Manual

Version 0.9.3
23 November 2011

Christophe Riccio
glm@g-truc.net

Summary

Introduction

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

GLM provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL so that when a programmer knows GLSL, he knows GLM as well which makes it really easy to use.

This project isn't limited by GLSL features. An extension system, based on the GLSL extension conventions, provides extended capabilities: matrix transformations, quaternions, half-based types, random numbers, etc...

This library works perfectly with OpenGL but it also ensures interoperability with other third party libraries and SDK. It is a good candidate for software rendering (Raytracing / Rasterisation), image processing, physic simulations and any context that requires a simple and convenient mathematics library.

GLM is written in C++98 but can take advantage of C++11 when supported by the compiler. It is a platform independent library with no dependence and officially supports the following compilers:

- Clang 2.6 and higher
- CUDA 3.0 and higher
- GCC 3.4 and higher
- LLVM 2.3 through GCC 4.2 front-end and higher
- Visual Studio 2005 and higher
- Any conform C++98 or C++11 compiler

The source code is under the MIT licence.

Thanks for contributing to the project by submitting tickets for bug reports and feature requests. (SF.net account required). Any feedback is welcome at glm@g-truc.net.

1. Getting started

1.1. Setup

GLM is a header only library, there is nothing to build to use it which increases its cross platform capabilities.

To use GLM, a programmer only have to include <glm/glm.hpp>. This provides all the GLSL features implemented by GLM.

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.

1.2. 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[3] = glm::vec4(1.0, 1.0, 0.0, 1.0);
	glm::vec4 Transformed = Model * Position;
	return 0;
}
```

1.3. Dependencies

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

When an extension is included, 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 or external headers like gl.h, gl3.h, glu.h or windows.h. However, if <boost/static_assert.hpp> is included, Boost static assert will be used all over GLM code to provide compiled time errors.

1.4. 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(
		glm::mat4(1.0f), glm::vec3(1.0f));
	glm::vec4 Transformed = Model * Position;
	return 0;
}
```

1.5. OpenGL interoperability

It could be possible to implement `glVertex3fv(glm::vec3(0))` in C++ with the appropriate cast operator. It would result as a transparent cast in this example, however cast operator may result of programs running with unexpected behaviors without build error or any notification.

`GLM_GTC_type_ptr` extension provides a safe solution:

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

void foo()
{
	glm::vec4 v(0.0f);
	glm::mat4 m(1.0f);
	...
	glVertex3fv(glm::value_ptr(v))
	glLoadMatrixfv(glm::value_ptr(m));
}
```

Another solution inspired by STL:

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

void foo()
{
	glm::vec4 v(0.0f);
	glm::mat4 m(1.0f);
	...
	glVertex3fv(&v[0]);
	glLoadMatrixfv(&m[0][0]);
}
```

1.6. GLM for CUDA

GLM 0.9.2 introduces CUDA compiler support allowing programmer to use GLM inside a CUDA Kernel. This support is automatic when a GLM header is included inside a CUDA kernel.

If necessary, a user can decided to force this support by defining `GLM_FORCE_CUDA` before any inclusion of `<glm/glm.hpp>`.

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

Some GLM functionalities might not be available and will return an error message if used in a CUDA kernel.

2. Advanced usages

2.1. 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.

```
vec4 A;
vec2 B;
...
B.yx = A.wy;
B = A.xx;
```

This functionality turns out to be really complicated, not to say impossible, to implement in C++ using the exact GLSL conventions. GLM provides three implementations for this feature.

C++98 implementation

The first implementation follows the GLSL convensions accurately however it uses macros which might generates name conflicts with system headers or third party libraries so that it is disabled by default. To enable this implementation, `GLM_SWIZZLE` has to be defined before any inclusion of `<glm/glm.hpp>`.

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

void foo()
{
	glm::vec4 ColorRGBA(1.0f, 0.5f, 0.0f, 1.0f);
	glm::vec4 ColorBGRA = ColorRGBA.bgra();
	…
	ColorRGBA.bgra() = ColorRGBA;
	ColorRGBA.bgra() = ColorRGBA.rgba();
	…
}
```

Note: Swizzle functions doesn’t return vector types (`glm::vec2`, `glm::vec3` and `glm::vec4`) when the swizzle operators are used as r-values. Instead, the swizzle functions return reference types (`glm::ref2`, `glm::ref3` and `glm::ref4`). Hence, when the reference type objects are generated, these objects can’t be used directly with a GLM functions and need to be cast into vector types.

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

void foo()
{
	glm::vec4 Color(1.0f, 0.5f, 0.0f, 1.0f);
	…
	// Need to cast the swizzle operator into glm::vec4
	glm::vec4 Clamped = glm::clamp(
		glm::vec4(ColorRGBA.bgra() = ColorRGBA), 0.f, 1.f);
	…
}
```

C++11 implementation

The first implementation follows the GLSL convensions accurately however it uses macros which might generates name conflicts with system headers or third party libraries so that

it is disabled by default. To enable this implementation, GLM_SWIZZLE has to be defined before any inclusion of <glm/glm.hpp>.

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

void foo()
{
	glm::vec4 ColorRGBA(1.0f, 0.5f, 0.0f, 1.0f);
	…
	// l-value:
	glm::vec4 ColorBGRA = ColorRGBA.bgra;

	// r-value:
	ColorRGBA.bgra = ColorRGBA;

	// Both l-value and r-value
	ColorRGBA.bgra = ColorRGBA.rgba;
	…
}
```

Note: Swizzle functions doesn't return vector types (glm::vec2, glm::vec3 and glm::vec4) when the swizzle operators are used as r-values. Instead, the swizzle functions return reference types (glm::ref2, glm::ref3 and glm::ref4). Hence, when the reference type objects are generated, these objects can't be used directly with a GLM functions and need to be cast into vector types.

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

void foo()
{
	glm::vec4 Color(1.0f, 0.5f, 0.0f, 1.0f);
	…
	// Need to cast the swizzle operator into glm::vec4
	glm::vec4 ClampedA = glm::clamp(
		glm::vec4(ColorRGBA.bgra = ColorRGBA), 0.f, 1.f);

	// Need to cast the swizzle operator into glm::vec4 (alternative
	method)
	glm::vec4 ClampedB = glm::clamp(
		(ColorRGBA.bgra = ColorRGBA)(), 0.f, 1.f);

	// Need to cast the swizzle operator into glm::vec4
	glm::vec4 ClampedC = glm::clamp(ColorRGBA.bgra(), 0.f, 1.f);
	…
}
```

Extension implementation

A safer way to do swizzling is to use the extension GLM_GTC_swizzle. In term of features, this extension is at the same level than GLSL expect that GLM support both static and dynamic swizzling where GLSL only support static swizzling.

Static swizzling is an operation which is resolved at build time but dynamic swizzling is revolved at runtime which is more flexible but slower especially when SSE optimisations are used.

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

```
void foo()
{
	glm::vec4 ColorRGBA(1.0f, 0.5f, 0.0f, 1.0f);
	…
	// Dynamic swizzling (at run time, more flexible)
	// l-value:
	glm::vec4 ColorBGRA1 =
		glm::swizzle(ColorRGBA, glm::B, glm::G, glm::R, glm::A);
	// r-value:
	glm::swizzle(ColorRGBA, glm::B, glm::G, glm::R, glm::A) = ColorRGBA;

	// Static swizzling (at build time, faster)
	// l-value:
	glm::vec4 ColorBGRA2 =
		glm::swizzle<glm::B, glm::G, glm::R, glm::A>(ColorRGBA);
	// r-value:
	glm::swizzle<glm::B, glm::G, glm::R, glm::A>(ColorRGBA) = ColorRGBA;
}
```

2.2. Notification system

GLM includes a notification system which can display some information at build time:
- Compiler
- Build model: 32bits or 64 bits
- C++ version
- Architecture: x86, SSE, AVX, etc.
- Included extensions
- etc.

This system is disable by default. To enable this system, define `GLM_MESSAGES` before any inclusion of `<glm/glm.hpp>`.

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

2.3. Force inline

To push further the software performance, a programmer can define `GLM_FORCE_INLINE` before any inclusion of `<glm/glm.hpp>` to force the compiler to inline GLM code.

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

2.4. SIMD support

GLM provides some SIMD optimization based on compiler intrinsics. These optimizations will be automatically utilized based on the build environment. These optimizations are mainly available through extensions, `GLM_GTX_simd_vec4` and `GLM_GTX_simd_mat4`.

A programmer can restrict or force instruction sets used for these optimizations using `GLM_FORCE_SSE2` or `GLM_FORCE_AVX`.

A programmer can discard the use of intrinsics by defining `GLM_FORCE_PURE` before any inclusion of `<glm/glm.hpp>`. If `GLM_FORCE_PURE` is defined, then including `a SIMD` extension will generate a build error.

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

2.5. Compatibility

Compilers have some language extensions that GLM will automatically take advantage of them when they are enabled. To increase cross platform compatibility and to avoid compiler extensions, a programmer can define `GLM_FORCE_CXX98` before any inclusion of `<glm/glm.hpp>`.

```
#define GLM_FORCE_CXX98
#include <glm/glm.hpp>
```

For C++11, an equivalent value is available: `GLM_FORCE_CXX11`.

```
#define GLM_FORCE_CXX11
#include <glm/glm.hpp>
```

If both `GLM_FORCE_CXX98` and `GLM_FORCE_CXX11` are defined then C++ 11 features will be utilized.

2.6. Default precision

With C++ it isn't possible to implement GLSL default precision (GLSL 4.10 specification section 4.5.3) the way it is specified in GLSL. Hence, instead of writing:

```
precision mediump int;
precision highp float;
```

With GLM we need to add before any include of `glm.hpp`:

```
#define GLM_PRECISION_MEDIUMP_INT;
#define GLM_PRECISION_HIGHP_FLOAT;
#include <glm/glm.hpp>
```

3. Deprecated function replacements

OpenGL 3.0 specification has deprecated some features that have been removed from OpenGL 3.2 core profile specification. GLM provides some replacement functions.

3.1. OpenGL functions (Section 2.11.2 Matrices, OpenGL 2.1 specification)

glRotate{f, d}:

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

glm::dmat4 glm::rotate(
	glm::dmat4 const & m,
	double angle,
	glm::dvec3 const & axis);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

glScale{f, d}:

```
glm::mat4 glm::scale(
	glm::mat4 const & m,
	glm::vec3 const & factors);

glm::dmat4 glm::scale(
	glm::dmat4 const & m,
	glm::dvec3 const & factors);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

glTranslate{f, d}:

```
glm::mat4 glm::translate(
	glm::mat4 const & m,
	glm::vec3 const & translation);

glm::dmat4 glm::translate(
	glm::dmat4 const & m,
	glm::dvec3 const & translation);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

glLoadIdentity:

```
glm::mat4(1.0) or glm::mat4();
glm::dmat4(1.0) or glm::dmat4();
```

From GLM core library: <glm/glm.hpp>

glMultMatrix{f, d}:

```
glm::mat4() * glm::mat4();
glm::dmat4() * glm::dmat4();
```

From GLM core library: <glm/glm.hpp>

glLoadTransposeMatrix{f, d}:

```
glm::transpose(glm::mat4());
glm::transpose(glm::dmat4());
```

From GLM core library: <glm/glm.hpp>

glMultTransposeMatrix{f, d}:

```
glm::mat4() * glm::transpose(glm::mat4());
glm::dmat4() * glm::transpose(glm::dmat4());
```

From GLM core library: <glm/glm.hpp>

glFrustum:

```
glm::mat4 glm::frustum(
        float left, float right,
        float bottom, float top,
        float zNear, float zFar);

glm::dmat4 glm::frustum(
        double left, double right,
        double bottom, double top,
        double zNear, double zFar);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

glOrtho:

```
glm::mat4 glm::ortho(
        float left, float right,
        float bottom, float top,
        float zNear, float zFar);

glm::dmat4 glm::ortho(
        double left, double right,
        double bottom, double top,
        double zNear, double zFar);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

3.2. GLU functions (Chapter 4: Matrix Manipulation, GLU 1.3 specification)

gluLookAt:

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

glm::dmat4 glm::lookAt(
        glm::dvec3 const & eye,
        glm::dvec3 const & center,
        glm::dvec3 const & up);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

gluOrtho2D:

```
glm::mat4 glm::ortho(
        float left, float right, float bottom, float top);

glm::dmat4 glm::ortho(
        double left, double right, double bottom, double top);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

gluPerspective:

```
glm::mat4 perspective(
        float fovy, float aspect, float zNear, float zFar);
```

```
glm::dmat4 perspective(
	double fovy, double aspect, double zNear, double zFar);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

gluPickMatrix:

```
glm::mat4 pickMatrix(
  glm::vec2 const & center,
  glm::vec2 const & delta,
  glm::ivec4 const & viewport);

glm::dmat4 pickMatrix(
	glm::dvec2 const & center,
	glm::dvec2 const & delta,
	glm::ivec4 const & viewport);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

gluProject:

```
glm::vec3 project(
  glm::vec3 const & obj,
  glm::mat4 const & model,
  glm::mat4 const & proj,
  glm::{i, ' ', d}vec4 const & viewport);

glm::dvec3 project(
	glm::dvec3 const & obj,
	glm::dmat4 const & model,
	glm::dmat4 const & proj,
	glm::{i, ' ', d}vec4 const & viewport);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

gluUnProject:

```
glm::vec3 unProject(
	glm::vec3 const & win,
	glm::mat4 const & model,
	glm::mat4 const & proj,
	glm::{i, ' '}vec4 const & viewport);

glm::dvec3 unProject(
	glm::dvec3 const & win,
	glm::dmat4 const & model,
	glm::dmat4 const & proj,
	glm::{i, ' ', d}vec4 const & viewport);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

4. Extensions

4.1. GLM_GTC_half_float

This extension provides half-precision floating-point types support for scalar (half), vectors (hvec*) and matrices (hmat*x*) for every GLM core functions that takes floating-point parameter.

Half float should be essentially used for storage as it isn't a native format and even if all the scalar operations are supported, there are implemented by converting half to float and then make this conversion back to half.

`<glm/gtc/half_float.hpp>` need to be included to use these functionalities.

4.2. GLM_GTC_matrix_access

Defines functions to access rows or columns of a matrix easily.

`<glm/gtc/matrix_access.hpp>` need to be included to use these functionalities.

4.3. GLM_GTC_matrix_integer

Provides integer matrix types. Inverse and determinant functions are not supported for these types.

`<glm/gtc/matrix_integer.hpp>` need to be included to use these functionalities.

4.4. GLM_GTC_matrix_inverse

Defines additional matrix inverting functions.

`<glm/gtc/matrix_inverse.hpp>` need to be included to use these functionalities.

4.5. GLM_GTC_matrix_transform

Defines functions that generate common transformation matrices.

The matrices generated by this extension use standard OpenGL fixed-function conventions. For example, the lookAt function generates a transform from world space into the specific eye space that the projective matrix functions (perspective, ortho, etc) are designed to expect. The OpenGL compatibility specifications defines the particular layout of this eye space.

`<glm/gtc/matrix_transform.hpp>` need to be included to use these functionalities.

4.6. GLM_GTC_noise

Defines 2D, 3D and 4D procedural noise functions.

`<glm/gtc/noise.hpp>` need to be included to use these functionalities.

Figure 4.6.1: glm::simplex(glm::vec2(x / 16.f, y / 16.f));

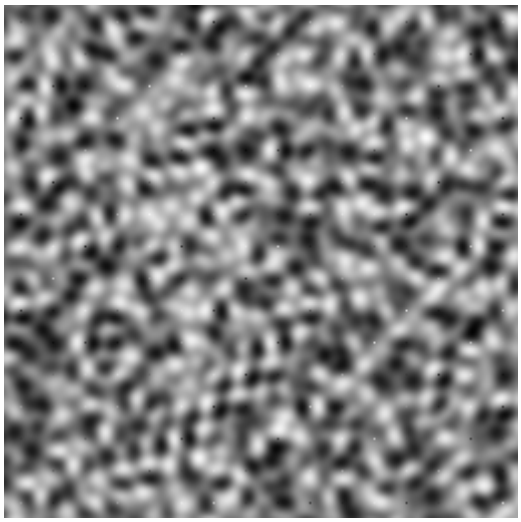


Figure 4.6.2: glm::simplex(glm::vec3(x / 16.f, y / 16.f, 0.5f));

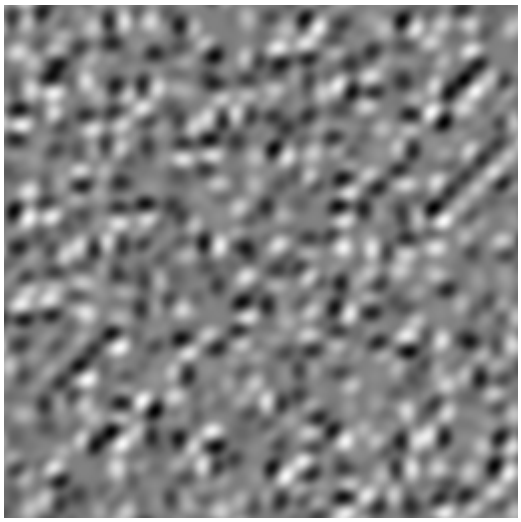


Figure 4.6.3: glm::simplex(glm::vec4(x / 16.f, y / 16.f, 0.5f, 0.5f));

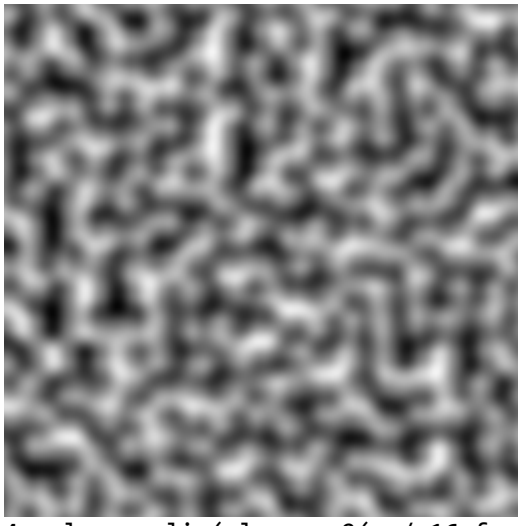


Figure 4.6.4: glm::perlin(glm::vec2(x / 16.f, y / 16.f));

Figure 4.6.5: glm::perlin(glm::vec3(x / 16.f, y / 16.f, 0.5f));

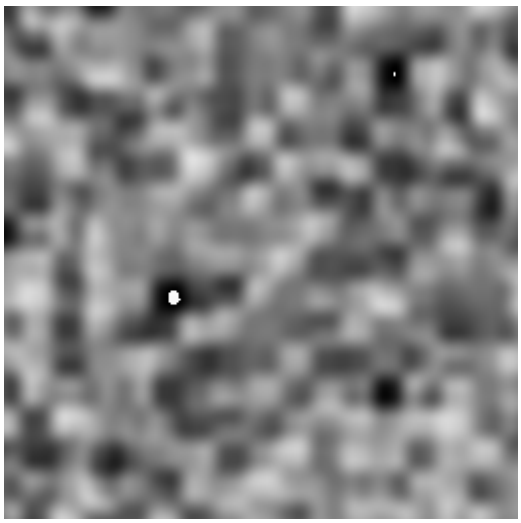


Figure 4.6.6: glm::perlin(glm::vec4(x / 16.f, y / 16.f, 0.5f, 0.5f)));

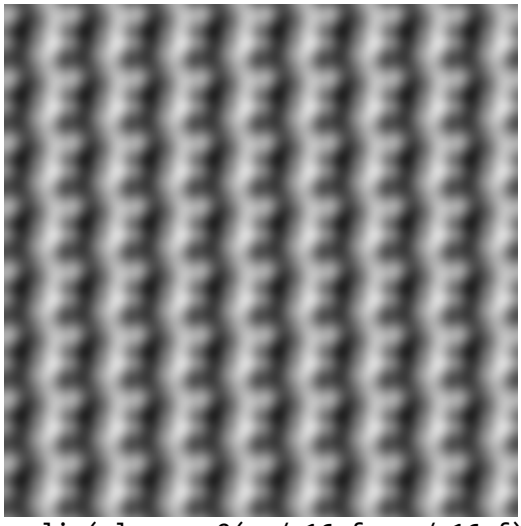


Figure 4.6.7: glm::perlin(glm::vec2(x / 16.f, y / 16.f), glm::vec2(2.0f));

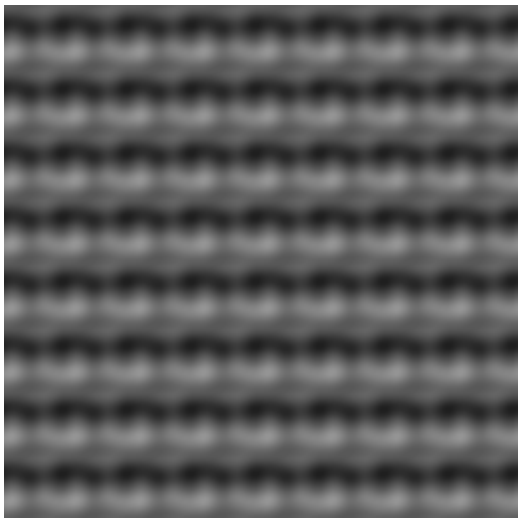


Figure 4.6.8: glm::perlin(glm::vec3(x / 16.f, y / 16.f, 0.5f),
glm::vec3(2.0f));

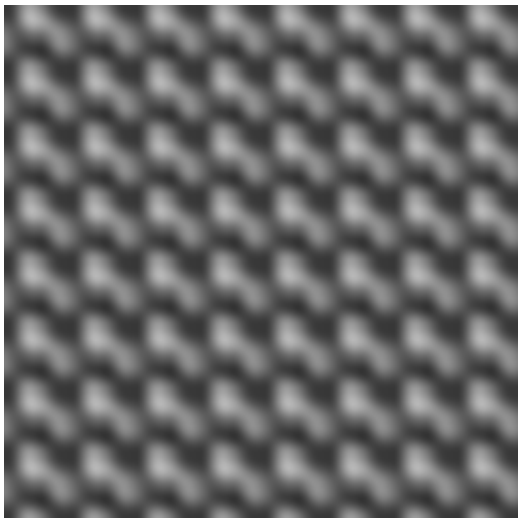


Figure 4.6.9: glm::perlin(glm::vec4(x / 16.f, y / 16.f, glm::vec2(0.5f)),
glm::vec4(2.0f));

4.7. GLM_GTC_quaternion

Defines a templated quaternion type and several quaternion operations.

<glm/gtc/quaternion.hpp> need to be included to use these functionalities.

4.8. GLM_GTC_random

Generate random number from various distribution methods.

<glm/gtc/random.hpp> need to be included to use these functionalities.

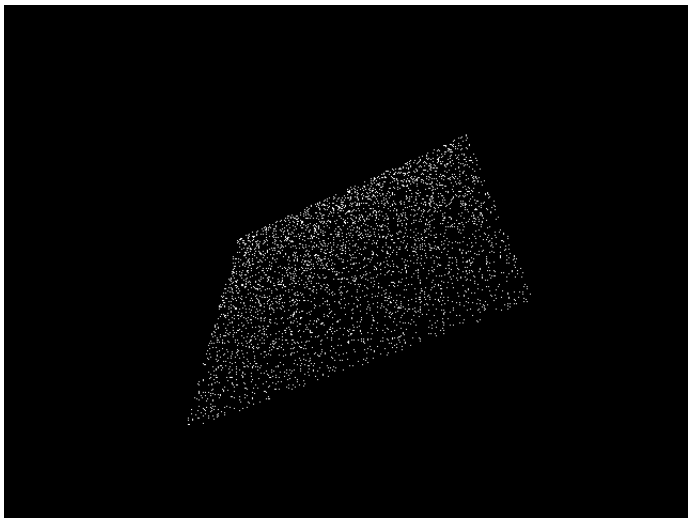


Figure 4.8.1: glm::vec4(glm::linearRand(glm::vec2(-1), glm::vec2(1)), 0, 1);

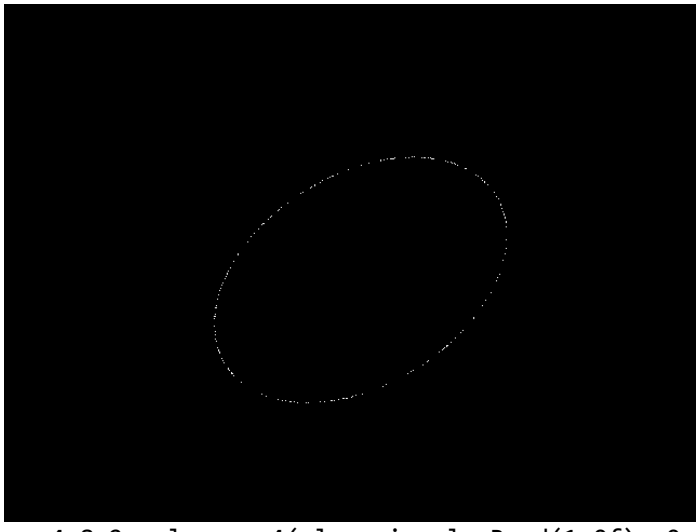


Figure 4.8.2: glm::vec4(glm::circularRand(1.0f), 0, 1);

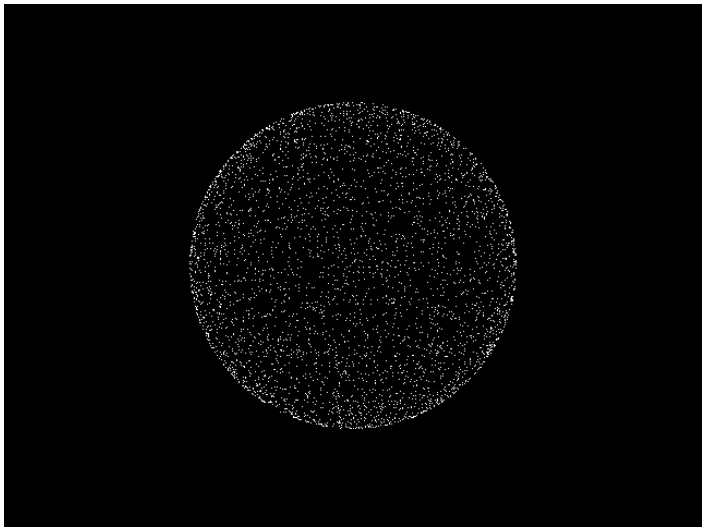


Figure 4.8.3: glm::vec4(glm::sphericalRand(1.0f), 1);

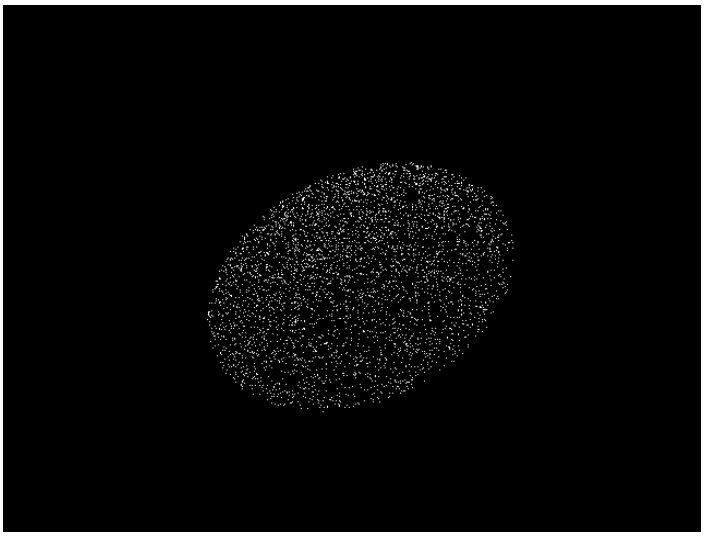


Figure 4.8.4: glm::vec4(glm::diskRand(1.0f), 0, 1);

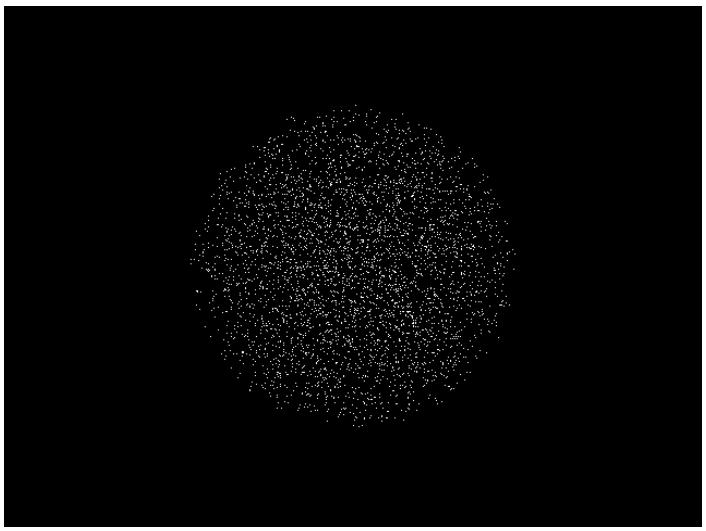


Figure 4.8.5: glm::vec4(glm::ballRand(1.0f), 1);

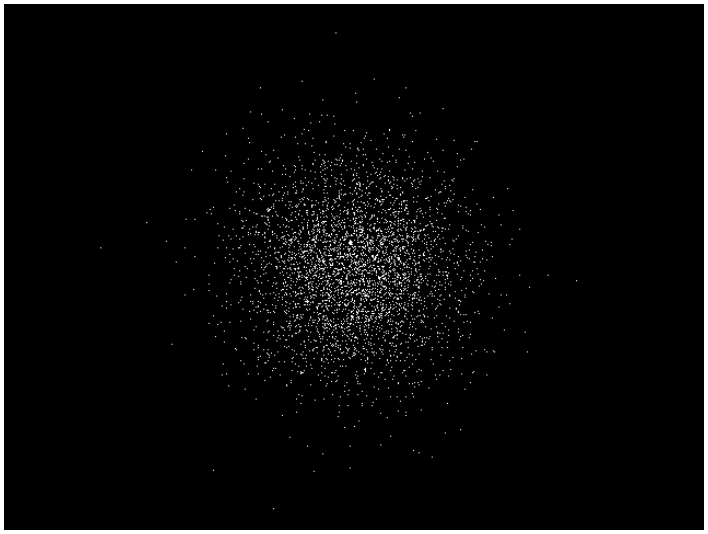


Figure 4.8.6: glm::vec4(glm::gaussRand(glm::vec3(0), glm::vec3(1)), 1);

4.9. GLM_GTC_swizzle

Provide functions to emulate GLSL swizzle operator fonctionalities but with a different syntax that feets C++ boundaries.

<glm/gtc/swizzle.hpp> need to be included to use these functionalities.

4.10. GLM_GTC_type_precision

Vector and matrix types with defined precisions. Eg, i8vec4: vector of 4 signed integer of 8 bits.

<glm/gtc/type_precision.hpp> need to be included to use these functionalities.

4.11. GLM_GTC_type_ptr

Handles the interaction between pointers and vector, matrix types.

This extension defines an overloaded function, glm::value_ptr, which takes any of the core template types (vec3, mat4, etc.). It returns a pointer to the memory layout of the object. Matrix types store their values in column-major order.

This is useful for uploading data to matrices or copying data to buffer objects.

Example:

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

glm::vec3 aVector(3);
glm::mat4 someMatrix(1.0);

glUniform3fv(uniformLoc, 1, glm::value_ptr(aVector));
glUniformMatrix4fv(uniformMatrixLoc,              1,              GL_FALSE,
glm::value_ptr(someMatrix));
```

<glm/gtc/type_ptr.hpp> need to be included to use these functionalities.

5. Known issues

5.1. not function

The GLSL keyword not is also a keyword in C++. To prevent name collisions, ensure cross compiler support and a high API consistency, the GLSL not function has been implemented with the name `not_`.

5.2. half based types and component accesses

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++98 specification doesn't support anonymous unions which limits `hvec*` vector components access to x, y, z and w.

However, Visual C++ does support anonymous unions if the language extensions are enabled (/Za to disable them). In this case GLM will automatically enables the support of all component names (x,y,z,w ; r,g,b,a ; s,t,p,q).

To uniformalize the component access across types, GLM provides the define `GLM_FORCE_ONLY_XYZW` which will generates errors if component accesses are done using r,g,b,a or s,t,p,q.

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

6. FAQ

6.1 Why GLM follows GLSL specification and conventions?

Following GLSL conventions is a really strict policy of GLM. It 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.

6.2. Does GLM run GLSL program?

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

6.3. Does a GLSL compiler build GLM codes?

No, this is not what GLM attends to do!

6.4. 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 any 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 and APIs and then are promoted to GTC extensions. This is fairly the way OpenGL features are developed; through extensions.

6.5. Where can I ask my questions?

A good place is the OpenGL Toolkits forum on OpenGL.org

6.6. Where can I find the documentation of extensions?

The Doxygen generated documentation includes a complete list of all extensions available. Explore this API documentation to get a complete view of all GLM capabilities!

6.7. Should I use 'using namespace glm;'?

NO! Chances are that if 'using namespace glm;' is called, especially in a header file, name collisions will happen as GLM is based on GLSL which uses common tokens for types and functions. Avoiding 'using namespace glm;' will an higher compatibility with third party library and SDKs.

6.8. Is GLM fast?

First, GLM is mainly designed to be convenient and that's why it is written against GLSL specification. Following the 20-80 rules where 20% of the code grad 80% of the performances, GLM perfectly operates on the 80% of the code that consumes 20% of the performances. This said, on performance critical code section, the developers will probably have to write to specific code based on a specific design to reach peak performances but GLM can provides some descent performances alternatives based on approximations or SIMD instructions.

6.9. When I build with Visual C++ with /W4 warning level, I have warnings...

You should not have any warnings even in /W4 mode. However, if you expect such level for you code, then you should ask for the same level to the compiler by as least disabling the Visual C++ language extensions (/Za) which generates warnings when used. If these extensions are enabled, then GLM will take advantage of them and the compiler will generate warnings.

7. Code samples

This series of samples only shows various GLM features without consideration of any sort.

7.1. Compute a triangle normal

```
#include <glm/glm.hpp> // vec3 normalize cross

glm::vec3 computeNormal(
	glm::vec3 const & a,
	glm::vec3 const & b,
	glm::vec3 const & c)
{
	return glm::normalize(glm::cross(c - a, b - a));
}

// A much faster but less accurate alternative:
#include <glm/glm.hpp> // vec3 cross
#include <glm/gtx/fast_square_root.hpp> // fastNormalize

glm::vec3 computeNormal(
	glm::vec3 const & a,
	glm::vec3 const & b,
	glm::vec3 const & c)
{
	return glm::fastNormalize(glm::cross(c - a, b - a));
}
```

7.2. Matrix transform

```
#include <glm/glm.hpp> //vec3, vec4, ivec4, mat4
#include <glm/gtc/matrix_transform.hpp> //translate, rotate, scale,
perspective
#include <glm/gtc/type_ptr.hpp> //value_ptr

void setUniformMVP
(
	GLuint Location,
	glm::vec3 const & Translate,
	glm::vec3 const & Rotate
)
{
	glm::mat4 Projection =
		glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.f);
	glm::mat4 ViewTranslate = glm::translate(
		glm::mat4(1.0f),
		Translate);
	glm::mat4 ViewRotateX = glm::rotate(
		ViewTranslate,
		Rotate.y, glm::vec3(-1.0f, 0.0f, 0.0f));
	glm::mat4 View = glm::rotate(
		ViewRotateX,
		Rotate.x, glm::vec3(0.0f, 1.0f, 0.0f));
	glm::mat4 Model = glm::scale(
		glm::mat4(1.0f),
		glm::vec3(0.5f));
	glm::mat4 MVP = Projection * View * Model;
	glUniformMatrix4fv(
		Location, 1, GL_FALSE, glm::value_ptr(MVP));
```

```
}
```

7.3. Vector types

```
#include <glm/glm.hpp> //vec2
#include <glm/gtc/type_precision.hpp> //hvec2, i8vec2, i32vec2
std::size_t const VertexCount = 4;

// Float quad geometry
std::size_t const PositionSizeF32 = VertexCount * sizeof(glm::vec2);
glm::vec2 const PositionDataF32[VertexCount] =
{
	glm::vec2(-1.0f,-1.0f),
	glm::vec2( 1.0f,-1.0f),
	glm::vec2( 1.0f, 1.0f),
	glm::vec2(-1.0f, 1.0f)
};
// Half-float quad geometry
std::size_t const PositionSizeF16 = VertexCount * sizeof(glm::hvec2);
glm::hvec2 const PositionDataF16[VertexCount] =
{
	glm::hvec2(-1.0f, -1.0f),
	glm::hvec2( 1.0f, -1.0f),
	glm::hvec2( 1.0f, 1.0f),
	glm::hvec2(-1.0f, 1.0f)
};
// 8 bits signed integer quad geometry
std::size_t const PositionSizeI8 = VertexCount * sizeof(glm::i8vec2);
glm::i8vec2 const PositionDataI8[VertexCount] =
{
	glm::i8vec2(-1,-1),
	glm::i8vec2( 1,-1),
	glm::i8vec2( 1, 1),
	glm::i8vec2(-1, 1)
};
// 32 bits signed integer quad geometry
std::size_t const PositionSizeI32 = VertexCount * sizeof(glm::i32vec2);
glm::i32vec2 const PositionDataI32[VertexCount] =
{
	glm::i32vec2 (-1,-1),
	glm::i32vec2 ( 1,-1),
	glm::i32vec2 ( 1, 1),
	glm::i32vec2 (-1, 1)
};
```

7.4. Lighting

```
#include <glm/glm.hpp> // vec3 normalize reflect dot pow
#include <glm/gtx/random.hpp> // vecRand3

// vecRand3, generate a random and equiprobable normalized vec3

glm::vec3 lighting
(
	intersection const & Intersection,
	material const & Material,
	light const & Light,
	glm::vec3 const & View
)
{
	glm::vec3 Color = glm::vec3(0.0f);
```

```
        glm::vec3 LightVertor = glm::normalize(
                Light.position() - Intersection.globalPosition() +
                glm::vecRand3(0.0f, Light.inaccuracy());
        if(!shadow(
                Intersection.globalPosition(),
                Light.position(),
                LightVertor))
        {
                float Diffuse = glm::dot(Intersection.normal(), LightVector);
                if(Diffuse <= 0.0f)
                        return Color;
                if(Material.isDiffuse())
                        Color += Light.color() * Material.diffuse() * Diffuse;
                if(Material.isSpecular())
                {
                        glm::vec3 Reflect = glm::reflect(
                                -LightVector,
                                Intersection.normal());
                        float Dot = glm::dot(Reflect, View);
                        float Base = Dot > 0.0f ? Dot : 0.0f;
                        float Specular = glm::pow(Base, Material.exponent());
                        Color += Material.specular() * Specular;
                }
        }
        return Color;
}
```

8. References

8.1. GLM development

- GLM website
- GLM HEAD snapshot
- GLM Trac, for bug report and feature request
- G-Truc Creation's page

8.2. OpenGL specifications

- OpenGL 4.1 core specification
- GLSL 4.10 specification
- GLU 1.3 specification

8.3. External links

- The OpenGL Toolkits forum to ask questions about GLM

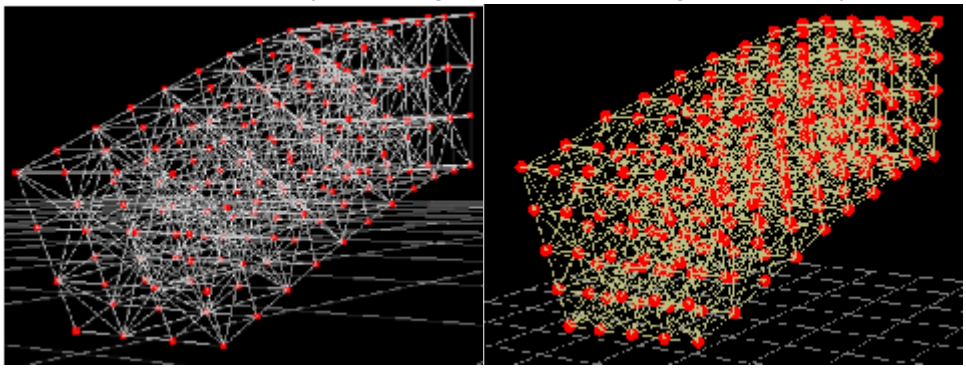
8.4. Projects using GLM

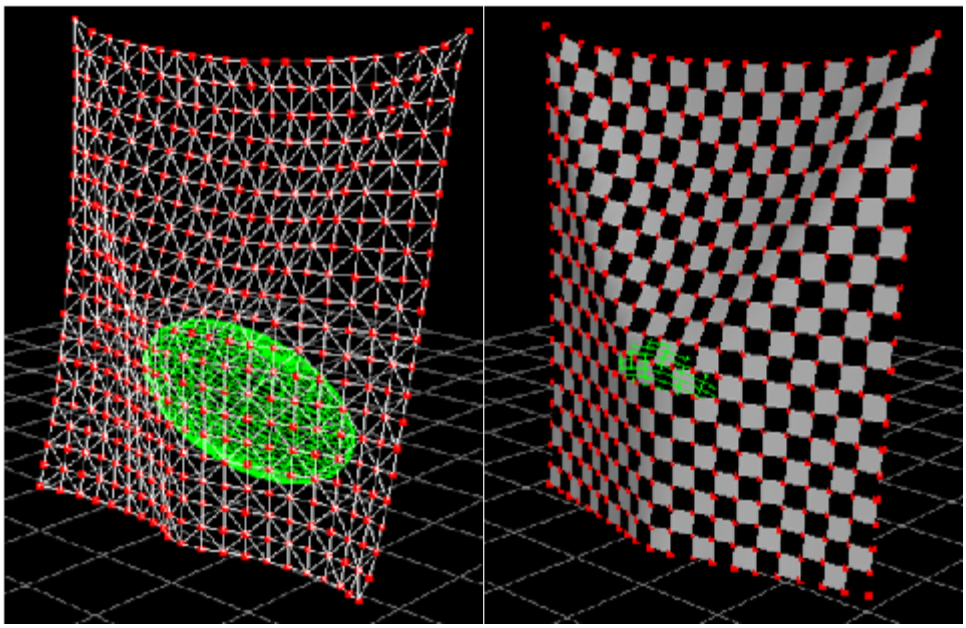
- Outerra:

3D planetary engine for seamless planet rendering from space down to the surface. Can use arbitrary resolution of elevation data, refining it to centimeter resolution using fractal algorithms.

- opencloth:

A collection of source codes implementing cloth simulation algorithms in OpenGL.





- OpenGL 4.0 Shading Language Cookbook:

A full set of recipes demonstrating simple and advanced techniques for producing high-quality, real-time 3D graphics using GLSL 4.0.

How to use the OpenGL Shading Language to implement lighting and shading techniques.

Use the new features of GLSL 4.0 including tessellation and geometry shaders.

How to use textures in GLSL as part of a wide variety of techniques from basic texture mapping to deferred shading.

Simple, easy-to-follow examples with GLSL source code, as well as a basic description of the theory behind each techniques.

- Are you using GLM in a project?

8.5. OpenGL tutorials using GLM

- The OpenGL Samples Pack, samples that show how to set up all the different new features
- Learning Modern 3D Graphics Programming, a great OpenGL tutorial using GLM by Jason L. McKesson
- Morten Nobel-Jørgensen's review and use an OpenGL renderer
- Swiftless' OpenGL tutorial using GLM by Donald Urquhart
- Rastergrid, many technical articles with companion programs using GLM by Daniel Rákos
- OpenGL Tutorial, tutorials for OpenGL 3.1 and later
- OpenGL Programming on Wikibooks: For beginners who are discovering OpenGL.
- 3D Game Engine Programming: Learning the latest 3D Game Engine Programming techniques.

- Are you using GLM in a tutorial?

8.6. Alternatives to GLM

- CML: The CML (Configurable Math Library) is a free C++ math library for games and graphics.
- Eigen: A more heavy weight math library for general linear algebra in C++.
- glhlib: A much more than glu C library.

- Are you using or working an alternative library to GLM?

8.7. Acknowledgements

GLM is developed and maintained by Christophe Riccio but many contributors have made this project what it is.

Special thanks to:
- Ashima Arts and Stefan Gustavson for their work on webgl-noise which has been used for GLM noises implementation.
- Arthur Winters for the C++11 and Visual C++ swizzle operators implementation and tests.
- Joshua Smith and Christoph Schied for the discussions and the experiments around the swizzle operator implementation issues.
- Guillaume Chevallereau for providing and maintaining the nightlight build system.
- Ghenadii Ursachi for `GLM_GTX_matrix_interpolation` implementation.
- Mathieu Roumillac for submitting the `GLM_VIRTREV_xstream` extension and providing some implementation ideas.
- Grant James for the implementation of all combination of none-squared matrix products.
- All the GLM user that has report bugs and hence help GLM to become a great library!

8.8. Quotes from the web

"I am also going to make use of boost for its time framework and the matrix library GLM, a GL Shader-like Math library for C++. A little advertise for the latter which has a nice design and is useful since matrices have been removed from the latest OpenGL versions"

Code Xperiments

"OpenGL Mathmatics Library (GLM): Used for vector classes, quaternion classes, matrix classes and math operations that are suited for OpenGL applications."

Jeremiah van Oosten

"Today I ported my codebase from my own small linear algebra library to GLM, a GLSL-style vector library for C++. The transition went smoothly."

Leonard Ritter

"A more clever approach is to use a math library like GLM instead. I wish someone had showed me this library a few years ago."

Morten Nobel-Jørgensen