering Techniques 2000 (Proceedings of the 11th Eurographics Workshop on Rendering)*, pages 299–306, 2000.

[61] Christophe Schlick. A customizable reflectance model for everyday rendering. In *Proceedings of the 4th Eurographics Workshop on Rendering*, pages 73–83, 1993.

[62] Andrei Sherstyuk. Fast ray tracing of implicit surfaces. *Computer Graphics Forum*, 18(2):139–147, 1999.

[63] Peter Shirley. *Fundamentals of Computer Graphics*. A K Peters, 2002.

[64] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. A K Peters, 2nd edition, 2005.

[65] Peter Shirley, Philipp Slusallek, Ingo Wald, et al. *SIGGRAPH 2006 Course Note #4: State of the Art in Interactive Ray Tracing*. 2006.

[66] Peter Shirley, Changyaw Wang, and Kurt Zimmerman. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1):1–36, 1996.

[67] Brian Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3(2):1–14, 1998.

[68] Brian Smits, Peter Shirley, and Michael M. Stark. Direct ray tracing of displacement mapped triangles. In *Rendering Techniques 2000 (Proceedings of the 11th Eurographics Workshop on Rendering)*, pages 307–318, 2000.

[69] Brian E. Smits, James R. Arvo, and David H. Salesin. An importance-driven radiosity algorithm. *Computer Graphics (Proceedings of SIGGRAPH '92)*, 26(2):273–282, 1992.

[70] Ian Stephenson, editor. *Production Rendering: Design and Implementation*. Springer-Verlag, 2005.

[71] Gordon Stoll, William R. Mark, Peter Djeu, Rui Wang, and Ikrima Elhassan. Razor: an architecture for dynamic multiresolution ray tracing. Technical Report TR-06-21, University of Texas at Austin, 2006. (Updated version to appear in *ACM Transactions on Graphics*).

[72] Frank Suykens and Yves D. Willems. Density control for photon maps. In *Rendering Techniques 2000 (Proceedings of the 11th Eurographics Workshop on Rendering)*, pages 11–22, 2000.

[73] Frank Suykens and Yves D. Willems. Path differentials and applications. In *Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering)*, pages 257–268, 2001.

[74] Steve Upstill. *The RenderMan Companion*. Addison Wesley Publishers, 1990.

[75] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An interactive out-of-core rendering framework for visualizing massively complex models. In *Rendering Techniques 2004 (Proceedings of the Eurographics Symposium on Rendering 2004)*, pages 81–92, 2004.

[76] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, 25(3):485–493, 2006.

[77] Ingo Wald and Steven G. Parker, editors. *Proceedings of the IEEE Symposium on Interactive Ray Tracing 2006*. IEEE, 2006. (Web page: www.sci.utah.edu-/RT06).

[78] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Michael Wagner. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering)*, pages 277–288, 2001.

[79] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Michael Wagner. Interactive rendering with coherent raytracing. *Computer Graphics Forum (Proceedings of Eurographics 2001)*, 20(3):153–164, 2001.

[80] Gregory J. Ward. Adaptive shadow testing for ray tracing. In *Proceedings of the 2nd Eurographics Workshop on Rendering*, pages 11–20, 1991.

[81] Gregory J. Ward. Measuring and modeling anisotropic reflection. *Computer Graphics (Proceedings of SIGGRAPH '92)*, 26(2):265–272, 1992.

[82] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. *Computer Graphics (Proceedings of SIGGRAPH '88)*, 22(4):85–92, 1988.

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

[84] Lance Williams. Pyramidal parametrics. *Computer Graphics (Proceedings of SIGGRAPH '83)*, 17(3):1–11, 1983.

[85] Sergei Zhukov, Andrei Iones, and Gregorij Kronin. An ambient light illumination model. In *Rendering Techniques '98 (Proceedings of the 9th Eurographics Workshop on Rendering)*, pages 45–55, 1998.