Lua basics

By Kubsy

Hi, and welcome to the basics of Lua! This page will launch straight to the point teaching you basic programming theory to get you started making Lua scripts for TombEngine.

Contents:

If you feel familiar with the material and would like to learn more complex and in-depth stuff about Lua, check out the official documentation: https://www.lua.org/pil/contents.html

If you feel ready to create interesting setups for your TombEngine level, check out the scripting tutorial: https://github.com/MontyTRC89/TombEngine/wiki/Lua-Script-Tutorials

Let's start! 😃

1) Variables and Data Types

Variables

Variables are an essential feature of every scripting language. They serve as containers that store specified information, be it text, numbers, or userdata, and allow you to manipulate their contents in many scenarios.

In Lua, you can define **local** and **global** variables:

- **Local** variables exist **only within** the code block in which they are defined.
- **Global** variables exist **script-wide** and can be accessed from anywhere.

(Quick aside: the following is an example of a **code block**. Instructions belonging to the block are typically **inset with a tab** for ease of readability. A key detail to remember: in Lua, **blocks are always closed with the keyword** `end` **at the bottom!**)

```
```

```
i = 0
while (i < 10) do
  local name = "TombEngine"
  print(name)
  i = i + 1
end
```

In the above example, the local variable `name`, defining the string of text `"TombEngine"`, exists **only within the block** of this `while` loop (loops will be explained later). This means it cannot be used anywhere **except** inside of it. **Every local variable must have the `local` keyword before the declaration of its name.**

Meanwhile, the global variable `i`, defining the integer `0`, exists in the **across the script**. This means it can be used **anywhere in the script**, inside and outside any block. **A variable without a keyword before the declaration of its name is global by default**.

Data Types

In Lua, there are 7 types of data that you may need to use during scripting:

1. Strings

- Strings are: characters, words, phrases or sentences which you can store, this is particularly useful if you want to have a dialogue and you want to store text dialogue in a variable.
- to declare a string you can either put double quotes **""** or single quotes **''**
  - `local LaraCurious = "Huh, what is this?"`
  - `local WernerAngry = 'Lara, what are you doing!!'`

2. Integers

- Integers are **whole numbers** which can be negative or non-negative (positive) again, useful for certain scenarios such as storing: health or timer, but also very useful to remove magic numbers (more on them later)
- To declare an integer, just declare a variable and initialise with a number:
  - `local health = 100`
  - `local negativeNumber = -6`

3. Floating-point number (decimals)

- Floating-point numbers are **decimal point numbers**, which also can be positive and negative. Same as integers, they can be used for many scenarios.
- To declare a floating-point number, do the same as declaring and initialising with a integer:
  - `local Rotationx = 23.9`
  - `local negativeRotationy = -55.5`

4. Booleans

- Boolean data type can have only 2 values: true or false. They are used to set a variable to be either true or false and use it in a particular scenario.

- example:

```
isLaraKilled = false
if isLaraKilled == true
   print("What have you done!!!???")
end
```

5. Tables

- Tables are an array or a hash tables (dictionaries), and an array is a list of values. Tables are quite complex and may not be needed but are still worth an explanation will be explained in Tables (Arrays) section

6. Userdata

- Userdata is a very essential data type and will be very common during TombEngine Lua scripting. Userdata are **anything that is not built-in lua type** hence in TombEngine, you will have functions and methods which are: `GetMoveableByName` or `Color.new` (explanation on these are in the Docs)
- Userdata can be used like this: `local raptor = TEN.Objects.GetMoveableByName("raptor1")` or `Color.new(255, 255, 255)`

  (pssst, you don't have to write `TEN.Objects.` if you write this in your Lua file:

  ```
  local Util = require("Util")
  Util.ShortenTENCalls()
  ```

  `Util` variable will store another Lua file which is "Util.lua" and gets its functions (**that file must be present in your script folder!**) which will then call the `ShortenTENcalls()` function so you can save your fingers and don't have to type `TEN.Objects.` 😃)

7. Functions

- Functions are another data type which will be required if you want to use volume triggers within Tomb Editor.
- Functions are blocks of code which can be reused as many times as you like in the script and in as many volume triggers as you like. More information on functions in the "Creating function for Volume triggers" and "Creating functions inside the script" sections.

2) Mathematical Operations

In programming, you can do arithmetic operators to Integers and decimal-point numbers in order to do some calculations. In Lua, there are 7 types of Arithmetic. Before explaining these Operators, let's define some variables:

```
```

```
local a = 10
local b = 5
```

now we can use these 2 variables to perform arithmetic:

1. Addition

- You can add variables as long as they have numbers:

```
local c = a + b
print(c)

Output: 15
```

- Explanation: since we declared and initialised 2 variables already (a = 10 and b = 5) we declared another variable (c) which will add these 2 variables together.

2. Subtraction

- You can subtract variables as long as they have numbers:

```
local c = a - b
print(c)

Output: 5
```

- Explanation: This will subtract 2 variables to give you c = 5

3. Multiplication

- You can multiply variables as long as they have numbers:
  - note: in programming, in order to do multiplication, you have to put an asterisk * not like in normal maths you put "x"

```
local c = a * b
print(c)

Output: 50
```

  - Explanation This will multiply 2 variables to give you c = 50

4. Division

- You can divide variables as long as they have numbers:

- Note: in programming, you need to put `/` to divide.

```
local c = a / b
print(c)

Output: 2
```

- Explanation: Here we have divided 2 variables to give you c = 2

5. Modulus

- A modulus (or a modulo) is a special operator which acts like division **but** it will return the **remainder**.
  - A modulus is performed with a `%` sign

```
local c = a % b
print(c)

Output: 0
```

  - Explanation: since 10 divided by 5 is 2 then the remainder is 0 because it doesn't give you a "leftover" number and 10 / 5 can be exactly divided which won't give you a fractional/decimal number
  - if you were to put:

```
local a = 10
local b = 6
local c = a % b
print(c)

Output: 4
```

  - then the remainder is 4 as 10 / 6 cannot be exactly divided and it's a decimal number.

6. Exponent (power operator)

- An exponent is an operator which will multiply the number by itself
  - An exponent is performed by: `a^b` where a = number and b = number of times to multiply itself by

```
local c = a^2
print(c)

Output: 100
```

- Explanation: since a = 10 then 10 to the power of 2 is 100 (because 10 x 10 = 100)

7. Unary (negation)

- Unary is a special operator which will convert the positive number to a negative
  - Unary is performed by putting `-` to a variable

```
local c = -a
print(c)

Output: -10
```

  - explanation: we negated the `a` variable so now it became -10 (think of it as -1 x 10 or -(10) = -10)

All of these operators come in handy if you want to perform some sort of calculations in-game for example you can perform a tricky calculation to move a static or rotate it or even to check distances precisely and etc.

3) Decision-Making

In programming, Decision-making implies making choices and evaluating if one condition is true or false, if either one of them is true then it will take that route however if it is false then it will take an alternative route.

in Lua there are 3 decision-making statements:

- `if` statements
  - an if statement will check if the condition is true and if it is then it executes the code
  -

```
local a = 2
local b = 1
local c = a + b

if c == 3 then print("c is 3") end

OR

if c == 3 then
   print("c is 3")
end
```

  - In the above example, the if statement determine if c = 3 if it is then it prints that c contains 3 value. as you can see, you can make an inline if or a block if statement.
- `elseif` statement
  - elseif statement will trigger if the if statement appears to be false, you can add as many elseifs as you like

```
local a = 2
local b = 2
local c = a + b

if c == 3 then
   print("c is 3")
elseif c == 4 then
   print("c is 4")
end
```

- `else` statement
  - else statement is used if neither if and elseif statement is true

```
local a = 1
local b = 1
local c = a + b

if c == 3 then
   print("c is 3")
elseif c == 4 then
   print("c is 4")
else
   print("c is neither a 3 or a 4"
end
```

4) Relational and Logical Operators

Relational and Logical operators are the types of operators which checks for comparison for both variables given the operator. They are commonly used within loops and if statements so let's go have a look at them:

Relational operators

In Lua (and in any other programming language) you have 6 relational operators:

1. Equal `==`

- The equal sign will check if 2 variables are equal to each other

```
local a = 10
local b = 10

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

2. does not equal to `~=`

- The does not equal to if the 2 variables don't equal each other

```
local a = 23
local b = 9

if a ~= b then
   print("a does not equal to b")
else
   print("a is equal to b")
end
```

3. greater than `>`

- greater than `>` will check if the left-hand side is equal to right-hand side.

```
local a = 23
local b = 9

if a > b then print("a is greater than b") end
```

4. less than `<`

- less than `<` is the same as greater than `>` but checks if left-hand side is less than the right-hand side.

```
local a = 23
local b = 9

if b < a then print("b is less than a") end
```

5. greater than or equal to `>=`

- greater than or equal to `>=` checks if the left-hand side is greater or equal to left-hand side.

```
local a = 23
local b = 22

if a >= b then print("a is either greater than or equal to b") end
```

6. less than or equal to `<=`

- less than or equal to `<=` checks if the left-hand side is less or equal to left-hand side.

```

```

```
local a = 23
local b = 22

if b <= a then print("b is either less than or equal to a") end
```

logical operators

Again, in Lua (and in any other programming language) you have 3 types of logical operators:

1. `AND` operator

- `AND` operator will check if both or more expressions are `true`

```
local score = 60
local health = 40

if score >= 60 and health >= 50 then
    print("score and health both are above minimum, well done!")
else
    print("Score or health is not above the minimum, game over!")
```

2. `OR` operator

- `OR` operator will evaluate if either one of the expressions (if there are 2 or more expressions) is true

```
local healthLeft = 25
local game_over = false

if healthLeft <= 25 or game_over == true then
   print("Game over!")
else
   print("Game is not over yet!")
end
```

3. `NOT` operator

- `NOT` operator will evaluate if the expression is not true

```
local isLaraDead = true

if not isLaraDead then
   print("Lara is still alive")
else
   print("She ded!")
end
```

5) Loops

Loops are another important programming concept used by all programming languages. Loops execute statements over and over until a certain condition has been reached or it has been set to true. Loops are useful to repeat the same stuff over again for example loop the script until Lara has picked up the artefact or killed an enemy.

In Lua there are 4 types of loops which may be used:

1. For loops

- For loops are types of loops which will loop the statements until something has been reached or something is completed.
- The syntax for For loops is:

```
for init,max/min value, n
do
statement(s)
end
```

  - Where:
    - init = Variable initialisation
    - max/min value = minimum or maximum value to loop
    - n = increments the loop by n number
- Example:

```
for i = 0, 5, 1 do
  if i == 5 then
      print("This is the final step!")
  else
      print("We are at step " .. i)
  end
end

 output:
 - We are at step 1
 - We are at step 2
 - We are at step 3
 - We are at step 4
 - This is the final step!
```

2. While loop

- A while loop is another type of loop which will loop endlessly while a **condition is true**. The while loop checks for the condition at the **top** of the block

- the syntax for a while loop is:

```
while(condition)
do
statement(s)
end
```

- Where the condition is the condition you specify for the while loop to test if it is true or not.
- Example:

```
local name = "Hi, I'm Lara Croft!"
local i = 10

while(i ~= 20) do
   print(name)
   i = i + 1
end

output:
- Hi, I'm Lara Croft!
- Hi, I'm Lara Croft!
- Hi, I'm Lara Croft!
- Hi, I'm Lara Croft!
- Hi, I'm Lara Croft!
- Hi, I'm Lara Croft!
- Hi, I'm Lara Croft!
- Hi, I'm Lara Croft!
- Hi, I'm Lara Croft!
- Hi, I'm Lara Croft!
```

- With a while loop, you can do an infinite loop but be careful as doing an infinite loop not correctly may perform some unintended behaviour (for example infinitely spawn enemies) and the engine may not handle this and the game will crash.

```
while(true) do
  print("Aaaaaaaa")
end
```

3. Repeat ... until loop

- Repeat ... until the loop is another loop which is similar to a while loop however, the repeat loop will check if the condition is true at the **bottom of the loop**
- this means that in a while loop if the condition is true initially, then the while loop will not execute. However, if the repeat until the loop is true initially, it will still run the statements but then it will stop.

- the syntax for a repeat until the loop is:

```
repeat
statement(s)
until( condition )
```

- Example:

```
local someName = "Von Croy"
local i = 0
repeat
  print(someName)
  i = i + 1
until(i == 10)

output:
- "Von Croy"
- "Von Croy"
- "Von Croy"
- "Von Croy"
- "Von Croy"
- "Von Croy"
- "Von Croy"
- "Von Croy"
- "Von Croy"
- "Von Croy"
```

- there is a keyword that may also be useful which is a `break` statement which will terminate the loop.
    - break statement can be used to break the loop if the loop has reached halfway to executing, for example, if you set a variable to i = 0 and put an if statement to check if the number has reached 5, then it will execute the break statement to terminate the loop.

4. Pairs and Ipairs

- `Pairs()` and `ipairs()` are functions which are mostly used for loops, they can be used for stuff that don't have an ending point and will print out key and values for both of them. This is similar to `enumerate()` in python
- ipairs() is used to iterate through an array:

```
local namesArray = {"Lara", "VonCroy", "Seth"}

for index, value in ipairs(namesArray) do
    print("name " .. value .. " has value " .. index)
end

output:
```

```
index Lara has value 1
index VonCroy has value 2
index Seth has value 3
```

- pairs() are similar to ipairs() but they are used with the dictionary element's key and value:

```
local inventory = {
["small medipack"] = 4,
["Big Medipack"] = 2,
["HK ammo"] = 100,
}

print("You have:")

for itemName, itemValue in pairs(inventory) do
    print(itemValue, itemName)
end

Output:
You have:
2       Big Medipack
4       small medipack
100     HK ammo
```

6) Tables (Arrays)

- In Programming, **arrays** are a list of values which contain multiple values with the same data type. In Lua, they are called **tables**. An index in an array is the position where the value is in for example:

```
local names = {"Lara", "Zip", "Winston"}
print(names[2])
Output: Zip
```

- in the above example, we have initialised a table (array) with 3 values, they must be enclosed with curly brace **{}**. To get a value from the table, you have to put the number in the square brackets with the table name (so in the above example, it is **names[2]**)
    - In Lua, the index will **always start at 1**. In most of the other popular programming languages, the index would start at 0.
    - to retrieve the length of the table, you can put a # operator in front of the table name `print(#names) -- 3`
- Tables could be a very useful feature to store multiple enemies in the same table in order to manipulate them at the same time such as: destroying them, modifying their health for each of them etc simultaneously.

- There are very essential methods which you can use for table manipulation which can be found here: https://www.tutorialspoint.com/lua/lua_tables.htm however these 2 will be very essential for table manipulation:
    - **table.insert (table, pos, value)**
        - this will insert the value given the table name, position value (for example inserting the value at position 7) and the value
        - `table.insert(names, 4, "Natla")` - this will insert the name "Natla" at position 4 in `names` table
        - Or for inserting, you can use a `#` operator

```
myTab[#names + 1] = "Larson"
print(#names, names[4]) output: Larson
```

        - Be careful with this operator though as leaving a gap will give you incorrect length of the array:

```
local myTab = {"Lara", "Zip", "Winston"}
myTab[5] = "Pierre"
print(#myTab) output: 3
```

    - **table.remove(table, pos)**
        - This simply removes the value given its position
        - `table.remove(names, 2)` - this removes the value "Zip" from `names`table

7) Lua's Mathematics Library

In Lua, you have a mathematics library which lets you do even more complex mathematics, similar to Scientific or Engineering mathematics. There is a full list of mathematical methods are explained in here: https://www.tutorialspoint.com/lua/lua_math_library.htm

note:

- 1 radian = 57,3 degrees approx, 2π radians = 360 degrees, π radians = 180 degrees and so on...
- general formula: **to convert radians to degrees: Radians x (180/π) and from degrees to radians: Degrees × (π/180)**
- radians should be rounded to 3 significant figures
    - 0.364621 radians = 0.365 radians

Here's a list of some useful math methods you might use:

- `math.pi` value of pi (π) to be used in math functions
- `math.rad(x)` value of x converted to radians given in degrees
    - `math.rad(90)` - 1.57 rad (π/2 radians)

- `math.sin(x)` returns the sine of x (x in **radians**)
    - `math.sin(math.pi) output: 0 (this is because the sine graph cuts the x-axis at π hence 0`
    - `math.sin(math.pi/3) output: 1/2 (exact value)`
    - `math.sin(29) output: -0.664 (to 3 significant figures)`
- `math.cos(x)` returns the cosine of x (x in **radians**)
    - `math.cos(math.pi) output: -1 (minimum amplitude of cosine graph at π hence -1)`
    - `math.cos(15) output: -0.760 (3 significant figures)`
- `math.tan(x)` returns the tan of x (x in **radians**)
    - `math.tan(math.pi/2) output: undefined (the tangent graph is undefined at math.pi/2 (90 degrees)`
    - `math.tan(math.pi/4) output: 1`
    - `math.tan(70) output: 1.22 (3 significant figures)`
- `math.random (m, n)` randomises the number given the m and n **where m = minimum value and n - maximum value**
    - `math.random(1,7) output: returns a random number which is between 1 and 7`
- Again, there are many functions in the maths library such as: `math.log(x)` - natural logarithm of x `math.asin(x)` - arc sine of x or `math.atan (x)` the arc tangent of x however these are very difficult to explain and I will leave it to you to do some research about them.

8) Creating Functions for Volume Triggers

- In the earlier section (Variables and data types) we learned that functions are blocks of code that can be used as often as you like and in many volume triggers.
- This section will show you have to create functions for volume triggers.

Creating Functions

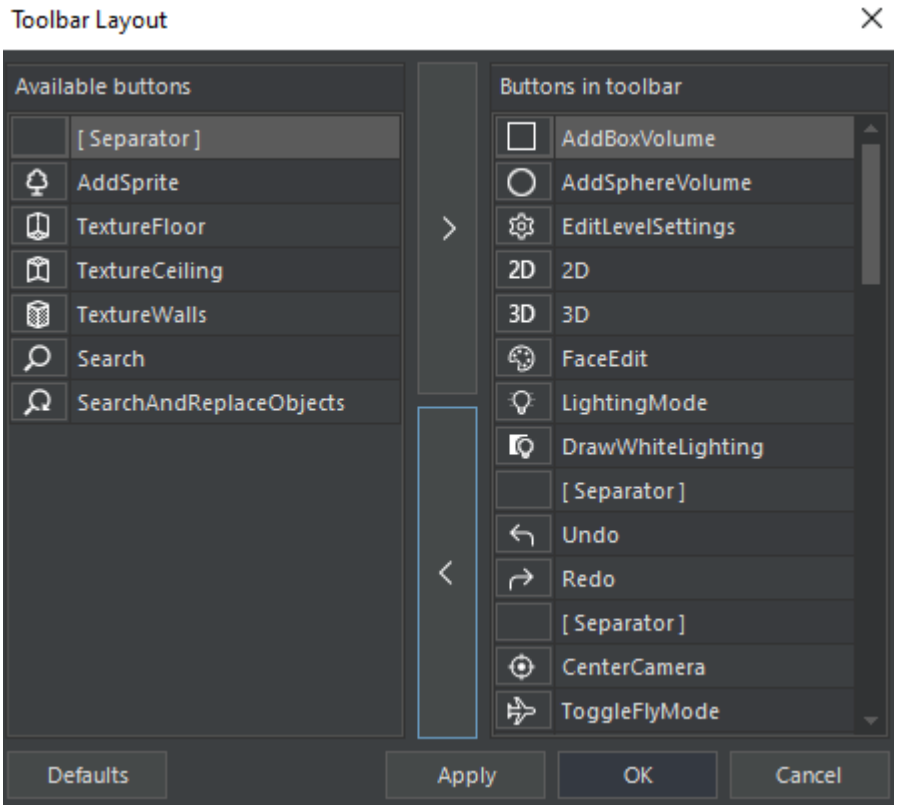
- In order to make functions, we need to make use of a method called `LevelFuncs` which will allow you to put the function in the volume trigger so make sure to have it or it will not appear! We can then call our function then make it equal to a `function()` and then you can make statements to execute when the function is called:
    - ```
      LevelFuncs.EnemySpawner = function()
         -- statements go here
      end
      ```
      ```

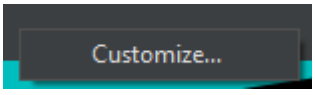
- Congrats, you now have an enemyspawner function and now you can insert it to your volume trigger

Volume Triggers and How to Use Them

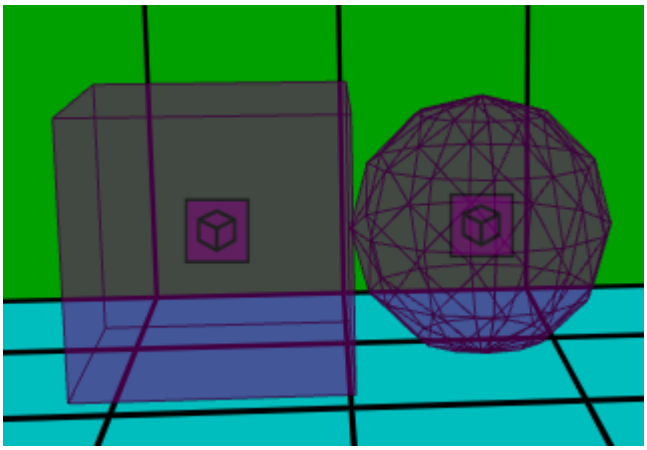
- Volume triggers are a new special trigger in TEN which allow you to put Lua functions inside, you have 2 kinds of triggers: **Sphere** and **Cube** Both can be resized and moved freely and they are not tied to the floor, this is great!
- To insert a volume trigger, you need to have 2 buttons on your top bar shown:

- if you do not then you can right-click on your toolbar -> customize -> move the sphere and cube triggers to the right-hand side of the window and press apply, as shown in the pictures:

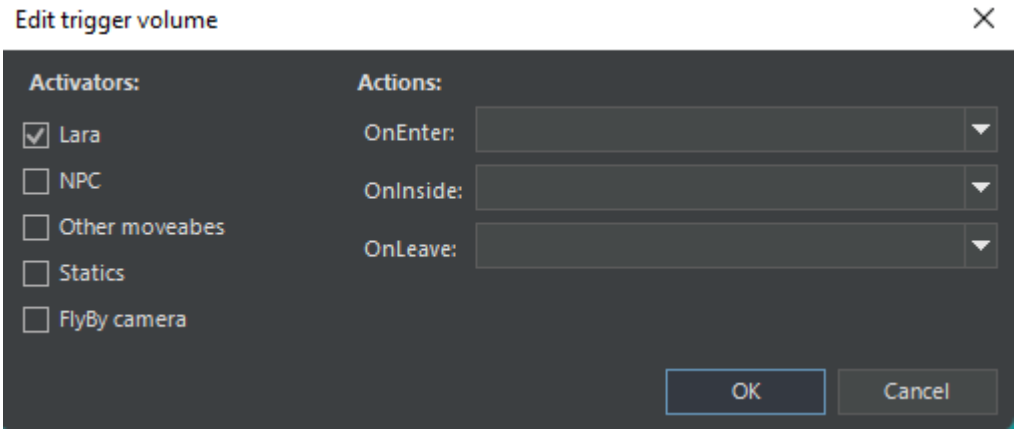




- Great you now can place your volume triggers on the map which will look like this:



- If you double-click on them, you will have a little window shown:



- As you can see, the volume triggers can be activated by anything, not just Lara, so you can have these triggers triggered by an enemy, a camera, moveable or by even a static! On the right-hand side is where you place your function accordingly. You have 3 states:
    - OnEnter - this will trigger the volume once Lara or another object **enters the volume trigger**
    - OnInside - Triggers the volume when Lara or other objects are **inside the volume trigger and it will trigger per game frame**
    - OnLeave - Triggers the volume when Lara or another object **leaves the volume trigger**
- So as you can see with volume triggers, you can do a lot of new and exciting stuff! I will be looking forward to what you create with these 😃

Special TombEngine Fields

- While scripting you will have 5 fields available at your disposal:
    - OnStart() - Calls the function when **the game starts**
    - OnLoad() - Calls the function when **the game is loaded via save**
    - OnSave() - Calls the function when **the game is saved**
    - OnExit() - Calls the function when **the player leaves the game** (includes: finishing a level, exiting to title menu and loading a save in different level)
    - OnControlPhase(delta) - a special field which will call the function per game frame (you can also pass in a delta time as argument, more on that in the next section)

What Is Delta Time and How Do I Use It?

- Imagine you have a 10x10 room and there's a static on the other end of the room and Lara on the another. You decide to create a function which will move the static across the room with a velocity of 2 clicks to the right in OnControlPhase() **per frame**. Everything goes right and it's all good however what if you have a potato pc and your game decides to lag? Well here's the problem the static will teleport instead of moving at a constant velocity whereas on another pc - powerful with constant 30 fps, will act normally. This is the problem and you have to solve it. The way to solve it is by using delta time (dt) for your OnControlPhase()
    - Delta time is the time difference between the last frame and the current time (i.e time difference to render the frame)

9) Creating Functions Inside the Script

- You can also create functions (also called as subprograms) while scripting, remember a function is a block of code which can be executed as many times as you like.
- the syntax for creating functions is as follows:

```
function_body
return parameter_result (separated by comma)
end
```

    - where:
        - Optional function scope - scope of the function, local or global.
        - function name - the name of your function
        - arguments - Arguments to be passed in to process the value (This is optional - you can leave it blank, it depends on what your function will do.)
        - function body - statements which will be done once the function is called and if any arguments have been passed in
        - return - returns the value from the process inside the function separated by commas meaning you can return multiple values (optional)

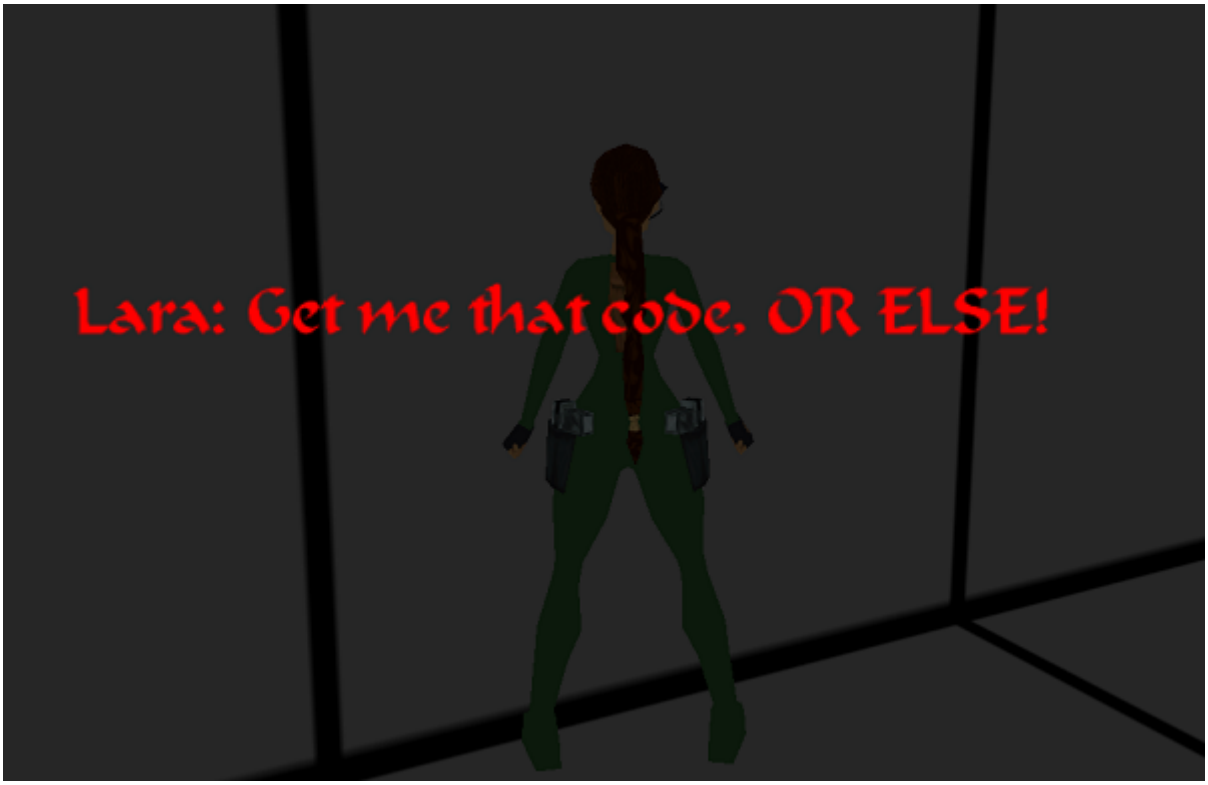
```
local Util = require("Util")
Util.ShortenTENCalls()

function InitSentence()
    local laraChatting = "Lara: Get me that code, OR ELSE!"
    local text = DisplayString(laraChatting, 500, 500, Color.new(255,0,0))
    ShowString(text, 6)
end

LevelFuncs.OnStart = function()
    InitSentence()
end
```

Explanation:

- The `LevelFuncs.Onstart = function()` will be executed once you start the game, it will call the function `InitSentence()`
- Inside the `InitSentence()` function, you have:
    - a variable `laraChatting` to make a string variable where Lara says something
    - another variable `text` which uses `DisplayString()` to customise the string, the text is 500 pixels both along the x and y-axis and colour of red.
    - `ShowString()` is used to actually display the string on screen for 6 seconds.
- This results in:



10) Script Readability

- Programming can be a fun task sometimes, you write code and bam it works as expected. However, there's one thing you need to consider especially if you want to share the code with others, and that is readability. You do not want a really messy code in your script, that is neither good for you nor for others when you share because:
    - 1. Will make debugging difficult
    - 2. will be difficult to read your own code and understand it.
- Here are some good techniques to improve readability in your code:

Magic Numbers

- Consider this script command:
    - `Color(245,134,100)`

- What can you notice about this command? Sure you know that you're defining a new colour which will have those RGB values, ok perfect! Now let's take a look at another example.
- Say you are creating a new object or an enemy:
    - `local newRaptor = Moveable.new(ObjID.RAPTOR, "raptor", Vec3(25674, 34, 12456), Rotation(0,0,0), 100, 0, 0, 10, 0, {0,0,0,0,0,0})`
- Now this is an extreme problem because you are now dealing with random numbers which you do not know what they mean, for example: what does - `{0,0,0,0,0,0}` mean? or a bunch of 0s or a 10? Now you're in panic mode.
- These numbers are called **Magic numbers**. Magic numbers is when you have numbers in your code but they do not give you a clear explanation of what they do, meaning they will hold you back because you need to fully understand what they mean and traverse either through code or the docs to find out if necessary and that will be time-consuming, and it's a bad practice in general.
- One way to fix this is to define variables which will help deal with those pesky magic numbers:

```
local raptorPosition = Vec3.new(25674, 34, 12456)
local raptorRotation = Rotation.new(0,0,0)
local raptorRoomLocation = 100
local raptorAnimNumber = 0
local raptorFrameNumber = 0
local raptorHealth = 10
local raptorOcb = 0
local raptorAINumber = {0,0,0,0,0,0}

local newRaptor = Moveable.new(ObjID.RAPTOR, "raptor", raptorPosition, raptorRotation, raptorRoomLocation, raptorAnimNumber, raptorFrameNumber, raptorHealth, raptorOcb, raptorAINumber)
```

- Now as you can see this is extremely nice and better, you now eliminated magic numbers and you know what each field do and make readability and debugging much easier for you and for other. Congrats 😃

Comments

- You can also comment your code or comment the code out completely to debug a certain issue, it will be ignored by the compiler (or TEN itself).
- To make a single-line comment, you should use a double dash `--` symbol
    - `local number = 12 -- This will comment on the variable line`
- Or if you can multiple lines commented out, you should use `--[[ --]]` symbol
    -
```
--[[
local hi = "yo"
local company = "Copro Dengise"
local name = "Del"
```

```
local holyObject = "Del's calculator"
--]]
```

- As you can see, comments are useful for commenting on what is going on in the code and debugging easily.

Naming Conventions

- Imagine you are reading a person's example script. let's say the example is about moving the static.

```
LevelFuncs.!@MOvESTatIC = function()
    local KiTCHenUTENSils = GetStaticByName(Utensils)
    local tOTHEriGHTSEctor = 1024
    KiTCHenUTENSils:SetPosition(Vec3(0,tOTHErIGHTSEctor,0))
```

    - What can you notice?

- Unfortunately, the variable naming and function naming is not great - you have lowercase and uppercase characters everywhere and also the special characters at the beginning of the function name. This is not great as this decreases readability a lot, especially in a larger script and most importantly: retains consistency.

- To combat this, in programming, we have a term called **Naming Conventions**. Naming Conventions are naming rules for variables, functions, and classes that you, your friend, your corporation, or anyone has set.

- There are several naming conventions but 3 most popular are:

PascalCase

- PascalCase is when the variable or function name has **uppercase** character in every connected word.
    - from above example: `LevelFuncs.MoveStatic = function()`
- I use PascalCase for function naming.

CamelCase

- CamelCase is similar to PascalCase but this time, the starting word character has a **lower-case** character, whilst other words have **upper-case** character.
    - From above example: `local kitchenUtensils = GetStaticByName(KitchenUtensils)`
- I use CamelCase for variable naming.

snake_case

- snake_case is also a popular name convention. Snake_case refers to names instead of being connected together, they are separated with an underscore (_) between them.

    - `LevelFuncs.move_static = function()`

- I don't use it, but you can use it for functions and variables.

- Naming conventions aren't tied to each thing for example you don't have to strictly use PascalCase for functions just because someone has that as well.

Conclusion

- Congratulations! You have reached the end of this tutorial. I hope it has helped you understand Lua and gain more confidence about scripting for TombEngine. We look forward to seeing the creative puzzles, contraptions, and effects you make. 😃

Credits

- ***Kubsy - Lua Basics Tutorial***

Proofreading and Error Checks

- ***JoeyQuint***
- ***RemRem***
- ***Sezz***
- ***Stranger1992***
- ***SquidShire (Hispidence)***