CMSC427 Notes on GLSL shading language

These notes have three purposes. To give references and readings on GLSL, and to give a very terse, quick intro to key elements.

Readings:
http://www.shaderific.com/glsl/
http://learnwebgl.brown37.net/12_shader_language/glsl_introduction.html
https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.3.30.pdf

Language structure and statements:
GLSL is structured like C/Java, with similar control structures including functions, if-elses, while and do-while loops, for loops and switches. Statements, declarations and comments are also similar, as is variable scope. A vanilla vertex shader has the form

```
// Version # should match what you have on your computer
#version 410

void main()
{
     color = vec4( position, 1.0f );
}
```

Data types:
The primitive data types in GLSL are **int**, **float**, **double**, **bool** and **uint**. Since GLSL is designed for numeric computation there are no chars, strings. Nor are there shorts, longs or other numeric type variants to complicate the language. There are arrays and structs, no objects.

For our purposes the interesting types are vectors (vecs) and matrices (mat), particularly floating point. They behave like a combination of arrays, objects and structs. You declare one as:

```
vec3 position = vec3(1.0, 1.0, 1.0);
```

You can use extend a vec3 to a vec4 by placing the vec3 in place of the first three positions:

```
vec4 color = vec3(normal, 1.0);
```

And you can name components of a vec, here using a normal vector to make a color:

```
vec4 color = vec4(normal.x, normal.y, normal.z, 1.0f);
```

One of the unique elements of GLSL is *swizzling*, using multiple element names to get a subsequence of a vector. Assume here color is a vec4, so this gives us the first 3 elements:

```
vec3 aVec3 = color.xyz; // color.xyz is a vec3
```

This line would normalize a vec4 homogenous vector:

```
vec4 h = vec4(color.xyz/color.w, 1.0f);
```

Since vecs can represent positions, colors and texture coordinates, the elements have three alternative names:

Position	**xyzw**
Color	**rgba**
Texture	**stpq**

Since texture coordinates are usually 2D, the **st** are the standard names – the **pq** is for second texture, or an orientation vector.

Swizzling has one additional trick – you can reorder or repeat components in a swizzle expression, here resetting color to green, red, red.

```
color = color.grr;
```

Matrices can be constructing by appending vectors, as in the following:

```
mat4 mv = mat4( xc, yc, zc, d );
```

Vector and matrix operations:
Vectors and matrices support appropriate vector and scalar operations:

```
gl_position = mv * vertex;

color = 0.4* color; // assume color, specular, diffuse vec4

color = specular + diffuse;

float cosTheta = dot(light, normal);

vec4 = reflect(normal, light);
```

The vector operations include standards like dot, cross, normalize, distance, length.

Numeric operations like sin, log, sqrt work component-wise on vectors:

```
color = sqrt(color);
```

But note that if you operate on all components of a color vector you modify the alpha channel, which you may not want to do. This is an alternative that keeps the alpha channel (transparency) at 1.0 by taking out the rgb, operating on it, and putting the vec4 back together:

```
color = vec4(sqrt(color.rgb),1.0f);
```

Pipeline and data passing:
Understanding shading programs requires understanding how data is passed from the client Java program to the vertex shader to the fragment shader. For us this version of the pipeline is adequate for the moment:

https://en.wikibooks.org/wiki/GLSL_Programming/OpenGL_ES_2.0_Pipeline

To get data from the client program into the vertex shader we have two options. The project 1 example code uses both.

1) Use Vertex Buffer Objects (VBOs) to get a sequence of vertices into the shader. The layout location associates the variable in the shader with the appropriate VBO in the client.

```
gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
gl.glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl.glEnableVertexAttribArray(0); // This 0 matches location 0
```

2) Use Uniforms to get a single value into a variable or matrices. This code in the client code passes **mvMat** to **mv_matrix** in the shader

```
int mv_loc = gl.glGetUniformLocation(rendering_program, "mv_matrix");
gl.glUniformMatrix4fv(mv_loc, 1, false, mvMat.getFloatValues(), 0);
```

Then in the shader we have:

```
#version 410

layout (location = 0) in vec3 position;
layout (location = 1) in vec2 tex_coord;
out vec2 tc;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;

void main(void)
{    gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
     tc = tex_coord;
}
```

To get data from the vertex to fragment shader we have the qualifier out in the vertex shader that connects a variable to an identically named in variable in the fragment shader:

```
#version 410
in vec2 tc;
out vec4 color;
uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
uniform sampler2D samp;
void main(void)
{   color = texture(samp,tc);  }
```