The Reyes Image Rendering Architecture

Robert L. Cook
Loren Carpenter
Edwin Catmull

Pixar
P. O. Box 13719
San Rafael, CA 94913

An architecture is presented for fast high-quality rendering of complex images. All objects are reduced to common world-space geometric entities called micropolygons, and all of the shading and visibility calculations operate on these micropolygons. Each type of calculation is performed in a coordinate system that is natural for that type of calculation. Micropolygons are created and textured in the local coordinate system of the object, with the result that texture filtering is simplified and improved. Visibility is calculated in screen space using stochastic point sampling with a z buffer. There are no clipping or inverse perspective calculations. Geometric and texture locality are exploited to minimize paging and to support models that contain arbitrarily many primitives.

CR CATEGORIES AND SUBJECT DESCRIPTORS: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism;

ADDITIONAL KEY WORDS AND PHRASES: image rendering, computer image synthesis, texturing, hidden surface algorithms, z buffer, stochastic sampling

1. Introduction

Reyes is an image rendering system developed at Lucasfilm Ltd. and currently in use at Pixar. In designing Reyes, our goal was an architecture optimized for fast high-quality rendering of complex animated scenes. By fast we mean being able to compute a feature-length film in approximately a year; high-quality means virtually indistinguishable from live action motion picture photography; and complex means as visually rich as real scenes.

This goal was intended to be ambitious enough to force us to completely rethink the entire rendering process. We actively looked for new approaches to image synthesis and consciously tried to avoid limiting ourselves to thinking in terms of traditional solutions or particular computing environments. In the process, we combined some old methods with some new ideas.

Some of the algorithms that were developed for the Reyes architecture have already been discussed elsewhere; these include stochastic sampling [12], distributed ray tracing [10, 13], shade trees [11], and an antialiased depth map shadow algorithm [32].

 ACM-0-89791-227-6/87/007/0095 $00.75

This paper includes short descriptions of these algorithms as necessary, but the emphasis in this paper is on the overall architecture.

Many of our design decisions are based on some specific assumptions about the types of complex scenes that we want to render and what makes those scenes complex. Since this architecture is optimized for these types of scenes, we begin by examining our assumptions and goals.

- **Model complexity.** We are interested in making images that are visually rich, far more complex than any pictures rendered to date. This goal comes from noticing that even the most complex rendered images look simple when compared to real scenes and that most of the complexity in real scenes comes from rich shapes and textures. We expect that reaching this level of richness will require scenes with hundreds of thousands of geometric primitives, each one of which can be complex.
- **Model diversity.** We want to support a large variety of geometric primitives, especially data amplification primitives such as procedural models, fractals [18], graftals [35], and particle systems [30, 31].
- **Shading complexity.** Because surface reflection characteristics are extremely varied and complex, we consider a programmable shader a necessity. Our experience with such a shader [11] is that realistic surfaces frequently require complex shading and a large number of textures. Textures can store many different types of data, including surface color [8], reflections (environment maps) [3], normal perturbation (bump maps) [4], geometry perturbation (displacement maps) [11], shadows [32], and refraction [25].
- **Minimal ray tracing.** Many non-local lighting effects can be approximated with texture maps. Few objects in natural scenes would seem to require ray tracing. Accordingly, we consider it more important to optimize the architecture for complex geometries and large models than for the non-local lighting effects accounted for by ray tracing or radiosity.
- **Speed.** We are interested in making animated images, and animation introduces severe demands on rendering speed. Assuming 24 frames per second, rendering a 2 hour movie in a year would require a rendering speed of about 3 minutes per frame. Achieving this speed is especially challenging for complex images.
- **Image Quality.** We eschew aliasing and faceting artifacts, such as jagged edges, Moiré patterns in textures, temporal strobing, and highlight aliasing.
- **Flexibility.** Many new image rendering techniques will undoubtedly be discovered in the coming years. The architecture should be flexible enough to incorporate many of these new techniques.

2. Design Principles

These assumptions led us to a set of architectural design principles. Some of these principles are illustrated in the overview in Figure 1.

1. **Natural coordinates.** Each calculation should be done in a coordinate system that is natural for that calculation. For example, texturing is most naturally done in the coordinate system of the local surface geometry (e.g., *uv* space for patches), while the visible surface calculations are most naturally done in pixel coordinates (screen space).
2. **Vectorization.** The architecture should be able to exploit vectorization, parallelism and pipelining. Calculations that are similar should be done together. For example, since the shading calculations are usually similar at all points on a surface, an entire surface should be shaded at the same time.
3. **Common representation.** Most of the algorithm should work with a single type of basic geometric object. We turn every geometric primitive into *micropolygons*, which are flat-shaded subpixel-sized quadrilaterals. All of the shading and visibility calculations are performed exclusively on micropolygons.
4. **Locality.** Paging and data thrashing should be minimized.
   a. **Geometric locality.** Calculations for a geometric primitive should be performed without reference to other geometric primitives. Procedural models should be computed only once and should not be kept in their expanded form any longer than necessary.
   b. **Texture locality.** Only the textures currently needed should be in memory, and textures should be read off the disk only once.
5. **Linearity.** The rendering time should grow linearly with the size of the model.
6. **Large models.** There should be no limit to the number of geometric primitives in a model.
7. **Back door.** There should be a back door in the architecture so that other programs can be used to render some of the objects. This give us a very general way to incorporate any new technique (though not necessarily efficiently).
8. **Texture maps.** Texture map access should be efficient, as we expect to use several textures on every surface. Textures are a powerful tool for defining complex shading characteristics, and displacement maps [11] can be used for model complexity.

We now discuss some of these principles in detail.

2.1. Geometric Locality.

When ray tracing arbitrary surfaces that reflect or refract, a ray in any pixel on the screen might generate a secondary ray to any object in the model. The object hit by the secondary ray can be determined quickly [20, 21, 34], but that object must then be accessed from the database. As models become more complex, the ability to access any part of the model at any time becomes more expensive; model and texture paging can dominate the rendering time. For this reason, we consider ray tracing algorithms poorly suited for rendering extremely complex environments.

In many instances, though, texture maps can be used to approximate non-local calculations. A common example of this is the use of environment maps [3] for reflection, a good approximation in many cases. Textures have also been used for refractions [25] and shadows [32, 36]. Each of these uses of texture maps represents some non-local calculations that we can avoid (principles 4a and 8).

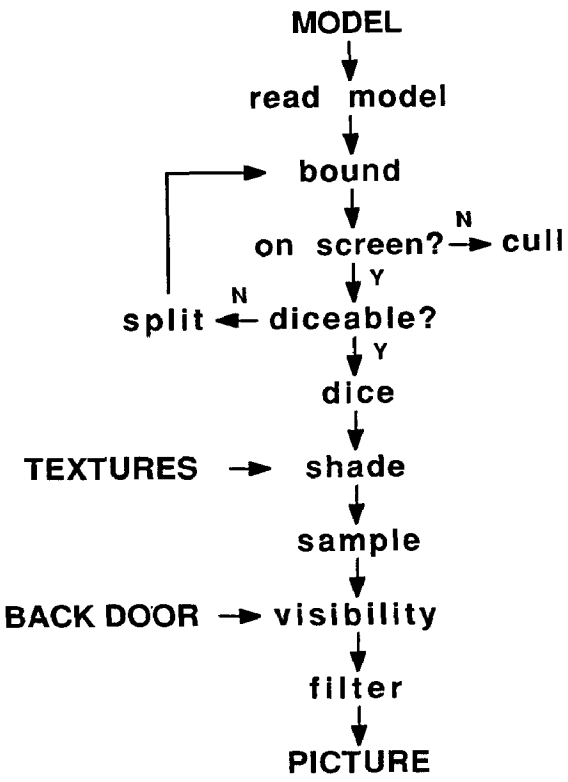

Figure 1. Overview of the algorithm.

2.2. Point sampling.

Point sampling algorithms have many advantages; they are simple, powerful, and work easily with many different types of primitives. But unfortunately, they have been plagued by aliasing artifacts that would make them incompatible with our image quality requirements. Our solution to this problem is a Monte Carlo method called *stochastic sampling*, which is described in detail elsewhere [12]. With stochastic sampling, aliasing is replaced with noise, a less objectionable artifact.

We use a type of stochastic sampling called *jittering* [12]. Pixels are divided into a number of subpixels (typically 16). Each subpixel has exactly one sample point, and the exact location of that sample point within the subpixel is determined by jittering, or adding a random displacement to the location of the center of the subpixel. This jittered location is used to sample micropolygons that overlap the subpixel. The current visibility information for each sample point on the screen is kept in a *z* buffer [8].

The *z* buffer is important for two reasons. First, it permits objects to be sent through the rest of the system one at a time (principles 2, 4, 5 and 6). Second, it provides a back door (principle 7); the *z* buffer can combine point samples from this algorithm with point samples from other algorithms that have capabilities such as ray tracing and radiosity. This is a form of 3-D compositing; it differs from Duff's method [15] in that the compositing is done before filtering the visible samples.

Glossary

CAT	a coherent access texture, in which s is a linear function of u and t is a linear function of v.
CSG	constructive solid geometry. Defines objects as the union, intersection, or difference of other objects.
depth complexity	the average number of surfaces (visible or not) at each sample point
dicing	the process of turning geometric primitives into grids of micropolygons.
displacement maps	texture maps used to change the location of points in a grid.
ε plane	a plane parallel to the hither plane that is slightly in front of the eye. The perspective calculation may be unreliable for points not beyond this plane.
eye space	the world space coordinate system rotated and translated so that the eye is at the origin looking down the $+z$ axis. $+x$ is to the right, $+y$ is down.
grid	a two-dimensional array of micropolygons.
geometric locality	the principle that all of the calculations for a geometric primitive should be performed without reference to other geometric primitives.
hither plane	the z=min plane that is the front of the viewing frustum.
jitter	the random perturbation of regularly spaced points for stochastic sampling
micropolygon	the basic geometric object for most of the algorithm, a flat-shaded quadrilateral with an area of about ¼ pixel.
RAT	a random access texture. Any texture that is not a CAT.
s and t	parameters used to index a texture map.
screen space	the perspective space in which the x and y values correspond to pixel locations.
shade tree	a method for describing shading calculations [11].
splitting	the process of turning a geometric primitive into one or more new geometric primitives.
stochastic sampling	a Monte Carlo point-sampling method used for antialiasing [12].
texture locality	the principle that each texture should be read off the disk only once.
u and v	coordinates of a parametric representation of a surface.
world space	the global right-handed nonperspective coordinate system.
yon plane	the z=max plane that is the back of the viewing frustum.

2.3. Micropolygons.

Micropolygons are the common basic geometric unit of the algorithm (principle 3). They are flat-shaded quadrilaterals that are approximately ½ pixel on a side. Since half a pixel is the Nyquist limit for an image [6, 26], surface shading can be adequately represented with a single color per micropolygon.

Turning a geometric primitive into micropolygons is called *dicing*. Every primitive is diced along boundaries that are in the natural coordinate system of the primitive (principle 1). For example, in the case of patches, micropolygon boundaries are parallel to u and v. The result of dicing is a two-dimensional array of micropolygons called a *grid* (principle 2). Micropolygons require less storage in grid form because vertices shared by adjacent micropolygons are represented only once.

Dicing is done in eye space, with no knowledge of screen space except for an estimate of the primitive's size on the screen. This estimate is used to determine how finely to dice, i.e., how many micropolygons to create. Primitives are diced so that micropolygons are approximately half a pixel on a side in screen space. This adaptive approach is similar to the Lane-Carpenter patch algorithm [22].

The details of dicing depend on the type of primitive. For the example of bicubic patches, screen-space parametric derivatives can be used to determine how finely to dice, and forward differencing techniques can be used for the actual dicing.

All of the micropolygons in a grid are shaded together. Because this shading occurs before the visible surface calculation, at a minimum every piece of every forward-facing on-screen object must be shaded. Thus many shading calculations are performed that are never used. The extra work we do is related to the *depth complexity* of the scene, which is the average number of surfaces at each sample point. We expect pathological cases to be unusual, however, because of the effort required to model a scene. Computer graphics models are like movie sets in that usually only the parts that will be seen are actually built.

There are advantages that offset the cost of this extra shading; the tradeoff depends on the particular scene being rendered. These are some of the advantages to using micropolygons and to shading them before determining visibility:

- **Vectorizable shading.** If an entire surface is shaded at once, and the shading calculations for each point on the surface are similar, the shading operations can be vectorized (principle 2).
- **Texture locality.** Texture requests can be made for large, contiguous blocks of texture that are accessed sequentially. Because shading can be done in object order, the texture map thrashing that occurs in many other algorithms is avoided (principle 4b). This thrashing occurs when texture requests come in small pieces and alternate between several different texture maps. For extremely complex models with lots of textures, this can quickly make a renderer unusable.
- **Texture filtering.** Many of the texture requests are for rectilinear regions of the texture map (principle 1). This is discussed in detail in the next section.
- **Subdivision coherence.** Since an entire surface can be subdivided at once, we can take advantage of efficient techniques such as forward differencing for patch subdivision (principles 1 and 2).
- **Clipping.** Objects never need to be clipped along pixel boundaries, as required by some algorithms.
- **Displacement maps** [11]. Displacement maps are like bump maps [4] except that the location of a surface can be changed as well as its normal, making texture maps a means of modeling surfaces or storing the results of modeling programs. Because displacement maps can change the surface location, they must be computed before the hidden surface calculation. We have no experience with the effects of large displacements on dicing.
- **No perspective.** Because micropolygons are small, there is no need to correct for the perspective distortion of interpolation [24]. Because shading occurs before the perspective transformation, no inverse perspective transformations are required.

2.4. Texture Locality.

For rich, complex images, textures are an important source of information for shading calculations [3, 8]. Textures are usually indexed using two parameters called u and v. Because u and v are also used for patch parameters, we will call the texture parameters s and t to avoid confusion. Surfaces other than patches may also have a natural coordinate system; we will use u and v for those surface coordinates too.

For many textures, s and t depend only on the u and v of the patch and can be determined without knowing the details of the shading calculations. Other textures are accessed with an s and t that are determined by some more complex calculation. For example, the s and t for an environment map depend on the normal to the surface (though that normal might in turn depend on a bump map that is indexed by u and v).

We accordingly divide textures into two classes: *coherent access textures* (CATs) and *random access textures* (RATs). CATs are textures for which $s=\mathbf{a}u+\mathbf{b}$ and $t=\mathbf{c}v+\mathbf{d}$, where **a**, **b**, **c**, and **d** are constants. All other textures are RATs. Many CATs have $s=u$ and $t=v$, but we have generalized this relationship to allow for single textures that stretch over more than one patch or repeat multiple times over one patch.

We make this distinction because CATs can be handled much more easily and often significantly faster than RATs. Because st order is the same as uv order for CATs, we can access the texture map sequentially if we do our shading calculations in uv order (principles 1 and 4b). Furthermore, if micropolygons are created so that their vertices have s and t values that are integer multiples of powers of ½, and if the textures are prefiltered and prescaled and stored as resolution pyramids [36], then no filtering calculations are required at run time, since the pixels in the texture line up exactly with the micropolygons in the grid (principle 1). Figure 2 shows a primitive diced into a 4x4 grid and the corresponding texture map; notice how the marked micropolygon corresponds exactly to the marked texture region because we are dicing along u and v, the texture's natural coordinate system.

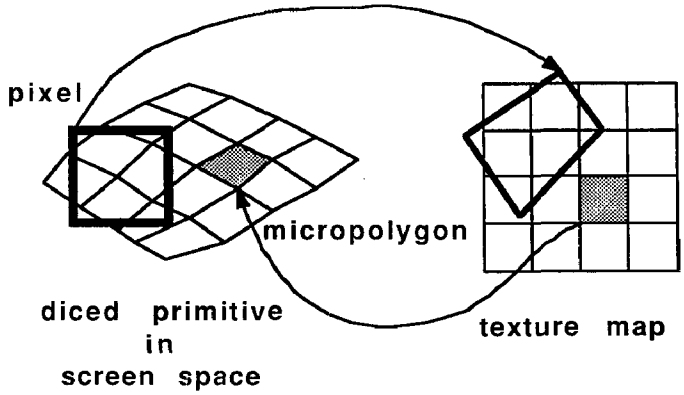

Figure 2. With CATs, micropolygons map exactly to texture map pixels. With the inverse pixel method, pixels map to quadrilateral areas of texture that require filtering.

By contrast, in the more traditional pixel texture access, the pixel boundary is mapped to texture space, where filtering is required. Filtering without a resolution pyramid gives good results but can be expensive [17]. Using a resolution pyramid requires interpolating between two levels of the pyramid, and the filtering is poor [19]. Summed area tables [14] give somewhat better filtering but can have paging problems.

RATs are more general than CATs, but RAT access is slower. RATs can significantly reduce the need for ray tracing. For example, reflections and refractions can frequently be textured onto a surface with environment maps. Environment maps are RATs because they are indexed according to the reflection direction. Another example of a RAT is a decal [2], which is a world-space parallel projection of a texture onto a surface, so that s and t depend on x, y and z instead of on u and v.

```
Initialize the z buffer.
For each geometric primitive in the model,
    Read the primitive from the model file
    If the primitive can be bounded,
        Bound the primitive in eye space.
        If the primitive is completely outside of the hither-yon z range, cull it.
        If the primitive spans the ε plane and can be split,
            Mark the primitive undiceable.
        Else
            Convert the bounds to screen space.
            If the bounds are completely outside the viewing frustum, cull the primitive.
    If the primitive can be diced,
        Dice the primitive into a grid of micropolygons.
        Compute normals and tangent vectors for the micropolygons in the grid.
        Shade the micropolygons in the grid.
        Break the grid into micropolygons.
        For each micropolygon,
            Bound the micropolygon in eye space.
            If the micropolygon is outside the hither-yon range, cull it.
            Convert the micropolygon to screen space.
            Bound the micropolygon in screen space.
            For each sample point inside the screen space bound,
                If the sample point is inside the micropolygon,
                    Calculate the z of the micropolygon at the sample point by interpolation.
                    If the z at the sample point is less than the z in the buffer,
                        Replace the sample in the buffer with this sample.
    Else
        Split the primitive into other geometric primitives.
        Put the new primitives at the head of the unread portion of the model file.
Filter the visible sample hits to produce pixels.
Output the pixels.
```

Figure 3. Summary of the algorithm.

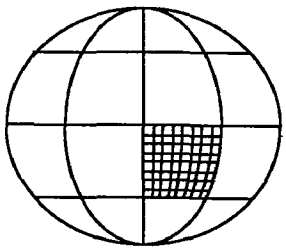

Figure 4a. A sphere is split into patches, and one of the patches is diced into a 8×8 grid of micropolygons.

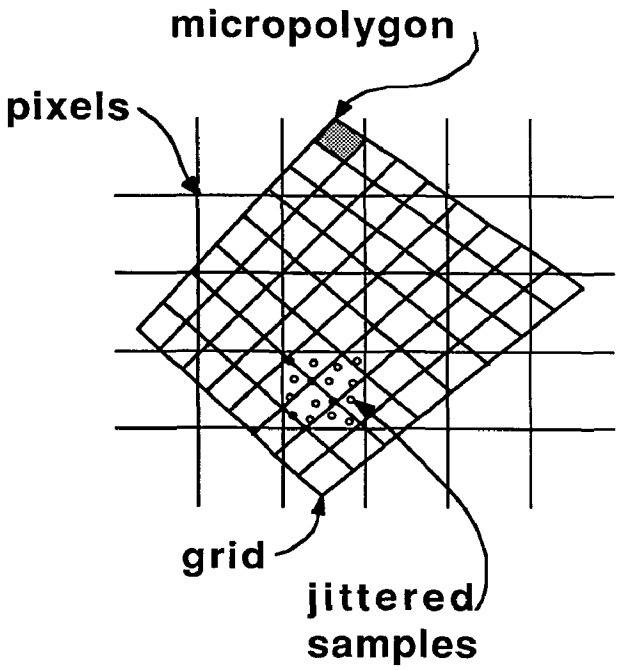

Figure 4b. The micropolygons in the grid are transformed to screen space, where they are stochastically sampled.

3. Description of the Algorithm

The algorithm is summarized in Figure 3. In order to emphasize the basic structure, this description does not include transparency, constructive solid geometry, motion blur, or depth of field. These topics are discussed later.

Each object is turned into micropolygons as it is read in. These micropolygons are shaded, sampled, and compared against the values currently in the z buffer. Since only one object is processed at a time, the amount of data needed at any one time is limited and the model can contain arbitrarily many objects.

Primitives are subdivided only in uv space, never in screen space. The first part of the algorithm is done in uv space and world space, and the second half is done in screen space. After the transformation to screen space, there is never any need to go back to world space or uv space, so there are no inverse transformations.

Each type of geometric primitive has the following routines:

- **Bound.** The primitive computes its eye-space bound; its screen-space bound is computed from the eye-space bound. A primitive must be guaranteed to lie inside its bound, and any primitives it is split into must have bounds that also lie inside its bound. The bound does not have to be tight, however. For example, a fractal surface can be bounded if the maximum value of its random number table is known [7, 18]. The fractal will be guaranteed to lie within this bound, but the bound probably will not be very tight. The effect of displacement maps must be considered in the calculation of the bound.
- **Dice.** Not all types of primitives need to be diceable. The only requirement is that each primitive be able to split itself into other primitives, and that this splitting eventually leads to primitives that can all be diced.
- **Split.** A primitive may split itself into one or more primitives of the same type or of different types.
- **Diceable test.** This test determines whether the primitive should be diced or split and returns "diceable" or "not diceable" accordingly. Primitives should be considered not diceable if dicing them would produce a grid with too many micropolygons or a large range of micropolygon sizes.

The bound, split, and dice routines are optional. If the diceable routine ever returns "diceable", the dice routine must exist; if the diceable routine ever returns "not diceable", the split routine must exist. If the bound routine exists, it is used for culling and for determining how finely a primitive should be diced in order to produce micropolygons of the correct size on the screen.

For example, consider one possible set of routines for a sphere. The sphere diceable routine returns "diceable" for small spheres and "not diceable" for large spheres. The sphere dice routine turns a sphere directly into micropolygons. The sphere split routine turns the sphere into 32 patches [16]. The patch dice routine creates a rectangular grid of micropolygons so that the vertices differ in u and v by integer multiples of powers of ½. This is done to obviate CAT filtering, but in this case it is also necessary for the prevention of patch cracks [9]. Figure 4a shows a sphere being split into patches and one of those patches being diced into an 8x8 grid of micropolygons. Figure 4b shows this grid in screen space with jittered sample locations in one of the pixels.

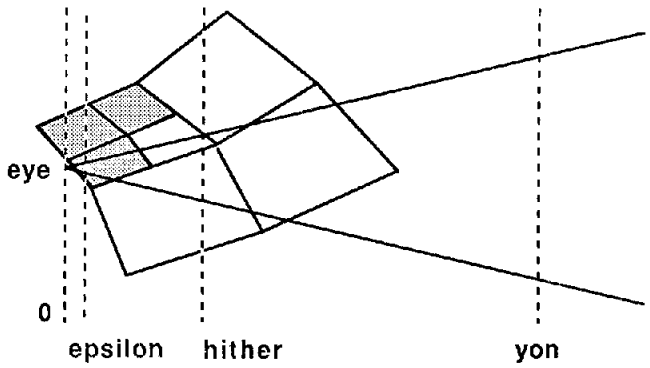

Figure 5. A geometric primitive that spans the ε and hither planes is split until its pieces can be culled or processed. The culled pieces are marked.

This algorithm does not require clipping. The viewing frustum consists of a screen space xy range and an eye space hither-yon z range. Objects that are known to be completely outside of this region are culled. Objects that are partly inside the frustum and partly outside are kept, shaded and sampled. Regions of these objects that are outside of the viewing frustum in the x or y directions are never sampled. Regions that are in front of or behind the viewing frustum may be sampled, but their hits are rejected if the sampled surface point lies outside the hither-yon z range. Note that if the filter that is used to sample the z buffer to produce pixels is wider than a pixel, the viewing frustum must be expanded accordingly because objects that are just off screen can affect pixels on the screen.

Sometimes an object extends from behind the eye to inside the viewing frustum, so that part of the object has an invalid perspective calculation and another part is visible. This situation is traditionally handled by clipping to the hither plane. To avoid clipping, we introduce the ε *plane*, a plane of constant z that lies slightly in front of the eye as shown in Figure 5. Points on the $z<\varepsilon$ side of this plane can have an invalid perspective calculation or an unmanageably large screen space x and y because of the perspective divide. If a primitive spans both the ε plane and the hither plane, it is considered "not diceable" and is split. The resulting pieces are culled if they are entirely outside of the viewing frustum, diced if they lie completely on the $z>\varepsilon$ side of the ε plane, and split again if they span both the ε plane and the hither plane. As long as every primitive can be split, and the splits eventually result in primitives with smaller bounds, then this procedure is guaranteed to terminate successfully. This split-until-cullable procedure obviates clipping. Objects that cannot be bounded can still be protected against bad perspective situations, since micropolygons are created in eye space. Their micropolygons can be culled or be run through a split-until-cullable procedure.

4. Extensions

Since this algorithm was first developed, we have found it easy to add a number of features that were not specifically considered in the original design. These features include motion blur, depth of field, CSG (constructive solid geometry) [1, 33], shadows [32] and a variety of new types of models. The main modification for transparency and CSG calculations is that each sample location in the z buffer stores multiple hits. The hits at each sample point are sorted in z for the transparency and CSG calculations. Motion blur and depth of field are discussed elsewhere in detail [10, 12, 13]. In the case of motion blur, micropolygons are moved for each sample point to a jittered time associated with that sample. For depth of field, they are moved in x and y according to a jittered lens location. Both motion blur and depth of field affect the bound calculations; the details are described elsewhere [13].

5. Implementation

We had to make some compromises to implement this algorithm on a general purpose computer, since the algorithm as described so far can require a considerable amount of z buffer memory. The screen is divided into rectangular buckets, which may be kept in memory or on disk. In an initial pass, each primitive is bounded and put into the bucket corresponding to the upper left corner of its screen space bound. For the rest of the calculations, the buckets are processed in order, left-to-right and top-to-bottom. First all of the primitives in the bucket are either split or diced; as primitives are diced, their micropolygons are shaded and put into every bucket they overlap. After all of the primitives in a bucket have been split or diced, the micropolygons in that bucket are sampled. Once a bucket is empty, it remains empty, so we only need enough z buffer memory for one bucket. The number of micropolygons in memory at any one time can be kept manageable by setting a maximum grid size and forcing primitives to be considered "not diceable" if dicing them would produce too large a grid.

We have implemented this revised version of the algorithm in C and have used it to make a number of animated films, including *The Adventures of André and Wally B.* [27], the stained glass man sequence in *Young Sherlock Holmes* [25], *Luxo Jr.* [28], and *Red's Dream* [29]. The implementation performs reasonably well, considering that the algorithm was designed as a testbed, without any requirement that it would run efficiently in C. For a given shading complexity, the rendering time is proportional to the number of micropolygons (and thus to screen size and to the number of objects).

An example of a image rendered with this program is shown in Figure 6. It is motion blurred, with environment maps for the reflections and shadow depth maps for the shadows [32]. The picture is by John Lasseter and Eben Osthy. It was rendered at 1024x614 pixels, contains 6.8 million micropolygons, has 4 light sources, uses 15 channels of texture, and took about 8 hours of CPU time to compute. Frames in *André* were 512x488 pixels and took less than ½ hour per frame. *Sherlock* frames were 1024x614 and took an hour per frame; *Luxo* frames were 724x434 and took 1½ hours per frame. Statistics on *Red's Dream* frames are not available yet. All of these CPU times are for a CCI 6/32, which is 4-6 times faster than a VAX 11/780.

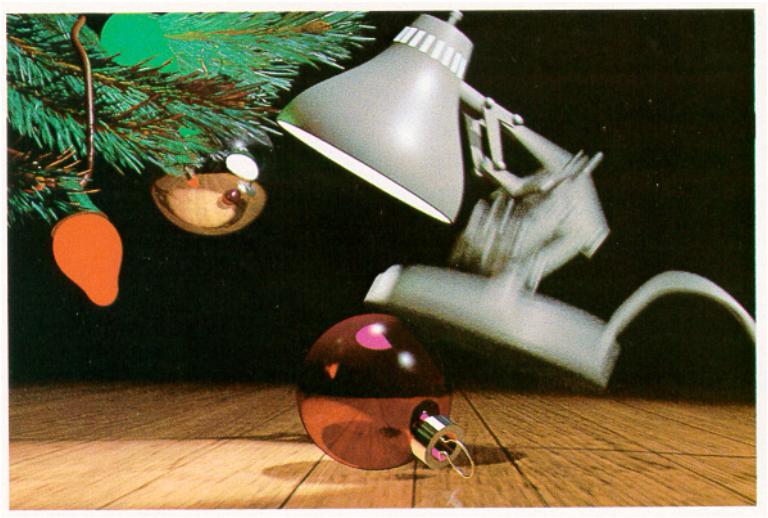

Figure 6. 1986 Pixar Christmas Card by John Lasseter and Eben Ostby.

6. Discussion

This approach has certain disadvantages. Because shading occurs before sampling, the shading cannot be calculated for the specific time of each sample and thus cannot be motion blurred correctly. Shading after sampling would have advantages if the coherency features could be retained; this is an area of future research. Although any primitive that can be scan-converted can be turned into micropolygons, this process is more difficult for some primitives, such as blobs [5]. The bucket-sort version requires bounds on the primitives to perform well, and some primitives such as particle systems are difficult to bound. No attempt is made to take advantage of coherence for large simply-shaded surfaces; every object is turned into micropolygons. Polygons in general do not have a natural coordinate system for dicing. This is fine in our case, because bicubic patches are our most common primitive, and we hardly ever use polygons.

On the other hand, our approach also has a number of advantages. Much of the calculation in traditional approaches goes away completely. There are no inversion calculations, such as projecting pixel corners onto a patch to find normals and texture values. There are no clipping calculations. Many of the calculations can be vectorized, such as the the shading and surface normal calculations. Texture thrashing is avoided, and in many instances textures require no run time filtering. Most of the calculations are done on a simple common representation (micropolygons).

This architecture is designed for rendering exceedingly complex models, and the disadvantages and advantages listed above reflect the tradeoffs made with this goal in mind.

Acknowledgements

Thanks to Bill Reeves, Eben Ostby, David Salesin, and Sam Leffler, all of whom contributed to the C implementation of this algorithm. Conversations with Mark Leather, Adam Levinthal, Jeff Mock, and Lane Molpus were very productive. Charlie Gunn implemented a version of this algorithm in *chas*, the assembly language for the Pixar Chap SIMD processor [23]. Paul Heckbert helped analyze the texture quality, and Ricki Blau studied the performance of the C implementation.

References

1. ATHERTON, PETER R., "A Scanline Hidden Surface Removal Procedure for Constructive Solid Geometry," *Computer Graphics (SIGGRAPH '83 Proceedings)* **17**(3), pp. 73-82 (July 1983).
2. BARR, ALAN H., "Decal Projections," in *SIGGRAPH '84 Developments in Ray Tracing course notes* (July 1984).
3. BLINN, JAMES F. AND MARTIN E. NEWELL, "Texture and Reflection in Computer Generated Images," *Communications of the ACM* **19**(10), pp. 542-547 (October 1976).
4. BLINN, JAMES F., "Simulation of Wrinkled Surfaces," *Computer Graphics (SIGGRAPH '78 Proceedings)* **12**(3), pp. 286-292 (August 1978).
5. BLINN, JAMES F., "A Generalization of Algebraic Surface Drawing," *ACM Transactions on Graphics* **1**(3), pp. 235-256 (July 1982).
6. BRACEWELL, RONALD N., *The Fourier Transform and Its Applications*, McGraw-Hill, New York (1978).
7. CARPENTER, LOREN, "Computer Rendering of Fractal Curves and Surfaces," *Computer Graphics (SIGGRAPH '80 Proceedings)* **14**(3), pp. 9-15, Special Issue (July 1980).
8. CATMULL, EDWIN E., "A Subdivision Algorithm for Computer Display of Curved Surfaces," Phd dissertation, University of Utah, Salt Lake City (December 1974).
9. CLARK, JAMES H., "A Fast Algorithm for Rendering Parametric Surfaces," *Computer Graphics (SIGGRAPH '79 Proceedings)* **13**(2), pp. 7-12, Special Issue (August 1979).
10. COOK, ROBERT L., THOMAS PORTER, AND LOREN CARPENTER, "Distributed Ray Tracing," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3), pp. 137-145 (July 1984).
11. COOK, ROBERT L., "Shade Trees," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3), pp. 223-231 (July 1984).
12. COOK, ROBERT L., "Stochastic Sampling in Computer Graphics," *ACM Transactions on Graphics* **5**(1), pp. 51-72 (January 1986).

13. COOK, ROBERT L., "Practical Aspects of Distributed Ray Tracing," in *SIGGRAPH '86 Developments in Ray Tracing course notes* (August 1986).
14. CROW, FRANKLIN C., "Summed-Area Tables for Texture Mapping," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3), pp. 207-212 (July 1984).
15. DUFF, TOM, "Compositing 3-D Rendered Images," *Computer Graphics (SIGGRAPH '85 Proceedings)* **19**(3), pp. 41-44 (July 1985).
16. FAUX, I. D. AND M. J. PRATT, *Computational Geometry for Design and Manufacture*, Ellis Horwood Ltd., Chichester, England (1979).
17. FEIBUSH, ELIOT, MARC LEVOY, AND ROBERT L. COOK, "Synthetic Texturing Using Digital Filtering," *Computer Graphics* **14**(3), pp. 294-301 (July 1980).
18. FOURNIER, ALAIN, DON FUSSELL, AND LOREN CARPENTER, "Computer Rendering of Stochastic Models," *Communications of the ACM* **25**(6), pp. 371-384 (June 1982).
19. HECKBERT, PAUL S., "Survey of Texture Mapping," *IEEE Computer Graphics and Applications* (November 1986).
20. KAPLAN, MICHAEL R., "Space-Tracing, A Constant Time Ray-Tracer," in *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes* (July 1985).
21. KAY, TIMOTHY L. AND JAMES T. KAJIYA, "Ray Tracing Complex Scenes," *Computer Graphics (SIGGRAPH '86 Proceedings)* **20**(4), pp. 269-278 (Aug. 1986).
22. LANE, JEFFREY M., LOREN C. CARPENTER, TURNER WHITTED, AND JAMES F. BLINN, "Scan Line Methods for Displaying Parametrically Defined Surfaces," *Communications of the ACM* **23**(1), pp. 23-34 (January 1980).
23. LEVINTHAL, ADAM AND THOMAS PORTER, "Chap - A SIMD Graphics Processor," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3), pp. 77-82 (July 1984).
24. NEWMAN, WILLIAM M. AND ROBERT F. SPROULL, *Principles of Interactive Computer Graphics (2nd ed.)*, McGraw-Hill, New York (1979). pp. 361-363
25. PARAMOUNT PICTURES CORPORATION, *Young Sherlock Holmes*, Stained glass man sequence by Pixar and Lucasfilm Ltd. 1985.
26. PEARSON, D. E., *Transmission and Display of Pictorial Information*, Pentech Press, London (1975).
27. PIXAR, *The Adventures of André and Wally B.*, July 1984.
28. PIXAR, *Luxo Jr.*, July 1986.
29. PIXAR, *Red's Dream*, July 1987.
30. REEVES, WILLIAM T., "Particle Systems – A Technique for Modeling a Class of Fuzzy Objects," *ACM Transactions on Graphics* **2**(2), pp. 91-108 (April 1983).
31. REEVES, WILLIAM T. AND RICKI BLAU, "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems," *Computer Graphics (SIGGRAPH '85 Proceedings)* **19**(3), pp. 313-322 (July 1985).
32. REEVES, WILLIAM T., DAVID H. SALESIN, AND ROBERT L. COOK, "Shadowing with Texture Maps," *Computer Graphics (SIGGRAPH '87 Proceedings)* **21** (July 1987).
33. ROTH, S. D., "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*(18), pp. 109-144 (1982).
34. RUBIN, STEVEN M. AND TURNER WHITTED, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics (SIGGRAPH '80 Proceedings)* **14**(3), pp. 110-116 (July 1980).
35. SMITH, ALVY RAY, "Plants, Fractals, and Formal Languages," *Computer Graphics (SIGGRAPH '84 Proceedings)* **18**(3), pp. 1-10 (July 1984).
36. WILLIAMS, LANCE, "Pyramidal Parametrics," *Computer Graphics (SIGGRAPH '83 Proceedings)* **17**(3), pp. 1-11 (July 1983).