GLM 0.9.9 Manual

Table of Contents

Licenses

The Happy Bunny License (Modified MIT License)

The MIT License

---

1. Getting started

1.1. Setup

GLM is a header-only library, and thus does not need to be compiled. We can use GLM’s implementation of GLSL’s mathematics functionality by including the `<glm/glm.hpp>` header. The library can also be installed with CMake, though the details of doing so will differ depending on the target build system.

Features can also be included individually to shorten compilation times.

```
#include <glm/vec2.hpp> // vec2, bvec2, dvec2, ivec2 and uvec2
#include <glm/vec3.hpp> // vec3, bvec3, dvec3, ivec3 and uvec3
#include <glm/vec4.hpp> // vec4, bvec4, dvec4, ivec4 and uvec4
#include <glm/mat2x2.hpp> // mat2, dmat2
```

```
#include <glm/mat2x3.hpp> // mat2x3, dmat2x3
#include <glm/mat2x4.hpp> // mat2x4, dmat2x4
#include <glm/mat3x2.hpp> // mat3x2, dmat3x2
#include <glm/mat3x3.hpp> // mat3, dmat3
#include <glm/mat3x4.hpp> // mat3x4, dmat2
#include <glm/mat4x2.hpp> // mat4x2, dmat4x2
#include <glm/mat4x3.hpp> // mat4x3, dmat4x3
#include <glm/mat4x4.hpp> // mat4, dmat4
#include <glm/common.hpp> // all the GLSL common functions
#include <glm/exponential.hpp> // all the GLSL exponential functions
#include <glm/geometry.hpp> // all the GLSL geometry functions
#include <glm/integer.hpp> // all the GLSL integer functions
#include <glm/matrix.hpp> // all the GLSL matrix functions
#include <glm/packing.hpp> // all the GLSL packing functions
#include <glm/trigonometric.hpp> // all the GLSL trigonometric functions
#include <glm/vector_relational.hpp> // all the GLSL vector relational functions
```

1.2. Faster compilation

GLM uses C++ templates heavily, and may significantly increase compilation times for projects that use it. Hence, source files should only include the headers they actually use.

To reduce compilation time, we can include `<glm/fwd.hpp>`, which forward-declares all types should their definitions not be needed.

```
// Header file (forward declarations only)
#include <glm/fwd.hpp>

// At this point, we don't care what exactly makes up a vec2; that won't matter
// until we write this function's implementation.
glm::vec2 functionDeclaration(const glm::vec2& input);
```

Precompiled headers will also be helpful, though are not covered by this manual.

1.3. Example usage

```
// Include GLM core features
#include <glm/vec3.hpp>
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>
#include <glm/trigonometric.hpp>

// Include GLM extensions
#include <glm/gtc/matrix_transform.hpp>

glm::mat4 transform(glm::vec2 const& Orientation, glm::vec3 const& Translate, glm::vec3 const& Up)
{
    glm::mat4 Proj = glm::perspective(glm::radians(45.f), 1.33f, 0.1f, 10.f);
    glm::mat4 ViewTranslate = glm::translate(glm::mat4(1.f), Translate);
    glm::mat4 ViewRotateX = glm::rotate(ViewTranslate, Orientation.y, Up);
    glm::mat4 View = glm::rotate(ViewRotateX, Orientation.x, Up);
```

```
    glm::mat4 Model = glm::mat4(1.0f);
    return Proj * View * Model;
}
```

1.4. Dependencies

GLM does not depend on external libraries or headers such as `<GL/gl.h>`, `<GL/glcorearb.h>`, `<GLES3/gl3.h>`, `<GL/glu.h>`, or `<windows.h>`.

---

2. Swizzling

Shader languages like GLSL often feature so-called swizzle expressions, 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), and `stpq` (usually for texture coordinates).

```
vec4 A;
vec2 B;

B.yx = A.wy;
B = A.xx;
vec3 C = A.bgr;
vec3 D = B.rsz; // 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_FORCE_SWIZZLE` before including any GLM header files, or as part of a 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.

```
#define GLM_FORCE_SWIZZLE // Or defined when building (e.g. -DGLM_FORCE_SWIZZLE)
#include <glm/glm.hpp>

void foo()
{
    glm::vec4 ColorRGBA(1.0f, 0.5f, 0.0f, 1.0f);
    glm::vec3 ColorBGR = ColorRGBA.bgr();

    glm::vec3 PositionA(1.0f, 0.5f, 0.0f, 1.0f);
    glm::vec3 PositionB = PositionXYZ.xyz() * 2.0f;

    glm::vec2 TexcoordST(1.0f, 0.5f);
    glm::vec4 TexcoordSTPQ = TexcoordST.stst();
}
```

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

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

void foo()
{
  glm::vec3 A(1.0f, 0.5f, 0.0f);

  // No compiler error, but A is not modified.
  // An anonymous copy is being modified (and then discarded).
  A.bgr() = glm::vec3(2.0f, 1.5f, 1.0f); // A is not modified!
}
```

2.2. Anonymous union member implementation

Visual C++ supports, as a *non-standard language extension*, anonymous `struct`s as `union` members. This permits a powerful swizzling implementation that both allows L-value swizzle expressions and GLSL-like syntax. To use this feature, the language extension must be enabled by a supporting compiler and `GLM_FORCE_SWIZZLE` must be `#define`d.

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

// Only guaranteed to work with Visual C++!
// Some compilers that support Microsoft extensions may compile this.
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;
}
```

This version returns implementation-specific objects that *implicitly convert* to their respective vector types. As a consequence of this design, these extra types **can't be directly used** by GLM functions; they must be converted through constructors or `operator()`.

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

using glm::vec4;

void foo()
{
  vec4 Color(1.0f, 0.5f, 0.0f, 1.0f);
```

```
  // Generates compiler errors. Color.rgba is not a vector type.
  vec4 ClampedA = glm::clamp(Color.rgba, 0.f, 1.f); // ERROR

  // Explicit conversion through a constructor
  vec4 ClampedB = glm::clamp(vec4(Color.rgba), 0.f, 1.f); // OK

  // Explicit conversion through operator()
  vec4 ClampedC = glm::clamp(Color.rgba(), 0.f, 1.f); // OK
}
```

---

3. Preprocessor options

3.1. GLM_PRECISION_**: Default precision

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

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

To use the default precision functionality, GLM provides some defines that need to added before any include of `glm.hpp` :

```
#define GLM_PRECISION_MEDIUMP_INT
#define GLM_PRECISION_HIGHP_FLOAT
#include <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 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. GLM_FORCE_MESSAGES: Compile-time message system

GLM includes a notification system which can display some information at build time:

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

This system is disabled by default. To enable this system, define GLM_FORCE_MESSAGES before any inclusion of <glm/glm.hpp>. The messages are generated only by compiler supporting #program message and only once per project build.

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

3.3. GLM_FORCE_CXX**: C++ language detection

GLM will automatically take advantage of compilers' language extensions when 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> to restrict the language feature set C++98:

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

For C++11 and C++14, equivalent defines are available: GLM_FORCE_CXX11, GLM_FORCE_CXX14.

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

// If the compiler doesn't support C++11, compiler errors will happen.
```

GLM_FORCE_CXX14 overrides GLM_FORCE_CXX11 and GLM_FORCE_CXX11 overrides GLM_FORCE_CXX98 defines.

3.4. SIMD support

GLM provides some SIMD optimizations based on compiler intrinsics. These optimizations will be automatically thanks to compiler arguments. For example, if a program is compiled with Visual Studio using /arch:AVX, GLM will detect this argument and generate code using AVX instructions automatically when available.

It's possible to avoid the instruction set detection by forcing the use of a specific instruction set with one of the fallowing define: GLM_FORCE_SSE2, GLM_FORCE_SSE3, GLM_FORCE_SSSE3, GLM_FORCE_SSE41, GLM_FORCE_SSE42, GLM_FORCE_AVX, GLM_FORCE_AVX2 or GLM_FORCE_AVX512.

The use of intrinsic functions by GLM implementation can be avoided using the define GLM_FORCE_PURE before any inclusion of GLM headers.

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

// GLM code will be compiled using pure C++ code without any intrinsics
```

```
#define GLM_FORCE_AVX2
#include <glm/glm.hpp>

// If the compiler doesn't support AVX2 instrinsics, compiler errors will happen.
```

Additionally, GLM provides a low level SIMD API in glm/simd directory for users who are really interested in writing fast algorithms.

3.5. GLM_FORCE_INLINE: 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>
```

3.6. GLM_FORCE_SIZE_T_LENGTH: Vector and matrix static size

GLSL supports the member function .length() for all vector and matrix types.

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

void foo(vec4 const& v)
{
    int Length = v.length();
    ...
}
```

This function returns a int however this function typically interacts with STL size_t based code. GLM provides GLM_FORCE_SIZE_T_LENGTH pre-processor option so that member functions length() return a size_t.

Additionally, GLM defines the type glm::length_t to identify length() returned type, independently from GLM_FORCE_SIZE_T_LENGTH.

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

void foo(vec4 const& v)
{
    glm::length_t Length = v.length();
    ...
}
```

3.7. GLM_FORCE_EXPLICIT_CTOR: Requiring explicit conversions

GLSL supports implicit conversions of vector and matrix types. For example, an ivec4 can be implicitly converted into vec4.

Often, this behaviour is not desirable but following the spirit of the library, this behavior is supported in GLM. However, GLM 0.9.6 introduced the define GLM_FORCE_EXPLICIT_CTOR to require explicit conversion for GLM types.

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

void foo()
{
    glm::ivec4 a;
    ...

    glm::vec4 b(a); // Explicit conversion, OK
    glm::vec4 c = a; // Implicit conversion, OK
    ...
}
```

With GLM_FORCE_EXPLICIT_CTOR define, implicit conversions are not allowed:

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

void foo()
{
    glm::ivec4 a;
    {
        glm::vec4 b(a); // Explicit conversion, OK
        glm::vec4 c = a; // Implicit conversion, ERROR
        ...
}
```

3.8. GLM_FORCE_UNRESTRICTED_GENTYPE: Removing genType restriction

By default GLM only supports basic types as genType for vector, matrix and quaternion types:

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

typedef glm::vec<4, float> my_fvec4;
```

GLM 0.9.8 introduced GLM_FORCE_UNRESTRICTED_GENTYPE define to relax this restriction:

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

#include "half.hpp" // Define “half” class with behavior equivalent to “float”

typedef glm::vec<4, half> my_hvec4;
```

However, defining GLM_FORCE_UNRESTRICTED_GENTYPE is not compatible with GLM_FORCE_SWIZZLE and will generate a compilation error if both are defined at the same time.

3.9. GLM_FORCE_SINGLE_ONLY: Removed explicit 64-bits floating point types

Some platforms (Dreamcast) doesn't support double precision floating point values. To compile on such platforms, GCC has the --m4-single-only build argument. When defining GLM_FORCE_SINGLE_ONLY before including GLM headers, GLM releases the requirement of double precision floating point values support. Effectivement, all the float64 types are no longer defined and double behaves like float.

---

4. Stable extensions

GLM extends the core GLSL feature set with extensions. These extensions include: quaternion, transformation, spline, matrix inverse, color spaces, etc.

To include an extension, we only need to include the dedicated header file. Once included, the 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;
}
```

When an extension is included, all the dependent core functionalities and extensions will be included as well.

4.1. GLM_GTC_bitfield

Fast bitfield operations on scalar and vector variables.

<glm/gtc/bitfield.hpp> need to be included to use these features.

4.2. GLM_GTC_color_space

Conversion between linear RGB and sRGB color spaces.

<glm/gtc/color_space.hpp> need to be included to use these features.

4.3. GLM_GTC_constants

Provide a list of built-in constants.

<glm/gtc/constants.hpp> need to be included to use these features.

4.4. GLM_GTC_epsilon

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

<glm/gtc/epsilon.hpp> need to be included to use these features.

4.5. GLM_GTC_integer

Integer variants of core GLM functions.

<glm/gtc/integer.hpp> need to be included to use these features.

4.6. GLM_GTC_matrix_access

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

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

4.7. GLM_GTC_matrix_integer

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

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

4.8. GLM_GTC_matrix_inverse

Additional matrix inverse functions.

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

4.9. GLM_GTC_matrix_transform

Matrix transformation functions that follow the 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.

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 define the particular layout of this eye space.

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

4.10. GLM_GTC_noise

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

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

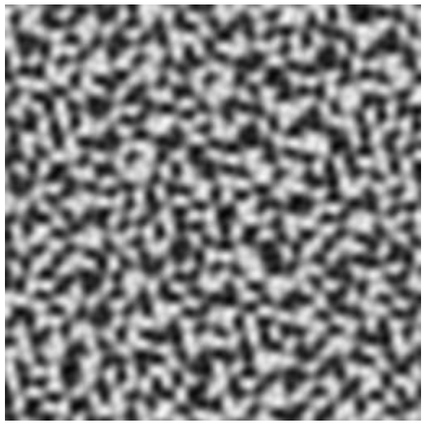


Figure 4.10.1: glm::simplex(glm::vec2(x / 16.f, y / 16.f));

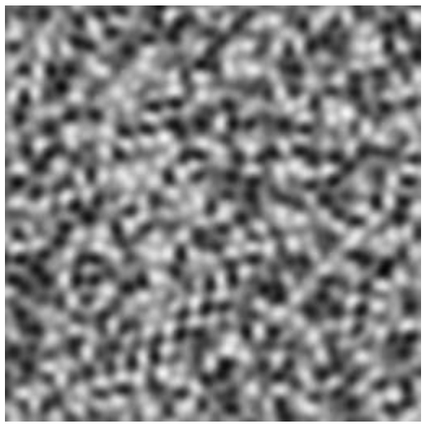


Figure 4.10.2: glm::simplex(glm::vec3(x / 16.f, y / 16.f, 0.5f));

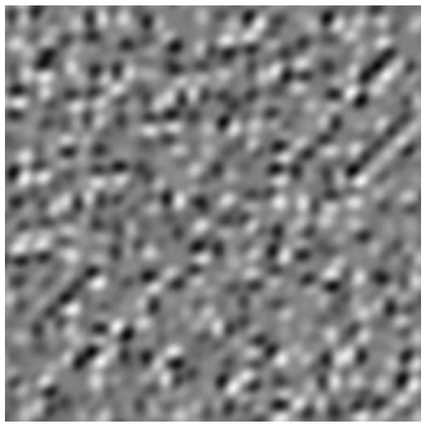


Figure 4.10.3: glm::simplex(glm::vec4(x / 16.f, y / 16.f, 0.5f, 0.5f));

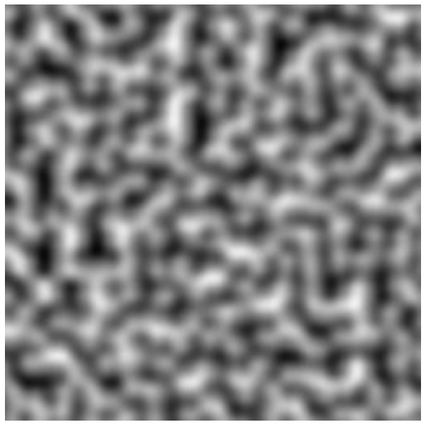


Figure 4.10.4: glm::perlin(glm::vec2(x / 16.f, y / 16.f));

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

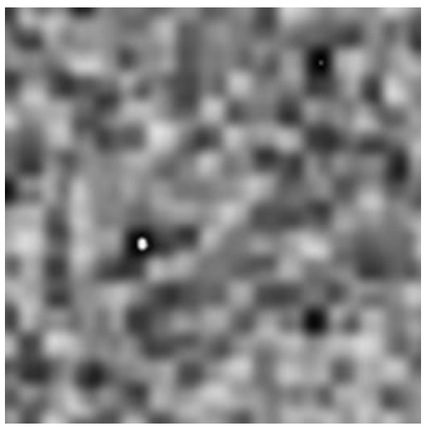


Figure 4.10.6: glm::perlin(glm::vec4(x / 16.f, y / 16.f, 0.5f, 0.5f)));

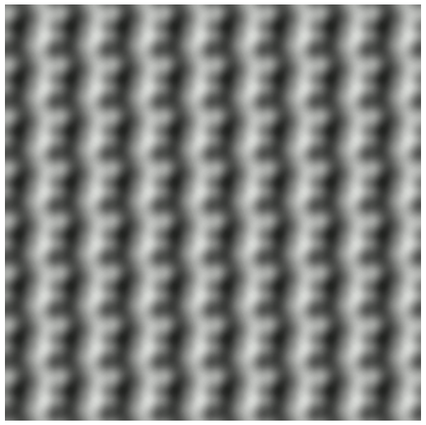


Figure 4.10.7: glm::perlin(glm::vec2(x / 16.f, y / 16.f), glm::vec2(2.0f));

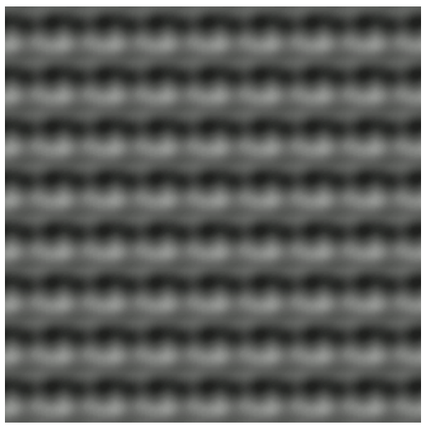


Figure 4.10.8: glm::perlin(glm::vec3(x / 16.f, y / 16.f, 0.5f), glm::vec3(2.0f));

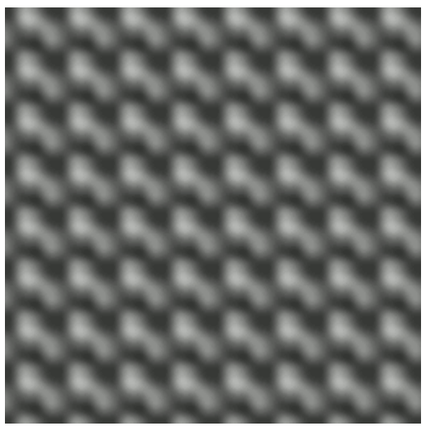


Figure 4.10.9: glm::perlin(glm::vec4(x / 16.f, y / 16.f, glm::vec2(0.5f)), glm::vec4(2.0f));

4.11. GLM_GTC_packing

Convert scalar and vector types to and from packed formats, saving space at the cost of precision. However, packing a value into a format that it was previously unpacked from is guaranteed to be lossless.

<glm/gtc/packing.hpp> need to be included to use these features.

4.12. GLM_GTC_quaternion

Quaternions and operations upon thereof.

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

4.13. GLM_GTC_random

Probability distributions in up to four dimensions.

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

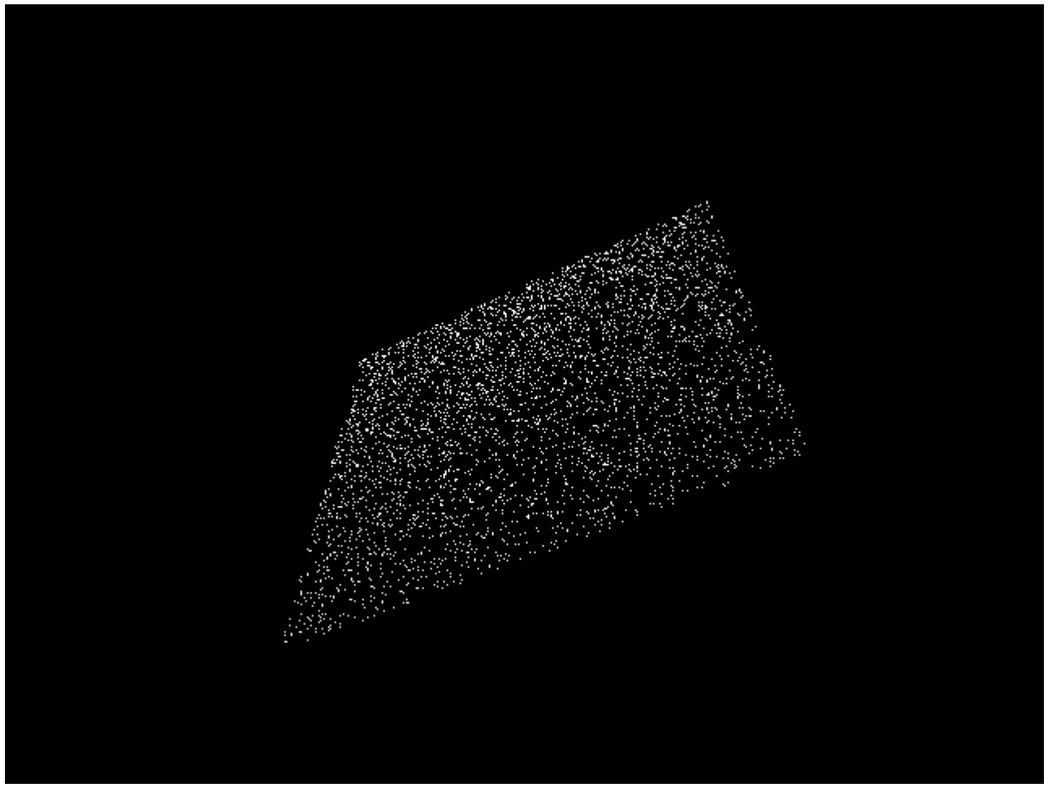


Figure 4.13.1: glm::vec4(glm::linearRand(glm::vec2(-1), glm::vec2(1)), 0, 1);

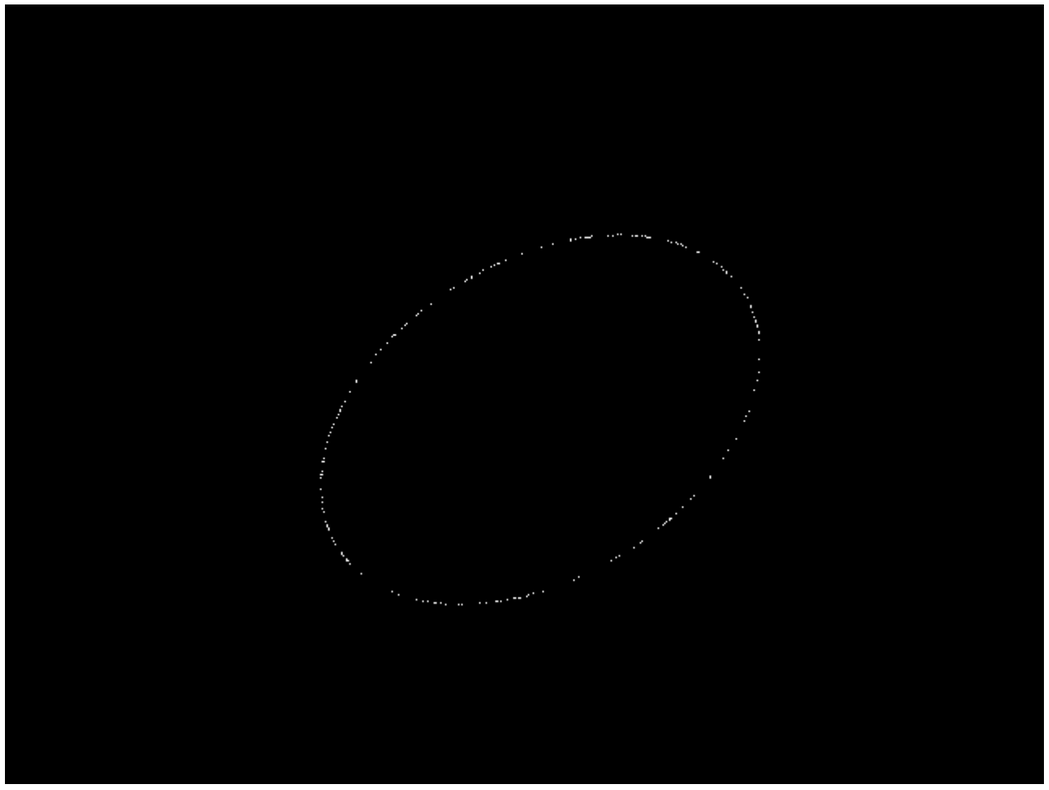


Figure 4.13.2: glm::vec4(glm::circularRand(1.0f), 0, 1);

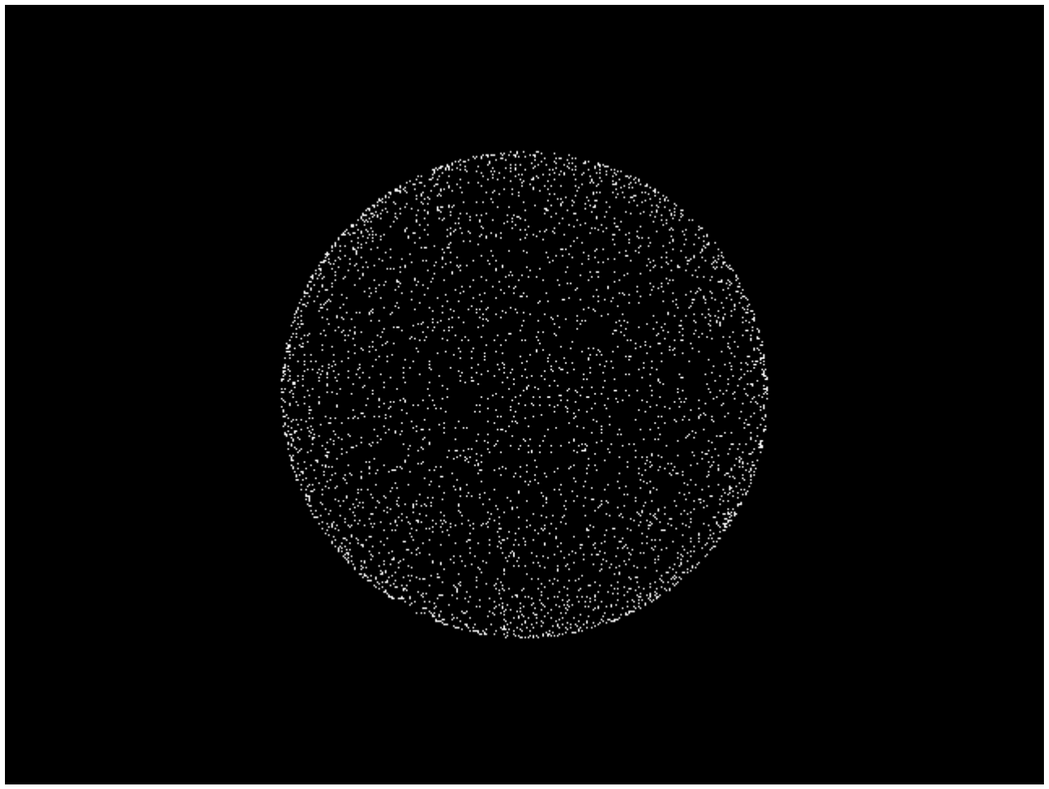


Figure 4.13.3: glm::vec4(glm::sphericalRand(1.0f), 1);

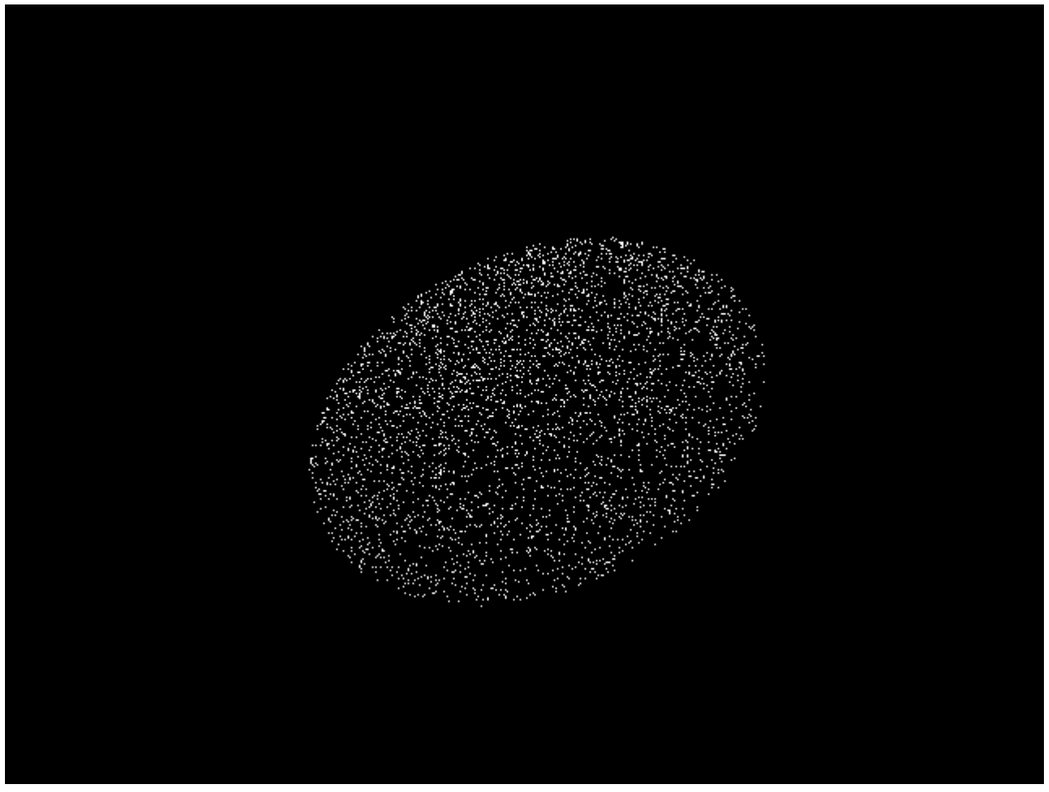


Figure 4.13.4: glm::vec4(glm::diskRand(1.0f), 0, 1);

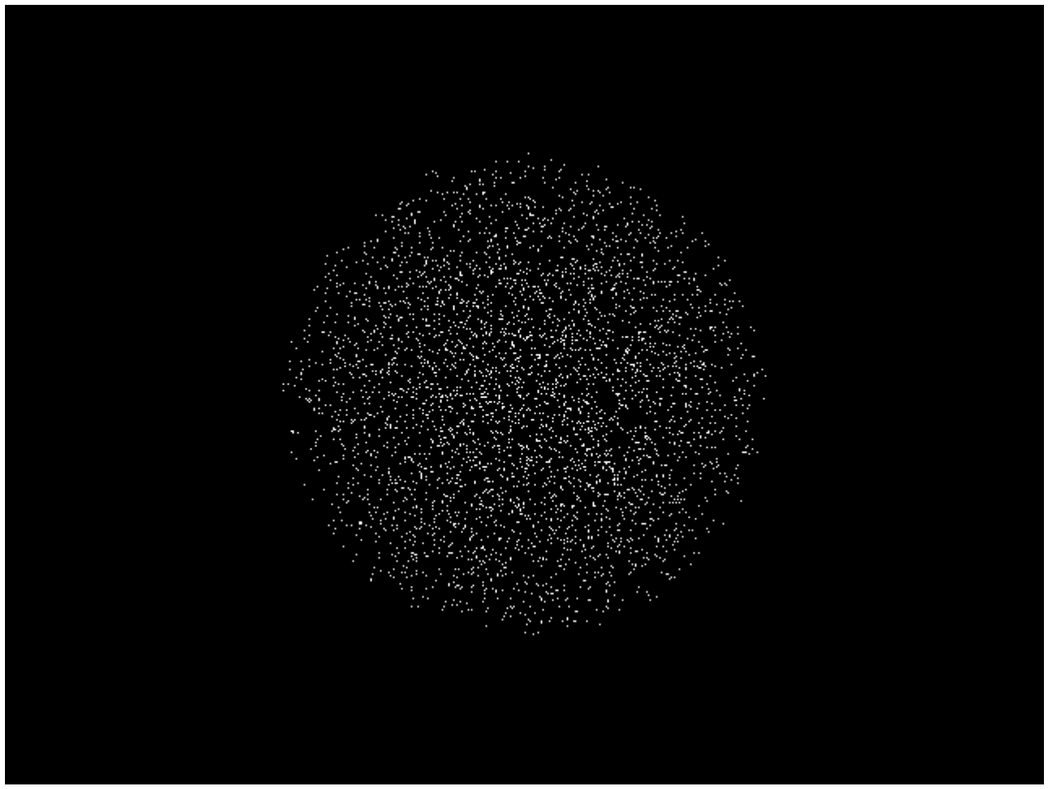


Figure 4.13.5: glm::vec4(glm::ballRand(1.0f), 1);

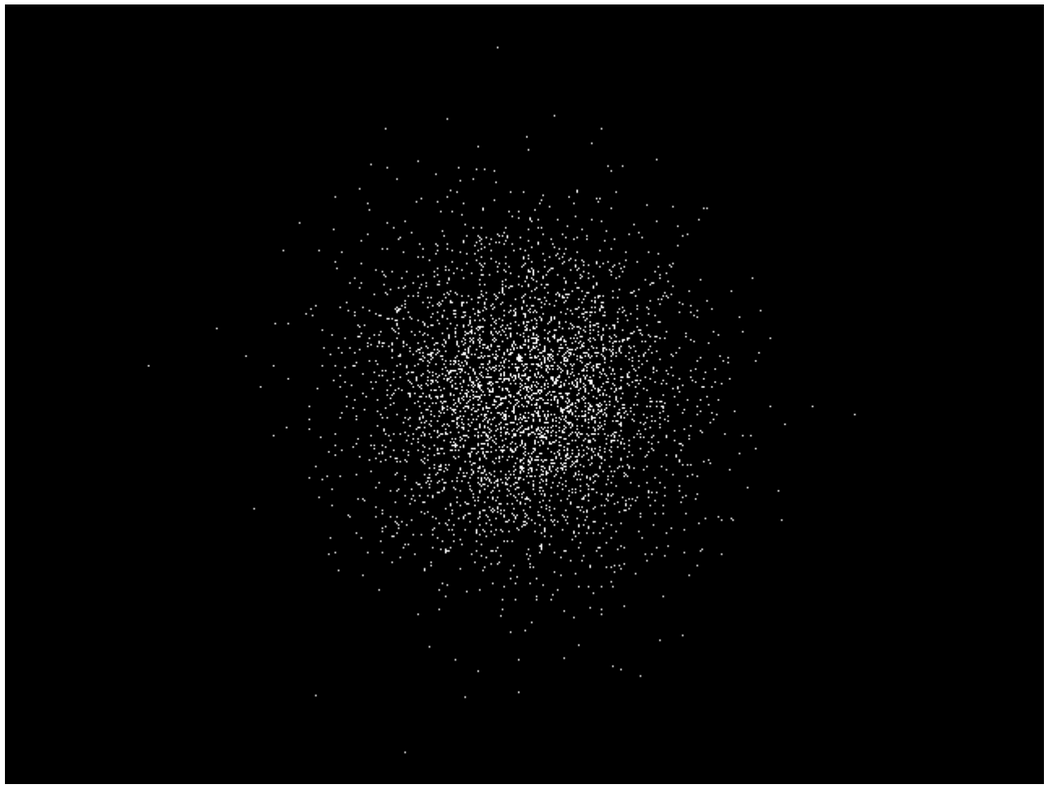


Figure 4.13.6: glm::vec4(glm::gaussRand(glm::vec3(0), glm::vec3(1)), 1);

4.14. GLM_GTC_reciprocal

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

<glm/gtc/reciprocal.hpp> need to be included to use the features of this extension.

4.15. GLM_GTC_round

Various rounding operations and common special cases thereof.

<glm/gtc/round.hpp> need to be included to use the features of this extension.

4.16. GLM_GTC_type_aligned

Aligned vector types.

<glm/gtc/type_aligned.hpp> need to be included to use the features of this extension.

4.17. GLM_GTC_type_precision

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 Default precision.

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

- GLM_PRECISION_LOWP_INT8: Low precision
- GLM_PRECISION_MEDIUMP_INT8: Medium precision
- 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)

<glm/gtc/type_precision.hpp> need to be included to use the features of this extension.

4.18. GLM_GTC_type_ptr

Facilitate interactions between pointers to basic types (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.

```
// GLM_GTC_type_ptr 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, this one inspired by the STL:
#include <glm/glm.hpp>

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

*Note: It would be possible to implement *glVertex3fv*(glm::vec3(0)) in C++ with the appropriate cast operator that would result as an implicit cast in this example. However cast operators may produce programs running with unexpected behaviours without build error or any form of notification. *

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

4.19. GLM_GTC_ulp

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

<glm/gtc/ulp.hpp> need to be included to use these features.

4.20. GLM_GTC_vec1

Add *vec1 types.

<glm/gtc/vec1.hpp> need to be included to use these features.

---

5. OpenGL interoperability

5.1. GLM replacements for deprecated OpenGL functions

OpenGL 3.1 specification has deprecated some features that have been removed from OpenGL 3.2 core profile specification. GLM provides some replacement functions.

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>

5.2. GLM replacements for GLU functions

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);
```

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

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::ivec4 const& viewport);
glm::dvec3 project(glm::dvec3 const& obj, glm::dmat4 const& model, glm::dmat4 const& proj, glm::ivec4 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::ivec4 const& viewport);
glm::dvec3 unProject(glm::dvec3 const& win, glm::dmat4 const& model, glm::dmat4 const& proj, glm::ivec4 const& viewport);
```

From GLM_GTC_matrix_transform extension: <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. 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 support

GLM supports GLSL precision qualifiers through prefixes instead of qualifiers. For example, GLM exposes \verb|lowp_vec4|, \verb|mediump_vec4| and \verb|highp_vec4| as variations of \verb|vec4|.

Similarly to GLSL, GLM precision qualifiers are used to trade precision of operations in term of ULPs for better performance. By default, all the types use high precision.

```
// Using precision qualifier in GLSL:

ivec3 foo(in vec4 v)
{
    highp vec4 a = v;
    mediump vec4 b = a;
    lowp ivec3 c = ivec3(b);
    return c;
}

// Using precision qualifier in GLM:

#include <glm/glm.hpp>

ivec3 foo(const vec4 & v)
{
    highp_vec4 a = v;
    medium_vec4 b = a;
    lowp_ivec3 c = glm::ivec3(b);
    return c;
}
```

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

7. FAQ

7.1 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.

7.2. Does GLM run GLSL program?

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

7.3. Does a GLSL compiler build GLM codes?

No, this is not what GLM attends to do.

7.4. 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.

Stating with GLM 0.9.9, to use experimental extensions, an application must define GLM_ENABLE_EXPERIMENTAL.

7.5. Where can I ask my questions?

A good place is stackoverflow using the GLM tag.

7.6. 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!

7.7. 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 a higher compatibility with third party library and SDKs.

7.8. Is GLM fast?

GLM is mainly designed to be convenient and that's why it is written against the GLSL specification.

Following the Pareto principle where 20% of the code consumes 80% of the execution time, GLM operates perfectly on the 80% of the code that consumes 20% of the performances. Furthermore, thanks to the lowp, mediump and highp qualifiers, GLM provides approximations which trade precision for performance. Finally, GLM can automatically produce SIMD optimized code for functions of its implementation.

However, on performance critical code paths, we should expect that dedicated algorithms should be written to reach peak performance.

7.9. 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 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.

7.10. Why some GLM functions can crash because of division by zero?

GLM functions crashing is the result of a domain error. Such behavior follows the precedent set by C and C++'s standard library. For example, it's a domain error to pass a null vector (all zeroes) to glm::normalize function, or to pass a negative number into std::sqrt.

7.11. What unit for angles is used in GLM?

GLSL is using radians but GLU is using degrees to express angles. This has caused GLM to use inconsistent units for angles. Starting with GLM 0.9.6, all GLM functions are using radians. For more information, follow the link.

7.12. Windows headers cause build errors...

Some Windows headers define min and max as macros which may cause compatibility with third party libraries such as GLM. It is highly recommended to define NOMINMAX before including Windows headers to workaround this issue. To workaround the incompatibility with these macros, GLM will systematically undef these macros if they are defined.

7.13. Constant expressions support

GLM has some C++ constant expressions support. However, GLM automatically detects the use of SIMD instruction sets through compiler arguments to populate its implementation with SIMD intrinsics. Unfortunately, GCC and Clang doesn't support SIMD instrinsics as constant expressions. To allow constant expressions on all vectors and matrices types, define GLM_FORCE_PURE before including GLM headers.

---

8. Code samples

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

8.1. 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));
}
```

8.2. 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));
}
```

8.3. 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)
};
```

8.4. Lighting

```
#include <glm/glm.hpp> // vec3 normalize reflect dot pow
#include <glm/gtc/random.hpp> // ballRand

// 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::ballRand(0.0f, Light.inaccuracy());

    if(!shadow(Intersection.globalPosition(), Light.position(), LightVertor))
    {
        float Diffuse = glm::dot(Intersection.normal(), LightVector);
        if(Diffuse &lt;= 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 &gt; 0.0f ? Dot : 0.0f;
            float Specular = glm::pow(Base, Material.exponent());
```

```
            Color += Material.specular() \* Specular;
        }
    }

    return Color;
}
```

---

9. References

9.1. OpenGL specifications

- OpenGL 4.3 core specification
- GLSL 4.30 specification {width="2.859722222222222in" height="1.6083333333333334in"}- *GLU 1.3 specification*

9.2. External links

- GLM on stackoverflow

9.3. Projects using GLM

Leo's Fortune

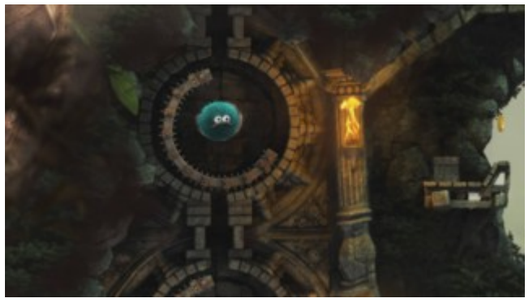
Leo's Fortune is a platform adventure game where you hunt down the cunning and mysterious thief that stole your gold. Available on PS4, Xbox One, PC, Mac, iOS and Android.

Beautifully hand-crafted levels bring the story of Leo 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



OpenGL 4.0 Shading Language Cookbook

A set of recipes that demonstrates a wide 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.

Outerra

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

Falcor

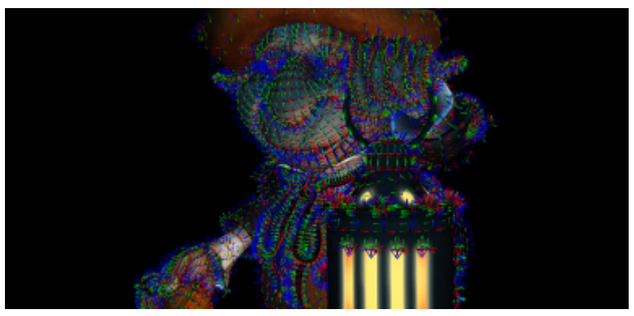
Real-time rendering research framework by NVIDIA.

Cinder

Cinder is a free and open source library for professional-quality creative coding in C++.

Cinder is a C++ library for programming with aesthetic intent - the sort of development often called creative coding. This includes domains like graphics, audio, video, and computational geometry. Cinder is cross-platform, with official support for OS X, Windows, iOS, and WinRT.

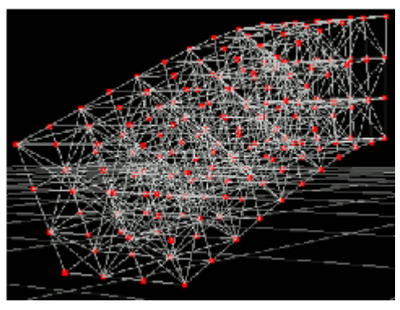
Cinder is production-proven, powerful enough to be the primary tool for professionals, but still suitable for learning and experimentation. Cinder is released under the 2-Clause BSD License.

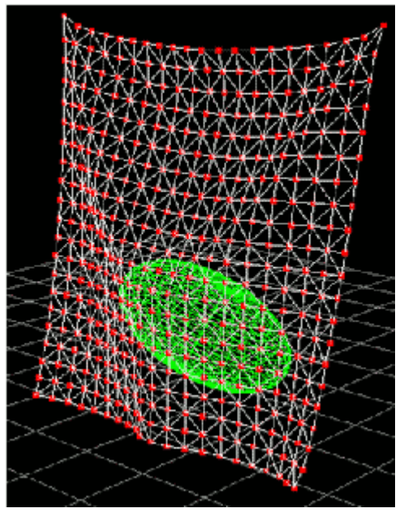


opencloth

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

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





LibreOffice

LibreOffice includes several applications that make it the most powerful Free and Open Source office suite on the market.

Are you using GLM in a project?

9.4. Tutorials using GLM

- Sascha Willems' Vulkan examples, Examples and demos for the new Vulkan API
- VKTS Vulkan examples using VulKan ToolS (VKTS)
- *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.
- Game Tutorials, graphics and game programming.
- open.gl, OpenGL tutorial
- c-jump, GLM tutorial
- Learn OpenGL, OpenGL tutorial
- ***Are you using GLM in a tutorial?***

9.5. Equivalent for other languages

- *cglm*: OpenGL Mathematics (glm) for C.
- *GlmSharp*: Open-source semi-generated GLM-flavored math library for .NET/C#.
- glm-js: JavaScript adaptation of the OpenGL Mathematics (GLM) C++ library interfaces
- JVM OpenGL Mathematics (GLM): written in Kotlin, Java compatible
- JGLM - Java OpenGL Mathematics Library
- SwiftGL Math Library GLM for Swift
- glm-go: Simple linear algebra library similar in spirit to GLM
- openll: Lua bindings for OpenGL, GLM, GLFW, OpenAL, SOIL and PhysicsFS
- glm-rs: GLSL mathematics for Rust programming language

- glmpython: GLM math library for Python

9.6. 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 developing an alternative library to GLM?

9.7. 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 operators implementation issues.
- Guillaume Chevallereau for providing and maintaining the *nightlight build system*.
- Ghenadii Ursachi for GLM_GTX_matrix_interpolation implementation.
- Mathieu Roumillac for providing some implementation ideas.
- *Grant James* for the implementation of all combination of none-squared matrix products.
- Jesse Talavera-Greenberg for his work on the manual amount other things.
- All the GLM users that have report bugs and hence help GLM to become a great library!