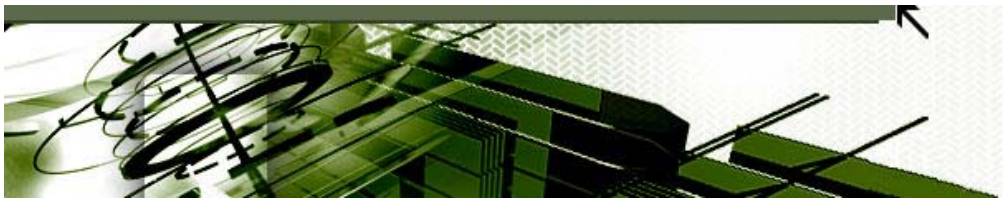

OpenGL Shading Language Course

Chapter 5 – Appendix

By
Jacobo Rodriguez Villar
jacobo.Rodriguez@typhoonlabs.com

APPENDIX

INDEX

Using GLSL Shaders Within OpenGL Applications

Though we have covered creating GLSL shaders, they have not yet been used in any OpenGL-based application outside of Shader Designer. We now need to look at loading, compiling, linking, and using the shaders from within the OpenGL API (more information can be found at http://developer.3dlabs.com/openGL2/slapi/index.htm).

Loading and Using Shaders

For a GLSL shader to work, it is important that the following four extensions are supported by the graphics card:

- GL_ARB_vertex_shader
- GL_ARB_fragment_shader
- GL_ARB_shader_objects
- GL_ARB_shading_language

This information can be found in one of two ways:

- Checking for these tokens using the extensions string `(glGetString(GL_EXTENSIONS);`.
- Using an extension enumeration library like GLEW (http://glew.sourceforge.net), which is much easier and the method we'll use for our example(s).

If the card does support GLSL, you can attempt to load/use the shader in the following way:

- Create a placeholder for your shader using the **`glCreateShaderARB`** call:

```
GLhandleARB vertexShader = glCreateShaderObjectARB(Glenum shaderType);
```

  `shaderType` must be either `GL_VERTEX_SHADER_ARB` for vertex shaders or `GL_FRAGMENT_SHADER_ARB` for fragment shaders.

- Load the source string into your shader using the **`glShaderSourceARB`** call:

```
void glShaderSourceARB(GLhandleARB shader,
                       GLsizei nstrings,
                       const GLcharARB **strings,
                       const GLint *lengths);
```

The shader will be initialized with a **`glCreateShaderARB`** call. `nstrings` holds the number of the strings contained within the shader (usually one); **`strings`** is a pointer to an array of strings; and **`lengths`** is an array which holds the length of the source strings. The shader is finally compiled with the function:

```
void glCompileShaderARB(GLhandleARB shader)
```

For example, to use a vertex shader called “generic.vert” (the file extension is not important), you would initialize it using the following code:

```
size_t ProgramSize(char *filename)
{
  int shader;
  size_t size = -1;

  shader = _open(filename,_O_RDONLY);
  if(shader == ENOENT)
     size = -1;

  size = _lseek(shader, 0, SEEK_END);
  _close(shader);
  return size;
}

{
  FILE *source = fopen("generic.vert","rb");
  size_t size = ProgramSize("generic.vert");
  GLcharARB *programSource = (GLcharARB *)malloc(size*sizeof(GLcharARB));
  fread(m_programSource,m_size,1,source);
  fclose(source);
  GLhandleARB vertexShader = glCreateShaderObjectARB(Glenum shaderType);
  glShaderSourceARB(vertexShader, 1,(const GLcharARB **)&programSource,NULL);
  free(programSource);
  glCompileShaderARB(vertexShader);
}
```

The shader is now compiled and ready to be attached to a program object (the executable unit that is a placeholder for compiled shaders). However, you may want to know the compilation result, as well as warning/error messages (if any).

```
int compiled = 0, length = 0, laux = 0;
glGetObjectParameterivARB(shader, GL_OBJECT_COMPILE_STATUS_ARB, &compiled);
glGetObjectParameterivARB(shader, GL_OBJECT_INFO_LOG_LENGTH_ARB, &length);
GLcharARB *logString = (GLcharARB *)malloc(length * sizeof(GLcharARB));
glGetInfoLogARB(shader, length, &laux, logString);
```

If the `compiled` variable returns **false** the shader will not compile due to errors (ill-formed data, resource issues, etc.). You may gain more information by retrieving the compilation log, but you must first find its length in order to allocate enough space for it.

We can now define an executable unit by creating and compiling the program object. This process involves linking global variables, varyings, etc.

A program object (able to contain shaders of both types), which is both linkable and executable, can be created using the following function:

```
GLhandleARB glCreateProgramObjectARB(void)
```

Once created and compiled, the vertex and fragment shaders must be 'inserted' into the program object (they can be inserted into multiple program objects). This can be achieved using the **glAttachObjectARB** call:

```
void glAttachObjectARB(GLhandleARB program, GLhandleARB shader)
```

```
GLhandleARB vertex = MyFuncThatLoadShaders("vertex.vert");
GLhandleARB fragment = MyFuncThatLoadShaders("fragm.frag");
GLhandleARB program = glCreateProgramObjectARB();
glAttachObjectARB(program,vertex);
glAttachObjectARB(program,fragment);
```

When the shaders are attached, you can link your program and leave it ready to be used:

```
void glLinkProgramARB(GLhandleARB program)
```

You can also query the link status and information log:

```
glLinkProgramARB(program);
glGetObjectParameterivARB(m_program,GL_OBJECT_LINK_STATUS_ARB,&resul);
glGetObjectParameterivARB(m_program,GL_OBJECT_INFO_LOG_LENGTH_ARB,&length);
glGetInfoLogARB(program,length,&laux,infoLog);
```

We now need to activate the program by making it part of the current state, which can be done using:

```
glUseProgramObjectARB(program);
// Render some stuff, with glVertex, glDrawElements...
glUseProgramObjectARB(0); //back to fixed function pipeline
```

The shaders themselves and programs can be deleted using:

```
void glDeleteObjectARB(GLhandleARB object)
```

If an object attached to another object is deleted, it will only be flagged for deletion and not actually deleted until it's no longer attached. This means we must detach all objects before any real deletion can occur:

```
void glDetachObjectARB(GLhandleARB container, GLhandleARB attached)
```

Using Uniforms

Passing variables from an OpenGL application to a GLSL object is done using **`glUniform`** calls, but it is important that the program object is linked first. If it is not linked, the calls will have no effect, as linking operations always initialize all uniforms at 0.

In addition to this, **glUniform** calls do not use strings for variable names, but instead use integers, which can be obtained using:

```
GLint glGetUniformLocationARB(GLhandleARB program, const GLcharARB
*name)
```

```
int location = glGetUniformLocationARB(program,"bumpSize");
glUniform1fARB(location, mybumpSizeVariable);
```

There are many **glUniform** calls:

- **glUniform{1|2|3|4}{f|i}ARB** is used to pass single values.
- **glUniform{1|2|3|4}{f|i}vARB** is used to pass an array of values.
- **glUniformMatrix{2|3|4}fvARB** is used to pass `mat2`, `mat3`, `mat4`, or arrays of them.

A complete description of these calls can be found at:
http://developer.3dlabs.com/openGL2/slapi/UniformARB.htm

Using Textures

Using textures involves passing both the texture unit (not the texture object) number to the shader and the correct binding of the texture to that unit.

We can pass the texture unit number by using the **glUniform1i(location,texture_unit);** call, but if you want use this as a sampler within the fragment (or vertex) shader, you must use the [1i] variant of the **glUniform** or else an error will occur.

You can find out the maximum number of textures a program object can support by querying the `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS_ARB` define with **glGetInteger** (which, once known, can simply activate each texture unit and bind the texture to it) (**glEnable** is not required here):

```
for(int I= 0;I< my number of textures;I++)
{
  glActiveTextureARB(GL_TEXTURE0_ARB + I);
  glBindTextureARB(GL_TEXTURE_2D,my_texture_array[I]);

  // glEnable(GL_TEXTURE_2D);  this is not needed, don't do it

 int location = glGetUniformLocationARB(myProgramObject, mySamplerNameArray[I]);
  glUniform1iARB(location, I );
}
glActiveTextureARB(GL_TEXTURE0_ARB);   // back to texture unit 0
```

Using Vertex Attributes

Every time the **glNormal, glTexCoord**, and **glColor** functions are called, you send pre-defined vertex attributes to the vertex shader. Vertex arrays (like **glTexCoordPointer** and **glNormalPointer**) can be used as well.

We will now look at some generic attributes which work in much the same way, but are more flexible, as you can assign descriptive names or complex types (like **mat4** instead of normal **vec** types).

Vertex attributes must be bound to a free 'slot,' which can be obtained via the OpenGL API, but this the program must be linked or else it will not work (as it is the link operation that decides which slots are free/not free):

```
GLint glGetAttribLocationARB(GLhandleARB program, const GLcharARB
*name)
```

For example:

```
int location = glGetAttribLocationARB(myProgram,"tangent");
```

This method will always retrieve a free slot, unless they are all occupied (we can force slots with the **glBindAttribLocationARB** function, but it's better to ask OpenGL for a free one).

Once a slot has been established, we can then send the attributes to OpenGL just like standard attributes and vertices. There are two ways to do this: immediate mode and vertex arrays.

Immediate mode:
Using **glVertexAttrib*ARB** between a **glBegin/glEnd** pairing, like **glNormal** or **glTexCoord** :

```
int location = glGetAttribLocationARB(myProgram,"myattrib");
glBegin(GL_TRIANGLES);
  glVertexAttrib3fARB(location,0,1,0); glVertex3f(0,0,0);
  glVertexAttrib3fARB(location,1,0,0); glVertex3f(0,0,0);
  glVertexAttrib3fARB(location,0,1,1); glVertex3f(0,0,0);
glEnd();
```

Vertex arrays:
To use a vertex array, you must first enable it for both the server and client states:

```
glActiveTextureARB(GL_TEXTURE5_ARB);
glClientActiveTextureARB(GL_TEXTURE5_ARB);
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glVertexPointer(3,GL_FLOAT,0,geometry);
glNormalPointer(GL_FLOAT,0,normals);
//draw stuff
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
glActiveTextureARB(GL_TEXTURE0_ARB);
glClientActiveTextureARB(GL_TEXTURE0_ARB);
```

To use generic vertex attribute arrays, a location is needed, which allows you to setup your arrays easily:

```
glActiveTextureARB(GL_TEXTURE5_ARB);
glClientActiveTextureARB(GL_TEXTURE5_ARB);

int location = glGetAttribLocationARB(myProgram,"myattrib");

glEnableVertexAttribArrayARB(location);
glVertexAttribPointerARB(location,size,type,false,0,myarray);
//draw some stuff
glDisableVertexAttribArrayARB(location);

glActiveTextureARB(GL_TEXTURE0_ARB);
glClientActiveTextureARB(GL_TEXTURE0_ARB);
```

Though this example may not be very good (as it assumes vertex attributes only, without vertices, so no drawing is actually performed), the objective is to show how to correctly setup a generic vertex attribute array.

The most complex function in this example is **glVertexAttribPointerARB**, which works the same as **glNormalPointer** but is generic.

A complete description of this function can be found at:
http://developer.3dlabs.com/openGL2/slapi/VertexAttribPointerARB.htm

How to Use the 3DSMax Mesh Exporter

Though Shader Designer has a series of default meshes, you may wish to import your own geometry. This can be done using the Flexporter (http://www.codercorner.com/Flexporter.htm) a plug-in manager for 3DSMax5, with the Shader Designer's export plugin.

This export plugin allows you to export an entire scene (without animations) to a GSD (Shader Designer mesh format) file. However, there are a couple of items to take into consideration before exporting any data:

- The default view frustum in Shader Designer is znear=1, zfar=500, fov=45, while the camera position is at (0,0,3) pointing at (0,0,0). If your mesh is too large and not centered correctly at (0,0,0), you may not see anything. If this is the case, reduce the mesh until it fits within a bounding box of 3 units, then place it (the scene) at the origin.

- Remember to perform a resetXform on every submesh/object of your scene before exporting it.

References:

Shader Designer homepage: http://www.TyphoonLabs.com

OpenGL Shading Language API man pages, GLSL examples and resources: http://developer.3dlabs.com

GameInstitute homepage: http://www.clockworkcoders.com/oglsl/tutorials.html

GLSL Shaders: http://3dshaders.com

GLSL Tutorials: http://www.clockworkcoders.com