# Student No: K00282339
# Student Name: Zach Ahearn

## CA1 Implementation Report

**Introduction**

The purpose of this CA is to create a 3D wireframe scene based on the theme 'Start with Nothing' using wireframe models and the processing language. I had to make 4 different shapes for my scene with one being static, and the other 3 each displaying a different form of transformation. For my scene, I chose a simple player model for my static shape. My other shapes include a translating gun, a rotating sign, and a scaling cube enemy.

```
54  //GUN 1
55  float[] p33 = {-15.0,60.0,0.0,1.0};//Top left
56  float[] p34 = {0.0,60.0,0.0,1.0};//Top right
57  float[] p35 = {0.0,65.0,0.0,1.0};//Barrel-bottom right
58  float[] p36 = {-10.0,65.0,0.0,1.0};//trigger
59  float[] p37 = {-10.0,75.0,0.0,1.0};//handle-bottom right
60  float[] p38 = {-15.0,75.0,0.0,1.0};//handle-bottom left
61
62  //GUN 2
63  float[] p39 = {-15.0,60.0,10.0,1.0};//Top left
64  float[] p40 = {0.0,60.0,10.0,1.0};//Top right
65  float[] p41 = {0.0,65.0,10.0,1.0};//Barrel-bottom right
66  float[] p42 = {-10.0,65.0,10.0,1.0};//trigger
67  float[] p43 = {-10.0,75.0,10.0,1.0};//handle-bottom right
68  float[] p44 = {-15.0,75.0,10.0,1.0};//handle-bottom left
69
```

*Figure 1 Snippet of plotted points code*

To start, I plotted out every point for the 2D versions of my shapes using float variables. Once the 2D versions are complete, making the 3D points is as simple as copy pasting all the points of a shape and changing their Z values. I separated the two sets of points with differing z values with comments as seen in figure 1 on lines 54 and 62. This process ended out with a total of 78 different points.

```
360 void drawBox () {
361   //STATIC PLAYER DRAW 1================================
362   strokeWeight(1);
363   stroke(255, 174,201);
364   //HEAD
365   line(p1[0],p1[1],p1[2], p2[0],p2[1],p2[2]);
366   line(p2[0],p2[1],p2[2], p3[0],p3[1],p3[2]);
367   line(p3[0],p3[1],p3[2], p4[0],p4[1],p4[2]);
368   line(p4[0],p4[1],p4[2], p1[0],p1[1],p1[2]);
369
370   //LEFT ARM
371   stroke(0, 162,232);
372   line(p4[0],p4[1],p4[2], p5[0],p5[1],p5[2]);//p5 to p6
373   line(p5[0],p5[1],p5[2], p6[0],p6[1],p6[2]); //p6 to p7
374   line(p6[0],p6[1],p6[2], p7[0],p7[1],p7[2]);//p7 to p8
375
376   //WAIST & FEET
377   stroke(63, 72,204);
378   line(p7[0],p7[1],p7[2], p8[0],p8[1],p8[2]);
379   line(p8[0],p8[1],p8[2], p9[0],p9[1],p9[2]);
380   line(p9[0],p9[1],p9[2], p10[0],p10[1],p10[2]);
381   line(p10[0],p10[1],p10[2], p11[0],p11[1],p11[2]);
382   line(p11[0],p11[1],p11[2], p12[0],p12[1],p12[2]);
383
384   //RIGHT ARM
385   stroke(0, 162,232);
386   line(p12[0],p12[1],p12[2], p13[0],p13[1],p13[2]);
387   line(p13[0],p13[1],p13[2], p14[0],p14[1],p14[2]);
388   line(p14[0],p14[1],p14[2], p15[0],p15[1],p15[2]);
389   line(p15[0],p15[1],p15[2], p3[0],p3[1],p3[2]);
390
391   //EXTRA
```

*Figure 2 Joining points.*

# Student No: K00282339
# Student Name: Zach Ahearn

The next step was to join all these points which is done in my drawBox method. I also separated the line joinings with comments which made it easier to add color to specific parts of the shapes afterwards.

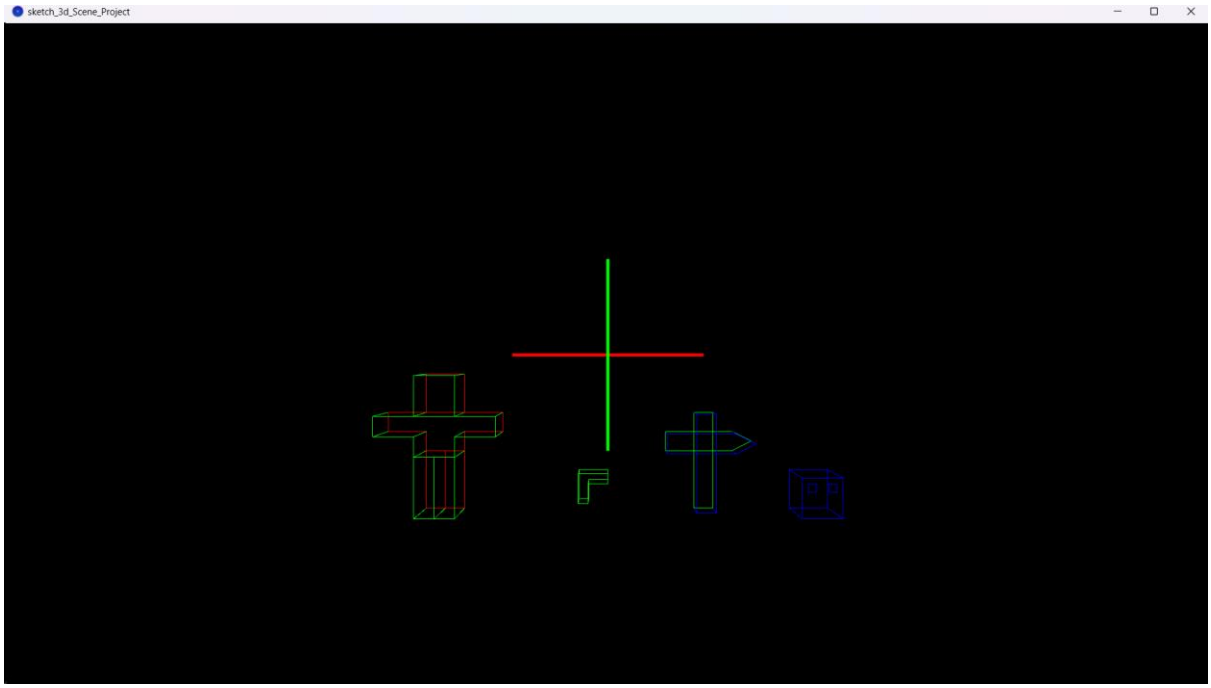



*Figure 3 Points plotted and joined only*

After all the points are plotted and appropriately joined to one another, figure 3 shows that result.

```
129 //TRANSLATING GUN----------------------------------------------------------------------------------------------------------------------------------------
130 int translateY = -2;
131
132 float[][] translateMatrix = {{1.0,0.0,0.0,0.0},{0.0,1.0,0.0,0.0},{0.0,0.0,1.0,0.0},{0,translateY,0,1.0}}; // translate by distance of -10
133 //----------------------------------------------------------------------------------------------------------------------------------------------------------
```

With a base now having been drawn, all that was left was to apply my transforms where necessary. The first transform I did was translating for my gun shape. In my translate matrix on line 132, the first 3 sets of numbers are left mostly 0, with x, y, and z in each set being set to 1.0 as to not change them in any way. The fourth set of numbers also has zeroes except for the y value. The Y value is instead equal to a variable I declared just above that on line 130 called ‘int translateY’. This variable is equal to -2 so shape will be translated by -2 on only its Y value every time it loops. The reason I have the Y value as a variable is because I will be changing this value.

# Student No: K00282339
# Student Name: Zach Ahearn

```
176  void draw()
177  {
178    // by commenting the background() call out,
179    //background(102);
180      //TRANSLATING LOOP----------------------
181      if(p38[1] < 40)
182      {
183        translateY = 2;
184      }
185      else if(p38[1] > 80)
186      {
187        translateY = -2;
188      }
189
190      translateMatrix[3][1] = translateY;
191    //----------------------------------------
```

*Figure 4 Translate Loop*

In my draw function, I have an if statement to check the value Y value of the lowest Y-valued point on my gun shape. If that Y value because less than 40, the translate variable is set to 2. If its greater than 80, its set to -2. Then on line 190, the translate value is set to equal the Y position for the translation matrix. This effectively loops the gun translation and makes then gun go up and down.

```
135  //ROTATING SIGN
136  float[][] translateToOrigin = {{1.0,0.0,0.0,0.0},{0.0,1.0,0.0,0.0},{0.0,0.0,1.0,0.0},{-50,-55,-5,1.0}}; // translate by certain distance
137  float[][] translateBackFromOrigin = {{1.0,0.0,0.0,0.0},{0.0,1.0,0.0,0.0},{0.0,0.0,1.0,0.0},{50,55,5,1.0}}; // translate by certain distance
138   float theta = radians(10); // convert degrees to radians
139
140
141  float[][] rotYMatrix = {{cos(theta),0.0,-sin(theta),0.0,0.0},{0.0,1.0,0.0,0.0,0.0},{sin(theta),0.0,cos(theta),0.0},{0.0,0.0,0.0,0.0,1.0}}; // roatate around y axis
142  /*
143  float[][] rotXMatrix = {{1.0,0.0,0.0,0.0},{0.0,cos(theta),sin(theta),0.0},{0.0,-sin(theta),cos(theta),0.0},{0.0,0.0,0.0,0.0,1.0}}; // roatate around x axis
144  float[][] rotZmatrix= {{cos(theta),sin(theta),0.0,0.0},{-sin(theta),cos(theta),0.0,0.0},{0.0,0.0,1.0,0.0},{0.0,0.0,0.0,0.0,1.0}}; // roatate around z axis
145  */
```

The next shape I moved onto was my sign shape. This is my shape that rotates. The first matrix on line 136 makes the shape translate back to point 0,0 on the axis. This is due to the fourth set of numbers in the matrix. The reason the numbers are -50,-55,-5 is because I had to get the centre coordinate for the x, y, and z values. Taking away the centre point values from x, y, and z translates the shape to 0,0. The second matrix on line 137 is done to bring the shape back to its original position. This is much the same as the first matrix except the previous negative values are set to positive. Next, a float called theta on line 138 is made which is equal to 10 radians. This is used in the rotYMatrix on line 141. Theta is incorporated into the matrix for the x and the z value. The values representing Y are left at 0 because when the shape is rotating around the Y axis, these values remain as 0.

```
208
209      m = scalingMatrix;
210      n = translateMatrix;
211
212      z = translateToOrigin;
213      applyTransform();
214
215      z = rotYMatrix;
216      applyTransform();
217
218      z =translateBackFromOrigin;
219  //     applyTransform();
220
221  //-------------------------
222      applyTransform();
223    //-----------------------
```

# Student No: K00282339
# Student Name: Zach Ahearn

For my code to apply transforms, I have float variables for each transform; m is for scaling, n is for translating, and z is for rotating. These variables are made equal to the appropriate matrices. Specifically for the rotation matrix, z is made equal the translateToOrigin matrix to move the shape's position to 0,0. Then z is made equal rotYMatrix, which rotates the shape. Finally, z is made equal translateBackFromOrigin which moves the shape back to its original position.

```
242 float[] transform_pointScale(float[][] m, float[] p)
243 {
244
245    float[] p_new = {0.0,0.0,0.0,0.0}; // handle 4 elements as its 3D calc
246
247    p_new[0] = m[0][0] * p[0] + m[1][0] * p[1] + m[2][0] * p[2] + m[3][0]*p[3];
248    p_new[1] = m[0][1] * p[0] + m[1][1] * p[1] + m[2][1] * p[2] + m[3][1]*p[3];
249    p_new[2] = m[0][2] * p[0] + m[1][2] * p[1] + m[2][2] * p[2] + m[3][2]*p[3];
250    p_new[3] = m[0][3] * p[0] + m[1][3] * p[1] + m[2][3] * p[2] + m[3][3]*p[3];
251
252
253    return p_new;
254 }
255
256 float[] transform_pointTranslate(float[][] n, float[] p)
257 {
258
259    float[] p_new = {0.0,0.0,0.0,0.0}; // handle 4 elements as its 3D calc
260
261    p_new[0] = n[0][0] * p[0] + n[1][0] * p[1] + n[2][0] * p[2] + n[3][0]*p[3];
262    p_new[1] = n[0][1] * p[0] + n[1][1] * p[1] + n[2][1] * p[2] + n[3][1]*p[3];
263    p_new[2] = n[0][2] * p[0] + n[1][2] * p[1] + n[2][2] * p[2] + n[3][2]*p[3];
264    p_new[3] = n[0][3] * p[0] + n[1][3] * p[1] + n[2][3] * p[2] + n[3][3]*p[3];
265
266    return p_new;
267 }
268
269
270 float[] transform_pointRotate(float[][] z, float[] p){
271
272    float[] p_new = {0.0,0.0,0.0,0.0}; // handle 4 elements as its 3D calc
273
274    p_new[0] = z[0][0] * p[0] + z[1][0] * p[1] + z[2][0] * p[2] + z[3][0]*p[3];
275    p_new[1] = z[0][1] * p[0] + z[1][1] * p[1] + z[2][1] * p[2] + z[3][1]*p[3];
276    p_new[2] = z[0][2] * p[0] + z[1][2] * p[1] + z[2][2] * p[2] + z[3][2]*p[3];
277    p_new[3] = z[0][3] * p[0] + z[1][3] * p[1] + z[2][3] * p[2] + z[3][3]*p[3];
```

Each variable m, n, and z are all sent to a different method which will handle 4 elements and its 3D calculations. A explain of how this works is that in the image, the first number of the p array, p_new[0], is used to hold the new value of x after it has been transformed. This applies to the y value of p_new[1], and the z value of p_new[2]. P_new is then returned as the new position of the shape after the transform. I made a new method for all 3 transforms although I believe I could have gotten the same final result for my project ad I only done one method to handle the 3D calculations.

```
123 //SCALING ENEMY-----------------------------------------------------------------------------------------------------------------
124 float scaleX = 0.99;
125 float scaleY = 0.99;
126 float scaleZ = 0.99;
127 float[][] scalingMatrix = {{scaleX,0.0,0.0,0.0},{0.0,scaleY,0.0,0.0},{0.0,0.0,scaleZ,0.0},{0.0,0.0,0.0,1.0}}; // scale by 0.99
128
```

The last shape to be transformed is my enemy shape which will be scaled. I started by changing the appropriate x, y, and z values in the matrix into variables which hold a value of 0.99. This means the shape will be scaled by 0.99.

Student No: K00282339
Student Name: Zach Ahearn

```
if(p63[1] < 50)
  {
    scaleX += 0.0099;
    scaleY += 0.0099;
    scaleZ += 0.0099;
  }
  else if(p63[1] >= 80)
  {
    scaleX -= 0.0099;
    scaleY -= 0.0099;
    scaleZ -= 0.0099;
  }
  scalingMatrix[0][0] = scaleX;
  scalingMatrix[1][1] = scaleY;
  scalingMatrix[2][2] = scaleZ;
```

In my draw function, I track one of the points on my enemy. If the Y value of that point becomes less than 50, the scaleX, Y, and Z all get 0.0099 added to them. This makes the shape increase in size. Also, if the Y value of that point becomes greater than 80, the values have 0.0099 taken away from them, decreasing the size of the shape. After the if statements, the changed x, y, and z values are set to the newly changed values. This essentially makes the shape increase in size to a certain point and decrease in size to a certain point.

```
293  void applyTransform(){
294
295    p33 = transform_pointScale(n, p33);
296    p34 = transform_pointScale(n, p34);
297    p35 = transform_pointScale(n, p35);
298    p36 = transform_pointScale(n, p36);
299    p37 = transform_pointScale(n, p37);
300    p38 = transform_pointScale(n, p38);
301    p39 = transform_pointScale(n, p39);
302    p40 = transform_pointScale(n, p40);
303    p41 = transform_pointScale(n, p41);
304    p42 = transform_pointScale(n, p42);
305    p43 = transform_pointScale(n, p43);
306    p44 = transform_pointScale(n, p44);
307
308    p45 = transform_pointRotate(z, p45);
309    p46 = transform_pointRotate(z, p46);
310    p47 = transform_pointRotate(z, p47);
311    p48 = transform_pointRotate(z, p48);
312    p49 = transform_pointRotate(z, p49);
313    p50 = transform_pointRotate(z, p50);
314    p51 = transform_pointRotate(z, p51);
315    p52 = transform_pointRotate(z, p52);
316    p53 = transform_pointRotate(z, p53);
317    p54 = transform_pointRotate(z, p54);
318    p55 = transform_pointRotate(z, p55);
319    p56 = transform_pointRotate(z, p56);
320    p57 = transform_pointRotate(z, p57);
321    p58 = transform_pointRotate(z, p58);
322    p59 = transform_pointRotate(z, p59);
323    p60 = transform_pointRotate(z, p60);
324    p61 = transform_pointRotate(z, p61);
325    p62 = transform_pointRotate(z, p62);
326
```

The applyTransform method then takes place which applies the new changes of the coordinates to all the points of my shapes. In the image above, p33 through p44 use the transform_pointScale

method which passes in the value on n and the related points. The same is done for m and z where they are passed into the appropriate transform methods.

```
float[][] scalingMatrix = {{0.97,0.0,0.0,0.0},{0.0,0.97,0.0,0.0},{0.0,0.0,0.97,0.0},{0.0,0.0,0.0,1.0}};
```

This matrix here works that the x, y, and z values are scaling factors. These values are less than 1, which means that the points of this matrix will be scaled down by 97% in all three dimensions. This is what applies the scaling transformation to the shape.

```
// transform matrix
 float theta = radians(10); // convert degrees to radians


float[][] scaleMatrix = {{1.1,0.0,0.0,0.0},{0.0,1.1,0.0,0.0},{0.0,0.0,1.1,0.0},{0.0,0.0,0.0,1}};  // scale 1.1,1.1,1.1
 /*    1.1  0   0  0
    =  0  1.1   0  0
       0  0   1.1  0
       0  0   0  1.1
 */
 float[][] rotateZMatrix = {{cos(theta),sin(theta),0.0,0.0},{-sin(theta),cos(theta),0.0,0.0},{0.0,0.0,1,0.0},{0.0,0.0,0.0,1.0}};
 /*    cos  -sin   0     0
    =  sin  cos   0     0
       0      0   1     0
       0      0    0     1
 */
```

The rotation transformation is a bit more complicated. It first scales the shape up by a factor of 1.1 in the x, y, and z, direction, increasing it by 10%. The rotateZMatrix makes the object rotate around the z axis. The shape rotates in the x and y plane at an angle according to the variable ‘theta’ which is stated above as being 10 radians. The z value is left at 1.0 which means it is unchanged.

# Student No: K00282339
# Student Name: Zach Ahearn

```
// rotation angle is 2 degrees
float theta = radians(2); // convert degrees to radians

// overwrite your rotate matrix values for each rotation type
rotateZMatrix[0][0] = cos(theta);
rotateZMatrix[0][1] = - sin(theta);
rotateZMatrix[1][0] = sin(theta);
rotateZMatrix[1][1] = cos(theta);
```

Inside the setup class is another matrix. This matrix overrides the previous matrix. The previous matrix rotated at 10 radians where as this matrix changes the values to rotate at 2 radians. The rotation transformation is the more complicated of the 3 with the inclusion of the theta variable and the radians.

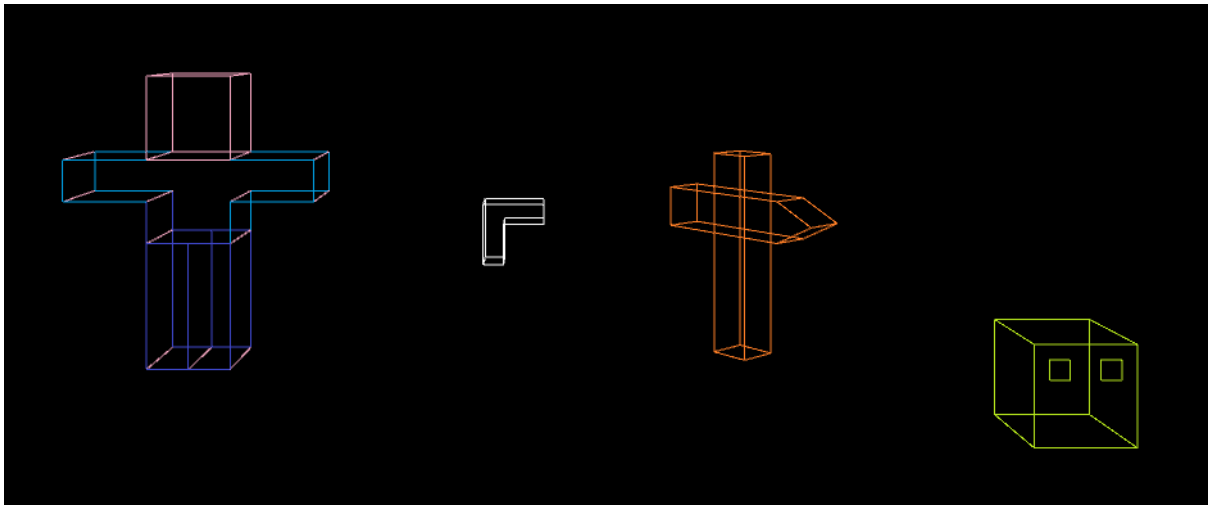
The user can view the rotation from different angles using the mouse with the help of the peasy cam library.

```
 9  import peasy.*;
10
11  PeasyCam cam;
```

The user can view the rotation from different angles using the mouse with the help of the peasy cam library. At the top of the code, the peasy library is imported and a peasyCam variable is made called cam.

```
47  void setup()
48  {
49    size(900, 900, P3D);
50    cam = new PeasyCam(this, 300);
51    cam.setMinimumDistance(200);
52    cam.setMaximumDistance(500);
```

Then in the setup method, the cam variable is made equal a new peasyCam object . The values set are related to the minimum and maximum distance of the camera from the shape drawn. The user can then change the view of the rotation by clicking and dragging the mouse around the screen.



This is the final result.

**Conclusion**

Student No: K00282339

Student Name: Zach Ahearn

Having completed 8 labs before this project, I was very prepared upon starting. The longest part of this project was having to plot out all 78 of my different points and then join the correct points to one another. The only difficulties I had with this project were making the scaling and translating shapes loop by increasing and decreasing their values. To solve this problem, I used debug statements to check what values were changing while the shape was transforming. Once I knew what values were changing and how they were changing, I could make appropriate if statements to make a max and minimum range of the shape's points can increase or decrease to during its transformation. Like I stated earlier, the project could have been shortened down had I used only one transform method to handle 3D calculations rather than making 3 separate methods for the 3 different transforms.