Assignment 2 Report

Problem:

For this assignment, we need to utilize openGL in C++ to recreate given images and create an image of our own. The main obstacle is figuring out how to effectively utilize x y z coordinates to mimic the images.

Algorithm and Methods:

All the problems use the glPushMatrix which pushes a matrix on a stack. The matrix is initially an identity matrix, then it can be further modified using glRotatef and glTranslated.

glRotatef modifies the matrix by adding cos and sin in the designated positions that cause the rotation on x, y, or z axis. glRotatef takes in an angle, along with x,y,z coordinates. If glRotatef(theta, 1,0,0) is called the result matrix would look like (if matrix modified is the identity matrix) :

$$\mathbf{R}_{x,\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

glTranslate modifies the matrix by adding a given value to the matrix to shift the object to another position. glTranslate takes in an x,y, and a z and adds those values to the matrix at the top of the stack.

glPopMatrix removes the top matrix from the stack, bringing the matrix below to the top. This is used to change between matrices, usually to an entirely different modification on a new object.

Nested glPushMatrix can be used (not using glPopMatrix). In this case, the matrix clones the previous matrix and further modifications can be done to it.

Implementation:

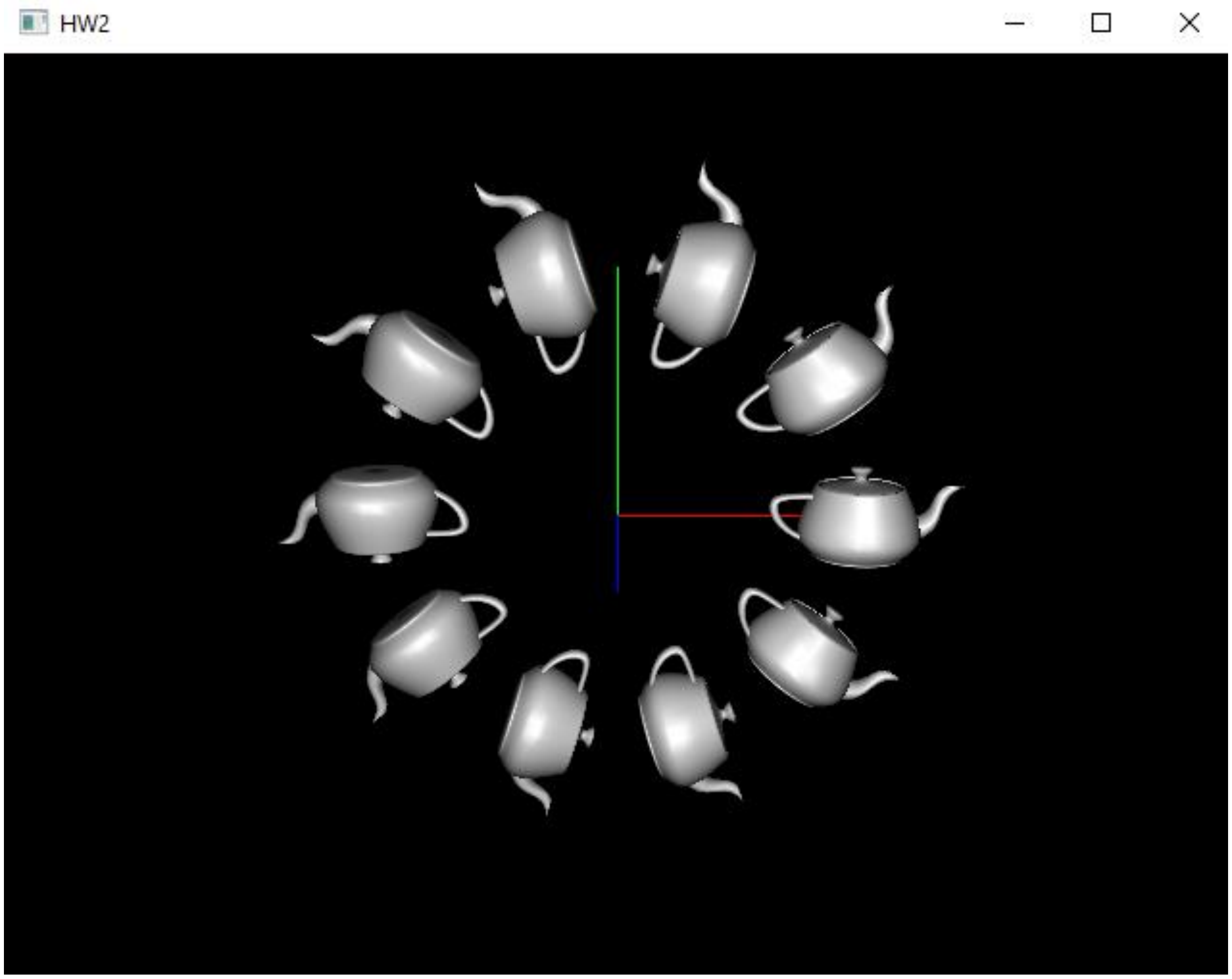
To solve problem 1, I noticed that there was ten equally spaced teapot in a circle formation. With that, I decided to create an algorithm that uses cos and sin for translated coordinates for x and y. For each increment for the angle, I used a=36 (360/10) and did a while loop until the incremented angle reached 360. Because the cos and sin functions in c++ take in radians, I added a quick function that converts degrees to radians called d2r(double a). And with each increment the function pushes a new matrix in to the stack, then translates with glTranslated(cos(d2r(a)), sin(d2r(a)), 0) and rotated with glRotatef(a, 0, 0, 1). Finally, the function uses glutSolidTeapot(.25) to produce to object, and then the angle is incremented.

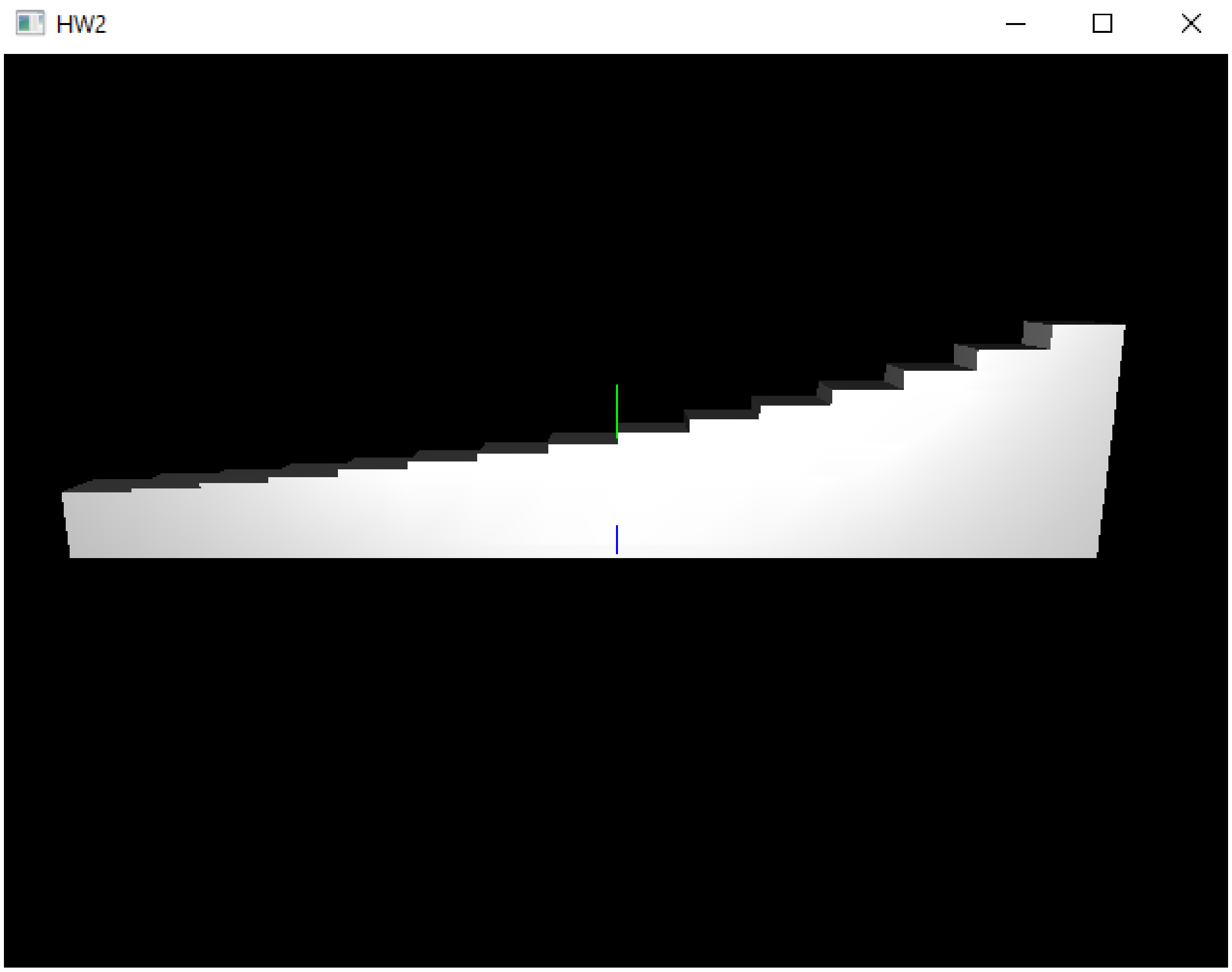
For problem 2, I noticed that the space between each step grows exponentially subtly. Because the image consists entirely of cubes, the algorithm needs to produce more cubes as the steps get higher. For this algorithm, I created 2 nested loops: one to determine the x of the step, one to increment the elevation of the step (y). The first loop increments the x linearly by .5 each time and the n by 1 (the number of steps). Also, Inside the first loop defines if y-increment value and the y value. The second loop pushes a matrix, translates with the x and y, produces the cube, and pops the matrix. Finally, the loop increments the y by multiplying the yinc by 1.13 the adds the yinc to the y. These loops continue until 15 steps have been created.

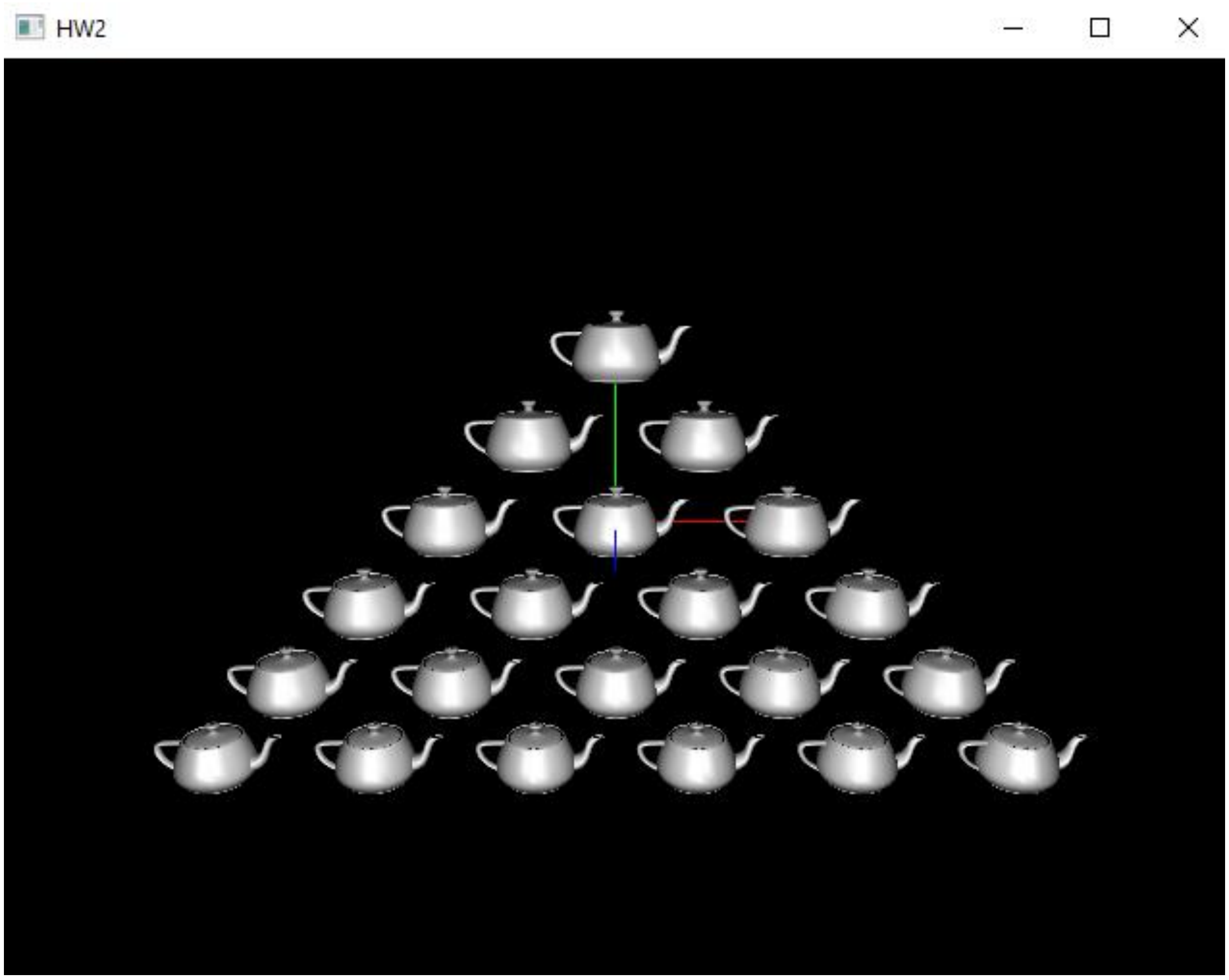
For problem 3, I created an additional function createRowTea(n,y) that creates a row of teapots. The function takes in an n which determines the number of teapots and a y which determines the y. The function first determines if the number of teapots is odd or even. If odd, the first teapot is the at x=0, if even, the first teapot starts at x=.5. Then the loops produce a teapot at the x and the -x and continues until the number of teapots is achieved. The original problem3 function then calls this function six times with 1-6 n and a y that matches the image.

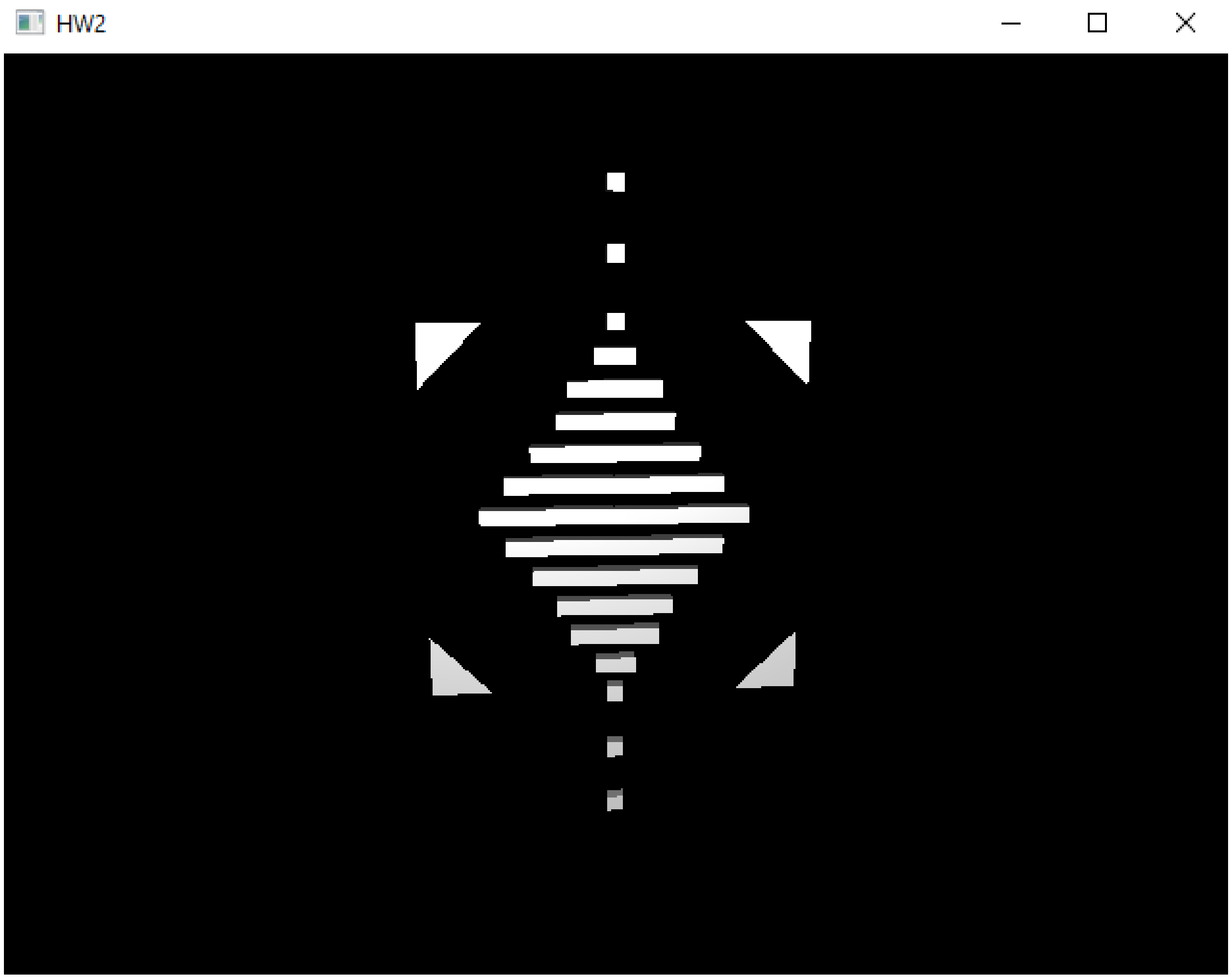
For problem 4, I utilized parts of the function I used for problem 3 but instead of teapots it's cubes. I messed around with function, experimenting with different values and created a cool image. I then added a new function that creates triangles called makeTri(). This function uses a glOrtho function that multiplies the matrix by an orthographic matrix which produces a parallel projection. Then, we use openGL immediate mode to produce triangles. For each triangle, I use glVertex2f to specify each vertex coordinate. The original function then calls both functions to create the image.

Results:


HW2


HW2


HW2



All the resulting images match the given images.