GLM: Manual

Version 0.9.0
April 30, 2010

Christophe Riccio
g.truc.creation[at]gmail.com

Summary

1. Introduction

OpenGL Mathematics (GLM) is a C++ mathematics library for 3D applications based on the OpenGL Shading Language (GLSL) specification.

GLM provides 3D programmers with math classes and functions that are similar to GLSL or any high level GPU programming language. The idea is to have a library that has identical naming conventions and functionalities than GLSL so that when developers know GLSL, they know how to use GLM.

However, this project isn't limited by GLSL features. An extension system, based on the GLSL extension conventions, allows extended capabilities.

This library can be used with OpenGL but also for software rendering (Raytracing / Rasterisation), image processing and as much contexts as a simple math library could be used for.

GLM is written as a platform independent library and supports the following compilers:

- GNU GCC 3.4 and higher
- Microsoft Visual Studio 8.0 and higher

The source code is under the MIT licence.

Any feedback is welcome and can be sent to g.truc.creation[at]gmail.com.

2. Getting started

2.1. Compiler setup

GLM is a header library. Therefore, it doesn't require to be built separately. GLM usage is achieved by simply directing the compiler to add the GLM install path to the include search paths. (-I option with GCC) Another option is to copy the GLM files directly into the project source directory.

GLM is a header only library that makes heavy usages of C++ templates. This design may significantly increase the compile time for files that use GLM. Precompiled headers are recommended to avoid this issue.

2.2. Core features

After initial compiler setup, all core features of GLM (core GLSL features) can be accessed by including the `glm.hpp` header. The line: `#include <glm/glm.hpp>` is used for a typical compiler setup.

Note that by default there are no dependencies on external headers like `gl.h`, `gl3.h`, `glu.h` or `windows.h`.

2.3. Setup of swizzle operators

A common feature of shader languages like GLSL is components swizzling. This involves being able to select which components of a vector are used and in what order. For example, "variable.x", "variable.xxy", "variable.zxyy" are examples of swizzling.

However in GLM, swizzling operators are disabled by default. To enable swizzling the define `GLM_SWIZZLE` must be defined to one of `GLM_SWIZZLE_XYZW`, `GLM_SWIZZLE_RGBA`, `GLM_SWIZZLE_STQP` or `GLM_SWIZZLE_FULL` depending on what swizzle syntax is required.

The swizzle defines are supplied in the file `setup.hpp` and a simple way of enabling swizzling is to edit this file. However to avoid settings being lost on future GLM upgrades, it is suggested that `setup.hpp` be included first, then custom settings and finally `glm.hpp`. For example:

```
#include  <glm/setup.hpp>
#define GLM_SWIZZLE  GLM_SWIZZLE_FULL
#include  <glm/glm.hpp>
```

These custom setup lines can then be placed in a common project header or pre-compiled header.

2.4. Use sample of GLM core

```
#include <glm/glm.hpp>

int foo()
{
	glm::vec4 Position = glm::vec4(glm::vec3(0.0), 1.0);

	glm::mat4 Model = glm::mat4(1.0);
	Model[4] = glm::vec4(1.0, 1.0, 0.0, 1.0);
	glm::vec4 Transformed = Model * Position;

	return 0;
}
```

2.5. GLM Extensions

GLM extends the core GLSL feature set with extensions. These extensions include: quaternion, transformation, spline, matrix inverse, color spaces, etc.
Note that some extensions are incompatible with other extension as and may result in C++ name collisions when used together.

GLM provides two methods to use these extensions.

This method simply requires the inclusion of the extension implementation filename. The extension features are added to the `glm` namespace.

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

int foo()
{
	glm::vec4 Position = glm::vec4(glm:: vec3(0.0f), 1.0f);

	glm::mat4 Model = glm::translate(1.0f, 1.0f, 1.0f);
	glm::vec4 Transformed = Model * Position;

	return 0;
}
```

2.6. Dependencies

When `<glm/glm.hpp>` is included, GLM provides all the GLSL features it implements in C++.

By including `<glm/ext.hpp>` all the features of all extensions of GLM are included.

When you include a specific extension, all the dependent extensions will be included as well. All the extensions depend on GLM core. (`<glm/glm.hpp>`)

There is no dependence with external libraries. However, if `<boost/static_assert.hpp>` is included boost static assert will be used all over GLM code.

3. FAQ

3.1 Why GLM follows GLSL specification and conventions?

Following GLSL conventions is a really strict policy of GLM. GLM has been designed following the idea that everyone does its own math library with his own conventions. The idea is that brilliant developers (the OpenGL ARB) worked together and agreed to make GLSL. Following GLSL conventions is a way to find consensus. Moreover, basically when a developer knows GLSL, he knows GLM.

3.2. Would it be possible to add my feature?

YES. Every feature request could be added by submitting it here:
https://sourceforge.net/apps/trac/ogl-math/newticket

These requests would mainly take the form of extensions and if you provide an implementation, the feature will be added automatically in the next GLM release.

A SourceForge.net account is required to create a ticket.

3.3. Does GLM run GLSL program?

No, GLM is a C++ implementation of a subset of GLSL.

3.4. Does a GLSL compiler build GLM codes?

Not directly but it can be easy to port. However, the difference between a shader and C++ program at software design level will probably make this idea unlikely or impossible.

3.5. Should I use GTX extensions?

GTX extensions are qualified to be experimental extensions. In GLM this means that these extensions might change from version to version without restriction. In practice, it doesn't really change except time to time. GTC extensions are stabled, tested and perfectly reliable in time. Many GTX extensions extend GTC extensions and provide a way to explore features and implementations before becoming stable by a promotion as GTC extensions. This is fairly the way OpenGL features are developed through extensions.

3.6. Would it be possible to change GLM to do glVertex3fv(glm::vec3(0))?

It's possible to implement such thing in C++ with the implementation of the appropriate cast operator. In this example it's likely because it would result as a transparent cast, however, most of the time it's really unlikely resulting of build with no error and programs running with unexpected behaviors.

`GLM_GTC_type_ptr` extension provide a safe solution:

```
glm::vec4 v(0);
glm::mat4 m(0);

glVertex3fv(glm::value_ptr(v))
glLoadMatrixfv(glm::value_ptr(m));
```

Another solution inspired by STL:

```
glVertex3fv(&v[0]);
glLoadMatrixfv(&m[0][0]);
```

3.7. Where can I ask my questions?

A good place is the OpenGL Toolkits forum on OpenGL.org:
http://www.opengl.org/discussion_boards/ubbthreads.php?ubb=postlist&Board=10&page=1

3.8. Where can I report a bug?

Just like feature requests:
https://sourceforge.net/apps/trac/ogl-math/newticket

A SourceForge account is required to create a ticket.

3.9. Where can I find the documentation of extensions?

The Doxygen generated documentation includes a complete list of all extensions available. Explore this documentation to get a complete view of all GLM capabilities!
http://glm.g-truc.net/html/index.html

4. Known issues

4.1. Swizzle operators

Enabling the swizzle operator can result in name collisions with the Win32 API. To prevent these issues, you can access to the internal swizzle operator functions without enabling the swizzle operator itself. This is done by defining:

```
#define GLM_SWIZZLE GLM_SWIZZLE_FUNC
```

4.2. `not` function

The GLSL keyword `not` is also a keyword in C++. To prevent name collisions, the GLSL not function has been implemented with the name `not_`.

4.3. `half` based types

GLM supports half float number types through the extension `GLM_GTC_half_float`. This extension provides the types `half`, `hvec*`, `hmat*x*` and `hquat*`.

Unfortunately, C++ norm doesn't support anonymous unions which limit `hvec*` vector components access to x, y, z and w.

However, Visual C++ does support anonymous unions. When `GLM_USE_ANONYMOUS_UNION` is define, it enables the support of all component names (x,y,z,w ; r,g,b,a ; s,t,p,q). With GCC it will result in a build error.

5. References

OpenGL 4.0 core specification:
http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf

GLSL 4.00 specification:
http://www.opengl.org/registry/doc/GLSLangSpec.4.00.8.clean.pdf

GLM HEAD snapshot:
http://ogl-math.git.sourceforge.net/git/gitweb.cgi?p=ogl-math/ogl-math;a=snapshot;h=HEAD;sf=tgz

GLM Trac:
https://sourceforge.net/apps/trac/ogl-math

GLM website:
http://glm.g-truc.net

G-Truc Creation page:
http://www.g-truc.net/project-0016.html