This equation is sometimes written in arrow notation, in which case it is known as the *three-point form* of the *light transport equation* and is given as

$$L(x' \to x'') = L_e(x' \to x'') + \int_{\mathcal{M}} f_r(x \to x' \to x'') \cdot L(x \to x') \cdot V(x \leftrightarrow x') \cdot G(x \leftrightarrow x') \cdot \partial A_{x'} \tag{10.182}$$

In this notation the arrows indicate the flow of light energy from point x, through point x', to point x''.

In the above equations, $\mathcal{M}$ is the union of all the surfaces in the scene, A_x is the area measure on $\mathcal{M}$, the function G, known as the *geometry function*, represents the change of variables from the original integration variable (step) $\partial\omega_i^{\perp}$ to the new integration variable $\partial A_{x'}$. This relationship is given as

$$\partial\omega_i^{\perp} = G(x, x') \cdot \partial A_{x'} \tag{10.183a}$$

$$\text{or} \quad \partial\omega_i^{\perp} = G(x \leftrightarrow x') \cdot \partial A_{x'} \tag{10.183b}$$

where

$$G(x, x') = \frac{|\cos(\theta_o) \cdot \cos(\theta_i')|}{|x - x'|^2} \tag{10.184a}$$

$$\text{or} \quad G(x \leftrightarrow x') = \frac{|\cos(\theta_o) \cdot \cos(\theta_i')|}{|x - x'|^2} \tag{10.184b}$$

For the radiosity family of rendering algorithms we can simplify the rendering Eq. (10.181), which we have developed for the Monte Carlo ray-tracing family of algorithms, and replace it by a much simpler expression of the outgoing radiosity.

With the radiosity family of rendering algorithms we assume that all the surfaces in the scene are perfect Lambertian reflectors and restrict ourselves to dealing with only diffuse inter-object reflections and diffuse emittance. Under these conditions the outgoing radiance from any point x' is constant in all directions, as the BRDF of a perfectly diffusing surface is independent of the incoming and outgoing directions of light. Thus, we can replace the complex computation of the outgoing radiance $L_o(x', \omega_o')$ of Eq. (10.181) by a much simpler expression for the outgoing radiosity $B(x')$. To explain how this simplification is justified, we rewrite Eq. (10.65) to get

$$B(x') = \int_{\mathcal{H}_i^2} L_o(x', \omega_o') \cdot \cos(\theta_r') \cdot \partial\omega_o \tag{10.185}$$

But since we are dealing with a perfectly diffuse surface, the outgoing radiance $L_o(x')$ is the same in all directions and can be moved outside the integral. Thus,

$$B(x') = L_o(x') \cdot \int_{\mathcal{H}_i^2} \cos(\theta_r') \cdot \partial\omega_o \tag{10.186}$$

Using Equ. (10.64), this reduces to

$$B(x') = L_o(x') \cdot \pi$$

$$\therefore \quad L_o(x') = \frac{B(x')}{\pi} \tag{10.187}$$

It should now be apparent that Eq. (10.181) can be simplified to

$$B(x') = B_e(x') + \int_{\mathcal{M}} f_{r,d}(x') \cdot B(x) \cdot V(x, x') \cdot G(x, x') \cdot \partial A_{x'} \tag{10.188}$$

where $B(x')$ is the emitted radiosity from a differential patch at point x'.

Since the BRDF $f_{r,d}(x')$ is independent of the incoming and outgoing directions, it can be moved outside the integral. Thus,

$$B(x') = B_e(x') + f_{r,d}(x') \cdot \int_{\mathcal{M}} B(x) \cdot V(x, x') \cdot G(x, x') \cdot \partial A_{x'} \tag{10.189}$$

Using Eq. (10.95) we can rewrite the above equation as

$$B(x') = B_e(x') + \frac{\rho_d(x')}{\pi} \cdot \int_{\mathcal{M}} B(x) \cdot V(x, x') \cdot G(x, x') \cdot \partial A_{x'} \tag{10.190}$$

This equation is known as the *radiosity equation*.

10.13 Monte Carlo Method and Monte Carlo Integration

Before we proceed with examining the implementation of physically based renderers we must introduce a mathematical technique that underlies almost all state of the art physically based rendering algorithms. This technique is known as the *Monte Carlo method*. We will use this method to compute accurate estimates of integrals which are required for the evaluation of the components of the rendering equation. We refer to this technique as *Monte Carlo integration*.

The concept of the Monte Carlo method has existed for a long time, but was first formalised by Metropolis and Ulam [Metropolis 49]. Monte Carlo methods allow us to solve problems by estimating the value of an equation using random numbers. Historically they have been applied to the solution of problems of a probabilistic nature.

Let us now examine how the Monte Carlo method can be used to perform numerical integration (also known as *quadrature*). Given the definite integral of a function $f(x)$ over the interval $[a, b]$ its value, known as the *estimand*, is given by

$$I = \int_a^b f(x) \cdot \partial x \tag{10.191}$$

The value of the integral, representing the area under the graph of the function (see Fig. 10.48), can be estimated by computing the mean value of the

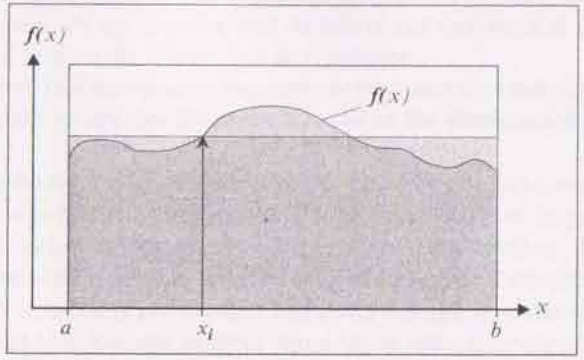

FIGURE 10.48. Monte Carlo Integration.

function $f(x)$ over the interval $[a, b]$ and then multiplying this mean value by the length of the interval $(b-a)$.

The value of this integral can be estimated by picking a uniform random number χ_i from the interval $[a, b]$ and evaluating $f(\chi_i)$. Here the value of the integral of $f(\chi_i)$ is called the *primary estimator* of the integral and is denoted as

$$\langle I \rangle_{prim} = \int_a^b f(\chi_i) \cdot \partial x = (b-a) \cdot f(\chi_i) \tag{10.192}$$

The evaluation of the primary estimator for a specific sample χ_i is known as an *estimate*. Thus, $(b-a) \cdot f(\chi_i)$ gives us an *estimate* of the area under the graph of $f(\boldsymbol{x})$ in Fig. 10.48. If we use N uniformly distributed random sample points $\chi_1, \chi_2, \ldots, \chi_N$ from the range $[a, b]$ to compute N estimates of the integral and average these, then we get a more accurate estimate of the integral $\int_a^b f(\boldsymbol{x}) \cdot \partial x$.

$$\langle I \rangle_{sec} = \frac{1}{N} \cdot \sum_{i=1}^{N} \langle I \rangle_{prim} = (b-a) \cdot \frac{1}{N} \cdot \sum_{i=1}^{N} f(\chi_i) \tag{10.193}$$

This is known as the *secondary estimate* or the *Monte Carlo estimate* of the integral. As we increase the number of samples in the computation of the secondary estimate of the integral, this estimate becomes more accurate and at the limit it becomes equal to the estimand, i.e.

$$\lim_{N \to \infty} \langle I \rangle_{sec} = I \tag{10.194}$$

The standard deviation σ of the secondary estimate $\langle I \rangle_{sec}$ from the true value of the integral I is proportional to the square root of the sample size, i.e.

$$\sigma \propto \frac{1}{\sqrt{N}} \tag{10.195}$$

which means that in order to half the estimation error, we must quadruple the sample size. See [Dutré 94].

10.14 Physically-Based Rendering Algorithms

The eventual output of any computer visualisation process is the production of a rendered image of the input scene. The geometry of a virtual scene can be described by a combination of geometric primitives which can either take the form of a *solid representation* or a *boundary representation*.

A solid representation is usually based on an *implicit functional representation* that describes the shape (i.e. the volume and boundary) of the object in question. A representation of this type is often referred to as an F-rep.

Complex object representations of this form are composed by blending and combining the functional representations of primitive solid objects, such as cubes, cylinders, cones, spheres and super-ellipsoids. Each of these primitives can be represented by a unique implicit function that allows us to test if a given point in space lies inside of, on the surface of or outside the solid object. These simple solid primitives can be combined through blending operations or Boolean algebra operations (such as union, intersection and difference) to form more complex solid objects. Most manufactured objects can easily be composed in this fashion. The F-rep is also ideally suited for the representation of soft objects and ethereal phenomena, such as steam, smoke and fire. It is common to use ray-tracing rendering techniques to render such arbitrary geometries.

A boundary representation, on the other hand, is composed of a set of boundary surfaces that represent the *skin* of the solid object. Such boundary surfaces are represented in the form of parametric equations that describe curved surfaces delimited by their boundary curves or polygons delimited by their outlines. A representation of this type is often referred to as a B-rep.

Boolean algebra operations can also be applied to B-reps to generate more complex forms. With a boundary representation it is much more difficult to determine if a given point in space lies inside, on the surface or outside a solid object. This makes the implementation of Boolean algebra operations on B-reps more difficult. B-reps are ideally suited, however, for the representation of free-form surfaces, which are best suited for the description of bodies of cars, ships and aeroplanes.

Other information that is required by the renderer, in order to be able to render a virtual scene, includes a description of the light scattering properties of the surfaces in the scene, a description of the light sources in the scene and a description of the virtual camera.

The description of the light scattering properties of the various surfaces is provided through the specification of their respective BRDFs, BTDFs and BSDFs, as we have seen earlier in this chapter.

The description of each light source in the scene should include its type (i.e. point, parallel, linear, area or solid shape light source), its position and direction of emission, and its emittance characteristics (i.e. its power or intensity and its colour).

Additionally, for each surface in the scene that is a light emitter we must describe its *emittance distribution function*.

Finally, the description of the virtual camera should include, its position, its direction of view, its up direction and its horizontal and vertical fields of view (i.e. the aspect ratio of the frame), and its resolution.

Having got all this information the renderer evaluates the rendering equation in order to generate an appropriate representation of the illumination in the virtual scene.

Ideally, physically-based renderers should be able to produce a rendering of the virtual scene which is indistinguishable from a photograph of its physical counterpart. Thus, achieving the illusive holy-grail of photorealism. Photorealistic results are best achieved by the accurate evaluation of global illumination effects, such as glossy reflections and caustics (achieved through *specular inter-object illumination*) and soft shadows, indirect illumination and *colour bleeding* (achieved through *diffuse inter-object illumination*). In physically-based rendering correctness of the resulting image is paramount, immaterial of the way in which it is achieved.

There are two main approaches for achieving the production of photo-realistic images. We can produce such images by using either an *object-space approach* or an *image space approach*.

10.14.1 Object-Space Rendering Algorithms

Object-space approaches compute and store a representation of the outgoing radiance function for each surface in the scene, in a pre-processing step. Then, in order to generate an image from a given viewpoint, for each pixel of the image we determine the visible surface or surfaces, using a scan-line, a depth-buffer or a ray-casting visibility algorithm. The average radiance for this pixel is determined by computing the average radiance radiated towards the camera from all the surfaces that are visible through the pixel using the stored radiance values computed in the pre-processing step.

Some examples of object-space rendering algorithms include the basic diffuse radiosity algorithm and other non-diffuse radiosity-style algorithms.

10.14.1.1 The Radiosity Algorithm

The radiosity method was first introduced by Goral et. al [Goral 84]. See also [Cohen 85] and [Nishita 85]. Radiosity is an object-space physically based rendering technique that deals exclusively with perfectly diffuse inter-object reflections (i.e. totally incoherent inter-object reflections). The surfaces of objects in the scene can only be Lambertian reflectors or emitters. Transmitted light and light reflected in a specular fashion cannot be handled by the basic radiosity technique.

In static diffuse scenes, of this sort, the emittance distribution function (EDF) and the BRDFs of the surfaces are direction independent and are only determined by their spatial location and the wavelength of the light incident on them. This simplification implies that the outgoing reflected radiance from a given point on a surface will be perceived with the same intensity and colour regardless of the viewing position. By reducing the dimension of the radiance, employing these simplifications, it becomes feasible to store accurate object-space representations of the radiance of complex scenes. This of course means that the output produced by the pre-processing step can be used to render images from different viewpoints in the scene.

With such diffuse environments it is more appropriate to use the radiosity (i.e. radiant exitance) rather than the radiance to quantify the illumination at a given location and wavelength.

Some non-diffuse radiosity-style algorithms compute the average radiance emitted by each surface in the scene towards all other surfaces in the scene. See [Aupperle 93] and [Stamminger 98]. Some other implementations use an angular rather than a spatial parameterisation to represent the directional dependence of the radiance. See [Immel 86] and [Sillion 91].

The major advantage of representing the scene illumination in object space is that we can reuse the results of the radiance computation, output from the pre-processing step of the these algorithms, to render images of the scene from different viewpoints. Thus these algorithms exhibit *frame-to-frame coherence*. 3D graphics hardware can be used to accelerate the rendering stage of these algorithms.

With these algorithms, a significant proportion of the pre-computed radiance can often be reused even if the geometry of the light emission or the surface scattering properties of the virtual scene are altered. See [George 90], [Chen 90] and [Drettakis 97].

The main disadvantage of the object-space rendering algorithms is the excessive amount of storage required to accurately represent the highly direction dependant radiance functions required to represent specular inter-object illumination. Even for the diffuse inter-object illumination of large virtual scenes, the storage requirements of these algorithms become prohibitive.

10.14.2 Image-Space Rendering Algorithms

Image-space rendering algorithms compute the average incoming radiance at each pixel on the fly, without relying on a pre-processing step to compute an object-space representation of the radiance reflected from each surface of the virtual scene. This family of algorithms solves the rendering equation for each pixel or group of pixels in the rendered image. Although the underlying global illumination method used by all the image-space algorithms is basically the same, the algorithms themselves are quite different. Examples of such algorithms are the *ray-tracing algorithm* [Whitted 80], the *distributed ray-tracing algorithm* [Cook 84], the *path-tracing algorithm* [Kajiya 86][Dutré 94], the *bi-directional path-tracing algorithm* [Lafortune 93][Veach 94], the *Metropolis light transport algorithm* [Veach 97b] and the *photon-mapping algorithm* [Jensen 95a].

The major advantage of the image-space algorithms is that they require very little storage, as they compute the visible surface solution on a per-pixel basis. In general these algorithms are more suitable than the object-space algorithms for more complex virtual scenes and can handle more complex illumination models.

The more sophisticated algorithms of this type can handle both specular inter-object and diffuse inter-object illumination, as well as, deal with volume rendering for participating media.

Let us now examine the family of image-space rendering algorithms in more detail.

10.14.2.1 The Recursive Ray-Tracing Algorithm

The concept of ray tracing first appeared in a 1971 paper by Goldstein and Nagel [Goldstein 71]. In the early eighties, Whitted introduced the concept of recursive ray tracing [Whitted 80].

Ray tracing is a simple and elegant algorithm that is able to handle specular inter-object illumination. Thus, it deals with specular reflection and transmission and generates sharp shadows.

For each pixel of the image, we spawn one *primary ray* that starts from the viewing point and passes through the centre of the pixel. If this ray intersects none of the surfaces in the scene, then the pixel is painted with the background colour and we proceed to the next pixel. If the ray, however, intersects some of the surfaces in the scene, we pick the surface whose intersection point is closest to the observer. Once we have identified the closest surface, we compute its unit normal at the point of intersection. This vector is then used in the shading calculations.

How the light is reflected from a surface depends on the type of surface that we are dealing with. If the surface is diffuse (rough), the reflected light will depend only on the illumination arriving directly from the light sources in the scene. Alternatively, if the surface is specular (smooth), the reflected light will depend on the illumination arriving directly from the light sources and on the indirect illumination arriving on this surface after having been reflected off other surfaces in the scene or been transmitted through the surface (if it is transparent). To discover if the point is directly illuminated, we have to trace a *shadow ray* to each of the light sources. If the surface point is visible from a light source, then it will receive direct illumination from this source, otherwise it will not.

If the surface in question is smooth, then we recursively spawn a ray in the reflection direction and if the surface is transparent, a second ray in the transmission direction. We follow these rays repeating the process recursively until we satisfy one of the recursion termination conditions. The recursion will terminate when one of the three following conditions is met. When the ray hits a rough surface, in which case the light reflected from this surface is returned by the ray since this will be the incoming light at the point of origin of the ray. When the ray misses all surfaces in the scene, in which case the ray returns the background colour. Finally, when a user defined maximum level of recursion has been reached, no new rays are generated. Upon returning from a recursive call we accumulate all the calculated radiance contributions.

An outline of the recursive ray-tracing algorithm is presented in Algorithm 10.1. In this algorithm, the function **emitter_shader()** computes the light emitted by an emitter surface and the function **direct_shader()** computes the direct illumination component of the light reflected from the surface.

```
function recursive_ray_tracing_renderer(scene, image)
{
 for each pixel in the image do
  {
   level = 0;

   generate a primary_ray from the eye to the centre of the
   pixel;

   pixel_colour = trace_ray(primary_ray);
  }
} /* recursive_ray_tracing_renderer */

function trace_ray(ray)
{
 find the closest point of intersection of the ray with a
 surface of the scene;

 if there are no intersections then return(background_colour);

 compute the unit normal of the closest surface at the point
 of intersection;
 if surface is emitter
        then colour = emitter_shader(ray, surface, point);
        else colour = 0;

 for each light source do
  {
   generate a shadow_ray to the light source;

   if light source is visible then
    colour = colour
    + direct_shader(point, surface, normal, light) / n_lights;
  }

 if surface is specular then
  {
   level = level + 1;

   if level > max_level then return(colour);
    {
     generate the reflected_ray;
     colour = colour + trace_ray(reflected_ray);
    }

   if surface is transparent then
    {
     generate the transmitted_ray;
     colour = colour + trace_ray(transmitted_ray);
    }
  }
 return(colour);
} /* trace_ray */
```

Algorithm 10.1 The outline of the recursive ray-tracing rendering algorithm.

398

10.14.2.2 The Distributed Ray-Tracing Algorithm

In 1984, Cook et. al introduced the distributed ray-tracing algorithm [Cook 84]. This algorithm is a refinement of the recursive ray-tracing algorithm that provided correct and easy solutions to a number of previously unresolved problems, including semi-coherent reflections and transmissions, shadows with penumbras, depth of field and motion blur.

With distributed ray-tracing we spawn a number of primary rays for each pixel using an appropriately selected probability. The precise probability distribution of the spawned rays, as well as, their individual direction and origin depend on the effect that we are attempting to simulate. For instance, to get a semi-coherent reflection (i.e. a fuzzy reflection) at a ray intersection point with a diffuse surface instead of spawning a single reflected ray we spawn a number of rays stochastically distributed around the mirror reflection direction. Similarly, to simulate a semi-coherent transmission (i.e. translucency) instead of spawning a single transmitted ray we spawn a number of rays stochastically distributed around the transmittance direction.

To generate penumbras (i.e. soft shadows), which result from the illumination produced by area light sources, instead of spawning a single shadow-ray towards the light source we spawn a number of shadow-rays stochastically distributed on the surface of the area light source. The illumination received from this light source is then made proportional to the number of shadow rays that are unobstructed by other surfaces in the scene.

An outline of the distributed ray-tracing algorithm is presented in Algorithm 10.2. In this algorithm, the function **emitter_shader()** computes the light emitted by an emitter surface and the function **direct_shader()** computes the direct illumination component of the light reflected from the surface.

```
function distributed_ray_tracing_renderer(scene, image)
{
 for each pixel in the image do
  {
   colour = 0;

   for each primary_ray sample do
    {
     generate a primary_ray from the eye through a random
     point in the pixel;

     /* Extension for depth of field */
     perturb the ray to account for the lens position;

     /* Extension for motion blur */
     pick a random time within the inter-frame interval to
     trace the ray;

     colour = colour + trace_ray(primary_ray);
    }
```



```
   pixel_colour = colour / n_ray_samples;
  }
} /* distributed_ray_tracing_renderer */

function trace_ray(ray)
{
 find the closest point of intersection of the ray with a
 surface of the scene;

if there are no intersections then return(background_colour);

compute the unit normal of the closest surface at the point
of intersection;

if surface is emitter then
 {
  if surface is diffuse then
   {
    colour = 0;

    for each ray sample do
     {
      stochastically perturb the ray direction;
      colour = colour +
               emitter_shader(perturbed_ray, surface, point)
               / n_ray_samples;
     }
   }
  else
   colour = emitter_shader(ray, surface, point);
 }
else
 colour = 0;

for each light source do
 {
  direct_colour = 0;

  for every shadow_ray do
   {
    generate a shadow_ray to a stochastically selected point
    on the light source;

    if the light source is visible then
     direct_colour = direct_colour +
                     direct_shader(point, surface, normal,
                                   light) / n_shadow_rays;
   }

  colour = colour + direct_colour / n_lights;
 }

determine if the ray is reflected or absorbed using a Russian
roulette procedure;

if ray is absorbed then return(colour);

generate the reflected_ray;

if surface is diffuse then
```

```
  {
   for each ray sampledo
    {
     stochastically perturb the reflected_ray direction;
     colour = colour + trace_ray(perturbed_reflected_ray)
                     / n_ray_samples;
    }
  }
 else
  colour = colour + trace_ray(reflected_ray);

 if surface is transparent then
  {
   generate the transmitted_ray;

   if surface is diffuse then
    {
     for each ray sample do
      {
       stochastically perturb the transmitted_ray direction;
       colour = colour + trace_ray(perturbed_transmitted_ray)
                       / n_ray_samples;
      }
    }
   else
    colour = colour + trace_ray(transmitted_ray);
  }

 return(colour);
} /* trace_ray */
```

Algorithm 10.2 The outline of the distributed ray-tracing rendering algorithm.

This algorithm represents the first effort in an attempt to introduce a certain degree of physical plausibility in the way the rendering equation is computed. The physically based image-space renderers that we will examine next attempt to be physically correct in the way they evaluate the rendering equation.

10.14.2.3 The Path-Tracing Algorithm

Path tracing was introduced by Kajiya in the mid-eighties to provide an efficient way to solve the rendering equation for both local and global illumination [Kajiya 86].

Path tracing is concerned with solving the integration of light energy resulting from the direct illumination arriving directly from area light sources and the indirect illumination arriving from other surfaces in the scene. These integration problems are solved using the Monte Carlo integration method, which as we have seen in Section 10.13, produces an estimate of the value of an integral by averaging a number of random primary estimates of the value of the integral. In the context of Monte Carlo ray-tracing algorithms, this means that we need to stochastically spawn a large number of rays within the integration domain in order to estimate the value of the integral representing the incoming light energy. As with all Monte Carlo methods, by spawning more stochastically scattered rays

(i.e. by evaluating more randomly selected primary estimates) we improve the accuracy of our result. Thus path-tracing can be seen as a progressive refinement of the distributed ray-tracing algorithm examined above.

In distributed ray-tracing we spawn a number of stochastically positioned primary rays through the pixel. When one of these rays is reflected from or transmitted through a diffuse (rough) surface it spawns a number of secondary rays, which are stochastically distributed around the specular reflection or transmission directions. This process is repeated recursively until it satisfies one of the recursion termination criteria. The recursion termination criteria are one of the following. A reflected or transmitted ray does not intersect any of the surfaces of the scene. Alternatively, a stochastic test, known as *Russian roulette*, is used to determine if a ray is reflected/transmitted or absorbed. When a ray is absorbed, no further rays are spawn. This recursive approach can very quickly lead to a combinatorial explosion of secondary rays. For instance, starting with 100 primary rays, in a perfectly diffuse scene, after the first reflective bounce we get $100^2 = 10{,}000$ rays, after the second reflective bounce we get $100^3 = 1{,}000{,}000$ rays and so on.

To avoid this type of combinatorial explosion the number of rays spawn at each bounce had to be kept to a minimum. Kajiya noticed that it was better to focus the bulk of our computing effort on the first few lighting events that have undergone the smallest number of reflection or transmission bounces. He decided that it was a lot more cost effective to spawn a large number of primary rays though the pixel and only spawn one ray for every secondary bounce stochastically distributed around the reflection or transmission directions. With this approach we could afford to spawn thousands of primary rays through each pixel. It is not uncommon to spawn between 1,000 and 10,000 rays per pixel.

The only problem with this approach is that we need a large number of primary rays to avoid noise in the generated image. This noise is related to the error in estimating the incoming radiance integral due to the use of Monte Carlo integration method. The better behaved the incoming radiance function is, the fewer primary rays we need to estimate its value. A well-balanced radiance function means that that there are few bright highlights coming from specific directions in the scene. Thus, in a perfectly diffuse environment we can get away with using as few as 100 primary rays per pixel. Trying to reduce the variance/standard deviation of the Monte Carlo integration method, thus allowing us to reduce the number of primary rays that are required to produce an accurate picture, remains an active research topic.

An outline of the path-tracing algorithm is presented in Algorithm 10.3. In this algorithm, the function **emitter_shader()** computes the light emitted by an emitter surface and the function **direct_shader()** computes the direct illumination component of the light reflected from the surface.

```
function path_tracing_renderer(scene, image)
{
 for each pixel in the image do
  {
   colour = 0;
```

```
    for each primary_ray sample do
     {
      generate a primary_ray from the eye through a random
      point in the pixel;

      /* Extension for depth of field */
      perturb the ray to account for the lens position;

      /* Extension for motion blur */
      pick a random time within the inter-frame interval to
      trace the ray;

      colour = colour + trace_ray(primary_ray);
     }

    pixel_colour = colour / n_ray_samples;
   }
} /* path_tracing_renderer */

function trace_ray(ray)
{
 find the closest point of intersection with a surface of the
 scene;

 if there are no intersections then return(background_colour);

 compute the unit normal of the closest surface at the point
 of intersection;

 if surface is emitter then
  {
   if surface is diffuse then
    {
     stochastically perturb the ray direction;
     colour = emitter_shader(perturbed_ray, surface, point);
    }
   else
    colour = emitter_shader(ray, surface, point);
  }
 else
  colour = 0;

 for each light source do
  {
   generate a shadow_ray to a stochastically selected point on
   the light source;

   if light source is visible then
   colour = colour + direct_shader(point, surface, normal,
                                   light) / n_lights;
  }

determine if the ray is reflected, transmitted or absorbed
using a Russian roulette procedure;

 if ray is absorbed then return(colour);

if surface is transparent and ray is transmitted then
  {
   compute the transmitted_ray;

   if surface is diffuse then
```



```
    {
     stochastically perturb transmitted_ray;
     colour = colour + trace_ray(perturbed_transmitted_ray);
    }
   else
    colour = colour + trace_ray(transmitted_ray);
  }
 else
  {
   compute the reflected_ray;

   if surface is diffuse then
    {
     stochastically perturb reflected_ray;
     colour = colour + trace_ray(perturbed_reflected_ray);
    }
   else
    colour = colour + trace_ray(reflected_ray);
  }

 return(colour);
} /* trace_ray */
```

Algorithm 10.3 The outline of the path-tracing rendering algorithm.

10.14.2.4 The Bi-directional Path-Tracing Algorithm

The bi-directional path-tracing algorithm was first developed by Lafortune and Willems [Lafortune 93] and a year later it was independently developed by Veach and Guibas [Veach 94]. Although both algorithms achieve very similar results, their underlying theoretical framework is quite different.

As we have seen above, with path tracing the primary estimator of the radiance for a given pixel is calculated by tracing a primary ray from the viewing point through the pixel being considered. At the intersection point of this ray with the surface of the scene, closest to the eye, one or more shadow rays are traced towards each of the light sources to determine the direct illumination contribution of each source. This contribution is only accumulated if the light source and the intersection point are mutually visible. Then, we use the Russian roulette stochastic test to determine if the ray (incident on this surface) is absorbed or if it continues its random walk, being reflected/transmitted from surface to surface. This process is repeated recursively until the ray misses all the surfaces in the scene or is absorbed by a surface. The primary estimator determined in this way by a single random walk is likely to have a large variance (i.e. it is a poor estimate of the true value of the radiance). This large variance is mainly due to the way indirect illumination is sampled. The path-tracing algorithm attempts to remedy this by computing a more accurate secondary estimate that is the average of a large number of primary estimates.

In a scene that is primarily illuminated by indirect illumination very few shadow rays are likely to reach any given light source. This will cause a large variance in both the primary and secondary estimates of the radiance due to direct illumination, resulting in high frequency noise in the rendered image. Bi-directional path tracing attempts to resolve this problem by the following technique. Instead of

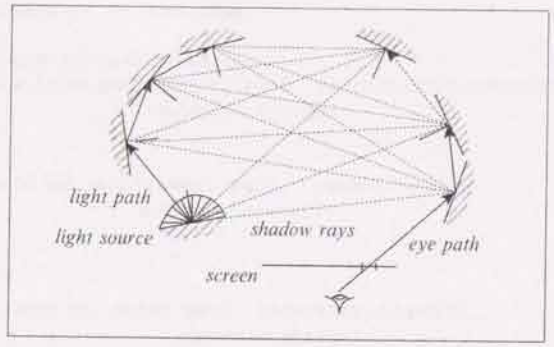

FIGURE 10.49. The geometry of bi-directional path tracing.

just tracing a random walk from the eye into the scene, known as the *eye-path*, in parallel we also trace a random walk from a randomly selected light source into the scene, known as the *light-path*. See Fig. 10.49.

In constructing the light-path, a light source is selected probabilistically depending on its power (brightness). Thus, brighter light sources are more frequently selected than dimmer sources. The starting point on the light source and the starting direction of the light-path are also selected probabilistically depending on the direction of the power distribution of the source. Thus, directions towards which the light source is brighter (i.e. emits more light) are selected more frequently than directions in which the source is dimmer.

In constructing the eye-path, the starting direction of the primary ray from the eye through the pixel is also selected probabilistically according to some random distribution.

The construction of both the eye and light-paths proceeds as follows. When a ray hits a surface it is reflected, transmitted or absorbed, depending on a probability computed by a Russian roulette procedure. This process is repeated recursively until the ray is absorbed or it misses all the surfaces in the scene.

When both the eye and light random walk paths have been constructed, we can proceed with the computation of the primary estimate of the incoming radiance at the pixel by following the eye-path and accumulating the radiance arriving at each hit point (intersection point) on this path. To compute the radiance arriving at a particular hit point, we connect this hit point with every hit point on the light-path (including the first point, which is a point on the light source). This allows us to accurately compute the incoming radiance contributions from both direct and indirect illumination. Each shadow ray that is unobstructed by another surface contributes to this computation. Having computed the incoming radiance at this point, we can, in turn, compute the outgoing radiance from this point. Once we have reached the end of the eye-path, the algorithm returns a primary estimate of the incoming radiance at the pixel.

Once again, as this is a probabilistic estimate, a more accurate secondary estimate of the radiance, incoming at the pixel, can be computed by averaging a large number of primary estimates, which is achieved by spawning a large number of such random walks per pixel.

An outline of the bi-directional path-tracing algorithm is presented in Algorithm 10.4. This algorithm uses three primitive functions that are not shown, namely: **store_light_hit_point()**, **store_eye_hit_point()** and **accumulate_shade()**.

```
function bidirectional_path_tracing_renderer(scene, image)
{
 for each pixel in the image do
  {
   colour = 0;
   for each ray sample do
    {
     light_path = generate_light_path(scene);
     eye_path = generate_eye_path(scene, pixel);
     colour = colour + combine_paths(light_path, eye_path);
    }
   pixel_colour = colour / n_ray_samples;
  }
} /* bidirectional_path_tracing_renderer */
function generate_light_path(scene);
{
 empty light_path;
 stochastically select a light source, a point on the source
 and an initial direction for the light_path;
 trace_light_path(scene, light_path, light_ray, light_energy);
 return(light_path);
} /* generate_light_path */
function trace_light_path(scene, path, ray, incoming_energy);
{
 compute intersection of the light ray with all the surfaces
 in the scene and select the closest intersection_point;
 if no intersections exist then return();
 determine if the ray is to be reflected, transmitted or
 absorbed using a Russian roulette procedure;
 if ray is absorbed then return();
 if ray is transmitted and surface is transparent then
  {
   compute the transmitted_ray and the outgoing_energy from
   the incoming_energy and the surface BTDF;
```

```
    if surface is diffuse then
     {
      perturb transmitted_ray;
      trace_light_path(scene, path, perturbed_transmitted_ray,
                       outgoing_energy);
     }
    else
     trace_light_path(scene, path, transmitted_ray,
                      outgoing_energy);
   }
  else
   {
    store_light_hit_point(path, intersection_point,
                          incoming_energy);

    compute the reflected_ray and the outgoing_energy from the
    incoming_energy and the surface BRDF;

    if surface is diffuse then
     {
      perturb reflected_ray;
      trace_light_path(scene, path, perturbed_reflected_ray,
                       outgoing_energy);
     }
    else
     trace_light_path(scene, path, reflected_ray,
                      outgoing_energy);
   }

  return();

} /* trace_light_path */

function generate_eye_path(scene, pixel);
{
 empty eye_path;

 generate a primary eye_ray from the eye through a random
 point in the pixel area;

 /* Extension for depth of field */
 perturb the eye_ray to account for the lens position;

 /* Extension for motion blur */
 pick a random time within the inter-frame interval to trace
 the eye_ray;

 transmission_factor = 1;

 trace_eye_path(scene, eye_path, eye_ray, transmission_factor);

 return(eye_path);

} /* generate_eye_path */

function trace_eye_path(scene, path, ray, transmission_factor)
{
 compute intersection of the eye ray with all the surfaces in
 the scene and select the closest intersection_point;
```



```
 if no intersections exist then return();

 determine if the ray is to be reflected, transmitted or
 absorbed using a Russian roulette procedure;
 if ray is absorbed then return();

 if ray is transmitted and surface is transparent then
  {
   compute the transmitted_ray and scale the
    transmission_factor using the surface BTDF;

   if surface is diffuse then
    {
     perturb transmitted_ray;
     trace_eye_path(scene, path, perturbed_transmitted_ray,
                    transmission_factor);
    }
   else
    trace_eye_path (scene, path, transmitted_ray,
                    transmission_factor);
  }
 else
  {
   store_eye_hit_point(path, intersection_point,
                       transmission_factor);

   compute the reflected_ray;

   if surface is diffuse then
    {
     perturb reflected_ray;
     trace_eye_path(scene, path, perturbed_reflected_ray,
                    transmission_factor);
    }
   else
    trace_eye_path(scene, path, reflected_ray,
                   transmission_factor);
  }

 return();

} /* trace_eye_path */

function combine_paths(light_path, eye_path)
{
 path_colour = 0;

 trace_path(light_path, eye_path, path_colour);

 return(path_colour);

} /* combine_paths */

function trace_path(light_path, eye_path, colour)
{
 if eye_path.next_node is not empty then
  {
   trace_path(light_path, eye_path.next_node, colour);
```

```
    for each node in the light path do
     {
      generate a shadow ray from the eye_path_node.point to the
      light_path_node.point;

      if these two points are mutually visible then
       accumulate_shade(eye_path_node, eye_path_node, colour);
     }
  }

 return();

} /* trace_path */
```

Algorithm 10.4 The outline of the bi-directional path-tracing rendering algorithm.

The **store_light_hit_point()** function progressively builds the light-path by creating a linked list of light-path hit points, while the **store_eye_hit_point()** function progressively builds the eye-path by creating a linked list of eye-path hit points.

As can be seen from the function **trace_light_path()**, when the light-ray is reflected from an opaque surface, then a hit point is entered in the light-path and the incoming light energy is recorded, and the energy of the reflected ray is attenuated to account for the reflectance characteristics of the surface. But, when the light-ray is transmitted through a transparent surface, then a hit point is not entered in the light-path and the energy of the transmitted ray is attenuated to account for the transmittance characteristics of the surface. See the left-hand side of Fig. 10.50. Figure 10.51a shows that after the light path has been constructed we now have one direct light source and three indirect light sources representing the four hit points on the light-path.

Analogously, as can be seen from the function **trace_eye_path()**, when the eye-ray is reflected from an opaque surface, then a hit point is entered in the eye-path and the cumulative transmission factor is recorded. But, when the eye-ray is transmitted through a transparent surface, then a hit point is not entered in the eye-path and the cumulative transmission factor is attenuated to account for

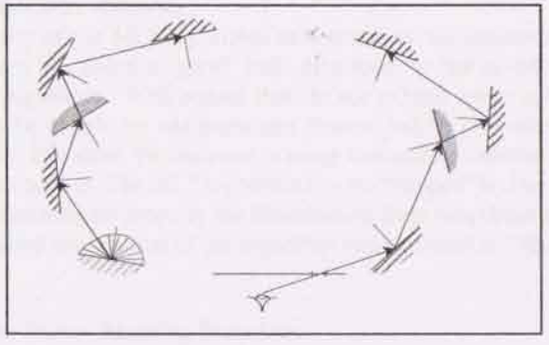

FIGURE 10.50. Tracing the eye-path and the light-path.

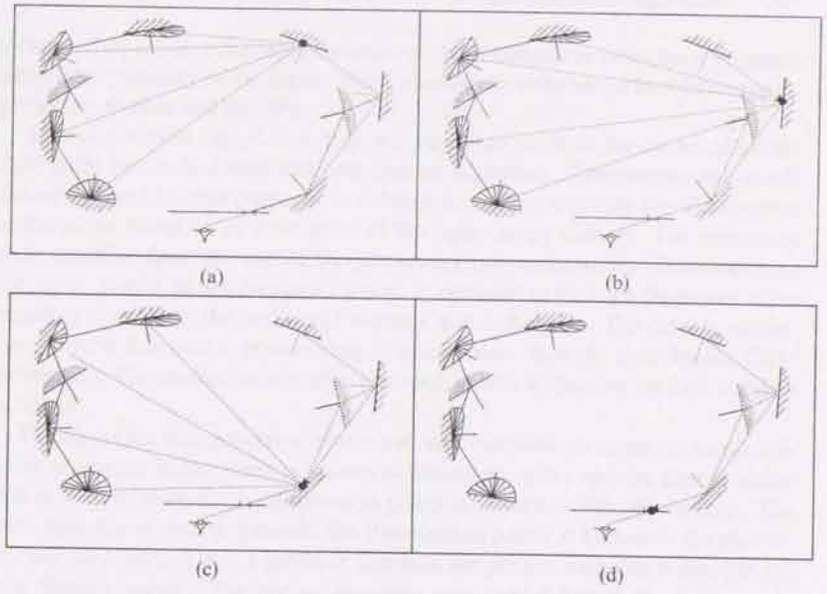

FIGURE 10.51. Tracing the eye-path from the deepest to the shallowest hit point.

the transmittance characteristics of the surface (thus, accounting for the amount of radiance being absorbed by this surface). See the right-hand side of Fig. 10.50.

The **accumulate_shade()** function computes the reflected radiance outgoing from a given eye-path hit point and arriving at the hit point that is positioned immediately before it on the eye-path (i.e. the hit point that is one ray-bounce closer to the eye than the given hit point). Thus, the computation of the reflected radiance proceeds from the deepest eye-path hit point (i.e. the hit point which is the largest number of bounces away from the eye) to the shallowest eye-path hit point (i.e. the hit point which is the smallest number of bounces away from the eye) and finally the eye. As this recursive computation proceeds, indirect illumination arriving from deeper hit points of the eye-path, indirect illumination arriving from light-path hit points and direct illumination arriving from the light source are taken into account. See Fig. 10.51(a)–(d).

A more detailed description of this algorithm and the expressions for the calculation of the illumination contributions can be found in [Lafortune 93].

10.14.2.5 The Metropolis Light Transport Algorithm

The Metropolis Light Transport (MLT) algorithm was introduced by Veach and Guibas [Veach 97b]. This algorithm, which is used to solve the light transport problem, is based on a Monte Carlo statistical simulation approach inspired by the Metropolis sampling technique. The Metropolis sampling technique was first used in computational physics by Metropolis and Ulam while they were working on the Manhattan Project [Metropolis 49], [Metropolis 53].

The general approach used in the MLT algorithm can be outlined as follows. To render an image of a 3D scene, we start with a single light transport path, known as the *current path*, and we progressively generate a set of alternative paths by stochastically mutating the current path. We use an appropriately selected probability, to accept or reject a mutated path, ensuring that the retained paths are sampled with an order that reflects their statistical contribution to the ideal image. Then, we estimate this ideal image by sampling a large number of the mutated paths and by recording their positions on the image plane that is represented by a 2D array in memory.

The MLT algorithm is unbiased, it is capable of handling the most general geometric and BSDF models, it is economical in storage and it can be significantly faster than other unbiased algorithms. The MLT algorithm is very different from both the path-tracing and the bi-directional path-tracing algorithms. Unlike other Monte Carlo methods, instead of randomly sampling the value of a function in order to estimate the value of its integral, the Metropolis method generates a distribution of samples to the unknown function value. To achieve this sampling distribution the MLT algorithm starts with a random sampling of the space of all light paths in the scene. These initial paths are generated using the bi-directional path-tracing algorithm and are subsequently cloned and mutated in order to compute the radiance of the final image.

The most important advantage of the MLP algorithm is that it explores the path space locally, selecting more frequently mutations arrived at by applying minor incremental modifications to the current path. Using this progressive refinement approach has the following beneficial consequences. The average cost for each sample path is relatively small, as very few rays are used. Once an important path is identified, nearby paths are employed as well, thus spreading the cost of determining such a "good" path over many neighbouring paths. The set of mutation operations applied, by the MLT algorithm, to a "good" path is easy to extend. By selecting mutations that retain certain of the properties of a given "good" path while changing other properties, we can take advantage of any type of coherence that is present in the scene. In this way, it is often possible to deal with different types of lighting problems more effectively by designing specialised mutations to handle these particular situations.

The propensity of the MLT algorithm to concentrate on incremental changes to a path, once it has found a "good" path, also leads to one of the main weaknesses of this algorithm. With scenes that do not exhibit space coherence, the algorithm may be caught by one particular feature and be prevented from converging quickly. Consider, for instance, a scene containing a surface with a grid of holes lit from behind. The MLT algorithm can be "trapped" by one of the holes and will fail to investigate properly the illumination from neighbouring holes.

A more detailed explanation of the algorithm can be found in [Veach 97a] and [Veach 97b].

10.14.2.6 The Photon-Mapping Technique

The photon-mapping technique was developed by Jensen and Christensen as an efficient alternative to pure Monte Carlo ray-tracing techniques [Jensen 95a]. This technique de-couples the representation of the illumination from the representation of the geometry of the scene. Thus, allowing us to handle arbitrarily complex geometric models and BRDFs.

To best visualise the *photon map* we may think of it as the cache of all the light paths in the bi-directional path-tracing algorithm. The photon map could indeed be used for this purpose. It is however used to estimate the illumination in the scene based on an estimation of the light energy density. The estimation error resulting from the use of the photon map, to estimate the illumination of the scene, results in low frequency noise, as opposed to the high frequency noise resulting from using the traditional Monte Carlo techniques. The density estimation method that uses a photon map is much faster than the pure Monte Carlo techniques. The main disadvantage, however, of this estimation method is that it is biased.

The algorithm that generates, stores and uses illumination as points on the surfaces of objects in the scene is known as *photon mapping* and the data structure that is used to store these illumination points is known as the photon map. The technique that is used to generate the illumination points is known as the *photon-tracing algorithm*. Thus, a renderer that uses the photon-mapping technique has two distinct passes. The *photon-mapping pass*, which builds the photon map data structure by spawning *photon rays* from the light sources and tracing them through the objects in the scene and the *rendering pass*, which renders the scene using the illumination information stored in the photon map (thus speeding up the rendering process).

As this has proved to be a very influential algorithm, we will examine it in some detail.

10.14.2.6.1 The Photon-Mapping Pass

The photon-mapping pass is an essential pre-processing step of any rendering algorithm that uses the photon-mapping technique. During this pass photons are emitted from the light sources, their paths are traced through the scene and when they hit a diffuse surface their location and power are recorded in the photon map.

10.14.2.6.2 Emission of Photons

A large number of photons are emitted by each light source in the scene. The power (i.e. the wattage) of a light source is divided equally among all the photons that it emits. Thus, each emitted photon transports a fraction of the power of the light source. The Jensen-Christensen model supports many different types of light source.

Diffuse point light sources emit photons uniformly in all directions using one of two Monte Carlo sampling techniques. *Explicit sampling*, which randomly selects two spherical coordinate angles, and *rejection sampling*, which randomly generates points inside a unit cube and selects the first such point that lies inside the unit sphere.

Spherical light sources emit photons in all directions. First a random point is selected on the surface of the light source sphere and then a random direction is

selected on the hemisphere above this point. A similar procedure is followed for polygonal *square light sources*.

For *directed light sources*, which are used to simulate very distant light sources, we enclose the scene in a bounding sphere which when projected onto the ground plane produces a circle. Random points in this circle can be used as the terminating points of incoming photon beams from the direction of the parallel light source.

Complex three-dimensional shapes can also be used as light sources. In this case, the photon ray emission points and directions are selected using a rejection-sampling scheme.

10.14.2.6.3 Scattering and Tracing of Photons

After a photon is emitted by a light source, it is traced through the scene by the photon-tracing algorithm. The photon tracing algorithm works in a very similar fashion to a ray tracing algorithm, except that photons distribute flux while rays accumulate radiance. This is significant in the case of refraction, where radiance changes according to the relative refractive index of the interface surface while flux does not.

When a photon arrives at an interface surface it can be reflected, transmitted or absorbed. This determination is made using a Russian roulette procedure, which acts as an importance-sampling technique. Here, a probability distribution function serves to eliminate the statistically insignificant parts of the domain of our problem.

10.14.2.6.4 Storage of Photons

As we have seen above, when a photon hits a specular surface it can be reflected, transmitted or absorbed. When it hits a non-specular surface, however, it is stored in the photon map. See Fig. 10.52. Photons represent incoming illumination (flux) at a given point on the surface. Thus, we can use the photons, stored in the photon map, to approximate the reflected illumination at several points on the surface.

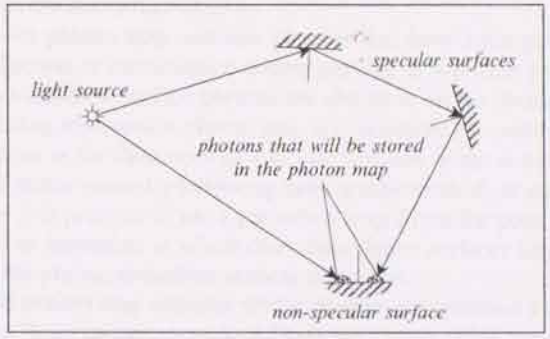

FIGURE 10.52. Photon tracing.

The photon map is stored as a *left-balanced kd-tree* data structure, which is very efficient to traverse [Bentley 79]. Essentially, the kd-tree is an axis-aligned *binary space partition tree* (*BSP-tree*). Each node of this tree stores a photon. Each photon is represented by the x, y, z coordinates of the point of incidence of the photon ray on a surface (which is stored as three floats), by its power (which is stored as four bytes), by its direction vector (which is given by the θ and ϕ angles and stored in compressed form as two bytes) and by a kd-tree flag (which is stored as a short).

Once the photon-tracing algorithm is completed the kd-tree is balanced to speed up the random access of its nodes.

10.14.2.6.5 Photon Density Estimation

The photon map represents the incoming flux on the surfaces of the scene. Each photon can be thought of as transporting a package of energy that represents a fraction of the power of the light source that emitted it. Thus, the photon map contains information indicating that a given region of the scene has received some direct or indirect illumination from a light source.

Looking at a single photon we can not tell how much light a given region has received and we must compute the photon density $\partial\Phi/\partial A$ and to estimate the irradiance for a small region surrounding a given point on a surface of the scene. We can approximate the incoming flux $\Phi_i(\boldsymbol{x})$ at a point $\boldsymbol{x}$ on a surface of the scene by finding the n photons, stored in the photon map, which are the closest neighbours of this point. All these photons will be enclosed in a sphere of radius r_x and each photon will have power $\Delta\Phi_p(\omega_p)$. See Fig. 10.53.

Now, the outgoing radiance $L_r(\boldsymbol{x}, \omega_x)$ reflected from this point can be approximated as

$$L_r(\boldsymbol{x}, \omega_x) \approx \sum_{p=1}^{n} f_r(\boldsymbol{x}, \omega_p, \omega_x) \cdot \frac{\Delta\Phi_p(\omega_p)}{\Delta A} \tag{10.196}$$

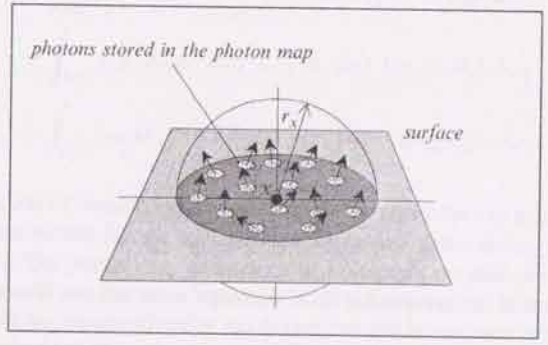

FIGURE 10.53. Photons in the neighbourhood of point $\boldsymbol{x}$.

If we assume that the region around point $\boldsymbol{x}$ is flat, then the intersection of the surface and the sphere containing the photons can be taken as

$$\Delta A = \pi \cdot r_x^2 \tag{10.197}$$

Substituting Eq. (10.197) into Eq. (10.196) we get an estimate of the outgoing reflected radiance from point $\boldsymbol{x} \cdot \nabla$ Thus,

$$L_r(\boldsymbol{x}, \omega_x) \approx \frac{1}{\pi \cdot r_x^2} \sum_{p=1}^{n} f_r(\boldsymbol{x}, \omega_p, \omega_x) \cdot \Delta\Phi_p(\omega_p) \tag{10.198}$$

The accuracy of this estimate depends on the number of photons in the photon map. The larger this number the better the estimate.

10.14.2.6.6 The Rendering Pass

The photon map created during the photon-tracing pass can now be used to render an image of the scene. The renderer is composed of a simple ray tracer that uses the radiance estimate to determine the reflected radiance component due to all the diffuse reflections and a recursive ray tracer that determines the reflected radiance component due to all the specular reflections and transmissions.

Unlike other Monte Carlo ray-tracing techniques, photon mapping is ideally suited for rendering *caustics*. Caustics occur when light that has been reflected from or transmitted through one or more smooth (specular) surfaces reaches a rough (diffuse) surface.

To improve the quality of the rendered diffuse inter-object reflections, often perceived as colour bleeding, we need to increase the number of photons that are stored in the photon map and the number of photons that are included in the neighbourhood of a given point when computing the diffuse component of the reflected radiance estimate at this point.

For scenes that exhibit an even balance of specular and diffuse reflections we can use two photon maps. One map, known as the *global photon map*, to store the indirect illumination, and a second map, known as the *caustics photon map*, to store caustics. The caustics photon map can be used by the recursive ray tracer that computes the direct illumination.

The caustics photon map contains photons that have undergone at least one specular reflection or transmission before arriving at a diffuse surface. After a collision with a diffuse surface photons are absorbed. In the photon-tracing pass, while populating the caustics photon map it is desirable to concentrate the emission of photons in the directions of specular surfaces in the scene. These could be identified either manually (allowing more artistic control) or automatically by the renderer. It is possible to use a projection map (from the point of view of the light source) to determine in which directions shinny surfaces lie so that we can concentrate the photon emissions in these directions.

The global photon map contains all the photons that reached a diffuse surface in the scene. These photons represent direct and indirect illumination, as well as caustics. The rendering algorithm must ensure that the caustics term is not added more than once in the rendering equation. The global photon map is populated by tracing photons towards all the surfaces in the scene and storing them when they reach a diffuse surface. Such photons may be absorbed or further reflected or refracted to reach other surfaces.

The final image is rendered using a distributed ray-tracing algorithm. Here, the incoming radiance at each pixel is computed by averaging a large number of sample estimates. As we have seen earlier in this chapter, a BRDF, f_r, is often composed of a specular term, $f_{r,s}$, and a diffuse term, $f_{r,d}$. Thus,

$$f_r(\boldsymbol{x}, \omega_i, \omega_o) = f_{r,s}(\boldsymbol{x}, \omega_i, \omega_o) + f_{r,d}(\boldsymbol{x}, \omega_i, \omega_o) \tag{10.199}$$

Similarly, the incoming radiance can be thought of as the sum of three incoming radiances. The *incoming direct illumination radiance* $L_{i,l}(\boldsymbol{x}, \omega_i)$, which is illumination arriving on the surface directly from the light sources. The *incoming caustics illumination radiance* $L_{i,c}(\boldsymbol{x}, \omega_i)$, which is indirect illumination arriving on the surface as a result of one or more specular reflections or transmissions. The *incoming indirect illumination radiance* $L_{i,d}(\boldsymbol{x}, \omega_i)$, which is indirect illumination arriving on the surface as a result of one or more diffuse inter-object reflections or transmissions. Thus, the incoming radiance can be written as

$$L_i(\boldsymbol{x}, \omega_i) = L_{i,l}(\boldsymbol{x}, \omega_i) + L_{i,c}(\boldsymbol{x}, \omega_i) + L_{i,d}(\boldsymbol{x}, \omega_i) \tag{10.200}$$

Recall, from Section 10.7.3, that the outgoing reflected radiance from a point $\boldsymbol{x}$ on a surface is given as

$$L_o(\boldsymbol{x}, \omega_o) = \int_{\mathcal{H}_i^2} f_r(\boldsymbol{x}, \omega_i, \omega_o) \cdot L_i(\boldsymbol{x}, \omega_i) \cdot (\omega_i \odot N_x) \cdot \partial\omega_i \tag{10.201}$$

where N_x is the unit normal vector of the surface at point $\boldsymbol{x}$.

Substituting Eqs. (10.199) and (10.200) into Eq. (10.201), we get

$$\begin{aligned} L_o(\boldsymbol{x}, \omega_o) = & \int_{\mathcal{H}_i^2} f_r(\boldsymbol{x}, \omega_i, \omega_o) \cdot L_{i,l}(\boldsymbol{x}, \omega_i) \cdot (\omega_i \odot N_x) \cdot \partial\omega_i \\ & + \int_{\mathcal{H}_i^2} f_{r,s}(\boldsymbol{x}, \omega_i, \omega_o) \cdot \left[L_{i,c}(\boldsymbol{x}, \omega_i) + L_{i,d}(\boldsymbol{x}, \omega_i) \right] \cdot (\omega_i \odot N_x) \cdot \partial\omega_i \\ & + \int_{\mathcal{H}_i^2} f_{r,d}(\boldsymbol{x}, \omega_i, \omega_o) \cdot L_{i,c}(\boldsymbol{x}, \omega_i) \cdot (\omega_i \odot N_x) \cdot \partial\omega_i \\ & + \int_{\mathcal{H}_i^2} f_{r,d}(\boldsymbol{x}, \omega_i, \omega_o) \cdot L_{i,d}(\boldsymbol{x}, \omega_i) \cdot (\omega_i \odot N_x) \cdot \partial\omega_i \end{aligned} \tag{10.202}$$

This equation or some approximation of it is used by the ray tracer to compute each sample estimate for the radiance incoming at a given point. At the closest intersection of the primary ray emanating from the eye, we evaluate Eq. (10.202). Ideally we would use the same equation at all subsequent ray bounces (ray hits), but this would be computationally too expensive and in any case would not be the most appropriate use of computing time. Thus, we try to use the accurate computation as infrequently as possible and use an approximate computation in all other cases. An accurate computation is only used on the first bounce of the primary

ray and on any subsequent bounces where the ray-surface intersection point is closer to the ray-origin than a given threshold. This latter condition is necessary to enhance the likelihood of accurate colour-bleeding occurring at convex corners of the scene, where the distance between two ray bounces is short.

Next, let us consider the individual components of the outgoing radiance from Eq. (10.202).

The *direct illumination reflected radiance* term, $L_{o,l}(\boldsymbol{x}, \omega_o)$, is given by

$$L_{o,l}(\boldsymbol{x}, \omega_o) = \int_{\mathcal{H}_i^2} f_r(\boldsymbol{x}, \omega_i, \omega_o) \cdot L_{i,l}(\boldsymbol{x}, \omega_i) \cdot (\omega_i \odot N_x) \cdot \partial\omega_i \qquad (10.203)$$

This term is frequently the most important part of the outgoing reflected radiance, since it is responsible for depicting shadows (to which the eye is most sensitive). So it must be computed accurately.

From every intersection point $\boldsymbol{x}$ shadow rays are cast towards each light source in the scene. With area light sources, more than one shadow rays are required per light source so as to generate convincing penumbra areas. This is an expensive process. In an attempt to improve its efficiency the algorithm can be modified to emit shadow photons (anti-photons) with negative light energy [Jensen 95b]. In the photon-tracing pass, when the photon map is being populated, shadow photons are emitted by each light source. When a shadow-photon ray is cast, starting from the second closest ray-surface intersection and including all subsequent intersections we deposit a negative photon if the intersected surface is facing the light source. In the rendering pass, if all the photons in the region of a surface point $\boldsymbol{x}$ are positive, then the point is deemed to be visible by the light source, if all the neighbouring photons are negative, then it is deemed to be hidden from this light source, otherwise it is deemed to be in the penumbra region of this light source. In the latter case we have to use a number of rays to discover the fraction of lighting that the point receives from this light source. Thus, we are only forced to perform the expensive shadow-ray casting operation for the regions of the scene that fall within the penumbra areas associated with this particular light source. This implementation of the algorithm requires a modification of the photon data structure to identify the light source that emitted the photon and to indicate whether the photon is positive or negative.

The *specular illumination reflected radiance* term, $L_{o,s}(\boldsymbol{x}, \omega_o)$, of the outgoing radiance of Eq. (10.202) is given by

$$L_{o,s}(\boldsymbol{x}, \omega_o) = \int_{\mathcal{H}_i^2} f_{r,s}(\boldsymbol{x}, \omega_i, \omega_o) \cdot \left[L_{i,c}(\boldsymbol{x}, \omega_i) + L_{i,d}(\boldsymbol{x}, \omega_i) \right] \cdot (\omega_i \odot N_x) \cdot \partial\omega_i \qquad (10.204)$$

This integral is evaluated using a Monte Carlo ray-tracing algorithm with an *importance sampling* optimisation, which is based on the specular BRDF $f_{r,s}$. Importance sampling is an optimisation technique employed to improve the performance of the Monte Carlo method [Jensen 95c].

The *caustic illumination reflected radiance* term, $L_{o,c}(\boldsymbol{x}, \omega_o)$, of the outgoing radiance of Equ. (10.202) is given by

$$L_{o,c}(\boldsymbol{x}, \omega_o) = \int_{\mathcal{H}_i^2} f_{r,d}(\boldsymbol{x}, \omega_i, \omega_o) \cdot L_{i,c}(\boldsymbol{x}, \omega_i) \cdot (\omega_i \odot N_x) \cdot \partial\omega_i \qquad (10.205)$$

When an accurate value for $L_{o,c}$ is required, then we evaluate this integral using Monte Carlo integration and the contents of the caustics photon map. When only an approximate value for $L_{o,c}$ is required, then we do not evaluate this equation at all but we rely on the caustics contribution included in the specular radiance estimate from the global photon map.

Finally, the *indirect illumination reflected radiance* term, $L_{o,d}(\boldsymbol{x}, \omega_o)$, of the outgoing radiance of Equ. (10.202) is given by

$$L_{o,d}(\boldsymbol{x}, \omega_o) = \int_{\mathcal{H}_i^2} f_{r,d}(\boldsymbol{x}, \omega_i, \omega_o) \cdot L_{i,d}(\boldsymbol{x}, \omega_i) \cdot (\omega_i \odot N_x) \cdot \partial\omega_i \qquad (10.206)$$

This outgoing radiance term represents light that since leaving the light source has been reflected, at least once, from a diffuse surface, resulting in incoherent (soft) illumination. When an accurate value for $L_{o,d}$ is required, then we evaluate this integral using Monte Carlo ray tracing. This is done by spawning a large number of rays, stochastically distributed around the reflection/transmission direction, and averaging the computed radiance from all the primary estimates. When only an approximate value for $L_{o,d}$ is required, then we compute it using a radiance estimate from the global photon map, which contains the direct, indirect and caustic illumination contributions.

10.14.2.6.7 Observations

Photon mapping is a very elegant technique that allows us to handle many different lighting phenomena and to generate photo-realistic images that are physically based or at least physically plausible. In conclusion we note that:

- Photon mapping is an elegant rendering technique that provides a complete global illumination solution for large scene geometries with complex material properties.
- The photon mapping technique separates the storage of the photons from the storage of the geometric representation of the scene. With complex scenes this represents a clear advantage over object-space finite element methods of rendering.
- Any rendering algorithm that uses photon maps must perform a final gather (aggregation) of light flux which requires a number of expensive near neighbourhood operations.
- Several optimisations exist that speed up the performance of this algorithm significantly.

A more detailed description of this technique can be found in Jensen's book [Jensen 01b] on which the above discussion is based.

10.14.3 Hybrid Multi-Pass Rendering Algorithms

A promising approach is to use a hybrid algorithm for rendering, which involves multiple passes. The first pass of such an algorithm adopts an object-space approach and deals with the diffuse inter-object illumination problem, by computing and storing the outgoing radiance diffusely reflected from the surfaces of the scene. The second pass of such an algorithm adopts an image-space approach and deals with specular inter-object illumination, by accessing the pre-stored object-space solution and by computing (on the fly) for each pixel the radiance contributed by the specular inter-object illumination, which would be impossible to compute and store in the first pass.

Such algorithms can be designed to take advantage of the strengths of both object-space and image-space techniques.

Examples of such multi-pass algorithms are found in [Wallace 87], [Sillion 89], [Chen 91], [Jensen 96] and [Suykens 99].

References

[Amanatides 84] Amanatides, J. Ray tracing with cones. *Proceedings of SIGGRAPH 84. Computer Graphics*, Vol. 18, No. 3, p.p. 129–135, 1984.

[Ashikmin 00a] Ashikmin, M. and Shirley, P. An anisotropic Phong light reflection model. Technical Report UUCS-00-014, Computer Science Department, University of Utah, p.p. 1–11, June 2000.

[Ashikmin 00b] Ashikmin, M. and Shirley, P. An anisotropic phong BRDF model. *Journal of Graphics Tools*, Vol. 5, No. 2, p.p. 25–32, 2000.

[Ashikmin 00c] Ashikmin, M., Premoze, S., and Shirley, P. A microfacet-based BRDF generator, *Proceedings of SIGGRAPH 2000, Computer Graphics, ACM Press/ACM SIGGRAPH/Addison Wesley Longman*, p.p. 65–74, 2000.

[Aupperle 93] Aupperle, L. and Hanrahan, P. A hierarchical illumination algorithm for surfaces with glossy reflection". *Proceedings of SIGGRAPH 93, Computer Graphics*, Vol. 27, No. 4, p.p. 155–162 1993.

[Beckmann 63] Beckmann, P. and Spizzichino, A. *The Scattering of Electromagnetic Waves from Rough Surfaces*. Pergamin Press, Oxford, England (1963).

[Bennett 61] Bennett, R. A. and Porteus, J. O. Relation between surface roughness and specular reflectance at normal incidence, *Journal of the Optical Society of America*, Vol. 51, p.p. 123–129, 1961.

[Bentley 79] Bentley, J. L. and Friedman, J. H. Data structures for range searching, *Computing Surveys*, Vol. 11, No. 4, p.p. 397–409, 1979.

[Blinn 77] Blinn, J. F. Models of light reflection for computer synthesized pictures. *Proceedings of SIGGRAPH 77, Computer Graphics*, Vol. 11, No. 2, p.p. 192–198, 1977.

[Chen 90] Chen, S. E. Incremental radiosity: An extension of progressive radiosity to an interactive image synthesis system. *Proceedings of SIGGRAPH 90, Computer Graphics*, Vol. 24, No. 4, p.p. 135–144, (1990).

[Chen 91] Chen, S. E., Rushmeier, H. E., Miller, G., and Turner, D. A progressive multi-path method for global illumination, *Proceedings of SIGGRAPH 91, Computer Graphics*, Vol. 25, No. 4, p.p. 165–174, 1991.

[Clarke 85] Clarke, F. J. J. and Parry, D. J. Helmholz reciprocity: Its VIvidity and application to reflectometry, *Lighting Research and Technology*, Vol. 17, No. 1, p.p. 1–11, 1985.

[Cohen 93] Cohen, M. F. and Wallace, J. R. *Radiosity and Realistic Image Synthesis*, Academic Press, San Diego, CA (1993).

[Cohen 85] Cohen, M. F. and Greenberg, D. P. The hemi-cube: A Rdriosity solution for complex environments. *Proceedings of SIGGRAPH 85, Computer Graphics*, Vol. 19, No. 3, p.p. 31–40, 1985.

[Cook 81] Cook, R. L. and Torrance, K. E. A reflection model for computer graphics. *Proceedings SIGGRAPH 81, Computer Graphics*, Vol. 15, No. 4, p.p. 307–316, 1981.

[Cook 82] Cook, R. L. and Torrance, K. E. A reflection model for computer graphics. *ACM Transactions on Graphics*, Vol. 1, No. 1, p.p. 7–24, 1982.

[Cook 84] Cook, R., L., Porter, T. and Carpenter, L. Distributed ray tracing, *Proceedings of SIGGRAPH 84, Computer Graphics*, Vol. 18, No. 3, p.p. 137–145, 1984.

[Ditchburn 76] Ditchburn, R. W. *Light*, Vols. 1 and 2. Academic Press, London (1976).

[Dorsey 99] Dorsey, J., Edelman, A., Jensen, H. W., Legakis, J., and Pedersen, H. K. Modelling and rendering of weathered stone. *Proceedings of SIGGRAPH 99*, pp. 225–234. Addison-Wesley, Reading, MA (1999).

[Drettakis 97] Drettakis, G. and Sillion, F., X. Interactive update of global illumination using a line-space hierarchy. *Proceedings of SIGGRAPH 97, Computer Graphics*, Vol. 31, No. 4, p.p. 57–64, 1997.

[Duderstadt 79] Duderstadt, J. J. and Martin, W. R. *Transport Theory*. John Wiley & Sons, New York (1979).

[Dutré 94] Dutré, Ph. and Willems, Y. D., Importance-Driven Monte Carlo light tracing. *Proceedings of the Fifth Eurographics Workshop on Rendering*, Darmstadt, Germany, Eurographics Association, p.p. 185–194 (1994).

[Dutré 98] Dutré, Ph. Mathematical Framework and Monte Carlo Algorithms for Global Illumination in Computer Graphics. Ph.D Thesis, University of Leuven (1998).

[George 90] George, D. W., Sillion, F. X., and Greenberg, D., P. Radiosity Redistribution for Dynamic Environments, *IEEE, Computer Graphics and Applications*, Vol. 10, No. 4, p.p. 26–34, July 1990.

[Goldstein 71] Goldstein, R. A. and Nagel, R. 3-D Visual Simulation. *Simulation*, p.p. 25–31 Jan 1971.

[Goral 84] Goral, C. M., Torrance, K. E., Greenberg, D. P., and Battaile, B. Modeling the interaction of light between diffuse surfaces. *Proceedings of SIGGRAPH 84, Computer Graphics*, Vol. 18, No. 3, p.p. 213–222, 1984.

[Hall 89] Hall, R. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York (1989).

[Hanrahan 93] Hanrahan, P. and Krueger, W. Reflection from layered surfaces due to subsurface scattering. *Proceedings of SIGGRAPH 93*, 165–174, ACM Press, New York (1993).

[Immel 86] Immel, D. S., Cohen, M. F. and Greenberg, D. P. A radiosity method for non-diffuse environments. *Proceedings of SIGGRAPH 86, Computer Graphics*, Vol. 20, No. 4, p.p. 133–142, 1986.

[Jenkins 76] Jenkins, F. A. and White, H. E. *Fundamentals of Optics*. McGraw-Hill, New York (1976).

[Jensen 95a] Jensen, H. W. and Christensen, N. J. Photon maps in bi-directional Monte Carlo ray tracing of complex objects. *Computers & Graphics*, Vol. 19, No. 2, p.p. 215–224, 1995.

[Jensen 95b] Jensen, H. W. and Christensen, N. J. Efficiently rendering shadows using the photon map. *Proceedings of CompuGraphics 95*, p.p. 285–291 Dec 1995.

[Jensen 95c] Jensen, H. W. Importance driven path tracing using the photon map. *Proceedings of the Eurographics Rendering Workshop 95*, Eurographics Association, p.p. 326– 335, June 1995.

[Jensen 96] Jensen, H. W. Global illumination using photon maps. *Proceedings of the Eurographics Rendering Workshop 1996*. Springer-Verlag, Vienna, p.p. 21–30, 1996.

[Jensen 99] Jensen, H. W., Legakis, J., and Dorsey, J. Rendering of wet materials, In: C. Lischinski and G. W. Larson (Eds.), *Rendering Techniques'99*. Springer-Verlag, Vienna (1999).

[Jensen 01a] Jensen, H. W., Marschner, S., Levoy, M., and Hanrahan, P. A practical model for subsurface light transport. *Proceedings of SIGGRAPH 2001*, p.p. 399–408. Addison-Wesley, Reading MA (2001).

[Jensen 01b] Jensen, H. W. *Realistic Image Synthesis using Photon Mapping*. A. K. Peters Ltd., Natick, Massachusetts (2001).

[Kajiya 86] Kajiya, J. T. The rendering equation. *Proceedings of SIGGRAPH 86, Computer Graphics*, Vol. 20, No. 4, p.p. 143–150, 1986.

[Lafortune 93] Lafortune, E. P. and Willems, Y. D. Bi-directional path tracing. *Proceedings of the International Computer Graphics and Visualisation Techniques, CompuGraphics 93*. Avlor, Portugal, p.p. 145–153 1993.

[Lafortune 94] Lafortune, E. and Willems, Y. Using the modified Phong reflectance model for physically based rendering, Technical Report CW 194, Department of Computer Science, K.U., Leuven, p.p. 1–18, Nov 1994.

[Lafortune 97] Lafortune, E. P. F., Foo, S. C., Torrance, K. E., and Greenberg D. P. Non-linear approximation of reflectance functions. *Proceedings of SIGGRAPH 97, Computer Graphics*, No. 31, p.p. 117–126, 1997.

[Lewis 93] Lewis, R. R. Making shaders more physically plausible. *Proceedings of the Fourth Eurographics Workshop on Rendering*, Paris, France. Eurographics Series EG 93 RW, pp. 47–62 June 1993.

[Metropolis 49] Metropolis, N. and Ulam, S. The Monte Carlo method, *Journal of the American Statistical Association*, Vol. 44, No. 247, p.p. 335–341, 1949.

[Metropolis 53] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. Equations of state calculations by fast computing machines, *Journal of Chemical Physics 21*, p.p. 1087–1091, 1953.

[Neumann 99a] Neumann, L., Neumann, A., and Szirmay-Kalos L. Compact metallic reflectance models. *Computer Graphics Forum (Eurographics '99)*. The Eurographics Association and Blackwell Publishers, Vol. 18, No. 3, p.p. 161–172, 1999.

[Neumann 99b] Neumann, L., Neumann, A., and Szirmay-Kalos L. Reflectance models with fast importance sampling, *Computer Graphics Forum*, Vol. 18, No. 4, p.p. 249–265, 1999.

[Nicodemus 77] Nicodemus, F. E., Richmond, J. C., Hsia, J. J., Ginburg, I. W., and Limperis, T. Geometric considerations and nomenclature for reflectance, *Monograph 161*, National Bureau of Standards (US) Oct 1977.

[Nishita 85] Nishita, T. and Nakamae, E. Continuous tone representation of three-dimensional objects taking account of shadows and inter-reflection. *Proceedings of SIGGRAPH 85, Computer Graphics*, Vol. 19, No. 3, p.p. 23–30, 1985.

[Palik 85] Palik, E. D. *Handbook of Optical Constants of Solids*. Academic Press, New York, NY (1985).

[Pharr 00] Pharr, M. and Hanrahan, P. Monte Carlo evaluation of non-linear scattering equations for subsurface reflection. *Proceedings of SIGGRAPH 2000*, pp. 75–84. Addison-Wesley, Reading, MA (2000).

[Sancer 69] Sancer, M. I. Shadow-corrected electromagnetic scattering from a randomly rough surface, *IEEE Transactions on Antennas and Propagation*, Vol. 17, No. 5, p.p. 577–585, 1969.

[Schlick 93] Schlick, C. A customizable reflectance model for everyday rendering. *Proceedings of the Fourth Eurographics Workshop on Rendering*. Series EG 93 RW. Paris, France, p.p. 73–84 June 1993.

[Shirley 91] Shirley, P. Physically Based Lighting Calculations for Computer Graphics, Ph.D Thesis, University of Illinois at Urbana Champaign Jan 1991.

[Sillion 89] Sillion, F. and Puech, C. A general two-pass method integrating specular and diffuse reflection. *Proceedings of SIGGRAPH 89, Computer Graphics*, Vol. 23, No. 4, p.p. 335–344, 1989.

[Sillion 91] Sillion, F., Avro, J., Westin, S., and Greenberg, D. P. A global illumination solution for general reflectance distributions. *Proceedings of SIGGRAPH 91, Computer Graphics*, Vol. 25, No. 4, p.p. 187–196 1991.

[Stamminger 98] Stamminger, M., Slusallek, Ph., and Seidel, P. H. Three point clustering for radiance computations. *Proceedings of the Eurographics Rendering Workshop 98*. Eurographics Association, p.p. 211–222, 1998.

[Suykens 99] Suykens, F. and Willems, Y. D. Weighted multi-pass method for global illumination. *Proceedings of Eurographics 99, Computer Graphics Forum*, Vol. 18, No. 3, p.p. 209–220, 1999.

[Torrance 67] Torrance, K. E. and Sparrow, E. M. Theory of off-specular reflection from roughened surfaces, *Journal of the Optical Society of America*, Vol. 57, No. 9, p.p. 1105–1114, 1967.

[Veach 94] Veach, E. and Guibas, L. J. Bi-directional estimates for light transport. *Proceedings of the Fifth Eurographics Workshop on Rendering*, Darmstadt, Germany. Eurographics Association, p.p. 147–162 1994.

[Veach 97a] Veach, E. Robust Monte Carlo Methods for Light Transport. Ph.D Thesis. Department of Computer Science, Stanford University (1997).

[Veach 97b] Veach, E. and Guibas, L. J. Metropolis light transport. *Proceedings of SIGGRAPH 97, Computer Graphics*, Vol. 31, No. 4, p.p. 65–76, 1997.

[Wallace 87] Wallace, J. R., Cohen, M. F., and Greenberg, D. P. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. *Proceedings of SIGGRAPH 97, Computer Graphics*, Vol. 21, No. 4, p.p. 311–320, 1987.

[Whitted 80] Whitted, T. An improved illumination model for shaded display, *Communications of the ACM*, Vol. 23, No. 6, p.p. 343–349, 1980.

[WWW 1] http://www.luxpop.com