GLM

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:

```cpp
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:

```cpp
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 ); // \{x',y',z'} = [v]*{x,y,z}
modelview = glm::rotate( modelview, D2R*Yrot, glm::vec3(0.,1.,0.) ); // \{x',y',z'} = [v]*[yr]*{x,y,z}
modelview = glm::rotate( modelview, D2R*Xrot, glm::vec3(1.,0.,0.) ); // \{x',y',z'} = [v]*[yr]*[xr]*{x,y,z}
modelview = glm::scale( modelview, glm::vec3(Scale, Scale, Scale) ); // \{x',y',z'} = [v]*[yr]*[xr]*[s]*{x,y,z}
```

This is exactly the same concept as OpenGL, but a different expression of it. Read on for details...

The Most Useful GLM Variables, Operations, and Functions

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

### Viewing Volume

```cpp
glm::mat4 modelview = glm::frustum( float(left), float(right), float(bottom), float(top), float(near), float(far) );
```

### Viewing

```cpp
glm::mat4 modelview = 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.

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>
```

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.

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.
Here's what that GLM folder looks like

Telling Visual Studio about where the GLM folder is

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.

GLM in the Vulkan sample.cpp Program

If( UseMouse )
{
    if( Scale < MINSCALE )
    {
        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 );
        }
    }
    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 );
}

Or, in matrix form:

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

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.

Or, in matrix form:

\[
\begin{bmatrix}
    x' \\
    y' \\
    z'
\end{bmatrix} =
\begin{bmatrix}
    x \\
    y \\
    z
\end{bmatrix}
\]

How Does this Matrix Stuff Really Work?

Translation

\[
\begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]

Rotation about X

\[
\begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & \cos \theta & -\sin \theta & 0 \\
    0 & \sin \theta & \cos \theta & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]

Rotation about Y

\[
\begin{bmatrix}
    \cos \theta & 0 & \sin \theta & 0 \\
    0 & 1 & 0 & 0 \\
    -\sin \theta & 0 & \cos \theta & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]

Rotation about Z

\[
\begin{bmatrix}
    \cos \theta & -\sin \theta & 0 & 0 \\
    \sin \theta & \cos \theta & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\]
The Rotation Matrix for an Angle ($\theta$) about an Arbitrary Axis ($Ax, Ay, Az$)

\[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

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.

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

One matrix to rule them all — the Current Transformation Matrix, or **CTM**

### Matrix Multiplication is Associative

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

### From the Data Buffer Noteset

```
layout( std140, set = 0, binding = 0 ) uniform sceneMatBuf
{
  mat4 uProjectionMatrix;
  mat4 uViewMatrix;
  mat4 uSceneMatrix;
}

layout( std140, set = 1, binding = 0 ) uniform objectMatBuf
{
  mat4 uModelMatrix;
  mat4 uNormalMatrix;
}
```

Here’s the vertex shader shader code to use the matrices:

```
  vNormal = uNormalMatrix * aNormal;
  gl_Position = uProjectMatrix * uViewMatrix * uSceneMatrix * uModelMatrix * aVertex;
```
Why isn't the Normal Matrix exactly the same as the Model Matrix?

```cpp
uNormalMatrix = glm::inverseTranspose( glm::mat3(Model) );
```

It is, if the Model Matrices are all rotations and uniform scalings, but if it has nonuniform scalings, then it is not. These diagrams show you why.