Eric Roberts
CS 151

Handout #14
October 23, 2022

# Project #3—ImageShop

*The section on the histogram-equalization algorithm is adapted from an assignment created by Keith Schwarz.*

**Due: Friday, November 4, 5:00 P.M.**

Most of you have probably had occasion to use some sort of image-editing software such as Adobe Photoshop™ or Adobe Illustrator™. In this assignment, you will have the chance to build a simple version of an image editor called ***ImageShop,*** which implements several simple operations on images along with a few more interesting ones.

Performing image manipulation in Python requires the use of the Pillow library, which is not included in the standard distribution. You should be able to load Pillow by typing the following command in the Mac Terminal window or the Windows command line:

```
pip3 install pillow
```

## Milestone #0—Understand the starter project

The starter project that you download from the course web site contains a skeleton version of `ImageShop.py`, which is the only module you need to modify (although you will have to create two others) and a few library files you will need for this project. Running the `ImageShop.py` application in the starter folder creates the screen image shown in Figure 1, which includes a button area with **Load** and **Flip Vertical** buttons and a blank image area. Clicking **Load** brings up a file chooser that lets you select an image. If, for example, you go to the `images` directory, you will see a list of image files supplied with this project. If you click on `VanGogh-StarryNight.png`, ImageShop will load that file and center it in the

**Figure 1. Screenshot of the initial version of the ImageShop application**

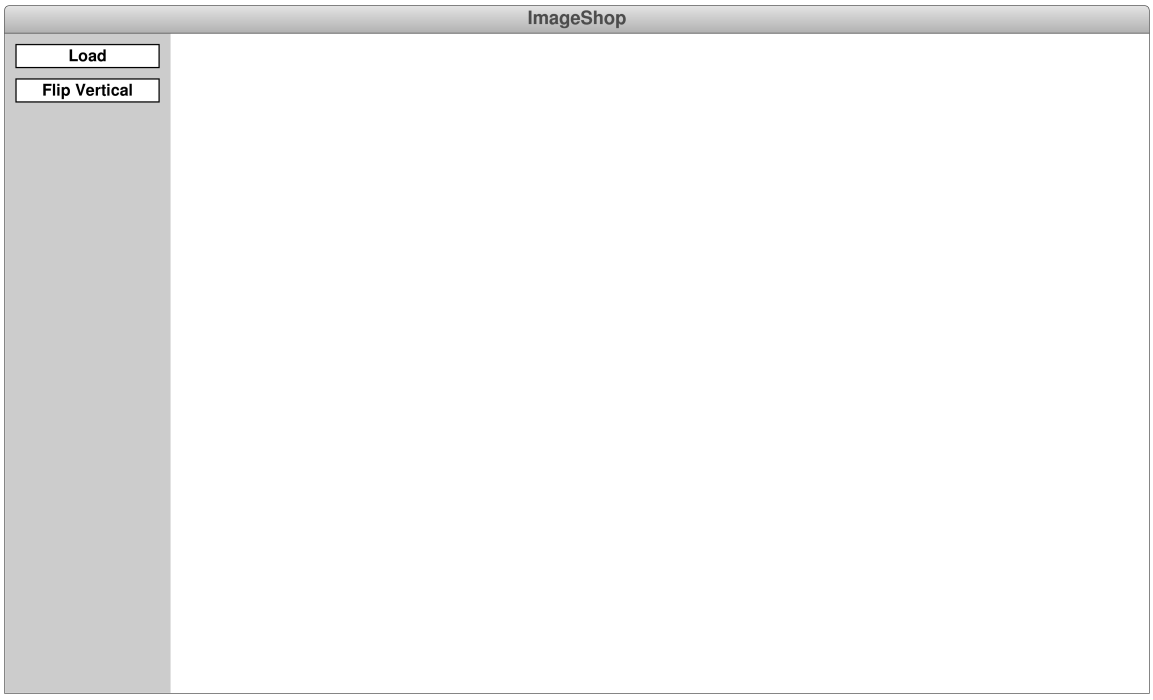


**Figure 2. The ImageShop screen after loading** `VanGogh-StarryNight.png`

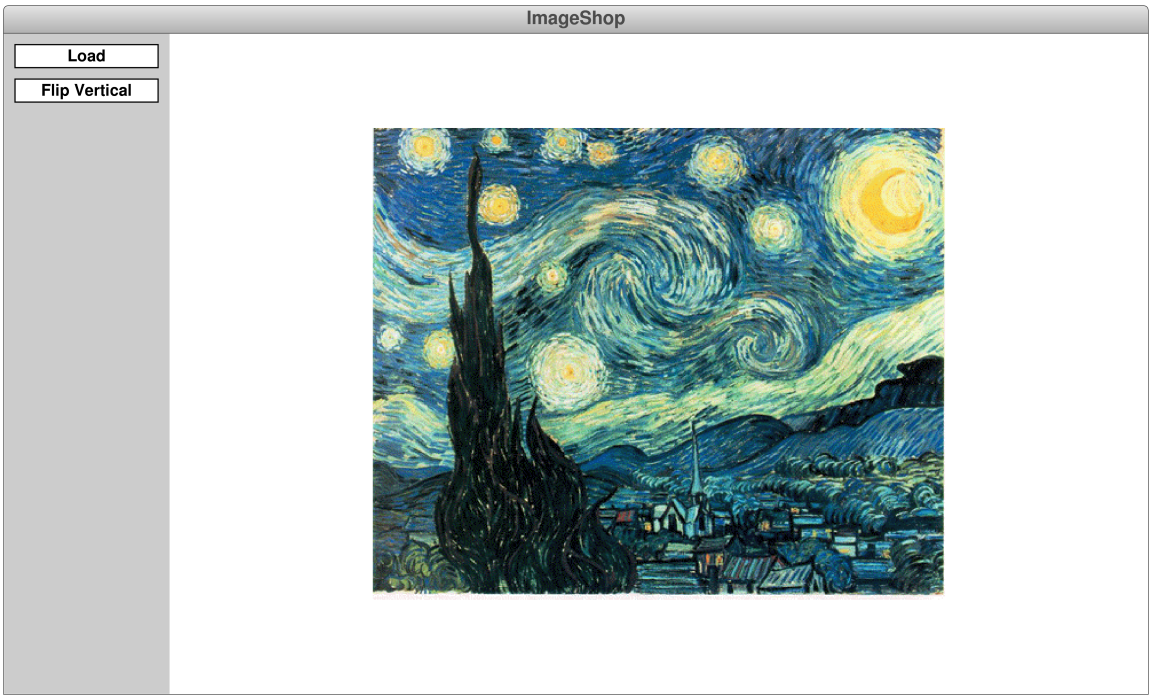
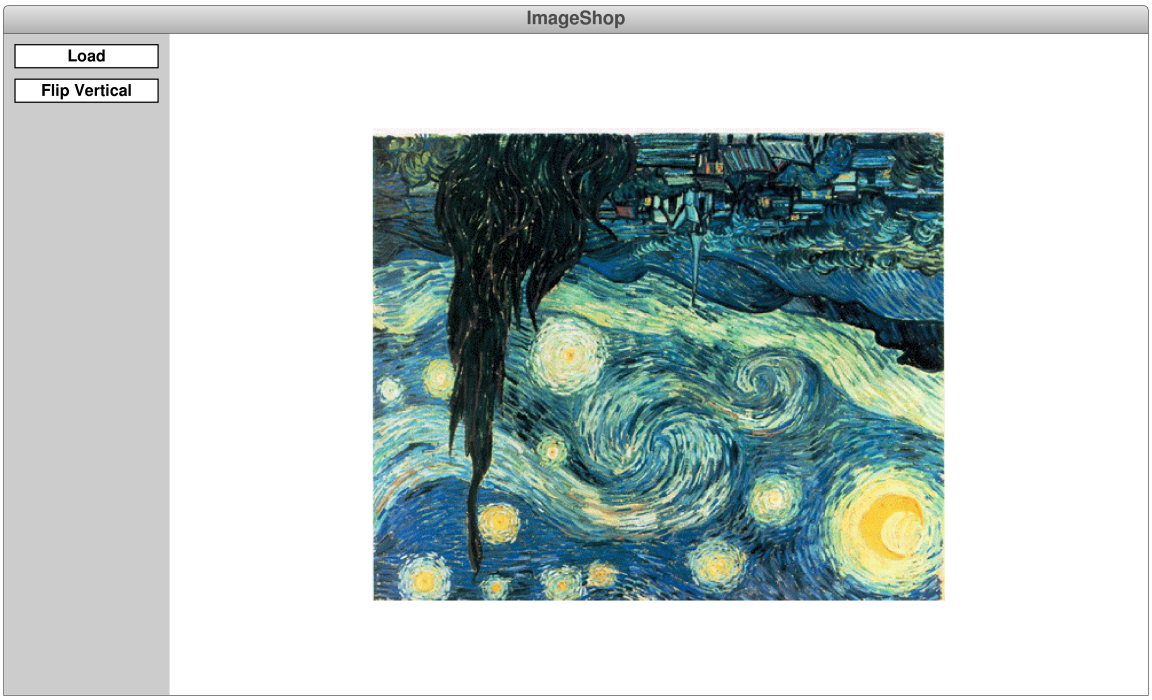



image area, as shown in Figure 2. If you then click the **Flip Vertical** button, you will get the picture shown in Figure 3. Clicking **Flip Vertical** again restores the original image.

The code for `ImageShop` in the starter project uses a new class called `GButton` in the `button.py` library, which is responsible for displaying the buttons on the screen. The constructor for `GButton` takes the text to display in the button, along with a function to call

**Figure 3. The ImageShop screen after flipping the image vertically**



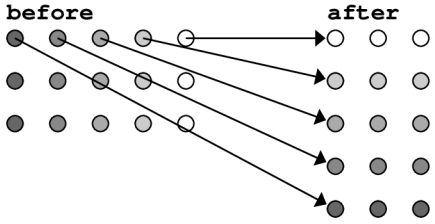
when the user clicks the button. In the `ImageShop` code, the call to the `GButton` constructor appears within the function `add_button`, which also takes care of placing each new button just underneath the previous one in the button area.

### Milestone #1—Add a Flip Horizontal button

To start you off with a relatively straightforward task, add a new **Flip Horizontal** button that flips the image from left to right. To display the button on the screen, you simply need to copy the code that displays the **Flip Vertical** button, adding the functions `flip_horizontal_action`, which responds to the button click, and `flip_horizontal`, which implements the image transformation. The only function that requires any real change from the `flip_vertical` model is `flip_horizontal`, where you have to reverse each row in the pixel array instead of reversing the array as a whole.

### Milestone #2—Add the Rotate Left and Rotate Right buttons

Your next step is to add two buttons that rotate the image on the screen by 90°. Once again, the most challenging part of this milestone is adding the functions that transform the `GImage`. Implementing these rotations is a bit tricky, since you have to figure out where each of the old pixels ends up in the new pixel array. Since the dimensions of the new image are inverted from the original—the new width is the old height and vice versa—you need to create a new array with the correct number of elements. Once you have done so, you need to copy each old pixel into the correct position in the new array. This process is illustrated in the following image, which shows where each pixel in the first row of the old array (`before`) moves to the new array (`after`) during a left rotation:




### Milestone #3—Add a Grayscale button

In this simplest of all the milestones, your job is to add a **Grayscale** button that replaces the image with a new one in which the image has been converted to grayscale. The code for the necessary image transformation appears not only in the book but also in the file `GrayscaleImage.py` in the starter folder. To implement this milestone, you should not copy and paste the code from `GrayscaleImage.py` but instead import the function or functions you need. The advantage of doing so is that there is then only one copy of the code, which is shared between the two applications. Copying code inevitably creates problems for software maintenance, since changes made to one copy may not be incorporated into the other, leading to incompatible versions.

### Milestone #4—Implement a Green Screen button

The **Green Screen** button implements an operation that is used all the time in movies to merge actors into a background scene. The basic technique is called ***chroma keying,*** in which a particular range of colors is used to represent a background that can later be made

transparent using a computational process. The most common colors used in chroma keying are green and blue (which give rise to the more specific names ***green screen*** and ***blue screen***) because those colors are most easily differentiated from flesh tones.

When studios use the green screen technique, for example, the actors are filmed in front of a green background. The digital images are then processed so that green pixels are made transparent, so that the background shows through when the partially transparent image is overlaid on top of the background image.

To illustrate this process, suppose that you are making *Star Wars: The Force Awakens* and that you want to superimpose an image of Daisy Ridley's character Rey on top of the following shot of the interior of the Millennium Falcon:

You then shoot an image of Rey in front of a green screen like this:

If you skip the green pixels in the image and copy all the others on top of the image of the Millennium Falcon, you get the following composite picture:

When the user clicks the **Green Screen** button, the first thing your program has to do is read in a new image using the `choose_input_file` function in the `filechooser` module in much the same way that the `load_action` function does in the supplied code from the starter file. Once you have the new `GImage`, you need to go through each pixel in both the old image and the new image and replace the old pixel value with the new one, unless the new pixel is green, at least by some definition. It is unlikely, however, that the pixels that appear in the portion of the new image shot in front of the green screen will have a color value exactly equal to the color `"Green"`. Instead, they will have pixel values that lie in a range of colors that appear to be "mostly green." For this part of the assignment, you should treat a pixel as green if its green component is at least twice as large as the maximum of its red and blue components.

It is not necessary for the old image and the new image to have the same size. Your program should assume that the upper left corners of the two images are at the same place and update only those pixels whose coordinates exist in both images. The final image, however, should be the same size as the original, and not the overlay. If the images are the same size—as they are for `MillenniumFalcon.png` and `ReyGreenScreen.png` in the images folder—the overlay operation will include every pixel in the image.

### Milestone #5—Implement the Equalize button

Digital processing can do an amazing job of enhancing a photograph. Consider, for example, the countryside image at the left in Figure 4 at the top of the next page. Particularly when you compare it to the enhanced version on the right, the picture on the left seems hazy. The enhanced version is the result of applying an algorithm called ***histogram equalization,*** which spreads out the intensities shown in the picture to increase its effective contrast and make it easier to identify individual features.

As described in section 7.7, the individual pixels in an image are represented using four single-byte values, one for the transparency of the image and three representing the intensity of the red, green, and blue components of the color. The human eye perceives some colors as brighter than others, much in the same way that it perceives tones of certain

**Figure 4. Before-and-after images illustrating histogram equalization**

*Image source:* `http://en.wikipedia.org/wiki/File:Unequalized_Hawkes_Bay_NZ.jpg`

frequencies as louder than others. The color green, for example, appears brighter than either red or blue.

### Luminance

The concept of brightness can be formalized using the idea of ***luminance,*** as described on page 256 in the book. That idea is implemented as a `luminance` function, which is defined in the `GrayscaleImage` module. The value returned by `luminance` is an integer between 0 and 255, just as the intensity values for red, green, and blue are. A luminance of 0 indicates black, a luminance of 255 indicates white, and any other color falls somewhere in between.

The histogram-equalization algorithm you need to write for this assignment uses luminosities rather than colors and therefore produces a grayscale image, much as you did when you implemented the **Grayscale** button. The process requires several steps, each of which is best coded as a helper method as described in the sections that follow.

### Milestone 5a. Calculate the image histogram

Given an image, there may be multiple different pixels that all have the same luminance. An ***image histogram*** is a representation of the distribution of luminance throughout that image. Specifically, the histogram is an array of 256 integers—one for each possible luminance—where each entry in the array represents the number of pixels in the image with that luminance. For example, the entry at index 0 of the array represents the number of pixels in the image with luminance 0, the entry at index 1 represents the number of pixels in the image with luminance 1, and so on.

Looking at an image's histogram tells you a lot about the distribution of brightness throughout the image. The example at the top of Figure 5, for example, shows the original low-contrast picture of the countryside, along with its image histogram. The bottom row shows an image and histogram for a high-contrast picture. Images with low contrast tend to have histograms more tightly clustered around a small number of values, while images with higher contrast tend to have histograms that are more spread out throughout the full possible range of values.

**Figure 5. Image histograms for a low-contrast and a high-contrast image**

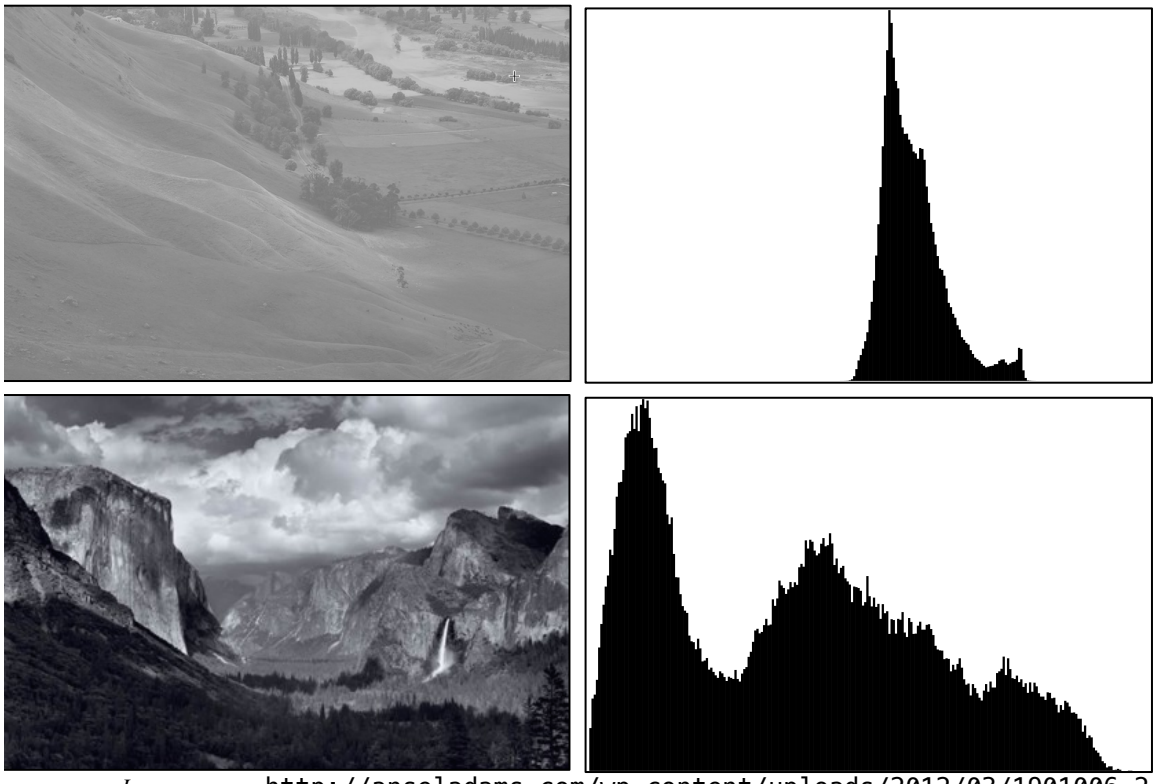


*Image source:* http://anseladams.com/wp-content/uploads/2012/03/1901006-2-

The point of making this intermediate subgoal into a milestone is to emphasize that you need to test it, even if it isn't easy to see the result. The histogram array will have 256 elements. You can either print the histogram on the console or use the graphical histogram function from Problem Set #5 to check whether your distribution matches these graphs.

**Milestone 5b. Calculate the cumulative histogram**

Related to the image histogram is the ***cumulative histogram,*** which shows not simply how many pixels have a particular luminance but rather the sum of all values at this luminance or below. Like the image histogram, the cumulative histogram is an array of 256 values—one for each possible value of the luminance. The cumulative histogram is computed purely from the image histogram. Each entry in the cumulative histogram is the sum of all entries in the image histogram up to and including that index position.

As an example, if the first six entries of the image histogram are

1, 3, 5, 7, 9, 11

the corresponding entries in the cumulative histogram would then be

1, 1 + 3, 1 + 3 + 5, 1 + 3 + 5 + 7, 1 + 3 + 5 + 7 + 9, 1 + 3 + 5 + 7 + 9 + 11

or

1, 4, 9, 16, 25, 36

**Figure 6. Cumulative histograms for the images from Figure 5**

Figure 6 shows the cumulative histograms for the two images from Figure 5. Notice how the low-contrast image has a sharp transition in its cumulative histogram, while the normal-contrast image tends to have a smoother increase over time.

**Milestone 5c. Implement the histogram-equalization algorithm**

The cumulative histogram provides just what you need for the histogram-equalization algorithm. To get a sense of how it works, it helps to start with an example. Suppose that you have a pixel in the original image whose luminance is 106. Since the maximum possible luminance for a pixel is 255, this means that the "relative" luminance of this pixel is 106 / 255 ≈ 41.5 percent, which means that this pixel's luminance is roughly 41.5 percent of the maximum possible. If you assume that all intensities are distributed uniformly throughout the image, you would expect this pixel to have a brightness that is greater than 41.5 percent of the pixels in the image.

Similarly, suppose that you find a pixel in the original image whose luminance is 222. The relative luminance of this pixel is 222 / 255 ≈ 87.1 percent, so we would expect that (in a uniform distribution of intensities) that this pixel would be brighter than 87.1 percent of the pixels in the image.

The histogram equalization algorithm works by trying to change the intensities of the pixels in the original image as follows: if a pixel is supposed to be brighter than $X$ percent of the total pixels in the image, then the algorithm maps it to a luminance that will make it brighter than as close to $X$ percent of the total pixels as possible. Making this process work turns out to be much easier than it might seem, especially if you have the cumulative histogram for the image.

Here's the key idea behind the algorithm. Suppose that an original pixel in the image has luminance $L$. If you look up the $L^{th}$ entry in the cumulative histogram for the image, you will get back the total number of pixels in the image that have luminance $L$ or less. You can then convert this into a fraction of pixels in the image with luminance $L$ or less by dividing by the total number of pixels in the image.

Once you have the fraction of pixels with intensities less than or equal to the current luminance, you can scale this number (which is currently between 0 and 1) so that it is between 0 and 255, which produces a valid luminance. The histogram equalization algorithm therefore consists of the following steps:

1. Compute the histogram for the original image.
2. Compute the cumulative histogram from the image histogram.
3. Replace each luminance value in the original image using the formula

$$\textit{new luminance} = \frac{255 \times \textit{cumulative histogram}[L]}{\textit{total number of pixels}}$$

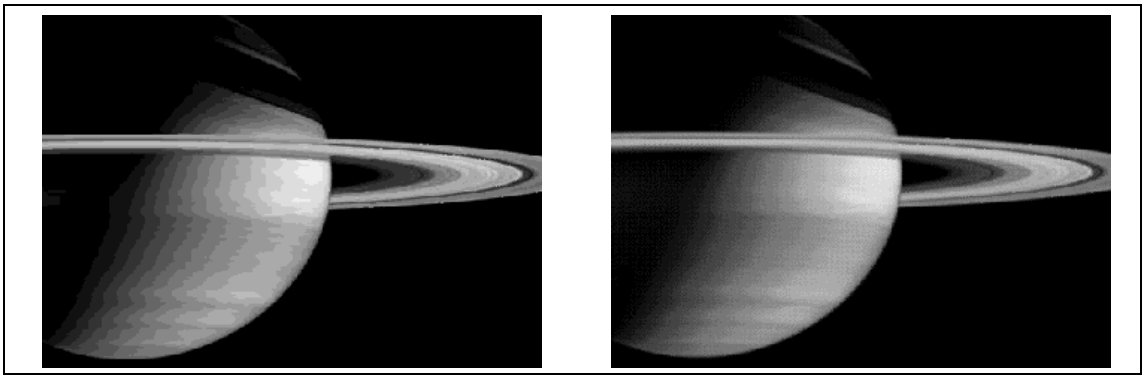
Remember that you are writing Milestone #5 as a sequence of three steps. If you test as you go, life will be much easier.

**Extensions**

This assignment offers essentially unlimited possibilities for extensions. All you need to do is implement features from your favorite image editor. Here are a few ideas:


- *Implement a "posterize" button.* Shepard Fairey's iconic design of the campaign poster for President Obama's 2008 campaign was widely adapted for other drawings. In this image, all pixels are converted to the closest equivalent chosen from a restricted set of colors. This image, for example, contains only red, an off-white ivory tone, and three shades of blue. Your application could, for example, replace all intermediate colors with the closest match in Java's predefined color palette or use some other strategy that you find by searching the web or that you dream up on your own.
- *Implement an averaging filter.* When using low-resolution digital cameras, images can look rather blotchy. Figure 7, for example, shows two images of Saturn taken by the

**Figure 7. Images of Saturn before and after smoothing**



Cassini probe. You can create images like the one on the right by replacing each pixel with a weighted average of its own luminance and that of its nearest neighbors. You can add a button to your ImageShop program that performs this sort of average.

- *Add a touch-up tool.* If you need to edit an image, it is particularly useful to have a pencil-like tool that allows you to drop a new color on any pixel in the image. The usual strategy is to allow the user to pick a color first and then change individual pixels to that color by clicking on them with the mouse.
- *Implement a crop box.* In the basic assignment, the mouse is used only for buttons, but you could also use it to draw a rectangle on the screen and then limit the functions of the other operators to the region inside the rectangle. It also makes sense to add a **Crop** button that eliminates all pixels outside the crop box. For example, if you load the image `DaVinci-LastSupper.png` and then use the mouse to draw a box as shown in Figure 8, clicking the **Crop** button should delete all the pixels outside of the box to leave only the following image on the screen:

If you saw the movie version of Dan Brown's novel *The Da Vinci Code,* you may remember that Ian McKellen's character used a cropping tool to highlight this image.

- *Whatever else you want.* Go wild! We should get a few ++ scores on this assignment.

**Figure 8. The ImageShop screen before hitting the Crop button**