# Programming Assignment #1

CS 200, Fall 2024

*Due Friday, September 13*

## Some words about GLM

### Vectors and points

We will be using the GLM API, which is the defacto standard math library for use with programming in OpenGL. However we will only be using the *core features* portion of the library, which does not include functions for creating the basic transformation matrices that we will be using in this course. The GLM extensions library does provide such functionality, but you are not allowed to use this!

We will use the `glm::vec4` class to store points and vectors. This class allows access to vector components through fields labeled $x$, $y$, $z$, and $w$. For example,

```
glm::vec4 v(1,2,3,4);
cout << v.x << ' ' << v.y << ' ' << v.z << ' ' << v.w << endl;
```

will print `1 2 3 4` to the standard output. For the most part, we will be doing 2D graphics, so we will typically set the $z$–component to 0, and only use the $x$, $y$, and $w$ components. Recall that the $w$–component tells use if a quantity is a point or a vector: a point has $w = 1$, and a vector has $w = 0$. So if we want to store the point $P = (1, 2)$ and the vector $\vec{v} = \langle 3, 4\rangle$, we would write

```
glm::vec4 P(1,2,0,1);
glm::vec4 v(3,4,0,0);
```

### Matrices

For storing transformations, we will use the `glm::mat4` class. This class allows access to matrix components through double subscripting. However we need to be careful, GLM stores matrices in *column major* order, instead of the usual row major order that you are probably familiar with. The only time when this becomes an issue is when we access the elements of a matrix directly: you will need to remember to reverse the row/column index order. For instance, consider the matrix

$$M = \begin{pmatrix} M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\ M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

In standard 0–based indexing notation, we have $M_{1,2} = 7$ (row index 1 and column index 2). With GLM, we would write $M[2][1] = 7$. The entirety of matrix $M$ can be specified by

```
glm::mat4 M = {{1,5,9,13},{2,6,10,14},{3,7,11,15},{4,8,12,16}};
```

As indicated above, we will primarily be using 2D affine transformations. We will encode this by extending a 2D to a 3D affine transformation. That is, to store the 2D affine transformation with homogeneous coordinate representation

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 0 & 0 & 1 \end{bmatrix}$$

We would write

```
glm::mat4 A(0);
A[0][0] = 1;  A[1][0] = 2;  A[2][0] = 0;  A[3][0] = 3;
A[0][1] = 4;  A[1][1] = 5;  A[2][1] = 0;  A[3][1] = 6;
A[0][2] = 0;  A[1][2] = 0;  A[2][2] = 1;  A[3][2] = 0;
A[0][3] = 0;  A[1][3] = 0;  A[2][3] = 0;  A[3][3] = 1;
```

We can also write

```
glm::mat4 A = {{1,4,0,0},{2,5,0,0},{0,0,1,0},{3,6,0,1}};
```

## Others

The GLM library is extensive. However, we will only have the need for a handful of functions in the core library. Documentation for the library can be found at

`https://glm.g-truc.net/0.9.9/index.html`

The page

`https://en.wikibooks.org/wiki/GLSL_Programming/Vector_and_Matrix_Operations`

is also useful. However, note this page is explicitly for GLSL (OpenGL shading language) instead of GLM. As GLM is meant to mimic GLSL, essentially everything on this page, except for swizzling, applies verbatim to GLM.

Prior to doing the assignment, you should spend a some time looking at the library documentation. In particular, familiarize yourself with the functions

`dot   length   normalize   inverse   radians   degrees`

as well the constructors for the `vec2`, `vec3`, `vec4`, `mat2`, `mat3`, and `mat4` classes. In particular, note that the constructors can be used to convert between different dimensions of the same type. E.g.,

```
glm::mat4 A(5);
glm::mat2 L = glm::mat2(A);
glm::mat4 B = glm::mat4(L);
```

results in

$$A = \begin{pmatrix} 5&0&0&0 \\ 0&5&0&0 \\ 0&0&5&0 \\ 0&0&0&5 \end{pmatrix}, \quad L = \begin{pmatrix} 5&0 \\ 0&5 \end{pmatrix}, \quad B = \begin{pmatrix} 5&0&0&0 \\ 0&5&0&0 \\ 0&0&1&0 \\ 0&0&0&1 \end{pmatrix}.$$

## Your task

I will provide you with a header file named `Affine.h`, that contains the following declarations and definitions.

```
namespace cs200 {
  inline bool near(float x, float y) { return std::abs(x-y)<1e-4f; }

  bool isPoint(const glm::vec4 &P);
  bool isVector(const glm::vec4 &v);
  bool isAffine(const glm::mat4 &A);

  glm::vec4 point(float x, float y);
  glm::vec4 vector(float x, float y);

  glm::mat4 affine(const glm::vec4 &u, const glm::vec4 &v,
                   const glm::vec4 &C);
  glm::mat4 rotate(float t);
  glm::mat4 translate(const glm::vec4 &v);
  glm::mat4 scale(float r);
  glm::mat4 scale(float rx, float ry);
}
```

Where the header file `glm/glm.hpp` header file has been included. You are to implement these functions. Here is a description of a what the functions should do.

`near(x,y)` — convenience function to compare two floating point numbers: returns `true` if $x$ and $y$ are close enough to be considered equal. [Implemented.]

*Remark on floating point numbers.* When using floating point numbers, you should keep in mind that computations are only accurate to a finite degree of precision. For example, suppose that after a computation we theoretically expect the value of a variable to be 2 exactly. However due to numerical precision issues, the value might actually be 1.9999. This is close enough for our purposes, but not exactly equal to the expected value. So we typically do not compare two floating point values using the equality operator `==`, but rather test for the two values to be sufficiently close.

`isPoint(P)` — returns `true` if $P$ represents a 2D point, and returns `false` otherwise.

`isVector(v)` — returns `true` if $v$ represents a 2D vector, and returns `false` otherwise.

`isAffine(A)` — returns `true` if $A$ represents a 2D affine transformation.

`point(x,y)` — returns the 2D point with components $(x, y)$.

`vector(x,y)` — returns the 2D vector with components $\langle x, y\rangle$.

`affine(u,v,C)` — returns the 2D affine transformation with *columns* given by $\vec{u}$, $\vec{v}$, and $C$ (two 2D vectors and one 2D point). I.e., the transformation $A$ such that $A(\mathbf{e}_x) = \vec{u}$, $A(\mathbf{e}_y) = \vec{v}$, and $A(\mathcal{O}) = C$ is returned.

`rotate(t)` — returns the 2D affine transformation $R_t$ for rotation by the angle $t$ with respect to the origin. The angle $t$ is assumed to be given in *degrees*.

`translate(v)` — returns the 2D affine transformation $T_{\vec{v}}$ for translation by the 2D vector $\vec{v}$.

`scale(r)` — returns the 2D affine transformation $H_r$ for uniform scaling by $r$ with respect to the origin.

`scale(rx,ry)` — returns the 2D affine transformation $H_{\langle r_x,r_y\rangle}$ for nonuniform scaling by factors $r_x$ and $r_y$ with respect to the origin.

## What to turn in

You are to implement the functions in the above header file (except for the one already implemented). Your implementation file should be named `Affine.cpp`. Only the `Affine.h` header file may be included (note that `Affine.h` already includes `cmath` and `glm/glm.hpp`). You may not alter the contents of this header file in any way.

## Remark on testing

For this assignment, and all assignments in general, the test drivers that I give you do not test all of the functions declared in the assignment header file. And for the functions that it does test, the tests are very simple: they only test the function with one or two calls. For the functions that the drivers do not test, you will need to design your own tests for these functions. Indeed, you should come up with ways of testing all of the functions in the assignment header file in a more robust manner than in the supplied test drivers.