What is GLM?

GLM is a set of C++ classes and functions to fill in the programming gaps in writing the basic vector and matrix mathematics for OpenGL applications. GLM isn’t really a library – it is all specified in *.hpp header files so that it gets compiled in with your source code.

You can find it at:

http://glm.g-truc.net/0.9.8.5/

You invoke GLM like this:

```cpp
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include  <glm/gtc/matrix_transform.hpp>
```

Or, you can #include only the specific GLM .hpp files you need.

If GLM is not installed in a system place, put it somewhere you can get access to. Later on, these notes will show you how to use it from there.

### Why are we even talking about this?

The OpenGL overlords have “deprecated” some of the OpenGL functions we have been using to perform transformations. In the desktop world, it means that the use of such functions is discouraged. In the OpenGL world, it means these functions are gone. You might as well become familiar with how to live without them. So, instead of saying:

```
    gluLookAt( 0., 0., 3.,     0., 0., 0.,     0., 1., 0. );
    glRotatef( (GLfloat)Yrot, 0., 1., 0. );
    glRotatef( (GLfloat)Xrot, 1., 0., 0. );
    glScalef( (GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale );
```

for OpenGL, you would now say:

```
    glm::mat4 modelview;
    glm::vec3   eye(0.,0.,3.);
    glm::vec3    look(0.,0.,0.);
    glm::vec3   up(0.,1.,0.);
    modelview = glm::lookAt( eye, look, up );
    modelview = glm::rotate( modelview, D2R*Yrot, glm::vec3(0.,1.,0.) );
    modelview = glm::rotate( modelview, D2R*Xrot, glm::vec3(1.,0.,0.) );
    modelview = glm::scale( modelview, glm::vec3(Scale,Scale,Scale) );
    glMultMatrixf(   glm::value_ptr( modelview )   );
```

Exactly the same concept, but a different expression of it. Read on for details …

### The Most Useful GLM Variables, Operations, and Functions

#### Constructor:

```
glm::mat4( 1. ); // identity matrix
glm::vec4( );
glm::vec3( );
```

#### Multiplication – the * operator has been overloaded:

```
glm::mat4
glm::rotate( glm::mat4 const & m, float angle, glm::vec3 const & axis );
glm::mat4
glm::scale( glm::mat4 const & m, glm::vec3 const & factors );
glm::mat4
glm::translate( glm::mat4 const & m, glm::vec3 const & translation );
```

#### Viewing:

```
glm::mat4
glm::ortho( float left, float right, float bottom, float top, float near, float far );
glm::mat4
glm::ortho( float left, float right, float bottom, float top );
glm::mat4
glm::frustum( float left, float right, float bottom, float top, float near, float far );
glm::mat4
glm::perspective( float fovy, float aspect, float near, float far );
```

#### Loading Matrices into OpenGL:

```
void LoadMatrix( glm::value_ptr( glm::mat4 ) );
gUniformMatrix4fv( Location, 1, GL_FALSE, glm::value_ptr( glm::mat4 ) );
```
Here's what that GLM folder looks like

Telling Linux about where the GLM folder is

```
g++ …
```

“minus-capital-eye-period” means “also look for the < > includes in this folder”

Instead of the period, you can list a full or relative pathname.

Telling Visual Studio about where the GLM folder is

1. A period, indicating that the project folder should also be searched when a `#include <xxx>` is encountered. If you put it somewhere else, enter that full or relative path instead.

Using Transformations, OpenGL-style, like in the sample.cpp Program

```c
glMatrixMode( GL_PROJECTION );
gLoadIdentity( );
if( WhichProjection == ORTHO )	glOrtho( -3., 3., -3., 3., 0.1, 1000. );
else
gluPerspective( 90., 1., 0.1, 1000. );
// place the objects into the scene:
gMatrixMode( GL_MODELVIEW );
gLoadIdentity( );
// set the eye position, look-at position, and up-vector:
gluLookAt( 0., 0., 3., 0., 0., 0., 0., 1., 0. );
// rotate the scene:
gRotatef((GLfloat)Yrot, 0., 1., 0.);
gRotatef((GLfloat)Xrot, 1., 0., 0.);
// uniformly scale the scene:
if( Scale < MINSCALE )
Scale = MINSCALE;
gScalef((GLfloat)Scale, (GLfloat)Scale, (GLfloat)Scale);
```

Using Transformations, GLM-style, like

```c
#include <glm/vec3.hpp>
#include <glm/mat4x4.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

const float D2R = M_PI/180.f; // 0.01745…

glMatrixMode( GL_PROJECTION );
gLoadIdentity( );
gluint projection;
if( WhichProjection == ORTHO )
projection = glm::ortho(-3., 3., -3., 3., 0.1, 1000.);
else
projection = glm::perspective( D2R*90., 1., 0.1, 1000.);
```

if convert degrees to radians:
```c
const float D2R = M_PI/180.f; // 0.01745…
```

```c
if apply the projection matrix:
gMatrixMode( glm::value_ptr(projection) );
```
// place the objects into the scene:
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// set the eye position, look-at position, and up-vector:
glm::vec3 eye(0.,0.,3.);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);
glm::mat4 modelview = glm::lookAt(eye, look, up);

// rotate the scene (warning -- unlike OpenGL's glRotatef, GLM's rotate method takes angles in "radians"):
modelview = glm::rotate(modelview, D2R*Yrot, glm::vec3(0.,1.,0.));
modelview = glm::rotate(modelview, D2R*Xrot, glm::vec3(1.,0.,0.));

// uniformly scale the scene:
if (Scale < MINSCALE)
    Scale = MINSCALE;
modelview = glm::scale(modelview, glm::vec3(Scale, Scale, Scale));

// apply the modelview matrix:
glMultMatrixf(glm::value_ptr(modelview));

// GLM for Vulkan

// if Vulkan's projected Y is inverted from OpenGL's

glm::mat4 projection = glm::perspective(D2R*90., 1., 0.1, 1000.);

projection[1][1] *= -1.; // Vulkan's projected Y is inverted from OpenGL's

glm::vec3 eye(0.,0.,3.);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);
glm::mat4 view = glm::lookAt(eye, look, up);

glm::mat4 model(1.); // identity
model = glm::rotate(model, D2R*Yrot, glm::vec3(0.,1.,0.));
model = glm::rotate(model, D2R*Xrot, glm::vec3(1.,0.,0.));

In the shader:

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;

mat4 PVM = projectionMatrix * viewMatrix * modelMatrix;

gl_Position = PVM * gl_Vertex;

In the C/C++ program:

Pattern->Use();
Pattern->SetUniformVariable("projectionMatrix", projection);
Pattern->SetUniformVariable("viewMatrix", view);
Pattern->SetUniformVariable("modelMatrix", model);

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;

mat4 PVM = projectionMatrix * viewMatrix * modelMatrix;

gl_Position = PVM * gl_Vertex;