An Introduction To GLM

This document describes the most important GLM data types & related functionality. It's not really possible to cover everything here – please check the online API documentation for further information. In general, if you need a maths function that seems like it'd be useful for a generic computer graphics problem, GLM probably implements it somewhere.
Basic stuff like cos, sin, tan, pow, exp, sqrt, PI etc. is available, but not listed here.

A. Vector Datatype

The types we're going to use most often are floating-point vectors:

- `glm::vec2`
- `glm::vec3`
- `glm::vec4`

The types above represent two, three and four component vectors, respectively. The name *vector* is a bit misleading; A `glm::vec4` is simply a class with four floating point member variables. While these vectors are most often used to represent either points in space or actual vectors - as in 'direction in space' - they may also be used to represent non-geometric variables, such as RGBA values and UV coordinates.

The vector class has pretty much no member functions. All utility functions are realized either as free functions in the glm namespace, or – in case of generic functionality - with the help of operator overloading. The reason for this is the fact that, as mentioned earlier, the vector class can be used to represent lots of completely different things; It would not make sense to add, say, functions for dealing with cross or dot products to a class that might be used for representing colour values.

Also, there are signed and unsigned integer versions, which we're going to use occasionally: signed integer vectors are prefixed with an 'i', unsigned integer vectors are prefixed with a *'u'* - e.g. `glm::uvec4` represents a four component unsigned integer vector.

Important Operators:

- the + and – operators implement component wise addition and subtraction of two vectors. *Both vectors must have the same number of components, and the same underlying data type (float, integer or unsigned integer).*
- the * and / operators implement component wise multiplication and division of two vectors. *Both vectors must have the same number of components, and the same underlying data type (float, integer or unsigned integer).*
- the * and / operators also implement multiplication and division of each component by a scalar. *The scalar value must have the same underlying data type as the vector (float, integer or unsigned integer).*

Utility Functions:

All of these are free functions located in the GLM namespace. Some of these functions operate on two or more vectors – *in that case, they all should have the same underlying data type (float, integer or unsigned integer) and the same number of components.*

- inner, or dot product of two vectors: `glm::dot`
- cross product of two vectors: `glm::cross`
- find unit vector of given vector: `glm::normalize`
- find length of given vector: `glm::length`
- reflect vector off a surface (defined by surface normal): `glm::reflect`
- distance between two points (also works for scalars): `glm::distance`

Basic Usage:

```
glm::vec3 a( 0.f, 1.f, 0.f );          // a position in 3d space
glm::vec3 b( 1.f, 2.f, 3.f );          // another position
glm::vec3 ab = b-a;                    // vector from a to b
                                       // ( use of operator - )
glm::vec3 dir = glm::normalize( ab );  // unit direction vector from a to b
float dist = glm::length( ab );        // distance between a and b
                                       // also: float dist = glm::distance( a, b );
glm::vec3 mid = a + 0.5f * ab;         // position exactly in between a and b
                                       // ( use of operators + and * with scalar )
```

In the above sample, the variables a and b both represent positions, while ab represents a direction; For the sake of avoiding logical errors, you should use distinct naming for positions, directions, colours, texture coordinates etc.

Accessing Components:

```
glm::vec4 position;                    // a position
glm::vec4 colour( 0.f, 0.f, 1.f );     // an RGBA value
glm::vec2 texcoord( 0.5f, 0.5f );      // a UV coordinate

// now the fun starts
position.x = colour[ 0 ];
position.y = colour.g * texcoord.s;
position.z = texcoord.t + colour.b;
position[ 3 ] = color.a;
// of course, this also works...
color.a = color.x;
```

As you can see, the individual components of a GLM vector can be accessed in different ways - if your vector represents a position, you can use x/y/z/w to access the components, if it's a colour you may want to use r/g/b/a instead, or if represents texture coordinates, you could use s/t/p/q. Of course, nothing stops you from using x/y/z/w for colour values and r/g/b/a for positions - but doing this would probably make your code harder to read and more error prone. Alternatively, you may use the []-operator to access the desired component by index.

In General, GLM closely mirrors GLSL. If a GLSL function exists, GLM problably offers the same function. There are just some things that the GLSL language allows, that C++ cannot easily mirror, like the example below which won't work in C++:

```
glm::vec3 v = vertexPosition.zyx //note the .zyx to change the order of the components
```

B. Matrix Datatype

We need matrices to represent affine transformations, as well as perspective/orthographic transformations.

- `glm::mat2`
- `glm::mat3`
- `glm::mat4`

The above types represent 2x2, 3x3 and 4x4 floating point matrices, respectively. Non-square matrices are supported, but we'll probably never use them.
Just like the vector class, the matrix class is ultimately just a set of values, devoid of any interpretation. E.g. in the case of a `glm::mat4`, we basically have an underlying array of 16 floats. Because of this, the GLM matrix class again has no member functions. Any utility functions are realized either with the help of free functions in the GLM namespace, or with the help of operator overloading.

Important Operators:

- the + and - operators implement addition and subtraction of matrices, respectively.
- the * and / operators implement multiplication and division by a scalar.
- the * operator also implements multiplication with another matrix, or a vector.

Utility Functions:
(http://glm.g-truc.net/0.9.4/api/a00151.html)

- transpose a matrix: `glm::transpose`
- invert a matrix: `glm::invert`
- compute a scaling matrix / apply scaling to a matrix: `glm::scale`
- compute a rotation matrix / apply rotation to a matrix: `glm::rotate`
- compute a translation matrix / apply translation to a matrix: `glm::translate`
- construct a perspective matrix: `glm::perspective`
- construct a view matrix: `glm::lookAt`

Special Matrix Constructors:

- construct an identity matrix: `glm::mat4( 1.f );`
- construct a matrix filled with zeros: `glm::mat4( 0.f );`

Composite Transformation:

```
// shortcut for identity matrix
const glm::mat4 id = glm::mat4( 1.f );
// constructs translation matrix -> 5 units in x, y, z each
glm::mat4 t = glm::translate( id, glm::vec3( 5.f, 5.f, 5.f ) );
// constructs rotation matrix – 45 degrees around positive y axis
glm::mat4 r = glm::rotate( id, glm::radians( 45.f ), glm::vec3( 0.f, 1.f, 0.f ) );
// constructs scaling matrix – 2 units in x, y, z
glm::mat4 s = glm::scale( id, glm::vec3( 2.f, 2.f, 2.f ) );
// computes composite transformation matrix
glm::mat4 composite = translate * rotate * scale;
```

Composite Transformation, Alternative:

```
// shortcut for identity matrix
const glm::mat4 id = glm::mat4( 1.f );
// beware, this one is much harder to read :-)
glm::mat4 composite = glm::scale( glm::rotate( glm::translate( id, glm::vec3( 5.f, 5.f,
5.f) ), glm::radians( 45.f ), glm::vec3( 0.f, 1.f, 0.f ) ), glm::vec3( 2.f, 2.f, 2.f ) );
```

The above samples demonstrate how basic and composite transformation matrices can be constructed. The three functions `glm::scale` / `glm::translate` / `glm::rotate` all accept a 4x4 matrix as their very first argument – the transformation itself is actually applied to that matrix. When we pass in the the identity matrix, the result will thus be the basic transformation matrix. Then, we just multiply the three transformation matrices to obtain the composite transformation matrix.
However, instead of constructing three basic transformation matrices first, we can just apply each transformation on top of the previous one, as shown in the alternate version. In that case, the order of transformations needs to be reversed to obtain the same result!

Accessing Elements

```
std::ostream operator<<( std::ostream &out, glm::mat4 const& m )
{
	for ( unsigned int row=0; row<4; ++row )
	{
		for ( unsigned int col=0; col<4; ++col )
			out << m[ col ][ row ] << " ";
		out << std::endl;
	}
	return out;
}
```

The above function shows how to overload the << operator so that you can print out a `glm::mat4` - this function is, sadly, lacking in GLM, you have to write it yourself if you need it (same for vectors).
It also demonstrates how to access individual elements of a matrix. The function is written so that prints out the matrix in a way that is easy to read for a programmer, namely in row-major layout. In reality, GLM stores matrices in column-major format, just like OpenGL. What this means is: in GLM, matrices are indexed first by column, second by row – m[0] returns the first column of the matrix m!. The way you typically learn matrix algebra, the content is indexed first by row, second by column.... if you talk about element a13 of a matrix in a maths class, you typically mean the element in the first row, third column. In GLM, it's the other way round.

Construction from Column Vectors:

```
glm::mat4 translation(	glm::vec4( 1.f, 0.f, 0.f, 0.f ),
			glm::vec4( 0.f, 1.f, 0.f, 0.f ),
			glm::vec4( 0.f, 0.f, 1.f, 0.f ),
			glm::vec4( 5.f, 5.f, 5.f, 1.f ) );
```

The above sample shows how to construct a via vectors. The matrix represents a translation, but the code looks as if the matrix were transposed (it isn't).

C. Quaternion Datatype

(http://glm.g-truc.net/0.9.4/api/a00153.html)

There isn't much to say about quaternions, except that they're often useful for storing rotations, because they avoid the well known gimbal lock problem and use up less space than a matrix. GLM supports easy 'casting' from matrices to quaternions and vice-versa.

Euler Angles → Quaternion:

```
float roll = glm::radians( 60.f );
float pitch = glm::radians( 30.f );
float yaw = glm::radians( 45.f );
glm::fquat rotation( glm::vec3( pitch, yaw, roll ) );
```

Rotation Matrix → Quaternion:

```
glm::mat4 rotation;
glm::fquat q = glm::quat_cast( rotation );
```

Quaternion → Matrices

```
glm::fquat rotation;
glm::mat4 m4 = glm::mat4_cast( rotation );
glm::mat3 m3 = glm::mat3_cast( rotation );
```

Quaternion → glm::rotate

```
const glm::mat4 id( 1.f );
glm::fquat rotation;
glm::mat4 m = glm::rotate( id, glm::angle( rotation ), glm::axis( rotation );
```