ME-C3100 Computer Graphics, Fall 2013

Lehtinen / Hölttä, Peussa

Math library handout

1 Intro

The lectures and lecture slides teach you the math theory and formulae you need to carry out the assignments. This handout is a guide to carrying out those computations in practice, using the Math library in the Nvidia framework and C++ standard library. We also briefly discuss computation in the OpenGL shader language, GLSL.

2 Technical preface

The Math library header `framework/base/Math.hpp` contains a large number of mathematical functions as well as vector and matrix classes. (The vector classes represent mathematical vectors; do not confuse them with `std::vector` which is a dynamic array in the C++ standard library.) The class names in the library specify dimension and type of the vector or matrix: for instance, `Vec3i` is vector of 3 signed integers, and `Mat4f` is a 4x4 matrix of floating point numbers.

This handout does not attempt to exhaustively cover the Math library. To learn its full capabilities you can read `Math.hpp`. It's easy to browse it for names and types of functions you need, but figuring out how it works exactly is unnecessary and quite hard if you are not experienced with C++. In this section we discuss the implementation; you do not need to understand everything here.

In `Math.hpp`, the patterns `VectorBase<T, L, S>` and `MatrixBase<T, L, S>` repeat often. These base classes are C++ *templates* and contain functionality that is shared between all vector and matrix types. The contents of the angle brackets are *template arguments*. `T` represents the type of elements in the vector or matrix, `L` is the dimension, and `S` is the actual type of the implementation used. E.g. for `Mat4f`, `T` is `F32` (32-bit float type), `L` is 4, and `S` is `Mat4f` itself. This complicated implementation stems from various performance reasons. On a practical level, knowing what these arguments are can occasionally help you: for instance, there is a static utility function `translate` in `MatrixBase<T, L, S>` which produces a translation matrix. It takes a `VectorBase<T, L-1, V>` argument. What's going on? To translate something in `L-1` dimensions, you need a `L`-dimension translation matrix. Once you know what the template argument `L` stands for, it's clear how to use `translate`: for instance, `Mat4f::translate` needs a `Vec3f` argument.

The matrices are stored in column-major order, i.e. the elements of the first column are stored consecutively first in the memory, then the elements of the second column, etc. In the following matrix:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

the single scalar values would be stored in memory in the following order: 1, 3, 5, 2, 4, 6. Note that C++'s native multidimensional arrays are stored in row-major order, i.e. 1, 2, 3, 4, 5, 6. Because in 3D calculations we care about matrices' columns, column-major order is more convenient. The data layout used by the library is also directly compatible with the OpenGL API. In Assignment 0 code you see us giving OpenGL a pointer to the raw data (`mymatrix.getPtr()`) of a matrix to be used as a shader uniform (via a `glUniformMatrix4fv` function call).

Some of the implementations of functions declared in `Math.hpp` are defined in `Math.cpp`. This is merely a C++ implementation detail; reading `Math.cpp` doesn't help you to use the library.

OpenGL shading language (GLSL) code is totally distinct from the C++ code, even if it looks a bit similar. Shader code runs on the GPU and is a critical part of the modern graphics pipeline. In the assignments, any shader code is enclosed in a `GLContext::Program` object. You do not have to write GLSL code, except possibly in the final assignment, but it is useful for extra credit work. More about GLSL in section 6.

3 Usage purposes

Note: The matrices or vectors are not constrained to just positions and normals, of course - they contain simply numbers. `Vec3f` could be a position, normal, or even an RGB or HSV color. `Vec3i` contains integers instead of floats, and can be used to store e.g. index numbers for a triangle's corners. When several related values are needed, vectors are usually more convenient than separate values (e.g. `x`, `y`, `z`, or `i`, `j`, `k`, or `int i[3]`). The assignment starter codes usually use the vectors extensively where it is meaningful, and you will see lots of good examples while reading the given sources.

4 Basic math

In math notation, we usually have row or column vectors. The vectors used in the code do not specify dimensions, however - they are just ordered tuples of numbers, and some calculations assume a specific format. You can e.g. replace a column or a row in a matrix with the same vector, provided just that their sizes match. The matrices are 2-dimensional arrays, though, and their size and indexing does matter.

Basic matrix and vector math rules apply, and multiplication `A * B` works for sizes $n*m$ and $m*k$ respectively. It's also okay to multiply a vector by a matrix (`M * v`); you get out another vector. More about this later.

4.1 Vectors

There are two ways to access individual elements of a vector. Indexing `v[i]` and members `x`, `y`, `z` and `w` access the same values; often one of the forms is more logical than another. We suggest using the member form only for vectors that actually have meaning in xyz space.

If you create a vector with no arguments, it is initially filled with zeroes. Giving one argument fills the vector with that value. (Watch out: this means *if you accidentally assign a scalar into a vector, the vector is filled with that scalar and you do not get a warning!*) Naturally, you can also specify a separate value for each element.

```
Vec3f z; // z = {0, 0, 0}
Vec3f z2(1); // z2 = {1, 1, 1}
Vec3f z3 = 1; // same as z2 - but probably not intended...
Vec4f v(8, 6, 4, 2);
v[0] = v[3]; // now v = {2, 6, 4, 2}
v.y = 4; // v = {2, 4, 4, 2}
```

The integral typed Vec3i is useful for e.g. indices:

```
// x coordinate of the centroid of a triangle defined by points in a
// std::vector of 3D positions indexed by 'corners'
float middle_x(const std::vector<Vec3f>& points, Vec3i corners) {
    return (points[corners[0]].x + points[corners[1]].x + points[corners[2]].x) / 3.0f;
}
```

There are some convenient accessors for subsets of elements, and for transforming to and from homogenous coordinates:

```
// v from the earlier listing
Vec3f v2(v.getXYZ()); // v2 = {2, 4, 4}
Vec3f v3(v.toCartesian()); // v3 = {2, 4, 4}/2 = {1, 2, 2}
Vec4f v4(v3, 1.0f); // v4 is a homogenous v3, w set to 1
Vec4f v5(v3.toHomogenous()); // v5 == v4
```

Adding two vectors together works as expected, element by element. Other operators that would not normally make sense for two column vectors, like division or multiplication, are also interpreted as element-wise. Same goes for operations for a vector and a scalar.

```
Vec3f a(2); // a = {2, 2, 2}
Vec3f b(10, 20, 30), c;
c = a * b;  // c = {20, 40, 60}
c *= 1.5;   // c = {30, 60, 90}
```

Dot and cross products are available in the library both as separate functions and as member functions. As a special case, cross product of two-dimensional vectors is defined as the length of the resulting vector (scalar).

```
Vec2f a(100, 10), b(4, 2);
float c = dot(a, b); // c = 420; can also write c = a.dot(b)
Vec3f x(1, 0, 0), y(0, 1, 0), z;
z = cross(x, y); // {0, 0, 1}
float z_magnitude = cross(x.getXY(), y.getXY()); // 1
```

Dot product is also sometimes called the inner product, and the vectors are interpreted so that the result is a scalar. If you need to multiply two vectors to produce a matrix, use the `outerProduct` function.

Frequently you need to normalize a vector to unit length. The framework offers multiple ways to do that:

```
Vec3f v;
// ...
Vec3f v2 = v.normalized(); // this returns a new vector
v.normalize(); // this normalizes v itself in place
v /= v.length(); // manual way
v *= 1.0f / v.length(); // multiplication is faster than division
```

The division operation is usually quite slow compared to others: multiplying all the vector's elements by a constant 1.0f / 3.0f is faster than dividing them by 3. You usually need to worry about this only when

performing divisions several times *per pixel*, though! (Computing the length needs a square root, which is also quite slow.)

Also note that when you want to *compare* lengths, you can instead compare *squared* lengths, which is much faster as it avoids taking the square root. The `lenSqr` function for vectors gives you a squared length.

4.2 Matrices

The Math header provides specializations for the most common sizes. With little effort, you could write your own 3x100 matrix if you need one in some corner case, using the template system. See the header file for details (not good for novices, though!).

The contents of a matrix are initialized to identity.

```
Mat3f m; // m is an identity matrix
m = Mat3f(); // also with explicit initialization like this
```

Remember that identity matrix has ones in the diagonal and zeros elsewhere. For 3x3:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Single elements can be accessed individually in many ways. The `Mat*`'s member functions `getCol`, `getRow`, `setCol` and `setRow` get and set individual rows and columns. The data is transferred via vectors. Indexing starts from zero as usual.

```
Mat3f m;
m(1, 0) = 42; // row 1, column 0
m.m01 = 1337; // even simpler. row 0, col 1.
m.setCol(2, Vec3f(2, 4, 6)); // replace column 2
Vec3f v(m.getRow(0));
```

Now m is:

$$\begin{pmatrix} 1 & 1337 & 2 \\ 42 & 1 & 4 \\ 0 & 4 & 6 \end{pmatrix}$$

and v becomes `{1, 1337, 2}`.

The `Mat4f` version has a special `getXYZ` member function (confusingly, the same name as with `Vec4f`) for getting a copy of the upper-left 3x3 submatrix.

5 3D transformations

Matrices work with practically no surprises if you compare them to the "textbook matrices".

The given classes have some utility members for building commonly used transformation matrices: `rotation`, `translation`, `fitToView`, `perspective`. The interested can look at their code to get ideas to maybe write own helper functions.

```
Vec3f orig(1, 0, 0);
Mat3f rotation(Mat3f::rotation(Vec3f(0, 0, 1), FW_PI/2.)); // angle in radians
Mat4f translation(Mat4f::translate(Vec3f(-1, 0, 0)));
// Note the order: rotate first, then translate
Vec3f new_p = translation * (rotation * orig);
// new_p rotates first by the Z axis to {0, 1, 0},
// and then translates to {-1, 1, 0}
```

Here you should be a bit concerned; `orig` has three dimensions, so multiplying by the 3 by 3 matrix `rotation` yields another vector in three dimensions, but this is then multiplied by a 4 by 4 matrix – surely this is not properly defined. The trick here is that the framework defines a matrix multiplication with a vector with one less dimension, substituting a one to the missing last element of the vector. Why this often makes sense will be clear after the first few lectures; for transforms with homogeneous coordinates, this is most often what you want – but not always, and you'll run into such cases. The safest course of action is to always cast vectors explicitly.

Also note that the framework only defines the form of matrix-vector multiplication shown in the snippet above; the vector is multiplied with a matrix from the left. To multiply from the right with a matrix, use this formula:

$$v^T R = ((v^T R)^T)^T = (R^T v)^T$$

Because the result from $R^T v$ is already a vector, the last transpose can be ignored. Thus, just transpose the matrix and multiply from left.

To obtain an inverted or transposed matrix without modifying the original, you can use the `inverted` or `transposed` functions:

```
Mat4f inv = mat.inverted(); // can also chain: .inverted().transposed()...}
```

To instead modify the same matrix in place, use the `invert` and `transpose` functions like so:

```
mat.invert();
```

You might have noticed already that this is consistent with the `normalize` vs. `normalized` and friends.

6 GLSL: the OpenGL shading language

In short, GLSL is a high-level (compared to ARB assembly) language for writing programs that run not on the CPU but on your *graphics processing unit*, the GPU. The shaders determine the color of the pixels on the screen.

The Wikipedia page on GLSL gives a decent overview. There is also an online textbook by Jason McKesson that teaches basic graphics concepts step by step while making use of shaders and raw OpenGL calls. To quickly find specific GLSL functions, use the online reference or downloadable reference cards or the wiki at OpenGL.org.

You can do a lot in GLSL by just making local modifications inside a single shader, but learning more about the pipeline and being able to pass values in and out of shaders adds to what you can accomplish. GLSL has rich functionality for linear algebra computations, with a slightly different syntax compared to our C++ code.

A small table of differences with our math library:

C++	GLSL
`Mat4f`	`mat4`
`Vec3f`	`vec3`
`Vec3i`	`ivec3`
`float`	`float`
`mat.inverted()`	`inverse(mat)`
`mat.transposed()`	`transpose(mat)`
`m * v`	`m * v`

Some useful shorthand notations:

```
vec4 v4 = vec4(1.0, 2.0, 3.0, 4.0);
vec3 v3 = v4.xyz; // picking components by "swizzling"
v3.yz = vec2(2.0, 3.0); // swizzling also works for assignment
vec4 v4b = vec4(v3, 1.0);
```

The shader code is compiled to machine code just as the C++ code is compiled for your CPU. The big difference is that the shader code is compiled by the graphics driver when the source code is fed to it (usually, once when the program is started). Some graphics drivers are less picky about the shader syntax than others, so be sure to test any modifications in the school's computers! For example; `float f = 1;` might not work - you need to use floating-point numbers explicitly: `float f = 1.0;` There are also some restrictions built into GLSL that you'd just have to know: assignment to a variable declared `uniform` is forbidden, for example.

6.1 Basic OpenGL pipeline

We first load data (vertex positions, vertex normals, textures, etc.) to the GPU, set a specific shader program that is supposed to act on it, and then issue any number of draw calls. Upon receiving a draw call, the GPU processes the data.

For each vertex, the GPU runs the *vertex shader*, which computes the position of the vertex but might also compute the surface color at that vertex or other material-specific information. Three subsequent vertices are taken to be a triangle.

For all pixels in the triangle that end up being visible (inside *clip space*), the GPU runs the *pixel shader* (or more correctly *fragment shader* because these still have the depth information). In addition to data given to it directly, the fragment shader can use data computed by the vertex shader. Frequently this data is interpolated smoothly between the vertices, so that neighboring fragments have subtly different inputs. The fragments produced by the shader are written in a buffer, which will normally then be displayed on the screen, but could also be used as input data for another round of computation.

There are other, advanced shader types in OpenGL, but these two are mandatory.

For more in-depth information, see OpenGL wiki or another intro.