AN INTELLIGENT SYSTEM TO ASSIST THE DRAWING COMPLETION AND COLORING SUGGESTION USING AI AND IMAGE PROCESSING TECHNIQUES

Jichuan Wang[1] and Yu Sun[2]

[1]Northwood High School, 4515 Portola PKWY, USA
[2]California State Polytechnic University,
Pomona, CA, 91768, Irvine, CA 92620, USA

ABSTRACT

Many times when we create any design or drawing on a digital platform, we face the problem that lines we created couldn't close up the shape correctly [1]. When we use the paint bucket tool, it fills the whole screen up with one click, and we need to re-look through the whole design again to find where it wasn't closed up correctly, not only taking up much of our time, but it's also very annoying. By determining a rule for whether or not a shape is completely closed or open, this application checks whether the shapes we created were closed by going through each pixel and detecting if there are holes around it [2]. The application could be provided for anyone creating 2d designs on a computer whether for hobby or job, to save the time looking through the whole design again and find the errors with much time are really tiny.

KEYWORDS

Shapes, checker, fixer, pixel extraction.

1. INTRODUCTION

As using any designing app that includes a tool to select or fill an area, the tool such as paint buckets often would take account of any tiny holes on the outline. This, in return, messes up the whole design by filling up or selecting not only the area wanted, but also the whole opening area connected to that tiny opening. When this occurs, you need to adjust it by hand, whether through erasing or finding and closing up the opening yourself, as it's more annoying to fill in the area using a pen tool which may draw on top or over the outlined shape [3]. And because of this, you often have to look through everything in the image to find the location of the problem, a process that could be even more difficult when the complexity of the design increases [4]. This is opposed when people have to look through the whole canvas and in the end find the opening at the smallest tip over, for example, the character's hair. Maybe over time people could get better at making designs to make no holes on the outline, but it's a common problem for most people who weren't trained for years, and they often thought they have completed their design, only to find that when filling in the desired color, the whole page is filled up with the color, and possibly covering other work they have already done. This can speed up the process of creating designs by letting people not put too much of their mind on finding problems in the closure of their design, and allow for more advanced novice design [5].

Some of the techniques and systems that have been proposed until now repeat and draw out existing shapes that allow the user to input their drawing into the program and draw the shape out, providing a canvas that allows the user to edit [7]. However, this implementation was limited as the user could only see the shape they drew. Furthermore, various other software applications only assist the process further by giving the user the tools to manually fix the problems with the image. This doesn't actually help the user with the original problem they had of wanting to fix their shape, but instead just draws what the user gives and asks the user to fix and find the problems themselves [8]. Tools such as Onshape and Goodnotes provide these useful tools, but do not provide fixtures to incomplete shapes. They only support fully closed shapes.

In this paper, we follow the same idea that helps the user make their shape more complete for further steps in creating their art. In this case, we have decided to utilize pixel art, or art that is drawn using single pixels that connect to form a complete image, for the drawings that are checked in the program. Instead of providing tools for users to manually do the drawing themselves, we first check their image for pixels in the shape that do not connect to other pixels, forming an incomplete shape. We implement a steady algorithm to check eight different directions that adjacent pixels can be from any given pixel in the image. Second, we tell the user where the problems are located in their image. In other words, we describe which pixels are not connected to other pixels in the image that cause our shape to be incomplete. Third, we help them fix their problem by filling in any gaps that they missed in their pixel drawing. We implement a similar algorithm to the one that determines completeness or incompleteness of a shape; however, we use this algorithm to now determine a start point and an end point to which a line will connect the two incomplete pixels from.

We used the graphing calculator to plot out all the coordinates that our program went through that determined the shape's pixels in a given image. With this graph, we saw what pixels were analyzed and how they were interpreted, and how closely it was to our predicted result. In terms of how the pixels were interpreted, we were using the graph to see if the program could extract all the colored pixels that we were supposed to analyze (this could be easily proven to be correct with closed images, since any gaps would prove an incorrect analysis and/or extraction in the program). If the plots on the graphing calculator are the same shape from the original picture (i.e. the coordinates on the graphing calculator match given coordinates extracted from the image in the program), we have proven that we can determine whether the shapes were closed or opened.

The rest of the paper is organized as follows: Section 2 gives the details on the challenges that we met during the experiment and implementing solid tests that check for the overall correctness of our project; Section 3 focuses on the details of our solutions corresponding to the challenges that we mentioned in Section 2; Section 4 presents the relevant details about the experiment we did, following by presenting the related work in Section 5. Finally, Section 6 gives the conclusion remarks, as well as pointing out the future work of this project.

2. CHALLENGES

In order to build the tracking system, a few challenges have been identified as follows.

2.1. Avoid Confusion

When implementing file-opening components in a program, we want to make sure that the file we open is the one we desire to check its shape for. To omit this confusion, we name our files specifically beforehand, naming them "complete"/open or "incomplete"/closed [6]. In real-world terms, the user will not know whether or not their shape is open or closed, and they are going to want the confidence of knowing that it is their shape that is being checked (not some other file); that is why they will use our application to check their shape. However, we use the os Python library, which guarantees file safety measures to ensure 100% accuracy regarding opening the correct file.

2.2. Identify the Pixel

For the Pixel Fixer design we made to close the opened shapes, we wanted to make sure that there are no pixels in the shape that do not meet the adjacency requirements (2 pixels adjacent). To identify a pixel to be an open pixel, the program checks for an adjacent pixel in eight different directions; if the number of adjacent pixels is less than two, the pixel was identified as an incomplete (or open) pixel. To make sure the pixels that were connecting are the incomplete pixels, we initialize an empty list and append each pixel that is identified as incomplete pixels into this list. When the user clicks to complete the image, the turtle object is placed at the first pixel in the list within the GUI and the program will go to the closest pixel. Once the turtle pen finishes its job of drawing, the program indicates a complete drawing in the GUI (there are no gaps in coordinates, and this is where incomplete images are most prominently indicated).

2.3. Make Sure the Image is Complete

In the program, we must make sure that an image that was identified as complete is actually a complete shape. The program will scan and record the coordinates of each pixel into an empty list that has a color that isn't white (this is supposed to be the color of the canvas). The program will go through every pixel in the list and check for the eight adjacent directions and see if these pixels are in the list of colored pixels. The pixel could only be identified as a closed pixel if two or more pixels adjacent were colored, and a shape will only be identified as closed if all pixels were identified as closed [9]. And when the shape was identified as closed, the program will then go on and display that the shape is a closed shape.

3. SOLUTION

Pixel Checker is an application that utilizes python libraries including turtle and tkinter. The Pixel retrieving component includes implementation of the PIL library, which is an image-based library used for photo-related data retrieval. The program, after retrieving the file in jpg format, goes through each pixel in the image using a for loop, and if a pixel's color isn't equal to white, or 0,0,0,0, the program will append the coordinate x and y of the pixel into a list for other parts of the program to use (this means that every coordinate in the list is represented as a tuple). Once all pixels have been retrieved, the program goes through a pixel checking process, which goes through each pixel to see if there are two pixels that are adjacent to the current pixel in the list. If a pixel has less than two colored pixels adjacent to it, it will be identified as an open pixel, and will be handled accordingly to fix the shape. The program only indicates a shape to be closed if and only if all of it's pixels are closed pixels; otherwise, the program indicates that the shape is open. The shape drawing component implements the turtle library, where the turtle will draw out lines when it's commanded to put it's pen down, it is implemented so that it puts its pen down as the first pixel of the list of colored pixels, and goes through all the pixels that were colored. Each pixel that was drawn was appended into another separate list to ensure the turtle didn't go on to redraw any pixel that was drawn. The shape fixer is similar to the shape drawer, in the sense that it goes through a list of pixels to move the turtle to its indicated location. Only in this component, however, the program goes through the list of opened pixels and draws it to the closest colored pixel to close up the shape. While there is a small portion of information being displayed onto the Console after drawing the image, the visual parts of the program that displays the drawing were put together with the GUI using the tkinter library (from a turtle screen).The GUI utilizes the screen with buttons for the user to interact with. The user can upload and select an image from a drop down menu. When the user clicks on the check image, the check image and draw image functions will be enacted to check the image for its closeness, and send an error message when there wasn't an image chosen. The program will draw out the image first, and if the shape is not closed, a pop up window will appear to tell the user that the shape isn't closed, and the user can click on the fix image button to fix and redraw a complete shape. Once the shape was identified as completed, a button will appear and show how the program goes through and check all the pixels of the image. After this, the program resets, the user can select another image to check.

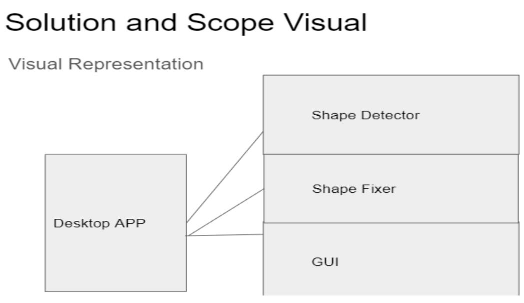
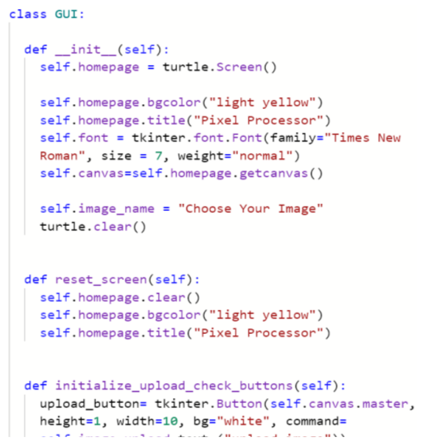
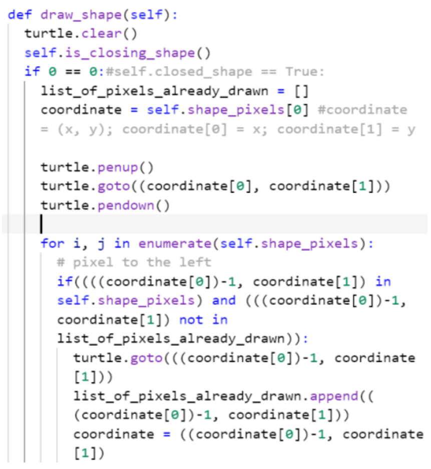


Figure 1. Overview of the system



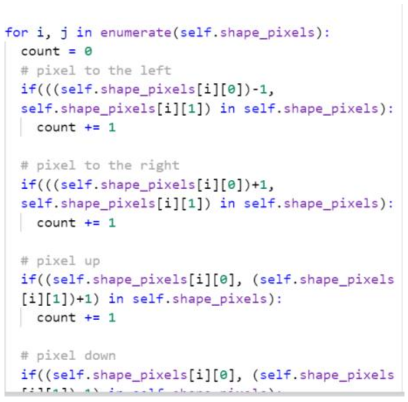
```
class GUI:

    def __init__(self):
        self.homepage = turtle.Screen()

        self.homepage.bgcolor("light yellow")
        self.homepage.title("Pixel Processor")
        self.font = tkinter.font.Font(family="Times New
        Roman", size = 7, weight="normal")
        self.canvas=self.homepage.getcanvas()

        self.image_name = "Choose Your Image"
        turtle.clear()


    def reset_screen(self):
        self.homepage.clear()
        self.homepage.bgcolor("light yellow")
        self.homepage.title("Pixel Processor")


    def initialize_upload_check_buttons(self):
        upload_button= tkinter.Button(self.canvas.master,
        height=1, width=10, bg="white", command=
```

Figure 2. Code of class GUI



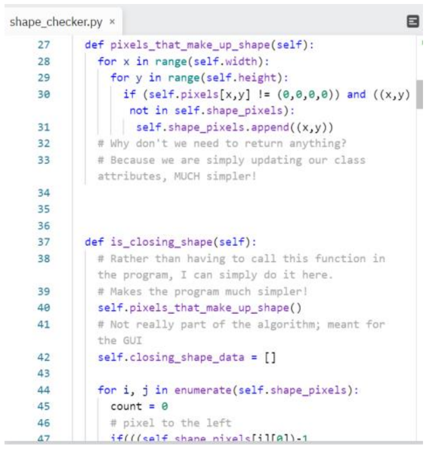
```
def draw_shape(self):
    turtle.clear()
    self.is_closing_shape()
    if 0 == 0:#self.closed_shape == True:
        list_of_pixels_already_drawn = []
        coordinate = self.shape_pixels[0] #coordinate
        = (x, y); coordinate[0] = x; coordinate[1] = y

        turtle.penup()
        turtle.goto((coordinate[0], coordinate[1]))
        turtle.pendown()

        for i, j in enumerate(self.shape_pixels):
            # pixel to the left
            if((((coordinate[0])-1, coordinate[1]) in
            self.shape_pixels) and (((coordinate[0])-1,
            coordinate[1]) not in
            list_of_pixels_already_drawn)):
                turtle.goto((((coordinate[0])-1, coordinate
                [1]))
                list_of_pixels_already_drawn.append((
                (coordinate[0])-1, coordinate[1]))
                coordinate = ((coordinate[0])-1, coordinate
                [1])
```

Figure 3. Code of draw shape



```
for i, j in enumerate(self.shape_pixels):
  count = 0
  # pixel to the left
  if(((self.shape_pixels[i][0])-1,
  self.shape_pixels[i][1]) in self.shape_pixels):
    count += 1

  # pixel to the right
  if(((self.shape_pixels[i][0])+1,
  self.shape_pixels[i][1]) in self.shape_pixels):
    count += 1

  # pixel up
  if((self.shape_pixels[i][0], (self.shape_pixels
  [i][1])+1) in self.shape_pixels):
    count += 1

  # pixel down
  if((self.shape_pixels[i][0], (self.shape_pixels
```

Figure 4. Code of enumerate



shape_checker.py ×

```
def pixels_that_make_up_shape(self):
  for x in range(self.width):
    for y in range(self.height):
      if (self.pixels[x,y] != (0,0,0,0)) and ((x,y)
       not in self.shape_pixels):
        self.shape_pixels.append((x,y))
  # Why don't we need to return anything?
  # Because we are simply updating our class
  attributes, MUCH simpler!



def is_closing_shape(self):
  # Rather than having to call this function in
  the program, I can simply do it here.
  # Makes the program much simpler!
  self.pixels_that_make_up_shape()
  # Not really part of the algorithm; meant for
  the GUI
  self.closing_shape_data = []

  for i, j in enumerate(self.shape_pixels):
    count = 0
    # pixel to the left
    if(((self.shape_pixels[i][0])-1
```

Figure 5. Code of pixels that make up shape

4. EXPERIMENT

4.1. Experiment 1

We check for every pixel in the picture that was colored (a pixel with a different RGB color value than the background's RGB color value). The program will go through each one of the shape pixels, and for the current pixel that we are checking, we check for the adjacent pixels around it (8 different directions). If there are two adjacent pixels to the current pixel that are in the shape pixels list (and thus meet the RGB value specification), we continue to check the next pixel in the list of shape pixels. If there is a pixel in the list of shape pixels that does not have two adjacent pixels to the current pixel that are in the shape pixels list, we end the function and the shape is not complete. If all of the pixels in the list of shape pixels meet the specified requirements, we can indicate that the shape is complete.

Using a professional graphing website: Desmos, we can specify all of the coordinates from the shape pixel (after projecting the coordinates in the program) to prove that there are two adjacent pixels in the direction. In the graph, the distance from each adjacent pixel should be no greater than 1 coordinate point away from each other.

That is, for a pixel, pixel1 that is adjacent to the pixel, pixel2, we expect that the distance equation below should follow for all of the pixels in the list of shape pixels:

$\text{Distance}_i \leq 1$, or:
$|\text{pixel1}(x) - \text{pixel2}(x)| \leq 1$ or $|\text{pixel1}(y) - \text{pixel2}(y)| \leq 1$
Coordinates from square_complete.py that are in shape_pixels list:
[(20, 20), (20, 21), (20, 22), (20, 23), (20, 24), (20, 25), (20, 26), (20, 27), (20, 28), (20, 29), (20, 30), (20, 31), (20, 32), (20, 33), (20, 34), (20, 35), (20, 36), (20, 37), (20, 38), (20, 39), (20, 40), (21, 20), (21, 40), (22, 20), (22, 40), (23, 20), (23, 40), (24, 20), (24, 40), (25, 20), (25, 40), (26, 20), (26, 40), (27, 20), (27, 40), (28, 20), (28, 40), (29, 20), (29, 40), (30, 20), (30, 40), (31, 20), (31, 40), (32, 20), (32, 40), (33, 20), (33, 40), (34, 20), (34, 40), (35, 20), (35, 40), (36, 20), (36, 40), (37, 20), (37, 40), (38, 20), (38, 40), (39, 20), (39, 40), (40, 20), (40, 40), (41, 20), (41, 21), (41, 22), (41, 23), (41, 24), (41, 25), (41, 26), (41, 27), (41, 28), (41, 29), (41, 30), (41, 31), (41, 32), (41, 33), (41, 34), (41, 35), (41, 36), (41, 37), (41, 38), (41, 39), (41, 40)]

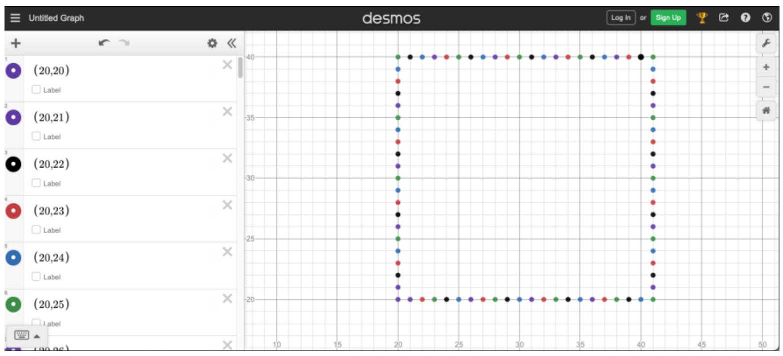


Figure 6. Square

For every pixel in the graph, there are two pixels in an adjacent direction with a distance (based on the formula above) of exactly 1; and thus, we have proven that the shape is complete [10].

4.2. Experiment 2

If the shape was identified as incomplete, (in other words, there is one pixel that did not have two adjacent pixels that were in the shape), a method in the shape checker known as draw_pixel is called. In this method, we start by storing all of the incomplete pixels to a separate list (this step is very similar to the is_complete_shape method; except now we are taking into consideration the incomplete pixels rather than the complete pixels). Using the GUI, the method will go to the specified coordinate of the first pixel in the list. Using a turtle, the turtle object goes to this coordinate. Next, the method moves onto the second pixel (the last pixel that is incomplete in the image), along with the turtle object moving forward to the specified coordinate in the GUI (it's important to note that the pen is down (turtle.pendown(), so there is a line being drawn from the first pixel in the list of incomplete pixels to the second pixel in the list of incomplete pixels). Once this line connects the two pixels together, we can then identify the shape as being complete. We know this to be true because for every x-y coordinate in the path of the line that was drawn, we assume there to be a coordinate in place so long as there are no gaps in the GUI within the shape's figure at this point.

Using a professional graphing website: Desmos, we can specify all of the coordinates from the shape pixel (after projecting the coordinates in the program) to prove that we have properly identified the two pixels that are open (incomplete). Open pixels are specified by the following requirement:
For every open pixel i in graph G:
$\sum(\text{adjacent pixels}_i) \leq 1$,
(Distance_i from any non adjacent pixel) > 1

[(34, 25), (34, 26), (34, 27), (34, 28), (34, 29), (34, 30), (34, 31), (34, 32), (34, 33), (34, 34), (35, 34), (36, 34), (37, 34), (38, 34), (39, 34), (40, 34), (41, 34), (42, 25), (42, 34), (43, 25), (43, 34), (44, 25), (44, 26), (44, 27), (44, 28), (44, 29), (44, 30), (44, 31), (44, 32), (44, 33), (44, 34)] ; coordinates for the line: (34, 25), (42, 25)

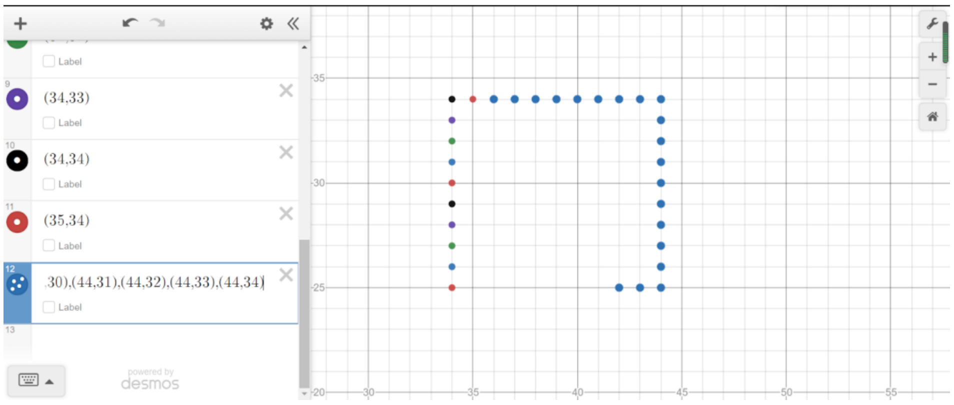


Figure 7. Desmos graph (screenshot after the line is drawn)

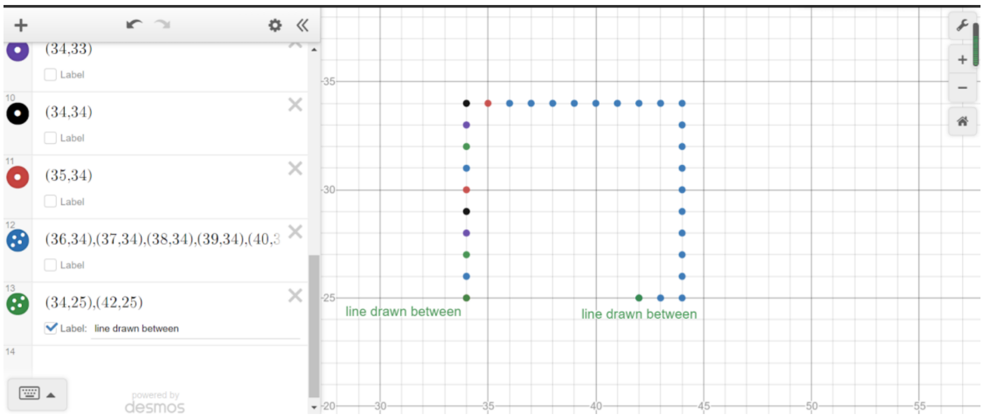


Figure 8. Desmos graph 2

We wanted to prove for the pixels that met the adjacency requirement, and based on the graphs for the image, that you can see these pixels have two other pixels adjacent with a distance on the graph next to each other equivalent to 1. For the pixels that did not meet the adjacency requirement, we can see that they have only one adjacent pixel with a distance of 1. The program identifies the two incomplete pixels, and it will draw a line between them (in the GUI we draw a connecting line between them). We have proven the summations of adjacent pixels for open and closed pixels, the distances for adjacent pixels for open and closed pixels (and how they determine a closed shape overall).

Through the experiment, we have proven that the program succeeds in determining whether the shape is closed or open, and how it could identify which pixels need to be connected with a line to close up the shape. The program will identify whether two of the pixels adjacent with a distance of 1 were colored to determine the pixel to be closed.

For example, the completed square could be identified correctly when every pixel in the shape has two adjacent pixels with a distance of 1; this requirement means that all of its pixels were complete.

In the incomplete square the program can identify the incomplete pixels (i.e. with only one pixel adjacent to it with distance of 1), and can draw a line from both pixels to connect them. The line drawn has no gaps (because a line is simply one solid shape that connects to the shape). And because a shape connects to another shape to close any gaps (and thus create another adjacent pixel with a distance of 1), our shape is complete after the line is drawn.

5. Related Work

Utilizes previously written data to fill in image gaps [11]. As stated in the article, "Based on the assumption that the same-class neighboring pixels around the un-scanned pixels have similar spectral characteristics, and that these neighboring and un-scanned pixels exhibit similar patterns of spectral differences between dates, we developed a simple and effective method to interpolate the values of the pixels within the gaps". They are attempting to fill in image gaps by selecting certain pixels. Given that they are searching for a similar pattern to fill in image gaps (scanned and unscanned pixels), we are searching for a similar pattern to fill in image gaps (pixels that meet a specific adjacency requirement to be determined as closed or open pixels). They are using a different form of technology to retrieve the pixels (they use Landsat ETM+, while we used a method of pixel extraction). And the images that they are testing are quite different as well (they use SLC-off ETM+ images, while we use .jpeg images created from a individualized pixel-focus drawing platform: pixlart). We can assume that they possibly used thousands of pixels, while our implementations, while they can handle this quantity of pixels, only used a few dozen.

The research tests how open and closed shapes affect the trend of judgement regarding the processing of shapes [12]. This research determines open and closed shapes for parts of their experiment. Rather than judging on whether the shape is open or closed, this tests how the open or closed shape affects the running time of finding a target that is a particular open or closed shape.

A method in Java attempting to extract edge end pixels for further image analyzing to link two edges together [13]. The research extracts pixels from images through methods focused on direction sensitives, as well as linking edges together from 2-d images (the edge segmentation on theirs, the line drawn on ours). The research was based on Java, while ours is based in python. This research focuses on analyzing the directions that the two edges are facing from each other, while our research gathers the information regarding the two edges without further analysis (simply draws a connecting line between the two).

6. Conclusions

The pixel checker utilizes a method to extract all colored pixels in a shape, and then holds on to certain pixels (the ones that make up the shape we are checking for) based on their RGB value [14]. The program will append all pixels into a list, and check through each of them. If each pixel has two pixels adjacent to it in any of the eight different directions horizontally, vertically, or diagonally (up, down, left, right, up-left, upright, down-left, or downright). If all the pixels in the shape meet this requirement, it will be identified as a closed pixel; otherwise, the pixel will be identified as open. The program will tell the user whether the shape is closed or not (using the specified method), and close the shape when it isn't. The program will close an open shape by appending all the open pixels into a separate list (in our experiments, there is a total of 2 pixels in the simple pixel-art based images). Then, we use a turtle.pen object to draw through all open pixels in the shape by going from the first pixel to the second pixel, which results in a line being drawn. Once the line is drawn, we have no pixels without two adjacent pixels in our shape, and thus we can conclude that we have closed an open image. Using images drawn on pixlart, we used the shape's pixels and plotted them on a graphing calculator. If the adjacency rules specified above in the experiments are met in the plotted points on the graph and in the coordinates in the program, we expected the results for both to determine closed shapes (our program would indicate this, while we would investigate every coordinate manually on the graphing calculator). For open shapes, we wanted to compare the two edge points (the two points that did not meet the adjacency requirement). If these two points did not meet the adjacency requirement manually by checking the graphing calculator and in the program, we can determine that our program correctly checks for both open and closed shapes.

For now, our program only draws lines through the pixels that are open, and cannot go back to check if the shape could be completed correctly. Our current program can only run simple pixel art images (our pixel checker's adjacency requirements allow us only to check for images with shapes that have two pixels adjacent to them without the possibility for more than this amount with complete accuracy). The current fixing method only closes up the shape by drawing a line that connects the two points together that are open. In terms of optimizing our program, we currently use brute-force to check for all of the given coordinates in our list of shape pixels. We have not implemented a dynamic-programming related concept nor a recursive algorithm to work our way through all of the pixels in the shape [15]. Doing so would drastically reduce the overall running time of the algorithm for the shape checker.

In the future, we could implement machine learning into the program to allow more advanced techniques to draw the pixels in our image and allow for more optimized solutions to fix open shapes. This will allow our program to handle more complex images, and to further our adjacency requirement for a higher accuracy rate on more complicated images (and hence we could possibly run this program on images that are not just single-pixels).

REFERENCES

[1] Geliskhanov, Islam Zelimkhanovich, and Tamara Nikolaevna Yudina. "DIGITAL PLATFORM: A NEW ECONOMIC INSTITUTION." Quality-Access to Success 19 (2018).

[2] Fisher, Peter. "The pixel: a snare and a delusion." International Journal of Remote Sensing 18.3 (1997): 679-685.

[3] Simader, Christian G. "Outline." On Dirichlet's Boundary Value Problem. Springer, Berlin, Heidelberg, 1972. 1-12.

[4] Gell-Mann, Murray. "What is complexity?." Complexity and industrial clusters. Physica-Verlag HD, 2002. 13-24.

[5] Hawthorne, John. "The case for closure." Contemporary debates in epistemology (2005): 26-43.

[6] D'MELLO, SIDNEY K., and Arthur C. Graesser. "Confusion." International handbook of emotions in education. Routledge, 2014. 299-320.

[7] Kendall, David George, et al. Shape and shape theory. Vol. 500. John Wiley & Sons, 2009.

[8] Osada, Robert, et al. "Shape distributions." ACM Transactions on Graphics (TOG) 21.4 (2002): 807-832.

[9] Sloan, Peter-Pike J., Charles F. Rose III, and Michael F. Cohen. "Shape by example." Proceedings of the 2001 symposium on Interactive 3D graphics. 2001.

[10] Hengl, Tomislav. "Finding the right pixel size." Computers & geosciences 32.9 (2006): 1283-1298.

[11] Chen, Jin, et al. "A simple and effective method for filling gaps in Landsat ETM+ SLC-off images." Remote sensing of environment 115.4 (2011): 1053-1064.

[12] Burlinson, David, Kalpathi Subramanian, and Paula Goolkasian. "Open vs. closed shapes: New perceptual categories?." IEEE transactions on visualization and computer graphics 24.1 (2017): 574-583.

[13] Pathegama, Mahinda, and Özdemir Göl. "Edge-end pixel extraction for edge-based image segmentation." ICSP, 2004.

[14] Süsstrunk, Sabine, Robert Buckley, and Steve Swen. "Standard RGB color spaces." Color and Imaging Conference. Vol. 1999. No. 1. Society for Imaging Science and Technology, 1999.
[15] Bellman, Richard. "Dynamic programming." Science 153.3731 (1966): 34-37.