How to execute

Part 1

1. Navigate to Part 1 folder

2. Open Part1EXE

3. Wait until white canvas with 20x20 gray square grid pops up

4. Click anywhere on the grid with left mouse button and hold it down

5. Hold down the button and drag towards any direction to resize the circle

6. Release the button

7. Blue circle is the circle that you created

8. Highlighted(blue) squares around the blue circle most closely approximate the location of the edge of the blue circle

9. Two red circles enclose all the highlighted squares

Part 2

1. Navigate to Part 2 folder

2. Run one of the following:

- **Part2 with optimized initial guess**: used helper function to get the optimized initial values of a circle (center x, center y, radius). Three highlighted squares are enough to get the best fit.

- **Part2 with rough initial guess**: set initial center of a circle as the center of the grid and radius as 300. More than 10 highlighted squares are required to get the best fit. Recommend generating first and highlight more squares as you visually see the progression.

3. Wait until white canvas with 20x20 gray square grid pops up

4. Click any square on the grid to highlight

5. Click highlighted square to unhighlight

6. After highlighted three or more squares, click generate button

7. You can always highlight more squares after you clicked generate (also unhighlight)

8. Blue circle is the best fits of the highlighted squares

9. If you want to clear all the highlighted squares, click reset button.

Part 1 – Algorithm

To enable user-interface, I used the package called fssimplewindow.cpp. This framework is open-source code that provides a set of functions that opens an OpenGL enabled window, and take keystrokes, mouse input, and sub-second timer.

I first created Grid class that contains checkCollision(..), highlightSquare(..) and draw(..) method.

checkCollision(..) method loops through all the points on the edge of the circle to check whether the certain square collide with the edge of the circle. However, to check not only the collision but also whether the edge is close, I expanded the territory of a square.

highlightSquare(..) uses return value of checkCollision(..) to find which squares need to be highlighted.

draw(..) method drew gird by drawing 20x20 squares. To highlight the square, I updated color array that stores color information of each square, so draw(..) method automatically update the color at every iteration.

And then I created parent Circle class with common variables and virtual methods that child classes must have.

Then I created BlueCircle that inherits Circle class and draws blue circle with the radius and center coordinate that obtained from calculateRadius(..) and updateMouseLoc(..) methods.

updateMouseLoc(..) method updates the coordinate of the circle’s center with the coordinate of the mouse location. (centerX = cursorX, century = cursorY)

calculateRadius(..) method uses the updated center coordinate to calculate the radius by using the equation: r = sqrt(cx^2+cy^2).

Next, I created RedCircle class that also inherits Circle class and contains checkDistance(..) and calculateRadius(..) method.

calulateRadius(..) method receives queue that contains left-top corner coordinate of highlighted squares. It calculates the distance between each highlighted square and the center using: dist = sqrt(x\*x+y\*y), and then, pass it to checkDistance(..). To get furthest or closest distance, it calculates distance from center to all four corners of the square.

checkDistance(..) method checks whether the input distance is either outer radius or inner radius.

And then I wrote main(..) function. It first opens canvas and enters infinite loop that terminate only when the user hit ESC or close window. At every iteration, it reads mouse event using FSGetMouseEvent function and when the user click left button and drag. And then, it uses above methods to create blue and red circles and highlight the squares.

Part 2 – Algorithm

In the revised part 2, I used Levenberg-Marquardt circle fit method (improved Newton method) to implement iterative least squares fit method. Not like the Taubin method, LM method required the initial value of a center, thus had a risk of divergence. Furthermore, it required more points than the Taubin method to get a desired best fit when the initial values were off. However, it was better than Taubin in finding the best fit for the big circle with a small arc. Furthermore, it got more accurate as data points accumulate and beyond certain level it got more accurate than the Taubin method.

Reference

N. Chernov, *Fast and numerically stable circle fit*, 2015