THREE-DIMENSIONAL TEXTURING USING LATTICES

Robert R. Lewis
Oregon Graduate Institute of Science and Technology[1]

7 September, 1990

Abstract

This paper describes a way to perform realistic three-dimensional texturing of ray-traced objects with irregular surfaces. Such texturing has been done in the past with texture mapping, particle systems, or volumetric methods. We propose an alternative to these called a *lattice*. Lattices work as fast but inexact ray tracers. As long as lattices are used for small objects, though, the inexactness doesn't show on the scale of the display, and the result is acceptable.
The paper shows how lattices can be integrated with a more traditional ray tracer, with several examples.
Time and memory space considerations are major constraints on lattices, preventing widespread practical application at the present time. The paper discusses these limitations and how they might be reduced.

1 Introduction

This paper discusses a new technique for adding surface detail to objects represented in a three-dimensional rendering system. We will confine the discussion to a ray-traced renderer. Such systems are capable of a high degree of realism. The kinds of objects we will focus our attention on are those which have features on their surface which are much smaller than, but not negligible in comparison to, the object itself.

Examples of such "hairy" objects are a fuzzy tennis ball, a field of grass, tree bark, a furry animal, or a human head with hair.

Two traditional approaches to generating images with these kinds of objects are texture mapping and particle systems.

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 state-of-the-art in texture mapping is highly sophisticated (cf. [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 hairy surface 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.

The other popular approach to rendering hairy objects, particle systems ([Reev83]), uses a set of easily-rendered, usually small objects that are allowed to "move" during rendering, tracing out paths on the display like a collection of generalized paintbrushes. Particle systems have also seen wide application. In general, particles are not ray-traced, but drawn directly on the display.

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 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 (except probabilistically, as in [Reev85]). In general, particles cannot shade each other.

[1] Author's current address:
University of British Columbia; Computer Science Dept.; Vancouver, BC V6T 1W5; CANADA
Internet: bobl@cs.ubc.ca

The reason is that a particle has no notion of the spatial relationship of its path to those of the others.

Recent non-traditional approaches to three-dimensional surface texturing ([Kaji89], [Perl89], and [Lewi89]) treat the problem as one of volume rendering. While this is capable of producing quite realistic results, the rendering time is considerable. Kajiya and Kay's "teddy bear" ([Kaji89]) required between one half and one cpu-hour on a configuration of 16 mainframes. (According to [Crow89], it works out to about 10^{12} floating point operations.)

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.

We will now discuss an alternative to both texture mapping and particle systems, called "lattices", inspired by that idea.

Imagine a small object ("primitive") surrounded by a rectangular box. 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 σ. These grids form the lattice. When measured on the faces of the lattice, lattice coordinates are always non-negative integers. Figure 1 shows an example of this: a 4 by 2 by 4 lattice surrounding a small object that resembles a small hook.

Lattices have an underlying small-scale ray tracer that is faster than but not so accurate as a conventional ray tracer. Later on, we'll consider how to combine the two.

2.1 Rasterization

The process we call "rasterization" constructs the small-scale ray tracer as follows: Before rendering the image, from every entry point $\mathbf{p}_i$ on the lattice to every exit point $\mathbf{p}_o$ on the lattice, determine the intersection, if any, with the object. Save the relevant information (see below) about all intersections in a lookup table. In case of multiple intersections for the same ($\mathbf{p}_i$, $\mathbf{p}_o$), save only the intersection that is closest to $\mathbf{p}_i$.

Then begin ray tracing as usual. Each lattice has a transform from world to lattice coordinates which allows scaling, orientation, and positioning of the lattice. 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 ($\mathbf{p}_i$, $\mathbf{p}_o$) in the table, and the intersection information is there.

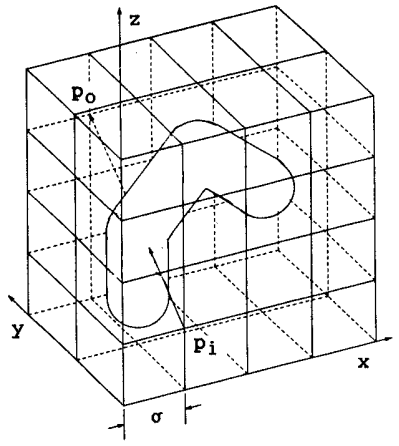

Figure 1: **A primitive object ("hook") embedded in a lattice. The object is represented as four spheres connected by cylinders. A ray (p_i, p_o) = ((1 0 1), (1 2 3)) intersects the object.**

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

2.2 Grid Spacing Considerations

σ plays a critical role in determining the appearance of the object, but σ also figures heavily into the storage space needed for the object representation. This space is large. To get a rough estimate of how large, suppose we are given a lattice of size N_{cx} by N_{cy} by N_{cz} grid cells.

primitive	σ^{-1}	N_{lp}	N_{ix}	N_{ix}:N_{lp}
sphere	1	2.4 (3)	5.1 (2)	0.21
diameter: 1	2	1.8 (4)	3.5 (3)	0.19
	3	7.2 (4)	1.4 (4)	0.19
cylinder	1	2.4 (4)	1.1 (4)	0.45
length: 10	2	2.4 (5)	9.4 (4)	0.39
diameter: 1				
cylinder	1	1.2 (6)	5.7 (5)	0.48
length: 100	2	1.3 (7)	5.4 (6)	0.41
diameter: 1				
spiral	1	3.4 (4)	1.1 (4)	0.33
radius: 5	2	3.5 (5)	1.1 (5)	0.32
height: 10				
Figure 1	1	9.9 (3)	3.2 (3)	0.32
"hook"	2	9.6 (4)	3.0 (4)	0.32
Plate 3	1	5.2 (5)	1.0 (5)	0.19
"hair"	2	6.9 (6)	1.3 (6)	0.20

Table 1: Some typical lattice intersection calculations.

$N_{lp\alpha\beta}$, the number of points on each face bounded by edges parallel to axes α and β is given by

$$N_{lp\alpha\beta} = (N_{c\alpha} + 1) \times (N_{c\beta} + 1) \qquad (1)$$

Intersecting rays can start of any of six faces and end on any of five faces. Thus N_{lp}, the number of pairs of ($\mathbf{p}_i$,$\mathbf{p}_o$) points to be tried, has 30 terms. Combining identical terms yields

$$N_{lp} = 8\left(N_{lpxy}N_{lpyz} + N_{lpyz}N_{lpzx} + N_{lpzx}N_{lpxy}\right) + 2\left(N_{lpxy}^2 + N_{lpyz}^2 + N_{lpzx}^2\right) \qquad (2)$$

This equation should be considered an upper limit. See [Lewi88] for an explanation. There is no easy analytical way to determine N_{ix}, the actual number of intersections.

So far, σ has been chosen to be equal to the smallest characteristic feature size (i.e., the thickness) of the primitive, but this need not be the case. As σ decreases, a finer and finer lattice surrounds the object. A smaller σ will increase the number of samplings and intersections, but it will also improve resolution. Even without decreasing σ, both N_{lp} and N_{ix} can be large. For the relatively simple object shown in Figure 1, $N_{cx} = 4$, $N_{cy} = 2$, and $N_{cz} = 4$, so, according to (1) and (2), $N_{lpxy} = 15$, $N_{lpyz} = 15$, and $N_{lpzx} = 25$. There are then Nlp = 9.950 ($\mathbf{p_i}$, $\mathbf{p_O}$) pairs to try! Table 1 shows similar calculations, along with the actual numbers of intersections found, for a variety of objects and σ^{-1} values.

For the sake of discussion, imagine a cubic lattice, with Nc cells on an edge, then $N_c = N_{cx} = N_{cy} = N_{cz}$. According to (1) and (2), the number of points to try goes like $O\left(N_c^4\right)$. 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 Nc depends on the nature of the object, but for a given object, the ratio of Nix to Nlp is roughly constant.

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 ($\mathbf{p}_i$, $\mathbf{p}_o$). This is the *Lattice Hash Table*, or LHT. The information this table has to contain is: ($\mathbf{p}_i$, $\mathbf{p}_o$) itself (since the hashing cannot be guaranteed to be perfect); the point of intersection, in real[1] lattice coordinates; the surface normal of the object at the intersection (this determines reflection and other optical properties discussed below); and 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 considerable amount of memory! With considerations described in [Lewi88], the size of an entry can be reduced to 14 bytes with minimal impact. The final section of this paper will discuss possible further reductions.

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

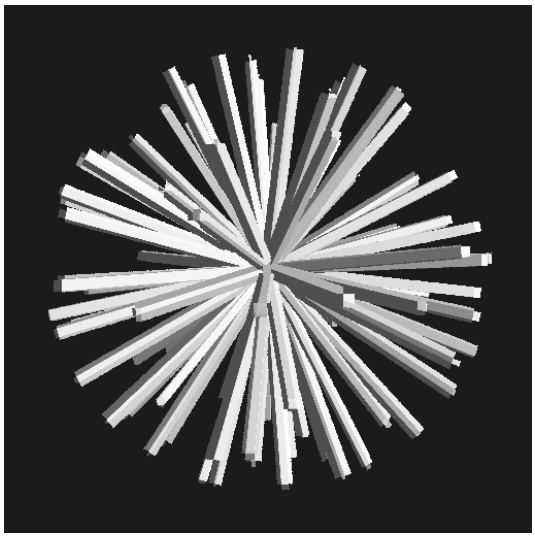

Plate 1: **100 randomly-oriented straight lattice primitives without self-shading**

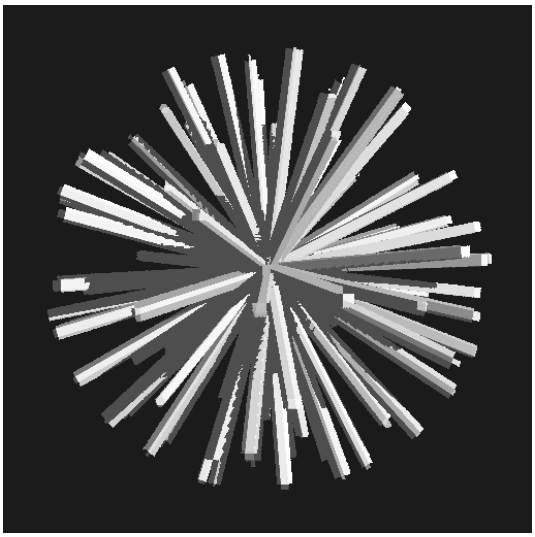

Plate 2: **100 randomly-oriented straight lattice primitives with self-shading**

2.4 Lighting Model

Given a way of determining intersections, the next step for the ray tracer is to determine the intensity of light at the intersection point. Realistic models of this interaction can be quite complex ([Cook82]), but there is a model dating back to [Phon75] that is comparatively simple and is good enough to illustrate the effectiveness of lattices.

The formula for Phong shading is well-known and we need not reproduce it here. Its connection to lattice information is in the major role played by the surface normal **N** in determining the intensity.

It may seem unnecessary to save the intersection point as well as **N** for each ($\mathbf{p}_i$, $\mathbf{p}_o$). There is, however, a subtle dependency on the intersection point in Phong shading. The exact intersection point is needed because it affects the shadowing calculation; that is, the set of sources over which the summation of specular and diffuse intensities takes place. Using the center point of the lattice as the intersection point, for instance, would never allow the primitive to shade itself.

2.5 Example: A Thistle

Plate 1 and Plate 2 show a simple application of a lattice: the rendering of a "thistle" made up of 100 randomly-oriented line segments, each of dimensions 1 by 1 by 20. For emphasis, both plates enlarge the segments more than the recommended few pixels.

Plate 1 shows this object without lattices shadowing each other, as would be the case if rendered by a particle system.

Plate 2 shows the effect of allowing lattices to shadow each other. (In both cases, 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.)

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 modelling and rendering techniques.

The first element in the resulting intersection list is the intersection point of the ray with the composite object. The lighting model, with shadowing, can then be applied.

Note that it does not matter how one gets intersection lists for the leaf nodes. The only requirement for primitive objects in a CSG system is the ability to build ray-object intersection lists for them. This is what will allow us to fit lattices into CSG.

3.1 Constructive Solid Geometry

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.

The CSG representation of any object is a binary tree whose leaf nodes are primitive objects and whose non-leaf nodes are logical operators acting on their child objects. 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 is 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 logical operator for the node. (For more details, see [Requ80].)

3.2 Primitives

There are many possible objects that can serve as primitive objects for a CSG system. We will choose lattices (obviously) and quartics. 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.

3.2.1 Quartics

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

$$f(x,y,z) = \sum_{i=j=k=0}^{i+j+k\le 4} a_{ijk} x^i y^j z^k \qquad (3)$$

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, including half-spaces, spheres, ellipsoids, paraboloids, hyperboloids, and toroids.

The restriction $i + j + k \le 4$ is purely practical. Intersecting a ray with a quartic produces polynomials of degree of at most 4 to be solved for intersections. 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. (3) also permits easy computation of the surface normal, $\frac{\nabla f}{\|f\|}$, in closed form by purely symbolic methods.

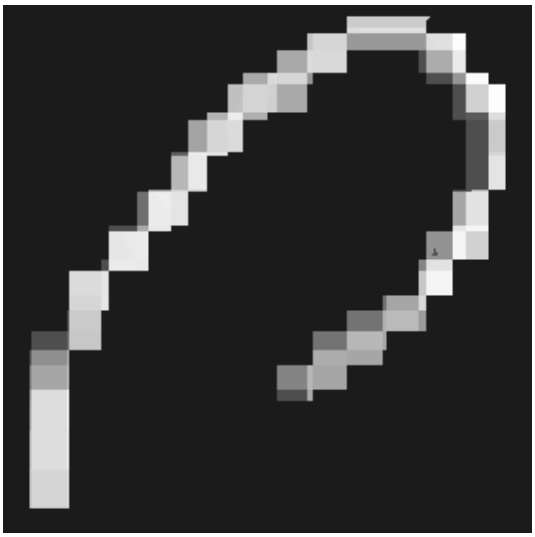

Plate 3: **A closeup of a "hair" lattice primitive. This is the same primitive used in both Plates 4 and 5.**

3.2.2 Lattices

Section 2 describes the representation of lattice primitives in an LHT. The lattice transform described there 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 the Phong formula.

3.3 Implementation

We have developed a system of programs, written in C and running under UNIX[tm], that implements the ideas of the previous sections. The major components are the ray tracer, *render*, and the program *rasterize*, which constructs the LHT.

3.4 Example: A Fuzzy Sphere

Plate 3 shows a closeup view of the "hair" lattice used in this example. Similar to Figure 1, but larger, the hair was modelled as 10 spheres connected by 9 cylinders.

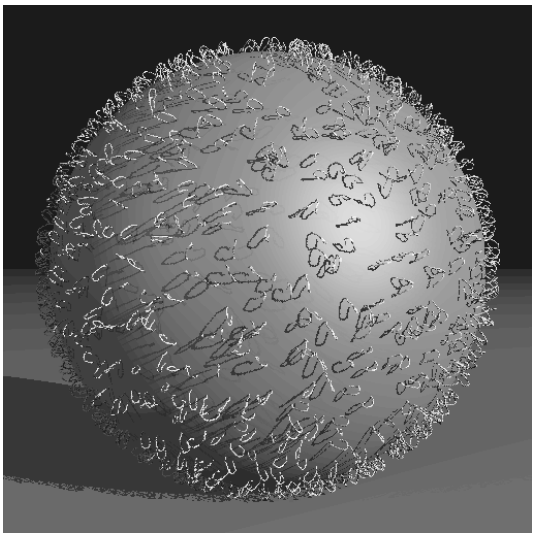

Plate 4: **1000 "hairs" placed randomly on a sphere. Notice how the texture at the edges of the sphere appears against the background.**

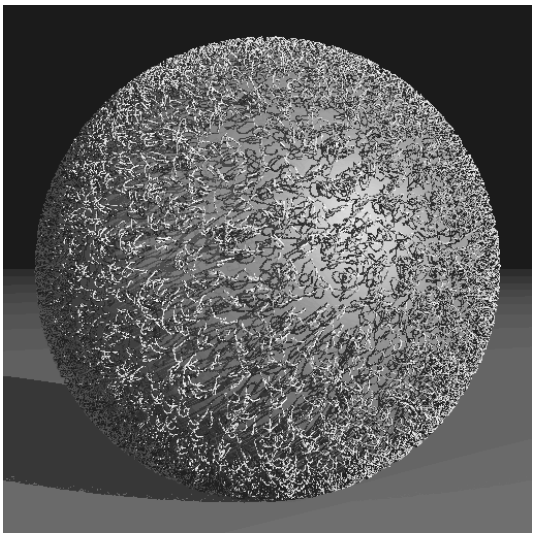

Plate 5: 5000 "hairs" placed randomly on a sphere.

Plate 4 shows 1000 almost-uniformly-spaced hairs on a sphere, the result of combining quartics with lattices. An infinite plane (actually a half-space) extends below the sphere. 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 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 *render* needed, so paging was minimal.

Plate 5 is similar to Plate 4, but with 5000 hairs. This took 25.5 hours of CPU time to render on the same configuration.

4 Conclusions

This paper 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.

Table 2 includes the option of "full ray trace": one that would remove the intermediary lattice primitives and represent them with CSG objects made up of the primitives used to construct the lattice primitive (spheres connected by cylinders in the case of the "hair" of Plate 3). The large-scale ray tracer would then render the result. The three most common criteria used in evaluating techniques of rendering realistic images are fidelity, speed, and, to a lesser degree, space. Table 2 qualitatively summarizes our findings.

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 4 and 5) 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.

	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.

5 Future Directions

We have shown the conceptual validity of lattice techniques. Continued work with lattices should move in the direction of practicality, with particular attention paid to improving time efficiency and memory usage.

5.1 Time Efficiency

Profiling *render* shows that very little time (less that 1%) is spent looking up intersections. Most time is spent computing intersections with bounding boxes[2].

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

5.2.1 Compress the LHT

Section 2.3 mentioned considerations that allowed reduction of the size of each entry in the LHT from 40 bytes to 14 bytes. Other reductions might be possible.

For example, it is necessary to keep ($\mathbf{p}_i$, $\mathbf{p}_o$) 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 ($\mathbf{p}_i$, $\mathbf{p}_o$) and the overflow pointer would reduce the size of each entry to 5 bytes.

5.2.2 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.

[2] Clearly, this arises from so many distinct objects being ray traced.

Acknowledgements

This work was done in partial fulfillment of the author's Master's Degree at the Oregon Graduate Institute. The thesis committee (Bart Butell, chair; Dr. David Maier; Dr. Jack Gjovaag; and Dr. Richard Hamlet) provided direction and offered some helpful suggestions.

Tom Hamilton, Mark Faust, Ron Lunde, and Jeff Hahs of Test Systems Strategies, Inc. were all generous in providing computer resources. System administrator Joe Pruett provided useful advice and cooperation.

References

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.

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.

Kaji89 Kajiya, J. T., and Kay, Timothy L., Rendering Fur With Three Dimensional Textures. Computer Graphics, 23, 3 (SIGGRAPH 89 Proceedings) 271-281.

Lewi88 Lewis, R. R., Three Dimensional Texturing Using Lattices. M. S. Thesis, Oregon Graduate Center, Dept. of Computer Science and Engineering Tech. Rpt. 88-038, 1988.

Lewi89 Lewis, J. P., Algorithms for Solid Noise Synthesis. Computer Graphics, 23, 3 (SIGGRAPH 89 Proceedings) 263-270.

Perl89 Perlin, K., and Hoffert, E. M., Hypertexture. Computer Graphics, 23, 3 (SIGGRAPH 89 Proceedings) 253-262.

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

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).

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

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