OpenGL Shader ISL Library Reference Page Index

Related User Documents

Shader SDK(1)	OpenGL Shader Software Development Kit
Interactive Shading Language	The language specification
ipf2ogl(1)	OpenGL Shader Interactive Shading Language translator
islc(1)	OpenGL Shader Interactive Shading Language compiler

ISL Reference Pages

islAppearance	OpenGL Shader standard appearance class
islAppearanceBase	OpenGL Shader base appearance class
islAppearanceCopy	OpenGL Shader appearance copy class
islAppearanceCopyData	OpenGL Shader copy appearance data
islAppearanceSnapshot	OpenGL Shader 'snapshot' appearance class
islAppearanceSnapshotData	OpenGL Shader 'snapshot' appearance data
islCompileAction	OpenGL Shader compiler class
islCopyAction	OpenGL Shader appearance copy action class
islDrawAction	OpenGL Shader rendering class
islError	OpenGL Shader error class
islMemory	OpenGL Shader memory manager class
islShader	OpenGL Shader Interactive Shading Language shader class
islShape	OpenGL Shader Interactive Shading Language shape class
islSnapshotAction	OpenGL Shader appearance snapshot class
isl::TexGen::copyNormToTex	OpenGL Shader TexGen Function: isl::TexGen::copyNormToTex
isl::TexGen::copyPosToTex	OpenGL Shader TexGen Function: isl::TexGen::copyPosToTex
isl::TexGen::tangentSpaceAxis	OpenGL Shader TexGen Function: isl::TexGen::tangentSpaceAxis
isl::Texture::ClearCoat360	OpenGL Shader ClearCoat360 Texture
isl::Texture::Fresnel	OpenGL Shader Fresnel Texture
isl::Texture::Image	OpenGL Shader Texture Generation Base class
isl::Texture::Noise	OpenGL Shader Noise Texture
isl::VertexContext	OpenGL Shader Vertex Shader Context class
isl::VertexShader	OpenGL Shader Vertex Shading class

Shader SDK(1)

NAME

Shader SDK - OpenGL Shader Software Development Kit

DESCRIPTION

The OpenGL Shader Software Development Kit is a suite of tools for supporting interactive, programmable shading on OpenGL systems. It consists of command line compilers and translators that can convert a set of Interactive Shading Language (ISL) shaders into an OpenGL function call, as well as an Interactive Shading Language Library that enables applications to access the compilers in an interactive system.

COMMAND LINE COMPILER

The command line compiler *islc*(1) translates an appearance description into a description of OpenGL passes. When converted to an OpenGL stream with a translator such as *ipf2ogl*(1), this intermediate pass description will render an object with the specified appearance. An appearance is defined as one or more of: a list of surface shaders, a list of ambient light shaders, and a list of direct light shaders. The shaders are written in the OpenGL Interactive Shading Language.

COMMAND LINE TRANSLATOR

The command line translator *ipf2ogl*(1) translates a description of OpenGL passes, as output by *islc*(1), into C code which implements the OpenGL passes described in the input. For a given intermediate pass file, one **.c** file and one **.h** file are generated by *ipf2ogl*(1). The **.c** file contains the definitions of the initialization, drawing and cleanup functions for the shader, while the **.h** file contains the prototypes for these functions.

ISL LIBRARY

The OpenGL Shader Interactive Shading Language Library provides a minimal interface for supporting interactive, programmable shading. The ISL Library consists of six classes that enable an application to define an appearance consisting of ISL shaders, compile that appearance into an OpenGL stream, associate the compiled appearance with geometry from the application, and, subsequently, to render the shaded geometry to an OpenGL rendering context opened by the application.

DOCUMENTATION

Documentation may be found in /usr/share/shader/doc. Documentation found here includes html man pages for the command line compiler, translator, and ISL Library as well as the ISL Specification.

EXAMPLE SOURCE CODE

Example source code may be found in /usr/share/shader/src. It includes examples for creating applications based on output from the command line compiler and translator, a stand-alone application based on the ISL Library, and an Inventor-based application using the ISL Library.

FILES

/usr/bin/islc

location of command-line ISL compiler

/usr/bin/ipf2ogl

location of OpenGL translator

/usr/lib32/libisl.so

ISL Library

/usr/lib32/debug/libisl.so

Debug ISL Library

/usr/share/shader/src/*

sample code and documentation

/usr/share/shader/doc/*

ISL Specification and html format man pages

SEE ALSO

islc(1), *ipf2ogl*(1), *islShader*(3), *islAppearance*(3), *islShape*(3), *islCompileAction*(3), *islDrawAction*(3), and *islError*(3),

Interactive Shading Language (ISL) Language Description Version 3.0 August 8, 2002

Contents

I. Introduction
II. Files
III. Data types
IV. Variables and identifiers
V. Uniform operations
VI. Parameter operations
VII. Varying operations
VIII. Built-in functions
IX. Variable declarations
X. Statements
XI. Functions
XII. Level of Detail

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 area of active debate. Over the next several releases of OpenGL Shader, we plan to extend ISL to more closely resemble the evolving OpenGL 2.0 language.

A. Features in common with off-line 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:

- Unlike most other interactive shading languages, the types of shading functions you write in ISL are based on the logical process of defining a surface appearance rather than the convenience of mapping to hardware. Describing what you want is your job, figuring out how to map it onto the hardware is our job. This is why we have *light* and *surface* shaders rather than *vertex* and *fragment* shaders.
- 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 ISL run-time library. The second is to use the command line compiler and translator. Both are documented in the shader(1) man page. 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. 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).

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.
parameter color lightVector	Within a light shader, the direction the light is shining. This vector may be modified by the light shader. Within a surface shader, the direction of the most recent light.
uniform float pi	The math constant.
uniform float numambientlights	Number of ambient lights in the current islAppearance.
uniform float numdirectlights	Number of direct lights (= both local and distant lights) in the current islAppearance.

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] = $\text{sum}_{j=0..3}$(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 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
\xH	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] = $\text{sum}_{j=0..3}$(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.

D. parameter relations

Operations producing a parameter relation closely parallel the uniform relations covered earlier. They can be used in control statements discussed later:

<table>
<tr><td>pf0 == pf1

pf0 != pf1

pf0 >= pf1

pf0 <= pf1

pf0 > pf1

pf0 < pf1</td><td>Traditional comparisons: equal, not equal, greater or equal, less or equal, greater, and less</td></tr>
<tr><td>pc0 == pc1</td><td>True if all elements of pc0 are equal to the corresponding elements of pc1</td></tr>
<tr><td>pc0 != pc1</td><td>true if any elements of pc0 does not equal the corresponding element of pc1</td></tr>
<tr><td>pm0 == pm1</td><td>True if all elements of pm0 are equal to the corresponding elements of pm1</td></tr>
<tr><td>pm0 != pm1</td><td>True if any elements of pm0 does not equal the corresponding element of pm1</td></tr>
<tr><td>(pr)</td><td>Grouping intermediate computations</td></tr>
<tr><td>pr0 && pr1</td><td>True if both pr0 and pr1 are true</td></tr>
<tr><td>pr0 || pr1</td><td>True if either pr0 or pr1 are true</td></tr>
<tr><td>!pr</td><td>True if pr is false</td></tr>
</table>

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

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):

<table>
<tr><td>FB = f(...)</td><td>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:
<table><tr><th>NO</th><th>OK</th></tr><tr><td>FB.r = f();</td><td>varying color a = FB;
FB = f();
FB.bga = a;</td></tr></table></td></tr>
<tr><td>lvalue = vc</td><td>Copy vc into lvalue</td></tr>
<tr><td>lvalue += vc

lvalue -= vc

lvalue *= vc</td><td>Add, subtract, or multiply lvalue and vc, putting the result in lvalue.</td></tr>
</table>

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 the 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 the set of provided functions returning varying color results.

<table>
<tr><td>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]])</td>
<td>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.

<table>
<tr><th>texture use</th><th>texgen code</th></tr>
<tr><td>texture()</td><td>>= 0</td></tr>
<tr><td>project()</td><td>-1</td></tr>
<tr><td>environment()</td><td>-2</td></tr>
<tr><td>varying variable use</td><td>-3</td></tr>
</table></td></tr>
<tr><td>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])</td>
<td>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.</td></tr>
<tr><td>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])</td>
<td>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.</td></tr>
<tr><td>varying color transform(parameter matrix xform)</td><td>Transform the varying color in the frame buffer by the given matrix</td></tr>
<tr><td>varying color lookup(parameter float lut[])

varying color lookup(parameter color lut[])</td><td>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]</td></tr>
<tr><td>varying color blend(varying color v)</td><td>Channel by channel blend: FB*(1-v) + v = v*(1-FB) + FB</td></tr>
<tr><td>varying color over(varying color v)</td><td>Alpha-based blend of FB over v:
v*(1-FB.a) + FB*FB.a</td></tr>
</table>

varying color under(varying color v)	Alpha-based blend of FB under v: FB*(1-v.a) + v*v.a
varying color setupLight(parameter float lightnum)	Configure a specific light for subsequent diffuse or specular calculations. After being called, the global lightVector is set with the current light's position. Light shaders can modify lightVector within their body
varying color ambient()	Return sum of ambient light hitting surface
varying color ambient(uniform float lightnum)	Return result of ambient light lightnum If lightnum<0 or lightnum>=numambientlights, ambient() returns black
varying color diffuse()	Return sum of diffuse light hitting surface
varying color diffuse(uniform float lightnum)	Return result of diffuse contribution from light lightnum If lightnum<0 or lightnum>=numdirectlights, diffuse() returns black diffuse(lightnum) is equivalent to setupLight(lightnum); runDiffuse(lightVector);
varying color runDiffuse(parameter color lvector)	Calculate diffuse effects of previously configured light (configured by using *setupLight*). Accepts a parameter lvector to specifiy the light position. Use the global lightVector to accept the value set by previous code or the *setupLight* routine.
varying color specular(parameter float e)	Return sum of specular light hitting surface, using e as the exponent in the Phong lighting model
varying color specular(uniform float lightnum, parameter float e)	Return result of specular contribution from light lightnum If lightnum<0 or lightnum>=numdirectlights, specular() returns black specular(lightnum, e) is equivalent to setupLight(lightnum); runSpecular(e,lightVector);
varying color runSpecular(parameter float e; parameter color lvector)	Calculate specular effects of previously configured light (configured by using *setupLight*). Accepts the parameter e as the exponent in the Phong lighting model.Accepts a parameter lvector to specifiy the light position. Use the global lightVector to accept the value set by previous code or the *setupLight* routine.

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.

<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>

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*
 	
if (pr) *statement*
	Execute statement only if uniform relation ur or parameter relation pr is true

if (ur) *statement* else *statement*
 	
if (pr) *statement* else *statement*
	Execute first statement if ur or pr is true, and second statement if ur or pr 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. Note: Any variable of any type assigned inside a varying if should only be used inside the if. The contents outside the if are undefined, and may change from release to release. Assignments into FB are still OK.
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. Note: Any variable of any type assigned inside a varying if should only be used inside the if. The contents outside the if are undefined, and may change from release to release. Assignments into FB are still OK.
repeat (uf) statement repeat (pf) statement	repeat statement max(0,floor(uf)) or max(0,floor(pf)) 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:

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.
ambientlight	Light contributing to ambient() function.
distantlight pointlight	distantlight is a light shining down the z axis. It is transformed by shadermatrix, which can be used by the application to point the light in other directions. Within the body of a distantlight, lightVector gives the light direction. It is initialized to shadermatrix[2], but can be changed by the shader. pointlight is a light positioned at the origin. It is transformed by shadermatrix, which can be used by the application to point the light in other directions. Within the body of a pointlight, lightVector gives the light direction. It is initialized to shadermatrix[3], but can be changed by the shader. Distant and point lights return the varying color and intensity of light falling on a surface. They do not compute the interaction of light with the surface itself, that interaction is computed in the surface shader through the diffuse() and specular() functions, or through setupLight() and runDiffuse() and runSpecular

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.

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;
}
```

XII. Level of Detail

Since complex shaders can sometimes be expensive in terms of texture use or rendering time, ISL includes several facilities to create several levels of detail for a single shader. The resulting LOD shader is used exactly as any normal shader, but has an extra parameter to control its rendered complexity. When an LOD shader is applied to an object, the application only needs to adjust the level parameter and the shader will handle the transitions between complex appearance when the object is close or important and simple appearance when the object is distant or unimportant.

A. Automatic LOD

The easiest form of level of detail to use is performed automatically by the OpenGL Shader compiler. If the API requirements for auto-LOD are satisfied (See the manual for *islCompileAction*). Auto-LOD is enabled for any appearance that contains a shader with the parameter:

parameter float autoLOD

When auto-LOD is enabled, shaders in the appearance will automatically be analyzed and simplified to create multiple levels of detail. These levels of detail can be controlled by setting the autoLOD variable of the **first** shader to a value between 0 (full complexity) and 1 (maximum simplification).

For example:

Original	AutoLOD

<pre>surface fancy()
{
    FB = environment("flowers.rgb");
    FB *= color(.5,.2,.0,0);
    FB = under(texture("marblebirds.rgba",
        scale(2.,2.,2.)));
}</pre>
	<pre>surface fancy(parameter float autoLOD=0)
	{
	    FB = environment("flowers.rgb");
	    FB *= color(.5,.2,.0,0);
	    FB = under(texture("marblebirds.rgba",
	        scale(2.,2.,2.)));
	}</pre>

B. Semi-automatic LOD

The next easiest form of level of detail uses building-block functions provided with OpenGL Shader that accept a simplification level parameter. These building block functions are found in the shader_include sample directory. To use semi-automatic level of detail, a shader should accept a level of detail parameter with a name **other than** autoLOD. This parameter has no special meaning to the shading compiler so can have any name you choose. Then just pass this level parameter into the building block functions.

For example:

```
surface brdf_with_fresnel (parameter float lodrange = 0; ...)
{
    // BRDF contribution
    FB = microfacetBRDF(brdfP, brdfQ, colorP, colorQ,brdfColor,
                        lodrange, lod_low, lod_mid, lod_high );

    // Fresnel contribution.
    FB = hdrFresnel (env,"fresnelRefract.bw", lodrange);
}
```

C. Manual LOD

The final method is to create level of detail shaders manually. The control mechanism for manual level of detail is the same as for semi-automatic level of detail, but instead of using LOD building blocks, you manually add conditionals to the shader to control the different levels. Manual and semi-automatic level of detail can be mixed in the same shader.

A manual level of detail shader might follow this outline:

```
surface LODshader (parameter float lodrange = 0; ...)
{
    if (lodrange < lod_low)
        ... most complex level ...
    else if (lodrange < lod_mid)
        ... second level ...
    else if (lodrange < lod_high)
        ... third level ...
    else
        ... simplest level ...
}
```

ipf2ogl(1)

NAME

ipf2ogl - OpenGL Shader Interactive Shading Language translator

SYNOPSIS

ipf2ogl [**-s** *shader-name*] [**-o** *out-file*] [*in-file*]

DESCRIPTION

The command line translator **ipf2ogl** translates a description of OpenGL passes, as output by **islc**, into C code which implements the OpenGL passes described in the input. For a given intermediate pass file, one **.c** file and one **.h** file are generated by **ipf2ogl**. The **.c** file contains the definitions of the initialization, parameter access, drawing and cleanup functions for the shader, while the **.h** file contains the prototypes for these functions. See below for a list of the generated functions.

An intermediate pass file is passed to **ipf2ogl** as the *in-file* command line argument. If *in-file* is not specified, input is read from stdin.

In addition to an input file, **ipf2ogl** can take the following command line arguments:

-s *shader-name*

Specifies the name of the shader defined by the intermediate pass file. If specified, *shader-name* will be used in place of **default** when naming all the externally visible functions defined in the generated **.c** and **.h** files. See below for a list of the generated functions.

-o *out-file*

Specifies the base name of the output files generated by **ipf2ogl**. The actual file names will be *out-file***.c** and *out-file***.h**. If **-o** *out-file* is not specified on the command line the output file names will be *shader-name***_shader.c** and *shader-name***_shader.h**.

The functions in the generated C code are defined as follows:

int **setup_default_shader** (

GLsizei *win_w*,

GLsizei *win_h*)

int **draw_default_shader** (

int (**draw_geometry*) (float, void*),

void **geometry*,

int (**load_texture*) (const char*, void*),

void **load_texture_user_data*,

GLsizei *win_w*,

GLsizei *win_h*,

GLint *rect_x*,

GLint *rect_y*,

GLint *rect_w*,

GLint *rect_h*)

int **cleanup_default_shader** (

void)

GLuint **get_default_shader_num_float_parameters** (

void)

GLuint **get_default_shader_num_color_parameters** (

void)

GLuint **get_default_shader_num_matrix_parameters** (

void)

const char* **get_default_shader_float_parameter_name** (

GLuint *param_num*)

const char* **get_default_shader_color_parameter_name** (

GLuint *param_num*)

const char* **get_default_shader_matrix_parameter_name** (

GLuint *param_num*)

GLint **set_default_shader_float_parameter** (

GLuint *param_num*,

GLfloat *param_val*)

GLint **set_default_shader_color_parameter** (

GLuint *param_num*,

GLfloat *param_val* [4])

GLint **set_default_shader_matrix_parameter** (

GLuint *param_num*,

GLfloat *param_val* [16])

setup_default_shader allocates and sets parameters for any temporary or 1D table textures used by the passes of the input intermediate pass file. If a texture originates in an image file, it is up to the user to allocate resources and set parameters for this texture. See the section on **draw_default_shader** for more information.

draw_default_shader implements the rendering of the passes defined in the input intermediate pass file. *draw_geometry* is a function that can be used to render the geometry pointed to by *geometry*. Although *geometry* is declared non-const, it is not changed by **draw_default_shader**. However, there is no restriction on what *draw_geometry* might do with it. The first argument to *draw_geometry* is a floating point number corresponding to the optional *texgen* argument to the ISL **texture()** function. The value of this floating point number is automatically filled in by **draw_default_shader**. *load_texture* is a function that can be used to load textures from image files when they are required by a shader pass. The first argument to *load_texture* is the texture name as specifed with the ISL **texture** function. The second argument to *load_texture* is a user data pointer which is specifed with the *load_texture_user_data* pointer. *win_w* and *win_h* specify the dimensions of the window. *rect_x*, *rect_y*, *rect_w* and *rect_h* specify the position and dimensions of the screen space bounding rectangle of the geometry pointed to by *geometry*.

cleanup_default_shader frees resources allocated by **setup_default_shader**. To avoid resource leaks, it is important to call **cleanup_default_shader** once **draw_default_shader** will no longer be called.

It is expected that a user application will call **setup_default_shader** once at application initialization followed by repeated calls to **draw_default_shader** followed by a call to **cleanup_default_shader** at application exit. However, calling these routines out of this expected order will not cause failures or resource leaks. For instance, calling **cleanup_default_shader** or **draw_default_shader** before calling **setup_default_shader** will simply have no effect. Also, calling **setup_default_shader** repeatedly without calling **cleanup_default_shader** in between will cause only the first **setup_default_shader** call to take effect. Other erroneous command sequences will be handled similarly.

get_default_shader_num_float_parameters, **get_default_shader_num_color_parameters** and **get_default_shader_num_matrix_parameters** return the number of float, color and matrix parameters used by the shader.

get_default_shader_float_parameter_name returns the name of the float parameter specified by *param_num*. If *param_num* is greater than or equal to the number of float parameters, NULL is returned.

get_default_shader_color_parameter_name returns the name of the color parameter specified by *param_num*. If *param_num* is greater than or equal to the number of color parameters, NULL is returned.

get_default_shader_matrix_parameter_name returns the name of the matrix parameter specified by *param_num*. If *param_num* is greater than or equal to the number of matrix parameters, NULL is returned.

set_default_shader_float_parameter sets the value of the float parameter specified by *param_num* to *param_val*. If *param_num* is greater than or equal to the number of float parameters, no state is changed and negative one is returned indicating failure. Zero is returned on success.

set_default_shader_color_parameter sets the value of the color parameter specified by *param_num* to *param_val*. If *param_num* is greater than or equal to the number of color parameters, no state is changed and negative one is returned indicating failure. Zero is returned on success.

set_default_shader_matrix_parameter sets the value of the matrix parameter specified by *param_num* to *param_val*. Matrix data should be specified in column-major order (as it is in OpenGL). If *param_num* is greater than or equal to the number of matrix parameters, no state is changed and negative one is returned indicating failure. Zero is returned on success.

EXAMPLES

The following command translates an intermediate pass file named *mytexture.ipf* and prints the generated C code to *default_shader.c* and *default_shader.h*:

```
ipf2ogl mytexture.ipf
```

The functions defined in *default_shader.c* will be named **setup_default_shader**, **draw_default_shader**, **cleanup_default_shader**, **get_default_shader_num_float_parameters**, **get_default_shader_num_color_parameters**, **get_default_shader_num_matrix_parameters**, **get_default_shader_float_parameter_name**, **get_default_shader_float_parameter_name**, **get_default_shader_float_parameter_name**, **set_default_shader_float_parameter**, **set_default_shader_color_parameter** and **set_default_shader_matrix_parameter**.

The following command translates an intermediate pass file named *mytexture.ipf* and prints the generated C code to *yourtexture.c* and *yourtexture.h*:

```
ipf2ogl -o yourtexture mytexture.ipf
```

The functions defined in *yourtexture.c* will be named **setup_default_shader**, **draw_default_shader**, **cleanup_default_shader**, etc.

The following command translates an intermediate pass file named *mytexture.ipf* and prints the generated C code to *mytexture_shader.c* and *mytexture_shader.h*:

```
ipf2ogl -s mytexture mytexture.ipf
```

This time the functions defined in *mytexture_shader.c* will be named **setup_mytexture_shader**, **draw_mytexture_shader**, **cleanup_mytexture_shader**, etc.

The following command translates an intermediate pass file named *mytexture.ipf* and prints the generated C code to *yourtexture.c* and *yourtexture.h*:

```
ipf2ogl -s mytexture -o yourtexture mytexture.ipf
```

The functions defined in *yourtexture.c* will be named **setup_mytexture_shader**, **draw_mytexture_shader**, **cleanup_mytexure_shader**, etc.

NOTES

The intermediate pass file which is read by **ipf2ogl** is not a standard and is subject to change. Applications should never depend on the format or content of this file. The intermediate pass file should not be generated by hand but always be generated by a compiler such as **islc**.

The OpenGL generated by **ipf2ogl** may not render properly on some graphics accelerators due to missing functionality, bugs, or constraints of their graphics drivers. The **ipf2ogl** translator depends heavily on OpenGL state management within the driver and strict compliance to the OpenGL specification.

It is the responsibility of the application to avoid OpenGL state conflicts with the code generated by **ipf2ogl**. The generated code makes no attempt to determine the current OpenGL state when it makes its own state changes nor can it prevent the *draw_geometry* callback from making state changes behind its back. The easiest way to avoid state conflicts is to restore OpenGL state to its default before calling the functions generated by **ipf2ogl**.

MACHINE DEPENDENCIES

If the environment variable **ISL_IMPACT_WORKAROUND** is set, **ipf2ogl** will include workarounds for known issues on systems with SGI Impact graphics (Indigo2 Impact, Octane)

If the environment variable **ISL_IR_WORKAROUND** is set, **ipf2ogl** will include workarounds for known issues on systems with SGI InfiniteReality graphics (Onyx InfiniteReality, Onyx2)

FILES

/usr/bin/ipf2ogl

  location of this command

/usr/bin/islc

  location of ISL compiler

/usr/share/shader/src/*

  sample code and documenation

/usr/share/shader/doc/*

  ISL Specification and html format man pages

SEE ALSO

shader(1), *islc*(1)

islc(1)

NAME

islc - OpenGL Shader Interactive Shading Language compiler

SYNOPSIS

islc *shader*

islc [**-I** *directory*] [**-s** *shader*] [**-a** *shader*] [**-d** *shader*] [**-l** *shader*] [**-f** *shader*] [**-v** *outfile_version*] [**-D** *hardware_capability_declaration*] ... [**-o** *outfile*]

DESCRIPTION

The command line compiler **islc** translates an appearance description into a description of OpenGL passes. When converted to an OpenGL stream with a translator such as **ipf2ogl**, this intermediate pass description will render an object with the specified appearance. An appearance is defined as one or more of: a list of surface shaders, a list of ambient light shaders, and a list of direct light shaders. The shaders are written in the OpenGL Interactive Shading Language.

Each *shader* is the name of a file containing the shader and an optional matrix:

file [*matrix*]

where the row-major *matrix* has the form:

(m00 m01 m02 m03 m10 m11 m12 m13 m20 m21 m22 m23 m30 m31 m32 m33)

If the matrix is included, the file name and matrix must together form a single argument. Since spaces are meaningful to the shell, the easiest way to achieve this is to surround the file name and matrix pair with quotation marks. The matrix specifies the default value of the *shadermatrix* global variable in the shader. If the matrix is omitted, the default *shadermatrix* is the identity. As *shadermatrix* is a parameter variable, it would typically be changed per-frame by the application. The default value is used for applications that don't set the *shadermatrix* parameter.

If only a single argument is given to **islc**, it is assumed to be a surface shader, and the compiler delivers the intermediate pass description to stdout. If more arguments are given, they are interpreted as follows:

-I *directory*

Specifies a directory to add to the end of the search path for shader or **#include** files. File names beginning with **/** are always interpreted as absolute file paths. For file names not beginning with **/**, **islc** first searches in the local directory, then any directories given in the **ISL_SHADER_PATH** environment variable, and finally in directories given with the **-I** option.

-s *shader*

Specifies the name of a file containing a surface shader. If more than one surface shader is defined on the command line, all shaders are included in the surface shader list and have effect.

-a *shader*

Specifies the name of a file containing an ambient light shader. If more than one ambient light shader is defined on the command line, all shaders are included in the appearance description and have effect.

-d *shader*

Specifies the name of a file containing a distant light shader. The direction of a distant light (before transformation by the shader matrix) points down the Z axis, a 'position' of *(0,0,1,0)*. If more than one distant light shader is defined on the command line, all shaders are included in the appearance description and have effect.

-l *shader*

Specifies the name of a file containing a local light shader. The position of a local light (before transformation by the shader matrix) is at the origin, *(0,0,0,1)*. If more than one local light shader is defined on the command line, all shaders are included in the appearance description and have effect.

-f *shader*

Specifies the name of a file containing a fog (atmosphere) shader. This shader is appended to the list of surface shaders in the appearance. If more than one fog shader is defined on the command line, all shaders are included in the surface shader list and have effect. This option is equivalent to **-s** and may be removed in a future release.

-o *outfile*

Specifies the name of a file to which the intermediate pass descriptions are written. If this argument is omitted, the result is sent to stdout.

-v *outfile_version*

Specifies the version of the file to which the intermediate pass descriptions are written. This option can be used to generate pass description files that are compatible with older versions of shader translator tools such as **ipf2ogl**. Legal file versions are 1.0, 2.0, 2.2, 2.3, 2.4, and 3.0. If no version is specified, the default version is the latest version. See the **ipf2ogl**(1) man page for more information on **ipf2ogl**.

-D *hardware_capability_declaration*

If *hardware_capability_declaration* is **current**, **islc** attempts to determine the capabilities of the current graphics hardware. **islc** must be run on a machine with graphics hardware and must have access to that hardware to use **-D current**.

-D can also be used to target hardware different from the current machine. For this option, *hardware_capability_declaration* takes one of the following values, determined by using **glGetString**(1). It is possible to get this information using **glxinfo**(1) on the target hardware, though glxinfo modifies the format of the extensions string. The list of extensions should be separated by spaces (no commas), and each should start with GL_ (glxinfo strips it off). Valid extensions should be of the form **GL_ARB_multitexture**):

ISL_GL_VENDOR=*GL_vendor_string*

ISL_GL_RENDERER=*GL_renderer_string*

ISL_GL_VERSION=*GL_version_string*

ISL_GL_EXTENSIONS=*GL_extensions_string*

It can also take the following value, determined using **glGet**(3) with an argument of **GL_MAX_TEXTURE_UNITS_ARB** or from documentation for the target hardware:

ISL_GL_TEXTURE_UNITS=*max_multitexture_units*

Use of an **ISL_GL_TEXTURE_UNITS** value other than 1 also requires a multi-texture geometry drawing function. If you are unsure, a value of 1 can be used even on hardware that does support multitexture.

Capabilities may also be set with **ISL_GL_VENDOR**, **ISL_GL_RENDERER**, **ISL_GL_VERSION**, **ISL_GL_EXTENSIONS** AND **ISL_GL_TEXTURE_UNITS** environment variables. Capabilities set with **-D** override those set using environment variables. Any capabilities not defined with an environment variable or **-D** will use generic multi-platform defaults.

EXAMPLES

The following command compiles a surface shader named *fire.isl* into a description of OpenGL passes delivered to stdout:

```
islc fire.isl
```

The following command compiles a surface shader named *cloth.isl*, illuminated by a single ambient light named *amb.isl* and two direct light sources named *pnt.isl* and *dst.isl*, into a description of OpenGL passes written to the file named *out.ipf*:

```
islc -a amb.isl -d pnt.isl -d dst.isl -s cloth.isl -o out.ipf
```

The following command compiles a surface shader named *matte.isl*, illuminated by a single direct light source named *dst.isl* having a *shadermatrix* that represents a rotation of 90 degrees around the *y* axis, into a description of OpenGL passes written to stdout:

```
islc -s matte.isl -d "dst.isl ( 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0 1)"
```

ENVIRONMENT VARIABLES

The compiler **islc** considers the following environment variables:

ISL_SHADER_PATH

This specifies a colon-separated list of directories which **islc** will search, in order, for shaders given on the command line and **#include** files within those shaders. For a more complete description of the **islc** search strategy, see the **-I** option.

ISL_GL_VENDOR

The GL vendor string, as returned by **glGetString**(3) or **glxinfo**(1). See the **-D** option.

ISL_GL_RENDERER

The GL renderer string, as returned by **glGetString**(3) or **glxinfo**(1). See the **-D** option.

ISL_GL_VERSION

The GL version string, as returned by **glGetString**(3) or **glxinfo**(1). See the **-D** option.

ISL_GL_EXTENSIONS

The GL extensions string, as returned by **glGetString**(3) or **glxinfo**(1). See the **-D** option.

ISL_GL_TEXTURE_UNITS

The GL extensions string, as returned by **glGet**(3) with the argument **GL_MAX_TEXTURE_UNITS_ARB**. See the **-D** option.

NOTES

The intermediate pass file is not a standard and is subject to change. Applications should never depend on the format or content of this file. The intermediate pass file should always be translated into another format with a program such as **ipf2ogl**. See the **ipf2ogl**(1) man page for more information about **ipf2ogl**.

FILES

/usr/bin/islc

location of this command

/usr/bin/ipf2ogl

location of OpenGL translator

/usr/share/shader/src/*

sample code and documenation

/usr/share/shader/doc/*

ISL Specification and html format man pages

BUGS

`islc` can be used to generate `IPF` code for use with OpenGL Performer v2.5 or eariler, however, these versions of Performer require v1.0 of `IPF` to be generated. `islc` incorrectly emits part of the `IPF` v1.0 specification, as reported in SGI bug #850415. A workaround is to post-process code emitted from an `islc -v 1.0` command with the following script. This script removes the non v1.0 compatible portions of code, and allows the processed `IPF` to be used with OpenGL Performer v2.5s pfShader loader.

```
#!/usr/bin/perl

# for each line in the ipf
while(<>) {
    # seen the start of a texgen block
```

```
        if (defined($texgen)) {
            # line with a USER token
            if (/ USER/) {
                $texgen=""; # kill texgen: line
                next;       # kill USER line
            }
            # non-USER line -- reset to normal
            else {
                print $texgen;
                undef($texgen);
            }
        }

        # look for new texgen line
        if (/^ *texgen:/) {
            $texgen = $_;   # remember this line & use as flag
            next;           # don't output yet
        }

        # normal line, just output
        print;
    }
```

SEE ALSO

shader(1), *ipf2ogl*(1)

NAME

islAppearance - OpenGL Shader standard appearance class

INHERITS FROM

islAppearanceBase

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islAppearance (void);
virtual ~islAppearance (void);

Setting and getting shader lists

void pushShader (ListType *type*, islShader* *shdr*);
islShader* popShader (ListType *type*);
islShader* getShader (ListType *type*, int *ii*);
int getNumShaders (ListType *type*) const;

CLASS DESCRIPTION

The islAppearance class object holds a collection of islShader objects that completely define an appearance. This includes a list of ambient light shaders, a list of distant light shaders, a list of local light shaders, and a list of surface shaders. These lists are maintained internally by the islAppearance.

Each list is specified uniquely with an enumerant of type `islAppearance::ListType` that is passed into each list management method of islAppearance. The `islAppearance::ListType` is one of: `islAppearance::AMBIENTLIGHT_LIST, islAppearance::DISTANTLIGHT_LIST, islAppearance::LOCALLIGHT_LIST, or islAppearance::SURFACE_LIST.` The islAppearance class provides an interface for setting and getting each of the list's objects. By default, all shader lists are empty.

The code to enable two local lights and a surface shader, for example, looks like:

```
islAppearance* appearance = new islAppearance();
surf->pushShader(islAppearance::SURFACE_LIST,surf);
surf->pushShader(islAppearance::LOCALLIGHT_LIST,light1);
surf->pushShader(islAppearance::LOCALLIGHT_LIST,light2);
```

The appearance also includes an implied ordering for shaders in each of the lists. The lists of shaders in the appearance are traversed in order, and each shader is visited in turn. This order is most relevant in the `islAppearance::SURFACE_LIST` because it determines the order of layered surface effects.

The ISL Library

The OpenGL Shader Interactive Shading Language Library provides a minimal interface for supporting interactive, programmable shading. The ISL Library consists of six classes that enable an application to define an appearance consisting of ISL shaders, compile that appearance into an OpenGL stream, associate the compiled appearance with geometry from the application, and, subsequently, to render the shaded geometry to an OpenGL rendering context opened by the application.

The appearance is specified through an islAppearance class object, which contains a list of active ambient light shaders, a list of active distant light shaders, a list of active local light shaders, and a list of surface shaders. Each of these shaders is contained in an islShader class object. An islAppearance is compiled into a stream of OpenGL commands held inside the ISL Library using an islCompileAction.

The compilation will take advantage of capabilities available on the current graphics hardare. It is possible to override the automatic capability detection through a set of environment variables: ISL_GL_VENDOR, ISL_GL_RENDERER, ISL_GL_VERSION, ISL_GL_EXTENSIONS, and ISL_GL_TEXTURE_UNITS, ISL_GL_ARBFP_LIMITS. See the islShape reference page for more details on these environment variables and their usage.

Application geometry is associated with the appearance through an islShape class object. The geometry is defined simply as a pointer to data and an associated user callback, which the application provides for delivering this data to the graphics pipeline. The appearance is a pointer to an islAppearance. An islShape class object can be rendered into the current OpenGL context with an islDrawAction. A simple example of drawing red geometry is shown below:

```
islShader* shader = new islShader();
shader->setShader("surface myshader() { FB = color(1,0,0,1); }");

islAppearance* appearance = new islAppearance();
appearance->pushShader(islAppearance::SURFACE_LIST, shader);

// for multi-texture capable hardware where we don't provide
// a multi-texture DrawGeometryFunc to the islShape (see below)
putenv("ISL_GL_TEXTURE_UNITS=1");
islCompileAction* compileaction = new islCompileAction();
compileaction->compile(appearance);

islShape* shape = new islShape();
shape->setAppearance(appearance);
shape->setDrawGeometryFunc(user_drawcallback);
shape->setGeometryData((void*)user_data);

islDrawAction* drawaction = new islDrawAction();
drawaction->draw(shape);
```

It is the responsibility of the application to compile the appearance when necessary (if, for example, the shaders have changed or the shader parameters have changed). It is also the responsibility of the application to ensure there are no OpenGL state collisions between the ISL Library and its own implementation. The ISL Library sets state only in the application of an islDrawAction. The islDrawAction restores all state to its original settings before returning, however it assumes most OpenGL state is set to its default values when the draw action is applied. The islDrawAction depends on the application properly setting the `glViewport` and `GL_PROJECTION_MATRIX`; these are read from the OpenGL state and possibly used during the draw action. Any errors during shader parsing, compiling, or drawing are trapped and can be queried with the help of the islError class.

There is a minor typing incompatibility between the versions of the standard template library provided with the MipsPro version 7.2 compilers and the 7.3 compilers. The OpenGL Shader ISL Library on IRIX is built with the 7.3 version compilers, but with compatibility options set to mimic the 7.2 STL types to allow use with either compiler version. If you are using the newer 7.3 compilers, you must `#define ::STL_USE_SGI_ALLOCATORS` and `STL_SGI_THREADS` **before** including isl.h in the files that directly use the OpenGL Shader API, or you can define these symbols using compiler flags. For example, using something like the following in a Makefile:

```
# these flags are required to build with version 7.3 of the
# MipsPro Compilers; they are ignored on version 7.2.1
LC++DEFS += -D::STL_USE_SGI_ALLOCATORS -DSTL_SGI_THREADS
```

These preprocessor symbols are ignored by the 7.2.1 standard template library headers, so code which may be compiled with either the 7.2.1 or 7.3 MipsPro compilers can safely define them in both cases.

METHOD DESCRIPTIONS

islAppearance()

islAppearance (void);

Constructs a new islAppearance.

~islAppearance()

virtual ~islAppearance (void);

Destroys the islAppearance.

getNumShaders()

int getNumShaders (ListType *type*) const;

Returns the number of shaders in the shader list of type *type*.

getShader()

islShader* getShader (ListType *type*, int *ii*);

Returns the shader at position *ii* in the shader list of type *type*.

popShader()

islShader* popShader (ListType *type*);

Pops the last shader on the shader list of type *type*.

pushShader()

void pushShader (ListType *type*, islShader* *shdr*);

Pushes the shader *shdr* onto the shader list of type *type*. The user must manage all memory for *shdr* explicitly - the shader lists neither delete nor copy the contents of this pointer at any point.

SEE ALSO

islAppearance, islAppearanceBase, islCompileAction, islDrawAction, islError, islShader, islShape

---

NAME

islAppearanceBase - OpenGL Shader base appearance class

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islAppearanceBase (void);
virtual ~islAppearanceBase (void);

CLASS DESCRIPTION

The islAppearanceBase class is a parent for derived islAppearance class objects that hold the complete description of the rendered appearance in a form specific to the derived appearance class.

Application geometry is associated with the appearance through an islShape class object. See the derived islAppearance class for an example.

METHOD DESCRIPTIONS

islAppearanceBase()

islAppearanceBase (void);

Constructs a new islAppearanceBase.

~islAppearanceBase()

virtual ~islAppearanceBase (void);

Destroys the islAppearanceBase.

SEE ALSO

islAppearance, islShape

NAME

islAppearanceCopy - OpenGL Shader appearance copy class

INHERITS FROM

islAppearanceBase

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islAppearanceCopy (void);
virtual ~islAppearanceCopy (void);

Setting and getting shader lists

virtual void setAppearanceCopyData (islAppearanceCopyData*);
virtual islAppearanceCopyData* getAppearanceCopyData (void) const;
islShader* getShader (islAppearance::ListType *type*, int *ii*);
int getNumShaders (islAppearance::ListType *type*) const;

CLASS DESCRIPTION

The islAppearanceCopy class object holds a deep copy of an appearance created through an islCopyAction. This appearance copy is identical to the original but at a different memory location, and with different accessors to shader members.

METHOD DESCRIPTIONS

islAppearanceCopy()

islAppearanceCopy (void);

Constructs a new islAppearanceCopy.

~islAppearanceCopy()

virtual ~islAppearanceCopy (void);

Destroys the islAppearanceCopy.

getAppearanceCopyData()

virtual islAppearanceCopyData* getAppearanceCopyData (void) const;

Returns a pointer to the islAppearanceCopyData for this appearance.

getNumShaders()

int getNumShaders (islAppearance::ListType *type*) const;

Returns the number of shaders in the shader list of type *type*.

getShader()

islShader* getShader (islAppearance::ListType *type*, int *ii*);

Returns the shader at position *ii* in the shader list of type *type*.

setAppearanceCopyData()

virtual void setAppearanceCopyData (islAppearanceCopyData*);

Set the appearance to be used when this islAppearanceCopy is applied to an islShape

SEE ALSO

islAppearanceBase, islCopyAction

NAME

islAppearanceCopyData - OpenGL Shader copy appearance data

HEADER FILE

#include <shader/isl.h>

CLASS DESCRIPTION

The islAppearanceCopyData class object holds an appearance copied from a compiled islAppearance by islCopyAction. This is an opaque data type, and cannot be explicitly constructed, destroyed, or manipulated by the application except through islAppearanceCopy and islCopyAction.

SEE ALSO

islAppearance, islAppearanceCopy, islCopyAction

---

sgi

NAME

islAppearanceSnapshot - OpenGL Shader 'snapshot' appearance class

INHERITS FROM

islAppearanceBase

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islAppearanceSnapshot (void);
virtual ~islAppearanceSnapshot (void);

Setting and getting shader lists

virtual void setAppearanceSnapshotData (islAppearanceSnapshotData*);
virtual islAppearanceSnapshotData* getAppearanceSnapshotData (void) const;

CLASS DESCRIPTION

The islAppearanceSnapshot class object holds an appearance 'frozen' from a compiled islAppearance by islSnapshotAction. In this appearance shapshot, all run-time parameter expressions and control constructs are pre-evaluated into object of the islAppearanceSnapshotData class. This allows a multi-threaded application to split the parameter evaluation and drawing portions of the normal

islAppearance draw into separate execution threads.

METHOD DESCRIPTIONS

islAppearanceSnapshot()

islAppearanceSnapshot (void);

Constructs a new islAppearanceSnapshot.

~islAppearanceSnapshot()

virtual ~islAppearanceSnapshot (void);

Destroys the islAppearanceSnapshot.

getAppearanceSnapshotData()

virtual islAppearanceSnapshotData* getAppearanceSnapshotData (void) const;

Returns a pointer to the islAppearanceSnapshotData for this appearance.

setAppearanceSnapshotData()

virtual void setAppearanceSnapshotData (islAppearanceSnapshotData*);

Set the appearance to be used when this islAppearanceSnapshot is applied to an islShape

SEE ALSO

islAppearance, islAppearanceBase, islAppearanceSnapshotData, islSnapshotAction

NAME

islAppearanceSnapshotData - OpenGL Shader 'snapshot' appearance data

HEADER FILE

#include <shader/isl.h>

CLASS DESCRIPTION

The islAppearanceSnapshotData class object holds an appearance 'frozen' from a compiled islAppearance by islSnapshotAction. This is an opaque data type, and cannot be explicitly constructed, destroyed, or manipulated by the application except through islAppearanceSnapshot and islSnapshotAction.

SEE ALSO

islAppearance, islAppearanceSnapshot, islSnapshotAction

NAME

islCompileAction - OpenGL Shader compiler class

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islCompileAction (const char* *compiler*="isl");
virtual ~islCompileAction (void);

Setting and getting image data-loading information

virtual void setLoadImageData (void* *user_data*);
virtual void* getLoadImageData (void);
virtual void setLoadImageFunc (LoadImageFunc *load_image*);
virtual LoadImageFunc getLoadImageFunc (void) const;

rendering methods

virtual int compile (const islAppearance* *appearance*);
virtual int isCompiled (const islAppearance* *appearance*);
virtual int getNumErrors (void) const;
virtual int getError (islError& *error*);

CLASS DESCRIPTION

The islCompileAction class provides an interface for compiling islAppearance objects. The compile() method compiles the appearance given in an islAppearance into a representation of a stream of OpenGL commands that is cached inside the ISL Library. This stream is completely independent of geometry. It can be associated with geometry in an islShape and drawn with an islDrawAction. The isCompiled() method can be used to query if a given appearance has been compiled and its data stream cached. It is up to the application to ensure the cached stream properly reflects the current appearance with calls to compile(). In general, this will be true if no shader source code or uniform shader parameters in the islAppearance have been modified since compile() was previously called, and no `#include` files changed on disk.

Image Data

An islCompileAction may be provided with a callback function to load image data for textures used by the shader. If this callback is provided, the image data may be used to improve shader performance or create levels of detail for compiled shaders. If this callback is not provided, no level of detail (LOD) simplifications using textures will be attempted for any shader. This function is of type :

```
bool (*LoadImageFunc)(
    const char* name, void* user_data,
    int &components,
    int &width, int &height, int &depth, int &border,
    unsigned int &format, unsigned int &type,
    int *&pixels);
```

The argument *name* is equivalent to the string passed to the texture, environment, or project operation, and the argument *user_data* is specified in the islCompileAction class object and passed through to the application callback without modification. The remaining parameters are equivalent to the parameters in the glTexImage* functions. Memory for the image data array is allocated by the application, but need only remain valid until control is returned to the application or the next call to LoadImageFunc. The callback should return true if image data is available for the given texture or false if no image data is available, if the texture is computed at run-time, or if this texture should not participate in the automatic level-of-detail simplifications.

Local textures may be created during level-of-detail simplifications. It is expected that these textures will also be managed by the application. Local textures are identified by their names: which begin with the prefix "islloctx_". If the LoadImageFunc is passed the *name* that exactly matches "islloctx_", it should return true if it is prepared to manage local textures, and false otherwise. For example, during level-of-detail simplification, the LoadImageFunc may be asked to load an image named "islloctx_3_foo.tx" (where 3 could be replaced with any integer). This means that the simplification is going to make a local texture, starting with "foo.tx" as the base texture. (The base texture is always one referenced by one of the loaded shaders).

The first time this particular file name is requested, the LoadImageFunc should recognize the "islloctx_integer_" prefix pattern, make a copy of the base image "foo.tx" (loading it from file if necessary), rename it to "islloctx_3_foo.tx", and return it. The new local texture should be maintained by the application, and returned the next time it is requested by name. The level-of-detail simplification will manipulate the data in the texture after it is copied, so the application needs to maintain that data (i.e. it is not sufficient to re-create it by copying the base texture again next time it is requested). The level-of-detail simplification may operate recursively on the local textures: i.e., it may later request image named "islloctx_0_islloctx_3_foo.tx". The application should similarly copy the local texture " islloctx_3_foo.tx", that it is maintaining, rename it "islloctx_0_islloc_3_foo.tx, and return a pointer to the data.

Note that to enable automatic level of detail, at least one shader in an appearance must also take parameter float autoLOD. The first autoLOD parameter in the appearance is the one that will be used. One easy way to control autoLOD is to create an empty shader for use as the first shader in an appearance solely to enable autoLOD and control the simplification level:

```
surface LOD(parameter float autoLOD=0) { }
```

The ISL Library

The OpenGL Shader Interactive Shading Language Library provides a minimal interface for supporting interactive, programmable shading. The ISL Library consists of six classes that enable an application to define an appearance consisting of ISL shaders, compile that appearance into an OpenGL stream, associate the compiled appearance with geometry from the application, and, subsequently, to render the shaded geometry to an OpenGL rendering context opened by the application.

The appearance is specified through an islAppearance class object, which contains a list of active ambient light shaders, a list of active distant light shaders, a list of active local light shaders, and a list of surface shaders. Each of these shaders is contained in an islShader class object. An islAppearance is compiled into a stream of OpenGL commands held inside the ISL Library using an islCompileAction.

The compilation will take advantage of capabilities available on the current graphics hardare. It is possible to override the automatic capability detection through a set of environment variables: ISL_GL_VENDOR, ISL_GL_RENDERER, ISL_GL_VERSION, ISL_GL_EXTENSIONS, and ISL_GL_TEXTURE_UNITS. The last is useful if you are running on multi-texture capable hardware, but do not have a multi-texture capable DrawGeometryFunc for your islShape

Application geometry is associated with the appearance through an islShape class object. The geometry is defined simply as a pointer to data and an associated user callback, which the application provides for delivering this data to the graphics pipeline. The appearance is a pointer to an islAppearance. An islShape class object can be rendered into the current OpenGL context with an islDrawAction. A simple example of drawing red geometry is shown below:

```
islShader* shader = new islShader();
shader->setShader("surface myshader() { FB = color(1,0,0,1); }");

islAppearance* appearance = new islAppearance();
appearance->setShaderList(islAppearance::SURFACE_LIST,shader);

// for multi-texture capable hardware where we don't provide
// a multi-texture DrawGeometryFunc to the islShape (see below)
putenv("ISL_GL_TEXTURE_UNITS=1");
islCompileAction* compileaction = new islCompileAction();
compileaction->compile(appearance);

islShape* shape = new islShape();
shape->setAppearance(appearance);
shape->setDrawGeometryFunc(user_drawcallback);
shape->setGeometryData((void*)user_data);

islDrawAction* drawaction = new islDrawAction();
drawaction->draw(shape);
```

It is the responsibility of the application to compile the appearance when necessary (if, for example, the shaders have changed or the shader parameters have changed). It is also the responsibility of the application to ensure there are no OpenGL state collisions between the ISL Library and its own implementation. The ISL Library sets state only in the application of an islDrawAction. The islDrawAction restores all state to its original settings before returning, however it assumes most OpenGL state is set to its default values when the draw action is applied. The islDrawAction depends on the application properly setting the `glViewport` and `GL_PROJECTION_MATRIX`; these are read from the OpenGL state and possibly used during the draw action. Any errors during shader parsing, compiling, or drawing are trapped and can be queried with the help of the islError class.

There is a minor typing incompatibility between the versions of the standard template library provided with the MipsPro version 7.2 compilers and the 7.3 compilers. The OpenGL Shader ISL Library on IRIX is built with the 7.3 version compilers, but with compatibility options set to mimic the 7.2 STL types to allow use with either compiler version. If you are using the newer 7.3 compilers, you must `#define ::STL_USE_SGI_ALLOCATORS` and `STL_SGI_THREADS` **before** including isl.h in the files that directly use the OpenGL Shader API, or you can define these symbols using compiler flags. For example, using something like the following in a Makefile:

```
# these flags are required to build with version 7.3 of the
# MipsPro Compilers; they are ignored on version 7.2.1
```

```
LC++DEFS += -D::STL_USE_SGI_ALLOCATORS -DSTL_SGI_THREADS
```

These preprocessor symbols are ignored by the 7.2.1 standard template library headers, so code which may be compiled with either the 7.2.1 or 7.3 MipsPro compilers can safely define them in both cases.

METHOD DESCRIPTIONS

islCompileAction()

islCompileAction (const char* *compiler*="isl");

Constructs a new islCompileAction. The *compiler* argument specifies the Interactive Shading Language compiler to be used to convert the islShader objects contained in the islAppearance into OpenGL. Currently, only a single compiler is supported, and *compiler* is ignored.

~islCompileAction()

virtual ~islCompileAction (void);

Destroys the islCompileAction.

compile()

virtual int compile (const islAppearance* *appearance*);

Recompiles all of the shaders that are given in *appearance* to generate a stream of OpenGL commands that is cached within the ISL Library. Returns -1 if an error condition has occurred; otherwise returns 0.

getError()

virtual int getError (islError& *error*);

Gets the next error from the list of errors found by compile(). Each subsequent call to getError gets the next error in the list until all errors have been returned. The return value is 1 if an error was available and 0 if no errors were left in the list.

getLoadImageData()

virtual void* getLoadImageData (void);

Gets the pointer to user data that is passed through to the `islCompileAction::LoadImageFunc` callback function.

getLoadImageFunc()

virtual LoadImageFunc getLoadImageFunc (void) const;

Returns the pointer to the current LoadImageFunc callback.

getNumErrors()

virtual int getNumErrors (void) const;

Returns number of errors from calls to compile() that can be read with getError().

isCompiled()

virtual int isCompiled (const islAppearance* *appearance*);

Returns 1 if *appearance* was successfully compiled and its results successfully cached with compile(); otherwise returns 0. It is the responsibility of the application to track the need for a recompile if there have been any changes to the shader lists in *appearance* or the individual shaders.

setLoadImageData()

virtual void setLoadImageData (void* *user_data*);

Sets a pointer to user data that is passed through to the `islCompileAction::LoadImageFunc` callback function. The data is unmodified by the islCompileAction.

setLoadImageFunc()

virtual void setLoadImageFunc (LoadImageFunc *load_image*);

Sets a pointer to an `islCompileAction::LoadImageFunc` callback function. If provided, this callback may be called during compilation to improve the performance of the compiled shader or for the creation of shader levels-of-detail.

ENVIRONMENT VARIABLES

When an islCompileAction is constructed, it queries the current graphics hardware for its capabilities. These queries can be overridden by the following environment variables:

```
    ISL_GL_VENDOR (overrides glGetString(GL_VENDOR))
    ISL_GL_RENDERER (overrides glGetString(GL_RENDERER))
    ISL_GL_VERSION (overrides glGetString(GL_VERSIION))
    ISL_GL_EXTENSIONS (overrides glGetString(GL_EXTENSIONS))
    ISL_GL_TEXTURE_UNITS (overrides glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB,&x))     ISL_GL_TEXTURE_UNITS
(overrides glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB,&x))
```

The latter can be particularly useful if you are running on multi-texture capable hardware, but do not have a multi-texture support in the DrawGeometryFunc for your islShape

SEE ALSO

islAppearance, islCompileAction, islDrawAction, islError, islShader, islShape

NAME

islCopyAction - OpenGL Shader appearance copy action class

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islCopyAction (islMemory* *mm*=NULL);
virtual ~islCopyAction (void);

Methods to manage snapshots

virtual islAppearanceCopyData* copy (const islAppearance*) const;
virtual void deleteCopy (islAppearanceCopyData*) const;

CLASS DESCRIPTION

The islCopyAction class provides an interface for deep-copying a compiled islAppearance for potential placement elsewhere in memory. An example use of an islCopyAction might be to place an in a shared memory arena.

islAppearanceCopy performs copies on *previously compiled* appearances. copy() will return NULL if the appearance specifed is not compiled.

Allocation of islAppearanceCopyData

The islAppearanceCopyData created by and deleted by deleteCopy(), can be allocated by specifying an islMemory to the islCopyAction constructor. If no is specified, a default islMemory will be used.

METHOD DESCRIPTIONS

islCopyAction()

islCopyAction (islMemory* *mm*=NULL);

Constructs a new islCopyAction. The object argument, if specified, will be used for allocating and freeing all memory used by the snapshot process. If no islMemory is specified (or NULL is specified) a default allocator will be used.

~islCopyAction()

virtual ~islCopyAction (void);

Destroys the islCopyAction. Does not delete any previously allocated islAppearanceCopyData that were not explicitly deallocated by calls to deleteCopy().

copy()

virtual islAppearanceCopyData* copy (const islAppearance*) const;

Copy the islAppearance. Returns a pointer an object of the islAppearanceCopyData class representing the copied appearance. The copy only works correctly on a compiled appearance. copy() will return NULL if the appearance specifed is not compiled or if any other error condition occurs.

deleteCopy()

virtual void deleteCopy (islAppearanceCopyData*) const;

Delete memory associated with copied appearance.

SEE ALSO

islAppearance, islAppearanceCopyData, islCopyAction, islMemory

---

NAME

islDrawAction - OpenGL Shader rendering class

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islDrawAction (void);
virtual ~islDrawAction (void);

Setting and getting texture loading information

virtual void setLoadTextureData (void* *user_data*);
virtual void* getLoadTextureData (void);
virtual void setLoadTextureFunc (LoadTextureFunc *load_texture*);
virtual LoadTextureFunc getLoadTextureFunc (void) const;

Setting and getting image data loading information

virtual void setLoadImageData (void* *user_data*);
virtual void* getLoadImageData (void);
virtual void setLoadImageFunc (LoadImageFunc *load_image*);
virtual LoadImageFunc getLoadImageFunc (void) const;

drawing methods

virtual int draw (const islShape* *shape*);
virtual int getNumErrors (void) const;
virtual int getError (islError& *error*);

CLASS DESCRIPTION

The islDrawAction class provides an interface for drawing islShape objects. The draw() method uses the OpenGL stream cached when the appearance of the shape was last successfully compiled with an islCompileAction or islSnapshotAction. This stream is applied to the geometry of the shape and rendered to the current OpenGL context. Nothing is drawn if the appearance has not been compiled previously. It is up to the application to ensure the cached stream properly reflects the current appearance in the islShape. In general, this will be true if no shader code or `uniform` shader parameters have changed in an islAppearance since it was compiled, and no `parameter` parameters or shader matrices have changed in an islAppearanceSnapshot. It is OK if the geometry data, geometry callback, and/or screen space bounding boxes of the geometry have changed.

Binding Textures

The islDrawAction class provides an interface to specify an application callback function to load textures into the current graphics context. This function will be called when the ISL Library encounters a texture name in a texture, environment, or project operation. This function is of type `islDrawAction::LoadTextureFunc`:

```
int (*LoadTextureFunc)(const char* name, void* user_data);
```

The argument *name* is equivalent to the string passed to the texture, environment, or project operation, and the argument *user_data* is specified in the islDrawAction class object and passed through to the application callback without modification. It is the responsibility of the callback to ensure that the desired texture is downloaded and ready to be used by the time it returns. The callback should return -1 if unsuccessful; otherwise it should return the dimension of the texture that was downloaded. The ISL Library uses the dimension to enable and disable texturing appropriately.

All management of named textures is the responsibility of the application through this callback. It can, for example, use the texture name to index into a cache of texture ids it generates with `glGenTextures`. If the texture has been downloaded previously, the callback need only bind the proper texture id and return. The callback should use only texture object OpenGL calls such as `glBindTexture`, `glTexParameter`, and `glTexImage2D` to specify and download the named texture into the current OpenGL context. It should not call any other OpenGL functions.

The ISL Library

The OpenGL Shader Interactive Shading Language Library provides a minimal interface for supporting interactive, programmable shading. The ISL Library consists of six classes that enable an application to define an appearance consisting of ISL shaders, compile that appearance into an OpenGL stream, associate the compiled appearance with geometry from the application, and, subsequently, to render the shaded geometry to an OpenGL rendering context opened by the application.

The appearance is specified through an islAppearance class object, which contains a list of active ambient light shaders, a list of active distant light shaders, a list of active local light shaders, and a list of surface shaders. Each of these shaders is contained in an islShader class object. An islAppearance is compiled into a stream of OpenGL commands held inside the ISL Library using an islCompileAction.

The compilation will take advantage of capabilities available on the current graphics hardare. It is possible to override the automatic capability detection through a set of environment variables: ISL_GL_VENDOR, ISL_GL_RENDERER, ISL_GL_VERSION, ISL_GL_EXTENSIONS, and ISL_GL_TEXTURE_UNITS. The last is useful if you are running on multi-texture capable hardware, but do not have a multi-texture capable DrawGeometryFunc for your islShape

Application geometry is associated with the appearance through an islShape class object. The geometry is defined simply as a pointer to data and an associated user callback, which the application provides for delivering this data to the graphics pipeline. The appearance is a pointer to an islAppearance. An islShape class object can be rendered into the current OpenGL context with an islDrawAction. A simple example of drawing red geometry is shown below:

```
islShader* shader = new islShader();
shader->setShader("surface myshader() { FB = color(1,0,0,1); }");

islAppearance* appearance = new islAppearance();
appearance->pushShader(islAppearance::SURFACE_LIST,shader);

// for multi-texture capable hardware where we don't provide
// a multi-texture DrawGeometryFunc to the islShape (see below)
putenv("ISL_GL_TEXTURE_UNITS=1");
islCompileAction* compileaction = new islCompileAction();
compileaction->compile(appearance);

islShape* shape = new islShape();
shape->setAppearance(appearance);
shape->setDrawGeometryFunc(user_drawcallback);
shape->setGeometryData((void*)user_data);

islDrawAction* drawaction = new islDrawAction();
drawaction->draw(shape);
```

It is the responsibility of the application to compile the appearance when necessary (if, for example, the shaders have changed or the shader parameters have changed). It is also the responsibility of the application to ensure there are no OpenGL state collisions between the ISL Library and its own implementation. The ISL Library sets state only in the application of an islDrawAction. The islDrawAction restores all state to its original settings before returning, however it assumes most OpenGL state is set to its default values when the draw action is applied. The islDrawAction depends on the application properly setting the `glViewport` and `GL_PROJECTION_MATRIX`; these are read from the OpenGL state and possibly used during the draw action. Any errors during shader parsing, compiling, or drawing are trapped and can be queried with the help of the islError class.

There is a minor typing incompatibility between the versions of the standard template library provided with the MipsPro version 7.2 compilers and the 7.3 compilers. The OpenGL Shader ISL Library on IRIX is built with the 7.3 version compilers, but with compatibility options set to mimic the 7.2 STL types to allow use with either compiler version. If you are using the newer 7.3 compilers, you must `#define ::STL_USE_SGI_ALLOCATORS` and `STL_SGI_THREADS` **before** including isl.h in the files that directly use the OpenGL Shader API, or you can define these symbols using compiler flags. For example, using something like the following in a Makefile:

```
# these flags are required to build with version 7.3 of the
# MipsPro Compilers; they are ignored on version 7.2.1
LC++DEFS += -D::STL_USE_SGI_ALLOCATORS -DSTL_SGI_THREADS
```

These preprocessor symbols are ignored by the 7.2.1 standard template library headers, so code which may be compiled with either the 7.2.1 or 7.3 MipsPro compilers can safely define them in both cases.

METHOD DESCRIPTIONS

islDrawAction()

islDrawAction (void);

Constructs a new islDrawAction.

~islDrawAction()

virtual ~islDrawAction (void);

Destroys the islDrawAction.

draw()

virtual int draw (const islShape* *shape*);

Draws the shape into the current OpenGL context using the OpenGL stream that was cached when the appearance of the shape was last compiled. Returns -1 if an error condition has occurred; otherwise returns 0.

getError()

virtual int getError (islError& *error*);

Gets the next error from the list of errors found by render() or redraw(). Each subsequent call to getError gets the next error in the list until all errors have been returned. The return value is 1 if an error was available and 0 if no errors were left in the list.

getLoadImageData()

virtual void* getLoadImageData (void);

Gets the pointer to user data that is passed through to the `islDrawAction::LoadImageFunc` callback function. (This method is reserved for future expansion)

getLoadImageFunc()

virtual LoadImageFunc getLoadImageFunc (void) const;

Returns the pointer to the current LoadImageFunc callback. (This method is reserved for future expansion)

getLoadTextureData()

virtual void* getLoadTextureData (void);

Gets the pointer to user data that is passed through to the `islDrawAction::LoadTextureFunc` callback function.

getLoadTextureFunc()

virtual LoadTextureFunc getLoadTextureFunc (void) const;

Returns the pointer to the current LoadTextureFunc callback function.

getNumErrors()

virtual int getNumErrors (void) const;

Returns number of errors from calls to draw() that can be read with getError().

setLoadImageData()

virtual void setLoadImageData (void* *user_data*);

Sets a pointer to user data that is passed through to the `islDrawAction::LoadImageFunc` callback function. The data is unmodified by the islDrawAction. (This method is reserved for future expansion)

setLoadImageFunc()

virtual void setLoadImageFunc (LoadImageFunc *load_image*);

Sets a pointer to an `islDrawAction::LoadImageFunc` callback function. (This method is reserved for future expansion)

setLoadTextureData()

virtual void setLoadTextureData (void* *user_data*);

Sets a pointer to user data that is passed through to the `islDrawAction::LoadTextureFunc` callback function. The data is unmodified by the islDrawAction.

setLoadTextureFunc()

virtual void setLoadTextureFunc (LoadTextureFunc *load_texture*);

Sets a pointer to an `islDrawAction::LoadTextureFunc` callback function. If this function is not specified, loading of textures is ignored entirely by the islDrawAction. The callback is responsible for using OpenGL calls (such as `glBindTexture` and `glTexImage2D`) to download a texture of a given name.

SEE ALSO

islAppearance, islAppearanceSnapshot, islCompileAction, islDrawAction, islError, islShader, islShape, islSnapshotAction

NAME

islError - OpenGL Shader error class

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islError (void);
virtual ~islError (void);

Getting error information

virtual const char* getFileName (void) const;
virtual int getLine (void) const;
virtual const char* getMessage (void) const;
virtual ErrorClass getErrorClass (void) const;

CLASS DESCRIPTION

The islError class object contains information about a single error encountered while compiling or drawing shaders. Errors are queried through the islShader::getError(), islCompileAction::getError() and islDrawAction::getError() methods.

Each error includes a file name or shader identifier, a line number, a user-readable message, and an error class from the enumerated type `islError::ErrorClass`.

Error classes

Each error has a class from `islError::ErrorClass`, which can be one of the following: `islError::NO_ERROR`, `islError::FATAL_ERROR`, `islError::FILE_ERROR`, `islError::SYNTAX_ERROR`, `islError::DECLARE_ERROR`, `islError::UNDECLARED_ERROR`, `islError::ARGUMENT_ERROR`, `islError::TYPE_ERROR`, `islError::UNSUPPORTED_ERROR` or `islError::RENDER_ERROR`.

`islError::NO_ERROR`: used only for newly created islError objects and when there are no more errors left to report from one of the getError functions. An error of the `islError::NO_ERROR` class will also have the file set and message set to an empty string and the line number set to -1.

`islError::FATAL_ERROR`: an error (such as out of memory) from which there is no chance of recovery.

`islError::FILE_ERROR`: a problem loading an include file.

`islError::SYNTAX_ERROR`: a shader syntax error.

`islError::DECLARE_ERROR`: an error in a variable or function declaration.

`islError::UNDECLARED_ERROR`: use of a variable or function that has not been defined in the shader.

`islError::ARGUMENT_ERROR`: an error in the arguments passed to a function.

`islError::TYPE_ERROR`: an attempt to perform a shading operation on an incompatible type (e.g. "string" + number).

`islError::UNSUPPORTED_ERROR`: an unsupported language feature.

`islError::RENDER_ERROR`: an error in rendering.

METHOD DESCRIPTIONS

islError()

islError (void);

Constructs a new islError of error class `islError::NO_ERROR`.

~islError()

virtual ~islError (void);

Destroys the islError.

getErrorClass()

virtual ErrorClass getErrorClass (void) const;

Returns the error class for this error, from the enumerated type `ErrorClass`

getFileName()

virtual const char* getFileName (void) const;

Get the name of the file or string identifying the shader where the error occurred. If there is no identifying string, an empty string is returned. The return value is never NULL.

getLine()

virtual int getLine (void) const;

Returns the line where the error occurred. If there is no line number associated with the error, returns -1.

getMessage()

virtual const char* getMessage (void) const;

Returns a human-readable message explaining the error. If there is no message (i.e. for `islError::NO_ERROR`), returns an empty string. The return value is never NULL.

SEE ALSO

islCompileAction, islDrawAction, islError, islShader

NAME

islShader - OpenGL Shader Interactive Shading Language shader class

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islShader (void);
virtual ~islShader (void);

Setting and getting shader information

virtual int setShader (const char* *shader*);
virtual char* getShader (void) const;
virtual void setIncludePath (const char* *path*);
virtual char* getIncludePath (void) const;
virtual void setShaderMatrix (const float* *matrix*);
virtual void getShaderMatrix (float* *matrix*);
virtual char* getName (void) const;
virtual int getNumErrors (void) const;
virtual int getError (islError& *error*);

Setting and getting shader parameters

virtual int getParameter (const char* *name*);
virtual int getNumParameters (void);
virtual ParameterType getParameterType (int *param*);
virtual char* getParameterName (int *param*) const;
virtual int getParameterFloat (int *param*, float& *val*);
virtual int setParameterFloat (int *param*, float *val*);
virtual int getParameterColor (int *param*, float& *r*, float& *g*, float& *b*, float& *a*);
virtual int setParameterColor (int *param*, float *r*, float *g*, float *b*, float *a*);
virtual int getParameterMatrix (int *param*, float* *val*);
virtual int setParameterMatrix (int *param*, const float* *val*);
virtual int getParameterString (int *param*, char*& *val*);
virtual int setParameterString (int *param*, const char* *val*);

CLASS DESCRIPTION

The islShader class object contains a single shader defined in the Interactive Shading Language (ISL) and supplies an interface to setting and getting its name, matrix, parameters, and the shader itself. A string containing an ISL shader is passed to the islShader with the setShader() method. The shader string is parsed immediately to extract any shader parameters. The number of parameters, their types, and their values can be queried through islShader methods, and their values can be queried and set through additional methods. Errors that are encountered during parsing can be queried with getError().

The islShader class object also contains a path in which it searches for files incorporated into the shader with an `#include` directive. This string is set by with the setIncludePath() method and is interpreted as a colon-separated list of directories that are searched, in order. If the `ISL_INCLUDE_PATH` environment variable is set, its value is prepended to that specified by setIncludePath(). If `ISL_INCLUDE_PATH` is not set and setIncludePath() has not been called, only the local directory is searched.

It is possible to use the `#include` directive to pull files into the islShader class object directly from disk by using code of the form (to load the shader `/usr/shaders/myshader.isl`):

```
islShader* shader = new islShader();
islShader->setIncludePath("/usr/shaders/");
islShader->setShader("#include \"myshader.isl\"");
```

Parameters are identified with unique integer indices from 0 to one less than the total number of parameters (which may be queried with getNumParameters()). The index of a parameter may be obtained from the name it has in the ISL shader with the getParameter() method. Parameter types are specified as enumerated values of type `islShader::ParameterType`, which can be one of the following: `islShader::PARAMETER_UNKNOWN` `islShader::PARAMETER_FLOAT`, `islShader::PARAMETER_COLOR`, `islShader::PARAMETER_MATRIX`, or `islShader::PARAMETER_STRING`.

The ISL Library

The OpenGL Shader Interactive Shading Language Library provides a minimal interface for supporting interactive, programmable shading. The ISL Library consists of six classes that enable an application to define an appearance consisting of ISL shaders, compile that appearance into an OpenGL stream, associate the compiled appearance with geometry from the application, and, subsequently, to render the shaded geometry to an OpenGL rendering context opened by the application.

The appearance is specified through an islAppearance class object, which contains a list of active ambient light shaders, a list of active distant light shaders, a list of active local light shaders, and a list of surface shaders. Each of these shaders is contained in an islShader class object. An islAppearance is compiled into a stream of OpenGL commands held inside the ISL Library using an islCompileAction.

The compilation will take advantage of capabilities available on the current graphics hardare. It is possible to override the automatic capability detection through a set of environment variables: ISL_GL_VENDOR, ISL_GL_RENDERER, ISL_GL_VERSION, ISL_GL_EXTENSIONS, and ISL_GL_TEXTURE_UNITS. The last is useful if you are running on multi-texture capable hardware, but do not have a multi-texture capable DrawGeometryFunc for your islShape

Application geometry is associated with the appearance through an islShape class object. The geometry is defined simply as a pointer to data and an associated user callback, which the application provides for delivering this data to the graphics pipeline. The appearance is a pointer to an islAppearance. An islShape class object can be rendered into the current OpenGL context with an islDrawAction. A simple example of drawing red geometry is shown below:

```
islShader* shader = new islShader();
shader->setShader("surface myshader() { FB = color(1,0,0,1); }");

islAppearance* appearance = new islAppearance();
appearance->pushShader( islAppearance::SURFACE_LIST, surf );

// for multi-texture capable hardware where we don't provide
// a multi-texture DrawGeometryFunc to the islShape (see below)
putenv("ISL_GL_TEXTURE_UNITS=1");
islCompileAction* compileaction = new islCompileAction();
compileaction->compile(appearance);

islShape* shape = new islShape();
shape->setAppearance(appearance);
shape->setDrawGeometryFunc(user_drawcallback);
```

```
shape->setGeometryData((void*)user_data);

islDrawAction* drawaction = new islDrawAction();
drawaction->draw(shape);
```

It is the responsibility of the application to compile the appearance when necessary (if, for example, the shaders have changed or the shader parameters have changed). It is also the responsibility of the application to ensure there are no OpenGL state collisions between the ISL Library and its own implementation. The ISL Library sets state only in the application of an islDrawAction. The islDrawAction restores all state to its original settings before returning, however it assumes most OpenGL state is set to its default values when the draw action is applied. The islDrawAction depends on the application properly setting the `glViewport` and `GL_PROJECTION_MATRIX`; these are read from the OpenGL state and possibly used during the draw action. Any errors during shader parsing, compiling, or drawing are trapped and can be queried with the help of the islError class.

There is a minor typing incompatibility between the versions of the standard template library provided with the MipsPro version 7.2 compilers and the 7.3 compilers. The OpenGL Shader ISL Library on IRIX is built with the 7.3 version compilers, but with compatibility options set to mimic the 7.2 STL types to allow use with either compiler version. If you are using the newer 7.3 compilers, you must `#define ::STL_USE_SGI_ALLOCATORS` and `STL_SGI_THREADS` **before** including isl.h in the files that directly use the OpenGL Shader API, or you can define these symbols using compiler flags. For example, using something like the following in a Makefile:

```
# these flags are required to build with version 7.3 of the
# MipsPro Compilers; they are ignored on version 7.2.1
LC++DEFS += -D::STL_USE_SGI_ALLOCATORS -DSTL_SGI_THREADS
```

These preprocessor symbols are ignored by the 7.2.1 standard template library headers, so code which may be compiled with either the 7.2.1 or 7.3 MipsPro compilers can safely define them in both cases.

METHOD DESCRIPTIONS

islShader()

islShader (void);

Constructs a new islShader.

~islShader()

virtual ~islShader (void);

Destroys the islShader.

getError()

virtual int getError (islError& *error*);

Gets the next error from the list of errors found by setShader(). Each subsequent call to getError gets the next error in the list until all errors have been returned. The return value is 1 if an error was available and 0 if no errors were left in the list.

getIncludePath()

virtual char* getIncludePath (void) const;

Gets the islShader include path.

getName()

virtual char* getName (void) const;

Gets the islShader name, which is extracted from the shader string. This value is NULL until setShader() has been called. This name is used to identify the shader when diagnostic information, such as an error message, is generated.

getNumErrors()

virtual int getNumErrors (void) const;

Returns number of errors from calls to setShader() that can be read with getError().

getNumParameters()

virtual int getNumParameters (void);

Returns the total number of parameters in the shader.

getParameter()

virtual int getParameter (const char* *name*);

Returns the index of the shader parameter with the given *name*. The value -1 is returned if *name* is not a parameter of the shader. The index is a unique identifier that can be used to get the parameter type (with getParameterType()), get the parameter name (with getParameterName()), and get and set the parameter value (with getParameterFloat(), setParameterFloat(), getParameterColor(), setParameterColor(), getParameterMatrix(), setParameterMatrix(), getParameterString(), and setParameterString().)

getParameterColor()

virtual int getParameterColor (int *param*, float& *r*, float& *g*, float& *b*, float& *a*);

Gets the value of the parameter whose index is *param* into *r*, *g*, *b*, and *a*. If *param* does not index a parameter of type `islShader::PARAMETER_COLOR`, -1 is returned; otherwise 0 is returned.

getParameterFloat()

virtual int getParameterFloat (int *param*, float& *val*);

Places the value of the parameter whose index is *param* into *val*. If *param* does not index a parameter of type `islShader::PARAMETER_FLOAT`, -1 is returned; otherwise 0 is returned.

getParameterMatrix()

virtual int getParameterMatrix (int *param*, float* *val*);

Places the value of the parameter whose index is *param* into *val*. The matrix is an array of 16 floating point values given in column-major form (as in OpenGL). The storage must be allocated by the application. If *param* does not index a parameter of type `islShader::PARAMETER_MATRIX`, -1 is returned; otherwise 0 is returned.

getParameterName()

virtual char* getParameterName (int *param*) const;

Returns the name of the parameter whose index is *param*.

getParameterString()

virtual int getParameterString (int *param*, char*& *val*);

Places the value of the parameter whose index is *param* into *val*. If *param* does not index a parameter of type `islShader::PARAMETER_STRING`, -1 is returned; otherwise 0 is returned.

getParameterType()

virtual ParameterType getParameterType (int *param*);

Returns the ParameterType of the parameter whose index is *param*. ParameterType is one of `islShader::PARAMETER_FLOAT`, `islShader::PARAMETER_COLOR`, `islShader::PARAMETER_MATRIX`, or `islShader::PARAMETER_STRING`. If *param* is not a valid parameter index, `islShader::PARAMETER_UNKNOWN` is returned.

getShader()

virtual char* getShader (void) const;

Gets the islShader shader string.

getShaderMatrix()

virtual void getShaderMatrix (float* *matrix*);

Gets the islShader shader matrix. The matrix is an array of 16 floating point values given in column-major form (as in OpenGL). The storage must be allocated by the application.

setIncludePath()

virtual void setIncludePath (const char* *path*);

Sets the islShader include path. If set, *path* is interpreted as a colon-separated list of directories in which the setShader() method will search, in order, for any header files included by the shader. If this method has not been called, only the local directory is searched. If the `ISL_INCLUDE_PATH` environment variable is set, its value is prepended to the path specified with setIncludePath().

setParameterColor()

virtual int setParameterColor (int *param*, float *r*, float *g*, float *b*, float *a*);

Sets the value of the parameter of index *param* to *r*, *g*, *b*, and *a*. If *param* does not index a parameter of type `islShader::PARAMETER_COLOR`, -1 is returned; otherwise 0 is returned.

setParameterFloat()

virtual int setParameterFloat (int *param*, float *val*);

Sets the value of the parameter whose index is *param* to *val*. If *param* does not index a parameter of type `islShader::PARAMETER_FLOAT`, -1 is returned; otherwise 0 is returned.

setParameterMatrix()

virtual int setParameterMatrix (int *param*, const float* *val*);

Sets the value of the parameter whose index is *param* to *val*. The matrix should be an array of 16 floating point values given in column-major form (as in OpenGL). If *param* does not index a parameter of type `islShader::PARAMETER_MATRIX`, -1 is returned; otherwise 0 is returned.

setParameterString()

virtual int setParameterString (int *param*, const char* *val*);

Sets the value of the parameter of index *param* to *val*. The string is copied, so storage an application has allocated for *val* may be freed. If *param* does not index a parameter of type `islShader::PARAMETER_STRING`, -1 is returned; otherwise 0 is returned.

setShader()

virtual int setShader (const char* *shader*);

Sets the islShader shader string. The *shader* argument is a string that contains a shader written in the Interactive Shading Language. This string is parsed immediately, and its parameters and name are extracted and can be queried by an application. Any parameters existing in the islShader before the call to setShader() are deleted along with their associated values. The *shader* string is copied, so storage an application has allocated for *shader* may be freed. Returns -1 if an error condition has occurred; otherwise returns 0.

setShaderMatrix()

virtual void setShaderMatrix (const float* *matrix*);

Sets the islShader shader matrix. The matrix is an array of 16 floating point values given in column-major form (as in OpenGL). This specifies the value of the variable shadermatrix for this shader. The value defaults to the identity matrix.

ENVIRONMENT VARIABLES

The setShader() method considers the `ISL_INCLUDE_PATH` environment variable. If set, this environment variable is prepended to the path specified with setIncludePath().

SEE ALSO

islAppearance, islCompileAction, islDrawAction, islError, islShader, islShape

sgi

NAME

islMemory - OpenGL Shader memory manager class

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Allocator/Deallocator specification and construction

islMemory (NewFunc *nfn*, DeleteFunc *dfn*);
~islMemory ( );

Setting and getting al/deallocator functions

NewFunc getNewFunc ( ) const;
DeleteFunc getDeleteFunc ( ) const;

CLASS DESCRIPTION

The islMemory class provides an interface for user-defined memory allocator/deallocator functions. These methods are used by certain classes throughout the libraries to place objects at user-defined locations. For example, this may be useful to place objects in shared-memory.

METHOD DESCRIPTIONS

islMemory()

islMemory (NewFunc *nfn*, DeleteFunc *dfn*);

The islMemory constructor takes two arguments which specify the allocator and deallocator which objects requiring an islMemory will use.

nfn is the allocator function pointer and must perform an allocation of

```
size_t
```

bytes when invoked. It's signature is:

```
void *(*NewFunc)(size_t);
```

nfn is the deallocator function pointer and must perform a deallocation of the specified

```
void *
```

when invoked. This deallocation must be symmetric with that performed in *nfn* or undefined results will occur:

```
typedef void (*DeleteFunc)(void*);
```

A reasonable replacement pair of new/delete functions would allocate a large chunk of memory then return sequential smaller pieces of the large chunk, to reduce the overhead of frequent small allocations.

~islMemory()

~islMemory ( );

Destructor.

getDeleteFunc()

DeleteFunc getDeleteFunc ( ) const;

Returns the deallocator function pointer.

getNewFunc()

NewFunc getNewFunc ( ) const;

Returns the allocator function pointer.

SEE ALSO

islMemory

NAME

islShape - OpenGL Shader Interactive Shading Language shape class

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islShape (void);
virtual ~islShape (void);

Setting and getting appearance

virtual void setAppearance (islAppearanceBase* *appearance*);
virtual islAppearanceBase* getAppearance (void) const;

Setting and getting geometry information

virtual void setGeometryData (void* *geometry_data*);
virtual void* getGeometryData (void) const;
virtual void setDrawGeometryFunc (DrawGeometryFunc *draw*);
virtual DrawGeometryFunc getDrawGeometryFunc (void) const;

Setting and getting screen space bound

virtual void setScreenBound (int *x*, int *y*, int *h*, int *w*);
virtual void setScissorScreenBound (ScreenBound *box*);
virtual void getScissorScreenBound (ScreenBound *box*) const;
virtual void getScreenBound (int& *x*, int& *y*, int& *w*, int& *h*) const;

Setting and getting object bounds on the screen

virtual void setObjectScreenBound (unsigned int *num_boxes*, ScreenBound* *boxes*);
virtual unsigned int getObjectScreenBound (unsigned int *num_boxes*, ScreenBound* *boxes*);

CLASS DESCRIPTION

The islShape class object provides an interface to associate an islAppearance class object with geometry retained by an application. The geometry is provided as a data pointer, an application callback that draws the data, and a set of screen space bounding boxes. The draw callback is of type `islShape::DrawGeometryFunc`:

```
int (*DrawGeometryFunc)(unsigned int num_tex,
                        const float* texcoords,
                        void* geometry_data);
```

The argument gives the number of multi-texture units that are active for this drawing pass. The maximum number of texture units to use is determined (when an appearance is compiled) based on the maximum number of texture units available and the number of texture units to use, as provided in the `ISL_GL_TEXTURE_UNITS` environment variable. If you may run on multi-texture capable hardware, but do not have a multi-texture capable DrawGeometryFunc, you can override the automatic hardware capability detection by setting the environment variable, `ISL_GL_TEXTURE_UNITS` to a value of 1 before creating the islCompileAction. See islCompileAction for details, and see below for further details.

The argument *texcoords* is an array of texture coordinate generation modes. There is one element in the array for each active multi-texture unit. Each element of the array specifies the texture coordinate generation mode for the corresponding texture unit. The values of *texcoords* can be interpreted as follows:

- -1: glTexGen has already been set for texture projection. no user texture coordinates should be set.
- -2: glTexGen has already been set for environment mapping. no user texture coordinates should be set.
- -3: glTexGen has already been set for use of a varying variable. no user texture coordinates should be set.
- -4: the corresponding texture unit is not in use. no user texture coordinates should be set.
- -5: the user must load the normals as texture coordinates.
- 0: normal drawing. user should load standard surface texture coordinates.
- >0: value set by a texture ISL operation with a user-defined pass-through variable. In this case, the corresponding element of *texcoords* takes on the value of the pass-through variable. This variable typically is used to select among texture coordinates that are computed by the application but can be used for any purpose.

The argument *geometry_data* passed to the callback is the unmodified geometry data given to the islShape with setGeometryData(). The callback should only draw the geometry and not set any OpenGL appearance state (such as current texture, framebuffer blend modes, and current color). It must restore any allowed OpenGL state it sets in the process of drawing the data (such as the modelview or projection matrices). The geometry must include texture coordinates if any shaders have texture ISL operations (indicated by the presence of a *texcoords* argument with elements greater than or equal to 0) and must include normal vectors if any shaders have diffuse or specular ISL operations. However, if an application also requests the normals in a texture coordinate set (*texcoords* argument equal to -5) for any texture unit, the normals (specified via any glNormal* coall) should be omitted. The presence of this argument implies that we are doing lighting wholly in fragment-hardware, and that base lighting will be Phong lighting, and that most lighting be fully hardware accelerated.

The DrawGeometryFunc function should return 0 if successful; otherwise it should return -1.

In addition, an application must specify screen space bounding boxes to define active pixels during rendering. Screen bounds are defined using data of the type

```
int ScreenBound[4];
```

where the four elements of the ScreenBound array are `{starting_x, starting_y, width, height}`. All pixel operations performed on the islShape by the ISL Library are scissored to this area. The screen space bounding box does not necessarily have to cover the entire object. For example, it can be used to tile the rendering of a single object for the purpose of load balancing or distribution using code of the form:

```
islDrawAction* drawaction = new islDrawAction();
islShape::ScreenBound ul = {0, 0,64,64}, ur = {64, 0,64,64},
islShape::ScreenBound ll = {0,64,64,64}, lr = {64,64,64,64},
shape->setScissorScreenBound(ul);
drawaction->draw(shape);
shape->setScissorScreenBound(ur);
drawaction->draw(shape);
shape->setScissorScreenBound(ll);
drawaction->draw(shape);
shape->setScissorScreenBound(lr);
drawaction->draw(shape);
```

If the geometry stored in `shape` spanned the (0,0)-(128,128) range, the code above would draw it in four separate pieces.

The application may also supply a list of tighter screen space bounding boxes for the actual geometry using setObjectScreenBound(). These boxes are used in pixel and texture block copy operations. Looser bounds will not affect appearance, but may affect performance.

The ISL Library

The OpenGL Shader Interactive Shading Language Library provides a minimal interface for supporting interactive, programmable shading. The ISL Library consists of six classes that enable an application to define an appearance consisting of ISL shaders, compile that appearance into an OpenGL stream, associate the compiled appearance with geometry from the application, and, subsequently, to render the shaded geometry to an OpenGL rendering context opened by the application.

The appearance is specified through an islAppearance class object, which contains a list of active ambient light shaders, a list of active distant light shaders, a list of active local light shaders, and a list of surface shaders. Each of these shaders is contained in an islShader class object. An islAppearance is compiled into a stream of OpenGL commands held inside the ISL Library using an islCompileAction.

The compilation will take advantage of capabilities available on the current graphics hardare. It is possible to override the automatic capability detection through a set of environment variables: ISL_GL_VENDOR, ISL_GL_RENDERER, ISL_GL_VERSION, ISL_GL_EXTENSIONS, and ISL_GL_TEXTURE_UNITS. Some of these can be useful to override the hardware queries and lower limits on certain capabilities, but true hardware limits are ultimately respected. If a user specifies an override of 8 for texture units when only 4 exist, the true capability of 4 will be respected. Examples of these environment variables which can be overridden:

- ISL_GL_TEXTURE_UNITS: Override the hardware texture unit count. This is useful if you are running on multi-texture capable hardware, but do not have a multi-texture capable DrawGeometryFunc for your islShape <\item>
- ISL_GL_TEXTURE_UNITS: Override the hardware texture unit count. This is useful if you are running on multi-texture capable hardware, but do not have a multi-texture capable DrawGeometryFunc for your islShape <\item>
- ISL_GL_TEXTURE_UNITS: Override the hardware texture unit count. This is useful if you are running on multi-texture capable hardware, but do not have a multi-texture capable DrawGeometryFunc for your islShape <\item>
- ISL_GL_ARBFP_LIMITS: Override the hardware fragment program limits. The space-delimited values in this list correspond to the following glGetParameterivARB query tokens, in-order:

  GL_MAX_PROGRAM_INSTRUCTIONS_ARB GL_MAX_PROGRAM_ALU_INSTRUCTIONS_ARB
  GL_MAX_PROGRAM_TEX_INSTRUCTIONS_ARB
  GL_MAX_PROGRAM_TEX_INDIRECTIONS_ARB
  GL_MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB
  GL_MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB
  GL_MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB
  GL_MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB
  GL_MAX_PROGRAM_TEMPORARIES_ARB GL_MAX_PROGRAM_NATIVE_TEMPORARIES_ARB
  GL_MAX_PROGRAM_PARAMETERS_ARB GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB
  GL_MAX_PROGRAM_ENV_PARAMETERS_ARB
  GL_MAX_PROGRAM_NATIVE_PARAMETERS_ARB GL_MAX_PROGRAM_ATTRIBS_ARB
  GL_MAX_PROGRAM_NATIVE_ATTRIBS_ARB

  An example of these limits might be set using:

```
setenv ISL_GL_ARBFP_LIMITS "94 63 31 4 96 64 32 4 16 32 32 32 32 32 10 10"
```

Application geometry is associated with the appearance through an islShape class object. The geometry is defined simply as a pointer to data and an associated user callback, which the application provides for delivering this data to the graphics pipeline. The appearance is a pointer to an islAppearance. An islShape class object can be rendered into the current OpenGL context with an islDrawAction. A simple example of drawing red geometry is shown below:

```
islShader* shader = new islShader();
shader->setShader("surface myshader() { FB = color(1,0,0,1); }");

islAppearance* appearance = new islAppearance();
appearance->pushShader(islAppearance::SURFACE_LIST, shader);

// for multi-texture capable hardware where we don't provide
// a multi-texture DrawGeometryFunc to the islShape (see below)
putenv("ISL_GL_TEXTURE_UNITS=1");
islCompileAction* compileaction = new islCompileAction();
compileaction->compile(appearance);

islShape* shape = new islShape();
shape->setAppearance(appearance);
shape->setDrawGeometryFunc(user_drawcallback);
shape->setGeometryData((void*)user_data);

islDrawAction* drawaction = new islDrawAction();
drawaction->draw(shape);
```

It is the responsibility of the application to compile the appearance when necessary (if, for example, the shaders have changed or the shader parameters have changed). It is also the responsibility of the application to ensure there are no OpenGL state collisions between the ISL Library and its own implementation. The ISL Library sets state only in the application of an islDrawAction. The islDrawAction restores all state to its original settings before returning, however it assumes most OpenGL state is set to its default values when the draw action is applied. The islDrawAction depends on the application properly setting the `glViewport` and `GL_PROJECTION_MATRIX`; these are read from the OpenGL state and possibly used during the draw action. Any errors during shader parsing, compiling, or drawing are trapped and can be queried with the help of the islError class. There is a minor typing incompatibility between the versions of the standard template library provided with the MipsPro version 7.2 compilers and the 7.3 compilers. The OpenGL Shader ISL Library on IRIX is built with the 7.3 version compilers, but with compatibility options set to mimic the 7.2 STL types to allow use with either compiler version. If you are using the newer 7.3 compilers, you must `#define ::STL_USE_SGI_ALLOCATORS` and `STL_SGI_THREADS` **before** including isl.h in the files that directly use the OpenGL Shader API, or you can define these symbols using compiler flags. For example, using something like the following in a Makefile:

```
# these flags are required to build with version 7.3 of the
# MipsPro Compilers; they are ignored on version 7.2.1
LC++DEFS += -D::STL_USE_SGI_ALLOCATORS -DSTL_SGI_THREADS
```

These preprocessor symbols are ignored by the 7.2.1 standard template library headers, so code which may be compiled with either the 7.2.1 or 7.3 MipsPro compilers can safely define them in both cases.

METHOD DESCRIPTIONS

islShape()

islShape (void);

Constructs a new islShape.

~islShape()

virtual ~islShape (void);

Destroys the islShape.

getAppearance()

virtual islAppearanceBase* getAppearance (void) const;

Returns the appearance of the islShape.

getDrawGeometryFunc()

virtual DrawGeometryFunc getDrawGeometryFunc (void) const;

Returns the single texture geometry draw function of the islShape.

getGeometryData()

virtual void* getGeometryData (void) const;

Returns the geometry data of the islShape.

getObjectScreenBound()

virtual unsigned int getObjectScreenBound (unsigned int *num_boxes*, ScreenBound* *boxes*);

Gets the list of non-overlapping screen space bounding boxes for the geometry. The *num_boxes* argument gives the number of boxes that have been allocated in the *boxes* array. The return value from getObjectScreenBound is the number of boxes actually used. Each element of the *boxes* array is set to the holds the position and size of one bounding box. If *num_boxes* is less than the total number of actual boxes used, only information on the first *num_boxes* boxes will be placed in the *boxes* array, though the total number of boxes in use will still be returned.

getScissorScreenBound()

virtual void getScissorScreenBound (ScreenBound *box*) const;

Use of this function is depricated, use getScissorScreenBound instead

getScreenBound()

virtual void getScreenBound (int& *x*, int& *y*, int& *w*, int& *h*) const;

Fill *box* with the screen space bounding box of the geometry.

setAppearance()

virtual void setAppearance (islAppearanceBase* *appearance*);

Sets the appearance of the islShape to *appearance*.

setDrawGeometryFunc()

virtual void setDrawGeometryFunc (DrawGeometryFunc *draw*);

Sets the geometry draw callback of the islShape to *draw*. This function is used by an islDrawAction to request that the application draw the geometry specified with setGeometryData().

setGeometryData()

virtual void setGeometryData (void* *geometry_data*);

Sets the geometry data of the islShape to *geometry_data*. This data may be drawn with a function of type `islShape::DrawGeometryFunc`.

setObjectScreenBound()

virtual void setObjectScreenBound (unsigned int *num_boxes*, ScreenBound* *boxes*);

Sets a list of non-overlapping screen space bounding boxes for the geometry. The *num_boxes* argument gives the number of boxes contained in the list. Each element of the *boxes* array contains an array of four integer pixel coordinates: `{left, bottom, width, height}`. The parameters are identical in order and interpretation to those for `glCopyPixels`. If setObjectScreenBound is never called, the single box given by `islShape::setScissorScreenBound` is used. However, at least one of setObjectScreenBound or setScissorScreenBound must be used.

setScissorScreenBound()

virtual void setScissorScreenBound (ScreenBound *box*);

Sets the screen space bounding box of the geometry. *box* is an array `(int[4])` containing the lower left corner x, lower left corner y, box width and box height. These elements are identical in meaning and order to the arguments for `glCopyPixels`. If setScissorScreenBound is never called, the total bounding box of all object screen bounding boxes is used (see setObjectScreenBound) However, at least one of setObjectScreenBound or setScissorScreenBound must be used.

setScreenBound()

virtual void setScreenBound (int *x*, int *y*, int *h*, int *w*);

Use of this function is depricated, use setScissorScreenBound instead

SEE ALSO

islAppearance, islCompileAction, islDrawAction, islError, islShader, islShape

---

NAME

islSnapshotAction - OpenGL Shader appearance snapshot class

HEADER FILE

#include <shader/isl.h>

PUBLIC METHOD SUMMARY

Construction and destruction

islSnapshotAction (islMemory* *mm*=NULL);
virtual ~islSnapshotAction (void);

Setting and getting texture freezing information

virtual void setSnapshotTextureData (void* *user_data*);
virtual void* getSnapshotTextureData (void) const;
virtual void setSnapshotTextureFunc (SnapshotTextureFunc *snapshot_texture*);
virtual SnapshotTextureFunc getSnapshotTextureFunc (void) const;

Methods to manage snapshots

virtual islAppearanceSnapshotData* snapshot (const islAppearance*) const;
virtual islAppearanceSnapshotData* snapshot (const islAppearanceCopy*) const;
virtual void deleteSnapshot (islAppearanceSnapshotData*) const;

CLASS DESCRIPTION

The islSnapshotAction class provides an interface for freezing the current run-time parameter settings for a islAppearance, for later use in a islAppearanceSnapshot.

It is not necessary to snapshot an appearance before use, and in a single-thread/single-processor application the combination of taking a snapshot and rendering the resulting frozen appearance will almost certainly be more expensive than just rendering the original appearance. Taking a snapshot of an appearance offers two benefits for multi-threaded applications. First, the snapshot mechanism allows parameter changes in one thread while rendering a previously snapped appearance in another thread. Second, the snapshot representation for a shader can be allocated using a user-provided allocator, allowing it to be allocated in shared memory if desired.

Allocation of islAppearanceSnapshotData

The islAppearanceSnapshotData created by

snapshot() and deleted by , can be allocated by specifying an to the islSnapshotAction constructor. If no islMemory is specified, a default islMemory will be used.

Texture Tracking

During the snapshot process, if provided, a callback function of type `islSnapshotAction::SnapshotTextureFunc` will be called for each external texture that will be used given the current parameter settings for the shader.

```
int (*SnapshotTextureFunc)(const char* name, float texgen_code,
    void* user_data);
```

name is the texture name, *texgen_code* is the texture 'code' for this texture call (see islDrawAction for more details on texgen codes), and *user_data* is an arbitrary data pointer that can be used by the callback. This callback can be used to tell which textures and texture generation codes will be used by the shader, given the current parameter settings. It is called for each texture use, so may be called multiple times for the same texture.

METHOD DESCRIPTIONS

islSnapshotAction()

islSnapshotAction (islMemory* *mm*=NULL);

Constructs a new islSnapshotAction. The islMemory object argument, if specified, will be used for allocating and freeing all memory used by the snapshot process. If no islMemory is specified (or NULL is specified) a default allocator will be used.

~islSnapshotAction()

virtual ~islSnapshotAction (void);

Destroys the islSnapshotAction. Does not delete any previously allocated snapshots that were not explicitly deallocated by calls to deleteSnapshot().

deleteSnapshot()

virtual void deleteSnapshot (islAppearanceSnapshotData*) const;

Delete memory associated with frozen appearance.

getSnapshotTextureData()

virtual void* getSnapshotTextureData (void) const;

Gets the pointer to user data that is passed through to the `islSnapshotAction::LoadTextureFunc` callback function.

getSnapshotTextureFunc()

virtual SnapshotTextureFunc getSnapshotTextureFunc (void) const;

Returns the pointer to the current SnapshotTextureFunc callback function.

setSnapshotTextureData()

virtual void setSnapshotTextureData (void* *user_data*);

Sets a pointer to user data that is passed through to the `islSnapshotAction::SnapshotTextureFunc` callback function. The data is unmodified by the islSnapshotAction.

setSnapshotTextureFunc()

virtual void setSnapshotTextureFunc (SnapshotTextureFunc *snapshot_texture*);

Sets a pointer to an `islSnapshotAction::SnapshotTextureFunc` callback function. If this function is not specified, loading of textures is ignored entirely by the islSnapshotAction. The texture uses will still exist in the frozen shader, but islSnapshotAction does no tracking beyond calling the SnapshotTextureFunction

snapshot()

virtual islAppearanceSnapshotData* snapshot (const islAppearance*) const;

Snapshot the islAppearance. Returns a pointer an object of the islAppearanceSnapshotData class representing the 'snapped' appearance. Returns 0 if an error condition has occurred.

snapshot()

virtual islAppearanceSnapshotData* snapshot (const islAppearanceCopy*) const;

Snapshot the islAppearanceData. Returns a pointer an object of the islAppearanceSnapshotData class representing the 'snapped' appearance. Returns 0 if an error condition has occurred.

SEE ALSO

islAppearance, islAppearanceData, islAppearanceSnapshot, islAppearanceSnapshotData, islDrawAction, islMemory, islSnapshotAction

---

sgi

NAME

isl::TexGen::copyNormToTex - OpenGL Shader TexGen Function: isl::TexGen::copyNormToTex

INHERITS FROM

isl::VertexShader

HEADER FILE

#include <shader/islvertexfn.h>

PUBLIC METHOD SUMMARY

virtual void init (void);
virtual void run (void);

INHERITED PUBLIC METHODS

Inherited from isl::VertexShader

inline VertexContext* getContext ( ) const;
virtual void init (void);
virtual void run (void);
inline void setContext (VertexContext* *cc*);

CLASS DESCRIPTION

The isl::TexGen::copyNormToTex class is a publically derived class of type `isl::VertexShader` which implements the texgen functionality, as it's name implies, of copying the current normal to the current texture.

To use this particular texgen mode, create an instance, and pass it to all isl::VertexContexts which are being used to draw geometry with this texture generation mode.

METHOD DESCRIPTIONS

init()

virtual void init (void);

Executes shader initialization. See `isl::VertexShader` for details.

run()

virtual void run (void);

Executes per-vertex computation. See `isl::VertexShader` for details.

SEE ALSO

isl::VertexContext, isl::VertexShader

NAME

isl::TexGen::copyPosToTex - OpenGL Shader TexGen Function: isl::TexGen::copyPosToTex

INHERITS FROM

isl::VertexShader

HEADER FILE

#include <shader/islvertexfn.h>

PUBLIC METHOD SUMMARY

virtual void init (void);
virtual void run (void);

INHERITED PUBLIC METHODS

Inherited from isl::VertexShader

inline VertexContext* getContext ( ) const;
virtual void init (void);
virtual void run (void);
inline void setContext (VertexContext* *cc*);

CLASS DESCRIPTION

The isl::TexGen::copyPosToTex class is a publically derived class of type `isl::VertexShader` which implements the texgen functionality, as it's name implies, of copying the current vertex to the current texture.

To use this particular texgen mode, create an instance, and pass it to all `isl::VertexShader` which are being used to draw geometry with this texture generation mode.

METHOD DESCRIPTIONS

init()

virtual void init (void);

Executes shader initialization. See `isl::VertexShader`

run()

virtual void run (void);

Executes per-vertex computation. See `isl::VertexShader` for details.

SEE ALSO

isl::VertexShader

sgi

NAME

isl::TexGen::tangentSpaceAxis - OpenGL Shader TexGen Function: isl::TexGen::tangentSpaceAxis

INHERITS FROM

isl::VertexShader

HEADER FILE

#include <shader/islvertexfn.h>

PUBLIC METHOD SUMMARY

void setAxis (ISLcolor *aa*);
ISLcolor getAxis ( );
virtual void init (void);
virtual void run (void);

PROTECTED MEMBER SUMMARY

ISLfloat _axis[4];

ISLfloat _binormal[4];

ISLfloat _tangent[4];

INHERITED PUBLIC METHODS

Inherited from isl::VertexShader

inline VertexContext* getContext ( ) const;
virtual void init (void);
virtual void run (void);
inline void setContext (VertexContext* *cc*);

CLASS DESCRIPTION

The isl::TexGen::tangentSpaceAxisclass is a publically derived class of type `isl::VertexShader` which implements the texgen functionality, as it's name implies, of copying the current normal to the current texture.

To use this particular texgen mode, create an instance, and pass it to all isl::VertexContexts which are being used to draw geometry with this texture generation mode.

METHOD DESCRIPTIONS

getAxis()

ISLcolor getAxis ( );

This function returns the current axis used to generate tangent-space.

init()

virtual void init (void);

Executes shader initialization. See `isl::VertexShader` for details.

run()

virtual void run (void);

Executes per-vertex computation. See `isl::VertexShader` for details.

setAxis()

void setAxis (ISLcolor *aa*);

Used to specify a particular axis from which to generate tangent-space. Both a tangent and binormal are generated, when combined with the normal, define a coordinate space at each vertex. The tangent and binromal are computed as:

```
Vtangent  = Normal cross Vtangent;
Vbinormal = Normal cross aa;
```

MEMBER DESCRIPTIONS

_axis[4]

ISLfloat _axis[4];

Storage for the specified axis vector used in generation of the tangent-space.

_binormal[4]

ISLfloat _binormal[4];

Storage for the computed binormal vector.

_tangent[4]

ISLfloat _tangent[4];

Storage for the computed tangent vector.

SEE ALSO

isl::VertexContext, isl::VertexShader

NAME

isl::Texture::ClearCoat360 - OpenGL Shader ClearCoat360 Texture

INHERITS FROM

isl::Texture::Image

HEADER FILE

#include <shader/isltexture.h>

PUBLIC METHOD SUMMARY

bool loadPaint (const std::string&);
void setViewMatrix (const float* *vm*);
virtual bool compute ( );
void restoreState ( );

INHERITED PUBLIC METHODS

Inherited from isl::Texture::Image

virtual bool compute ( );
int getDepth ( );
unsigned char* getDstImg ( ) const;
int getHeight ( );
int getNumChannels ( );
unsigned char* getSrcImg ( ) const;
int getWidth ( );
void setDstImg (unsigned char* *dst*);
void setImgDims (int *ww*, int *hh*, int *dd*=1);
void setSrcImg (unsigned char* *src*);

CLASS DESCRIPTION

The isl::Texture::ClearCoat360 class creates an 360 degree environment reflection map, based on a previously captured or simulated paint simulation. This class allows ClearCoat360 paints to be used by an islShader.

METHOD DESCRIPTIONS

compute()

virtual bool compute ( );

Computes the view-dependent texture environment. Requires complete access to the framebuffer to do this, and overwrites current contents of the framebuffer. This call completely manages all state necessary for the texture to be computed correctly. The framebuffer must be greater than or equal to the width and height of the paint (getWidth(), getHeight()) for the resultant environment map to be properly calculated and sized.

After compute() is called the image should copied into a texture for subsequent application as an environment map. Use the most efficient texture copy method available for your platform. InfiniteReality performs very well with glCopyTexSubImage2D on an existing texture, for instance.

After the texture has been extracted from the framebuffer, the context state should be returned to it's pre-compute() state with restoreState().

To summarize, there are several steps in creating and computing an isl::Texture::ClearCoat360:

1. Create a new ClearCoat360 object (ClearCoat360()).
2. Load a paint (loadPaint()).
3. Render the paint to a buffer (compute()).
4. Extract the image from the buffer (glCopyTexSubImage2D or equivalent).
5. Restore the buffer state (restoreState()).

The same steps, in pseudo-code:

```
using namespace isl::Texture;

isl::Texture::ClearCoat360 *cctex = new isl::Texture::ClearCoat360;

bool loaded = cctex->loadPaint( "paint.cc360" );
if ( loaded == false )
{
  cerr << "couldn't load cc360 paint. exiting." << endl;
  exit( -1 );
}

cctex->compute();

glBindTexture( GL_TEXTURE_2D, application_allocated_texture_obj );
glCopyTexSubImage2D( GL_TEXTURE_2D, 0,
                     0, 0,
                     0, 0,
                     cctex->getWidth(), cctex->getHeight() );

cctex->restoreState();
```

loadPaint()

bool loadPaint (const std::string&);

This method will attempt to load the named .cc360 paint file at the path specified. Returns true or false for success or failure.

Paints may only be loaded when the GL context in which they will be used is current. This restriction is due to texture binding done in the load process. Textures and texture names are not shared across pipes, so this further requires that each pipe in which a ClearCoat360 texture is used have a v new instance of a particular isl::Texture::ClearCoat360 created and loaded.

restoreState()

void restoreState ( );

Returns the state to it's previous setting after a has been issued. May only be called after a or the GL context will be in an indeterminate state.

setViewMatrix()

void setViewMatrix (const float* *vm*);

This method sets the view matrix from which the resultant ClearCoat360 environment texture is calculated.

SEE ALSO

islShader, isl::Texture::Image

---

sgi

NAME

isl::Texture::Fresnel - OpenGL Shader Fresnel Texture

INHERITS FROM

isl::Texture::Image

HEADER FILE

#include <shader/isltexture.h>

PUBLIC METHOD SUMMARY

void setIndexOfRefraction (float *idx*);
void setContrastScaleBias (float *ss*, float *bb*);
virtual bool compute ( );

INHERITED PUBLIC METHODS

Inherited from isl::Texture::Image

virtual bool compute ( );
int getDepth ( );
unsigned char* getDstImg ( ) const;
int getHeight ( );
int getNumChannels ( );
unsigned char* getSrcImg ( ) const;
int getWidth ( );
void setDstImg (unsigned char* *dst*);
void setImgDims (int *ww*, int *hh*, int *dd*=1);
void setSrcImg (unsigned char* *src*);

CLASS DESCRIPTION

The isl::Texture::Fresnel class creates a fresnel refraction map from an input environment (sphere) map. The resultant blend parameters are stored in the alpha channel of the destination image. isl::Texture::Fresnel uses the image set/query methods from isl::Texture::Image. This implementation is derived from the original SGI ClearCoat implementation, and can be used to achieve identical effects.

METHOD DESCRIPTIONS

compute()

virtual bool compute ( );

Computes the fresnel map, using the source image and storing results in the alpha-channel of the destination image.

setContrastScaleBias()

void setContrastScaleBias (float *ss*, float *bb*);

Sets a contrast enhancement scale and bias to the results.

setIndexOfRefraction()

void setIndexOfRefraction (float *idx*);

Sets the index of refraction to use in computing the map. A value of 1.8 is used by default, which is approximately that of a polyurethane.

SEE ALSO

isl::Texture::Image

NAME

isl::Texture::Image - OpenGL Shader Texture Generation Base class

HEADER FILE

#include <shader/isltexture.h>

PUBLIC METHOD SUMMARY

int getNumChannels ( );
int getWidth ( );
int getHeight ( );
int getDepth ( );
void setImgDims (int *ww*, int *hh*, int *dd*=1);
void setSrcImg (unsigned char* *src*);
void setDstImg (unsigned char* *dst*);
unsigned char* getDstImg ( ) const;
unsigned char* getSrcImg ( ) const;
virtual bool compute ( );

CLASS DESCRIPTION

The isl::Texture::Image class defines the base class for dynamically calculated textures within the ISL framework. isl::Texture::Image is a pure virtual class and therefore cannot be directly instantiated. isl::Texture::Image is instead a template providing base image sizing and depth functionality and requires it's derived classes to supply the compute() method.

METHOD DESCRIPTIONS

compute()

virtual bool compute ( );

Method to be supplied by any derived class. Do any/all texture generation work here.

getDepth()

int getDepth ( );

Returns the depth of the computed image in pixels.

getDstImg()

unsigned char* getDstImg ( ) const;

Returns a pointer to the currently set destination image.

getHeight()

int getHeight ( );

Returns the height of the computed image in pixels.

getNumChannels()

int getNumChannels ( );

Returns the number of channels in the computed image.

getSrcImg()

unsigned char* getSrcImg ( ) const;

Returns a pointer to the currently set source image.

getWidth()

int getWidth ( );

Returns the width of the computed image in pixels.

setDstImg()

void setDstImg (unsigned char* *dst*);

Sets the pointer to the memory in which the destination image will be stored. Must be allocated by the user and be of at least

```
width*height*num_channels
```

in extents.

setImgDims()

void setImgDims (int *ww*, int *hh*, int *dd*=1);

Configures the width, height, and optional depth of the computed image, in pixels. Both the source and destination images must be of this size.

setSrcImg()

void setSrcImg (unsigned char* *src*);

Sets the pointer to the memory in which the source image (if any) is stored.

---

sgi

NAME

isl::Texture::Noise - OpenGL Shader Noise Texture

INHERITS FROM

isl::Texture::Image

HEADER FILE

#include <shader/isltexture.h>

PUBLIC METHOD SUMMARY

void setSeed (unsigned int *seed*);
unsigned int getSeed ( );
virtual bool compute ( );

INHERITED PUBLIC METHODS

Inherited from isl::Texture::Image

virtual bool compute ( );
int getDepth ( );
unsigned char* getDstImg ( ) const;
int getHeight ( );
int getNumChannels ( );
unsigned char* getSrcImg ( ) const;
int getWidth ( );
void setDstImg (unsigned char* *dst*);
void setImgDims (int *ww*, int *hh*, int *dd*=1);
void setSrcImg (unsigned char* *src*);

CLASS DESCRIPTION

The isl::Texture::Noise class creates a noise texture using the Perlin noise technique. Noise textures of this sort are simply scalar values at any point in the texture into which these are computed, so luminance textures are enough to capture the entirety of the noise calculated by this class.

METHOD DESCRIPTIONS

compute()

virtual bool compute ( );

Computes the noise map.

getSeed()

unsigned int getSeed ( );

Returns the seed currently used by the random number generator.

setSeed()

void setSeed (unsigned int *seed*);

Sets the seed to the random number generator. All noise generated from a particular seed will be identical.

SEE ALSO

isl::Texture::Image

---

sgi

NAME

isl::VertexContext - OpenGL Shader Vertex Shader Context class

HEADER FILE

#include <shader/islvertex.h>

PUBLIC METHOD SUMMARY

VertexShaders configuration methods

void enable (ProgramType *pp*);
void disable ( );
ProgramType typeProgram ( ) const;
void setVertexShader (VertexShader* *fn*);
VertexShader* getVertexShader ( );
void init ( );

Light context methods

void extractLightPositions (islAppearance* *aa*);
const ISLvertexVector& getDistantLights ( ) const;
const ISLvertexVector& getLocalLights ( ) const;

Matrix context methods

void setModelviewMatrix (ISLmatrix *mv*);
void setProjectionMatrix (ISLmatrix *pp*);
inline ISLmatrix getModelviewMatrix ( ) const;
inline ISLmatrix getProjectionMatrix ( ) const;
inline ISLmatrix getInvModelviewMatrix ( ) const;
inline ISLmatrix getInvProjectionMatrix ( ) const;

Vertex Array data set methods

void setTexCoordPointer (GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);
void setNormalPointer (GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);
void setColorPointer (GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);
void setVertexPointer (GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);
void setIndexPointer (GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);

Vertex Array draw & calculate methods

void drawElements (GLenum *mode*, GLsizei *count*, GLenum *type*, const GLvoid* *indices*);
void drawArrays (GLenum *mode*, GLint *first*, GLsizei *count*);

Per-component data get methods

inline ISLvertex getNormal ( );
inline ISLvertex getVertex ( );
inline ISLvertex getColor ( );

inline ISLvertex getTexCoord (const int *ii*);

Per-component data get methods

inline ISLvertex getNormalResult ( );
inline ISLvertex getVertexResult ( );
inline ISLvertex getColorResult ( );
inline ISLvertex getTexCoordResult (const int *ii*);

Per-component data set methods

inline void setNormal3f (GLfloat *xx*, GLfloat *yy*, GLfloat *zz*);
inline void setTexCoord2f (GLfloat *ss*, GLfloat *tt*);
inline void setMultiTexCoord2f (int *ii*, GLfloat *ss*, GLfloat *tt*);
inline void setColor3f (GLfloat *xx*, GLfloat *yy*, GLfloat *zz*);
inline void calcVertex3f (GLfloat *xx*, GLfloat *yy*, GLfloat *zz*);
inline void setVertex3f (GLfloat *xx*, GLfloat *yy*, GLfloat *zz*);

CLASS DESCRIPTION

VertexContext is the basis for run-time user computation of various vertex-based parameters. VertexContext provides a mapping between standard OpenGL state and programmable equivalents and also acts as a framework by which per-vertex computations are executed.

Vertex Programming Motivation

Per-vertex texture-coordinate calulation is necessary in a wide variety of scenarios. One common example is the sphere-map, a dynamically calculated set of texture coordinates which vary per-frame, using a normal as an index into a texture. Others include BRDFS, cube-maps, or any surface property dependent upon the combined positions of light, eye, and object vertex. Some of these techniques (sphere-map, cube-map, etc.) exist in dedicated OpenGL hardware today, but many require custom code to perform. OpenGL graphics hardware is increasingly supporting a large degree of custom programmability per-vertex, and the VertexContext and VertexShader classes are designed to expose that to the OpenGL Shader community, in a cross-platform, portable, and compatible fashion.

Though some graphics hardware currently supports custom vertex shading capability in hardware, not all does, nor do all support it in the same fashion. Therefore, to achieve program compatibility across hardware platforms, a software developer is faced with a set of painful choices. Either write custom programs for each platform or write to some lowest-common-denominator programmability set. However, another choice exists - similarly to the way a high-level shading language such as **ISL** provides an abstraction of hardware shading capabilties, so does the isl::VertexShader, for vertex shading.

OpenGL Shader currently supports c-language vertex programs and ARB_vertex_program programs.

VertexContexts are the essential mechanisms by which per-vertex texture generation and vertex processing occurs. Describing the details of how the particular functions operate follows, but first, a quick example, in a pseudo-application `draw()` will be presented to give an overview of how a context is used.

```
void drawFrame()
{
  // setup various matrices
  glMatrixMode( GL_MODELVIEW );
  glLoadMatrixf( mvm );

  glMatrixMode( GL_PROJECTION );
  glLoadMatrixf( pm );

  vertex_context->setModelviewMatrix( mvm );
```

```
vertex_context->setProjectionMatrix( pm );

// position the light
distantLightShader->setShaderMatrix( lmat );

vertex_context->extractLightPositions( shadedShape->getAppearance() );

// init the vertex shaders
vertex_context->init();

// exceute the drawaction
drawAction->draw( shadedShape );
}

void shadedShapeDraw()
{
  for( int ii=0; iisetTexCoord2f( tc[ii][0], tc[ii][1] );
    vertex_context->setNormal3f( nn[ii][0], nn[ii][1], nn[ii][2] );
    vertex_context->setVertex3f( vv[ii][0], vv[ii][1], vv[ii][2] );
  }
}
```

A complete example can be found in `/usr/share/shader/src/` in the `geometry` and `viewer_lib` directories.

The `ProgramType` enumerant is used througout isl::VertexContext to specify a particular shading mode. The value is one of:

- `isl::VertexShader::NONE`: No program is currently set to execute.
- `isl::VertexShader::TEXGEN`: When using texgen shaders, the isl::VertexContext will pass-through and render all parameters specified. These include, vertex, normal, color, and texture coordinate.
- `isl::VertexShader::VERTEX`: When using vertex shaders, the isl::VertexContext will pass-through and render some parameters specified. These include only vertex, color, and texture coordinates. It is the application's responsibility, as on any platform supporting vertex shaders, to ensure that the particular vtx shader written transforms vertices to clip-space. Further, as vertex shaders bypass the traditional lighting and transformation, any lighting calculations must be performed by the shader in use, and assigned per-vertex to it's color output.

A isl::VertexContext is used first to configure the environment in which geometry will be drawn, then to actually draw the geometry, executing the specified isl::VertexShaders.

To use the isl::VertexContext to generate texture coordinates, an application must replace it's usage of glTexCoordPointer, glVertexPointer, glNormalPointer, etc. with the following equivalent methods.

A isl::VertexContext can also be used to simply operate on the specified data, without actually rendering the geometry specified throug the various OpenGL vertex array APIs.

To use a isl::VertexContext and isl::VertexShader together, an application must modify existing OpenGL code which looks like:

```
glTexCoordPointer(3,GL_FLOAT,0,tri->_uv);
glNormalPointer(GL_FLOAT,0,tri->_n);
glVertexPointer(3,GL_FLOAT,0,tri->_v);
glDrawArrays(GL_TRIANGLE_STRIP,k,tri->_stripLength[i]);
```

to use isl::VertexContext code. Here, for example, we show using a previously-allocated context **vert_context** to issue the calls:

```
vert_context->setTexCoordPointer( 3,GL_FLOAT,0,tri->_uv );
vert_context->setNormalPointer( GL_FLOAT,0,tri->_n );
vert_context->setVertexPointer( 3,GL_FLOAT,0,tri->_v );
vert_context->drawArrays( GL_TRIANGLE_STRIP,k,tri->_stripLength[i] );
```

Notice that all the arguments remain identical. However, only `GL_FLOAT` data types are fully-supported at this time. Please submit any requests for other data formats to `shader-feedback@sgi.com`.

An alternate set of methods for specifying per-vertex data exist within VertexContext which parallel the single-component glNormal, glVertex, glColor, and glTexCoord functions.

As for the vertex array methods above, these methods are designed to be used in code such as:

```
glTexCoord2f( .56, .13 );
glNormal3f( 0, 0, 1 );
glVertex3f( 23.0, 14.0, 2.718 );
```

This code, when slightly modified, will then issue the same geometry, but execute the VertexContext's VertexShaders on each vertex. The above code is simply converted, when specified using a previously-allocated context `vert_context` to issue the calls, as follows:

```
vertex_context->setColor3f( .1, .3, .5 );
vertex_context->setTexCoord2f( .56, .13 );
vertex_context->setNormal3f( 0, 0, 1 );
vertex_context->setVertex3f( 23.0, 14.0, 2.718 );
```

METHOD DESCRIPTIONS

calcVertex3f()

inline void calcVertex3f (GLfloat *xx*, GLfloat *yy*, GLfloat *zz*);

Performs per-vertex isl::VertexShader operations, as specified to this isl::VertexContext. Does not render geometry through OpenGL pipeline - no results are displayed. Calculated results may subsequently be queried.

disable()

void disable ( );

Disable specified `ProgramType` *pp*.

drawArrays()

void drawArrays (GLenum *mode*, GLint *first*, GLsizei *count*);

Calculates & issues per-vertex operations, as specified to this . Issues results to OpenGL pipeline equivalently to glDrawArrays.

drawElements()

void drawElements (GLenum *mode*, GLsizei *count*, GLenum *type*, const GLvoid* *indices*);

Calculates & issues per-vertex isl::VertexShader operations, as specified to this isl::VertexContext. Issues results to OpenGL pipeline equivalently to glDrawElements.

enable()

void enable (ProgramType *pp*);

Enable specified `ProgramType` *pp*.

extractLightPositions()

void extractLightPositions (islAppearance* *aa*);

Extracts all distant and local light positions from the specified islAppearance. These positions are copied into distant and local light arrays, described below.

getColor()

inline ISLvertex getColor ( );

Returns the current input color. This is the preferred accesor for this data from within an . This data should not be overwritten.

getColorResult()

inline ISLvertex getColorResult ( );

Returns the color calculation result. This is the preferred accesor for this data from within an isl::VertexShader. This data should not be overwritten.

getDistantLights()

const ISLvertexVector& getDistantLights ( ) const;

Returns a reference to an **ISLvertexVector** containing the previously extracted distant light positions. The light vector is initialzed with values from computed by extractLightPositions.

`ISLvertexVector`s are simply `std::vector<ISLVertex>` typedefs, implemented as **STL** lists. For convenience, an `ISLvertexVectorIter` typedef is provided as well.

getInvModelviewMatrix()

inline ISLmatrix getInvModelviewMatrix ( ) const;

Returns the inverse modelview ISLmatrix for this isl::VertexContext.

getInvProjectionMatrix()

inline ISLmatrix getInvProjectionMatrix ( ) const;

Returns the inverse modelview ISLmatrix for this isl::VertexContext.

getLocalLights()

const ISLvertexVector& getLocalLights ( ) const;

Returns a reference to an **ISLvertexVector** containing the previously extracted local light positions. The light vector is initialzed with values from computed by extractLightPositions.

getModelviewMatrix()

inline ISLmatrix getModelviewMatrix ( ) const;

Returns the current modelview ISLmatrix for this isl::VertexContext.

getNormal()

inline ISLvertex getNormal ( );

Returns the current input normal. This is the preferred accesor for this data from within an . This data should not be overwritten.

getNormalResult()

inline ISLvertex getNormalResult ( );

Returns the normal calculation result. This is the preferred accesor for this data from within an . This data should not be overwritten.

getProjectionMatrix()

inline ISLmatrix getProjectionMatrix ( ) const;

Returns the current projection ISLmatrix for this isl::VertexContext.

getTexCoord()

inline ISLvertex getTexCoord (const int *ii*);

Returns the current input texcoord as specified by *ii*. This is the preferred accesor for this data from within an isl::VertexShader. This data may be overwritten by a user vertex shader.

getTexCoordResult()

inline ISLvertex getTexCoordResult (const int *ii*);

Returns the texcoord as specified by *ii* calculation result. This is the preferred accesor for this data from within an isl::VertexShader. This data may be overwritten by a user vertex shader.

getVertex()

inline ISLvertex getVertex ( );

Returns the current input vertex. This is the preferred accesor for this data from within an isl::VertexShader. This data should not be overwritten.

getVertexResult()

inline ISLvertex getVertexResult ( );

Returns the vertex calculation result. This is the preferred accesor for this data from within an . This data should not be overwritten.

getVertexShader()

VertexShader* getVertexShader ( );

Returns the current isl::VertexShader in use.

init()

void init ( );

Executes the current isl::VertexShader `init()` functions. This method should be called per-frame.

setColor3f()

inline void setColor3f (GLfloat *xx*, GLfloat *yy*, GLfloat *zz*);

Equivalent to glColor3f, though no OpenGL state is modified.

setColorPointer()

void setColorPointer (GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);

Equivalent to glColorPointer. Replace calls to glColorPointer to setColorPointer.

setIndexPointer()

void setIndexPointer (GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);

Equivalent to glIndexPointer. Replace calls to glIndexPointer to setIndexPointer.

setModelviewMatrix()

void setModelviewMatrix (ISLmatrix *mv*);

Sets the modelview matrix for this isl::VertexContext. Many shaders rely on this to do their work, so this must be set by the application for each object with a unique ShaderMatrix. This will ensure that each object being drawn is also shaded with it's corresponding modelview matrix. Computes the inverse of this matrix and stores it for subsequent queries.

setMultiTexCoord2f()

inline void setMultiTexCoord2f (int *ii*, GLfloat *ss*, GLfloat *tt*);

Equivalent to glMultiTexCoord2f, though no OpenGL state is modified. Only supported on platforms which support multitexture.

setNormal3f()

inline void setNormal3f (GLfloat *xx*, GLfloat *yy*, GLfloat *zz*);

Equivalent to glNormal3f, though no OpenGL state is modified.

setNormalPointer()

void setNormalPointer (GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);

Equivalent to glNormalPointer. Replace calls to glNormalPointer to setNormalPointer.

setProjectionMatrix()

void setProjectionMatrix (ISLmatrix *pp*);

Sets the projection matrix for this isl::VertexContext. Computes the inverse of this matrix and stores it for subsequent queries.

setTexCoord2f()

inline void setTexCoord2f (GLfloat *ss*, GLfloat *tt*);

Equivalent to glTexCoord2f, though no OpenGL state is modified.

setTexCoordPointer()

void setTexCoordPointer (GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);

Equivalent to glTexCoordPointer. Replace calls to glTexCoordPointer to setTexCoordPointer.

setVertex3f()

inline void setVertex3f (GLfloat *xx*, GLfloat *yy*, GLfloat *zz*);

Issues per-vertex isl::VertexShader operations, as specified to this isl::VertexContext, and draws the specified vertex, using current state from prior set* calls. Issues post-computed results to OpenGL pipeline equivalently to glVertex3f.

setVertexPointer()

void setVertexPointer (GLint *size*, GLenum *type*, GLsizei *stride*, const GLvoid* *pointer*);

Equivalent to glVertexPointer. Replace calls to glVertexPointer to setVertexPointer.

setVertexShader()

void setVertexShader (VertexShader* *fn*);

Specifies the current isl::VertexShader to be used.

typeProgram()

ProgramType typeProgram ( ) const;

Returns the current type of the VertexContext program.

SEE ALSO

islAppearance, isl::VertexContext, isl::VertexShader

NAME

isl::VertexShader - OpenGL Shader Vertex Shading class

HEADER FILE

#include <shader/islvertex.h>

PUBLIC METHOD SUMMARY

isl::VertexContext manipulation

inline void setContext (VertexContext* *cc*);
inline VertexContext* getContext ( ) const;

Shading methods

virtual void init (void);
virtual void run (void);

PROTECTED MEMBER SUMMARY

VertexContext* _ctxt;

CLASS DESCRIPTION

The isl::VertexShader class defines the base class for operations which are performed per-vertex over a set of geometric primitives. These primitives are specified through the isl::VertexContext class and methods.

Though isl::VertexShader is not a pure-virtual base class, and can be instantiated directly, it performs no operations, and is designed to be used only as a base-class from which concrete vertex shading classes are derived. For example, a contrived vertex shader might be implemented as:

```
class myTexGen : public isl::VertexShader
{
  protected:
    float scale

  public:
    void run()
    {
      memcpy( _ctxt->getTexCoord( 0 ), _ctxt->getNormal( 0 ),
              2*sizeof( ISLfloat );
    }
};
```

This shader would then, per-vertex, simply use the X- and Y-components of the per-vertex normal as texture coordinates.

For details on the operation and interaction between VertexShaders and isl::VertexContexts, please read the isl::VertexContext man page.

Both init() and run() methods can use any data in the _ctxt to do per-vertex work. To operate on this data, the **ISL math libraries**, as found in **islmath.h** and **libislmath.so**, are provided which package a wide variety of common matrix math. Please read about the islmath library and it's functionality in associated man-pages for details.

METHOD DESCRIPTIONS

getContext()

inline VertexContext* getContext ( ) const;

Returns the current isl::VertexContext in which this shader is being used.

init()

virtual void init (void);

Executes custom vertex shading code initialization.

Could be used to setup lights of interest, perform a custom calcluation required at each vertex, or simply do nothing.

run()

virtual void run (void);

Executes the vertex shading code in this method, once per-vertex.

setContext()

inline void setContext (VertexContext* *cc*);

Sets the isl::VertexContext in which this vertex shader will operate.

MEMBER DESCRIPTIONS

_ctxt

VertexContext* _ctxt;

Points to the context in which this isl::VertexShader is being used. Protected so that derived classes can access it directly.

This variable is the primary means a isl::VertexShader has for accessing data about its operand and its environment. See the isl::VertexContext man page for more details on what data is provieded through a VertexContext.

SEE ALSO

isl::VertexContext, isl::VertexShader