PixelFlow Shading Language
Marc Olano
Revised: 15 September 1997

1 Overview

The PixelFlow shading language is a special purpose C-like language for describing the shading of surfaces on the PixelFlow graphics system. On PixelFlow, some shading function written in the shading language is associated with each primitive. The shading function is executed for each visible pixel (or sample for antialiasing) to determine its color. The language is based heavily on the RenderMan shading language[1].

2 Data

2.1 Built in types

Only a few simple data types are supported. The simplest type is **void**. As with C, it is only used as a return type for functions that have no return value. There is a floating point type, **float**, used for most scalar values. There is a fixed point type, **fixed**, provided for efficiency. And there are literal strings, useful for print formatting[2]. Note that, unlike RenderMan, the 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 bits. Note however, that an exponent larger than the size or less than zero is perfectly legal. A two byte integer would be **fixed<16, 0>**, while a two byte pure fraction would be **fixed<16, 16>**. It is possible to translate back and forth between the real value and stored value using these equations:

$$\text{real\_value} = \text{stored\_value}^{-\text{exponent}}$$
$$\text{stored\_value} = \text{real\_value}^{\text{exponent}}$$

However, it is much less confusing to always think of the real value. For example, with a **fixed<8,8>**, never think of the value as 128, instead think 0.5. An unspecified fixed point type can also be used, declared simply as **fixed**, and its size and exponent will be chosen automatically[3].

It is also possible to have arrays of these basic types, declared in a C-like syntax (i.e. **float color[3]**). The declaration **float color[3]**, declares color to be a 1D array of three **float**s, **color[0]**, **color[1]**, and **color[2]**. You can also look at **color** as a variable of type **float[3]**, and an equivalent definition would be **float[3] color**. Note the behavior of mixing these two types of definitions: **float[2][3] color_list**, **float[3] color_list[2]** and **float color_list[2][3]** are all equivalent. 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. Where RenderMan uses separate types for points, vectors, normals, and colors, pfman uses arrays.

2.2 Type attributes

As with RenderMan, types may be declared to be either **uniform** or **varying**. A **varying** variable is one that might vary from pixel to pixel, similar **plural** in MasPar's mpl. A **uniform** variable is one that will not vary from pixel to pixel, similar to **singular** in MasPar's mpl. It deserves mentioning again that 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**.

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**.

---

[1] Upsill, Steve, The RenderMan Companion, Addison-Wesley, 1990.

[2] As of September 13, 1997, strings for calls to printf are not supported.

[3] As of September 13, 1997, automatic fixed point variables are not supported. The sizes produced by automatic fixed types will have to be pessimistic in their size estimation. Error analysis and explicit fixed point sizes is sure to make better use of memory.

There are a number of additional attributes for shader parameters. One transformation type can be given for any parameter. These are **transform_as_vector**, **transform_as_normal**, **transform_as_point**, **transform_as_plane**, or **transform_as_texture**[4]. A parameter can also be declared to be **unit** if it should be unit length[5]. For example, you might declare a parameter

unit transform_as_vector float v[3];

These attributes only affect what happens to the parameter before it is passed to the shader. They do not affect how the parameter is used inside the shader. For example, a **unit** parameter will not remain unit length. These attributes also cannot be used to distinguish versions of an overloaded function.

2.3 User defined types

Aliases can be defined for types with a C-like **typedef** statement. **typedef** is only legal outside function definitions. The **typedef** statement only provides aliases for types, 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 which can be used completely interchangably with **float[3]**.

3 Expressions

3.1 Operators

The set of operators and operator precedence is fairly 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 1.

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[6]
`* / %`	left	multiplication, division, mod
`+ -`	left	addition, subtraction
`&`	left	bitwise and[7]
`|`	left	bitwise or[8]
`<< >>`	left	shift[9]
`< <= >= >`	left	comparison
`== !=`	left	comparison
`&&`	left	logical and
`||`	left	logical or
`?:`	right	conditional expression
`= += -= *= /= ^=`	right	assignment[10]
`,`	—	expression list

Figure 1. Operator precedence

3.2 Operations on arrays[11]

Operations on arrays are defined as the corresponding vecor, matrix, or tensor operation. The unary operations act on all elements of the array. Addition, subtraction, and assignment require arrays of equal

---

[4] As of March 4, 1995, vectors and points are transformed the same and normals and planes are transformed the same.

[5] As of September 13, 1997, **unit** has no affect (parameters declared **unit** are not normalized).

[6] As of March 4, 1995, none of xor, cross product, or wedge product are implemented.

[7] & only works between identical fixed point types.

[8] | only works between identical fixed point types.

[9] As of September 13, 1997, left and right shift are only implemented for varying integer shift values

[10] As of September 13, 1997, ^= is not implemented

[11] As of September 13, 1997, Array cross product, and inverse do not work.

dimension and do 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 defined.

Multiplication between vectors gives a dot product, between vector and matrix, matrix and vector, or matrix and matrix gives the appopriate 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. In other words, for **float a[3][3][3]**, **float b[3][3][3]** and **float c[3][3][3][3]**,

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

is equivalent to

```
float i, j, k, l;
for(i=0; i<3; i++)
    for(j=0; j<3; j++)
        for(k=0; k<3; k++)
            for(l=0; l<3; l++) {
                c[i][j][k][l] = 0;
                for(m=0; m<3; m++)
                    c[i][j][k][l] += a[i][j][m] * b[m][k][l];
            }
```

Division can also be used as a matrix inverse. **1 / a** is the inverse of a square matrix **a** and **b / a** multiplies **b** by the inverse of square matrix **a**.

Finally, the ^ operator gives the cross product between two vectors or the tensor wedge product between two arrays.

3.3 Inline arrays[12]

C-style array initializers are allowed in any expression as an anonymous array. So 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}**.

3.4 Einstein summation notation[13]

Inside any statement block, the uniform integer variables **$1**, **$2**, … are automatically defined. For example for **float a[3], b[3]**, the expression **a[$1] * b[$1]** is equivalent to **a[0]*b[0] + a[1]*b[1] + a[2]*b[2]** (which in this case, is equivalent to **a * b**).

4 Statements

4.1 Compound statements

As with 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 **}**.

4.2 Expression statements

Any expression followed by a **;** is a legal statement.

4.3 Standard control statements

Most of the control statements are borrowed directly from C.[14]

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;

---

[12] As of September 13, 1997, inline arrays can only have constants for their array elements.

[13] As of September 13, 1997, Einstein summation notation is not implemented

[14] Due to limitations of PixelFlow, the condition_expression's must be **uniform** for all of the looping control statements. The condition for an **if** can be either **uniform** or **varying**.

return;
return return_value_expression**;**

In addition, there are several control statements taken from the RenderMan shading language to aid in shading. They are **illuminance**, **illuminate**, and **solar**.

The **illuminance** statement,

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

acts like a loop over the available light sources. It can also be thought of as an integral over the incoming light. For each light that can hit a pixel at the given position, or can hit a surface at the given position with the given orientation and visibility angle, the light source function is run, returning a light color and intensity that can be used in the statement. The light direction can be accessed using the **px_rc_l** parameter to the shader. The light color can be accessed using the **px_rc_cl** parameter to the shader.

The **illuminate** and **solar** statements,

illuminate (position_expression) statement
illuminate (position_expression**,** axis_angle**,** angle_expression) statement
solar (axis_angle**,** angle_expression) statement
solar ( ) statement

provide the information the **illuminace** statement uses to tell if a light source function should be run or not. They can also be thought of as conditional statements that only execute the associated statement if the current pixel position falls within the light's area. The four statements above correspond to a point light, a spot light, a directional light, and an ambient light[15].

4.4 Declaration statements

Variable declarations can occur anywhere a statement can. They consist of a type and a list of new variable names to declare. Each variable name can have additional array dimensions and an expression for the initial value.

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 whatever is in the **x** variable at the declaration time, and **c** as an uninitialized **float**.

Each compound statement defines a new scope, so variables can be redefined within a compound statement without conflicting with function or variable names in other scopes. It is illegal, however, to have a variable in any scope with the same name as any user defined type. This is true even if the **typedef** occurs after the variable declaration.

5 Functions

5.1 Overloading

Function overloading similar to C++ is supported. So functions of the same name that can be destinguished by their input parameters are considered distinct. This provides the ability to have seperate 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 parameters and operator overloading is not supported.

5.2 Definition

A function definition gives the return type, name, parameters, and body that define the function. Function definitions cannot be nested. By default, function parameters and return types are **uniform**. A simple function definition:

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

---

[15] I don't really like the way this works in RenderMan. Is there a use to placing some of the light code within an **illuminate** statement and some outside? Is it too specialized for a couple of particular light types? Whether I understand it or not, it's there.

```
    else
        return 1;
}
```

The formal parameters to a function have their own scope level between the global scope and the function body, so their names can hide the global function names. As with variables, it is illegal to have a function or parameter with the same name as a user defined type, regardless of where in the source the **typedef** occurs.

5.3 Shading functions[16]

There are several special return types to indicate that a function has some special rendering purpose and may need to be called by the PixelFlow rendering library. These are **primitive**, **interpolator**, **surface**, **light**, and **image**. A **primitive** function computes which pixels are in some rendering primitive like a triangle or sphere; an **interpolator** function computes the value for some shading parameter across a number of pixels; a **surface** function computes the shading on a surface (the archetypal shading function); a **light** function computes the color and intensity of a light; and an **image** computes the final color and location of the image pixels (handling image warping, fog effects, etc.). For all of these functions, each parameter can have a default value in case the graphics library is not given a value for that parameter. These are given just by putting an **= value** (just like variable initialization) in the parameter list. These default values must be compile-time constants. It is perfectly legal to call a surface shading function from inside another surface shading function[17]. In this case, only one illuminance statement can occur in either the original surface shader or any called by it.

5.4 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. A function prototype is just like a function definition, but with a **;** instead of the function body

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

5.5 Internal details and External linkage

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 created by the compiler and function calls made by it 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 types that the shading language supports.

Pfman adds some additional arguments added by the compiler. The new first argument is a pointer to the PixelFlow IGCStream where the instruction stream for the pixel processors should go. The new second argument is a pointer to a PixelFlow GLStage class, which contains information about the rendering context. The new third argument is a pointer to the PixelFlow pixel memory map class. For functions with a varying return value, the new third argument is the address for the return value. All the other arguments follow. There are C++ classes for varying float and fixed parameters giving their address, and in the case of fixed parameters, their size and binary point position. Details of these types and the prototypes for the different kinds of shading functions are beyond the scope of this document.

Standard C or C++ functions can be used by pfman by prefixing their prototype with **extern "C"** or **extern "C++"**. All of the uniform math library routines are declared this way. These tell pfman not to add the extra function parameters. Similarly, pfman functions that contain only uniform operations can be declared **extern "C"** or **extern "C++"** for use by code outside of pfman.

---

[16] As of September 13, 1997, only surface and light are supported.

[17] As of September 13, 1997, it is not possible to call either surface shaders from inside surface shaders.