# 4 SURFACE SHADING

Surface shading is the most heavily explored procedural graphics technique. There are several reasons for this.

- Procedural shading makes it easy to add the noise and random variability to a surface that make it look more realistic.
- It can be easier to create a procedural shader for a complicated surface than to try to eliminate the distortions caused by wrapping a flat scanned texture over the surface.
- If a procedural shader does not produce quite the right effect, it is easier to tweak it than to rescan or repaint an image texture.
- It is often easier to create detail on an object using a procedural shader instead of trying to modify the object geometry.
- A procedural surface shader can change with time, distance, or viewing angle.

As was explained in Chapter 2, the job of a surface shader is to produce a color for each point on a surface, taking into account the color variations of the surface itself and the lighting effects. We have created a language named *pfman* for writing these surface shaders (*pf* for PixelFlow, *man* for its similarity to Pixar's RenderMan shading language). Our language is close, in syntax and purpose, to the RenderMan shading language [Hanrahan90][Upstill90], and consequently to the C programming language. As in the RenderMan shading language, we include several constructs (not found in C) intended to make shading easier. Pfman is described in Section 4.1.

Section 4.1.5 covers the interface used by the graphics application to control the procedural shaders. The remaining sections of this chapter describe the optimizations that make pfman shaders better able to run on the graphics hardware (or in some cases that enable the shaders to run at all). The optimizations for PixelFlow cover one of the following key areas: memory usage (Section 4.3), communication bandwidth (Section 4.4), or execution speed (Section 4.5).

## 4.1. Pfman language

The PixelFlow shading language is a special purpose C-like language for describing the shading of surfaces on the PixelFlow graphics system. On PixelFlow, a shading function is associated with every primitive. The shading function is executed for each visible pixel (or each sample for antialiasing) to determine its color. This section describes the pfman language and points out where it differs from the RenderMan shading language

### 4.1.1. Data

Variable declarations in pfman follow this basic form:

```
type_specification[array_dimensions] identifier[array_dimensions]
```

Where `type_specification` is

```
[type_modifier|basic_type] type_specification
```

Only one `basic_type` can appear in any `type_specification`.

#### Basic types

Only a few simple data types are supported: `void`, `float`, and `fixed`. The simplest type is `void`. It is only used as a return type for functions that have no return value. Where RenderMan has a single floating point type used for all scalar values, we have two types, `float` and `fixed`. The floating point type is easier to use, but fixed point is more efficient. Unlike RenderMan, a string type is not used as an identifier for texture maps; instead a scalar ID is used.

The `fixed` type has two parameters: the size in bits and an exponent. So it is really a class of types, given as `fixed<size,exponent>`. For exponents between zero and the bit size, the exponent can also be thought of as the number of fractional binary digits. A two byte integer would be `fixed<16,0>`, while a two byte pure fraction would be `fixed<16,16>`. An exponent larger than the size of the fixed point number or less than zero is also perfectly legal. Conversion between the real value and stored value uses these equations:

```
represented_value = stored_value * 2^-exponent
stored_value = represented_value * 2^exponent
```

It is much less confusing to always work with the real value. For example, a `fixed<8,8>`, representing 0.5 is not “128”, any more than a floating point number representing 0.5 is “2,113,929,216”.

Variables of the `fixed` type may be declared `signed` or `unsigned`. The size of a fixed point type does not include the extra sign bit added by `signed`. So a `signed fixed<15,0>` takes 16 bits. If not specified, all fixed point variables default to `signed`.

**Arrays**

Pfman supports arrays of the basic types, declared in a C-like syntax. For example, the declaration `float color[3]` declares color to be a 1D array of three floats, `color[0]`, `color[1]`, and `color[2]`.

Arrays can be multi-dimensional, for example `float color_list[2][3]`. As with C, it is not necessary to give all of the indices for an array at once. While `color_list[1][1]` is a `float`, `color_list[0]` and `color_list[1]` are each `float[3]` 1D arrays. In this case, `color_list` in interpreted as two three-element vectors. Pfman allows the array sizes to be specified with the type or with the variable, so `color_list` could also have been declared as `float[2][3] color_list` or `float[3] color_list[2]`. The last version makes explicit the idea that `color_list` is two three-element vectors.

RenderMan uses separate types for points, vectors, normals, and colors. Pfman uses arrays. This allows us to declare any of these quantities to be either floating point or fixed point as appropriate. Arrays also make matrices and lists of points easy to represent. Recent versions of the RenderMan shading language have added a new matrix type and arrays of the basic types (float, point, vector, normal, color) for just this purpose.

**Uniform and varying**

As with RenderMan, pfman includes the `uniform` or `varying` type modifiers. A `varying` variable is one that might vary from pixel to pixel, similar to `plural` in MasPar's MPL [MasPar90] or `poly` in Thinking Machines' C* [ThinkingMachines89]. For example, the texture coordinates across a surface would be `varying`.

A `uniform` variable is one that will not vary from pixel to pixel, similar to `singular` in MPL or `mono` in C*. For the brick shader presented in Figure 1.3, the uniform parameters are the width, height and color of the bricks and the thickness and color of the mortar. These control the appearance of the brick, and allow us to use the same brick shader for a variety of different brick styles.

Declaring a variable to be `varying` does not imply that it will vary, only that it might. If not specified, shader parameters default to `uniform` and local variables default to `varying`.

**Other type modifiers**

There are a number of additional modifiers for pfman shader parameters. These affect what happens to the parameter before it is passed to the shader. For example, the `unit` modifier indicates that the parameter should be normalized before it is passed to the shading function. This does not imply that a `unit` parameter will remain unit length if it is changed within the shader.

The remaining modifiers each declare a type of transformation to be applied to a parameter before it is passed to the shading function. The first four of these transformation types are discussed in more detail on page 2.4.2. The `transform_as_point` modifier applies the current geometric transformation to a point in homogeneous coordinates. The inverse of this is given by `transform_as_plane` (so named because, for the perspective transformations prevalent in graphics, plane coefficients transform by the inverse of the homogeneous point transform). The `transform_as_vector` and `transform_as_normal` modifiers apply the correct transformations for vectors or surface normals [ONeill66][Koenderink90]. For example, you might declare a parameter

```
unit transform_as_vector float v[3]
```

The final transformation modifier is `transform_as_texture`. This transformation allows translation, scaling, or various other effects between the original texture coordinates and the coordinates used by the shader [Segal92].

The RenderMan shading language does not include any similar form of type modification. The transformations are implicit in the basic RenderMan types, and RenderMan has no equivalent of `unit` or `transform_as_texture`.

#### User-defined types

Pfman also supports aliases for types with a C-like `typedef` statement. `typedef` is only legal outside function definitions, and no distinction is made between equivalent types with different names. The statement

```
    typedef float Point[3], Normal[3];
```

declares `Point` and `Normal` to both be types that can be used completely interchangeably with `float[3]`. A number of such type definitions can be found in the pfman include file, `pftypes.h`. Pfman does not allow any function, parameter, or variable to have the same name as a user-defined type.

### 4.1.2. Functions

#### Overloading

Function overloading, similar to that in C++, is supported by both pfman and RenderMan. Functions of the same name that can be distinguished by their input parameters are considered distinct. This provides the ability to have separate versions of functions for `uniform` and `varying` parameters, `float` and `fixed`, or different fixed point types. Note that functions cannot be overloaded based on their return types and operator overloading is not supported.

#### Definition

A function definition specifies the return type, name, parameters, and body that define the function. These function definitions cannot be nested. A simple function definition is

```
float factorial(float n) {
   if (n > 1)
      return n * factorial(n);
   else
      return 1;
}
```

**Shading functions**

There are several special pseudo-return-types that indicate that a function is actually a procedure for some procedural stage (Section 2.3). For the procedure types discussed in this chapter, these are `surface` and `light`. Chapter 5 covers `primitive` and `interpolator` functions. Instead of returning the result of the procedural stage, these functions place their results in special `output` parameters. For `surface` procedures, the output color is declared

```
output varying Color px_rc_co[3]
```

Where `Color` is a `typedef` alias for `unsigned fixed<8,8>`. We restrict varying surface shader parameters to only use fixed point types. This is an implementation restriction that may be relaxed in the future.

Light procedures produce both a color and direction, which are declared

```
output varying Short px_rc_l[3]
output varying Color px_rc_cl[3],
```

`Short` is a `typedef` alias for `signed fixed<11,8>`. These type names are defined for convenience in the `pfman_types.h` header file, included by most pfman procedures. The parameter names are defined by other portions of the PixelFlow system software. One of the biggest complaints of our users is that the names of these parameters do not match those of RenderMan (`Ci`, `L`, and `Cl`). In future efforts, even if we kept the pfman language, we would choose to convert these names within the pfman compiler to match the user's expectations.

The procedural stage functions are not called explicitly. Their parameters are set by name from a global state kept by the graphics library (discussed in more detail in Section 4.1.5). It is possible that an application program will never set the value for some of the parameters of a shader. As is done in RenderMan, we allow default values for the parameters of these functions. These are given in the parameter list as `parameter = value` (just like variable initialization). e.g.

```
float brick_width = 0.05
```

These default values must be compile-time constants. As with RenderMan, if no default value is given, it is assumed to be zero.

These ideas are demonstrated in the code for a simple light that uses the same color and light direction for all pixels (Figure 4.1).

```
light simple_light(
   uniform float px_light_diffuse[3] = {0.99, 0.99, 0.99},
   uniform float px_light_position[3] = {.408, -.408, .816},
   output varying Color px_rc_cl[3],
   output varying Short px_rc_l[3])
{
   px_rc_cl = px_light_diffuse;
   px_rc_l = px_light_position;
}
```

**Figure 4.1. Code for a simple light.**

### Prototypes

Any function that is to be used before it is defined, or that is defined in a different source file, must have a prototype, for example

```
float factorial(float n);
```

Prototypes for the common math and shading functions (as defined for RenderMan in [Upstill90]) are defined in the standard pfman include file `pfman.h`.

### External linkage

As mentioned in Section 3.3.1, The pfman shading language compiler turns shading language source code into C++ source code that must be further compiled with a C++ compiler. The function definitions and function calls created by the compiler correspond directly to C++ function definitions and function calls. It is possible (and supported) to call C++ functions from shading language functions and to call shading language functions from C++. This facility is limited to functions using pfman's types. For example, the ability to call C++ functions from shading code is used to allow access to the standard math functions for `uniform float` variables. The ability to call shading language code from

C++ allows shaders written with the testbed interface to call functions that have been created in pfman.

The pfman compiler adds some additional arguments to most functions when creating the corresponding C++ function. To create and call ordinary C++ functions, we use `extern` directives similar to the C++ `extern` directive:

```
    extern “C++” uniform float factorial(float x);
```

These directives appear immediately before a function definition or declaration, and modify the C++ code generated for that function. Legal strings for the `extern` are `“C”`, `“C++”`, `“inline”` or `“static”`. The `extern “C++”` and `extern “C”` directives indicate that the function should be compiled as a regular C++ or C function without the extra arguments normally added by pfman. Functions of either of these types cannot include any varying parameters or variables. The `extern “inline”` directive indicates that the function should be compiled as a C++ inline function. The `extern “static”` directive indicates that the function should be visible only in the file where it is defined. Combinations of these are also acceptable (e.g. `extern “static C++”`), as long as C and C++ do not both appear. Thus the `extern` specification is

```
    extern “[inline] [static] [C|C++]”
```

### 4.1.3. Expressions

#### Operators

The set of operators and operator precedence is similar to that of C (it was based on a grammar for ANSI C). The full list of operators and their precedence is given in Figure 4.2.

#### Operations on arrays

Operations on arrays are defined as the corresponding vector, matrix, or tensor operation. The unary operations act on all elements of the array. Addition, subtraction, and assignment require arrays of equal dimension and perform the operation between corresponding elements (i.e. `a + b` gives the standard matrix addition of `a` and `b`). The comparison operations also require arrays of equal dimension, though only `==` and `!=` are permitted.

| Operation | Associativity | Purpose |
|---|---|---|
| `( )` | — | expression grouping |
| `++ -- []` | — | postfix increment and decrement, array index |
| `++ -- - !` | — | prefix increment and decrement, arithmetic and logical negation |
| `( )` | — | type cast |
| `^` | left | xor / cross product / wedge product |
| `* / %` | left | multiplication, division, mod |
| `+ -` | left | addition, subtraction |
| `&` | left | bitwise and |
| `\|` | left | bitwise or |
| `<< >>` | left | shift |
| `< <= >= >` | left | comparison |
| `== !=` | left | comparison |
| `&&` | left | logical and |
| `\|\|` | left | logical or |
| `?:` | right | conditional expression |
| `= += -= *= /= ^=` | right | assignment |
| `,` | — | expression list |

**Figure 4.2. Pfman operator precedence**

Multiplication between vectors gives a dot product, between vector and matrix, matrix and vector, or matrix and matrix gives the appropriate matrix multiplication. More generally, multiplication between any two arrays gives the tensor contraction of the last index of the first array against the first index of the second array (e.g. a generalized inner product). In other words, for `float a[3][3][3]`, `float b[3][3][3]` and `float c[3][3][3][3]`,

```
    c = a * b;
```

computes

$$\texttt{c[i][j][k][l]} = \sum_{\texttt{m=0}}^{3} \texttt{a[i][j][m]*b[m][k][l]}$$

The `/` and `^` operators have special meaning for certain array types. `1/a` is the inverse of a square matrix `a`, and `b/a` multiplies `b` by the inverse of square matrix `a`. The `^` operator gives the cross product between two 3D vectors.

**Inline arrays**

C-style array initializers are allowed in any expression as an anonymous array. A 3x3 identity matrix might be coded as `{{1,0,0},{0,1,0},{0,0,1}}`, while the computed elements of a point on a paraboloid might be filled in with `{x, y, x*x+y*y}`. As a variable initializer, this would be

```
float v[3] = {0,0,0};
```

In an expression, it would be

```
v = p - {1,1,1};
```

### 4.1.4. Statements

As in C, anywhere a statement is legal, a compound statement is legal as well. A compound statement is just a list of statements delimited by `{` and `}`. Any expression followed by a `;` is also a legal statement. The remaining types of statements closely mimic C or the RenderMan shading language.

**Standard control statements**

Most of the control statements are borrowed directly from C.

```
if (condition_expression) statement_for_true
if (condition_expression) statement_for_true
      else statement_for_false
while (condition_expression) loop_statement
do loop_statement until (condition_expression);
for (initial_expression; condition_expr; increment_expression)
   loop_statement
break;
continue;
return;
return return_value_expression;
```

In addition, the `illuminance` statement is taken from RenderMan, to aid in shading. This statement,

```
illuminance () statement
illuminance (position_expression) statement
illuminance (position_expression, axis_expression,
      angle_expression) statement
```

can be thought of as an integral over the incoming light. For our implementation (as well as Pixar's RenderMan implementation), `illuminance` acts as a loop over the available light sources. Since the lights are also procedural, this means the function for each light source is run, producing a light color and intensity. A group at the University of Erlangen has produced a RenderMan implementation that computes a *global illumination* rendering, including all of the inter-reflections between different surfaces [Slusallek94]. In their implementation, the `illuminance` statement really does numerically compute the integral over all incoming light.

Within the body of the `illuminance` statement, the light direction can be accessed with the shading parameter, `px_rc_l`, and the light color can be accessed as `px_rc_cl`. An example of the `illuminance` construct implementing an approximation to Phong shading from [Lyon93] is given in Figure 4.3.

```
illuminance() {
   float L[3] = normalize(px_rc_l);
   float n_dot_l = Nf * L; // Nf = unit surface normal
   float v_dot_l = V * L;  // V = unit "view" vector from surface to eye

   // specular contribution
   varying float spec = 1 + v_dot_l - two_n_dot_v * n_dot_l;// D.D / 2
   spec = 1 - spec * px_material_shininess / 4;
   if (spec < 0) spec = 0;
   spec *= spec;
   spec *= spec;

   if (n_dot_l < 0)
      n_dot_l = spec = 0;

   // add in diffuse and specular contributions with appropriate colors
   diffuse += px_rc_cl * n_dot_l;
   specular += px_rc_cl * spec;
}
```

**Figure 4.3. Use of the illuminance construct.**

**Declaration statements**

As in C++, variable declarations can occur anywhere a statement can. For example,

```
float a[3], b=2*x, c;
```

declares `a` as an uninitialized 1D `float` array with 3 elements, `b` as a `float` with an initial value twice the value of variable `x` at the declaration time, and `c` as an uninitialized `float`. Each compound statement block, enclosed by { and }, defines a new scope. Variables can be redefined within a compound statement without conflicting with function or variable names in other scopes.

### 4.1.5. Antialiasing

Pfman has minimal support for shader antialiasing similar what is available in the RenderMan shading language. None of the automatic antialiasing techniques discussed in 2.4.5 are used. Filtered `step` functions are available to analytically antialias a shader. Band-limited noise functions are also available as a creative tool for writing shaders, and these functions can be faded by the user-written pfman code as their base frequency approaches the pixel size. To use these tools, it is necessary to know the size of pixel being shaded. The pixel size is available in the varying input parameter, `px_shader_f_sqr`, equivalent to the RenderMan variable `area`.

## 4.2. Application interface

The RenderMan standard [Upstill90] defines not only the shading language, but also a graphics application program interface (API). This is a library of graphics functions that the graphics application can call to describe the scene to the renderer. We elected to base our API on OpenGL [Neider93] instead of RenderMan. OpenGL is a popular API for interactive graphics applications, supported on a number of graphics hardware platforms. It provides about the same capabilities as the RenderMan API with a similar collection of functions, but with more focus on interactive graphics. By using OpenGL as our base we can also easily port applications written for other hardware.

We extended OpenGL to support procedural shading. We required that these procedural shading extensions have no impact on applications that do not use procedural shading. We also endeavored to make them fit into the framework and philosophy of OpenGL. The design of these extensions was a group effort. The primary contributors were Jon Leech, Lee Westover, Anselmo Lastra, Roman Kuchkuda, Paul Layne, Rich Holloway, and the author. These PixelFlow extensions to OpenGL are described in much more detail in [Leech98].

Following the OpenGL standard, all of our extensions have the suffix `EXT`. We will follow that convention here to help make it clear what functions are already part of OpenGL and which we added. OpenGL functions also usually include additional suffix letters to indicate the operand types (`f`, `i`, `s`, etc.). For brevity, we will generally omit these in the text, though we will use them in the code examples.

### 4.2.1. Shading parameters

Applications that do not employ procedural shading use a default *OpenGL shader*. This built-in procedural shader supports the standard OpenGL shading model. Parameters to the OpenGL shading model are set using the `glMaterial` call or one of a handful of other parameter-specific calls (`glColor`, `glNormal`, and `glTexCoord`). The OpenGL shading model uses a number of different color parameters (`GL_AMBIENT_-COLOR`, `GL_DIFFUSE_COLOR`, `GL_SPECULAR_COLOR` or `GL_EMMISIVE_COLOR`). `glColor` can be assigned to set any one of these or the combination of ambient and diffuse. The other colors can still be set by `glMaterial`. Figure 4.4 shows some OpenGL code for one vertex using these calls. A triangle includes three similar vertices.

```
glNormal3f(1.0, 0.0, 0.0);
glMaterialfv(GL_EMISSIVE_COLOR, white); // white is a float array
glVertex3f(0.0, 0.0, 1.0);
```

**Figure 4.4. Typical OpenGL code for a vertex.**

We use these same functions for other shaders. To handle arbitrary shading parameters, we assign each parameter a *parameter ID*, which is used to identify it in the `glMaterial` call. The application can find a parameter ID using the `glMaterialParameterNameEXT` function. The `glNormal`, and `glTexCoord` functions are equivalent to using `glMaterial` with specific parameters of the OpenGL shader (`px_material_-normal` and `px_material_texcoord`). For example,

```
glNormalfv(normal);
glTexCoordfv(texcoord);
```

is equivalent to

```
      glMaterialfv(glMaterialParameterNameEXT(
                        "px_material_normal"), normal);
      glMaterialfv(glMaterialParameterNameEXT(
                        "px_material_texcoord"), texcoord);
```

The various color parameters are equivalent to parameters named `px_material_-ambient`, `px_material_diffuse`, `px_material_specular` and `px_mate-rial_emissive`. By using parameters with these same names, user-written shaders can make use of the values that were set in the application using `glColor`, `glNormal`, and `glTexCoord`.

### 4.2.2. Shader instances

The RenderMan API allows some parameter values to be fixed when a shader function is chosen. Our equivalent is to allow certain *bound* parameter values. A shading function along with its bound parameters together make a *shader instance* (or sometimes just *shader*) that describes a particular type of surface. Shader instances with bound parameter values allow us to define several surface types using the same shading function, for example fat red bricks and thin yellow bricks (Figure 4.5), both using the brick function of Figure 1.2. We can easily choose one kind of brick or the other within the application by referring to the right shader instance.

A shader function describes how to create a certain class of surfaces, (e.g. "bricks"). To set bound parameter values, we add a `glBoundMaterialEXT` function, equivalent to `glMaterial` for bound parameters.

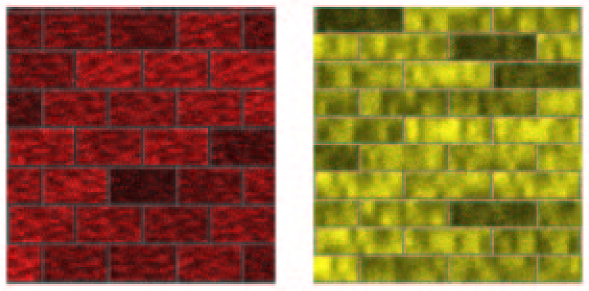

**Figure 4.5. Instances of a brick surface shader**

We load a shader function by calling the new API function `glLoadExtension-Code`. To create instances, we provide three new functions. The instance definition is contained in a `glNewShaderEXT`, `glEndShaderEXT` pair. This is similar to other OpenGL capabilities, for example display list definitions are bracketed by calls to `glNew-List` and `glEndList`. `glNewShaderEXT` takes the shading function to use and returns a *shader ID* that can be used to identify the instance later. Between the `glNew-ShaderEXT` and `glEndShaderEXT` we allow calls to `glShaderParameterBin-dingEXT`. `glShaderParameterBindingEXT` takes a parameter ID and one of `GL_MATERIAL_EXT` or `GL_BOUND_MATERIAL_EXT`. This indicates that the parameter should be set by calls to `glMaterial` or `glBoundMaterialEXT` respectively. Figure 4.6 shows the code to create a shader instance.

```
// load the shader function
GLenum phong = glLoadExtensionCodeEXT(GL_SHADER_FUNCTION_EXT, "phong");

// create a new instance called red_phong
GLenum red_phong = glNewShaderEXT(phong);
   glShaderParameterBindingEXT(
        glGetMaterialParameterNameEXT("px_material_normal"),
        GL_MATERIAL_EXT);
    glShaderParameterBindingEXT(
        GL_DIFFUSE,
        GL_BOUND_MATERIAL_EXT);
glEndShaderEXT();

// set the bound value for the diffuse color parameter
float red[3] = {.4,0,0,1};
glBoundMaterialfvEXT(red_phong, GL_DIFFUSE, red);
```

**Figure 4.6. Application code to create a shader instance.**

To choose a shader instance, we use the `glShaderEXT` call. This function takes a shader ID returned by `glNewShaderEXT.` Primitives drawn after the `glShaderEXT` call will use that shader instance.

### 4.2.3. Lights

OpenGL normally supports up to eight lights, `GL_LIGHT0` through `GL_LIGHT7`. These lights are turned on and off through calls to `glEnable` and `glDisable`. Parameters for the lights are set by calls to `glLight`, which takes the light ID, the parameter, and the new value. We use all of these calls for our procedural lights. New light functions are loaded with `glLoadExtensionCodeEXT`, in the same way that new shader functions are loaded. New light IDs beyond the eight pre-loaded lights are created with `glNewLightEXT`.

Since OpenGL only supports eight lights, many applications reuse these lights within a frame. For example, all eight lights may be used in a single room of an architectural model. The positions and directions of the lights can be changed before rendering any polygons for the next room, giving the effect of more than eight lights even though only eight at a time shine on any one polygon.

PixelFlow's use of deferred shading means that we cannot easily handle light changes within a frame. Each light, along with its parameter settings, is effectively a *light instance* in the same way that a shader with its bound shader parameters is a *shader instance*. This encourages a PixelFlow application to create a large number of lights, since each change in parameters requires a new light. To handle the enabling and disabling of a large set of lights, we create *light groups*. A new light group is created by a call to `glNewLightGroup`. Lights are enabled and disabled in the light group using the normal `glEnable` and `glDisable` calls. Within the frame we use `glEnable` with a light group ID to switch the active set of lights. Light groups may be a useful shorthand on other systems, but their primary purpose is to make light changes within a frame possible on PixelFlow

### 4.3. Memory optimizations

The most limited resource for shaders on PixelFlow is the pixel memory. The texture memory size (64 megabytes) affects the size of image textures a shader can use in its computations, but does not affect the shader complexity. The microprocessor memory (128 megabytes), is designed to be sufficient to hold large geometric databases. For shading purposes it is effectively unlimited. However, the pixel memory, at only 256 bytes, is quite limited. From that 256 bytes, we further subtract the shader input parameters and an area used for communication between the light shaders and surface shaders. In this section we highlight some of the pfman features and optimizations made by the pfman compiler to make this limited amount of memory work for real shaders.

### 4.3.1. Paging

It would be possible to increase the available pixel memory by paging to texture memory, and even further by paging texture memory completely out of the PixelFlow machine. Writing, then reading, four bytes of texture memory for every pixel in a 128x64 region takes 380 μs. For a system with four shaders, a single swapping operation in each of the 40 regions of a non-antialiased NTSC video image would take over 10% of the 33 ms available for rendering a frame at 30 frames per second. For a single swapping operation in each of the 160 regions of a high-resolution image or an antialiased NTSC video image, the total jumps to over 45% of the available time. Consequently, paging could help for running arbitrary shaders at faster than off-line rendering (but slower than interactive) speeds. Since our primary focus is interactive rendering, we have not pursued any paging methods. Any shader we want to run interactively **must** fit in the 256 bytes of real pixel memory.

### 4.3.2. Uniform and varying

RenderMan divides parameters into *uniform*, and *varying* in part for the efficiency of their off-line uni-processor renderer. Uniform parameters are used to control the overall operation and appearance of the shader while varying parameters control variations across the surface. A single expression can use a mix of uniform and varying parameters. We define a *uniform expression* (or sub-expression) as one with a uniform result, and a *varying expression* as one with a varying result. As Pixar's prman renderer evaluates the shading on a surface, it computes uniform expressions only once, sharing the results with all of the surface samples. Then it loops over all of the surface samples to compute the varying expressions.

We can make use of a similar division of labor. We store all uniform variables in the microprocessor memory, so operations between them can be done once, by the microprocessor. Thus operations and storage for uniform variables are shared by all pixels. Varying computations must be done by the pixel processors since they can potentially have different values at every pixel. Consequently, these variables must exist in pixel memory. Their storage and operations are replicated across the SIMD array. This same distinction between shared (*uniform*) and SIMD array (*varying*) memory has been made by other SIMD compilers [MasPar90][ThinkingMachines89] (Section 4.1.1).

The division between uniform and varying provides some execution speed gain, since computing a single math operation on the PA-RISC processor is faster than computing the same math operation simultaneously on all of the pixel processors. However, the memory savings is the primary motivation for our interest in this division.

### 4.3.3. Fixed point

RenderMan has one representation for all numbers: floating point. Pfman also supports floating point (32-bit IEEE single precision format) because it is such a forgiving representation. This format can represent numbers as large as about $10^{38}$ or as small as $10^{-38}$, with about $10^{-7}$ relative error throughout the range. Smaller numbers can be represented but with greater relative error.

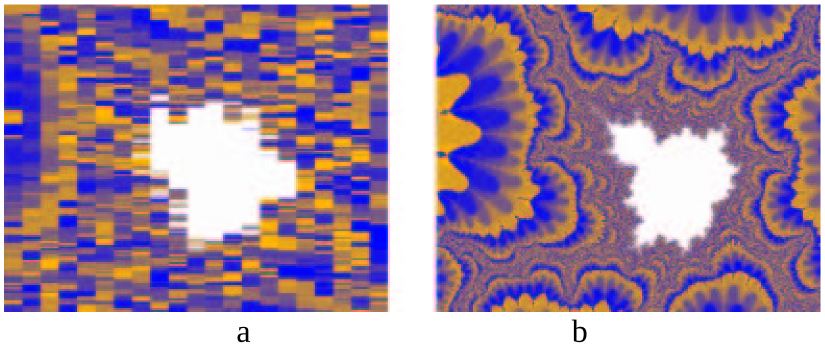

**Figure 4.7. Fixed point vs. floating point comparison.**
**a) Mandelbrot set computed using floating point. b) Mandelbrot set computed using fixed point**

For some quantities used in shading this range is overkill. For colors, an 8 to 16 bit fixed point representation is sufficient [Hill97]. But floating point takes four bytes, regardless of the necessary range. Worse, there are cases where floating point has too much range but not enough precision. For example, a Mandelbrot fractal shader has an insatiable appetite for precision, but only over the range [–2,2] (Figure 4.7). In this case, it makes much more sense to use a 32 bit fixed point format instead of a 32 bit floating point format, since the floating point format wastes one of the four bytes for an exponent that is hardly used. In general, it is easiest to prototype a shader using floating point, then make changes to fixed point as necessary for memory usage, precision, and speed (the speed advantages will be covered in more detail later in this chapter).

To help further, we specify the size of our fixed point numbers in bits. PixelFlow can only allocate and operate on multiples of single bytes. However, we can do a much better job of limiting the sizes of intermediate results in expressions with a more accurate idea of the true range of the values involved. For example, if we add two arbitrary two-byte integers, we need to allocate three bytes for the result. However, if we know the integers really only use 14 bits, the result can be at most 15 bits, which fits in two bytes instead of three.

The analysis to determine the sizes of intermediate fixed-point results happens in two passes. The first, *bottom-up* pass determines the sizes necessary to keep all available precision. It starts with the sizes it knows, variable references or the result of type casts. It combines them according to simple rules (e.g. multiplication adds the bit sizes and adds exponents). The second, *top-down* pass limits the fixed point types used for the intermediate results to only what is necessary for the assignment or type cast used for the final result of the expression.

```
fixed<10,10> x,y;  fixed<20,20> t1 = x*y;   fixed<12,12> t1 = x*y;
fixed<15,12> z,r;  fixed<24,20> t2 = t1+z;  fixed<15,12> t2 = t1+z;
r = x*y + z        fixed<15,12> r = t2;     fixed<15,12> r = t2;
```
a b c

**Figure 4.8. Example of fixed point size determination.**
**a) original code. b) after bottom-up pass. c) after top-down pass**

Figure 4.8 demonstrates the procedure used for determining fixed point types for intermediate results in a complex expression. Figure 4.8a shows the original expression and sizes. The variables `x` and `y` are signed 10-bit pure fractions, representing numbers from –1 to just under 1. Both `z` and the result, `r`, are 15 bits (3-bit integer part and 12-bit fractional part), representing numbers from -8 to just under 8. In Figure 4.8b, we have done the bottom-up pass. The result of `x*y` could be as big as 20 bits, all of which would still be fractional. The result of adding `z` to this could have 4 integer bits (e.g. adding a number just under 8 to a number just under 1, gives a result that is almost 9), and we might want to keep all 20 fraction bits from `x*y`.

Finally, Figure 4.8c demonstrates the top-down part of the algorithm. Since `r` only has three integer bits and 12 fraction, there is no point in keeping `t2` to any greater precision. Since `t2` has now been trimmed down to `fixed<15,12>`, there is no point to keeping a full 20 bits for `t1` since only 12 will be used. The result is a set of fixed point types for all parts of the expression that are as small as possible, while still conforming to the constraints set by the input and output types.

Strictly speaking, since the representation of `z` is only accurate to 12 bits, the actual number that `z` represents could be all 1's instead of all 0's (or any other choice of random bits) for the extra eight bits in Figure 4.8b. However, most users expect `1.0+0.01` to be `1.01`, not `1.1`, `1.05`, or `1.08`. If these excess bits affect the result, we set them to zero to conform to this expected behavior. The ability to ask for more precision in the result than it is possible to accurately compute is one of the pitfalls of fixed point and numeric computing in general. We do not protect the user from these types of problems.

### 4.3.4. Memory allocation

We gain some memory savings from placing uniform variables and expressions in microprocessor memory and from using fixed point instead of floating point, yet the primary feature that allows shaders to have any hope of working on PixelFlow is the memory allocation done by the compiler. Since every surface shader is running different code, there is no reason why the *memory maps* used by all shaders should be the same. We use a different memory map for each shader, with the memory maps determined at compile time.

We have found that, while even the simplest of shaders may define more than 256 bytes worth of varying variables, most shaders do not use that many variables at once. We effectively treat pixel memory as one giant register pool, and perform register allocation on it during compilation. This is one of the most compelling reasons to use a compiler when writing surface shaders to run on graphics hardware. While it is possible to manually do the analysis of which variables can coexist in the same place in memory, it is not easy. It took the author about a month to do just such an analysis for the Pixel-Planes 5 shading code (combined with an analysis of the fixed point sizes for all intermediate results). This code was written by hand at approximately the level that the pfman compiler produces. With automatic allocation, it suddenly becomes possible to prototype and change shaders in minutes instead of months.

### 4.3.5. Memory allocation method

The pfman memory allocator was written by Voicu Popescu. It performs a variable lifetime analysis using a static single assignment (SSA) representation of the procedure [Muchnick97] [Briggs92] (see Figure 4.9). First, we go through the shader, creating a new temporary variable for the result of every assignment. This is where the method gets its name: we do a static analysis, resulting in one and only one assignment for every variable. In some places, a variable reference will be ambiguous, potentially referring to one of several of these new temporaries. At these points we replace the reference with a pseudo-function called a *$\phi$-function*. This indicates that, depending on the control flow, one of several variables could be referenced. The arguments to the $\phi$-function are each of the temporary variables that the original reference could potentially use. In these cases, we merge the separate temporaries back together into a single variable. What results is a program with many more variables, but each having as short a lifetime as possible.

```
i = 1;              i1 = 1;             i1 = 1;
i = i + 1;          i2 = i1 + 1;        i2_3 = i1 + 1;
if (i > j) {        if (i2 > j1) {      if (i2_3 > j1) {
   i = 5;              i3 = 5;             i2_3 = 5;
}                   }                   }
j = i;              j2 = φ(i2,i3);      j2 = i2_3;
        a                   b                   c
```

**Figure 4.9. Example of SSA analysis.**

**a) original code fragment. b) code fragment in SSA form. Note the new variables used for every assignment and the use of the $\phi$-function for ambiguous assignment. c) final code fragment with $\phi$-functions merged.**

Following the conversion to SSA form, we make a linear pass through the code, mapping the new variables to free memory as soon as they become live, and unmapping them when they are no longer live. Variables can only become live at assignments and can only die at their last reference. As a result of these two passes, variables with the same name in the user's code may shift from memory location to memory location. We only allow these shifts when the SSA name for the variable changes. This costs us a little in the amount of memory available to called functions, but eliminates excess copies whose sole purpose would be to keep active memory compact. One of the major effects of the SSA analysis is that a variable that is used independently in two sections of code may not actually reside anywhere between the two sections.

Every shader and function allocates memory from an offset of zero relative to a *frame pointer*. This allows us to make the best use of our aggressive allocation without locking the exact memory locations. Therefore, the successive calls of the recursive factorial function in Section 4.1.2 would have progressively higher frame pointers and would use progressively higher actual memory locations for its computations. This places a hard limit on how big a number this factorial function can compute, and shows that general recursion is not practical in pfman.

```
//*** 79:     uInt1 row = px_shader_texcoord[1] / brick_height; ***//

emc_ufixed2fp(pfman_p, pfman_map,
   /* pfman_tmp0 */ pfman_frameptr,
   /* px_shader_texcoord */ px_shader_texcoord + (int)(1*2));
emc_fp_intomem(pfman_p,
   /* pfman_tmp1 */ pfman_frameptr + 4,
   (1. / brick_height));
emc_fp_mul(pfman_p,
   /* pfman_tmp2 */ pfman_frameptr,
   /* pfman_tmp0 */ pfman_frameptr,
   /* pfman_tmp1 */ pfman_frameptr + 4,
   /* 14 byte temp */ pfman_map->mark());
emc_fp2fixed(pfman_p, pfman_map,
   /* row */ ufixed<8,0>(pfman_frameptr) + 4,
   /* pfman_tmp2 */ pfman_frameptr);
```

**Figure 4.10. Generated code**

Figure 4.10 shows a short section of code generated by the pfman compiler for part of the brick shader of Chapter 1. All of the temporary values and local variables are relative to the frame pointer, but the shader parameters are referenced directly. Several temporaries, as well as the local variable result of the expression are stored in the same place (at different times). Finally, the reciprocal of the uniform variable, `brick_height`, is computed at the time this C++ code is executed, while the instructions for the pixel computations are placed in the SIMD instruction stream buffer, `pfman_p`.

## 4.3.6. Memory allocation results

Figure 4.11 shows the performance of the memory allocator on an assortment of shaders. Images generated with these shaders are shown in Figure 4.12.

| shader | uniform + varying | varying | varying with allocation |
|---|---|---|---|
| simple brick | 171 | 97 | 16 |
| fancy brick | 239 | 175 | 101 |
| ripple reflection | 341 | 193 | 137 |
| planks | 216 | 152 | 97 |

**Figure 4.11. Shader memory usage in bytes.**

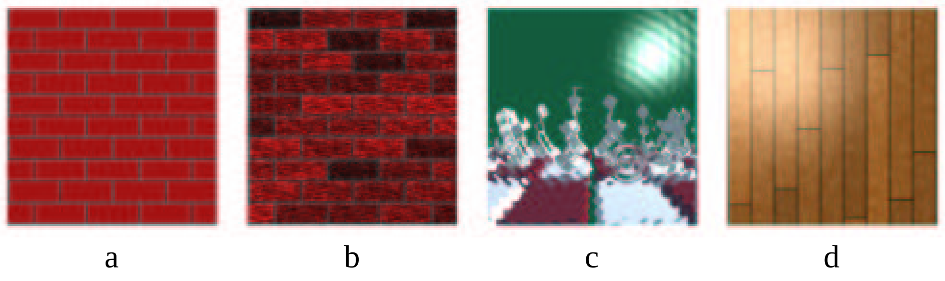

**Figure 4.12. Example surface shaders**
**a) simple brick. b) fancy brick. c) ripple reflection. d) wood planks**

These numbers show that the distinction between uniform and varying variables makes a large difference in the memory use by the shaders, while the success of the memory allocation can vary from shader to shader. The actual memory usage numbers are hard to interpret because they do not include the space taken by the shaders parameters, the shader-light communication area, and the other pixel-memory overhead. An easier statistic to interpret is the amount of free pixel memory available during the shader execution. We have collected this information, along with the SIMD execution time for some of the shaders in Figure 4.12, as well as some of the user-written shaders from Chapter 6. These are shown in Figure 4.13

The second nanoManipulator shader is particularly revealing. With only one byte free, it is barely able to fit. This shader has already been largely converted to fixed-point to get to the point where it would run at all. This shows that with only 256 bytes of memory, even with memory allocation, it is not always possible to prototype an entire shader in floating point. For larger shaders, an incremental approach is necessary, converting pieces of the shader to fixed point as they are developed to make room. Further, the UNC nanoManipulator project wants to combine the second nanoManipulator shader with the BRDF shader. That large a shader may not be possible at all in the space available on PixelFlow.

| **shader** | **bytes free** | **time** |
|---|---|---|
| fancy brick | 46 | 613.15 µs |
| ripple reflection | 59 | 1058.07 µs |
| planks | 105 | 532.30 µs |
| bowling pin | 86 | 401.96 µs |
| nanoManipulator (texture and bumps) | 75 | 567.95 µs |
| nanoManipulator (texture, bumps, and spot noise) | 1 | 2041.44 µs |
| BRDF | 51 | 1638.67 µs |

**Figure 4.13. Shader execution time and memory.**

## 4.4. Bandwidth optimizations

There are two communication paths between boards in the PixelFlow system (see Section 3.3. The *geometry network* allows communication between the microprocessors and the *composition network* allows communication between pixels. We are most concerned with the bandwidth of the composition network. The total effective bandwidth of the composition network is 11.2 GB/s if we use simultaneous transfers in both directions or 5.6 GB/s if we only send data in one direction at a time.

As mentioned in Section 3.1, PixelFlow uses deferred shading. The rendering boards store varying shading parameters and a *shader ID* in the pixels. The complete set of data for each visible pixel must be transferred over the composition net from the rendering boards to a shader board, then a final color from the shader board to the frame buffer. The design of the composition network allows these two transfers to be overlapped, so we really only pay for the bandwidth to send data for each visible pixel from the rendering boards to shading boards. At 30 frames per second on a 1280x1024 screen (or 640x512 screen with 4 sample antialiasing), and accounting for the transfer chunk size, this results in a maximum communication budget of 280 bytes per pixel for bi-directional compositions or 140 bytes per pixel for uni-directional compositions. To deal with this limited communication budget, we have to perform some optimizations to reduce the number of parameters that need to be sent from renderer board to shader board.

### 4.4.1. Shader-specific maps

Even though each 128x64 pixel region is sent as a single transfer, each pixel could potentially be part of a different surface. Rather than use a transfer that is the union of all the parameters needed by all of those surface shaders, we allow each to have its own tailored transfer map. The first two bytes in every map contain the *shader ID*, which indicate what transfer map was used and which surface shader to run.

### 4.4.2. Bound parameters

The bound parameters of any shader instance cannot change from pixel to pixel (Section 4.2.2), so they are sent over the geometry network directly to the shading nodes. Since the shading nodes deal with visible pixels without any indication of when during the frame they were rendered, we must restrict bound parameters to only be changed between frames. The bound uniform parameters are used directly by the shading function running on the microprocessor. Any bound varying parameters must be loaded into pixel memory. Based on the shader ID stored in each pixel, we identify which pixels use each shader instance and load their bound varying parameters into pixel memory before the shader executes.

Any parameter that is bound in every instance of a shader should probably be uniform, since this gives other memory and execution time gains. Yet, it is occasionally helpful to have bound values for varying shading parameters. For example, our brick shader may include a `dirtiness` parameter. Some brick walls will be equally dirty everywhere. Others will be dirtiest near the ground and clean near the top. The instance used in one wall may have `dirtiness` as a bound parameter, while the instance used in a second wall allows `dirtiness` to be set using `glMaterial` with a different value at each vertex.

However, the set of parameters that should logically be bound in some instances and not in others is small. Allowing bound values for varying parameters would be only a minor bandwidth savings, were it not for another implication of deferred shading. Since bound parameters can only change once per frame, we find parameters that would otherwise be uniform are being declared as varying solely to allow them to be changed with `glMaterial` from primitive to primitive (instead of requiring hundreds of instances). This means that someone writing a PixelFlow shader may make a parameter varying for flexibility even though it will never actually vary across any primitives. Allowing instances to have bound values for all parameters helps counter the resulting explosion of pseudo-varying parameters.

### 4.4.3. Explicit shader parameters

RenderMan defines a certain set of *standard parameters* that are implicitly available for use by every surface shader. The surface shader does not need to declare these parameters and can just use them as if they were global variables. In pfman, these parameters must be explicitly declared. This allows us to construct a transfer map that contains only those parameters that are actually needed by the shader.

In retrospect, it would have been possible to do a static analysis of the shader function to tell which of the built-in parameters is used. This would have had the positive effect of making pfman that much more like RenderMan, and consequently that much easier for new users already familiar with RenderMan. On the other hand, it would also have prolonged the time it took to develop pfman.

## 4.5. Execution optimizations

Execution-time optimizations are the final type of optimization necessary for the practical use of shaders on PixelFlow. A frame rate of 30 frames per second translates to 33 ms per frame. The system pipelining means that most of this time is actually available for shading. Based on 160 regions to shade for a high-resolution or antialiased NTSC video display, each shading node is responsible for 40 regions on a system with four shading nodes, and can take an average of 825 μs to shade each region. On a larger system with 16 shading nodes, each is responsible for 10 regions and can spend an average of 3.3 ms shading a region. To put these times in perspective, the rippled reflection shader (see Figure 4.12c) takes 1.1 ms to run. Even if this is the only shader, we cannot achieve our target frame rate using only four shading nodes. With more shading nodes we can achieve the target frame rate with time left over to shade other surfaces (though hopefully not too many others of that shading complexity).

### 4.5.1. Deferred shading

Deferred shading is the technique of performing shading computations on pixels only after the visible pixels have been determined. It provides several advantages for the execution of surface shading functions. First, no time is wasted doing shading computations on pixels that will not be visible. Second, our SIMD array can simultaneously evaluate an instance of a surface shader on every primitive that uses it in a 128x64 region. Finally, it decouples the rendering performance and shading performance of the system. To handle more complex shading, add more shading hardware. To handle more complex geometry, add more rendering hardware. On PixelFlow, where the boards for both are identical, the balance between rendering performance and shading performance can be changed on an application by application basis.

### 4.5.2. Fixed point

In addition to their memory advantages, we can achieve significant speed improvements by using fixed point operations instead of floating point. Our pixel processors do not support floating point in hardware, so every floating point operation is built from basic integer math operations. In contrast, fixed point operations correspond to a single integer math operation and a small number of shifts for alignment. Essentially, a fixed point number is like a floating point number where the exponent is a compile-time constant. As a result, some of the run-time pixel computations required for floating point become compile-time constants. Some operations present a bigger advantage for fixed point than others. Figure 4.14 shows a comparison for several operations. Addition (and subtraction) have the highest penalty for floating point. Multiplication and division have similar costs for both fixed and floating point because they do not require the shifts for alignment that are necessary for addition and subtraction. As expected, the fixed-point advantage for more complex operations falls in between these extremes. Here, `sqrt` is a square-root operation and `noise` is a band-limited noise function, a common building block for shaders.

| Operation | 16-bit fixed | 32 bit fixed | 32-bit float |
|---|---:|---:|---:|
| `+` | 0.07 μs | 0.13 μs | 3.08 μs |
| `*` | 0.50 μs | 2.00 μs | 2.04 μs |
| `/` | 1.60 μs | 6.40 μs | 7.07 μs |
| `sqrt` | 1.22 μs | 3.33 μs | 6.99 μs |
| `noise` | 5.71 μs | — | 21.64 μs |

**Figure 4.14. Fixed point and floating point execution times.**

The fixed point noise function listed in Figure 4.14 was implemented by Yulan Wang, and the remaining fixed point operations were written by Peter McMurry and Greg Pruett. The floating point noise function was implemented by the author and the remaining floating point operations were written by John Eyles and Steve Molnar.

### 4.5.3. Math functions

To round out the varying math capabilities, the author created floating point versions of the remaining standard math library functions. Efficient SIMD implementation of these functions requires a slightly different approach than a serial implementation would. The typical way to implement a transcendental math function (`sin`, `asin`, `exp`, `log`, …) is with a piece-wise polynomial approximation. First the domain is folded using identities of the particular function. For example, for `log(x)`, we write `x` as $m*2^e$ with $m \in [1,2)$. If `x` is floating point, it is already in this form.

$$\log(x) = \log(m*2^e) = \log(2^e) + \log(m) = e \log(2) + \log(m)$$

It is enough to approximate `log(m)` between 1 and 2. Normally, the domain is further reduced with a table of polynomials. For example, the math library distributed with SunOS 4.1 [Sun89] divides this [1,2) domain into 32 segments and fits each with a different fifth-order polynomial.

This approach presents a problem on PixelFlow due to the handling of conditionals on a SIMD array. For a typical `if`/`else`, a normal serial processor evaluates the condition, then executes either one branch or the other. On a SIMD array, the condition determines which processing elements are enabled. The true part is executed with some processing elements enabled, then the set of enabled processors is flipped and the false part is executed. Thus the SIMD array spends the time to execute both branches of the `if`.

For the `log` function example, this means that using a table of 32 polynomials takes as much time as a single polynomial for the entire [1,2) domain with 32 times as many terms. Even so, a polynomial with 160 terms is not practical. For each PixelFlow math function, we reduce the function domain using identities (e.g. getting an approximation domain from 1 to 2 for the `log` function), but do not reduce it further. We fit this domain with a single polynomial. Each polynomial is chosen to use as few terms as possible while remaining accurate to within the floating point precision.

Each approximation must have relative error that is less than the error in the floating point representation. The mantissa of a floating point number has an error of $2^{-24}$. The full floating point number $m*2^e$ has an absolute error of $2^{-24}*2^e$, or a relative error that

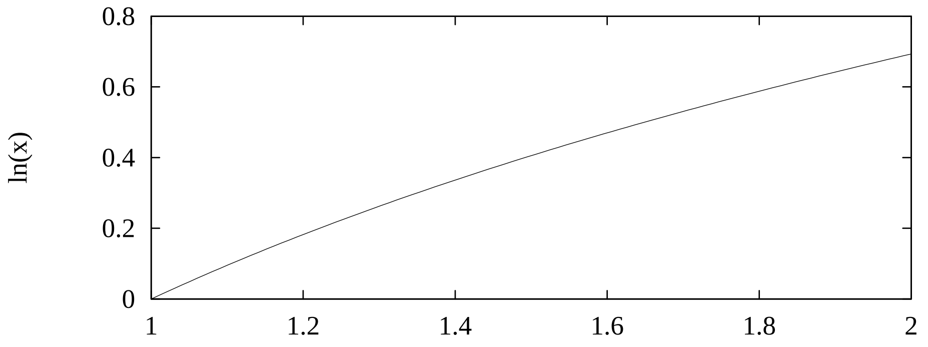

**Figure 4.15. Natural log function over the approximation domain.**

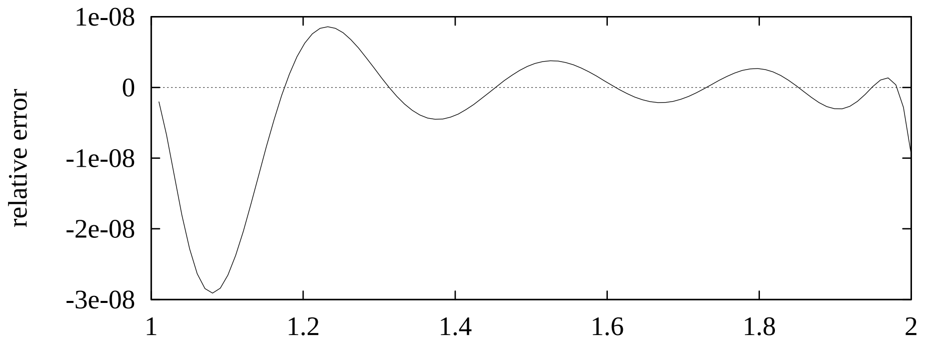

**Figure 4.16. Relative error in natural log approximation.**

ranges from $2^{-24}$ to $2^{-25}$ (about $6*10^{-8}$ to about $3*10^{-8}$) as `m` ranges from 1 to 2. The major determining factor in the accuracy of the polynomial approximation is the number of terms. Once we have selected the right number of terms for p(x), the actual coefficients can vary quite a bit and still be within floating point accuracy.

We want to minimize the maximum relative error. This can become quite difficult. Since we only need a solution within the floating point accuracy, we instead solve the easier least-squares minimum relative error over the approximation domain. So to approximate f(x) with p(x), we want to solve for the p(x) that minimizes

$$\int_{\mathbb{D}} \left(\frac{p(x)-f(x)}{f(x)}\right)^2 dx \tag{4.1}$$

over the approximation domain, $\mathbb{D}$. We find p(x) by solving for the coefficient vector, $\vec{v}$ in

$$\frac{d}{d\vec{v}} \int_{\mathbb{D}} \left(\frac{p(x)-f(x)}{f(x)}\right)^2 dx = \vec{0} \tag{4.2}$$

For most of the math functions, even Equation 4.2 has no closed form solutions. However, we can factor it into terms that are linear in the elements of $\vec{v}$ and solve with numeric integration.

This works well for areas where f(x) does not approach 0. If $f(x_0)=0$ for some $x_0 \in \mathbb{D}$, the relative error goes to infinity as x approaches $x_0$. In these cases, we salvage the approximation by constraining $p(x_0)$ and $p'(x_0)$ to match exactly while still minimizing the least-squares relative error. Figure 4.15 shows the natural log function over the approximation domain. Figure 4.16 shows the relative error of a 10th order approximation made by constraining the value and first derivative at x=1 and minimizing the least-squares relative error over the rest of the domain.

| **function** | **exact** | **fast** |
|---|---:|---:|
| `sin` | 81.36 µs | 45.64 µs |
| `cos` | 81.36 µs | 48.77 µs |
| `tan` | 93.25 µs | 52.65 µs |
| `asin, acos` | 78.52 µs | 47.50 µs |
| `atan` | 66.41 µs | 35.34 µs |
| `atan2` | 66.17 µs | 35.15 µs |
| `exp` | 53.37 µs | 37.86 µs |
| `exp2` | 51.09 µs | 35.58 µs |
| `log` | 57.76 µs | 21.57 µs |
| `log2` | 57.68 µs | 21.49 µs |

**Figure 4.17. SIMD execution time for floating point math functions.**

We provide these accurate versions of the math functions, but often shaders do not really need the “true” function. With the ripple reflection shader in Figure 4.12c, it is not important that the ripples **be** sine waves. They just need to **look like** sine waves. For that reason, we also provide faster, visually accurate, but numerically poor versions of the math functions. The fast versions use simpler polynomials, just matching value and first derivative at each endpoint of the range fit by the more exact approximations. This provides a function that appears visually correct while not requiring an excessive number of terms.

### 4.5.4. Combined execution

Many shading functions follow the same general mold. On a SIMD system, the shaders are executed sequentially, so the time spent shading is the sum of the time for all of the shaders. Combining the execution of the common sections of code in multiple shaders can lead to large gains in performance. If we find an expensive operation performed by each of ten shaders in a scene, and manage to execute that expensive operation only once, it is equivalent to making the expensive operation ten times faster.

This combination of operations is similar to the work of Dietz for combining execution of code within a single SIMD procedure [Dietz92]. On a SIMD processor, even a simple `if` statement requires executing the `then` clause and `else` clause sequentially. Dietz' work with common subexpression induction allows code that appears in both to be combined and executed only once.

Rather than attempt common subexpression induction at a fine grain within a shader or between shaders, we have focused on combining the large, expensive operations shared by different shaders. The easiest and most automatic of these types of optimizations is the combined execution of lights for all surface shaders. For some of the more "traditional" surface shaders, involving image texture lookups and Phong shading, we can do further overlapped computation. Different surface types that share the same surface shading function can sometimes be executed together. Finally, there is some interesting possible future work with generalizing this class of overlapped execution optimizations.

#### Lights

One of the jobs of a surface shader is to incorporate the effects of each light in the scene. This is accomplished through the `illuminance` construct, which behaves like a loop over the active lights (see Figure 4.18). The `illuminance` construct is covered in more detail in Section 4.1.4. This means that each surface shader effectively includes a loop over every light. For `m` shaders and `n` lights, this would result in the execution of `m*n` lights. This would be quite expensive since the lights themselves are procedural, and could be arbitrarily complex. However, since the lights are the same for each of the `m` shaders, we can compute each light just once and share its results among all of the shaders, resulting in the execution of only `n` lights. We do this by interleaving the execution of all of the lights and shaders.

```
// setup, compute base surface color
illuminance() {
   // incorporate the contribution of one light
}
// wrap up
```

**Figure 4.18. Outline of a typical surface shader.**

We accomplish this interleaving by having the compiler generate three SIMD instruction streams for each shader function. The first stream, which we call *`pre-illum`*, contains only the setup code (until the `illuminance` in Figure 4.18). The second stream contains the body of the `illuminance` construct. We call this the *`illum`* stream. Finally, the *`post-illum`* stream contains everything after the `illuminance`. The lights themselves create their own stream of SIMD commands. The interleaving pattern of these streams is shown in Figure 4.19.

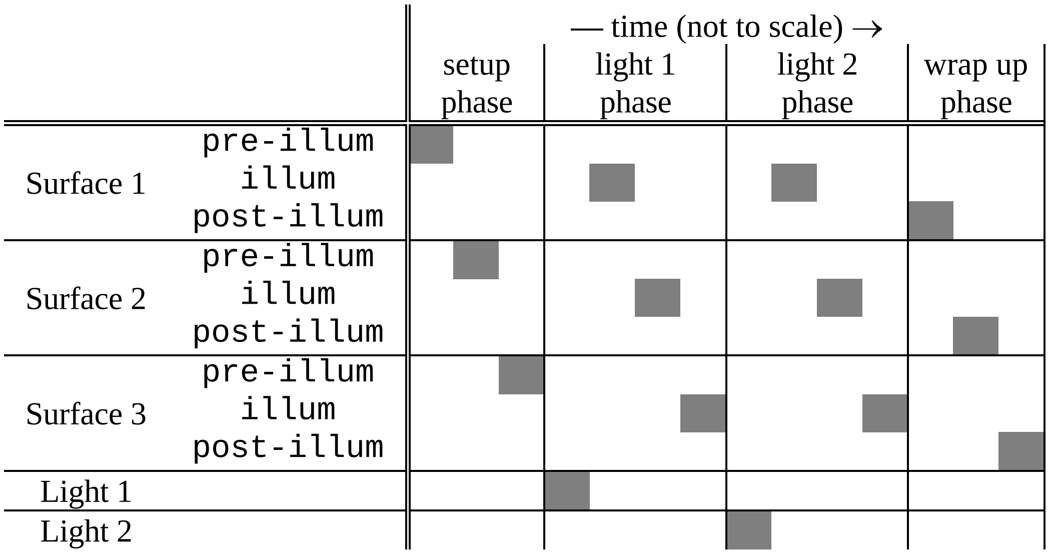

**Figure 4.19. Interleaving of surface shaders and lights**

The memory usage of the surfaces and lights must be chosen in such a way that each has room to operate in SIMD memory, but none conflict. The surface shaders cannot interfere with each other since any one pixel can only use one surface shader. Different surface shaders already use different allocations of pixel memory. Lights, however, must operate in an environment that does not disturb any surface shader, but provides results in a form that all surface shaders can use. The results of the lighting computation, the color and direction of the light hitting each pixel, are stored in a special communications area to be shared by all surface shaders. The light functions themselves operate in the SIMD memory left over by the greediest of the surface shader `pre-illum` stages. Above this *high water mark*, the light can freely allocate whatever memory it needs. The `illum`, and `post-illum` streams of all shaders can use all available memory without interfering with either the other surfaces or the lights.

**Surface position**

For image composition, every pixel must contain Z-buffer depth of the closest surface visible at that pixel. This Z value, along with the position of the pixel on the screen, is sufficient to compute the location of the surface sample in 3D. Since the surface position can be reconstructed from these two pieces of information, we do not explicitly store the surface position in pixel memory during rendering, or waste composition bandwidth sending it from the rendering boards to the shading boards. Instead, we compute it on the shading boards in a phase we call `pre-shade`, which occurs before any shading. Thus, we save memory and bandwidth early in the graphics pipeline, and share the execution time necessary to reconstruct the surface position once we need it.

**Support for traditional shaders**

Two optimizations have been added to assist in cases that are common for forms of the OpenGL shading model. These can be used by procedural shaders as well, though the interface is not as automatic as the shared lighting computations. Shared lighting computations can be applied to any shader without special coding by the shader writer, while these special-purpose optimizations rely on the shader-writer to declare certain “magic” parameters to enable the optimization.

Surface shaders that use only the typical Phong shading model can use a shared `illum` stream. Use of this shared stream is switched on for any shader that declares the `px_rc_spec_co` and `px_rc_phong_co` parameters. In the `post_illum` portion of the shaders, these parameters contain the results of the shared illuminance computation. This optimization is not easily detected automatically, so it is much easier to rely on the intelligence of the shader-writer directly.

Surface shaders that perform a certain class of texture lookups can share the lookup computations. The PixelFlow hardware does not provide any significant improvement in actual lookup time for shared lookups, but the shared lookup allows illuminance computations to happen while the lookup is happening. The optimized texture lookup is used by a number of variants of the OpenGL shading model. These shaders know what texture they want to look up in the `pre-illum` phase, but don't require the results until the `post-illum` phase. To share the lookup processing, they place their *texture ID* and *texture coordinates* in special shared "magic" parameters. The results of the lookup are placed in another shared magic parameter by the start of the `post-illum` stage.

### 4.5.5. Cached instruction streams

As mentioned in Section 4.3.2, PixelFlow shading code consists of two conceptual parts. Any operations involving only uniform quantities execute only on the shading board microprocessor. Any operations involving varying quantities, or a mixture of uniform and varying, occur on the SIMD pixel processors. The microprocessor code computes the uniform expressions and all of the uniform control flow (`if`'s with uniform conditions, `while`'s, `for`'s, etc.), generating a stream of SIMD processor instructions. This SIMD instruction stream is buffered for later execution. The set of SIMD instructions only changes when some uniform parameter of a shader changes, so we cache the instruction stream and re-use it. Any parameter change sets a flag that indicates that the stream must be regenerated. For most non-animated shaders, this means that the uniform code executes only once when the application starts.

### Shader groups

Different *shader instances* that use the same shader function can sometimes be executed together. The set of shader instances that may be executed together is called a *shader instance group*. We have elected not to use this optimization. The payoff when it succeeds can be high, but it is costly to find the groups. The merging of instances into instance groups can only be done while the application is running, and may change from frame to frame. By contrast, the earlier optimizations in this section can all be statically determined at compile-time. Since, in our limited experience, we have yet to see any applications with a group size larger than one, any effort spent finding groups would be wasted. However, we will discuss some of the issues in using shader groups, should they prove practical in the future.

Shader instances that share a pfman function produce different streams of SIMD instructions if their uniform parameters differ. For example, one uniform parameter might be the number of times to execute a loop in the shader. Consequently, a shader group consists only of those instances that do not differ in any uniform parameter values, though they may differ in bound values for varying parameters. Since the values of the bound parameters can change from frame to frame, a particular instance may change from shader group to shader group. In addition, the number of shader groups may change.

To utilize the optimization, we must be able to rapidly identify which group to use for an instance when one of its uniform parameters changes. This may also involve creating a new group or removing an old one. We can rapidly limit the number of groups to check by creating a simple hash function of all of the uniform parameters. The more complete, parameter by parameter, comparison need only be done for groups that match the hash value. A particularly attractive hash is a simple checksum. When a parameter value changes, it is easy to recompute the checksum by subtracting the contribution of the old parameter value and adding in the contribution of the new parameter value. Once we identify the shader instance groups, it is easy to combine their execution by enabling a set of *shader IDs* instead of only one.

### 4.5.6. Identification of active shaders

Just because a shader is loaded does not mean that it is being used. Even if it is used, all of the primitives that use it may be entirely off screen. Even if they are on screen, they may not fall in a particular region. Even if they fall in the region, they may not be visible. Time spent running shaders that do not even appear in the image is wasted. There are several levels of active shader detection that we could support, corresponding to the ways that a loaded shader might not be used.

The most advanced and accurate technique is to detect which shaders actually appear in image pixels in a region. The PixelFlow hardware includes an Enable OR (`EOR`) test, provided specifically to make this possible. `EOR` can tell whether any processors in the SIMD array are enabled.

Normally, the code running on each board's microprocessor generates a buffer of SIMD instructions to be sent at some later time. To use the `EOR` test, we must generate instructions to be sent to the SIMD array while we wait. The basic algorithm is to enable only the pixels using a particular shader, then check `EOR` to see if such pixels exist. Since only one `EOR` test can be done at a time on PixelFlow, its use is further complicated by the buffering that exists in hardware and software. Before any such test can be started, we must wait for the queued and buffered instructions to complete. For this reason, an `EOR` test would be best done once per region, for all shaders, instead of on a shader-by-shader basis. Even if the `EOR` test were only done once per region, it still upsets the buffering and load balancing of the machine.

Shaders tend to be applied on an object-by-object basis, so the pixels using a particular shader tend to be close together. Even with the buffering and load balancing concerns, the `EOR` test would be worthwhile for scenes with large numbers of shaders since all shaders are unlikely to appear in a single region. However, we did not run into any such scenes in our testing, so we instead focused our energies on other forms of optimization.

## 4.6. Stages in shading

All of the attempts to share execution between different shaders or other parts of the system mean that the software stages actually run on PixelFlow are quite different from the simple abstract pipeline presented to the users and covered in Chapter 2. Figure 4.20 shows the full set of stages

| | | |
|---|---|---|
| rendering node | `rasterize` | `primitive` |
| | `post-rast` | — |
| | `compress` | — |
| | \| | |
| shading node | `uncompress` | — |
| | `pre-shade` | — |
| | `pre-illum` | `shade` |
| | `light` | `light` |
| | `illum` | `shade` |
| | `post-illum` | `shade` |
| | `post-shade` | — |
| | `fog` | `atmospheric` |
| | `blend` | — |
| | \| | |
| frame buffer node | `frame-buffer` | `warp` |
| | a) | b) |

**Figure 4.20. Comparison of PixelFlow software stages and abstract stages**

**a) All of the stages present in the PixelFlow software. b) The mapping of these stages into the abstract pipeline stages.**

The `rasterize` stage handles the transformation, rasterization, and interpolation stages of the abstract pipeline of Chapter 2. The `post-rast` stage handles operations that can be shared by many or all primitives and that reduce the required composition network bandwidth. For example, perspective correction of shading parameters is done in `post-rast`. `Compress` removes empty spaces in the memory map before composition. `Uncompress` expands the map back out again and fills in the bound parameter values for any pixel-memory bound parameters. As mentioned in Section 4.5.4, `pre-shade` handles common tasks like the computation of surface position that can be shared by all shaders. `pre-shade` and `post-rast` serve similar functions on opposite sides of the composition network transfer. Particular computations are placed in one or the other, based on which will allow the minimum composition network message size from rendering node to shading node. `Pre-illum`, `light`, `illum`, and `post-illum` were discussed in Section 4.5.4. `Fog` corresponds to the fog or atmospheric stage of the abstract pipeline of Chapter 2. `Blend` does blending of image samples into a single pixel value for anti-aliasing. Blending occurs on the shading board to reduce the composition network bandwidth required between the shading board and frame buffer boards. Finally the `frame-buffer` stage handles copying the incoming image pixels into the frame buffer, including any required warping.