UM EECS 487
Lab 4: Viewing and Perspective Transformations

FIGURE 1. The degrees of freedom in specifying camera tracking motion: *dolly*, *crab*, and *boom* and orientation/pivot motion: *pan* (or *yaw*), *tilt* (or *pitch*), and *roll*.

In this lab, we simulate the movements of a camera around a group of objects. As with real cameras, our imaginary camera has the following six degrees of freedom, as depicted in Fig. 1:

1.) **Dolly**: *translation* forward or backward.
2.) **Crab**: *translation* left or right.
3.) **Boom**: *translation* up or down.
4.) **Pan**: *rotation* left or right, gaze/heading change.
5.) **Tilt**: *rotation* up or down, gaze change.
6.) **Roll**: central *rotation*, no change in gaze direction.

Given the standard OpenGL camera setup, that is, looking down the $-w$-axis, with $+v$ up, and $+u$ to the right, the above six degrees of freedom are, in order: translation along w-, u-, and v-axis, and rotation around v-, u-, and w-axis. Each of these motions can be simulated by manipulating the eye-position **e**, gaze-direction **g**, and the top-vector **t** (for instance, roll: rotate the top vector around the gaze vector). The directions of motion here pertain to *the camera*, which means when the camera moves one way, the displayed objects move the other way. All translational movements are to be made in `DELTA` increments and all rotational movements are to be made in `THETA` increments. Both `DELTA` and `THETA` are defined in `transforms.cpp`. Every time you change any of the eye position or vectors, you need to call `Camera::orient()` to "re-orient" your camera by re-building its coordinate frame. Table 1 summarizes the camera tracking and pivot motions and the respective keyboard and mouse shortcuts. It may seem intensive to implement so

Camera Motion (C)	Direction (shortcut(s))	Direction (shortcut(s))
Dolly	forward (w/z)	backward (s/Z)
Crab	left (h/x)	right (l/X)
Boom	up (k/Y)	down (j/y)
Pan	left (mouse left click-hold and drag left)	right (mouse left click-hold and drag right)
Tilt	up (mouse left click-hold and drag up)	down (mouse left click-hold and drag down)
Roll	ccw (a)	cw (d)

TABLE 1. Camera tracking and pivot motions.

many movements, but each is quite simple, most are one-liners. Do not over-think them! Search for "YOUR CODE HERE" in `transforms.cpp`. You should leave the implementation of the "dolly" camera movement to the next section because the dolly effect is more visible after you have perspective projection, which you will implement in the next section.

Each time you render a 3D scene to screen, you first transform the scene from the world coordinate frame to the camera's coordinate frame (viewing transform), then you project the scene onto the 2D viewing plane (projection transform). You are to implement viewing transform in `setup_view()` (see online comments in `transforms.cpp` for further instructions on how to do this). In this section, we use orthographic projection, implemented by calling `glOrtho()` in the `reshape()` handler in `viewing.cpp`. In the same `reshape()` function, the call to `glViewPort()` transforms the canonical view volume (cvv) to the view port.

The group of objects displayed consists of an array of cubes and a pyramid (both defined in `objects.cpp`). Pressing the '`I`' key resets the display.

Play with the viewing transformations a bit and contrast it to the modeling transformations of the previous lab. Contemplate how in modeling transformations, you move the object by applying transformations to it; whereas in viewing tranformations, the group of objects stays fixed, it's the camera and the camera's coordinate frame that you move about.

PERSPECTIVE TRANSFORMATION

To implement perspective projection, go to `objects.cpp:transformWorld()` and put in your code to perspective transform the `cube_array` and `pyramid` objects. Search for "YOUR CODE HERE" in the source files for further instructions. Pressing the '`P`' key switches your projection to perspective mode. Pressing the '`O`' key switches projection to orthographic mode.

The 2D viewing plane is the near-plane of the view volume; it is also your Open GL window. the near plane is placed at `zNear0` distance away from the initial camera/eye position. You only need to implement the perspective projection that "squeezes" the view frustum into a rectangular view volume. The part that transforms the rectangular box into the unit canonical view volume (cvv) and the transformation of the cvv into the view port are already done for you when the `reshape()` handler calls `glOrtho()` and `glViewPort()` respectively, in `viewing.cpp`. The far plane of the view frustum is not specified and is assumed to be at ∞. You want to work out the results of applying the projection matrix to a point and what the z-coordinate would be instead of applying the perspective matrix blindly (and multiply by ∞!).

Once you have perspective projection working, implement the *dolly* motion for the camera in `transforms.cpp:dolly()` and invoke it from `transforms.cpp:movecam()`.

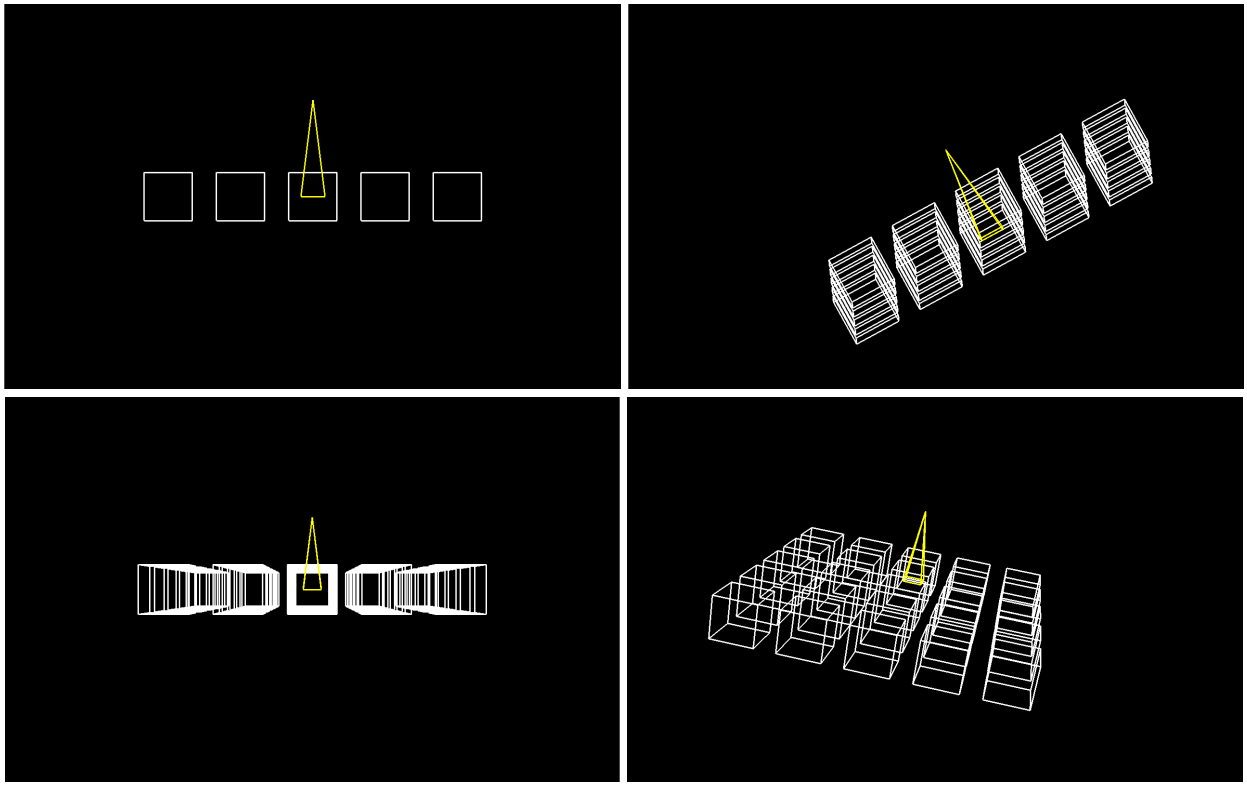

FIGURE 2. Screenshot showing [top] the cube array and pyramid with orthographic projection and [bottom] perspective projection. The two figures on the left are views down the $-z$-axis. The two figures on the right are from camera off the $-z$-axis.

So far you have computed the viewing and projection transforms in software. To use the hardware, follow the instructions `transforms.cpp:setup_view()` to setup viewing transforms with OpenGL. Then follow the instructions in `viewing.cpp:reshape()` to set up OpenGL for perspective projection. In both cases, search for "`HARDWARE`" and you should see "YOUR CODE HERE". Once you have both functions set up, pressing the '`H`' key tells the program to use hardware transforms with OpenGL (hit '`O`' or '`P`' to switch back to using your software transforms). Since the parameters we use to set up the orthographic and perspective projections under OpenGL do not exactly match, the resulting images will not be the same. As long as the objects are perspective projected when you press '`H`', you're fine.

VECTOR AND MATRIX HELPER CLASSES

The same `XVec*` and `XMat*` helper classes used in the previous lab are again available for your use in this lab.