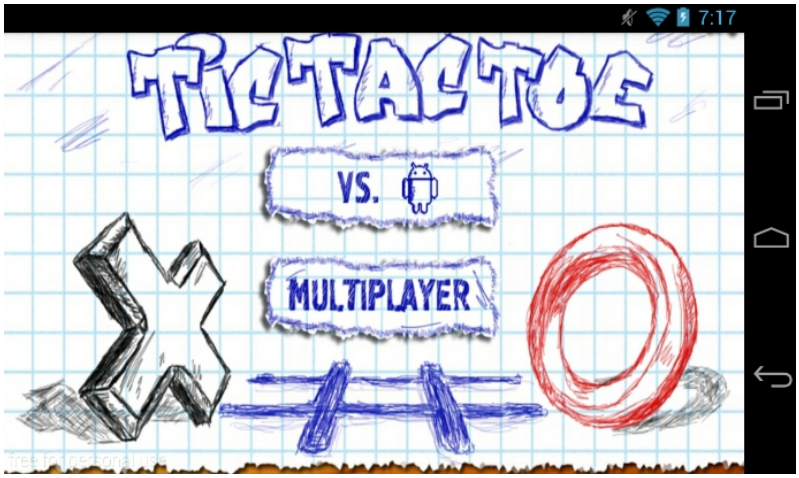
Phillip Godzin pgg2105

COMS W4735y: Visual Interfaces to Computers

Final Project: Tic Tac Toe CV

Introduction and How to Play

For this final project, I created an Android Tic-Tac-Toe game in which a player challenges either the Artificial Intelligence of the app or another human player for a multiplayer game.



Once the mode is selected, the player uses haptic input to draw their move on the screen. There are 7 possible moves a player can make: X, O, +, →, ♡, △, and ☐. The application waits for 1.5 seconds once the user lifts their finger before processing the move, allowing for multiple strokes. In each player's first move of every game, the identified shape will appear in the upper left hand corner. The player must continue drawing this shape for all the subsequent turns in that particular game. The user must wait until the next player's move is identified or until the AI

makes its move before drawing again. The game continues until one of the players win, force a tie, or make a mistake that forces the game to restart.

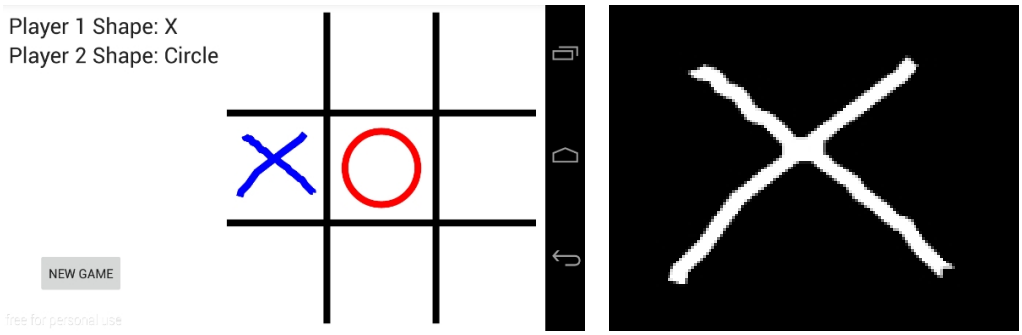
There are several cases that will force a restart. My initial goal was to erase the offending move in the case of error and allow the user to try again, but I was unable to get this functionality working well enough. Instead, the game will restart with a clean board.

The error cases are:

1. The move is not contained within a single cell on the board. If it touches the gridlines or ventures, even slightly, into a second cell, the game will restart.
2. The initial move is not recognized as one of the 7 possible shapes.
3. The subsequent moves do not match the user's initial move of the game.
4. The user drew in a cell that has already been played.

Isolating a Move

I used Android's Canvas functionality to register the moves drawn on the grid. Once a move is completed, I save the resulting Canvas view in a bitmap called board and compare it to a bitmap taken prior to the move being drawn, called oldBoard. I load the two bitmaps into openCV matrices, and subtract them using `Core.absdiff(oldBoard, newBoard, move);` From there, I create a new matrix diffMat. I place a white pixel into the matrix for every corresponding pixel in which the distance between two color values is greater than 30. For example, this process isolates the following move:

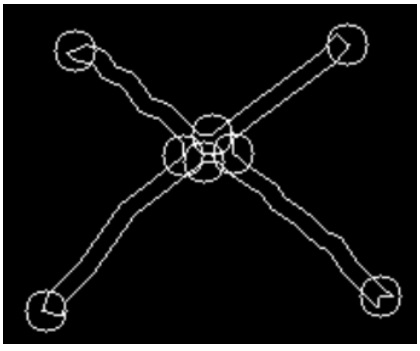


Identifying a Move

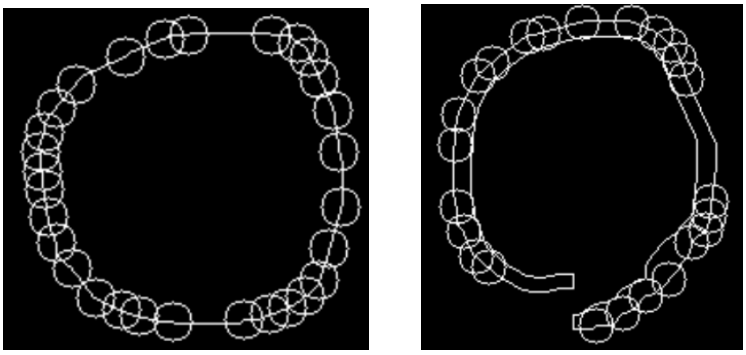
Once I have isolated a move, I find its contours by compressing similarly-colored segments into only their edges:

```
Imgproc.findContours(diffMat, contours, mHierarchy, Imgproc.RETR_EXTERNAL,
        Imgproc.CHAIN_APPROX_SIMPLE);
```

Next, I use openCV's erode function to dilate the black background and thus make the white lines thinner. This was necessary for having satisfactory results with shapes that were not closed. In the case of closed shapes, such as a closed circle, my contour function identified the outside contours. However, for open shapes, such as a circle with endpoints that don't meet, the entire contours were identified. Since a lot of my shape identification involves identifying empty space, the added width of the contours ruined many of my results. Next, I find the contours' points of interest using OpenCV's convexity defects function, which finds points associated with deviations in the convex hull. The points are then filtered to make sure no two points are very close together. For the X example above, the contours and convexity points are:



Below is an example showing the differing contours of a closed and open circle. Prior to the eroding, the difference was much more drastic.



Once the convexity defects are obtained, I make sure that all the convex points are located in the same cell. If not, the game is restarted:

```
int cellNum = -1;
for (int i = 0; i < cells.length; i++) {
     if (convexPoints.length > 0 && convexPoints[0].inside(cells[i])) {
         cellNum = i;
         break;
     }
 }
for (Point p : convexPoints) {
     for (int i = 0; i < cells.length; i++) {
         if (p.inside(cells[i]) && i != cellNum && cellNum != -1) {
             restart();
         }
     }
 }
```

Otherwise, these contours and convex points are used to recognize the specific shapes in the order below.

Recognizing a + and an X

The first thing I do before I try to identify any shape is find the bounding box of the contours. By knowing the width and height of the drawn shape, I can adjust my expectations of where certain elements should be based on the size of the drawn move rather than using a fixed approach independent of size. The bounding box is shown in the examples below in blue.

The next step is to identify the center of gravity of the shape:

```
    public static Point centerOfGravity(Mat m) {
        Moments p = Imgproc.moments(m, false);
        int x = (int) (p.get_m10() / p.get_m00());
        int y = (int) (p.get_m01() / p.get_m00());
        return new Point(x, y);
    }
```

For a plus and an X, I know that the center of gravity must be very close to the intersection of the two lines forming it. This intersection forms 4 corners in the contour of the shape, which are consistently picked up by my function that identifies the convexity defects. Thus, identifying +'s and X's hinges on finding these 4 convex points near the center of gravity.

```
Point cog = centerOfGravity(contours);
Rect cogRect = new Rect(new Point(cog.x - w, cog.y - h), new Point(cog.x + w, cog.y + h));
int count = 0;
for (Point p : convexPoints) {
    if (p.inside(cogRect)) {
        count++;
    }
}

if (count != 4) return false;
```

One element that uniquely identifies a + in conjunction with the 4 convex points is the lack of

convex points found in the corners of the bounding rectangle.

```
Rect ulRect = new Rect(boundingRect.tl(), new Point(boundingRect.tl().x + w / 2,
                    boundingRect.tl().y + h / 2));
Rect urRect = new Rect(new Point(boundingRect.br().x - w / 2, boundingRect.tl().y),
                    new Point(boundingRect.br().x, boundingRect.tl().y + h / 2));
Rect blRect = new Rect(new Point(boundingRect.tl().x, boundingRect.br().y - h / 2),
                    new Point(boundingRect.tl().x + w / 2, boundingRect.br().y));
Rect brRect = new Rect(new Point(boundingRect.br().x - w / 2, boundingRect.br().y - h / 2),
                    boundingRect.br());

for (Point p : convexPoints) {
     if (p.inside(ulRect) || p.inside(urRect) || p.inside(blRect) || p.inside(brRect)) {
         return false;
     }
}
```

Contrastingly, an X is identified by the lack of convex points to the left, right, above and below

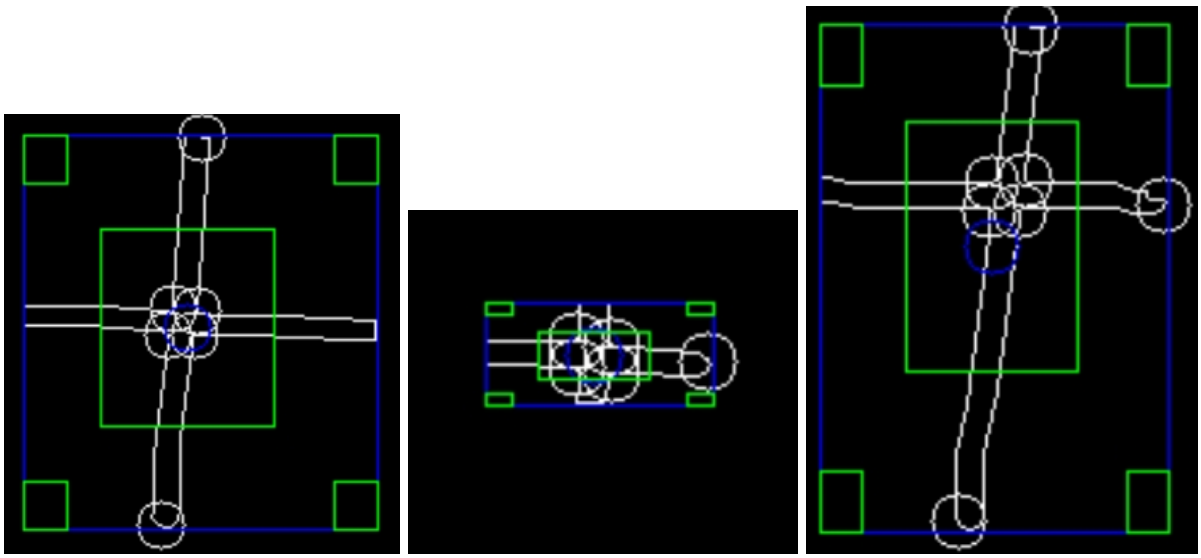
the center of gravity, so I perform the same test using the following rectangles:

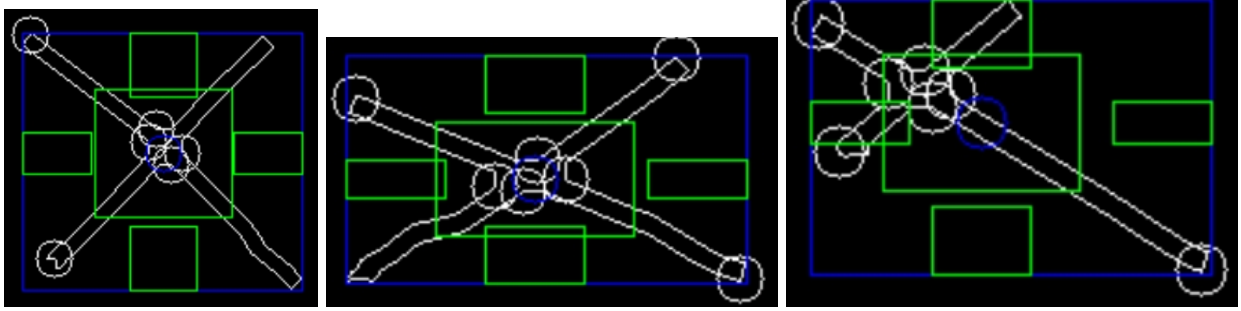
```
Rect lMid = new Rect(new Point(boundingRect.tl().x, cog.y - h / 12),
                    new Point(boundingRect.tl().x + w / 4, cog.y + h / 12));
Rect rMid = new Rect(new Point(boundingRect.br().x - w / 4, cog.y - h / 12),
                    new Point(boundingRect.br().x, cog.y + h / 12));
Rect tMid = new Rect(new Point(cog.x - w / 8, boundingRect.tl().y),
                    new Point(cog.x + w / 8, boundingRect.tl().y + h / 4));
Rect bMid = new Rect(new Point(cog.x - w / 8, boundingRect.br().y - h / 4),
                    new Point(cog.x + w / 8, boundingRect.br().y));
```

Despite these clear-cut distinctions, one important thing I realized after my first iteration of user testing was that **false-positives are much better than false-negatives**, especially in a game where it is likely that the user is playing in good faith and is actually trying to draw the correct move. As such, for every shape I try to identify, I loosen the requirements once my program knows what shape to be look for. So for the very first move of the game, recognizing a + or X requires all the previous conditions to be fulfilled. However, if the user already drew a recognized plus sign or X, all subsequent moves will only need to meet the criteria of the 4 convex points near the center of gravity. Thus, any pair of intersecting lines will be accepted.

A similar methodology is applied by humans in a manual game of Tic-Tac-Toe. Knowing that a player is playing as an X, even a sloppy X will be identified since the other player is expecting that shape.

Below are the contours and convex points of several plus signs and X's along with the associated areas of interest (green rectangles) and center of gravity (blue).





Recognizing a Circle and Square

The two main features of both squares and circles are the ratio of the bounding boxes and the emptiness of the middle of the shapes. Since the sides of a square are equal and the radius of a circle is consistent throughout, I gave a bit of leeway on the ratios of the width and height of the shape. I found .8 to be a reasonable ratio cutoff.

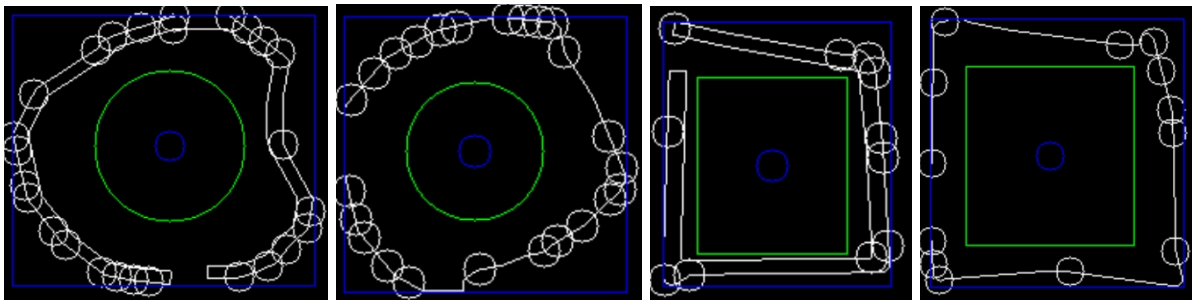
```
int w = boundingRect.width;
int h = boundingRect.height;
double ratio = ((double) w) / h;
if (ratio > 1) ratio = ((double) h) / w;
boolean circular = ratio > .8;
if (!circular) return false;
```

Next, my program checks that the inside of the shape is empty. For a square, I create a rectangle around the center of gravity and make sure no contour points are inside the rectangle.

```
Rect cogRect = new Rect(new Point(cog.x - w / div, cog.y - h / div),
               new Point(cog.x + w / div, cog.y + h / div));
Point[] allPoints = contours.toArray();
for (Point p : allPoints) {
    if (p.inside(cogRect)) {
        return false;
    }
}
```

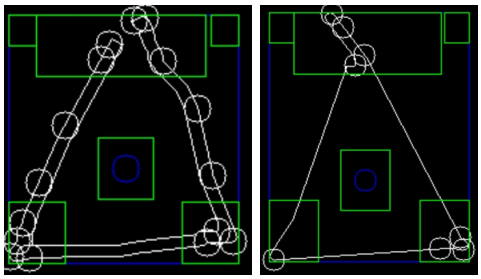
For a circle, I do the same thing except I check that the distance between the center of gravity and every contour point is greater than a certain distance. I decrease the size of the inner rectangle and circle on the moves following the first identification.

The main attribute by which I differentiate between a square and a circle is the number of convex points. Since a square should most be made up of straight lines, it should have far fewer convex points than a circle, which is drawn by constantly changing direction. A circle requires at least 20 convex points. This becomes a bit of an issue for small circles, as they often will result in a false positive because they do not have enough convex points to pass the circle requirements.



Recognizing a Triangle

Once we test for a triangle, we know that the shape failed the tests for all the previous shapes. With this knowledge, we test for a triangle using similar methods. The program checks that there is a contour point on the bottom left and bottom right corners of the bounding box, as well as in the upper middle section. Then, the program checks that the area near the center of gravity is empty. If the identification is for the first move of the game, it also makes sure that there are no contour points in the upper left and upper right corners of the bounding rectangle.



Recognizing a Heart

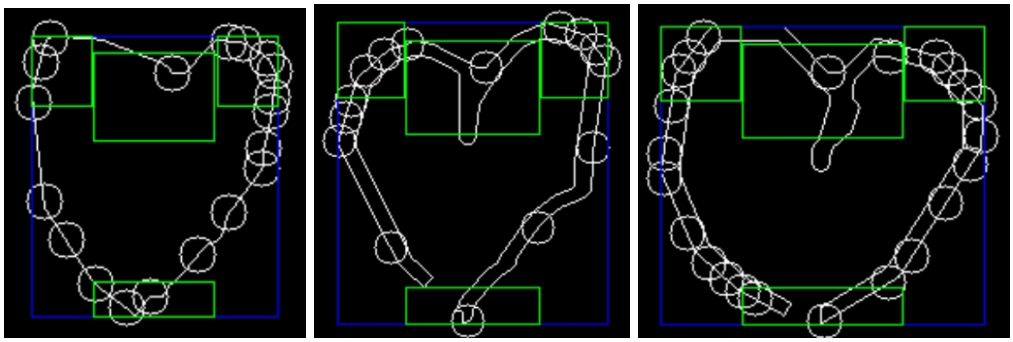
Once the other shape tests fail, the next test is the heart. The recognize a heart, there needs to be a convex point at the bottom middle of the bounding box, the upper middle dip, and the upper left and right corners where the arches are. I removed the check for emptiness under the center of gravity because an open heart created a contour that had the dip reach down rather far in the bounding box, ruining many results.

```
for (Point p : convexPoints) {

    if (p.inside(bottomMid)) bMid = true;
    else if (p.inside(upperMid)) uMid = true;
    else if (p.inside(ulRect)) ul = true;
    else if (p.inside(urRect)) ur = true;
}

return ul && ur && bMid && uMid;
```

For all the checks after the initial heart identification, the search boxes increase slightly.



Recognizing an Arrow

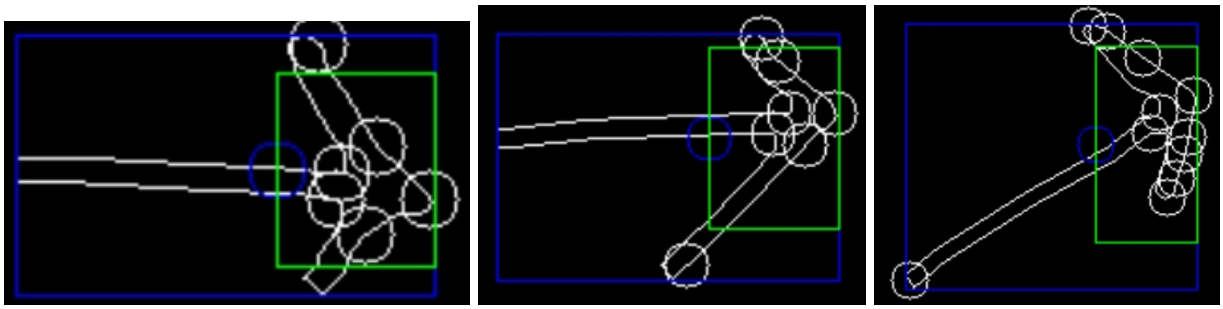
The final test is to determine if the move is an arrow. I required the arrow to be rightward-facing, so I made sure the center of gravity was to the right of the middle of the bounding box:

```
Point cog = centerOfGravity(contours);
if (cog.x < boundingRect.x + boundingRect.width / 2) return false;
```

The only other check I made was that the area to the right of the center of gravity had at least three convex points - the point of the arrow and the 2 corners created by the intersection of the three lines that make up the arrow:

```
Rect midRight = new Rect(new Point(cog.x, cog.y - h),
                         new Point(boundingRect.br().x, cog.y + h));
int count = 0;
for (Point p : convexPoints) {
    if (p.inside(midRight)) {
        count++;
    }
}
return count >= 3;
```

At first, I also tried to make sure that there were no contour points in the upper left and lower left corners of the bounding box, but after user testing, it was clear that most people's arrow barbs were not the same length and arrows were not always horizontal. As such, the shaft of the arrow was often in one of those two corners.



Tic Tac Toe Artificial Intelligence

I used the minimax algorithm found at https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaGame_TicTacToe_AI.html as my implementation for my AI mode. The algorithm looks ahead 2 turns during the first iteration (depth of 4) and ahead one turn (depth of 2) the rest of the game and applies a heuristic in which it evaluates a score for each of the 8 lines on the grid, adding 100 to the state score for each of its winning lines, 10 for each line in which it has two of its pieces, and 1 for each single

element line. It then does the same with the opponent's pieces, except negating the score. It then chooses a move in order to minimize the maximum possible loss.

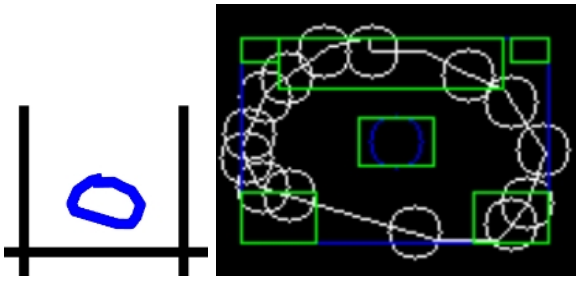
Results and Conclusions

My AI implementation worked quickly and flawlessly. Since Tic-Tac-Toe is a solvable game, the best a human opponent can do is force a tie. If the human ever makes a mistake, the AI will always win the game.

Overall, my move recognition worked better than I expected given the variety of different drawings my users produced for each shape. I had initially underestimated how difficult it was to draw a closed circle and heart on a phone, so my first algorithm failed often on open shapes. My biggest breakthrough come on the realization that false positives were much better than false negatives. My users were frustrated that every shape they drew had to be perfect in order to be recognized, since I was applying the same criteria throughout the game. I compared this to the human mindset of "close-enough" when playing Tic-Tac-Toe, and figured out which restrictions I could loosen to recognize the shapes more often. Furthermore, my users were not trying to trick my program, they were simply trying to draw the same shape again. My second implementation following this realization created much better results.

In my proposal, I considered acceptable results to be an >80% recognition rate for X, O, +, ☐ and △, with slightly worse results for → and ♡ due to the variety of ways they could be drawn. When the first move has already been identified, the subsequent recognition rate exceed 80% for all 7 moves. On the very first turn, results were slightly more mixed. X and + were by far the easiest to recognize, as the 4 convex points formed by the intersection worked almost

flawlessly. Circle and triangle were somewhat more difficult, resulting in the greatest amount of false positives. For example, the following small circle did not have enough convex points to be classified a circle, and instead passed the test for a triangle:



Square recognition worked very well when drawn with relatively straight lines. On rare occasion, the squares had enough convex points to count as a circle, as they both had similar empty space in the center. Hearts began as the most difficult to identify since I started by checking for emptiness on the middle, which was ruined by the mostly-open hearts drawn. Once I got rid of the criteria and added contour erosions, their success rates increased over 80%. Finally, arrow criteria was general enough that it too resulted in a better than 80% success rate.

Considering this project used no machine learning to find the most likely shape, I would deem this project a success. I have shown that distinct shapes can be easily distinguished from each other by understanding user tendencies and the features that are critical to a shape. I think it would have been better to replace square with a different shape given its similarity with a circle in both ratio and empty space. Still, this project could be extended to identify many different kinds of geometric shapes and symbols.