Abstractanator: Final Report

Team 01
Members: Hunter Black, Evan Johnston

Introduction:

When coming up with an idea for our final project, our team eventually landed on the general concept of abstraction. We felt that although there are programs currently that allow users to apply filters to images to "abstract" them, often these programs are overly complicated and are often very expensive, such as Photoshop. Because of this, users may be turned off from the idea of exploring the concept of abstraction. In order to solve this problem, we wanted to create a free application with a simple UI that would allow users to import their own images and abstract them to explore who applying multiple different abstractions would affect an image.

After going through the development process for the Abstractanator, we were able to deliver on the basic premise of our goal. We designed and developed a simple image abstraction application that is easy and intuitive to use while allowing users to explore and discover the effects of applying different abstractions to an image in sequence. We ended up developing algorithms for 4 different types of abstractions that could be applied to an image: an Red Green Blue (RGB) pixel randomizer, a black and white polarizer, a color polarizer, and a folder/unfolder. Although we had plans to develop more abstraction functions, we underestimated the scope of this project for a team of two, and thus had to cut the ideas for other abstractions that we had come up with during our brainstorming phase of this project.

In conclusion, we were able to develop a working build of our initial concept that allows for basic abstraction of imported images and saving of said images. We were unable to fully integrate all of the abstraction methods that we had originally come up with, however our final product stands on its own as a fully functional tool for exploring abstraction. In the future we hope to add additional abstraction functionality, as well as possible 3D integrations so that abstracted images can be examined from a different perspective.

Example:

Abstractanator: Final Report

Team 01

Members: Hunter Black, Evan Johnston

To get a basic idea of how our application works, we have developed a simple use case scenario below that walks through some simple steps on importing an image, abstracting that image, undoing some abstractions that had been performed, and then exporting that image for later use:

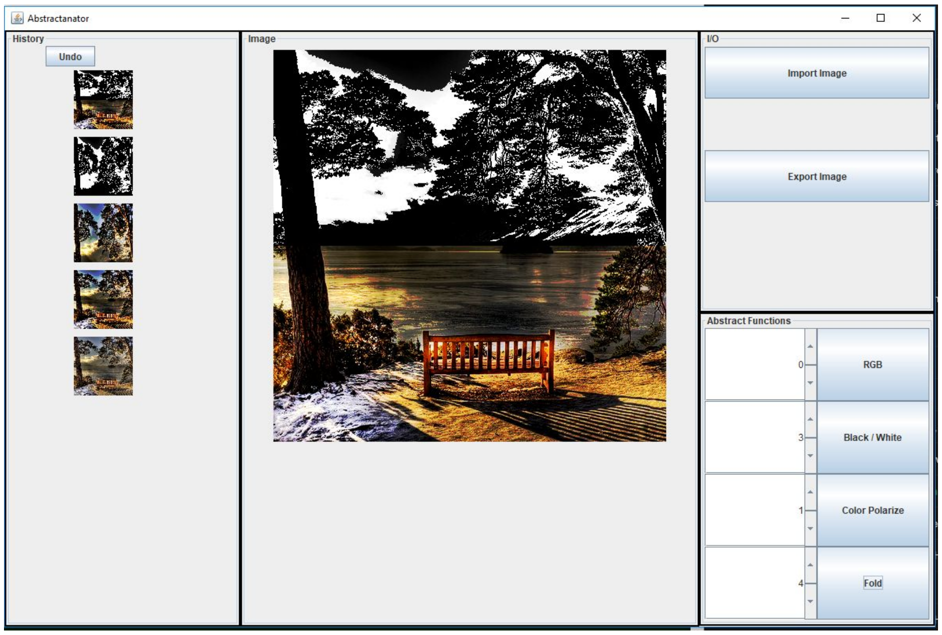
Jeff Kim, an amatuer photographer and college student, has decided that he would like to explore different ways of altering the photos that he takes to create more abstract and thought provoking pieces that he will display at his school's art gallery. To do so, he has decided to use the Abstractanator.

First, he imports one of his photos that he has stored on his computer into the Abstractanator application by clicking on the "Import" button on the top right of the application, navigating through the file structure of his computer to where the photo is saved, and clicking "open". His photo is now displayed in the center panel of the application, as well as in the history panel as a small thumbnail version of the larger image.

Once this is done, he decides to first apply the color polarization abstraction to the image by clicking on the "Color Polarize" button on the bottom right of the application. Since he does not want too extreme of a polarization, he leaves the iteration counter to the left of the color polarize button at 1. After clicking on the color polarize button, the image gets the associated abstraction applied to it, and the application updates. Now, the center panel will display the color polarized image, and the history panel will get updated with the change that was made.

Now, Kim decides that he wants to apply a black and white abstraction to the image, but only to the top half of it. To do so, he sets the fold side counter next to the "Fold" button to 4 (which indicates that the bottom half of the image will be folded back), and clicks the fold button. The application now displays the image folded in half, with only the top portion showing. He then applies the black and white abstraction by clicking on the "Polarize" button, setting the buttons associated iteration counter to 3 to apply a more extreme black and white abstraction to the folded image. Satisfied with that, he then unfolds the image by clicking on the "Unfold" button (which is the same button he clicked to fold the image that has now been renamed to unfold since the application has detected a fold). This updates the application and shows the new image that has different abstractions applied to both the top and bottom half of the image.

He then decides that he does not like the abstractions that he has applied, and clicks the "Undo" button on the left hand side of the application to return to the color polarized image. Satisfied with that, he exports the image by clicking on the "Export" button and choosing a destination to save the newly abstracted photo. Below is an example screenshot of the above scenario.



Process:

When we first came up with the idea for the Abstractanator, we laid out some basic goals for the project as a whole, a short timeline that accounted for holidays and predicted course load from all of our members' other schoolwork, and the expected scope of this project based on our current knowledge of the tool sets that we had decided to use, and finally a set of smaller tasks that broke down the project into manageable chunks for a smoother development process. We decided to approach the development of the Abstractanator by dividing the work into to major sections: the UI development of the application, and the algorithm design for our abstraction functions. We did this so as to prevent work overload and confusion if we were to both work on the same parts of our application at the same time, as well as allow us to simultaneously work on the project without relying on one another's individual progress to contribute to the goals we had laid out at the beginning of this project. Additionally, we approached the our code base with the idea of making it as modular as possible, so that future improvements and additions could be made with ease.

As discussed above, to manage our project's scope effectively, we broke down the overarching goals into smaller tasks that could be completed sequentially. These tasks can be divided into three major groups: UI and UX development, backend/algorithm development, and integration.

For the UI and UX tasks, we broke down the tasks into small, achievable things such as creating the main display window, dividing the window into subsections to better distinguish button groups, creating the buttons, and ensuring an intuitive and easily accessible feel throughout. The backend tasks involved creating the RGB randomizer, black and white polarizer, color polarizer, and folding abstraction algorithms, as well as creating necessary objects to hold our abstracted images. Finally, the integration tasks involved wiring the backend with the UI and ensuring that the functionality of the application as a whole was correct. These tasks included making sure that all buttons in the UI applied their designated abstraction function to the image, making sure that the history list was displaying abstracted images correctly, and insuring image I/O worked correctly.

After coming up with these tasks, we came up with an initial timeline to complete these tasks. This timeline allowed 2 weeks to develop one abstraction function, starting in mid February, while at the same time allowing roughly 1 month to get the UI designed. Finally, we allowed for roughly 1 month to hook up the backend of our application with the UI. Additionally, in our original project plan we had set aside time to integrate a 3D/VR implementation of this project to further explore these abstracted images. We were unable to stick to this timeline, however. Due to unexpectedly high class work loads for two of our team members, as well as losing one of our team members roughly halfway through the semester, we ended up not allotting as much time for the development of our abstraction algorithms. In the end, we ended up doing the majority of our algorithm development for the abstraction functions in the last month of the project.

In addition to having to shift our timeline, we also decided to cut back largely on our project scope by getting rid of the initially planned 3D/VR environment implementation and sticking to a 2D application, as well as cutting our planned geometric abstraction function. We mainly decided to do this due to the increased workload on the team after one of our team members had to unenroll from the course roughly halfway through the semester without notice.

Detailed Results:

The special features of the product are its set of abstraction features. The features are listed below:

The first feature that was implemented was the color randomizer. Over the course of a predetermined set of iterations by the user, the algorithm loops through the entire image and adjusts each pixel's RGB values by a random amount for each of the three color parameters. The randomness has an upper limit as to how far each color parameter can be adjusted in each iteration, to ensure that despite randomness, there is some consistency to be expected from certain sets of iterations. One iteration makes it appear "off," ten iterations makes it look like a camera trying to detect light in a dark area without the flash, and one hundred iterations makes it almost incomprehensible static. However, due to the fact that there is no wrap-around (that is, a color parameter of 0 would not loop below 0 and become 255), pixel values tend to become stratified. This has the side effect of the changed image still bearing some resemblance to the original if the colors of the foreground are distinct enough from those in the background.

The second feature that was implemented was the black-and-white polarizer. Much like the algorithm above, this feature iterates on the entire image and adjusts each pixel. In this case, it first checks if the image is already in grayscale (via a boolean). If it is not, it makes it so via a helper function. After the image is guaranteed grayscale, it checks the red value. If the value is below 127, 32 is subtracted from its amount; otherwise, 32 is added. Naturally, a check is put in place to make sure the value does not escape the bounds of a color parameter. After this adjustment is made, the green and blue values are set equal to the red (to ensure that the pixel is still a shade of gray). Due to the way this algorithm is structured, each pixel will eventually take on the value [0, 0, 0] (black) or [255, 255, 255] (white).

The helper function mentioned above (simply called "grayscale") takes the average values of the RGB parameters and sets each color parameter equal to that value; this is done for every pixel in the image. Once every pixel has equal values in its RGB parameters, they are all some shade of gray. The image then sets its boolean checking if it is grayscale to true so that this function does not have to be called with every iteration of the black-and-white polarizer. If any other function is called, however, this boolean is set to false, as there is no way of knowing without iterating through the image again whether or not it is still in grayscale (and barring the astronomically unlikely event that the color randomizer adjusts every pixel value by nothing, there is really no chance of another function besides folding leaving the image in grayscale).

The third feature implemented was the color polarizer. This algorithm operates on a similar idea to the black-and white-polarizer; however, this time the color values are not averaged (therefore not making them gray), and the RGB values are checked individually and adjusted by the same limits as the black-and-white polarizer. For example, if a pixel's values were [90, 137, 25], one iteration of the value would take it to [58, 169, 0], and after four the value would be [0, 255, 0] (green). Just like the black-and-white function, all of the RGB values would become stratified, but on a per-parameter basis instead. In practice, this tends to have the effect of making an image look much brighter and simplistic than before.

All of these abstraction functions are listed as private for simplicity and security reasons. Instead, whenever the user wishes to call any of these functions, the button that is pressed passes two integers to a function called “abstractinate.” The first integer is the type of abstraction to be performed. The integers are so named by defined constants according to the abstraction function, such as “POLARIZE,” which made it easier for the developers to tell which integer was meant to do what. The second integer is the amount of iterations that the given abstraction algorithm is meant to go through; if the number is for some reason below 1 (or if the overloaded function without an iteration parameter is called), the number of iterations defaults to 1.

The last feature implemented was the “folder” and “unfolder,” a twin set of functions. The folder reads in a value from the GUI that determines which direction should be folded “behind” the image. The values 1, 2, 3, and 4 correspond to left, right, top, and bottom, respectively. For instance, in the user story given above, Kim selected 4, as he wanted to fold the bottom half behind the image. The folded image then kept track of which half of it was folded, and all further abstraction functions would operate only on the top half (that is, the horizon). Of course, if a new image was loaded in via the import function, the image would no longer be counted as folded.

The “unfolder” feature takes the place of the fold button if the program detects that the image is folded. Realistically what this does is recombine the two halves of the images, so that the dimensions of the unfolded image match those before it was folded. These two features, when paired, give the user the capability of operating on only one half of an image to generate further contrasting images.

The other major part that bears mentioning is the “AbstractImage” class. An AbstractImage holds one image and also stores the relevant information with it. The necessary information each AbstractImage must hold is the image itself, the folded part of the image (if there is one), the position from which the fold was made (if the image is folded), the thumbnail, and whether or not the image is in grayscale.

The appearance of the GUI itself was designed to be as simple and intuitive as possible with no bells and whistles attached. The left panel is the history of the images. Within that panel, there are up to five thumbnails showing the previous iterations of the abstraction, with the top one being the most recent. The largest panel in the middle displays the image currently being iterated on. The two panels to the right separate the user interaction between input/output and using abstraction functions on the image.

In terms of implementation, we were somewhat shackled by our lack of knowledge of other technology at the beginning of the course. This caused us to resort to Java2D for any graphical purpose. By the time we had enough experience with the new technology, it was late enough in the semester that it was not feasible to change the tools we worked with.

For documentation, we believe that we have sufficiently commented our code without disrupting the flow of the logic behind it. Most functions (other than getters and setters) describe what they do via Javadoc, such that the developers may hover over the function anywhere to get a quick reminder of what the function does. Any logic that might bear explanation is given a concise line or two before it is executed.

As for the structuring of the code itself, we attempted to keep each part as distinct as possible. The Abstractanator and AbstractImage would handle all of the logic and book-keeping of the abstraction, whereas the View and the Controller class would handle the front end that would then pass the information along to the abstraction classes. Almost every time the View interacts with the abstraction classes, it is calling functions that tell the classes what to do, rather than handling the logic itself.

Lastly, the requirements for effectively utilizing this program are few. Any device with a reasonably recent version of Java should be able to run the program without any problems (although the GUI itself is optimized for computers rather than cell phones or tablets). Any user with basic capabilities at operating a computer and reading a manual should be able to effectively use the program.

Future Direction:

In terms of major features for the Abstractanator that ended up not being implemented and/or designed, there were 2 that got cut. These included the 3D/VR environment implementation for the Abstractanator and the geometric simplification abstraction method. The 3D/VR environment implementation had originally been planned as a supplement to the main 2D application that allowed for image abstraction, and was intended to serve as a secondary way of exploring the images that had been abstracted by being able to place them in a 3D, interactable environment with movable light sources to see how these images would appear when viewed from different angles and with different light sources. The geometric simplification abstraction was intended to examine an image passed into it, detect general shapes within the image (circles, squares, etc), and change the image so that these shapes appeared more pronounced, either by manipulating the surrounding pixels of the detected shapes or by drawing an entirely new image closely related to the original but with less detail in terms of color.

One feature that we have yet to plan but that would be ideal in adding next would be a simpler new abstraction method. This could include a function that applied Floyd-Steinberg Dithering to the image, for example. This type of abstraction would be ideal and easy to add due to the modularity of our code base, as well as abundance of academic sources related to this specific type of image manipulation. Additionally, because we become more familiar with pixel color control through the creation of our already implemented abstraction methods, developing our own algorithm to apply Floyd-Steinberg Dithering would prove less difficult than developing an algorithm to affect the overall shapes within the image itself.

In the future, some potential directions for expanding the development of our application could include modifying our code to be able to display in a web browser and thus allow us to reach a wider audience of users, research and implement new ways of abstracting images into our application, adding the ability to "redo" any undo action that is unwanted, and improving the overall look and feel of the application by moving away from the Java Swing library to a more visually appealing UI library.

Conclusion:

In recap, the Abstractanator aims at solving the issue of there not being very many cheap, easy to use and simple applications that allow users to explore the ideas of abstraction iteratively. To go about developing this application, we broke the development process into smaller tasks that involved UI and UX design, algorithm development, and backend integration. As a result of our work, we have produced a working application that allows users to import their own images, apply as many abstraction filters iteratively as they like to the loaded in image, undo any changes that they want to, and then export the newly abstracted image to a specified location on the user's computer.

Team Summary:

Originally, our team consisted of three members. However, a little less than halfway through the semester one of our team members could no longer continue with the course. The two team members remaining, Hunter Black and Evan Johnston, had to rearrange the workload of our project to accommodate this.

Hunter Black took on the role of the lead UI and UX designer in charge of developing the main design of our application as well as hooking up the abstractanator functions up with the UI in a way that was intuitive and that would allow new users to quickly start using the application. Additionally, he was in charge of formatting the final source code to ensure a unified look, as well as formatting the final documentations and reports for submission, including the project plan, progress report, and final report.

Evan Johnston took on the role of the main backend developer, and was in charge of creating the algorithms for our abstraction functions. Additionally, he was in charge of a majority of the source code documentation, as well as editing the project's submitted reports, including the project plan, progress report, and final report.