An Efficient Scanline Visibility Implementation

Andrew Woo
Steve Chall

Style! Division, Alias Research Inc.
110 Richmond Street East
Toronto, Ontario
M5C 1P1

1. Abstract (Résumé)

In this paper, we present an efficient scanline visibility implementation, one which requires minimal memory and little precomputation, yet provides high quality anti-aliasing for both silhouettes and shading. We also suggest several improvements on conventional scanline strategies.

Voici, une version efficace de l'algorithme de visibilité par balayage. Elle utilise un minimum de mémoire, peu de calculs préliminaires, et produit pourtant des résultats de grande qualité pour l'anti-aliasage des bords de silhouette ainsi que pour l'ombrage. Nous présentons plusieurs améliorations des stratégies conventionelles des algorithmes de balayage.

Keywords: anti-aliasing, normalization, parameterization, point sampling, scanline, shading, span, tessellation, texture mapping, visibility.

2. Introduction

We sought a versatile renderer, one that could be used as a fast previewer and still provide high quality results when called upon to do so. The personal computers for which our implementation was intended imposed additional constraints: relatively modest processing power and a small amount of available memory. Having read papers describing several prior implementations [Watk70] [Fium83] [Croc84] [Fole84], we decided that a scanline visibility algorithm offered what we needed: an efficient means of generating complex images. The following is a discussion of our own scanline-based renderer, its background and implementation.

The fundamental scanline visibility algorithm comes in two basic flavors: a spanning approach [Watk70] and a point sampling approach (similar to ray tracing [Whit80]). The spanning approach gives more accurate horizontal edge anti-aliasing since it provides an exact analytic solution. It is very costly, however: span-scanline intersection calculations can be as expensive as $O(m^2)$ when dealing with highly tessellated surfaces, where m is the maximum number of polygon spans that can exist in any one scanline. Another question that arises, concerning finely tessellated surfaces, is whether the exact intersected span size calculations are worth it. In other words, why bother with intersected span sizes for edge anti-aliasing when they rarely contribute to the silhouette anyway? Besides, the intersected spans are usually very small and numerous for closed surfaces.

Additionally, spanning approaches only take care of horizontal edge anti-aliasing. In most implementations, vertical filtering is done to complete the edge anti-aliasing, with results that are usually unacceptable in terms of image quality. More sampling is needed, which leads us to sample more subscanlines within each row of pixels.

Our scanline uses a point sampling approach. Samples are processed in horizontal scanline order, top to bottom, left to right. This approach provides effective edge anti-aliasing with super-sampling, and linear transparency can trivially be included in the illumination model. Not only does super-sampling permit edge anti-aliasing, it also lends itself to the effective shading anti-aliasing of highlights, spotlights, texture noise, shadows, and so forth. These are shading phenomena which are usually very poorly handled in spanning scanline implementations (which generally offer only one shading call per span within a pixel).

Furthermore, we feel that past implementors have dwelt excessively on the virtues of coherence, but have given inadequate attention to issues involving tessellated surfaces and large numbers of tiny polygons in general. Our approach does take some advantage of coherence, but it also deals effectively with circumstances involving tessellated surfaces, in which coherence considerations are of little use.

We have concentrated heavily on memory conservation and contiguity. This is another issue that has been ignored in the past, but is of crucial importance in dealing with tessellated surfaces. The processing of large numbers of small polygons may run up the primitive count very quickly, and as a consequence put severe strains on the memory capabilities of even the largest machines.

3. Our Scanline Approach

Our scanline method deals with triangles only, usually resulting from the tessellation of curved surfaces. The following is pseudocode which demonstrates its general flow. The variables (in bold face) will be explained in upcoming sections.

```
Build ySortList;
Sort ySortList in decreasing Y (sec 3.3);

For each scanline in decreasing Y order
     {
     Get active triangles from ySortList;
     Compute x spans of active triangles and
          store in yActiveList (sec 4.3);
     Sort yActiveList in decreasing X (sec 3.3);

     For each sample that contains geometry
          in increasing X order
          {
          Get active triangles of current
               sample from yActiveList
               and place into xActiveList;
          Compute Z depths in xActiveList (sec 4.2);

          /* To deal with transparency */
          Do until visible surface is opaque
               or if background
               {
               Find closest Z depth;
               Interpolate Normal and Shade
                    Point on closest surface
                    (sec 4.4);
               Mark surface implicitly deleted;
               }
          Combine shading information into rgb;
          }
     }
```

3.1. Memory usage

Scanline data structures usually consume massive amounts of memory. However, by rendering triangles, we can reduce memory significantly, since we know that a triangle can contribute at most one span to a given scanline (a fact which, of course, holds true for any convex polygon). There are three arrays for which we need to allocate memory. The first is an O(n)-sized array called *ySortList*, where n is the number of triangles in the scene. An element of this array contains only three fields: a pointer to minimal world-space coordinate information (in our case, shared vertex information including vertices, normals and texture indices) and the y bounds of the triangle in screen space.

The other two arrays, *xActiveList* and *yActiveList*, are of size O(m), where m is the maximum number of triangles that can possibly intersect any one scanline. Bounding box evaluations of the triangles give us the value of m. We know that $m \leq n$, and that usually m is significantly smaller, especially when the triangles are evenly distributed throughout the scene (see section 6.1).

yActiveList contains the screen-space geometry of the current scanline's active triangles. *xActiveList* contains pointers to the currently active triangles with respect to the current pixel. The following offers a quick view of some of the relationships between the above arrays.

```
struct ySortListType
     {
     WorldCoord *tri;
     unsigned short minY;
     unsigned short maxY;
     } ySortList[n];

struct yActiveListType
     {
     struct ySortListType *tri;
     unsigned short minX;
     unsigned short maxX;
     ScreenGeom info;
     } yActiveList[m];

struct xActiveListType
     {
     struct yActiveListType *tri;
     } xActiveList[m];
```

ySortList is sorted according to the *maxY* field in decreasing order. *maxY* determines the topmost scanline in which the triangle is active, and *minY* determines the final scanline past which the triangle is no longer active, that is to say, is no longer used in the scene. Many implementations use an array of linked lists for *ySortList*. However, this requires much more memory, and the memory used is non-contiguous, whereas our array data structure is small, compact and (possibly) contiguous. In addition, with this data structure, we can easily skip empty scanlines to get to the next non-empty scanline.

yActiveList is sorted according to the *minX* field in increasing order, and may also be stored in contiguous memory. *minX* indicates the leftmost pixel of the current scanline in which the triangle is active, and *maxX* indicates the final intersected pixel in the current scanline, beyond which the triangle is no longer used. *info* represents the triangle's parametric and geometric information in screen space, and will be discussed in detail in section 4 (it may also contain colour interpolation information [Gour71]). This array can be O(m) since the information it contains is dynamically generated and compacted. This compaction causes implicit deletion of triangles no longer used, without having to explicitly free the memory.

3.2. Integerizing Parts of the Scanline

Since we are computing the scene one scanline at a time, we can integerize each scanline, and thus the *maxY* and *minY* values can also be integerized. For a triangle with floating point bounds of *floatYMax* and *floatYMin*, we simply perform the following assignments: $maxY = floor(floatYMax)$ and $minY = ceil(floatYMin)$. We can also restrict *maxY* and *minY* to 16 bits, which means a maximum y resolution of 2^{16} (65536), which, for the immediate future, will not inconvenience us excessively.

In the x direction, we can repeat the same trick since we are point sampling. Thus, all span computations with

floating point bounds (see section 4.3) will be *ceil*ed and *floor*ed into *minX* and *maxX* values, respectively. This means that we sample at the corners. This is no different than sampling at the middle of the pixel (a more popular approach) other than a half pixel shift.

Why do we bother with integerizing these four fields? First, we need to sort on *maxY* and *minX* (see next subsection), and floating point sorting tends to be much slower than 2-byte integer sorting. In addition, we need to traverse the arrays that contain the fields in order to locate the active triangles (*maxY*, *minX*), as well as release the inactive triangles (*minY*, *maxX*). So a lot of processing is done using these fields.

3.3. The Sorting Schemes

We have two integer lists to sort: *ySortList* and *yActiveList*. Some implementations need to sort the z values for each pixel too, but we feel this is unnecessary since only the smallest z value is desired for opaque surfaces, or only a small set of candidates in the case of transparency (see pseudo-code at the beginning of section 3). We would have liked to use bin sort for both *ySortList* and *yActiveList*, but an effective bin sort requires a linked list data structure [Aho83], and compaction to our array structure would not be worth the expense. Thus, we have had to resort to other sorting schemes.

For *ySortList*, we cannot assume any partial ordering. Furthermore, we want a sorting scheme which does not require too much additional memory. A fast $O(n\log n)$ method should be implemented. A non-recursive quicksort [Knut73] [Sedg84] appears to be a good choice, with insertion sort to deal with small subsets. In terms of additional memory used, only a couple of hundred bytes are needed for a large n around 1 million. It should be mentioned that we are not too upset at our inability to use $O(n)$ bin sort for *ySortList*, since this sort is only done once per scene and on a 2-byte key.

For *yActiveList*, most implementations use bubble or insertion sort because it is felt that a partial ordering can be sustained from the previous scanline. We do keep information from the previous scanline, then add newly active triangles for the current scanline. With finely tessellated surfaces, however, this partial ordering does not help speed up the sorting in the majority of cases, and both these sorting schemes are a slow $O(m^2)$. Thus, we want a fast sort whose performance suffers neither from partially ordered (e.g., not quicksort) nor unordered input. We chose heap sort ($O(m\log m)$) for *yActiveList*, but on closer examination, it proved to be slow for small lists. We then decided to use insertion sort $O(m^2)$ if there are fewer than 200 triangles to be sorted.

3.4. Anti-Aliasing

We use a super-sampling scheme for anti-aliasing, where a uniform grid of points $l \times l$ is set up within each pixel, and l can be a user input for controlling the rendering quality. A sampling grid of 3×3 or 4×4 is usually sufficient for NTSC resolution displays. Accommodating the integer data structures is just a matter of scaling the data structure values by l. We sample l subscanlines per row of pixels.

We can also sample at the edges of pixels both vertically and horizontally, so that visibility and *rgb* results can be shared between neighbouring pixels in all directions. This resembles adaptive anti-aliasing as it is used in most ray tracing implementations [Whit80]. Thus an $l \times l$ grid only requires $(l-1)(l-1)$ samples, with $l-1$ subscanlines.

With a point sampling approach, we know that the geometry visibility determination is still fast (see section 4). However, to compute the shading information at each such point within a pixel is very expensive, and can be overkill. Usually, the shading information is already anti-aliased; we would like to avoid unnecessary shading calls. So we only super-sample the geometry on the $l \times l$ grid (for guaranteed high quality edge anti-aliasing), and conserve on the shading computations (i.e., avoid super-sampling for shading anti-aliasing). Conservation of shading computations is discussed in the following subsections.

3.4.1. One Shading Call per Pixel

To avoid an excessive number of shading calls, we require another $O(x)$-sized array (call it *keepObj*), where x is the horizontal resolution. In this array, we record the *rgb* illumination value most recently generated while sampling the current pixel, as well as the surface (not triangle) most recently hit. If the next sample in the same pixel belongs to the same visible surface as before, then we avoid shading by simply reusing the saved *rgb* value. If the next sample belongs to a different surface, then we need to shade and save the new *rgb* as well as the visible surface in the data structure. This usually results in only one shading call per pixel.

Note that the array information is nulled for each new row of pixels to be processed, so that at least one shading call will be made per pixel. In addition, note that we need an $O(x)$ array because we are sampling in a strictly horizontal direction per subscanline - recall that l subscanlines constitute a row of pixels.

3.4.2. Adaptive and Stochastic Shading Calls

With only one shading call per pixel, there is a danger of aliased highlights, spotlights, and shadows, plus noisy textures (in particular, procedural textures), and so forth. Adaptive sampling [Whit80] provides a simple technique for deciding whether additional shading calls are necessary. Unfortunately, we cannot use it directly since we are sampling in a strictly horizontal direction.

We can, however, use a quasi-adaptive [Whit80] and stratified [Lee85] sampling approach for shading. For each sample on the $l \times l$ grid, it is stochastically determined with some probability P whether or not to perform shading calculations. We used the probability

$$P = \frac{l-1}{l^2-1}.$$

If, for a given sample, the probability P dictates that shading is not to be performed, then the strategy described in section

3.4.1 is followed. With each shading call, we also compare the current *rgb* values with those of the previous sample in the current pixel as well as in the left and right neighbouring pixels: we use the *rgb* values stored in *keepObj*. If it is the first sample taken in the current pixel, then we can also compare with the previous row of information to check *rgb* discrepancies. If some large discrepancy in *rgb* values results, then we will be forced to super-sample the shading for the current pixel and its neighbour.

In short, we super-sample the geometry for edge anti-aliasing, and adaptively-stochastically sample on the $l \times l$ grid for shading anti-aliasing. The following pseudo-code illustrates our per-sample anti-aliasing scheme:

```
Sample Geometry to Determine Visibility;

if (geometry belongs to different surface ||
    stratified probability "P" to sample ||
    forced to super-sample shading in
        current pixel)
    {
    RGB = ShadePoint;
    Compare RGB with previous sample and
        neighbour pixels in keepObj;

    if (RGBs differ > some tolerance)
        force super-sample shading for
            current & neighbour pixels;
    }
else {
    RGB = use keepObj's RGB information for
        current pixel;
    }
```

4. Parametric Representation of Triangles

Parameterization of triangles is nothing new. However, to the authors' knowledge, it has never been used in a scanline implementation. We will show, in the following subsections, the advantages of using parameterization information for calculating visibility information, depth computation and normal interpolation.

For tessellated surfaces, coherence does not help much. Since it usually requires a great deal of precomputing time per scanline and per triangle, it may not be worth it. We have attempted to achieve a good balance between precomputation and lazily evaluated components. In addition, all our precomputation is done only once per triangle. Nothing extra needs to be computed per scanline.

4.1. Parameterization

Let A, B and C be the screen coordinate vertices of a triangle. Instead of storing the above vertices (or edge information), we will store a parameterized version: D, E, F vectors, plus a few other pieces of information.

These equations set up the needed parameterization. Let

$$d = (B_x - A_x)(C_y - A_y) - (B_y - A_y)(C_x - A_x),$$

$$D = A,$$

$$E_x = (B_x - A_x)/d, \quad E_y = (B_y - A_y)/d, \quad E_z = (B_z - A_z),$$

$$F_x = (C_x - A_x)/d, \quad F_y = (C_y - A_y)/d, \quad F_z = (C_z - A_z).$$

We need to perform this precomputation for each triangle when it becomes active with respect to a scanline.

The parameters s, t can be defined from the above equations for any (x,y) location to be rendered as $s = (x - D_x)F_y - (y - D_y)F_x$, $t = (y - D_y)E_x - (x - D_x)E_y$, where s is the unit distance on the normalized E–D axis and t is the unit distance on the normalized F–D axis. Note that $0 \le s,t \le 1$ and $0 \le s+t \le 1$ must be valid for a visibility hit.

4.2. Fast Z Depth Computations

Since we already compute the spans in the scanline approach, we do not need to check the range of the s,t values for a visibility hit. Thus we can compute the z depth values directly without the s,t values. Three additional values are precomputed: $d_x = F_yE_z - E_yF_z$, $d_y = E_xF_z - F_xE_z$, $d_z = \delta\, d_x$, where δ represents the change in x from one sample to the next.

A simple z depth computation costing 5 floating point calculations is $z = D_z + (x - D_x)\,d_x + (y - D_y)\,d_y$. Applying horizontal scanline coherence, we get $z_{i+1} = z_i + d_z$.

4.3. Calculating Spans

We can also compute triangle spans for the current scanline very quickly using our parameterization. It takes 7-11 floating point evaluations per triangle to compute span information, without assuming any coherence. The bounds of the span will then be scaled and integerized, and inserted into the scanline data structure.

By definition, the boundaries of the triangle are determined by the cases: $s = 0$, $t = 0$ and $s + t = 1$. On each scanline, two of the above three cases must be valid for any active triangle. We test for conditions $s = 0$ and $t = 0$, and if one of these two conditions fails, we can use the $s + t = 1$ case.

For the $s = 0$ case, we know that $0 \le t \le 1$ in order to qualify, where $t = (y - D_y)\,[E_x - (F_x/F_y)\,E_y]$. If this condition holds, then one of the x boundaries of the span is $(y - D_y)\,(F_x/F_y) + D_x$.

For the $t = 0$ case, we know that $0 \le s \le 1$ in order to qualify, where $s = (y - D_y)\,[-F_x - (E_x/E_y)\,F_y]$. If this condition holds, then one of the x boundaries of the span is $(y - D_y)\,(E_x/E_y) + D_x$.

For the $s + t = 1$ case, the x boundary is $(y - D_y)\,[(E_x - F_x)/(F_y - E_y)] + [1/(F_y - E_y) + D_x]$.

Note that most of the above expressions are precomputed, so that the actual span computation is very fast. We could have also kept vertical coherence information as to which two edges are likely to intersect the next scanline, but this feature was omitted due to our memory constraints.

4.4. Normal Interpolation for Shading

After the visibility winner is declared, we need to interpolate vertex normals during shading to achieve the appearance of smooth surfaces [Bui75]. Some papers [Plet89] have described this to be the most expensive phase of the rendering process. However, we can quickly disprove this claim. Actually, for our scanline renderer, the process which tends most to hog the CPU is sorting the *yActiveList* for very complex scenes.

Recall that s,t can be computed as follows: $s = (x - D_x) F_y - (y - D_y) F_x$, $t = (y - D_y) E_x - (x - D_x) E_y$. Then the interpolated vertex normal N is simply $N = (1-s-t) N_D + s N_E + t N_F$. The weights can thus be computed in 8 floating point evaluations, and do not depend on any coherence. This is helpful for the adaptive-stochastic anti-aliasing scheme we have designed.

Furthermore, the weights $1-s-t$, s, t are normalized. Then N is guaranteed to be almost normalized†. Thus, instead of normalizing N using the standard *sqrt* method, we can compute the expression below (Newton's iterative method) with an initial guess $x_0 = 1$, where x will converge to $1 / |N|$ in 1-2 iterations:

$$x_{i+1} = 1.5\, x_i - 0.5\, N \cdot N\, x_i^3$$

Thus $x_1 = 1.5 - 0.5\, N \cdot N$.

While others, such as [Duff79] [Bish86], have employed faster normalizations without the need for *sqrt*, they assumed limitations about the environment such as a fixed illumination model, an orthographic view and directional lights. With our approach, there is no need to assume anything about the environment.

5. Scanline Information Assisting Texture Mapping

In most complicated scenes, there is not only a large polygon count, but plenty of texture mapping to contend with as well. This compounds the memory problem.

We can do a little better, with the help of scanline information. Instead of reading the texture information into memory before rendering any scanlines, this processing is done only when a surface that needs the information is first encountered during a scanline. Therefore no memory is used until necessary. If we are lucky, the surface is totally hidden and no texture loading is required at all.

For each different texture, we also calculate the topmost scanline beyond which the texture will no longer be applied. This can be done in conjunction with the bounding box evaluation to compute *minY* (mentioned in sections 3.1, 3.2). When the last scanline to use the texture is completed, the texture information in memory is freed. Thus, any new texture that is introduced beyond the current scanline can reuse the previous texture's memory. We can sort these texture references in decreasing *minY* order, so that our searches for opportunities to free texture storage should be very fast.

† Note that normalized $N' = N / |N|$, where $|N| = \sqrt{N \cdot N}$.

The rationale for dynamically loading and freeing textures is as follows. Most scenes that have lots of different textures usually have them mapped onto surfaces in small and discrete regions of the scene. Thus there is no need to keep the information around all the time, and we can reuse the memory as new textures are introduced. And even in the worst case when this scenario is not valid, there is no penalty for this optimization: texture processing is just delayed a little until an unloaded texture is encountered.

This approach is analogous to dynamic loading as discussed in [Cook87]. However, they applied it to a Z-buffer approach (actually, to an A-buffer). This means that they render in surface order, while our implementation renders in horizontal scanline order. They can take advantage of sequentially dealing with surfaces that contain the same texture, then free memory for the next set of textures. This is very powerful, but it is unclear how similar sets of multiple textures per surface are handled. It may be quite complicated or inefficient.

6. Testing and Analysis

Our scanline renderer is built on top of the *Alias ray caster*, version 3.0. The original implementation was done on a *Silicon Graphics Personal Iris*. It has since been ported onto the *Mac II*, on which memory conservation is of particular interest to us. Testing was done on a *Mac IIfx* in the *MPW* environment, system 6, with 6 megabytes of memory allocated to the renderer.

A hidden-line removal option has been added to the scanline implementation with minor modifications to the code. This feature is used to reveal the fineness of the tessellations and the speed of the visibility determination (free of shading computations).

6.1. Memory Usage

The value of m is usually much smaller than n, where m is the maximum number of triangles intersecting any scanline and n is the total number of triangles. Thus the memory necessary to store the geometric information tends to be quite small. Table 1 shows some comparisons of n with m from our test images.

Image	Resolution	#Lights	n	m
Rockets	640x480	9	18020	624
Spirals	640x480	3	4320	277
Room	640x480	1	5178	233
Lamp	640x480	3	29057	835

Table 1: maximum number of spans in a scanline vs. number of triangles

6.2. Performance of Aliased vs. Anti-aliased Images

Our tests indicate that the adaptive and stochastic shading calls were fairly effective. In most images, the ratio of shading to geometric super-sampling is small but the results

are still very high quality. The ratio is particularly small for the Room image (Figures 5 & 6), where the highlighting is already anti-aliased. This is also true for the Lamp image (Figures 8 & 9), but a little more shading is needed on the highlights.

For the Spirals image (Figures 3 & 4), this ratio is close to 1. This additional expense is understandable since a lot of light intensity changes occur. For the Rockets image (Figures 1 & 2) a lot more shading calls were necessary because of the large number of highlights. Nonetheless, many calls were avoided. These examples indicate to us that our shading call scheme does properly sample as needed. And this shading quality cannot possibly be achieved using only a single shading call per pixel, or by vertically filtering in a scanline implementation.

Also note that the visibility part of the scanline is very fast in all cases. All timings shown in table 2 are recorded in CPU seconds for the scanline process. The *sampling* field indicates the level of anti-aliasing, with *hidden* meaning hidden line removal at 1×1 sampling rate. *#Shading* represents the total number of shading calls, and *#Geometry* represents the number of times the geometry is sampled.

Image	Sampling	Time	#Shading	#Geometry
Rockets	3x3	524	191473	390342
Rockets	1x1	232	97550	97550
Rockets	hidden	84	0	97550
Spirals	4x4	162	85290	86008
Spirals	1x1	24	5363	5363
Spirals	hidden	19	0	5363
Room	3x3	542	309023	1193977
Room	1x1	300	298587	298587
Room	hidden	88	0	298587
Lamp	3x3	294	120410	438724
Lamp	1x1	160	109481	109481
Lamp	hidden	67	0	109481

Table 2: Execution times of various modes of rendering

6.3. Dynamically Loading and Freeing Textures

In the Texture image (Figure 10), note that four separate textures are applied. However, due to our dynamic loading and freeing scheme (section 5), the renderer only needed to allocate memory for three textures. The potential exists for even greater memory reuse with more complex scenes.

7. Conclusions and Future Work

This paper described the implementation of a scanline-based renderer. We have stressed issues such as memory conservation (sections 3.1, 5) so that tessellated surfaces are effectively dealt with, the benefits of increased efficiency provided by separating geometric visibility from shading computations for anti-aliasing (section 3.4), and the advantages of using triangle parameterization for a scanline approach (section 4). Such considerations lead to an efficient implementation which also provides very high quality images.

Since our scanline implementation is a point sampling approach, it lends itself quite straightforwardly to ray tracing [Whit80]. As in [Wegh84], the scanline can be used to compute the visibility of a cast ray, and then to spawn off reflection and refraction rays for producing ray tracing effects. This is done to avoid expensive ray-surface intersection calculations for cast rays.

However, this has not been implemented on top of our renderer. Anyone who wishes to employ such an approach still needs to address several important problems. The first problem is memory usage: we would need to store the scanline structures, geometric information for world-space triangles, and ray tracing intersection culler structures. No present-day machine can function effectively under such demands. For complex scenes, excessive swapping and paging would easily cripple any system. The second problem is anti-aliasing: we would be forced to spawn off reflection and refraction rays at the same location as the scanline. This would restrict our anti-aliasing mainly to super-sampling, which can be very expensive. In addition, there is no simple way to apply powerful anti-aliasing techniques such as stochastic sampling [Cook86].

8. Acknowledgements

Thanks to Andrew Pearce and Jim Craighead for their help and advice during our implementation of the scanline visibility algorithm. Thanks also to Andrew Pearce, Laurent Ruhlmann, Rob Krieger and Paul Philp for their suggestions on this paper. Brian MacGowan is responsible for the design of the Lamp image.

9. References

[Aho83] A. Aho, J. Hopcroft, J. Ullman, "Data Structures and Algorithms", Addison-Wesley, 1983.

[Bish86] G. Bishop, D. Weimer, "Fast Phong Shading", Computer Graphics, 20(4), August 1986, pp. 103-106.

[Bui75] P. Bui-Tuong, "Illumination for Computer Generated Pictures", Communications of the ACM, 18(6), June 1975, pp. 311-317.

[Cook86] R. Cook, "Stochastic Sampling in Computer Graphics", ACM Transactions on Graphics, 5(1), January 1986, pp. 51-72.

[Cook87] R. Cook, L. Carpenter, E. Catmull, "The Reyes Image Rendering Architecture", Computer Graphics, 21(4), July 1987, pp. 95-102.

[Croc84] G. Crocker, "Invisibility Coherence for Faster Scanline Hidden Surface Algorithms", Computer Graphics, 18(3), July 1984, pp. 95-102.

[Duff79] T. Duff, "Smoothly Shaded Renderings of Polyhedral Objects on Raster Displays", Computer Graphics, 19(2), August 1979, pp. 270-275.

[Fium83] E. Fiume, A. Fournier, L. Rudolph, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer", Computer Graphics, 17(3), July 1983, pp. 141-150.

[Fole84] J. Foley, A. Van Dam, "Fundamentals of Interactive Computer Graphics", Addison-Wesley, 1984, 1st edition.

[Gour71] H. Gouraud, "Computer Display of Curved Surfaces", IEEE Transactions on Computers, c-20(6), June 1971, pp. 623-629.

[Knut73] D. Knuth, "The Art of Computer Programming, Vol. 3: Sorting and Searching", Addison-Wesley, 1973.

[Lee85] M. Lee, R. Redner, S. Uselton, "Statistically Optimized Sampling for Distributed Ray Tracing", Computer Graphics, 19(3), July 1985, pp. 61-67.

[Plet89] D. Pletinckx, "Quaternion Calculus as a Basic Tool in Computer Graphics", The Visual Computer, vol. 5, 1989, pp. 2-13.

[Sedg84] R. Sedgewick, "Algorithms", Addison-Wesley, 1984.

[Watk70] G. Watkins, "A Real-Time Visible Surface Algorithm", Computer Science Dept., University of Utah, UTECH-CSC-70-101, June 1970.

[Wegh84] H. Weghorst, G. Hooper, D. Greenberg, "Improved Computational Methods for Ray Tracing", ACM Transactions on Graphics, 3(1), January 1984, pp. 52-69.

[Whit80] T. Whitted, "An Improved Illumination Model for Shaded Display", Communications of the ACM, 23(6), June 1980, pp. 343-349.

Figure 1. 1×1 Rockets

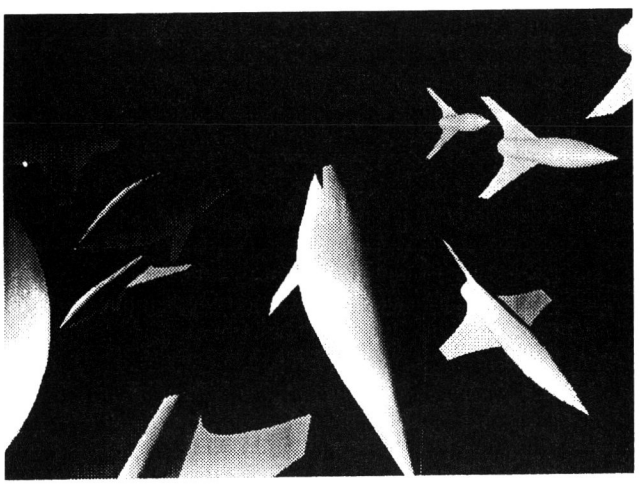

Figure 2. 3×3 Rockets

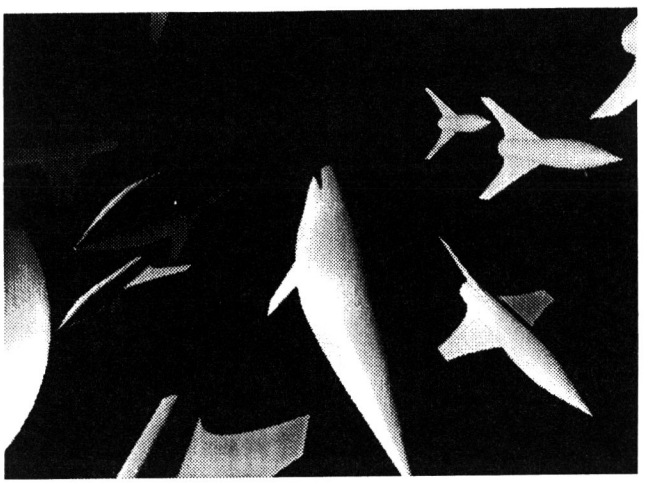

Figure 3. 1×1 Spirals

Figure 4. 4×4 Spirals

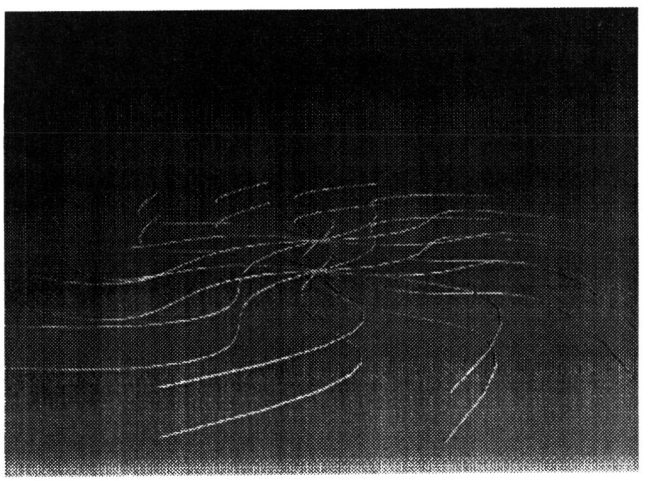

Figure 5. 1×1 Room

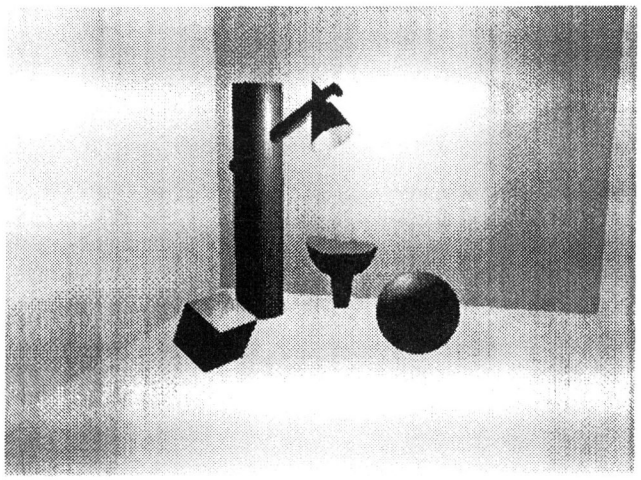

Figure 6. 3×3 Room

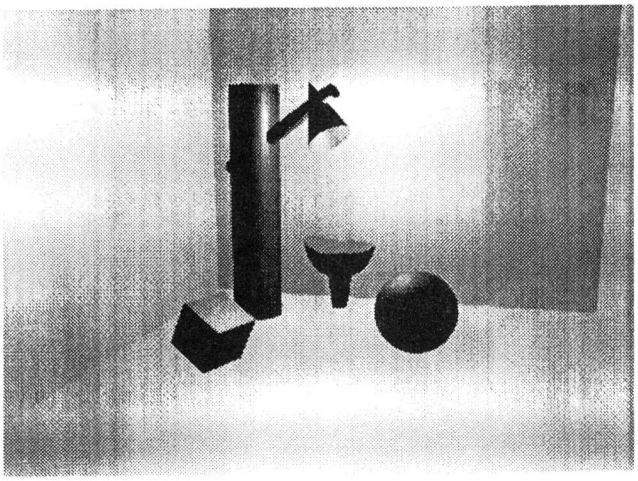

Figure 7. Hidden-line Lamp

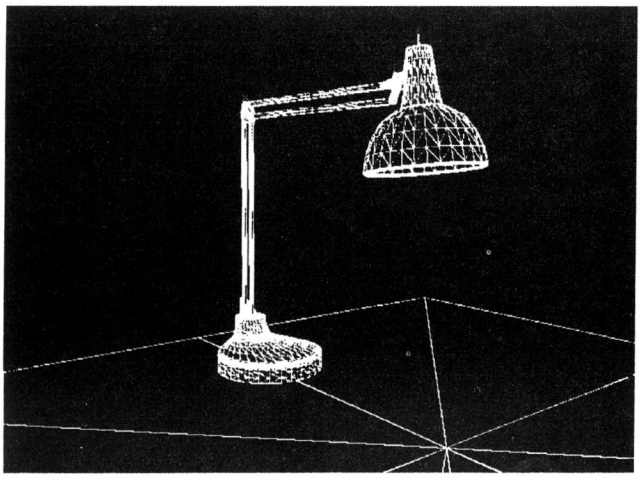

Figure 8. 1×1 Lamp

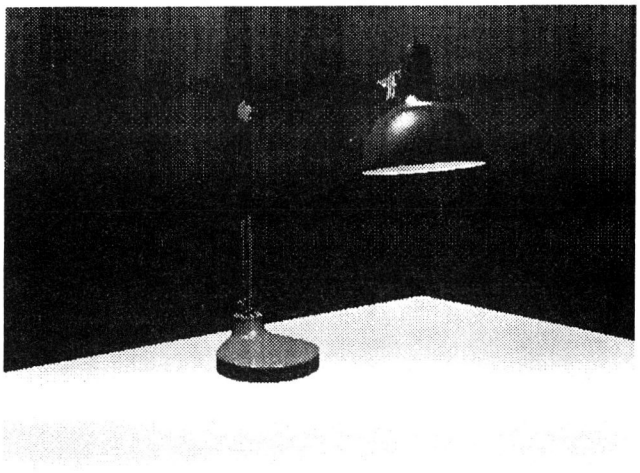

Figure 9. 3×3 Lamp

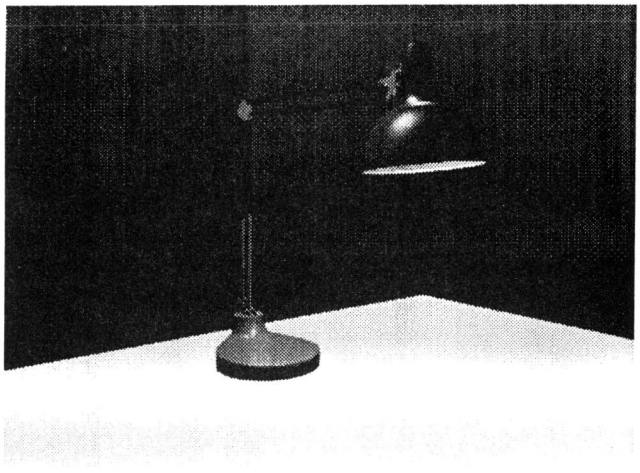

Figure 10. Texture Mapping