Neocis Software Challenge Documentation

Position: Internship - Software Engineering

Tyrone Gallardo

Overview

The software assessment was divided into two parts. For the sake of simplicity, each part was made into its own individual project. The challenge was coded using Java16 in its entirety. Here is a list of artifacts that can be found in the email submission:

- The project documentation (this document)
- CircleGrid.zip, the compressed file containing all the artifacts related to Part 1
  - The source code for Part 1
  - ‘runPart1.bat’, a Windows Batch File that will execute the code
  - ‘runPart1.jar’, a runnable .jar file that will execute the code
- CircleGrid.zip, the compressed file containing all the artifacts related to Part 2
  - The source code for Part 2
  - ‘runPart2.bat’, a Windows Batch File that will execute the code
  - ‘runPart2.jar’, a runnable .jar file that will execute the code

Part 1

Overview:

Upon execution, a window will show displaying a 20x20 grid of `MyButton` squared buttons. The user can click a button and drag the cursor to display a dynamically-sized circle. Upon releasing the mouse click, an algorithm will map the circle to the squared buttons beneath it, creating a replica of it using said buttons in blue.

Additionally, two additional circles, sharing the same origin as the blue circle, will display in red when the mouse click is released. The outer red circle represents the smallest possible circle that **completely encapsulate*s*** all the squares marked in blue. The inner red circle represents the largest possible circle that is **completely encapsulat*ed*** by the squares marked in blue. The origin of these three circles will be marked in yellow upon release as well.

Parts of the source code will be commented and labeled as **IGNORE**; these areas of the code are not relevant to the implementation, but served as valuable experiences.

Algorithm

A public static variable named `MoE` (Margin of Error) determines which squares will be marked blue. Each square has an object attribute named `center`, which helps determine the square’s relative position on the window. By knowing each square’s position, the distance between each square and the circle’s origin (another square) can be found using a simple method; the `distanceFrom()` method found in the `Coordinate` class serves this function. Only squares for which $d - r \leq MoE$, where d is the distance between the square and the origin, and r is the circle’s radius, were set to blue. The `MoE` variable can be changed in-code, but a balance between effectiveness and precision is preferred: `8px`.

To make the red circle implementation work, additional attributes had to be added to each square. Rather than just knowing the coordinates for their center, knowing the coordinates for each corner of the square would be useful. For this purpose, four additional attributes `tl`, `bl`, `tr`, `br` (for top-left, bottom-left, top-right, bottom-right, respectively) were added. Next, all four corners of each blue square was measured against the origin, and the shortest and longest distances were recorded; these distances would become the radii of the inner and outer red circle, respectively

Synopsis

When the project is executed, the following steps carry out in order:

1. The `main()` method is executed, which runs the `gui()` method in Line 90.
2. The `gui()` method will create a new frame instance, and call its class constructor in line 97.
    1. The constructor will set up every square button that will be displayed later, setting things like bounds, and backgrounds.
    2. The constructor will add a `MouseListener` and a `MouseMotionListener` using instances of the `Part1Listener` inner class.
    3. The constructor will then add each square button to the frame in line 55
3. The `gui()` method initiates the frame to be displayed, setting background, window size, and more.
    1. The `CirclePanel1()` panel is added to the frame in line 101, setting up a `Graphics` object that will act as the brush in the frame.
    2. The `Graphics` object will immediately draw the three circles (1 blue and 2 red), but since the radii variables they are using are momentarily set to 0, the circles will not be visible. The circles will display when these variables are given a value in a future step.
    3. The frame will be set visible in line 103.
4. As soon as the window is visible and interactable, the listeners in each of the 400 square buttons will be awaiting events.
5. The user will proceed to click and drag from one of the squares, where a `MousePressed()` and a `MouseDragged`() method will be called through the `MouseListener` and the `MouseMotionListener`.
    1. The `MousePressed()` method will store the position of the origin button that was first pressed.
    2. The `MouseDragged()` method will dynamically keep track of the mouse cursor's relative position to the circle's origin, and store this distance in the `currentRadius1` variable. The `Graphics` object, activated by the `frame.update()` method, will use this newly set distance $d \neq 0$ and change the radius of the blue circle.
    3. The user can see the blue circle being expanded dynamically as the mouse click is being dragged.
6. When the user releases the mouse click, a `MouseEvent` will trigger the `mouseReleased()` method, triggering three main additional methods:
    1. The `setBtnBlue()` method will be called in line 190, turning squares blue as indicated in the algorithm explanation above.
    2. The `findRedCircles()` method will be called in line 191, assigning values to the `farthestRadius1` and `closestRadius1` variables as indicated in the algorithm explanation above. These radii are then used by the `Graphics` object to display the two red circles.
    3. The `frame.update()` method will update the graphics to the newly set radii.

Part 2

Overview

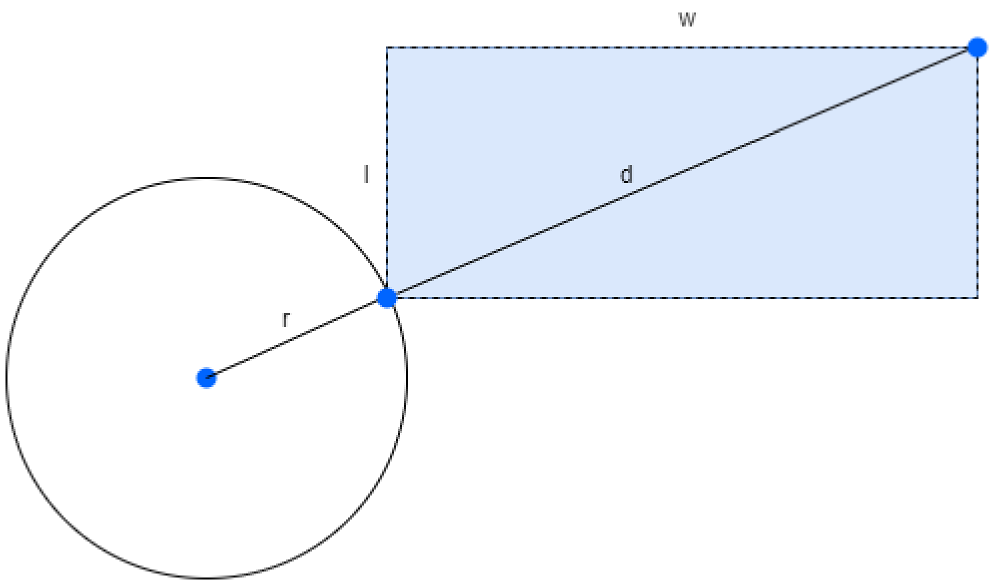
The second part of the project requires that the user be able to select an arbitrary amount of points in the grid; the same 20x20 grid will be used for this part as well. After selecting any amount of points, the user will be able to click a **Generate** button, which will generate a circle that best fits the selected points. To determine the best-fit circle, a geometric least squares-based algorithm was required.

After generating a circle, the user can continue to generate additional circles by either adding or removing points and reclicking the **Generate** button. Marked points will display in blue, while un-marked buttons will display in gray.

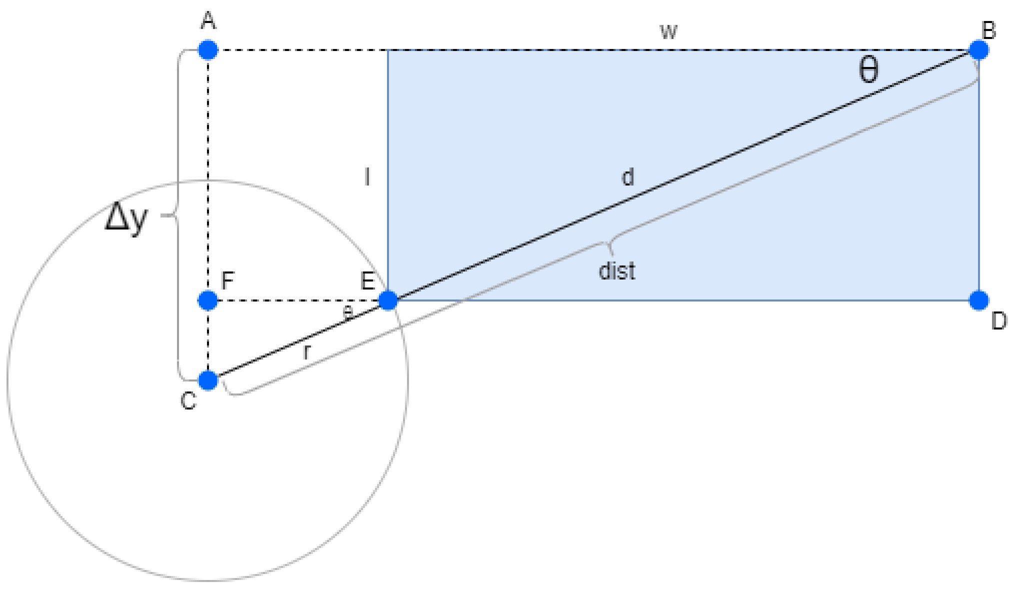
Algorithm

Geometry

As requested, Part 2 uses a very effective least square-based algorithm. The idea was to be able to find the area of a rectangle with diagonal d, where d is equal to the distance between an arbitrary selected point and a point in the desired circle. A diagram helps visualize this better:



If it was possible to obtain either w or l, it would be possible to find the area of that square using the formula $A = l\sqrt{d^2 - l^2}$. To accomplish this, some basic geometry was necessary. The following diagram helps visualize the procedure:



Our first step is to prove that $\Delta ABC \sim \Delta FEC$. Since $\overline{AB} \parallel \overline{DF}$, and both θ angles are formed by $\overline{CB}$, we can conclude that the angles are congruent. Furthermore, $\angle CAB = \angle CFE = 90^o$, proving that $\Delta ABC \sim \Delta FEC$ through the Angle-Angle Postulate. What follows is to set up the equation:

$$\frac{l}{\Delta y} = \frac{d}{d+r} \implies l = \frac{d\Delta y}{d+r}$$

Once we find l, we can easily use l and d to find the area of the highlighted rectangle.

Finding an Approximated Origin

To find the best origin to start the circle from, I found the average of the x- and y-coordinates of every selected square. The idea was to start the circle at the center-most place in the grid with relation to the selected points.

Finding the Best Circle

To find the best-fit circle, the algorithm found two values first: the closest and the farthest selected square from the approximated origin; the distance from the closest circle to the origin was stored in the variable `closestRadius2`, and the distance from the farthest circle was stored in the variable `farthestRadius2`. From there, the algorithm would try every circle, starting at a radius of `closestRadius2` and incrementing by `RoC` until a radius of `farthestRadius2` was reached; `RoC` can be incremented to increase the number of attempted radii in the algorithm at a larger cost, but a value of 1 is appropriate.

The algorithm goes through the defined range of radii, and calculates the least square area of every selected point in relation to the circle as described in the ***Geometry*** subsection. The sum of these areas are compared at the end of every iteration, and the smallest area is kept, along with the designated radius for that area. The blue circle will start from the approximated origin and have the designated radius.

Synopsis

When the project is executed, the following steps carry out in order:

1. The `main()` method is executed, which runs the `gui()` method in Line 63.
2. The `gui()` method will create a new frame instance, and call its class constructor in line 70.
    1. The constructor will set up every square button that will be displayed later, setting things like bounds, and backgrounds.
    2. The constructor will add a `MouseListener` using instances of the `Part2Listener` inner class.
    3. The constructor will then add each square button to the frame in line 42.
3. The `gui()` method initiates the frame to be displayed, setting background, window size, and more.
    1. The `CirclePanel2()` panel is added to the frame in line 74, setting up a `Graphics` object that will act as the brush in the frame.
    2. The `Graphics` object will immediately draw the circle, but since the radius variable it is using is momentarily set to 0, the circle will not be visible. The circle will display when this is given a value in a future step.
    3. The frame will be set visible in line 76.
4. As soon as the window is visible and interactable, the listeners in each of the 400 square buttons will be awaiting events.
5. The user will now begin clicking the desired squares to be selected. A `MouseEvent` will activate the `mouseClicked()` method, which will turn the squares blue on click. The squares can be de-selected by being clicked it again.
6. Once the user has selected all the squares, he/she will click the **Generate** button, which will once again trigger a `MouseEvent` and call the `mouseClicked()` method, which has a specific function when it detects the **Generate** button:
    1. A variable `btnCoo` will be set to hold the approximated origin, determined by the `findOrigin()` method, called in line 137. This method's algorithm was discussed in the ***Algorithm*** section.
    2. After an origin is set, the `findRadius()` method will be called, which finds the best possible radius that fits the given points. This method's algorithm was discussed in the ***Algorithm*** section.
    3. Finally, the `frame.update()` method is called to display the circle with the newly updated radius.