UM EECS 487
Lab 3: 3D Modeling and Modeling Transformations

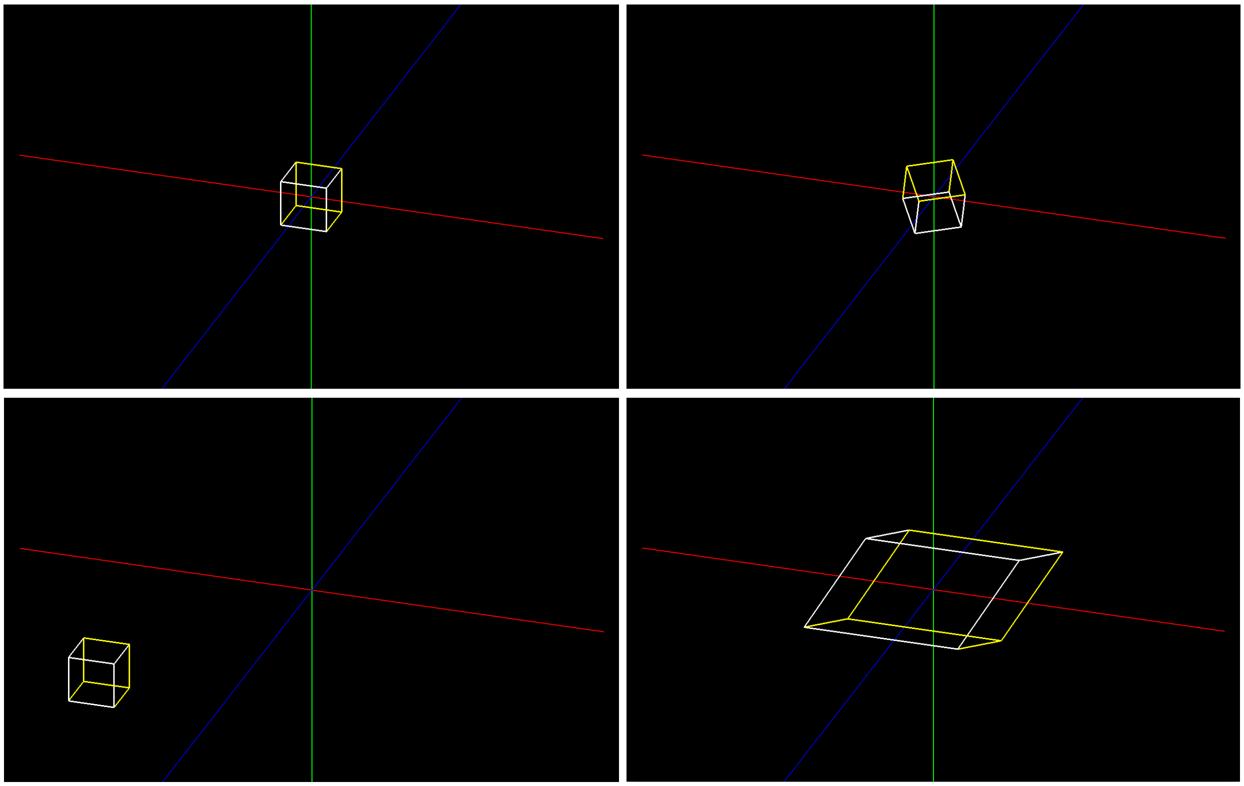

FIGURE 1. Screenshot showing [top-left] the original cube, [top-right] after various rotations, [bottom-left] x-,y-,z-translations, and [bottom-right] several scaling followed by shear transformations.

This lab exercises your knowledge of the basic modeling and modeling transformations. You can enter one of the four modeling transformation modes of the supplied program by pressing the '`M`', '`R`', '`S`', or '`T`' keys for magnification/scaling, rotation, shear, or translation respectively.

MODELING TRANSFORMATIONS

In all of the modeling transformation modes, the program displays a wireframe cube centered at the origin, as depicted in the top left figure of Fig. 1. Table 1 summarizes the transformations you must implement, along with the default values by which the transformations are applied. The header of the table lists the key bound to each transformation, e.g., pressing '`h`' in rotation mode rotates the cube $9°$ counter-clock wise (ccw) about the $-z$-axis per rotation.

Pressing the '`I`' key resets the cube's position and orientation. The cube consists of eight 3D vertices stored in the global array `cube[8][4]`, defined in `objects.cpp`; each vertex is described by 3D homogeneous coordinates. The cube is initialized in the `Cube()` constructor. The display-callback, `display()` reads the present value of these vertices and renders the cube. Each time a key is pressed the keyboard-callback `kbd()` calls functions to transform these vertices according to the mode and key selected. To help you make sense of the transformed cube, the edges

Mode/Key	`l/X`	`h/x`	`k/Y`	`j/y`	`w/z`	`s/Z`
Magnification (`M`)	scale x up 50%	scale x down 50%	scale y up 50%	scale y down 50%	scale z down 50%	scale z up 50%
Shear (`S`)	shear x by 10% y	shear x by -10% y	shear y by 10% z	shear y by -10% z	shear z by -10% y	shear z by 10% y
Translation (`T`)	+10 along x	-10 along x	+10 along y	-10 along y	-10 along z	+10 along z

TABLE 1. Transformation invocations along with their default values.

Mode/Key	`l/Z`	`h/z`	`k/X`	`j/x`	`w/Y`	`s/y`
Rotation (`R`)	9° cw about $-z$	9° ccw about $-z$	9° cw about x	9° ccw about x	9° cw about y	9° ccw about y

TABLE 2. Rotation invocations along with their default values.

forming the back and bottom sides of the cube in its original orientation are drawn in yellow, the rest in white.

Search for "YOUR CODE HERE" in the `.cpp` files. Each transformation consists of appropriately initializing the global "current tranformation matrix" (`CTM`), which is of type `MatModView`, before calling `transformWorld()`. For example, when the program is in rotation mode and the user hits '`h`', your `rotate()` function should first call the `rotateZ()` method of the `MatModView` class to initialize the `CTM` then you can call `transformWorld()`, which in turn should call your `Cube::transform()`. The `rotate()` function is called by the `kbd()` handler. You are to write the code to initialize the tranformation matrix for each type of transformation. You are also to write the code for `Cube::transform()`. Do *not* call `glTranslate*()` or `glRotate*()` or any of the other transformation functions that come with OpenGL. Basically, aside from the already provided code, your code should not call OpenGL functions *at all*. It may seems like a lot of functions to write, but most are four-liners that are slight variations of a theme. Don't over-think them!

Play with the rotation mode of the program for awhile. If you rotate a cube that has been moved away from the origin, it doesn't rotate about its own axis. Modify your `transformWorld()` and `setupWorld()` such that a moved cube rotates about its own axis.

A CONE

In this part of the assignment, you should replace the cube above with a cone. Both the `Cube` and `Cone` classes are defined in `objects.h`. The `Cube()` constructor and `Cube::draw()` method are given to you in `objects.cpp` and you have written the `Cube::transform()` method in the above. For the cone, you must define the `Cone()` constructor and you **must** use a vertex array and `GL_TRIANGLE_FAN` in `Cone::draw()` to pass the cone's vertices to openGL. Use `glDrawElements()` to index the appropriate vertices in the vertex array. See the comments in `Cone::draw()` for further instructions. You are then to write `Cone::transform()` also (hint: it is very similar to `Cube::transform()`).

Hitting the '1' and '2' keys switches between displaying the cube and the cone.

VECTOR AND MATRIX HELPER CLASSES

The file `xvec.h` defines the `XVec4f` class along with several useful methods for vector operations such as `dot`, to compute the dot product, `cross`, to compute the cross product, and `normalize`, to normalize the vector. Accessing elements of `XVec4f` uses the "`()`" operator instead of the more common "`[]`" operator, e.g., `v(3)` refers to the last element of the 4-vector. You can both store to and read from vector element accessed in this way. This is the same vector support code used in PA1.

The file `xmat.h` defines the `XMat4f` class along with several other useful methods for matrix operations. The '*' operator is defined such that given matrix `M` of type `XMat4f` and vector `v` of type `XVec4f`, `M*v` returns a 4-vector containing the result of the computation, as expected. You may also find useful methods that implement other matrix operations such as `Identity()`, `setCol()`, `setRow()`, etc. As with `XVec4f`, element access of `XMat4f` uses "()" instead of the more common "[]", e.g., `M(3,3)` gives the bottom right-most element of the 4x4 matrix `M`. Unlike C/C++, matrix access is row major order: `M(row, col)`. You can both store to and read from element accessed in this way.

You should look through both of these files to see all the functionalities they provide. It can save you a lot of time not having to re-implement basic vector and matrix manipulation code!