Shading System Immediate-Mode API v2.2

William R. Mark and C. Philipp Schloter

August 29, 2001

1 Introduction

This document describes modifications to the OpenGL API to support the immediate-mode use of the Stanford real-time shading language. We collectively refer to these extensions as the shading-language immediate-mode API. These extensions are implemented as a layer on top of regular OpenGL.

The immediate-mode API supports the following major operations:

1. Loading the source code for a light shader or surface shader from a file.
2. Associating one or more light shader(s) with a surface shader to create a combined surface/light shader.
3. Compiling a combined surface/light shader for the current graphics hardware.
4. Selecting a compiled surface/light shader to use as the current shader for rendering.
5. Setting values of shader parameters.

When a shader is active, many OpenGL commands are no longer allowed, because their functionality is provided through the shading language. The disallowed commands fall into four major categories:

1. Fragment-processing commands (e.g. fog, texturing modes)
2. Texture-coordinate generation and transformation commands
3. Lighting commands.
4. Material-property commands.

When using our “lburg” multi-pass fragment backend, commands that configure framebuffer blending modes are also forbidden (instead, use the `Cprev` builtin variable within a shader). However, these commands are allowed with our “nv” fragment backend.

Using a forbidden OpenGL command while a programmable-shader is active will result in undefined behavior.

2 Initialization

`sglInit()`

The application must initialize the programmable shading system by calling `sglInit()` once, before calling any other `sgl*` routines.

3 Loading shader source code

`int sglShaderFile(GLuint shaderSourceID, char *shaderName, char *filename)`

Loads the source code for the shader named `shaderName` from file `filename`, and assigns it the identifier `shaderCodeID`. Any other shaders that are specified in the file are ignored. The loaded shader source code becomes the active shader source code. The specified shader may be either a light shader or a surface shader. `shaderCodeID` must be unused when this routine is called. The return code is 0 if there were no errors, 1 if there was an error.

4 Compiling and activating shaders

After the source code for a shader is loaded, but before it is used, the shader must be compiled. Our system treats shader source code and compiled shaders as largely separate entities.

`sglCompileShader(GLuint shaderID)`

Compiles the current shader source code. The compiled shader is assigned the user-specified identifier `shaderID`. If the shader is a surface shader, it incorporates any currently associated light shaders (discussed in the next section).

The newly created shader becomes the 'active' shader, as if sglBindShader() had been called. If the shader is a light shader, it is only active in the sense that subsequent `sglParameterHandle()` calls will apply to it. A light shader can only be activated for rendering purposes by associating it with a surface shader using `sglUseLight()`.

The current shader source code remains unchanged by this call.

Note that `shaderID` may not be -1, because this value is reserved for `SGL_STD_OPENGL`, the standard OpenGL lighting/shading model.

`sglBindShader(GLuint shaderID)`

Changes the currently active shader to that specified by `shaderID`. Note that it is illegal to render geometry when a light shader is bound.

Specifying `SGL_STD_OPENGL` reverts to the standard OpenGL lighting/shading model.

5 Associating lights with a surface

For efficiency reasons, the shading system must know which light shaders will be used with a surface shader before the surface shader is compiled.

`sglUseLight(GLuint lightShaderID)`

This command binds a “compiled” light shader to the current surface-shader source code. `lightShaderID` indicates the light that is to be associated with the surface.

More than one light can be associated with a surface, by calling `sglUseLight()` multiple times.

However, the same (compiled) light shader may not be used more than once with a single surface. If two identical lights are required, compile the light shader twice. Our system imposes this requirement because the *lightShaderID* is used to specify how light parameters are modified. “Identical” lights will usually have different parameter values (e.g. position).

5.1 Setting parameter values

For performance reasons, shader parameters are identified at rendering-time with numeric identifiers rather than names. For each compiled shader, the programmer can choose the bindings from names to numeric identifiers, within some constraints. We refer to the numeric parameter identifiers as *parameter handles*. Each compiled surface or light shader has its own parameter-handle space.

There is an important advantage to allowing the programmer to choose values of parameter handles. It facilitates the use of a single geometry rendering routine (e.g. renderSphere) with different surface shaders, as long as the programmer chooses a consistent mapping of parameter handles to actual parameters for all of the relevant shaders.

`sglParameterHandle(char *paramName, GLuint paramHandle)`

Assigns the parameter handle `paramHandle` to the current shader’s parameter `paramName`. The value of paramHandle must be between 0 and `SGL_MAX_PARAMHANDLE`. The value of `SGL_MAX_PARAMHANDLE` is guaranteed to be no less than 15.

`sglParameter*(GLuint paramHandle, TYPE v, ... )`

`sglParameter*v(GLuint paramHandle, TYPE *v)`

Assigns a value to the shader parameter(s) specified by `paramHandle`. For a per-vertex parameter, this routine may be called at any time. For a per-primitive parameter, this routine may only be called outside of a begin/end pair.

Because our shading language does not explicitly identify shader parameters as “colors” or “texture coordinates”, the shading system can not automatically assign default values in the manner that OpenGL does. For example, in OpenGL a `glColor3f` command automatically sets the fourth value (alpha) to the default value of 1.0. When using our system, the user must always specify all four components of the color value. Likewise, the user must always specify all four components of a texture value. For a 2D texture, the third and fourth values should usually be set to 0.0 and 1.0 respectively.

The `sglParameter*` routine is available in `sglParameter1*`, `sglParameter4*`, and `sglParameter16*` variants. The `sglParameter16*` variants are used to specify matrix parameters, using OpenGL's array format.

If the shading language specifies a parameter's type as either `clampf` or `clampfv`, type conversions are performed in the same manner as they are for the OpenGL `glColor*` routines (see OpenGL Red Book, 3rd edition, Table 4-1).[1]. In summary, integer-to-float conversions are performed such that the maximum integer value (e.g. 255 for an unsigned byte) maps to 1.0. This behavior allows colors and normals to be stored in unsigned bytes in a natural manner.

Our shading language uses textures, but the contents of the textures are not defined using the language. Textures are defined by the application program, then passed to the shading-language routine as a 'texref' parameter. Our system relies on OpenGL's texture object facility (`glBindTexture()`). The `sglParameter1i` or `sglParameter1iv` routines are used to specify 'texref' parameters. The value of the integer parameter is the *textureName* created using `glBindTexture()`.

```
sglLightParameter*(GLuint lightShaderID, GLuint paramHandle, TYPE v, ... )
sglLightParameter*v(GLuint lightShaderID, GLuint paramHandle, TYPE *v)
```

Assigns a value to the light parameter specified by `paramHandle`. The "compiled" light shader is specified by `lightShaderID`. For a per-vertex parameter, this routine may be called at any time. For a per-primitive parameter, this routine may only be called outside of a begin/end pair.

5.2 Light Pose

The pose of a light (position, direction, and orientation) is set using a routine defined for that purpose.

```
sglLightPosefv(GLuint lightShaderID, GLuint pname, GLfloat *v)
```

`pname` can be `SGL_POSITION`, `SGL_DIRECTION`, or `SGL_UPAXIS`. The light direction defines the $-Z$ axis in light space, and the up axis defines the Y axis in light space.

The vector `v` should always be a four-element vector, and is considered to be in modelview space (i.e. transformed by the modelview matrix or its inverse transpose, as appropriate). For `SGL_POSITION`, the fourth element of the vector should usually be set to 1.0. For `SGL_DIRECTION` and `SGL_UPAXIS`, the fourth element should usually be set to 0.0.

[1]For implementation simplicity, our system deviates from the behavior in Table 4-1 in a minor way. Our system treats negative and positive values symmetrically. For example, a signed-byte value of -127 maps to -1.0, whereas in OpenGL the value of -128 maps to -1.0

5.3 Ambient Light

```
sglAmbient*( ... )
```

Specify the global ambient color. This color is accessible in surface shaders using the pre-defined `Ca` variable. If a surface shader does not use the `Ca` variable, the ambient color will be ignored. This routine can not be called inside a Begin/End pair.

6 Replacements for Standard OpenGL routines

6.1 Begin/End and Flush/Finish

Use `sglBegin()`, `sglEnd()`, `sglFlush()`, and `sglFinish()` instead of the corresponding standard OpenGL routines. Using the standard OpenGL routines while a programmable shader is active will result in undefined behavior.

6.2 Vertices, Normals, Tangents, Binormals

```
sglVertex*(TYPE v, ... )
sglVertex*v(TYPE v, ... )

sglNormal3*(TYPE v, ... )

sglNormal3*v(TYPE v, ... )

sglTangent3*(TYPE v, ... )

sglTangent3*v(TYPE v, ... )

sglBinormal3*(TYPE v, ... )

sglBinormal3*v(TYPE v, ... )
```

Vertices and local coordinate-frame vectors are passed using our versions of the classical OpenGL routines. The results of calling one of the standard OpenGL routines while a programmable shader is active are undefined.

6.3 *Vertex arrays*

To attain higher frame rates when using large models, the shading system provides `sglParameterPointer`, `sglEnableClientState`, `sglGetClientState`, `sglDisableClientState` and `sglDrawArrays`. These routines differ from the OpenGL routines in that they support not only arrays of vertices, normals, binormals or tangents, but also of any other shader parameter. To setup vertex arrays, you have to follow a basic three step procedure, consisting of calls to:

1. `sglParameterPointer`
2. `sglEnableClientState`
3. `sglDrawArrays`.

First, the pointers to the parameter arrays have to be specified by `sglParameterPointer(int handle, GLsizei size, GLenum type, GLsizei stride, float *pointer)`. Valid handles are `SGL_VERTEX, SGL_NOMRAL, SGL_BINORMAL, SGL_TANGENT` or any parameter handle obtained from `sglParameterHandle`. The parameters `size`, `type`, `stride` and `pointer` follow standard OpenGL vertex-array conventions. Please note that in the current version of the immediate-mode API:

- GL_FLOAT is the only supported type.
- Stride should always be set to 4 for `SGL_VERTEX`, and to 3 for `SGL_NORMAL`, `SGL_BINORMAL` or `SGL_TANGENT` arrays.

After specifying all parameter arrays, they must be activated for rendering by calling `sglEnableClientState(int handle)`. Similar, an activated parameter array can be disabled again by calling `sglDisableClientState(int handle)`.

To render the actual vertex array using all activated parameter arrays, call `sglDrawArrays(GLenum mode, GLint first, GLsizei count)` which again follows standard OpenGL conventions. Please note that rendering will only occur if an SGL_VERTEX array was both specified and activated. All other parameter arrays are optional. `SGL_NOMRAL, SGL_BINORMAL` and `SGL_TANGENT` are set to constant default values if not provided.

7 Advanced features

7.1 Manual backend configuration

To offer manual control over which backends the shading system should use, the immediate-mode interface provides `sglSetBackEndType(char* perprimitivegroup, char* vertex, char* fragment)`. This routine presents a wraper for the internal, low-level functions `set_bcodegen`, `set_vcodegen` and `set_fcodegen`.

Currently, there are two primitive-group backends ("cc" and "x86"), three vertex backends ("cc", "x86", and "nv20"), and two fragment backends ("lb" and "nv"). "lb" is a standard-OpenGL backend; "nv" is a register-combiner backend.

7.2 Shader parameter list retrieval

The following two routines allow a program to retrieve the lists of parameters required by a surface shader. To retrieve the number of parameters for the current surface shader:
`sglGetParameterCount(int *count)` where count returns the total number of parameters for the shader.

To retrieve the name and number of values for a specific parameter:
`sglGetParameterInfo(int p, char **name, int *vcnt)` where p defines the parameter of the current shader, ranging from 0 to (1-count), name returns the name of the parameter and vcnt returns the number of values for this parameter. E.g. for a float4 parameter, vcnt would return 4.

To retrieve the lists of parameters required by a light shader, use the following routines:

```
sglGetLightParameterCount(int lightid, int *count)
sglGetLightParameterInfo(int lightid, int p, char **name, int *vcnt)
```

Both routines take a `lightid` as parameter that specifies the light for which information should be retrieved.

8 Depth testing

Ideally, depth testing works exactly as it does in standard OpenGL. However, in some implementations, incorrect shading may occur if two (potentially visible) fragments at a pixel have exactly the same depth. This problem only occurs if an implementation uses the framebuffer for inter-pass temporary storage in a multi-pass shader.

9 Error Handling

The shading system has a flexible method for handling errors. Errors are divided into two classes, minor and major. For each class of error, the application can choose one of four behaviors:

- `SG_MSG_NONE` – No message is printed, and program execution continues. Errors can only be detected by polling for them using `sglGetError`.
- `SGL_MSG_WARN_ONCE` – A message is printed for the first error that occurs, and program execution continues. No message is printed for subsequent errors.
- `SGL_MSG_WARN` – A message is printed for every error that occurs, and program execution continues.
- `SGL_MSG_ABORT` – When an error occurs, a messsage is printed and program execution is halted.

`sglDebugLevel(int minor, int major)`

Specify the behavior for minor and major errors. The default is `sglDebugLevel(SGL_MSG_WARN_ONCE, SGL_MSG_ABORT)`.

`GLenum sglGetError(void)`

Poll for an error. If no error has occurred, `GL_NO_ERROR` is returned. If an error has occurred, the error code is returned.

`const GLubyte* sglErrorString(GLenum errorCode)`

Returns a descriptive string corresponding to an error code.

10 System Tips

In our current implementation, every `sglBegin`/`sglEnd` pair is expensive. If possible, group all primitives into one such pair.

Because of restrictions in current graphics hardware, if a translucent shader is implemented using more than one hardware pass, overlapping transparent primitives will not render correctly. You must call `sglFlush` between each group of potentially overlapping primitives to avoid this problem.