THREE-DIMENSIONAL TEXTURING USING LATTICES

Robert R. Lewis
B.S., Harvey Mudd College, 1974
M.A., University of California, 1979

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

October, 1988

The dissertation "Three-Dimensional Texturing Using Lattices" by Robert R. Lewis has been examined and approved by the following Examination Committee:

Bart Butell, Thesis Advisor
Adjunct Professor

[illegible]

Jack Gjovaag
Principal Scientist (Tektronix)

Richard Hamlet
Professor (Portland State University)

David Maier
Associate Professor

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to the many people who helped during the production of this thesis.

The thesis committee provided much-needed direction and offered some helpful suggestions.

My managers at TSSI: Tom Hamilton, Mark Faust, Ron Lunde, and Jeff Hahs were all generous in providing computer resources. System administrator and programmer Joe Pruett provided useful advice and cooperation. (Okay, guys, you can trash that DN660 now!) Many people at TSSI provided encouragement ("Hey, Bob! That looks cool!")

Chris Rose of the Beaverton sales office of Apollo Computer graciously provided time on a DN580 to produce some of the color plates shown here. Mark Faust provided the photographic equipment.

Special thanks go to Jules Bloomenthal of Xerox PARC, who pointed out the need for a change of direction much later than I should have seen it myself.

TABLE OF CONTENTS

Introduction 1
  Ray Tracing 1
  Texture Mapping 5
  Particle Systems 8
Lattices 12
  Rasterization 13
  Grid Spacing Considerations 15
  The Lattice Hash Table 18
  Lighting Model 20
  Example 23
Lattices in a Conventional Ray-Tracing System 26
  Constructive Solid Geometry 26
  Primitives 29
    Quartics 29
    Lattices 30
  Implementation Details 30
  Example: A Fuzzy Sphere 31
Conclusions 36
Future Directions 40
  Time Efficiency 40
  Memory Usage 40
    Compress the LHT 41
    Compute the LHT on Demand 41
    Construct a Stochastic Lattice Hash Function 44
References 45

ABSTRACT

The thesis investigates a way to perform realistic three-dimensional texturing of ray-traced objects that is especially useful for objects with irregular surfaces. Two similar existing methods are texture mapping and particle systems. The thesis describes these methods and points out their strengths and weaknesses. It then proposes an alternative to these methods that uses a construct called a *lattice*. Lattices make as fast ray tracers, but they are inexact. As long as lattices are used for small objects, though, their inexactness doesn't show on the scale of the display, and the result is acceptable.

The thesis also shows how lattices fit with the more traditional large-scale ray tracer, combining lattices with a Constructive Solid Geometry (CSG) object model to produce several examples.

Time and memory space considerations are major constraints on lattices. The thesis discusses these limitations and how they can be reduced.

CHAPTER 1

INTRODUCTION

This thesis discusses a new technique for adding surface detail to objects represented in a three-dimensional rendering system.

1.1. Ray Tracing

Ray tracing is the most widely used image generation technique in realistic image synthesis. Although the fundamental ideas ("ray casting") behind go back as far as Goldstein and Nagel ([Gold71]), widespread use of ray tracing dates from the seminal paper [Whit80]. Since that time, investigators have done a considerable amount of work extending the basic notions of ray tracing to cover a wide range of applications.

It is impossible to cover all of them here. There are good overviews of ray tracing in several popular computer graphics texts (e.g., [Roge85] and [Fole82]). Every SIGGRAPH conference since 1980 has had numerous ray-tracing contributions. The discussion here will confine itself to the necessary mathematical notation.

The geometry relating what an observer in 3-space is looking at is as follows. The observer at a point $\mathbf{o}$ is looking at a display a distance D away. This display is made up of $N_x \times N_y$ pixels. A ray $\mathbf{r}$ begins at $\mathbf{o}$ and passes through the center of one of the pixels. If d is a vector from o to that pixel, the mathematical ("parametric") description of the ray is:

$$\mathbf{r} = \mathbf{o} + \mathbf{d}t \tag{1.1}$$

where $t \geq 0$ is the "parameter". $T = 0$ at the origin **o** (obviously) and $t = 1$ at the location of the pixel on the display. Let W be the width of the display and let H be its height. Figure 1 shows this geometry. Given D, W, H, N_x, and N_y, one can construct a ray through any point

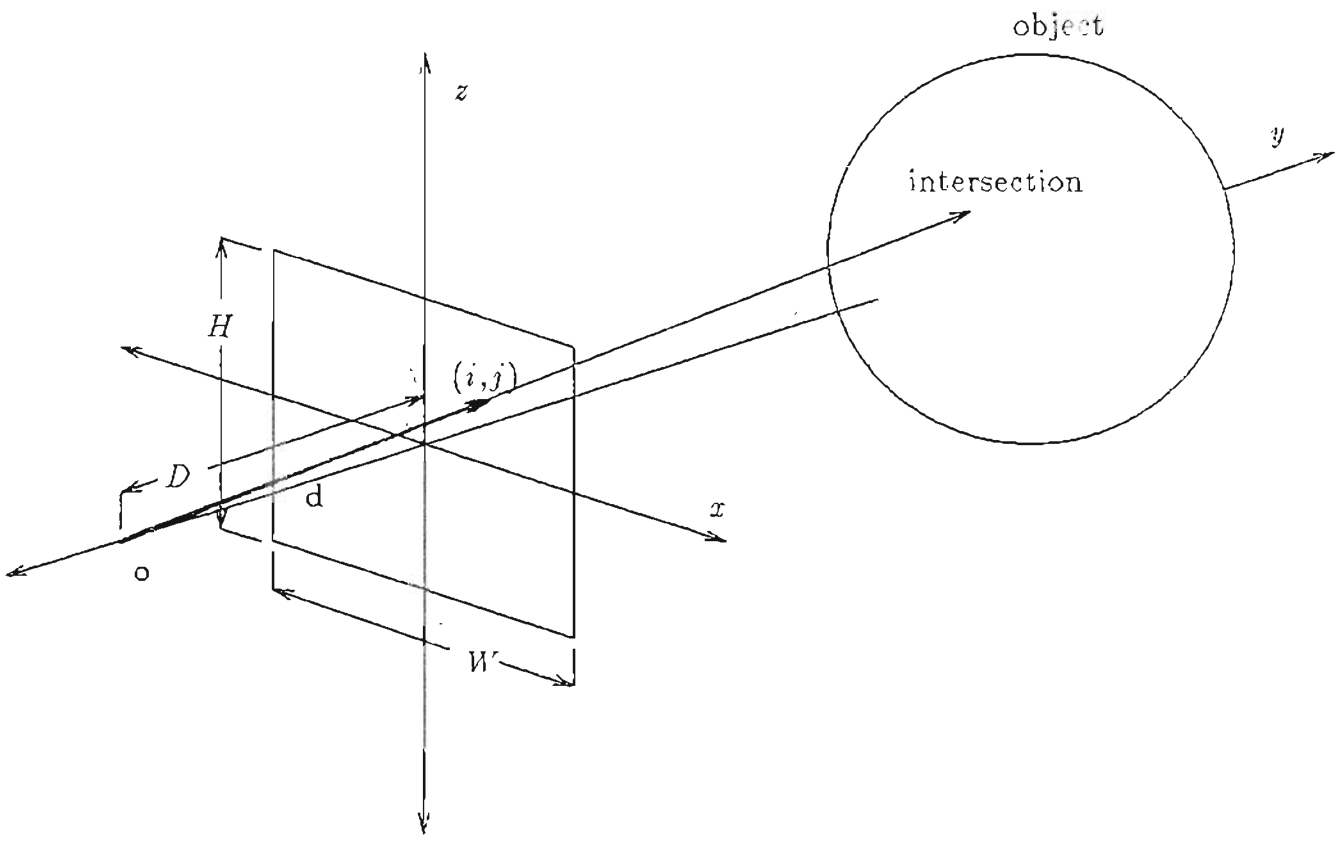

Figure 1 Basic geometry of ray tracing. Quantities are defined in the text.

(i, j), $0 \leq i < N_x$, $0 \leq j < N_y$ on the display using

$$\mathrm{o} = \left\{0, 0, -D\right\} \tag{1.2a}$$

$$\mathrm{d} = \left\{W\left[\frac{i}{N_x - 1} - \frac{1}{2}\right], H\left[\frac{j}{N_y - 1} - \frac{1}{2}\right], D\right\} \tag{1.2b}$$

The essence of ray tracing is the ability to determine the intersections of rays with a mathematical model of whatever it is that is being rendered and then calculate the appearance of the model at that intersection. For example, the model may be a simple opaque sphere $x^2 + y^2 + z^2 = 1$. For a given ray, x, y, and z are all constrained to lie along it, so it is easy to plug the values of $r_x \rightarrow x$, $r_y \rightarrow y$, and $r_z \rightarrow z$ from (1.1) into the equation of the sphere. The result is a quadratic in t with everything else known. Valid rays must have positive, real t values, so this will lead to 0, 1, or 2 such solutions.

If there are no solutions, the ray misses the sphere entirely. There is one solution if the ray is exactly tangential to the sphere or o lies within the sphere (this is perfectly okay). If there are two solutions, the desired one is the one that is closer to o.

The example ends here. If it were to go on, it would be necessary to provide more information about the model: how many lights there are, where they are, and how they interact optically with the object. A physically sound "lighting model" is important to realistic image generation, and has had considerable study in its own right ([Phon75], [Cook82]).

The example above used an analytical surface for the model. Ray tracing, however, can be done with a wide variety of models, including fractals ([Kaji83]), particles (see below), and polyhedra. All that are required are a way of determining intersections and a lighting model.

Ray tracing, then, consists of making this intersection calculation at least once for each pixel on the display and often more than once to allow for intricate lighting models (i.e., reflection, refraction, shadows), and other optical effects.

On a high-resolution display with $N_x \approx N_y \approx 1000$ pixels, ray tracing is expensive even for single images, and if the images are to appear animated (on film at 24 frames/second, say), the expense increases by a factor of 24 for every second of the final sequence.

Much of the work done on the technique has concentrated on improving its efficiency, but even so, without considerable computer resources or specialized hardware, ray tracing is not interactive.

This thesis presents an alternative to a couple of well-established enhancements that speed up ray tracing. The next two sections discuss those enhancements.

1.2. Texture Mapping

Texture mapping was developed by Blinn [Blin76]. Although it has been used with rendering techniques other than ray tracing (sorted polygons, for instance), it fits nicely into the ray tracing environment. The principle is straightforward: given a two-dimensional image in (u,v) called a *texture*, a three-dimensional object in (x,y,z), and a mapping $u = u(x, y, z)$, $v = v(x, y, z)$, do a standard ray-tracing calculation on the object and then transform the intersection points (x_i, y_i, z_i) back into the texture at point $(u(x_i, y_i, z_i), v(x_i, y_i, z_i))$ to define the optical characteristics at each pixel.

Thus one can take a image database of a world with (u, v) being longitude and latitude and map that onto a sphere to produce the image of a globe.

Texture mapping is a fundamental technique of ray-tracing systems, and the current state-of-the-art is highly sophisticated (cf. [Perl85], [Bloo85], [Heck86]).

Nevertheless, a major shortcoming of texture mapping is that it is *two-dimensional*. It works fine when the object has a smooth surface like glass, metal, or plastic, but a rugged or hairy surface such as bark, fur, or rind causes problems. In particular, the edges of the object appear smooth and featureless when they should show the small scale structure silhouetted against whatever is behind the object.

It is also difficult to render shadows caused by an unsmooth surface. For example: Unless the underlying texture took account of the position of the sun[1], a texture-mapped half moon would have a smooth, gradual terminator with neither bright mountain peaks in the shadowed area nor darkened crater interiors in the sunlit area.

Plate 1, a reproduction of Perlin's "Stucco Donut" (from [Perl85]), shows a stucco texture mapped onto a toroid. Note the lack of texture at the edges of the object.

[1]Thus eliminating the whole point of texture mapping in the first place!

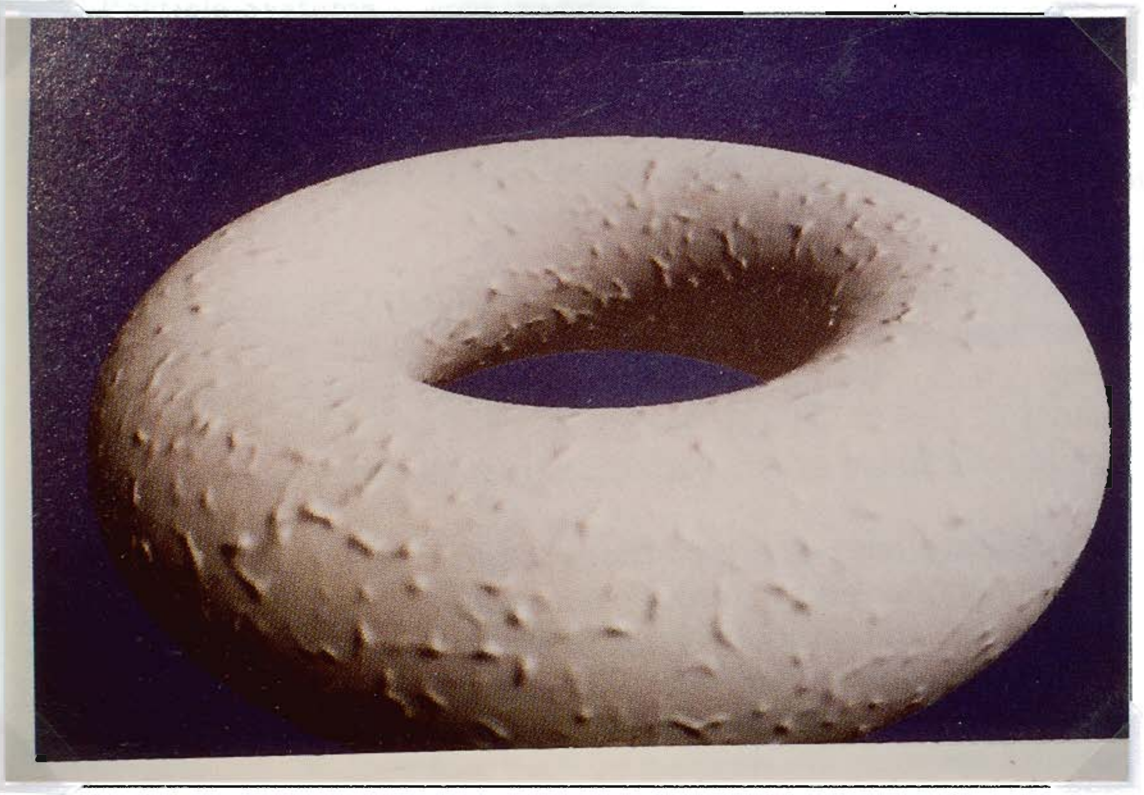

Plate 1 Perlin's "Stucco Donut", showing a stucco texture mapped onto a toroid. Note that the edges of the donut appear smooth and belie the rough texture one would expect for such an object. (from [Perl85])

1.3. Particle Systems

Particle systems, as developed by Reeves [Reev83], are made up of a set of easily-rendered, usually small objects that are allowed to move, tracing out paths on the display. They are like having a generalized paintbrush or collection of brushes, in that there can be any number of them, with arbitrary optical characteristics, tracing paths through the modelling space during image production. It is often useful to constrain particle motion to obey physics ([Reyn87]). Particle systems make motion blur easy to calculate. They have seen wide application ([Yaeg86], [Reyn87]).

In general, particles are not ray-traced, but drawn directly on the display. The only time a ray from the observer to the particle might be calculated would be to see whether some obscuring (ray-traced, non-particle) object blocked the observer's view of the particle.

Particle systems are best used at the limits of resolution. A particle at close range looks no more realistic than a brushstroke. But if a particle system describes small or thin objects (i.e., those whose thickness maps to a few pixels on the display), the result is acceptable.

Particles are especially useful for phenomena like explosions and fireworks, where there many small objects, each of which is easily rendered individually. They do have one major drawback that limits their usefulness. While a particle can interact with its environment by emitting, reflecting, refracting, and blocking light coming from that environment, it cannot interact with other *particles* in the same way. Particles cannot shade each other^2.

The reason is that a particle has no notion of the spatial relationship of its path to those of the others. At a given stage in its motion, a particle knows its own position and that of the other particles, but it would be computationally prohibitive to have the particle reconstruct the trajectories of the others in the past to see if their paths interacted (by one shading the other, say).

Plate 2, a reproduction of Figure 12 of [Reev83], Alvy Ray Smith's well-known "white.sand" image, shows this problem clearly: the particle-generated clump of grass casts a shadow on the sand, but there are no shadows evident on the clump itself.

In addition, when rendering a dense collection of particles, such as a forest or field of grass, the rendering must be done from back-to-front. Suppose particle A's path lies between the path of particle B and the observer: A obstructs B. If the scene were being ray-traced, A's obstruction of B would appear naturally as a result of the distance of the intersection points

^{2}With some foreknowledge of the scene being rendered, Reeves and Blau ([Reev85]) have shown that the effect of self-shaded particles can be produced probabilistically. This does not, however, solve the general problem.

of the ray with the two particles. But particles are not ray-traced -- they are drawn directly on the display, so the only way for A to obstruct B is if B is drawn before A, imposing an observer-dependent ordering on image production. If the observer changes position, this ordering must, in general, be recalculated.

Such an ordering might not even be possible. Suppose A and B were rendering a double helix. The proper rendering order would vary, depending on where along their paths one considered the particles to be. A fixed drawing order would not exist.

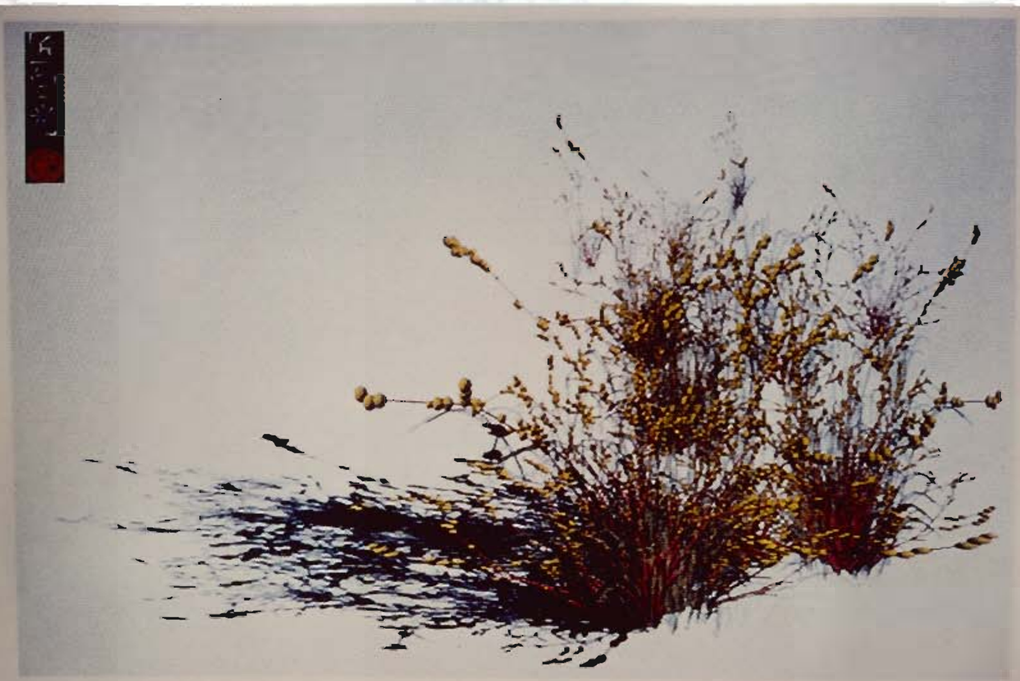

Fig. 12. *white.sand.*

Plate 2 From [Reev83], Alvy Ray Smith's "white.sand" image. Although the particle system grass casts shadows on the ground, it does not shade itself.

CHAPTER 2

LATTICES

Developers of particle systems postulated that on small-scale features one need not go to the same lengths as on large-scale objects to produce acceptable images. This idea is important. This thesis will now discuss an alternative to particle systems, called "lattices", inspired by the same idea.

As with particle systems, lattices are built with an underlying small-scale ray tracer. Instead of turning a set of particles loose in the modelling space, however, let small-scale objects be represented by a small-scale ray tracer that is much faster but not so accurate as the large-scale ray tracer. An interface between the two ray tracers is necessary, so that the large scale ray tracer knows when to invoke the smaller one.

Imagine a small object ("primitive") defined in its own Cartesian coordinate system, called *object coordinates.* Imagine also that a rectangular box completely bounds this object. Let there be a grid on each of the six faces of this box. The x, y, and z extents of the box are integer multiples of the grid spacing (expressed in object coordinates) λ. These grids form the lattice.

One corner of the box becomes the origin of a second coordinate system, called *lattice coordinates*, which, for the time being, is a simple translation of object coordinates[1]. When measured on the faces of the lattice, lattice coordinates are always non-negative integers.

Figure 2 shows an example of this, a 4 by 2 by 4 lattice surrounding a small object that may, for instance, be a small Velcro (tm) hook.

2.1. Rasterization

The process we call "rasterization" constructs the simple, fast ray tracer as follows:

Before rendering the image, for every entry point p_i on the lattice to every possible exit point p_o on the lattice, determine the intersection, if any, with the object. Save the relevant information (the intersection point and the surface normal, discussed below) about all intersections in a lookup table. When there are multiple intersections for the same (p_i, p_o), save only the intersection that is closer to p_i.

Once the table is constructed, begin ray tracing as usual. Each lattice has associated with it a transform from world to lattice coordinates. This allows scaling and positioning of the lattice anywhere, in any orientation.

[1]At some future time, the object-to-lattice transform may be optimized by rotation so that the volume of the lattice is minimized.

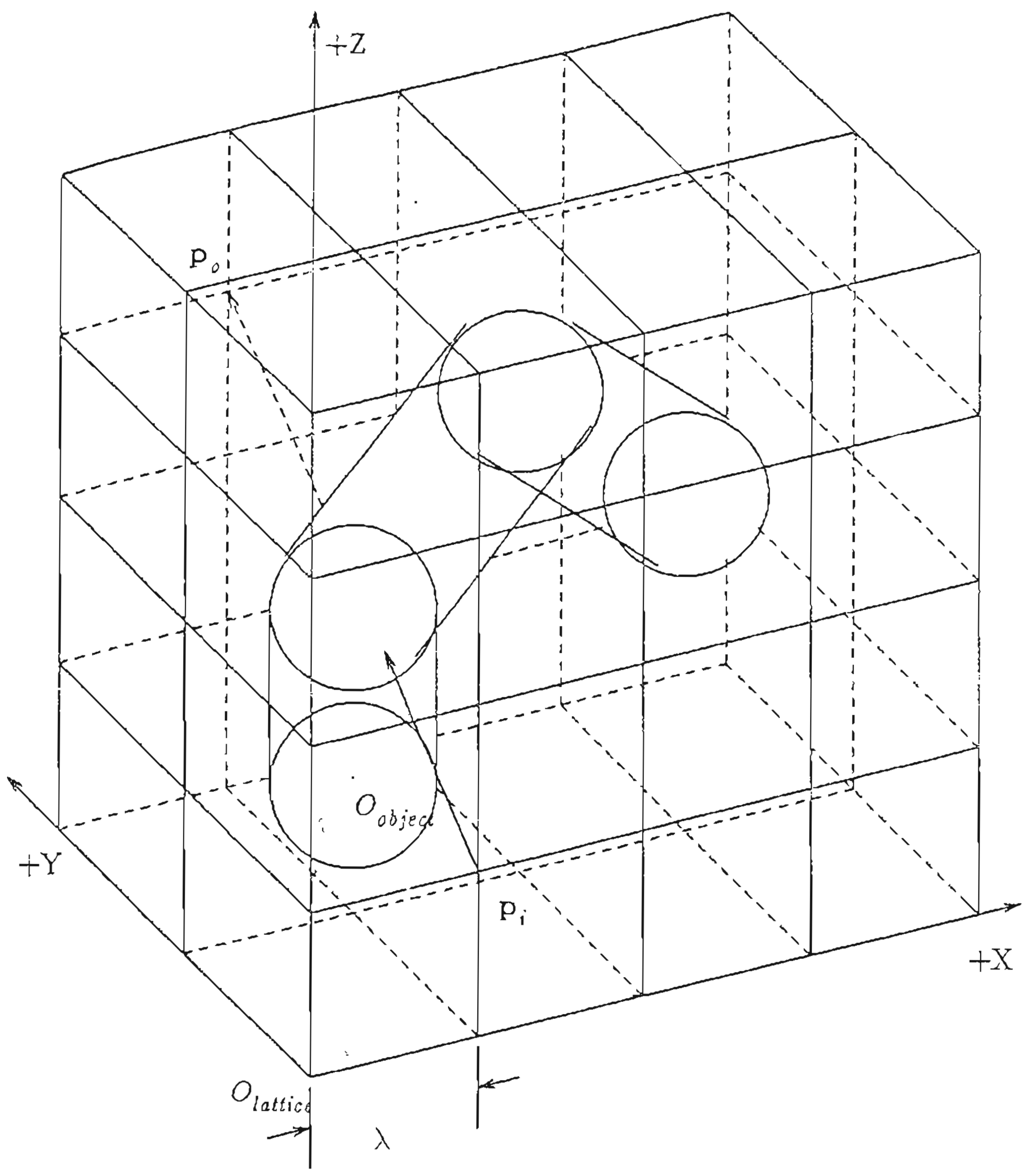

Figure 2 A primitive object ("hook") embedded in a lattice. The object is represented as three spheres connected by cylinders. A typical ray $(p_i, p_o) = ((1\ 0\ 1), (1\ 2\ 3))$ intersects the object. O_{object} and $O_{lattice}$, the origins of the object and lattice coordinate systems, are also shown.

Apply this transform to each ray to convert it to lattice coordinates and see if the ray intersects the lattice's bounding box. If it does, the ray intersects the object if there is an entry for the ray's (p_i, p_o) in the table, and the intersection information is there (although some will need to be transformed, to be discussed below).

The fundamental assumption about lattices in this is that the projection of λ (transformed into world coordinates) subtends no more than a few pixels on the display. The main source of error that would otherwise be encountered would be in the rounding of the ray's initial world coordinates, which are real, to lattice coordinates, which are integer.

2.2. Grid Spacing Considerations

Obviously the grid spacing, λ, plays a critical role in determining the appearance of the object on the display, but the λ also figures heavily into the storage space needed for the lattice hash table (see below).

As the reader may already have guessed, this table is large. Given a lattice of size $N_{cx} \times N_{cy} \times N_{cz}$ grid cells, N_{lp}, the number of pairs of (p_i, p_o) points to be tried[2] is

$$N_{lp} = 2 \left[4 \left(N_{lpxy} \, N_{lpyz} + N_{lpyz} \, N_{lpzx} + N_{lpzx} \, N_{lpxy} \right) + N_{lpxy}^2 + N_{lpyz}^2 + N_{lpzx}^2 \right] \qquad (1.3$$

where

$$N_{lp\alpha\beta} = \left(N_{c\alpha}+1\right)\left(N_{c\beta}+1\right).$$

The initial factor of 2 in (1.3) reflects the fact that (p_i, p_o) must be distinct from (p_o, p_i) for any $p_i \neq p_o$, since, for a primitive object of finite dimension, intersections usually come in pairs, one which is closer to p_i and one which is closer to p_o. Furthermore even though they are often spatially close, their surface normals, which play a major role in determining illumination, may vary widely.

So far, λ has been chosen to be equal to the smallest characteristic feature size (i.e., the thickness) of the primitive object, but this need not be the case. A larger λ might cause parts of the object to be missed during rasterization, but a smaller λ will increase the number of samplings and thus

[2]Actually, this equation should be considered an upper bound on the number of points to be tried. Consider two adjacent faces F1 and F2 and their common edge E. This equation considers only face-face interactions, so the points in E are counted twice. Furthermore, since it includes all rays starting on F1 and ending on F2, it counts rays that begin on E (as part of F1) and are contained entirely within F2. Such rays could not possibly intersect the object, and an efficient tabulation of the intersections could avoid them entirely. The equation also counts corner points three times. All of these considerations would reduce the number of points to try from the value of N_{lp} given in (1.3). Nevertheless, if one considers a cube with N_c cells on a side and hence N_c^2 points on a face, face-face interactions are $O(N_c^4)$, while all other interactions are $O(N_c^k)$, where $k \leq 3$, so that as the lattice gets larger and larger, the quartic terms which make up (1.3) dominate and the actual number of points to try would converge linearly towards the value of N_{lp} given by this equation.

improve resolution. Let $f_{lat} \geq 1$ the *lattice oversampling factor*, be the number of object diameters per λ. As f_{lat} increases, a finer and finer lattice is surrounding the object.

Even without oversampling, both N_{lp} and N_{iz}, the number of intersections, can be large. For the relatively simple object shown in Figure 2, $N_{cx} = 4$, $N_{cy} = 2$, and $N_{cz} = 4$, so, according to (1.3), $N_{lpxy} = 15$, $N_{lpyz} = 15$, and $N_{lpzx} = 25$. There are then $N_{lp} = 9{,}950$ (p_i, p_o) pairs to try! Table 1 shows similar calculations, along with the actual numbers of intersections found, for a variety of objects and f_{lat} values.

	f_{lat}	N_{lp}	N_{ir}
sphere radius: 0.5λ	1 2 3	2430 18750 72030	510 3486 13614
straight line length: 10λ	1 2	24030 238750	10766 93614
straight line length: 100λ	1 2	1190430 13018750	570926 5368334
spiral radius: 5λ height: 10λ	1 2	33698 353950	11190 111804
Figure 2 "hook"	1 2	9950 95742	3224 30266
hair	1 2	522450 6928158	100504 1366229

Table 1 Some typical lattice intersection calculations.

For the sake of discussion, imagine a cubic lattice, with N_c being the number of cells on an edge, then $N_c = N_{cx} = N_{cy} = N_{cz}$. According to (1.3), the number of points to try goes like $O(N_c^4)$. This is not, however, a measure of the size of the table, only of the number of points to try. The way the number of intersections scales with N_c depends on the of the object. For a hair, or filament, it is $O(N_c^3)$. There is an easy way to see this intuitively.

Imagine an observer at a particular entry point $\mathbf{p}_i$ on the lattice, looking at the object. The observer sees the projection of the filament intersecting a certain number of exit points, $\mathbf{p}_o$, on the lattice. Doubling the length of the filament (and keeping λ equal to the filament thickness), the lattice must also double in each dimension to continue to surround the filament. But, from the observer's point of view, the number of intersections $\mathbf{p}_o$ for the given $\mathbf{p}_i$ has only doubled, i.e., it has only increased linearly with N_c. Since the number of possible $\mathbf{p}_i$ points has squared, the net effect on the number of intersections is that of $O(N_c^3)$.

2.3. The Lattice Hash Table

How best to store intersections for the fast ray tracer? There are six (integer) lattice coordinates in $(\mathbf{p}_i, \mathbf{p}_o)$, so one could implement the table as a 6-dimensional array. But as seen in Table 1, this table would be huge for most non-trivial lattices. Note, however, that the table is sparse for filamentary structures.

The obvious way to store the information, then, is in a hash table whose index is constructed from (p_i, p_o). This is the *lattice hash table*, or LHT. The information this table has to contain is:

- (p_i, p_o) itself, since the hashing cannot be guaranteed to be perfect
- the point of intersection, in real[3] lattice coordinates
- the surface normal of the object at the intersection (this determines reflection and other optical properties discussed below)
- a pointer or index into the hash overflow area

In theory, this would require 4 integers and 6 reals for each entry. If each quantity required 4 bytes, each entry would require 40 bytes. For tables with ~100,000 entries or so, this would use a lot of memory! One can reduce the size of an entry with the following constraints:

- Restrict (integer) lattice coordinates to fit in a single byte. Assuming 8 bits/byte, this means that the maximum extent of the object can be no more than 255 λ[4]. This constraint is not significant. The pair (p_i, p_o)

[3]Although lattice coordinates are integers on the surrounding grids of the lattice, the intersection points are, in general, real.

[4]Since the bounding lattice must surround the object completely, it is necessary to allow 2 coordinate values, 0 and 255, which are guaranteed to be outside the object.

then takes up 6 bytes.

- Since the intersection point lies along a line from p_i to p_o, express this intersection as a scalar value s, $0 \leq s \leq 1$ such that the intersection is $p_i + \left(p_o - p_i\right)s$ and further scale s by 255 to fit in a single byte. There is a small but acceptable truncation error here.

- Since the surface normal has unit magnitude, scale each component by 127 to fit in a signed 8-bit quantity with, again, a small, acceptable truncation error.

Probably nothing should be done with the overflow pointer. If the lattice were small, it might be restricted to 16 bits, but Table 1 shows that typical lattices are likely to run over 2^{16} intersections.

One can thus get the table entry size down to 14 bytes. The final chapter of this thesis will discuss possible further reductions.

2.4. Lighting Model

Given way of determining intersections, the next step for the ray tracer is to determine the intensity of light at the intersection point. For what follows, see the ray diagram in Figure 3.

The interaction of light with the material an object is made of determines the object's appearance. To cover a wide range of materials, models for this interaction can be complex ([Cook82]). Fortunately, there is a

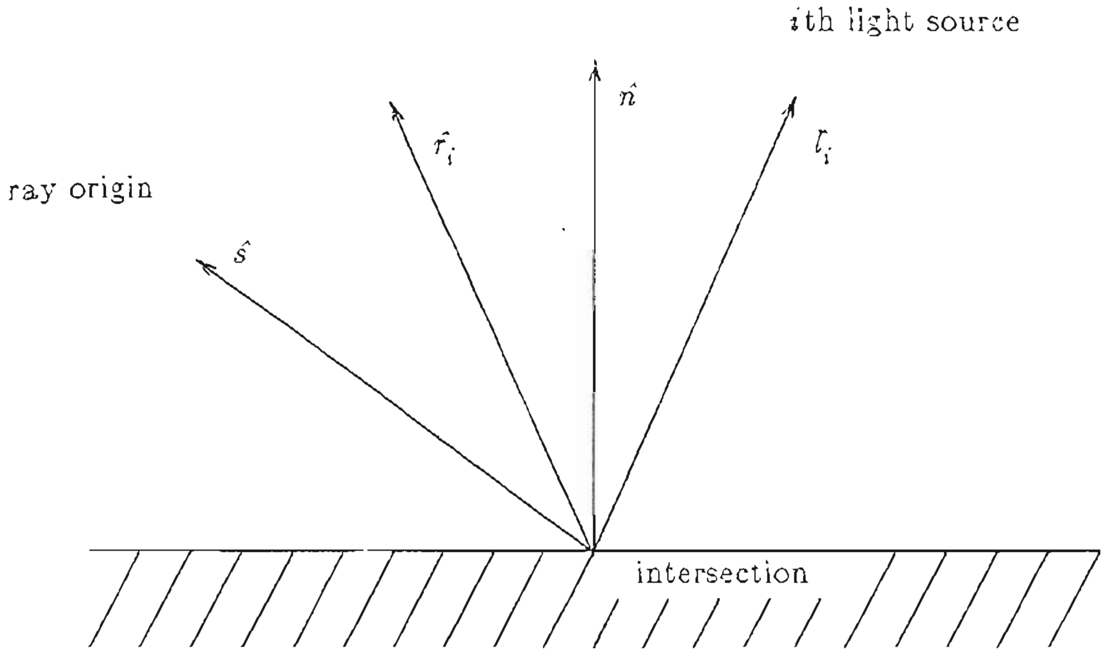

Figure 3 The geometry at the intersection point. The text explains the various directions.

simple model dating back to [Phon75] that is comparatively fast to calculate and will be good enough to illustrate the effectiveness of lattices. The basic equation is (see Figure 3):

$$I = k_{amb} I_{amb} + \sum_{i=1}^{N_{vs}} \frac{I_{inc,i}}{D_i + K} \left[k_{diff} \left(\hat{n} \cdot \hat{l}_i \right) + k_{spec} \left(\hat{r}_i \cdot \hat{l}_i \right)^{n_{spec}} \right] \qquad (2.3$$

where k_{amb} is the ambient reflection coefficient, I_{amb} is the ambient light intensity, k_{diff} is the diffuse reflection coefficient, N_{vs} is the number of (non-ambient) light sources visible from (i.e., not shading) the intersection, $I_{inc,i}$ is the the incident intensity of the ith source at some canonical distance, D_i is the distance from the ith source to the intersection, K is an arbitrary constant (mainly to fudge the inverse-square law, see [Roge85] for more information), $\hat{n}$ is the surface normal of the object at the intersection, $\hat{I}_i$ is the normal in the direction of the ith light source, k_{spec} is the specular reflection coefficient, $\hat{s}$ points from the observer to the intersection, $\hat{r}_i = 2\left(\hat{I}_i \cdot \hat{n}\right)\hat{n} - \hat{I}_i$ is the reflected ray from the ith source, as given by the Fresnel equation (angle of incidence = angle of reflection), and n_{spec} is the specular reflection exponent. k_{diff}, k_{spec}, and n_{spec} are all empirically derived.

In this simple model, (2.3) is monochromatic and the resulting intensity at a given wavelength λ is dependent only on the incident intensities and the optical properties of the object at that wavelength, not at any other wavelength. The wavelength-dependent quantities in (2.3) are k_{amb}, I_{amb}, k_{diff}, $I_{inc,i}$, k_{spec}, and n_{spec}.

Equation (2.3) does not include the possibility of transmission, which would be an additional term, and since this model assumes that the incident light could only be coming directly from an unshadowed light source, it will not allow the refractive or mirror reflection effects so popular in other ray tracers. Neither of these features is necessary, however, for demonstrating lattices, and there is no conceptual problem with their co-existence with lattices. Both of them could be added to the ray tracer with little difficulty.

The relationship of (2.3) to lattice information is in the major role played by the surface normal $\hat{n}$ in determining the intensity.

It may seem unnecessary to save the intersection point as well as $\hat{n}$ for each (p_i, p_o). One might think it would be enough to use the center of the lattice as the intersection point for the purposes of the ray-object intersection calculation. For that calculation, it *would* probably suffice, but there is a more subtle dependency on the intersection point in (2.3). The more exact intersection point is needed because it affects the shadowing calculation; that is, the set of N_{vs} sources over which the summation in (2.3) takes place. Using the center point of the lattice would never allow the primitive to shade itself.

2.5. Example: A Thistle

Plates 3 and 4 show a simple application of a lattice: the rendering of a "thistle" made up of 100 randomly-oriented line segments, each of length 20λ. For emphasis, both plates enlarge the segments more than the recommended $\lambda \approx$ a few pixels. Plate 3 shows this object without lattices shadowing each other, as would be the case if rendered by a particle system. Plate 4 shows the effect of allowing lattices to shadow each other.

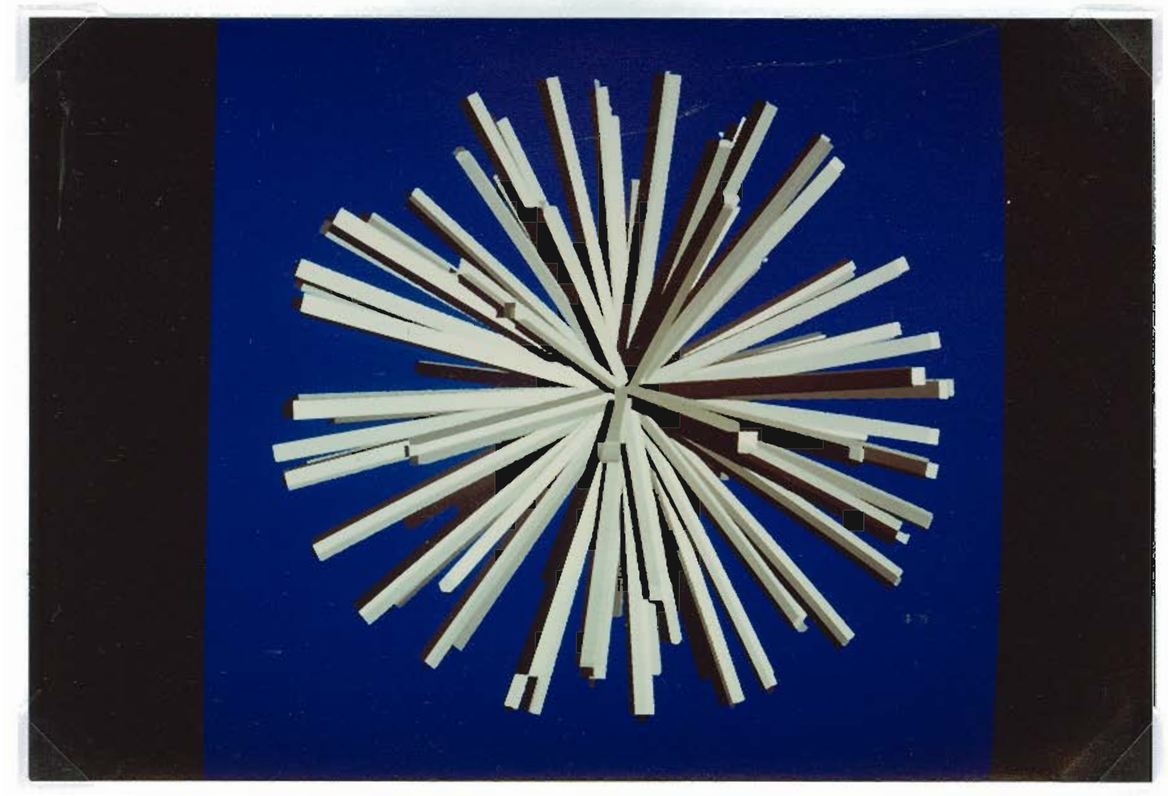

Plate 3 100 randomly-oriented straight lattice primitives without self-shading. This is equivalent to what a particle system would produce.

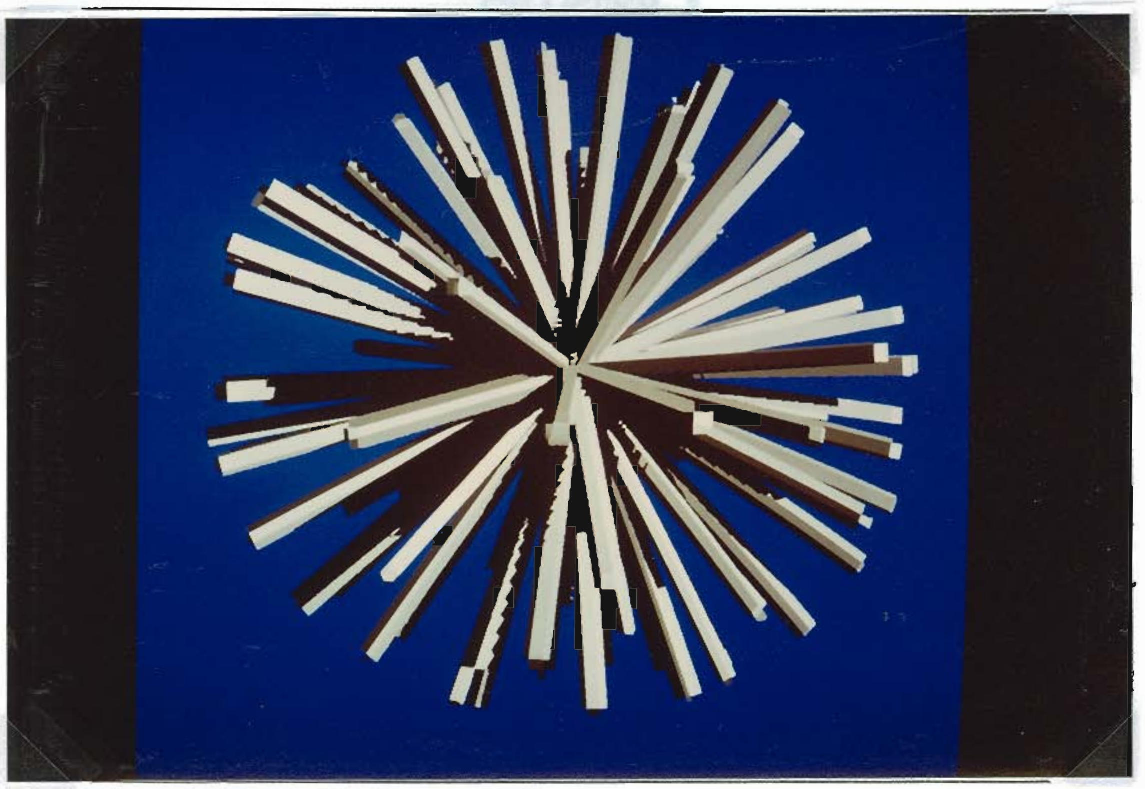

Plate 4 100 randomly-oriented straight lattice primitives with self-shading. Particles in a particle system cannot shade each other. (The jagged edges are a result of the enlargement of the figure for purposes of illustration and are not detectable at the recommended scale for lattice primitives of a few pixels.)

CHAPTER 3

LATTICES IN A CONVENTIONAL RAY-TRACING SYSTEM

In order for lattices to be useful, it should be possible to combine them with more traditional rendering techniques. That is what this chapter covers.

3.1. Constructive Solid Geometry

Section 1.1 discussed how to compute ray-object intersections when the object is a sphere. The kinds of objects usually rendered, however, are more complicated than spheres. A sphere has a simple analytical representation, but rendering a sphere with a hole through it, for example, is difficult at best if one is restricted to purely analytical means.

The *constructive solid geometry* (CSG) (cf. [Requ80]) approach starts with simple primitive objects like spheres and defines a set of logical operations that combine those simple objects into more complicated ones. Although the choice of primitives varies from system to system (the ones chosen here will be discussed below), the operations are invariably the same.

Let A and B be two objects and $O = A\ op\ B$ be the result of the dyadic logical operator op acting on A and B. If op is *union*, O consists of all points that belong to either A or B (or both). If op is *intersection*, O

consists of only those points that belong in both A and B. If *op* is *difference*, O consists of only those points in A that are not in B.

For example, to model a sphere with a hole in it, let A be a sphere, B be a cylinder (transformed to the proper location), and *op* be *difference*.

The CSG representation of any object, then, is a binary tree whose leaf nodes are primitive objects (spheres and such) and whose non-leaf nodes are logical operators acting on their child objects. Transformations between world and primitive (or lattice) coordinates need only be kept at the leaf nodes, although during construction of the tree, it is useful to allow transformations to be performed on non-leaf nodes and propagated downward at that time. Figure 4 illustrates this.

Ray tracing a CSG tree is straightforward. Given a ray, for each leaf node build a list of intersections of the ray with that primitive object. The list should be ordered in increasing distance from the observer. Then, proceeding bottom-up from the leaf nodes, for each non-leaf node build a list of intersections by merging the lists of its children according to rules determined by the value of *op* for the node. (For more details, see [Requ80].)

The first element in the resulting intersection list is the intersection point of the ray with the composite object. The lighting model can then be applied. This step will include tracing an additional ray from the intersection point in the direction of each of the light sources to determine whether

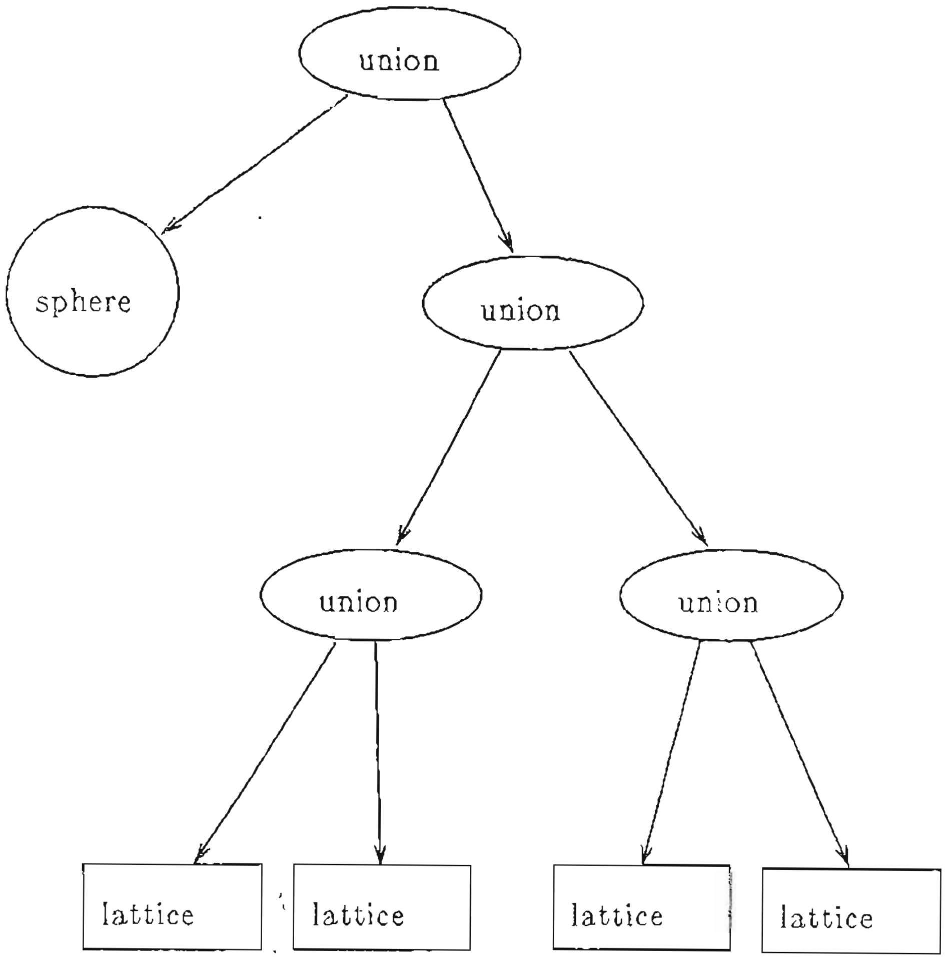

Figure 4 A CSG tree. There are four lattice primitives and a spherical quartic primitive.

or not that ray intersects the composite object, i.e., whether or not the point is in that light source's shadow.

The point to note here is that it does not matter how one gets intersection lists for the leaf nodes. This means that the only criterion for primitive objects in a CSG system is the ability to build ray-object intersection lists for them.

3.2. Primitives

There are many possible objects that can serve as primitive objects for a CSG system. This thesis will choose lattices (obviously) and quartics (defined below). Other types of primitive (fractals, polyhedra, splines, or whatever) are feasible, but only one non-lattice primitive is sufficient to demonstrate how well lattices fit into a CSG scheme. Also, quartics provide flexibility.

3.2.1. Quartics

A quartic is a polynomial in x, y, and z of the form

$$f(x, y, z) = \sum_{i=j=k=0}^{i+j+k \leq 4} a_{ijk}\, x^i y^j z^k \tag{3.1}$$

The surface of the quartic is defined by $f = 0$, and, by convention, $f > 0$ outside the quartic and $f < 0$ inside it. Quartics can represent a wide variety of figures. If f contains only constant and linear $(i + j + k \leq 1)$ terms, it represents a half-space. Several halfspaces can be intersected to form a polyhedron. although this may not be the most efficient way to do so. Spheres are obviously quartics, as are ellipsoids, paraboloids, hyperboloids. and toroids.

If one extends i, j, and k, to take on any positive values, (3.1) will describe a general 3-d analytical surface. The reason for restricting $i + j + k$ to be less than or equal to 4 is purely practical. Extending the example from Section 1.1, intersecting a ray with a quartic will produce polynomials of degree of at most $i + j + k = 4$ to be solved for t, hence the name "quartic". Closed-form solutions do not exist for arbitrary polynomials of degree greater than 4. The quartic restriction thus avoids having to deal with the iterative and comparatively slow rootfinding techniques needed at higher degrees.

A useful feature of quadrics is that the surface normal, which analytic geometry shows to be ∇f, is easy to compute in closed form by purely symbolic methods.

3.2.2. Lattices

Chapter 2 describes the representation of lattice primitives in an LHT. As was mentioned there, each lattice also has an arbitrary transform from world coordinates (in which we initially define the rays) into lattice coordinates. This transform serves two purposes. First, to clip the ray against the lattice. Second, to transform the normal from the LHT's lattice coordinates back into world coordinates to find the illumination from (2.3).

3.3. Implementation Details

There is now a system of programs, written in C and running under the UNIX(TM) operating system, that implements the ideas of the previous sections. The ray tracer, *model*, is about 11,700 lines long. The program *rasterize*, which constructs the LHT, is about 1,400 lines long. In addition, both programs use a vector and matrix arithmetic package about 1,700 lines long and an LHT construction and lookup table package about 1,900 lines long.

Several minor programs generate tables of transformations (special purpose for the examples presented in this thesis), map the 18-bit colors that *model* produces to a color-mapped display with fewer than 18 planes (using the algorithm in [Heck82]), and render the result on either an Apollo DN660 workstation or a Tektronix 4125 graphic terminal[1].

Out of necessity, the software compiles and runs on Sun workstations under SunOS(TM), Apollo workstations under Domain/IX(TM), VAXen under BSD UNIX(TM), and Tektronix workstations under UTek(TM), although not all programs have been tested on all platform/OSes.

3.4. Example: A Fuzzy Sphere

Plate 5 shows a closeup view of the "hair" lattice used in this example. Similar to Figure 2, but larger, the hair was modelled as 10 spheres connected by 9 cylinders.

[1]For the author's own amusement, a version for the long-obsolete Tektronix 4027 graphic terminal is now under development.

Plate 6 shows 1000 hairs placed randomly on a sphere, the result of combining quartics with lattices. An infinite plane (actually a half-space) extends below a sphere that has been covered with almost-uniformly-spaced hairs. Unlike texture mapping, the edges show three-dimensional texture. The hairs cast shadows from the two light sources (of differing intensity) onto the plane, the sphere, and, unlike particle systems, themselves. It took 12.1 hours of CPU time to render this 512 by 512 pixel image on an unloaded Sun 3/60. The system had 8MB of real memory, much more than *model* needed, so paging was minimal.

Plate 7 is similar to Plate 6, but with 5000 hairs. This image took 25.5 hours of CPU time to render on the same configuration.

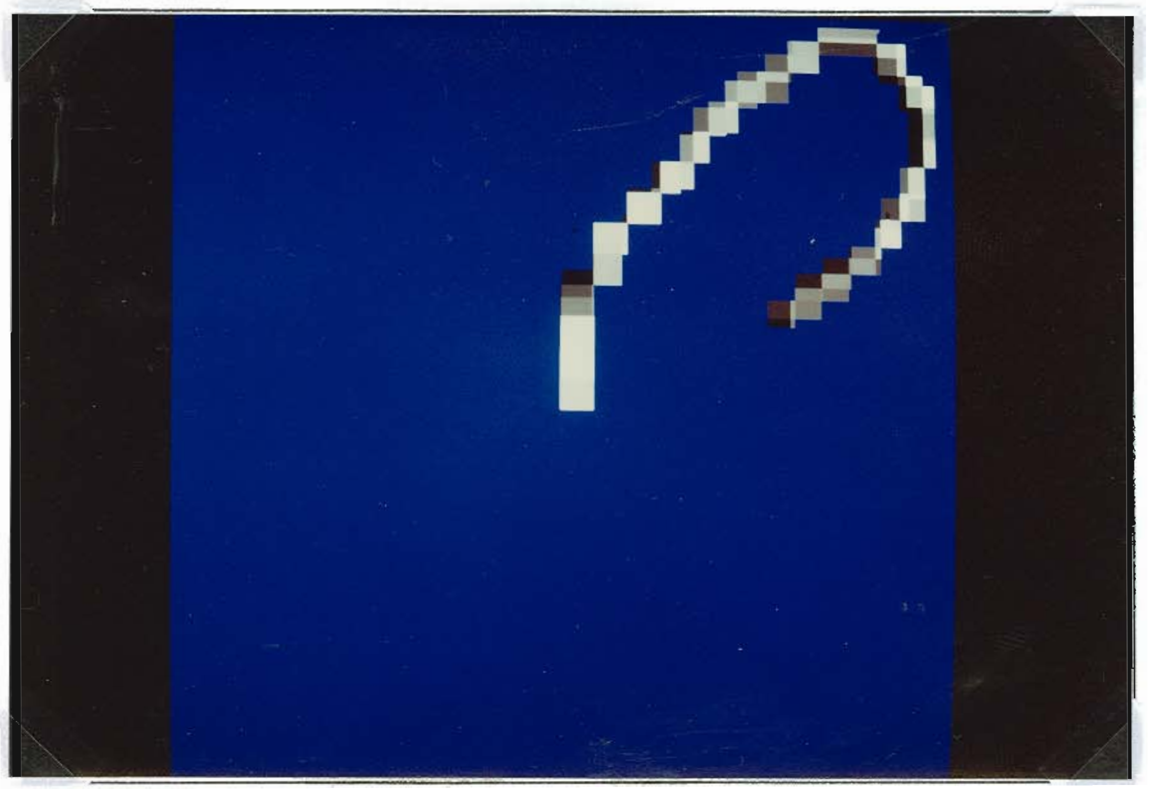

Plate 5 A closeup of a "hair" lattice primitive. This is the same primitive used in both Plates 6 and 7.

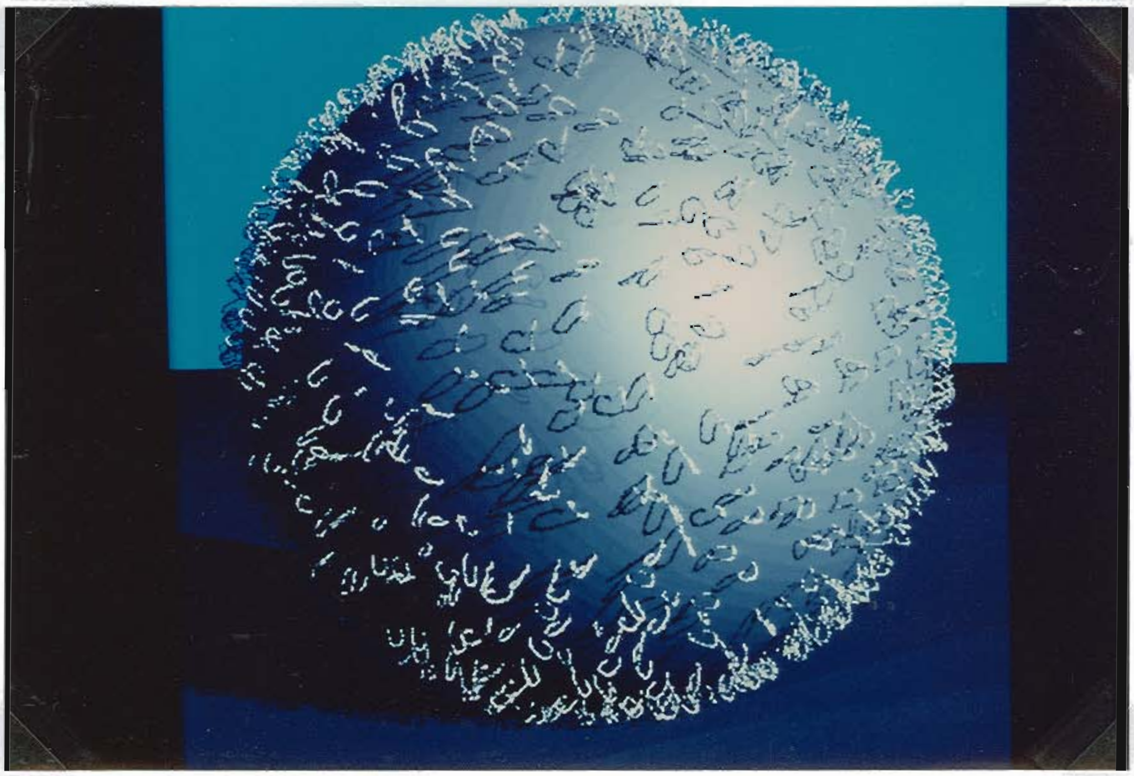

Plate 6 1000 "hairs" placed randomly on a sphere. Notice how the texture at the edges of the sphere appears against the background.

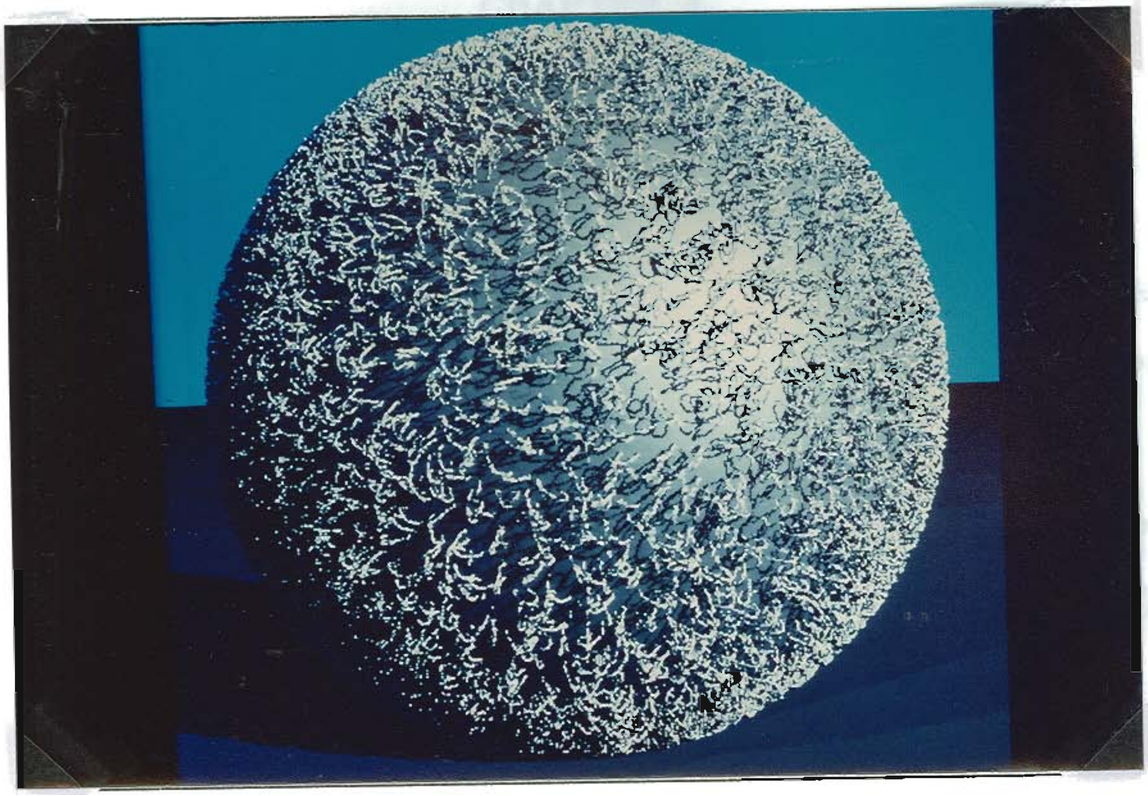

Plate 7 5000 "hairs" placed randomly on a sphere.

CHAPTER 4

CONCLUSIONS

This thesis has shown that using lattices for three-dimensional texturing can produce acceptable results. For rough objects, this technique produces more realistic images than either texture mapping or particle systems.

The three most common criteria used in evaluating techniques of rendering realistic images are fidelity ("Does the image look realistic?"), speed ("How quickly was the image rendered?"), and, to a lesser degree, space ("How much memory did rendering the image require?"). Table 2 qualitatively summarizes the findings of this thesis.

	texture mapping	particle system	full ray trace	lattice
fidelity	low	medium	high	high
speed	very fast	fast	slow	moderate
space	small	small	medium	large

Table 2 Qualitative summary of results.

Note that Table 2 has included the option of "full ray trace". A full ray trace would remove the intermediary lattice primitives and represent them with composite objects made up of the primitives used to construct the lattice primitive (spheres connected by cylinders in the case of the "hair" of

Plate 5). The large-scale ray tracer would then render the result. For any but the most trivial primitives, this would increase the number of leaf nodes by a large factor (19 in the case of Plates 6 and 7) to a value that would slow down ray tracing and use more memory. Not having to do this is the main rationale for texture mapping, particle systems, and lattices in the first place, so full ray tracing is included in the table only for purposes of comparison.

What about a more quantitative comparison? As far as fidelity is concerned, there is no well-established quantitative criterion. Readers must judge Plates 3, 4, 6, and 7 for themselves. If one is willing to use lattice oversampling, a degree of fidelity arbitrarily close to that of a full ray trace can be achieved (at a cost of increased memory usage).

Preceeding sections presented a quantitative analysis of memory usage. In summary, a typical lattice takes up a few megabytes. An application that did not have the memory space available would be ill-advised to use lattices.

Something quantitative needs to be said about time efficiency. The times cited above amount to 170 msec/pixel for Plate 6 and 350 msec/pixel for Plate 7. The only way to do a direct comparison between lattices and a ray tracer with surface texturing or particles would be to build such a system and run it on the same configuration rendering the same image. This is beyond the scope of this thesis, but even it it were to be done, the comparison might still be considered "biased" because there would be no guarantee that either of the traditional techniques was implemented in its most efficient form.

Nevertheless, we can make a semi-qualitative judgement by citing a couple of results in the literature. Even the very early [Whit80] has examples of surface texturing requiring between 9 and 24 msec/pixel on a VAX 780 (a slower CPU than the Sun 3/60). depending on the scene complexity. For particle systems, [Reev85] shows results between 12 and 143 msec/pixel on a VAX 750 (even slower than the 780), depending on the number of particles and the scene complexity.

The time required to render a surface texture is independent of the appearance of the texture. The time to render a particle system is proportional to the number of particles. Profiling indicates that (as with most ray tracers) intersection calculations for lattices and their bounding boxes dominate time usage. The time required to render a lattice system, then, is proportional to the mean number of intersections encountered tracing the CSG tree. Ideally, this should be the logarithm of the number of lattice primitives. The fact that increasing the number of lattice primitives by a factor of 5 from Plate 6 to Plate 7 only increased the rendering time by a factor of about 2 illustrates this.

Using these considerations, it is possible to scale data given in [Whit80] and [Reev85] to get the very approximate comparisons shown in Table 3.

	texture mapping	particle system	full ray trace	lattice
Plate 6	1	1.5	51	12
Plate 7	1	3.5	120	25

Table 3 Gross estimation of relative efficiency. All times are relative to that of texture mapping ≡ 1, which should be the same absolute amount of time for both images.

Surface texturing will probably always be the fastest technique of the three, but because of the linear-vs.-logarithmic dependencies, for a large enough number of primitives, lattices should be more efficient than particle systems. This number may, however, be too large for practical usefulness.

CHAPTER 5

FUTURE DIRECTIONS

This thesis has shown the conceptual validity of lattice techniques. Continued work with lattices should move in the direction of practicality. There are two principle aspects of lattices that need improvement before the technique can be said to be practical: time efficiency and memory usage.

5.1. Time Efficiency

Profiling *model* shows that very little time (less that 1%) is spent looking up intersections. Most time is spent computing intersections with bounding boxes[1]. This is consistent with other work ([Whit80], for example). Apart from the bounding boxes, no optimizations such as described in [Kaji83] or other literature have been implemented to speed up intersection computations. This should be done.

5.2. Memory Usage

The other drawback to using lattices is the space required for the LHT. When each primitive requires about one megabyte of storage, this can eat

[1]Clearly, this arises from having so many distinct objects to be ray traced.

up virtual memory quickly. There are three possible ways in which the memory requirements could be reduced.

5.2.1. Compress the LHT

Section 2.3 presented some considerations that allowed reduction of the size of each entry in the LHT from 40 bytes to 11 bytes. Other reductions might be possible.

For example, it is necessary to keep (p_i, p_o) around because the hashing scheme is not perfect. If the set of values to put in the table is known from the start, as it is now, it should be possible to devise a perfect hashing scheme. Omitting (p_i, p_o) would reduce the size of each entry to 5 bytes.

Alternatively, it would be interesting to study how many entries in the LHT are actually referenced while tracing a typical object. It may be advantageous to reject points that would otherwise go into the overflow area, if there is only a small chance that they would ever be needed.

Another way to reduce the size of the LHT might be to treat the problem as one of compression of a digital signal.

5.2.2. Compute the LHT on Demand

Along these lines, one might try constructing the LHT on demand during ray tracing. This means that the LHT would start out empty, and as rays intersected lattices at (p_i, p_o) pairs, make the intersection calculation then. The LHT would grow accordingly. An approximate calculation of N_{lookup}, the number of times the LHT is consulted, will show why this is useful.

Without allowing for transmission or multiple reflections, the total number of rays generated during rendering is $N_{ray} \approx (1 + N_s)N_x N_y$ where N_s is the total number of light sources. For a typical display, $N_x \approx N_y \approx 1000$. Generously suppose there are $N_s = 3$ light sources. That means that $N_{ray} \approx 4 \times 10^6$. N_{lookup} is $N_{ray}\overline{n_{iz}}$, where $\overline{n_{iz}}$ is the average number of lattice intersections per ray. This will depend on the number of primitives in the image as well as their size. Assume that, as above, the CSG tree has a bounding-box test so that rays that miss a lattice completely are quickly rejected. If there were 10,000 primitives placed randomly within the viewing volume and the projection of each one subtended an area of 100 pixels, $\overline{n_{iz}}$ would still be of order unity, so $N_{lookup} \approx 4 \times 10^6$. Compare this number with N_{iz}, the actual size of the lookup table required, shown Table 1. Even for some of the simple primitives here, N_{iz} approaches 4×10^6. This means that the ray trace is looking up each entry in the LHT once on the average. This is not even considering the fact that many times one would be looking up the *same* (p_i, p_o), so that in practice many of the N_{iz} entries in the LHT are never referenced at all.

Furthermore, consider what happens for larger lattices. Section 2.2 showed that N_{iz} is $O(N_c^3)$ for a lattice with N_c cells on an edge. Since it only depends on the projection of the lattice on the display, $\overline{n_{iz}}$ is proportional to the area of that projection, which is $O(N_c^2)$. N_{ray} does not depend on N_c, so N_{lookup} is $O(N_c^2)$ also. Hence, N_{iz}/N_{ray} is $O(N_c)$. This means that as N_c increases, it will eventually become desirable to avoid the lookup table altogether and calculate intersections during ray tracing. Where the break-even point occurs depends on both the model and the relative efficiency of the lookup and intersection calculations.

One complication on-demand LHT computation would add would be that the LHT would have to contain information on both hits and misses, but in the latter case, of course, there would be no intersection information to save[2].

The additional time required to build the LHT this way would not be a a problem. While the rendering time would increase, the overall time to produce the LHT and render it would decrease since not as many lattice points

[2]It would be efficient to maintain two LHTs of differing entry sizes: one for hits, containing the usual intersection information, and one for misses, with only (p_i, p_o) and an overflow pointer. One could place arbitrary limits on the size of either table, rejecting (p_i, p_o) pairs after a table had been filled up. The only consequence would then be that one might have to recompute the same (p_i, p_o) more than once, but even so, this would yield more flexibility in trading space for time than the current scheme.

would have to be evaluated. In any case, the time it takes to build the LHT is typically a few minutes, whereas ray tracing takes hours, so doing both at once should not hurt.

This approach would link the space used by the LHT to the complexity of the model being rendered.

5.2.3. Construct a Stochastic Lattice Hash Function

This alternative approach would seek the replacement of the LHT entirely by replacing the data in it with a rapidly-evaluated function whose parameters were derived statistically from the original data. Instead of seeking the exact representation that the previous approach would maintain, this approach would trade some exactness for recovering some address space.

A fractal approach might prove useful here.

REFERENCES

Bloo85 Bloomenthal, J., Modeling the Mighty Maple. *Computer Graphics, 19,* 3 (SIGGRAPH 85 Proceedings) 305-311.

Blin76 Blinn, J. F., and Newell, M. E., Texture and Reflection in Computer-Generated Images. *Commun. ACM 19,* 10 (October, 1976), 542-547.

Cook82 Cook, R. L., and Torrance, K. E., A Reflectance Model for Computer Graphics. *ACM Trans. Gr. 1,* 1 (January, 1982) 7-24.

Fole82 Foley, J. D., and Van Dam, A., *Fundamentals of Interactive Computer Graphics.* Addison-Wesley Publishing Company, Reading, Mass., 1982.

Gold71 Goldstein, R. A., and Nagel, R., 3-D Visual Simulation. *Simulation 16,* 1 (January, 1971) 25-31.

Heck82 Heckbert, P. S., Color Image Quantization for Frame Buffer Display. *Computer Graphics, 16,* 3 (SIGGRAPH 82 Proceedings) 297-307.

Heck86 Heckbert, P. S., Survey of Texture Mapping. *IEEE Computer Graphics and Applications 6,* 11 (November, 1986).

Kaji83 Kajiya, J. T., New Techniques for Ray Tracing Procedurally Defined Objects. *ACM Trans. Gr. 2,* 3 (July, 1983) 161-181.

Phon75 Phong, B. T., Illumination for Computer Generated Pictures. *Commun. ACM 18,* 6 (June, 1975), 311-317.

Perl85 Perlin, K., An Image Synthesizer. *Computer Graphics, 19,* 3 (SIGGRAPH 85 Proceedings) 287-296.

Reev83 Reeves, W. T., Particle Systems -- A Technique for Modeling a Class of Fuzzy Objects. *ACM Trans. Gr. 2,* 3 (April, 1983) 91-108.

Reev85 Reeves, W. T., and Blau, R., Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems. *Computer Graphics, 19,* 3 (SIGGRAPH 85 Proceedings) 313-322.

Requ80 Requicha, A. A. C., Representations for Rigid Solids: Theory, methods, and systems. *ACM Comp. Surv., 12,* 4, Dec. 1980.

Reyn87 Reynolds, C., Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics, 21,* 4 (SIGGRAPH 87 Proceedings) 25-34.

Roge85 Rogers, D. F., *Procedural Elements for Computer Graphics.* McGraw-Hill Book Company, New York, 1985.

Yaeg86 Yaeger, Y., Upson, C., and Myers, R., Combining Physical and Visual Simulation -- Creation of the Planet Jupiter for the Film "2010", *Computer Graphics, 20,* 4 (SIGGRAPH 86 Proceedings) 85-93.

Whit80 Whitted, T. An Improved Illumination Model for Shaded Display *Commun. ACM 23,* 6 (June, 1980), 343-349.

VITA

Bob Lewis received his BS in Physics at Harvey Mudd College in 1974. He received an MA in Astronomy at the University of California at Berkeley in 1979. His part-time graduate studies in Computer Science began at the University of Santa Clara in 1981 and continued at the Oregon Graduate Center in 1983 when he moved to Portland.

He first worked with computer graphics in 1977, when, as a research assistant in the Astronomy Department at UC Berkeley, he developed an interactive program for the analysis of stellar spectral data using the venerated Tektronix 4010 storage tube terminal.

Making a career change from Astronomy to Software Engineering, he has worked for several firms specializing in Computer-Aided Design for Electrical Engineering, including GE-Calma, Comsat General Integrated Systems, and Tektronix. While at the latter, he was project leader for the TekniVIEWS(TM) graphics terminal-based window manager.

He is currently employed as a Senior Software Engineer at Test Systems Strategies, Inc. and is a member of the IEEE Computer Society, ACM/SIGGRAPH, and the Portland Computer Arts Resource Committee.