CHAPTER 7

EXTENSIONS TO BASIC SOLUTIONS

The previous chapters have assumed that the scene has vacuum between the surfaces, that the solution is needed at only one wavelength, and that the scene is static. This chapter addresses what to do when these assumptions do not hold.

Section 7.1 discusses zonal techniques for participating media, with emphasis on extending the Monte Carlo techniques of the last chapter. In Section 7.2, solutions for multiple wavelengths are discussed. Section 7.3 examines solutions for dynamic environments. Finally, the contents of the chapter are summarized in Section 7.4.

7.1 Including Participating Media

Solutions with primary scattering from participating media have been discussed by Blinn[11] and by Kajiya and Von Herzon[62]. There have also been many methods that use primary and secondary scattering in the Heat Transfer literature, but most of these make many simplifying assumptions about either geometry or scattering function[54]. A zonal method has been applied to the case of an isotropically scattering medium by Rushmeier[93]. Her approach was not based on the energy shooting strategy, so arbitrary scattering functions were not allowed. Rushmeier

also developed Monte Carlo methods that fully account for arbitrary scattering functions, but these methods are not as efficient as the zonal method she used for the isotropic case[92].

The power shooting techniques of the last chapter can be extended to media with general scattering functions. As with surfaces, the media can be divided into zones, where each zone located at x_i is considered to contain small particles with number density N (number of particles per unit volume), scattering cross sectional area A, and a *SPF* $s(x_i, \psi_{in}, \psi_{out}, \lambda)$. Values of A and s for many real world gasses as well as an excellent treatment of scattering in general can be found in the paper by Klassen[67].

When a ray carrying power Φ passes a distance t through a zone, it will have some amount of power Φ_s scattered by the zone:

$$\Phi_s = \Phi(1 - e^{-NAt})$$

This energy can be entered into the $I(x_i, \psi, \lambda)$ table for the zone. During the shooting phase of the algorithm, volume zones can fire their accumulated power according to $I(x_i, \psi, \lambda)$.

After the zonal phase is finished, the radiant intensity functions can be interpolated to zone vertices. When a viewing ray passes through the media, it will pick up some of the radiance of the zones, and some of the background radiance will be attenuated.

If the radiant intensity functions $I(x_i, \psi, \lambda)$ of the zones are converted to $L(x_i, \psi, \lambda)$ radiance functions for the particles, then calculating the color seen along a viewing ray is determined by:

$$L(x, \psi, \lambda) = L_b(\psi, \lambda)e^{-NAt} + (1 - e^{-NAt})L(x_i, \psi, \lambda)$$

where the ray passes through a distance t of the zone, and L_b is the radiance seen through the zone. If N, A, or $L(x_i, \psi, \lambda)$ vary along the ray (which will happen if we interpolate between zone vertices), then ray integration can be done in small steps as done by Kajiya and Von

Herzen[62]. This machinery is also used for scalar volume visualization, though the lighting models used are not necessarily physically motivated[68, 114].

Because the volume zones are treated similarly to surface zones, it is straightforward to allow volume zones to be light sources such as fire.

One potential drawback that arises when volumes are used is that a volume and surface object may overlap. This means care must be taken when emitting energy from a volume element to avoid sending energy from the inside of an opaque object. A possible solution is to allow volume geometries that are not rectilinear, so the volumes can be wrapped around a surface. This unfortunately makes ray tracking more difficult[101]. Rushmeier has suggested that any power carrying rays be dissallowed if their origins are inside an object[92]. Her approach would at least prevent physically impossible rays.

As can be done with surfaces, we can calculate the indirect lighting of volumes with a low resolution zonal phase, and calculate the detailed direct lighting of the medium in the viewing phase. This is similar to calculating lighting on textured surface zones where average reflectance is used in the zonal calculation. With volumes, average number density and cross sectional area can be used.

7.2 Wavelength Dependencies

Because of the many perceptual issues involved, wavelength sampling and conversion to the proper format for display are perhaps the least well-understood part of the image generation process. Color in most graphics applications simply means three separate solutions for the red, green, and blue channels. What these channels really mean, and how they relate to spectral sampling is subtle[36]. Many authors have asserted that some form of spectral sampling, with

later conversion to RGB is preferable[44, 71], especially for certain pathological cases. Just what degree of sampling is needed is unclear, but most practitioners use a small number of samples (4-20), and this seems to be adequate for many applications. Meyer has suggested a group of four wavelengths chosen with perceptual motivation[71], and these wavelengths have been used for the figures in this work.

In some situations, it is not possible to generalize to multiple wavelengths simply by maintaining an array of radiances for a selection of wavelengths. One obvious example is when refraction causes dispersion, so that the different wavelengths require different rays to represent their paths. However, *usually* we will want to treat the different wavelengths as a group, so that the number of rays traced is minimized. Thomas has simulated refraction and dispersion effects in a way that maintains coherence of the wavelengths until dispersion occurs, at which point an adaptive splitting occurs[112]. Thomas' approach is complicated to implement, but there is probably no easy method to capture dispersion effects while maintaining coherence.

Another situation where spectral coherence problems arise is when the general energy transport methods of Section 6.3 are used. If a zonal surface is glossy, the light leaving it may have very different directional distributions for different wavelengths. Because of this difference, there is no way to send out a bundle of rays while having the rays carry constant spectral quantities. There are two solutions to this problem. The naive, and inefficient, approach would be to send separate rays for each wavelength. A better approach would be to send rays based on the total power integrated over all wavelengths (or perhaps luminance), and to weight each ray according to the spectral distribution for its direction.

Some of the color effects caused by physical optics have been treated as special cases. Opalescence has been simulated using a simplified model of scattering[125]. Interference effects

(e.g. thin-film effects) have been simulated using analytical models on surfaces where a priori knowledge that interference will occur exists[104]. Because interference produces almost pure spectral colors, care must be taken when displaying images containing interference effects, because they may be outside the gamut of the display device. Smits and Meyer have observed that the quality of the conversion depends on in what color space the extrapolation to a displayable color takes place[104]. Diffraction effects, such as the colored bands around headlights, have been modeled by Nakamae et al.[77].

The issue of wavelength sampling will only increase in importance. Currently, most input data to graphics programs is non-physical data that is often reasonably smooth. Once more varying spectral data, such as fluorescent light spectra, are used, the relation of spectral sample point location to the peaks and valleys in the spectral function will become more important. It is unclear what the resolution to this issue will be, but it is possible that for many applications the efficient solution of smoothing the input spectral data to minimize the number of spectral samples will be acceptable.

7.3 Time Dependencies

When the image is formed over a time interval, and objects are moving in that interval, blurring and dynamic shading can occur. This can be handled by image based methods in a fairly straightforward (but not that easy to implement) manner. This is done by integrating over time in addition to pixel area, lens area, reflection direction, and light source areas. Distributed ray tracing accomplishes this by associating a time with each ray, thus sampling time as another dimension of the integrand[26, 25].

Zonal solution methods do not adapt so easily to time dependent solutions. One of their fundamental assumptions is that the system is either solved for in a very small time period, or is in steady state. Several researchers have looked into solving for a given configuration and then modifying this solution after changes in the geometry of emitted energy distributions take place[9, 88, 19]

An approximate solution method for dynamic environments would be to solve for set time intervals and linearly interpolate for times between these intervals. A viewing ray at a certain time could approximate a zonal value for a given time. This method would be more robust if the direct light calculations were image based.

7.4 Summary

The ray tracing based energy transport techniques of Chapter 6 can be generalized to include volumes of media.

When wavelength dependencies are added into image based methods, the wavelength information can be carried along with each ray. This idea breaks down when dispersion of different wavelengths occurs at a surface. In this case the ray can be split into rays representing different spectral bands. Exactly how one should handle spectral sampling is an open issue, but some small number (4-20) of samples usually is sufficient.

When wavelength dependencies are added to zonal methods, the spectral information can be stored as a group for each patch. A complication arises in the case where the **BRDF** of a zone depends on incident directions. In this case, the luminance of the radiant intensity distribution can be used as a distribution for energy carrying rays sent by the zone.

When the geometry or lighting in a scene are time dependent, stochastic image based methods can treat time as just another dimension to be integrated over. Time dependency in zonal methods is much more of a problem. If the time dependency can be localized, then the partial zoning method of Chapter 6 could be used to find an approximate solution. Another possibility is that separate zonal solutions can be made at specified sample points, and approximate solutions could be found using interpolation.

CHAPTER 8

IMPLEMENTATION

All of the raster pictures in the previous chapters were produced using a moderate size C++ program (approximately 9000 lines). The program follows three basic phases: the input of the geometric and viewing parameters, the zonal transport of power, and the formation of an image for a specified set of viewing parameters.

Two basic techniques associated with object oriented programming were used in designing the code. The first was the implementation of a set of 'utility' classes, such as points and vectors. The second was the use of subclassing (inheritance) to generate cleaner, more extensible code than usually results from conventional programming practice.

Section 8.1 outlines the utility classes implemented, and how these classes were implemented. The ray tracing primitives are described in Section 8.2. The material property classes are discussed in Section 8.3. The Zonal calculation phase of the code is described in Section 8.4. The viewing phase is then covered in Section 8.5. Finally, this chapter is summarized in Section 8.6.

8.1 Toolbox Classes

Some utility classes were 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 Ronald Goldman. The allowed expressions are:

1. vector + vector = vector
2. vector - vector = vector
3. point + vector = point
4. vector + point = point
5. scaler * vector = vector
6. vector * scaler = vector
7. point - vector = point
8. rotation-matrix * rotation-matrix = rotation-matrix
9. point * rotation-matrix = point
10. vector * rotation-matrix = 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. In our experience, the benefits of these operational restrictions have greatly outweighed the deficits.

It is often useful to restrict a vector to be of unit length, such as when dealing with direction vectors in a ray. The unit-vector class has been defined as a subclass of vector. The unit-vector

class can be confusing because some operations such as a matrix multiplication can change its length. A partial resolution to the problems of rigor versus efficiency is to let the unit-vector inherit all of the operations of the vector type. For example, we can have 'vector = vector + unit-vector'. A unit vector can be initialized by an assignment operator 'unit-vector = vector', where the assignment occurs after automatic normalization of the vector value. Unit vectors themselves have a subclass, surface-normal-vector, which responds to transformations differently than unit-vectors.

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 for a 'typical' image). Next, we switched to twenty evenly spaced nodes were used 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 [72]. 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 only using a few sample locations.

Another basic utility class is the ray class, represented by a point of origin, and a unit-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 [44] 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. The basis is defined by three orthonormal vectors, $\vec{\mathbf{u}}$, $\vec{\mathbf{v}}$, and $\vec{\mathbf{w}}$.

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 [84] 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 [85], and found it to be quite mechanical to implement. While it

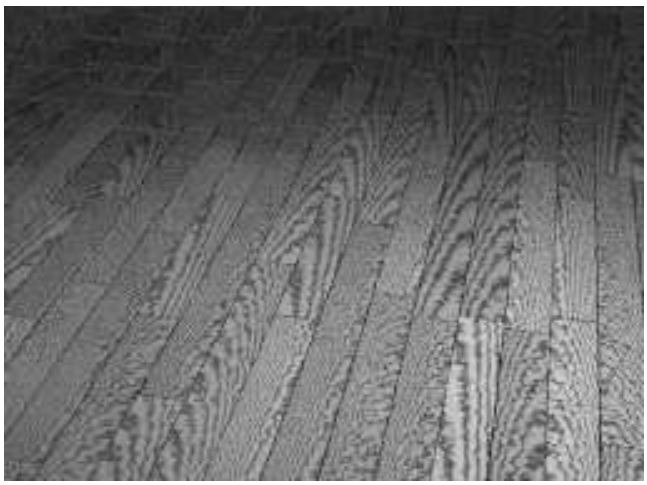

Figure 8.1: A board floor generated from a procedural solid texture.

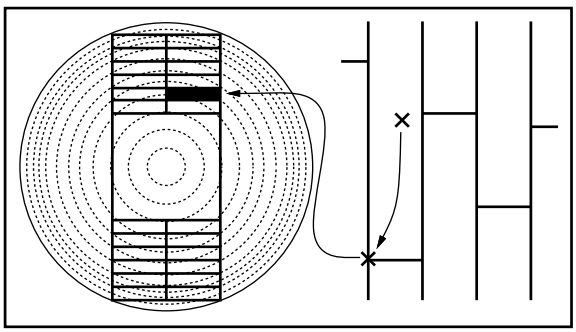

Figure 8.2: Use one corner of o board to find a solid noise seed for an index into the tree.

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. a specific brick) can be modeled using various transformations on solid noise. This needs to be added 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 planer surface 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). The 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 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 8.2). 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. An example floor generated by this method is shown in Figure 8.1.

This same basic idea 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. To generate the basic 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 [7]. When we have existing rgb data, we use Glassner's conversion method [39]. It has been our experience that using smooth curves for this conversion is highly preferable to using impulses.

8.2 Ray Tracing Primitives

As discussed in Chapters 5 and 6, finding the object first seen along a ray is a fundamental operation. Paul Heckbert has noted that this operation should be associated with an abstract 'geometrical object' class[51]. Specific objects, such as spheres and polygons, should be subclasses of the abstract class. This allows the addition of new geometrical object types without muddying existing code that uses ray tracing. Arvo and Kirk have generalized this idea by making the spatial ray intersection structures, such as Glassner's octree[38], themselves subclasses of geometrical object[65]. This makes sense because the program modules that use the ray tracing primitives do not need to know about underlying representations.

One useful thing that Arvo and Kirk have emphasized is that having the geometrical primitives (spheres, polygons, etc.) and the collection structures (octrees, lists, etc) be subclasses of the same abstract class, is that storage structures can be nested. This allows a particular storage class (a bounding volume hierarchy, for example), to contain, as members, other storage classes (octrees, regular grids, simple lists). Intelligent use of this generality can reduce running time. Unfortunately, just how to attain 'intelligent use' is not obvious.

Another useful characteristic of this class structure is that data hiding is quite natural. One of the basic primitives in the program is the triangle. A connected set of triangles forms a new 'mesh' primitive. If an octree data structure were used to store the primitives at a global level, it would be natural to want to add a mesh as a member of the octree. At appropriate times, the octree can query the mesh on whether a ray intersects the mesh. The mesh can then call on its list of triangles to decide whether the ray hits it. If there were many triangles, it would be essential that the mesh maintain its own internal efficiency structure (perhaps an octree). The general class structure allows such nesting, and is more modularized than it would be if the global octree were allowed access to the mesh's list of triangles.

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 very different, should not be grouped together, as seen with points and vectors in the last section.

8.3 Ray-Material Interaction

Several researchers have noted that reflection behavior should be encapsulated as one unit of a rendering system [66, 115, 47]. 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 one shading model with parameters including ks, kt, and kd [43]. 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, 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 effect 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 by asking, 'given an incoming ray r, a point p on the surface, and a surface normal $\vec{\mathbf{n}}$, 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 query a material, 'what is your radiance, $L(\vec{\mathbf{v}}_{out})$, that comes from a source of radiance $L(\vec{\mathbf{v}}_{in})$ 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 data for conductors in [82]. The behavior of both conductors and dielectrics is determined using the Fresnel Equations, the full form of which can be found in [102, 98]. 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 stochasticly if $e \neq \infty$.

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

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

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

translucent: Generate two rays stochastically, one reflected, one transmitted, each with a stochastic 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 lambertain, translucent, and luminaire surfaces, and would turn off reflections off 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 , to be used as a basis for any probabilistic reflection that might occur.

8.4 Zonal Calculations

Several recent zonal (radiosity: global lighting information is stored at a finite set of 'zones') systems are based on progressive refinement techniques [22, 2]. The theoretical basis for such systems is straightforward to extract. If the progressive refinement is viewed as power transport simulation, which implies fairly direct non-diffuse zonal solutions [5, 100, 42, 99, 95]. 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.

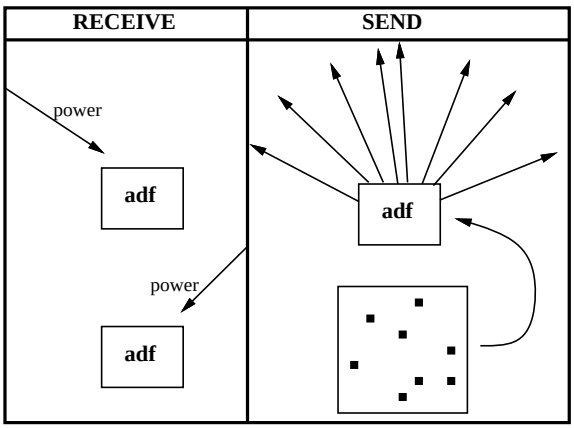

Figure 8.3: The two crucial methods of and adf (angular distribution function): receive power, and send power according to some set of sample points.

For a lambertian zone, the zonal-data module is easy to implement because there is no dependancy 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 done with pixel sampling, 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 [42, 99, 95]. A simple way to do this is a simple spherical coordinate array of bins, with the total power going through each bin. The unsent and total power of the diffuse case must be generaized 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 8.3).

The receiving method of a directionally dependent adf can be implemented by using the ray-material interaction module of Section 8.3. 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 sampling transformation methods, or by independantly 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 it would still 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 extendable 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 a adf module that internally only stores 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 in invisible. This allows the programmer to detatch the local lighting models from the global light transport.

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, stochastic techniques are used to point sample the zone to estimate the average properties.

8.5 Viewing Calculations

There are two basic abstractions in the viewing calculations. These abstractions can apply to any ray tracing program, even if no zonal calculations are used. 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. This flexibility helped in the comparison tests of Sections 5.1 and 5.5. The second basic abstraction is to have the material (e.g. steel, glass, etc.) control the shading of a surface. Shading includes both responding to light (e.g. phong shading), and generating reflected or transmitted light. A similar abstraction was used by Arvo and Kirk[65]. This simplifies the shading calculations, and allows new material types and shading models to be added in a flexible manner.

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 shade[115, 47], 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.

8.6 Summary

Two basic programming strategies have been used to implement the code used for this work. The first was the creation of toolbox classes of points, vectors, colors, orthonormal bases, texture coordinates, and textures. These toolbox classes were used as primitive types, much like integers and floats in numerical codes, in the creation of the image generation code.

The second basic strategy was the creation of abstract classes of geometrical-object, sample-generator, zonal-data, and material-shader. These objects, practically speaking, replace explicit case statements with calls to virtual functions of the abstract classes. In the case of geometrical-objects, this allows the addition of new object types and search hierarchies. For sample-generators, this allows different sampling strategies and distributions to be plugged into the basic ray tracing module. The zonal-data class accomplishes data hiding on the accumulated power distributions, so efficient storage systems can be implemented for each surface type.

The least satisfying part of the implementation is the material-shader abstract class, which seems to need to much global information as currently implemented. It remains to be seen

whether it is possible to have a more modular shading class structure while maintaining the power that can come with the global information.

CHAPTER 9

CONCLUSION

In this work I have attempted to put realistic image generation into a usable theoretical formulation, and provide an overview of some previous and new solution techniques for global illumination.

The basic physics of reflection and light transport were used to derive the rendering equation. The problem of generating an image was phrased in terms of evaluating the Global Radiance Function. The standard reflection models of computer graphics were reviewed, and modifications were made to the combined specular and diffuse reflection model to allow for Fresnel effects. This formulation of the rendering equation differs from previous formulations by explicitly accounting for transparent surfaces.

The physical rules governing reflection were also used to make improvements in reflection models. In diffuse transmission it was shown that light is filtered to the same extent regardless of which side of the surface the light comes from. This eliminates one of the parameters from previous diffuse transmission models. The microscopic structure of polished surfaces was used to justify coupling the diffuse and specular coefficients according to the Fresnel Equations. The Fresnel Equations are commonly used to vary the reflectivity of metal and transparent

dielectrics, but have not been used before to vary the reflectivity of the polish and underlying diffuse substrate.

Image-based solution methods were phrased as a lazy evaluation of the Global Radiance Function; evaluation only took place for visible points. Several constraints were outlined for what part of the image function should contribute to each pixel, and a new separable, symmetric filter was developed that satisfies these constraints.

A stochastic shadow ray generation method was introduced that greatly reduces the number of shadow rays needed for scenes with multiple light sources. The sampling distributions used for shadow rays and other dimensions of the integral were evaluated by introducing to computer graphics the notion of discrepancy from numerical integration theory. The use of discrepancy provided some insight not given by the signal processing theory traditionally used in computer graphics. As part of this discussion a new sampling scheme, separate sampling, was introduced. Separate sampling was shown to be as efficient to generate as jittered sampling, while often outperforming Poisson disk sampling. It also can generate distributions for any positive integer number of samples, including primes.

The peculiarities of the sampling spaces used in distributed ray tracing were shown to preclude naive hierarchical sampling. It was demonstrated that hierarchical sampling can greatly reduce noise, however, if we have sufficient knowledge of the sampling space.

Zonal methods represent the opposite extreme of image methods, where all function values are computed and stored, and each evaluation is a table lookup. The zonal method was phrased as a transport simulation, similar to progressive refinement radiosity methods. Using this direct simulation model, it is straightforward to generate zonal methods for anisotropic reflection. This requires storing accumulated power in a directional table for each zone.

It was also shown that simulation allows for surfaces which are not zoned to interact with those that are. This is a generalization of the diffuse and specular ray tracing transport work of Malley. This technique can be useful for highly complex or difficult to zone surfaces such as a human face. For ray tracing methods, $O(N)$ rays are required for scenes with N zones that have a bounded area ratio.

The zonal solution methods can be applied to participating media in a fairly natural manner. This also applies to media with anisotropic scattering characteristics, but such a solution requires a large amount of storage.

Wavelength dependent solutions introduce some complications, but can be handled by traditional point sampling techniques. Time dependent solutions are easily handled by image-based solution methods, but are very difficult to apply using zonal methods.

In implementing a global renderer, it is possible encapsulate the local illumination code so that it is independent of the global illumination code. The standard techniques of object oriented programming are useful in accomplishing this.

In closing, there are many open problems that are worth looking into, of which a partial list follows:

- Is a sophisticated model of glossy reflection needed for Computer Graphics applications, or will simple empirical models suffice?
- Are anisotropically scattering media needed for Computer Graphics?
- How should indirect lighting in outdoor scenes be handled?
- How should interaction between indoor and outdoor scenes (at windows) be handled?
- How realistically must we model the characteristics of light sources?
- How should we restrict the allowed permutations of separately generated samples? Can discrepancy analysis be of use?
- What automatic gridding techniques could be used for zonal methods?
- How can discrepancy be generalized to non-rectangular sampling spaces?

- How should modeling interact with reflection models? E.g. should wet wood become darker automatically?
- Should discretization of a complex zonal environment be view dependent?
- What is a valid error metric for a computed image?
- For the purposes of complexity analysis, what is an 'average' scene?
- How do we determine the error of an approximate image?
- How can perceptual science guide us in displaying an image, so that it evokes the same response as a real scene, rather than that of of a photograph?
- Is there an algorithm that can efficiently calculate solutions for indoor, outdoor, and combined scenes?