# 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).

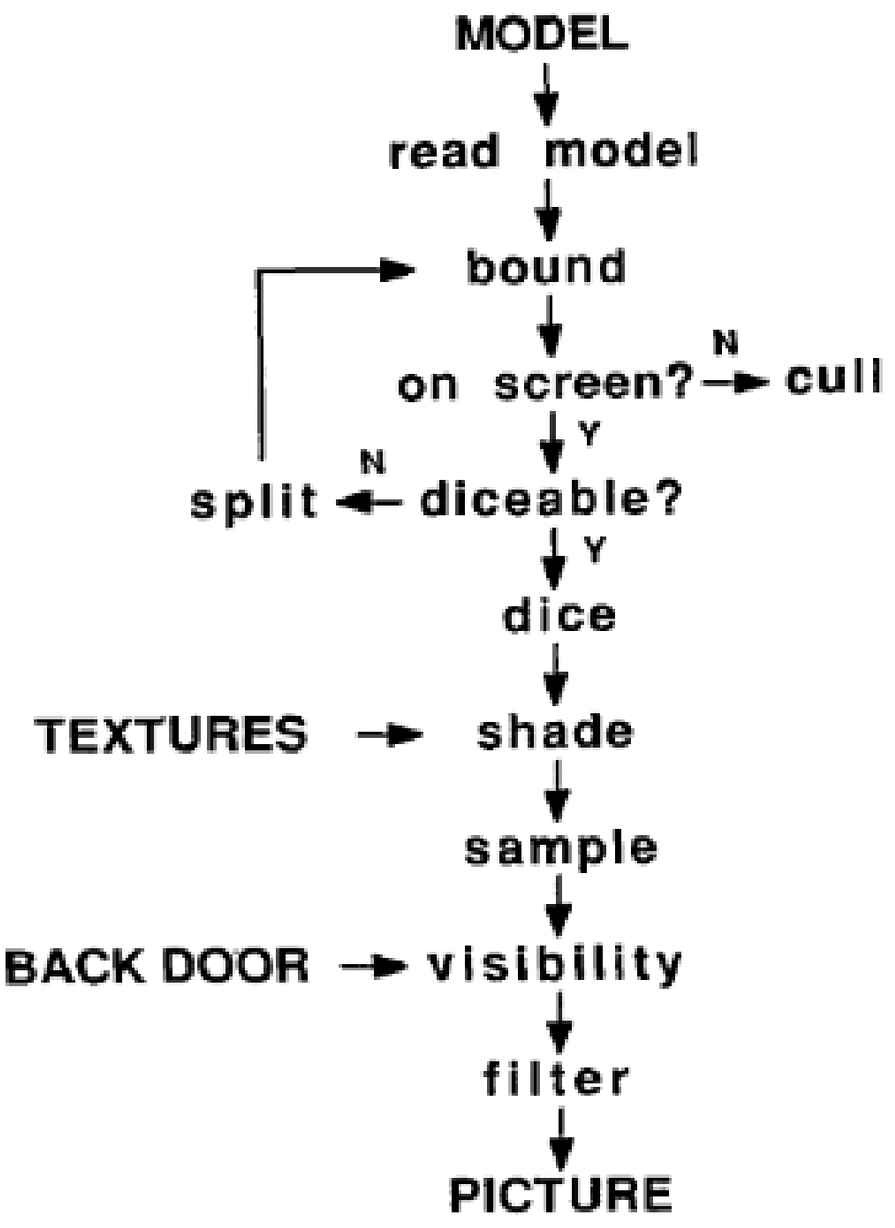


*Figure 1. Overview of the algorithm.*

### 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.