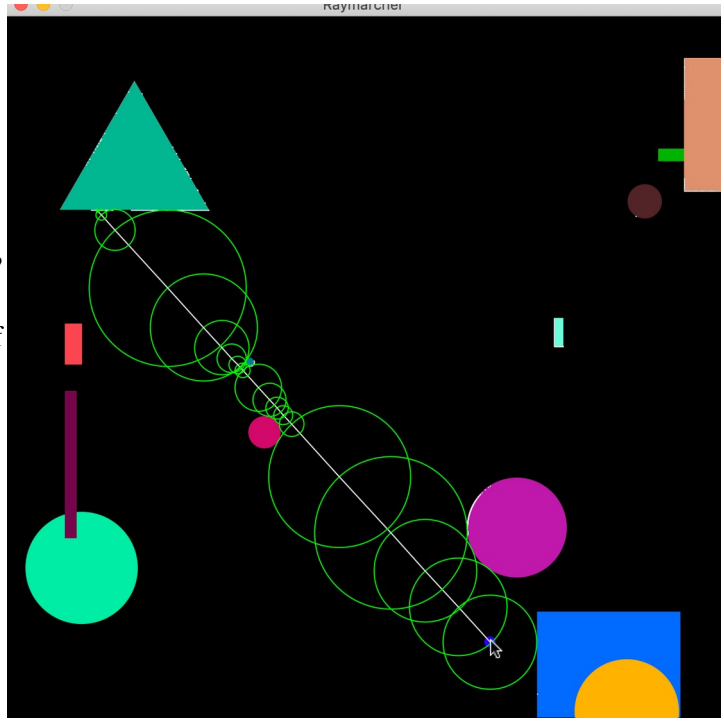
CSCI165 Computer Science II
Final Project
Raymarching in a Two Dimensional Plane

Objective:

- Object Orientation: Encapsulation, Composition, Inheritance, Abstraction and Polymorphism
- Determine the distance between a point and multiple objects in a 2D plane
- Change angle-of-view and position via mouse input
- Modify graphics/shapes in a 2D plane.

What You Will Create:



Background: Today's world of computing is filled to the brim with colorful and life-like graphics; continuing to blur the line from fantasy and reality. Has it ever occurred to you to think about how games, animators, and others create these amazing works of art?

In this lab, we're going to explore the topic of sphere-tracing: a type of ray marching algorithm.

First, let's back up and understand what ray tracing is, since ray marching is a derivation. Ray tracing involves a camera and an environment (also called a world). The camera projects simulated lines called rays into the world which then interact with objects in the world. Think of it like a light source; light travels from its source to objects which is then reflected and altered based on the colliding object. Try to imagine how a computer would do this. A computer has to mathematically determine when a collision with an object occurs so as to not erroneously pass through that object (we're ignoring translucency).

Note that the main component is the ***camera*** (i.e., where our ray begins, aligned with the cursor), and the purple line is a wall, or an object. Each green circle through which the wall passes is considered a "march". The march emanates from the wall until it collides with an object. The marches in this graphic will always travel towards the object that is the closest to the wall.

TASK ONE:

1. Extract the `RaymarcherLab.zip` file. Inside, you will find three Java classes: `RaymarcherRunner`, `RaymarcherPanel`, and `SwingApplication`. The latter `SwingApplication` initializes boilerplate code for the Swing UI components, so unless you are interested, it is not necessary to investigate this code further.

2. After setting up the project, get accustomed to the two other classes. The main class, `RaymarcherRunner`, as the name suggests, runs the application, and initializes all GUI components. On the other hand, `RaymarcherPanel` is where you will be doing most of the laborious work. Rendering and drawing should occur on this JPanel object. To test your environment, a couple lines of code are included that draws a blue rectangle on the screen inside the `paintComponent` method using the `Graphics2D` class. Play around with this to see if you can try different colors or even different shapes. Remove these lines (except the first two) once you are done experimenting.

3. Since we're going to be creating an environment of shapes for rays to collide with, we obviously need objects for the ray to collide with, right? So, let's do that by populating our world with random rectangles, triangles and circles. You will be using classes from the previous Shape lab

    a. Create an abstract class `Shape`. A `Shape` should have, at minimum, an *x*/*y* coordinate pair represented by the `Point` class from an earlier lab, a `Color` and the ability to be filled or not. Our *x*/*y* coordinate pair will need to be changed to float or double (explained below)
    b. Create subclasses called `Rectangle, Triangle` and `Circle` that define the fields to manage the size of each object. Add as many different subclasses and shapes as you wish.

⚠ **Warning!** Make absolutely sure that you use `double` or `float` variables when initializing positions. When drawing with Swing, you can cast doubles to integers and use other methods in `Graphics2D`. Later on, when we perform arithmetic on the positions and dimensions, floating-point operations are crucial to ensure we don't encounter integer truncation issues.

⚠ **Warning!** If you're trying to set the positioning based on the screen/window size, you'll need to use `this.getPreferredSize().width` and `this.getPreferredSize().height`. This is because the `JPanel` hasn't been packed into the parent `JFrame` component when the `Shapes` are instantiated.

⚠ **Warning!** Keep note of whether your objects are instantiated at the center or the top-left. The choice is up to you, but if you instantiate them at the center now, it's slightly less effort later. Otherwise, you have to do a bit more later on since we'll be working with the center of objects.

4. Now, you may be wondering: "Where do we instantiate these objects?" Well, we can populate them in the `RaymarcherPanel` class. Create a list of `Shapes` with random dimensions, colors and positions. The size of the list doesn't necessarily matter but try to keep it lower than twenty (20) objects. You may want to modify the code so that you can easily specify how many shapes to create when it runs. Also, make sure that objects do not generate outside the world! Do not worry if they overlap . . . in fact you want this in order to test your logic.

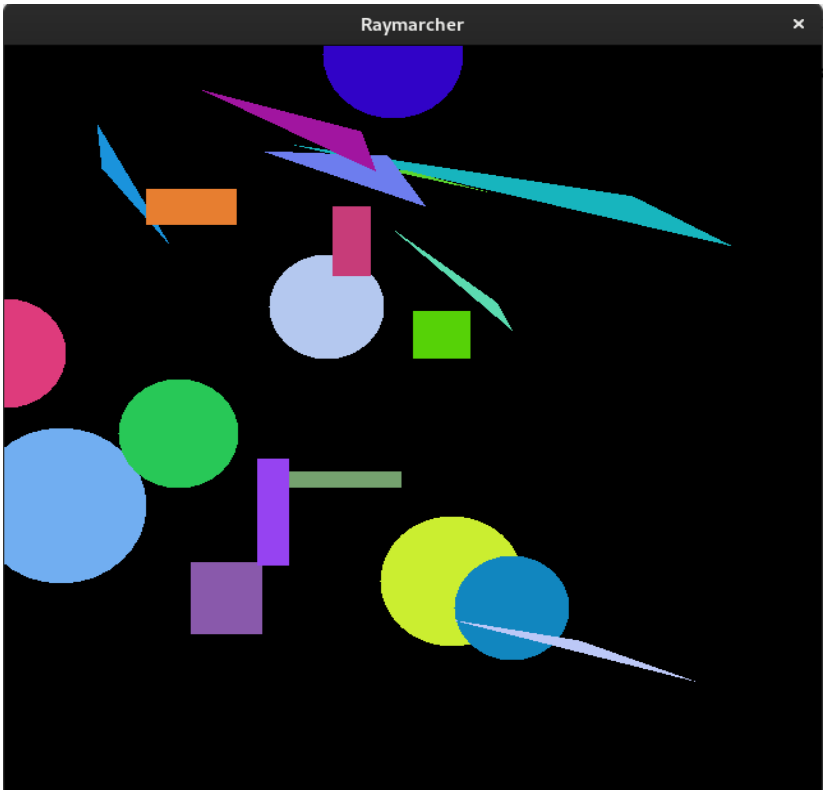
5. At this point, you should have a fully populated list of `Shapes`. It is now time to draw them! Note that `JComponents` have the `paintComponent(Graphics g)` method for drawing. Since we're going to be drawing objects besides `Shapes`, we should create an interface that says a class is "drawable".

    a. Create an interface called `Drawable` with the method signature

    ```
    void drawObject(Graphics2D g2d).
    ```

    From here, implement the interface in `Shape` and override its method in your subclasses. Now, add the functionality to draw the shapes using their position and color. Feel free to use the **isFilled** behavior too.

    b. Finally, in your panel class, iterate over your list of objects and call `drawObject` on each one. When drawing the objects, draw them at their center! Drawing them at the top-left can cause problems down the road. So, make sure you apply the correct math offsets to draw the shape at its center (note that I said <u>draw</u> at the center; not <u>position</u> at the center. If you position at the center then you've already done this part!).



TASK TWO:

6. We're now ready to start our ray marcher! The first thing we need is some type of "camera" or perspective to serve as the beginning point for the march. It also would be a little boring if we could only march rays in one direction, right? So, we'll need to add angle change functionality to our camera, but we'll get to that as we go.

   Create a class called `Camera` and another called `March`. `Camera` will be where the ray begins marching (it will eventually be associated with the cursor), and a `March` object will represent a single step, or iteration, in the ray march. Both of these will have *x*, *y* coordinates and radii. This is almost identical to the `Circle` class, and you can subclass it but these new objects do serve different purposes. We'll first write the `Camera` class since it is more interesting.

7. `Camera`, as mentioned earlier, is the starting point of our ray march. So, like `Shape`, we're going to implement `Drawable`. The camera is just a small circle, so giving it a fixed radius of, say, ten (10) pixels is sufficient. Do the same thing you did for `Circle`: draw the camera at the provided *x* and *y* coordinates.

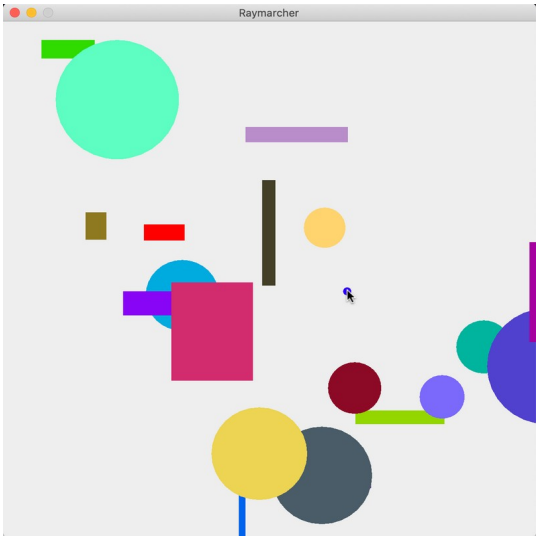
8. Now, we're ready to move our camera! There are two ways we can do this: with keyboard input or mouse input. Mouse input is more appropriate here, so let's do that. As we move the mouse around the world, we want our camera to follow so we want our camera to "listen" to the movement of the mouse. The Java API provides a `MouseMotionListener` interface for us to implement and define this behavior.

   Once `Camera` implements this, you will be required to override two methods, but we only need to write code inside one: `mouseMoved(MouseEvent)`. Whenever we move the mouse, we want to update the x and y coordinates of `Camera`. Any time the mouse is moved, the `mouseMoved` method is called, and the `MouseEvent` parameter contains two methods: `getX()` and `getY()`. So, assign the coordinate instance variables of Camera to these values in this method.

9. The only thing that's left is to register the motion listener with the panel so the panel will know which class to notify that the mouse has moved. So, create an instance of `Camera` inside the `RaymarcherPanel` constructor. Call `addMouseMotionListener` and pass it the `Camera` object. Also, don't forget to call `drawObject` from `Camera` inside `RaymarcherPanel`'s `paintComponent` method or you won't see anything! Run the program and you should see your camera move as you move the mouse.

Warning! If you place the camera's draw method above the loop where you draw the objects, you won't see it if your mouse is over an object. Can you deduce why?

NOTE: If you assign the x and y coordinates to the exact position of the mouse event's x and y coordinates, it will be slightly offset. Fix this



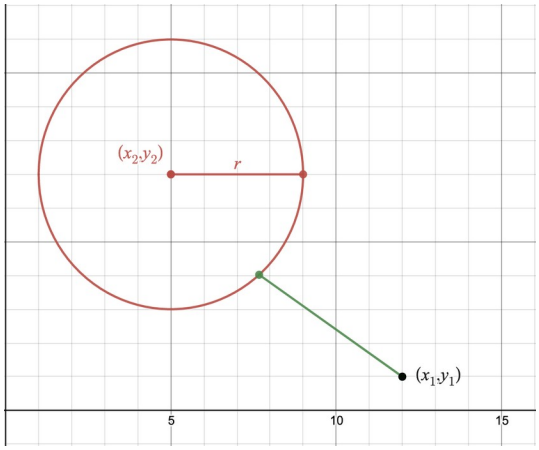
TASK THREE

10. Now, let's begin the raymarching! We're going to implement sphere tracing, where we compute the minimum distance between the mouse and ***all objects in the scene***. So, we first need to understand this calculation. We're essentially computing the hypotenuse of the triangle formed from the center of the camera to an object's center. So, let's look at this for both cases.

    For circles, we need to account for only one thing: the radius. Take the distance (also called the magnitude if you're familiar with vectors and linear algebra) from the camera to the center of the circle and subtract its radius. The formula is as follows and you should already have this implemented in the Point class from previous labs:

    $$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} - r$$

    Where x_1, y_1 represent the coordinates of the camera's center, and x_2, y_2 represent the center of the circle. r is the radius of the circle. In the figure below, we want to compute the magnitude (length) of the green line). This is d in the above equation.



Rectangles are a bit more complicated since we're involving both width and height instead of just a radius. The simplest way to do it is to compute the distance between the camera and the line segments that make up the rectangle. The `Line2D` class provides a great method for computing this distance: `ptSegDist`. But working the math out yourself is great practice

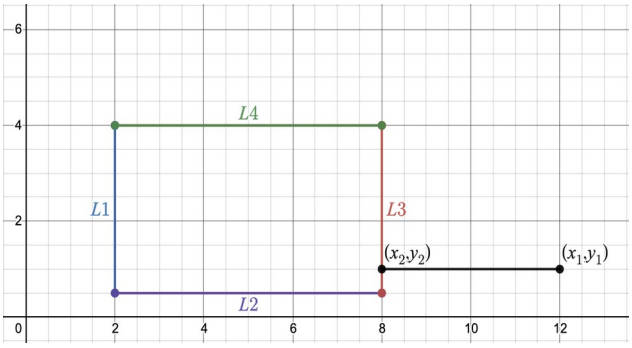
If the line passes through two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ then the distance of (x_0, y_0) from the line is:

$$\text{distance}(P_1, P_2, (x_0, y_0)) = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}.$$

There are four line segments that make up each rectangle, so just take the minimum of all four segments.

NOTE: This math is for a continuous line segment that would extend infinitely through two points. To factor in our ***finite lines*** you will need include the termination points of the segments.

Note that creating the line segments is slightly harder if you chose to center the rectangle instead of using its top-left coordinate (but not by much at all!). In the figure below, note that each line segment is denoted by *L*1, *L*2, *L*3, and *L*4. Recreate this in your program.



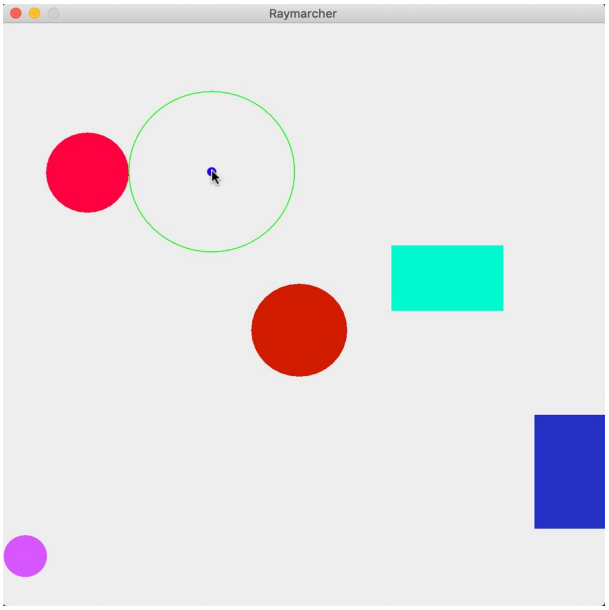
A good idea would be to create an abstract `Shape` method

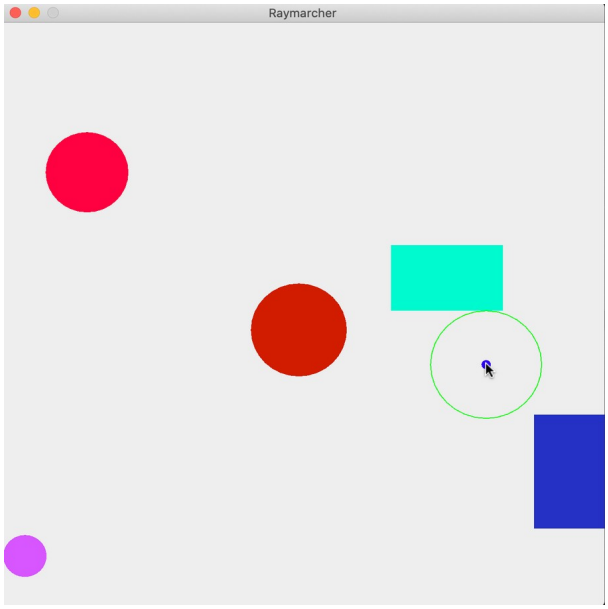
```
computeDistance(Point cameraPosition)
```

which is overridden in subclasses.

11. Now, iterate through your list of objects and compute the minimum distance between the camera's position and each object. Use this distance to draw a circle at the camera's center with a radius of the minimum distance multiplied by two *(think about why we do this!)*. As you move the mouse around the screen, you should notice that the circle is drawn out to touch the nearest object.

⚠ **Warning!** If you only multiply the distance of the circle by two and don't adjust where the circle is drawn, your circle will be drawn at an incorrect spot! So, be sure to multiply the distance by two, then draw it at the camera's center.





TASK FOUR:

12. Now cast multiple marches out into the world instead of just one. The idea is as follows:

    a. march out as far as you can until you collide with something.
    b. Compute the minimum distance from that point to every other object in the world and march out to that point.
    c. We need to eventually stop marching if the march has a small enough radius (say, 0.01). If the minimum distance from the current point to any other object is smaller than this threshold, we can deduce that we have collided with something and cast the ray. Let's start with the `March` class.

13. A march consists of four primary things:
    a. a circle
    b. a line
    c. a starting point
    d. an ending point.
    e. The line's length should be equal to the radius of the circle. Create the `March` class with these properties. Then, implement the drawing functionality.

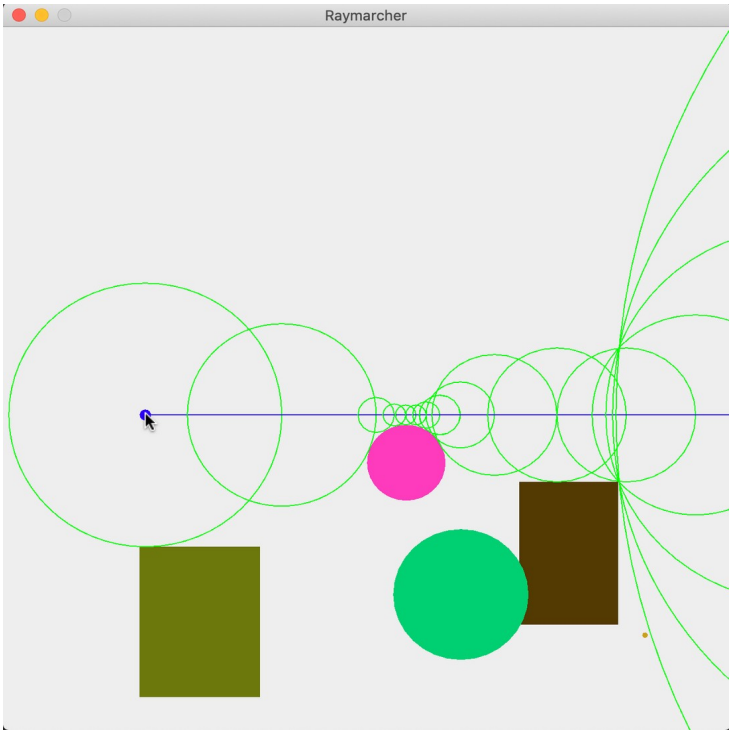
14. A ray consists of multiple marches. So, we can create a `Ray` class that receives a list of `March` objects. When drawing the ray, draw all the marches that are in its list.

15. Lastly, we need to add this new functionality to the panel. You should use a loop to keep track of the minimum distance between the current iteration point and any object in the world, and once

this goes below that threshold mentioned in step 12, break out, then initialize and draw a `Ray`. When marching, the next point should be created at the current point plus the length of the march (with no alterations to the y coordinate – see step 16 for more on this!).

Tips: if your program is freezing, check to make sure that your distance functions are correctly computing the minimum distance, and that your threshold isn't too low (below 0.01 can cause floating-point precision errors). Also, when setting up the loop to continue until the minimum distance is below the threshold, you most likely want a do-while loop because you want at least one iteration to complete prior to breaking out. Further, you may want to keep track of the ending position of the current point in the march – if it goes beyond the screen, you should terminate the loop. Finally, if you're noticing that, as you move the mouse closer to a point that it suddenly locks up, check to make sure all coordinates are floating point and non-integer.

⚠ **Warning!** The same bug with the circle's rendering location occurs here if you don't offset it like step 11 informed you.



TASK FIVE:

16. This is nice, but wouldn't it be neater if we could rotate that ray? First, let's consider what is going on when marching. As mentioned previously, we're only advancing along the x-axis and not the y-axis. This makes drawing our marches (and hence the ray) easy since there's no trigonometry involved. But, to advance along both axes, we need a new field in the `Camera` class to keep track of the angle. After this, we need a way of modifying said angle. There are a few ways to do this, but we'll go with a mouse approach again. Let's suppose that when the user

clicks the left mouse button, their camera angle will increase, and decrease if they click the right mouse button. To implement this, we need to use the `MouseListener` interface, and again, it will require that you implement a whole bunch of methods but we only need to use one: `mousePressed(MouseEvent)`.

If you aren't keen on writing a bunch of dummy overrides for interface methods you won't be using check out the `MouseAdapter` class

https://docs.oracle.com/javase/8/docs/api/java/awt/event/MouseAdapter.html

17. In `mousePressed`, a `MouseEvent` is supplied, just like `mouseMoved`. The difference is that we will be using the `getButton` method to determine which button was pressed instead of checking for position. To check which button was pressed, use

    ```
    event.getButton() == MouseEvent.BUTTONX
    ```

    where `X` represents the button (1 is the left mouse button, 3 is the right button). Write the code to increment the angle by 1 if the left mouse button was pressed, and decrement by 1 if the right button was pressed.

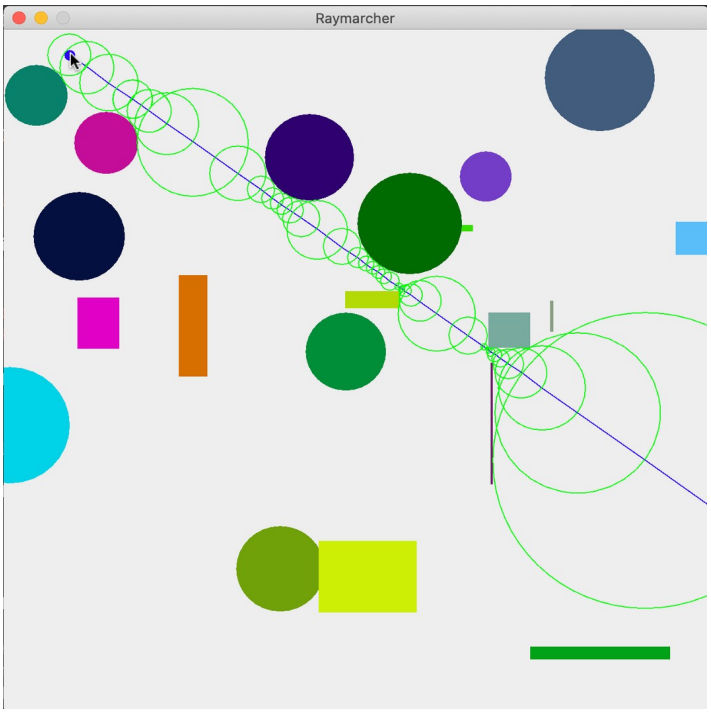
    After this, add the `Camera` instance as a `mouseListener` object to the panel.

18. So, this doesn't change much if you run it. However, now we can update our ray drawing procedure. To do this, we can use polar coordinates.

    We have our starting coordinate pair $P1$, and the minimum distance from $P1$ to any object in the plane is l. The camera's angle is t in radians. We wish to compute $P2$, the ending coordinate pair to this line. Thus,

    $$P2_x = P1_x + l * \cos(t)$$
    $$P2_y = P1_y + l * \sin(t)$$

    Use this logic to update your code.



Try this out: You may notice that constantly clicking the mouse to change the angle is a bit cumbersome. Use some of the other `MouseListener` *methods to change this functionality!*

RUBRIC

Criteria	Points
Program successfully compiles and executes without any errors.	Out of 25 points.
Program has high-quality style (i.e., appropriate comments and indentation).	Out of 15 points.
Inheritance hierarchy, abstraction and polymorphic behaviors created properly	Out of 5 points.
A list of `Shape`s exists in the `RaymarcherPanel` class	Out of 5 points.
The `Shape` list is populated with random shapes, with random colors	Out of 5 points.
The `Camera` and `March` classes are correctly created.	Out of 5 points.
The `Drawable` interface is used. `Shape` (and subclasses) , `Camera`, and `March` implement `Drawable`.	Out of 5 points.
The `MouseMotionListener` is created and correctly instantiated and implemented.	Out of 5 points.
A polymorphic solution to compute the minimum distance from the cursor to all objects in the scene is correctly implemented.	Out of 10 points.
A circle extends out from the cursor to the closest object in the scene, measured from the method created in the previous step.	Out of 5 points.
The ray is constructed correctly where each circle is drawn at the appropriate distance.	Out of 10 points.
The ray can be rotated via user input (either mouse or keyboard).	Out of 5 points.