Abe Arends
CSC 340 Assignment 1 Report

Angle Step Size	**# Rotations**	**Absolute Color Error**	**Pixel Rounding Error**	**(# Rotations) * (Pixel Displacement)**
45	8	0.348	0.5	4.0
60	6	0.565	0.5	3.0
90	4	0.0	1.68×10^{-14}	6.72×10^{-14}
120	3	0.559	0.5	1.5
180	2	0.0	5.8×10^{-7}	1.16×10^{-13}
360	1	0.0	1.16×10^{-13}	1.16×10^{13}

Figure 1: Scaling factor: 60 degrees, second step

Rotation Factor: 180

In this Assignment, we were tasked with utilizing the matrix multiplication algorithm we learned in class to create a program that utilizes the OpenCV and Numpy libraries to copy, rotate, and display a bordered image.

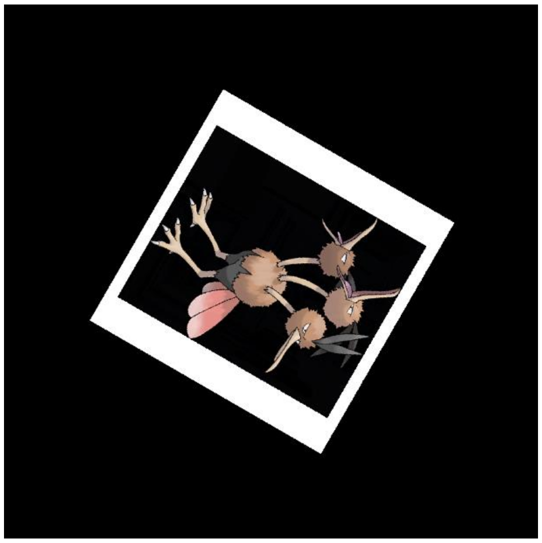
Figure 2: Scaling factor: 60 degrees, second step

For this program, I used a two-dimensional array as my data structure containing the pixel data that I express as a matrix. A two-dimensional array is effective for the matrix operations required in this assignment. It also allows for better code readability, as accessing row i, column j for some matrix A (A_{ij}) is equivalent to accessing index i, j in a two-dimensional Python array (A[i][j]).

The procedure for a single rotation is as follows:

Save the previous image. If this is the first rotation, the previous image will be the original, bordered image. Next, consider that all pixel data is not reflective of where it would map to a Cartesian plane. That is to say, the origin's index is [0][0]. In order to apply trigonometric functions to the image, we must translate the image up and to the left by half of the image's length (after the bordering operations are complete, we always have a square image/matrix). Next, matrix multiplication is applied to each row (Y) and column (X) value against the rotation matrix, which is specifically designed to rotate an image by a given theta. Finally, these pixels are mapped back into a blank image of the same size. While iterating through i for rows and j for columns, pixel color data is mapped as follows: new_image[i, j, …] = old_image[r, c, …]. This insures that every pixel in the new image is mapped.

Abe Arends
CSC 340 Assignment 1 Report

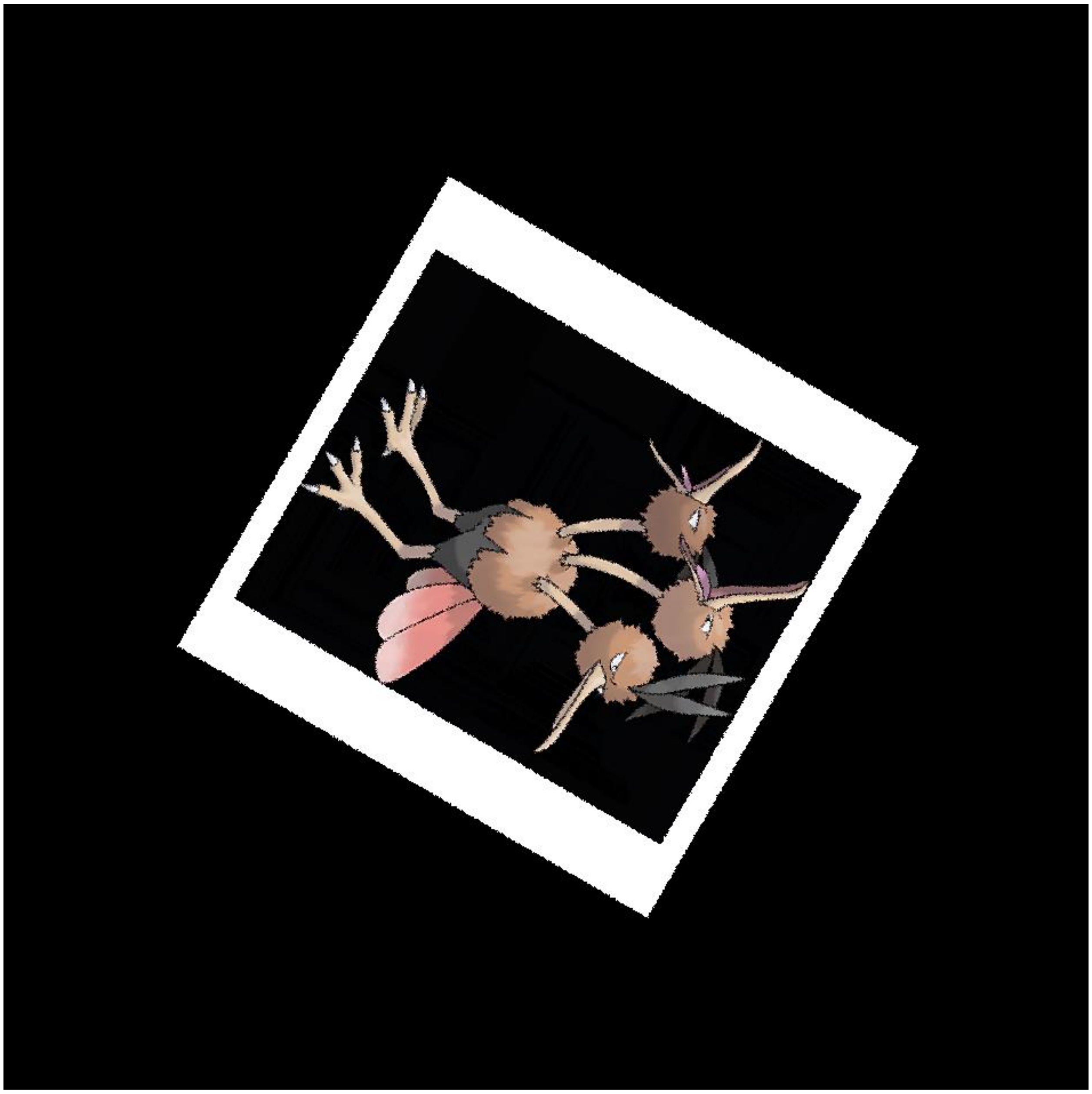

Figure 3 Blown up to illustrate. 40 degrees done 3 times. Blur is increased compared to 60 degrees done twice.

For rounding, I did not use any special algorithms, so there is certainly room for improvement. However, I am slightly impressed with how well a simple round() function worked before casting the given floating-point numbers from the matrix multiplication into integers. I doubt this would have been nearly as effective in a smaller image, especially when values in the rotation matrix are close to 0.5 (cases like 45 degrees). If I were to make an improvement, I would consider splitting the work of my cosine and sine into multiple multiplication matrices involving tangents, and/or scale the image up, place pixels, and use some sort of grouping algorithm to reprocess the image in its original scale.

Abe Arends
CSC 340 Assignment 1 Report

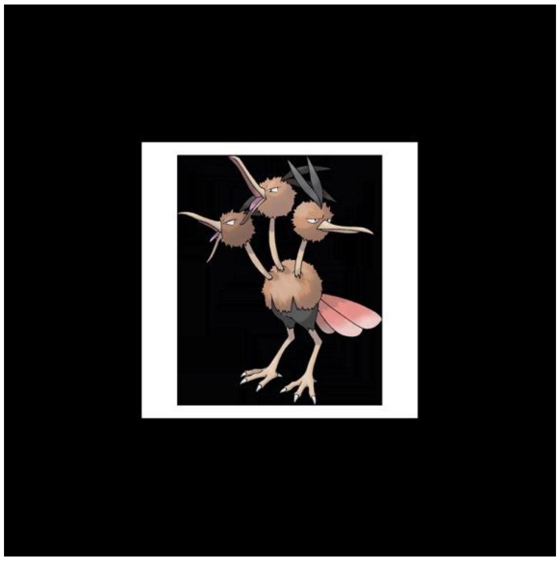
Figure 3: Scaling Factor: 180 degrees w/ no color error

There is a strong correlation between color error and perceived blurriness. The higher the error, the less distinguishable the image is. However, for non-full rotations, pixel rounding error does especially well to highlight the issues brought on by referring straight lines to a diagonal and proceeding to straighten them out again (three 30 degree turns highlights this). My image has straight borders on top of the black border that is generated at the beginning of the program, so it was very clear that there could be work done to improve these corners.

As a side-note, I wonder what algorithms computer programs such as Microsoft word use to rotate images, as they are so fast and accurate. I will likely be looking into optimizing an image rotation as a portfolio project in the near future.