Salutations, I’m Jesco and you’re watching Game Dev Made Easy. In the video today, we are going to explore state in DragonRuby.

DragonRuby is very special with how it can handle objects and has some built in features that makes moving and manipulating objects easier.

Args dot state is one of those features. What is args dot state? It is an instance state that GTK holds internally and hands to your tick function as an argument every frame. It lives longer than the tick function, as a matter of fact, it lives for the entire life of the application and is stored and always available inside of the engine but cannot be accessed like a global function.

Without further ado, let’s dive into the code and see how args dot state works.

Sign up for my discord server to chat with me and other likeminded individuals. Link is in the description!

Now… Before we write code that utilizes state, we need to write code that doesn’t.

Inside of your main dot rb file, clear out everything.

Is it clear yet? Did you do it? Ugh, fine… I’ll wait…

(Insert impatient waiting meme https://www.youtube.com/watch?v=EAzeFCxs3IM)

Alright, now that you have FINALLY caught up with me

(insert tongue sticking out here)

Write def tick args and directly underneath it, write end.

We are basically defining the main entry point for DragonRuby at this point and since ending a function block requires the end keyword, we went ahead and did so.

Create a line space in between the def and end lines.

Within that line spacing, write x underscore position is set to be 100.

We are creating this variable to store the horizontal position of the object.

Now on the next line, let’s create the object we want to manipulate.

By writing Args dot outputs dot solids less than sign less than sign then adding an opening bracket, which is proceeded by the x underscore position variable comma 250 comma 50 comma 50 comma zero comma zero comma zero and a closing bracket.   
  
We have successfully told DragonRuby to draw the solid at this specific x position, y position, width, height with this specific red, green and blue value. Which, in this case is black.

Now, for the final line of code to make this all complete.   
x underscore position plus equals 10. Which should make the x position update every frame by 10 right?

(Insert yelling RIGHT meme https://www.youtube.com/watch?v=JWQsmbxc0\_s)

Let’s hit play mode and see what happens.

Alright, so the solid is drawn at the x position being at 100, but it never changes. Now why is that?

(My name Jeff meme https://www.youtube.com/watch?v=6CKoBtMsdSw)

The reason this happens is due to how DragonRuby works internally. So what happens is that every frame, the array values are wiped and repopulated with the default values to draw on screen every frame (which in this case is a tick). The x underscore position is a local variable which goes away at the end of the function since it is a local variable and thus will always return the value of 100.

Utilizing state fixes this issue and allows for values to increment every frame. So, let’s just get into how to write the code and how it works shall we?

(Get on with it already! https://www.youtube.com/watch?v=sXE8LdXzeHM)

By the way, don't forget that if you find this tutorial useful, please consider throwing a buck or two towards my PayPal or even consider joining my patreon? Links are in the description box below.

\*Ahem\*, enough shilling, back to the tutorial.

Let’s clear out all the code that resides within the tick function.  
Got it done? Good. Time to continue.

We are going to write args dot state dot x double pipe equals 60. The “double-pipe equals” is an operator that assigns a value, much like a plain equals symbol which is our classic assignment operator but will only complete the assignment if the left side of our operation returns false or nil. In Ruby, nil is the same thing as null.

The cool thing about state is that we don’t have to define what anything after it is attached to. We can write it and use it the way we want and DragonRuby will apply it as we call it.

In this scenario, x is going to designate horizontal axis position.

On the next line we are going to add args dot state dot y double pipe equals 250. We want this to be the vertical axis and position on said axis.

The next line is going to be args dot state dot w double pipe equals 50. The W will stand for width.

Following that is going to be args dot state dot h double pipe equals 50. The H stands for height.

We have 4 lines left to write out.

This line is going to be args dot state dot r double pipe equals 0. The R is the red color designated in RGB values.

The next line is going to be args dot state dot g double pipe equals 0. The G is the green color designated in RGB values.

The following line is going to be args dot state dot b double pipe equals 0. The B is the blue color designated in RGB values.

The last line is going to be args dot state dot speed double pipe equals 10. This is going to store our speed value.

That wasn’t too bad. But now we need to use these values somewhere.

Enter one of the 6 main primitive types in DragonRuby. We will utilize the solids primitive to just draw a filled in square.

Write args dot outputs dot solids. This is the basic syntax for accessing the primitive types found in DragonRuby.

Add a space in between solids and write two less than symbols. In Ruby, the two less than symbols back to back has many meanings. For our case, we are appending values to an array.

Add an opening and closing bracket. This will create an array. Inside we will set the values for what the solids primitive type requires.

It requires an x position, y position, width, height, r, g, and b value in its’ full form.

Inside of the brackets, write args dot state dot x comma args dot state dot y comma args dot state dot w comma args dot state dot h comma args dot state dot r comma args dot state dot g comma and args dot state dot b.

We have defined our solid with the values it requires and if we were to launch the DragonRuby dot exe file, it would just show a black box in the scene. Let’s actually make it move.

To do so, all we need to do is add one more line.

Write args dot state dot x plus equals 10. This is saying that we want to add 10 to the value every time this state is called.

And there we have it, we have successfully made this box move across the screen. But why don’t we go ahead and expand upon it a little more?

Remove the args dot state dot x plus equals 10 from the editor.

What we are going to do is write a simple if statement and then add our movement code back. The if statement is just going to check if our position is greater than a specific amount, if it is, then we want to reset the position.

Write, if args dot state dot x is greater than or equal to 1080. Make sure to have an end statement below this line.

Inside of the if statement, make sure to indent one time and write args dot state dot x is set to be 60.

Outside of the end statement, write args dot state dot x plus equals 10.

If you press the play button, you will see the black box move across the screen and when it reaches the end, it will reset the position to be 60 and then continue moving to the right.

That was pretty cool and all, but… We can go one step further with this. Let’s create our own lerp function similar to what we would see in other game engines.

Go ahead and delete the if statement and the movement code as we don’t need that anymore.

Outside of the tick’s end statement, we will define a new function.

Write def lerp parenthesis start comma stop comma step, closing parenthesis. While Ruby does not need the parenthesis, I find it easier to read with multiple parameters.

Make sure to add another line and write end to close the function declaration.

Inside of the lerp function write parenthesis 1 point 0 minus step closing parenthesis multiplied by start plus step multiplied by stop.

This is a more precise lerp function that guarantees that our v is going to equal our vector value when step is equal to 1.

Simple graph mathematics really. Linear interpolation is very interesting and fun, which also happens to have extensions for more spatial dimensions as well as forms the basis for spline interpolation and Bezier surfaces as well as 3D meshes. I would suggest getting the book called Curves and Surfaces for Computer Graphics by David Salomon to really dive into this.

Let’s go back to the tick function and add our code that utilizes this function to move the position.

Write args dot state dot x is set to be lerp, open parenthesis 0 comma 1200 comma open parenthesis Math dot sin open parenthesis args dot state dot speed multiplied by args dot state dot tick underscore count divided by 200 closing parenthesis. Plus one closing parenthesis divided by 2 closing parenthesis.

Now let’s break this beast of a function call down.

We are setting the x position of the object to be the results of lerping between the values of 0 and 1200.

The step is the most complicated part of this entire source. We sin the speed multiplied by the tick count and divide those results by 200. This will smooth out the speed to a more practical speed range. Then we add the sin results by 1 and divide by 2. This will always lock the movement range to be on screen.

Launch the DragonRuby executable file and watch as our little black box scrolls side to side on our screen in a loop.

Now you know how to utilize args dot state in DragonRuby. I sincerely hope you have enjoyed this tutorial today. If you did, give it a thumbs up, leave a comment, share with your friends and subscribe for more game dev related content with various game engines here on Game Dev Made Easy.