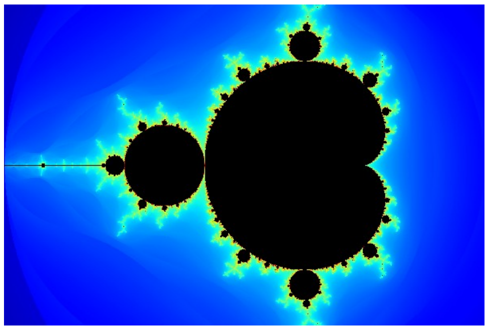
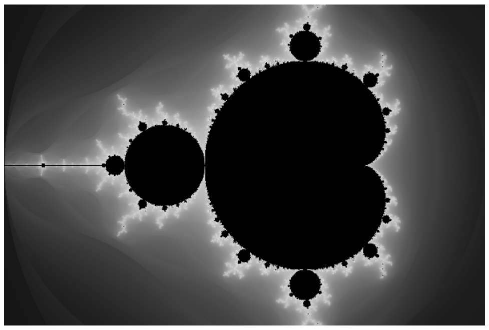
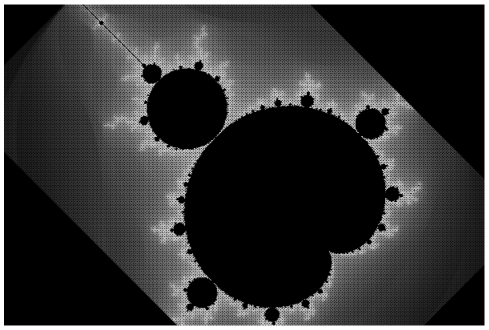
Colton Watson
May 7, 2018
Math 2270

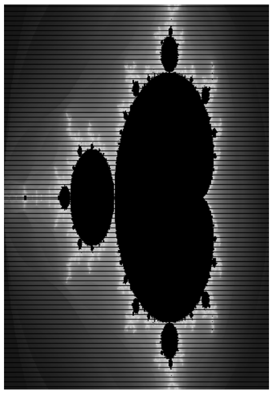
Image Translations: Rotating, Shearing, and Scaling

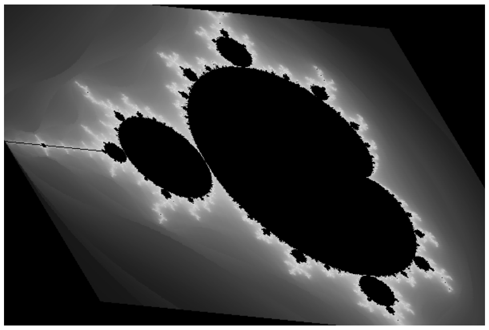
Nearly everything related to digital images can be manipulated with linear algebra. A typical image is made from a series of data that encodes the image as a matrix of color values, which then can be used in a high level programming language such as MATLAB or GNU Octave. This project is an implementation of various transformation matrices in Octave. Currently, the project uses forward mapping for all of the transformations, which means that the destination pixel is found from a transformation of the source pixel. The advantage of using this mapping is that the transformation is usually easier to understand and implement, but comes at the cost that the image from the transformation most likely will have holes – pixels that are not written into. Most images are stored as a three dimensional array, having the height, width, and three layers for the red, green, and blue component of the image. For simplicity, this projects makes the images grayscale, which only has one layer. An example of the transformations are provided below.


Original color image


Conversion to grayscale


Rotation


Scale


Shear

One of the most fundamental transformations is scaling. Start with the identity matrix, and then on the main diagonal, replace one of the factors with the scale in one of the axes. The transformation is given below. To use this, go through each pixel in the source image, multiply the current pixel by the matrix, and that will return the pixel destination of transformed image. It should be noted that this is not what is typically used in most image manipulation tools, since it is not nearly as fast or as accurate as other methods.

$$\begin{pmatrix} x_{destination} \\ y_{destination} \end{pmatrix} = \begin{pmatrix} horizontal\,scale & 0 \\ 0 & vertical\,scale \end{pmatrix} \begin{pmatrix} x_{source} \\ y_{source} \end{pmatrix}$$

Another transformation is rotation, displayed below. Functionally, it is identical to the previous transformation, simply multiply the current pixel by the transformation matrix, and that will return the destination pixel, and do that for every pixel in the source image. However, that will rotate the image about one of its corners, which is not useful when only half or less of the image is visible on the screen, so to mitigate the problem, first subtract the current pixel by the center pixel location, multiply it by the transformation matrix, then add the location of the center pixel back. This will rotated the image about the center of the image, which is much more useful on a finite workplace, such as an array.

$$\begin{pmatrix} x_{destination} \\ y_{destination} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \left(\begin{pmatrix} x_{source} \\ y_{source} \end{pmatrix} - \begin{pmatrix} x_{center} \\ y_{center} \end{pmatrix} \right) + \begin{pmatrix} x_{center} \\ y_{center} \end{pmatrix}$$

Finally in this example is shearing. Again, the basic transformation matrix is directly below. It is used almost identically as rotation, the only difference is the transformation matrix.

$$\begin{pmatrix} x_{destination} \\ y_{destination} \end{pmatrix} = \begin{pmatrix} 1 & horizontal\,shear \\ vertical\,shear & 1 \end{pmatrix} \left(\begin{pmatrix} x_{source} \\ y_{source} \end{pmatrix} - \begin{pmatrix} x_{center} \\ y_{center} \end{pmatrix} \right) + \begin{pmatrix} x_{center} \\ y_{center} \end{pmatrix}$$