Computer Graphics

The OpenGL Mathematics (GLM) Library

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a
Creative Commons
Attribution-NonCommercial-NoDerivatives 4.0
International License

What is GLM?

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

You can find GLM at:
http://glm.g-truc.net/0.9.8.5/
or you can get a zip file of it on our Class Resources page.

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 mobile world of OpenGL-ES, it means those 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:

elm::mat4 modelview;
elm::vec3 eye(0.,0.,3.);
elm::vec3 look(0.,0.,0.);
elm::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) );

Exactly the same concept, 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 not use “using namespace” syntax because they have not made any effort to create unique function names.

If constructor:
glm::mat4( 1. ); // identity matrix
glm::vec4();

If multiplications – the * operator has been overloaded:
glm::mat4 *= glm::mat4

Emulating OpenGL transformations with concatenation:
glm::mat4 glm::translate( glm::vec3 const & translation );
glm::mat4 glm::scale( glm::vec3 const & factors );
glm::mat4 glm::rotate( glm::vec3 const & axis, float angle );
glm::mat4 glm::perspective( float fovy, float aspect, float near, float far);

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

Telling Linux about where the GLM folder is

```
p++ ... . . .
```

‘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 <xx> 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

```
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:
glMatrixMode( 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, I

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

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

// apply the projection matrix:
gMat4(Mat4) glm::value_ptr( projection );
```

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

// apply the projection matrix:
gMat4(Mat4) glm::value_ptr( projection );
```
Using Transformations, GLM-style, II

// place the objects into the scene:
glm::mat4 modelview = glm::identity();

// set the eye position, look-at position, and up-vector:
gl::vec3 eye(0.,0.,3.);
gl::vec3 look(0.,0.,0.);
gl::vec3 up(0.,1.,0.);
gl::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:
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;
vec4 Position = PVM * vec3(Vertex);

In the C/C++ program:

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.);
gl::vec3 look(0.,0.,0.);
gl::vec3 up(0.,1.,0.);
gl::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.));

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