A Ray Tracing Framework for Global Illumination Systems

Peter Shirley *
Department of Computer Science
Indiana University
Bloomington, IN 47405, USA

Kelvin Sung †
William Brown ‡
Deptartment of Computer Science
University of Illinois
Urbana, IL 61801, USA

Abstract

The fundamental software components useful for a zonal ray tracing system are described. The interface protocols and some implementational observations are outlined for each of the key components. Components for sampling, ray-object intersection, and zonal (radiosity) calculations are emphasized. Some results from a global illumination program assembled from the components are discussed.

CR Categories and Subject Descriptors: I.3.0 [Computer Graphics]: General; I.3.6 [Computer Graphics]: Methodology and Techniques.

Additional Key Words and Phrases: Ray tracing, radiosity, object-oriented design, software components, zonal method, visual realism.

1 Introduction

In recent years many researchers have investigated ray tracing solutions to the global illumination problem [54, 13, 12, 29, 53, 52, 1, 47, 25]. Other authors have applied object-oriented design to ray tracing programs [32, 24]. In this paper, we discuss the major components[1] of an object oriented ray tracing system for global illumination.

There are several existing systems that enable programmers to construct user interfaces from basic components (e.g menus, scrollbars, windows). In a similar fashion, graphics programmers should be able to compose a global illumination renderer from basic components. Such components are specified only by their interface protocols, so different implementations of the same component can be invisibly exchanged in a particular system, which allows isolated testing of particular component implementations. This, of course, is simply an instance of abstract data typing.

The design strategy outlined above is not novel. We are simply applying this strategy to the global illumination problem, and thus identifying the key components of such a system. In this paper, we focus on the functionality and interface protocols of these components. We also discuss some implementation strategies for particular components.

The fundamental components of a distributed ray tracing system are described in Section 2. This extends the basic framework presented by Kirk and Arvo [32]. In Section 3 we describe the new components that are needed for zonal calculations, and discuss how the ray tracing components can be reused. Some results using our components and some open issues are discussed in Section 4.

The terminology we use in this paper is borrowed from C++, however the ideas should be easily translated to most object-oriented languages. The word *class* will be used for particular components and their associated implementation. As discussed in [55], an *abstract class* specifies the behavior of an object (i.e. a component interface) without defining how the behavior should be implemented and a *subclass* is a specialization of an abstract class because it defines the implementation of the behavior.

*E-mail: shirley@cs.indiana.edu
†E-mail: ksung@cs.uiuc.edu
‡Now at Mentor Graphics. E-mail: brown@cs.uiuc.edu

[1] Components are defined by their behavior and their interface protocols. This is the same as the *black box framework* described by Johnson and Foote [28].

2 Ray Tracing Components

In this section, we describe the basic components of our ray tracing framework. In Section 2.1 we present several simple classes that are useful in graphics programs. The components which manage sample distributions on the pixel are discussed in Section 2.2. The ray-object intersection component is described in Section 2.3. The ray-material interaction component, our variation of a shader [51, 23], is outlined in Section 2.4.

2.1 Graphics Utility Classes

Some utility classes are obviously needed: points, vectors, 4 by 4 transformation matrices, and colors. Operations involving these classes are defined using operator overloading mechanisms. Only operations that have mathematical meaning are allowed, as suggested by Goldman [19]. The allowed expressions include addition and subtraction on vectors, addition and subtraction between a point and a vector, subtraction between points, and operations between a scalar and a vector. Denying operations such as the addition of two points makes mistakes in expressions easier to catch. Unfortunately, it also makes it difficult to take the centroid of a set of points. However, it is always possible to create a special purpose function to perform such a task.

Colors have the operators of addition, multiplication, division, and subtraction with themselves, and multiplication and division with scalars. Initially, a RGB color model was used. This was later replaced by a very general spectral model, where each color was approximated by a piecewise linear approximation with arbitrary node locations. When two colors were combined by an operation, a new color with possibly more nodes was generated. This meant that unbounded lists had to be used to store the node locations and amplitudes. Though this general color representation was well suited for complicated spectral distributions, such as the emission spectrum of a fluorescent light, and was spatially efficient for simple spectra, the time needed to manage the color variables was too large to justify the switch from RGB (typically, the run time more than doubled). Next, we switched to twenty evenly spaced nodes to represent color spectra. Because this had a high storage cost (especially for images using zonal calculations), the four unevenly spaced node locations suggested by Meyer was used [34]. There was no qualitative loss in image quality observed going from the twenty to four samples, but this may say more about the arbitrary nature of most of the input spectral curves than about the quality of the spectral approximation. We suspect that filtering the input spectra before point sampling them will avoid most problems associated with using only a few sample locations.

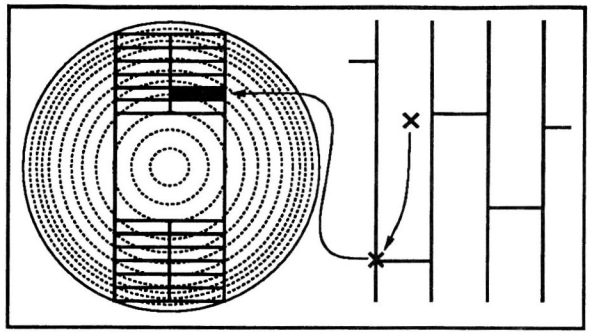

Figure 1: Use one corner of a board to find a solid noise seed point for an index into the tree.

Another basic utility class is the ray class, represented by a point of origin, and a vector representing direction. The ray class has no allowed operations, but does have the basic 'method' (member function) that finds the point a certain distance along the ray. We have found that it is helpful to associate two other characteristics with a ray. The first is a material id which stores the material the ray is in (e.g. 'this ray is now traveling through glass'). The second characteristic is the attenuation of the ray. This makes the bookkeeping associated with adaptive ray tree pruning [22] straightforward.

A utility class that has been surprisingly useful is an orthonormal basis of three vectors. Though any two of these vectors uniquely defines the basis, all three are explicitly stored, trading space for execution time. These bases are used in viewing calculations, and to assign a local coordinate system to a surface.

Another utility class is the (u, v) pair. This class is useful for pixel sampling and texture mapping. The texture class itself adds another useful utility. In this implementation, the texture abstract class takes both a point in 3D, and a (u, v) pair. Surface textures and solid textures [38] are both subclasses of the texture class. A solid texture will use the point for texture generation, and the surface texture will use the (u, v) coordinate for color lookup. This 'send all information, needed or not' strategy can be inefficient, but it is a simple way to guarantee that the needed information is passed to a texture module.

A very useful utility class is a solid noise generator. We have implemented the generator given in Perlin's 1989 paper [39], and found it to be mechanical to implement. While it has been demonstrated that solid noise is useful in generating uniform textures such as marble, we have found that these techniques are too simple for semistructured patterns such as board floors, brick walls, and carpeted floors. Each element of these patterns (e.g. an individual brick) can be modeled using various transformations on solid noise. To accomplish this, one adds one

or more randomly or semi-randomly varying parameters to make the elements vary among themselves.

As an example, a single board has a random grain pattern associated with a plane passing through a tree. The woodgrain of a tree is modeled in a manner similar to Perlin's marble: the basic structure of the tree is light and dark concentric rings, where the area (not distance) between rings is roughly constant. The radius of a given ring will decrease slightly as it moves up the tree (it is visually important that the rings are not perfect cylinders). Noise is used to add some irregularity to the geometry of the rings. The particular location of a board in a tree is what gives it its unique character. Since the tree is of finite volume there are a finite set of specific boards that come form a tree, and each can be given a specific integer id.

We create our board texture by putting down an algorithmic (predefined) board pattern, and associating a specific board id to each board. This is easily accomplished by using the solid noise at a particular corner of the board as a seed to generate the board location in the tree (Figure 1). This guarantees that all points on a board are mapped to the same id in the tree. This procedure gives some irregularity to what types of boards (fine-grained versus loose-grained) are adjacent in the pattern. To ensure that there is no visible correlation between the ids of adjacent boards, we scale the corner points by some large factor before calling the solid noise function.

A similar approach can be used to generate bricks. Solid noise evaluated at one corner of a brick will give a seed to control some global parameter for that brick. If more than one parameter is needed, the other corners (or midpoints, etc.) can be also used for solid noise seed points. All of the textures in Figures 7 and 6 used this type of procedural texture.

To generate the colors of wood or brick, we attempt to use physical spectral curves. Unfortunately this data can be hard to come by, and directly specifying spectral curves is not intuitive. In these cases we generate curves by 'mixing' standard artist's pigments, the curves for which can be found in [9]. When we have existing RGB data, we use Glassner's conversion method [18]. It has been our experience that using smooth curves for this conversion is highly preferable to using impulses.

2.2 Sample Point Generation

A basic module in any ray tracing code selects sample points on the pixel (in some systems the samples are chosen more globally on the screen). In a stochastic ray tracing system, sample points must be chosen from a larger multidimensional space (e.g. screen-lens-reflection-shadow-time space). Cook recommends choosing sample point module one or two dimensions at a time and then combining these choices for full multidimensional sample distributions. For example, we might choose screen locations uniformly on the pixel area, and lens points uniformly on the lens disk. These points could then be paired randomly or deterministically to form sample points in four-dimensional screen-lens space. Cook calls his specific version of this method *uncorrelated jittering*. There has been much discussion on how to get a 'good' set of sample points on a square two-dimensional region [13, 35].

Many spaces we need to sample (e.g lens area, reflection ray direction) are not square. One way to generate sample points on a non-square region is a special purpose algorithm, such as the one used by Cook for reflected rays [12]. The other is to generate points on the square and then apply a warping function so that their distribution is changed. An example of this method is the transformation used by Ward et al. to generate a cosine distribution of sample points on a hemisphere [53]. The first abstraction is to make the sampling distribution come from an abstract class. This lets the user flexibly choose and add sampling methods and filter functions in a natural way.

In this section we discuss two basic modules useful for sample point generation. The first generates sample points on the unit square, and the second module acts as a filter to create specific sample point distributions by 'warping' uniform distributions.

2.2.1 Uniform Sample Point Generation

The sample-generator module is passed a non-negative integer, n, and generates n (u, v) sample points that are equidistributed on the unit square ($0 \leq u \leq 1, 0 \leq v \leq 1$). The basic implementations of this module are commonly regular sampling, jittering, and poisson disk sampling. All of these strategies have problems. Regular sampling can cause extreme aliasing. Jittering requires n to be a perfect square for optimum performance, or requires some factoring system otherwise ($n = 17$ would imply a 1 by 17 sampling partition). Poisson disk has non-deterministic running time, and choosing the disk radius is not straightforward. Mitchell addressed the first problem with an approximation to poisson disk sampling based on error diffusion [35].

We have several strategies, and most often use *n-rooks sampling*. This scheme, to our knowledge, was first used in the OPTIK system [56], and first reported in the Monte Carlo literature by Kalos [30]. N-rooks is a variation of Cook's uncorrelated jittering. We generate the u coordinates of the n points by jittering in one dimension, and then generate n v coordinates in the same way. We then randomly link the u and v coordinates to form n (u, v) pairs. This forms n sample points in a way that

is unbiased in relation to the probability of selecting any particular point. Another way to look at the sample generation is to draw an n by n chessboard, and randomly place n rooks on the board subject to the constraint that the rooks cannot capture each other in one move. This results in a pattern where every row and column has exactly one rook. We now select one (u, v) pair from each square where a rook sits.

The n-rooks strategy is really a simplification of uncorrelated jittering (rather than a generalization), because none of the dimensions are really linked. N-rooks has the advantages that it can generate a set of n samples for any n, that it takes a deterministic amount of time, and that there are no parameters that need to be set other than n. It is not surprising that n-rooks performs well for pixels containing horizontal or vertical edges, because it fully jitters each dimension. To our surprise, it has so far almost always been more accurate than other sampling methods on the images we have tested. These test results, and some analysis on why this might be true can be found in [45, 48].

2.2.2 Sample Distribution Transformations

Suppose we want to apply weighted area averaging to a pixel rather than simple area averaging [14]. We could simply apply weights to each sample point based on its (u, v) coordinate. If we sample each pixel independently it is better to apply importance sampling by placing the sample points in a way that is distributed according to the weighting function [12]. In this section, we describe a module that takes a set of n uniformly distributed points, and warps them in such a way that they are distributed according to the desired weighting function.

As a simple example, suppose we want the trivial 'box' weighting function, which simply samples the pixel area uniformly ($w(x, y) = 1$ inside the pixel). The warping module would take n sample points generated by a module described in the last section, and apply the mapping $x = u - 0.5$, $y = v - 0.5$. This assumes we are using a coordinate system where the pixel center is the origin and the pixel width is 1.

In practice we will want non-uniform weighting functions. For example, suppose we have the width 2 weighting function:

$$w(x, y) = (1 - |x|)(1 - |y|) \qquad (1)$$

If we just want to generate independent random points with density w, we apply standard techniques to transform canonical[2] random numbers into these points [33]. Since this particular w is separable, we can generate x according to $w(x) = (1 - |x|)$, and y the same way. We

[2] A canonical random number ξ is uniformly distributed between 0 and 1.

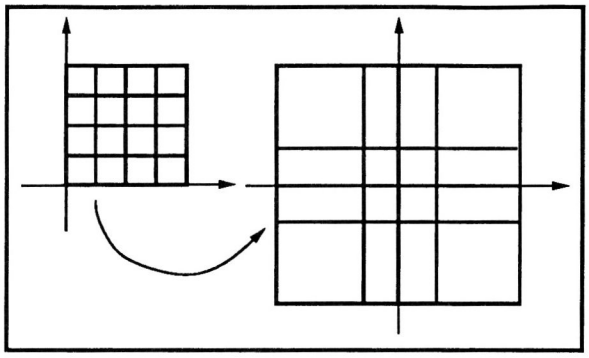

Figure 2: A transformation for nonuniform filter sampling.

define the distribution function, F, associated with w:

$$F(x) = \int_{-1}^{x} (1 - |x'|)dx' = \frac{1}{2} + x - \frac{1}{2}x|x| \qquad (2)$$

To get our desired independent (x, y) distributed according to w, we simply take canonical (ξ_1, ξ_2) and apply $x = F^{-1}(\xi_1)$, and $y = F^{-1}(\xi_2)$, where F^{-1} is the inverse function of F.

This idea can be extended by taking our set of sample points and transforming them as if they were canonical. This will preserve some of the good qualities of the original sampling distribution, but will have our desired distribution properties. Applying this idea to the filter in Equation 1, we can take our n uniform sample points and apply the transformation:

$$x = \begin{cases} -1 + \sqrt{2u} & \text{if } u < 0.5 \\ 1 - \sqrt{2(1-u)} & \text{if } u \geq 0.5 \end{cases}$$

followed by a similar transformation involving y and v. This transformation on 16 jitter cells is shown in Figure 2. Note that this will transform nonuniformly within each cell, so the sample point selection must take place before the transformation.

The uniform sampling transformation can also apply to non-separable density functions, and to functions defined on non-Cartesian manifolds. This requires dealing with the joint distribution function and non-constant metrics, but otherwise the same techniques apply. Details can be found in [44, 45].

We have found several other transformations to be useful. For example, to choose points uniformly from a disk of radius R, apply the transformation $\theta = 2\pi u$, $r = R\sqrt{v}$. To choose points in a cosine distribution on a hemisphere $(p(\theta, \phi) = (1/\pi)\cos\theta)$, apply the transformation $\theta = \arccos(\sqrt{1-u})$, $\phi = 2\pi v$. A generalization is to choose directions according to a 'phong' distribution $(p(\theta, \phi) = ((n+1)/(2\pi))\cos^n\theta)$, by applying the transformation $\theta = \arccos((1-u)^{1/(n+1)})$,

$\phi = 2\pi v$. To choose random points on a triangle defined by vertices p_0, p_1, and p_2, apply the transformation: $a = 1 - \sqrt{1-u}$, $b = (1-a)v$, and the random point p will is: $p = p_0 + a(p_1 - p_0) + b(p_2 - p_0)$.

2.3 Ray-Object Intersection

From a functional point of view, the ray-object intersection component simply finds the first object, if any, hit by a ray[3]. With this simple definition, this component can be implemented as a black box framework [28]. Our goal here is to present the design of a framework to facilitate the independent development of different algorithms for this component. We use the *Faster Ray Intersection Techniques* as described by [8] as a base to discuss our design and show that our design unifies the approaches. This section concludes with some implementation observations.

2.3.1 Class Hierarchy

Heckbert observed that geometrical objects should be viewed as basic components with common interface protocols, so that a ray tracing system can be designed independently from the geometrical primitive types (e.g. spheres, polygons) [24]. Like Kirk and Arvo [32], we have implemented geometrical primitives (e.g. sphere, polygon) and collection structures (e.g. octree, bounding volume) as subclasses of the same *geom-object* abstract class. In this way, they can have the same interface protocols and behaviors. This allows a particular collection structure (e.g. octree) to contain, as members, other collection structures (e.g. bounding volume, regular grids). For example, both geometrical primitives and collection structures respond to the *Hit(ray)* message by returning the *geom-object* hit by the input ray. Since both geometrical primitives and collection structures are subclasses of geom-object, the result of *Hit* message could be either. In this way, it is possible to have nested octrees or have regular grid nested inside an octree structure. Intelligent use of this generality can reduce the system execution time. Unfortunately, just how to attain intelligent use is not obvious [32].

This general grouping of spheres, polygons, octrees, etc. emphasizes an important point: classes with common access (member) functions should be subclasses of the same abstract class, even if their underlying representations are very different. Classes with different access rules, even if their underlying representations are identical, should not be grouped together, as seen with points and vectors in the previous section.

[3] If we want to do constructive solid geometry, we might want the intersection routine to find the list of all intersection points[40].

2.3.2 Functional Modules

It is well known that linear exhaustive testing of primitives with rays for potential intersection is inadequate. The geometric model should be processed into some internal representation (e.g. collection structures like octree) for efficient candidate primitive look up during intersection testings. Since current ray tracers and zonal renderers only model geometrical optics [29], the *processed internal representation* could be viewed as an efficient *primitive/geometric characteristic* association list storage and look up mechanism.

From the design level, the ray-object intersection component consists of three functional modules: Builder, Structurer, and Traverser. For each primitive, the builder module asks for certain geometric characteristic from the primitive, and sends requests to the structurer module to associate the characteristic with the primitive. The traverser module extracts the relevant geometric characteristic from an input ray and requests the structurer module to look up the associated primitives stored. It is a primitive's responsibility to calculate for possible intersections. Under this design, the ray-object intersection component becomes the *coordinator* that assists and thus speeds up the process of searching for potential intersecting primitives.

The acceleration techniques for faster intersection calculations as described in [8] are unified under this design:

Bounding Slabs [31]. The *Builder module* asks for the bounding slab distances from the primitive. In this case, the bounding slab is the 'geometric characteristic'. The *Structurer module* associates the primitive with the corresponding slab and slab distances. The construction of the bounding slab hierarchy is also the responsibility of the structurer module. The *Traverser module* takes a ray geometry and pierces the bounding slab hierarchy to locate potential intersecting primitives. Notice that a bounding slab is also a *primitive* in the sense that it must respond to the *Hit* protocol message by returning the primitives (including children slabs) bounded by the current slab.

Spatial Subdivision [17, 15]. The *Builder module* asks for the extent from a primitive and determines which spatial cell should be associated with the primitive. The actual implementation of spatial cell and the association between the primitive is performed by the *Structurer module*. The Structurer module stores the association of primitives and spatial cell units in such a way that later retrieval of primitives can be done efficiently (when given a ray). The *Traverser module* takes a ray geometry and traverses the spatial structure to locate the potential intersection primitives. Again, a spatial cell unit must be able to respond to the *Hit* protocol message by returning primitives (including children spatial cells in the case of adaptive subdivision algorithms) contained in the current

spatial cell.

New algorithms could be formulated by identifying the geometric characteristic that the approach is taking advantage of. For example, the *ray coherence theorem* [36] uses a direction and an acute angle as the geometric characteristic. In this case, the builder module finds out the list of candidate primitives that are visible from a direction/angle pair of a primitive. When tracing a secondary ray, the traverser module uses the ray direction and reflecting angle to request the structurer module to extract the candidate primitives associated with the corresponding direction/angle pair for potential intersection calculations.

The reason to separate this component into three modules is such that the implementation of each module can be isolated and replaced without affecting the others. For example, for an octree spatial subdivision implementation, the builder can build the octree structure statically before ray tracing starts, or dynamically during ray tracing. The structurer module is the underlying implementation of an octree, this could be built around a hash table, or a hierarchy of pointers. The traverser module knows how to traverse an octree, but does not need to know the detailed implementation of the octree structure. Some of the examples of octree traversers are: tree walking [16], Glassner's algorithm [17], and DDA octree traverser [50].

2.3.3 Implementation Notes

Following the design approach described in this section, it is possible to replace a module in this component with the rest of the system remaining unchanged. As a result, we are able to isolate and observe the effect of different algorithms.

Octree Traversal. It is believed that the original octree traversal algorithm [17] can be improved by realizing *tree location coherence* [16]. In his original octree approach [17], Glassner proposed always beginning the search for next octant from the root of an octree. It has been pointed out by various researchers [27, 16] that theoretically, starting the next octant search from the parent of current octant should be faster (this approach has been called *tree walking* [16]). We note that the only difference between the two algorithms is in the process of getting the next octant: tree walking recognizes the tree location coherence and starts searching from the parent of the current octant, while Glassner's algorithm always starts searching from the root of the octree. A fact that generally has been overlooked is that it takes extra time for the tree walking algorithm to ascend the octree [7]. When traversing between two octants of different first level parents, the two algorithms take the same amount of time to descend the octree, but tree walking needs extra time to ascend the octree. The recursion the extra time needed to ascend the octree usually causes the implementation of the tree walking algorithm to result in a slower system [50].

Mail box. The mail box concept was proposed independently by [5, 4]. The idea is to avoid multiple ray-object intersection calculation between the same object and the same ray in different spatial cell units. The underlying assumption is that the primitive objects usually stretch across many spatial cell units, thus the time saved in avoiding multiple intersection calculations offsets the overhead involved in maintaining the extra information. We note that when the size of the objects in a scene is small in comparison to the size of the spatial cell units, the probability of objects spanning multiple spatial cell boundaries also becomes small. The overhead involved in performing the pre-intersection checking and post-intersection information recording eventually offsets the time saved in avoiding the small number of multiple intersection calculations. In some cases, the mailbox implementation actually results in a slower system [50], and its utility may be highly dependent on the general spatial character of the geometric primitives used.

2.4 Ray-Material Interaction

Several researchers have noted that reflection behavior should be encapsulated as one unit of a rendering system [32, 51, 23]. We treat light-material interaction as a component, where the reflection behavior is determined strictly from a set of material parameters. Traditionally this might be accomplished with parameters including ks, kt, and kd [21]. One problem with such an approach is that physically implausible parameter combinations can be chosen by the user (e.g. kd = ks = 0, kt = 1). Implausible combinations may be useful for many applications, but if realism is desired, we think it is better to limit the user's choices.

We have used the idea that materials can be classed as families, each grouped by the parameters that affect their behavior. This way the user only needs to choose the relevant parameters for a particular material. Once the material is chosen it is treated as a black box component that responds to a limited protocol (much like geometrical objects for ray-object intersection). The first way in which a material can be queried is, given an incoming ray r, a point p on the surface, and a surface normal $\vec{n}$, asking what rays r_i are reflected/transmitted by the material, and what is the attenuation k_i for each ray'. This will allow us to handle building the ray propagation code. For other lighting calculations, such as the direct lighting component, we need to ask a material, 'what is your radiance, $L(\vec{v}_{out})$, that comes from a source of radiance $L(\vec{v}_{in})$ that subtends a solid angle ω?'. We also need to

ask a material if it is a luminaire (source of light), and if so, how much light it emits in a particular direction.

The materials that we have implemented are:

conductor: Parameters n (refractive index), k (extinction coefficient), e (phong-style exponent). Example: aluminum.

dielectric: Parameters n (refractive index), a (filter coefficient), e (phong-style exponent). Example: glass.

lambertian: Parameter k_d (diffuse coefficient). Example: matte paint.

polished: Parameters k_d (diffuse coefficient of substrate), n (refractive index of polish), e (phong-style exponent). Example: gloss paint.

translucent: Parameters k_d^1 (diffuse coefficient of first side), k_d^2 (diffuse coefficient of first side), kt (transmission coefficient). Example: lampshade.

luminaire: Parameters k_d (diffuse coefficient), e (phong-style exponent). Example: light bulb.

These basic materials can be extended, but they have proven to be fairly good approximations to common real world materials. Conductors are sometimes a little difficult because the parameters n and k are not intuitively controllable. We have found most of the data we use for conductors in [37]. The behavior of both conductors and dielectrics is determined using the Fresnel Equations, the full form of which can be found in [49, 45]. The polished surface is an approximation to a diffuse substrate with a thin dielectric covering. This means that for a given direction we first calculate the specular reflectivity $k_s(\theta)$, and then the remaining light is reflected diffusely, giving a diffuse reflectance of $(1 - k_s(\theta))k_d$. This allows glare effects to be approximated accurately. The phong-style exponent e is used to allow some spread in the reflected component of conductors, dielectrics, and polished materials. For smooth surfaces e is set to a large number.

The translucent surface reflects light diffusely from either side, and also allows some light to be diffusely transmitted. The luminaire acts as a diffuse reflector, and also emits power in a phong-style distribution. Large exponents are used if spot lights are desired.

The ray reflection/transmission behavior can be summarized as:

conductor: Generate one reflected ray with attenuation determined by Fresnel Equations. Perturb ray randomly if $e \neq \infty$.

dielectric: Generate one reflected ray and one transmitted ray with attenuations determined by Fresnel Equations. Perturb both rays randomly if $e \neq \infty$.

lambertian/luminaire: Generate one reflected ray randomly with a cosine distribution.

polished: Generate one reflected ray from the polish with attenuation determined by Fresnel Equations. Perturb ray randomly if $e \neq \infty$. Generate one reflected ray from the diffuse substrate randomly with a cosine distribution.

translucent: Generate two rays randomly, one reflected, one transmitted, each with a random cosine distribution.

It is useful to be able to turn off reflections from a particular material. We allow this to be done when the material is initialized. A conventional Whitted-style ray tracer would turn off reflections for the lambertian, translucent, and luminaire surfaces, and would turn off reflections from the substrate (but not the polish) of the polished surfaces. To maintain some form of dependent sampling, such as uncorrelated jittering, the reflection protocol should also accept a canonical (u, v) pair (Section 2.2), to be used as a basis for any probabilistic reflection that might occur.

3 Zonal Calculations

Several recent zonal[4] systems are based on progressive refinement techniques [11, 2]. The theoretical basis for such systems is straightforward to extract. The progressive refinement technique can be viewed as power transport simulation, which implies fairly direct non-diffuse zonal solutions [6, 47, 20, 46, 42]. These solutions are easy to construct if we view the zone as a black box which collects power carrying rays, and later emits a group of power carrying rays that represent reflected power accumulated since the previous emission step.

This abstraction underlying zonal calculations can be stated: zones should receive, accumulate, and send power, and the mechanics of how this happens should be hidden. This is accomplished by defining a zonal-data module. In addition, the module should, after the zonal calculations are completed, be able to provide the radiance of the patch when viewed from a certain direction.

For a lambertian zone, the zonal-data module is easy to implement because there is no dependency on the incoming direction of intensity. We need to store the total power, Φ, and the unsent accumulated power, Φ_u. Each new incoming ray carrying power Φ_i will imply $\Phi = \Phi + k_d\Phi_i$ and $\Phi_u = \Phi_u + k_d\Phi_i$. When it is time for the zone to emit, it will send N rays each carrying power Φ_u/N. These rays will be sent in a cosine distribution. Just as was done with pixel sampling in Section 2.2,

[4] In a zonal system lighting information is stored at a finite set of zones. Radiosity programs use zonal methods.

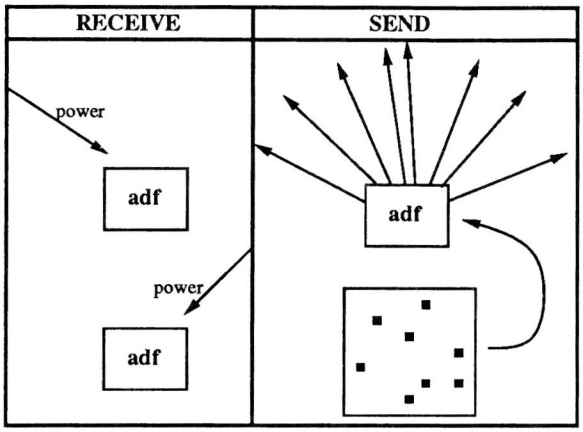

Figure 3: The two crucial methods of an adf (angular distribution function): receive power, and send power according to some set of sample points.

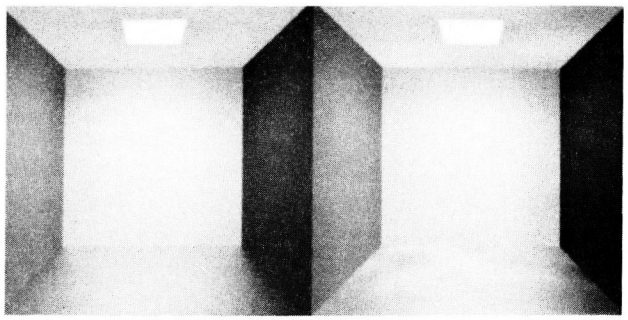

Figure 4: The imperfect floor on the right is modeled with zones, while the one on the left uses distributed ray tracing. All diffuse surfaces use zones with no directional tables.

we can derive N (u, v) pairs and then transform these to the appropriate (θ, ϕ) pairs. The radiance of the zone will just be $\Phi/(\pi A)$, where A is the area of the zone.

For a zone with directionally dependent reflection behavior, such as brushed steel, we must maintain the total and unsent power as some kind of directional table [20, 46, 42]. A simple way to do this is a spherical coordinate array of bins, with the total power going through each bin. The unsent and total power of the diffuse case must be generalized to a new black box, the angular distribution function (adf). This function maintains whatever information is necessary to receive power, and later send power as a set of rays (Figure 3). Other possible tables include hemicubes [26] and spherical harmonics [10].

The receiving method of a directionally dependent adf can be implemented by using the ray-material interaction module of Section 2.4. Simply reflect the incoming ray using the ray-material interaction module as a black box, and add the attenuated reflected power to whichever angular bin(s) the reflected ray(s) land in. The sending stage can be implemented using the warping methods of Section 2.2.2, or by independently sending a pattern of N rays from each angular bin. Because the adf actually stores spectral values, it can only be converted to a probability density by converting the entries to scalars. We use luminance to do this.

It would be very wasteful of storage to store an explicit directional table for diffuse surfaces, though that would work. We therefore implement a lambertian adf by storing only the total and accumulated power. The black box interface still looks the same to the zonal module. To accomplish this in an extendible way, we add an access function to the ray-material interaction class which tells whether the reflection behavior is directionally dependent or independent, and whether the surface is reflective or reflective and transmissive. If the material is reflective and directionally independent (e.g. lambertian), it will use an adf module that stores only total and unsent power. If it is reflective and transmissive and directionally independent (e.g. translucent), then these quantities will be maintained both above and below the surface. If directionally dependent, the directional tables will be maintained either for $(0 < \theta < \pi/2)$ or $(0 < \theta < \pi)$ depending on whether the material is transmissive. If the material can provide an estimate of specularity (e.g. the phong exponent), then this can be used to choose the resolution of the table.

Once the adf modules have been initialized in each zonal module, their internal representation is invisible. This allows the programmer to detach the local lighting models from the global light transport. Figure 4 shows an environment containing both zones with and without directional tables.

The zonal-data abstract class accomplishes two important functions. First, it removes any reference to surface reflection type from the light transport code. This makes the code more readable and allows new reflection types to be added in a modular fashion. The second important function is that it allows variable storage for the zonal-data of different reflection types. Thus, adding a surface with a large directional table does not force the lambertian surfaces to use more memory.

If the zone is textured, its average reflection properties must be found. To avoid aliasing problems, the methods of Section 2.2 are used to point sample the zone to estimate the average properties.

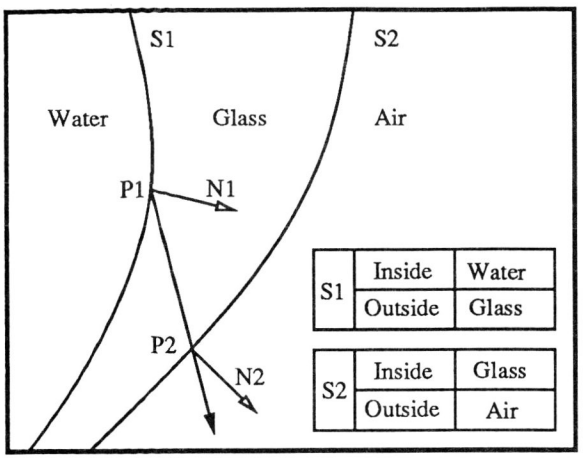

Figure 5: Elimination of ϵ-test.

Figure 6: A lamp with translucent lampshade.

3.1 Reducing Precision Problems

One advantage of ray tracing is that the programmer does not have to be concerned with maintaining the 'correct' outward facing surface normals, because we know whether a ray is 'inside' or 'outside' simply by counting surface crossings. For zonal methods, however, we need to emit power carrying rays toward the 'outside', so we need to maintain an outward facing normal. This is unfortunate, but it can be used to our advantage by eliminating the infamous ϵ test. A strategy similar to the one described in this section has been developed by Schlick [43].

When generating a parametrically defined ray $(\mathbf{o}+t\vec{\mathbf{v}})$ on a surface, as is done with reflected rays, power carrying rays, and shadow rays, we will always have an intersection at point $\mathbf{o}$ at approximately $t = 0$. To avoid complications that arise when imprecision causes the hit to be represented as a slightly positive number, we look for the first hit where $t > \epsilon$, where ϵ is a small positive number that bounds the possible roundoff error. Amanitides and Mitchell[3] showed that problems can arise when there are real surfaces closer than $t = \epsilon$ and provided strategies to use in these cases.

We can eliminate the ϵ test by using the surface normal information. Each surface can be viewed as an interface between two materials[5]. The first material can be viewed as the inside material (facing away from the normal), and the second as the outside material. As mentioned in Section 2.1, each ray stores the material it is currently in. This means a dot product of the ray direction with the normal of a surface will tell the ray which material is in front of the surface from the point of view of the ray. If this material is not the same as the material the ray is traveling through, then the ray cannot hit that surface. This idea is illustrated in Figure 5, where a ray originates at point P1 and propagates into glass (the outside material). When the ray is tested for intersection against S1, the dot product between the ray direction and N1 indicates that the ray is going inside to outside and is thus approaching from the water side. Since the air is in glass, we know there can be no intersection, so no ϵ test is needed. For S2, the inside material is glass, so the intersection point P2 is taken to be valid.

[5] Our translucent surface does not fit this definition, so some care must be taken when assuming such a model

4 Conclusion

While experimenting with different ways of combining ray tracing and zonal methods, it became clear to us that it is desirable to be able to assemble a global illumination renderer from some basic components. With such a system, at a global level, the graphics programmer will be able to experiment with different ways of assembling the renderer. while at a local level, different implementations of the same component can be exchanged invisibly. In this way, it would be possible to independently test different algorithms.

Figures 6 and 7 show two pictures generated using components in our framework. Figure 8 shows a picture with a zonal participating medium. The zones in the medium have volume angular distribution functions similar to the surface functions described in Section 3. The basic method follows [41], but uses progressive refinement ray tracing to implicitly calculate form factors.

Two programming strategies have been used to achieve our goal. The first was the creation of utility classes of points, vectors, colors, rays, orthonormal bases, noise generators, texture coordinates, and textures. These utility classes were used as primitive types, much like integers

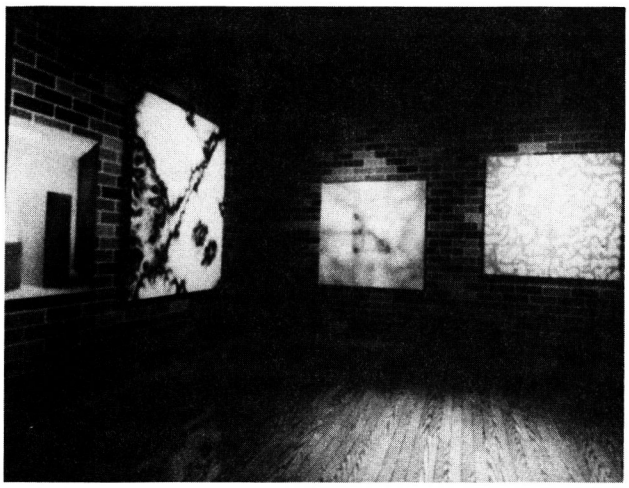

Figure 7: A gallery with several textures.

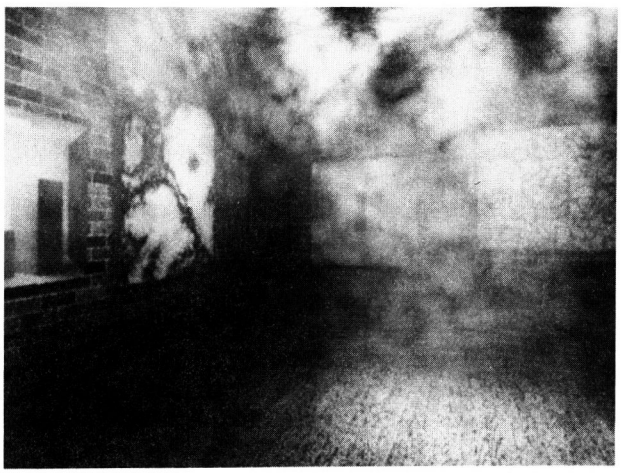

Figure 8: Gallery with participating medium.

and floats in numerical codes, in the creation of the image generation code. The second basic strategy was the identification of key components in a global illumination system and the creation of abstract classes to support these components.

We identified the key components of a stochastic ray tracer to be a sample point generator, a ray-object intersector, and a ray-material interactor. Corresponding abstract classes are defined to support these components: sample-generator allows different sampling strategies and distributions to be plugged into the basic ray tracing module, geom-object allows the addition of new object types and collection structures, and ray-material interaction class allows the definition of new materials with distinct light interaction characteristics.

When identifying the key components for supporting zonal calculations, we realize that most of the ray tracing components are reusable. For example, the reflected energy ray distribution uses the sample point generator, and the surface energy reflection behavior is implemented using the ray-material interaction class. The zonal-data class hides the mechanics of power accumulation and redistribution, and thus allows efficient storage systems to be implemented for each surface type.

One issue that remains unresolved is how to handle surfaces with complex material properties. For example, we could define a floor surface with alternating tiles made of marble and steel. The steel would respond to light as a metal, and the marble as a polished surface. It would also be desirable to be able to add a layer of dust (probably using a procedural texture), that would cover the marble and steel in a nonuniform manner, reducing the specularity of both. If the 'material' were to handle all shading in this situation, it would need access to steel, marble, and dust reflectance behavior, as well as the procedural texture describing the dust. This could be accomplished in a manner similar to a Renderman shader [51, 23], where the shading routine has access to the internals of reflection models and textures. Unfortunately, such a shader does not hide much information, and can become quite unwieldy. It would be very desirable to put the capabilities of a general shader in a class structure that preserves modularity and data hiding, but exactly how to create such a class structure is still a research topic.

Our discussion is summarized by the description of a global illumination system assembled from the defined components. Without altering other components in the system, we have implemented different octree traversal algorithms and storage strategies. The different implementations of the component were plugged in and tested in a way that is invisible to the rest of the system. In this way, some observations were made that contradict common beliefs. For example, the tree walking algorithm is not always the best octree traversal choice, jittering may not be the best simple sampling strategy, and mail box may not always be a worthwhile effort. During the course of the system development, we changed our color models and sampling strategies several times. This was accomplished with only local code alterations.

We believe we have identified a useful set components for the construction of global illumination system. Note that there is no 'correct' set of components for any software system. As new algorithms are invented, the component design should be refined to support the implementation of the new algorithms.

5 Acknowledgments

Thanks to Greg Rogers for general help with object-oriented design, Claude Puech for discussion about the ray-object intersection, John Airey, Holly Rushmeier,

John Wallace, and Greg Ward for numerous radiosity discussions, Jim Arvo for providing insight and information about his ray tracing work, Andrew Woo, for help with traversal and sampling issues, Don Mitchell for some expert guidance in sampling theory, Ralph Johnson for motivating a design based on something besides pure hacking, Eric Ost and Jean Buckley for improvements on the drafts of this paper, and special thanks to William Kubitz, the advisor for this project.

References

[1] John M. Airey and Ming Ouh-young. Two adaptive techniques let progressive radiosity outperform the traditional radiosity algorithm. Technical Report TR89-20, University of North Carolina at Chapel Hill, August 1989.

[2] John M. Airey, John H. Rohlf, and Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics*, 24(1):41–50, 1990. ACM Workshop on Interactive Graphics Proceedings.

[3] John Amanatides and Don P. Mitchell. Antialiasing of interlaced video animation. *Computer Graphics*, 24(3):77–86, August 1990. ACM Siggraph '90 Conference Proceedings.

[4] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, 1987.

[5] Bruno Arnaldi, Thierry Priol, and Kadi Bouatouch. A new space subdivision method for ray tracing csg modelled scenes. *Visual Computer*, 3:98–107, 1987.

[6] James Arvo. Backward ray tracing. *Developments in Ray Tracing*, pages 259–263, 1985. ACM Siggraph '85 Course Notes.

[7] James Arvo. Linear-time voxel walking for octrees. *Ray Tracing News*, 1(2), February 1988. e-mail Edition, available under anonymous ftp from weedeater.math.yale.edu.

[8] James Arvo and David Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

[9] Norman F. Barnes. Color characteristics of artists' pigments. *Journal of the Optical Society of America*, May 1939.

[10] Brian Cabral, Nelson Max, and Rebecca Springmeyer. Bidirectional reflectance functions from surface bump maps. *Computer Graphics*, 21(4):273–282, July 1987. ACM Siggraph '87 Conference Proceedings.

[11] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A progressive refinement approach to fast radiosity image generation. *Computer Graphics*, 22(4):75–84, August 1988. ACM Siggraph '88 Conference Proceedings.

[12] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[13] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics*, 18(4):165–174, July 1984. ACM Siggraph '84 Conference Proceedings.

[14] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, second edition, 1990.

[15] Akira Fujimoto, Takayu Tanaka, and Kansei Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, pages 16–26, April 1986.

[16] Andrew Glassner. Implementation notes for ray tracers. *Advanced Topics in Ray Tracing*, 1990. ACM Siggraph '90 Course 24 Notes.

[17] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[18] Andrew S. Glassner. How to derive a spectrum from an rgb triplet. *IEEE Computer Graphics and Applications*, 9(7):95–99, 1989.

[19] Ronald N. Goldman. Illicit expressions in vector algebra. *ACM Transactions on Graphics*, 4(3):223–243, July 1985.

[20] David Edward Hall. An analysis and modification of shao's radiosity method for computer graphics image synthesis. Master's thesis, Department of Mechanical Engineering, Georgia Institute of Technology, March 1990.

[21] Roy Hall. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, N.Y., 1988.

[22] Roy Hall and Donald P. Greenberg. A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications*, 3(8):10–20, 1983.

[23] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. *Computer Graphics*, 24(3):289–298, August 1990. ACM Siggraph '90 Conference Proceedings.

[24] Paul S. Heckbert. Writing a ray tracer. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

[25] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics*, 24(3):145–154, August 1990. ACM Siggraph '90 Conference Proceedings.

[26] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. *Computer Graphics*, 20(4):133–142, August 1986. ACM Siggraph '86 Conference Proceedings.

[27] Frederik W. Jansen. Data structures for ray tracing. In L. R. A. Kessener, F. J. Peters, and M. L. P. van Lierop, editors, *Data Sturctures for Raster Graphics*, pages 57–373. Springer-Verlag, Netherlands, 1986.

[28] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object Oriented Programming*, pages 22–35, June 1988.

[29] James T. Kajiya. The rendering equation. *Computer Graphics*, 20(4):143–150, August 1986. ACM Siggraph '86 Conference Proceedings.

[30] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo Methods*. John Wiley and Sons, New York, N.Y., 1986.

[31] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, August 1986. ACM Siggraph '86 Conference Proceedings.

[32] David Kirk and James Arvo. The ray tracing kernel. In *Proceedings of Ausgraph*, pages 75–82, July 1988.

[33] Donald Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley, New York, N.Y., 1981.

[34] Gary W. Meyer, Holly E. Rushmeyer, Michael F. Cohen, Donald P. Greenberg, and Kenneth E. Torrance. An experimental evaluation of computer graphics imagery. *ACM Transactions on Graphics*, 5(1):30–50, January 1986.

[35] Don P. Mitchell. Generating antialiased images at low sampling densities. *Computer Graphics*, 21(4):65–72, July 1987. ACM Siggraph '87 Conference Proceedings.

[36] Masataka Ohta and Mamoru Maekawa. Ray coherence theorem and constant time ray tracing algorithm. In Toshiyasu Kunii, editor, *Computer Graphics 1987*, pages 303–314. Springer-Verlag, Tokyo, Japan, 1987.

[37] Edward D. Palik. *Handbook of Optical Constants of Solids*. Academic Press, New York, N.Y., 1985.

[38] Ken Perlin. An image synthesizer. *Computer Graphics*, 19(3):287–296, July 1985. ACM Siggraph '85 Conference Proceedings.

[39] Ken Perlin and Eric M. Hoffert. Hypertexture. *Computer Graphics*, 23(3):253–262, July 1989. ACM Siggraph '89 Conference Proceedings.

[40] S. D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144, February 1982.

[41] Holly E. Rushmeier. *Realistic Image Synthesis for Scenes with Radiatively Participating Media*. PhD thesis, Cornell University, May 1988.

[42] Bertrand Le Saec and Christophe Schlick. A progressive ray-tracing-based radiosity with general reflectance functions. In *Proceedings of the Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 103–116, June 1990.

[43] Christophe Schlick. The acne problem. *Ray Tracing News*, 4(1), March 1991. e-mail Edition, available under anonymous ftp from weedeater.math.yale.edu.

[44] Y. A. Screider. *The Monte Carlo Method*. Pergamon Press, New York, N.Y., 1966.

[45] Peter Shirley. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois at Urbana-Champaign, November 1990.

[46] Peter Shirley. Physically based lighting calculations for computer graphics: A modern perspective. In *Proceedings of the Eurographics Workshop on Photosimulation, Realism and Physics in Computer Graphics*, pages 67–81, June 1990.

[47] Peter Shirley. A ray tracing algorithm for global illumination. *Graphics Interface '90*, May 1990.

[48] Peter Shirley. Discrepancy as a quality measure for sampling distributions. In *Eurographics '91*, September 1991.

[49] Robert Siegel and John R. Howell. *Thermal Radiation Heat Transfer*. McGraw-Hill, New York, N.Y., 1981.

[50] Kelvin Sung. A dda octree traversal algorithm for ray tracing. In *Eurographics '91*, September 1991.

[51] Steve Upstill. *The Renderman Companion*. Addison-Wesley, Reading, MA, 1990.

[52] John R. Wallace, Kells A. Elmquist, and Eric A. Haines. A ray tracing algorithm for progressive radiosity. *Computer Graphics*, 23(3):335–344, July 1989. ACM Siggraph '89 Conference Proceedings.

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

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

[55] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, September 1990.

[56] Andrew Woo. The world of optik. Technical report, Department of Computer Science,University of Toronto, February 1989.