

A Rendering Architecture

2

2.1 Introduction

Writing a production renderer is a large software development project that requires a balance between the creativity of designing algorithms and the discipline of writing robust code. A renderer often starts out as a toy program to demonstrate a new optical effect. Over time, the software evolves into a large, poorly modularized system or gets locked into a particular architecture that is difficult to extend. Starting with a sound design will save effort in the long run.

This chapter presents an object-oriented design for a production renderer with a micropolygon architecture. It should be considered a skeleton upon which the content of the subsequent chapters may be hung. Our sample renderer will adhere to the RenderMan standard (Pixar, 2000) and example header files will be written in C++. We assume that the reader is familiar with both.

The first section will chart the course from the RenderMan Application Programmers' Interface (API) to the front door of the rendering engine. This part is independent from the choice of rendering algorithms, and is applicable to anything from a real-time RIB previewer to a global illumination renderer.

Next, micropolygon architectures and the Reyes pipeline will be discussed from a theoretical point of view. In preference to immediately presenting a Reyes implementation, a supporting cast of classes will be introduced that take on the burden of many steps. Each has been designed to create interfaces at logical points and promote modularity. An overview of the rendering architecture is shown in Figure 2.1.

In the final section, the Reyes framework will be coded in terms of these supporting classes. A ray-tracing framework will then be shown, and the two will ultimately be combined to create a hybrid production renderer. These final steps are simplified by the versatility of the object-oriented design presented earlier in the chapter.

2.2 The Hierarchical Graphics State

Just below the RenderMan API sits the *state* subsystem. It manages the hierarchical graphics state using five stack data structures:

- Mode
- Option

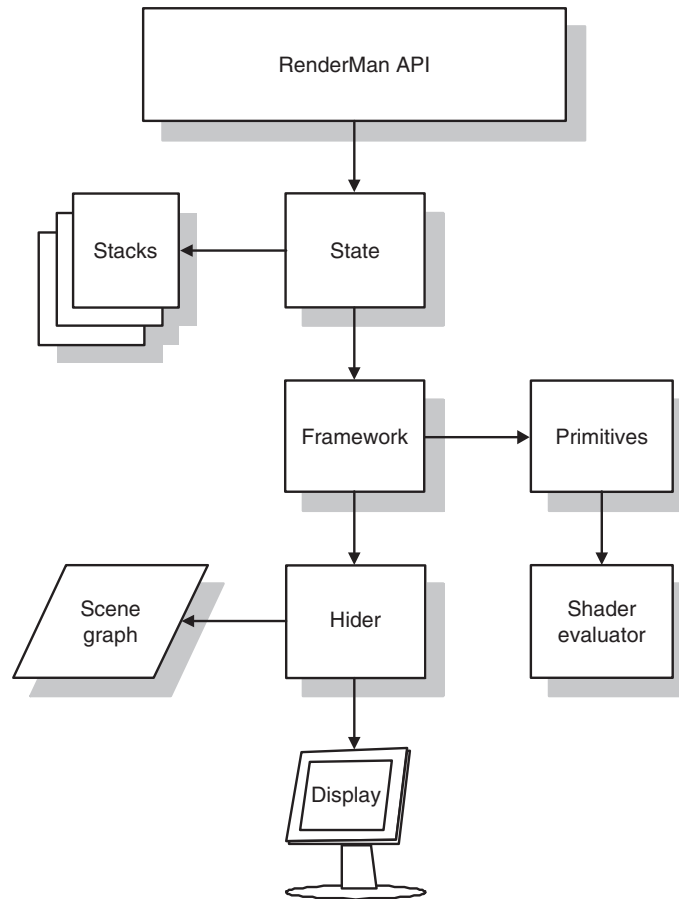


Figure 2.1 An overview of the architecture.

- Attribute
- Transformation
- Object and Light

After examining each of these stacks, we will show how to assemble them into a single class that manages the hierarchical graphics state. Finally, how the state impacts the process of inserting primitives into the scene will be covered.

2.2.1 The Mode Stack

The interface supports the concept of modes to prevent bad input. Functions with names like `RiXxxBegin()` and `RiXxxEnd()` change the mode of the interface. For example, between `RiMotionBegin()` and `RiMotionEnd()` the interface is in motion mode. The complete set of modes are as follows:

<undefined>	Before <code>RiBegin</code> or after <code>RiEnd</code>
base	Between <code>RiBegin</code> and <code>RiEnd</code>

frame	Between	RiFrameBegin	and	RiFrameEnd
world	Between	RiWorldBegin	and	RiWorldEnd
attribute	Between	RiAttributeBegin	and	RiAttributeEnd
transform	Between	RiTransformBegin	and	RiTransformEnd
solid	Between	RiSolidBegin	and	RiSolidEnd
object	Between	RiObjectBegin	and	RiObjectEnd
motion	Between	RiMotionBegin	and	RiMotionEnd

Each Begin function sets the current mode as shown above. The corresponding End function restores the current mode to what it was before the Begin-End block. Therefore, a stack of modes is necessary to handle nested blocks. The Begin function pushes the appropriate mode onto the stack. All of the End functions pop the mode stack. The current state is always the one at the top of the stack.

One type of bad input that must be recognized is improper nesting of Begin and End. This can happen when the user forgets to explicitly end a block or incorrectly orders RiXxxEnd() operations:

```
AttributeBegin
TransformBegin
    Translate 10000 0 0
    ReadArchive "planet.rib"
AttributeEnd          # WRONG: Must close Transform block
                     # first
TransformEnd          # These are in the wrong order
```

Detecting improper nesting is trivial with the mode stack. RiAttributeEnd() simply confirms that the current state is attribute before popping the mode stack. RiTransformEnd() looks for transform mode, and so on. If the current mode does not match, the error handler is called with the error code RIE_NESTING.

Functions that set RenderMan options are only allowed while in base or frame mode. Attributes may be set in any mode except object. Primitives may be specified while in world, attribute, transform, solid, object or motion modes. Violations of these three rules cause the error handler to be called with codes RIE_NOTOPTIONS, RIE_NOTATTRIBS, and RIE_NOTPRIMS, respectively. Attempts to call functions while in the <undefined> mode result in the RIE_NOTSTARTED error code. Most other mode-related errors fall under the RIE_ILLSTATE catch-all. For a complete (but sometimes inaccurate) account of which functions are allowed in each mode, see the table at the end of Chapter 3 in *The RenderMan Companion* (Upstill, 1990).

While the interface can be in only one mode at a time, there are often two or more acceptable states for each function in the API. It is convenient to create a class called ModeSet to represent a set of modes. Then we can easily verify if the current mode is in the set of acceptable modes. The stack need only contain a single mode, so it can be declared using C++ templates as:

```
Stack<Mode> modeStack;
```

2.2.2 The Option Stack

State variables such as raster resolution which apply to the whole scene and not to individual primitives are called *options* in RenderMan. Options are fixed before `RiWorldBegin()` and remain in effect throughout rendering. At first glance it might seem that an option stack is unnecessary and that all options could be implemented as global variables. Upon closer examination of the RenderMan Interface specification it becomes clear that `RiFrameBegin()` preserves the options and `RiFrameEnd()` restores them. Since frame blocks cannot be nested a stack of size two will suffice (a backup and restore model would also be viable).

Unlike a mode, the set of all options cannot be represented by a single integer. Each element in the option stack is an aggregation of variables. By defining a suitable class all of the RenderMan options can be bundled into a convenient package:

```
class Options {
public:
    Options();
    Options(const Options &options);
    ~Options();
    Options &operator= (const Options &options);

    // Camera options
    RtInt xRes, yRes, pixelAspectRatio;      // RiFormat
    RtFloat frameAspectRatio;               // RiFrameAspectRatio
    RtFloat left, right, bottom, top;       // RiScreenWindow
    RtFloat xMin, xMax, yMin, yMax;        // RiCropWindow
    RtToken projection;                    // RiProjection
    RtFloat nearClip, farClip;             // RiClipping
    RtFloat fStop, focalLength,            // RiDepthOfField
        focalDistance;
    RtFloat openShutter, closeShutter;     // RiShutter

    // Display options
    .
    .
    .
};
```

When the mode stack is pushed, a new value is stored at the top of the stack and the current contents need not be examined. In this respect, the mode stack resembles a traditional stack from computer architecture or a container library. In the case of the option stack, there is no new set of values to store. When `RiFrameBegin()` is called the top of stack is simply duplicated to preserve the old values, without actually changing them.

The push operation on the option stack is therefore defined as follows:

- take the options object currently at the top of the stack,
- copy it,
- put the copy on the top of the stack.

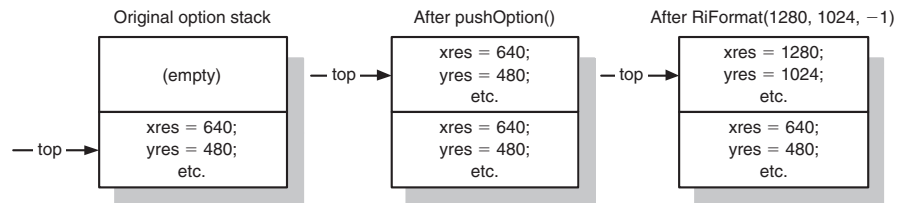


Figure 2.2 The “copy on push” semantics of the option stack.

For the moment, the top two elements of the stack are identical. Subsequent calls to functions such as `RiFormat()` will only manipulate the options at the top of the stack, as depicted graphically in Figure 2.2. The push serves to protect the options at the bottom of the stack until `RiFrameEnd()`. Because the copy-on-push semantics are non-standard, the option stack must be implemented as a custom class that takes care of the copy in the push method:

```
CopyStack<Options> optionStack;
```

2.2.3 The Attribute Stack

In *RenderMan*, *attributes* are state variables such as colour and opacity that can be changed after `RiWorldBegin()`, when the user is defining primitives. Each primitive can have its own set of attributes, distinct from all of the other primitives. Whatever the prevailing values of the attributes are at the time the primitive is added to the scene remain associated with that primitive until rendering is complete.

The requirements for the attribute stack are much the same as for the option stack except that attribute blocks can be nested arbitrarily deep. A stack of size two is therefore inadequate. The attribute stack must be able to grow unbounded. Again, for convenience, all of the attributes are grouped into a single class:

```
class Attributes {
public:
    Attributes();
    Attributes(const Attributes &attributes);
    ~Attributes();
    Attributes &operator= (const Attributes &attributes);

    // Geometry attributes
    RtBound bound;                // RiBound
    RtBound detail;              // RiDetail
    RtFloat minVisible, lowerTransition, // RiDetailRange
        upperTransition, maxVisible;
    RtToken typeApproximation;    // RiGeometricApproximation
    RtFloat valueApproximation;
```

```

    RtToken orientation;           // RiOrientation
    RtInt sides;                  // RiSides

    // Shading attributes
    .
    .
    .
};

```

The attribute stack also uses copy-on-push to protect lower values in the stack without changing any of the current values of the attributes.

```
CopyStack<Attributes> attributeStack;
```

`RiAttributeBegin()` pushes the attribute stack, as do `RiFrameBegin()`, `RiWorldBegin()`, `RiSolidBegin()` and `RiObjectBegin()`. All of the corresponding End functions pop the attribute stack.

For *immediate mode* renderers (those that render each piece of geometry as it is passed through the API) the stack data structure is sufficient. Any stage of the pipeline can simply check the top of the stack and know that it applies to the primitive currently being processed.

Production renderers are more often *retained mode*, where geometry is stored in some sort of scene database and rendering does not begin until the entire scene has been defined. Retained mode renderers cannot check the attribute stack while rendering because the stack is only valid during the scene description stage. Instead, primitives must be tagged with the prevailing top-of-stack attributes as they are added to the scene database. These attributes must be stored in memory that will not be destroyed when the attribute stack is popped.

Since many primitives may share a set of identical attributes, a little bit of ingenuity can go a long way to reduce the demand on memory. The first primitive added to the scene will make a copy of the current attributes object. Within the stack, the pointer to that copy is cached. When the next primitive is added, it can be given the previously stored pointer and a reference count incremented. The same pointer may be reused until one of the attributes changes, at which time the cache is invalidated. The next primitive will get a fresh copy. An attributes object at the top of the stack that is never picked up by a primitive need not be copied to long-term memory.

Sometimes modelling programs generate RIB that alternates between two sets of attributes (or rotates among several) foiling the single cache strategy above. To make best use of memory in these cases, insert the durable copies of attributes objects into a hash table indexed by a digest (or checksum) of the individual attribute values in the object.

2.2.4 The Transformation Stack

A stack data structure of 4×4 matrices neatly captures the hierarchical coordinate systems described in Chapter 1. Like the option and attribute stacks, the transformation stack must have copy-on-push semantics.

```
CopyStack<Transform> transformStack;
```

Calls to `RiTranslate()`, `RiScale()` and `RiRotate()` only affect the matrix at the top of the stack. Like attributes, retained-mode renderers should tag primitives with the top-of-stack matrix as they are added to the scene database. The previous discussion on caching attributes pointers applies equally to transformations. In fact, the object-to-world transformation matrix can be regarded as simply another `RenderMan` attribute. However, transformations are treated separately from all of the other attributes so that the transformation stack can be pushed and popped without affecting the attribute stack (i.e. `RiTransformBegin()` only pushes the transformation stack). The converse is not true. `RiAttributeBegin()` pushes both the transformation stack and the attribute stack, as do `RiFrameBegin()`, `RiWorldBegin()`, `RiSolidBegin()` and `RiObjectBegin()`.

Pay attention to whether transformations are in *Utah* order, with points expressed as rows and multiplied by matrices on the right. If so, then the top of the stack should be pre-multiplied (i.e. on the left) by new transformations. If points appear as columns to the right of matrices, then the stack must follow a post-multiply discipline.

For the purpose of transformational motion blur, it may be advantageous to maintain two (or more) transformation stacks, one for each motion sample. For example, the primary transformation stack could represent the object-to-world matrix at shutter open time while a second stack tracks the same information for shutter close time.

```
CopyStack<Transform> transformOpenStack;  
CopyStack<Transform> transformCloseStack;
```

2.2.5 The Object and Light Stacks

Objects are created with `RiObjectBegin()` and light sources are created with `RiLightSource()` or `RiAreaLightSource()`. These functions return opaque data handles for future reference (in the case of RIB, the user must provide a unique identifier). With few exceptions, objects and lights can be created at any time between `RiBegin` and `RiEnd`.

Object and light handles are scoped by both frame blocks and world blocks. Handles created during the world block go out of scope upon `RiWorldEnd()` – likewise for the frame block and `RiFrameEnd()`. In order to correctly scope these handles, create a stack where each element consists of a set of object and light handles. An empty set is pushed onto the stack by `RiFrameBegin()` and `RiWorldBegin()` (there is no need for a copy-on-push).

```
Stack<ObjectsAndLights> objectLightStack;
```

Whenever an object or light handle is created, it is added to the set at the top of the stack. Upon `RiFrameEnd()` or `RiWorldEnd()`, free the memory associated with the objects and lights in the set and pop the stack.

The world block may be contained within a frame block, but otherwise frame and world blocks cannot be nested. Therefore a stack of size three is sufficient (remember, objects can be defined in base mode prior to `RiFrameBegin()`).

2.2.6 The State Object

Functions in the RenderMan API that begin and end blocks are redirected to the state object. All of the API functions use the state object to verify proper mode prior to carrying out their orders. Functions that modify options, attributes or transformations gain access to the top of the appropriate stack here. As primitives are being added to the scene graph, they can be annotated with a dynamically allocated copy of the top-of-stack attributes and transformations.

```
class State {
// Singleton design pattern (Gamma et al., 1995),
public:
    static State *Instance();
protected:
    State();
    ~State();
private:
    static State *instance;
public:
    // Changing states
    RtVoid begin(RtToken name);          RtVoid end();
    RtVoid frameBegin(RtInt frame);      RtVoid frameEnd();
    RtVoid worldBegin();                RtVoid worldEnd();
    RtVoid attributeBegin();            RtVoid attributeEnd();
    RtVoid transformBegin();            RtVoid transformEnd();
    RtVoid solidBegin(RtToken operation); RtVoid solidEnd();
    RtVoid motionBegin(RtInt n, RtFloat times[]);
                                           RtVoid motionEnd();

    RtObjectHandle objectBegin();        RtVoid objectEnd();
    RtVoid addLight(RtLightHandle newLight);

    // Verify current mode is in set of acceptable modes.
    // If not, throw error code and return false.
    bool verifyMode(ModeSet allowableModes,
        RtInt errnumIfDifferent);

    // Accessing tops of stacks
    Options &topOptions();
    Attributes &topAttributes();
    Transform &topTransformOpen();
    Transform &topTransformClose();
    // Get durable copy of top-of-stack attributes or transform
    Attributes *cloneAttributes();
    Transform *cloneTransformOpen();
    Transform *cloneTransformClose();
}
```



```

    // Add constructed primitive to object, light, blur, or
    // framework.
    void insert(Primitive *prim);

protected:
    Stack<Mode> modeStack;
    CopyStack<Options> optionStack;
    CopyStack<Attributes> attributeStack;
    CopyStack<Transform> transformOpenStack, transformCloseStack;
    Stack<ObjectsAndLights> objectLightStack;

    // Internal methods called by begin and end methods above.
    // For example, frameBegin calls all of the push functions.
    void pushMode(Mode m);           void popMode();
    void pushOption();               void popOption();
    void pushAttrib();               void popAttrib();
    void pushTransform();            void popTransform();
    void pushObjectLight();          void popObjectLight();

    //
    // In object mode, motion mode, or when defining an area light,
    // gather primitives here. Otherwise, these pointers will all
    // be NULL and the primitive will pass through to the
    // Framework.
    //
    RtObjectHandle openObject;
    RtLightHandle openAreaLight;
    BlurredPrimitive *openBlurPrim;
};

```

The `State` class has a member function called `insert()` that takes a pointer to a `Primitive`. `Primitive` is the root class for all geometric primitives such as `Polygon`, `Sphere`, and `NuPatch`, and will be considered in a later section. Functions in the API that define geometric primitives should instantiate a class derived from `Primitive` and pass it to the `State` object for insertion into the scene graph. For example, `RiPolygonV()` might be implemented as follows:

```

RtVoid RiPolygonV(RtInt nverts,
    RtInt n, RtToken tokens[], RtPointer parms[])
{
    State *state = State::Instance();
    Primitive *prim = new Polygon(nverts, n, tokens, parms);
    state->insert(prim);
}

```

The `State` object will normally just forward the primitive to the framework for rendering, but there are exceptions where it will gather up individual primitives to build a composite of some sort.

If the current mode is `object` (i.e. after `RiObjectBegin()` and before `RiObjectEnd()`) then the new primitive will be appended to the currently open object. The object itself is an aggregate primitive: a primitive made up of other primitives. It subsequently may be inserted into the hider one or more times, or it may never get inserted at all, in which case the component primitives will not be rendered.

In the case of an area light source, primitives are used to define the shape of the light. They are not intended to be rendered as objects. The `State` class is responsible for creating the handle to the area light, which it caches internally. Subsequent primitives are added to the area light definition until `RiAttributeEnd()` is called. At that point, the cached handle is cleared and the `State` object returns to the mode of passing primitives through to the framework.

The final exception is the motion mode. Inside a motion block, two or more poses of a deforming primitive can be defined. If each pose were simply forwarded to the framework then they would be rendered as individual unblurred objects. It is the `State` object's responsibility to collect the poses and then insert a single deforming primitive into the framework upon `RiMotionEnd()`.

The `insert()` member function just needs to handle the special cases and pass all other primitives through to the framework:

```
void State::insert(Primitive *prim)
{
    if (openObject != NULL)
        append prim to openObject
    else if (openAreaLight != NULL)
        append prim to openAreaLight
    else if (openBlurPrim != NULL)
        append prim to openBlurPrim
    else
        insert prim into Framework
}
```

A straightforward way to represent objects, area lights or blurred primitives is to derive a class from `Primitive` and give it a member variable that is a set of other `Primitives`. For example:

```
class BlurredPrimitive : public Primitive {
protected:
    vector<Primitive *> poses; // Deforming primitive keyframes
public:
    void append(Primitive *p) { poses.push_back(p); }
};
```

2.3 Micropolygon Architectures

Most modern production renderers share what could be called a *micropolygon architecture*. The traditional definition of a micropolygon is a “flat-shaded sub-pixel

quadrilateral” (Cook et al., 1987) but all three aspects of that definition have been challenged by various implementations over the years:

- Shading is still performed at every vertex of a micropolygon, but the interior is often Gouraud shaded instead of flat shaded. Only in the case of ray tracing are micropolygons shaded at points in their interior.
- When projected to raster space, micropolygons are often the size of a pixel. The requirement is that micropolygons be small enough that their shape is not discernible in the image, even after stretching due to displacement mapping. Abandoned is the notion that micropolygons with edges half the width of a pixel will satisfy the Nyquist limit and somehow eliminate shader aliasing.
- The traditional four-sided micropolygon need not be planar, so the name “micro-bilinear-patch” would be more accurate but harder to communicate. The name “micropolygon” has stuck. A particular renderer may choose to use micropolygons that have three sides but it must be consistent in that choice.

Users of a renderer do not create micropolygons directly. Instead, they specify higher-order geometric primitives that are often curved, such as NURBS or subdivision surfaces. Partway down the pipeline, the renderer converts these higher-order primitives into a mesh of micropolygons through a process known as *dicing*. Dicing is a particular kind of tessellating that is done adaptively within the renderer and results in micropolygons, not multiple-pixel facets as are found in low polygon count models for real-time graphics. The broader term *tessellating* refers to any process that converts a curved surface to a polygonal representation, including what is done in a modelling program to prepare the geometry for a polygon-oriented renderer. Polygons as a geometric primitive are discouraged in production rendering for several reasons:

- Information about the underlying curved surface has already been lost.
- The coarseness of tessellation must be decided by the user or modelling program.
- Tessellation that is constant in parametric space may result in facets that are very large near the camera and very small in the background.

Dicing, on the other hand, is always an adaptive process done by the renderer, where more information about how many pixels the surface will cover is available. Parts of a primitive near the camera will be diced more finely than those in the background to maintain the approximate size of one micropolygon per pixel.

Another term related to micropolygons and dicing is *grid*. A grid is simply a two-dimensional array (or *mesh*) of micropolygons. Just as you might wash your car in sections, dicing works on a section of a geometric primitive to create a grid of micropolygons. The grid representation is more compact than storing micropolygons independently because data at the interior vertices is shared.

2.3.1 Advantages

Micropolygons serve as a standard data format to isolate the primitives from the latter stages of the Reyes pipeline, such as shading and hiding. Having a common

data structure allows these complex stages of the pipeline to be written for a single geometry type, and not be further complicated by having to directly support every available geometric primitive. Adding a new primitive to the renderer is also simplified. It just needs to know how to convert itself into micropolygons.

Because all of the micropolygons in a grid are the same size in parametric (or *uv*) space, parametric surfaces may take advantage of subdivision coherence by applying fast forward differencing algorithms. The decision of whether to optimize primitives in this way can be made on a case-by-case basis, as the code will be isolated to the `dice()` function of each primitive.

The most obvious advantage of a micropolygon architecture is the ease with which it can support displacement mapping, a feature that has become popular in production renderers. Since the geometry is already diced into pixel-sized elements, it is relatively straightforward to move each vertex in a direction and distance prescribed by a procedural displacement shader. To distinguish it from lower quality approximations, micropolygon displacement is often called “true sub-pixel displacement”: “true” because the surface is actually deformed and not just the shading normal, and “sub-pixel” because micropolygons are usually the size of a pixel or smaller.

Shading performance is improved by keeping unshaded micropolygons in their grid form. Being able to shade a surface at the same time is an example of *vectorization*, where similar calculations are performed together. Vectorized shading amortizes various costs associated with shading over the entire grid, rather than repeating the work for every vertex that gets shaded. Shading one grid at a time also improves the performance of accessing texture files. The vertices in a grid will usually access contiguous parts of the same texture maps, reducing the number of texture files that need to be open at one time and maximizing cache performance by obeying locality of reference. Since micropolygons are often aligned with the axes of texture coordinates, the need for run-time texture filtering is reduced.

Micropolygons overcome two difficulties associated with perspective projections. If the micropolygons are small in pixel space (as they are intended to be), then the perspective distortion of interpolated values is negligible and it is unnecessary to correct for it. Because micropolygons are shaded in world or camera space before the perspective projection, there is no need for an inverse perspective transformation.

2.4 Reyes Pipeline

Reyes¹ is the original micropolygon framework and remains a popular choice today. Loren Carpenter wrote the Reyes “prototype” renderer in 1981. He later

¹Reyes is named after Point Reyes in California and is also an acronym for Renders Everything You Ever Saw. While it has become popular to use all capitals to emphasize REYES as an acronym, we will use the traditional capitalization from the 1987 paper, “The Reyes Image Rendering Architecture”. Unfortunately, Reyes is pronounced exactly the same as “rays”.

recast Reyes as a test-bed for the research and development of hiders. Rob Cook's stochastic sampler (Cook et al., 1984), Ed Catmull's analytic hider (Catmull, 1984), and Carpenter's A-buffer (Carpenter, 1984) were all developed under the Reyes test-bed. By the time Cook, Carpenter and Catmull published their paper (Cook et al., 1987), they had taken a preference for Cook's stochastic hider and presented it as part of the architecture along with the Reyes framework.

This text will use the definition of Reyes as a framework (or test-bed) independent of the choice of hider. A hider takes shaded micropolygons as input and produces pixels as output. The hider could be a Z-buffer, super sampler, stochastic sampler, A-buffer, analytic hider, ray tracer, or some other technique. Reyes will interface to the hider through an abstract interface. How a hider uses pixel filters and whether or not it resamples are independent of the Reyes process.

The five main operations of the Reyes framework are bound, split, dice, shade and hide. If a renderer does all of those steps in that order, then it is certainly a Reyes renderer. In performing an object-oriented analysis of Reyes, it is most important to understand the data types which flow from one stage of the pipeline to the next. In Figure 2.3 the process blocks will be implemented by methods and the data types passed between them will be objects.

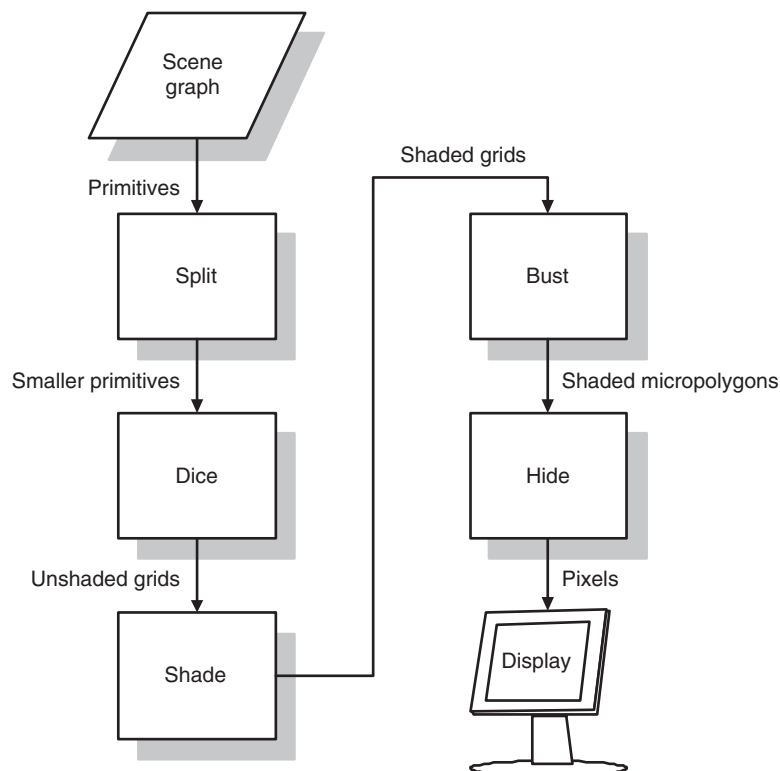


Figure 2.3 Reyes data flow.

2.4.1 Bound

Early in the Reyes pipeline, a bounding volume is calculated for primitives. Some primitives such as infinite planes may not be boundable at all. The usual solution for this problem is to simply not allow unboundable primitives in a Reyes renderer. A more flexible solution is to ask each primitive whether it is boundable or not and only calculate a bound for those that are finite. Unbounded primitives can be processed by the Reyes framework, but not nearly as elegantly and efficiently as those with well-defined bounding volumes.

2.4.2 Split

Splitting is an example of a divide and conquer approach. To split a primitive is to reduce it to one or more simpler primitives. The straightforward way to split a parametric surface such as a bicubic patch is to cut it along its centre parametric lines (where $u = 0.5$ and $v = 0.5$) into four smaller bicubic patches. An aggregate primitive, such as a set of polygonal faces, can be directly split into individual objects or repartitioned into two or more smaller sets.

The primitives that result from a split need not be the same type as the original, unsplit primitive. For example, a sphere might be split into NURBS patches. Ideally the new primitives will represent exactly the same surface as the original. However, it is sometimes acceptable to approximate the original surface. Because bicubic patches cannot accurately represent quadric surfaces, splitting a sphere into bicubic patches would be an approximation but might be close enough. Fractals and other procedural geometry may have infinite detail in theory, but eventually they must be split into finite surfaces if they are to be rendered.

The requirements of the split operation are that the child primitives stay within the parent's bound, and that splitting must eventually lead to primitives that are diceable.

2.4.3 Dice

Armed with good split routines, we could design a renderer that just keeps splitting until the primitives are as small as a pixel and then replace each with a single micropolygon. That is essentially the way the renderer described in Ed Catmull's thesis (Catmull, 1974) worked. However, splitting quickly reaches a point of diminishing returns. This is often the case with divide and conquer algorithms. Consider the case of quicksort. It can be written to partition the input down to single elements but in practice a lightweight sorting algorithm (such as insertion sort) is called upon once quicksort gets down to sets of seven items or so. The overhead of partitioning begins to dominate. Sometimes you just have to stop dividing and start conquering!

With Reyes, conquering comes in the form of dicing. Dicing is the process of converting a high-order primitive into a grid of micropolygons. The process is driven by the natural coordinate system for the primitive, such as uv coordinates for parametric primitives. Vertices generated by dicing will initially be in world or camera space. Raster space is only used for an estimate of how finely to dice in order to maintain a micropolygon size of about one pixel.

2.4.4 Shade

The result of dicing is an unshaded grid. The role of the shading system is to turn that unshaded grid into a shaded grid. Shading occurs only at the vertices of the micropolygon grid. Micropolygon interiors may either be flat shaded (by taking the shade at the upper left vertex) or Gouraud shaded by interpolating the shading results at the four vertices.

Shading may be done in world or camera coordinates but it must not be done in screen or raster space. Angles between vectors are important in shading. For example, the angle between the vector to a light source and the normal of a surface is used in diffuse shading. The perspective transformation does not preserve angles between vectors. Therefore, shading in screen space would yield incorrect results. Inverse transforming vectors back to camera space while shading is awkward. It is more natural and efficient to shade in world or camera coordinates.

All of the input variables to the shading system must be available or derivable from data in the unshaded grid. Add user-programmable shaders, and this requirement places a hefty demand for flexibility in the data structure design. The most important outputs from the shading system are colour and opacity (or, in the case of displacement shaders, position and normal). Production renderers often support final output of quantities other than colour and opacity, such as arbitrary variables defined in shaders. Again, this requires a flexible data structure.

Displacement shaders are evaluated before all other shaders. To support ray tracing, displacement shading should be separated and called independently by the framework. The result of displacement shading is a displaced but otherwise unshaded grid.

2.4.5 Hide

Computer graphics pioneers named a procedure for figuring out what geometry was visible and what was hidden behind other geometry a “hidden surface algorithm”. Apparently inspired by a glass-is-half-full notion, optimists relabelled the process “visible surface determination”. We prefer the terse name *hider*. A production renderer might provide several hidere and give the user the option of selecting one based on tradeoffs such as speed versus quality.

Hidere tend to have memory requirements that are proportional to the number of pixels in the raster. For example, a Z-buffer hider might store colour, alpha and depth for each pixel. That is a manageable amount of memory, but more sophisticated hidere store a list of micropolygons (or micropolygon fragments) per supersample. When hider memory requirements become intractable, the solution is to divide the raster up into bite-sized chunks called *buckets*. Buckets can be scanlines or rectangular areas of pixels. The idea is to finish processing one bucket in its entirety, then reuse the hider memory to process the next bucket. If a hider needs to use buckets, it must communicate that fact and the desired bucket size back to the framework so that it can cooperate in processing the scene in bucket order.

The input to a hider is a grid of micropolygons and the output is pixels. The hider does not return the pixels to the framework. Instead, it communicates

directly with the display driver(s) to output the pixels to a framebuffer or graphics file.

2.5 Primitives

A retained mode renderer must store all primitives in a scene graph before rendering begins. Therefore, the top-level representation of primitives should be as concise as possible to conserve resources. The representation passed across the API is rather concise, so it is hard to go wrong by allocating space and copying the data verbatim. As primitives are processed by a rendering pipeline, they will generally be expanded into a more memory-intensive representation and then discarded (or at least kept in a geometry cache of a limited size).

As of revision 3.2, the RenderMan specification defined 20 primitive types (including procedural primitives and instanced objects). The early stages of a rendering framework manipulate these top-level primitives. Clearly this is a good place to use a base class, rather than hardcoding 20 cases throughout the code and needing to modify each occurrence to support whatever primitives may be required in the future.

Primitives that support the following operations will fit easily into a variety of frameworks:

```
class Primitive {
public:
    virtual bool boundable();
    virtual Bound bound();
    virtual bool splitable();
    virtual void split(Framework &f, bool usplit, bool vsplit);
    virtual bool diceable(MicroPolygonGrid &g, Hider &h, bool
        &usplit, bool &vsplit);
    virtual void dice(MicroPolygonGrid &g);
protected:
    virtual Point evalP(float ugrid, float vgrid);
    virtual Vector evalDPdu(float ugrid, float vgrid);
    virtual Vector evalDPdv(float ugrid, float vgrid);
};
```

2.5.1 Bounding

The `boundable()` function returns `true` if the primitive can be bounded, and `bound()` returns a bounding volume in camera coordinates. The framework will not call `bound()` unless `boundable()` returns `true`. Most primitives should be boundable except things like infinite planes or half-spaces.

The typical choice of bounding volume is a box aligned with the axes of camera space. While it is permissible to bound in world or object coordinates, camera coordinates are preferable because they maximize the probability that the primitive can be trivially rejected in a frustum or occlusion culling step.

For correct results the bound must completely contain the primitive, including displacements. If motion blur is in effect, the bound must contain the primitive at all positions along its motion path while the shutter is open. If the primitive is split, then all of the child primitives must also be contained in the bound.

The good news is that the bound does not need to be perfectly tight. Bounds are used for culling and for diceability estimates, so a loose bound may hurt performance but at least it will not make the image incorrect. Many primitives obey the convex-hull property, which means a bound that contains all of the control vertices will also contain the surface. If a primitive does not have the convex-hull property, it can often be converted to an equivalent surface that does.

2.5.2 Splitting

`Splitable()` returns `true` if the primitive can be reduced to simpler primitives. It is a good idea to ensure that all primitives are splitable. A flexible framework will ask a primitive if it is splitable before telling it to split itself. In theory, a primitive that is always diceable need not be splitable, but there are rare occasions when the framework must split a primitive or discard it (even if it is visible).

`split()` creates one or more primitives and inserts them back into the scene graph by calling `f.insert(newprim)`. Parameters `usplit` and `vsplit` are hints on whether to split parametric primitives in the u dimension, v dimension or both. The framework will not call `split()` if `splitable()` returns `false`. After calling `prim->split()`, the framework will delete `prim`.

2.5.3 Dicing

`diceable()` returns `true` if the primitive can be diced into a micropolygon grid. It can ask the hider for hints such as desired micropolygon size, maximum number of micropolygons per grid, how finely an example camera-space line or bound should be diced, etc. If it returns `false`, `diceable()` must set `usplit` and `vsplit` to recommend how the primitive should be split. If it returns `true`, it must set `g.xdim` and `g.ydim` to the recommended grid dimensions.

`dice()` creates an unshaded grid of micropolygons. The dimensions are passed in as `g.xdim` and `g.ydim`. `dice()` fills the points of the grid in camera coordinates. The base class implementation of `dice()` can be something like:

```
Primitive::dice(MicroPolygonGrid &g)
{
    for vgrid = 0 to 1 step 1.0/g.ydim
        for ugrid = 0 to 1 step 1.0/g.xdim
            add evalP(ugrid, vgrid) to g
}
```

Derived classes that do not support `evalP()` must override `dice()`. Other classes may choose to override `dice()` to implement faster algorithms such as forward differencing.

`evalP()` takes parameters `ugrid` and `vgrid`, which range from 0 to 1 over a single grid (as opposed to the entire surface). It returns the point P on the surface at parametric coordinates `ugrid` and `vgrid`.

`evaldPdu()` and `evaldPdv()` return partial derivatives dP/du and dP/dv at coordinates `ugrid` and `vgrid`. Ideally, these are derived analytically rather than through finite differentials. An analytic normal can be obtained from the cross product of `dPdu` and `dPdv`.

2.6 Grids

A grid's lifespan starts when the `diceable()` function sets its dimensions and ends when the `hider` busts it into individual micropolygons. In between, dicing fills in the values of vertex variables such as the position P at each point in the grid. It may get trimmed and displaced prior to being shaded, which sets final values for colour and opacity (C_i and O_i).

```
class MicroPolygonGrid {
public:
    MicroPolygonGrid();
    ~MicroPolygonGrid();

    int xdim, ydim; // Grid dimensions
    int nverts; // Total vertices = (xdim+1) * (ydim+1)

    void addVariable(RtToken name, RtPointer value, int type,
        int detail);
    RtPointer findVariable(RtToken name, int type, int detail);
    bool isbackfacing();

    bool trim();
    void displace();
    void shade();
    MicroPolygon *bust();

protected:
    const Primitive *parent; // Primitive from which we came
    TokenValueList vertexvars;
};
```

Grid dimensions are so frequently accessed that they are shown here as public member variables. `Primitive`'s `diceable()` function sets them and also sets the total number of vertices as a convenience. Subsystems such as the shader evaluator usually just want to know how big the grid is, but SL functions like `calculateNormal()` need to distinguish the grid's width from its height.

A micropolygon grid only becomes really usable when the `dice()` function sets the values of the vertex variables. It stores these in the private `vertexvars` member variable by calling the `addVariable()` function. The position P will be always be diced. If motion blur is in effect, another position at shutter close time `Pclose` will also be added. If the user declares reference geometry as a vertex

variable `Pref`, that should be diced as well, and so on. Uniform variables should not be stored in `vertexvars` because it would just contain redundant values. Varying variables can be stored here, but can also be interpolated on the fly from u , v , and the values at the four corners of the grid.

The `isbackfacing()` method returns `true` if every micropolygon in the grid faces away from the camera. However, it must always return `false` if backface culling was turned off for this primitive with `RiSides(2)`. The `Sides` attribute is available through `parent`, which points to the primitive that was diced to create this grid.

The `trim()` member function actually applies trim curves to the grid. Again, the trim curves are attributes and can be accessed through the parent pointer. If the entire grid gets trimmed away, then `trim()` will return `true`.

Grids know how to displace and shade themselves by calling the shader evaluator. Displacement shading is separated from the others (light, surface and atmosphere) to accommodate ray tracing hiders that wish to intersect displaced but otherwise unshaded grids.

Finally, the `bust()` member function can be called to break up the grid into individual micropolygons. Although the grid representation is more compact, individual micropolygons can actually save memory if most are consumed by the current bucket and only a handful are forwarded to other buckets for further processing.

2.7 Shader Evaluator

The shader evaluator converts unshaded grids to shaded grids by executing shader code that has been preprocessed offline by the shader compiler. By some estimates, shading accounts for half of the design complexity and 90% of the CPU cycles in a production renderer (and two chapters of this book). Fortunately, the interface to the shading subsystem can be summed up in two functions:

```
SIShader *loadShader(const char *shadername);
void runShader(SIShaderInstance *shader, MicroPolygonGrid
              *grid);
```

where `SIShaderInstance` is defined as:

```
class SIShaderInstance {
    SIShader *theShader;
    TokenValueList ParameterValues;
}
```

The `loadShader()` function is called as a result of one of the shader functions in the API such as `RiSurface()`, `RiDisplacement()` or `RiLightSource()`. It returns a pointer to an `SIShader` that represents only the shader code, and no data. This should be augmented by the token-value list that was passed across the `RenderMan` API, and stored with the attributes as an `SIShaderInstance`.

Later, the `displace()` and `shade()` member functions of the `MicroPolygonGrid` class will call `runShader()`. They will retrieve the appropriate `SShaderInstance` from the attributes of the parent primitive and pass this as the second argument.

The shader evaluator will turn around and access vertex variables in the grid by calling `findVariable()`. It must have read and write access to the vertex variables because displacement shaders update `P` and `N` while surface shaders set `Ci` and `Oi`.

2.8 Micropolygons

Once a grid is shaded it may get busted into micropolygons.

```
class MicroPolygon {
public:
    MicroPolygon(MicroPolygonVertex *v0, MicroPolygonVertex *v1,
                MicroPolygonVertex *v2, MicroPolygonVertex *v3);
    MicroPolygon(const MicroPolygon &mp);
    ~MicroPolygon();
    VertexProxy v[4];                // Four corners
    int xmin, xmax, ymin, ymax;      // Bucket coordinates
    float zmin, zmax;                // Camera coordinates
    // Add renderer-specific fields here
};
```

Each micropolygon has four vertices, represented by the `VertexProxy` class described below. The `x` and `y` bounds of the micropolygon are specified in bucket coordinates – not pixel coordinates within each bucket but the position of the bucket itself within the 2D array of buckets. This allows the hider to quickly determine which buckets the micropolygon overlaps and whether it is necessary to forward it to another bucket. The `z` bounds are simply in camera coordinates.

Any number of renderer-specific fields can be added after the bounds. A link back to the primitive which led to this micropolygon is possible, but it may be preferable for micropolygons to be self-contained. Therefore, flags such as matte object, smooth interpolation, and backface culling should be copied to the micropolygon when it is created. If micropolygons are to be trimmed, then information regarding trim curves that intersect the micropolygon should be stored here as well.

2.8.1 Shaded Vertices

Micropolygons that have already been shaded do not need to carry the full complement of vertex variables that were present in the original grid for the benefit of the shader evaluator. All that is necessary are those values that the hider needs in order to do its job of colouring each pixel.

```

class MicroPolygonVertex {
public:
    MicroPolygonVertex();
    virtual ~MicroPolygonVertex();

    Point P;
    int referenceCount;           // Used by VertexProxy
};

class ShadedVertex : public MicroPolygonVertex {
public:
    ShadedVertex();
    virtual ~ShadedVertex();

    Point Pclose;
    Colour Ci;
    Colour Oi;
    // Add arbitrary output variables here
};

```

Pclose is here because it is the hider's job to do motion blur. If the micropolygon is not blurred then Pclose will equal P from the base class (or a flag can be cleared in the MicroPolygon class indicating that there is no motion blur). Obviously, colour and opacity must be provided to the hider for it to resolve hidden surfaces and colour the pixels.

Supporting arbitrary output variables really requires a more complex design. A way must be found to identify them at run time, store them with each shaded vertex, and have the hider create multiple display outputs.

2.8.2 Unshaded Vertices

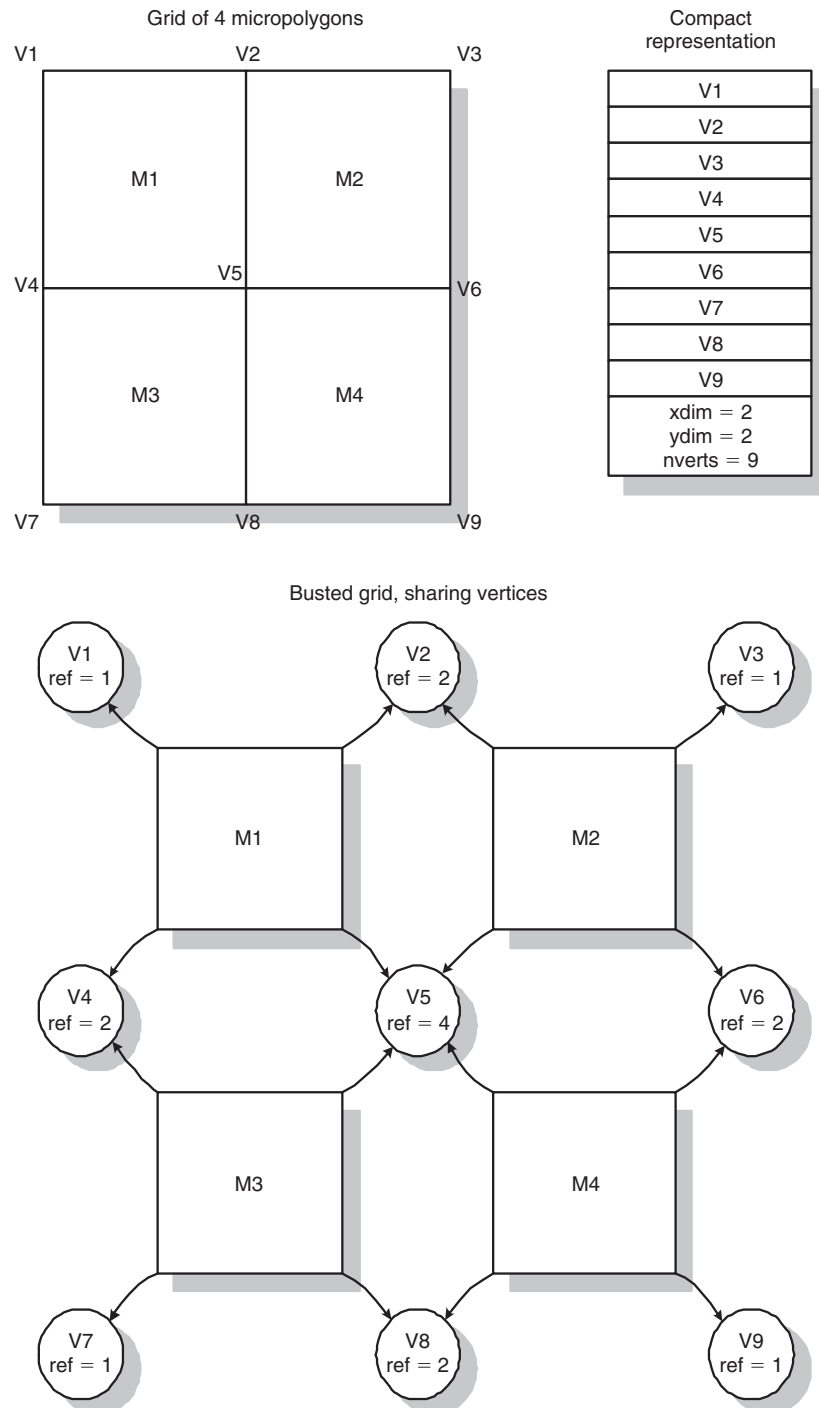
Ray tracing hidiers may wish to displace and bust grids, intersecting the resulting unshaded micropolygons. Since shading happens after the point of intersection has been determined, unshaded micropolygons must carry all of the inputs to the surface shader in their vertices.

```

class UnshadedVertex : public MicroPolygonVertex {
public:
    UnshadedVertex();
    virtual ~UnshadedVertex();
    Vector N;
    Vector dPdu, dPdv;
    // Add surface shader parameters here
};

```

The set of shader parameters to include must be determined at run time because it depends on the choice of surface shader and may vary from primitive to primitive.

**Figure 2.4** Micropolygon representations.

2.8.3 Vertex Proxy

The increase in resource requirements that comes with busting a grid can be somewhat mitigated by sharing `MicroPolygonVertex` objects between adjacent micropolygons.

```
// Proxy design pattern for smart reference (Gamma et al., 1995)
class VertexProxy {
public:
    VertexProxy(MicroPolygonVertex *v);
    ~VertexProxy();

    MicroPolygonVertex *operator->();
    MicroPolygonVertex &operator*();

private:
    MicroPolygonVertex *barePointer;
};
```

The `VertexProxy` class acts as a *smart pointer* (Edelson, 1992), counting references and deleting the actual `MicroPolygonVertex` object when all `MicroPolygon` objects that shared it go out of scope. Figure 2.4 shows how busted micropolygons may share vertices. Operator overloading allows the standard `->` pointer syntax to be used. `Grid`'s `bust()` method sets all of this up by creating one `MicroPolygonVertex` for every vertex in the grid, and then constructing `MicroPolygon`'s from sets of four. The constructor for `MicroPolygon` creates four `VertexProxy` objects from them.

2.9 Hiders

The primary function of a hider is to take grids of shaded micropolygons, determine visible surfaces, and output pixels. It also manages the scene graph and may perform occlusion culling. The choice of scene graph data structure is up to the hider. Ray trace hiders will generally use a 3D spatial partitioning data structure such as a BSP tree, octtree, kd-tree or 3D voxel array. Other hiders often use a 2D array of buckets.

A production renderer may support many hiders and allow the user to select one at run time (although the ray trace framework must be paired with a ray trace hider). Hiders may be immediate or retained mode. Examples of hiders include Z-buffer, super sampler, stochastic sampler, A-buffer, analytic, ray tracer and scan converter.

```
class Hider {
public:
    virtual void worldBegin();
    virtual void insert(Primitive *);
    virtual void remove(const Primitive *);
```

```

virtual bool bucketBegin();
virtual Primitive *firstPrim();
virtual void hide(MicroPolygonGrid &g);
virtual Colour trace(Point p, Vector r);
virtual void bucketEnd();

virtual bool isRasterOriented();
virtual float shadingRate();
virtual int gridSize();
virtual float rasterEstimate(Bound &b);
virtual float rasterEstimate(Point &p0, Point &p1);
};

```

The `worldBegin()` function initializes the scene graph and hider state for a new frame. For example, a stochastic hider may set up tables of pseudorandom numbers here.

For immediate mode hidere, the `insert()` member function actually renders the primitive and then discards the data. Retained mode hidere will usually examine the primitive's bound to decide where to place it within the scene graph. The `remove()` method must be able to locate the primitive within the scene graph and delete it.

`bucketBegin()` signals the start of a new bucket. It returns `false` if all buckets have been exhausted and there are no more to render. Immediate mode hidere will have already rendered all the primitives and will return `false` immediately. Hidere that do not use buckets treat the entire raster as one big bucket. They return `true` from the first call of `bucketBegin()` and `false` thereafter.

The purpose of `bucketBegin()` is to give the hider a chance to perform bucket-specific initialization. For example, a bucket-oriented Z-buffer hider would only need as many depth values as there are pixels in a bucket, and all would be reset to infinity here.

The list of primitives in the current bucket is traversed by `firstPrim()`. The head is removed from the list and returned. When there are no more primitives in the current bucket, `firstPrim()` returns `NULL`.

The `hide()` function does most of the work in a retained mode hider. If the hider is bucket-oriented, the first thing it will do is ask the grid to bust itself into individual micropolygons. Micropolygons which only overlap the current bucket will be processed and discarded immediately. An individual micropolygon that lands in another bucket will be *forwarded* to that bucket to be processed later when the hider gets around to working on it. If the grid had not been busted into individual micropolygons, then the entire grid would need to be kept in memory until the hider finished processing the last bucket that overlapped any part of the grid. Busting the grid allows the hider to discard most of it, thus saving memory.

Hidere that do not use buckets are free to leave the micropolygons in grid form. For example, a hider that simply passes micropolygons to OpenGL for a quick preview rendering would be better off converting the grid to a quadrilateral strip data structure than busting it into individual micropolygons.

As shown here, the `trace()` function is straight out of the RenderMan specification. It takes an origin point and a direction to trace a ray, returning the

shaded colour of the first intersection (black if none are found). In practice, there will be additional parameters. For example, the recursion depth will often be passed down so that ray tracing can be terminated after a user-specified maximum number of bounces. See Chapter 6 on ray tracing for more information.

It is the `bucketEnd()` function that finalizes the colour of each pixel. Sampling hidens perform filtering here. If the display driver accepts arbitrary rectangles, then the pixels which make up the bucket can be sent for display and the memory reused by the hider. For scanline-oriented display drivers, the hider must buffer the pixels until it has a complete row of buckets.

Rounding out the `Hider` class are several functions used by the `diceable()` test in the `Primitive` class. Raster-oriented hidens are those that allow micropolygons to be larger (in world space) if they are near silhouette with respect to the camera and thus have a small footprint in raster space. Ray tracing hidens are not raster oriented because rays can come from any direction and the micropolygon size needs to be consistent.

Each hider can define its own shading rate (i.e. the desired size of micropolygons in terms of pixels) and limit on number of micropolygons per grid. Ray tracers tend to use large micropolygons to conserve memory. They make up for the sacrifice in image quality by shading at the actual intersection point of a ray and the micropolygon, not just at the micropolygon corners.

Two functions are provided to help the diceability test estimate raster space projections. The first takes a primitive bound in camera coordinates, projects to raster coordinates, and returns the area in terms of pixels. The second form takes two points in camera coordinates and returns the distance between them in pixels after being projected to raster coordinates. It is useful for determining independent dicing rates in the parametric u and v directions.

2.9.1 Occlusion Culling

It is within the `firstPrim()` method that the hider may optionally perform occlusion culling. Before returning a primitive, it checks to see if it is potentially visible within the current bucket. If not, it forwards that primitive to the next bucket it overlaps. The hider keeps examining primitives in the list until a potentially visible one is found or the list is exhausted. Some occlusion culling algorithms are more effective if the primitives are processed from front to back, so consider sorting the list in `bucketBegin()`. Note that a split operation might result in new primitives being added to the current bucket after it was sorted. Adding them to the front of the list is a good approximation to keeping the list sorted because that is where the parent primitive was before it got split.

For a simple example, consider what it would take to add occlusion culling to a Z-buffer hider. It already maintains the depth of the nearest object rendered thus far on a pixel-by-pixel basis. For a quick occlusion culling test, it only needs to examine the front of a primitive's bounding box. If the bound is axis-aligned then the front face will have a single depth. If all of the pixels in the current bucket which overlap the front face of the bounding box have a nearer depth, then the primitive cannot be visible and can be culled.

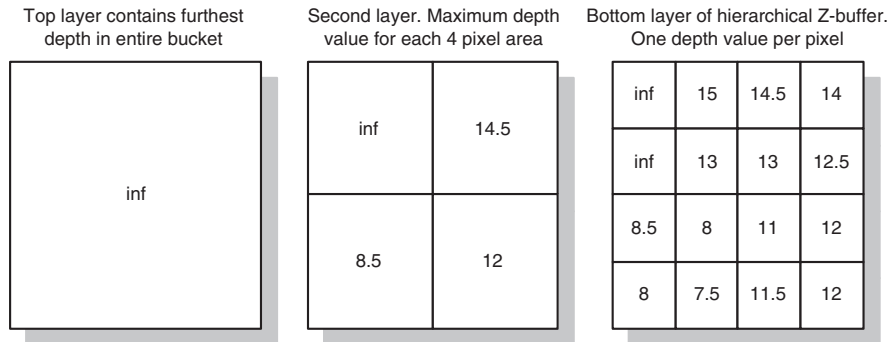


Figure 2.5 Construction of a hierarchical Z-buffer.

If the hider supports transparency, then only opaque (or nearly opaque) samples should contribute to the occlusion culling test. Even if the primitive is determined to be potentially visible within this bucket, the resulting grids and even individual micropolygons can be tested for occlusion later. Of course, there are diminishing returns the further down the pipeline geometry gets before being culled.

In the example above, an occluded primitive would require a depth test of every single pixel overlapped by its bounding volume. In a scene of high depth complexity where many primitives are occluded, the number of pixels tested could become a bottleneck.

A more advanced occlusion culling scheme is based upon the Hierarchical Z-Buffer Visibility algorithm (Green et al., 1993). Think of this data structure as a pyramid whose foundation is the Z-buffer for the bucket. In the second layer, each 2×2 square from the Z-buffer is summarized by a single z value equal to the maximum of the 4 below. Continuing this process results in each layer being one-fourth the size of the layer below (half as many rows and half as many columns) until the top of the pyramid is a single z value. This number represents the maximum depth of all the pixels in this bucket. Having bucket dimensions that are both powers of two simplifies this process. An example of constructing the data structure for a bucket with only 16 pixels is shown in Figure 2.5.

Now when a z value is updated in the Z-buffer, the z value in the second layer is re-evaluated to see if the maximum for that 2×2 square has decreased. If it has, the second layer is updated and the process continues upward until a value that need not change is encountered.

Occlusion culling starts at the top of the pyramid and works its way down only as necessary. If the value at the top of the pyramid is closer than the front face of the primitive's bound, then the primitive is culled for this bucket after comparing only two values. Otherwise, the hider drops down a layer and examines values that represent areas overlapped by the bound. Nodes closer than the front face of the primitive, are determined to be hiding that part of the object, and require no further examination. Only those areas with greater depths result in recursion to lower layers. In Figure 2.6, note how the bottom-right quadrant is removed from consideration after processing the second layer. Only two pixels

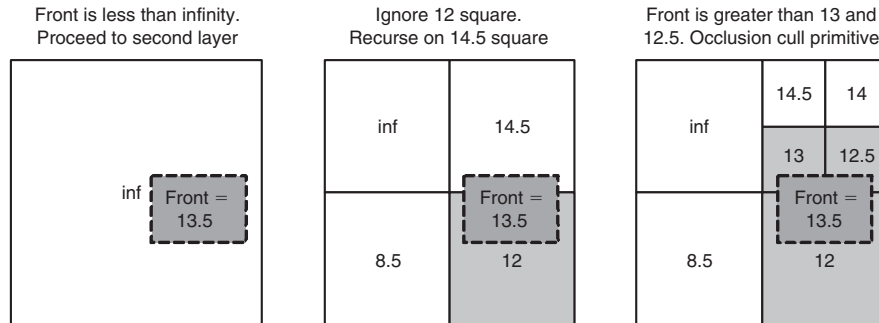


Figure 2.6 Occlusion culling.

in the bottom layer need to be examined in order to cull the primitive in this example.

2.10 Frameworks

A framework is the part of a renderer that manages the pipeline. It determines which operations will be performed on primitives, and in what order those operations will take place. Most renderers are built upon a single framework but with good object-oriented design, a renderer can support multiple frameworks that can be selected by the user at run time.

```
class Framework {
public:
    virtual void worldBegin();
    virtual void insert(Primitive *);
    virtual void remove(const Primitive *);
    virtual void worldEnd();
    virtual Colour trace(Point p, Vector r);

protected:
    Hider *hider;
};
```

The abstract base class for frameworks has member functions that are called by the State class at `worldBegin()` and `worldEnd()` time. Primitives are added and deleted from the scene graph by calling `insert()` and `remove()`, respectively. The `trace()` member function is only implemented by frameworks that support ray tracing.

Note that the `remove()` member function is never called from the API. Inserted primitives are generally consumed by the rendering pipeline. The `remove()` method exists for situations where the framework or hider internally deletes a primitive from the scene graph, for example to replace it with a more refined model.

2.10.1 Reyes Framework

It is straightforward to write the Reyes framework in terms of the `Framework` base class, given primitives that know how to bound, split and dice themselves. Three of the five functions are trivial:

```
void ReyesFramework::worldBegin()
{
    hider->worldBegin();
}

void ReyesFramework::remove(const Primitive *prim)
{
    hider->remove(prim);
}

Color ReyesFramework::trace(Point p, Vector r)
{
    return colourBlack;
}
```

The scene graph will be maintained by the hider. Since Reyes does not support ray tracing, calls to the `trace()` member function always return black (indicating that the ray did not hit anything).

Inserting a primitive with Reyes requires some special handling:

```
void ReyesFramework::insert(Primitive *prim)
{
    if prim is visible to camera rays AND
        prim's z coordinates are between clipping planes AND
        prim's bounds overlap viewing frustum AND
        prim's level of detail falls within visible range
    {
        if prim passes in front of epsilon plane
        {
            if (prim->splittable())
                prim->split(*this, true, true);
        }
        else
            hider->insert(prim);
    }
}
```

There are several situations where Reyes can cull a primitive without adding it to the scene graph. Sometimes primitives are flagged as invisible to camera rays, because they are only intended to block shadow rays, for example. Such primitives will be totally invisible to the Reyes framework and can be culled. A more common situation is that the primitive is entirely in front of the near clipping

plane, behind the far clipping plane, or otherwise outside of the viewing frustum. Because Reyes does not ray trace reflected objects, primitives off screen do not contribute to the final image and are also culled. Finally, a primitive whose level of detail is out of range gets culled. For example, a highly detailed representation should not be inserted when the bounds project to a single pixel on-screen.

The other special handling has to do with the notorious eye-splits problem in Reyes. If the primitive already passed the clipping test, then at least part of it is on the far side of the near clipping plane. If part of it is also on the near side of the epsilon plane (immediately in front of the camera) then it must be split in order to avoid a perspective divide by zero situation.

Primitives that pass all of the above tests are forwarded to the hider for insertion into the scene graph. If the hider happens to be immediate mode, it will actually consume and render the primitive before returning from its `insert()` member function.

Reyes is traditionally paired with a retained mode hider. In that case, `worldEnd()` is where the work is actually done:

```
void ReyesFramework::worldEnd()
{
    while (hider->bucketBegin()) {
        Primitive *p;
        while ((p = hider->firstPrim()) != NULL) {
            MicroPolygonGrid g;
            bool usplit, vsplit;
            if (p->diceable(g, *this, usplit, vsplit)) {
                p->dice(g);
                if (!g.isbackfacing() && !g.trim()) {
                    g.displace();
                    g.shade();
                    hider->hide(g);
                }
            }
            else if (p->splitable())
                p->split(*this, usplit, vsplit);
            delete p;
        }
        hider->bucketEnd();
    }
    hider->worldEnd();
}
```

The outer loop iterates over the hider's buckets. When the buckets are exhausted, `bucketBegin()` returns false. If a hider is not bucket-oriented, then `bucketBegin()` will only return true one time – representing the entire raster. An immediate mode hider will have already rendered everything with its `insert()` member function and will return false immediately.

The inner loop iterates over the primitives in the bucket. Those that are diceable are diced. If the resulting grid is entirely backfacing or entirely trimmed away by trim curves, then the grid is ignored. Otherwise, it is displaced, shaded and passed back to the hider for final hiding.

Undiceable primitives are split. This results in more primitives being inserted into the framework, possibly into the current bucket. Primitives that are neither diceable nor splittable should never happen, and are duly ignored.

2.10.2 Ray Tracing Framework

Use of the ray tracing framework implies that even primary rays will be traced. This framework must be used in conjunction with a ray tracing hider. Like Reyes, most of the member functions inherited from the `Framework` class are direct pass-throughs to the hider:

```
void RayTraceFramework::worldBegin()
{
    hider->worldBegin();
}

void RayTraceFramework::insert(Primitive *prim)
{
    hider->insert(prim);
}

void RayTraceFramework::remove(const Primitive *prim)
{
    hider->remove(prim);
}

Color RayTraceFramework::trace(Point p, Vector r)
{
    return hider->trace(p, r);
}

void RayTraceFramework::worldEnd()
{
    while (hider->bucketBegin())
        hider->bucketEnd();
    hider->worldEnd();
}
```

The `worldEnd()` member simply iterates over the buckets. It is up to the ray tracing hider to seed the ray tracing process by creating rays that originate at the camera and pass through the pixels on the image plane. It may choose to do this in `bucketEnd()` or `worldEnd()`, depending on whether the hider is bucket-oriented. If more than one ray per pixel is generated, the hider must filter the results before sending the final pixels on to the display subsystem.

Note that the ray tracing framework never sends grids to the hider. Ray tracing hiders will either intersect rays with high level primitives (such as NURBS) or tessellate the primitives on demand when a ray passes nearby.

2.10.3 Hybrid Framework

The Reyes framework does primary visibility efficiently, while the ray tracing framework handles shadow and reflection rays. By combining the two, the resulting hybrid framework can trivially provide the best features of each approach:

```
class HybridFramework : public Framework {
public:
    HybridFramework();
    ~HybridFramework();

    virtual void worldBegin();
    virtual void insert(Primitive *);
    virtual void remove(const Primitive *);
    virtual void worldEnd();
    virtual Color trace(Point p, Vector r);

protected:
    ReyesFramework *reyes;
    RayTraceFramework *raytrace;
};

HybridFramework::HybridFramework()
{
    reyes = new ReyesFramework();
    raytrace = new RayTraceFramework();
}

HybridFramework::~~HybridFramework()
{
    delete reyes;
    delete raytrace;
}

void HybridFramework::worldBegin()
{
    reyes->worldBegin();
    raytrace->worldBegin();
}

void HybridFramework::insert(Primitive *prim)
{
    reyes->insert(prim);
    raytrace->insert(prim);
}

void HybridFramework::remove(const Primitive *prim)
```

```
{
    reyes->remove(prim);
    raytrace->remove(prim);
}

void HybridFramework::worldEnd()
{
    reyes->worldEnd();
}

Color HybridFramework::trace(Point p, Vector r)
{
    return raytrace->trace(p, r);
}
```

The work is handed off symmetrically between the Reyes and ray tracing frameworks with the exceptions of `worldEnd()` and `trace()`. Instead of letting the ray trace hider generate primary rays, the Reyes side does the job of primary visibility more efficiently. When a request to trace a secondary ray comes along, the ray trace side handles it. Such requests are generated by shaders to determine shadows, reflections and refractions.

2.11 Conclusion

The architecture presented in this chapter certainly isn't the only way to design a production renderer, but it should provide a flexible foundation on which to build various rendering algorithms. Details on how to implement primitives, shader evaluators, ray tracing, global illumination and hiderefs will be covered in the remaining chapters. Those writing a renderer from scratch will find that those modules fit neatly into this architecture. Others who are maintaining mature renderers may find inspiration for improvements to their designs.

Production Rendering

Design and Implementation

Stephenson, I. (Ed.)

2005, XIII, 302 p. 115 illus., 32 illus. in color., Hardcover

ISBN: 978-1-85233-821-3