Infinite Photo Collage

A Major Qualifying Project submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science in Computer Science and Degree of Bachelor of Science in Data Science

Liliana Foucault, Frank McShan, Molly Sunray

Advised by Professor Jacob Whitehill

December 16, 2022

Abstract

The Infinite Photo Collage explores zooming into the center of an image infinitely. Once the image is zoomed into past a certain threshold, the image is replaced by a collage of images, chosen to match sections of pixels of the original image based on their average color. This process can repeat infinitely. This was accomplished using a program we wrote using Java Swing.

Table of Contents

Abstract 2

Table of Contents 3

Introduction 4

Background 5
- Computational Art Examples 5
- Similar Projects 8

Specification 11
- Terminology 11

Implementation 16
- JavaFX Versus Java Swing 16
- Loading Images 17
- Development Process 18
  - Initial Version 18
  - Image Cache 21
  - Cropping to the Size of the Window 23
- User Interface 24
- Computational Challenges 25

Evaluation of Progress and Next Steps 27
- Assessment 27
- Unexplored Ideas 27

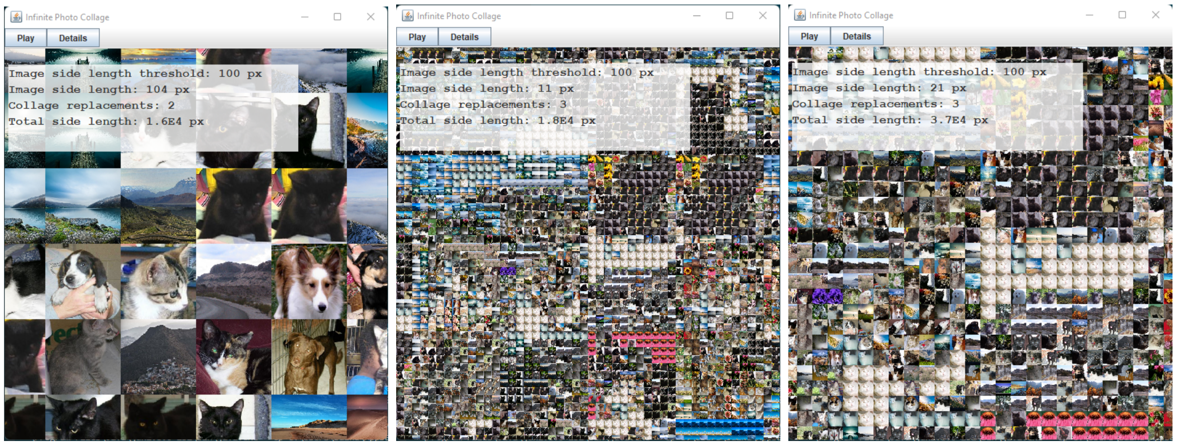
Conclusions 29

References 31

Introduction

This project is a Computer Science and Data Science Major Qualifying Project that explores one implementation of computational art. Computational art uses a computer to generate artwork. A collage of images is repeatedly computed after zooming into an image with the result being an Infinite Photo Collage. To understand the impact of displaying many images on screen, the runtime cost associated with loading images was analyzed and improvements were made where possible.

The idea behind the project is that users begin with an image, and as they zoom in to a certain threshold, pixels of the image are replaced with other images. The result is a collage-like image that still ultimately reflects the look of the original image. If they choose, users can continue zooming in on the collage, and as the new images become closer in view, they too will be replaced with other images. Users can zoom infinitely into the collage, allowing for an infinite repetition of image replacement to take place. Figure 1 showcases an example of this process of collage replacement and zooming.



```
Image side length threshold: 100 px
Image side length: 104 px
Collage replacements: 2
Total side length: 1.6E4 px
```

```
Image side length threshold: 100 px
Image side length: 11 px
Collage replacements: 3
Total side length: 1.8E4 px
```

```
Image side length threshold: 100 px
Image side length: 21 px
Collage replacements: 3
Total side length: 3.7E4 px
```

Figure 1: Images being replaced with collages of images then zoomed into.

Although quite challenging to understand programmatically and in the context of implementation, our team wanted to take on this challenge. The project itself piqued our interest immediately when envisioning how it would work.

Background

The implementation of our project is encompassed by the greater context of computational art. Computational art combines art and computer programming, using programming and computation as the medium (Bright, 2019). Analyzing projects with similar functionalities to our project allowed for a better understanding and visual of how the project should behave. Examples that were different from our project also allowed us to explore what exists in the realm of computational art.

Computational Art Examples

Computational art has been around for many years, although the forms it has taken have evolved and expanded. An early example is Georg Nees' "Schotter," an image showing orderly squares descending into disorder created in 1968. Nees, who is viewed as one of the founders of computer art, created a program that introduced random variables to draw the geometric pattern shown in Figure 2 using a drawing machine (Victoria and Albert Museum, 2009).

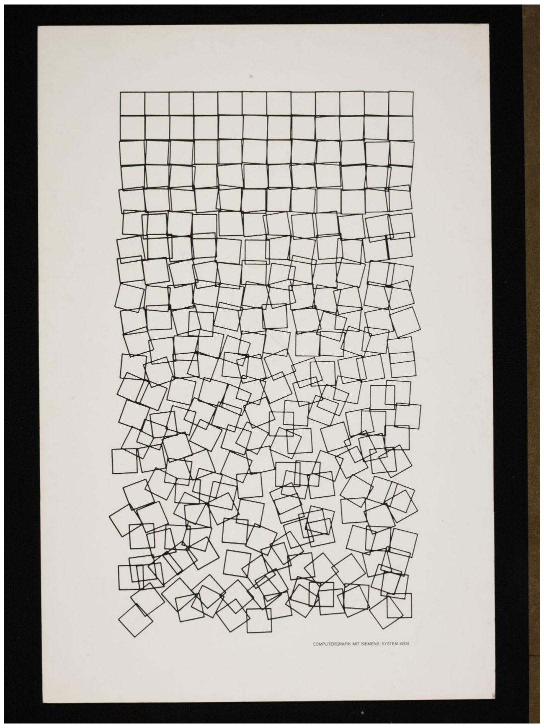


Figure 2: "Schotter," a geometric pattern created by Georg Nees.

Vera Molnar also used randomization in her drawing “Structure of Squares” created in 1974. This drawing, shown in Figure 3, consists of several groups of squares where the number of squares in each group was randomized (Victoria and Albert Museum, 2011).

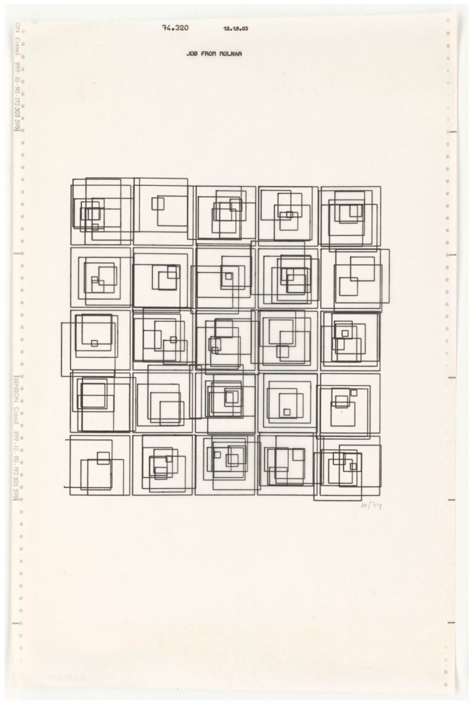


Figure 3: “Structure of Squares,” created by Vera Molnar.

More recently, many computational art approaches use artificial intelligence to create art from algorithms. One earlier example is Harold Cohen’s artificial intelligence system AARON, which has been in development since the 1970s. At first, the system was only capable of producing drawings in black and white so Cohen often colored them by hand. A small robot used a marker to create the drawings on paper. He later developed a version that could perform the coloring itself and in a manner that was almost identical to Cohen’s coloring in earlier drawings. Some of the drawings produced by AARON were displayed in museums such as the Computer Museum in Boston and the San Francisco Museum of Modern Art (Garcia, 2016).

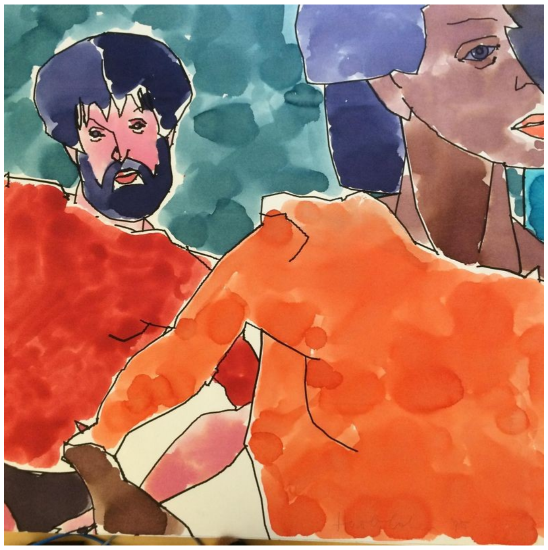


Figure 4: A drawing created and colored by AARON.

Another example is La Famille de Belamy, a series of eleven portraits created by the art collective called *Obvious*. The portraits represent aristocrats from different time periods and were created using generative adversarial networks (GANs), which are generative models that perform training by competing two algorithms against each other (Obvious, n.d.).

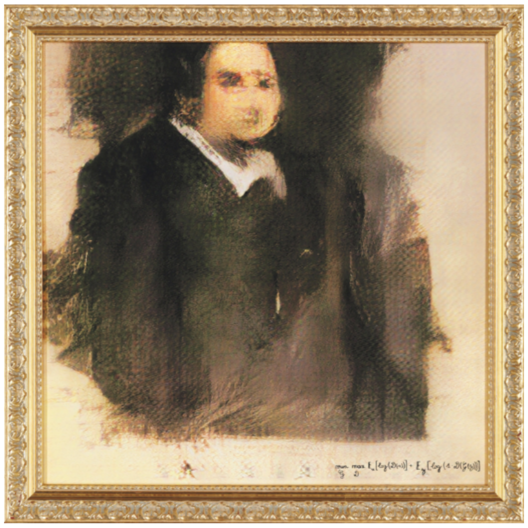


Figure 5: “Edmond de Belamy,” one of the portraits from La Famille de Belamy created by *Obvious*.

Recently, artificial intelligence has taken over social media through Lensa AI, a popular iPhone app that uses selfies to produce computer-generated portraits in a variety of styles. The app uses Stable Diffusion, which is an image generator that uses image prompts and text prompts to produce high-quality images. Lensa trains Stable Diffusion using artwork from existing artists, creating concerns among them and others (Kircher & Holtermann, 2022).

Similar Projects

Several projects were found that presented similar functionalities to our project. The first example is a project that uses the idea of infinite zooming and DALL-E, an artificial intelligence tool that creates images based on natural language descriptions (Osinga, 2022). The result is an "infinite zoom movie" that zooms into a single image, displaying more details of specific parts of the image once they come into view. While described as infinite, the zooming stops after a certain point and loops back to the starting image. This project does not include image replacement based on pixel colors. The images used to create this movie are shown in Figure 6.

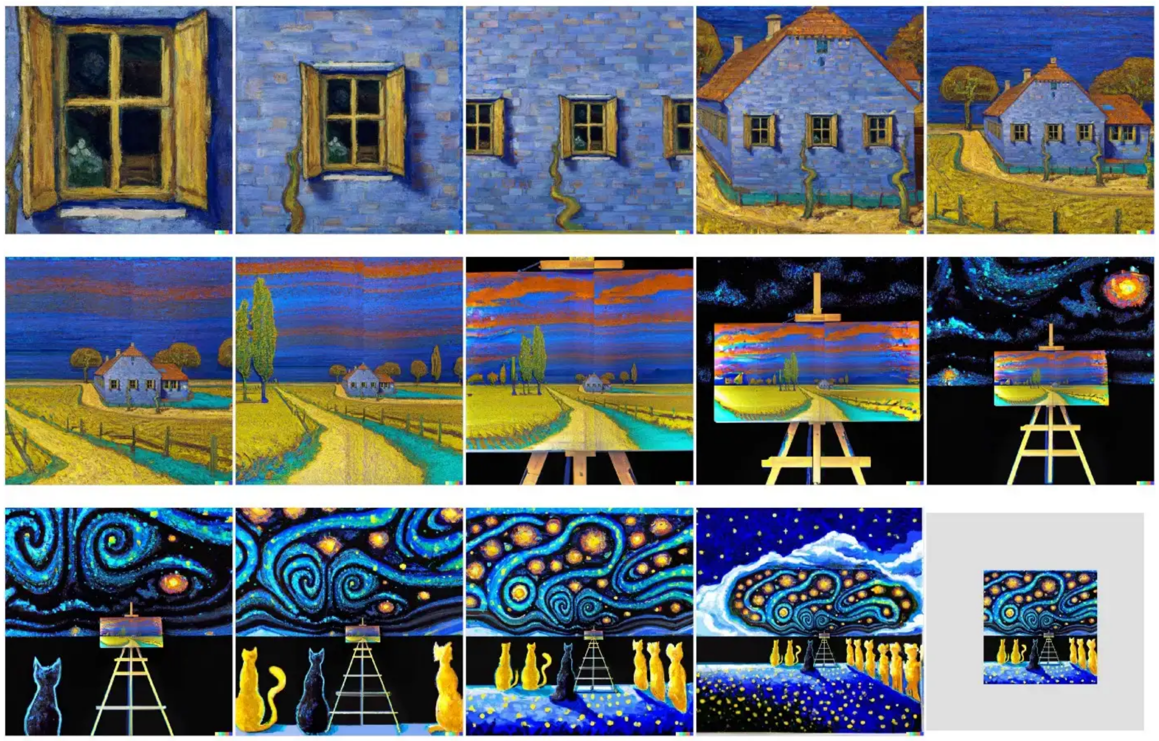


Figure 6: Frames used to create an "infinite zoom movie" from DALL-E.

Another project found publicly on GitHub uses image replacement but not zooming by generating a photomosaic image (Dawson, 2022). This program takes an image to be transformed into a mosaic and a directory of images to use as tiles for the mosaic. The mosaic tiles can be matched to the original image based on the average color of each tile, which is similar to our implementation for color matching. However, this transformation only happens once, allowing for a single level of image replacement with no infinite zoom. An example is shown in Figure 7. There is no UI for running this project but changing variables in the source code influences the outcome resolution and overall computing time. These three variables are tile size, enlargement factor, and tile match resolution. Tile size is referred to as the side length in pixels of one image in the resulting mosaic. Enlargement factor is described as the factor of how much larger the resulting width and height of the mosaic is. Tile match resolution is not explicit in units, but describes how well of a resolution the images per tile are rendered; a larger input results in better resolution but a longer runtime. Because the tile size and enlargement factor can be altered, the resolution of the final mosaic in images is not always equal to the resolution of the original image in pixels. The consistent factor is the width and height ratio between the original and final mosaic.

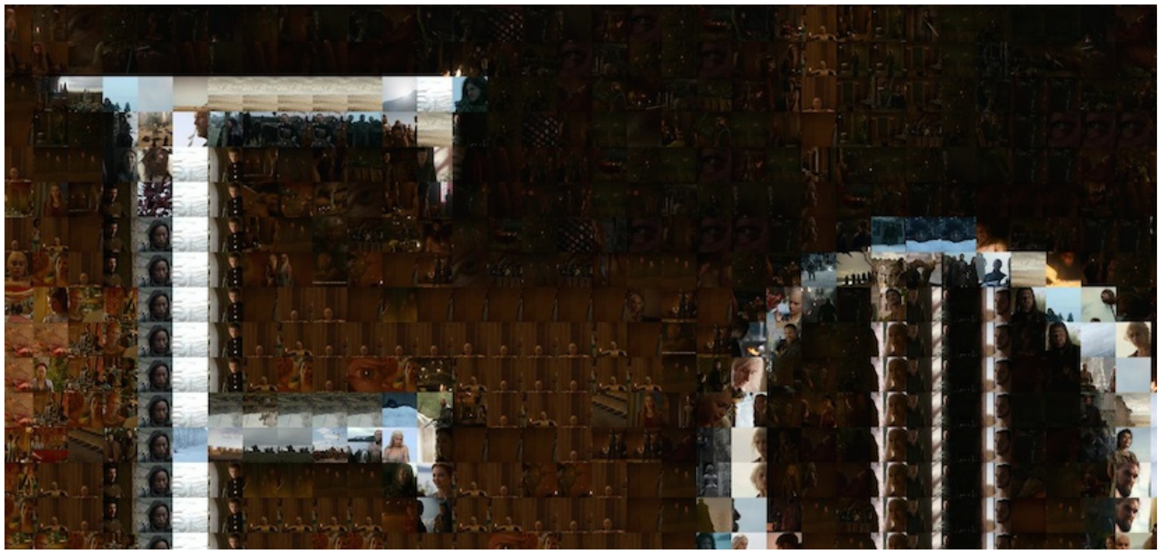


Figure 7: An image transformed into a mosaic created by Dawson.

Lastly, another project was found that performs mosaic transformation in addition to infinite zooming (Furstenheim, 2020). The project gives the user control over zooming in, zooming out, and panning, and can alternatively be viewed on autoplay. The image dataset is artwork from the Art Institute Chicago. When the project is viewed in the browser, a random image is selected as the starting image and occupies the full view window. The image is displayed just past the threshold of image replacement so each pixel has been replaced with an image with the average color of the corresponding pixel. When the user zooms into one of these pixel areas so that a new image is in full view and takes up almost the entire view window, the image is about to reach the threshold of replacement. This threshold is the image size being the size of the view window. Figure 8 includes the image after this threshold when the pixel areas are replaced with images. However, if the user is not zooming into the center of a single image, the user will see the replacement for multiple images occurring at this same threshold.

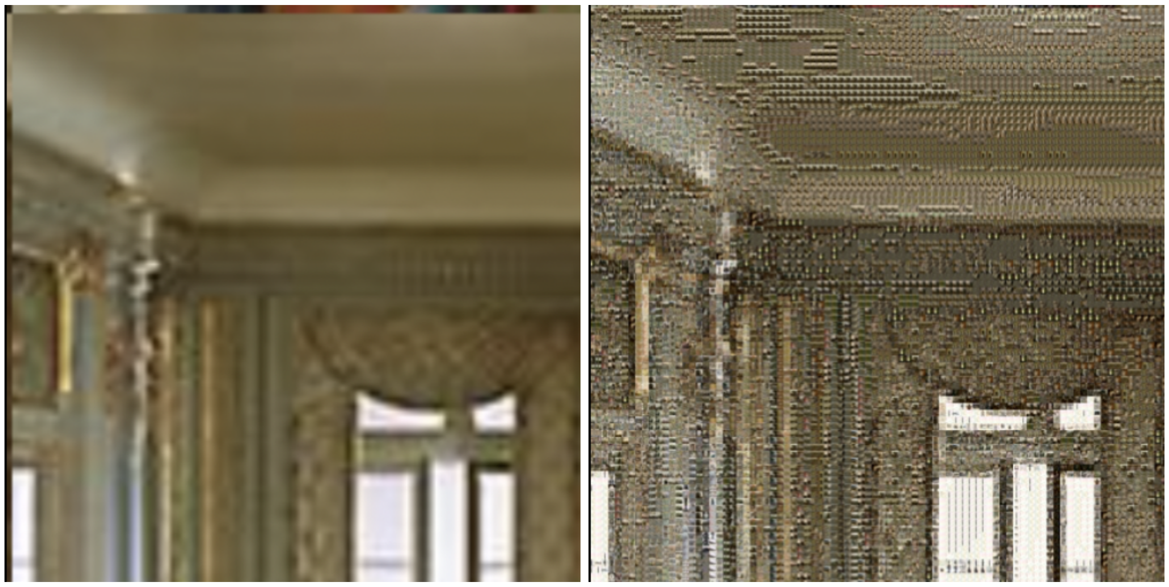


Figure 8: A portion of an image before (left) and after (right) the threshold of replacement from Furstenheim's application.

Specification

Our project revolves around two major concepts: zooming into images infinitely and replacing an image with a collage of images. Once a user has chosen a directory of images from their local device, a random selection from the dataset will be the first visible image. Users are then able to infinitely zoom into that image. After a certain amount of zooming into this singular image, it will be replaced with a collage of images. This process repeats and users can continue zooming in infinitely.

Terminology

Multiple terms are used consistententently throughout the project. This portion will define a comprehensive list describing each step of the infinite zoom process.

When a directory of images is selected, the file pathname of each is stored for later reference along with its respective average color. The **average color** of an image is the average of each color channel over every pixel in that image. This is calculated by iterating over each pixel of an image and summing each of the red, green, and blue values individually. These totals are divided by the total number of pixels in the image giving the average value for each color channel. One edge case is if the image contains “outlier” areas of color. One example could be a border around the edges of the image that may skew the calculated value away from what is visible. Instead of using every pixel, only the center pixels of an image were used by ignoring a set percentage of pixels around the edges. This average color, represented as an integer, is calculated then stored as a key mapped to its respective image filename. The initial image will then be displayed for users to interact with.

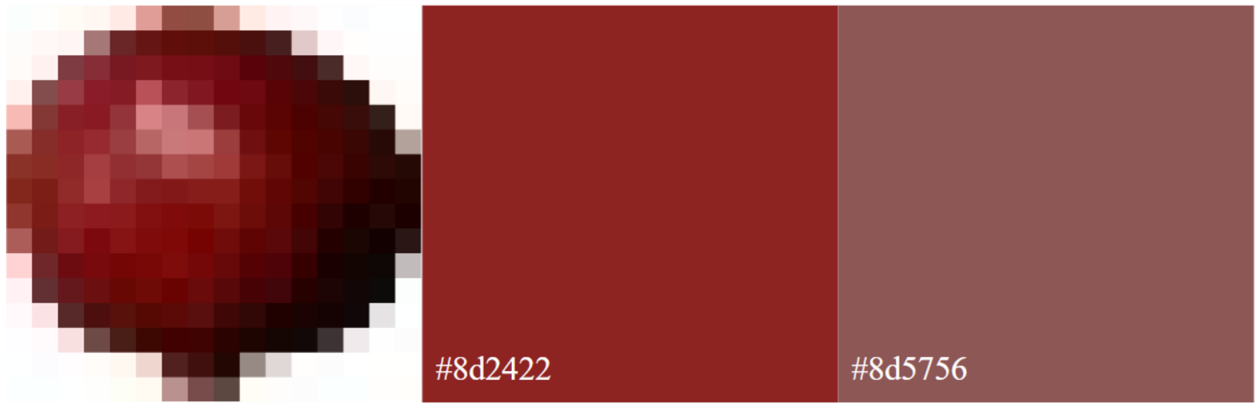


Figure 9: Image (left) and calculated average color ignoring edge pixels (middle) vs. using all pixels (right).

The image can be zoomed into using the spacebar, mouse, or autoplay feature. **Zooming** increases the rendered resolution of the visible image by a constant percent increment. The rendered resolution is therefore multiplied by 1 + the percent increment. The **original resolution** of an image refers to an image's width and height in pixels exactly as it is uploaded by the user whereas the **rendered resolution** of an image refers to an image's width and height in pixels as displayed in the project window. The rendered resolution is stored using the **image side length** since all images are displayed as squares. The space the image(s) occupy inside the window is referred to as the **canvas**. The canvas width or height in pixels at any point is the **total side length**. When this value is larger than the window frame, images not visible are cropped out by calculating the side length in images that can fit inside the area of the window. Cropping out images is necessary to avoid heap space errors and allows zooming to be infinite. This process is described in further detail in later sections. The image side length and the total side length can then be used to calculate the **side length in images** of the canvas which is the total side length divided by the image side length.

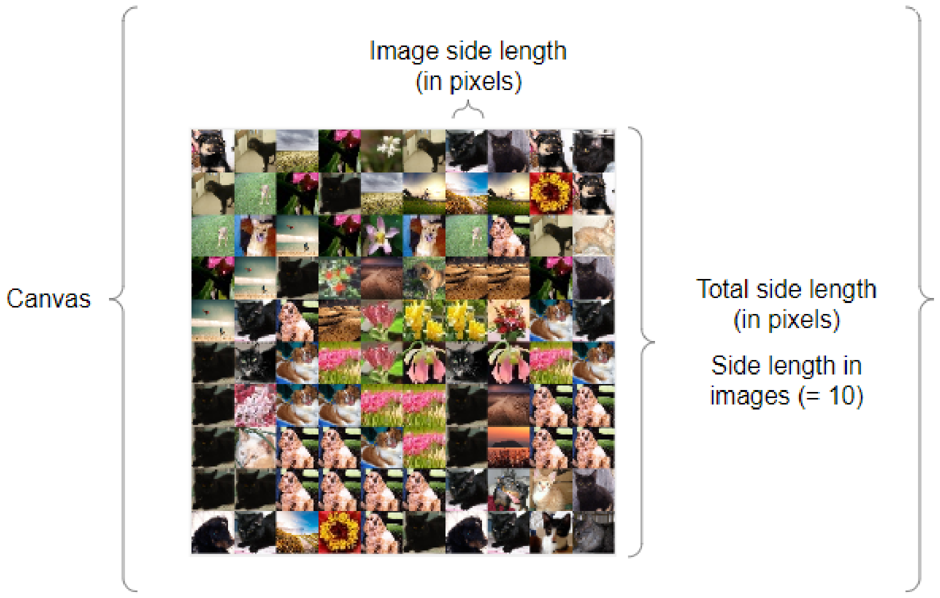


Figure 10: Visualization of image side length, total side length, and side length in images on the canvas.

The **approximate original image side length** is also stored and updated in a similar fashion to the image side length. This value represents the side length in pixels that the starting image would theoretically be rendered at if cropping did not occur. Each time a zoom or collage replacement occurs, this value is updated.

After the image side length is larger than a maximum value, the **replacement threshold**, a collage replacement occurs. A **collage replacement** uses the map of average colors to select a set of images to replace the pixels of some parent image.

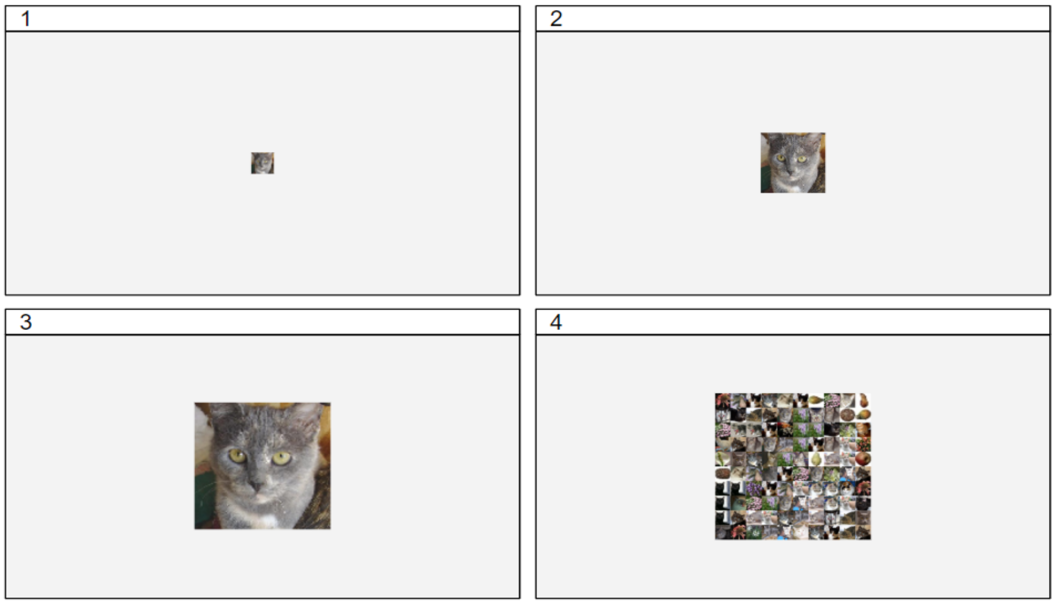


Figure 11: Visualization of first image rendered (1), zoomed into (2, 3), and replaced with collage of images (4) in the project window.

For each pixel of some parent image, the average color map is searched through for the key with the smallest difference to the target pixel color. This equals the smallest sum of the difference of each of the RGB channels of the key compared to the target color; the absolute value of the sum of (target red - image red) + (target green - image green) + (target blue - image blue). The filename of the best matching image is cached in a hashmap alongside the integer coordinate(s) of where the image is located. This coordinate value matches the (x, y) coordinate of the target pixel relative to the top left corner of the parent image. The cache of images is referenced to draw the new collage image by image on the canvas to replace the previous zoomed in image. This new collage at its current rendered resolution becomes the visual reference, the parent image, for the next collage replacement. The process of zooming and collage replacement can repeat indefinitely with this method.

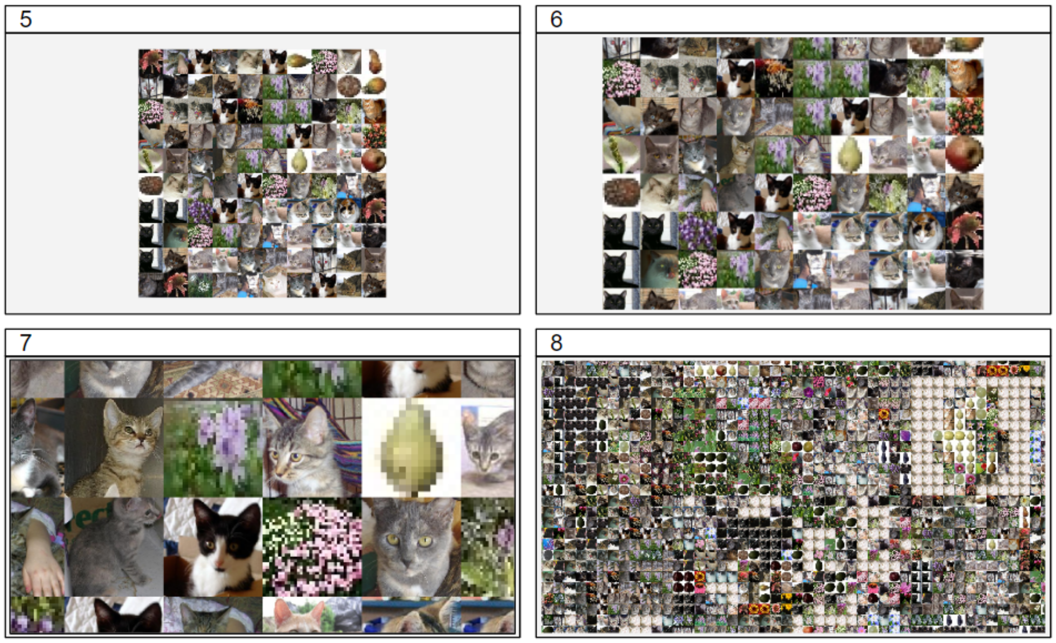


Figure 12: Continuation of zooming into collage (5-7) and a second collage replacement (8) in the project window.

Implementation

The final approach to the project involved distinct steps to implement infinite zooming. These methods included determining a way to calculate the average color of an image, efficiently storing and retrieving images, and cropping out images from the collage that are no longer in view of the window.

JavaFX Versus Java Swing

In planning the development of the application, two development toolkits were considered: JavaFX and Java Swing. Although both JavaFX and Java Swing can be used to create a Graphical User Interface (GUI), there are pros and cons of using each.

Java Swing was released in the late 1990s and has many lightweight components that function similarly across platforms (Oracle, 2020). Components can be compounded and used to create complex programs. Java Swing lets users easily customize components and offers more advanced features like color pickers. While Java Swing is relatively simple to use, it has become dated as many have opted for the use of popular tool JavaFX instead.

JavaFX was released in the late 2000s and provides additional features not found in Java Swing. First, it supports customization using CSS and XML, which can be incredibly useful in styling an application (Singh, 2018). With the development of JavaFX, Oracle created a Scene Builder application that lets users easily develop a user interface with a drag and drop approach that removes that coding aspect altogether (Oracle, n.d.). Users can modify component properties and apply style sheets all without having to modify code.

After analyzing the various features and functionalities offered by both JavaFX and Java Swing, we opted for the use of Java Swing. While JavaFX provides many features not found in Swing, it was determined that the application's user interface would not be incredibly complex. As a result of this, there were few Swing components actually needed for our application. JavaFX also contains many more libraries and components than Swing, making it much more time consuming when figuring out the exact components to use. Java Swing was easier to understand and simple in aiding in the development of the application. A more intricate and dynamic user interface could require the use of JavaFX, as it provides many more tools that can be more easily customized.

Loading Images

The efficiency of our project was analyzed by examining the time it takes to load several images of different sizes. These sizes range from 16 by 16 pixels to 2048 by 2048 pixels.

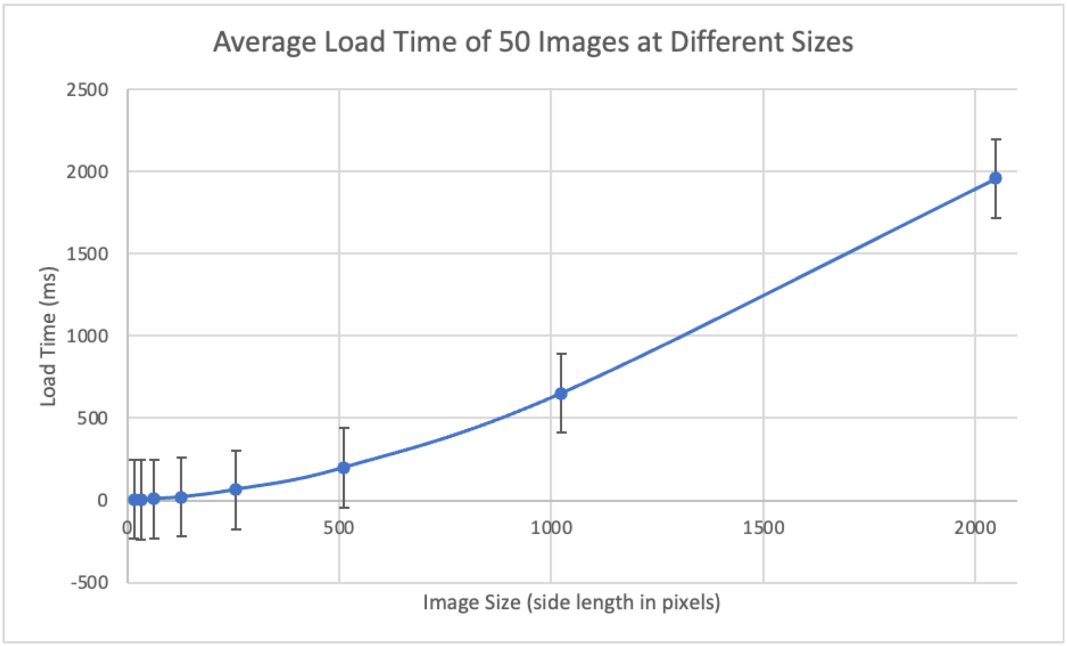


Figure 13: Average Load Time of 50 Images at Different Sizes.

For each of these image sizes, 10 trials of loading 50 images of that size were performed and the average load time was recorded in milliseconds. Figure 13 displays these measurements in a graph with image size on the x-axis and load time on the y-axis. As the image size increases, the amount of time it takes to load the image also increases. The error bars indicate the standard error of the mean, which is calculated as 240.6 milliseconds using $\frac{\sigma}{\sqrt{n}}$, where σ is the standard deviation of the sample and n is the sample size.

Development Process

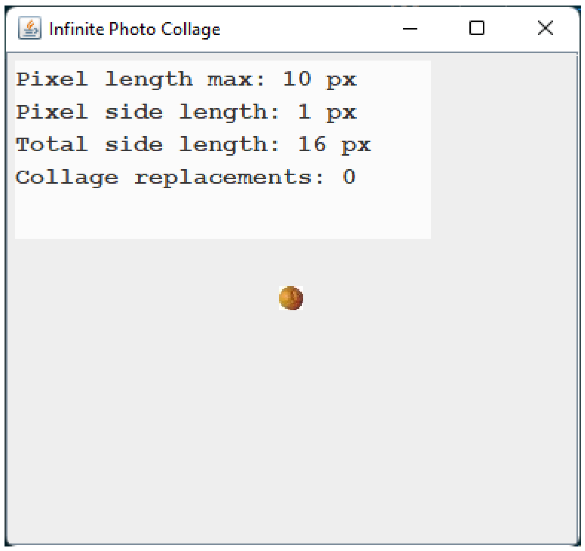
An initial brainstorming of ideas led to the first version of the infinite photo collage. Many aspects of this initial version were improved upon for the final implementation. Firstly, the goal was to maintain the original resolution of each image during zooming and collage replacements. As a result, the canvas was drawn image by image instead when zooming to account for this. This required keeping track of the image side length as well as the percent increment at each zoom level. From there, storing the total side length and side length in images in combination with the image side length helped to display images at the correct sizes during zooming and collage replacements. Also, since the method of loading images was inefficient, an image cache was created to address this.

The rest of the process focused on cropping because Java heap space errors would occur due to the large number of images that were trying to be displayed on the screen. Additionally, the scroll bars that were implemented in the initial version were removed because cropping eliminated the need and altered the centering of the collage in the window. The last major challenge was that, although the image was being cropped, it was being cropped to display the upper left hand corner of the image rather than the center.

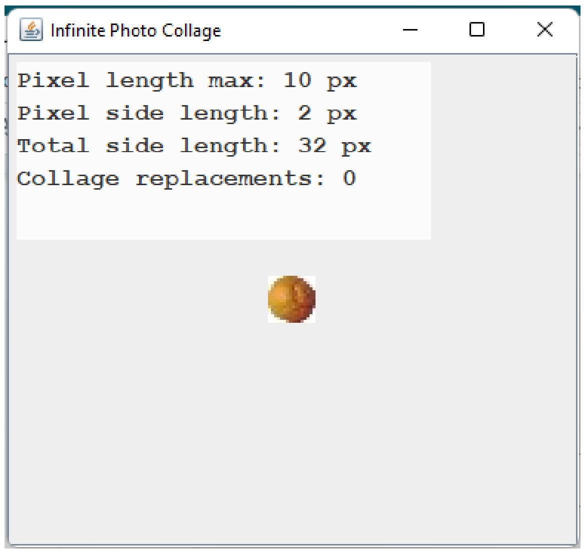
Throughout the development process, we enhanced the usability of our application by implementing, in order, the ability to zoom by using a mouse wheel, clicking the spacebar, and adding an autoplay option. We also modified the way the application started. While the initial version prompts the user for the pathname to a starting image file in the terminal, our final implementation has a separate window that requires the user to navigate to a directory of images.

Initial Version

The initial implementation of this project began by prompting the user for the pathname of some arbitrary image file that exists within the project's images folder. If a valid pathname is provided, the image is read and stored as a BufferedImage. The map of average colors is calculated at this point using the project's images folder. Users are shown the selected image, and the image grows larger when zooming in using a mouse wheel, which is equivalent to scrolling up on a trackpad. The procedure of zooming at this initial level increases the rendered resolution of one pixel in the image by one pixel. Figure 14 shows how one increment of zoom, equivalent to one mouse wheel turn, takes each pixel initially at 1x1 resolution and displays it as a 2x2 pixel length area.



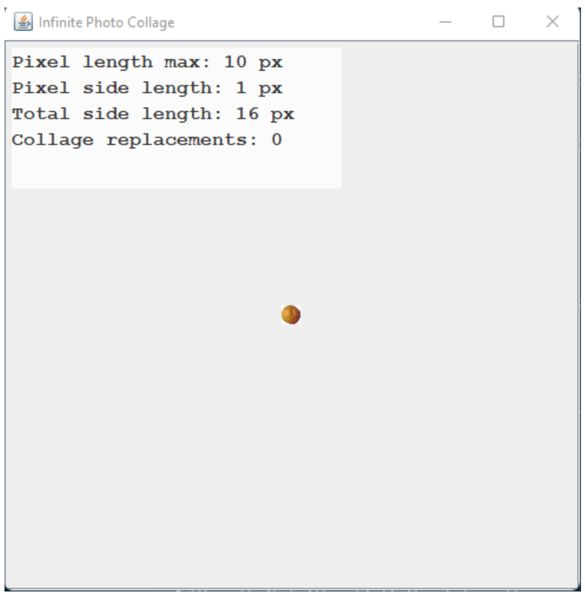
```
Pixel length max: 10 px
Pixel side length: 1 px
Total side length: 16 px
Collage replacements: 0
```



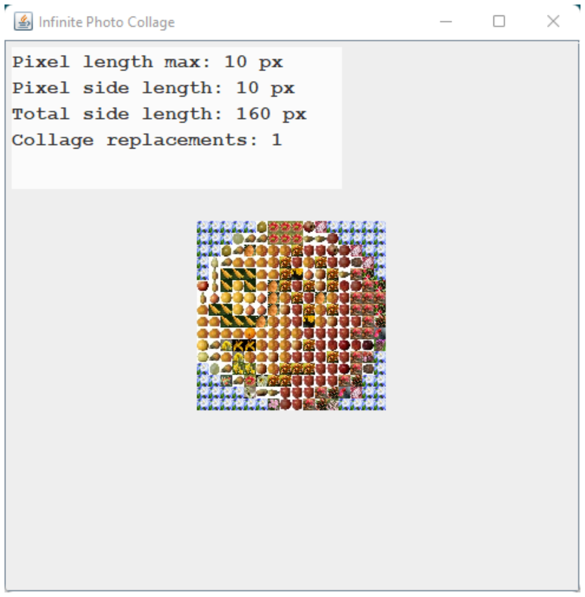
```
Pixel length max: 10 px
Pixel side length: 2 px
Total side length: 32 px
Collage replacements: 0
```

Figure 14: Image before (left) and after (right) 1 increment of zoom.

This pixel side length is used to determine when to perform a collage replacement in this version. After a certain threshold, displayed in Figure 14 as “pixel length max,” a collage replacement occurs. The parent image is the initial resolution the image was rendered at, given as “10 px” in Figure 14. A nested for loop is used to iterate over every pixel in the parent image to perform color matching for that pixel. Immediately after an image is matched to a pixel of the parent, that image is read in as a BufferedImage and displayed in its respective space on the canvas. The size of the canvas, shown as the “total side length” in Figure 14, is calculated as the image side length multiplied by the side length in images. The location of where to draw a single image is calculated by multiplying the coordinate location of the pixel that was matched, given as the current indices in the for loops, by the current pixel side length. This is repeated for every pixel area of the image which results in a collage of images drawn on the canvas. The canvas at this resolution becomes the parent image for the next collage replacement.



```
Pixel length max: 10 px
Pixel side length: 1 px
Total side length: 16 px
Collage replacements: 0
```



```
Pixel length max: 10 px
Pixel side length: 10 px
Total side length: 160 px
Collage replacements: 1
```

Figure 15: Image at initial render (left) vs. after 1 collage replacement (right).

At a certain point, only a portion of the collage is replaced when zooming into the center. All images outside of the center images of the collage are disregarded to avoid heap space errors, keeping only the middle images as the parent for the collage replacement. This is calculated by finding the center of the parent image. The center is calculated by dividing the image width and height in half. If an image is located at coordinates within a certain number of images from the center, it is added to a list of images that will be referenced for the next collage replacement. Rather than cropping to the window, we tested what number of images made sense visually and this allowed for experimentation with how cropping would look in the final implementation. This crops the collage and prevents heap space errors.

There are a few points of improvement where the initial working version is lacking. The first issue is how the collage is being cropped, which is hard coded to a specific size. We wanted the final implementation to crop to the window to make the zooming and collage replacement smoother, which would better simulate the infinite portion of the infinite photo collage. The second major issue was the rendered resolution of the images in the collage after replacements. Our initial method of rendering the collage when zooming does not respect the original resolutions of images being displayed. For instance, two images adjacent to each other will be at the same rendered resolution in a collage. If the original resolution of one image is higher than another, we could expect the first to always look higher resolution than the second regardless of the rendered resolution. We resolved this issue by implementing a cache of images to use as a reference to draw the collage image by image every time after zooming or during a collage replacement. This also allowed collage replacements to be drawn more quickly.

Image Cache

In performing a collage replacement, the same image is likely to occur multiple times throughout a collage based on color matching. Loading an image for every occurrence decreases the performance of the application in comparison to implementing a method to load every unique image only once.

Table 1 shows the time it takes to perform collage replacements when loading each image per every occurrence. This is how our initial version was implemented. Five starting images were examined over four collage replacements and the time taken to perform the replacement, by reading the image file and drawing the image on the canvas, was measured using a system timer from Java. For example, when Image 1 was used as the starting image, the first replacement took 324 milliseconds to complete while the second replacement took 2990 milliseconds to complete. Table 1 also shows the average replacement time and standard error for each image, where the standard error indicates how each image's average time would vary if more replacements were examined. The average time taken to perform a replacement across all five images is 1829.9 milliseconds.

	Image 1	**Image 2**	**Image 3**	**Image 4**	**Image 5**
Replacement 1	324	257	308	264	276
Replacement 2	2990	1423	2136	1759	1968
Replacement 3	3384	2968	3447	2235	2360
Replacement 4	2055	2028	2240	2031	2145
Mean	2188.25	1669	2032.75	1572.25	1687.25
Standard Error	681.041161	567.9306	647.33291	446.848478	477.19447

Table 1: Time in milliseconds for five different starting images to perform collage replacements without using a cache.

To improve the performance time, we created a cache to hold reference to each unique image present in the collage. This cache is implemented as a map where the keys are the filenames of each unique image and the values are array lists of coordinates. The coordinates represent the (x, y) locations of all occurrences of this image relative to the parent image.

Table 2 shows the time it takes to perform collage replacements when this cache is used. The measurements were taken in the same way as the measurements in Table 1. The measurements for Image 1 show that the first replacement took 39 milliseconds to complete while the second replacement took 148 milliseconds to complete. Table 2 also shows the average replacement time and standard error for each image. The average time taken to perform a replacement across all five images is 109.75 milliseconds. Compared to the implementation that doesn't use a cache, it took significantly less time on average to perform a replacement when the cache was used. This indicates that utilizing the cache decreased the time needed to perform a replacement, overall improving the efficiency of our code and creating smoother zooming.

	Image 1	**Image 2**	**Image 3**	**Image 4**	**Image 5**
Replacement 1	39	53	28	42	67
Replacement 2	148	146	153	149	135
Replacement 3	146	131	138	132	137
Replacement 4	114	141	95	98	103
Mean	111.75	117.75	103.5	105.25	110.5
Standard Error	25.4701623	21.8073955	28.0074395	23.5986405	16.4595464

Table 2: Time in milliseconds for five different starting images to perform collage replacements using a cache.

In our application, the cache is created before a collage replacement occurs. To create a cache for this collage, the parent image is used as the starting point. For each pixel in the parent image, we find the image filename with the corresponding average color based on the created average color map. If this filename already exists in the cache, the program retrieves the current list of coordinates and adds the current pixel location as a coordinate in the list. Otherwise, the new location is simply added. With the updated list, the program inserts the key image filename with the value list of coordinates in the cache.

For each filename in the cache, we read in the file as a BufferedImage. Next, we draw the image at every associated coordinate. The coordinates are in units of images and are multiplied by an offset value, the current image side length, to get the pixel coordinate location on the current rendered collage. Using the cache in this way vastly improves performance for both zooming and collage replacements. As previously mentioned, this also allowed us to maintain the highest resolution of the images at any given rendered resolution. This is because the cache provides a way to retrieve the images at their original resolutions and render them relative to these original resolutions. In our initial version, we redraw and resize the pixels on each zoom, causing the pixels to be zoomed into with no improvement to the resolution of the image they belong to.

Cropping to the Size of the Window

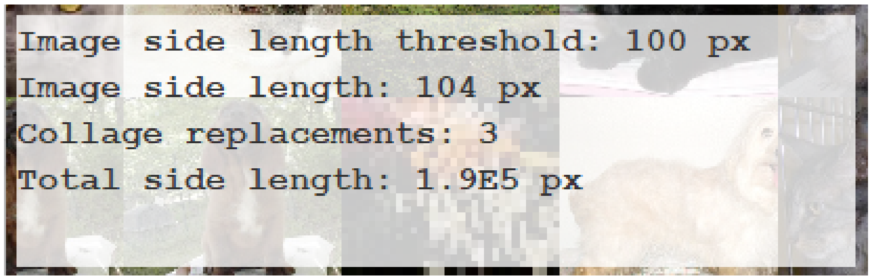
As with our initial implementation, cropping was necessary to make zooming infinite. To do this, we needed to find the difference between the side length in images that could fit into the window compared to the total side length in images that should exist for that rendered collage. The total number was already stored as a variable that was updated when a collage replacement happens. The window's width in images was calculated by taking the max of the window's width and height in pixels and then dividing by the current image side length, resulting in units of images. We round this value up to the next even number to accomplish two things: rounding up guarantees the width in images is always greater than or equal to the window size, and keeping an even number ensures the collage will always have the same center location of the images for visual consistency.

The total side length in images for a collage will always be the side length in pixels of the parent image. Therefore, the number of images to crop from the collage is the side length in pixels of the parent image minus the number of images that fits in the window. Since we want to crop to the center of the collage, this number is divided by two and is referenced as the cropping bound. The indices in both the x and y direction of images to render begins at the cropping bound and ends at the side length of the parent image minus the cropping bound.

User Interface

The project is initialized with a separate window that requests the user to select a directory of images to use for the infinite photo collage. Having a separate window allowed for more clarity in instructing the user how to start compared to requesting the image dataset in the same space as the main visual. For general information about the project, we provided a description of the implemented methods of zooming a user can perform and a high-level overview of how the collage works. In the window of the main visual, we added a "Details" button on a top menu bar that opens a new window with this information. A user can quickly and more easily understand the process they will see visually. After a valid image directory is chosen, a new window with the main visual of the images will be displayed.

The main window features a text display that overlays the frame shown in Figure 16. The font is black with a low-opacity white background to make the text more readable while also maintaining visibility on the collage behind it. This contains the current values for the replacement threshold, image side length, number of collage replacements, and approximate original image side length.



```
Image side length threshold: 100 px
Image side length: 104 px
Collage replacements: 3
Total side length: 1.9E5 px
```

Figure 16: Close up of debug info menu at some point of zooming.

The approximate original image side length is included to give the user a sense of how deep they are zooming into the starting image. Due to the infinite nature of this project, this value gets increasingly large with each zoom increment. Therefore, we needed a data type that could store integers larger than the limit of the long data type in Java. This can be accomplished with BigInteger, which is a class in Java used exactly for this purpose. BigInteger represents immutable arbitrary-precision integers and supports values in the range $-2^{\text{Integer.MAX\_VALUE}}$ to $2^{\text{Integer.MAX\_VALUE}}$ exclusive, where Integer.MAX_VALUE represents the maximum possible integer value in Java (Oracle, 2019). It stores the number in two's-complement binary representation and uses this representation to perform operations. These operations can be performed through several methods, which cover Java's primitive integer operators as well as other mathematical operations (Oracle, 2019). In our project, we used the BigInteger.multiply() method to multiply the original side length at the previous zoom level by 1 + the zoom percent increment to approximate the original image side length.

Additionally, we added an autoplay feature that will perform one increment of zooming based on a timer. A user can toggle the Play/Pause button on the top menu bar that will trigger it. This addition is useful in cases where other methods of zooming may be unaccessible, and provides a more streamline visualization of the collage replacement.

Computational Challenges

In the duration of our application's development, several computational challenges arose that were crucial in attaining the desired application behavior.

One of the issues that arose early on in development was that we were consistently receiving the Java heap space error, and we also noticed that the program slowed tremendously as the collage grew larger. After investigating the issue, we noticed that each image of the collage, regardless if unique or not, was being loaded from memory every time a zoom or collage replacement occurred.

To overcome this issue, we decided to build a cache that would store an image file and the coordinates of each occurrence of that image in the collage. With each collage replacement, the cache is reconstructed to store the images of the former collage. The cache increases the application's speed because it is used in both zooming in on the collage and collage replacements. When the collage is resized or is replaced, each image of the former collage needs to be placed back on the canvas in its new dimensions. The cache is referenced so that an image filename can be loaded only once and can then be drawn on all associated coordinates, significantly improving overall speed of the program if an image is to be displayed multiple times. As a result of utilizing the cache, we avoided the heap space error for a time and our program's speed visibly improved.

Another topic we struggled with involved calculating the bound for cropping the collage during zooming and after collage replacement. Cropping was a necessary step for rendering the images as we needed to simulate infinite zooming and the size of a BufferedImage creates a heap space error after a certain render resolution. We started with the same approach as in the initial version and instead calculated the number of images that could fill the width and height of the window. This was calculated by rounding up the max of the window's width and height divided by the current image side length. We then used the center coordinate of the collage, equal to the parent image's side length divided by 2, plus or minus half the number of images that could fit in the window as the cropping bound; like in the initial version, this cropped the collage outward from the center. This method involved recalculating values for the canvas's total side length, side length in images, and image side length. The new values altered the pixel coordinates of where images were being placed on the canvas, therefore we needed to rethink this approach. The final method instead calculated the number of images from the edges to ignore, as mentioned in a previous section.

Evaluation of Progress and Next Steps

Throughout our application's development phase, we came up with several ideas for future development that we didn't have the time to actually implement. The work was evaluated and assessed by referencing our initial goals for the application.

Assessment

After analyzing our finished application as a whole, we have achieved everything we set out to with the project. A central component of the project was to create an application that would let users zoom infinitely on an image, effectively turning that image into a collage of images that can be continuously zoomed in on. Our finished application does just that, and we also added some other features to augment the overall application user experience. For example, we added an autoplay feature to the program so that users can watch the collage grow without pressing the spacebar or using a mouse. Along with additional features and overcoming several computational challenges throughout the course of development, our work has allowed us to achieve our goals for the project.

Unexplored Ideas

Part of finalizing the infinite photo collage required prioritizing which methods or ideas we would implement or remove for computational or complexity purposes. Among these include zooming out, visual improvements, a tree structure, and panning.

One idea was the ability to zoom back out. After zooming in for a period of time, a user could zoom back out of the collage, which would eventually result in the starting image again. Our current implementation does not keep track of images outside of where collage cropping takes place, so zooming back out of the collage would require a different image storage method.

Another idea for a potential new feature was a toggle or input that would allow users to control how fine-grained collages would appear. In our application's development stages, adjusting the initial side length of the starting image on the canvas controlled the resolution in images of the collage replacements. With a smaller image side length value, more images will be needed to fit in the same size collage. More images increase the collage's level of detail, as additional, smaller images with more accurate average color values are shown. In our current implementation, all images, regardless of original resolution, are replaced with a collage whose resolution in images is the same. Therefore, the resolution of an image does not affect the resolution of its collage. The user experience could be visually improved if collage resolutions depended on original image resolutions.

The resolution of a collage could also vary based on its parent's original resolution in a different approach. Adjacent images would therefore be replaced at different points in time where an image could be next to a collage, as shown in Figure 17. A possible implementation for this would be maintaining a tree structure with nodes as images. Nodes would store the image filename it represents, its current image side length, the coordinate locations of itself relative to its parent node, and a list of child nodes that would be the collage to replace itself. A collage replacement on a single node could occur when the ratio between the original resolution and current image side length crosses a certain threshold.

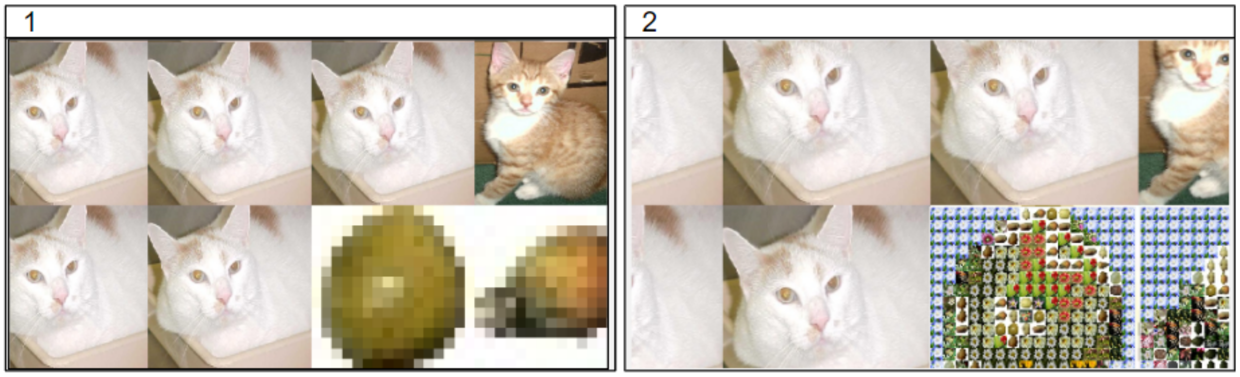


Figure 17: Example of the tree implementation before and after one zoom increment.

Panning using scroll bars is another feature we hoped to implement. Rendering and updating the scrollbars affected the centering of the collage. In developing the remainder of the main functionality of the project, we decided to not pursue fixing this issue. However, it would be visually interesting to be able to move around the rendered collages in future work.

Conclusions

In this Major Qualifying Project, we created an Infinite Photo Collage. The application lets users choose a directory of images from their file system, which will be displayed in a new window. Users can zoom into this image, and eventually, the image will become a collage of images that collectively resemble the original image. After further zooming in, the collage will be broken into a collage of even more images, allowing users to zoom in infinitely. After working on the application for an entire semester, our team has come to several conclusions. We have learned much about software development and working efficiently in a group. Additionally, we have constructed several recommendations for future MQP teams that helped our team succeed.

We have gained incredible knowledge by developing an entire application from scratch. As a team, we brainstormed how we wanted the application to look and behave, and we used this as a reference point throughout our development phase. Transforming a seemingly simple idea into a full-fledged application has helped us become more familiar with working on an intricate team project. As a result of working in a team for an extended period of time, we have also learned how to balance a team project with our individual obligations. Finally, we have learned much in the area of software development. None of our team members had previously worked with images in Java, so we took the project as an opportunity to learn how to load and display images. Additionally, one of our initial goals for the project was to gain experience with new libraries and technologies. The primary driver of our application is Java Swing, which is used to construct the application windows and display collages. We heavily researched how to utilize Java Swing to implement our application efficiently. Our team then used this knowledge to build the components necessary to hold and display collages in our application.

Additionally, through conducting experiments to measure the timing of loading images, we were able to explore the effects of loading images of increasing sizes as well as the effects of using a cache to store unique images. As expected, images of larger sizes had a longer load time. We also discovered that caching improves the efficiency of our application by comparing the time it takes to perform collage replacements with and without a cache. These analyses provided insight into how our application behaved as a whole.

When reflecting on the MQP process, the project ran smoothly overall. We met consistently at the exact times and days each week so that the project would feel more like a course in our schedules, and we found this was crucial in maintaining progress each week. For future MQP teams, we recommend finding times to meet and work on the application early on in the project so that the team can shift into a routine. Another recommendation is to work on different aspects of the project each week. During our development phase, we worked on both the application and report, which allowed us to stay caught up and consistent.

We have learned so much about software development and ourselves during this MQP project, and we are pleased with the application overall.

References

Bright, R. (2019, September). *On Computational Art*. Interalia Magazine. https://www.interaliamag.org/interviews/ernest-a-edmonds/

Dawson, R. (2022, January 10). *Mosaic*. GitHub. https://github.com/codebox/mosaic

Furstenheim. (2020, December 29). *Infinite-mosaic*. GitHub. https://github.com/furstenheim/infinite-mosaic

Garcia, C. (2016, August 23). *Harold Cohen and AARON—A 40-Year Collaboration*. Computer History Museum. https://computerhistory.org/blog/harold-cohen-and-aaron-a-40-year-collaboration/

Kircher, M. M., & Holtermann, C. (2022, December 7). How Is Everyone Making Those A.I. Selfies? *The New York Times*. https://www.nytimes.com/2022/12/07/style/lensa-ai-selfies.html

Obvious. (n.d.). *Obvious: Artificial Intelligence for Art*. http://obvious-art.com/wp-content/uploads/2020/04/MANIFESTO-V2.pdf

Oracle. (2019). *Class BigInteger*. https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/math/BigInteger.html

Oracle. (n.d.). *JavaFX Scene Builder*. https://www.oracle.com/java/technologies/javase/javafxscenebuilder-info.html

Oracle. (2020). *Package Javax.Swing*. https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

Osinga, D. (2022, July 3). *Use DALL-E to create infinite zoom movies*. Medium. https://dosinga.medium.com/use-dall-e-to-create-infinite-zoom-movies-234fb72c85ab

Singh, J. (2018, September 24). *What is JavaFX and How is it Different from Swing and AWT?* Medium.

https://medium.com/@japkeerat21/what-is-javafx-and-how-is-it-different-from-swing-and-awt-54de995e4869

Victoria and Albert Museum. (2009, April 8). *Schotter*. http://collections.vam.ac.uk/item/O221321/schotter-print-nees-georg/

Victoria and Albert Museum. (2011, March 28). *Structures of Squares*. https://collections.vam.ac.uk/item/O1193777/structures-of-squares-drawing-vera-molnar/