A Safer Way To Track Forest Fires

New Mexico

SuperComputing Challenge

Final Report

April 4th, 2018

Team ATC55

Academy for Technology and the Classics

Team Members:

Etta Pope, 10th Grade

Mindy Bilbo, 12th Grade

Savannah Valerio, 12th Grade

Shyla Sharma, 11th Grade

Teacher:

Jenifer Hooten

Table of Contents

Executive Summary

Due to the dangerous nature of fires, countless civilians and even trained professionals die every year when faced with forest fires. Some of these deaths are, in part, caused by the necessity of having these trained professionals get dangerously close to the fire in order to gather important information about it. This information then in turn offers firefighters intel into where the fire started, how it's moving, and where it's going. The goal of our project is to provide a safer, more efficient way to track forest fires, in which all information is collected and processed technologically, eliminating the need for human lives to be put in harm's way. Our project started with research on the current techniques used to gather and share information about the fires, as well as their efficiency. Information was also gathered about the average size and temperature of forest fires, which was then used as a basis for research on drone technology, as we plan for our code to be implemented using drones. All research is further detailed in the *Background Research* section.

Our code was written in Java using the program jGrasp, where we analyzed each individual pixel of different pictures of fires. From each pixel, we were able to gather the red, green, and blue values of the pixel's color. From there, we associated these values with specific colors. Then, we were able to take the colors of the pixels, and match them with the average temperature of a flame that color. We plan to extend this project into a two year project, as we would like to be able to develop a code that can not only analyze the pixels of the picture, but can also stitch together multiple images from different angles, all received from a drone, to create one complete image of the fire.

Problem Statement

When beginning to think about a topic for our project, we decided the most important thing to us was finding a way to stop unnecessary deaths. From there, looking at forest fires was an obvious next step, as many preventable deaths every year are caused in relation to fires. For example, 201 firefighters died in 2016 alone while responding to fires [1].

Of all of these deaths, some of the most preventable revolve around mapping out the fire. While this is an essential part of the process of fighting fires, perhaps even the basis for all following actions, it should not cause people to lose their lives. The mapping of the fire is what allows the firefighters to create a plan of action and find the best, most efficient way to stop the fire from growing and continuing to wreak havoc. Because mapping is such an incredibly important aspect of fighting fires, it is necessary to come up with new, creative ways to take the life-threatening characteristic out of the process. Because of all of this, we decided that we would like to find a way to digitize all of the firefighters' mapping. By doing this, we hope to help lower the amount of firefighters killed each year while working to stop wildfires.

Background Research

The two main causes of wildfires are humans and lightning. Human-caused fires are typically the result of campfires left unattended, burning debris, discarded cigarettes, and sometimes arson. Humans are the reason for ninety percent of wildfires. Lightning makes up a majority of the remaining ten percent. Lightning has two components: the leader, a probing feeler sent from the cloud, and strokes, which are the return streaks of light that create the visible flash. There are also two different types of lightning--cold and hot lightning. Cold lightning is a series of strokes with an intense electrical current, but a relatively short duration. Hot lightning has a current with less voltage, but occurs for a longer period of time. Extremely long-lasting hot lightning bolts cause wildfires. [2]

On average, there are more than 100,000 wildfires that take place around the world per year. These wildfires clear 4 million to 5 million acres of land in the U.S. every year. Wildfires can move at speeds of up to 14 miles an hour, consuming everything in their paths. An average surface fire on the forest floor might have flames reaching one meter in height and can reach temperatures of 800°C. Under extreme conditions, a fire can give off 10,000 kilowatts or more per meter of fire front. These conditions make fighting wildfires extremely dangerous, as the fires can be fast moving and unpredictable. Worldwide, wildfires kill around 339,000 people per year. This is makes firefighting a very perilous field. In the United States, eighty-nine firefighters died while on duty in 2016, ten of which were part of the wildland agency. To prevent deaths and assist firefighters; surveillance, command, and control aircraft are often employed above a fire [2].

The most useful information about wildfires is provided through maps. There are two main types of maps: perimeter maps and infrared maps. Perimeter maps are usually created by GPS data collected by various sources, such as fixed wing aircraft that fly over the fire and personnel on the ground. However, there are several issues with perimeter maps, including smoke and wind. The second type of map, infrared, detects hotspots and temperature. They then provide this information through the infrared spectrum. This data can be collected through satellites or scanners. The major advantage of infrared imaging is that it does not rely on sunlight, but instead detects emitted heat energy. Heat spots need to be 600 degrees to be detected. Infrared scanning is done at night due to favorable temperature contrasts, and scanners can cover almost 1 million acres in one hour of flight time. In infrared maps, warm spots are depicted as white, while cooler spots are displayed in shades of yellow, red, and purple. Infrared maps are more commonly used with wildfires. The maps can be carried on handheld devices, making them more easily accessible to firefighters. [3]

Currently, two U.S. Polar Orbiter (POES) satellites are used to collect data for infrared maps. POES satellites orbit the earth fourteen times each day at an altitude of approximately 520 miles. A satellite is able to view a 1,600 mile wide area of the earth with each orbit. Because of Earth's rotation, the satellites are able to view the same area twice a day, once during the day and once during the night. This provides daily readings of an area, which can be used to track a wildfires movement and behavior. Although infrared imaging itself does not require light, a visible satellite image does. A visible satellite image is created by looking at the visible portion of the light spectrum, meaning that it is only useful during daylight hours. Data from multiple orbits are stitched together to provide wide scale global

views of the Earth in a single image. This can result in occasional dark spots on POES images due to gaps in data transmitted from the orbiters. [4]

In addition to POES, there are a number of other satellites that help detect changes in the environment. Meteorological, or weather, satellites are primarily used to monitor the weather and climate of Earth. Meteorological satellites can detect fires, sand and dust storms, auroras, and other environmental information. Other environmental satellites can sense changes in Earth's vegetation, sea state, ocean color, and ice fields. The satellite image displays objects based on the temperature of the object, with warm temperatures appearing in dark shades and cold temperatures appearing in light shades. These maps are used to prevent and inform people of natural disasters. [5]

MODIS, an Active Fire Mapping Program, uses data from satellites to track fires. Pictures of the Earth are taken everyday at latitudes less than 30º, and every few days at latitudes greater than 30º. Thermal information is collected twice a day. It features two different sensors: one for collecting daytime data and one for nighttime data. Observations are made of the United States and Canada. The sensors and satellites not only provide information about fires, but are also used to monitor land and sea. The program is managed by The USDA Forest Service Geospatial Technology and Applications Center, based in Salt Lake City, Utah. Another method is using U.S. Forest Service aircrafts for infrared flights, but method is expensive, with each flight costing between $1000 to $5000. [6]

An alternative way to collect data for fire mapping would be using drones. However, this method can be expensive depending on the type of drone. Military drones, Unmanned Aerial Vehicles (UAVs), can stay aloft for up to 17 hours at a time, loitering over an area and

sending back real-time imagery of activities on the ground. These drones are more durable than normal drones, and are able to withstand higher temperatures, enabling them to get better views of wildfires. This drones are usually equipped with a camera, which can provide infrared images. The cost per flight depends on the type of drone. Currently, the Pentagon has 7,000 aerial drones, with $2.9 billion for drone research, development, and procurement in the year 2016. In addition to that, the Department of Homeland Security has at least ten unarmed Predator drones, costing approximately $62 million a year. Predator and Reaper drones cost about $2,500 to $3,500 per flight hour. [7]

Normal drones are limited by their flight time, which ranges from eight to fifteen minutes before the battery needs to be changed. Drone batteries use lithium polymer technology which allows considerable energy to be stored in a small package. However, these are associated with fire risk and have been known to spontaneously catch fire while charging or if punctured. Most drones have a maximum speed of 30mph, which means they can only be used with wind speeds less than 20mph. For this reason, drones cannot be used in snow or even a slight drizzle. Cold weather can also cause a reduction in battery life and give shorter flying times. Another factor is altitude, in which the thin air in the mountains means that special rotors are needed and battery life is reduced.

A new technology is the Fireproof Aerial Robot System (FAROS), which was recently developed by a research team at the Korea Advanced Institute of Science and Technology. This drone was designed to help detect and combat fires in high-rise buildings, so it can both fly and climb walls. It contains a thermal-imaging camera that enables it to locate people trapped inside buildings, in addition to a range of other image-processing

technology that allows it to identify a fire's point of origin. FAROS is able to withstand extreme heat, enduring temperatures of up to 1,000°C for over a minute without losing its functional capacity. This is because its coating is made up of aramid fibers, which also commonly used by the automotive, aerospace and military sectors to create heat-resistant surfaces. The drone is able to access fires in skyscrapers, entering the burning building in order to conduct a search and transmit information back to firefighters on the ground, who can then decide on an appropriate course of action. [8]

Original Code

In the following sections, the first code we created will be broken down piece by piece for an in-depth explanation. Code that does not require explanation will be omitted. The full code can be found under *Appendices* as *Appendix A*.

To start off our original code, we implemented multiple libraries to work with an image in Java.

```
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.image.PixelGrabber;
import java.awt.Color;
import java.awt.image.ImageConsumer;
import java.io.*;
import java.util.*;
import java.util.Arrays;

// Based off of: [9] Tips, Java PixelGrabberTest [Source code]
<https://www.java-tips.org/java-se-tips-100019/23-java-awt-image
/1950-how-to-use-pixelgrabber-class-to-acquire-pixel-data-from-a
n-image-object.html>.
```

Image, ImageConsumer, Color, and PixelGrabber are all imported specifically to allow for the analysis of a picture. The other items imported are basic java libraries that are not specific to our program, but can instead be found in many programs with varying outputs.

```
public static void processImage(String inFile, String outFile)
     {
       System.out.println("Test 1");
Image image = Toolkit.getDefaultToolkit().getImage("Fire.jpg");
```

```
        System.out.println("Test 2");
          try
              {
    PixelGrabber grabber =
new PixelGrabber(image,0,0,20,10,false);
            System.out.println("Test 3");
```

First, a method is made with two method parameters that do not yet affect the code. Then, using an implemented library, the image is stored as the "image" variable. Next, a new object called "grabber" is created to store data from individual pixels. To start, it is filled with default information. The size of the picture is also set here. All "System.out.println()" are used to test if the code has passed that point. They simply print out the words in the quotes if the program has reached that far. The "try" found above "grabber" is used so that the code is only run if there are no exceptions. If there is an exception, the program will be redirected instead of crashing. The code that is used to catch exceptions will be explained later and, additionally, can be viewed under *Appendices* as *Appendix B*.

```
if(grabber.grabPixels())
 {
     int width = grabber.getWidth();
     int height = grabber.getHeight();
     System.out.println(width + ", " + height);
```

The if statement at the beginning of the above code is used to ensure that the following code will only execute if there are pixels available to analyze. If there are pixels present, then the following code obtains the height and width of the selected pixels and prints out the values.

```
if (isGreyscaleImage(grabber))
                {
                   byte[] data = (byte[]) grabber.getPixels();
                    System.out.println("Image is greyscale.");
                }
```

The if statement at the beginning of the above section of code is used in order discern between grayscale and color pictures. If the pixels stored in “grabber” are grayscale, the code will be enacted, and the pixels will be put into an array made out of bytes. If the pixels are not grayscale, the above code will be ignored, and the below code will be followed instead.

```
 else
     {
      int[] data = (int[]) grabber.getPixels();
      int x = 0;
      int y = 1;
      int z = 0;
      Color color = new Color( x,  y, z);
      String rgb = Integer.toHexString(color.getRGB());
```

The above code is only run if the image is in color. To start, an array of integers is created with the information from the pixels. Following that, we set our own values and then use them to create a color that mimics the possible colors of the pixels. This color is then changed to a hex value in order to be stored in a string data type. In our completely finished code, we would not need to create a fake color, but would instead be able to analyze this color from the pixels stored. We created our own test color for the purpose of continuing on with the code and getting as far as possible and achieving as much as possible.

```
            for(int i = 0; i<rgb.length();i++)

            {
            System.out.println("The color of the pixel at position
" + i++ + " is " + rgb.substring(i,i++));
            }


                }
             }
```

After the hex value of the color of each pixel has been stored in the string "rgb", the above code goes through the entire code and prints out the color of each pixel individually, with the placement of the pixel noted.

```
             else
            {
             System.out.println("Test 5");
            }
}
```

This part of the program is run if the first if statement, shown under *Appendices* as *Appendix C,* is not true; in other words, if no pixels can be found. It simply prints a message so the user knows that there are no pixels available to be analyzed.

```
     catch (InterruptedException e1)

         {
            e1.printStackTrace();
        }
   }
}
```

If there is an exception with the try statement, the code will jump to the catch statement until a solution to the exception can be found. This prevents the program from crashing and allows for the program to continue running after a solution is found for the problem.

The main issue with this code is that we have yet to find a way to gather the color from the pixel itself. We know we can collect each individual pixel, and we know that we can plug a color into our program to have it printed out, but we could not find a way to connect the two pieces to make one full program. Because of this, we moved on and created many different alterations, as well as entirely new programs to try to fix this problem. After many different attempts, we came up with a final code that does everything we need it to do to make a digital map of the forest fire.

Final Code

After trying many different approaches to fill the array with the colors stored in actual pixels, we decided that starting a new code entirely from scratch would be more efficient than continuing to work with the code we already had. After many failed attempts, we finally came up with one that was able to print out the red, green, and blue values of each individual pixel. From there, we were able to associate the majority of the pixels with a color, and from that, associate them with a temperature.

As with the first code, our final code will be broken down piece by piece with explanation. The full code can be found under *Appendices* as *Appendix D.*

```
import java.io.*;
import java.awt.*;
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
```

Once again, we started off by importing multiple libraries, but for our final code we did not use PixelGrabber, as we did in our first code. We made this decision because we could find few java mentors who had worked with PixelGrabber before, and information about it was limited. This time, we used both ImageIO and BufferedImage to work with an image in our code. The other two libraries are commonly used in most java programs.

```
public class ATC55_Final_Code

{
  public static void main(String args[]) throws IOException
   {
      File file= new File("fire2.jpg");
      BufferedImage image = ImageIO.read(file);
```

```
     //[10] Shadow, Black (2014)GetPixelColor [Source code]
<https://stackoverflow.com/questions/22391353/get-color-of-each-
pixel-of-an-image-using-bufferedimages>.
      int width = image.getWidth()-1;
      int height = image.getHeight()-1;
      System.out.println("The width is " + width+1 + " and the
height is " + height+1 + ".");
```

Next, we took our picture, made it into a file, and then converted the file into an image using the BufferedImage class. We then used the methods getWidth and getHeight from the BufferedImage class to get the dimensions of the image. These were then printed out.

```
for(int i = 0; i <= width; i++)
         {
            for(int j = 0; j<= height; j++)
               {
                  int color=  image.getRGB(i,j);//Width goes
first, then height.
                  int  red   = (color & 0x00ff0000) >> 16;
                  int  green = (color & 0x0000ff00) >> 8;
                  int  blue  =  color & 0x000000ff;
     //[10] Shadow, Black (2014)GetPixelColor [Source code]
<https://stackoverflow.com/questions/22391353/get-color-of-each-
pixel-of-an-image-using-bufferedimages>.
                  System.out.println("The pixel at " + i + "," +
j + " has a red value of " + red + ", a green value of " + green
+ ", and a blue value of " + blue + ".");
```

Then, using a nested for loop, we went through each pixel of the image. The first for loop increases the x value of the picture by one after the inner for loop has increased the y value until it reaches the final pixel in that row. This process repeats until the program reaches the pixel in

the lower right corner of the image. For each pixel, the red, green, and blue values of its color is gathered. These values are gathered in the lines

```
int  red   = (color & 0x00ff0000) >> 16;
int  green = (color & 0x0000ff00) >> 8;
int  blue  =  color & 0x000000ff;
```

All three colors are stored together in the variable color, and then in the first line,

```
0x00ff0000
```

is used to extract the red value from the three values being stored together. The zeros are used to mask all the other color values, and the two ff's, which are hexadecimal for 256, allow the full value of red to be present. The

```
>>16
```

is used to move the value to right 16 positions so there is no blank space left over from where the other values were previously. The two other lines follow the same process, just masking out the other values so that specific color is the only one that can be seen. Because the blue value is already at the end, there is no need to shift it over either 16 or 8 spaces, so

```
>> 16
```

or

```
 >>8
```

is missing. All three values are then printed out for each pixel, with labels.

```
          //Values associated with Bright White

                  if(red>228&&red==green&&green==blue)
                    brightWhite();
                  else if(red>=240&&green>245&&blue>240)
                    brightWhite();
```

After that, the program checks if the pixel has the red, green, and blue values that are associated with bright white. If the pixel does have any of these exact values, the brightWhite class will be called, where it prints a statement saying that the pixel is white, as well as the approximate temperature of that pixel based on its color. If the pixel does not match any of these values, it will move on to dull white and so on, until it reaches light blue. If by then the pixel has not matched with any of the colors, nothing will be printed. Many of the colors also have ranges of values associated with them, not just specific numbers. For example, dark red, as shown below, has ranges of values for red, green, and blue associated with it.

```
 //Values associated with Dark Red

                  else if(red<180&&red>130&&green<30&&blue<30)
                     darkRed();
                  else
if((red>90&&red<130)&&(green<19)&&(blue<10))
                     darkRed();
```

References [11], [12], [13], and [14] were used to associate red, green, and blue values with specific colors. After translating the red, green, and blue values into actual, readable colors, we researched how the temperature of fires can be discerned from the color of the flame [15], [16]. This allowed us to assign a temperature value to each of the possible colors of the pixels. The temperatures are all rough estimates.

To test the validity of the code, we tested aerial pictures of old fires. When cross referenced with the picture, the code was able to produce accurate readings of the color of each area, and the temperatures associated with each were also found to be correct.

Real World Application

Considering the fact that firefighters currently have to fly over wildfires to map them out and decide how to attack them, their data isn't always accurate and they can lose their lives too easily. Plus, flying over the fires and creating maps of them is time consuming, so realistically, they can only fly over the fire once a day. This means that they cannot provide real-time data, even though wildfires can change drastically within a few hours.

If firefighters were able to use our code in combination with drone technology, information about wildfires could be gathered through a much safer, quicker, and easier process. The information could also be quickly shared across many devices. This would result in the information being shared much more efficiently and could greatly reduce the amount of lives put in danger while firefighters are waiting to receive information. Also, firefighters wouldn't have to spend so much time and effort to map out the fires and would be able to put that time into actually fighting the fire. With being able to redirect that effort, their time would be spent much more efficiently and effectively.

Future Work

Because we did not achieve one hundred percent of our desired results, we plan to expand our project into a two-year project. Although half of the group's members will be graduating, the other half will continue on with the project, hopefully eliciting new mentors and other support in order to achieve full completion of the project.

When fully completed, our project should be able to stitch together multiple drone pictures in the same code as is used to analyze the picture. Our vision is that the drone will be able to take multiple pictures of the fire and our code will be able to take them all in and stitch them together to create a full image of the fire, where all angles can be analyzed for the most accurate information.

Not only will this increase the overall efficiency of the project, as well as making it more streamlined, but it will also increase the number of drones compatible with the program, making it cheaper and more easily available to a wider audience. This is because, if the images can be stitched together in the same code, it will not be necessary to get the drone as close to the fire in order to get a clear picture, as multiple pictures from a farther distance can be stitched together to achieve a complete picture. Because the drone will not need to get as close, it will not need to withstand as high a temperature, so lower quality, cheaper drones can be used just as effectively.

Also, we would like to continue improving our code. In our final code, we were able to take in an image and go through the picture, pixel by pixel, to analyze the color in terms of rgb values. We then associated ranges of rgb values with colors, and from there associated the colors with temperatures. For the best results, we would like to continue working on our code so that we can come up with a wider range of colors, so that every pixel is ensured to be associated with

one. We would also like to use more specific measurements of temperature to associate with each color. This will make our results closer to the actual fire itself, and make the work of the firefighters more efficient.

Additionally, we would like to find a way to make this code available to firefighters. This way, less human lives would be put at risk and the overall process will be faster. This would involve finding a way for the information to be shared across many devices nearly instantaneously, allowing for the process to be made much more effective.

References

[1] “U.S. Fire Statistics.” *U.S. Fire Administration*, 18 Dec. 2017, <www.usfa.fema.gov/data/statistics/>.

[2] Thiessen, Mark. “Wildfires Information and Facts.” *National Geographic*, 18 Jan. 2018, <www.nationalgeographic.com/environment/natural-disasters/wildfires/>.

[3] “Maps.” *NM Fire Info*, WordPress, 12 June 2017, <nmfireinfo.com/links/maps/>.

[4] “What Is an IR Satellite Image?” *UCLA Atmospheric & Oceanic Sciences*, UC Regents, <www.atmos.ucla.edu/weather/satellite/about>.

[5] “National Infrared Satellite .” *Intellicast*, TWC Product and Technology, <www.intellicast.com/National/Satellite/Infrared.aspx>.

[6]“Active Fire Mapping Program.” *Remote Sensing Applications Center*, USDA Forest Service , <fsapps.nwcg.gov/afm/faq.php>.

[7] “Understanding Drones.” *Friends Committee on National Legislation*, FCNL, <www.fcnl.org/updates/understanding-drones-43>.

[8] Taub, Ben. “Firefighting Drone Can Fly, Climb And Withstand Intense Heat.” *IFLScience*, IFLScience, 20 Mar. 2018, <www.iflscience.com/technology/firefighting-drone-can-fly-climb-and-withstand-intense-heat/>.

[9] Tips, Java PixelGrabberTest [Source code] <https://www.java-tips.org/java-se-tips-100019/23-java-awt-image/1950-how-to-use-pixelgrabber-class-to-acquire-pixel-data-from-an-image-object.html>.

[10] Shadow, Black (2014) GetPixelColor [Source code]
<https://stackoverflow.com/questions/22391353/get-color-of-each-pixel-of-an-image-using-bufferedimages>.

[11] Walsh, Kevin J. "RGB to Color Name Mapping (Triplet and Hex)." *RGB to Color Name Mapping (Triplet and Hex)*, 2010,
<web.njit.edu/~kevin/rgb.txt.html#columns>.

[12] *The Other RGB Color Chart*, Taylored Marketing, Jan. 2003,
<www.tayloredmktg.com/rgb/#YE>.

[13] "RGB Color Codes Chart." *Rapid Tables*, RapidTables,
<www.rapidtables.com/web/color/RGB_Color.html>.

[14] "RGB Explorer." *Stanford*,
<web.stanford.edu/class/cs101/image-rgb-explorer.html>.

[15] Maggio, Maggie. "Fire II: Color and Temperature." *Maggie Maggio*, 30 Aug. 2011,
<maggiemaggio.com/color/2011/08/fire-ii-color-and-temperature/>.

[16] *Flame Colors*. Polymer Science Learning Company,
<pslc.ws/fire/howwhy/flameco.htm>.

Acknowledgements

We thank Talaya White, for her assistance and input on our code, specifically working with colored images.

We thank Kevin Coen, our Computer Science teacher at ATC, for helping us understand our code and helping us work with new methods.

We would also like to acknowledge SimTables and their previous work being done to map wildfires. They have greatly influenced our focus for this project.

Additionally, we would like to thank Drew Einhorn for his insightful feedback.

Finally, we would like to thank our teacher Jenifer Hooten for all of the support and enthusiasm she has provided us throughout the duration of our project.

Appendices

Appendix A:

```
// Based off of: Tips, Java PixelGrabberTest [Source code]
<https://www.java-tips.org/java-se-tips-100019/23-java-awt-image
/1950-how-to-use-pixelgrabber-class-to-acquire-pixel-data-from-a
n-image-object.html>.
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.image.PixelGrabber;
import java.awt.Color;
import java.awt.image.ImageConsumer;
import java.io.*;
import java.util.*;
import java.util.Arrays;

public class PixelGrabberTest
{
   public static void processImage(String inFile, String
outFile) {
        System.out.println("Test 1");
 Image image = Toolkit.getDefaultToolkit().getImage("Fire.jpg");
//Takes in the image.
        System.out.println("Test 2");//This tests if the program
made it this far. We have them set throughout the program.
       try {
           PixelGrabber grabber = new PixelGrabber(image, 0,
0,20,10,false);//Creates a new variable that stores the image we
sent it as well as default information about its size.
            System.out.println("Test 3");//This tests if the
program made it this far.
```

```
          if (grabber.grabPixels())

          // Used so the program only runs if there are pixels
present.
           {
               int width = grabber.getWidth();
               int height = grabber.getHeight();
               System.out.println(width + ", " + height);
          // Takes the height and width of the given pixels and
prints the values.

              if (isGreyscaleImage(grabber))
                {
                   byte[] data = (byte[]) grabber.getPixels();
                    System.out.println("Image is greyscale.");
                   // Processes greyscale image. If the image is
Greyscale, it will come here, and then the pixels will be
obtained from that image.
               }
               else

          {
          //If the image is in color it will come here. Right
     now, it has been coming here.
               int[] data = (int[]) grabber.getPixels();

//Fills an array with the information from the pixels.
                  int x = 0;
                   int y = 1;
                   int z = 0;
                  Color color = new Color( x,  y, z);
              String rgb = Integer.toHexString(color.getRGB());
//We created our own color to test if we could get the RGB hex
values from the array we filled. It does work; however, the
```

values we are getting of course are just the ones we manually entered (0,1,0 above).

```
     for(int i = 0; i<rgb.length();i++)
          {
          System.out.println("The color of the pixel at position
" + i++ + " is " + rgb.substring(i,i++));
          }
// The loop is used to print the value stored for each
individual pixel at the index of the array.
               }
           }
           else
          {
           System.out.println("Test 5");
          }
//Only runs if there are no pixels available for analysis.
       }
      catch (InterruptedException e1)
        {
           e1.printStackTrace();
       /* Runs when there is an exception for the try statement.
Once solution is found, goes back and runs through program as
originally intended. Prevents program from crashing. */
     }
   }
}
```

Appendix B:

```
    catch (InterruptedException e1)
        {
           e1.printStackTrace();
       /* Runs when there is an exception for the try statement.
Once solution is found, goes back and runs through program as
originally intended. Prevents program from crashing. */
     }
```

Appendix C:

```
  if (grabber.grabPixels())
          // Used so the program only runs if there are pixels
present.
           {
               int width = grabber.getWidth();
               int height = grabber.getHeight();
               System.out.println(width + ", " + height);
          // Takes the height and width of the given pixels and
prints the values.

              if (isGreyscaleImage(grabber))
                {
                   byte[] data = (byte[]) grabber.getPixels();
                    System.out.println("Image is grayscale.");
                   // Process grayscale image. If the image is
Grayscale, then it will come here, and then the pixels will be
obtained from that image.
               }
```

Appendix D:

```
import java.io.*;
import java.awt.*;
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;

public class ATC55_Final_Code
{
  public static void main(String args[]) throws IOException
   {
      File file= new File("fire.jpg");
      BufferedImage image = ImageIO.read(file);
      //Shadow, Black (2014) GetPixelColor [Source code]

//<https://stackoverflow.com/questions/22391353/get-color-of-eac
h-pixel-of-an-image-using-bufferedimages>.
      int width = image.getWidth()-1;
      //One is subtracted from the width because pixels start at
0,0.
      int height = image.getHeight()-1;
      //One is subtracted from the height because pixels start
at 0,0.
      System.out.println("The width is " + width+1 + " and the
height is " + height+1 + ".");
      //One is added back to the width and height to account for
one taken away previously.

      for(int i = 0; i <= width; i++)
      //Increases width by one.
         {
            for(int j = 0; j<= height; j++)
            //Increases height by one.
               {
                  int color=  image.getRGB(i,j);
                  //Width goes first, then height.
                  int  red   = (color & 0x00ff0000) >> 16;
                  //Masks all colors except for red, then moves
the value 16 spaces over.
```

```
                int  green = (color & 0x0000ff00) >> 8;
                //Masks all colors except for green, then
moves the value 8 spaces over.
                int  blue  =  color & 0x000000ff;
                //Masks all colors except for blue.
                //Shadow, Black (2014) GetPixelColor [Source
code]

//<https://stackoverflow.com/questions/22391353/get-color-of-eac
h-pixel-of-an-image-using-bufferedimages>.
                System.out.println("The pixel at " + i + "," +
j + " has a red value of " + red + ", a green value of " + green
+ ", and a blue value of " + blue + ".");

                //Values associated with Bright White
                if(red>228&&red==green&&green==blue)
                  brightWhite();
                else if(red>=240&&green>245&&blue>240)
                  brightWhite();

                //Values associated with Dull White
                else
if((red>=217&&red<228)&&(green==red)&&(blue==green))
                   dullWhite();

                //Values associated with Deep Orange
                else
if((red>240&&red<250)&&(green>105&&green<115)&&(blue>=0&&blue<10
))
                   deepOrange();
                else
if((red>120&&red<190)&&(green>15&&green<60)&&(blue>=0&&blue<10))
                   deepOrange();
                //Values associated with Bright Orange
                else
if((red>=190)&&(green>30&&green<115)&&(blue>=0&&blue<15))
                   brightOrange();
                else if(red>200&&green<150&&blue<50)
                   brightOrange();
```

```
                    //Values associated with Yellow
                    else if(red>240&&green>=165&&blue<50)
                       yellow();
                    else if(red>250&&green>230&&blue<150)
                       yellow();

                    //Values associated with Dark Red
                    else if(red<180&&red>130&&green<30&&blue<30)
                       darkRed();
                    else
if((red>90&&red<130)&&(green<19)&&(blue<10))
                       darkRed();

                    //Values associated with Bright Red
                    else if(red>180&&green<35&&blue<35)
                       brightRed();

                    //Values associated with Black
                    else if(red<20&&green<20&&blue<20)
                       black();

                    //Values associated with Brown
                    else
if((red<=90&&red>15)&&(green>=0&&green<11)&&(blue>=0&&blue<10))
                       brown();

                    //Values associated with Light Blue
                    else
if((red>150&&red<250)&&(green>150&&green<250)&&blue>200)
                       lightBlue();

                }

            }

        }
      public static void brightWhite()
      {
```

```
   //Called when red, green, and blue values are seen as bright
white.

      System.out.println("The color is bright white.");
      System.out.println("The temperature is around 1,400°C.");
   }
   public static void dullWhite()
   {
   //Called when red, green, and blue values are seen as dull
white.
      System.out.println("The color is dull white.");
      System.out.println("The temperature is around 1,350°C.");
   }
   public static void deepOrange()
   {
   //Called when red, green, and blue values are seen as deep
orange.
      System.out.println("The color is deep orange.");
      System.out.println("The temperature is around 1,100°C.");
   }
   public static void brightOrange()
   {
   //Called when red, green, and blue values are seen as bright
orange.
      System.out.println("The color is bright orange.");
      System.out.println("The temperature is around 1,200°C.");
   }
   public static void yellow()
   {
   //Called when red, green, and blue values are seen as yellow.
      System.out.println("The color is yellow.");
      System.out.println("The temperature is around 1,300°C.");
   }
   public static void darkRed()
   {
   //Called when red, green, and blue values are seen as dark
red.
      System.out.println("The color is dark red.");
      System.out.println("The temperature is around 700°C.");
```

```
   }
   public static void brightRed()
   {
   //Called when red, green, and blue values are seen as bright
red.
      System.out.println("The color is bright red.");
      System.out.println("The temperature is around 950°C.");
   }
   public static void black()
   {
   //Called when red, green, and blue values are seen as black.
      System.out.println("The color is black.");
   }
   public static void brown()
   {
   //Called when red, green, and blue values are seen as brown.
      System.out.println("The color is brown.");
   }
   public static void lightBlue()
   {
   //Called when red, green, and blue values are seen as light
blue.
      System.out.println("The color is light blue.");
      System.out.println("The temperature is around 1500°C.");
   }
}
```