GLM Manual

Version 0.9.7

Christophe Riccio

March 11, 2016

Licensing

The Happy Bunny License (Modified MIT License)

© 2008 - 2016 G-Truc Creation

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

Restrictions: By making use of the Software for military purposes, you choose to make a Bunny unhappy.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The MIT License

© 2008 - 2016 G-Truc Creation

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Contents

Introduction

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

GLM provides classes and functions to mimic the conventions and functionality provided by GLSL. An extension system (inspired by OpenGL's) also provides extra capabilities including (but not limited to) matrix transformations, quaternions, data packing, random numbers, and noise.

GLM works perfectly with OpenGL, but is also well-suited for use with any project that demands a simple (yet flexible) mathematics framework such as software rendering (ray-tracing/rasterisation), image processing, and physics simulation.

GLM is written in C++98, but can take advantage of C++11 where support exists. GLM is platform independent, has no dependencies, and supports the following compilers:

- Apple Clang 4.0 and higher
- GCC 4.2 and higher
- Intel C++ Composer XE 2013 and higher
- LLVM 3.0 and higher
- Visual C++ 2010 and higher
- CUDA 4.0 and higher (experimental)
- Any conforming C++98 or C++11 compiler

The source code and the documentation (including this manual) are licensed under both the Happy Bunny License (Modified MIT) and the MIT License.

Feedback, bug reports, feature requests, and acts upon thereof are highly appreciated. The author may be contacted at glm@g-truc.net.

1 Getting Started

1.1 Setup

GLM is a header-only library, and thus does not need to be compiled. To use GLM, merely include the `<glm/glm.hpp>` header, which provides GLSL's mathematics functionality.

- `<glm/vec2.hpp>`: `vec2`, `bvec2`, `dvec2`, `ivec2` and `uvec2`
- `<glm/vec3.hpp>`: `vec3`, `bvec3`, `dvec3`, `ivec3` and `uvec3`
- `<glm/vec4.hpp>`: `vec4`, `bvec4`, `dvec4`, `ivec4` and `uvec4`
- `<glm/mat2x2.hpp>`: `mat2`, `dmat2`
- `<glm/mat2x3.hpp>`: `mat2x3`, `dmat2x3`
- `<glm/mat2x4.hpp>`: `mat2x4`, `dmat2x4`
- `<glm/mat3x2.hpp>`: `mat3x2`, `dmat3x2`
- `<glm/mat3x3.hpp>`: `mat3`, `dmat3`
- `<glm/mat3x4.hpp>`: `mat3x4`, `dmat3x4`
- `<glm/mat4x2.hpp>`: `mat4x2`, `dmat4x2`
- `<glm/mat4x3.hpp>`: `mat4x3`, `dmat4x3`
- `<glm/mat4x4.hpp>`: `mat4`, `dmat4`
- `<glm/common.hpp>`: all the GLSL common functions
- `<glm/exponential.hpp>`: all the GLSL exponential functions
- `<glm/geometry.hpp>`: all the GLSL geometry functions
- `<glm/integer.hpp>`: all the GLSL integer functions
- `<glm/matrix.hpp>`: all the GLSL matrix functions
- `<glm/packing.hpp>`: all the GLSL packing functions
- `<glm/trigonometric.hpp>`: all the GLSL trigonometric functions

- `<glm/vector_relational.hpp>`: all the GLSL vector relational functions

1.2 Faster Compilation

GLM makes heavy use of C++ templates, which may significantly increase the compile time for projects that use GLM. Hence, source files should only include the GLM headers they actually use.

To further reduce compilation time, include `<glm/fwd.hpp>`, which provides forward declarations of all types should their full definitions not be needed.

```
c+c1// Header file (forward declarations only)
c+cp#include c+cpf<glm/fwd.hpp>
```

```
c+c1// Source file (actual types included)
c+cp#include c+cpf<glm/glm.hpp>
```

1.3 Example Usage

```
c+c1// Include GLM core features
c+cp#include c+cpf<glm/vec3.hpp>
c+cp#include c+cpf<glm/vec4.hpp>
c+cp#include c+cpf<glm/mat4x4.hpp>

c+c1// Include GLM extensions
c+cp#include c+cpf<glm/gtc/matrix_transform.hpp>

kusing nglmo::nvec2p;
kusing nglmo::nvec3p;
kusing nglmo::nmat4p;

nmat4 n+nftransformp(
  nvec2 kconst o& nOrientationp,
  nvec3 kconst o& nTranslatep,
  nvec2 kconst o& nUpp)
p{
  nmat4 nProjection o= nglmo::nperspectivep(l+m+mf45.0fp, l+m+mf4.0fo/l+m+mf3.0fp, l+m+m
  nmat4 nViewTranslate o= nglmo::ntranslatep(nmat4p(l+m+mf1.0fp), nTranslatep);
  nmat4 nViewRotateX o= nglmo::nrotatep(nViewTranslatep, nOrientationp.nyp, nUpp);
  nmat4 nView o= nglmo::nrotatep(nViewRotateXp, nOrientationp.nxp, nUpp);
  nmat4 nModel o= nmat4p(l+m+mf1.0fp);

  kreturn nProjection o* nView o* nModelp;
p}
```

1.4 Dependencies

The `<glm/glm.hpp>` header provides all standard GLSL features.

GLM does not depend on external libraries or external headers such as `gl.h`, `glcorearb.h`, `gl3.h`, `glu.h` or `windows.h`. However, if `<boost/static_assert.hpp>` is included, then `Boost.StaticAssert` will be used to provide compile-time errors. Otherwise, if using a C++11 compiler, the standard `static_assert` will be used instead. If neither is available, GLM will use its own implementation of `static_assert`.

2 Swizzling

Shader languages like GLSL often feature so-called swizzle expression, which may be used to freely select and arrange a vector's components. For example, `variable.x`, `variable.xzy` and `variable.zxyy` respectively form a scalar, a 3D vector and a 4D vector. The result of a swizzle expression in GLSL can be either an R-value or an L-value. Swizzle expressions can be written with characters from exactly one of `xyzw` (usually for positions), `rgba` (usually for colors), or `stpq` (usually for texture coordinates).

```
1 kvec4 nAp;
2 kvec2 nBp;
3
4 nBp.nyx o= nAp.nwyp;
5 nB o= nAp.nxxp;
6 kvec3 nC o= nAp.nbgrp;
7 kvec3 nD o= nBp.nrszp; c+c1// Invalid, won't compile
```

GLM optionally supports some of this functionality via the methods described in the following sections. Swizzling can be enabled by defining `GLM_SWIZZLE` before including any GLM header files, or as part of your project's build process.

Note that enabling swizzle expressions will massively increase the size of your binaries and the time it takes to compile them!

2.1 Default C++98 Implementation

When compiling GLM as C++98, R-value swizzle expressions are simulated through member functions of each vector type.

```
c+cp#define GLM_SWIZZLE c+c1// Or define when building (e.g. -DGLM_SWIZZLE)
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop()
p{
  nglmo::nvec4 nColorRGBAp(l+m+mf1.0fp, l+m+mf0.5fp, l+m+mf0.0fp, l+m+mf1.0fp);
  nglmo::nvec3 nColorBGR o= nColorRGBAp.nbgrp();

  nglmo::nvec3 nPositionAp(l+m+mf1.0fp, l+m+mf0.5fp, l+m+mf0.0fp, l+m+mf1.0fp);
  nglmo::nvec3 nPositionB o= nPositionXYZp.nxyzp() o* l+m+mf2.0fp;

  nglmo::nvec2 nTexcoordSTp(l+m+mf1.0fp, l+m+mf0.5fp);
  nglmo::nvec4 nTexcoordSTPQ o= nTexcoordSTp.nststp();
p}
```

Swizzle operators return a **copy** of the component values, and thus **can't** be used as L-values to change a vector's values.

```
c+cp#define GLM_SWIZZLE
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop()
p{
  nglmo::nvec3 nAp(l+m+mf1.0fp, l+m+mf0.5fp, l+m+mf0.0fp);

  c+c1// No compiler error, but A is not modified.
  c+c1// An anonymous copy is being modified (and then discarded).
  nAp.nbgrp() o= nglmo::nvec3p(l+m+mf2.0fp, l+m+mf1.5fp, l+m+mf1.0fp); c+c1// A is not m
p}
```

2.2 Anonymous Union Member Implementation

Visual C++ supports, as a *non-standard language extension*, anonymous `structs` in `unions`. This enables a very powerful implementation of swizzle expressions on Windows, which both allows L-value swizzle operators and makes the syntax for it closer to GLSL's. You must enable this language extension in a supported compiler and define `GLM_SWIZZLE` to use this implementation of swizzling.

```
c+cp#define GLM_SWIZZLE
c+cp#include c+cpf<glm/glm.hpp>

c+c1// Only guaranteed to work with Visual C++!
c+c1// Some compilers that support Microsoft extensions may compile this.
k+ktvoid n+nffoop()
p{
  nglmo::nvec4 nColorRGBAp(l+m+mf1.0fp, l+m+mf0.5fp, l+m+mf0.0fp, l+m+mf1.0fp);

  c+c1// l-value:
  nglmo::nvec4 nColorBGRA o= nColorRGBAp.nbgrap;

  c+c1// r-value:
  nColorRGBAp.nbgra o= nColorRGBAp;

  c+c1// Both l-value and r-value
  nColorRGBAp.nbgra o= nColorRGBAp.nrgbap;
p}
```

This versions returns implementation-specific objects that *implicitly convert* to their respective vector types. Unfortunately, these extra types can't be directly used by GLM functions; you must instead convert a swizzle-made "vector" to a conventional vector type or call the swizzle object's `operator()`.

```
c+cp#define GLM_SWIZZLE
c+cp#include c+cpf<glm/glm.hpp>

kusing nglmo::nvec4p;

k+ktvoid n+nffoop()
p{
  nvec4 nColorp(l+m+mf1.0fp, l+m+mf0.5fp, l+m+mf0.0fp, l+m+mf1.0fp);

  c+c1// Generates compiler errors. Color.rgba is not a vector type.
  nvec4 nClampedA o= nglmo::nclampp(nColorp.nrgbap, l+m+mf0.fp, l+m+mf1.fp); c+c1// ERRO

  c+c1// Explicit conversion through a constructor
  nvec4 nClampedB o= nglmo::nclampp(nvec4p(nColorp.nrgbap), l+m+mf0.fp, l+m+mf1.fp); c+c

  c+c1// Explicit conversion through operator()
  nvec4 nClampedC o= nglmo::nclampp(nColorp.nrgbap(), l+m+mf0.fp, l+m+mf1.fp); c+c1// OK
p}
```

3 Preprocessor Options

3.1 Default Precision

C++ does not provide a way to implement GLSL default precision selection (as defined in the GLSL 4.10 specification, section 4.5.3) with GLSL-like syntax.

```
kprecision kmediump kintp;
kprecision khighp kfloatp;
```

To set the default precision of a given arithmetic type, use any of the following defines:

```
c+cp#define GLM_PRECISION_MEDIUMP_INT
c+cp#define GLM_PRECISION_HIGHP_FLOAT
c+c1// Or, again, as part of your build process

c+cp#include c+cpf<glm/glm.hpp>
```

Available defines for floating point types (`glm::vec*`, `glm::mat*`):

- `GLM_PRECISION_LOWP_FLOAT`: Low precision
- `GLM_PRECISION_MEDIUMP_FLOAT`: Medium precision
- `GLM_PRECISION_HIGHP_FLOAT`: High precision (default)

Available defines for double-precision floating point types (`glm::dvec*`, `glm::dmat*`):

- `GLM_PRECISION_LOWP_DOUBLE`: Low precision
- `GLM_PRECISION_MEDIUMP_DOUBLE`: Medium precision
- `GLM_PRECISION_HIGHP_DOUBLE`: High precision (default)

Available defines for signed integer types (`glm::ivec*`):

- `GLM_PRECISION_LOWP_INT`: Low precision
- `GLM_PRECISION_MEDIUMP_INT`: Medium precision
- `GLM_PRECISION_HIGHP_INT`: High precision (default)

Available defines for unsigned integer types (`glm::uvec*`):

- `GLM_PRECISION_LOWP_UINT`: Low precision

- `GLM_PRECISION_MEDIUMP_UINT`: Medium precision
- `GLM_PRECISION_HIGHP_UINT`: High precision (default)

3.2 Compile-Time Messages

GLM can optionally display the following at compile-time:

- Platform: Windows, Linux, Native Client, QNX, etc.
- Compiler: Visual C++, Clang, GCC, ICC, etc.
- Build model: 32-bit or 64-bit
- C++ version: C++98, C++11, MS extensions, etc.
- Architecture: x86, SSE, AVX, etc.
- Included extensions

To enable compile-time messaging, define `GLM_MESSAGES` before any inclusion of `<glm/glm.hpp>`. The messages are generated once per build, assuming your compiler supports `#pragma message`.

```
c+cp#define GLM_MESSAGES  c+c1// Or as part of your build
c+cp#include c+cpf<glm/glm.hpp>
```

3.3 C++ Language Detection

GLM may implement certain features that require the presence of a minimum C++ standard. You can mandate compatibility with particular revisions of C++ by defining `GLM_FORCE_CXX**` before any inclusion of `<glm/glm.hpp>` (where ** is one of `98`, `03`, `11`, and `14`).

```
c+cp#define GLM_FORCE_CXX98
c+cp#include c+cpf<glm/glm.hpp>
c+c1// Nothing that was introduced after 1998 will be used in GLM.
```

```
c+cp#define GLM_FORCE_CXX14
c+cp#include c+cpf<glm/glm.hpp>
c+c1// Live life on the bleeding edge; go big or go home!
```

Later standards will override earlier ones, like so:

GLM_FORCE_CXX14 > GLM_FORCE_CXX11 > GLM_FORCE_CXX03 > GLM_FORCE_CXX98

3.4 SIMD Support

GLM provides some SIMD (Single instruction, multiple data) optimizations based on compiler intrinsics, which will be automatically utilized based on compiler's arguments. For example, if a program is compiled in Visual C++ with `/arch:AVX` set, certain GLM functionality will use AVX instructions.

In addition, GLM provides specialized `vec4`, `quat`, and `mat4` implementations through the `GLM_GTX_simd_vec4`, `GLM_GTX_simd_quat`, and `GLM_GTX_simd_mat4` extensions.

You can force GLM to use a particular set of intrinsics with the following defines: `GLM_FORCE_SSE2`, `GLM_FORCE_SSE3`, `GLM_FORCE_SSE4`, `GLM_FORCE_AVX` or `GLM_FORCE_AVX2`.

You may also disable intrinsics entirely by defining `GLM_FORCE_PURE` before any inclusion of `<glm/glm.hpp>`. If `GLM_FORCE_PURE` is defined, then including any SIMD extension will generate a compiler error.

```
c+cp#define GLM_FORCE_PURE
c+cp#include c+cpf<glm/glm.hpp>

c+c1// GLM code without any form of intrinsics.
```

Useful as `GLM_FORCE_PURE` is, I suggest you enforce this with compiler arguments instead.

```
c+cp#define GLM_FORCE_AVX2
c+cp#include c+cpf<glm/glm.hpp>

c+c1// Will only compile if AVX2 intrinsics are supported.
```

3.5 Force Inline

To gain a bit of performance, you can define `GLM_FORCE_INLINE` before any inclusion of `<glm/glm.hpp>` to force the compiler to inline GLM code.

```
c+cp#define GLM_FORCE_INLINE
c+cp#include c+cpf<glm/glm.hpp>
```

3.6 Compile-Time Type Info

The member function `length()` returns the dimensionality (number of components) of any GLSL matrix or vector type.

```
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop(nglmo::nvec4 kconst o& nvp)
p{
  k+ktint nLength o= nvp.nlengthp();  c+c1// returns 4
p}
```

There are two problems with this function.

The first problem is that `length()` returns an `int` (signed) despite typically being used with code that expects a `size_t` (unsigned). To force a `length()` member function to return a `size_t`, define `GLM_FORCE_SIZE_T_LENGTH`.

GLM also defines the `typedef glm::length_t` to identify the returned type of `length()`, regardless of whether `GLM_FORCE_SIZE_T_LENGTH` is set.

```
c+cp#define GLM_FORCE_SIZE_T_LENGTH
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop(nglmo::nvec4 kconst o& nvp)
p{
  nglmo::k+ktsize_t nLength o= nvp.nlengthp();
p}
```

The second problem is that `length()` shares its name with `glm::length(*vec* const & v)`, which is used to return a vector's Euclidean length. Developers familiar with other vector math libraries may be used to their equivalent of `glm::length(*vec* const & v)` being defined as a member function, and may thus ask for a vector's dimensionality when they intend to ask for its Euclidean length.

```
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop(nglmo::nvec3 kconst o& ntargetp)
p{
  k+ktfloat ndistance o= nvp.nlengthp();  c+c1// Always returns 3.  Oops!
p}
```

To resolve this, define `GLM_FORCE_SIZE_FUNC` to use `size()` for dimensionality queries.

```
c+cp#define GLM_FORCE_SIZE_FUNC
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop(nglmo::nvec4 kconst o& nvp)
p{
  nglmo::k+ktsize_t nSize o= nvp.nsizep();  c+c1// Always returns 4
p}
```

As of GLM 0.9.7.0, GLM also provides the `GLM_META_PROG_HELPERS`, which enables `static` members that provide information about vector, matrix, and quaternion types at compile time in a manner suitable for template metaprogramming.

```
c+cp#define GLM_META_PROG_HELPERS
c+cp#include c+cpf<glm/glm.hpp>

ktemplateo<ktypename nVecTo>
k+ktvoid nfoop(nVecT kconst o& nvp)
p{
  kstatic_assertp(nVecTo::ncomponents o< l+m+mi4p, l+s"4D doesn't make sense!"p);

  nglmo::nvec2o<nVecTo::nvalue_typep, nVecTo::npreco> nsomethingp;
  c+c1// vec2 with the same component type and precision as VecT
p}

ktemplateo<ktypename nMatTo>
k+ktvoid nbarp(nMatT kconst o& nmp)
p{
  kstatic_assertp(nMatTo::nrows o== nMatTo::ncolsp, l+s"Square matrices only!"p);
p}
```

3.7 Disabling Default Initialization

By default, the nullary (zero-argument) constructors of vectors and matrices initialize their components to zero, as demanded by the GLSL specification. Such behavior is reliable and convenient, but sometimes unnecessary. Disable it at compile time by defining `GLM_FORCE_NO_CTOR_INIT`.

GLM's default behavior:

```
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop()
p{
  nglmo::nvec4 nvp; c+c1// v is (0.0f, 0.0f, 0.0f, 0.0f)
p}
```

GLM behavior using `GLM_FORCE_NO_CTOR_INIT`:

```
c+cp#define GLM_FORCE_NO_CTOR_INIT
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop()
p{
  nglmo::nvec4 nvp; c+c1// v's components are undefined
p}
```

Alternatively, individual variables may be left undefined, like so:

```
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop()
p{
  nglmo::nvec4 nvp(nglmo::nuninitializep); c+c1// v's components are undefined
p}
```

3.8 Requiring Explicit Conversions

GLSL allows implicit conversions of vector and matrix types (e.g. from `ivec4` to `vec4`).

```
kivec4 nap;
kvec4 nb o= nap; c+c1// Implicit conversion, OK
```

Such behavior isn't always desirable in C++, but in the spirit of GLM's mission it is fully supported.

```
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop()
p{
  nglmo::nivec4 nap;

  nglmo::nvec4 nbp(nap); c+c1// Explicit conversion, OK
  nglmo::nvec4 nc o= nap; c+c1// Implicit conversion, OK
p}
```

To instead require all conversions between GLM types to be explicit (making implicit conversions a compiler error), define `GLM_FORCE_EXPLICIT_CTOR`.

```
c+cp#define GLM_FORCE_EXPLICIT_CTOR
c+cp#include c+cpf<glm/glm.hpp>

c+c1// This function is the same as above.
k+ktvoid n+nffoop()
p{
  nglmo::nivec4 nap;

  nglmo::nvec4 nbp(nap); c+c1// Explicit conversion, OK
  nglmo::nvec4 nc o= nap; c+c1// Implicit conversion, ERROR
p}
```

3.9 Static Constants

Occasionally, you may need certain common vectors or matrices with unit-sized components, e.g. for working with coordinate systems. To enable them, define `GLM_STATIC_CONST_MEMBERS`.

```
c+cp#define GLM_STATIC_CONST_MEMBERS
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop()
p{
  nglmo::nivec4 na o= nglmo::nivec4o::nX o+ nglmo::nivec4o::nZp;
  nassertp(na o== nglmo::nivec4o::nXZp);
p}
```

For vectors and quaternions, constants for all possible combinations of the component's values being zero and one are defined. For example, `vec3::XYZ == vec3(1.0f)`. For matrices, the identity and zero matrices are defined.

4 Stable Extensions

GLM provides additional functionality on top of GLSL's, including (but not limited to) quaternions, matrix transformations, random number generation, and color space conversion.

All extra features are part of the `glm` namespace, and can be enabled by including the relevant header file.

```
c+cp#include c+cpf<glm/glm.hpp>
c+cp#include c+cpf<glm/gtc/matrix_transform.hpp>

kusing nglmo::nvec3p;
kusing nglmo::nvec4p;
kusing nglmo::nmat4p;

k+ktvoid n+nffoop()
p{
  nvec4 nPosition o= nvec4p(nvec3p(l+m+mf0.0fp), l+m+mf1.0fp);
  nmat4 nModel o= nglmo::ntranslatep(nmat4p(l+m+mf1.0fp), nvec3p(l+m+mf1.0fp));
  nvec4 nTransformed o= nModel o* nPositionp;
p}
```

Descriptions of the most stable extensions follow.

4.1 `GLM_GTC_bitfield`

Header: `<glm/gtc/bitfield.hpp>`

Fast bitfield operations on scalar and vector variables.

4.2 `GLM_GTC_color_space`

Header: `<glm/gtc/color_space.hpp>`

Conversion between linear RGB and sRGB color spaces.

4.3 `GLM_GTC_constants`

Header: `<glm/gtc/constants.hpp>`

Built-in constants such as π, e, and ϕ.

4.4 GLM_GTC_epsilon

Header: <glm/gtc/epsilon.hpp>

Approximate equality comparisons for floating-point numbers, possibly with a user-defined epsilon.

4.5 GLM_GTC_integer

Header: <glm/gtc/integer.hpp>

Integer variants of core GLM functions.

4.6 GLM_GTC_matrix_access

Header: <glm/gtc/matrix_access.hpp>

Functions to conveniently access the individual rows or columns of a matrix.

4.7 GLM_GTC_matrix_integer

Header: <glm/gtc/matrix_integer.hpp>

Integer matrix types similar to the core floating-point matrices. Some operations (such as inverse and determinant) are not supported.

4.8 GLM_GTC_matrix_inverse

Header: <glm/gtc/matrix_inverse.hpp> Additional matrix inverse functions.

4.9 GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

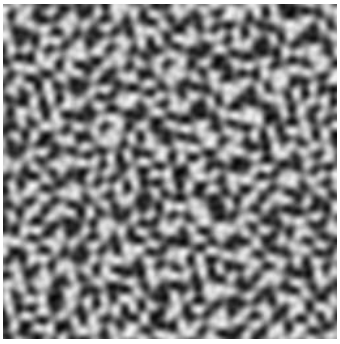


Figure 4.10.1: `simplex(vec2(x / 16.f, y / 16.f));`

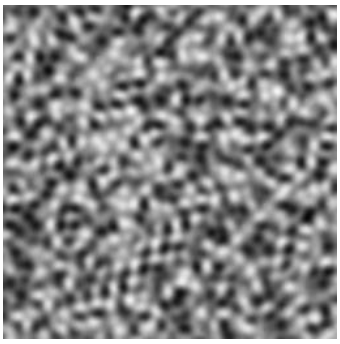


Figure 4.10.2: `simplex(vec3(x / 16.f, y / 16.f, 0.5f));`

Matrix transformation functions that follow the old OpenGL fixed-function conventions. For example, the `lookAt` function generates a transformation matrix that projects world coordinates into eye coordinates suitable for projection matrices (e.g. `perspective`, `ortho`). See the OpenGL compatibility specifications for more information about the layout of these generated matrices.

4.10 `GLM_GTC_noise`

Header: `<glm/gtc/noise.hpp>`

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

4.11 `GLM_GTC_packing`

Header: `<glm/gtc/packing.hpp>`

Convert scalar and vector types to and from packed formats, saving space at the cost of

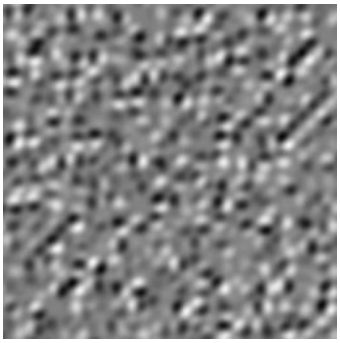


Figure 4.10.3: `simplex(vec4(x / 16.f, y / 16.f, 0.5f, 0.5f));`

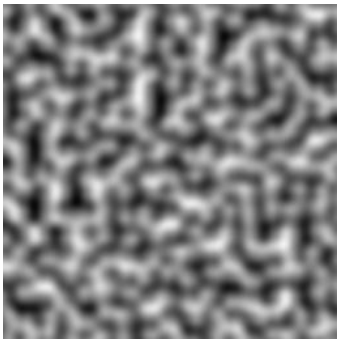


Figure 4.10.4: `perlin(vec2(x / 16.f, y / 16.f));`

Figure 4.10.5: `perlin(vec3(x / 16.f, y / 16.f, 0.5f));`

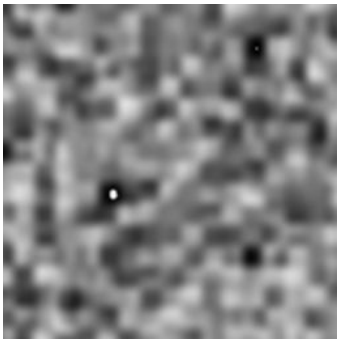


Figure 4.10.6: `perlin(vec4(x / 16.f, y / 16.f, 0.5f, 0.5f)));`

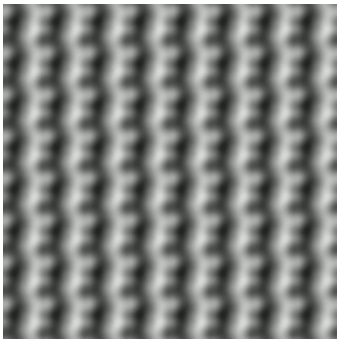


Figure 4.10.7: `perlin(vec2(x / 16.f, y / 16.f), vec2(2.0f));`

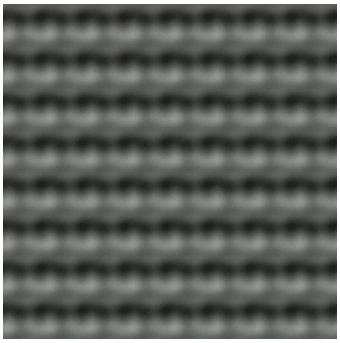


Figure 4.10.8: `perlin(vec3(x / 16.f, y / 16.f, 0.5f), vec3(2.0f));`

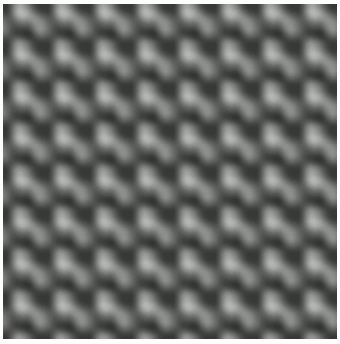


Figure 4.10.9: `perlin(vec4(x / 16.f, y / 16.f, vec2(0.5f)), vec4(2.0f));`

precision. However, packing a value into a format that it was previously unpacked from is **guaranteed** to be lossless.

4.12 `GLM_GTC_quaternion`

Header: `<glm/gtc/quaternion.hpp>`

Quaternions and operations upon thereof.

4.13 `GLM_GTC_random`

Header: `<glm/gtc/random.hpp>` Probability distributions in up to four dimensions..

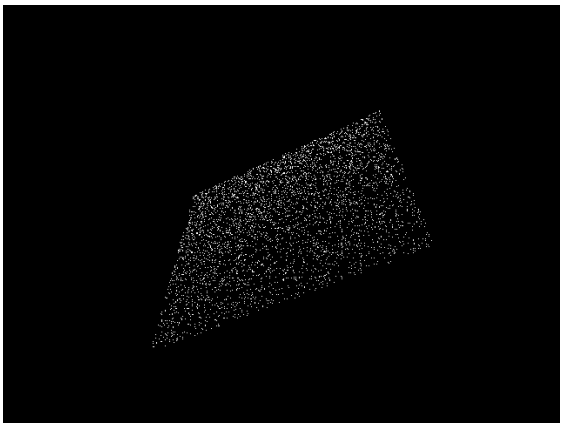


Figure 4.13.1: `vec4(linearRand(vec2(-1), vec2(1)), 0, 1);`

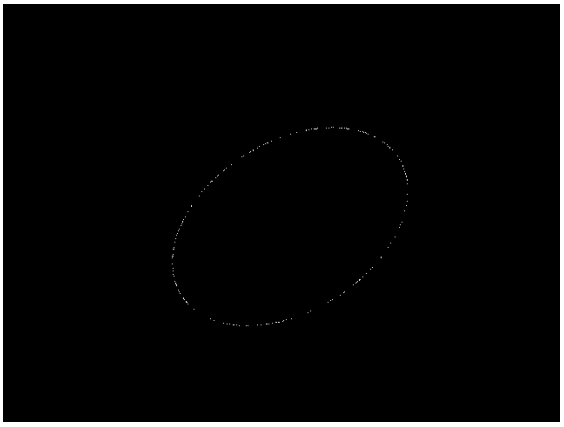


Figure 4.13.2: `vec4(circularRand(1.0f), 0, 1);`

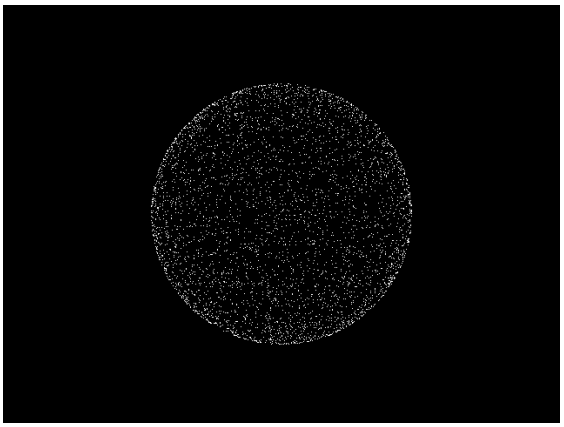


Figure 4.13.3: `vec4(sphericalRand(1.0f), 1);`

4.14 `GLM_GTC_reciprocal`

Header: `<glm/gtc/reciprocal.hpp>`

Reciprocal trigonometric functions (e.g. secant, cosecant, tangent).

4.15 `GLM_GTC_round`

Header: `<glm/gtc/round.hpp>`

Various rounding operations and common special cases thereof.

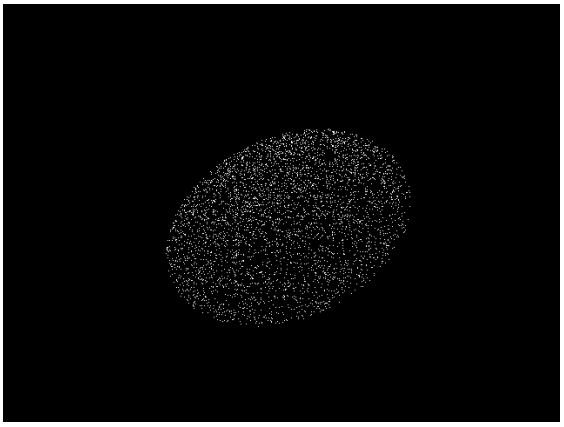


Figure 4.13.4: `vec4(diskRand(1.0f), 0, 1);`

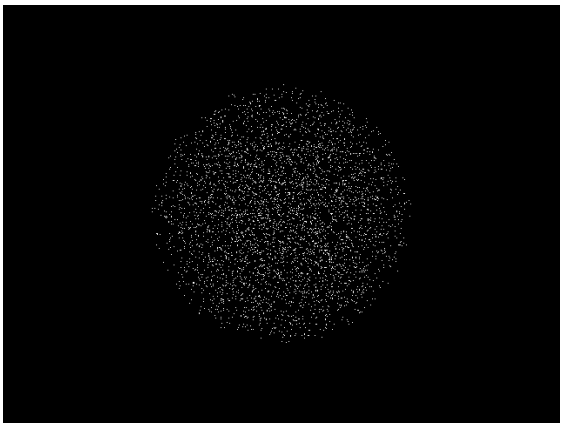


Figure 4.13.5: `vec4(ballRand(1.0f), 1);`

4.16 `GLM_GTC_type_precision`

Header: `<glm/gtc/type_precision.hpp>`

Vector and matrix types with defined precisions, e.g. `i8vec4`, which is a 4D vector of signed 8-bit integers.

This extension adds defines to set the default precision of each class of types added, in a manner identical to that described in section 3.1.

Available defines for signed 8-bit integer types (`glm::i8vec*`):

- `GLM_PRECISION_LOWP_INT8`: Low precision
- `GLM_PRECISION_MEDIUMP_INT8`: Medium precision

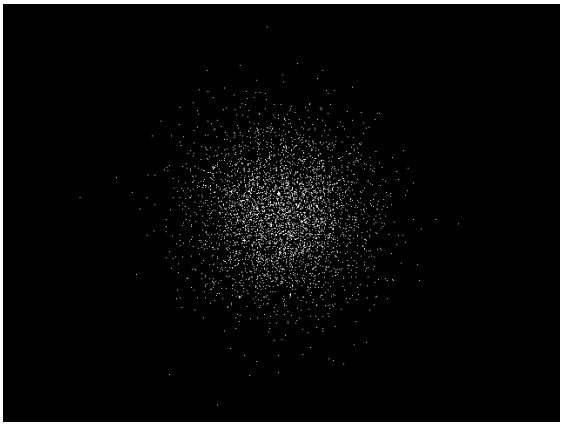


Figure 4.13.6: `vec4(gaussRand(vec3(0), vec3(1)), 1);`

- `GLM_PRECISION_HIGHP_INT8`: High precision (default)

Available defines for unsigned 8-bit integer types (`glm::u8vec*`):

- `GLM_PRECISION_LOWP_UINT8`: Low precision
- `GLM_PRECISION_MEDIUMP_UINT8`: Medium precision
- `GLM_PRECISION_HIGHP_UINT8`: High precision (default)

Available defines for signed 16-bit integer types (`glm::i16vec*`):

- `GLM_PRECISION_LOWP_INT16`: Low precision
- `GLM_PRECISION_MEDIUMP_INT16`: Medium precision
- `GLM_PRECISION_HIGHP_INT16`: High precision (default)

Available defines for unsigned 16-bit integer types (`glm::u16vec*`):

- `GLM_PRECISION_LOWP_UINT16`: Low precision
- `GLM_PRECISION_MEDIUMP_UINT16`: Medium precision
- `GLM_PRECISION_HIGHP_UINT16`: High precision (default)

Available defines for signed 32-bit integer types (`glm::i32vec*`):

- `GLM_PRECISION_LOWP_INT32`: Low precision

- GLM_PRECISION_MEDIUMP_INT32: Medium precision
- GLM_PRECISION_HIGHP_INT32: High precision (default)

Available defines for unsigned 32-bit integer types (glm::u32vec*):

- GLM_PRECISION_LOWP_UINT32: Low precision
- GLM_PRECISION_MEDIUMP_UINT32: Medium precision
- GLM_PRECISION_HIGHP_UINT32: High precision (default)

Available defines for signed 64-bit integer types (glm::i64vec*):

- GLM_PRECISION_LOWP_INT64: Low precision
- GLM_PRECISION_MEDIUMP_INT64: Medium precision
- GLM_PRECISION_HIGHP_INT64: High precision (default)

Available defines for unsigned 64-bit integer types (glm::u64vec*):

- GLM_PRECISION_LOWP_UINT64: Low precision
- GLM_PRECISION_MEDIUMP_UINT64: Medium precision
- GLM_PRECISION_HIGHP_UINT64: High precision (default)

Available defines for 32-bit floating-point types (glm::f32vec*, glm::f32mat*, glm::f32quat*):

- GLM_PRECISION_LOWP_FLOAT32: Low precision
- GLM_PRECISION_MEDIUMP_FLOAT32: Medium precision
- GLM_PRECISION_HIGHP_FLOAT32: High precision (default)

Available defines for 64-bit floating-point types (glm::f64vec*, glm::f64mat*, glm::f64quat*):

- GLM_PRECISION_LOWP_FLOAT64: Low precision
- GLM_PRECISION_MEDIUMP_FLOAT64: Medium precision
- GLM_PRECISION_HIGHP_FLOAT64: High precision (default)

4.17 GLM_GTC_type_ptr

Header: <glm/gtc/type_ptr.hpp>

Facilitate interactions between pointers to raw values (e.g. `float*`) and GLM types (e.g. `mat4`).

This extension defines an overloaded function, `glm::value_ptr`, which returns a pointer to the memory layout of any GLM vector or matrix (`vec3`, `mat4`, etc.). Matrix types store their values in **column-major order**. This is useful for uploading data to matrices or for copying data to buffer objects.

```
c+c1// GLM_GTC_type_ptr provides a safe solution:
c+cp#include c+cpf<glm/glm.hpp>
c+cp#include c+cpf<glm/gtc/type_ptr.hpp>

k+ktvoid n+nffoop()
p{
  nglmo::nvec4 nvp(l+m+mf0.0fp);
  nglmo::nmat4 nmp(l+m+mf1.0fp);

  nglVertex3fvp(nglmo::nvalue_ptrp(nvp))
  nglLoadMatrixfvp(nglmo::nvalue_ptrp(nmp));
p}
```

```
c+c1// Another solution, this one inspired by the STL:
c+cp#include c+cpf<glm/glm.hpp>

k+ktvoid n+nffoop()
p{
  nglmo::nvec4 nvp(l+m+mf0.0fp);
  nglmo::nmat4 nmp(l+m+mf1.0fp);

  nglVertex3fvp(o&nvp[l+m+mi0p]);
  nglLoadMatrixfvp(o&nmp[l+m+mi0p][l+m+mi0p]);
p}
```

Note: It's possible to implement `glVertex` and similar functions by defining an implicit cast from GLM types to pointer types. However, you risk triggering undefined behavior by doing so.

4.18 GLM_GTC_ulp

Header: <glm/gtc/ulp.hpp>

Measure a function's accuracy given a reference implementation of it. This extension works on floating-point data and provides results in ULP.

4.19 GLM_GTC_vec1

Header: <glm/gtc/vec1.hpp>

Add *vec1 types.

5 Substituting Deprecated OpenGL Functions

5.1 OpenGL Substitutes

Most fixed-function APIs were deprecated in OpenGL 3.1, and then removed entirely in OpenGL 3.2. GLM provides substitutes for some of this lost functionality.

5.1.1 glRotate

```
nglmo::nmat4 nglmo::nrotatep(
  nglmo::nmat4 kconst o& nmp,
  k+ktfloat nanglep,
  nglmo::nvec3 kconst o& naxis
p);

nglmo::ndmat4 nglmo::nrotatep(
  nglmo::ndmat4 kconst o& nmp,
  k+ktdouble nanglep,
  nglmo::ndvec3 kconst o& naxis
p);
```

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

5.1.2 glScale

```
nglmo::nmat4 nglmo::nscalep(
  nglmo::nmat4 kconst o& nmp,
  nglmo::nvec3 kconst o& nfactors
p);

nglmo::ndmat4 nglmo::nscalep(
  nglmo::ndmat4 kconst o& nmp,
  nglmo::ndvec3 kconst o& nfactors
p);
```

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

5.1.3 glTranslate

```
nglmo::nmat4 nglmo::ntranslatep(
  nglmo::nmat4 kconst o& nmp,
  nglmo::nvec3 kconst o& ntranslation
p);

nglmo::ndmat4 nglmo::ntranslatep(
  nglmo::ndmat4 kconst o& nmp,
  nglmo::ndvec3 kconst o& ntranslation
p);
```

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

5.1.4 glLoadIdentity

```
nglmo::nmat4p(l+m+mf1.0p) nor nglmo::nmat4p();
nglmo::ndmat4p(l+m+mf1.0p) nor nglmo::ndmat4p();
```

Extension: None, part of GLM core

Header: <glm/glm.hpp>

5.1.5 glMultMatrix

```
nglmo::nmat4p() o* nglmo::nmat4p();
nglmo::ndmat4p() o* nglmo::ndmat4p();
```

Extension: None, part of GLM core

Header: <glm/glm.hpp>

5.1.6 glLoadTransposeMatrix

```
nglmo::ntransposep(nglmo::nmat4p());
nglmo::ntransposep(nglmo::ndmat4p());
```

Extension: None, part of GLM core

Header: <glm/glm.hpp>

5.1.7 glMultTransposeMatrix

```
nglmo::nmat4p() o* nglmo::ntransposep(nglmo::nmat4p());
nglmo::ndmat4p() o* nglmo::ntransposep(nglmo::ndmat4p());
```

Extension: None, part of GLM core

Header: <glm/glm.hpp>

5.1.8 glFrustum

```
nglmo::nmat4 nglmo::nfrustump(
  k+ktfloat nleftp, k+ktfloat nrightp,
  k+ktfloat nbottomp, k+ktfloat ntopp,
  k+ktfloat nzNearp, k+ktfloat nzFar
p);

nglmo::ndmat4 nglmo::nfrustump(
  k+ktdouble nleftp, k+ktdouble nrightp,
  k+ktdouble nbottomp, k+ktdouble ntopp,
  k+ktdouble nzNearp, k+ktdouble nzFar
p);
```

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

5.1.9 glOrtho

```
nglmo::nmat4 nglmo::northop(
  k+ktfloat nleftp, k+ktfloat nrightp,
  k+ktfloat nbottomp, k+ktfloat ntopp,
  k+ktfloat nzNearp, k+ktfloat nzFar
p);

nglmo::ndmat4 nglmo::northop(
  k+ktdouble nleftp, k+ktdouble nrightp,
  k+ktdouble nbottomp, k+ktdouble ntopp,
  k+ktdouble nzNearp, k+ktdouble nzFar
p);
```

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

5.2 GLU Substitutes

5.2.1 gluLookAt

```
nglmo::nmat4 nglmo::nlookAtp(
  nglmo::nvec3 kconst o& neyep,
  nglmo::nvec3 kconst o& ncenterp,
  nglmo::nvec3 kconst o& nup
p);

nglmo::ndmat4 nglmo::nlookAtp(
  nglmo::ndvec3 kconst o& neyep,
  nglmo::ndvec3 kconst o& ncenterp,
  nglmo::ndvec3 kconst o& nup
p);
```

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

5.2.2 gluOrtho2D

```
nglmo::nmat4 nglmo::northop(
  k+ktfloat nleftp, k+ktfloat nrightp, k+ktfloat nbottomp, k+ktfloat ntop
p);

nglmo::ndmat4 nglmo::northop(
  k+ktdouble nleftp, k+ktdouble nrightp, k+ktdouble nbottomp, k+ktdouble ntop
p);
```

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

5.2.3 gluPerspective

```
nglmo::nmat4 nperspectivep(
  k+ktfloat nfovyp, k+ktfloat naspectp, k+ktfloat nzNearp, k+ktfloat nzFar
p);

nglmo::ndmat4 nperspectivep(
  k+ktdouble nfovyp, k+ktdouble naspectp, k+ktdouble nzNearp, k+ktdouble nzFar
p);
```

Note that in GLM, fovy is expressed in *radians*, not degrees.

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

5.2.4 gluPickMatrix

```
nglmo::nmat4 npickMatrixp(
  nglmo::nvec2 kconst o& ncenterp,
  nglmo::nvec2 kconst o& ndeltap,
  nglmo::nivec4 kconst o& nviewport
p);

nglmo::ndmat4 npickMatrixp(
  nglmo::ndvec2 kconst o& ncenterp,
  nglmo::ndvec2 kconst o& ndeltap,
  nglmo::nivec4 kconst o& nviewport
p);
```

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

5.2.5 gluProject

```
nglmo::nvec3 nprojectp(
  nglmo::nvec3 kconst o& nobjp,
  nglmo::nmat4 kconst o& nmodelp,
  nglmo::nmat4 kconst o& nprojp,
  nglmo::nvec4 kconst o& nviewport
p);

nglmo::ndvec3 nprojectp(
  nglmo::ndvec3 kconst o& nobjp,
  nglmo::ndmat4 kconst o& nmodelp,
  nglmo::ndmat4 kconst o& nprojp,
  nglmo::nvec4 kconst o& nviewport
p);
```

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

5.2.6 gluUnProject

```
nglmo::nvec3 nunProjectp(
  nglmo::nvec3 kconst o& nwinp,
  nglmo::nmat4 kconst o& nmodelp,
  nglmo::nmat4 kconst o& nprojp,
  nglmo::nvec4 kconst o& nviewport
p);

nglmo::ndvec3 nunProjectp(
  nglmo::ndvec3 kconst o& nwinp,
  nglmo::ndmat4 kconst o& nmodelp,
  nglmo::ndmat4 kconst o& nprojp,
  nglmo::nvec4 kconst o& nviewport
p);
```

Extension: GLM_GTC_matrix_transform

Header: <glm/gtc/matrix_transform.hpp>

6 Known Issues

This section reports GLSL features that GLM can't accurately emulate due to language restrictions.

6.1 The `not` Function

The GLSL function `not` is a keyword in C++. To prevent name collisions and ensure a consistent API, the name `not_` (note the underscore) is used instead.

6.2 Precision Qualifiers

GLM supports GLSL precision qualifiers through prefixes instead of keywords. For example, GLM provides `lowp_vec4`, `mediump_vec4` and `highp_vec4` as variations of `vec4`.

As in GLSL, GLM precision qualifiers are used to exchange precision for performance. By default, all types use high precision.

```
c+c1// Using precision qualifiers in GLSL:

kivec3 nfoop(kin kvec4 nvp)
p{
  khighp kvec4 na o= nvp;
  kmediump kvec4 nb o= nap;
  klowp kivec3 nc o= kivec3p(nbp);

  kreturn ncp;
p}
```

```
c+c1// Using precision qualifiers in GLM:
c+cp#include c+cpf<glm/glm.hpp>

nivec3 n+nffoop(kconst nvec4 o& nvp)
p{
  nhighp_vec4 na o= nvp;
  nmedium_vec4 nb o= nap;
  nlowp_ivec3 nc o= nglmo::nivec3p(nbp);

  kreturn ncp;
p}
```

The syntax for default precision specifications in GLM differs from that in GLSL; for more information, see section 3.1.

7 FAQ

7.1 Why does GLM follow the GLSL specification?

Everyone and their dog has their own idea of what should constitute a math library. The designers of GLSL (the OpenGL ARB) make a living on deciding what its own should offer; why not learn from the best and stick to a proven convention?

7.2 Does GLM actually run GLSL?

No. GLM implements a subset of GLSL's functionality.

7.3 Can GLM be compiled to GLSL (or vice versa)?

No. That is not one of GLM's goals.

7.4 What's the difference between GTX, GTC, and Core?

GTX extensions are considered *experimental*, and may thus introduce breaking changes without warning (though in practice this doesn't happen often). GTC extensions are considered *stable*, and are therefore unlikely to introduce radical changes. Core libraries are based entirely on functionality provided by GLSL, and can safely be relied upon. The GTX and GTC extension system provides a way to experiment with new GLM functionality, in the hopes of eventually promoting it to a GTC extension. OpenGL itself is developed in much the same way.

7.5 Where's the documentation?

The Doxygen-generated documentation includes a complete list of all extensions, and can be found here.

7.6 Should I use `using namespace glm;`?

Absolutely not! GLM uses many common names, such as `any`, `scale`, and `length`. Haphazardly writing `using namespace glm;` will almost certainly result in name collisions. Instead, either prefix GLM calls with `glm::`, or pull individual types or functions into your namespace with `using glm::*`, where * is some desired name.

7.7 Is GLM fast?

GLM is designed for convenience over performance. *That being said*, the most frequently-used operations are optimized to the fullest reasonable extent. Approximations and SIMD-flavored structures are provided as well, in case they're needed.

7.8 Visual C++ gives me lots of warnings on warning level `/W4`.

You should not have any warnings, even in `/W4` mode. However, if you expect such level for your code, then you should ask for the same level to the compiler by at 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.

7.9 Why are some GLM functions vulnerable to division by zero?

Such behavior follows the precedent set by C and C++'s standard library, in that it's treated as a domain error. For example, it's a domain error to pass a null vector (all zeroes) to `glm::normalize`, or to pass a negative number into `std::sqrt`.

7.10 What unit does GLM use for angles?

All angles in GLM are expressed in radians unless otherwise noted. GLSL does the same thing. GLU uses degrees, however. This used to cause a lot of confusion. For more information, see here.

8 Code Samples

8.1 Triangle Normal

```
c+cp#include c+cpf<glm/glm.hpp> // vec3 normalize cross
c+cp#include c+cpf<glm/gtx/fast_square_root.hpp> // fastNormalize

kusing nglmo::nvec3p;

nvec3 n+nftriNormalp(nvec3 kconst o& nap, nvec3 kconst o& nbp, nvec3 kconst o& ncp)
p{
  kreturn nglmo::nnormalizep(nglmo::ncrossp(nc o- nap, nb o- nap));
p}

c+c1// A faster (but less accurate) alternative:
nvec3 n+nffastTriNormalp(nvec3 kconst o& nap, nvec3 kconst o& nbp, nvec3 kconst o& ncp)
p{
  kreturn nglmo::nfastNormalizep(nglmo::ncrossp(nc o- nap, nb o- nap));
p}
```

8.2 Matrix Transformations

```
c+cp#define GLM_STATIC_CONST_MEMBERS c+c1// For unit vectors
c+cp#include c+cpf<glm/glm.hpp> // vec3, vec4, ivec4, mat4
c+cp#include c+cpf<glm/gtc/matrix_transform.hpp> // translate, scale, etc.
c+cp#include c+cpf<glm/gtc/type_ptr.hpp> // value_ptr

kusing knamespace nglmp; c+c1// DON'T DO THIS; only done for brevity!

k+ktvoid n+nfsetUniformMVPp(nGLuint nLocationp, nvec3 kconst o& nTp, nvec3 kconst o& nRp
p{
  nmat4 nProjection o= nperspectivep(l+m+mf45.0fp, l+m+mf4.0f o/ l+m+mf3.0fp, l+m+mf0.1f
  nmat4 nViewTranslate o= ntranslatep(nmat4p(l+m+mf1.0fp), nTp);
  nmat4 nViewRotateX o= nrotatep(nViewTranslatep, nRp.nyp, o-nvec3o::nXp);
  nmat4 nView o= nrotatep(nViewRotateXp, nRp.nxp, nvec3o::nYp);
  nmat4 nModel o= nscalep(nmat4p(l+m+mf1.0fp), nvec3p(l+m+mf0.5fp));
  nmat4 nMVP o= nProjection o* nView o* nModelp;

  nglUniformMatrix4fvp(nLocationp, l+m+mi1p, nGL_FALSEp, nvalue_ptrp(nMVPp));
p}
```

8.3 Vector Types

```
c+cp#include c+cpf<glm/glm.hpp> //vec2
c+cp#include c+cpf<glm/gtc/type_precision.hpp> //hvec2, i8vec2, i32vec2

kusing knamespace nglmp; c+c1// DON'T DO THIS; only done for brevity!

nstdo::k+ktsize_t kconst nVertexCount o= l+m+mi4p;

c+c1// Float quad geometry
nvec2 kconst nPositionDataF32p[nVertexCountp] o= p{
  nvec2p(o-l+m+mf1.0fp, o-l+m+mf1.0fp),
  nvec2p( l+m+mf1.0fp, o-l+m+mf1.0fp),
  nvec2p( l+m+mf1.0fp,  l+m+mf1.0fp),
  nvec2p(o-l+m+mf1.0fp,  l+m+mf1.0fp)
p};

c+c1// Half-float quad geometry
nhvec2 kconst nPositionDataF16p[nVertexCountp] o= p{
  nhvec2p(o-l+m+mf1.0fp, o-l+m+mf1.0fp),
  nhvec2p( l+m+mf1.0fp, o-l+m+mf1.0fp),
  nhvec2p( l+m+mf1.0fp,  l+m+mf1.0fp),
  nhvec2p(o-l+m+mf1.0fp,  l+m+mf1.0fp)
p};

c+c1// 8 bits signed integer quad geometry
ni8vec2 kconst nPositionDataI8p[nVertexCountp] o= p{
  ni8vec2p(o-l+m+mi1p, o-l+m+mi1p),
  ni8vec2p( l+m+mi1p, o-l+m+mi1p),
  ni8vec2p( l+m+mi1p,  l+m+mi1p),
  ni8vec2p(o-l+m+mi1p,  l+m+mi1p)
p};

c+c1// 32 bits signed integer quad geometry
ni32vec2 kconst nPositionDataI32p[nVertexCountp] o= p{
  ni32vec2p(o-l+m+mi1p, o-l+m+mi1p),
  ni32vec2p( l+m+mi1p, o-l+m+mi1p),
  ni32vec2p( l+m+mi1p,  l+m+mi1p),
  ni32vec2p(o-l+m+mi1p,  l+m+mi1p)
p};
```

8.4 Lighting

```
c+cp#include c+cpf<glm/glm.hpp> // vec3, normalize, reflect, dot, pow
c+cp#include c+cpf<glm/gtx/random.hpp> // vecRand3

kusing knamespace nglmp; c+c1// DON'T DO THIS; only done for brevity!

nvec3 n+nflightingp(
  nintersection kconst o& nIp,
  nmaterial kconst o& nMp,
  nlight kconst o& nLp,
  nvec3 kconst o& nView
p)
p{
  nvec3 nIntersectionPos o= nIp.nglobalPositionp();
  nvec3 nLightPos o= nLp.npositionp();
  nvec3 nColor o= nvec3p(l+m+mf0.0fp);
  nvec3 nLightVector o= nnormalizep(
    nLightPos o- nIntersectionPos o+ nvecRand3p(l+m+mf0.0fp, nLp.ninaccuracyp())
  p);
  c+c1// vecRand3 generates a uniformly random normalized vec3

  kif p(o!nshadowp(nIntersectionPosp, nLightPosp, nLightVectorp)) p{
    k+ktfloat nDiffuse o= ndotp(nIp.nnormalp(), nLightVectorp);
    kif p(nDiffuse o<= l+m+mf0.0fp)
      kreturn nColorp;

    kif p(nMp.nisDiffusep())
      nColor o+= nLp.ncolorp() o* nMp.ndiffusep() o* nDiffusep;

    kif p(nMp.nisSpecularp()) p{
      nvec3 nReflect o= nreflectp(o-nLightVectorp, nIp.nnormalp());
      k+ktfloat nDot o= ndotp(nReflectp, nViewp);
      k+ktfloat nBase o= nDot o> l+m+mf0.0f o? n+nlDot p: l+m+mf0.0fp;
      k+ktfloat nSpecular o= npowp(nBasep, nMp.nexponentp());
      nColor o+= nMp.nspecularp() o* nSpecularp;
    p}
  p}

  kreturn nColorp;
p}
```

9 References

9.1 Official GLM Resources

- Official website
- Latest stable release
- HEAD snapshot
- Issue tracker
- G-Truc Creation's Website

9.2 OpenGL Specifications

- OpenGL 4.3 core specification
- GLSL 4.30 specification
- GLU 1.3 specification

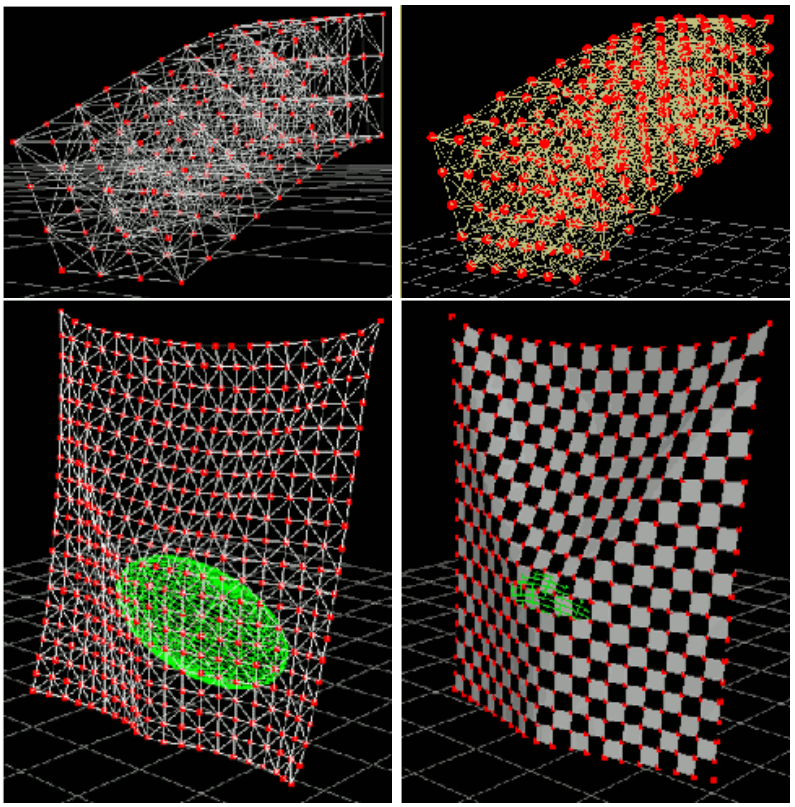
9.3 Projects Using GLM

9.3.1 Outerra

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

9.3.2 opencloth

A collection of cloth simulation demos, visualized with OpenGL.



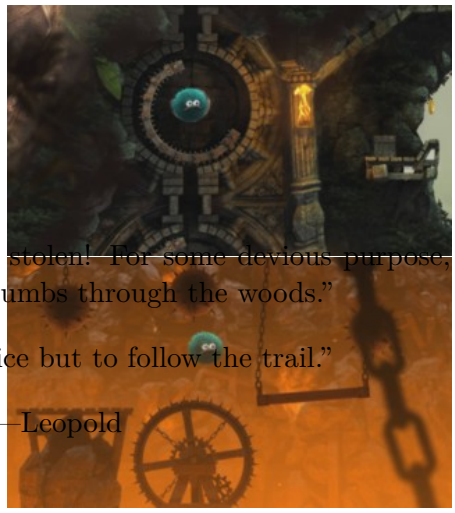
9.3.3 OpenGL 4.0 Shading Language Cookbook

A set of recipes that demonstrates a wide variety of techniques for producing high-quality, real-time 3D graphics with GLSL 4.0, such as:

- Using GLSL 4.0 to implement lighting and shading techniques.
- Using the new features of GLSL 4.0 including tessellation and geometry shaders.
- Using 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 are provided, as well as a basic description of the theory behind each technique.

9.3.4 Leo's Forture



Leo's Fortune is a platform adventure game in which you hunt down the cunning and mysterious thief that stole your gold. Available on PS4, Xbox One, PC, Mac, iOS and Android.

Beautiful hand-crafted levels bring Leo's story to life in this epic adventure.

"I just returned home to find all my gold has been stolen! For some devious purpose, the thief has dropped pieces of my gold like breadcrumbs through the woods."

"Despite this pickle of a trap, I am left with no choice but to follow the trail."

"Whatever lies ahead, I must recover my fortune." —Leopold

Are you using GLM in a project? Tell us!

9.4 OpenGL Tutorials using GLM

- The OpenGL Samples Pack: Examples that demonstrate OpenGL's many features

- Learning Modern 3D Graphics Programming: A great OpenGL tutorial by Jason L. McKesson
- Morten Nobel-Jørgensen's review and OpenGL renderer
- Swiftless' OpenGL tutorial by Donald Urquhart
- Rastergrid: Technical articles (with companion programs) by Daniel Rákos
- OpenGL Tutorial: OpenGL tutorials
- OpenGL Programming on Wikibooks: For beginners just discovering OpenGL
- 3D Game Engine Programming: 3D game engine programming techniques
- Game Tutorials: Graphics and game programming tutorials
- open.gl: An OpenGL tutorial
- c-jump: A GLM tutorial
- Learn OpenGL: An OpenGL tutorial

Are you using GLM in a tutorial? Tell us!

9.5 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.

Are you using or developing an alternative to GLM? Tell us!

9.6 Acknowledgements

GLM is developed and maintained by Christophe Riccio, but relies on the support of its many contributors to stay great.

Special thanks to:

- Ashima Arts and Stefan Gustavson for their work on webgl-noise, which which is the base of `GLM_GTC_noise`.
- Arthur Winters for contributing to the swizzle operator implementation.
- Joshua Smith and Christoph Schied for their contributions to the swizzle expression implementation.
- Guillaume Chevallereau for providing and maintaining the nightly build system.
- Ghenadii Ursachi for `GLM_GTX_matrix_interpolation`.
- Mathieu Roumillac for providing some implementation ideas.
- Grant James for non-square matrix multiplication (e.g. `mat3 * mat3x2`)
- GLM's users! Without you, there is no GLM!

9.7 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 Mathematics 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 code base 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