How PhotoRealistic RenderMan Works

. . . and What You Can Do about It

6.1 History

Pixar's *PhotoRealistic RenderMan* renderer is an implementation of a scanline rendering algorithm known as the *REYES* architecture. REYES was developed in the mid-1980s by the Computer Graphics Research Group at Lucasfilm (now Pixar) with the specific goal of creating a rendering algorithm that would be applicable to creating special effects for motion pictures. The algorithm was first described by Cook et al. in their 1987 Siggraph paper "The REYES Image Rendering Architecture." They developed this novel rendering algorithm because they felt that the other algorithms generally in use at the time (polygon *z*-buffer algorithms, polygon scanline algorithms, and ray tracers) had various flaws and constraints that really limited their use in this venue.

Images used in motion picture special effects need to be photorealistic — that is, appear of such high quality that the audience would believe they were filmed with a real camera. All of the image artifacts that computer graphics researchers had to live with up to this point were unacceptable if the images were going to fool the critical eyes of the movie-going masses. In particular, REYES was designed to overcome the following problems with existing rendering algorithms:

- **Vast visual complexity**: Photographs of the real world contain millions of objects, and every object has minute details that make it look real. CG images must contain the same complexity if they are to blend seamlessly with live camera work.
- **Motion blur**: Photographs of moving objects naturally exhibit a blur due to the camera shutter being open for a period of time while the object moves through the field of view. Decades of stop-motion special effects failed to take this into account, and the audience noticed.
- **Speed and memory limitations**: Motion pictures contain over 100,000 frames and are filmed in a matter of days or weeks. Fast computers are expensive, and there is never enough memory (particularly in retrospect). Implementability on special-purpose hardware was a clear necessity.

The resulting design brought together existing work on curved surface primitives and scanline algorithms with revolutionary new work in flexible shading and stochastic antialiasing to create a renderer that could produce images that truly looked like photographs. It was first used in a film titled *Young Sherlock Holmes* in 1985, drew rave reviews for its use in *The Abyss* in 1989, and in 1993 was given an Academy Award for contributions to the film industry.

6.2 Basic Geometric Pipeline

The REYES algorithm is a geometric pipeline, not entirely unlike those found in modern-day hardware graphics engines. What sets it apart is the specific types of geometric operations that occur in the pipeline, and the way that, as the data streams through the system, it gains and retains enough geometric and appearance fidelity that the final result will have very high image quality. Figure 6.1 shows the basic block diagram of the architecture.

The first step, of course, is loading the scene description from the modeler. Typically, the scene description is in a RIB file, loaded from disk. In that case, the RIB file is read by a RIB parser, which calls the appropriate RI routine for each line of the RIB file. Notice that since the RIB file is a simple metafile of the RI API, it is extremely easy to parse. The only minor complexity arises from the handling of parameter list data, which is dependent on the parameter type declarations that appear earlier in the RIB file. Alternatively, a program that is linked to the renderer can call the RI API directly, in which case the parser is simply bypassed.

The second step is the processing of the RenderMan Interface calls themselves. This stage of the pipeline maintains the hierarchical graphics state machine. RI calls fall into two classes: attributes or options that manipulate the graphics state machine, and geometric primitives whose attributes are defined by the then-current version of the graphics state machine. The hierarchical graphics state machine

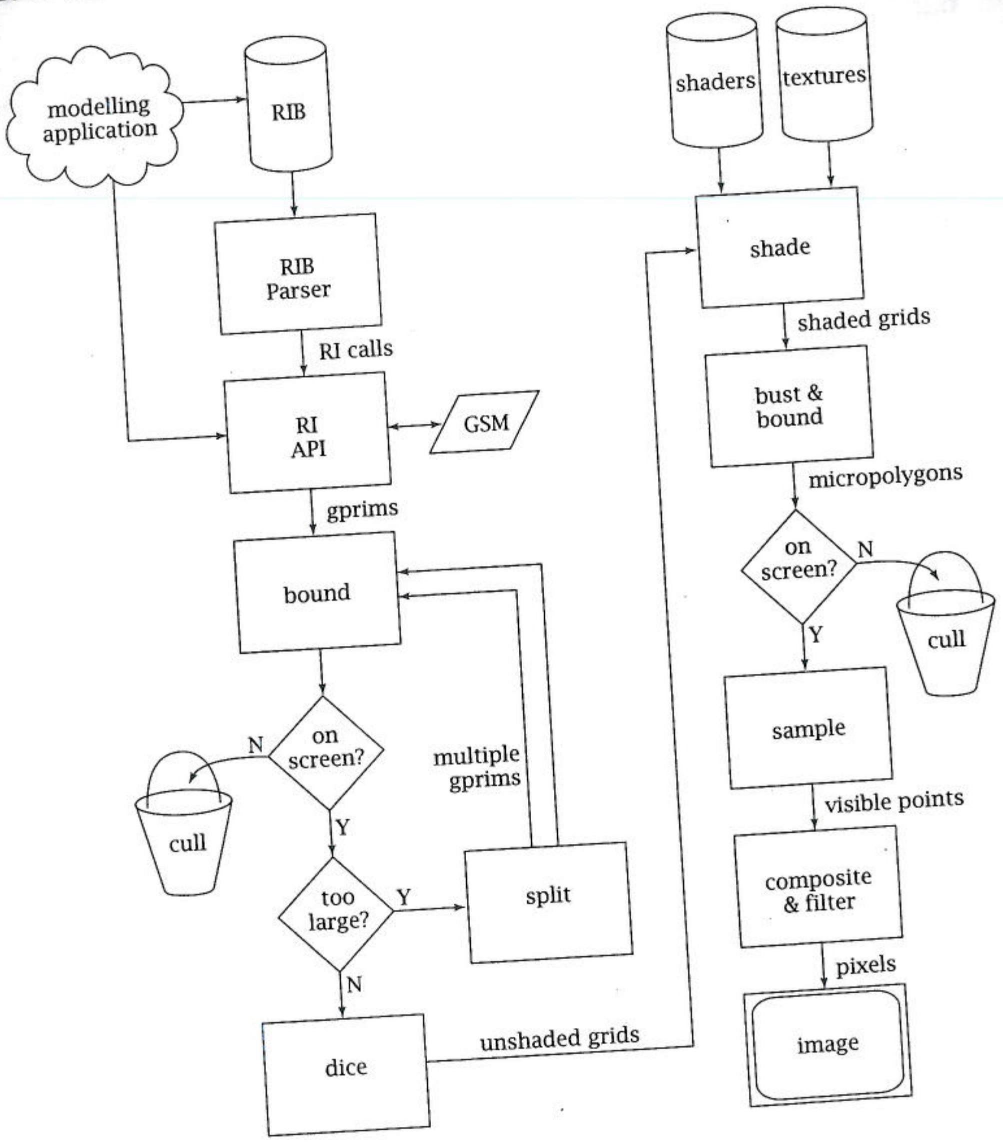

Figure 6.1 The REYES rendering pipeline.

is kept in a stack-based data structure within the RI layer. Whenever a geometric primitive arrives, the current top-of-the-stack set of attributes is attached to the primitive before it proceeds into the main REYES geometric processing engine.

6.2.1 Splitting Loop

The first thing that REYES does to arriving primitives is *bound* them. The renderer computes a camera-space axis-aligned bounding box that is guaranteed to contain the entire primitive. RenderMan does not contain any unbounded primitives (such as infinite planes), so this is generally straightforward. Most RenderMan primitives have the convex-hull property, which means that the primitive is entirely contained within the volume outlined by the vertices themselves. Primitives that don't have this property (such as Catmull-Rom patches) are converted to equivalent primitives that do (such as Bezier patches).

Next, the bounding box is checked to see if the primitive is actually on-screen. The camera description in the graphics state gives us the *viewing volume*, a 3D volume of space that contains everything that the camera can see. In the case of perspective projections, this is a rectangular pyramid, bounded on the sides by the perspective projection of the screen window (sometimes called the *screen space viewport* in graphics texts) and truncated at front and back by the near and far clipping planes; for an orthographic projection this is a simple rectangular box. It is important to note at this point that REYES does not do any global illumination or global intervisibility calculations of any kind. For this reason, any primitive that is not at least partially within the viewing volume cannot contribute to the image in any way and therefore is immediately culled (trivially rejected). Also, any one-sided primitives that are determined to be entirely back facing can be culled at this stage, because they also cannot contribute to the image.

If the primitive is (at least partially) on-screen, its size is tested. If it is deemed "too large" on-screen, according to a metric described later, it is *split* into smaller primitives. For most parametric primitives, this means cutting the primitive in two (or possibly four) along the central parametric line(s). For primitives that are containers of simpler primitives, such as polyhedra, splitting may mean roughly dividing into two containers each with fewer members. In either case, the idea is to create subprimitives that are simpler and smaller on-screen and more likely to be "small enough" when they are examined. This technique is often called "divide and conquer."

The resulting subprimitives are then dropped independently into the top of the loop, in no particular order, to be themselves bound, cull-tested, and size-tested. Eventually, the progeny of the original primitive will pass the size test and can move on to the next phase called *dicing*. Dicing converts the small primitive into a common data format called a *grid*. A grid is a tessellation of the primitive into a rectangular array of quadrilateral facets known as *micropolygons* (see Figure 6.2). (Because of the geometry of the grid, each facet is actually a tiny bilinear patch, but we call it a micropolygon nonetheless.) The vertices of these facets are the points that will be shaded later, so the facets themselves must be very small in order for the renderer to create the highly detailed, visually complex shading that we have come to expect. Generally, the facets will be on the order of one pixel in area. All primitives, regardless of original type, are converted into grids that look

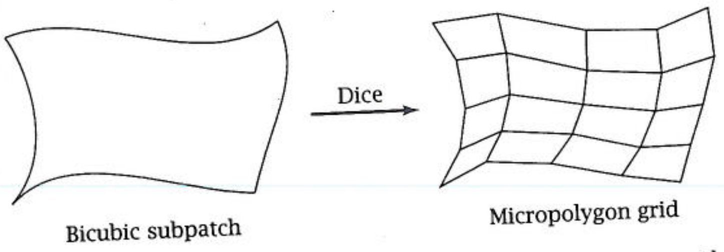

Figure 6.2 A primitive that is small enough will be diced into a grid of tiny bilinear facets known as micropolygons.

essentially the same to the remainder of the rendering pipeline. At this stage, each grid retains all of the primitive's attributes, and any primitive vertex variables that were attached to the primitive have been correctly interpolated onto every vertex in the grid, but the vertices have not yet been shaded.

6.2.2 Shading

The grid is then passed into the shading system to be shaded. *PRMan* first evaluates the displacement shader, which may move grid vertices and/or recompute shading normals. Then the surface shader is evaluated. In a typical surface shader, there are calls to `diffuse` or `illuminance` somewhere in the code. These routines require the evaluation of all of the light shaders attached to the object. The light shaders run as "coroutines" of the surface shader the first time that they are needed, but their results are cached and reused if they are accessed again (for example, in a subsequent `specular` call). When the surface shader finishes, it has computed the color and opacity of every grid vertex. Finally, the atmosphere shader is evaluated, making adjustments to the vertex color and opacity to simulate fog or other volumetric effects. The end result of all of these evaluations is that the grid now has final color and opacity assigned to each vertex, and the vertices themselves might be in different places than when originally diced. The other attributes of the original primitive are now mostly superfluous.

The details of the method that the shading system uses to assign color and opacity to each grid vertex are of very little consequence to the REYES pipeline itself but, of course, are of central importance to the users of *PRMan*. However, readers who are not very familiar with the material in Chapters 7 and 11 will probably want to review those chapters before spending too much time worrying about these details.

PRMan's shading system is an interpreter for the RenderMan Shading Language, which reads the byte-codes in the `.slo` file previously created by the Shading Language compiler (Hanrahan and Lawson, 1990). Notice that the data structure that the interpreter operates on, a grid, is a bunch of large rectangular arrays of floating-point numbers. For this reason, it may not be surprising that the interpreter actually executes as a virtual SIMD (single instruction, multiple data) vector math unit. Such vector pipelines were typical in 1980s-vintage supercomputers.

The run-time interpreter reads shader instructions one at a time and executes the operator on all grid vertices. It is important to contrast this scheme with the alternative, which would run the entire shader on the first grid vertex, then run it on the second grid vertex, and so on. The advantages of the breadth-first solution is efficiency. We are able to make near-optimal use of the pipelined floating-point units that appear in modern processors, and we have excellent cache performance due to strong locality of reference. In addition, for `uniform` variables, the grid has exactly one data value, not *nvertices* values. As a result, the interpreter is able to compute operations on uniform data exactly once on each grid, saving significantly over the redundant calculations that would be necessary in a depth-first implementation.

However, life is never all gravy. When a conditional or loop instruction is reached, the results may require that not all points on the grid enter the protected block. Because of this, the SIMD controller has *run flags*, which identify which grid vertices are "active" and which are "inactive" for the current instruction. For any instruction where the run-flag vector has at least one bit on, the instruction is executed, but only on those grid vertices that require it. Other operators such as `else`, `break`, and `continue` manipulate the run flags to ensure that the SIMD execution accurately simulates the depth-first execution.

Another advantage of the SIMD execution model is that neighborhood information is available for most grid vertices (except those on the grid edges), which means that differentials can be computed. These differentials, the difference between values at adjacent grid vertices, substitute for derivatives in all of the Shading Language operators that require derivative information. Those operators, known generically as *area operators*, include `Du`, `calculatenormal`, `texture`, and their related functions. Notice that grid vertices on edges (actually on two edges, not all four) have no neighbors, so their differential information is estimated. *PRMan* version 3.8 and lower estimated this poorly, which led to bad grid artifacts in second-derivative calculations. *PRMan* version 3.9 has a better differential estimator that makes many of these artifacts disappear, but it is still possible to confuse it at inflection points.

6.2.3 Texturing

The number and size of texture maps that are typically used in photorealistic scenes are so large that it is impractical to keep more than a small portion of them in memory at one time. For example, a typical frame in a full-screen CGI animation might access texture maps approaching 10 Gb in total size. Fortunately, because this data is not all needed at the highest possible resolution in the same frame, mip-maps can be used to limit this to 200 Mb of texture data actually read. However, even this is more memory than can or needs to be dedicated to such transient data. *PRMan* has a very sophisticated texture caching system that cycles texture data in as necessary, and keeps the total in-core memory devoted to texture to under 10

Mb in all but extreme cases. The proprietary texture file format is organized into 2D tiles of texture data that are strategically stored for fast access by the texture cache, which optimizes both cache hit rates and disk I/O performance.

Shadows are implemented using shadow maps that are sampled with *percentage closer* filtering (Reeves, Salesin, and Cook 1987). In this scheme, grid vertices are projected into the view of the shadow-casting light source, using shadow camera viewing information stored in the map. They are determined to be in shadow if they are farther away than the value in the shadow map at the appropriate pixel. In order to antialias this depth comparison, given that averaging depths is a nonsensical operation (because it implies that there is geometry in some halfway place where it doesn't actually exist), several depths from the shadow map in neighboring pixels are stochastically sampled, and the shadowing result is the percentage of the tests that succeeded.

6.2.4 Hiding

After shading, the shaded grid is sent to the hidden-surface evaluation routine. First, the grid is *busted* into individual micropolygons. Each micropolygon then goes through a miniature version of the main primitive loop. It is bounded, checked for being on-screen, and backface culled if appropriate. Next, the bound determines in which pixels this micropolygon might appear. In each such pixel, a stochastic sampling algorithm tests the micropolygon to see if it covers any of the several predetermined point-sample locations of that pixel. For any samples that are covered, the color and opacity of the micropolygon, as well as its depth, are recorded as a *visible point*. Depending on the shading interpolation method chosen for that primitive, the visible-point color may be a Gouraud interpolation of the four micropolygon corner colors, or it may simply be a copy of one of the corners. Each sample location keeps a list of visible points, sorted by depth. Of course, keeping more than just the frontmost element of the list is only necessary if there is transparency involved.

Once all the primitives that cover a pixel have been processed, the visible-point lists for each sample can be composited together and the resulting final sample colors and opacities blended together using the reconstruction filter to generate final pixel colors. Because good reconstruction kernels span multiple pixels, the final color of each pixel depends on the samples not merely in that pixel, but in neighboring pixels as well. The pixels are sent to the display system to be put into a file or onto a frame buffer.

6.2.5 Motion Blur and Depth of Field

Interestingly, very few changes need to be made to the basic REYES rendering pipeline to support several of the most interesting and unique features of *PRMan*. One of the most often used advanced features is motion blur. Any primitive may be motion blurred either by a moving transformation or by a moving deformation (or both). In the former case, the primitive is defined as a single set of control points with multiple transformation matrices; in the latter case, the primitive actually contains multiple sets of control points. In either case, the moving primitive when diced becomes a moving grid, with positional data for the beginning and ending of the motion path, and eventually a set of moving micropolygons.

The only significant change to the main rendering pipeline necessary to support this type of motion is that bounding box computations must include the entire motion path of the object. The hidden-surface algorithm modifications necessary to handle motion blur are implemented using the *stochastic sampling* algorithm first described by Cook et al. in 1984. The hidden-surface algorithm's point-sample locations are each augmented with a unique sample time. As each micropolygon is sampled, it is translated along its motion path to the position required for each sample's time.

PRMan only shades moving primitives at the start of their motion and only supports linear motion of primitives between their start and stop positions. This means that shaded micropolygons do not change color over time, and they leave constant-colored streaks across the image. This is incorrect, particularly with respect to lighting, as micropolygons will "drag" shadows or specular highlights around with them. In practice, this artifact is rarely noticed due to the fact that such objects are so blurry anyway.

Depth of field is handled in a very similar way. The specified lens parameters and the known focusing equations make it easy to determine how large the circle of confusion is for each primitive in the scene based on its depth. That value increases the bounding box for the primitive and for its micropolygons. Stochastically chosen lens positions are determined for each point sample, and the samples are appropriately jittered on the lens in order to determine which blurry micropolygons they see.

6.2.6 Shading before Hiding

Notice that this geometric pipeline has a feature that few other renderers share: the shading calculations are done before the hidden-surface algorithm is run. In normal scanline renderers, polygons are depth-sorted, the visible polygons are identified, and those polygons are clipped to create "spans" that cover portions of a scanline. The end points of those spans are shaded and then painted into pixels. In ray tracing renderers, pixel sample positions are turned into rays, and the objects that are hit by (and therefore visible from) these rays are the only things that are shaded. Radiosity renderers often resolve colors independently of a particular viewpoint but nonetheless compute object inter visibility as a prerequisite to energy transfer. Hardware *z*-buffer algorithms do usually shade before hiding, as REYES does; however, they generally only compute true shading at polygon vertices, not at the interiors of polygons.

One of the significant advantages of shading before hiding is that displacement shading is possible. This is because the final locations of the vertices are not needed by the hider until after shading has completed, and therefore the shader is free to move the points around without the hider ever knowing. In other algorithms, if the shader moved the vertices after the hider had resolved surfaces, it would invalidate the hider's results.

The biggest disadvantage of shading before hiding is that objects are shaded before it is known whether they will eventually be hidden from view. If the scene has a large depth complexity, large amounts of geometry might be shaded and then subsequently covered over by objects closer to the camera. That would be a large waste of compute time. In fact, it is very common for this to occur in *z*-buffer renderings of complicated scenes. This disadvantage is addressed in the enhanced algorithm described in Section 6.3.2.

6.2.7 Memory Considerations

In this pipeline, each stage of processing converts a primitive into a finer and more detailed version. Its representation in memory gets larger as it is split, diced, busted, and sampled. However, notice also that every primitive is processed independently and has no interaction with other primitives in the system. Even sibling subprimitives are handled completely independently. For this reason, the geometric database can be streamed through the pipeline just as a geometric database is streamed through typical *z*-buffer hardware. There is no long-term storage or buffering of a global database (except for the queue of split primitives waiting to be bounded, which is rarely large), and therefore there is almost no memory used by the algorithm. With a single exception: the visible point lists.

As stated earlier, no visible-point list can be processed until it is known that all of the primitives that cover its pixel have, in fact, been processed. Because the streaming version of REYES cannot know that any given pixel is done until the last primitive is rendered, it must store all the visible-point lists for the entire image until the very end. The visible-point lists therefore contain a point-sampled representation of the *entire* geometric database and consequently are quite large. Strike that. They are absolutely huge—many gigabytes for a typical high-resolution film frame. Monstrously humongous. As a result, the algorithm simply would not be usable if implemented in this way. Memory-sensitive enhancements are required to make the algorithm practical.

6.3 Enhanced Geometric Pipeline

The original REYES paper recognized that the memory issue was a problem, even more so in 1985 than it is now. So it provided a mechanism for limiting memory use, and other mechanisms have been added since, which together make the algorithm much leaner than most other algorithms.

6.3.1 Bucketing

In order to alleviate the visible-point memory problem, a modified REYES algorithm recognizes that the key to limiting the overall size of the visible-point memory is to know that certain pixels are done before having to process the entire database. Those pixels can then be finished and freed early. This is accomplished by dividing the image into small rectangular pixel regions, known as *buckets*, which will be processed one by one to completion before significant amounts of work occur on other buckets.

The most important difference in the pipeline is in the bounding step, which now also sorts the primitives based on which buckets they affect (that is, which buckets the bounding box overlaps). If a primitive is not visible in the current bucket of interest, it is put onto a list for the first bucket where it will matter and is thereby held in its most compact form until truly needed.

After this, the algorithm proceeds in the obvious way. Buckets are processed one at a time. Objects are removed from the list for the current bucket and either split or diced. Split primitives might be added back to the list or might be added to the lists of future buckets, depending on their bounding boxes. Diced primitives go through the normal shading pipeline and are busted. During busting, the micropolygons are bound and similarly bucket-sorted. Micropolygons that are not in the current bucket of interest are not sampled until the appropriate bucket is being processed. Figure 6.3 shows four primitives whose disposition is different. Primitive A will be diced, shaded, and sampled in the current bucket. Primitive B needs to be split, and half will return to the current bucket while half will be handled in a future bucket. Primitive C is in the current bucket because its bounding box touches it (as shown), but once split, you can see that both child primitives will fall into future buckets. Primitive D will be diced and shaded in the current bucket, but some of the micropolygons generated will be held for sampling until the next bucket is processed.

Eventually, there are no more primitives in the current bucket's list, because they all have either been sampled or transferred to future buckets. At that point, all of the visible-point lists in that bucket can be resolved and the pixels for that bucket displayed. This is why *PhotoRealistic RenderMan* creates output pixels in little blocks, rather than in scanlines like many algorithms. Each block is a bucket. The algorithm does not require that the buckets be processed in a particular order, but in practice the implementation still uses a scanline-style order, processing buckets one horizontal row at a time, left to right across the row, and rows from top to bottom down the image.

The major effect of this pipeline change is the utilization of memory. The entire database is now read into memory and sorted into buckets before any significant amount of rendering is done. The vast majority of the geometric database is stored in the relatively compact form of per-bucket lists full of high-level geometric primitives. Some memory is also used for per-bucket lists of micropolygons that have already been diced and shaded but are not relevant to the current bucket. The

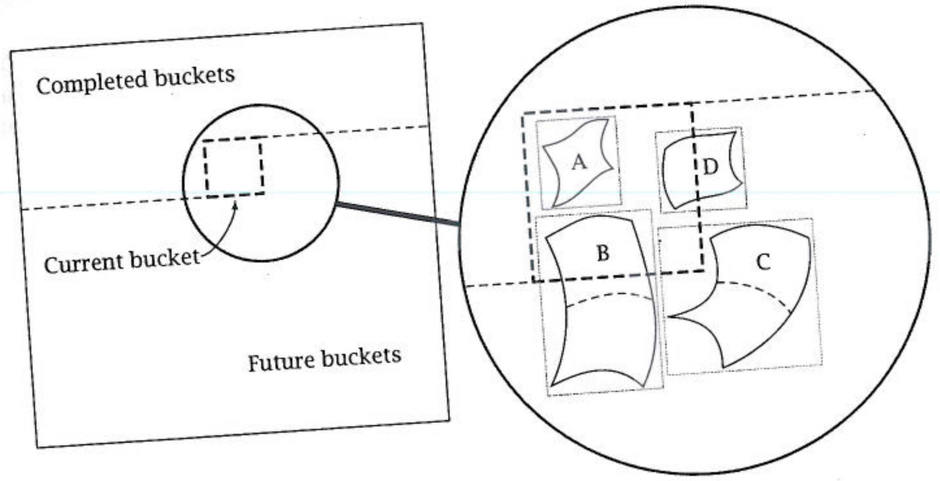

Figure 6.3 When the renderer processes the primitives that are on the list for the currrent bucket, their size and positions determine their fates.

visible-point lists have been reduced to only those that are part of the current bucket, a small fraction of the lists required for an entire image. Thus we have traded visible-point list memory for geometric database memory, and in all but the most pathological cases, this trade-off wins by orders of magnitude.

6.3.2 Occlusion Culling

As described so far, the REYES algorithm processes primitives in arbitrary order within a bucket. In the preceding discussion, we mentioned that this might put a primitive through the dicing/shading/hiding pipeline that will eventually turn out to be obscured by a later primitive that is in front of it. If the dicing and shading of these objects takes a lot of computation time (which it generally does in a photorealistic rendering with visually complex shaders), this time is wasted. As stated, this problem is not unique to REYES (it happens to nearly every *z*-buffer algorithm), but it is still annoying. The enhanced REYES algorithm significantly reduces this inefficiency by a process known as *occlusion culling*.

The primitive bound-and-sort routine is changed to also sort each bucket's primitives by depth. This way, objects close to the camera are taken from the sorted list and processed first, while farther objects are processed later. Simultaneously, the hider keeps track of a simple hierarchical data structure that describes how much of the bucket has been covered by opaque objects and at what depths. Once the bucket is completely covered by opaque objects, any primitive that is entirely behind that covering is occluded. Because it cannot be visible, it can be culled before the expensive dicing and shading occurs (in the case of procedural primitives, before they are even loaded into the database). By processing primitives in front-to-back order, we maximize the probability that at least some objects will be occluded and culled. This optimization provides a two- to ten-times speedup in the rendering times of typical high-resolution film frames.

6.3.3 Network Parallel Rendering

In the enhanced REYES algorithm, most of the computation—dicing, shading, hiding, and filtering—takes place once the primitives have been sorted into buckets. Moreover, except for a few details discussed later, those bucket calculations are generally independent of each other. For this reason, buckets can often be processed independently, and this implies that there is an opportunity to exploit parallelism. *PRMan* does this by implementing a large-grain multiprocessor parallelism scheme known as *NetRenderMan*.

With NetRenderMan, a parallelism-control client program dispatches work in the form of bucket requests to multiple independent rendering server processes. Server processes handle all of the calculation necessary to create the pixels for the requested bucket, then make themselves available for additional buckets. Serial sections of the code (particularly in database sorting and redundant work due to primitives that overlap multiple buckets) and network latency cut the overall multiprocessor efficiency to approximately 70–80% on typical frames, but nevertheless the algorithm often shows linear speedup through 8–10 processors. Because these processes run independently of each other, with no shared data structures, they can run on multiple machines on the network, and in fact on multiple processor architectures in a heterogeneous network, with no additional loss of efficiency.

6.4 Rendering Attributes and Options

With this background, it is easy to understand certain previously obscure rendering attributes and options, and why they affect memory and/or rendering time, and also why certain types of geometric models render faster or slower than others.

6.4.1 Shading Rate

In the RenderMan Interface, the `ShadingRate` of an object refers to the frequency with which the primitive must be shaded (actually measured by sample area in pixels) in order to adequately capture its color variations. For example, a typical `ShadingRate` of 1.0 specifies one shading sample per pixel, or roughly Phong-shading style. In the REYES algorithm, this constraint translates into micropolygon size. During the dicing phase, an estimate of the raster space size of the primitive is made, and this number is divided by the shading rate to determine the number of micropolygons that must make up the grid. However, the dicing tessellation is

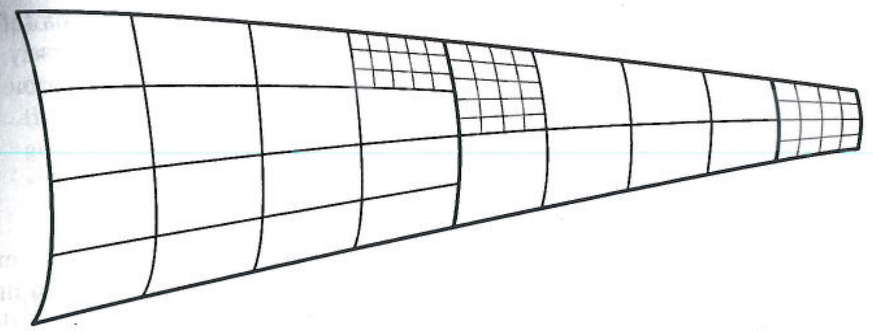

Figure 6.4 Adaptive parametric subdivision leads to adjacent grids that are different sizes parametrically and micropolygons that approximate the desired shading rate.

always done in such a manner as to create (within a single grid) micropolygons that are of identically sized rectangles in the parametric space of the primitive. For this reason, it is not possible for the resulting micropolygons in a grid to all be exactly the same size in raster space, and therefore they will only approximate the shading rate requested of the object. Some will be slightly larger, others slightly smaller than desired.

Notice, too, that any adjacent sibling primitive will be independently estimated, and therefore the number of micropolygons that are required for it may easily be different (even if the sibling primitive is the same size in parametric space). In fact, this is by design, as the REYES algorithm fundamentally takes advantage of *adaptive subdivision* to create micropolygons that are approximately equal in size in raster space independent of their size in parametric space (see Figure 6.4). That way, objects farther away from the camera will create a smaller number of equally sized micropolygons, instead of creating a sea of inefficient nanopolygons. Conversely, objects very close to the camera will create a large number of micropolygons, in order to cover the screen with sufficient shading samples to capture the visual detail that is required of the close-up view. For this reason, it is very common for two adjacent grids to have different numbers of micropolygons along their common edge, and this difference in micropolygon size across an edge is the source of some shading artifacts that are described in Section 6.5.4.

In most other rendering algorithms, a shading calculation occurs at every hidden-surface sample, so raising the antialiasing rate increases the number of shading samples as well. Because the vast majority of calculations in a modern renderer are in the shading and hidden-surface calculations, increasing `PixelSamples` therefore has a direct linear effect on rendering time. In REYES, these two calculations are decoupled, because shading rate affects only micropolygon dicing, not hidden-surface evaluation. Antialiasing can be increased without spending any additional time shading, so raising the number of pixel samples in a REYES image will make a much smaller impact on rendering time than in other algorithms (often in the range of percentage points instead of multiplicative factors). Conversely, adjusting the shading multiplicative rate will have a large impact on rendering time in images where shading dominates the calculation.

6.4.2 Bucket Size and Maximum Grid Size

The bucket size option obviously controls the number of pixels that make up a bucket and inversely controls the number of buckets that make up an image. The most obvious effect of this control is to regulate the amount of memory devoted to visible-point lists. Smaller buckets are more memory efficient because less memory is devoted to visible-point lists. Less obviously, smaller buckets partition the geometric database into larger numbers of shorter, sorted primitive lists, with some consequential decrease in sorting time. However, this small effect is usually offset by the increase in certain per-bucket overhead.

The maximum grid size option controls dicing by imposing an upper limit on the number of micropolygons that may occur in a single grid. Larger grids are more efficient to shade because they maximize vector pipelining. However, larger grids also increase the amount of memory that can be devoted to shader global and local variable registers (which are allocated in rectangular arrays the size of a grid). More interestingly, however, the maximum grid size creates a loose upper bound on the pixel area that a grid may cover on-screen—a grid is unlikely to be much larger than the product of the maximum grid size and the shading rate of the grid. This is important in relation to the bucket size because grids that are larger than a bucket will tend to create large numbers of micropolygons that fall outside of the current bucket and that must be stored in lists for future buckets. Micropolygons that linger in such lists can use a lot of memory.

In the past, when memory was at a premium, it was often extremely important to optimize the bucket size and maximum grid size to limit the potentially large visible-point and micropolygon list memory consumption. On modern computers, it is rare that these data structures are sufficiently large to concern us, and large limits are perfectly acceptable. The default values for bucket size, 16×16 pixel buckets, and maximum grid size, 256 micropolygons per grid, work well except under the most extreme situations.

6.4.3 Transparency

Partially transparent objects cause no difficulty to the algorithm generally; however, they can have two effects on the efficiency of the implementation. First, transparent objects clearly affect the memory consumption of the visible-point lists. Due to the mathematical constraints of the compositing algebra used by *PRMan*, it is not possible to composite together the various partially transparent layers that are held in the visible-point list of a sample until the sample is entirely complete. Notice that an opaque layer can immediately truncate a list, but in the presence of large amounts of transparency, many potentially visible layers must be kept around. Second, and more importantly, transparent layers do not contribute to the occlusion culling of future primitives, which means that more primitives are diced and shaded than usual. Although this should be obvious (since those primitives are probably going to be seen through the transparent foreground), it is often quite surprising to see the renderer slow down as much as it does when the usually extremely efficient occlusion culling is essentially disabled by transparent foreground layers.

6.4.4 Displacement Bounds

Displacement shaders can move grid vertices, and there is no built-in constraint on the distance that they can be moved. However, recall that shading happens halfway through the rendering pipeline, with bounding, splitting, and dicing happening prior to the evaluation of those displacements. In fact, the renderer relies heavily on its ability to accurately yet tightly bound primitives so that they can be placed into the correct bucket. If a displacement pushes a grid vertex outside of its original bounding box, it will likely mean that the grid is also in the wrong bucket. Typically, this results in a large hole in the object corresponding to the bucket where the grid "should have been considered, but wasn't."

This is avoided by supplying the renderer a bound on the size of the displacement generated by the shader. From the shader writer's point of view, this number represents the worst-case displacement magnitude—the largest distance that any vertex might travel, given the calculations inherent in the displacement shader itself. From the renderer's point of view, this number represents the padding that must be given to every bounding box calculation prior to shading, to protect against vertices leaving their boxes. The renderer grows the primitive bounding box by this value, which means that the primitive is diced and shaded in a bucket earlier than it would normally be processed. This often leads to micropolygons that are created long before their buckets need them, which then hang around in bucket micropolygon lists wasting memory, or primitives that are shaded before it is discovered that they are offscreen. Because of these computational and memory inefficiencies of the expanded bounds, it is important that the displacement bounds be as tight as possible, to limit the damage.

6.4.5 Extreme Displacement

Sometimes the renderer is stuck with large displacement bounds, either because the object really does displace a large distance or because the camera is looking extremely closely at the object and the displacements appear very large on-screen. In extreme cases, the renderer can lose huge amounts of memory to micropolygon lists that contain most of the geometric database. In cases such as these, a better option is available. Notice that the problem with the displacement bound is that it is a worst-case estimate over the primitive as a whole, whereas the small portion of the primitive represented by a single small grid usually does not contain the worst-case displacement and actually could get away with a much smaller (tighter) bound. The solution to this dilemma is to actually *run the shader* to evaluate the true displacement magnitude for each grid on a grid-by-grid basis and then store those values with the grid as the *exact* displacement bound. The disadvantage of this technique is that it requires the primitive to be shaded twice, once solely to determine the displacement magnitude and then again later to generate the color when the grid is processed normally in its new bucket. Thus, it is a simple space-time trade-off.

This technique is enabled by the `extremedisplacement` attribute, which specifies a threshold raster distance. If the projected raster size of the displacement bound for a primitive exceeds the extreme displacement limit for that primitive, the extra shading calculations are done to ensure economy of memory. If it does not, then the extra time is not spent, under the assumption that for such a small distance the memory usage is transient enough to be inconsequential.

6.4.6 Motion-Factor

When objects move quickly across the screen, they become blurry. Such objects are indistinct both because their features are spread out over a large region and because their speed makes it difficult for our eyes to track them. As a result, it is not necessary to shade them with particularly high fidelity, as the detail will just be lost in the motion blur. Moreover, every micropolygon of the primitive will have a very large bounding box (corresponding to the length of the streak), which means that fine tessellations will lead to large numbers of micropolygons that linger a long time in memory as they are sampled by the many buckets along their path.

The solution to this problem is to enlarge the shading rate of primitives if they move rapidly. It is possible for the modeler to do this, of course, but it is often easier for the renderer to determine the speed of the model and then scale the shading rates of each primitive consistently. The attribute that controls this calculation is a `GeometricApproximation` flag known as `motionfactor`. For obscure reasons, `motionfactor` gives a magnification factor on shading rate per every 16 pixels of blurring. Experience has shown that a motion-factor of 1.0 is appropriate for a large range of images.

The same argument applies equally to depth of field blur, and in the current implementation, `motionfactor` (despite its name) also operates on primitives with large depth of field blurs as well.

6.5 Rendering Artifacts

Just as an in-depth understanding of the REYES pipeline helps you understand the reason for, and utility of, various rendering options and attributes, it also helps you understand the causes and solutions for various types of geometric rendering artifacts that can occur while using *PhotoRealistic RenderMan*.

6.5.1 Eye Splits

Sometimes *PhotoRealistic RenderMan* will print the error message "Cannot split primitive at eye plane," usually after appearing to stall for quite a while. This error message is a result of perhaps the worst single algorithmic limitation of the modified REYES algorithm: in order to correctly estimate the shading rate required for a primitive, the primitive must first be projected into raster space in order to evaluate its size. Additionally, recall that the first step in the geometric pipeline is to bound the primitive and sort it into buckets based on its position in raster space, which requires the same projection. The problem is that the mathematics of perspective projection only works for positions that are in front of the camera. It is not possible to project points that are behind the camera. For this reason, the renderer must divide the primitive into areas that are in front of and areas that are behind the camera.

Most rendering algorithms, if they require this projection at all, would clip the primitive against the near clipping plane (hence the name) and throw away the bad regions. However, the entire REYES geometric and shading pipelines require subprimitives that are rectangular in parametric space, which can be split and diced cleanly. Clipping does not create such primitives and cannot be used. Instead, REYES simply splits the primitive hoping that portions of the smaller subprimitives will be easier to classify and resolve.

Figure 6.5 shows the situation. Notice that primitives that lie entirely forward of the eye plane are projectable, so can be accepted. Primitives that lie entirely behind the near clipping plane can be trivially culled. It is only primitives that span both planes that cannot be classified and are split. The region between the planes can be called the "safety zone." If a split line lies entirely within this zone (in 3D, of course), both children of the bad primitive are classifiable, which is the situation we are hoping for. If the split line straddles a plane, at least we will shave some of the bad primitive away and the remaining job, we hope, is slightly easier. REYES splits as smartly as it can, attempting to classify subprimitives.

Unhappily, there are geometric situations where the splitting simply doesn't work in a reasonable number of steps. "Reasonable" is defined as a small integer because each split doubles the number of subprimitives, so even 10 attempts create $2^{10} = 1024$ primitives. If, after splitting the maximum permitted number of times, the primitive still cannot be classified, REYES gives up and throws the primitive away and prints the "Cannot split" message. If that primitive was supposed to be visible, the lost section will leave a hole in the image.

Primitives that have large displacement bounds, or are moving rapidly toward the camera, will exacerbate the eye-splitting problem because the parametric splitting process will do very little to reduce the bounding box. Indeed, for primitives for which the camera is inside the displacement bound of part of the surface, or primitives whose motion path actually goes through the camera, splitting can never succeed.

In order to reduce the artifacts due to eye-split culling, the key is to give the renderer the largest possible safety zone. Place the near clipping plane as far forward

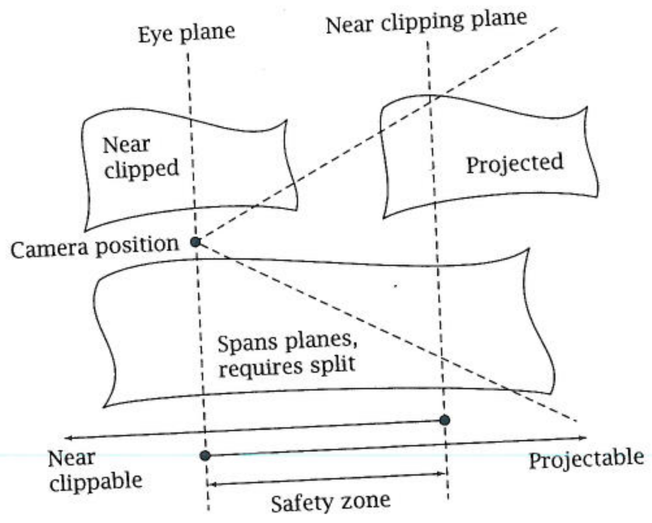

Figure 6.5 The geometric relationship of the near and eye planes gives rise to three categories of primitives: the cullable, the projectable, and the "spanners," which require splitting.

as is possible without otherwise affecting the image. The near clipping plane can be placed surprisingly far forward for most shots made with cameras that have reasonable fields of view. If you don't normally set the clipping plane, set it to some small but reasonable value immediately—the default value of 1e-10 is just about as bad as you could get! The number of splitting iterations that will be permitted can be controlled with a rendering option, and if you permit a few more, you can sometimes help handling of large but otherwise simple primitives. Beware, however, of the displacement and motion cases, because upping the limit will just let the renderer waste exponentially more time before it gives up.

Also, make sure that displacement bounds for primitives near the camera (for example, the ground plane) are as tight as possible and that the shader is coded so that the displacement itself is as small as possible. If you place the camera so close to some primitive that the displacement bound is a significant portion of the image size, there will be no end of trouble with both eye splits and displacement stretching (discussed later). In flyovers of the Grand Canyon, the canyon should be modeled, not implemented as a displacement map of a flat plane!

It will also help to keep the camera as high off the ground as possible without ruining the composition of the shot. REYES simply has lots of trouble with worm's-eye views. And make sure that no object is flying through the camera (you wouldn't do that in live-action photography, would you?). Generally, if you pretend that the CG camera has a physical lens that keeps objects in the scene at least a certain distance away and respect that border, you will have fewer problems with eye splits.

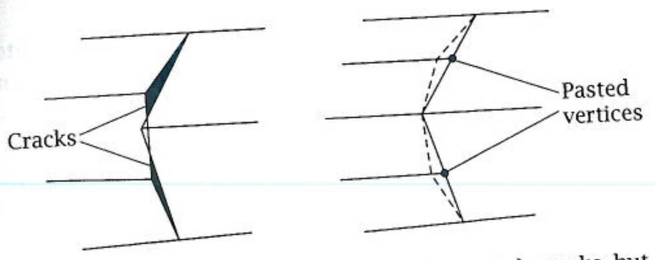

Figure 6.6 Tessellation differences result in patch cracks, but these can be repaired by moving binary-diced vertices.

6.5.2 Patch Cracks

Patch cracks are tiny holes in the surface of objects that are caused by various errors in the approximation of primitives by their tessellations (we'll use the term loosely to include tessellation-created cracks on primitives other than patches). Patch cracks usually appear as scattered pinholes in the surface, although they can sometimes appear as lines of pinholes or as small slits. Importantly, they always appear along parametric lines of primitives. They are recognizably different from other holes created by clipping, culling, or bucketing errors, which are usually larger, often triangular, and occur in view-dependent places (like on silhouettes or aligned with bucket boundaries).

Patch cracks occur because when objects are defined by sets of individual primitives, the connectedness of those primitives is only implied by the fact that they abut, that they have edges that have vertices in common. There is no way in the RenderMan Interface to explicitly state that separate primitives have edges that should be "glued together." Therefore, as the primitives go through the geometric pipeline independently, there is a chance that the mathematical operations that occur on one version of the edge will deviate from those on the other version of the edge, and the vertices will diverge. If they do, a crack occurs between them.

The deviations can happen in several ways. One is the tessellation of the edge by adjacent grids being done with micropolygons of different sizes. Figure 6.6 shows that such tessellations naturally create intermediate grid vertices that do not match, and there are tiny holes between the grids. For many years, *PRMan* has had a switch that eliminated most occurrences of this type of crack. Known as *binary dicing*, it requires that every grid have tessellations that create a power-of-two number of micropolygons along each edge. Although these micropolygons are smaller than are required by shading rate alone, binary dicing ensures that adjacent grids have tessellations that are powers-of-two multiples of each other (generally, a single factor of two). Thus, alternating vertices will coincide, and the extra vertices on one side are easily found and "pasted" to the other surface.

Another way that patch cracks can happen is when the displacement shader that operates on grid vertices gets different results on one grid from another. If a common vertex displaces differently on two grids, the displacement literally rips the surface apart, leaving a crack between. One common reason for such differing results is displacement mapping using texture filter sizes that are mismatched (see Section 6.5.4). The texture call then returns a slightly different value on the two grids, and one grid displaces to a different height than the other. Another common reason is displacement occurring along slightly different vectors. For example, the results of `calculatenormal` are almost guaranteed to be different on the left edge of one grid and on the right edge of the adjacent grid. If displacement occurs along these differing vectors, the vertices will obviously go to different places, opening a crack. Unfortunately, only careful coding of displacement shaders can eliminate this type of cracking.

Notice that patch cracks cannot happen on the interiors of grids. Because grid micropolygons explicitly share vertices, it is not possible for such neighbor micropolygons to have cracks between them. For this reason, patch cracks will only occur along boundaries of grids, and therefore along parametric edges. In *PRMan* versions prior to 3.9, patch cracks could occur on any grid edge, including those that resulted from splitting a patch into subpatches. Later versions of *PRMan* have a crack-avoidance algorithm that glues all such subpatches together. Therefore, modern versions of *PRMan* will only exhibit patch cracks along boundaries of original primitives, not along arbitrary grid boundaries.

6.5.3 Displacement Stretching

Another problem that displacement shaders might create is stretching of micropolygons. This is caused when displacement shaders move the vertices of a micropolygon apart, so that it no longer obeys the constraint that it is approximately the size specified by the shading rate.

In the process of dicing, the renderer estimates the size of the primitive onscreen and makes a grid that has micropolygons that approximately match the shading rate. Shading then occurs on the vertices of the grid (the corners of the micropolygons). Displacement shaders are permitted to move grid vertices, but there is no constraint on where they are moved. If two adjacent grid vertices are moved in different directions (wildly or subtly), the area of the micropolygon connecting them will change. Generally, this change is so small that the micropolygon is still safely in the range expected of shading rate. However, if the displacement function has strong high frequencies, adjacent grid vertices might move quite differently. For example, an embossing shader might leave some vertices alone while moving vertices inside the embossed figure quite a distance. The micropolygons whose corners move very differently will change size radically, and sometimes will be badly bent or twisted (see Figure 6.7).

Twisted micropolygons have unusual normal vectors, and this alone may be enough to cause shading artifacts. For example, highly specular surfaces are very sensitive to normal vector orientation, so a micropolygon that is twisted in an unusual direction may catch an unexpected highlight.

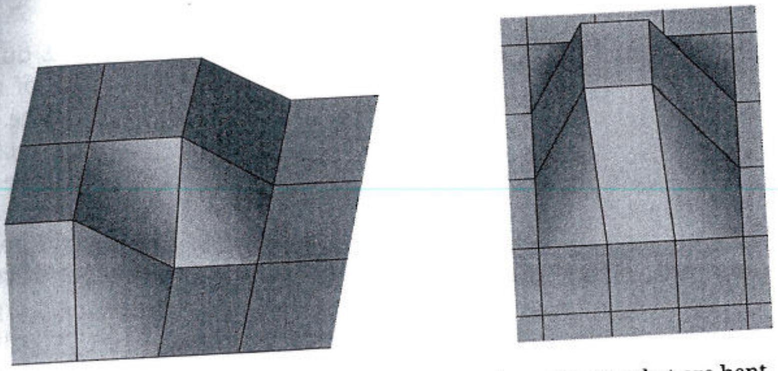

Figure 6.7 Displacement stretching leads to micropolygons that are bent, twisted, or significantly larger than anticipated.

More common, however, is that the stretching of the micropolygons will itself be visible in the final image. An individual flat-shaded micropolygon creates a constant-colored region in the image. With a standard shading rate of around a pixel, every pixel gets a different micropolygon and the flat shading is not visible. But large stretched or long twisted micropolygons will cover many pixels, and the constant-colored region will be evident. Sometimes this takes the form of alternating dark and light triangles along the face of a displacement "cliff." Corners of micropolygons that are shaded for the top of the plateau hang down, while corners of micropolygons that are shaded for the valley poke up, interleaved like teeth of a gear.

The visual artifacts of these problems can be somewhat ameliorated by using smooth shading interpolation, which will blur the shading discontinuities caused by the varying normals. However, the geometric problems remain. The primary solution is to lower the frequency content of the displacement shader so that adjacent micropolygons cannot have such wildly varying motion (see Chapter 11 on antialiasing shaders for hints). If this cannot be done, the brute-force approach is to reduce the shading rate to values such as 0.25 pixels or smaller so that even stretched micropolygons stay under one pixel in size. However, this will have significant performance impact, because shading time is inversely proportional to shading rate. Fortunately, shading rate is an attribute of individual primitives, so the extra expense can be limited to the part of the model that requires it.

6.5.4 Texture Filter Mismatches

As primitives are split, their subprimitives proceed through the rendering pipeline independently, and when the time comes to dice them, their own individual size on-screen determines the tessellation rate. As a result, it is often the case that the adjacent grids resulting from adjacent subprimitives will project to different sizes on the screen and as a result will tessellate at different rates during dicing. Tessellated micropolygons are rectangular in the parametric space of the original primitive, and all of the micropolygons in a single grid will be the same size in that parametric space, but due to the differing tessellation rates, the micropolygons that make up the adjacent grids will have different sizes in parametric space.

One of the artifacts that results from this difference is that filtering calculations based on parametric size will change discontinuously across a grid boundary. In Chapter 11 on shader antialiasing, various filtering techniques are discussed that use parametric size as part of the calculation of filter width. This type of size discontinuity will result in filtering discontinuities, which can be visible in the final image. For example, in simple texturing, the texture filter size defaults to the micropolygon size. The resulting texture can have visible changes in sharpness over the surface of an otherwise smooth primitive. If the result of a texture call is used as a displacement magnitude, a displacement crack can result (Section 6.5.2).

Recent versions of *PRMan* have significantly reduced problems such as these by the introduction of smooth derivatives. The derivatives and the parametric size values that are available to shaders now describe a smoothly varying idealized parametric size for the micropolygon. That is, the values do not exactly match the true size of the micropolygon in parametric space, but instead track closely the desired parametric size given the shading rate requested (in some sense compensating for the compromises that needed to be made to accommodate binary dicing or other tessellation constraints at the time of dicing). These smoothly varying parametric size estimates ameliorate texture filter size mismatches, both in built-in shading functions and in antialiased procedural textures. Generally, the fact that micropolygons are not exactly the size that they advertise is a small issue, and where it is an issue, it can be compensated for by minor modifications to the shader.

6.5.5 Conclusion

The REYES rendering architecture is so general and flexible that new primitives, new graphics algorithms, and new effects features have been added modularly to the existing structure almost continously for over 15 years. The system has evolved from an experimental testbed with a seemingly unattainable dream of handling tens of thousands of primitives into a robust production system that regularly handles images a hundred times more complex than that. The speed and memory enhancements that have been added may appear to have been short-term requirements, as Moore's law allows us to run the program on computers that are faster and have more memory without any additional programming. However, this is shortsighted, for our appetite for complexity has scaled, too, as fast or faster than Moore's law allows. Undoubtedly, in 15 more years, when computers with a terabyte of main memory are common and optical processors chew up 1 billion primitives without flinching, we will still be using the REYES architecture to compute our holofilms.