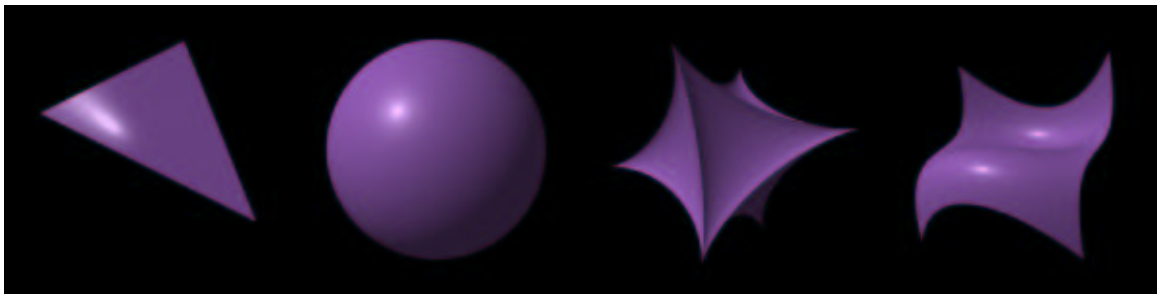
5 PRIMITIVES AND INTERPOLATION

A large variety of different graphical primitives are routinely used for high-quality, off-line rendering: spline patches, blobs, polygons, spheres, superquadrics, and many others [Watt92] (Figure 5.1 shows examples of several). No set of built-in primitives can ever be complete. There are many more that might be useful, both entirely new primitive types and variations on existing ones. Although libraries for interactive rendering, such as OpenGL [Neider93], support a subset of these primitives, graphics acceleration hardware can usually only directly render polygons. The more complex primitives are tessellated to polygons before being sent to the graphics hardware. Unfortunately, this may not be the most efficient way to render complex primitives. Rendering smooth surfaces in this manner results in a large number of polygons.



a b c d

Figure 5.1. Examples of several primitive types.
a) triangle, b) sphere, c) superquadric, d) spline patch

We can address both of these problems with a procedural interface for defining new primitives. As with procedural shading, a user of the graphics system writes short procedures to create new primitives or new versions of old ones. Our definition of a primitive in this context is fairly broad, including both primitives that are rendered directly and ones that are converted into other primitives for rendering.

We believe strongly in the utility of procedural primitives because of our experience with Pixel-Planes 5 [Fuchs89]. Our rendering library, PPHIGS (based on the PHIGS API), provided a standard set of primitives. However a number of users found this inadequate and wrote their own primitives or special purpose modifications of standard primitives on Pixel-Planes 5. The ability to create new primitives was not a part of the original Pixel-Planes 5 software design. Because we designed the hardware and software, we were able to provide the ability to add arbitrary user-written primitives. Some of the primitives were created as additions to the standard graphics library – peers to the standard triangle and other built-in primitives. Others were created using *display list callbacks.* This callback interface was added in the year after Pixel-Planes 5 became operational, and allowed new primitive code to be used without requiring a private copy of the graphics library source code or requiring the user to create all of the API drawing and inquiry functions of a fully supported primitive. Our experience with Pixel-Planes 5 is summarized in Section 5.1.

We would like to make the process of defining and using a new primitive as easy as possible. Based on our experience with Pixel-Planes 5, we have provided a general primitive interface in the PixelFlow graphics API. Our Pixel-Planes 5 experience did not answer the question of how to make it easier to write the primitives themselves. To address this, we carefully define what portions of the graphics pipeline should be included as part of the definition of the primitive and adapt our shading language to provide additional support for writing primitives.

On PixelFlow, new primitives can be defined using a machine-specific, testbed interface similar to the one used on Pixel-Planes 5, though on PixelFlow this interface is designed to fit the primitive stage of the abstract pipeline of Chapter 2. Because both of these testbed interfaces are machine specific, it is not possible to use primitives from Pixel-Planes 5 on PixelFlow. We have also defined new extensions to the pfman language to allow primitives to be written in a device-independent high-level form. The pfman compiler does not completely support these language extensions for reasons that will be discussed in Section 5.5.

In Section 5.1 we expand on our experiences with Pixel-Planes 5. In Section 5.2 we explain how the procedural primitive fits in the abstract pipeline of Chapter 2, and how it can fit into other graphics pipelines. The information in Section 5.2 applies to both the low-level testbed interface, which requires intimate knowledge of the PixelFlow internals, and the high-level language. In Section 5.3, we introduce some procedural primitive extensions to the OpenGL graphics API. Section 5.4 covers the language extensions for procedural primitives. Finally, Section 5.5 discusses the implementation of our language extensions for PixelFlow, including both the successes and difficulties.

5.1. Historical perspectives

The anecdotal evidence from primitives created on Pixel-Planes 5 provides insight into what capabilities are required for useful procedural primitives. In this section, we give an overview of the primitives that were defined, what they were used for, and what form of graphics library support they utilized.

Originally, the only option for creating new primitives was to incorporate them into the standard library. This style of primitive development requires a private copy of the source for the graphics library. This is not acceptable for most users. It is difficult to update the private libraries with new releases and bug fixes, and compilation times are rather long (on the order of half an hour for the Pixel-Planes 5 PPHIGS libraries). Most primitives created this way were incorporated back into the main graphics library. This requires a close relationship with the library maintainers. It also requires the user to write several additional API calls to save, load, and inquire the state of the primitive. The *featured polygon* and *disk* primitives were both created this way.

John Alspaugh wrote the *featured polygon* primitive for the UNC walkthrough project. A featured polygon is a convex polygon with convex holes cut into it. Each featured polygon was rendered directly by the hardware instead of being decomposed into ordinary convex polygons. The primitive was created to allow simple and efficient rendering of environments created with the Virtus WalkThrough program, which used featured polygons as a primary primitive (the current version of this program from the Virtus Corporation is

called WalkThrough Pro). For example, a wall with holes for a door and two windows is a single featured polygon.

The author wrote a *flat disk* primitive for use by the UNC ultrasound project [Bajura92]. The primitive renders flat disks, always aligned to face the screen. It was used to render pixels in captured video images from an ultrasound machine, where each non-black pixel was rendered as an appropriately colored disk. The author was working for the Pixel-Planes project at the time, and the primitive was built into the PPHIGS library as a standard primitive. However, given its specialized purpose, it would have been more appropriate to include using a procedural primitive interface.

The callback mechanism was added to make creation and use of new primitives easier. Pixel-Planes 5 renders a scene by traversing a database of geometry and transformations called a *display list*. The display list represents the scene as a directed acyclic graph. Each object is described by a section of display list called a *structure*. To render a structure, its *display list elements* are handled in order. Elements include transformations, primitives, and calls to render other display list structures. The callback mechanism adds a new type of display list element. When it is time to render the callback, a user-written function is called. The callback has access to all of the internal functions and data structures of the rendering library itself, as well as an arbitrary block of data from the application.

The major advantage of the callback method is that the user can create a new primitive without having to write all of the extra API support functions, or indeed having to modify the API library at all. Yet it does not make writing the primitive itself any easier. The callback mechanism completely exposes the internal structure of the rendering library. This requires the user who wants to write a primitive to have a good understanding of both the graphics hardware and the rendering library. Any primitive written as a callback was, by necessity, tailored for the Pixel-Planes 5 machine.

A 4D polygon primitive was created by David Banks using a combination of the callback mechanism and a custom copy of the graphics library. He used this primitive as part of his dissertation work in visualizing surfaces in 4-space [Banks92][Banks93]. For rendering, the 4D polygon is projected onto the 2D screen. To help in understanding the 4D

object, the primitive detects and colors silhouette edges and intersections with other polygons. The primitive itself is a callback, but a custom copy of the graphics library was required to change the pixel memory map to support intersection detection.

Subodh Kumar wrote a Beziér patch primitive as part of his dissertation work [Kumar95][Kumar96]. It was originally written using the callback interface, but later added as a full-fledged primitive. This is an example of a primitive that does not use the graphics hardware directly; it dynamically tessellates the patch into triangles, based on the current view of the patch.

Russ Taylor wrote a height field primitive for the UNC nanoManipulator project [Taylor93][Taylor94]. It was created to display large surfaces created by the scanning tunneling and atomic force microscopes. The surfaces have a regular grid structure with different elevations at each point in the grid. The data received by the primitive are an array of height values. The heights are used as one coordinate for each vertex in a grid of triangles.

Jon Leech wrote a particle system primitive. This primitive handles the creation, destruction, and movement of hundreds or thousands of particles. The particles are rendered using standard sphere primitives. This is an interesting example of a procedural primitive because it violates some of our normal conceptions of what a primitive is. Even though there may be thousands of disconnected particles all over the screen, they are still part of a single primitive. This primitive stretches the line drawn in Chapter 4 between procedural primitives and procedural models.

Finally, David Luebke and Chris Georges wrote a portal primitive as several cooperating callbacks [Luebke95]. This primitive is used for doors or mirrors in a scene. If the door or mirror is visible, it renders the display list structures for everything that appears behind the door or reflected in the mirror. In this case, rendering a single primitive may cause large portions of the database to be rendered (or re-rendered in mirror reflection).

From these examples, we draw several conclusions. We should support both direct rendering, where a primitive directly computes the visible pixels, and rendering through decomposition into simpler primitives. We should allow extra values (beyond just Z for Z-

buffering) to be used during rasterization. It may even be possible for a primitive to render a disconnected set of pixels as was done by the particle system, and we want to allow for this as well. In the sections that follow, we define an interface for procedural primitives that supports all of these requirements.

5.2. Creating primitives

In Chapter 2, we presented an abstract pipeline that included a stage for procedural primitives. In this section, we describe how procedures written using the abstract pipeline model can fit into two other graphics pipelines that are not fully procedural. We also provide more detail on what is included in the primitive stage and what is included in the surrounding transformation and interpolation stages.

5.2.1. Graphics pipeline

Figure 5.2 shows two simple graphics pipelines designed for polygon-based rendering. Figure 5.2a is based on the pipeline used in Pixel-Planes 5 [Fuchs89], while Figure 5.2b is a simplified view of the OpenGL graphics pipeline [OpenGL92].

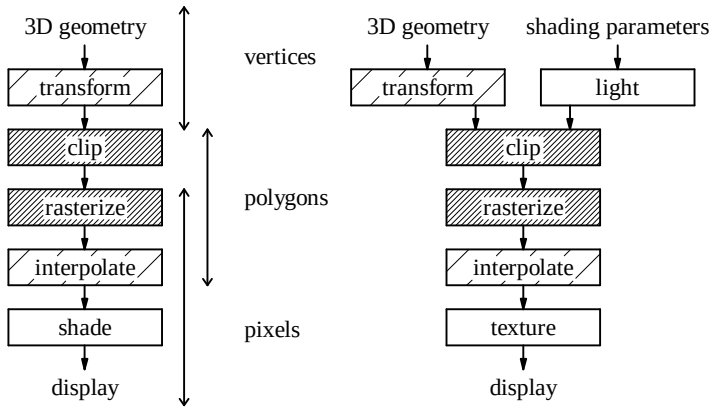


Figure 5.2. Some typical graphics pipelines.

a) Pipeline based on the Pixel-Planes 5 graphics system. b) Pipeline based on OpenGL.

I have renamed some of the stages in Figure 5.2 to emphasize the similarities between the two graphics pipelines. The *transform* stage converts 3D vertex positions to vertex positions on the screen. The *clip* stage takes collections of vertices representing a polygon

and clips off parts that extend behind the eye, off screen, or through a user-specified clip-plane. The *rasterize* stage finds the pixels that are in the polygon. This stage also does Z-buffering to compute visibility of each pixel. The *interpolate* stage for the Pixel-Planes pipeline computes the value of the parameters of the built-in Phong shader, as they vary across the polygon. For the OpenGL pipeline, this stage computes the value of the texture coordinates and shaded color as they vary across the polygon. The *shade* stage computes color from the various shading and lighting parameters. In the Pixel-Planes pipeline, this happens at each pixel after all of the polygons have been rendered. In the OpenGL pipeline, this happens at each vertex before the polygons are rendered, and it is the resulting color that is interpolated across the polygon. Because shading is done at each vertex in the OpenGL pipeline, it also has a *texture* stage after rasterization to apply an image texture to the surface.

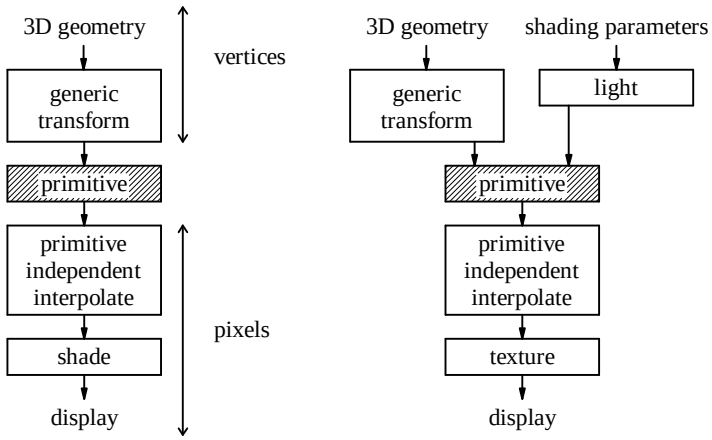


Figure 5.3. Pipeline modified for procedural primitives.

Procedural primitives should be orthogonal to the non-primitive stages. For true device-independence, primitives defined for one pipeline should be usable by a similar system using another pipeline. The abstract pipeline can serve as an effective mental model as users write new primitives for any of these actual pipelines. The procedural primitive will replace only some of the stages of the polygon pipelines in Figure 5.2. It will completely replace the darkly shaded stages, partially replace the lightly shaded stages, and will not affect the unshaded stages. Figure 5.3 shows the modified pipelines. Transformation and interpolation are divided because some forms of each depend on the primitive being rendered while other forms work for all primitives.

5.2.2. Transformation

For generality, and to make the process of creating a new primitive as simple as possible, we would prefer a completely separate transformation stage. If a renderer only supports a single primitive type (e.g. polygons), transformations and primitive rendering can easily be split into separate stages. However, different primitives require different types of transformation, making the division less straightforward. As a compromise, we support a handful of common transformation types as a separate stage. We require the procedural primitive to perform any more specific transformations itself, typically building on the supported common transformations.

One of the common transformations that we support is the transformation of points from object-space to screen-space. Many primitives have point-like input parameters; both triangles and spline patches have such parameters (vertices and control points respectively). Following [Fleischer87], the common transformations are the point transform and its inverse, for transforming points and planes back and forth between object-space and screen-space, and the vector transform and its inverse, for vectors and normals. This set of common transformations can be shared by all primitives.

Primitive-specific transformations are harder to characterize in any general fashion. One example involves quadric surfaces. Quadric surfaces include spheres, ellipsoids, hyperboloids and paraboloids. Each of these can be defined by a single 4x4 matrix, Q. Points on the surface, $\vec{p} = [x\ y\ z\ 1]^T$, obey the equation

$$\vec{p}^T Q \vec{p} = 0$$

Proper rendering of the quadric surface requires that Q be transformed from object-space to screen-space. This transformation does not fit any of the standard types. If points are transformed by a 4x4 matrix, M, so

$$\vec{p}' = M \vec{p}$$

the correct transformation for Q is

$$Q' = M^{-1\,T} Q M^{-1}$$

Since we cannot predict the arbitrary transformations (such as this one) that different primitives will require, we do not attempt to handle them in the general transformation stage. A quadric surface primitive would build the transformation for its input parameter, Q, from the provided common transformation, M^{-1}.

5.2.3. Interpolation

When a shader runs, its job is to compute color using all of its input parameters. An *interpolator* is a function that provides the *varying* inputs for the shader. If a parameter varies across the surface of the primitive, the interpolator produces this variation. Just as with transformations, it is easy to separate interpolation from rasterization when only a single primitive type is considered. As soon as we allow more types of primitives, it becomes apparent that some kinds of interpolation can be used for any primitive while others are quite specific to one kind of primitive.

Primitive-independent interpolation

A *Primitive-independent interpolator* should be able to create parameter values across any primitive without knowing anything except the positions of the pixels that have been determined to be in the primitive. Thus, it should not know the type of primitive (triangle, spline patch, quadric surface, …), nor any of the primitive's input parameters (vertices, control points, quadric matrices, …).

The simplest example is the *constant interpolator*. This is used for parameters that have a single constant value for all pixels. It is common to have parameters that have been declared varying in the shader, but are constant across a primitive. The procedural interpolation stage covers any method used to determine the value of varying shading parameters within a primitive. Consequently, while the terminology we have chosen may sound strange, a constant interpolator is the most useful of the primitive-independent interpolators

Using the location of the pixels on the screen, primitive-independent interpolators can also include any arbitrary function of the screen position. Since all primitives must also have a Z value at each pixel for Z-buffering, primitive-independent interpolators can also include arbitrary functions of the 3D position of each pixel. For example, we might have a

fog thickness parameter that thins with height above the ground. This linear function of the height in world space is completely independent of the individual primitives. The OpenGL *texture coordinate generators* are another example [Neider93]. A texture coordinate generator projects texture coordinates onto a surface. Figure 5.4 shows spherical projection of texture coordinates onto two different objects. Both are shaded using the same simple checkerboard shader.

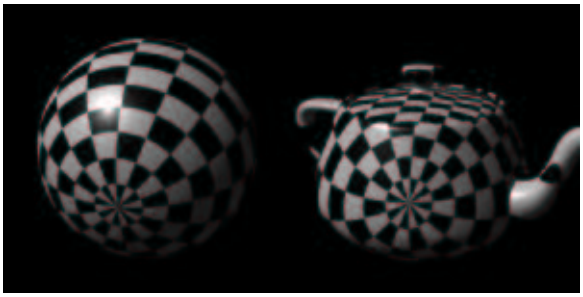


Figure 5.4. Example of primitive-independent interpolation.

Primitive-independent interpolators are a close relative to volume shaders [Cook84]. They define a scalar field in space; any primitive can use parameter values from this field based on where it is located in this space. Ebert used similar parameter fields, which he called *solid spaces* [Ebert94] for shading volume-rendered smoke and fog.

Primitive-dependent interpolation

As with transformations, it is harder to provide a concise description of the arbitrary kinds of interpolation that are primitive-dependent. One common example is linear interpolation across a triangle. This interpolator uses a value at each vertex to define parameter values across the triangle. For a different primitive without vertices (e.g. a sphere), linear interpolation makes no sense., We require interpolation that depends on features of the primitive itself to be done by the code for the procedural primitive.

5.3. Application interface for arbitrary primitives

To use procedural primitives, there must be a way to describe an arbitrary primitive in the graphics API. It should be general enough for any primitive we may want to create, and must fit well into the existing API.

The OpenGL API already supports several primitive types (triangles, triangles strips, triangle fans, quadrilaterals, …). The OpenGL code for a triangle is shown in Figure 5.5. For new primitive types, we simply replace `GL_TRIANGLES` with an identifier for the new primitive. This identifier comes from the same `glLoadExtensionCodeEXT` call that is used for loading new surface shaders and lights.

```
glBegin(GL_TRIANGLES);
   glVertexfv(v0);
   glVertexfv(v1);
   glVertexfv(v2);
glEnd();
```

Figure 5.5. OpenGL code for a triangle.

For primitives that require more information than just a set of vertices, we add a `glRastParamEXT` call. `glRastParamEXT` provides a value at each vertex for some new rasterization parameter to be used by the procedural primitive. For example, we might define a triangular slab primitive by the vertices of a base triangle and the thickness of the primitive at each vertex. The API code for such a primitive appears in Figure 5.6.

```
GLenum triangle_wedge = glLoadExtensionCodeEXT("triangleWedgePrim",
      PRIMITIVE_CODE_EXT);
…
glBegin(triangle_wedge);
   glRastParamfEXT(thickness, t0);
   glVertexfv(v0);

   glRastParamfEXT(thickness, t1);
   glVertexfv(v1);

   glRastParamfEXT(thickness, t2);
   glVertexfv(v2);
glEnd();
```

Figure 5.6. API code demonstrating `glRastParamEXT`.

The `glVertex` call serves two purposes. It provides the value for the vertex rasterization parameter, and it binds the current values for the other rasterization parameters and for the shading parameters (from `glMaterial`, Section 4.1.5). Not every primitive has a rasterization parameter that could be conveniently called a vertex. We define a *sequence point* as a place in the application code where the current rasterization and shading pa-

rameters are bound. This terminology comes from the compiler literature. For example, in C++, the ‘`,`’ operator serves as a sequence point in an expression and ‘`;`’ serves as a sequence point between expressions. In our extensions to the API, `glSequencePointEXT` serves the same purpose as `glVertex`, but without the included vertex coordinates. In fact, using `glVertex` is equivalent to using `glRastParamEXT` to set the vertex coordinates, then `glSequencePointEXT` to bind the current set of rasterization and shading values. A sphere primitive using `glSequencePointEXT` is shown in Figure 5.7.

```
glBegin(sphere);
   glRastParamfvEXT(center, ctr);
   glRastParamfEXT(radius, r);
   glSequencePointEXT();
glEnd();
```

Figure 5.7. Example of a rasterizer using `glSequencePointEXT`.

Finally, we add the `glMaterialInterpEXT` call. With this function, the application can change the type of interpolation to use for each material appearance parameter and each primitive type.

5.4. Language details

The procedural primitive literature is not as developed as the procedural shading literature, In particular there has been little work using special purpose languages for writing new primitives. Consequently, we had to define the language features and interface. Since two of the key advantages of a special purpose language are device-independence and the related hiding of architecture details, some effort was taken to ensure that primitives would not be specific only to the PixelFlow system.

We add a new pfman function type, `primitive`, which is parallel to the existing `surface` and `light` functions. This function is responsible for determining which pixels are inside the primitive, updating the Z-buffer, and triggering the interpolation of the surface shading parameters across the primitive. In the next few sections, we provide details on the exact responsibilities of a `primitive` function, and the reasons for some of these choices.

5.4.1. The pixel-centric view

`Primitive` functions operate with the same *pixel-centric* view as surface shaders. In the *pixel-centric* view, a primitive or surface-shading function is written as if it is running on only a single sample on the surface. A surface shader's job is to determine the color of that one sample. This description for a single sample is extended to the rest of the surface by the compiler and renderer. Compare this pixel-centric view to the *incremental* one where the shader must describe explicitly how to find the color of a sample given the color of the neighboring sample.

A procedural primitive's job is to determine whether a pixel is in the primitive, whether the pixel is visible, and what value a single arbitrary shading parameter should take there. The compiler and renderer use this single-pixel description to create the full set of visible pixels in the primitive, and the full set of shading appearance parameters across the primitive. Compare this to the incremental approach where the procedural primitive steps from one pixel to the next and one scan line to the next.

Some RenderMan surface shaders (notably the wall paper in Pixar's KnickKnack animation [Pixar89b]) operate by drawing geometric shapes on the surface. Since the shader itself has a pixel-centric view, the shading code to render these shapes must also take a pixel-centric view.

The Pixel-Planes polygon scan conversion algorithm uses a linear function of the pixel coordinates (`a*x + b*y + c`) to define each edge of the polygon [Fuchs85]. This function is positive inside the edge and negative outside. The Pixel-Planes algorithm is pixel-centric; each pixel evaluates the edge functions to determine whether it is inside or outside the polygon. Pineda demonstrated how this pixel-centric algorithm can be converted into an incremental one [Pineda88]. His algorithm incrementally computes the linear edge functions, depth, and shading parameters from one pixel to the next across the scan line by adding `a` at each step, and from one scan line to the next by adding `b` at each step. As the algorithm steps across the polygon, if an edge function changes sign, it knows that it has crossed outside the polygon and should step down to the next line. This poly-

gon rendering algorithm has been used by Silicon Graphics [Akeley93] in at least one of their graphics systems.

Obviously, the fact that the PixelFlow SIMD array operates best with a pixel-centric view influenced our choice to adopt this view for the `primitive` functions. Given the existence of similar code in RenderMan shaders and the possibility of an incremental evaluation of a pixel-centric description, we believe that this choice satisfies the desire for a language interface for primitives that need not be restricted solely to PixelFlow.

5.4.2. Parameter types

Primitives use more different types of parameters than surface shaders or lights. We classify them into varying, per-vertex, interpolation, and per-primitive parameters.

Since one of a primitive's responsibilities is to set the current Z-buffer depth, primitives will have Z as a *varying* parameter. Some primitives may also use extra varying parameters for storing partial results or communication between successive calls to the primitive function. For example, the fast sphere drawing algorithm used on Pixel-Planes 5 [Fuchs85] and implemented on PixelFlow by Kenny Hoff, uses a *Q-buffer*, which contains the value x^2+y^2. This value is required for every sphere, but since it never changes it is computed once and stored in the Q-buffer.

Per-vertex (or *per-sequence-point*) parameters provide a set of uniform values to the primitive. Polygon vertices and spline control points are two examples of this kind of parameter. These parameters correspond to the values set using `glVertex` or our `glRastParamEXT` API extension and bound by a call to `glVertex` or `glSequencePointEXT`.

Interpolation parameters contain uniform shading data to be interpolated across the primitive. The primitive must have a way to indicate how interpolated shading parameters should be computed, but it does not need to know what these parameters are or how many there are. The interpolation parameters provide a way to represent these computations in the code without requiring the primitive to know these superfluous details. Variables of a new `interpolation_data` type represent the shading data to interpolate.

These variables are used in the `primitive` function as if they each contained just single value, but operations apply to every shading appearance parameter. Each primitive can have a special per-sequence-point parameter of this type, `interp_data`, which represents the values of all per-sequence-point shading appearance parameters, set by `glMaterial`, `glColor`, `glNormal`, or `glTexCoord` (Figure 5.11).

The final type, *per-primitive* parameters, are used for overall control of the primitive. Per-primitive parameters are equivalent to uniform parameters for surface or light shaders. With our API extensions, these parameters are set by any calls to `glRastParamEXT` that appear outside of the primitive's `glBegin` / `glEnd` (Figure 5.8).

```
glRastParamfEXT(radius, r);
glRastParamfvEXT(center, c);
glRastParamfvEXT(axis, a);
glBegin(surface_of_revolution);
   glVertex3f(v_1);
   …
   glVertex3f(v_n);
glEnd();
```

Figure 5.8. Example per-primitive parameters (radius, center, and axis).

5.4.3. Subdivision to other primitives

Figure 5.9. Primitive using subdivision to triangles.

One type of procedural primitive just produces a subdivision into new primitives. This is the method commonly used for rendering spline patches on graphics hardware. Patches are successively subdivided into smaller patches until they meet some size or flatness constraint and can be rendered as triangles. It is also the method that the Reyes algorithm used by Pixar's PhotoRealistic RenderMan renderer uses for rendering all primitives [Cook87] (some other examples were given in Section 5.1). For this type of procedural

primitive, one primitive simply includes a call to another. A simple example is shown in Figure 5.9. The pfman code for this primitive appears in appendix A.

5.4.4. Direct rendering

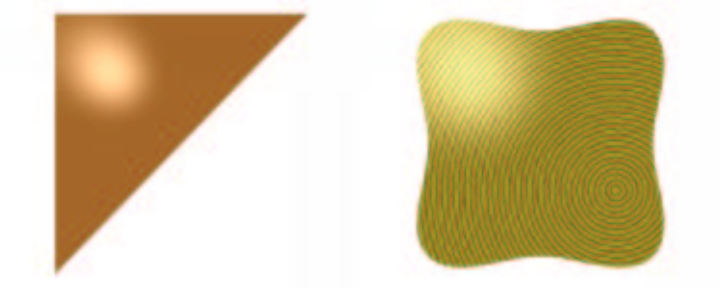


Figure 5.10. Primitives using direct rendering.

Other primitives are rendered directly. Figure 5.10 shows two simple example. The code for these primitives also appears in appendix A.

Directly rendered primitives use a new `interpolation` construct (Figure 5.11). The `interpolation` construct is similar to a C `switch` / `case` statement. Each case computes a different form of interpolation that the primitive supports. The `interpolation` construct generally appears in the innermost of a series of comparisons that determine the coverage of the primitive. Each case of the `interpolation` construct includes an `interpolator_return` statement. This statement indicates the resulting value for any shading appearance parameters using that type of interpolation.

The `interpolation` construct should be placed at the point in the primitive function where it has been determined that the current pixel is in the primitive. The cases of the `interpolation` statement describe how to do each different kind of interpolation that the primitive supports. It also indicates the point where the primitive-independent interpolators should be run for parameters that use them.

Figure 5.11 shows the outline of a triangle written with these extensions. We would prefer to use direct expressions of the pixel location, but for ease of implementation we currently require the use of a function, `linear_expr(a,b,c)`, which computes `a*x + b*y + c`.

```
primitive triangle(
      transform_as_point float vertex[][3],
      interpolation_data interp_data[],
      varying fixed<32,32> px_shader_z)
{
   … transformation and clipping of vertices …
   … compute linear expression coefficients for edges and Z …

   // compare against pixel screen coordinates to find out if the
   // current pixel is inside the triangle
   if (linear_expr(edge[0][0],edge[0][1],edge[0][2]) >= 0  &&
         linear_expr(edge[1][0],edge[1][1],edge[1][2]) >= 0  &&
         linear_expr(edge[2][0],edge[2][1],edge[2][2]) >= 0) {
      // compare with Z-buffer to see if the current pixel is visible
      if (linear_expr(az, bz, cz) > px_shader_z) {
         px_shader_z = linear_expr(az, bz, cz);

         // if the current pixel is active at this point, it is visible
         // and in the triangle. Interpolate the shading parameters
         interpolation {
            case PXlinear:
               interpolator_return(linear_expr(…));
         }
      }
   }
```

Figure 5.11. Pseudo-code for a triangle rasterizer.

5.4.5. Interpolator functions

We also introduce a new `interpolator` function type for primitive-independent interpolations. An `interpolator` function appears quite similar to one case of the `interpolation` construct. It uses the `interpolation_data` type for variables and returns the interpolated results with the `interpolator_return` statement. It uses varying parameters for things like pixel location and Z; it uses uniform parameters for general control of the interpolator; and it uses a special `interp_data` parameter for data specific to each shading parameter it will interpolate. For an example of a simple interpolator, see Figure 5.12.

```
interpolator radial(
      interpolation_data interp_data;
      uniform float radial_scale;
      varying fixed<32,32> px_shader_z)
{
   float x= linear_expr(1,0,0), y= linear_expr(0,1,0), z= px_shader_z;
   return(data * radial_scale * sqrt(x*x + y*y + z*z));
}
```

Figure 5.12. A simple interpolator function

5.5. Implementation

We have added the `primitive` function type to the pfman language to enable new primitives to be added to the PixelFlow machine. Direct rendered primitives are limited to using simple linear expressions for their interpolated shading parameters, and `interpolator` functions are not supported. In the remainder of this section, we discuss one optimization that we included to make pfman primitives more efficient and outline the problems with the current implementation.

5.5.1. Primitive bounds

The *uniform* code for a shader instance is executed at most once per frame. It only needs to be rerun to generate a new SIMD instruction stream when the uniform parameters to the instance change. The resulting SIMD instruction stream, consisting of the *varying* operations performed by the shader, is executed at most once per region on the screen. In contrast, uniform primitive code must be re-executed for every primitive, potentially thousands to millions of times per frame. Each primitive may be quite small, landing in just one region, or large, encompassing the entire screen.

Without further information, the SIMD instruction stream for each primitive must be executed for every region on the screen. If the individual primitives are small, the extra execution by the SIMD array is a huge waste of computing resources. A graphics system that does not use a SIMD array would still have a similar problem since any Pineda-style scan line algorithm would need to scan over large portions of the screen looking for the first pixel inside the primitive [Pineda88].

One of the advantages of a shading language over hand-coded shaders is that the shading language is device and implementation-independent. We believe that including either a bounding-box or top-most point in our language extensions would force too much of the implementation to seep into the language definition. Instead, we provide a *hint* mechanism to help indicate the actual screen coverage of the primitive. The hint is provided through an extra `output` parameter of the primitive, `pfman_primitive_bounds_hint`. Primitives that set this hint parameter will only be executed on a subset of the regions. Primitives that do not set the hint will still be rendered completely correctly, but less efficiently, since they will be executed on every region of the screen. If another implementation used a different hint parameter, it will be ignored on PixelFlow.

5.5.2. Interpolator functions

We have not implemented the primitive-independent `interpolator` functions because each `interpolator` function may be called upon to produce values for shading parameters of any type. The current pfman compiler does not have provisions for doing operations without knowing the types of the variables at compile-time. We were able to allow limited use of the `interpolation` construct in `primitive` functions only for the special case where the `interpolator_return` is limited to linear expressions using a `linear_expression` function.

Since PixelFlow restricts varying shading parameters to only use fixed-point types, it would be possible to create a fully functional implementation of both `interpolator` functions and the `interpolation` construct with the addition of a new fixed-point type. In Chapter 4, we compared fixed point to floating point. For quantities in SIMD pixel memory, fixed point is similar to floating point with a constant exponent. The problem is that each call to an `interpolator` functions could potentially involve a parameter with a different size and exponent. The existing fixed point type insists that both of these are compile time constants. With a new fixed point type that allowed them to be uniform instead of just constant, `interpolation` functions could work as they were designed.

type	mantissa	exponent
uniform float	`uniform`	`uniform`
varying float	`varying`	`varying`
pfman fixed point	`varying`	`constant`
interpolator fixed point	`varying`	`uniform`

Figure 5.13. Comparison of fixed and floating point types

We considered implementing this form of fixed point for shaders, where it would provide a middle ground between the efficiency of fixed point and the ease-of-use of floating point. Adding any new type requires creating new versions of all of the basic math and comparison operations, array versions of these operations, and conversions between the new type and the existing types. Since we already had fixed point and floating point, we elected to concentrate our efforts on other facets of the compiler. As a result, surface and light shaders are more successful, but correct and generic pfman `interpolation` functions are not possible.

5.5.3. Code efficiency

Because both the uniform and varying code for primitives are executed many more times than the corresponding shader code, their efficiency is much more critical to the machine performance. The only SIMD instructions executed by the hand-coded triangle primitive on PixelFlow are linear expressions and loads into pixel memory. Both are single operations: the linear expressions are used to identify pixels inside the triangle edges and to interpolate parameters across the triangle; the load instructions are used for shading parameters that are constant across the triangle. Further, both the edge comparison instructions and the shading parameter load instructions are optimized versions, which allow the individual operations to be pipelined across several instructions. Every instruction and every cycle counts.

The code produced by the pfman compiler is not up to these standards. For pfman optimizations, detailed in Chapter 4, we focused our limited resources on those optimizations that would most aid surface shading and lighting. Few of those optimizations help for primitives. The pfman compiler produces several less-efficient instructions for each of the special-purpose instructions that appear in the hand-coded triangle.

As mentioned in Section 3.3.1, the low-level interface for shaders, lights, and primitives all consist of C++ with embedded *EMC functions* (Figure 5.14). The shader's EMC functions allocate the SIMD instruction buffer space incrementally, while those for the primitives compute the total required buffer space before their first EMC function. This avoids costly comparisons in every EMC function, comparisons that disrupt the RISC processor's instruction pipeline. Yet, it also severely limits the complexity of the EMC operations that can be done. The instruction count for a triangle can be simply characterized. However, without introducing restrictions to the primitives, finding the instruction count of a user-written primitive is equivalent to solving the halting problem [Lewis81].

```
…
pfman_map->pushenab(pfman_p);

EMC_MemNETree_Si(pfman_p,
    (/* pfman_tmp0 */ sfixed<15,0>(pfman_frameptr)).addr,
    2, 0);

pfman_map->pushenab(pfman_p);

{

    //*** 32:   interpolation { ***//

    // do interpolation
    {
        // set shader ID
        EMC_TreeIntoMem_Si(pfman_p,
            pfman_map->find_pm_address(pxid_shader_id),
            pfman_map->find_pm_length(pxid_shader_id),
            pfman_map->get_shader_id());

        ShaderParmEntry *pfman_sp = pfman_parmbuffer->
            get_shader_parm_table();
        for(PXint pfman_i = 0;
            pfman_i < pfman_parmbuffer->get_num_shader_parms();
            ++pfman_i, ++pfman_sp)
        {
            switch(pfman_sp->interpolator) {
                case PXlinear:
…
```

Figure 5.14. Code generated by the pfman compiler.

In addition to the difficulty of computing the total instruction buffer size, the two styles of EMC commands cannot easily be intermixed. We cannot call functions using the shader-style EMC commands from a primitive using the faster EMC commands. These concerns, and the fact that the same pfman compiler is used for shading, lighting, and primitives, led to our decision to use the less-efficient commands for primitives written in the pfman language.

The best solution for optimal efficiency would be to convert the pfman compiler and the entire PixelFlow system to use the faster versions of the EMC functions for everything. While the compiler cannot easily compute the complete SIMD instruction count for an entire function, it could compute the counts for sequences of EMC commands without C++ control flow or function calls. This would require some additional compile-time analysis, but would create more efficient code for pfman primitives, surface shaders, and lights.

5.5.4. Buffer space

Given the dynamic nature of scenes in computer graphics, it is difficult if not impossible to perfectly balance the load between several rendering nodes. All of the rendering nodes must send the results for a region over the composition network at the same time (Chapter 2), which is a point of required synchronization between the processors. If rendering node A has more primitives or more complicated primitives in one region than any of the other nodes, a naïve implementation would require all of the other rendering nodes to wait for A. In the next region, rendering node B may be the one with the most work, but once again all of the other nodes, including A, would have to wait. If the synchronization could be relaxed, node B could start early and node A could run late. All of the nodes would have less idle time and the total performance of the machine would improve.

To relax the synchronization, we buffer several regions as they are rendered. Thus, while node A is still rendering the region in buffer 0, the other nodes, including node B can begin on the next region in buffer 1. When A is finished, buffer 0 is sent for all nodes, allowing it to be reused. With only two buffers, if a third node, C, finishes both regions before node A finishes the first, C will have to wait until a buffer is free. More buffers give a

more balanced execution. With eight buffers, node C would wait if it renders eight regions in less time than it takes node A to render the first.

The question of how many buffers to use was addressed in [Molnar91]. According to his simulations on a set of sample data-sets, four regions of buffering improves the load-balance by 50%, while eight regions of buffering provided almost the full parallelism. Based on this analysis, we recognized that region buffering is important for PixelFlow. These buffers are held pixel memory.

As discussed in Section 4.3, even at 256 bytes, pixel memory is one of the most limiting factors for compiled shaders. On rendering nodes, region buffers are configured to use $4/5^{ths}$ of the available pixel memory (the exact number of buffers is determined by the size of the largest set of shading parameters). After overhead, this leaves at most 50 bytes for any varying computations by the primitive. Compare this to the memory usage of the shaders in Figure 4.11.

This is not a problem for simple primitives like polygons; most of their computation is uniform and does not occupy pixel memory. However, buffering must be reduced for primitives that use more pixel memory. A primitive that use a great deal of pixel memory (similar to that used by the shaders) requires that we dispense with region buffering entirely.

Three options are to use no buffering, to change the allocation ratio between working memory for the primitives and memory for buffered regions, or to change the buffering dynamically.

Using no buffering at all will hurt the basic polygon performance. Since we expect that polygons will always remain one of the primary primitives used by our machine, this is not an acceptable alternative.

The current $4/5^{ths}$ ratio was chosen empirically based on the needs of the current primitives and the post-rasterization computation done on the rendering node. The easiest way to accommodate a primitive that needs more pixel memory than this is to change the

allocation ratio. This requires recompiling the PixelFlow libraries with the new ratio, a somewhat expensive prospect.

Since primitive functions can contain arbitrary code, including recursive calls to other primitives, it is not possible to know their pixel memory requirements in advance. The ideal solution would be to dynamically allocate rendering buffers and pixel memory for primitive computations from the same pool. If a primitive needs more memory than is currently available and there are multiple buffers currently allocated, it would block until an extra buffer becomes free, then use that buffer as working memory. Unfortunately, this solution requires significant changes to the software infrastructure. It would have been much easier if dynamic buffer allocation had been a requirement from the beginning of the software development.

Instead of changing the buffering strategy by any of these methods, we limit the types of primitives we can support. This decision affects both primitives written with pfman and hand-coded primitives.

5.5.5. Data organization

A final shortcoming of the current implementation of pfman primitives is caused by the differing data organizations used by the pfman compiler and the low-level primitive interface. The pfman compiler expects each rasterizer parameter to be grouped as an array of values, one value for each sequence point. This is the natural ordering for pfman since this is the interface it presents for the rasterization parameters: each is an array of values. The testbed primitive interface presents a table of all rasterization parameters grouped by sequence point, with a new group of rasterization parameters for each sequence point. This is the natural ordering as the rasterization parameters are collected since they are all specified for the first sequence point, then again for the second, etc.

This is just an implementation problem, which could be solved by either changes to the pfman compiler or the lower-level primitive interface. The current solution is for the pfman compiler to create wrappers around each primitive it compiles to shuffle the rasterization parameters into the order it expects. We also require wrappers around each low-level primitive that will be called from a pfman primitive. For a less time-critical applica-

tion, this solution would suffice. However, reading and writing data for the sole purpose of rearranging data formats is a waste of memory bandwidth that we would prefer to avoid.

This could be fixed by either rearranging the data structures of the low-level interface, or at the pfman compiler level, by treating rasterization parameters as a special data type. Rearranging the data structures of the low-level interface would cause fairly extensive changes to the existing software since these data structures are widely used. Treating rasterization parameters as their own data type would allow the compiler to generate different code to access them than it does for accessing other data or shading parameters. Math and support functions would need to be written for the new type. The choice of regular or rasterization data types would be inferred by the definition context so the addition of the extra internal type could remain invisible to the user.

5.6. Demonstration of procedural primitives

The latter part of this chapter has focused on the shortcomings of the pfman compiler for writing new primitives. This work can provide useful lessons for future attempts at high-level support for primitives. Code efficiency is important for primitives; plan to spend time for optimizations. To avoid limiting the kinds of primitives, plan on providing similar resources for rasterization as are used for shading. Finally, avoid unnecessary copying of data from application through the frame buffer, but particularly from the primitive specification in the application through the interpolation of shading parameters.

It should be emphasized that PixelFlow does allow fast and fully-functional procedural primitives using the testbed interface. This is similar to the interface provided by Pixel-Planes 5 (the use of which was documented in Section 5.1). Since both interfaces are low-level, they require knowledge of the graphics hardware. They also lack the device-independence provided by a language interface, so the Pixel-Planes 5 primitives are not usable on PixelFlow. Kenny Hoff has written sphere and flat disk primitives at this level, and all of the PixelFlow OpenGL primitives are also written with this interface.