Using Trigonometry and p5.js to Make a Simple Game

1. **In the setup() function, type the following lines of code.**

```
createCanvas(500,500);
background('lightblue');
angleMode(DEGREES);
rectMode(CENTER);
strokeWeight(2);
frameRate(20);
```

2. **The drawSoldier() function is already defined for us:**

```
function drawSoldier(x, y, dir){
  translate(x, y);
  rotate(dir
  stroke('black');
  rect(0,-20,10,20);
  ellipse(0,0,30,30);
  resetMatrix();
}
```

3. **In the draw() function, *call* the drawSoldier() function using the line of code below. Then, experiment with different numbers as arguments.**

```
drawSoldier(100, 200, 90);
```

4. **Eight global variables have already been defined for us and set to 0. Take a look at them:**

```
var soldierX = 0;
var soldierY = 0;
var soldierDir = 0;

var goalX = 0;
var goalY = 0;

var xDistanceToGoal = 0;
var yDistanceToGoal = 0;
var triangleAngle = 0;
```

5. **In the draw() function, use soldierX, soldierY, and soldierDir as arguments in the drawSoldier() function.**

```
drawSoldier(soldierX, soldierY, soldierDir);
```

Then, try controlling the position and direction of the soldier using the lines of code mentioned in step 4.

6. **In the mousePressed() function, type the following lines of code. The first two lines will capture the coordinates of the mouse when it is pressed and save those coordinates as `goalX` and `goalY`. The next two lines calculate the distance between the soldier and the goal.**

```
goalX = mouseX;
goalY = mouseY;

xDistanceToGoal = Math.abs(goalX - soldierX);
yDistanceToGoal = Math.abs(goalY - soldierY);
```

We using Math.abs() to convert the distance into positive numbers, which will prevent us from getting confused by negative numbers. The other lines of code in mousePressed() are for debugging purposes. Make sure your web console is open so that you can see the debugging code in action!

7. **The pointSoldier() function is partially completed for us. Think about what the line `line(soldierX, soldierY, goalX, goalY)` means.**

```
function pointSoldier(){
  stroke('green');
  line(soldierX, soldierY, goalX, goalY);

}
```

***Call* the pointSoldier() function in the draw() function. Then, try clicking around on the canvas.**

```
pointSoldier();
```

8. **Add `background('lightblue')` above the other lines of code in the draw() function. Then, click around on the canvas. Think about why we needed to add this line *above* the other lines.**

```
background('lightblue');
drawSoldier(soldierX, soldierY, soldierDir);
pointSoldier();
```

9. **Add these lines to the pointSoldier() function. Then, click around on the canvas.**

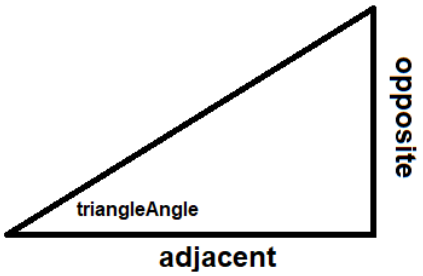
```
stroke('blue');
line(soldierX, soldierY, goalX, soldierY);
stroke('red');
line(goalX, soldierY, goalX, goalY);
```

These lines will help us visualize the triangle that we'll use to move and turn the soldier. The blue line is a visualization of `xDistanceToGoal` and the red line is a visualization of `yDistanceToGoal`.

10. **Now, it's finally time for some trigonometry! We need the soldier to point in the same direction as the green line (the "hypotenuse"), which means we need the angle made by the green and blue lines. There are many ways to find this angle, but we're going to use the lengths of the <u>blue and red lines</u> (the "adjacent" and "opposite" lines), since we've already stored their lengths as `xDistanceToGoal` and `yDistanceToGoal`. In the mousePressed() function, add this line below where you set `xDistanceToGoal` and `yDistanceToGoal`:**

```
triangleAngle = atan(yDistanceToGoal / xDistanceToGoal);
```

The atan() function is the reverse of the tan() function. You might remember it as $\tan^{-1}$ from math class.



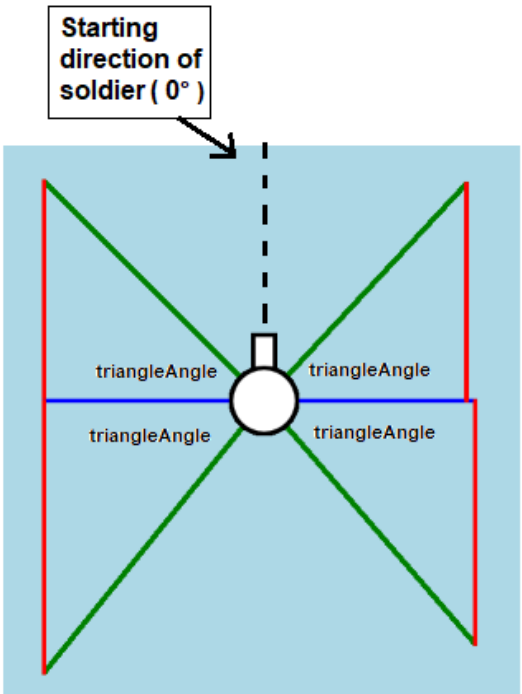
$$\tan(\text{triangleAngle}) = \frac{\text{opposite}}{\text{adjacent}} \quad \Rightarrow \quad \text{triangleAngle} = \text{atan}\left(\frac{\text{opposite}}{\text{adjacent}}\right)$$

11. **Now we need to use `triangleAngle` to change the direction that the soldier is pointing. Add this line of code to the pointSoldier() function:**

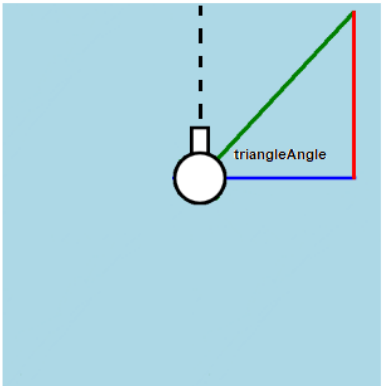
```
soldierDir = triangleAngle;
```

Try clicking around on the canvas. You'll notice that it's not working yet! That's because `triangleAngle` can only give us a measurement between 0° and 90°. But our soldier needs to be able to turn between 0° and 360°.

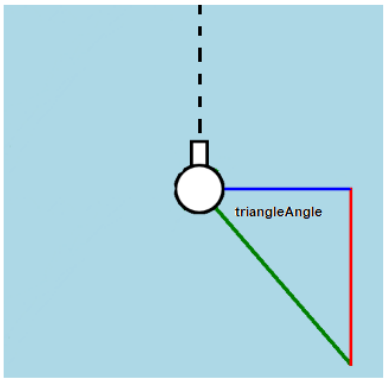
Furthermore, the soldier is turning from an initial position of 0°, or straight-up (because of how the teacher initially draw it in the drawSoldier() function). In contrast, the `triangleAngle` we're using for our calculations is NOT BEING MEASURED FROM THAT SAME 0° REFERENCE POINT. Instead, `triangleAngle` is being measure in exactly four different ways (depending on where we click).



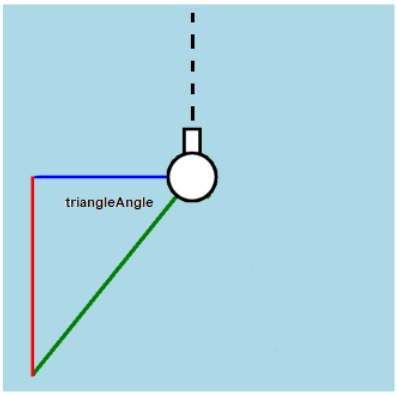
We'll need to do some angle addition and subtraction to ensure that the soldier is pointing the way we want it to point.



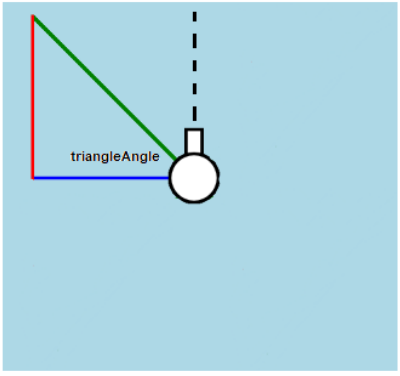
If we click to the **top-right** of the soldier, we'll need to turn the soldier `90 - triangleAngle.`



If we click to the **bottom-right** of the soldier, we'll need to turn the soldier `90 + triangleAngle.`



If we click to the **bottom-left** of the soldier, we'll need to turn the soldier `270 - triangleAngle.`



If we click to the **top-left** of the soldier, we'll need to turn the soldier `270 + triangleAngle.`

12. **Delete the line** `soldierDir = triangleAngle`. **Then, uncomment the if-statements that are already in your pointSoldier() function. Two of these if-statements are completed for you. See if you can make the other two work.**

13. **Now it's time to make the soldier move. Start by writing these two lines of code in your moveSoldier() function.**

```
var singleXMovement = 1;
var singleYMovement = 1;
```

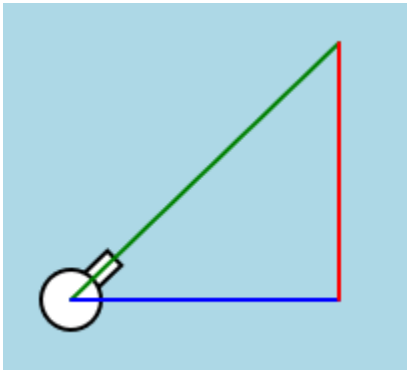
Next, uncomment the if-statements in your moveSoldier() function. Replace "YOUR CONDITIONS" with the conditions that you wrote in step 12. Try to complete the two empty if-statements by looking at the other if-statements in the moveSoldier() function.

```
function moveSoldier(){
  var singleXmovement = 1;
  var singleYmovement = 1;

  if (goalX > soldierX && goalY < soldier){
     soldierX = soldierX + singleXMovement;
     soldierY = soldierY - singleYMovement;
  }
  if ("YOUR CONDITIONS"){

          TRY TO COMPLETE THIS IF STATEMENT

  }
  if ("YOUR CONDITIONS"){

          TRY TO COMPLETE THIS IF STATEMENT

  }
  if (goalX < soldierX && goalY < soldierY){
     soldierX = soldierX - singleXMovement;
     soldierY = soldierY - singleYMovement;
  }
}
```

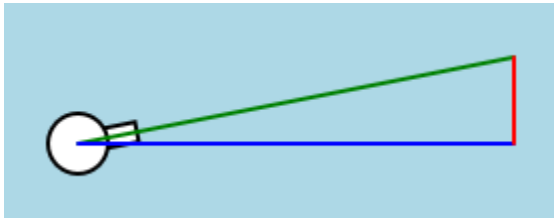
14. **Call the moveSoldier() function in the draw() function. The draw() function should now look like this:**

```
background('lightblue');
makeSoldier(soldierX, soldierY, soldierDir);
pointSoldier();
moveSoldier();
```

15. **Click around on the canvas. You'll notice that the soldier doesn't yet move correctly. Currently, it only moves properly if you click at exactly a 45° angle from the soldier. This is because our** `singleXMovement` **and** `singleYMovement` **variables are static—they don't change according to the shape of the triangle. They're always equal to 1 even when** `xDistanceToGoal` **(blue line) and** `yDistanceToGoal` **(red line) are different sizes.**

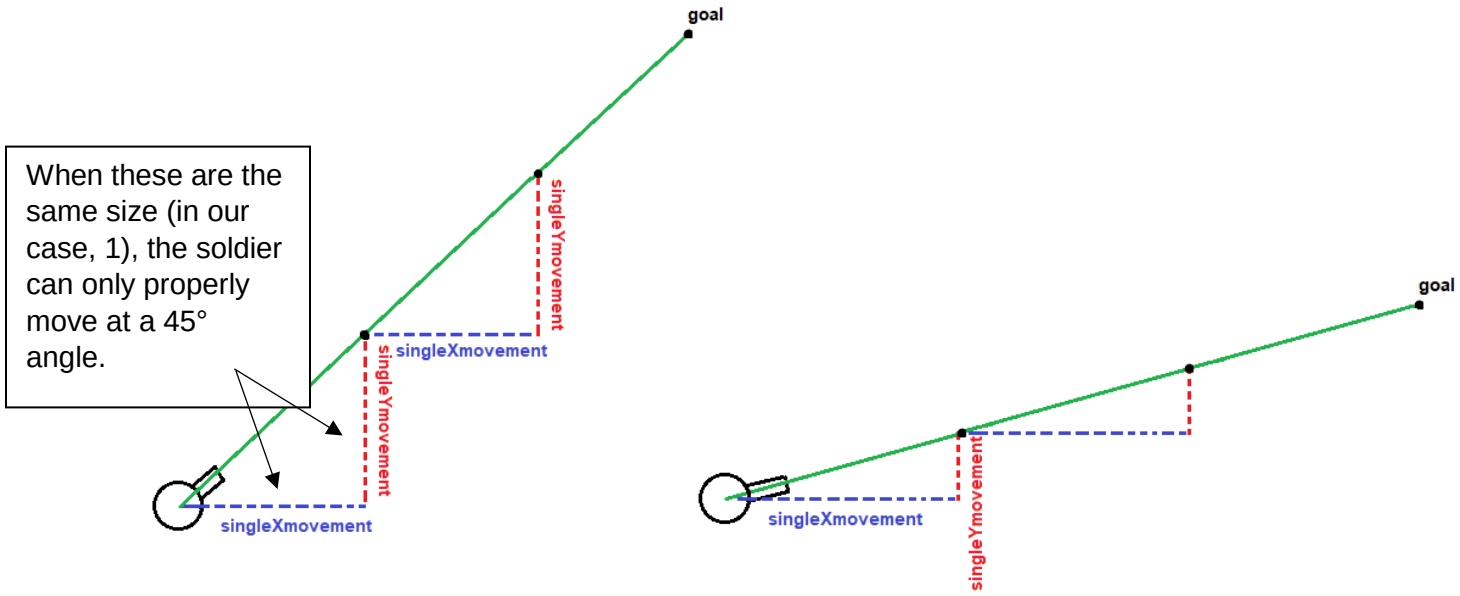


This sort of works.

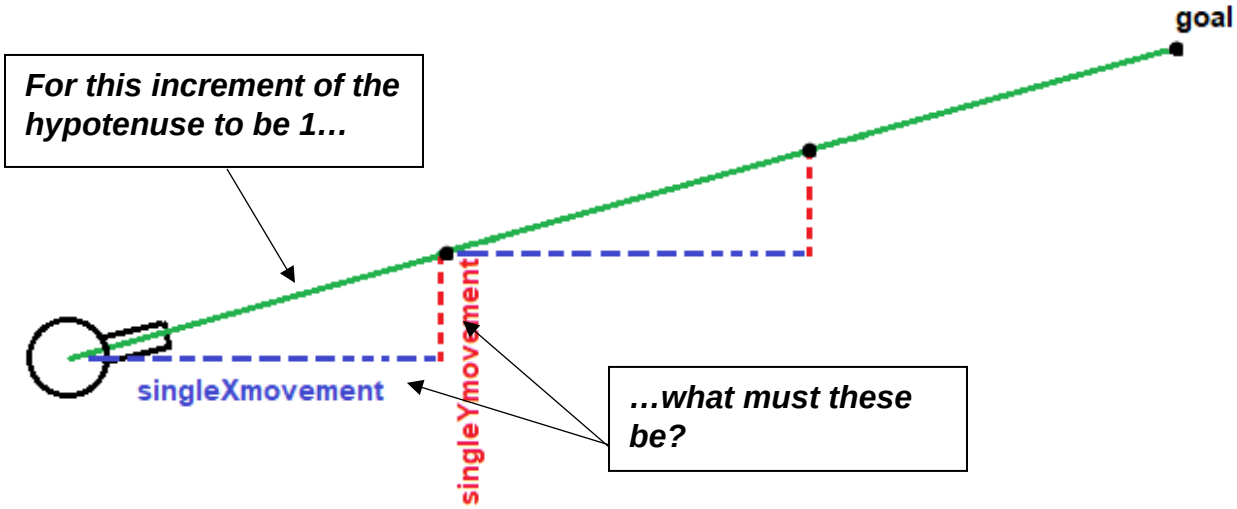


This definitely does not work.

If the red line is shorter (as seen above), then the soldier's `singleYMovement` needs to be shorter than its `singleXMovement`, since it has less Y distance to cover. The opposite is true if the blue line is shorter.



Therefore, instead of setting `singleXmovement` and `singleYmovement` to 1, let's calculate what those variables would be IN ORDER TO MAKE THE HYPOTENUSE EQUAL TO 1.



To calculate those variables, we'll use the sin() and cos() functions.

In the image above, `triangleAngle` **is about 15°. Furthermore,** `singleXmovement` **can be called the "<u>adjacent</u> side". Let's use that information to demonstrate how** `singleXmovement` **is calculated using the cos() function.**

$$\cos(\text{triangleAngle}) = \frac{\text{adjacent}}{\text{hypotenuse}} \Rightarrow \cos(15°) = \frac{\text{adjacent}}{1} \Rightarrow 0.9659 = \text{adjacent}$$

$$0.9659 = \text{singleXMovement}$$

or

$$\cos(15°) = \text{singleXMovement}$$

The same process is used to calculate `singleYmovement` **(a.k.a. opposite side) using the sin() function.**

16. **Therefore, to ensure that the soldier is moving properly, we simply need to set** `singleXmovement` **and** `singleYmovement` **to the following:**

```
var singleXMovement = cos(triangleAngle);
var singleYMovement = sin(triangleAngle);
```

BONUS MATERIAL:
ADD ALIENS!

17. **Create an Alien class:**

```
class Alien{
  constructor(){
    this.x = random(500);
    this.y = random(500);
    this.distanceFromSoldier;
    this.show = true;
  }

  drawAlien(){
    if (this.show === true){
      translate(this.x, this.y);
      stroke('black');
      ellipse(0,0,30,10);
      ellipse(0,0,10,30);
      resetMatrix();
    }
  }

  calcDistanceFromSoldier(){
    var result = dist(this.x, this.y, soldierX, soldierY);
    this.distanceFromSoldier = result;
  }
}
```

18. **Create a empty global array called** `alienList`**:**

var alienList = [ ];

19. **In the setup() function, create some aliens and add them to your** `alienList`**:**

```
for (var i = 0; i < 10; i++){
  var anAlien = new Alien;
  alienList.push(anAlien);
}
```

20. **Have the draw function continuously loop through your** `alienList` **in order to draw the aliens and in order to remove an alien when the soldier gets close to it.**

```
for (var i = 0; i < alienList.length; i++){
    alienList[i].calcDistanceFromSoldier();
    if (alienList[i].distanceFromSoldier <= 40){
      alienList[i].show = false;
    }
    alienList[i].drawAlien();
  }
```