**Partial Image Placement Software (PIPS)**

By: Bradley Miller, Matt Nicol and James Merrill

**Abstract**

A fluid object to picture placement and blending method is something many people could use in their daily lives. With how expensive Photoshop and other alternatives are, and the difficulty of using them, having a program that can do everything from cutting out the object in the image to placing it a second image and blending it is something worth having.

**Motivation**  
The motivation for this project was to have a challenging project so we could learn to use matlab more efficiently. We wanted an idea that would be a cool project but also something that we could use ideas from throughout the semester to help guide us in making the project.

**Approach**

The approach we used to implement this project was quite simple. First we had a basic user interface where the user could select an image. The user would then draw a box around the image to represent what wanted to be cut out. Then the user picks a pixel on the second image where the blended image should be placed. The UI then calls the matlab function to segment the image and the blend it into the new image. Below are the three sections of the project in more detail.

**Image Selection**

The first part of the project is to select part of an Image so that it can be refined and placed into another image. From our initial research creating a UI in MATLAB was not feasible for what we wanted in our application so we created a user interface using java. The code for the java program can be found in Paint.java and DrawArea.java. Paint.java is the main application that runs and DrawArea.java is a helper class. The overall application is around 500 lines of code and uses the java swing library for creating the user interface. The program starts by prompting the user to input two images. The first image will be the one you select from and the second will be the image that you place into. This can be seen in figure 1 below.

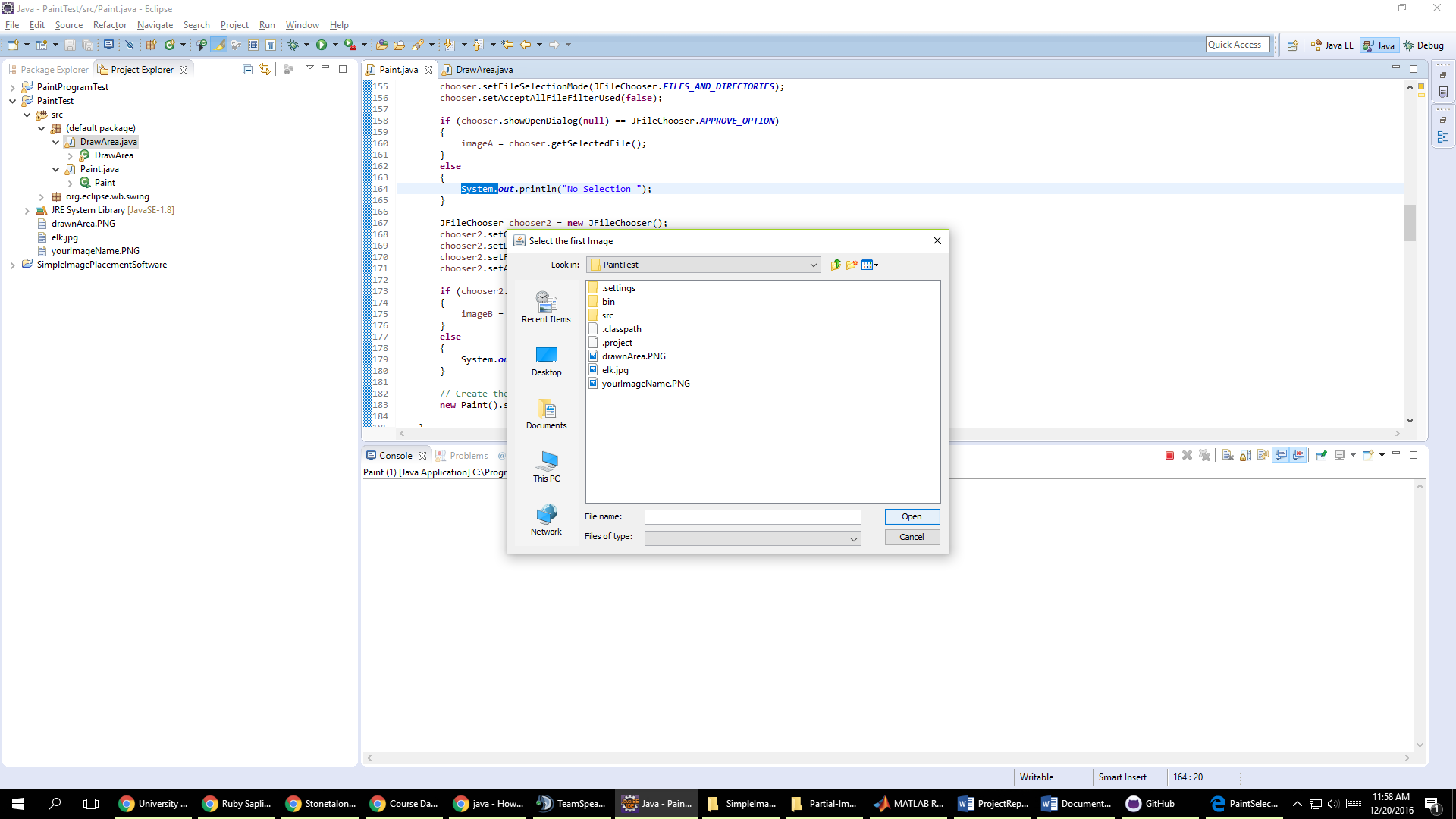


Figure 1: Selecting an image

After the user selects the two images that they want to use a new window pops up for the main user interface as shown in figure 2. This interface is how the user will select portions of the image. Due to difficulty and complications we were not able to integrate the java application directly into the MATLAB functions. The java application does output all the needed information for the MATLAB functions through the command line.

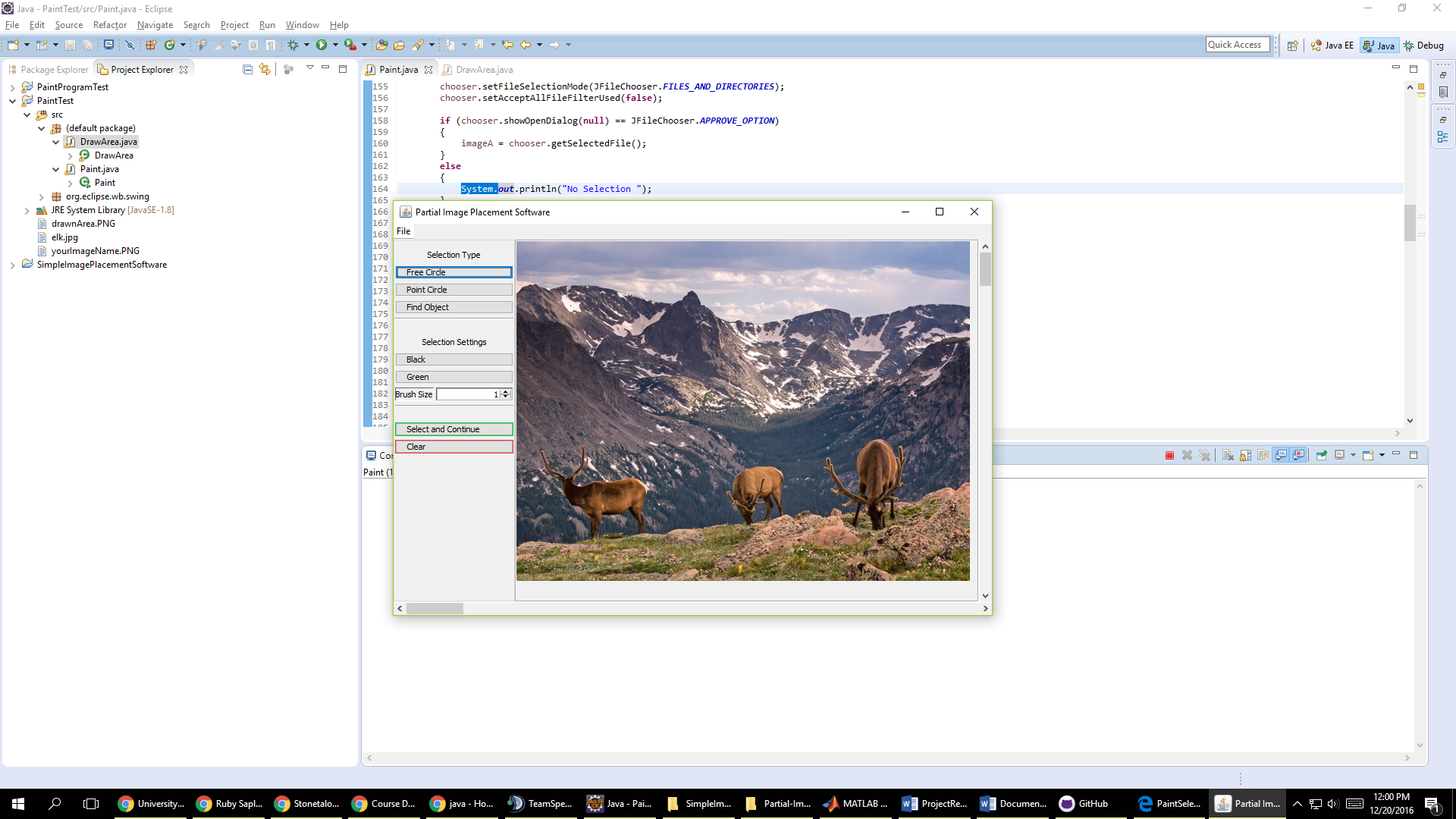


Figure 2: The main user interface

Our initial goals for image selection were to implement lazy snapping or a paint selection method. Lazy snapping, as outlined in a paper by Li, Yin, et al called Lazy Snapping published in ACM Transactions on Graphics, allows the user to draw a line around an object and have the line snap to the object. The second method we wanted to implement was paint selection, as outlined in a paper by Jiangyu Liu, et al, which allows a user to draw inside of objects that they wanted to select. However due to time constraints and difficulties in working with images in java we moved to allowing the user to select an area with a box that will later be refined by the automatic cut out algorithm. The box can be seen in figure 3 and the ability to freely draw around images can be seen in figure 4.

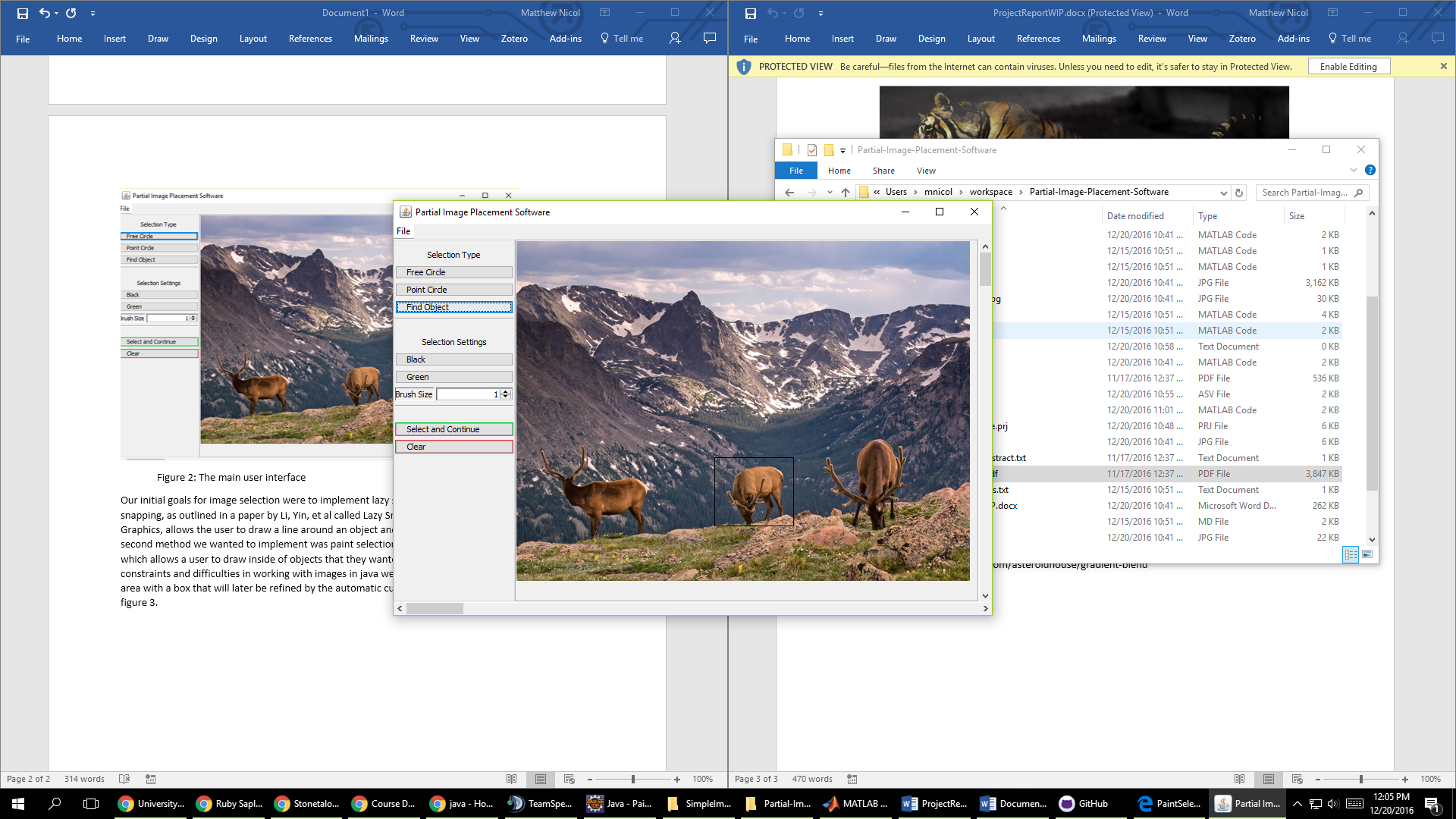


Figure 3: User placed box around an object

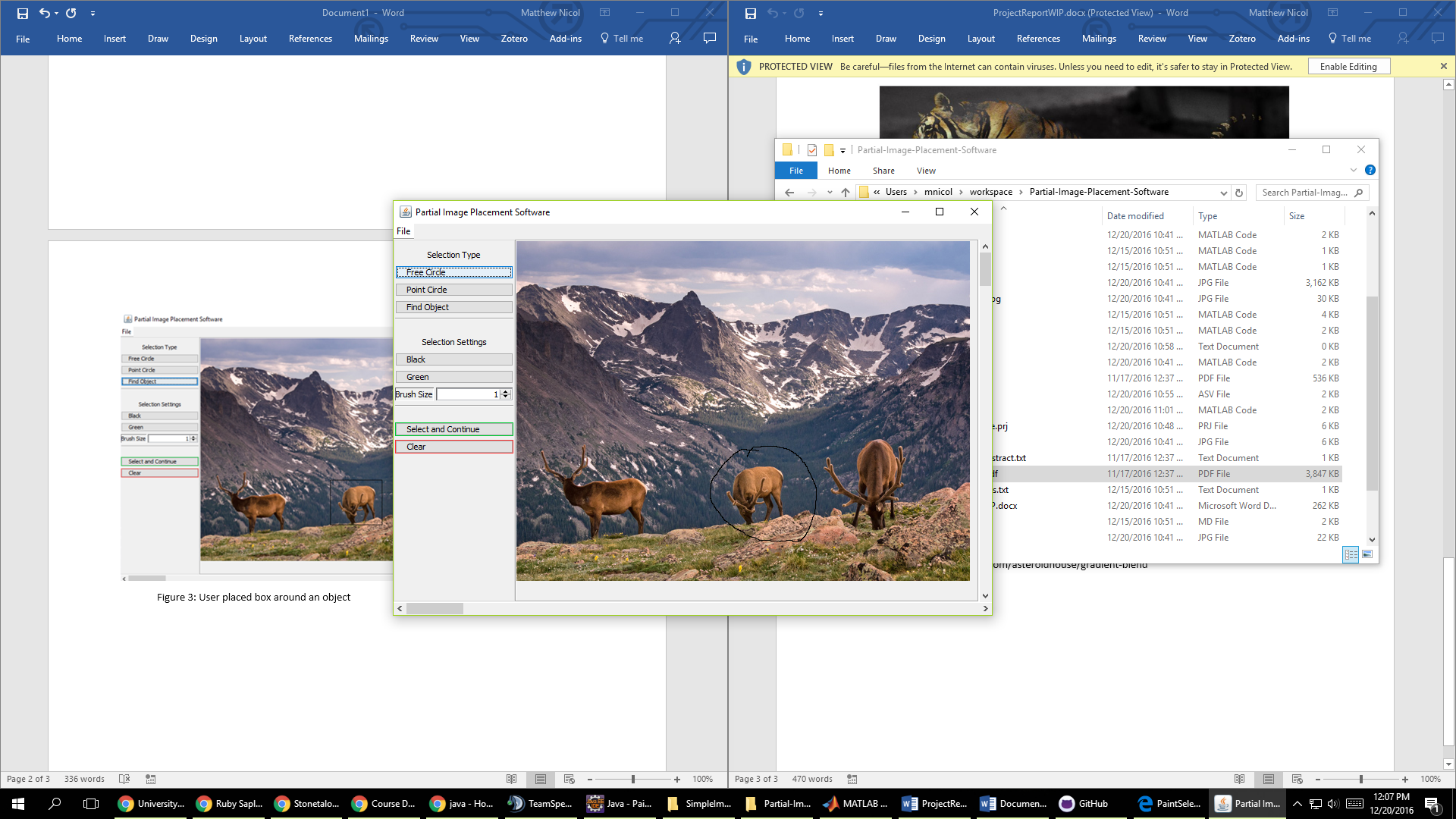


Figure 4: Freely drawn circle around object

The resulting information for these actions can be found as outputs in the command line as shown in figure 5. The information that is output is where the user clicked which will be needed for placing the image, and the upper left corner and bottom right corner of the rectangle to select the area for cutting. You can also save the drawn area of the image with a blank background to be processed later as seen in figure 6.

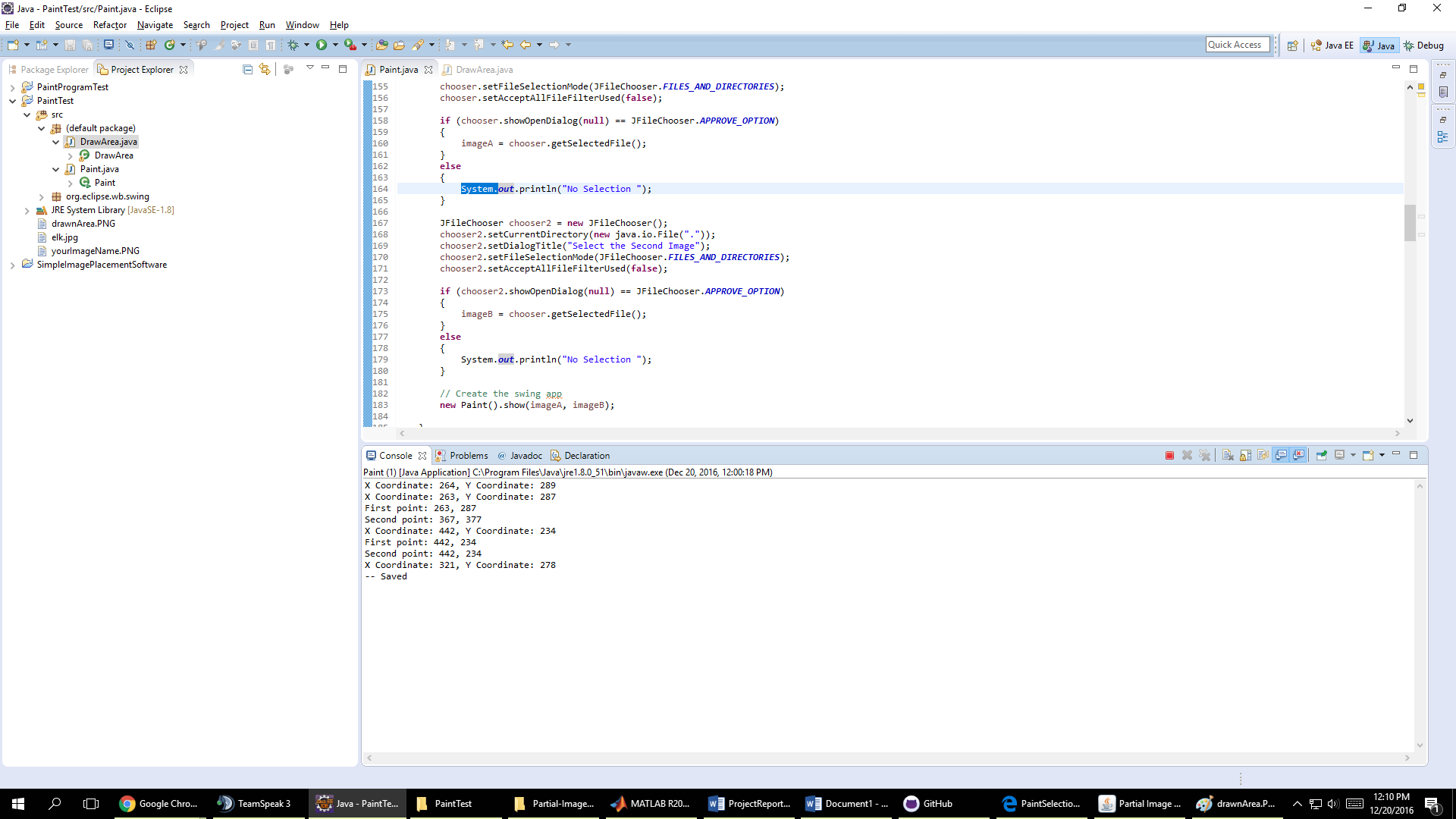


Figure 5: The coordinate outputs from using the program

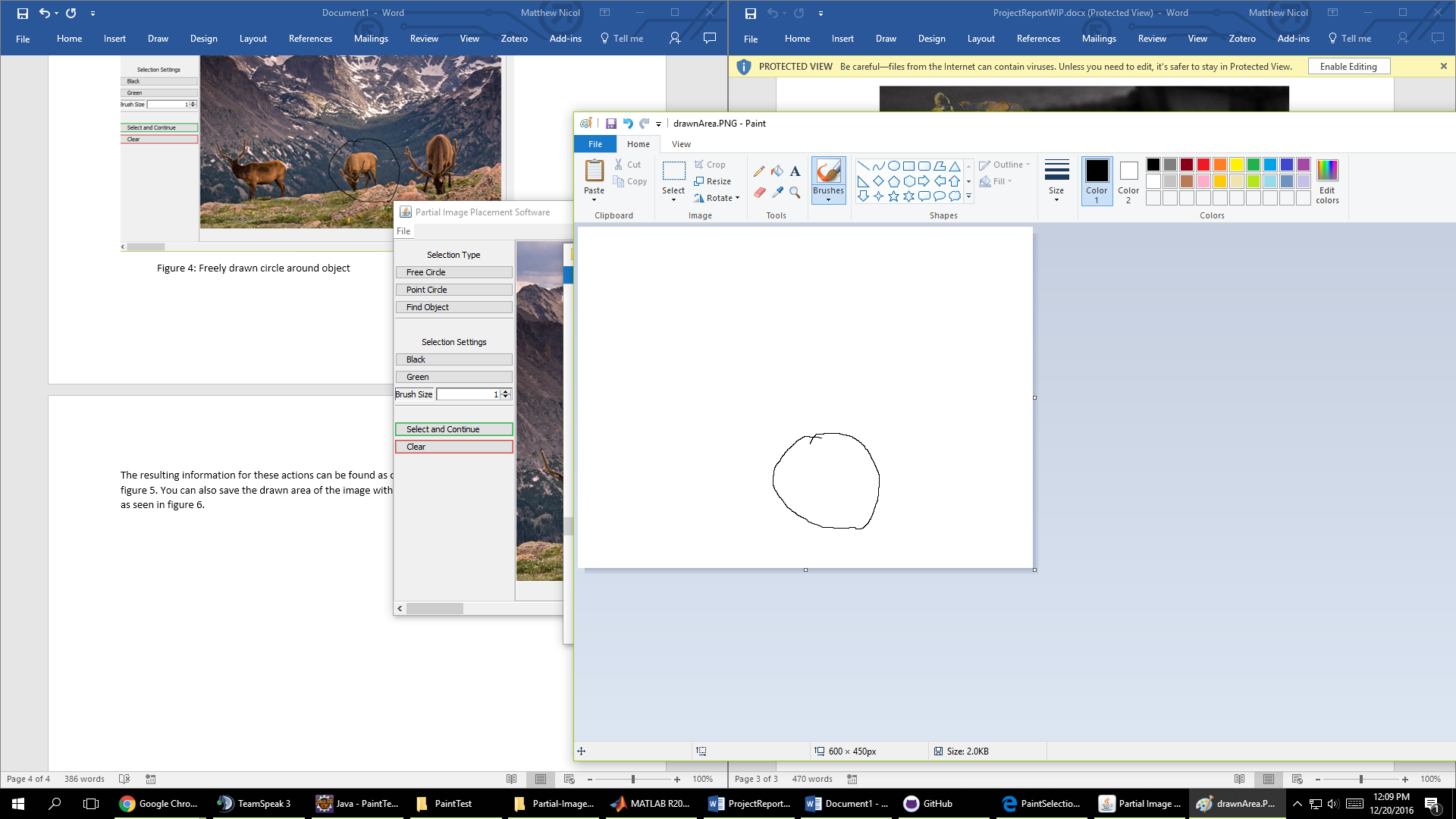
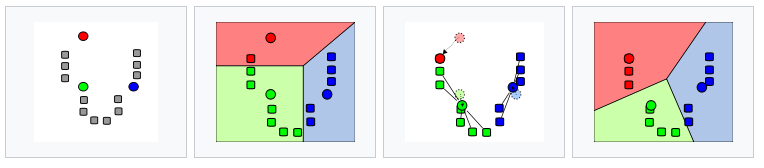


Figure 6: The drawn area from figure 4

**Segmentation**

To segment the picture there were quite a few different options that were good options. The first choice that was explored was K-Means. K-Means is a way to segment the image based on color association. To begin with you can choose a “k” value which signifies how many different colors you cluster the image into. As you can see below this is a very basic examples that segments the colors of the image into 3 different “k” values.

(<https://en.wikipedia.org/wiki/K-means_clustering>)

A way to improve upon this algorithm is to use a lab color space which stands for Lightness, with a representing green and red values at negative and positive values respectively and b represetngin blue and yellow at negative and positive values repsetively.

This method work decently well and produced examples like this:



Now obviously these results weren’t ideal but they did a decent job at selecting the object from the frame. The biggest issue with this is that it cut a bit too much out of the image.

The next method I attempted to implement was similar to a graph cut. The idea behind this method was to split the image into a foreground and background and then compare the pixels in the foreground that are neighbors to a background pixel. To do this I compared their color to see if they were close enough to be considered part of the same object. If they were then I added them to the background. I did this for all the pixels in the image and then continued to do it until there were no changes for any of the pixels. This result was not as good as I originally thought it would be. I could’ve implemented a method that removes the pixels that don’t connect to the main image, but I didn’t think that was worth it consider the image wasn’t entirely what I was looking for.





The final method I wrote and tested was a method was a mixture of both of the previous methods. Since we had the user drawing a box around the object we can assign a foreground and background object fairly easily. Due to this we can implement a method where we take the background pixels and use matlab’s built in k-means function on them to determine what the background pixels colors are. We can then do the same to the foreground pixels. After we have the center of each of these we can construct a method that compares each of these centers to see if they are close enough to be considered the same or similar color. The idea behind this is if we have a distinct object in the foreground we are trying to cut out that does not share colors with what’s in the background then the K-means comparison should never select that object colors are part of the selection process. We can run this a few times to get a better cut out of the image. Here are a few examples I did that showcase this methods strengths and weaknesses. As you can see from the images below that the method works pretty well for cutting out the object, however when the object has colors from the background in it seems to also cut those out or if the object has similar colors that blend with some background objects, like the fire hydrant and the red sidewalk it seems to leave those in. So the method isn’t perfect but works for sufficiently for what we need for this project.







**Blending**

We implemented a Poisson Blend as described in *Pérez et al.* to combine our target and source images. For our purposes, the Poisson Blend simplifies to: for each pixel in our blended region, the result of a Laplacian operator on that pixel should be equal to the Laplacian if the corresponding pixel in the source region. Thus, the gradient across the inserted object should be preserved as well as the gradient between that object and its original background.

We are able to calculate these gradients across each color channel using a system of linear equations. Each variable in the system of equations represents one pixel in a rectangle which contains the source object placed over the target background. Using a binary mask we obtained from our image segmentation, we determine which pixels will be taken directly from the background and which pixels are from the source object and need to be blended.

* For each blended section pixel *x*
  + If pixel is background: ***x* = *t***, where *t* is the corresponding pixel in the target image.
  + If pixel is blended object: **laplacian(*x*) = laplacian(*s*)**, where *s* is the corresponding pixel in the source image.

We are able to solve this quite large system of equations using matrix division. ***A\b = x*** where *A* is the matrix that represents the left side of the equations above and b is the vector that represents the right side. Since the matrices involve get to be quite large, but have at most five non-zero values in any given row, we create them using the sparse() command in Matlab. This stores the zero values in our matrices more efficiently and helps our blender run in a reasonable amount of time (as well as prevents it from crashing James’ computer if used incorrectly).

**Blending Results**

There are some circumstances where the blending does not produce seamless results as intended. For instance, if the source object is segmented such that there is too much of its original background blended, the result image will have sort of a blurry halo around the source object which is especially noticeable if the target background image has complex patterns. 

There is a blurry halo around Bucky's head since too much of the source was selected. The underwater plants make this blur much more noticeable.

Also, if something goes wrong with segmentation and the segmented object has holes in it, the holes will be filled by the background, and the object will appear semi-transparent. 

While this segmented tiger looks nice on its own, it will actually only select the orange parts of the tiger.



The tiger is a little more transparent than we intended.

**Conclusion**

In conclusion the methods we choose to do this are the correct ones. Our implementation worked fairly well and as you can see from the picture below, looks fairly good. We had a bit of issue calling the actual matlab function from the GUI which was an issue and if we continued to work on this then we would work towards fixing that problem. The segmentation worked well but sometimes produces a poor image when the background mixes into the object that’s going to be cut. Overall the project as a whole works fairly well together but could use a bit of refining.

****

**Citiation**

* Poisson Image Editing by Patrick Pérez, Michel Gangnet and Andrew Blake

**Our Code**

GUI ~ 400 lines of code – Matt’s Part

Segmentation ~ 200 lines of code – Brad’s Part

Poisson Blender code – Roughly 80 lines. – James’ Part

Total = About 700-800 lines of code

Poisson Implementation took Inspiration from:

* <http://cs.brown.edu/courses/cs129/asgn/proj2/>
* http://eric-yuan.me/poisson-blending/
* <https://github.com/asteroidhouse/gradient-blend>

**Team Contribution**

Matt wrote the code for the GUI, Brad wrote the code for Segmentation, and James wrote the code for the blending method.