Blueprint Platformer Scripting Guide

Table of Contents

Introduction

This document contains the essentials on scripting using the Lua programming language in Blueprint Platformer. It is not designed to be a comprehensive guide to the Lua programming language but a reference for learning Lua in the context of Blueprint Platformer and a reference for the Blueprint Platformer functionality.

Why use Lua? Lua is widely supported scripting language used in game development both by game engines for less-technical users to extend the game and by games to allow users to write mods for that game. It is extremely simple to learn.

Basic Syntax

So what does Lua look like? Here is an example of a simple Lua script:

```lua
counter = 0

-- Each frame print the value of counter to the console
function Tick()
    counter = counter + 1
    Print("counter is " .. counter)
end
```

This example shows a number of important Lua concepts: - Variables are defined

using the `=` operator - Comments are defined using `--` so anything after `--` on a line is ignored - Functions are defined using the `function` keyword, these are reusable blocks of code that can be called from anywhere in the script using the function name and `()` - `Tick` is a special function that is called every frame - Strings are defined using `"`, these store text and can be concatenated (combined) using `..` - Numbers are defined as you would expect, `1`, `2`, `3`, etc and basic maths operations can be performed on them using `+`, `-`, `*`, `/`, etc.

You will learn more about each of these in the following sections of this guide.

Functions

A function, as mentioned previously, is a reusable piece of code. You will learn about creating custom functions later in this guide so for now will focus on the functions provided by Blueprint Platformer which you can use to solve puzzles.

As previously mentioned, you can invoke a method using the name of the function followed by `()` and any parameters that the function requires. For example, to print a message to the console you would use the `Print` function like so:

```
Print("Hello World!") -- Prints "Hello World!" to the console
```

Some functions take several parameters, these parameters are separated by `,`. For example, to move a platform, you will need to give the name of the object to move, x and y values to move it by e.g.:

```
Move("platform_1", 0, 1) -- Move platform_1 0 on the x axis and 1 on the y axis
```

The following functions are provided by Blueprint Platformer:

```
Move("obj_name", x, y) -- Move an object relative to its current position
MoveAbs("obj_name", x, y) -- Move an object to an absolute position
Rotate("obj_name", angle) -- Rotate an object relative to its current rotation
RotateAbs("obj_name", angle) -- Rotate an object to an absolute rotation
Scale("obj_name", x, y) -- Scale an object relative to its current scale
ScaleAbs("obj_name", x, y) -- Scale an object to an absolute scale
SetColour("obj_name", r, g, b, a) -- Set the colour of an object with a red, green, blue an
Print("text to log") -- Log a message to the in-game console (useful for debugging)
Destroy("obj_name") -- Destroy an object
```

Tick Function

The `Tick` function is a special function that is provided by Blueprint Platformer. This function is not defined by the engine but when present in a script will be called a stable 60 times per second. This is useful to moving an object over time rather than moving it instantly when the script is run.

Here is an example of using `Tick` to move a platform over time from a starting point e.g.:

```lua
-- This code is called when the script is run
MoveAbs("platform_1", 0, 0)

-- This code is called every frame
function Tick()
    Move("platform_1", 0.05, 0)
end
```

This will move "platform_1" to 0, 0 when the script is run and each frame will move it 0.05 on the x axis - moving the platform to the right gradually.

Variables

Variables are crucial to Lua and any programming language, they allow you to easily store a value and use it later (within the scope it was defined) e.g.:

```lua
counter = 0 -- Define a variable called counter and set it to 0

function Tick()
    counter = counter + 1 -- Set counter to its current value plus 1
    Print("counter is " .. counter) -- Print the value of counter to the console
end
```

Variables defined at the top of the file as scoped to the entire file. This means anywhere in that file you can access the variable. If you want to define a variable inside a function so it can not be used outside of that function, you use the `local` keyword e.g.:

```lua
function Tick()
    local counter = 0 -- Define a variable called counter and set it to 0

    counter = counter + 1 -- Set counter to its current value plus 1
    Print("counter is " .. counter) -- Print the value of counter to the console
end
```

This means if we try to use counter outside of `Tick`, it will throw an error as that variable is not defined outside of that function.

Note: `local` should not be used when defining variables at the top of the file and is not supported in this version of Lua.

Maths

You learnt about variables but what makes variables powerful is the ability to redefine them. Here are some basic operations we can use to redefine variables:

```lua
counter = 0 -- Re-define counter to 0
counter = counter + 1 -- Add 1 to counter
counter = counter - 1 -- Subtract 1 from counter
```

```
counter = counter * 2 -- Multiply counter by 2
counter = counter / 2 -- Divide counter by 2
```

Additionally, you can use these operations on pure numbers e.g.:

```
counter = 34 + 1 -- counter is now 35
counter = 34 - 1 -- counter is now 33
```

Note: If a variable is a string (text), we cannot add them together e.g.:

```
myVariable = "Hello" + "World" -- This will throw an error
```

Instead, to combine strings together, In Lua you use `..` e.g.:

```
myVariable = "Hello" .. "World" -- counter is now "HelloWorld"
```

Defining Custom Functions

You should have a basic understanding of using variables but you may want to write your own if there is code you will use in multiple places.

Functions are defined using the `function` keyword followed by a name for the function and `()` e.g.:

```
function MyFunction()
    -- Code goes here
end
```

Which we can call using `MyFunction()` e.g.:

```
MyFunction()
```

Functions can be nested inside of other functions e.g.:

```
function MyFunction()
    function MyNestedFunction()
        Print("Hello World!")
    end
end
```

However to keep things simple, we will not be using this feature in this guide.

Function Parameters

Sometimes you may want a function to take input in some way, we can give variables parameters (also known as arguments) which can then be used inside that function e.g.:

```
function MyFunction(myParameter)
    Print(myParameter)
end
```

The above functions takes one parameter, this parameter can then be referred to in the function using the name given in the function definition.

But sometimes you need more than one parameter, we can define multiple parameters by separating them with `,` e.g.:

```
function JoinStringsTogetherAndPrint(string1, string2)
    print(string1 .. string2)
end
```

Note: These parameters are only accessible inside the current function and each one must have a unique name.

Return Values

Say we want to create a function to join two strings together but instead of logging that to the console, we want to return it. We can use the `return` keyword to do this e.g.:

```
function JoinStringsTogether(string1, string2)
    return string1 .. string2
end
```

We can then use the function like so:

```
joinedString = JoinStringsTogether("Hello", "World") -- joinedString is now "HelloWorld"
```

Return is core to how functions work in any programming language as without it, we would need to create a file scoped variable to store the result of the function and then set that variable to the result of the function. Which is less than ideal.

Conditionals

Conditionals are essential to programming. They let you do different things based on a condition.

Before we jump into conditionals, we will talk about equality in Lua. In Lua we can check equality using `==` e.g.:

```
IsOneEqualToOne = 1 == 1 -- IsOneEqualToOne is now true
IsOneEqualToTwo = 1 == 2 -- IsOneEqualToTwo is now false
```

what these lines do is set the variable on the left to the results of the equality check e.g. `1 == 2` is always false so `IsOneEqualToTwo` is set to false.

When you see `==` think of it as "is equal to". Additionally, we can check if something not equal to something else using `~=`, e.g.:

```
IsOneNotEqualToOne = 1 ~= 1 -- IsOneNotEqualToOne is now false
IsOneNotEqualToTwo = 1 ~= 2 -- IsOneNotEqualToTwo is now true
```

When you see `~=` think of it as "is not equal to".

Finally, we can use more than, less than and their combined more than or equal to and less than or equal to e.g.:

```
IsOneGreaterThanOne = 1 > 1 -- IsOneGreaterThanOne is now false
IsOneLessThanOne = 1 < 1 -- IsOneLessThanOne is now false
IsOneGreaterThanOrEqualToOne = 1 >= 1 -- IsOneGreaterThanOrEqualToOne is now true
IsOneLessThanOrEqualToOne = 1 <= 1 -- IsOneLessThanOrEqualToOne is now true
```

When you see `>` think of it as "is greater than". When you see `<` think of it as "is less than". When you see `>=` think of it as "is greater than or equal to". When you see `<=` think of it as "is less than or equal to".

Now we can learn about if statements. An if statement will run code if a condition is true e.g.:

```
if 1 == 1 then
    print("1 is equal to 1")
end
```

As you can see, if statements are formed using the `if` keyword, the condition to evaluate, the `then` keyword and then the code to run if the condition is true. To end an if statement, you use the `end` keyword.

We can also run code if an if statement is not met using the `else` keyword e.g.:

```
if 1 == 2 then
    print("1 is equal to 2")
else
    print("1 is not equal to 2")
end
```

We replace the `end` keyword with the `else` keyword and then add the code to run if the condition is not met. After the `else` keyword, we still need to use the `end` keyword to end the if statement.

Finally, we can check multiple conditions using the `elseif` keyword e.g.:

```
if 1 == 2 then
    print("1 is equal to 2")
elseif 1 == 1 then
    print("1 is equal to 1")
end
```

`elseif` is similar to else but if also performs a check to see if the condition is also true. We can optionally add an `else` statement to run code if none of the conditions are met e.g.:

```
if 1 == 2 then
    print("1 is equal to 2")
elseif 1 == 1 then
```

```
    print("1 is equal to 1")
else
    print("1 is not equal to 1 or 2")
end
```

Additionally, we can negate a condition using the `not` keyword e.g.:

```
if not 1 == 2 then
    print("1 is not equal to 2")
end
```

And can check two conditions using the `and` keyword e.g.:

```
if 1 == 1 and 2 == 2 then
    print("1 is equal to 1 and 2 is equal to 2")
end
```

Or check if one of two conditions are true using the `or` keyword e.g.:

```
if 1 == 1 or 1 == 2 then
    print("1 is equal to 1 and/or 1 is equal to 2")
end
```

That covers the basics of conditionals.

Loops

Loops are a common form of interaction in programming. They allow us to run a piece of code multiple times.

Lua supports multiple forms of iteration.: - `while` loops - `repeat` loops - `for` loops - recursive functions (calling a function from within itself)

Before we continue, it is important to note that is it trivial to create an infinite loop e.g.

```
while true do
    print("Hello World!")
end
```

This code above will freeze up the game and will require you to terminate the application to stop it. So be careful when using loops.

While Loops

While loops are the simplest form of iteration. They will run a piece of code while a condition is true e.g.:

```
i = 0
while i < 10 do
    print(i)
```

```
    i = i + 1
end
```

They are defined by typing `while` followed by the condition to check and then the do keyword. After this you write your code to repeat and then end the loop with the `end` keyword.

Repeat Loops

Similar to the while loops, are `repeat` loops which will always run at least once and then run while a condition is true e.g.:

```
i = 0
repeat
    print(i)
    i = i + 1
until i >= 10
```

So the difference between these is that while will not run even once if the condition is not met but repeat will always run at least once.

For Loops

`for` loops are a little more complex but the most powerful form of iteration.

To define a for loop, you need to define a variable to iterate over, the start value, the end value and the step value separated by commas and with the `do` keyword like with all other loops e.g.:

```
for i = 0, 10, 1 do
    print(i)
end
```

This code simply creates a variable `i` with the value `0`, while `i` is less than `10` it will run the code inside the loop and then add `1` to `i` after each iteration.

Not Covered

Lua has many more features we will not cover in this document. Some of more useful features are: - Tables - Modules - Coroutines - Order of operations (precedence)

Among many others.