Interactive Shading Language (ISL) Language Description April 12, 2001

Contents

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 two function types. A *light shader* computes the color and intensity for a light hitting the surface. Light shaders can be used for ambient, distant and local lights. 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. The term *shader* is used to refer to either of these 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 point on the surface, 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.

ISL is also different from most other shading languages in that more than one surface shader may be applied to each surface. The shaders are applied in turn and may composite or blend their results. ISL no longer supports explicit atmosphere shaders. Any light transmission effects between the surface and eye can be handled in the final shader applied to each surface.

II. Files

The appearance of a shaded surface is defined by one or more ISL surface shaders and possibly one or more ISL light shaders. Each shader is defined in its own ISL source files, which should have the file name extension .isl.

A. File contents

Only one shader definition (whether light or surface) can appear in each .isl file. The .isl file may include C preprocessor-like *#include* directives to get access to functions or global variable definitions stored in another file.

Comments in isl may be either C or C++-style (*/*comment*/* or *// comment to end of line*)

B. File compilation

There are two ways to compile a set of ISL files into the rendering passes used to compute surface appearance. The first is to use the command line compiler and translator. The second is to use the ISL run-time library. Both are documented in the shader(1) man page. The ISL compiler, islc, converts a set of ISL files into a pass description (.ipf) file. Information on running islc can be found on the islc(1) man page. The pass description file can be converted either to C OpenGL code with the command line translator ipf2ogl (see the ipf2ogl(1) man page), or to a Performer pass file with the command line translator ipf2pf (shipped with Performer 2.4 or later). The ISL Library consists of a set of C++ classes that enable an application to compile that appearance consisting of ISL shaders into an OpenGL stream. The compiled appearance can be associated with geometry from the application, and rendered to an OpenGL rendering context opened by the application.

III. Data types

All ISL data is classified as either *varying*, *parameter* or *uniform*. Varying data may hold a different value at each pixel. Parameter data must have the same value at every pixel on a surface, but can differ from surface to surface or from frame to frame. Changes to varying or parameter data do not require recompiling the shader. Uniform data also has the same value at every pixel on the surface, but changes to uniform data only take effect when the shader is recompiled.

The complete list of ISL data types is:

uniform float *uf*	*uf* and *pf* are each a single floating point value
parameter float *pf*	
uniform color *uc*	*uc* and *pc* are each a set of four floating point values, representing a color, vector or point. For colors, the components are ordered red, green, blue and alpha. For points, the components are ordered x,y,z and w.
parameter color *pc*	
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
uniform matrix *um*	*um* and *pm* are each 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, ...)
parameter matrix *pm*	
uniform string *us*	*us* is a character string, used for texture names.

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

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

IV. Variables and identifiers

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

Several variables are predefined with special meaning:

varying color FB	Current frame buffer contents. This is the intermediate result location for almost all varying operations.
parameter matrix shadermatrix	Arbitrary matrix associated with the shader at compile time. This may be used to control the coordinate space where the shader operates.
uniform float pi	The math constant.

V. 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 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	One of the following non-case-sensitive patterns: `0x`*`H`* (hex integer); `0`*`O`* (octal integer); *`D`*; *`D`*`.`; `.`*`D`*; *`D`*`.`*`D`*; *`D`*`e`*`SD`*; *`D`*`.e`*`SD`*; `.`*`D`*`e`*`SD`*; *`D`*`.`*`D`*`e`*`SD`* 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
(*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*
uc[*uf0*]	Gives channel *floor(uf0)* of color *uc*, where red is channel 0, green is channel 1, blue is channel 2 and alpha is channel 3.
um[*uf0*][*uf1*]	Gives element *floor(4*uf0 + uf1)* of matrix *um*
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 or a floating point element from a variable (*var[uf0]* for a uniform color or a uniform float array, *var[uf0][uf1]* for a uniform matrix or uniform color array or *var[uf0][uf1][uf2]* for a uniform matrix array):

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
um[*uf*]	Row *floor(uf)* of matrix *um*
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 a color element from a variable (*var[uf0]* for an element of a color array or row of a uniform matrix or *var[uf0][uf1]* for a uniform matrix array):

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*

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

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)*
uf	*matrix(uf,0,0,0, 0,uf,0,0, 0,0,uf,0, 0,0,0,uf)*
(*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] = 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 variable, 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 sections A and B 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 variable, accessed by *var[uf]*:

lvalue = *us*	Simple assignment

F. uniform relations

Operations producing a uniform relation (used in control statements discussed later):

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

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

VI. Parameter operations

In the following, *pf* and *pf0*-*pf15* are parameter floats; *pfa* is an array of parameter floats; *pc*, *pc0* and *pc1* are parameter colors; *pca* is an array of parameter colors; *pm*, *pm0* and *pm1* are parameter matrices; and *pma* is an array of parameter matrices. Also, *uf0* and *uf1* are uniform floats and *uc* is a uniform color as defined above.

A. parameter float

Operations producing a parameter float:

variable reference	Value of parameter float variable.
uf	Convert uniform float to parameter float.
(*pf*)	Grouping intermediate computations.
-*pf*	Negate *pf*
pf0 + *pf1*	Add *pf0* and *pf1*
pf0 - *pf1*	Subtract *pf1* from *pf0*
pf0 * *pf1*	Multiply *pf0* and *pf1*
pf0 / *pf1*	Divide *pf0* by *pf1*
pc[*pf0*]	Gives channel *floor(pf0)* of color *pc*, where red is channel 0, green is channel 1, blue is channel 2 and alpha is channel 3.
pm[*pf0*][*pf1*]	Gives element *floor(4*pf0 + pf1)* of matrix *pm*
pfa[*uf*]	Element *floor(uf)* of array *pfa* where element 0 is the first element. Note that currently the array index must be uniform. Behavior is undefined if *floor(uf0)* falls outside the array.
f(...)	Function call to a function returning parameter float result

Parameter float assignments take the following forms, where *lvalue* is either a parameter float variable or a floating point element from a variable (*var[uf0]* for a parameter float array):

lvalue = *pf*	Simple assignment
lvalue += *pf*	Equivalent to *lvalue* = *lvalue* + *pf*
lvalue -= *pf*	Equivalent to *lvalue* = *lvalue* - *pf*
lvalue *= *pf*	Equivalent to *lvalue* = *lvalue* * *pf*
lvalue /= *pf*	Equivalent to *lvalue* = *lvalue / pf*

B. parameter color

Operations producing a parameter color:

variable reference	Value of parameter color variable
uc	Convert uniform color to parameter color.
color(*pf0*,*pf1*,*pf2*,*pf3*)	*red=pf0*; *green=pf1*; *blue=pf2*; *alpha=pf3*
pf	*color(pf,pf,pf,pf)*
(*pc*)	Grouping intermediate computations
-*pc* *pc0* + *pc1* *pc0* - *pc1* *pc0* * *pc1* *pc0* / *pc1*	Each parameter float operation is applied component-by-component
pm[*pf*]	Row *floor(pf)* of matrix *pm*
pca[*uf*]	Element *floor(uf)* of array *pca*, where element 0 is the first element. Note that currently the array index must be uniform. Behavior is undefined if *floor(uf0)* falls outside the array.
f(...)	Function call to a function returning parameter color result

Parameter color assignments take the following forms, where *lvalue* is either a parameter color variable or a color element from a variable (*var[uf0]* for an element of a color array):

lvalue = *pc*	Simple assignment
lvalue += *pc*	Equivalent to *lvalue = lvalue + pc*
lvalue -= *pc*	Equivalent to *lvalue = lvalue - pc*
lvalue *= *pc*	Equivalent to *lvalue = lvalue * pc*
lvalue /= *pc*	Equivalent to *lvalue = lvalue / pc*

Unlike uniform colors, parameter colors cannot currently be set by element.

C. parameter matrix

Operations producing a parameter matrix:

variable reference	Value of parameter matrix variable
um	Convert uniform matrix to parameter matrix.
matrix(*pf0,pf1,pf2,pf3, pf4,pf5,pf6,pf7, pf8,pf9,pf10,pf11, pf12,pf13,pf14,pf15*)	Matrix with rows *(pf0,pf1,pf2,pf3)*, *(pf4,pf5,pf6,pf7)*, *(pf8,pf9,pf10,pf11)* and *(pf12,pf13,pf14,pf15)*
pf	*matrix(pf,0,0,0, 0,pf,0,0, 0,0,pf,0, 0,0,0,pf)*
(*pm*)	Grouping intermediate computations
-*pm* *pm0* + *pm1* *pm0* - *pm1*	Each parameter float operation is applied component-by-component
pm0 * *pm1*	Matrix multiplication: $result[i][k] = sum_{j=0..3}(um0[i][j] * um1[j][k])$
pma[*uf*]	Element *floor(uf)* of array *pma* where element 0 is the first element. Note that currently the array index must be uniform. Behavior is undefined if *floor(uf0)* falls outside the array.
f(...)	Function call to a function returning parameter matrix result

Parameter matrix assignments take the following forms, where lvalue is either a parameter matrix variable or one element of a parameter matrix array variable, accessed as *var[uf]*:

lvalue = *pm*	Simple assignment
lvalue += *pm*	Equivalent to *lvalue* = *lvalue* + *pm*
lvalue -= *pm*	Equivalent to *lvalue* = *lvalue* - *pm*
lvalue *= *pm*	Equivalent to *lvalue* = *lvalue* * *pm*

Unlike uniform matrices, parameter matrices cannot currently be set by element.

VII. Varying operations

In the following, *vc* is a varying color. Also, *pf0* and *pf1* are parameter floats and *pc* is a parameter color as defined above.

A. varying color

Operations producing a varying color:

variable reference	Value of varying color variable Note: when a varying variable is used, *texgen* value of -3 is passed to the application geometry drawing function (see the description under *texture()*). While the geometry drawing function may choose to act on this value, OpenGL Shader will set the texture generation mode appropriately.
pc	Convert parameter 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
	
	All varying functions also implicitly have access to the value of FB when the function is called.
	
	Except for certain built-in functions explicitly noted later, varying functions can **only** be assigned directly into all channels of the framebuffer. To combine the results of a varying function with the existing frame buffer contents, you must save the existing frame buffer into a variable. For example:
	
	NO FB.r = f();
	OK varying color a = FB;
	FB = f();
	FB.bga = a;

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

B. varying relations

Operations producing a varying relation (used in control statements discussed later):

FB[*vf0*] == *vf1* FB[*vf0*] != *vf1* FB[*vf0*] >= *vf1* FB[*vf0*] <= *vf1* FB[*vf0*] > *vf1* FB[*vf0*] < *vf1*	Traditional comparisons: equal, not equal, greater or equal, less or equal, greater, and less Performs per-pixel comparison between frame buffer channel *uf0* and reference value *uf1*. Frame buffer channel 0 is red, channel 1 is green, channel 2 is blue and channel 3 is alpha.

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

VIII. Built-in functions

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

uniform float abs(uniform float *x*) parameter float abs(parameter float *x*)	absolute value of x
uniform float acos(uniform float *x*) parameter float acos(parameter float *x*)	inverse cosine, radian result is between 0 and pi
uniform float asin(uniform float *y*) parameter float asin(parameter float *y*)	inverse sine, radian result is between -pi/2 and pi/2
uniform float atan(uniform float *f*) parameter float atan(parameter float *f*)	inverse tangent, radian result is between -pi/2 and pi/2
uniform float atan(uniform float *y*; uniform float *x*) parameter float atan(parameter float *y*; parameter float *x*)	inverse tangent of y/x, radian result is between -pi and pi
uniform float ceil(uniform float *x*) parameter float ceil(parameter float *x*)	round x up (smallest integer i >= x)
uniform float clamp(uniform float *x*; uniform float *a*; uniform float *b*) parameter float clamp(parameter float *x*; parameter float *a*; parameter float *b*)	clamp x to lie between a and b

uniform float cos(uniform float *r*) parameter float cos(parameter float *r*)	cosine of *r* radians
uniform float exp(uniform float *x*) parameter float exp(parameter float *x*)	e^x
uniform float floor(uniform float *x*) parameter float floor(parameter float *x*)	round x down (largest integer i <= x)
uniform matrix inverse(uniform matrix *m*) parameter matrix inverse(parameter matrix *m*)	matrix inverse *m*inverse(m) = inverse(m)*m* = identity matrix
uniform float log(uniform float *x*) parameter float log(parameter float *x*)	natural log of x
uniform float max(uniform float *x*; uniform float *y*) parameter float max(parameter float *x*; parameter float *y*)	maximum of x and y
uniform float min(uniform float *f*; uniform float *g*) parameter float min(parameter float *f*; parameter float *g*)	minimum of x and y
uniform float mod(uniform float *n*; uniform float *d*) parameter float mod(parameter float *n*; parameter float *d*)	Remainder of division *n/d* *n - d*floor(n/d)*
uniform matrix perspective(uniform float *d*) parameter matrix perspective(parameter float *d*)	matrix to perform perspective projection looking down the Z axis with a field of view of *d* degrees. *matrix(cotan(d/2),0, 0, 0,* *0, cotan(d/2),0, 0,* *0, 0, 1, 1,* *0, 0, -2,0)*
uniform float pow(uniform float *x*; uniform float *y*) parameter float pow(parameter float *x*; parameter float *y*)	x^y

uniform matrix rotate(uniform float *x*; uniform float *y*; uniform float *z*; uniform float *r*) parameter matrix rotate(parameter float *x*; parameter float *y*; parameter float *z*; parameter float *r*)	rotate *r* radians around axis *(x,y,z)*
uniform float round(uniform float *x*) parameter float round(parameter float *x*)	round x to the nearest integer
uniform matrix scale(uniform float *x*; uniform float *y*; uniform float *z*) parameter matrix scale(parameter float *x*; parameter float *y*; parameter float *z*)	*matrix(x,0,0,0, 0,y,0,0, 0,0,z,0, 0,0,0,1)*
uniform float sign(uniform float *x*) parameter float sign(parameter float *x*)	sign of x: -1, 0 or 1
uniform float sin(uniform float *r*) parameter float sin(parameter float *r*)	sine of *r* radians
uniform float smoothstep(uniform float *a*; uniform float *b*; uniform float *x*) parameter float smoothstep(parameter float *a*; parameter float *b*; parameter float *x*)	smooth transition between 0 and 1 as *x* changes from *a* to *b*. 0 for $x < a$, 1 for $x > b$
uniform color spline(uniform float *x*; uniform color *c*[]) uniform float spline(uniform float *x*; uniform float *c*[]) parameter color spline(parameter float *x*; parameter color *c*[]) parameter float spline(parameter float *x*; parameter float *c*[])	evaluate Catmull-Rom spline at *x* based on control point vector, *c*. A Catmull-Rom spline passes through all of the control points. The derivative of the curve at each control point is half the difference between the next and previous control points. The full curve is covered between *x=0* and *x=1*
uniform float sqrt(uniform float *x*) parameter float sqrt(parameter float *x*)	square root of *x*

uniform float step(uniform float *a*; uniform float *x*) parameter float step(parameter float *a*; parameter float *x*)	0 for *x<a* 1 for *x>=a*
uniform float tan(uniform float *r*) parameter float tan(parameter float *r*)	tangent of *r* radians
uniform matrix translate(uniform float *x*; uniform float *y*; uniform float *z*) parameter matrix translate(parameter float *x*; parameter float *y*; parameter float *z*)	*matrix(1,0,0,0, 0,1,0,0, 0,0,1,0, x,y,z,1)*

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

varying color texture(uniform string *texturename[*; parameter matrix *xform[*; uniform float *texgen]]*) varying color texture(uniform float *texturearray*[]*[*; parameter matrix *xform[*; uniform float *texgen]]*) varying color texture(uniform color *texturearray*[]*[*; parameter matrix *xform[*; uniform float *texgen]]*)	Map texture onto surface, using texture coordinates defined with object geometry. Versions with array textures are 1D texturing only (using the *s* texture coordinate). Optional float *texgen* (>= 0) is passed to the geometry drawing function so it can generate a different (application defined) set of per-vertex texture coordinates. If *texgen* is not given, a value of 0 will be passed to the geometry drawing function. Optional matrix *xform* is a matrix for transforming the texture coordinates. If *xform* is not given, the identity matrix is used (i.e. texture coordinates are used as given). Note: negative *texgen* values are used for built-in texture generation modes. These negative values are also passed to the geometry drawing function. While the geometry drawing function may choose to act on these value, OpenGL Shader will set the texture generation mode appropriately.

texture use	**texgen code**
texture()	>= 0
project()	-1
environment()	-2
varying variable use	-3

varying color environment(uniform string _texturename[_; parameter matrix _xform]_) varying color environment(uniform float _texturearray_[]_[_; parameter matrix _xform]_) varying color environment(uniform color _texturearray_[]_[_; parameter matrix _xform]_)	Map texture onto surface, as a spherical environment map. Versions with array textures are 1D texturing only (using the _s_ texture coordinate). Optional matrix _xform_ is a matrix for transforming the texture coordinates. For example, it can be used to set the map _up_ direction. If _xform_ is not given, the identity matrix is used (i.e. texture coordinates are used as generated). Note: _environment_ also passes a _texgen_ value of -2 to the application geometry drawing function.
varying color project(uniform string _texturename[_; parameter matrix _xform]_) varying color project(uniform float _texturearray_[]_[_; parameter matrix _xform]_) varying color project(uniform color _texturearray_[]_[_; parameter matrix _xform]_)	Project texture onto surface using parallel projection down the Z axis. Versions with array textures are 1D texturing only (using the X coordinate only). Optional matrix _xform_ is a matrix for transforming before projection. For example, to project in shader space, use _inverse(shadermatrix)_. If _xform_ is not given, the identity matrix is used. Note: _project()_ also passes a _texgen_ value of -1 to the application geometry drawing function.
varying color transform(parameter matrix _xform_)	Transform the varying color in the frame buffer by the given matrix
varying color lookup(parameter float _lut_[]) varying color lookup(parameter color _lut_[])	Lookup each frame buffer 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(varying color _v_)	Channel by channel blend: _FB*(1-v) + v = v*(1-FB) + FB_
varying color over(varying color _v_)	Alpha-based blend of _FB_ over _v_: _v*(1-FB.a) + FB*FB.a_
varying color under(varying color _v_)	Alpha-based blend of _FB_ under _v_: _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(parameter float _e_)	Return sum of specular light hitting surface, using _e_ as the exponent in the Phong lighting model

IX. Variable declarations

A variable declaration is a type name followed by one (and only one) variable name. Each variable name may optionally be followed by an initial value. Some examples:

```
uniform float fvar;
uniform float farray[3];
uniform float fvar = 3;
parameter matrix = 1;
uniform string = "mytexture"
varying color cvar;
```

Variable and functions names are bound using static scoping rules similar to *C*. The same name cannot occur more than once within the same block of statements (bounded by ’*{*’ and ’*}*’), but can be redefined within a nested block:

not legal	**legal**
`{`	`{`
`    uniform float x;`	`    uniform float x;`
`    uniform float x;`	`    {`
`}`	`        uniform color x;`
	`    }`
	`}`

X. Statements

In the following, *uf* is a uniform float, *ur* is a uniform relation and *vr* is a varying relation as defined above.

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 (*vr*) *statement*	Restricts the currently active set of pixels to those where the given varying relation is true. The active set of pixels starts as all visible pixels within the shaded object, but may be restricted by one or more *if* statements. Frame buffer operations in *statement* only operate on the active subset of pixels. Any uniform operations or varying variable assignments are still applied for all pixels.
if (*vr*) *statement* else *statement*	The first statement executes with the same restricted set of pixels as the previous *if* statement. The second statement executes with the active pixels restricted to those that were active when the *if* statement was reached but where the varying relation was false. In both statements, the active set of pixels only restricts frame buffer operations. Uniform operations and assignment to varying variables are not affected by the set of active pixels.
repeat (*uf*) statement	repeat statment *max(0,floor(uf))* times.

XI. Functions

Every function has this form:
type function_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	Equivalent to surface. Atmospheric effects like fog are handled in the last surface shader in the shader list.

The set of formal parameter declarations are a semi-colon separated list of uniform variable declarations, with initial values. *Initial values are required for all formal parameters*. For shaders, the initial values are interpreted as defaults for any variable not set explicitly by the application. Arrays in the formal parameter list for a shader are not currently visible to the application. The initial values for parameters of ordinary functions are not currently used, but they are still required.

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 function should be the special statement
`return` *value;*.

The *return* statement can only appear as the last statement in a function, and the type of *value* should match the function type. For functions returning a varying color, the *return* is optional. If *return* is omitted on a varying color function, the function return value is the value of *FB* at the end of the function.

Light shaders return a varying color giving the light that reaches the surface. This color may include effects like shadowing, but not the interaction with the surface itself.

Surface shaders return a varying color giving the final color of the surface. 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 return the attenuated color.

An example shader:

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