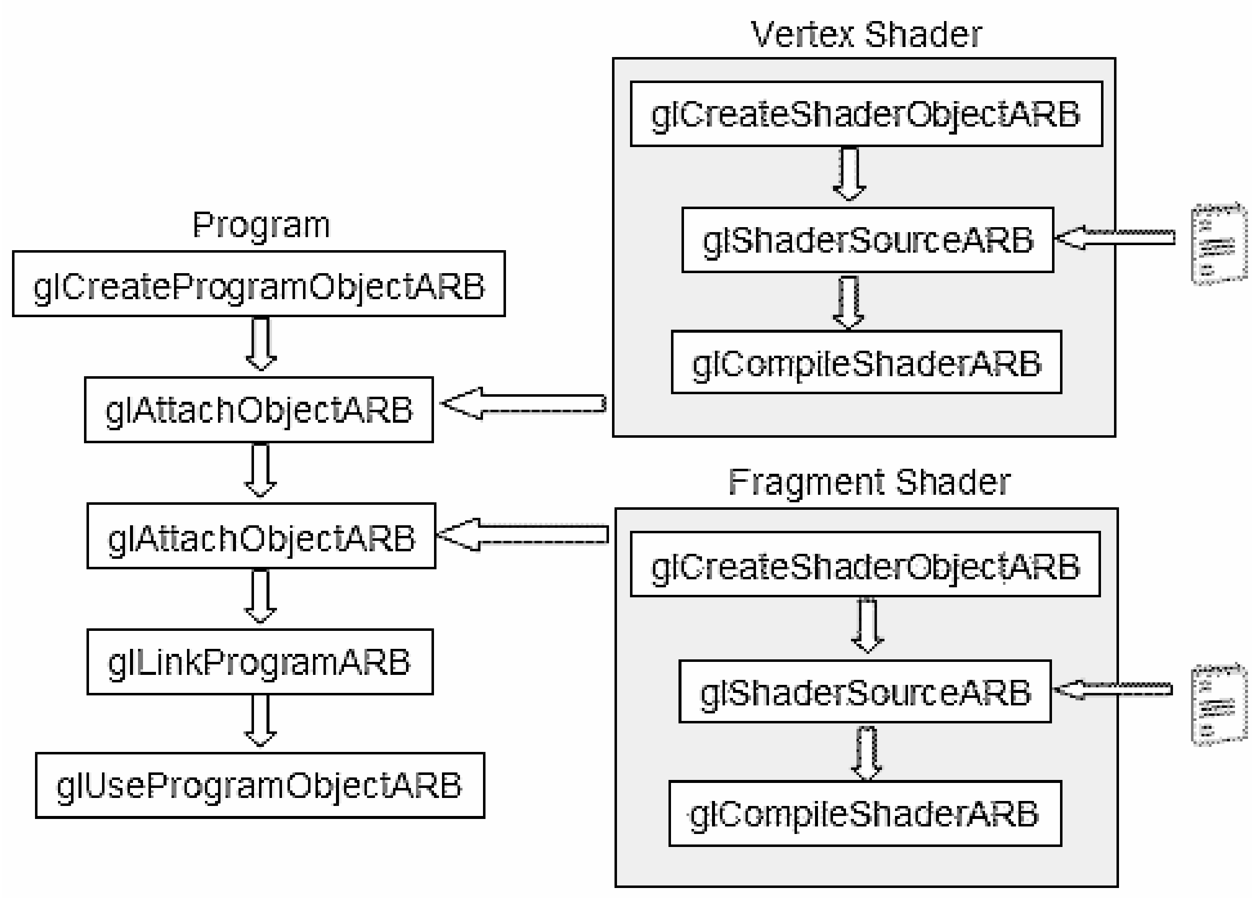
Introduction to GLSL

Marco Benvegnù
hiforce@gmx.it

The Overall Process



Creating a Shader

- The first step is creating an object which will act as a shader container. The function available for this purpose returns a handle for the container

```
GLhandleARB glCreateShaderObjectARB(GLenum shaderType);
```

Parameter:

shaderType - GL_VERTEX_SHADER_ARB or GL_FRAGMENT_SHADER_ARB.

Adding the source code

void glShaderSourceARB(GLhandleARB shader, int numOfStrs, const char **strings, int *lenOfStrs);

Parameters:

shader	**- the handler to the shader.**
numOfStrings	**- the number of strings in the array.**
strings	**- the array of strings.**
lenOfStrs	**- an array with the length of each string, or NULL if NULL terminated.**

Compiling

- The final step, the shader must be compiled.
- The function to achieve this is:

```
void glCompileShaderARB(GLhandleARB program);
```

Parameters:

program - the handler to the program.

Creating a Program

- The first step is creating an object which will act as a program container.

```
GLhandleARB glCreateProgramObjectARB(void);
```

- One can create as many programs as needed. Once rendering, you can switch from program to program, and even go back to fixed functionality during a single frame.
  - For instance one may want to draw a teapot with refraction and reflection shaders, while having a cube map displayed for background using OpenGL's fixed functionality.

Creating a Program

- The 2nd step is to attach the shaders to the program you've just created.
- The shaders do not need to be compiled nor is there a need to have src code. For this step only the shader container is required

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

Parameters:

program - the handler to the program.
shader - the handler to the shader you want to attach.

- If you have a pair vertex/fragment of shaders you'll need to attach both to the program (call attach twice).
- You can have many shaders of the same type (vertex or fragment) attached to the same program, but only one of them can define the main() function.

Creating a Program

- The final step is to link the program.

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

Parameters:

program - the handler to the program.

Using a Program

- Each program is assigned an handler, and you can have as many programs linked and ready to use as you want (and your hardware allows).

void glUSeProgramObjectARB(GLhandleARB prog);

Parameters:

prog - the handler to the program to use, or zero to return to fixed functionality

Summing up

```
void setShaders()
{
    const char *vs,*fs;

    GLhandleARB v = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
    glLoadShaderSource(v, "phong.vert");
    glCompileShaderARB(v);

    GLhandleARB f = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
    glLoadShaderSource(f, "phong.frag");
    glCompileShaderARB(f);

    p = glCreateProgramObjectARB();
    glAttachObjectARB(p,v);
    glAttachObjectARB(p,f);
    glLinkProgramARB(p);
}
```

Cleaning Up

- A function to detach a shader from a program is:

  void glDetachObjectARB(GLhandleARB program, GLhandleARB shader);

  Parameter:

  program - The program to detach from.

  shader - The shader to detach.

- To delete a shader use the following function:
- Only shaders that are not attached can be deleted

  void glDeleteShaderARB(GLhandleARB shader);

  Parameter:

  shader - The shader to delete.

Getting Error

- There is alos an info log function that returns compile & linking information, errors

```
void glGetInfoLogARB(GLhandleARB object,
                     GLsizei maxLength,
                     GLsizei *length,G
                     GLcharARB *infoLog);
```

GLSL Data Types

- Three basic data types in GLSL:
  - float, bool, int
  - float and int behave just like in C,and bool types can take on the values of true or false.
- Vectors with 2,3 or 4 components, declared as:
  - vec{2,3,4}:   a vector of 2, 3,or 4 floats
  - bvec{2,3,4}: bool vector
  - ivec{2,3,4}: vector of integers
- Square matrices 2x2, 3x3 and 4x4:
  - mat2
  - mat3
  - mat4

GLSL Data Types

- A set of special types are available for texture access, called sampler
  - sampler1D - for 1D textures
  - sampler2D - for 2D textures
  - sampler3D - for 3D textures
  - samplerCube - for cube map textures
- Arrays can be declared using the same syntax as in C, but can't be initialized when declared. Accessing array's elements is done as in C.
- Structures are supported with exactly the same syntax as C

GLSL Variables

- Declaring variables in GLSL is mostly the same as in C

```
float a,b; // two vector (yes, the comments are like in C)
int c = 2; // c is initialized with 2
bool d = true; // d is true
```

- Differences: GLSL relies heavily on constructor for initialization and type casting

```
float b = 2; // incorrect, there is no automatic type casting
float e = (float)2;// incorrect, requires constructors for type casting
int a = 2;
float c = float(a); // correct. c is 2.0
vec3 f; // declaring f as a vec3
vec3 g = vec3(1.0,2.0,3.0); // declaring and initializing g
```

- Initializing variables using other variables

```
vec2 a = vec2(1.0,2.0);
vec2 b = vec2(3.0,4.0);
vec4 c = vec4(a,b) // c = vec4(1.0,2.0,3.0,4.0);
vec2 g = vec2(1.0,2.0);
float h = 3.0;
vec3 j = vec3(g,h);
```

GLSL Variables

- Matrices also follow this pattern

```
mat4 m = mat4(1.0) // initializing the diagonal of the matrix with 1.0
vec2 a = vec2(1.0,2.0);
vec2 b = vec2(3.0,4.0);
mat2 n = mat2(a,b); // matrices are assigned in column major order
mat2 k = mat2(1.0,0.0,1.0,0.0); // all elements are specified
```

- The declaration and initialization of structures is demonstrated below

```
struct dirlight // type definition
{
   vec3 direction;
   vec3 color;
};
dirlight d1;
dirlight d2 = dirlight(vec3(1.0,1.0,0.0),vec3(0.8,0.8,0.4));
```

GLSL Variables

- Accessing a vector can be done using letters as well as standard C selectors.

```
vec4 a = vec4(1.0,2.0,3.0,4.0);
float posX = a.x;
float posY = a[1];
vec2 posXY = a.xy;
float depth = a.w;
```

- One can use the letters x,y,z,w to access vectors components; r,g,b,a for color components; and s,t,p,q for texture coordinates.

GLSL Variable Qualifiers

- Qualifiers give a special meaning to the variable. In GLSL the following qualifiers are available:
  - **const** - the declaration is of a compile time constant
  - **attribute** – (only used in vertex shaders, and read-only in shader) global variables that may change per vertex, that are passed from the OpenGL application to vertex shaders
  - **uniform** – (used both in vertex/fragment shaders, read-only in both) global variables that may change per primitive (may not be set inside glBegin,/glEnd)
  - **varying** - used for interpolated data between a vertex shader and a fragment shader. Available for writing in the vertex shader, and read-only in a fragment shader.

GLSL Statements

- Control Flow Statements:

```
if (bool expression)
  ...
else
  ...

for (initialization; bool expression; loop expression)
  ...

while (bool expression)
  ...

do
  ...
while (bool expression)
```

Note: only “if” are available on most current hardware

GLSL Statements

- A few jumps are also defined:
  - •continue - available in loops, causes a jump to the next iteration of the loop
  - •break - available in loops, causes an exit of the loop
  - •Discard - can only be used in fragment shaders. It causes the termination of the shader for the current fragment without writing to the frame buffer, or depth.

GLSL Functions

- As in C, a shader is structured in functions. At least each type of shader must have a main function declared with the following syntax: void main()
- User defined functions may be defined.
- As in C a function may have a return value, and use the return statement to pass out its result. A function can be void. The return type can have any type, except array.
- The parameters of a function have the following qualifiers:
  - **in** - for input parameters
  - **out** - for outputs of the function. The return statement is also an option for sending the result of a function.
  - **inout** - for parameters that are both input and output of a function
  - If no qualifier is specified, by default it is considered to be *in*.

GLSL Functions

- A few final notes:
  - A function can be overloaded as long as the list of parameters is different.
  - Recursion behavior is undefined by specification.
- Finally, let's look at an example

```
vec4 toonify(in float intensity)
{
        vec4 color;
        if (intensity > 0.98)
            color = vec4(0.8,0.8,0.8,1.0);
        else if (intensity > 0.5)
            color = vec4(0.4,0.4,0.8,1.0);
        else if (intensity > 0.25)
            color = vec4(0.2,0.2,0.4,1.0);
        else color = vec4(0.1,0.1,0.1,1.0);
        return(color);
}
```

Uniform Variables

- Uniform variables, this is one way for your C program to communicate with your shaders (e.g. what time is it since the bullet was shot?)
- A uniform variable can have its value changed by primitive only, i.e., its value can't be changed between a *glBegin* / *glEnd* pair.
- Uniform variables are suitable for values that remain constant along a primitive, frame, or even the whole scene.
- Uniform variables can be read (but not written) in both vertex and fragment shaders.

Uniform Variables

- The first thing you have to do is to get the memory location of the variable.
  - Note that this information is only available after you link the program. With some drivers you may be required to be using the program, i.e. *glUSeProgramObjectARB* is already called
- The function to use is:

GLint glGetUniformLocationARB(GLhandleARB program, const char *name);

Parameters:

program - the handler to the program

name - the name of the variable.

The return value is the location of the variable, which can be used to assign values to it.

Uniform Variables

- Then you can set values of uniform variables with a family of functions.
- A set of functions is defined for setting float values as below. A similar set is available for int's, just replace "f" with "i"

```
void glUniform1fARB(GLint location, GLfloat v0);
void glUniform2fARB(GLint location, GLfloat v0, GLfloat v1);
void glUniform3fARB(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
void glUniform4fARB(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);

GLint glUniform{1,2,3,4}fvARB(GLint location, GLsizei count, GLfloat *v);
```

Parameters:

location - the previously queried location.
v0,v1,v2,v3 - float values.
count - the number of elements in the array
v - an array of floats.

Uniform Variables

- Matrices are also an available data type in GLSL, and a set of functions is also provided for this data type:

GLint glUniformMatrix{2,3,4}fvARB(GLint location, GLsizei count, GLboolean transpose, GLfloat *v);

Parameters:

location - the previously queried location.

count - the number of matrices. 1 if a single matrix is being set, or *n* for an array of *n* matrices.

transpose - 1 for row major order, 0 for column major order

v - an array of floats.

Uniform Variables

- Note: the values that are set with these functions will keep their values until the program is linked again.
- Once a new link process is performed all values will be reset to zero.

Uniform Variables

- A sample:

Assume that a shader with the following variables is being used:

```
uniform float specIntensity;
uniform vec4 specColor;
uniform float t[2];
uniform vec4 colors[3];
```

In the application, the code for setting the variables could be:

```
GLint loc1,loc2,loc3,loc4;
float specIntensity = 0.98;
float sc[4] = {0.8,0.8,0.8,1.0};
float threshold[2] = {0.5,0.25};
float colors[12] = {0.4,0.4,0.8,1.0, 0.2,0.2,0.4,1.0, 0.1,0.1,0.1,1.0};
loc1 = glGetUniformLocationARB(p,"specIntensity");
glUniform1fARB(loc1,specIntensity);
loc2 = glGetUniformLocationARB(p,"specColor");
glUniform4fvARB(loc2,1,sc);
loc3 = glGetUniformLocationARB(p,"t");
glUniform1fvARB(loc3,2,threshold);
loc4 = glGetUniformLocationARB(p,"colors");
glUniform4fvARB(loc4,3,colors);
```

Attribute Variables

- Attribute variables also allow your C program to communicate with shaders
- Attribute variables can be updated at any time, but can only be read (not written) in a vertex shader.
- Attribute variables pertain to vertex data, thus not useful in fragment shader
- To set its values, (just like uniform variables) it is necessary to get the location in memory of the variable.

```
GLint glGetAttribLocationARB(GLhandleARB program,char *name);
Parameters:
    program - the handle to the program.
    name - the name of the variable
```

Attribute Variables

- As uniform variables, a set of functions are provided to set attribute variables (replace “f” with “i” for integers)

```
void glVertexAttrib1fARB(GLint location, GLfloat v0);
void glVertexAttrib2fARB(GLint location, GLfloat v0, GLfloat v1);
void glVertexAttrib3fARB(GLint location, GLfloat v0, GLfloat v1,GLfloat v2);
void glVertexAttrib4fARB(GLint location, GLfloat v0, GLfloat v1,,GLfloat v2, GLfloat v3);
```

or

```
GLint glVertexAttrib{1,2,3,4}fvARB(GLint location, GLfloat *v);
```

Parameters:

location - the previously queried location.

v0,v1,v2,v3 - float values.

v - an array of floats.

Attribute Variables

- A sample snippet

Assuming the vertex shader has:

```
attribute float height;
```

In the main Opengl program, we can do the following:

```
loc = glGetAttribLocationARB(p,"height");
glBegin(GL_TRIANGLE_STRIP);
glVertexAttrib1fARB(loc,2.0);
glVertex2f(-1,1);
glVertexAttrib1fARB(loc,2.0);
glVertex2f(1,1);
glVertexAttrib1fARB(loc,-2.0);
glVertex2f(-1,-1);
glVertexAttrib1fARB(loc,-2.0);
glVertex2f(1,-1); glEnd();
```

Sample vertex shader

```
uniform vec4 lightPos;

varying vec3 normal;
varying vec3 lightVec;
varying vec3 viewVec;

void main(){
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
   vec4 vert = gl_ModelViewMatrix * gl_Vertex;

   normal   = gl_NormalMatrix * gl_Normal;
   lightVec = vec3(lightPos - vert);
   viewVec  = -vec3(vert);
}
```

Sample fragment shader

```
varying vec3 normal;
varying vec3 lightVec;
varying vec3 viewVec;

void main(){
     vec3 norm = normalize(normal);

     vec3 L = normalize(lightVec);
     vec3 V = normalize(viewVec);
     vec3 halfAngle = normalize(L + V);

     float NdotL = dot(L, norm);
     float NdotH = clamp(dot(halfAngle, norm), 0.0, 1.0);

     // "Half-Lambert" technique for more pleasing diffuse term
     float diffuse  = 0.5 * NdotL + 0.5;
     float specular = pow(NdotH, 64.0);

     float result = diffuse + specular;

     gl_FragColor = vec4(result);
}
```

Built-in variables

- Attributes & uniforms
- For ease of programming
- OpenGL state mapped to variables
- Some special variables are required to be written to, others are optional

Special built-ins

- Vertex shader

```
vec4  gl_Position;       // must be written
vec4  gl_ClipPosition;   // may be written
float gl_PointSize;      // may be written
```

- Fragment shader

```
float gl_FragColor;      // may be written
float gl_FragDepth;      // may be read/written
vec4  gl_FragCoord;      // may be read
bool  gl_FrontFacing;    // may be read
```

Built-in Attributes

- Vertex shader

```
attribute vec4  gl_Vertex;
attribute vec3  gl_Normal;
attribute vec4  gl_Color;
attribute vec4  gl_SecondaryColor;
attribute vec4  gl_MultiTexCoordn;
attribute float gl_FogCoord;
```

Built-in Uniforms

```
uniform   mat4  gl_ModelViewMatrix;
uniform   mat4  gl_ProjectionMatrix;
uniform   mat4  gl_ModelViewProjectionMatrix;
uniform   mat3  gl_NormalMatrix;
uniform   mat4  gl_TextureMatrix[n];

struct gl_MaterialParameters
{
  vec4  emission;
  vec4  ambient;
  vec4  diffuse;
  vec4  specular;
  float shininess;
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```

Built-in Uniforms

```
struct gl_LightSourceParameters
{
  vec4  ambient;
  vec4  diffuse;
  vec4  specular;
  vec4  position;
  vec4  halfVector;
  vec3  spotDirection;
  float spotExponent;
  float spotCutoff;
  float spotCosCutoff;
  float constantAttenuation
  float linearAttenuation
  float quadraticAttenuation
};
Uniform gl_LightSourceParameters gl_LightSource[gl_MaxLights];
```

Built-in Varyings

```
varying   vec4  gl_FrontColor        // vertex
varying   vec4  gl_BackColor;        // vertex
varying   vec4  gl_FrontSecColor;    // vertex
varying   vec4  gl_BackSecColor;     // vertex

varying   vec4  gl_Color;            // fragment
varying   vec4  gl_SecondaryColor;   // fragment

varying   vec4  gl_TexCoord[];       // both
varying   float gl_FogFragCoord;     // both
```

Built-in functions

- Angles & Trigonometry
  - **radians, degrees, sin, cos, tan, asin, acos, atan**
- Exponentials
  - **pow, exp2, log2, sqrt, inversesqrt**
- Common
  - **abs, sign, floor, ceil, fract, mod, min, max, clamp**

Built-in functions

- Interpolations
  - **mix**(x,y,a)   **x*(1.0-a) + y*a**)
  - **step**(edge,x)   **x <= edge ? 0.0 : 1.0**
  - **smoothstep**(edge0,edge1,x)

```
t = (x-edge0)/(edge1-edge0);
t = clamp( t, 0.0, 1.0);
return t*t*(3.0-2.0*t);
```

Built-in functions

- Geometric
  - **length, distance, cross, dot, normalize, faceForward, reflect**
- Matrix
  - **matrixCompMult**
- Vector relational
  - **lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual, notEqual, any, all**

Built-in functions

- Texture
  - **texture1D, texture2D, texture3D, textureCube**
  - **texture1DProj, texture2DProj, texture3DProj, textureCubeProj**
  - **shadow1D, shadow2D, shadow1DProj, shadow2Dproj**
- Vertex
  - **ftransform**