Graphics Programming

Assignment 02

First, the original images….

Had to cram them in there! We got raccoons, promiscuous gremlins, a cube, and some generic cereal!!!! From left to right, then up to down, is image 1, 2, 3, and 4.

Mirror Transformation

```
mirror = getT()
mirror[axis, axis] *= -1
mirror[axis, axis+1] = img.shape[0]+1

out = cv2.warpPerspective(img,
mirror,img.shape[:2])
```

The code for this was relatively straightforward. All I had to do was create a 3x3 identity matrix with getT(), then modified that matrix to flip it across the y-axis.

@ axis param = specified axis to flip across. 0 = x axis, and 1 = y axis. I also added a 1 pixel translation to the flip to make it clear where the mirrored image was.

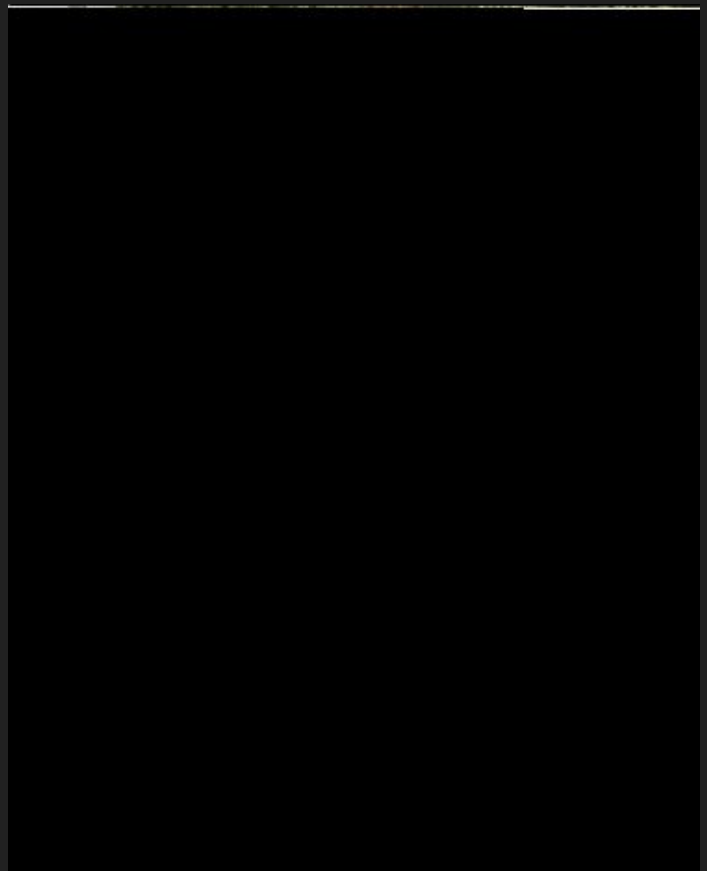
Mirror pt. 2

You can see where the image is mirrored at the top of this output image.

For clarification, the transformation matrix looks like
[1, 0, 0]

[0, -1, 1]

[0, 0, 1]



Rotate

```
    t1[0, 2], t1[1, 2] = -w, -h
    t2[0, 2], t2[1, 2] = w, h

    r[0, 0], r[0, 1] = cos, -sin
    r[1, 0], r[1, 1] = sin, cos

    out = cv2.warpPerspective(img, t2@r@t1, (w, h))
```

w and h are the width and height of the input image. t1 is the matrix that will transform the lower-right corner to the upper-left, and t2 does the opposite so that the lower-right corner is where it should be. r is the matrix that actually does the rotating. Cos and Sin variables are the cosine and sine of the angle given. The angle is input in degrees but turned into radians. Lastly, the warpPerspective() function puts the transformations in right-to-left since that's just how code goes.

```
t1 = [1, 0, -w]
     [0, 1, -h]
     [0, 0, 1]

r = [cos, -sin, 0]
    [sin, cos , 0]
    [0,    0  , 1]

t2 = [1, 0, w]
     [0, 1, h]
     [0, 0, 1]
```

These are the respective matrices.

Rotate but EPIC!! (rotate 2)

```
a = w*sin
b = w*cos

dX = -(w-b)
t3[0, 2] = dX
dW = int(h*sin+w*cos)

dY = int((a+(h*cos))-h)
t3[1, 2] = dY #e = dH and dY
dH = dY+h

m = t3@t2@r@t1

out = cv2.warpPerspective(img, m, (dW, dH))
```

```
t3 = [1,  0,  dX]
     [0,  1,  dY]
     [0,  0,   1]
```

The code from the first rotate applies. The only difference is that another matrix is made and the new image is resized to have the rotated image fit within the borders of the frame.

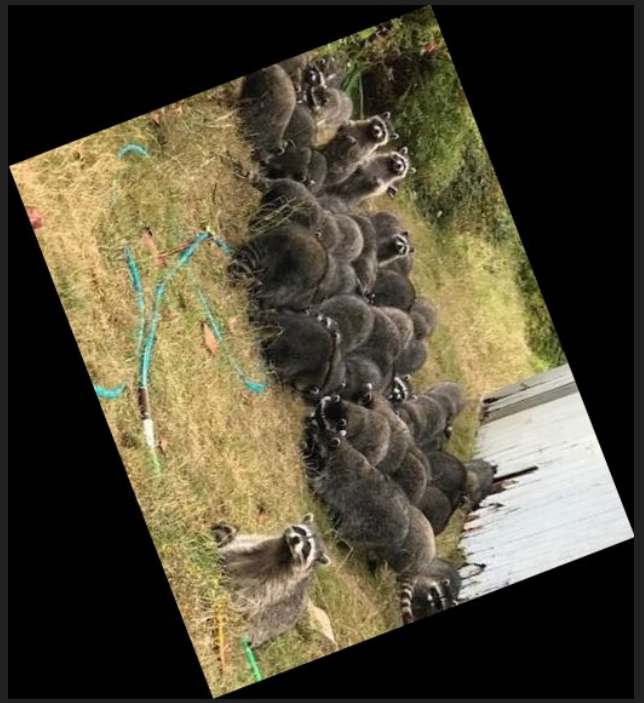
dX is the change in the image's x, and the dY is the change in y.

dW and dH are the change in width and height of the window.

These values were found using trig to figure out how much to move and modify the image according to an angle, theta.

Rotate but EPIC!!!(rotate 2)

Rotated by 30 degrees



Rotated by 69 degrees

Putting da squares on da cub3

```
coords_1 = [[167, 236], [391, 204], [410, 433], [176, 493]]
coords_2 = [[23, 170], [154, 246], [158, 489], [27, 400]]

#raccoons
srcPoint = np.float32([[0, 0], [w1, 0], [w1, h1], [0, h1]])
destPoint = np.float32(coords_1)
M = cv2.getPerspectiveTransform(srcPoint, destPoint)
out1 = cv2.warpPerspective(sq, M, (iw, ih), flags=cv2.INTER_NEAREST)

#gremlin
srcPoint2 = np.float32([[0, 0], [w2, 0], [w2, h2], [0, h2]])
destPoint2 = np.float32(coords_2)
N = cv2.getPerspectiveTransform(srcPoint2, destPoint2)
out2 = cv2.warpPerspective(sq2, N, (iw, ih), flags=cv2.INTER_NEAREST)

mask = (out1 > 0)
mask2 = (out2 > 0)

p1 = (out1*mask+img3*(1-mask))

out = out2*mask2+p1*(1-mask2)
```

This is a lot of code, but to break it down: The coords vars are the coords of the corners of the cubes the images are being put on. The images being put onto the cube have been sliced from a larger image, which is why their srcPoints have [0, 0] in them.

The width and height variables are also just the sizes of those sliced images. After getting those images warped onto the cube, the last bit of code creates two masks. These masks make it possible to add the warped image onto the cube.

Putting da squares on da cub3 pt. 2

Flattening my cereal!

```
h, w = img.shape[:2]
w_to_h = 8 /(12+(1/8))

srcPoint = np.float32([(135, 65), (424, 81), (414, 572), (148, 504)])
destPoint =  np.float32([(0, 0), (int(h*w_to_h), 0), (int(h*w_to_h), h), (0, h)])
M = cv2.getPerspectiveTransform(srcPoint, destPoint)

out = cv2.warpPerspective(img, M, (int(h*w_to_h), h))
```

To flatten a perspective image(cereal, in this case), the part that would be flat (the front) needs to be transposed onto a new image so that it appears flat. Basically, take the coordinates of the front of the cereal box and put it on a new image where the top right corner is the origin and the rest is relative to that.

The magic comes in when it has to have the aspect ratio modified so it still looks like a cereal box. I got a generic value for the dimensions of a cereal box (8 x 12 ⅛) and divided those numbers. Then used those numbers to create a proportionally sized image.

Flattening my cereal!!

Putting promiscuous gremlin on my cereal and unflatting it

```
flat[h//2 - 100: h//2 + 100, w//2 - 100: w//2 + 100] = sq

srcPoint = np.float32([(0, 0), (w, 0), (w, h), (0, h)])
destPoint = np.float32([(135, 65), (424, 81), (414, 572), (148, 504)])

M = cv2.getPerspectiveTransform(srcPoint, destPoint)
out = cv2.warpPerspective(flat, M, (w0, h0))

mask = cv2.warpPerspective(flat*0+255, M, (w0, h0), flags=cv2.INTER_NEAREST)
mask = mask > 0
out = out*mask+img*(1-mask)
```

I put a square image in the center of the flattened cereal box. Then I essentially did the reverse of what I did to flatten the cereal box. I took the src point and transformed that into what the points were originally.
h,w = height and width of the flat image
h0, w0 = height and width of the original, unflattened image.
The out is just the flat image transposed to the original coordinates, with the resulting image being the same size. Then I create a fancier mask and use that to composite onto my original, unflattened image.

Unflattening my cereal pt.2