Alright!

Hopefully by now you understand the basics of what a game loop is, and how we look at the screen as this Cartesian plane with an x and y starting from the top-left corner of the display.

But we don’t always think of problems only using the x-y Cartesian approach. There are problems where it makes more sense to define points in space by saying how far away they are from the origin and how much they have rotated from the x-axis.

We can express these points with two values… a “distance” from the origin, and an “angle”.

<gif angle-distant> <wait!!!>

So, instead of saying an object is at point (x=7, y=5), we can now say that the object is 6 units away from the origin and it travelled an angle of 45 degrees counter-clockwise from the x-axis.

The distance “6 units” from the origin can be seen as the radius of the circle that is produced by the rotating the object.

<gif rotation>

This is what mathematicians usually call Polar coordinates. It is nothing more than a different way of describing the position of something in space. Before we used x-y values, and when we think now of Polar coordinates, we use a distance from the origin, and an angle.

This is useful for us, as programmers, because, as mentioned, some problems are easier to solve when we think of them in Polar coordinates.

<gif animation solar system>

But there is one gotcha…

Even though it’s easier to think of the problem as angle and distance from the origin, Lua still expects us to provide an x-y point when we want to draw an object on the screen.

Therefore, in order to draw the object in screen, we need to convert the Polar distance-angle value to a Cartesian x-y point.

<wait>

To solve this, we need a bit of help from mathematics.

<image angle-distance, x-y>

Let’s look at how to convert a Polar coordinate (distance/angle) to Cartesian coordinate (x/y).

If we look at the Polar coordinate, we have the radius of the circle, and the travelled angle. That gives us our point in the plane.

We need to find the x and the y values that are equivalent values of that point.

So, if we project the values down to the x-axis and left to the y-axis, we will end up with a triangle. This triangle will help us find the x and y coordinates of our point.

The base of this triangle is the x value, and the height of the triangle is the y value.

Mathematicians have a thing for weird names. When they talk about right-triangles they use terms like the adjacent side, or the opposite side, and the hypotenuse.

It is really very easy. The opposite side is the side that is opposite to the angle we are talking about. In this case, the opposite side is equal to the height of the triangle. Meaning that the opposite side is our y value.

<opposite>

The adjacent is the other side, the one that runs along the base. Which means, the adjacent will be the x value we are looking for.

<adjacent>

The hypotenuse is the missing side. The longest one, which we already have. This is the radius of the circle, the distance from the object to the origin.

Ok, so… since the adjacent side of the triangle is the x coordinate, and the opposite side is the y coordinate, how do we discover the adjacent and the opposite values?

Here we use other useful properties straight from the high school math book.

The ratio between the adjacent side and the hypotenuse gives us something called the “cosine” of the angle.

<wait>

And the ratio between the opposite side and the hypotenuse is the “sine” of the angle.

<wait>

Lua has built-in functions to calculate the Sine and Cosine of an angle. That means we have everything we need to calculate the value of the adjacent and opposite sides of our triangle, which also means we will be able to find the x and y position of our game object.

<image ratios>

With a little bit of algebra, we shuffle the ratio formula from before, and we get:

Adjacent = cos(angle) \* hypotenuse

Opposite = sin(angle) \* hypotenuse

Which is the same as:

x = cos(angle) \* radius

y = sin(angle) \* radius

<wait>

There we go! This formula gives us the Cartesian x and y coordinates based on the Polar distance and angle values.

x = cos(angle) \* distance

y = sin(angle) \* distance

We use the converted x and y values to tell Lua where to draw the object in the screen.

x = 0

y = 0

angle = 0

distance = 50

function update()

angle = angle + 0.05

x = math.cos(angle) \* distance

y = math.sin(angle) \* distance

end

function draw()

circle(x, y, 10)

end

<wait…>

In the above example, the variable angle starts at 0 and we add 0.05 on each frame of the update loop.

Observe that we use the Lua built-in functions “math.cos()” and “math.sin()” to get the cosine and sine of that angle.

Ok, you are probably looking at this example and thinking “Wait a minute! An angle that grows only 0.05 each frame? Isn’t that value too small? Aren’t angles usually measured in 45 degrees, 90 degrees, 180 degrees, etc.?”

I hear you, and you are absolutely correct! We are used to think and talk to other people about angles using the “degrees” unit. But computers usually expect angles to be provided in a unit called “radians.”

Radians are a different way of thinking of angles, and their value is linked to the radius *r* of the circle. If we travel our circle a distance equivalent of 1 *r* (length), that gives us the measurement of 1 radian (angle).

Important to notice that, if we travel 180 degrees in the circle, that is the equivalent of approximately 3.14 radians. That also means that a full 360 degree rotation is approximately 6.28 radians.

<wait for 10 seconds>

180 degrees = PI radians

360 degrees = 2\*PI radians

We can use the properties above to convert from radians to degrees, and back from degrees to radians.

For now, always remember that when we talk angles with the computer, it will expect us to provide the values of the angles in radians.

When you talk with your friends or other programmers, chances are you will still talk about angles as degrees. I never heard anyone say they performed a radical 2PI radians maneuver with a skateboard.

Ok, we really covered a lot in this talk!

Now I have a quick task for you to put in practice what we just learned.

I have a small game that needs to draw the sun in the center of the screen, and a blue planet that rotates around the sun object…

<gif image sun-earth>

Open the code editor and analyze what the script is doing.

You will see a variable angle, that is being incremented in 0.05 on each frame of the game loop, and you will also notice that we have something new.

This is the first time we are using tables in Lua. Tables are an option to organize our code a bit better.

You will see that we wrap values inside two tables. In this example, the tables names are “sun” and “planet”.

<image sun planet objects>

The table “sun” has the values x, y, and radius. And the table “planet” has its own values x, y, and radius.

<objects-sun-planet>

Currently, the planet object is moving to the right and down.

<image planet-moving-right>

See if you can use the concepts we just studied to make the planet rotate around the sun.

<image sun planet rotates>

I will try to write some helpful tips and diagrams in the whiteboard.

As always, once you are done, just type “exit” in the terminal and we can proceed to the next task.

Have fun…