MATH 3333: Project - Computer Graphics

Justin Santos

1 July 2019

1 Introduction

Computer graphics is a field of computer science that plays an important role in everyday life. It is used in video games, medical imaging, model visualization, and much more. The mathematics behind computer graphics is dominated by linear algebra and matrix manipulation. Think of a screen as a huge matrix, with each pixel being a single entry in that matrix. Moving an object on the screen elsewhere would involve geometric transformation of that object's pixels, which is done through a combination of matrix multiplication and addition. There are three main geometric transformations: translation, scaling, and rotation. Translation is a lateral movement of the object, scaling makes the object bigger or smaller, and rotation, obviously, rotates the object.

This project will be primarily focused on the transformation of two-dimensional graphical objects as it is much simpler than three-dimensional transformations, but will take a glimpse into three-dimensional transformations at the end.

2 The Mathematics

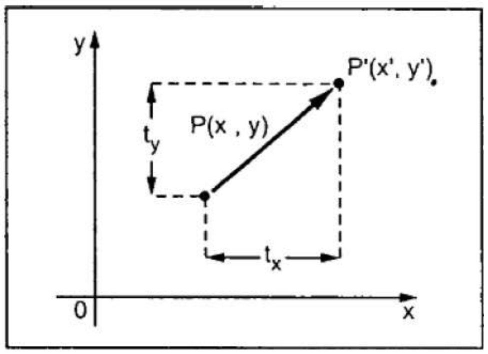
2.1 Translation

Translation is, arguably, the simplest of the three transformations. The formula for a two-dimensional translation of an object is as follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

where x and y is the the original coordinate location of the point, t_x and t_y are the transformations in x and y, respectively, and x' and y' is the new coordinate location. The matrix containing t_x and t_y is called the translation matrix. This formula must be applied to every point in the object. Simply put, translation is just simple matrix addition. Figure 1 shows this formula visually.

Figure 1: Translation of a point in two-dimensions



2.2 Scaling

There is an important term that comes up when talking about scaling and that is the scale factor. The scale factor is the amount the object is to be scaled by. This value must be greater than 0. Fractional values ($0 < x < 1$) will decrease an object's size, whereas values greater than 1 will increase its size. The formula for the two-dimensional scaling of an object follows:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

where x and y is the the original coordinate location of the point, s_x and s_y are the scale factors in x and y, respectively, and x' and y' is the new coordinate location. This formula must be applied to every point in the object. The matrix containing s_x and s_y is called the scaling matrix. Note that if $s_x = s_y = s$ the scaling matrix becomes:

$$sI = s \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

which is just a scalar multiplication of the identity matrix. A visualization of the scaling formula is available in figures 2 and 3.

Figure 2: Before the scaling of an object in two-dimensions

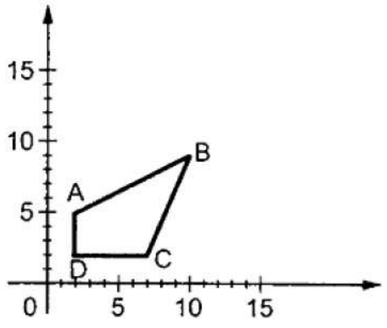
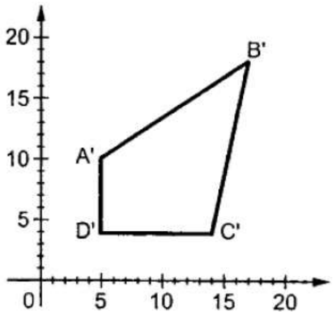


Figure 3: After the scaling of an object in two-dimensions with $s = 2$



2.3 Rotation

Rotation is the toughest of the three transformations because it involves trigonometry. Figure 4 will help with the understanding of the derivation for the rotation formula. Any arbitrary point (x, y) can be transformed to polar coordinates by the use of these two equations:

$$x = rcos(\phi) \tag{1}$$
$$y = rsin(\phi) \tag{2}$$

A rotation would be adding another term, namely θ, to ϕ. As such, the new point after the rotation is:

$$x' = rcos(\phi \ + \ \theta) = rcos(\phi)cos(\theta) - rsin(\phi)sin(\theta) \tag{3}$$
$$y' = rcos(\phi \ + \ \theta) = rcos(\phi)sin(\theta) + rsin(\phi)cos(\theta) \tag{4}$$

Now, substitute equations (1) and (2) into equations (3) and (4):

$$x' = xcos(\theta) - ysin(\theta) \tag{5}$$
$$y' = xsin(\theta) + ycos(\theta) \tag{6}$$

Now, we can turn equations (5) and (6) into its matrix multiplication counterpart to get the formula for a rotation about (0, 0):

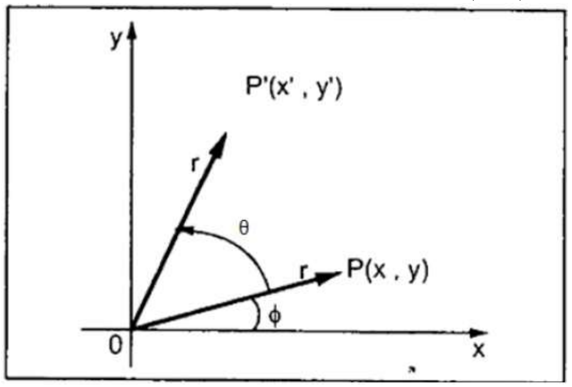
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix}$$

where x and y is the the original coordinate location of the point, θ is the rotation amount in radians, and x' and y' is the new coordinate location. To make this formula more generalized, however, we can add a pivot point to this equation so that we can rotate the object about any arbitrary point on the Cartesian plane, not just (0,0). When adding a pivot point, the final equation will look like this:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \left(\left(\begin{bmatrix} x \\ y \end{bmatrix} \text{-} \begin{bmatrix} p_x \\ p_y \end{bmatrix} \right) \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta) & cos(\theta) \end{bmatrix} \right) + \begin{bmatrix} p_x \\ p_y \end{bmatrix}$$

where p_x and p_y are the x and y coordinates of the pivot point, respectively. On a side note, the matrix with the $sin(\theta)$ and $cos(\theta)$ is called the rotation matrix. Again, this formula must be applied to every point in the object.

Figure 4: Rotation of a point about (0, 0)



3 The Code

In order to code these formulas, I used a library called NumPy (which is imported as ”np” in the code). NumPy adds support for large arrays and matrices, and also provides high-level mathematical functions to operate on these matrices. In order to visualize the transformations, I used another library called Matplotlib (which is imported as ”pyplot” in the code). This allowed me to plot the object into a coordinate grid to help see the transformations.

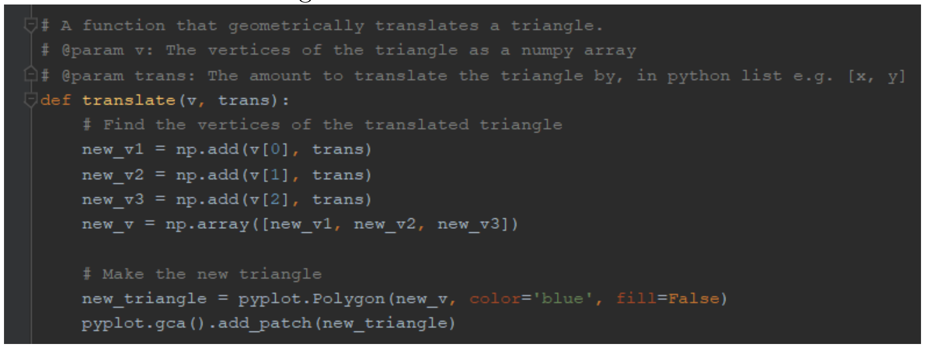
The code uses triangles as its object. A triangle has three vertices, which are the important points when defining the new, transformed triangle. As such, the transformation formulas will be applied to three points — the original triangle's vertices — in order to form the transformed triangle.

Here are some important functions from the aforementioned libraries that are used:

$$\begin{aligned} np.array() &- \text{forms a new matrix} \\ np.add() &- \text{adds two matrices} \\ np.subtract() &- \text{subtracts two matrices} \\ np.matmul() &- \text{multiplies two matrices (if possible)} \\ pyplot.Polygon() &- \text{creates a triangle using a matrix of vertices} \\ pyplot.gca().add\_patch() &- \text{updates the plot to include the new triangle} \end{aligned}$$

3.1 Translation

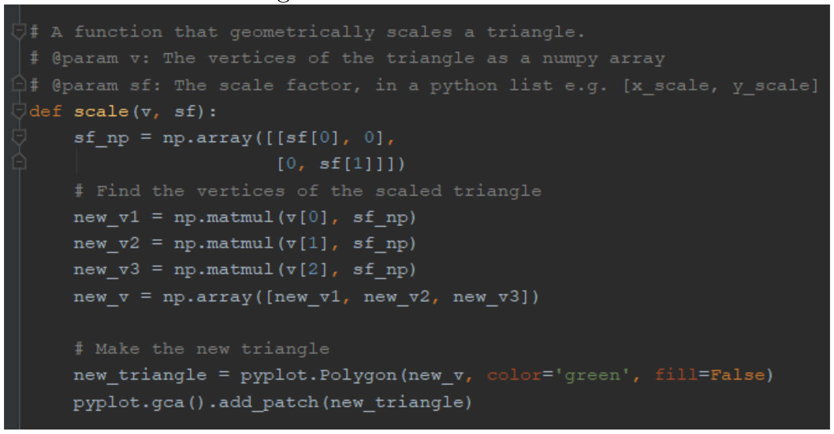
Figure 5: The translate function



```python
# A function that geometrically translates a triangle.
# @param v: The vertices of the triangle as a numpy array
# @param trans: The amount to translate the triangle by, in python list e.g. [x, y]
def translate(v, trans):
    # Find the vertices of the translated triangle
    new_v1 = np.add(v[0], trans)
    new_v2 = np.add(v[1], trans)
    new_v3 = np.add(v[2], trans)
    new_v = np.array([new_v1, new_v2, new_v3])

    # Make the new triangle
    new_triangle = pyplot.Polygon(new_v, color='blue', fill=False)
    pyplot.gca().add_patch(new_triangle)
```

3.2 Scaling

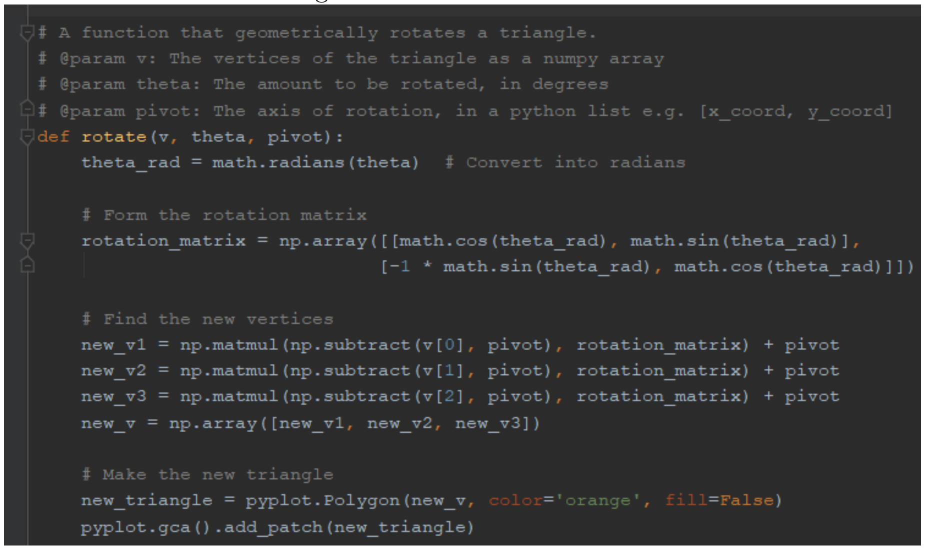
Figure 6: The scale function



```
# A function that geometrically scales a triangle.
# @param v: The vertices of the triangle as a numpy array
# @param sf: The scale factor, in a python list e.g. [x_scale, y_scale]
def scale(v, sf):
    sf_np = np.array([[sf[0], 0],
                      [0, sf[1]]])
    # Find the vertices of the scaled triangle
    new_v1 = np.matmul(v[0], sf_np)
    new_v2 = np.matmul(v[1], sf_np)
    new_v3 = np.matmul(v[2], sf_np)
    new_v = np.array([new_v1, new_v2, new_v3])

    # Make the new triangle
    new_triangle = pyplot.Polygon(new_v, color='green', fill=False)
    pyplot.gca().add_patch(new_triangle)
```

3.3 Rotation

Figure 7: The rotate function



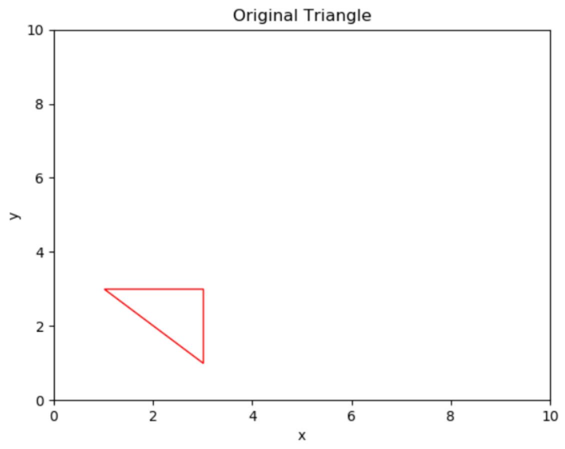
```
# A function that geometrically rotates a triangle.
# @param v: The vertices of the triangle as a numpy array
# @param theta: The amount to be rotated, in degrees
# @param pivot: The axis of rotation, in a python list e.g. [x_coord, y_coord]
def rotate(v, theta, pivot):
    theta_rad = math.radians(theta)  # Convert into radians

    # Form the rotation matrix
    rotation_matrix = np.array([[math.cos(theta_rad), math.sin(theta_rad)],
                                [-1 * math.sin(theta_rad), math.cos(theta_rad)]])

    # Find the new vertices
    new_v1 = np.matmul(np.subtract(v[0], pivot), rotation_matrix) + pivot
    new_v2 = np.matmul(np.subtract(v[1], pivot), rotation_matrix) + pivot
    new_v3 = np.matmul(np.subtract(v[2], pivot), rotation_matrix) + pivot
    new_v = np.array([new_v1, new_v2, new_v3])

    # Make the new triangle
    new_triangle = pyplot.Polygon(new_v, color='orange', fill=False)
    pyplot.gca().add_patch(new_triangle)
```

4 The Application

Now, we can see these transformation functions in action. As a simple example, I will be performing all three transformations on a triangle with vertices at (1, 3), (3, 1), and (3, 3). When the triangle is transformed, the old location will be denoted by red, dashed outline. All of these plots were produced using Matplotlib.

4.1 Original Triangle

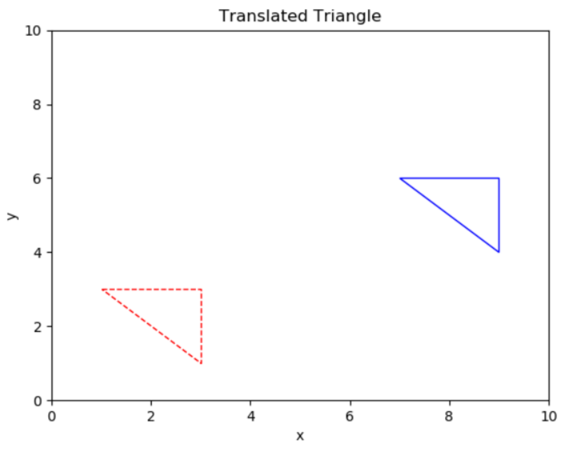
Figure 8: The original triangle with vertices at (1,3), (3, 1), and (3, 3)



4.2 Translation

Here is a translation of the original triangle by 6 in the positive-x direction and 3 in the positive-y direction:

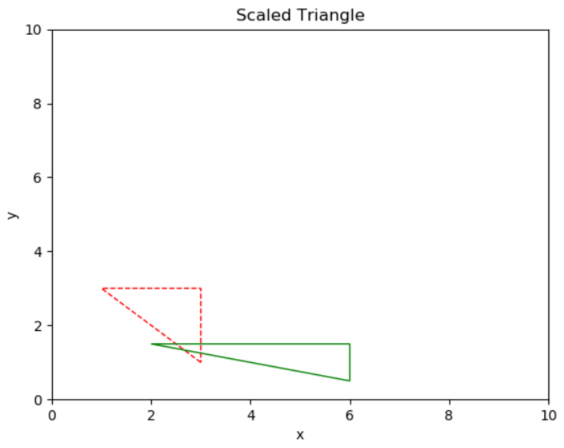
Figure 9: A translation of the original triangle with $t_x = 6$ and $t_y = 3$



4.3 Scaling

Now, here is a scaling of the original triangle by 2 in x and 0.5 in y:

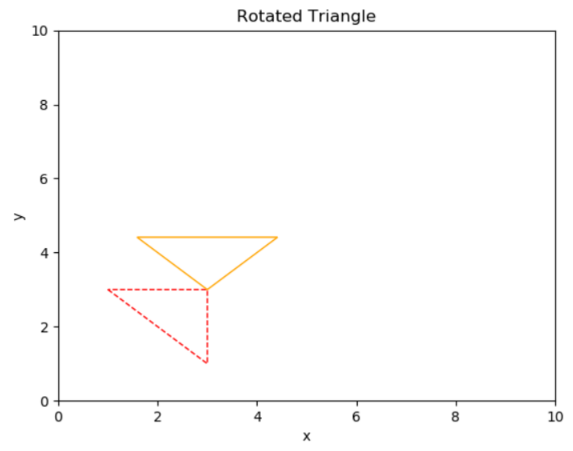
Figure 10: A scale of the original triangle with $s_x = 2$ and $s_y = 0.5$



4.4 Rotation

Lastly, here is a 225 degree rotation of the original triangle about its vertex at (3, 3):

Figure 11: A rotation of the original triangle with $p_x = p_y = 3$ and $\theta = 225°$



5 A Glimpse Into Three-Dimensional Geometric Transformations

Geometric transformations in three dimensions are significantly harder to calculate than in two dimensions. Here are just a few examples of what the transformation matrices would be like.

5.1 Translation Matrix

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.2 Scaling Matrix

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5.3 Rotation Matrix

$$\begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6 Conclusion

Hopefully by now it is evident how useful linear algebra is in computer graphics. However, geometric transformations are only a small part of computer graphics. There are way more uncovered topics that require even more linear algebra to understand. This was only a small, yet highly important, application of linear algebra towards computer science and computer graphics.