SoftArt Shader Interfaces

Enums

|  |  |
| --- | --- |
| **Enum** | **Description** |
| cbuffer\_type | Indicates a constant buffer's type. |
| shader\_variable\_class | These flags identify the shader-variable class. |
| shader\_variable\_type | These flags identify the shader-variable type. |
| system\_value\_name | These flags identify shader parameters that use system-value semantics. |
| register\_component\_type | The register component types, usually used in shader\_param\_desc. |
| shader\_input\_type | These flags identify a shader-resource type. |
| shader\_input\_flags | These flags identify shader-input options. |
| resource\_return\_type | Indicates return value type. |
| srv\_dimension | These flags identify the type of resource that will be viewed. |

# cbuffer\_type

Indicates a constant buffer's type.

enum cbuffer\_type{  
 ct\_cbuffer,  
 ct\_tbuffer,  
 ct\_resource\_bind\_info   
};

### Constants

*ct\_cbuffer*A buffer containing scalar constants.

*ct\_tbuffer*A buffer containing texture data.

*ct\_resource\_bind\_info*A buffer containing binding information.

# shader\_variable\_class

These flags identify the shader-variable class.

enum shader\_variable\_class{  
 svc\_scalar,  
 svc\_vector,  
 svc\_matrix\_rows,  
 svc\_matrix\_columns,  
 svc\_object,  
 svc\_struct  
};

### Constants

*svc\_scalar*The shader variable is a scalar.

*svc\_vector*The shader variable is a vector.

*svc\_matrix\_rows*The shader variable is a row-major matrix.

*svc\_matrix\_columns*The shader variable is a column-major matrix.

*svc\_object*The shader variable is an object.

*svc\_struct*The shader variable is a structure.

### Remarks

These flags describe the class of a shader variable. This is not a programming class; the class identifies whether the variable class such as scalar, vector, object, etc. The shader-variable class is used in a shader-input-signature description (see shader\_input\_bind\_desc).

# shader\_variable\_type

These flags identify the shader-variable type.

enum shader\_variable\_type{  
 svt\_void,  
 svt\_bool,  
 svt\_int,  
 svt\_float,  
 svt\_string,  
 svt\_texture,  
 svt\_texture\_1d,  
 svt\_texture\_2d,  
 svt\_texture\_3d,  
 svt\_texture\_cube,  
 svt\_sampler,  
 svt\_vertex\_shader,  
 svt\_pixel\_shader,  
 svt\_blend\_shader,  
 svt\_uint,  
 svt\_uint8,  
 svt\_geometry\_shader,  
 svt\_rasterizer,  
 svt\_depth\_stencil,  
 svt\_blend,  
 svt\_buffer,  
 svt\_cbuffer,  
 svt\_tbuffer,  
 svt\_texture\_1d\_array,  
 svt\_texture\_2d\_array,  
 svt\_render\_target\_view,  
 svt\_depth\_stencil\_view,  
 svt\_texture\_2d\_ms,  
 svt\_texture\_2d\_ms\_array,  
 svt\_texture\_cube\_array,  
 svt\_double,  
};

### Constants

*svt\_void*The variable is a void pointer.

*svt\_bool*The variable is a boolean.

*svt\_int*The variable is a integer.

*svt\_float*The variable is a floating-point number.

*svt\_string*The variable is a string.

*svt\_texture*The variable is a texture.

*svt\_texture\_1d*The variable is a 1D texture.

*svt\_texture\_2d*The variable is a 2D texture.

*svt\_texture\_3d*The variable is a 3D texture.

*svt\_texture\_cube*The variable is a texture cube.

*svt\_sampler*The variable is a sampler.

*svt\_vertex\_shader*The variable is a vertex shader.

*svt\_pixel\_shader*The variable is a pixel shader.

*svt\_blend\_shader*The variable is a blend shader.

*svt\_uint*The variable is an unsigned integer.

*svt\_uint8*The variable is an 8-bit unsigned integer.

*svt\_geometry\_shader*The variable is a geometry shader.

*svt\_rasterizer*The variable is a rasterizer-state object.

*svt\_depth\_stencil*The variable is a depth-stencil-state object.

*svt\_blend*The variable is a blend-state object.

*svt\_buffer*The variable is a buffer.

*svt\_cbuffer*The variable is a constant buffer.

*svt\_tbuffer*The variable is a texture buffer.

*svt\_texture\_1d\_array*The variable is a 1D-texture array.

*svt\_texture\_2d\_array*The variable is a 2D-texture array.

*svt\_render\_target\_view*The variable is a render-target view.

*svt\_depth\_stencil\_view*The variable is a depth-stencil view.

*svt\_texture\_2d\_ms*The variable is a 2D-multisampled texture.

*svt\_texture\_2d\_ms\_array*The variable is a 2D-multisampled-texture array.

*svt\_texture\_cube\_array*The variable is a texture-cube array.

*svt\_double*The variable is a double.

### Remarks

These flags are used by a shader-type description (see shader\_type\_desc).

# system\_value\_name

These flags identify shader parameters that use system-value semantics.

enum system\_value\_name {  
 svn\_undefined,  
 svn\_position,  
 svn\_clip\_distance,  
 svn\_cull\_distance,  
 svn\_vertex\_id,  
 svn\_primitive\_id,  
 svn\_instance\_id,  
 svn\_is\_front\_face,  
 svn\_sample\_index,  
 svn\_target,  
 svn\_depth,  
 svn\_coverage  
};

### Constants

*svn\_undefined*This parameter does not use a predefined system-value semantic.

*svn\_position*This parameter contains position data.

*svn\_clip\_distance*This parameter contains clip-distance data.

*svn\_cull\_distance*This parameter contains cull distance data.

*svn\_vertex\_id*This parameter contains a vertex ID.

*svn\_primitive\_id*This parameter contains a primitive ID.

*svn\_instance\_id*This parameter contains a instance ID.

*svn\_is\_front\_face*This parameter contains data that identifies whether or not the primitive faces the camera.

*svn\_sample\_index*This parameter a sampler-array index.

*svn\_target*This parameter contains render-target data.

*svn\_depth*This parameter contains depth data.

*svn\_coverage*This parameter contains alpha-coverage data.

### Remarks

These flags identify shader parameters that have predefined system-value semantics. These flags are used in a shader-signature description (see shader\_param\_desc).

# register\_component\_type

The register component types, usually used in shader\_param\_desc.

enum register\_component\_type {  
 rct\_unknown = 0,  
 rct\_uint32 = 1,  
 rct\_sint32 = 2,  
 rct\_float32 = 3,  
};

### Constants

*rct\_unknown*Reserved.

*rct\_uint32*32-bit unsigned integer.

*rct\_sint32*32-bit signed integer.

*rct\_float32*32-bit floating-point number.

# shader\_input\_type

These flags identify a shader-resource type.

enum shader\_input\_type {  
 sit\_cbuffer,  
 sit\_tbuffer,  
 sit\_texture,  
 sit\_sampler,  
};

### Constants

*sit\_cbuffer*The shader resource is a constant buffer.

*sit\_tbuffer*The shader resource is a texture buffer.

*sit\_texture*The shader resource is a texture.

*sit\_sampler*The shader resource is a sampler.

### Remarks

These flags describe a shader resource that is used as an input in a shader-input-signature description (see shader\_input\_bind\_desc).

# shader\_input\_flags

These flags identify shader-input options.

enum shader\_input\_flags {  
 sif\_user\_packed = 1 << 0,  
 sif\_comparison\_sampler = 1 << 1,  
 sif\_texture\_component\_0 = 1 << 2,  
 sif\_texture\_component\_1 = 1 << 3,  
 sif\_texture\_components = sif\_texture\_component\_0 | sif\_texture\_component\_1  
};

### Constants

sif\_user\_packed  
Assign a shader input to a register based on the register assignment in the SASL code (instead of letting the compiler choose the register).

sif\_comparison\_sampler   
Use a comparison sampler, which uses the SampleCmp and SampleCmpLevelZero sampling functions.

sif\_texture\_component\_0   
A 2-bit value for encoding texture components.

sif\_texture\_component\_1   
A 2-bit value for encoding texture components.

sif\_texture\_components   
A 2-bit value for encoding texture components.

### Remarks

These flags are used in a shader-input-signature description (see shader\_input\_bind\_desc).

# resource\_return\_type

Indicates return value type.

enum resource\_return\_type {  
 rt\_unorm = 1,  
 rt\_snorm = 2,  
 rt\_sint = 3,  
 rt\_uint = 4,  
 rt\_float = 5,  
 rt\_mixed = 6,  
 rt\_double = 7,  
 rt\_continued = 8,  
};

### Constants

*rt\_unorm*Return type is UNORM.

*rt\_snorm*Return type is SNORM.

*rt\_sint*Return type is SINT.

*rt\_uint*Return type is UINT.

*rt\_float*Return type is FLOAT.

*rt\_mixed*Return type is unknown.

*rt\_double*Return type is DOUBLE.

*rt\_continued*Return type is a multiple-dword type, such as a double or uint64, and the component is continued from the previous component that was declared. The first component represents the lower bits.

# srv\_dimension

These flags identify the type of resource that will be viewed.

enum srv\_dimension {  
 srv\_dimension\_unknown = 0,  
 srv\_dimension\_buffer = 1,  
 srv\_dimension\_texture\_1d = 2,  
 srv\_dimension\_texture\_1d\_array = 3,  
 srv\_dimension\_texture\_2d = 4,  
 srv\_dimension\_texture\_2d\_array = 5,  
 srv\_dimension\_texture\_2d\_ms = 6,  
 srv\_dimension\_texture\_2d\_ms\_array = 7,  
 srv\_dimension\_texture\_3d = 8,  
 srv\_dimension\_texture\_cube = 9,  
};

### Constants

*srv\_dimension\_unknown*The type is unknown.

*srv\_dimension\_buffer*The resource is a buffer.

*srv\_dimension\_texture\_1d*The resource is a 1D texture.

*srv\_dimension\_texture\_1d\_array*The resource is an array of 1D textures.

*srv\_dimension\_texture\_2d*The resource is a 2D texture.

*srv\_dimension\_texture\_2d\_array*The resource is an array of 2D textures.

*srv\_dimension\_texture\_2d\_ms*The resource is a multisampling 2D texture.

*srv\_dimension\_texture\_2d\_ms\_array*The resource is an array of multisampling 2D textures.

*srv\_dimension\_texture\_3d*The resource is a 3D texture.

*srv\_dimension\_texture\_cube*The resource is a cube texture.

### Remarks

These flags are used by a shader-resource-view description (see shader\_resource\_view\_desc), or a shader-input-signature description (see shader\_input\_bind\_desc).

Structs

|  |  |
| --- | --- |
| **Struct** | **Description** |
| shader\_buffer\_desc | Describes a shader constant-buffer. |
| shader\_variable\_desc | Describes a shader variable. |
| shader\_type\_desc | Describes a shader-variable type. |
| shader\_desc | Describes a shader. |
| shader\_param\_desc | Describes a shader signature. |
| shader\_input\_bind\_desc | Describes how a shader resource is bound to a shader input. |

# shader\_buffer\_desc

Describes a shader constant-buffer.

struct shader\_buffer\_desc {  
 std::string name;  
 cbuffer\_type type;  
 uint32\_t variables;  
 uint32\_t size;  
 uint32\_t flags;  
};

### Members

*name*The name of the buffer.

*type*The intended use of the constant data.

*variables*The number of unique variables.

*size*Buffer size (in bytes).

*flags*Shader buffer properties.

### Remarks

Constants are supplied to shaders in a shader-constant buffer. Get the description of a shader-constant-buffer by calling shader\_reflection\_constant\_buffer::get\_desc.

# shader\_variable\_desc

Describes a shader variable.

struct shader\_variable\_desc {  
 std::string name;  
 uint32\_t start\_offset;  
 uint32\_t size;  
 uint32\_t flags;  
 void\* default\_value;  
};

### Members

*name*The variable name.

*start\_offset*Offset from the start of the parent structure, to the beginning of the variable.

*size*Size of the variable (in bytes).

*flags*Flags, which identify shader-variable properties

*default\_value*The default value for initializing the variable.

### Remarks

Get a shader-variable description using reflection, by calling shader\_reflection\_variable::get\_desc.

# shader\_type\_desc

Describes a shader-variable type.

struct shader\_type\_desc {  
 shader\_variable\_class class;  
 shader\_variable\_type type;  
 uint32\_t rows;  
 uint32\_t columns;  
 uint32\_t elements;  
 uint32\_t members;  
 uint32\_t offset;  
};

### Members

*class*Identifies the variable class as one of scalar, vector, matrix or object. See shader\_variable\_class.

*type*The variable type. See shader\_variable\_type.

*rows*Number of rows in a matrix. Otherwise a numeric type returns 1, any other type returns 0.

*columns*Number of columns in a matrix. Otherwise a numeric type returns 1, any other type returns 0.

*elements*Number of elements in an array; otherwise 0.

*members*Number of members in the structure; otherwise 0.

*offset*  
Offset, in bytes, between the start of the parent structure and this variable.

### Remarks

Get a shader-variable-type description by calling shader\_reflection\_type::get\_desc.

# shader\_desc

Describes a shader.

struct shader\_desc {  
 uint32\_t version;  
 uint32\_t flags;  
 uint32\_t constant\_buffers;  
 uint32\_t bound\_resources;  
 uint32\_t input\_params;  
 uint32\_t output\_params;  
};

### Members

*version*Shader version.

*flags*Shader compilation/parse flags.

*constant\_buffers*The number of shader-constant buffers.

*bound\_resources*The number of resource (textures and buffers) bound to a shader.

*input\_params*The number of parameters in the input signature.

*output\_params*The number of parameters in the output signature.

### Remarks

A shader is written in SASL and compiled into an intermediate language by the SASL compiler. The shader description returns information about the compiled shader. Get a shader description by calling shader::get\_desc.

# shader\_param\_desc

Describes a shader signature.

struct shader\_param\_desc {  
 std::string semantic\_name;  
 uint32\_t semantic\_index;  
 uint32\_t register;  
 system\_value\_name system\_value\_type;  
 register\_component\_type component\_type;  
 uint8\_t mask;  
 uint8\_t read\_write\_mask;  
 uint32\_t stream;  
};

### Members

*semantic\_name*A per-parameter string that identifies how the data will be used.

*semantic\_index*Semantic index that modifies the semantic. Used to differentiate different parameters that use the same semantic.

*register*The register that will contain this variable's data.

*system\_value\_type*A predefined string that determines the functionality of certain pipeline stages. See system\_value\_name.

*component\_type*The per-component-data type that is stored in a register. See register\_component\_type. Each register can store up to four-components of data.

### Remarks

A shader can take n inputs and can produce m outputs. The order of the input (or output) parameters, their associated types, and any attached semantics make up the shader signature. Each shader has an input and an output signature.

When compiling a shader or an effect, some API calls validate shader signatures That is, they compare the output signature of one shader (like a vertex shader) with the input signature of another shader (like a pixel shader). This ensures that a shader outputs data that is compatible with a downstream shader that is consuming that data. Compatible means that a shader signature is a exact-match subset of the preceeding shader stage. Exact match means parameter types and semantics must exactly match. Subset means that a parameter that is not required by a downstream stage, does not need to include that parameter in its shader signature.

Get a shader-signature from a shader or an effect by calling APIs such as shader::get\_input\_param\_desc.

# shader\_input\_bind\_desc

Describes how a shader resource is bound to a shader input.

struct shader\_input\_bind\_desc {  
 std::string name;  
 shader\_input\_type type;  
 uint32\_t bind\_point;  
 uint32\_t bind\_count;  
 uint32\_t flags;  
 resource\_return\_type return\_type;  
 srv\_dimension dimension;  
 uint32\_t num\_samples;  
};

### Members

name   
Name of the shader resource.

type   
Identifies the type of data in the resource. See shader\_input\_type.

bind\_point   
Starting bind point.

bind\_count   
Number of contiguous bind points for arrays.

flags   
Shader input-parameter options. See shader\_input\_flags

return\_type   
If the input is a texture, the return type. See resource\_return\_type.

dimension   
Identifies the amount of data in the resource. See srv\_dimension.

num\_samples   
The number of samples for a multisampled texture; otherwise 0.

### Remarks

Get a shader-input-signature description by calling shader::get\_resource\_binding\_desc.

Classes

# shader\_reflection\_constant\_buffer class

This shader-reflection interface provides access to a constant buffer. This interface does not inherit from anything, but does declare the following methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| shader\_reflection\_constant\_buffer::get\_desc | Get a constant-buffer description. |
| shader\_reflection\_constant\_buffer::get\_variable | Get a shader-reflection variable |

To create a constant-buffer interface, call shader::get\_constant\_buffer.

# shader\_reflection\_constant\_buffer::get\_desc

Get a constant-buffer description.

const shader\_buffer\_desc& get\_desc();

### Parameters

None.

### Return Values

A constant reference to a shader\_buffer\_desc, which represents a shader-buffer description.

# shader\_reflection\_constant\_buffer::get\_variable

Get a shader-reflection variable by index or name.

shader\_reflection\_variable\* get\_variable(uint32\_t index);  
shader\_reflection\_variable\* get\_variable(const std::string& name);

### Parameters

*index*[in] Zero-based index.

*name*[in] Variable name.

### Return Values

A pointer to a shader-reflection variable interface (see shader\_reflection\_variable).

# shader\_reflection\_variable class

This shader-reflection interface provides access to a variable. This interface does not inherit from anything, but does declare the following methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| shader\_reflection\_variable::get\_desc | Get a shader-variable description. |
| shader\_reflection\_ variable::get\_type | Get a shader-variable type. |

To get a shader-reflection-variable interface, call a method like shader::get\_variable.

# shader\_reflection\_variable::get\_desc

Get a shader-variable description.

const shader\_variable\_desc& get\_desc();

### Parameters

None.

### Return Values

A constant reference to a shader-variable description (see shader\_variable\_desc).

# shader\_reflection\_variable::get\_type

Get a shader-variable type.

shader\_reflection\_type\* get\_type();

### Parameters

None.

### Return Values

A pointer to a shader\_reflection\_type.

# shader\_reflection\_type class

This shader-reflection interface provides access to variable type. This interface does not inherit from anything, but does declare the following methods:

|  |  |
| --- | --- |
| Method | Description |
| shader\_reflection\_type::get\_base\_class | Gets a variable base class type. |
| shader\_reflection\_type::get\_desc | Gets a shader-type description. |
| shader\_reflection\_type::get\_member\_type | Gets a variable type by index. |
| shader\_reflection\_type::get\_member\_type\_name | Gets a variable type. |
| shader\_reflection\_type::get\_sub\_type | Gets the base class of a class. |
| shader\_reflection\_type::operator== | Indicates whether two Shader\_reflection\_type Interface pointers have the same underlying type. |
| shader\_reflection\_type::is\_of\_type | Indicates if a variable is of the specified type. |

The get a shader-reflection-type interface, call shader\_reflection\_variable::get\_type.

# shader\_reflection\_type::get\_base\_class

Gets an Shader\_reflection\_type object containing the variable base class type.

shader\_reflection\_type\* get\_base\_class();

### Parameters

None.

### Return Values

A pointer to a shader\_reflection\_type object.

# shader\_reflection\_type::get\_desc

Get the description of a shader-reflection-variable type.

const shader\_type\_desc& get\_desc();

### Parameters

None.

### Return Values

A constant reference to a shader-type description (see shader\_type\_desc).

# shader\_reflection\_type::get\_member\_type

Get a shader-reflection-variable type by index or name.

shader\_reflection\_type\* get\_member\_type(uint32\_t index);  
shader\_reflection\_type\* get\_member\_type(const std::string& name);

### Parameters

*Index*[in] Zero-based index.

*name*[in] Member name.

### Return Values

A pointer to a shader\_reflection\_type object.

# shader\_reflection\_type::get\_member\_type\_name

Get a shader-reflection-variable type.

const std::string& get\_member\_type\_name(uint32\_t Index);

### Parameters

*index*  
[in] Zero-based index.

### Return Values

The variable type.

# shader\_reflection\_type::get\_sub\_type

Gets the base class of a class.

shader\_reflection\_type\* get\_sub\_type();

### Parameters

None.

### Return Values

Returns a pointer to a shader\_reflection\_type Interface containing the base class type. Returns NULL if the class does not have a base class.

# shader\_reflection\_type::operator==

Indicates whether two shader\_reflection\_type Interface pointers have the same underlying type.

friend bool operator==(const shader\_reflection\_type& lhs , const shader\_reflection\_type& rhs);

### Parameters

*lhs, rhs*[in] A constant reference to a shader\_reflection\_type object.

### Return Values

Returns *true* if the objects have the same underlying type; otherwise returns *false.*

### Remarks

operator== indicates whether the sources of the shader\_reflection\_type objects have the same underlying type. For example, if two shader\_reflection\_type Interface pointers were retrieved from variables, operator== can be used to see if the variables have the same type.

# shader\_reflection\_type::is\_of\_type

Indicates whether a variable is of the specified type.

bool is\_of\_type(const shader\_reflection\_type& type);

### Parameters

*type*[in] A constant reference to a shader\_reflection\_type Interface.

### Return Values

Returns *true* if object being queried is equal to or inherits from the type in the *type* parameter; otherwise returns *false*.

# shader class

|  |  |
| --- | --- |
| **Method** | **Description** |
| shader::get\_constant\_buffer | Gets a constant buffer. |
| shader::get\_desc | Gets a shader description. |
| shader::get\_input\_param\_desc | Gets an input-parameter description for a shader. |
| shader::get\_output\_param\_desc | Gets an output-parameter description for a shader. |
| shader::get\_resource\_binding\_desc | Gets a description of the resources bound to a shader. |
| shader::get\_variable | Gets a variable. |

# shader::get\_constant\_buffer

Get a constant buffer by index or name.

shader\_reflection\_constant\_buffer\* get\_constant\_buffer (uint32\_t index);  
shader\_reflection\_constant\_buffer\* get\_constant\_buffer (const std::string& name);

### Parameters

*index*   
[in] Zero-based index.

*name*  
[in] The constant-buffer name.

### Return Values

A pointer to a constant buffer (see shader\_reflection\_constant\_buffer).

### Remarks

A constant buffer supplies either scalar constants or texture constants to a shader. A shader can use one or more constant buffers. For best performance, separate constants into buffers based on the frequency they are updated.

# shader::get\_desc

Get a shader description.

const shader\_desc& get\_desc() const;

### Return Values

A constant reference to a shader description. See shader\_desc.

# shader::get\_input\_param\_desc

Get an input-parameter description for a shader.

const shader\_param\_desc& get\_input\_param\_desc (uint32\_t index) const;

### Parameters

*index*[in] A zero-based parameter index.

### Return Values

A constant reference to a shader input description. See shader\_param\_desc.

### Remarks

An input-parameter description is also called a shader signature. The shader signature contains information about the input parameters such as the order or parameters, their data type, and a parameter semantic.

# shader::get\_output\_param\_desc

Get an output-parameter description for a shader.

const shader\_param\_desc& get\_output\_param\_desc (uint32\_t index) const;

### Parameters

*index*[in] A zero-based parameter index.

### Return Values

A constant reference to a shader output description. See shader\_param\_desc.

### Remarks

An output-parameter description is also called a shader signature. The shader signature contains information about the output parameters such as the order or parameters, their data type, and a parameter semantic.

# shader::get\_resource\_binding\_desc

Get a description of the resources bound to a shader.

const shader\_input\_bind\_desc& get\_resource\_binding\_desc(uint32\_t index);  
const shader\_input\_bind\_desc& get\_resource\_binding\_desc(const std::string& name);

### Parameters

*index*  
[in] A zero-based resource index.

*name*  
[in] The constant-buffer name.

### Return Values

A constant reference to an input-binding description. See shader\_input\_bind\_desc.

### Remarks

A shader consists of executable code (the compiled SASL functions) and a set of resources that supply the shader with input data. This API gets a list of the resources that are bound as inputs to the shader.

# shader::get\_variable

Gets a variable by name.

shader\_reflection\_variable\* get\_variable(const std::string& name);

### Parameters

*name*  
[in] A string containing the variable name.

### Return Values

A pointer to shader\_reflection\_variable object (see shader\_reflection\_variable).

Compiler Interfaces

Structs

# shader\_macro

Defines a shader macro.

struct shader\_macro {  
 std::string name;  
 std::string definition;  
};

### Members

*name*  
The macro name.

*definition*  
The macro definition.

### Remarks

You may use macros in your shaders. This enables the application to #define tokens at runtime, before the file is parsed. This structure defines a single macro. For example:

shader\_macro Shader\_Macros[1] = { "zero", "0" };

The shader creation methods *compile\_shader* take an array of macros as an input parameter.

Classes

# shader\_include

A shader\_include base class allows an application to create user-overridable methods for opening and closing files when loading an effect from memory. This interface does not inherit from anything, but does declare the following methods:

|  |  |
| --- | --- |
| **Method** | **Description** |
| shader\_include::close | A user-implemented method for closing a shader #include file. |
| shader\_include::open | A user-implemented method for opening and reading the contents of a shader #include file. |

To use this class, create a class that inherits from shader\_include, and implement custom behavior for the methods.

# shader\_include::close

A user-implemented method for closing a shader #include file.

void close(  
 std::istream& stream  
);

### Parameters

*stream*  
[in] A stream that contains the include directives. This is the input stream that was returned by the corresponding shader\_include::open call.

### Remarks

If shader\_include::open was successful, shader\_include::close is guaranteed to be called before the API using this interface returns.

# shader\_include::open

A user-implemented method for opening and reading the contents of a shader #include file.

void open(  
 const std::string& file\_name,  
 const void\* parent\_data,  
 std::istream& stream  
);

### Parameters

*file\_name*[in] Name of the #include file.

*parent\_data*[in] Pointer to the container that includes the #include file.

*stream*[out] The returned stream that contains the include directives. This stream remains valid until shader\_include::close is called.

Functions

# compile\_shader

Compile a shader from a stream.

shader\* compile\_shader(  
 std::istream& stream,  
 const shader\_macro\*defines,  
 shader\_include\* include\_handler,  
 const std::string& func\_name,  
 uint32\_t flags,  
 std::string\* error\_msgs,  
);

### Parameters

*stream*[in] The stream that contains the shader code.

*defines*[in] Optional. Pointer to an array of macro definitions (see shader\_macro). The last structure in the array serves as a terminator and must have all members set to 0. If not used, set *defines* to NULL.

*include\_handler*[in] Optional. Pointer to a class for handling include files. Setting this to NULL will cause a compile error if a shader contains a #include.

*function\_name*   
[in] Name of the shader-entrypoint function where shader execution begins.

*flags*[in] Shader compile flags. Reserved.

*error\_msgs*[out] A pointer to string which contains a listing of errors and warnings that occurred during compilation.

### Return Values

A pointer to a shader object which contains the compiled shader, as well as any embedded debug and symbol-table information.

Host Interfaces

|  |  |
| --- | --- |
| **Method** | **Description** |
| renderer::set\_vertex\_shader | Set a vertex shader to the renderer. |
| renderer::set\_pixel\_shader | Set a pixel shader to the renderer. |
| renderer::set\_blend\_shader | Set a blend shader to the renderer. |
| renderer::get\_vertex\_shader | Get the vertex shader currently set on the renderer. |
| renderer::get\_pixel\_shader | Get the pixel shader currently set on the renderer. |
| renderer::get\_blend\_shader | Get the blend shader currently set on the renderer. |

# renderer::set\_vertex\_shader

Set a vertex shader to the renderer.

void set\_vertex\_shader(  
 shader\*vs  
);

### Parameters

*vs*[in] Pointer to a vertex shader (see shader). Passing in NULL disables the shader for this pipeline stage.

# renderer::set\_pixel\_shader

Set a pixel shader to the renderer.

void set\_pixel\_shader(  
 shader\*ps  
);

### Parameters

*ps*[in] Pointer to a pixel shader (see shader). Passing in NULL disables the shader for this pipeline stage.

# renderer::set\_blend\_shader

Set a blend shader to the renderer.

void set\_blend\_shader(  
 shader\*bs  
);

### Parameters

*bs*[in] Pointer to a blend shader (see shader). Passing in NULL disables the shader for this pipeline stage.

# renderer::get\_vertex\_shader

Get the vertex shader currently set on the renderer.

shader\* get\_vertex\_shader();

### Parameters

None.

### Return Values

Address of a pointer to a vertex shader (see shader) to be returned by the method.

# renderer::get\_pixel\_shader

Get the pixel shader currently set on the renderer.

shader\* get\_pixel\_shader();

### Parameters

None.

### Return Values

Address of a pointer to a pixel shader (see shader) to be returned by the method.

# renderer::get\_blend\_shader

Get the blend shader currently set on the renderer.

shader\* get\_blend\_shader();

### Parameters

None.

### Return Values

Address of a pointer to a blend shader (see shader) to be returned by the method.