Programming Assignment #1

CS 250, Summer 2020

Due Tuesday, May 12

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. 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 $P = (1, 2, 3)$ and the vector $\vec{v} = \langle 4, 5, 6\rangle$, we would write

```
glm::vec4 P(1,2,3,1);
glm::vec4 v(4,5,6,0);
```

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}};
```

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 few minutes looking at the library documentation. In particular, familiarize yourself with the functions

`cross   dot   length   normalize   inverse   radians   degrees`

as well the constructors for the `vec3`, `vec4`, `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::mat3 L = glm::mat3(A);
glm::mat4 B = glm::mat4(L);
```

results in

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

Your task

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

```
namespace cs250 {
  inline bool near(float x, float y) { return std::abs(x-y) < 1e-4f; }

  glm::vec4 point(float x, float y, float z);
  glm::vec4 vector(float x, float y, float z);
  bool isPoint(const glm::vec4 &v);
  bool isVector(const glm::vec4 &v);

  float angle(const glm::vec4 &u, const glm::vec4 &v);
  glm::vec4 cross(const glm::vec4 &u, const glm::vec4 &v);

  glm::mat4 affine(const glm::vec4 &Lx, const glm::vec4 &Ly,
                   const glm::vec4 &Lz, const glm::vec4 &C);
  bool isAffine(const glm::mat4 &M);

  glm::mat4 rotate(float deg, const glm::vec4 &v);
  glm::mat4 scale(float r);
  glm::mat4 scale(float rx, float ry, float rz);
  glm::mat4 translate(const glm::vec4 &v);
  glm::mat4 affineInverse(const glm::mat4 &A);
}
```

Where the header files `glm/glm.hpp` and `cmath` have been included. You are to implement these functions. Here is a description of a what the functions 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.]

`point(x,y,z)` — returns the 3D point with components (x, y, z).

`vector(x,y,z)` — returns the 3D vector with components $\langle x, y, z\rangle$.

`isPoint(P)` — returns `true` if P represents a 3D point, and returns `false` otherwise.

`isVector(v)` — returns `true` if v represents a 3D vector, and returns `false` otherwise.

`angle(u,v)` — returns the angle, measured in degrees, between the 3D vectors $\vec{u}$ and $\vec{v}$. The function assumes that the passed–in parameters u and v are actually 3D vectors (w–components are 0). No error checking need be done.

`cross(u,v)` — returns the cross product $\vec{u} \times \vec{v}$ of $\vec{u}$ and $\vec{v}$. There is already a `cross` function in the GLM library. However, the two arguments of `glm::cross` are both of type `vec3`, and the function return type is also `vec3`. In contrast, the function here takes and returns objects of type `vec4`. You can either implement this function from scratch, or use `glm::cross` with appropriate type conversions.

`affine(u,v,n,C)` — returns the 3D affine transformation with *columns* (in standard matrix representation) given by $\vec{u}$, $\vec{v}$, $\vec{n}$, and C (three 3D vectors and one 3D point). I.e., the transformation A such that $A(\mathbf{e}_x) = \vec{u}$, $A(\mathbf{e}_y) = \vec{v}$, $A(\mathbf{e}_z) = \vec{n}$, and $A(\mathcal{O}) = C$ is returned.

`isAffine(A)` — returns `true` if A represents a 3D affine transformation.

`rotate(t,v)` — returns the 3D affine transformation $R_{t\vec{v}}$ for counterclockwise rotation by the angle t about the vector $\vec{v}$. The angle t is assumed to be given in *degrees*.

`translate(v)` — returns the 3D affine transformation $T_{\vec{v}}$ for translation by the 3D vector $\vec{v}$.

`scale(r)` — returns the 3D affine transformation H_r for uniform scaling by r with respect to the origin.

`scale(rx,ry,rz)` — returns the 3D affine transformation $H_{\langle r_x,r_y,r_z\rangle}$ for nonuniform scaling by factors r_x, r_y, r_z with respect to the origin.

`affineInverse(A)` — returns the inverse of the 3D affine transformation A. It assumed the passed–in parameter A is actually an affine transformation. Again, no error check need be performed. GLM has matrix inverse functions. However, inverting a general 4×4 matrix is computationally more expensive than computing the inverse of a 3D affine transformation. You are expected to implement the method discussed in class for inverting a 3D affine transformation using the inverse of a 3×3 matrix. You may use the `glm::inverse` function, but only for 3×3 matrices.

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.