Procedural Primitives in a High Performance, Hardware Accelerated, Z-Buffer Renderer

Marc Olano, Anselmo Lastra, Jonathan Leech

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

In the past, very few types of primitives have been directly supported by graphics accelerators, in spite of the fact that a great variety of complex primitives are routinely used for high-quality rendering. We explore the issues associated with the support of arbitrary procedural primitives, and describe a machine-independent language as well as an extension to OpenGL designed to support user-defined procedural primitives. We have created a prototype implementation on PixelFlow, a high-performance graphics accelerator.

1. Introduction

A large variety of different graphical primitives are routinely used for high-quality, off-line rendering. These primitives include polygons, spheres, superquadrics, spline patches, blobs, and many others (figure 1 shows examples of several). Although libraries for interactive rendering, such as OpenGL [Akeley92], support a subset of these primitives, graphics acceleration hardware can usually only directly render polygons. Thus, 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 because rendering smooth surfaces in this manner can result in a very large number of polygons (tessellated spheres are a good example). An alternative is to directly use the rasterization hardware.

Using a high-level language, we provide a way for users to create new procedural primitives that can render directly using the acceleration hardware or indirectly by tessellation into other primitives. With extensions to OpenGL, we allow the programmer to supplement the existing primitives in a well-integrated fashion. We have built a prototype implementation on PixelFlow [Molnar92], a machine for high-performance interactive 3D graphics. For generality, we have avoided exposing into the language the details of the PixelFlow machine itself, beyond the fact that it uses Z-buffer rendering. Thus

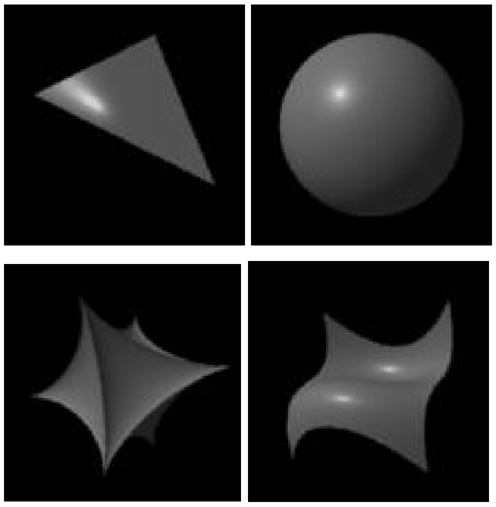
Figure 1. Examples of several primitive types.

this language is suitable for use with other hardware or with software-based renderers.

This primitive-description language is an extension of the shading language that we designed for PixelFlow in an effort to demonstrate user-programmable procedural shading in real time [Lastra95]. We based the language on Pixar's RenderMan shading language [Hanrahan90]. Following the philosophy of the RenderMan design, we have added only the language constructs that seemed necessary to support the task.

The strongest evidence we have for the utility of procedural primitives lies in our experience from Pixel-Planes 5, our last graphics machine. A number of people wrote code for their own primitives or for special purpose modifications of standard primitives on Pixel-Planes 5. All of these primitives were created without the benefit of any higher-level interface at all. By providing a high-level interface, the task of creating these primitives will be much easier on PixelFlow. Many of the arguments used to justify procedural shading hold just as well for procedural primitives. In fact, the first sentence of [Hanrahan90], the paper that introduced the RenderMan shading language, says, "The appearance

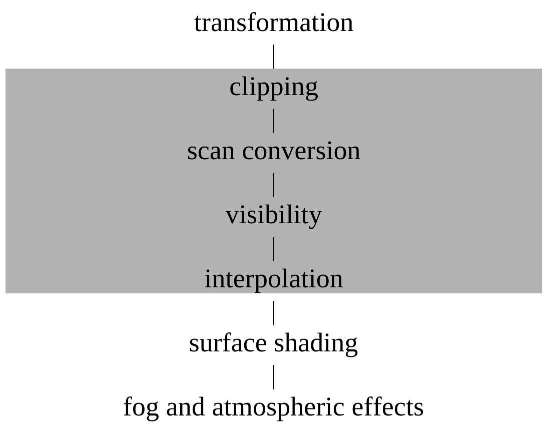

Figure 2. A typical graphics pipeline. Shaded portions are included in the procedural primitive.

of objects in computer generated imagery ... depends on both their shape and shading."

We are, of course, not the first to consider procedural primitives. There have been a number of papers on procedural models [Hedelman84] [Amburn86] [Green88] [Upstill90]. These are procedures for higher level objects that generate lower level primitives. They can provide some advantages for high-level culling and changes in the level of detail of the model.

A number of ray tracers allow new primitives to be added [Rubin80] [Hall83] [Wyvill85] [Kuchkuda88] [Kolb92]. In all of these, adding a new primitive consists of adding a new ray intersection function to the ray tracer. None that we are aware of use a separate language to define the new primitives.

There have also been a handful of scan-line and Z-buffer systems that have allowed the addition of new primitives. [Crow82] treated primitives as separate processes, whose result is later composited separately. [Whitted82] used a well defined interface in a scan line renderer to allow new primitives to be easily written in C and added to the renderer. [Fleischer88] allowed primitives to be defined as LISP functions defining the surfaces parametrically and implicitly. [Glassner93] defines primitives with C++ code, which can be linked in fairly arbitrary ways with the other stages of his renderer.

Finally, [Perlin89] extended procedural shading to define one class of new primitives, which he called *hypertexture*. Hypertexture primitives are a form of implicit function primitives, like blobs or metaballs. They are defined with a special purpose language, with the surface of the primitive defined by the zero-crossings of the hypertexture function.

In the remainder of this paper, we first examine requirements that must drive the specification of procedural primitives on any 3D Z-buffer hardware. Next, we discuss the procedural primitive specification derived from these requirements. Then we give an example of a procedural primitive written using our language. Finally, we comment on performance and present our conclusions.

2. Requirements

In this section, we examine the question of what we want our procedural primitives to do. As much as possible, we will make only general observations, applicable to both interactive and non-interactive systems. Concerns about interactivity will be deferred to section 3, where we define the specifications for our primitives.

2.1 Procedural primitive

Figure 2 shows a typical graphics pipeline. What portions of this pipeline should our proposed procedural primitive include? Some parts of the pipeline can be run without knowing what type of primitive is being rendered, while others cannot. For example, RenderMan and other procedural shading systems demonstrate that shading does not require any information about the primitive. Even if we are using a fixed shading model, say Phong shading with image textures, the shading does not rely on special knowledge of the primitive type. On the other hand, the scan conversion stage clearly requires information about the primitive. It is impossible to identify which pixels are inside the primitive without knowing the geometry of the primitive.

Those parts, and only those parts, of the pipeline that require information about the specific primitive will be included in our procedural primitive. We will now examine the stages of the pipeline to see which will be included in the procedural primitive and which will be left out.

Transformation: Transformation will (usually) not be part of the procedural primitive. Most of the time, we need to transform vectors, normals, points, and planes. In the interest of reducing the coding effort required, the code for the procedural primitive need not be involved in the transformation beyond indicating which things are vectors, which are normals, etc. Any more complex specialized transformation tasks will be handled by the procedural primitive. For example, a quadric surface renderer would have to handle the transformation of the matrix that describes the quadric.

Clipping: We cannot separate the clipping task from the domain of the procedural primitive. To see why, consider clipping a polygon. We find the intersections of the edges with each clip plane, and add or remove vertices to get a new polygon. Now consider clipping a spline patch. One method would be to find the curves of intersection between the patch and clip planes, and to use these to produce a trimmed patch. Thus, the polygon clipper cannot be used with the spline patch, and the spline patch clipper cannot be used with the polygon.

Scan conversion and visibility: These are trivially part of the procedural primitive.

Interpolation: By interpolation, we refer to interpolation of shading parameters across the primitive. These may be the many arbitrary parameters used by a procedural shader or the few specific parameters used by some hard-coded shader. Whatever the shading model, its parameters are interpolated in just a few ways. The interpolation stage uses a number of *interpolation parameters* to compute the values of each shading parameter.

Interpolation must be part of the procedural primitive. To see why, compare interpolation methods for a triangle and a sphere. Linear interpolation across a triangle is natural and makes sense. On the other hand, linear interpolation across a sphere does not make much sense — linear between where and where? Similarly, interpolation based on spherical coordinates makes sense on the sphere but is meaningless on the triangle.

Surface shading, fog and atmospheric effects: As already mentioned, we need not include shading as part of the primitive. Similarly, fog and atmospheric effects will also be excluded.

As we have now seen, to define a procedural primitive we need code for the four middle stages of the pipeline in figure 2: clipping, scan conversion, visibility, and interpolation.

2.2 Interface to the application

Not only do we need code for the procedural primitive, but we need some way for the application code to access the new primitive. This interface should be something that is easy to use. It should match, or at least be similar to the traditional methods of defining scan conversion and shading parameters for the built-in primitives. Yet it should not make too many assumptions about what a new primitive might require.

There are two parts to this problem. We need some mechanism to assign values for the parameters that control the scan conversion. One example of such a parameter is the vertex of a triangle. We also need some mechanism to control shading parameter interpolation, for example to control the linear color interpolation across the triangle. We have looked at some example primitives to gain an understanding of what is needed for both.

2.2.1 Scan conversion parameters

What parameters are needed to control the scan conversion? For insight, we'll look at the parameters used to define polygons, spline patches, spheres, and implicit surfaces.

Polygons: Polygons are defined by three or more vertices. The positions of the vertices completely define the geometry of the polygon.

Spline patches: The definition of a spline patch is based on its control points. Just as the vertex positions defined the geometry of the polygon, for some patch types the positions of the control points completely define the geometry of the patch. Other patch types also have a weight at each control point. Still others also use a knot vector.

Spheres: The geometry of a sphere is defined by a center and radius.

Implicit Surfaces: Implicit surfaces are defined by an isosurface of some 3D density function. Sometimes they are defined through the position and parameters of several density function kernels.

From these examples, we distinguish two kinds of scan-conversion parameters for the primitive. There are parameters associated with a control point (we call these *per-sequence-point* parameters for reasons described in section 3.1), such as vertex coordinates or spline weights, and *per-primitive* parameters that have a single value for the whole primitive, such as the radius of a sphere.

2.2.2 Interpolated values

What parameters are necessary to control interpolation? Once again, we'll look for insight at interpolation for polygons, spline patches, spheres, and implicit surfaces.

Polygons: A natural method of interpolating across a polygon is linear interpolation. For this, a different value is given for the shading parameter at each vertex. Other interpolation methods, such as perspective-correct, are also desirable.

Spline patches: A natural parameter interpolator for a spline patch uses spline interpolation from parameter values at each control point.

Spheres: Unlike polygons and spline patches, there is no natural way to interpolate a shading parameter across the sphere based on a value at either the center or associated with both the center and radius. However, there is an *implicit* interpolation that is independent of any points on the primitive. Surface normals on the sphere are an example of this implicit interpolation.

```
    // declare that my_shading_parameter should use linear interpolation
glMaterialInterpEXT(my_shading_parameter, GL_TRIANGLES, GL_LINEAR_INTERPOLATOR_EXT);
glBegin(GL_TRIANGLES);
  glMaterialf(GL_FRONT_AND_BACK, my_shading_parameter, shading_value0);
  glNormal3f(n0x, n0y, n0z);
    // glNormal3f(...) is equivalent to glMaterial3f(GL_FRONT_AND_BACK, GL_NORMAL, ...)
  glVertex3f(v0x, v0y, v0z);
    // similarly for other three vertices
glEnd();
```

Figure 4. Example showing extensions to OpenGL for arbitrary shading parameters.

```
glBegin(my_sphere_primitive);
  glRastParamfEXT(radius_name, radius_value);
  glRastParam3fEXT(center_name, cx, cy, cz);
  glSequencePointEXT();
glEnd();
```

Figure 3. Example showing sequence point extension to OpenGL.

Implicit Surfaces: There is no natural way to interpolate a shading parameter based on values at vertices or control points since the implicit surface has neither. Nor can we expect a handy coordinate system such as spherical coordinates. One possibility for interpolation is to have the parameter value defined in space by another function.

From these examples, we see that sometimes the *interpolation parameters* that control the interpolation must be bound at the vertex or control point, sometimes they have a single value for the whole primitive, and sometimes no value at all. To interpolate a single shading parameter may take several interpolation parameters.

3. Procedural primitives

We have used these requirements to create a specification for our procedural primitives. This section provides some of the details of that specification as well as reasoning for the decisions we made that were not mandated by the requirements.

3.1 Vertices, control points and sequence points

First, we will address the problem of the application programmer's interface (API). The PixelFlow API is an extension to OpenGL [Akeley92]. Our goal was to divorce the abstract programmable-primitive interface from the PixelFlow specific implementation, and to support the existing OpenGL primitives as a special case. In OpenGL, there is a notion of the current state of attributes such as color, normals, etc. The current attributes are bound to a vertex at the glVertex call.

We have introduced a generalized parameter-setting call, glMaterial (figure 3), to change the current parameter state of any attribute. One of the arguments to glMaterial is the name of the attribute to change. As with OpenGL, at the vertex call, the current values of all parameters are bound and used for that vertex.

This mechanism covers most of what we want. We would also like to have the ability to bind attributes as with glVertex, but without an associated point. For example, an implicit surface primitive may be defined by a series of grouped sets of blob coefficients. These coefficients must be bound as a group, but there are no vertices involved. To perform this binding function, we borrow the compiler idea of a *sequence point.* At the glSequencePointEXT call (see figure 4), the current values of all parameters are bound, as with the glVertex call. Note that the OpenGL glVertex call is equivalent to setting the vertex position parameter, then making a glSequencePointEXT call.

3.2 Special-purpose language

Should the procedural primitive be written in a special-purpose language or a general-purpose programming language such as C? We have elected to follow the example of Pixar’s RenderMan and use a special-purpose language. The language can include features specific to the writing of procedural primitives. This reduces the effort required to write each new procedural primitive. The language also allows us to hide the details of the hardware. This allows a user familiar with graphics, but not familiar with our hardware, to write new primitives. It also may make the language and interface portable to other hardware. Finally, since we compile our primitives, we can perform optimizations on the code, especially important for hardware-based interactive rendering.

The main disadvantage is that some effort must be spent learning the new language. To minimize this cost, our language is similar to our shading language, thus to the RenderMan shading language and to C.

3.3 The pixel-centric view

In addition to adopting a language similar to the RenderMan shading language, we have also adopted their *pixel-centric* point of view. A RenderMan surface-shading function is written as if it were running on only a single sample on the surface. Its job is to determine the color of that one sample. The 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.

Our procedural primitive is also written as if it were running at a single pixel. Its job is to determine whether that pixel is in the primitive, whether the pixel is visible, and what value each shading parameter takes there. Compare this to the incremental approach where the procedural primitive steps from one pixel to the next and one scan line to the next. Most scan conversion algorithms have traditionally been defined in this incremental fashion. One disadvantage of the incremental scheme is that the code for the primitive needs to know about the image sampling.

One example is the traditional incremental polygon scan conversion algorithm. The algorithm follows the edges from one scan line to the next, and interpolates depth and shading parameters from one pixel to the next across the scan line. One such algorithm was introduced by Pineda [Pineda88], and has been used by Silicon Graphics [Akeley93]. This algorithm uses a linear function ($ax + by + c$) of the pixel coordinates for each edge. This function is positive inside the edge and negative outside. The values of the edge functions are incrementally computed by adding b for each step from one scan line to the next and a for each step from one pixel to the next in a line. When an edge function changes sign, we know that we've crossed outside the polygon. This algorithm can be cast, instead, in a pixel-centric form. Then an individual pixel is determined to be in the polygon if all edge functions are positive. This can also be thought of as the pixel-centric version of the Pixel-Planes scan conversion algorithm [Fuchs82], where the polygon is defined implicitly as the region where all edge functions are positive.

Some RenderMan surface shaders (notably the wall paper in Pixar's KnickKnack) operate by drawing geometric shapes on the surface. It is satisfying to note the consistency between the pixel-centric view used by these mini-scan converters in the surface shader and the pixel-centric view of our procedural primitives.

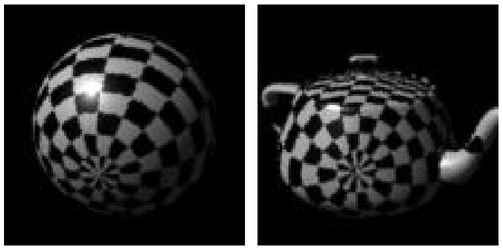

Figure 5. Primitive independent interpolation of texture coordinates.

3.4 Sharing interpolators

Earlier, we decided that shading parameter interpolation should be part of the procedural primitive. This was based on a two examples of interpolators that work only for a specific primitive. While interpolators that work across the surface of the primitive typically cannot be generalized to work for all primitives, some interpolators can work on any primitive. We would like to be able to share these general interpolators across all primitives.

What allows an interpolator to be shared with any primitive? It cannot depend at all on the parameters that define the primitive, on the number of sequence points, or on the per-sequence point parameters. All that is left are per-primitive parameters. Obviously these can be used to define constant values, though that's not terribly interesting. They can also be used to define arbitrary value fields in space. This is equivalent to the Ebert's *solid spaces* [Ebert94]. Using this technique, a surface shader can be turned into a volume shader, as is shown in Figure 5. The texture coordinates are determined by two functions which are defined everywhere in space. Any surface can use these texture coordinates, no matter what primitives generated the surface. Note that some of the shading parameters for the surface may be generated by primitive-specific interpolators, while others may be generated by primitive-independent interpolators.

4. System accommodations

The previous sections describe a language and an interface that are independent of the fact that, in our prototype, they are running with graphics acceleration hardware. There are a few specific requirements made of the hardware and of the software to accommodate hardware-based rendering.

```
primitive polygon(int vertex_count;                          // number of vertices
                  Point vertex[];                            // vertices — transformed since they are a point type
                  interpolator_data interp_data[];           // when this is used, it is really applied to all
                                                             //   interpolation parameters
                  varying fixed<32,16> X, Y;                 // screen coordinates (passed into screen_triangle)
                  inout varying fixed<32,32> Z)              // Z-buffer (passed into screen_triangle)
{

  // check clipping against hither plane
  for each vertex
    check vertex against hither plane

  // draw triangle fan
  for each vertex
    if vertex is unclipped
      divide by W to get 3D non-homogeneous vertex
      add vertex to screen triangle
      add interp_data value for this vertex to screen triangle
      if there are three vertices in the current screen triangle
        call screen_triangle to render it
        reset current screen triangle for the next triangle in the fan
    if current vertex and next vertex are on opposite sides of hither plane
      compute new 3D non-homogeneous vertex at intersection with clipping plane
      add this vertex to screen triangle
      for linear or perspective_linear interpolators
        add new interp_data value at clipping plane
      if there are three vertices in the current screen triangle
        call screen_triangle to render it
        reset current screen triangle for the next triangle in the fan
}
```

Figure 6. Polygon primitive pseudo-code. Important points: vertices are pre-transformed since they are listed with a Point type; any operations performed with the interp_data are applied for all per-vertex interpolation parameters; further computations that should also apply to all per-vertex parameters use the interpolator_data type.

Modern interactive graphics hardware must make heavy use of parallelism in the rasterization stage of the pipeline to achieve the expected performance. Our first requirement is that these processors be programmable. Our second requirement is that they have enough memory to execute interesting primitives.

We have added a fixed-point arithmetic construct to the shading and procedural-primitive language in order to decrease the time spent in computations. In addition to the RenderMan float, we have a fixed data type with length (in bits) and fractional arguments. The parameters X, Y, and Z in figure 6 are all declared as fixed.

The software framework has also been designed with procedural primitives in mind. For generality, all primitives, including the "built-in" ones must register themselves with the software system at startup time. All primitives are also driven by a general parameter setting mechanism. This generality is invisible to the user since the standard OpenGL constructs and primitives are all supplied.

5. Example primitives

We will show how a generalized polygon primitive might look if programmed using our language. We show two versions of the polygon primitive. The first turns the polygon into screen-space clipped triangles, demonstrating a primitive that is decomposed into simpler primitives. The second part renders screen-space triangles directly.

Figure 6 is pseudo-code for the polygon primitive. It is assumed that the vertices are coplanar and the polygon is simple and convex. Note that this polygon handles two kinds of interpolation. It can perform perspective corrected or uncorrected linear interpolation. The basic algorithm is to clip the polygon against the hither plane, then split into a fan of triangles.

Figure 7 is pseudo-code for the screen-space triangle primitive. Remember that the procedure executes as if it were running on at a single pixel. The pixel's location is found in the X and Y parameters. As with the RenderMan shading language, varying variables are those that have different values at different

```
primitive screen_triangle(float screen_vertex[3][3];          // vertices — not point type, not transformed
                          interpolator_data interp_data[3];   // shorthand for all interpolation parameters
                          varying fixed<32,16> X, Y;          // screen coordinates
                          inout varying fixed<32,32> Z)       // Z-buffer
{
    for each pair of vertices
        compute edge expression
        return if pixel is outside of edge

    compute triangle Z
    return if triangle Z < Z buffer
    Z = triangle Z

    for perspective-corrected linear interpolation
        compute interpolation value
    for uncorrected linear interpolation
        compute interpolation value
    for all other types of interpolation
        execute shared interpolators here
}
```

Figure 7. Pseudo-code for screen_triangle primitive.

```
// declaration
interpolation_data screen_data[3];
...
// if vertex is unclipped
//   add interp_data for this vertex to screen triangle
screen_data[screen_vertex_count] = interp_data[current_vertex];
...
// if current vertex and next vertex are on opposite sides of the hither plane
//   for linear or perspective_linear interpolators
//       add new interp_data value at clipping plane
interpolation {
    perspective_linear:
    linear:
        screen_data[screen_vertex_count] = (1-t) * interp_data[current_vertex] + t * interp_data[next_vertex];
}
```

Figure 8. Interpolation details from polygon primitive.

pixels. Only the varying computations need be performed at every pixel. All other computations are done once for all the pixels.

The subroutine, screen_triangle, tests the pixel's position against each of the edges and the Z-buffer. If the pixel is outside the triangle or fails the Z-test, the primitive function returns without changing the Z-buffer or interpolating any shading parameters for that pixel.

5.1 Dealing with interpolated shading parameters

There are four main features of the language designed to support shading parameter interpolation.

The first is the interpolation_data type. This type is used by the primitive in any shading parameter computations. Any computations using the interpolation_data type are applied to all per-vertex interpolation parameters. In other words, for any expression using this data type, our compiler generates code to compute the same expression, using either fixed point or floating point as appropriate, for arbitrary sets of shading parameters. The polygon primitive uses this to compute interpolation parameter values at the hither clipping plane. For these computations, it uses a variable of the interpolation_data type called screen_data. This can be seen in figure 8.

The second feature is the magic interp_data parameter to the primitive. This parameter is of the interpolation_data type. It holds the interpolation-parameter data for all of the per-vertex shading parameters. This can be seen in figure 6 and figure 7. The per-primitive parameters are not mentioned because the primitive itself does not need to do anything with them.

The third feature is the interpolation construct. This looks similar to a C switch statement, but the branches are the different kinds of interpolation that

```
interpolation {
    // for perspective corrected linear interpolation
    //   compute interpolation value
    perspective_linear:
        a = interp_data[0]/vertex[0][2] * (vertex[1][1] - vertex[2][1])
            + interp_data[1]/vertex[1][2] * (vertex[2][1] - vertex[0][1])
            + interp_data[2]/vertex[2][2] * (vertex[0][1] - vertex[1][1]);
        b = ...
        c = ...
        interpolator_return(a * X + b * Y + c);

    // for uncorrected linear interpolation
    //   compute interpolation value
    linear:
        a = interp_data[0] * (vertex[1][1] - vertex[2][1])
            + interp_data[1] * (vertex[2][1] - vertex[0][1])
            + interp_data[2] * (vertex[0][1] - vertex[1][1]);
        b = ...
        c = ...
        interpolator_return(a * X + b * Y + c);

    // for all other types of interpolation
    //   execute shared interpolators here
    default:
        interpolator_return;
}
```

Figure 9. Interpolation details from screen_triangle primitive.

```
interpolator cylindrical_interpolation(float cylinder_transform[4][4] = {{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}},
                                       float cylinder_start = 0, cylinder_end = 1;
                                       fixed<32,4> X, Y;
                                       fixed<32,32> Z)
{
    float screen_space_point[4] = {X, Y, Z, 1};
    float cylinder_space_point[4] = cylinder_transform * screen_space_point;
    interpolator_return((cylinder_end - cylinder_start) * atan(cylinder_space_point[1], cylinder_space_point[2]) / 2 / PI
                        + cylinder_start);
}
```

Figure 10. A simple shared interpolator

the primitive handles. For example, in figure 9, there are branches for linear, and perspective_linear, the two kinds of interpolation the screen_triangle primitive supports. There is also a default branch, which can be used for actions for any kinds of interpolation not known to the primitive. Figure 8 and figure 9 both show examples of the interpolation construct in use.

The final feature is the interpolation_return statement. It is typically used inside an interpolation construct to set the interpolated shading-parameter values. The action of interpolator_return is similar to the action of the interpolation_data type. When it executes, it sets the actual values of the shading parameters for each parameter in the primitive using that particular kind of interpolation. Figure 9 has three interpolator_return statements: one for the shading parameters using perspective-correct linear interpolation, one for the shading parameters using uncorrected linear interpolation, and finally one (with no value given) for the any shading parameters using shared interpolators.

5.2 Example of a shared interpolator

Figure 10 shows an example of a simple shared interpolator. This interpolator produces shading parameter values based on a cylindrical coordinate system. It is similar to a primitive, but its only result is a single value, given in an interpolator_return statement.

6. Performance

We expect our compiled primitives to be fast enough for prototyping and low-volume uses. In applications requiring hundreds of thousands of instances of a custom primitive, we expect that it will be further hand-tweaked (or even coded in assembly language) after prototyping is complete.

Performance of a user-programmable primitive coded in the high-level language will be directly related to the optimizations performed by the compiler. Not all of the optimizations that we plan to support have been written. We expect to see steadily increasing performance as the compiler matures.

As mentioned earlier, we currently have PixelFlow boards in hardware testing and software running in simulation. Therefore, we do not have performance numbers at this time. By the final paper deadline, we expect to have an operational machine to run our procedural primitives. Procedural primitives have been a goal of the project throughout its development. Even our fast, hand optimized primitives use the same internal interfaces as the procedural primitives. At system initialization, the built-in primitives are installed in the same fashion as user-defined primitives.

7. Conclusions

We have designed extensions to a special-purpose language and to an existing graphics API for creating new primitive types and have built a prototype implementation for the PixelFlow 3D graphics machine. To design the interface, we examined the general characteristics required to support a variety of procedural primitives for a Z-buffer renderer. Our interface is a direct result of this analysis and of some concerns mandated by the nature of general graphics-acceleration hardware. As a result, we have an interface that does not require the user to have detailed knowledge of the underlying graphics hardware.

8. References

[Akeley92] Kurt Akeley, K. P. Smith, J. Neider, *OpenGL Reference Manual*, Addison-Wesley, 1992.

[Akeley93] Kurt Akeley, "RealityEngine Graphics", *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, August 1993, pp. 109–116.

[Amburn86] Phil Amburn and Eric Grant and Turner Whitted, "Managing Geometric Complexity with Enhanced Procedural Models", *Computer Graphics (SIGGRAPH '86 Proceedings)* , volume 20(4), August 1986, pp. 189–195.

[Crow82] F. C. Crow, "A More Flexible Image Generation Environment", *Computer Graphics (SIGGRAPH '82 Proceedings)* , volume 16(3), July 1982, pp. 9–18.

[Ebert94] David Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin and Steven Worley, *Texturing and Modeling: A Procedural Approach*, Academic Press, 1994.

[Fleischer88] K. Fleischer and A. Witkin, "A Modeling Testbed", *Proceedings of Graphics Interface '88* , Canadian Inf. Process. Society 1988, pp. 137–137.

[Fuchs82] Henry Fuchs, John Poulton, "Pixel-Planes: A VLSI-Oriented Design for a Raster Graphics Engine," *VLSI Design*, 2(3), 1982, pp. 20-28.

[Fuchs89] Henry Fuchs and John Poulton and John Eyles and Trey Greer and Jack Goldfeather and David Ellsworth and Steve Molnar and Greg Turk and Brice Tebbs and Laura Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories", *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, 1989, pp. 79–88.

[Glassner93] Andrew S. Glassner, "*Spectrum*: An Architecture for Image Synthesis Research, Education, and Practice", *SIGGRAPH '93 Developing Large-scale Graphics Software Toolkits seminar notes*, 1993.

[Green88] Mark Green and Hanqiu Sun, "MML: A language and system for procedural modeling and motion", *Proceedings of Graphics Interface '88*, June 1988, pp. 16–25.

[Hall83] R. A. Hall and D. P. Greenberg, "A Testbed for Realistic Image Synthesis", *IEEE Computer Graphics And Applications*, volume 3, November 1983, pp. 10–20.

[Hanrahan90] Pat Hanrahan and Jim Lawson, "A Language for Shading and Lighting Calculations", *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, August 1990, pp. 289–298.

[Hedelman84] H. Hedelman, "A Data Flow Approach to Procedural Modeling", *IEEE Computer Graphics and Applications* , volume 3, January 1984, pp. 16–26.

[Kolb92] Craig E. Kolb, *Rayshade User's Guide and Reference Manual* , January 1992.

[Kuchkuda88] Roman Kuchkuda, "An Introduction to Ray Tracing", *Theoretical Foundations of Computer Graphics and CAD* , volume F40, Springer-Verlag 1988, pp. 1039–1060.

[Lastra95] Anselmo Lastra, Steven Molnar, Marc Olano and Yulan Wang, "Real-Time Programmable Shading", *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, ACM SIGGRAPH, 1995, pp. 59 – 66.

[Molnar92] Steven Molnar, John Eyles and John Poulton, "PixelFlow: High-speed Rendering using Image Composition", *Computer Graphics (SIGGRAPH '92 Proceedings)* , volume 26, 1992, pp. 231–240.

[Perlin89] Ken Perlin and Eric M. Hoffert, "Hypertexture", *Computer Graphics (SIGGRAPH '89 Proceedings)* , volume 23, July 1989, pp. 253–262.

[Pineda88] Juan Pineda, "A Parallel Algorithm for Polygon Rasterization", *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, August 1988, pp. 17–20

[Rubin80] S. M. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes", *Computer Graphics*, volume 14(3), July 1980, pp. 110–116.

[Upstill90] Steve Upstill, *The RenderMan Companion*, Addison-Wesley 1990.

[Whitted82] T. Whitted and D. M. Weimer, "A Software Testbed for the Development of 3D Raster Graphics Systems", *ACM Trans. on Graphics (USA)*, volume 1(1), January 1982, pp. 43–57.

[Wyvill85] Geoff Wyvill and Tosiyasu L. Kunii, "A Functional Model for Constructive Solid Geometry", *The Visual Computer* , volume 1(1), July 1985, pp. 3–14.