Interactive Shading Language (ISL)
Language Description
April 6, 2000

Copyright 2000, SGI
all rights reserved

I. Introduction

ISL is a shading language designed for interactive display. Like other shading languages, programs
written in ISL describe how to find the final color for each pixel on a surface. ISL was created as a
simple restricted shading language to help us explore the implications of interactive shading. As such,
the language definition itself changes often. While this may be a snapshot specification for ISL, ISL is
not proposed as a formal or informal language standard. Shading language design for interactive
shading is still an open area of research.

A. Features in common with other shading languages

The final pixel color comes from the combined effects of three function types. A light shader computes
the color and intensity for a light hitting the surface. Several light shaders may be involved in finding the
final color for a single pixel. A surface shader computes the base surface color and the interaction of the
lights with that surface. Finally, an atmosphere shader computes any changes to the color between the
surface and camera, attenuation from fog, for example. The term shader is used to refer to any of these
three special types of function.

All shading code is written with a single instruction, multiple data (SIMD) model. ISL shaders are
written as if they were operating on a single surface pixel in isolation. The same operations are
performed for all pixels on the surface, but the computed values can be different at every pixel.

Like other shading languages that follow the SIMD model, ISL data may be declared varying or
uniform. Varying values may vary from pixel to pixel, while uniform values must be the same at every
pixel on the surface.

B. Major differences from other shading languages

ISL has several differences and limitations that distinguish it from more full-featured shading languages:

- The primary varying data type in ISL is limited to the range [0,1]. Results outside this range are
  clamped.
- ISL does not allow texture lookups based on computed results.
- ISL does not allow user-defined parameters that vary across the surface. Such parameters must
  either be computed or loaded as texture.
II. Data types

All ISL data is classified as either varying or uniform. Varying data may hold a different value at each pixel, while uniform data must have the same value for every pixel on a surface. Uniform values may still differ from surface to surface, or from frame to frame.

The complete list of ISL data types is:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>uniform float</strong> uf</td>
<td>uf is a single floating point value</td>
</tr>
<tr>
<td><strong>uniform color</strong> uc</td>
<td>uc is a set of four floating point values, representing a color, vector or point. For colors, the components are ordered red, green, blue and alpha.</td>
</tr>
<tr>
<td><strong>uniform matrix</strong> um</td>
<td>um is a set of sixteen floating point values, representing a 4x4 matrix in row-major order (all four elements of first row, all four elements of second row, ...)</td>
</tr>
<tr>
<td><strong>uniform string</strong> us</td>
<td>us is string, as for a texture name</td>
</tr>
<tr>
<td><strong>varying color</strong> vc</td>
<td>vc is a four element color, vector or point that may have different values at each pixel on the surface. Elements of the color are constrained to lie between 0 and 1. Negative values are clamped to zero and values greater than one are clamped to one</td>
</tr>
</tbody>
</table>

ISL also allows 1D arrays of all uniform types, using a C-style specification:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>uniform float</strong> ufa[n]</td>
<td>ufa is an array with n floating point elements, ufa[0] through ufa[n-1]</td>
</tr>
<tr>
<td><strong>uniform color</strong> uca[n]</td>
<td>uca is an array with n uniform color elements, uca[0] through uca[n-1].</td>
</tr>
<tr>
<td><strong>uniform matrix</strong> uma[n]</td>
<td>uma is an array with n uniform matrix elements, uma[0] through uma[n-1]</td>
</tr>
<tr>
<td><strong>uniform string</strong> usa[n]</td>
<td>usa is an array with n uniform string elements, usa[0] through usa[n-1]</td>
</tr>
</tbody>
</table>

III. Variables and identifiers

Identifiers in ISL are used for variable or function names. They begin with a letter or underscore, and may be followed by additional letters, underscores or digits. For example a, abc, C93d, _4, and d_e_f are all legal identifiers.

Several variables are predefined with special meaning:
varying color FB  current frame buffer, intermediate result location for almost all varying operations.

uniform float frame  current integer frame number
uniform float time  current elapsed time, in seconds
uniform matrix objectmatrix  matrix to transform from the space where the object was defined to camera space, equivalent to the OpenGL ModelView matrix
uniform matrix shadermatrix  Arbitrary matrix associated with the shader by the application. This may be used to allow the shader to operate in a common space for many independently transformed objects

IV. Uniform Operations

In the following, uf and uf0-uf15 are uniform floats; ufa is an array of uniform floats; uc, uc0 and uc1 are uniform colors; uca is an array of uniform colors; um, um0 and um1 are uniform matrices; uma is an array of uniform matrices; us, us0 and us1 are a uniform strings; usa is an array of uniform strings; and ur, ur0 and ur1 are uniform relations.

A. uniform float

Operations producing a uniform float:
<table>
<thead>
<tr>
<th>variable reference</th>
<th>value of uniform float variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>float constant</td>
<td>Where</td>
</tr>
<tr>
<td></td>
<td>( H = 1 ) or more hex digits (0-9 or a-f)</td>
</tr>
<tr>
<td></td>
<td>( O = 1 ) or more octal digits (0-7)</td>
</tr>
<tr>
<td></td>
<td>( D = 1 ) or more decimal digits (0-9)</td>
</tr>
<tr>
<td></td>
<td>( S = +, - ) or nothing</td>
</tr>
<tr>
<td></td>
<td>One of the following non-case-sensitive patterns:</td>
</tr>
<tr>
<td></td>
<td>( 0xH ) (hex integer);</td>
</tr>
<tr>
<td></td>
<td>( 0O ) (octal integer);</td>
</tr>
<tr>
<td></td>
<td>( D; D.; .D; D.D; )</td>
</tr>
<tr>
<td></td>
<td>( D\eSD; D.eSD; .D\eSD; D.D\eSD )</td>
</tr>
<tr>
<td>(uf)</td>
<td>Grouping intermediate computations</td>
</tr>
<tr>
<td>-uf</td>
<td>negate uf</td>
</tr>
<tr>
<td>uf0 + uf1</td>
<td>add uf0 and uf1</td>
</tr>
<tr>
<td>uf0 - uf1</td>
<td>subtract uf1 from uf0</td>
</tr>
<tr>
<td>uf0 * uf1</td>
<td>multiply uf0 and uf1</td>
</tr>
<tr>
<td>uf0 / uf1</td>
<td>divide uf0 by uf1</td>
</tr>
<tr>
<td>um[uf0][uf1]</td>
<td>Gives element ( \text{floor}(4*uf0 + uf1) ) of matrix um</td>
</tr>
<tr>
<td></td>
<td>Behavior is undefined if ( \text{floor}(4*uf0 + uf1) ) is not in the range 0 to 15</td>
</tr>
<tr>
<td>ufa[uf]</td>
<td>element ( \text{floor}(uf) ) of array ufa where element 0 is the first element.</td>
</tr>
<tr>
<td></td>
<td>Behavior is undefined if ( \text{floor}(uf0) ) falls outside the array.</td>
</tr>
<tr>
<td>f(...)</td>
<td>function call to a function returning uniform float result</td>
</tr>
</tbody>
</table>

Uniform float assignments take the following forms, where lvalue is either a uniform float variable, one element of a uniform matrix variable, accessed as \( \text{var[uf0][uf1]} \), or one element of a uniform float array, accessed as \( \text{var[uf]} \):

<table>
<thead>
<tr>
<th>lvalue = uf</th>
<th>simple assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>lvalue += uf</td>
<td>equivalent to ( \text{lvalue} = \text{lvalue} + \text{uf} )</td>
</tr>
<tr>
<td>lvalue -= uf</td>
<td>equivalent to ( \text{lvalue} = \text{lvalue} - \text{uf} )</td>
</tr>
<tr>
<td>lvalue *= uf</td>
<td>equivalent to ( \text{lvalue} = \text{lvalue} * \text{uf} )</td>
</tr>
<tr>
<td>lvalue /= uf</td>
<td>equivalent to ( \text{lvalue} = \text{lvalue} / \text{uf} )</td>
</tr>
</tbody>
</table>

### B. uniform color

Operations producing a uniform color:
<table>
<thead>
<tr>
<th>variable reference</th>
<th>value of uniform color variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>color(uf0,uf1,uf2,uf3)</td>
<td>(red=uf0, green=uf1, blue=uf2, alpha=uf3)</td>
</tr>
<tr>
<td>uf</td>
<td>color(uf,uf,uf,uf)</td>
</tr>
</tbody>
</table>

C. uniform float

Operations producing a uniform float:

<table>
<thead>
<tr>
<th>variable reference</th>
<th>value of uniform float variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>-uc</td>
<td>Each uniform float operation is applied component-by-component</td>
</tr>
<tr>
<td>uc0 + uc1</td>
<td></td>
</tr>
<tr>
<td>uc0 - uc1</td>
<td></td>
</tr>
<tr>
<td>uc0 * uc1</td>
<td></td>
</tr>
<tr>
<td>uc0 / uc1</td>
<td></td>
</tr>
</tbody>
</table>

| uca[uf]            | element \(\text{floor}(\text{uf})\) of array uca, where element 0 is the first element. Behavior is undefined if \(\text{floor}(\text{uf0})\) falls outside the array. |
| f(...)             | function call to a function returning uniform color result |

Uniform color assignments take the following forms, where lvalue is either a uniform color variable or one element of a uniform color array, accessed as var[uf]

| lvalue = uc       | simple assignment |
| lvalue += uc      | equivalent to lvalue = lvalue + uc |
| lvalue -= uc      | equivalent to lvalue = lvalue - uc |
| lvalue *= uc      | equivalent to lvalue = lvalue * uc |
| lvalue /= uc      | equivalent to lvalue = lvalue / uc |

C. uniform matrix

Operations producing a uniform matrix:

<table>
<thead>
<tr>
<th>variable reference</th>
<th>value of uniform matrix variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix(uf0,uf1,uf2,uf3, uf4,uf5,uf6,uf7, uf8,uf9,uf10,uf11, uf12,uf13,uf14,uf15)</td>
<td>matrix with rows ((uf0,uf1,uf2,uf3), (uf4,uf5,uf6,uf7), (uf8,uf9,uf10,uf11) \text{ and } (uf12,uf13,uf14,uf15))</td>
</tr>
<tr>
<td>(um)</td>
<td>Grouping intermediate computations</td>
</tr>
<tr>
<td>-um</td>
<td>Each uniform float operation is applied component-by-component</td>
</tr>
<tr>
<td>um0 + um1</td>
<td></td>
</tr>
<tr>
<td>um0 - um1</td>
<td></td>
</tr>
</tbody>
</table>

| uma[uf]            | element \(\text{floor}(\text{uf})\) of array uma where element 0 is the first element. Behavior is undefined if \(\text{floor}(\text{uf0})\) falls outside the array. |
| f(...)             | function call to a function returning uniform matrix result |

Uniform matrix assignments take the following forms, where lvalue is either a uniform matrix variable
or one element of a uniform matrix array accessed as `var[uf]`

<table>
<thead>
<tr>
<th>lvalue = um</th>
<th>simple assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>lvalue += um</td>
<td>equivalent to <code>lvalue = lvalue + um</code></td>
</tr>
<tr>
<td>lvalue -= um</td>
<td>equivalent to <code>lvalue = lvalue - um</code></td>
</tr>
<tr>
<td>lvalue *= um</td>
<td>equivalent to <code>lvalue = lvalue * um</code></td>
</tr>
</tbody>
</table>

Matrix elements can also be set individually. See section A above.

### E. uniform string

Operations producing a uniform string:

<table>
<thead>
<tr>
<th>variable reference</th>
<th>value of uniform string variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant string</td>
<td>string inside double quotes (&quot;string&quot;)</td>
</tr>
<tr>
<td><code>usa[uf]</code></td>
<td>element <code>floor(uf)</code> of array <code>usa</code> where element 0 is the first element. Behavior is undefined if <code>floor(uf0)</code> falls outside the array.</td>
</tr>
<tr>
<td><code>f(...)</code></td>
<td>function call to a function returning uniform string result</td>
</tr>
</tbody>
</table>

Strings can include escape sequences beginning with `\`:

<table>
<thead>
<tr>
<th>character sequence</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\0</td>
<td>octal character code</td>
</tr>
<tr>
<td>\x/#</td>
<td>hex character code</td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\v</td>
<td>vertical tab</td>
</tr>
<tr>
<td>\b</td>
<td>backspace</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
<tr>
<td>\f</td>
<td>form feed</td>
</tr>
<tr>
<td>\a</td>
<td>alert (bell)</td>
</tr>
<tr>
<td>\</td>
<td>backslash character</td>
</tr>
<tr>
<td>?</td>
<td>question mark</td>
</tr>
<tr>
<td>'</td>
<td>single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>embedded double quote</td>
</tr>
</tbody>
</table>

Uniform string assignments take the following forms, where `lvalue` is either a uniform string variable or one element of a uniform string array, accessed by `var[uf]`

| lvalue = us | simple assignment |

### F. uniform relations

Operations producing a uniform relation (used in control statements discussed later):
traditional comparisons: equal, not equal, greater or equal, less or equal, greater and less

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uf_0 == uf_1</td>
<td>true if all elements of uf_0 are equal to the corresponding elements of uf_1</td>
</tr>
<tr>
<td>uf_0 != uf_1</td>
<td>true if any elements of uf_0 does not equal the corresponding element of uf_1</td>
</tr>
<tr>
<td>uf_0 &gt;= uf_1</td>
<td>true if all elements of uf_0 are greater or equal to the corresponding elements of uf_1</td>
</tr>
<tr>
<td>uf_0 &lt;= uf_1</td>
<td>true if all elements of uf_0 are less or equal to the corresponding elements of uf_1</td>
</tr>
<tr>
<td>uf_0 &gt; uf_1</td>
<td>true if any elements of uf_0 does not equal the corresponding element of uf_1</td>
</tr>
<tr>
<td>uf_0 &lt; uf_1</td>
<td>true if any elements of uf_0 does not equal the corresponding element of uf_1</td>
</tr>
<tr>
<td>uc_0 == uc_1</td>
<td>true if all elements of uc_0 are equal to the corresponding elements of uc_1</td>
</tr>
<tr>
<td>uc_0 != uc_1</td>
<td>true if any elements of uc_0 does not equal the corresponding element of uc_1</td>
</tr>
<tr>
<td>um_0 == um_1</td>
<td>true if all elements of um_0 are equal to the corresponding elements of um_1</td>
</tr>
<tr>
<td>um_0 != um_1</td>
<td>true if any elements of um_0 does not equal the corresponding element of um_1</td>
</tr>
<tr>
<td>us_0 == us_1</td>
<td>traditional string comparison: equal and not equal</td>
</tr>
<tr>
<td>us_0 != us_1</td>
<td>traditional string comparison: equal and not equal</td>
</tr>
</tbody>
</table>

It is not possible to save uniform relation results to a variable.

**V. Varying operations**

In the following, uc is as defined above, and vc is a varying color, resulting from one of the operations:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uc</td>
<td>convert uniform color to varying, clamping the resulting color to [0,1]. After this conversion, every pixel has its own copy of the color value.</td>
</tr>
</tbody>
</table>

Possible targets for varying assignments are:

<table>
<thead>
<tr>
<th>Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB</td>
<td>all channels of the framebuffer</td>
</tr>
<tr>
<td>FB.C</td>
<td>set only some channels, leaving the others alone. C is a channel specification, consisting of some combination of the letters r,g,b and a to select the red, green, blue and alpha channels. Each letter can appear at most once, and they must appear in order. This can be used to isolate individual channels: FB.r, FB.g, FB.b, FB.a, or to select arbitrary groups of channels: FB.rgb, FB.rb, FB.ga.</td>
</tr>
</tbody>
</table>

Varying assignments into the framebuffer can take the following forms, where lvalue is FB or FB.C (as described above):

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB = f(...)</td>
<td>function call to a function returning varying color result</td>
</tr>
<tr>
<td>lvalue = vc</td>
<td>copy vc into lvalue</td>
</tr>
<tr>
<td>lvalue += vc</td>
<td>Add, subtract, or multiply lvalue and vc, putting the result in lvalue.</td>
</tr>
<tr>
<td>lvalue -= vc</td>
<td>Add, subtract, or multiply lvalue and vc, putting the result in lvalue.</td>
</tr>
<tr>
<td>lvalue *= vc</td>
<td>Add, subtract, or multiply lvalue and vc, putting the result in lvalue.</td>
</tr>
</tbody>
</table>
Assignments into varying variables can only take this form:

```
variable = FB  
```
copy framebuffer to variable

---

### VI. Built-in functions

The following is a preliminary set of provided functions returning uniform results.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uniform float sin(uniform float radians)</code></td>
<td>Trigonometric sine and cosine functions</td>
</tr>
<tr>
<td><code>uniform float cos(uniform float radians)</code></td>
<td></td>
</tr>
<tr>
<td><code>uniform float mod(uniform float x, uniform float modulus)</code></td>
<td>remainder of division by modulus</td>
</tr>
<tr>
<td><code>uniform matrix inverse(uniform matrix m)</code></td>
<td>matrix inverse</td>
</tr>
<tr>
<td><code>uniform matrix scale(uniform float x, y, z)</code></td>
<td>matrix to rotate by given angle about axis (x, y, z)</td>
</tr>
<tr>
<td><code>uniform matrix translate(uniform float x, y, z)</code></td>
<td></td>
</tr>
<tr>
<td><code>uniform matrix rotate(uniform float x, y, z, radians)</code></td>
<td></td>
</tr>
<tr>
<td><code>uniform matrix perspective(uniform float degree)</code></td>
<td>matrix to perform perspective projection looking down the Z axis with degree field of view</td>
</tr>
</tbody>
</table>

The following is a preliminary set of provided functions returning varying color results.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>varying color texture(uniform string texturename)</code></td>
<td>map texture onto surface, using texture coordinates defined with object geometry.</td>
</tr>
<tr>
<td><code>varying color texture(uniform float texturearray[])</code></td>
<td>Versions with array textures are 1D texturing only (using the s texture coordinate).</td>
</tr>
<tr>
<td><code>varying color texture(uniform color texturearray[])</code></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>varying color texture(uniform string texturename, uniform matrix xform)</td>
<td>same as above, but transform the texture coordinates through xform first.</td>
</tr>
<tr>
<td>varying color texture(uniform float texturearray[], uniform matrix xform)</td>
<td>map texture onto surface as an spherical environment map.</td>
</tr>
<tr>
<td>varying color texture(uniform color texturearray[], uniform matrix xform)</td>
<td>same as above, but transforming the texture coordinates through xform first. Note that this is of questionable utility given the place in the chain where OpenGL applies the transform.</td>
</tr>
<tr>
<td>varying color environment(uniform string texturename)</td>
<td>project texture onto surface using parallel projection down z axis.</td>
</tr>
<tr>
<td>varying color environment(uniform string texturename, uniform matrix xform)</td>
<td>same as above, but transforming the texture coordinates through xform before projection. For example, to project in object space, use inverse(objectmatrix).</td>
</tr>
<tr>
<td>varying color project(uniform string texturename)</td>
<td>Transform the varying color in the framebuffer by the given matrix.</td>
</tr>
<tr>
<td>varying color project(uniform string texturename, uniform matrix xform)</td>
<td>lookup each framebuffer channel in the given lookup table. Each channel is handled independently, so the resulting red component of the result comes from the red component lut[n<em>FB.r]. Similarly, for green from lut[n</em>FB.g] and blue from lut[n*FB.b].</td>
</tr>
<tr>
<td>varying color transform(FB, uniform matrix xform)</td>
<td>channel by channel blend: FB*(1-v) + v = v*(1-FB) + FB</td>
</tr>
<tr>
<td>varying color lookup(FB, uniform float lut[])</td>
<td>alpha-based blend: v*(1-FB.a) + FB*FB.a</td>
</tr>
<tr>
<td>varying color lookup(FB, uniform color lut[])</td>
<td>alpha-based blend: FB*(1-v.a) + v*v.a</td>
</tr>
<tr>
<td>varying color blend(FB, varying color v)</td>
<td>return sum of ambient light hitting surface</td>
</tr>
<tr>
<td>varying color blend(varying color v, FB)</td>
<td>return sum of diffuse light hitting surface</td>
</tr>
<tr>
<td>varying color ablend(FB, varying color v)</td>
<td>return sum of specular light hitting surface, using shininess as the exponent in the Phong lighting model</td>
</tr>
</tbody>
</table>

VII. Variable declarations
A variable declaration is a type name followed by one or more comma-separated variable names. Each variable name may optionally be followed by an initial value. Some examples:

```plaintext
uniform float fvar, gvar;
uniform float farray[3];
uniform float fvar = 3, gvar;
uniform matrix = identity;
uniform string = "mytexture"
varying color cvar;
```

Variable and functions have distinct name spaces, so variables and functions may exist with the same name. The same variable name cannot occur more than once within the same block of statements (bounded by '()' and '), but can be redefined within a nested block:

<table>
<thead>
<tr>
<th>not legal</th>
<th>legal</th>
</tr>
</thead>
</table>
| { uniform float x; uniform float x; } | { uniform float x; {
| |  uniform color x; |
| | } } |

### VIII. Statements

Legal ISL statements are:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment;</td>
<td>performs assignment</td>
</tr>
<tr>
<td>variable declaration;</td>
<td>creates and possibly initializes variable</td>
</tr>
<tr>
<td>{list of 0 or more statements}</td>
<td>executes statements sequentially</td>
</tr>
<tr>
<td>if (ur) statement</td>
<td>execute statement if uniform relation ur is true</td>
</tr>
<tr>
<td>if (ur) statement else statement</td>
<td>execute first statement if ur is true, and second statement if ur is false.</td>
</tr>
<tr>
<td>if (FB) statement</td>
<td>execute statement only for pixels where FB != 0. Any uniform operations in the statement are applied for all pixels.</td>
</tr>
<tr>
<td>if (FB) statement else statement</td>
<td>execute first statement for pixels where FB != 0 and second statement for pixels where FB == 0. Any uniform operations in either statement are applied for all pixels.</td>
</tr>
<tr>
<td>repeat (uf) statement</td>
<td>repeat statement ( \text{max}(0, \text{floor}(uf)) ) times.</td>
</tr>
</tbody>
</table>

### IX. Functions

Every function has this form:

type name(formal_parameters) { body }
The type is one of the ordinary types or a shader type:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambientlight</td>
<td>light contributing to <code>ambient()</code> function.</td>
</tr>
<tr>
<td>distantlight</td>
<td>light shining down the z axis. It is transformed by <code>shadermatrix</code>, which can be used to point in other directions. Contributes to the <code>diffuse()</code> and <code>specular()</code> functions.</td>
</tr>
<tr>
<td>pointlight</td>
<td>light positioned at the origin. It is transformed by <code>shadermatrix</code>, which can be used to position it in the scene. Contributes to <code>diffuse()</code> and <code>specular()</code> functions.</td>
</tr>
<tr>
<td>surface</td>
<td>surface appearance. Should compute the base surface color and lighting contribution (though calls to <code>ambient()</code>, <code>diffuse()</code> and <code>specular()</code>).</td>
</tr>
<tr>
<td>atmosphere</td>
<td>Atmospheric effects like fog.</td>
</tr>
</tbody>
</table>

The set of formal parameter declarations are a semi-colon separated list of uniform variable declarations, with initial values. For shaders, the initial values are interpreted as defaults for any variable not set explicitly by the application.

The body is just a list of statements. The result of each shader is just the value left in `FB` when the shader exits.

The last statement of any functions returning a uniform result should be the special statement `return value;`.

Functions returning a varying color should leave their result in `FB`.

Light shaders should leave the color of light that reaches the surface in `FB`. This color includes things like shadowing, but not the interaction with the surface itself.

Surface shaders should leave the final surface color in `FB`. At the start of the shader, `FB` contains the color of the closest surface previously seen at each pixel. Shaders with transparency should handle any blending with this existing color. In order for surfaces with varying opacity to work, it is also necessary that the application and/or scene graph sort transparent surfaces, and surfaces with varying opacity should be treated as transparent.

Atmosphere shaders start with `FB` set to the final rendered color for each pixel. They should leave the attenuated color in `FB`.

An example shader:

```cpp
surface shadertest(
    uniform color c = color(1,0,0,1);
    uniform float f = .25)
{
    FB = diffuse();
    FB *= c*f;
}
```

X. Files
ISL source files should have the file name extension .isl. Only one shader definition (whether light, surface, or atmosphere) can appear in each .isl file. The .isl file is processed through the C preprocessor, so all cpp directives may be used, as well as both C-style and C++-style comments.

The .isl file itself consists of two sections. All global variable declarations and function definitions must appear first, followed by a single shader function. Only one shader function may appear in any isl file.