Zena's CS251 Project 3

This goal of this project was to create a 3D space to view data in, although we only made the axes for now. We implemented controls to zoom, pan, and rotate the view, and we did this using matrices that represented the vpn, vrp, u, vup, and extent, as well as matrices to represent the screen size and offset from the edges of the window. We had these values inside the view class, and manipulated them with methods for normalization and rotation. In the other class, display, we built upon project 1 but added in a way to draw the axis lines as well as methods that user controls triggered, which accessed and manipulated the matrices in formulas that we used to, for example, recalculate the extent in the zooming method. In my project. I also implemented a reset button, included axis labels, and had the zoom level be shown to the user.

For the first task, we built a matrix in view that stores the axes' starting and ending points (plus the homogeneous coordinate), meaning there were six columns in total; the x axis, for example, was (0,0,0,1) and (1,0,0,1). We also had objects representing the canvas objects for the axes (initialized as None) that were stored in a list so that we could later initialize and manipulate them. Then, we made the buildAxes formula, which draws the axes onto the canvas, and calls the build method of the view class, which sets up the matrices in a way that provides us with a kind of starting camera view. Lastly, we made an updateAxes method, which would put changes in place after transformations by building a vtm, multiplying that by each axis's start-point and endpoint, then looping through each line in the axes list and changing their location on the canvas. To get the correct line from the list, I started indexing variables at 0 and 2 and added 2 to each at the end of every time through the loop.

For the second task, we had to build all of the user control capabilities.
The first one we made was the handleButton1 method, which let us press the left mouse/trackpad button and lead into the handleButton1Motion if we moved. For this motion method, we followed a provided algorithm. For this, we first calculated the difference in former and current position, allowing us to get the distance moved. With this, we could get the delta x and delta y, which were scaled to the screen by being divided by the screen height then being multiplied by the current extent. Then, we calculated the new vrp, updated the axes, and replaced the self.baseclick location.
The second function we made was to have the button 3 of the mouse implement scaling. Moving up makes the items come closer, and moving down makes them appear farther away. As before, we calculated the distance moved, but this time, we calculated a scale factor, as well. This would be multiplied by the base extent to determine the new extent. Therefore, it had to start at 1 to ensure the extent remained the same by default, and since it should range from .1 to 3 times the original view size, I subtracted no more than the amount moved divided by the screen size, and added no more than 3 times the amount moved divided by the screen size.
The third subtask concerned making the pipeline for rotation around the view volume in the view class. We made a series of matrices - one to translate, one to align, one to rotate about y, one to rotate about x, one to unalign, and one to represent the starting, untranslated view reference coordinates. We multiply them all in the end, and then reassign the results to vrp, u, vup, and vpn, and normalize them all at the end.
The last part of task 2 was to give the user the ability to utilize this method. They would press button 2 on the mouse, and moving the mouse around would give the mousebutton2 motion-handling method the amount moved, which it would use to calculate the angle (math.pi*((difference)/200)). Then we use the original view matrix and transform that using the angles as input to the view class's rotation method.

My first extension was to implement a reset button. I simply linked the spacebar to a method that calls the reset method of view, which resets all the matrices to their default values, and then I updated the axes to reflect the changes.

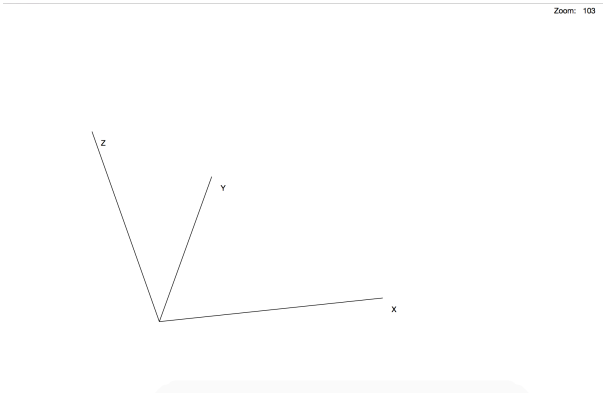
My second extension was to add axis labels, so we'd know which was x, which was y, and which was z. For this, I decided to define the letters in relation to the lines. In the updateAxes method, I added the logic shown below:

```
for line in self.axisList:
    # update the coordinates of the object
    #print self.axisMatrix
    self.canvas.coords(line, (points[0,startCol], points[1,startCol],
                              points[0,endCol], points[1,endCol]))
    letterX = self.canvas.coords(line)[2] + 20
    letterY = self.canvas.coords(line)[3] + 20
    self.canvas.coords(self.letterList[letterIdx],letterX, letterY)

    letterIdx += 1
    startCol += 2
    endCol += 2
```

As you can see, I defined the x and y location of the letter as the second and third indices of the coordinates of lines. Each line started at the origin, and so its first two coordinates reflected that, while its last two were where its end currently was. Therefore, the letter would be moved there, with a 20 pixel offset to look cleaner. (I also had variables to store the anchors for the line endpoints, too, but in retrospect, I don't need them because I don't use them anywhere except in one method.)

The third extension I did was showing the user how zoomed-in they are. They can see a number in the top right that starts at 100 and gets bigger when closer and smaller when farther from the axes. For this, I had a field for the zoom level int, and one for the object to draw. Upon reset, the int would go back to 100, and the text object would be reset to 100. In the method setZoomInfo, I set the default zoom and drew "Zoom: " and set the zoom level text object to a drawn canvas object. Unlike the other items on the canvas, which were simply moved, this would stay in the same spot and just keep getting erased and replaced when the zoom level changed in the relevant method. The challenge I had for this extension was finding a way to invert the direction, because I was basing the zoom level upon the extent, but that got smaller as the objects on the canvas came closer. I ended up getting the reciprocal of the extent and multiplying it by 100, then rounding that to an int to determine the current zoom.



The picture above shows my program with axis labels and zoom level displayed.

In this project, I learned more about binding controls to methods in tkinter and how to draw letters and lines in tkinter, and I became more familiar with using numpy matrices. It also helped my understanding of the view matrices and the math behind them to visualize and manipulate them through the view class.

Thank you to: Melody, Stephanie