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

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:

uniform float `uf`	`uf` is a single floating point value
uniform color `uc`	`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.
uniform matrix `um`	`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, ...)
uniform string `us`	`us` is string, as for a texture name
varying color `vc`	`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

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

uniform float `ufa`**[**`n`**]**	`ufa` is an array with `n` floating point elements, `ufa[0]` through `ufa[n-1]`
uniform color `uca`**[**`n`**]**	`uca` is an array with `n` uniform color elements, `uca[0]` through `uca[n-1]`.
uniform matrix `uma`**[**`n`**]**	`uma` is an array with `n` uniform matrix elements, `uma[0]` through `uma[n-1]`
uniform string `usa`**[**`n`**]**	`usa` is an array with `n` uniform string elements, `usa[0]` through `usa[n-1]`

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:

variable reference	value of uniform float variable

float constant	Where
	`H` = 1 or more hex digits (0-9 or a-f)
	`O` = 1 or more octal digits (0-7)
	`D` = 1 or more decimal digits (0-9)
	`S` = +, - or nothing
	
	One of the following non-case-sensitive patterns:
	`0xH` (hex integer);
	`0O (octal integer);`
	`D; D.; .D; D.D;`
	`DeSD; D.eSD; .DeSD; D.DeSD`

`(uf)`	Grouping intermediate computations
`-uf`	negate `uf`
`uf0 + uf1`	add `uf0` and `uf1`
`uf0 - uf1`	subtract `uf1` from `uf0`
`uf0 * uf1`	multiply `uf0` and `uf1`
`uf0 / uf1`	divide `uf0` by `uf1`
`um[uf0][uf1]`	Gives element `floor(4*uf0 + uf1)` of matrix `um` Behavior is undefined if `floor(4*uf0 + uf1)` is not in the range 0 to 15
`ufa[uf]`	element `floor(uf)` of array `ufa` where element 0 is the first element. Behavior is undefined if `floor(uf0)` falls outside the array.
`f(...)`	function call to a function returning uniform float result

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

`lvalue = uf`	simple assignment
`lvalue += uf`	equivalent to `lvalue = lvalue + uf`
`lvalue -= uf`	equivalent to `lvalue = lvalue - uf`
`lvalue *= uf`	equivalent to `lvalue = lvalue * uf`
`lvalue /= uf`	equivalent to `lvalue = lvalue / uf`

B. uniform color

Operations producing a uniform color:

variable reference	value of uniform color variable
color(uf0,uf1,uf2,uf3**)**	(red=uf0, green=uf1, blue=uf2, alpha=uf3)
uf	color(uf,uf,uf,uf)
(uc**)**	Grouping intermediate computations
-uc uc0 **+** uc1 uc0 **-** uc1 uc0 ***** uc1 uc0 / uc1	Each uniform float operation is applied component-by-component
uca**[**uf**]**	element floor(uf) of array uca, where element 0 is the first element. Behavior is undefined if floor(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:

variable reference	value of uniform matrix variable
matrix(uf0,uf1,uf2,uf3, uf4,uf5,uf6,uf7, uf8,uf9,uf10,uf11, uf12,uf13,uf14,uf15**)**	matrix with rows (uf0,uf1,uf2,uf3), (uf4,uf5,uf6,uf7), (uf8,uf9,uf10,uf11) and (uf12,uf13,uf14,uf15)
(um**)**	Grouping intermediate computations
-um um0 **+** um1 um0 **-** um1	Each uniform float operation is applied component-by-component
um0 * um1	matrix multiplication result[i][k] = $\text{sum}_{j=0..3}$(um0[i][j] * um1[j][k])
uma**[**uf**]**	element floor(uf) of array uma where element 0 is the first element. Behavior is undefined if floor(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]`

`lvalue = um`	simple assignment
`lvalue += um`	equivalent to `lvalue = lvalue + um`
`lvalue -= um`	equivalent to `lvalue = lvalue - um`
`lvalue *= um`	equivalent to `lvalue = lvalue * um`

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

E. uniform string

Operations producing a uniform string:

variable reference	value of uniform string variable
constant string	string inside double quotes (`"string"`)
`usa[uf]`	element `floor(uf)` of array `usa` where element 0 is the first element. Behavior is undefined if `floor(uf0)` falls outside the array.
`f(...)`	function call to a function returning uniform string result

Strings can include escape sequences beginning with '`\`':

character sequence	name
`\`*O*	octal character code
`\x`*H*	hex character code
`\n`	newline
`\t`	tab
`\v`	vertical tab
`\b`	backspace
`\r`	carriage return
`\f`	form feed
`\a`	alert (bell)
`\\`	backslash character
`\?`	question mark
`\'`	single quote
`\"`	embedded double quote

Uniform string assignments take the following forms, where lvalue is either a uniform string variable or one element of an 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):

<table>
<tr><td>uf0 == uf1
uf0 != uf1
uf0 >= uf1
uf0 <= uf1
uf0 > uf1
uf0 < uf1</td><td>traditional comparisons: equal, not equal, greater or equal, less or equal, greater and less</td></tr>
<tr><td>uc0 == uc1</td><td>true if all elements of uc0 are equal to the corresponding elements of uc1</td></tr>
<tr><td>uc0 != uc1</td><td>true if any elements of uc0 does not equal the corresponding element of uc1</td></tr>
<tr><td>um0 == um1</td><td>true if all elements of um0 are equal to the corresponding elements of um1</td></tr>
<tr><td>um0 != um1</td><td>true if any elements of um0 does not equal the corresponding element of um1</td></tr>
<tr><td>us0 == us1
us0 != us1</td><td>traditional string comparison: equal and not equal</td></tr>
<tr><td>(ur)</td><td>Grouping intermediate computations</td></tr>
<tr><td>ur0 && ur1</td><td>true if both ur0 and ur1 are true</td></tr>
<tr><td>ur0 || ur1</td><td>true if either ur0 or ur1 are true</td></tr>
<tr><td>!ur</td><td>true if ur is not true</td></tr>
</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:

variable reference	value of varying color variable
`uc`	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.

Possible targets for varying assignments are:

`FB`	all channels of the framebuffer
`FB.`*`C`*	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`.

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

`FB = f(...)`	function call to a function returning varying color result
`lvalue = vc`	copy `vc` into `lvalue`

`lvalue += vc`
`lvalue -= vc`
`lvalue *= vc`
	Add, subtract, or multiply `lvalue` and `vc`, putting the result in `lvalue`.

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.

`uniform float sin(uniform float radians)`
 	
`uniform float cos(uniform float radians)`
	Trigonometric sine and cosine functions

`uniform float mod(uniform float x, uniform float modulus)`	remainder of division by modulus `x - modulus*floor(x/modulus)`
`uniform matrix inverse(uniform matrix m)`	matrix inverse `inverse(m)*m = identity`
`uniform matrix scale(uniform float x, y, z)`	`matrix(x,0,0,0, 0,y,0,0, 0,0,z,0, 0,0,0,1)`
`uniform matrix translate(uniform float x, y, z)`	`matrix(1,0,0,0, 0,1,0,0, 0,0,1,0, x,y,z,1)`
`uniform matrix rotate(uniform float x, y, z, radians)`	matrix to rotate by given angle about axis `(x,y,z,0)`

`uniform matrix perspective(uniform float degree)`	matrix to perform perspective projection looking down the Z axis with `degree` field of view
	`matrix(cotan(degree/2),0, 0, 0,`
	`0, cotan(degree/2),0, 0,`
	`0, 0, 1, 1,`
	`0, 0, -2,0)`

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

`varying color texture(uniform string texturename)`
 	
`varying color texture(uniform float texturearray[])`
 	
`varying color texture(uniform color texturearray[])`
	map texture onto surface, using texture coordinates defined with object geometry. Versions with array textures are 1D texturing only (using the `s` texture coordinate).

`varying color texture(uniform string texturename, uniform matrix xform)`
 	
`varying color texture(uniform float texturearray[], uniform matrix xform)`
 	
`varying color texture(uniform color texturearray[], uniform matrix xform)`
	same as above, but transform the texture coordinates through `xform` first.

`varying color environment(uniform string texturename)`	map texture onto surface as an spherical environment map.
`varying color environment(uniform string texturename, uniform matrix xform)`	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.
`varying color project(uniform string texturename)`	project texture onto surface using parallel projection down z axis
`varying color project(uniform string texturename, uniform matrix xform)`	same as above, but transform by `xform` before projection. For example, to project in object space, use inverse(objectmatrix).
`varying color transform(FB, uniform matrix xform)`	Transform the varying color in the framebuffer by the given matrix

`varying color lookup(FB, uniform float lut[])`
 	
`varying color lookup(FB, uniform color lut[])`
	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*FB.r]`. Similarly, for green from `lut[n*FB.g]` and blue from `lut[n*FB.b]`

`varying color blend(FB, varying color v)` `varying color blend(varying color v,FB)`	channel by channel blend: `FB*(1-v) + v = v*(1-FB) + FB`
`varying color ablend(FB, varying color v)`	alpha-based blend: `v*(1-FB.a) + FB*FB.a`
`varying color ablend(varying color v, FB)`	alpha-based blend: `FB*(1-v.a) + v*v.a`
`varying color ambient()`	return sum of ambient light hitting surface
`varying color diffuse()`	return sum of diffuse light hitting surface
`varying color specular(uniform float shininess)`	return sum of specular light hitting surface, using shininess as the exponent in the Phong lighting model

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:

```
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>
<tr><th>not legal</th><th>legal</th></tr>
<tr><td><pre>{
    uniform float x;
    uniform float x;
}</pre></td><td><pre>{
    uniform float x;
    {
        uniform color x;
    }
}</pre></td></tr>
</table>

VIII. Statements

Legal ISL statements are:

assignment`;`	performs assignment
variable declaration`;`	creates and possibly initializes variable
`{`list of 0 or more statements**`}`**	executes statements sequentially
`if (``ur`**`)`** `statement`	execute statement if uniform relation `ur` is true
`if (``ur`**`)`** `statement` **`else`** `statement`	execute first statement if `ur` is true, and second statement if `ur` is false.
`if (FB)` `statement`	execute statement only for pixels where `FB != 0`. Any uniform operations in the statement are applied for all pixels.
`if (FB)` `statement` **`else`** `statement`	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.
`repeat (``uf`**`)`** `statement`	repeat statment `max(0,floor(uf))` times.

IX. Functions

Every function has this form:
type name(formal_parameters) { body }

The type is one of the ordinary types or a shader type:

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

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:

```
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.