2 ABSTRACT INTERFACE

To support procedural interfaces throughout the graphics pipeline, we need a reasonable decomposition into stages where we may substitute user-defined procedures. These stages represent the programmer's view of the system and may not directly correspond to the hardware pipeline stages. We call this programmer's view the *abstract pipeline*. Any procedural interfaces supported by a system are implemented by the system designers to look like a stage of the abstract pipeline. The abstract pipeline defines only the input and output of each stage, so a hardware implementation is free to add additional hidden stages or change the execution order of the stages as long as the stage input and output do not change.

We want it to be possible to use the same procedures on different graphics systems or different generations of PixelFlow hardware, so we need to ensure that the decomposition is general and device-independent. Once we have a general, device-independent abstract pipeline, an individual graphics system can support one, several, or all of the procedural stages. Even if two systems do not support different subsets of the stages, procedures for stages they share will be usable by both.

In this chapter, we develop an abstract pipeline that satisfies these needs. In Section 2.1, we look at some of the fully procedural systems that have been created in the past. In Section 2.2, we discuss device independence and how we achieve it. In Section 2.3, we present our abstract pipeline. Finally in Section 2.4, we look at each stage in more detail, using the abstract pipeline as a taxonomy to classify the prior work.

2.1. Prior systems

Varying degrees of programmability have been provided at points in the graphics pipeline. The examples range from high-level *shading languages* down to *testbed* graphics systems that are designed to be reprogrammed only by the original author. Both can shed light on what procedures should be included in a fully programmable pipeline.

2.1.1. Testbeds

Most experiments with programmability, even for off-line renderers, have been testbed systems. These are designed to allow the system creator to experiment with a variety of shading and rendering techniques. Creating a testbed system requires a certain sophistication in graphics. A useful testbed requires a good characterization of the common themes of all of the rendering or shading techniques that will be explored, yet must not be so restrictive that new ideas cannot be tried. The testbed is created as a modular rendering system based on these assumptions. New shaders, primitives, or transformations are implemented as replacement modules, generally in the same programming language used for the rest of the testbed. As long as the new modules obey the assumptions of the testbed, they are easy to create and investigate. To some degree, every rendering system that is not implemented entirely in hardware is part testbed, but we are interested in the ones that were planned that way.

Whitted and Weimer published a description of one of the earliest systems designed explicitly as a testbed [Whitted82]. They supported experimental primitives and shaders. Each primitive supplied the first scan-line where it would become active. At this scan-line, it could either render using a scan-line algorithm or split into simpler pieces, each with its own starting scan-line. For shading, the primitive would *interpolate* arbitrary *shading parameters,* producing values at each pixel. A separate shading process would compute a color for each pixel. Since rendering could be slow and costly, a single set of rendered shading parameters could be reused for tests of several different shading algorithms. Crow presented a system that used a separate UNIX process for each primitive [Crow82]; and Potmesil and Hoffert presented one that used a UNIX process for each stage in the graphics pipeline [Potmesil87].

Hall and Greenberg [Hall83] and Trumbore, Lyttle and Greenberg [Trumbore93] produced testbed systems for Cornell’s global illumination research. These frameworks allowed easy testing of different *light transport* algorithms. The Hall and Greenberg system was based on ray tracing, and was similar to the ray-tracing testbeds that will be described next, though with different goals. The Trumbore, Lyttle, and Greenberg system was interesting in its reversal of the normal testbed organization. It provided a collection of library routines that could be called by user code. This places the user's code in charge of the scheduling of the different rendering tasks. This makes it more like the data-flow systems described later in this section.

A modular framework is built into the ray tracing algorithm. Rays are shot into the scene. Each primitive reports its closest intersection with a ray. Once a closest intersection has been chosen from among the primitives, either a color or shading parameter values are produced at the intersection point. As a result of this built-in framework, there have been a number of testbed ray tracers. The [Hall83] system already mentioned is one of these. Others include [Rubin80], [Wyvill85], [Kuchkuda88], and [Kolb92]. Rubin and Whitted provided an environment for testing bounding volume hierarchies and surfaces defined through recursive subdivision (i.e. the surface can be split into smaller pieces, which can be split into still smaller pieces, which eventually can be converted into triangles or some other directly ray-traceable primitive). Wyvill and Kunii had a system for ray tracing complex CSG (constructive solid geometry) objects. The complex objects are built of Boolean combinations of simpler objects, in their case, defined implicitly (by an arbitrary function – the surface is as the locus of points where the function). Kuchkuda created a simple modular ray tracer in C, which was expanded by Kolb into his popular, free Rayshade ray tracer.

Finally, there have been a number of *data-flow* testbeds. These allow changes to the order of the pipeline of graphics operations. The pipeline stages can be hooked together into any reasonable order. This effectively allows a single testbed renderer to support several different rendering algorithms. A data-flow testbed also allows other interesting reorganizations; adding an extra transformation just before shading, for example.

Hedelman created a data-flow-based system allowing procedural models, transformations, and shaders to be connected together [Hedelman84]. Fleischer and Witkin wrote a conceptually elegant LISP system, allowing fully programmable LISP modules to be connected together arbitrarily [Fleischer87]. It has a generalization for transformations which we will adopt. It represents each procedural transformation as a set of four functions that compute the transformation and inverse transformation for points and vectors. This system also requires all surfaces to have both implicit and parametric formulations so the system can be free to choose either ray-tracing or scan-line rendering. The only interpolated shading parameters available in this system are the 3D surface position, the parametric surface position, and the surface normal. This limits the shading possibilities as compared to some of the more advanced procedural shaders, which will be discussed in Section 2.4.5. [Nadas87], [Glassner93] and [Trumbore93] are other data-flow systems, all of which allow C or C++ procedures to be connected together in arbitrary ways.

2.1.2. User-programmable systems

Sometimes there is very little difference between a testbed system and one designed to be user-programmable. All of the concerns of the testbed still hold, particularly the need for a general, non-restrictive framework for the parts that will be replaced. However, a user-programmable system has extra considerations. The user will not be as familiar with the workings of the rendering system as a whole. Therefore, an easy to understand interface is necessary. This is one of the main reasons that a simpler language is used (often called a shading language, even when used for more than shading).

RenderMan

The RenderMan standard is presented in [Hanrahan90] and described in detail in [Upstill90]. RenderMan provides a geometry description library similar to OpenGL, a geometric file format (called RIB), and a shading language. The library (or API for application program interface) is used by application developers to describe the geometry to be rendered. There is a one-to-one correspondence between the library function calls and the lines in a RIB file. In Pixar’s PhotoRealistic RenderMan, calls to the library generate the RIB file. The renderer reads the RIB file to render the scene. RIB files are also used by other animation and modeling applications to create geometric descriptions that can be rendered with the RenderMan renderer.

Several implementations of the RenderMan standard exist, though all are off-line renderers. The first implementation, and the one against which all others are judged, is Pixar's own PhotoRealistic RenderMan. PhotoRealistic RenderMan uses the *REYES* rendering algorithm [Cook87]. There is also an implementation using ray-tracing called the Blue Moon Rendering Tools (BMRT) [Gritz96], and another using a global-illumination rendering algorithm [Slusallek94]. Each implementation has its own strengths, but the same application will run on all three without change to produce comparable, if not identical, images. RenderMan effectively hides the details of the implementation. Not only does this have the advantage of allowing multiple implementations using completely different rendering algorithms, but it also means that the user writing the application and shaders doesn't need to know anything about the rendering algorithm being used. Knowledge of basic graphics concepts suffices. This is a huge departure from normal testbed systems, where the programmer who writes the procedural pieces needs to know both details of the rendering algorithm and details of the implementation.

One key to this generality is the shading language. The shading language is a high-level language, similar to a subset of C, with extensions useful for shading and other procedures. It is designed to handle transformations, deformations (small-scale perturbations of the actual geometry), lighting, surface shading, and atmospheric effects.

Pixar also uses a procedural interface for animation, called MENV. MENV has been described in [Reeves90], though it is not part of the RenderMan standard or product.

2.2. Device independence

At the beginning of this chapter, we mentioned the desire for a device-independent interface for procedures. This would enable the use of the same procedures on multiple graphics machines. We attack device-independence on two fronts. First, we use a special purpose language to write all of our procedures. Second, we place these procedures into an abstract pipeline. The design of the abstract pipeline is based on a partitioning of the rendering process into logical procedures, instead of on the physical organization of our graphics system.

2.2.1. Using a "shading" language

We use a special purpose language to write procedure for all stages of the abstract pipeline. This language looks similar to C or the RenderMan shading language, minimizing the effort required to learn it. It also contains special purpose constructs to make it easier to write procedures for the different stages. In the literature, since [Cook84], languages used for simple procedures anywhere in the graphics pipeline have been called *shading languages*, and the procedures, *shaders*. This nomenclature can cause confusion, so we will try to avoid it, except where it is necessary to relate our work to the prior work.

Whatever it is called, a special purpose language has advantages, both from the procedure-writer's point of view, and from the graphics-system architect's point of view. It allows a compiler to hide the details of the actual rendering system. Code written without such an intermediary must use the low-level interfaces provided by the graphics system. The compiler can convert standard graphics paradigms into the interface required by the hardware. Our compiler also fills in tables used by the rendering system to tell what the procedural parts are doing and what parameters they use, and it looks in the rendering system data structures to find the shader parameters to pass to the shader. In general, it hides the internal implementation from the user.

Since graphics hardware usually has limited resources, a compiler can also perform optimizations necessary to allow procedures to run effectively on the hardware. The users could make some of these same optimizations themselves. However, this would reduce the portability of the procedure code and make it much more difficult to do any kind of rapid prototyping and modification. For example, our system is quite short on memory. Without the compiler's memory optimization, many shaders would not fit. Such memory optimization can be done by hand; but after spending a day or two optimizing memory for a shader, the shader writer is not going to want to make any major code changes.

Finally, our language has the same benefits as the RenderMan shading language [Hanrahan90]. It provides a uniform interface for writing shaders or other procedures that is simpler than a full-blown language, yet also has special constructs to make procedures easier to write. Thus the user’s programming effort is reduced, making shader development and rapid prototyping easier.

2.2.2. Abstract models

Once we place any kind of programmable processor into an interactive graphics system, we have achieved the goal of having programmable graphics hardware. However, programming at this low level requires intimate knowledge of the particular hardware. This is the realm of the graphics library implementers, not the user. Porting software written for such a system to other graphics systems is difficult at best. In addition, supporting the same internal interface on later generations of the same graphics hardware would put a heavy burden on the hardware system designers.

Pixel-Planes 5 [Fuchs89] is an example of a machine where user written code was allowed to run on the internal processors of the graphics engine. However, even though the Pixel-Planes 5 graphics library interface is well defined for the application running on the host computer, the interface for writing code to be run on the hardware itself is not as well defined. In fact, this interface was created after the system was operational based on user demands. Consequently, programming the Pixel-Planes 5 hardware requires detailed knowledge of the Pixel-Planes 5 hardware and the internal implementation of the graphics library. In many cases, it even requires writing code for custom chips found only in Pixel-Planes 5. Our users wrote code for this environment anyway, because it was the only way for them to do what they wanted to do. Several examples of their efforts are given in Section 5.1.

We would like an interface that is more general than was found in Pixel-Planes 5, both for the users sake and for our sake as graphics system designers. To achieve device-independence, we require an abstract model of the graphics process. This model must be high-level so it can be useful on many architectures, and so it can be usable by someone with an average knowledge of graphics. Countering the desire for a high-level interface, the model must remain low-level enough to be implementable and fast.

For our abstract model, we use a version of the common *graphics pipeline* [Foley90]. This is the set of independent processes that converts 3D geometry information into pixels on the screen. For speed and implementability, we restrict our scope to cover only z-buffer based graphics hardware.

Just because the steps in the graphics process are described as a pipeline, they do not have to be implemented as an actual pipeline with the stages given. It may be more appropriate to think of it as the *graphics flow chart*. We will continue to use the common name, graphics pipeline, with the understanding that any hardware pipelining need not conform to the stage divisions of our abstract pipeline. The stages allow us to define the interfaces for the different procedures. If a particular graphics system follows the interface recommendations, the procedures will be usable by other graphics systems that support procedural access to the same stages.

2.3. Abstract Pipeline

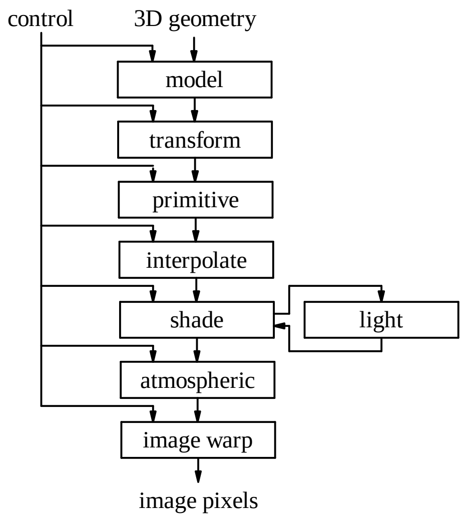

Figure 2.1. Abstract pipeline for procedural rendering.

Figure 2.1 shows the pipeline our procedural rendering system will use. Each stage can be implemented as a procedure. The procedures are orthogonal, so a new procedure for one stage will not require any changes to the procedures for the other stages. A particular hardware implementation can therefore just allow procedural access to those stages that it is able to support. Not every hardware system will be programmable in every stage, yet we still want a consistent interface for those stages that can be programmed.

2.4. Pipeline stages

In the following sections, we provide a brief description of each of the stages in our abstract pipeline. We also provide a brief survey of prior work for each procedure classification.

2.4.1. Modeling

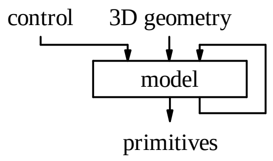

Figure 2.2. Model stage.

Modeling is the construction of objects and scenes out of basic geometric primitives. Usually, objects are defined as a rigid set of primitives. A procedural interface can be used to define rigid models, but it is more interesting for models that move, change shape, possess awareness of their environment and automatically interact with it, or possess awareness of their viewing conditions and change accordingly.

Procedural models use a set of control parameters to generate a description of the model in terms of geometric primitives or other procedural models. Most polygon based systems implement spline patches in this way. These smooth patches are defined by a small set of control points. For rendering, the patch is first split into smaller patches. When the patches become small enough, they are rendered as polygons. A few other possibilities for procedural models are particle systems [Reeves83], fractals [Mandelbrot77] [Hart89], L-systems [Prusinkiewicz88], hypertextures [Perlin89], and generative models [Snyder92].

The graphics API library itself usually serves as a procedural model interface. However, several systems have been created that provide an alternate form for defining procedural models [Newell75][Hedelman84][Amburn86][Green88][Perlin89][Upstill90]. Each of these will be covered in more detail below.

Newell's procedure models were an object-oriented representation (in the data-structures sense) of the objects in the scene. Each model is an opaque entity, capable of responding to two messages: draw the object and provide an approximate 3D extent for the object.

Hedelman's procedural models could be linked with transformations and shaders into a tree. He presents procedural models capable of smart culling. Such a model can use viewing information to choose one of several representations at different levels of detail.

Amburn, Grant and Whitted described two ideas for procedural modeling. They present a generalization of subdivision, where a representation is subdivided until a transition test triggers a change to a different representation. The new representation can be another procedural model or a set of primitives. The spline algorithm mentioned previously is one example of such a model – it subdivides into similar spline patches until some test triggers the change to a polygonal representation. They also used communication between procedural models. Two interacting model procedures send messages to each other to adjust each other.

Green and Sun use a pre-processor to C called MML to generate rule or grammar-based models. For example, a tree might be defined with a rule to split a branch into a trunk and two branches. By recursively applying the rules, they can create a large model.

Perlin is one of the leaders in application of procedural techniques to graphics. In 1989, Perlin and Hoffert presented *hypertexture*, a modeling technique where an ordinary surface is thickened into a region of space with some density. A procedure modulates the density, creating a 3D texture effect on the surface. The hypertexture surface can then be rendered by traditional volume-rendering techniques [Watt92].

The RenderMan API, as documented by Upstill, includes procedural models. These are similar to Newell's procedure models. A RenderMan procedural model is created through a pair of functions. One to subdivide into other models or primitives, and the other to give a bound on the volume that the model will occupy. The latter function aids the renderer in knowing when to render the model. Only recently has any RenderMan implementation supported procedural models as defined by the standard. The application and RenderMan API library communicate with the renderer through a RIB file, and may not even run at the same time. This makes it is difficult for the renderer to call the procedural model functions. Only the latest release of Pixar's PhotoRealistic RenderMan supports procedural models through dynamic loading of the procedural model code by the renderer.

We have summarized several systems that provide a specialized interface for procedural modeling. These systems all fit our stage definition to produce geometric primitives or other procedural models from the input parameters. Only a handful of such systems, providing a specialized interface exist, but procedural modeling using the graphics API is widely used on both interactive and off-line rendering systems.

2.4.2. Transformation

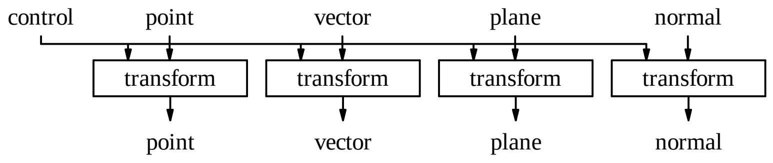

Figure 2.3. Transform stage.

Transformations are mappings of an object from one coordinate system to another. They are used to build models from component parts, for object motion, and ultimately to find the object's projection onto the screen. A transformation procedure takes a 3D point or vector as its input and produces a new 3D point or vector.

The most common set of transformations used for graphics are the simple linear mappings of homogeneous points. For 3D points, (x, y, z), the basic linear mappings create a new set of x, y, and z coordinates as a linear combination of the original x, y, and z. These linear mappings allow rotation and scaling of objects. With a homogeneous representation, (x, y, z, 1), the linear mappings can be extended to include translation and perspective projection [Foley90].

Since transformation is just a mapping between coordinate systems, there is no reason to use only linear mappings. Because they deform the objects, nonlinear transformations are more commonly known as deformations. Two different kinds of deformations are in fairly common use ([Barr84][Sederberg86]). These are sometimes used for static modeling (positioning and sculpting of the parts of a model), but more often for animation of flexible objects.

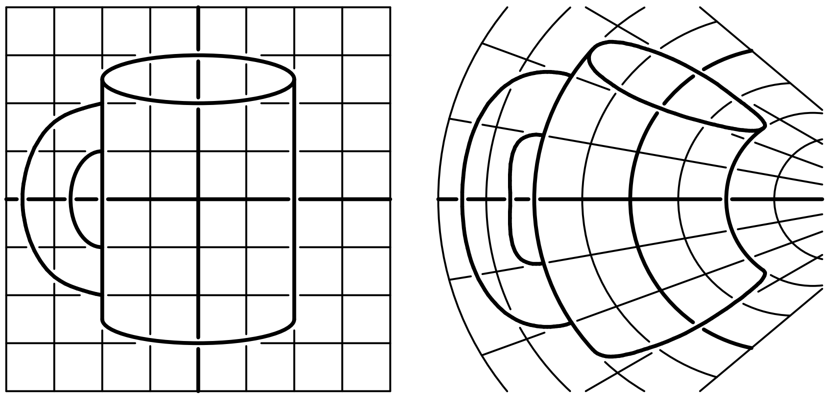

Figure 2.4. Bend deformation in 2D.

a) before deformation, b) after deformation

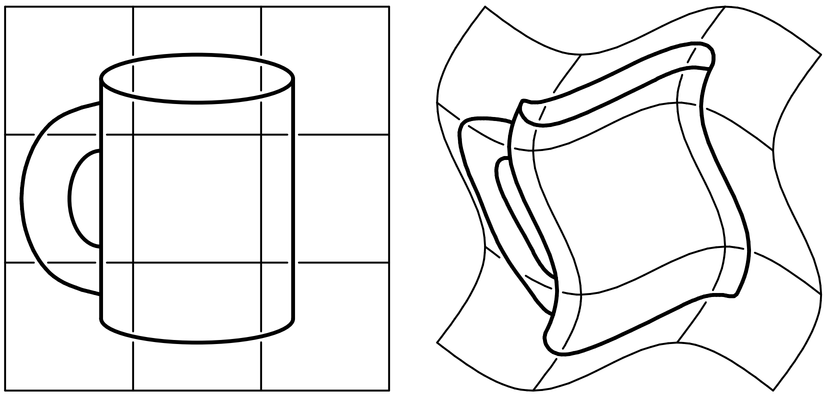

Figure 2.5. Bicubic free form deformation in 2D.

a) before deformation, b) after deformation

Barr defines global and local deformations based on standard linear transformations where the transformation parameters are simple functions of position. For example, a twist around the Y axis is defined as a rotation around the Y axis where the rotation angle is proportional to the Y coordinate. This is more easily seen in a 2D example (Figure 2.4). This figure shows a bend of the Y axis, defined as a rotation around a point on the X axis, where the rotation angle depends on the Y coordinate. This illustration also shows one of the common features of all transformations and deformations. The transformation deforms the space itself. The object is only transformed by virtue of its embedding in the space.

Sederberg defines *free form deformations* (FFDs) as a spline warping of the space around an object. Just as a spline patch defines a mapping from a 2D parameter space to positions on the patch, a spline volume defines a mapping from a 3D parameter space to a 3D point in the volume. Thus, it is a mapping from 3D to 3D, and can be used as a transformation (Figure 2.5).

The full range of deformations is much larger than the subset covered by these two types. In addition, the Barr and Sederberg deformations do not always provide very intuitive control. A tricubic FFD has 64 control points (4^3). In practice, simplified parameterization of their controls are preferred [Watt92][Reeves90]. For example, a single bending parameter might control the positions of the FFD control points. Procedural transformations can support the full set of possible transformations, while also providing more intuitive controls.

The progression of more and more complex shading methods has been used as a justification for procedural shading [Hanrahan90]. A fixed, non-procedural shading model is sufficient for some surfaces, but will lack features needed to handle the surface shading of others. A more complex fixed shading model will be able to handle still more surface characteristics, but there will always be surfaces it cannot handle. With procedural shading, a procedure can be written to match the fixed shading model, but new procedures can also be written to handle new types of surfaces. A similar argument applies to procedural transformations. Rigid motion is the most basic of transformations. On top of the rigid motion, we can add uniform scaling. Non-uniform object scaling includes and extends the results possible with uniform scaling. 3D linear transformations includes rotation, uniform and non-uniform scaling, as well object shearing. Homogeneous linear transformation incorporates all of these, and adds perspective. Deformations can do all of these, but also add the ability to bend and warp the object. Procedural transformation encompasses all of these and more.

Some definitions and implementations of procedural transformation exist. Fleischer and Witkin [Fleischer88] define true user-accessible programmable transformations as a set of four functions. These four functions provide transformations for points and vectors, and the inverse transformation for each. The inverse transformations are necessary for correctly transforming planes and surface normals [ONeill66]. They are also useful for transforming from the distorted space back to the undistorted object.

The RenderMan specification defines transformation shaders [Upstill90], though they are not implemented in Pixar's PhotoRealistic RenderMan, or any of the other RenderMan implementations. The transformation shaders specified by RenderMan include only transformations of points and not normals, planes, or vectors. Version 3.5 of the RenderMan specification [Pixar97] adds vector and normal transformation functions for use by the shading procedures, however the transformation procedure specification did not change accordingly. In a pinch, RenderMan's displacement mapping can be used as a form of procedural transformation. Pixar's MENV system allows procedural control of the parameters of the classic types of deformations [Reeves90].

Interactive graphics systems are limited to just the linear homogeneous transformations. Interactive deformations have been created by having the application do the transformations instead of the graphics hardware. Off-line renderers often use one of several fixed forms of deformation, but only a handful of systems have attempted anything more general.

2.4.3. Primitives

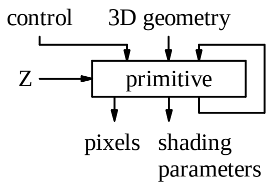

Figure 2.6. Primitive stage.

Primitives are the basic building blocks, the atoms, of the graphics system. They are the basic units that are rendered directly. A primitive procedure decides which pixels are inside the primitive. It may also use a specialized method to compute values for some shading parameters.

The Pixel-Planes 5 graphics system provides good anecdotal support for the need for procedural primitives; covered in more detail in Chapter 5. No user-level support was provided for writing primitives, yet a number of users went through the trouble to write their own custom primitives.

Given the number of primitive types currently available, the chances are good that any given rendering system will have missed several of them. A few possibilities are polygons, spline patches, spheres, quadrics, superquadrics, metaballs and other implicit models, generative models, smart points, and volume elements. Some of the more interesting candidates can be found in [Lane80][Max81][Blinn82][Kajiya83].

Ray tracing renders by shooting rays into the scene and rendering the closest intersection with the ray. New primitives in a ray tracer only have to support a ray-object intersection test, so procedural primitives are easily supported. Examples may be found in [Rubin80][Hall83][Wyvill85][Kuchkuda88][Kolb92]. Examples are harder to find for non-ray-traced renderers. A handful of testbed systems have provided some degree of primitive programmability: [Whitted82][Crow82][Hall83][Fleischer87][Nadas87]. These testbeds require the new primitive to be created in the language of the testbed, using some general internal interface.

Chapter 5 covers our procedural primitive support. Like the prior systems, our primary interface for creating new primitives is a testbed-style one. These primitives can operate by selecting pixels within the primitive to render or by subdividing into simpler primitives. In Chapter 5, we also define language extensions to allow new primitives to be written in a high-level, special-purpose language.

We have listed some of the prior testbed-style interfaces for creating procedural primitives. These primitives determine the visible pixels and their shading parameters or color. More detail on one of these prior systems, Pixel-Planes 5, will be presented in Chapter 5, along with details of the PixelFlow procedural primitive implementation.

2.4.4. Interpolate

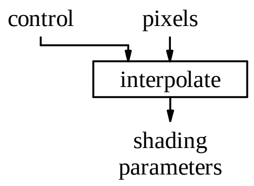

Figure 2.7. Interpolation stage.

Interpolation is the computation of shading parameter values across each primitive, such as surface normals, colors, or texture coordinates. The prior work in procedural shading (discussed in the next section) demonstrates that this interpolation is independent of the shading procedure or its parameters. Since all that is important to the surface shader is the resulting parameter values at each pixel, any method may be used to compute the values. Thus, a procedural interpolator must produce shading parameter values for the set of pixels determined by the primitive.

The OpenGL API allows the texture coordinates to be interpolated using *texture coordinate generators* [Neider93]. These texture coordinate generators include planar projection, spherical projection, or cylindrical projection. Any of these methods can be used to project the texture coordinates onto any primitive. Ebert allows arbitrary procedural interpolation with his *solid spaces* [Ebert94]. A solid space defines values throughout a 3D volume. Ebert uses solid space to construct procedural models of smoke and fog.

The texture coordinate generators found in OpenGL are not an example of procedural interpolation, but they do demonstrate that more flexible interpolation techniques have been found useful enough to appear in interactive graphics hardware systems. It is easy to see how the fixed set of texture coordinate generators can be extended to other shading parameters or made as flexible as Ebert's solid spaces.

2.4.5. Shading

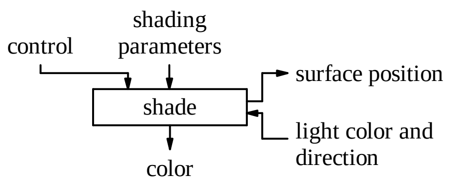

Figure 2.8. Shading stage.

Shading is the process of computing a color for each point on a surface. This color depends on the lighting, the surface properties, and the raw surface color. Unlike some of the other procedure types, surface shading appears in essentially the same form in any kind of rendering, whether z-buffer, ray tracing, or volume rendering. After modeling (at least at the level provided by a graphics API), surface shading is the facet of the rendering process that is most often attacked by procedural methods.

Procedural shading describes the shading of a surface through a simple function to turn the surface attributes and shading parameters into a color. Over the past several years there has been a trend among high-quality rendering programs to include shading languages and procedural shading capabilities.

In early systems, programmability was supported by rewriting the shading code for the renderer ([Max81], for example). Whitted and Weimer explicitly allowed this in their test-bed system [Whitted81][Whitted82]. Their *span buffers* are an implementation of a technique now called *deferred shading*. In this technique, the parameters for shading are scan converted producing an image-buffer full of shading parameters. Shading is done after all of the primitives have been scan converted. For their software system, this allowed them to run multiple shaders on the same scene without having to spend the time to rerender.

More recently, easier access to procedural shading capability has been provided to the graphics programmer. Cook's *shade trees* [Cook84] are the base of most later shading works. He turned simple expressions describing the shading at a point on the surface into a parse tree form, which was interpreted. A powerful set of precompiled library functions helped produce a reasonable rendering speed. He introduced the name *appearance parameters* for the parameters that affect the shading calculations. He also proposed an orthogonal subdivision of types of programmable functions into surface shade trees, light trees, and atmosphere trees. This corresponds exactly to the shading, lighting, and fog stages of Figure 2.1. Abram and Whitted created an implementation of Cook's shade trees called *building block shaders* [Abram90]. They had a graphical interface to construct the trees by plugging blocks together instead of using an expression parser.

Perlin’s image synthesizer extends the simple expressions in Cook’s shade trees to a full language with control structures [Perlin85]. He also introduced the powerful Perlin noise function, which produces random numbers with a band-limited frequency spectrum. This style of noise plays a major role in many procedural textures.

The RenderMan shading language [Hanrahan90], [Upstill90] extends the work of Cook and Perlin even further. They extend the procedure types introduced by Cook to also include displacement mapping, transformations, image operations, and volume effects. The RenderMan shading language is presented as a standard so shaders can be portable to any conforming implementation.

Even on existing interactive systems, the shading process involves turning shading parameters into color. Interactive systems primarily use a simple Phong shading model with Gouraud shading, where a fixed lighting model is applied at each vertex and the resulting color is interpolated across each triangle. In contrast, the history of shading, outlined above, shows the progression to include more and more flexibility. The off-line shading systems also have become increasingly well-defined and easier to use, with a corresponding increase in actual use.

Shading capabilities

Since surface shading is the most explored procedural method, we have a good understanding of the capabilities provided by procedural shading that are not possible with image texturing or other simple models previously used in interactive graphics systems. These include animated shaders, volume shaders, shaders with great computed detail, and shaders that do automatic antialiasing.

Procedural shading allows animated shaders, where the appearance of the surface changes with time. In animated shaders, time is just another control parameter for the shader. The animated ripple, shown in Figure 1.4a and b is an example of this. It is possible to get similar effects by flipping through a set of image textures, but the animated procedural texture can react to changes on the fly. The animated ripple texture creates ripples under interactive user control.

Procedural shading also allows volume shaders, where the surface is appearance is derived from the position of the surface. This can be contrasted to typical image texturing techniques that attempt to map a 2D image onto a surface. Sometimes this mapping can be quite difficult. The wood texture, show in Figure 1.4d and e is an example of a procedural volume texture. Image volume textures, as a stack of 2D image textures, have appeared in interactive graphics hardware. For good results, they require stacking a large number of 2D textures.

Since procedural shaders are only computed for a limited number of samples, they can contain much greater detail than image textures. As we zoom closer to the surface, we can see more of the detail, but the additional detail computation only has to be done for a limited portion of the surface. For example, the Mandelbrot shader shown in Figure 4.7 is based on a mathematical function that has infinite detail. The shader cannot have truly infinite detail, but it does have visible detail to the precision limits of the computations involved. It would be prohibitively expensive to compute and store an image texture at the resolution limit (consisting of about $1.8*10^{19}$ pixels); a procedural shader only computes the subset of those pixels that appear in each frame.

Procedural shaders also have the ability to do automatic antialiasing, as is seen in Figure 1.4c. Some procedural shaders do no antialiasing, and as a result are only useful at a limited range of distances. Others do much more sophisticated antialiasing. Both procedural shaders and image textures have a tradeoff between cost and antialiasing quality. The point along this spectrum typically chosen for interactive graphics systems is antialiasing using the MIP-mapping technique [Williams83]. MIP-maps contain a pyramid of filtered texture images, each level a quarter of the size of the one above it. This takes just 25% more space than the original full-size texture and provides good antialiasing at any distance as long as the textured surface is directly faces the viewer, with some degree of over-blurring if the texture is tilted away from the viewer.

Antialiasing

Several techniques have been used to antialias surface shaders. Since the surface shader can be an arbitrarily complex function, correct antialiasing has the potential to be much more difficult than for image texturing. The major techniques are analytical filtering, attenuation of high-frequency elements, and super-sampling. Much of the work in this area is covered in [Ebert94].

Analytical filtering attempts to convolve a simple shader with a filter kernel (for discussion of antialiasing as filtering, see [Glassner95]). If this can be done exactly, the result is as good an antialiased shader as possible. The basic approach is presented by Peachy [Ebert94]. First `if`'s and other conditional expressions are replaced by `step` functions. (`step(t)` is 0 for `t<0` and 1 for `t>0`). Then the step functions are replaced by a filtered form, RenderMan provides `boxstep`, `smoothstep`, and `filterstep`; versions of `step` convolved with different filter kernels.

Perlin discusses replacing conditionals with filtered step functions automatically [Ebert94]. Heidrich, Slusallek and Seidel extend this idea by also using affine arithmetic to keep track of the possible range values for all of the expressions in the shader [Heidrich96]. This provides information on the possible error in the resulting filtered shader.

The second technique for antialiasing shaders is to selectively attenuate high-frequency elements in the shader, as discussed by Peachy in [Ebert94]. This method is assisted by the use of a *band-limited noise* function introduced by Perlin [Perlin85]. Band-limited noise is a common building block for shaders. It can be thought of as white noise that has been put through a band-pass filter to limit the frequencies to the octave between a base frequency and half the base frequency. As the base frequency approaches the pixel size, the shader fades the noise function to zero. Other features can also be smoothly removed. For example, the mortar lines in a brick wall might fade to match the color of the brick as the mortar thickness approaches the width of a pixel.

The final technique is shader super-sampling. Super-sampling is a common technique for antialiasing geometry, both in interactive and off-line rendering. With super-sampling, several samples are rendered for each pixel, then combined to produce the final pixel color. Shader super-sampling is discussed by both Peachy and Worley in [Ebert94]. In this method, the shading procedure itself takes several samples at slightly perturbed points, which it blends to produce the color for a single image sample or pixel.

2.4.6. Lighting

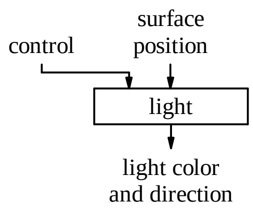

Figure 2.9. Light stage.

Procedural lighting functions determine the intensity and color of light that hits a surface point from a light source. They can be used for a variety of shadow and slide projector techniques.

RenderMan [Hanrahan90] and Cook’s shade trees [Cook84] make an important conceptual distinction between lighting and shading. The same light procedure may be used by all of the shading procedures. A prime example of the power of lighting procedures is the window pane light in Pixar’s Tin Toy [Upstill90]. A version of this light, rendered on PixelFlow, is shown in Figure 1.4

Recent advances in procedural lighting have been made by Slusallek, et. al. at the University of Erlangen [Slusallek98]. They create a graphical tree of *LightOps*, similar to the building-block shader representation [Abram90]. Each LightOp represents a light or surface that reflects light in the scene. The connections between the LightOps represent the light that bounces from surface to surface. The resulting lighting network, which may include cycles, describes all of the light interactions as light bounces from object to object in a global illumination simulation.

In contrast, interactive systems typically support only a small fixed set of lights. With directional lights, all of the light comes from the same direction. Point lights emanate from a single point. Some interactive systems also include spot lights that shine in a directed cone of light. Some also allow the light to be masked or colored by an image texture. Procedural lighting extends these to include arbitrary masking, shadowing, or even changes in the direction the light shines from point to point.

2.4.7. Volume and Atmospheric Effects

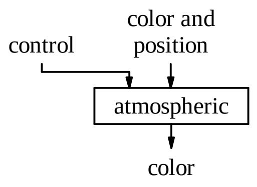

Figure 2.10. Atmospheric stage.

Procedural volume and atmospheric shaders handle the behavior of light as it passes through a medium, normally the space between a visible surface and the viewer. A few examples are fog, haze, atmospheric color shift, and density thresholding.

Cook [Cook84] defines atmospheric shade trees to handle fog and similar color effects between the surfaces and the viewer. RenderMan [Hanrahan90] extends this to two types of shaders, volume and atmospheric, to handle the passage of light through and outside of objects in the scene.

Interactive systems generally support only a simple model for fog, a linear or exponential falloff with distance. Near surfaces are unobscured, while surfaces farther from the viewer fade to some single fog color. With the addition of procedural effects, a more complex array of effects is possible.

2.4.8. Image Warping and Filtering

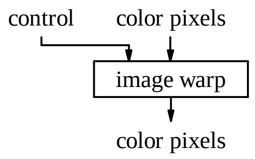

Figure 2.11. Warping stage.

Procedural warping can be used to support a host of video-warping special effects; water droplets on a camera lens, for example. Procedural warping can also compensate for the barrel distortion caused by the lenses used in some head mounted displays. Image warping effects require grabbing the final pixel color for one place in the image from another place in the image.

For image filtering the image pixels do not move, but are combined to achieve effects like blurring, sharpening, or brightening. Warping and filtering are similar because both use the values of one or more pixels in the input image to compute the value of each pixel in the output image.

Simple image modifications can be done with the RenderMan *image shaders* [Upstill90]. As with all RenderMan procedures, image shaders are written from the point of view of a single pixel. Since they do not allow access to the color results of other pixels, image shaders do things like change the image brightness, but cannot do either filtering or warping effects.

Better procedural warping and filtering capabilities can be found in image manipulation programs. Adobe Photoshop has a plug-in interface, which allows new filtering and warping functions to be written in C [Knoll90a]. The GIMP (Gnu Image Manipulation Program) has a similar C plug-in interface, and also has a Scheme scripting interface [Kylander97].

The main filtering present in interactive graphics systems is anti-aliasing. The results of several image samples are combined to create the result color for a single pixel. By incorporating the procedural methods present in image manipulation software, we can provide a rich set of post-processing alternatives for 3D images.

2.5. Maps

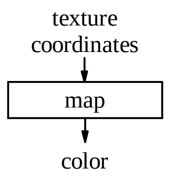

Figure 2.12. Typical map.

Maps are not a stage, per se, but a type of procedure that may be used by any of the stages. Image maps are common in graphics. For example, a texture map is a 2D image. The texture map is pasted onto a surface like a decal. The mapping is accomplished through a pair of texture coordinates that parameterize the surface. The same 2D texture coordinates can be used to map a bump map onto the surface. Instead of color, the bump map has information that is used to perturb the surface normal to change the surface shading [Blinn78].

Other mapping techniques do not rely on texture coordinates. Reflection mapping also determines a color to apply to the surface. However, with reflection mapping, the location in the 2D image is determined by the direction of a reflection off of the surface [Williams83]. Shadow maps are projected onto the surface from a light in the scene, and determine whether each point of the surface is lit or in shadow [Williams78].

All of these techniques have a common theme. A map exists, generally in the form of a 2D image. One of a variety of techniques is used to decide where to look in the 2D image. The resulting value is used as a parameter to the shading model. Procedural maps are just the programmable version of this idea. A procedural map is a function instead of a 2D image. Instead of looking up texture results in an image map, any stage can call a map function to obtain the parameter value.

With image texturing, the same shader can be used with many different image textures to get many different effects. Similarly, many different procedural maps can be used with the same procedural stage. Procedural maps can be used wherever image maps would be. Procedural maps can also be generalized beyond simple replacements for 2D image maps. The procedural maps can be functions of a different size (e.g. 1D or 3D instead of 2D), though only procedural maps of the same input and output dimensions can be substituted for each other. In other words, a procedural map is just a function that can be replaced under application control.

The procedural shading capability described in [Rhoades92] is really just an instance of procedural maps in a fixed Phong shader. The map function complexity was limited by the pixel memory on Pixel-Planes 5 (208 bits). The shading language was also not very high level. It included simple operations (add, multiply, copy) and some more complicated built in functions (wood, Julia set, Gardner textures, Perlin noise).

So, procedural maps are essentially just procedural replacements for the more common image texture maps. Other than the Pixel-Planes 5 system, procedural maps have not been well explored. In many cases, the procedures can be run initially to create 2D images from the maps, which can be used by current hardware to simulate the same results.

2.6. Using the pipeline

In this chapter, we have presented a summary of our abstract pipeline. This pipeline is just one possible decomposition of the work required for 3D graphics. Our particular decomposition was driven by the following criteria: each stage is an independent procedure; the pipeline can be easily mapped to a range of hardware systems; the pipeline presents a simple model of the system operation; and even if procedural support is only available at a subset of the stages, the abstract pipeline can still be used as a model of the system.

We also used the abstract pipeline as a taxonomy to classify the previous work in procedural approaches to graphics. In the remaining chapters, we will demonstrate the use of the abstract pipeline by the mapping it onto one interactive graphics hardware system, PixelFlow.