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. However, even though it was written for OpenGL, it works fine with Vulkan.

Even though GLM looks like a library, it actually isn’t – 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>
#include <glm/gtc/matrix_inverse.hpp>
```

OpenGL treats all angles as given in degrees. This line forces GLM to treat all angles as given in radians.

I recommend this so that all angles you create in all programming will be in radians.

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?

All of the things that we have talked about being **deprecated** in OpenGL are really **deprecated** in Vulkan -- built-in pipeline transformations, begin-end, fixed-function, etc. So, where you might have said in OpenGL:

```gl
glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );
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 );
```

you would now say:

```gl
glm::mat4 modelview = glm::mat4( 1. ); // identity
glm::vec3 eye(0.,0.,3.);
glm::vec3 look(0.,0.,0.);
glm::vec3 up(0.,1.,0.);
modelview = glm::lookAt( eye, look, up ); // [mv] = [v]*[x,y,z]
modelview = glm::rotate( modelview, D2R*Yrot, glm::vec3(0.,1.,0.) ); // [mv] = [v]*[yr]*[x,y,z]
modelview = glm::rotate( modelview, D2R*Xrot, glm::vec3(1.,0.,0.) ); // [mv] = [v]*[yr]*[xr]*[x,y,z]
modelview = glm::scale( modelview, glm::vec3(Scale,Scale,Scale) ); // [mv] = [v]*[yr]*[xr]*[s]*[x,y,z]
```

This is 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( );

GLM recommends that you use the “glm:” syntax and avoid “using namespace” syntax because they have not made any effort to create unique function names

// multiplications:
glm::mat4 * glm::mat4
glm::mat4 * glm::vec4
glm::mat4 * glm::vec4( glm::vec3, 1. ) // promote a vec3 to a vec4 via a constructor

// emulating OpenGL transformations with concatenation:
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 );
```
The Most Useful GLM Variables, Operations, and Functions

// viewing volume (assign, not concatenate):

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 );

// viewing (assign, not concatenate):

glm::mat4 glm::lookAt( glm::vec3 const & eye, glm::vec3 const & look, glm::vec3 const & up );

Installing GLM into your own space

I like to just put the whole thing under my Visual Studio project folder so I can zip up a complete project and give it to someone else.
Here's what that GLM folder looks like

1. Telling Visual Studio about where the GLM folder is

2.
A *period*, indicating that the **project folder** should also be searched when a

```c
#include <xxx>
```

is encountered. If you put it somewhere else, enter that full or relative path instead.

---

### GLM in the Vulkan sample.cpp Program

```c
if( UseMouse )
{
  if( Scale < MINSCALE )
  
    Matrices.uModelMatrix = glm::mat4( 1. );  // identity
  
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Yrot, glm::vec3( 0.,1.,0.) );
    Matrices.uModelMatrix = glm::rotate( Matrices.uModelMatrix, Xrot, glm::vec3( 1.,0.,0.) );
    Matrices.uModelMatrix = glm::scale( Matrices.uModelMatrix, glm::vec3(Scale,Scale,Scale) );
    // done this way, the Scale is applied first, then the Xrot, then the Yrot
}
else
{
  if( ! Paused )
  
    const glm::vec3 axis = glm::vec3( 0., 1., 0. );
    Matrices.uModelMatrix = glm::rotate( glm::mat4( 1. ),  (float)glm::radians( 360.f*Time/SECONDS_PER_CYCLE ),   axis );
}
```

```c
glm::vec3 eye(0.,0.,EYEDIST );
glm::vec3 look(0.,0.,0.);
glm::vec3   up(0.,1.,0.);
Matrices.uVewMatrix = glm::lookAt( eye, look, up );
Matrices.uProjectionMatrix = glm::perspective( FOV, (double)Width/(double)Height, 0.1f, 1000.f );
Matrices.uProjectionMatrix[1][1] *= -1.;  // Vulkan’s projected Y is inverted from OpenGL
Matrices.uNormalMatrix = glm::inverseTranspose( glm::mat3( Matrices.uModelMatrix );  // note: inverseTransform!
fill05DataBuffer( MyMatrixUniformBuffer, (void *) &Matrices );
Misc.uTime = (float)Time;
Misc.uMode = Mode;
fill05DataBuffer( MyMiscUniformBuffer, (void *) &Misc );
```
### How Does this Matrix Stuff Really Work?

This is called a “Linear Transformation” because all of the coordinates are raised to the 1st power, that is, there are no $x^2$, $x^3$, etc. terms.

\[
\begin{align*}
x' &= Ax + By + Cz + D \\
y' &= Ex + Fy + Gz + H \\
z' &= lx + Jy + Kz + L
\end{align*}
\]

Or, in matrix form:

\[
\begin{pmatrix}
x' \\
y' \\
z'
\end{pmatrix} =
\begin{pmatrix}
A & B & C & D \\
E & F & G & H \\
I & J & K & L \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\]

### Transformation Matrices

**Translation**

\[
\begin{pmatrix}
x' \\
y' \\
z'
\end{pmatrix} =
\begin{pmatrix}
1 & 0 & 0 & T_x \\
0 & 1 & 0 & T_y \\
0 & 0 & 1 & T_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\]

**Scaling**

\[
\begin{pmatrix}
x' \\
y' \\
z'
\end{pmatrix} =
\begin{pmatrix}
S_x & 0 & 0 & 0 \\
0 & S_y & 0 & 0 \\
0 & 0 & S_z & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\]

**Rotation about X**

\[
\begin{pmatrix}
x' \\
y' \\
z'
\end{pmatrix} =
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\]

**Rotation about Y**

\[
\begin{pmatrix}
x' \\
y' \\
z'
\end{pmatrix} =
\begin{pmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\]

**Rotation about Z**

\[
\begin{pmatrix}
x' \\
y' \\
z'
\end{pmatrix} =
\begin{pmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z
\end{pmatrix}
\]

1/13/2020
The Rotation Matrix for an Angle ($\theta$) about an Arbitrary Axis ($A_x$, $A_y$, $A_z$)

$$
M = \begin{bmatrix}
A_x A_x + \cos \theta (1 - A_x A_x) & A_x A_y - \cos \theta A_y A_x - \sin \theta A_z & A_x A_z - \cos \theta A_z A_x + \sin \theta A_y \\
A_y A_x + \cos \theta A_x A_y + \sin \theta A_z & A_y A_y + \cos \theta (1 - A_y A_y) & A_y A_z - \cos \theta A_z A_y - \sin \theta A_x \\
A_z A_x - \cos \theta A_x A_z - \sin \theta A_y & A_z A_y + \cos \theta A_y A_z + \sin \theta A_x & A_z A_z + \cos \theta (1 - A_z A_z)
\end{bmatrix}
$$

For this to be correct, $A$ must be a unit vector
Compound Transformations

Q: Our rotation matrices only work around the origin. What if we want to rotate about an arbitrary point (A,B)?

A: We create more than one matrix.

\[
\begin{pmatrix}
  x' \\
  y' \\
  z'
\end{pmatrix}
= \begin{pmatrix}
  T_{A_1+B_1} & \cdot & T_{A_2+B_2} \\
  R_{\theta}
\end{pmatrix}
\begin{pmatrix}
  x \\
  y \\
  z
\end{pmatrix}
\]

Matrix Multiplication is not Commutative

- Rotate, then translate
- Translate, then rotate
Matrix Multiplication is Associative

\[
\begin{pmatrix}
    x' \\
    y' \\
    z' \\
    1
\end{pmatrix} = \left[ T_{+A,+B} \right] \cdot \left[ R_{\theta} \right] \cdot \left[ T_{-A,-B} \right] \cdot
\begin{pmatrix}
    x \\
    y \\
    z \\
    1
\end{pmatrix}
\]

One matrix – the Current Transformation Matrix, or CTM

One Matrix to Rule Them All

\[
\begin{pmatrix}
    x' \\
    y' \\
    z' \\
    1
\end{pmatrix} = \left[ T_{+A,+B} \right] \cdot \left[ R_{\theta} \right] \cdot \left[ T_{-A,-B} \right] \cdot
\begin{pmatrix}
    x \\
    y \\
    z \\
    1
\end{pmatrix}
\]

```cpp
glm::mat4 Model = glm::mat4(1.);
Model = glm::translate(Model, glm::vec3(A, B, 0.));
Model = glm::rotate(Model, thetaRadians, glm::vec3(Ax, Ay, Az));
Model = glm::translate(Model, glm::vec3(-A, -B, 0.));

glm::vec3 eye(0., 0., EYEDIST);
glm::vec3 look(0., 0., 0.);    glm::vec3 up(0., 1., 0.);
glm::mat4 View   = glm::lookAt(eye, look, up);
glm::mat4 Projection = glm::perspective(FOV, (double)Width/(double)Height, 0.1, 1000.);
Projection[1][1] *= -1.;

glm::mat3 Matrix = Projection * View * Model;
glm::mat3 NormalMatrix = glm::inverseTranspose(glm::mat3(Model));
```
Why Isn’t The Normal Matrix exactly the same as the Model Matrix?

It is, if the Model Matrix is all rotations and uniform scalings, but if it has non-uniform scalings, then it is not. These diagrams show you why.

Original object and normal

Wrong!

Right!

\[
glm::mat3 \text{ NormalMatrix} = \text{glm::mat3(Model)};\]

\[
glm::mat3 \text{ NormalMatrix} = \text{glm::inverseTranspose(glm::mat3(Model))};\]