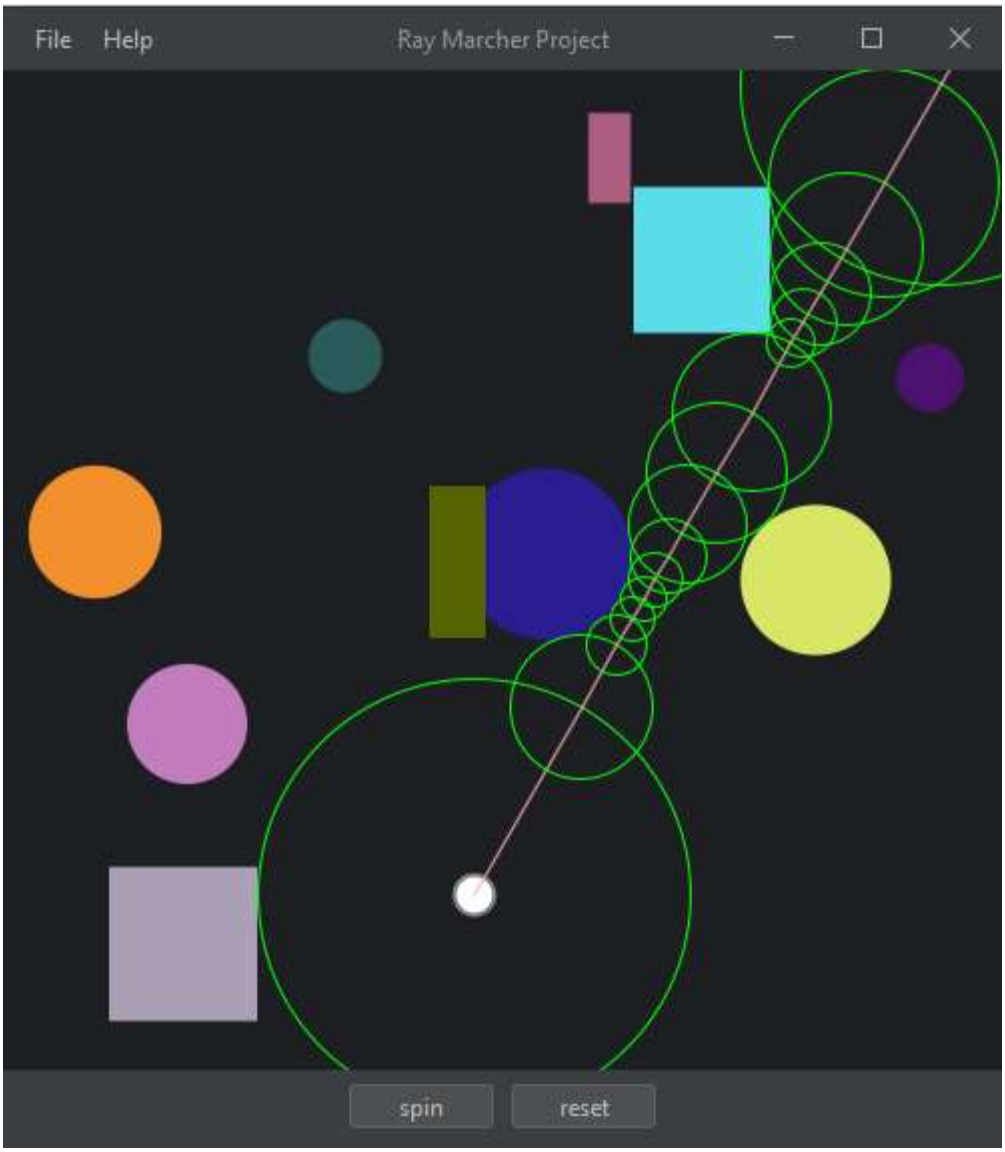
Two-Dimensional Ray Marching

Objective

By the end of this lab, students should be able to use iteration to determine the distance between a point and several objects in a plane, change angle-of-view and position via mouse input, and modify graphics/shapes in a plane. Students should also be able to understand the benefits of raymarching over other two-dimensional collision detection techniques.

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 need to do this. A computer must, mathematically, determine when a collision with an object occurs to not erroneously pass through that object (we're ignoring translucency!). A naive solution is to extend, or march, the ray outward in intervals of a predetermined unit, checking for collision along its path. This raises an obvious question: what is a unit, and what about the interval?

What to do.

1. First, build the graphical user interface. The drawing will be done on a JPanel via overriding the paintComponent method. You may build this from scratch or use a template from a previous lab or project. We'll be using Java's Graphics2D class which extends the Graphics class. It provides more sophisticated control over geometry, coordinate transformations, color management, and text layout.

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2D = (Graphics2D)g;
    // todo
}
```

2. Since we're going to be creating an environment for rays to collide with, we obviously need objects for the ray to collide with, right? So, let's do that. Create two subclasses called RectangleObject and CircleObject that extend an abstract superclass CollisionObject. The idea is this: we're going to populate our world with random rectangles and circles. A CollisionObject should have, at minimum, an *x*/*y* coordinate pair. RectangleObject should have width/height fields, and CircleObject should have a radius/diameter field, whichever you prefer.

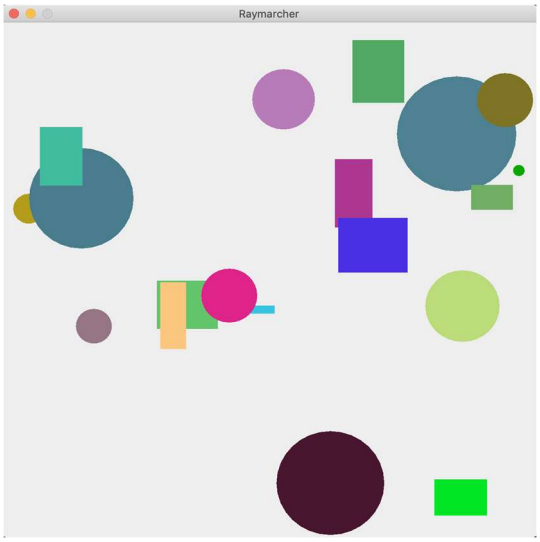
⚠ **Warning!** Make sure that you use double variables when initializing positions. When drawing with Swing, you can do one of a few things: either create objects with the java.awt.geom package that allow for explicit doubles when instantiating shape objects (e.g., Line2D, Rectangle2D, Ellipse2D) and then draw those with g2d.drawShape(), or cast doubles to integers and use other methods in Graphics2D. Later, when we perform arithmetic on the positions and dimensions, floating-point operations are crucial to ensure we don't encounter integer truncation issues.

```
Line2D.Double line2D = new Line2D.Double(x1, y1, x2, y1);
Rectangle2D.Double rectangle2D = new Rectangle2D.Double(x, y, w, h);
Ellipse2D ellipse2D new Ellipse2D.Double(x, y, w, h);
```

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

3. Create a list of CollisionObjects with random dimensions and positions. The size of the list doesn't necessarily matter but try to keep it lower than twenty (20) objects. Also, make sure that objects do not generate outside the world! And the rectangles may have any orientation.

4. At this point, you should have a fully populated list of CollisionObjects. It is now time to draw them! Note that JComponents have the paintComponent(Graphics g) method for drawing. We're going to do something similar. Since we're going to be drawing objects besides CollisionObjects, we should create an interface that says something is "drawable". Create an interface called Drawable with the method signature void drawObject(Graphics2D g2d). From here, implement the interface in CollisionObject and override its method in your subclasses. Now, add the functionality to draw the shapes. Finally, in your panel class, iterate over your list of objects and call drawObject on each one. When drawing the objects, make sure you apply the correct math offsets.



5. We're now ready to start our ray marcher! The first thing we need is some type of "camera" or perspective to start at. It also would be a little boring if we could only march rays in one direction, right? So, we'll need to add a listener to our camera, but we'll get to that as we go. Firstly, create a class called Camera and another called March. Camera will be where the ray begins marching, and March will be a single step, or iteration, in the ray march. Both will have *x*, *y* coordinates and radii. This is almost identical to the CircleObject class, and we could reuse it, but because they serve different purposes (and we're going to add more to it), we'll just rewrite a new class. We'll first write the Camera class since it is more interesting.

6. Camera, as we mentioned earlier, is the starting point of our ray march. So, like CollisionObject, 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 CircleObject: draw the camera at the provided *x* and *y* coordinates.

7. Now, we're ready to move our camera! There are two ways we can do this: with keyboard input or mouse input. We will choose the latter. As we move the mouse around the world, we want our camera to follow us. Thankfully, Java provides a very nice MouseMotionListener interface for us to implement. You will be required to override two methods(unless using MouseAdaptor), 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 and call repaint(). Run the program and you should see your camera move as you move the mouse.

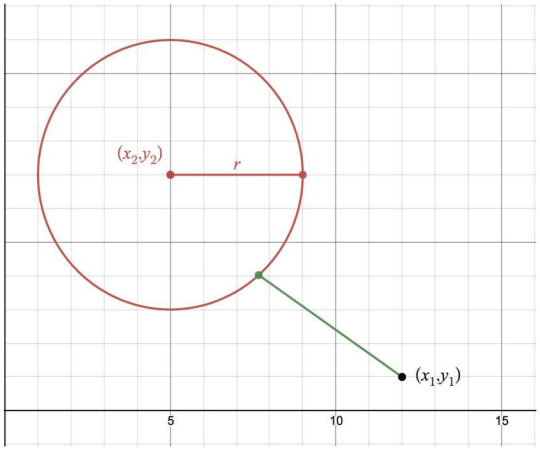
⚠ **Warning!** If you place the camera's draw method above the loop where you draw the objects, you won't see it if your mouse is over an object.

8. Now, let's begin the raymarching! As we mentioned, 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 how to compute this. 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.

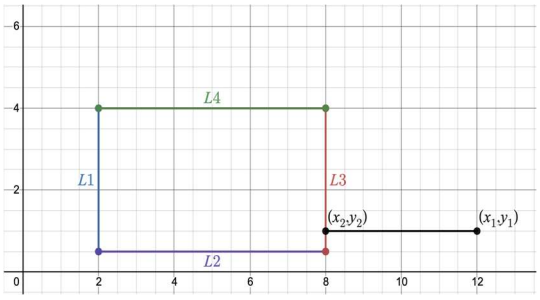
   For circles, we need to account for only one thing: the radius. Take the distance (also called the magnitude if you're familiar with vectors) from the camera to the center of the circle and subtract its radius. The formula is as follows:

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

   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.

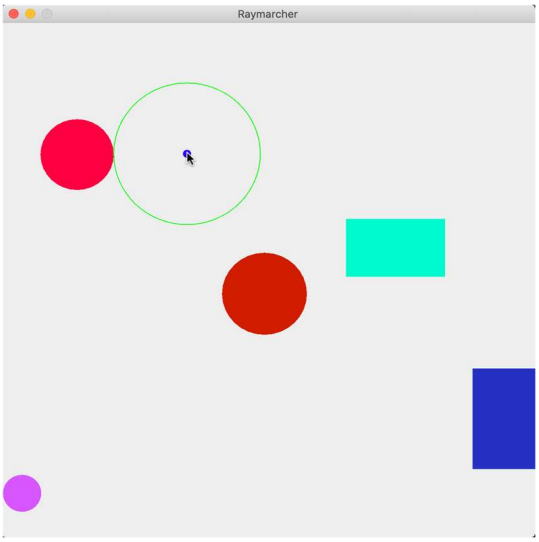


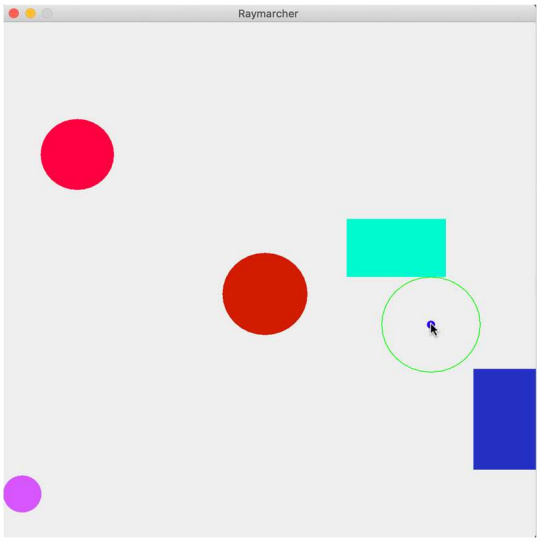
   Rectangles are a bit more complicated since we're involving both width and height instead of just a radius. The simplest way to do it is to compute the distance between the camera and the line segments that make up the rectangle. Line2D provides a great method for computing this distance: ptSegDist. There are four line segments that make up each rectangle, so just take the minimum of all four segments. 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 using the aforesaid methods.



   A good idea would be to create an abstract method computeDistance(double cameraX, double cameraY) in CollisionObject which is overridden and implemented in your subclasses.

9. 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. As you move the mouse around the screen, you should notice that the circle is drawn out to touch the nearest object.





10. We're almost there! What we need to do now is cast multiple marches out into the world instead of just one. The idea is as follows: march out as far as you can until you collide with something. Compute the minimum distance from that point to every other object in the world and march out to that point. 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.

⚠ **Warning!** The ray marching has 2 base cases to stop. You may use a while or a do while loop, but you should stop when the distance is less than the threshold of .01 or when the march moves off the screen(eventually in all 4 directions).

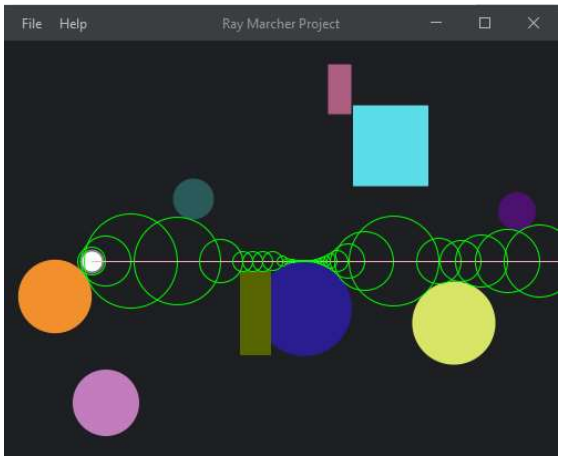
11. A march consists of two primary things: a circle and a line. The line's length should be equal to the radius of the circle. Create the March class with these properties. Then, implement the drawing functionality.

12. 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.

13. 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 10, break out. 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 yet – first test along the x-axis).

    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!



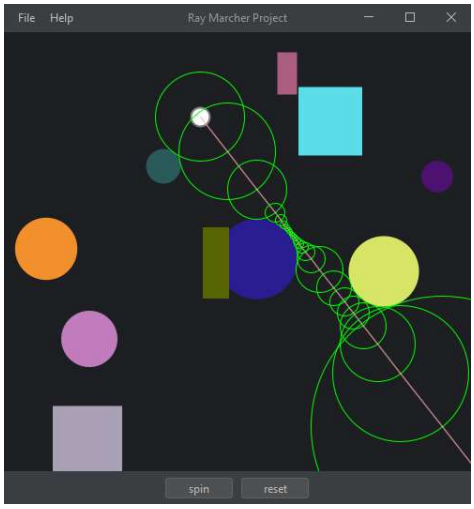
14. This is nice, but wouldn't it be neater if we could rotate that ray? It sure would be! First, let's consider what is going on when marching. Currently, we're only advancing along the x-axis and not the y-axis. This makes drawing our marches (and hence the ray) easy since there's no trigonometry involved. 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 you must choose to rotate with the left/right arrow keys or left/right mouse clicks. Use KeyListener and MouseListener respectively.

15. We need to update our ray drawing procedure. To do this, we can use polar coordinates.

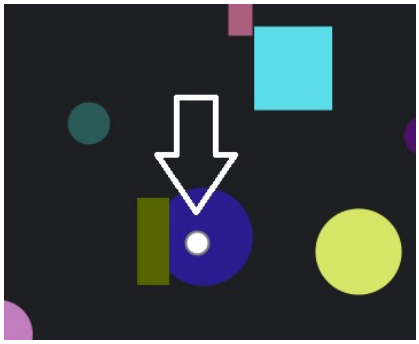
    We have our starting coordinate pair *P*1, and the minimum distance from *P*1 to any object in the plane is *l*. The camera's angle is *t* in radians. We wish to compute *P*2, the ending coordinate pair to this line. Thus,

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

    Use this logic to update your code and see what it does now.



16. You may notice that your camera tries to ray march inside of objects. Add an abstract method contains(int x, int y) to the abstract class CollisionObject and implement it in the subclasses. Ensure the camera is not inside a shape prior to marching.



17. And that's it! You've successfully created a sphere tracing ray marcher. Continue to add new things such as different shapes! You can add triangles and polygons in the same way that we did the rectangles. Add a MouseWheelListener to rotate using the mouse wheel. Add a reset, spin, and addShape buttons. Spinning can be implemented with a Timer.

```
Timer timer;

timer = new Timer(10, x -> {
      // add code to be called every xx milliseconds(10 above)
});

// isRunning(), start() and stop() will be useful
```