Making an ease script for Davinci Fusion: Part 2

Table of contents

Intro

In the previous article, we discussed fundamental motion graphics concept, and the ins and out of the script that needs to be made. In this section, we will tackle some LUA and Fusion scripting fundamentals.

A primer on LUA

LUA is the scripting language that is used by Davinci Fusion, either that or python, but LUA is built-in, requiring no additional installation or configuration.

LUA is fairly basic in its capabilities and data structures, the standard library is more limited than other scripting languages, but it makes up for it in efficiency.

Types

Lua has all the classic types:

- string
- number
- function
- boolean
- nil (this one represents undefined in javascript)

The `type()` method easily lets you check what is of what type.

Here is an extract from the official documentation.

```
print(type("Hello world"))  --> string
print(type(10.4*3))         --> number
print(type(print))          --> function
print(type(type))           --> function
print(type(true))           --> boolean
print(type(nil))            --> nil
print(type(type(X)))        --> string
```

Variables

Variables are not strongly typed, you can (at your own risk) reassign a value of another type to a variable containing a value of a certain type.

Local variables are defined by appending the `local` prefix to the definition, otherwise it's global. Locality works as you would expect with some basic knowledge of blocks and chunks (just like `let` in javascript). LUA has no "constant" prefix.

`nil` is not `null`, `nil` is inexistence, if you want to destroy a global variable, you assign `nil` to it.

Tables

LUA tables are the be-all and end-all of data structures, there is no other construct. Arrays, matrices, dictionaries, all can be implemented with a table.

An array can be made with the shorthand constructor:

```
array = {1, 4, 9, 16, 25, 36, 49, 64, 81}

print(array[1]) --> 1
```

The table constructor initializes arrays at 1, not 0. Lua recommends indexing at 1

Indices are automatically created for each value, a more customized key-value correspondance is obtained by defining the keys:

```
table = {5,8,2}

print(table[1]) --> 5

table = {test = 5,8,2} -- You can mix indices definitions

print(table[1]) --> 8
print(table["test"]) --> 5
```

Iterating over a table with `pairs` does it in an arbitrary order, ordered iteration requires an array (ordered indices starting at 1) and the use of the `ipairs()` function. More info can be obtained here (you probably should read the entire thing if you want to get serious about lua scripting).

The standard library

LUA's standard library is smaller than average, implementing basic functions and math, but not much more, you'll probably be copy-pasting helpers from online if you want to do more complex scripting.

This much should be enough LUA knowledge to start experimenting with it inside of Davinci Fusion

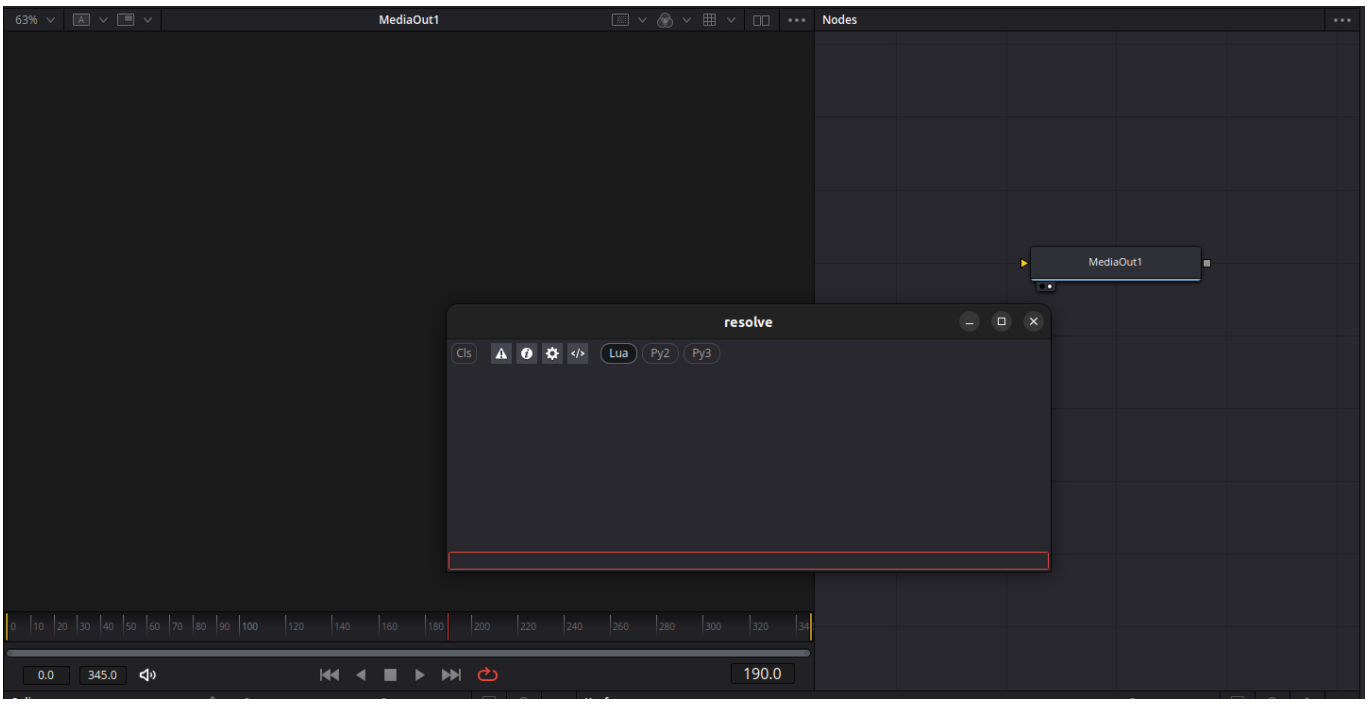
Classes and methods

In LUA, an object instances' method is often invoked using `:` instead of `.`. This is because in OOP, the instance itself should be passed to a method if we want to only affect the instance's lifecycle, `:` does that under the hood (more details here).

```
myObject:callMethod()
myObject.callMethod(myObject) --same thing
```

A primer on Fusion Scripting

Once you're in a new Fusion composition, you can bring up the scripting console with `Workspace > console` from the top menu.



It's configured to use LUA by default.

The scripting manual

Fusion's scripting console does not have any kind of intellisense or autocomplete, there is no VSCode extension either if you want to write extensions externally.

The most important piece of documentation you have is a two hundred pages long scripting manual. This manual teaches the fundamentals of Fusions Object Model and available scripting methods/properties. I had to read through a big portion of it to get my bearings and understand how to navigate around objects and properties.

Printing to the console

Because the Fusion scripting environment has a lot of tables, the fusion console comes prepackaged with a handy helper, the `dump()` function, this can be simplified even more with the `==` shorthand if the command is a single-line expression.

```
dump(comp:GetToolList())
-- table: 0x771a1d281eb8
--  1 = Media Out (0x0x77195ecbf040) [App: 'Resolve' on 127.0.0.1,
UUID:46c45bb2-1243-47b4-8202-de50228e0034]
==comp:GetToolList()
-- table: 0x771a1d2822b8
--  1 = Media Out (0x0x77195ecbf040) [App: 'Resolve' on 127.0.0.1, UUID:
46c45bb2-1243-47b4-8202-de50228e0034]
```

This is useful for debugging (or should I say, this is your **only** debugging tool).

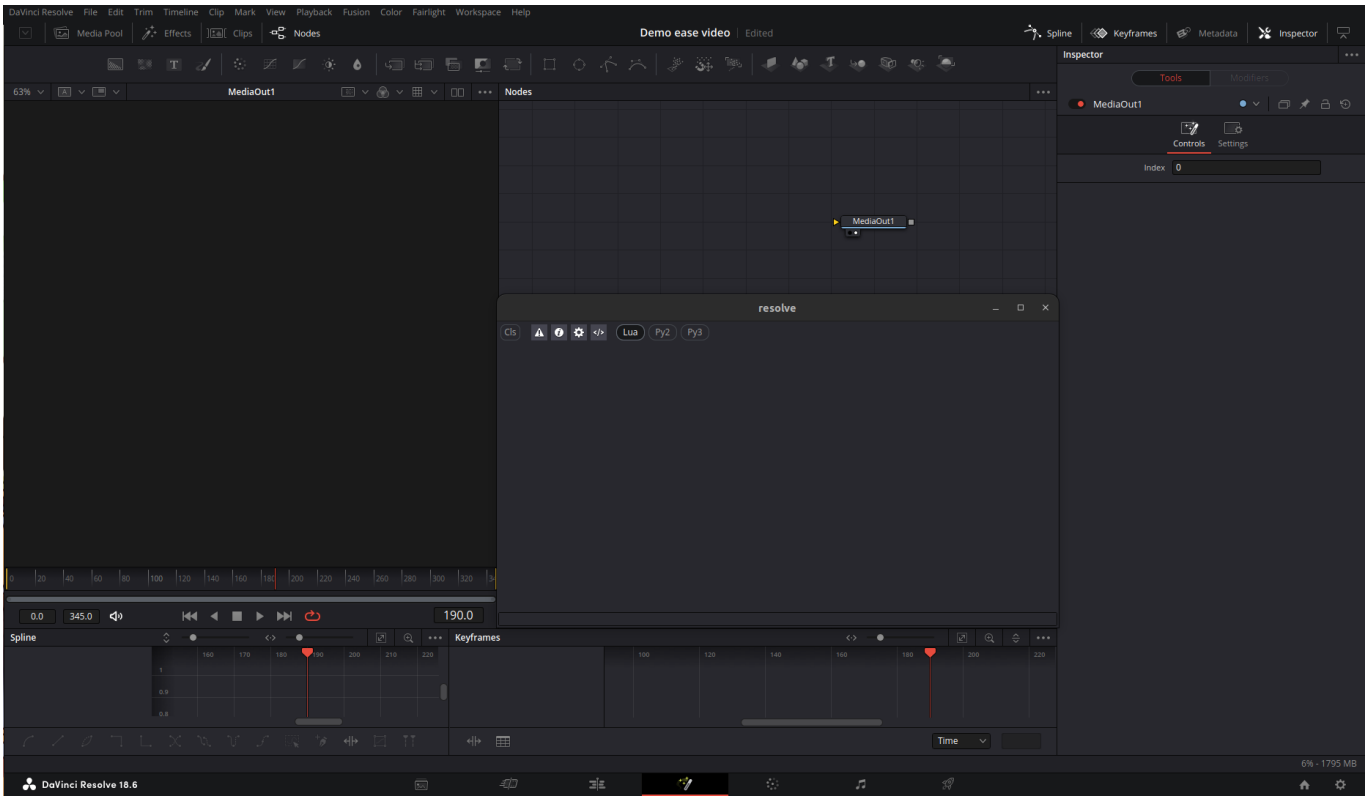
The Fusion Object Model

Knowing how to navigate the Object Model is crucial to scripting in fusion. A few global variables are made available to you from the get-go:

Object	Description
FusionUI	Fusion represents the Fusion application state, accessible via `fusion`
Composition	The current active composition in the script's execution context, accessible via `comp` or `fusion:GetCurrentComp()`
Tool/Operator	Represents a node in fusion's node editor
MainInput/MainOutput	Inputs and outputs that appear as connections between nodes on the Flow
Input	Properties that can appear on a tool's properties view, can be a controlled input or a modifier
Output	An output is the final value of a tool's property

Understanding the object model with a practical example

Let us start with a blank composition.



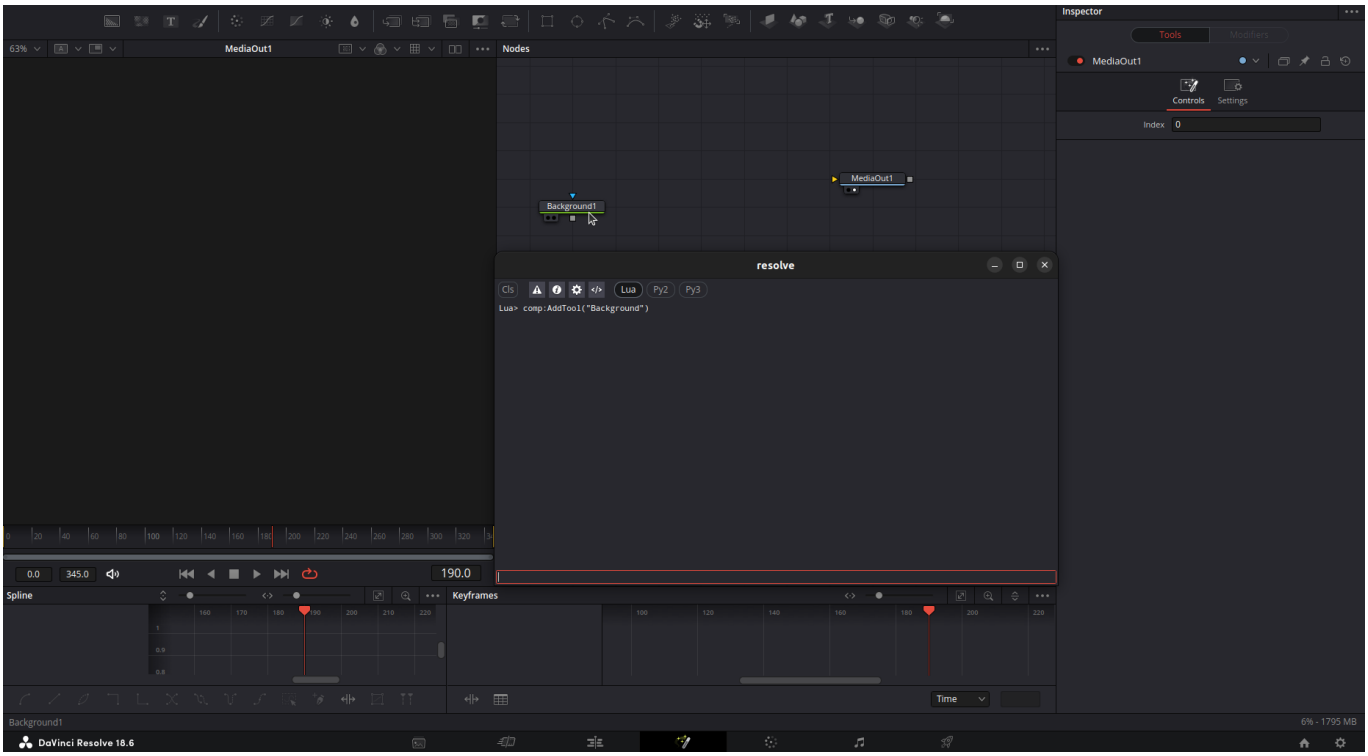
Let's say our goal there is to create a solid background and change its color from blue to green over the course of one second, but we can only do it using the console.

Creating a node

First, create the tool with the `AddTool()` method.

```
comp:AddTool("Background")
```

By convention, you can expect all class methods and attributes to use CamelCase.



Hovering on the newly created node, you can see its name in the bottom left corner, this name can be used to globally access the node.

```
==Background1
-- Background (0x0x77195d739600) [App: 'Resolve' on 127.0.0.1, UUID:
46c45bb2-1243-47b4-8202-de50228e0034]
```

Connecting inputs and outputs

Then, we need to connect our Background1 node to our MediaOut1 node, this operation is done from the input of the receiving node (MediaOut1, that is). It is not our input that is connecting to the output (like you would usually do with your mouse by dragging from source to target), but rather, our output that is requesting the input.

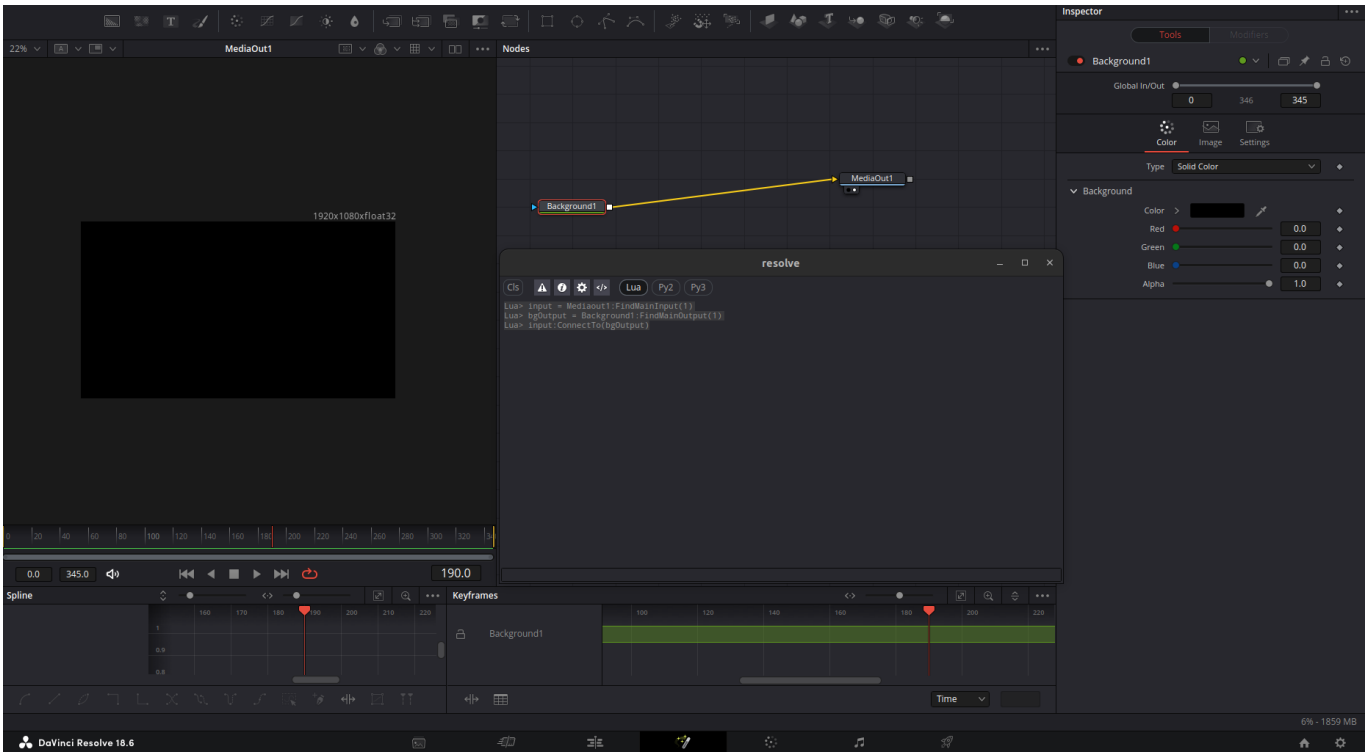
```
input = Mediaout1:FindMainInput(1)
bgOutput = Background1:FindMainOutput(1)
input:ConnectTo(bgOutput)
```

With this, the background is now connected to the output, and displayed as a solid black color in the preview window!

Changing properties (no keyframes)

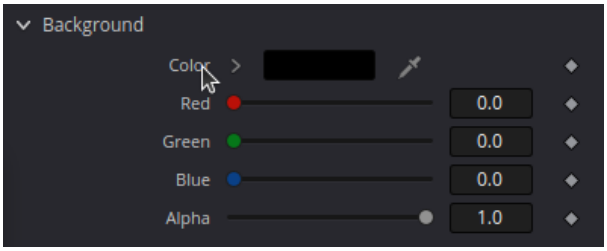
Now to change the color from black to blue.

Click on the background node to show its properties in the inspector.



The same "hover" trick can also be used on properties name in order to show the name of input parameters.

If you hover over the "Color" property, you will see that nothing shows up on the bottom left corner.



The reason behind this is that "Color" is not a property that actually *exists* on the background node, but rather, it is a user control that is linked to the Red, Green, Blue, and Alpha values just below it.

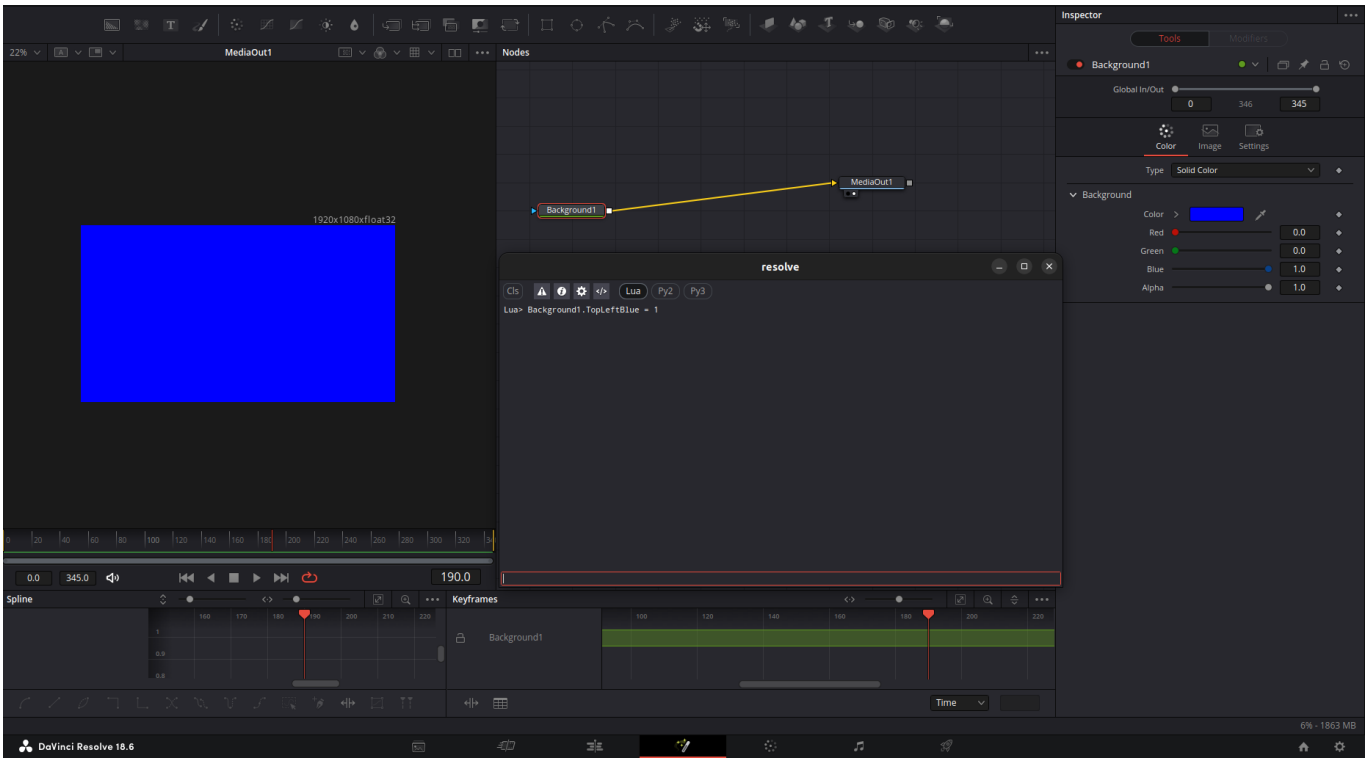
Once you hover over "Blue", you will see its real name show up on the bottom left.

> Background1.TopLeftBlue

Now why the `TopLeft` prefix? That is because background nodes can also be set to be 4-color-gradients, and when they are set to the `Solid Color` type, then it is the top-left corner that is used.

You can make the background blue by setting the new value of the Blue property to `1` (color values are normalized, they don't use the `0-255` range)

```
Background1.TopLeftBlue = 1
```

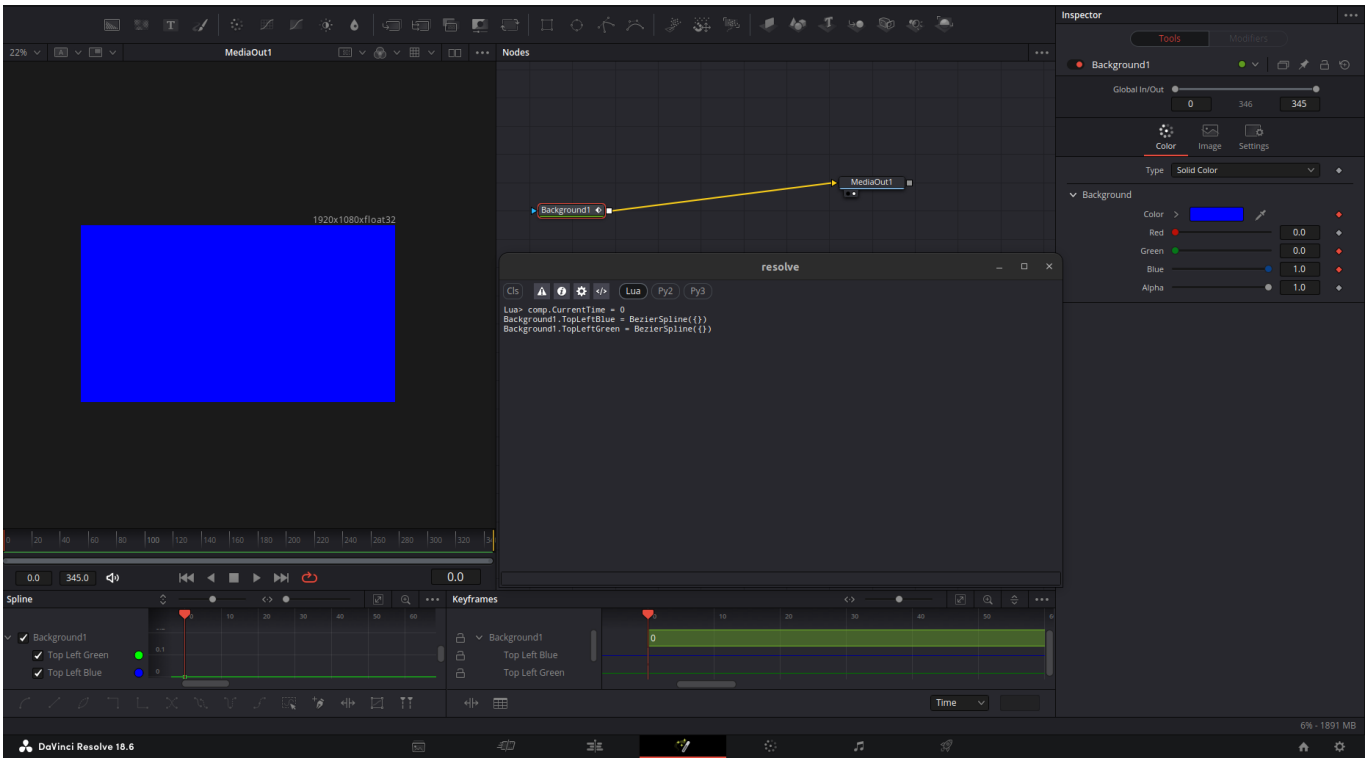


Animating an input

We now need to animate the background from blue to green, that means, animating blue from 1 to 0 and inversely for green.

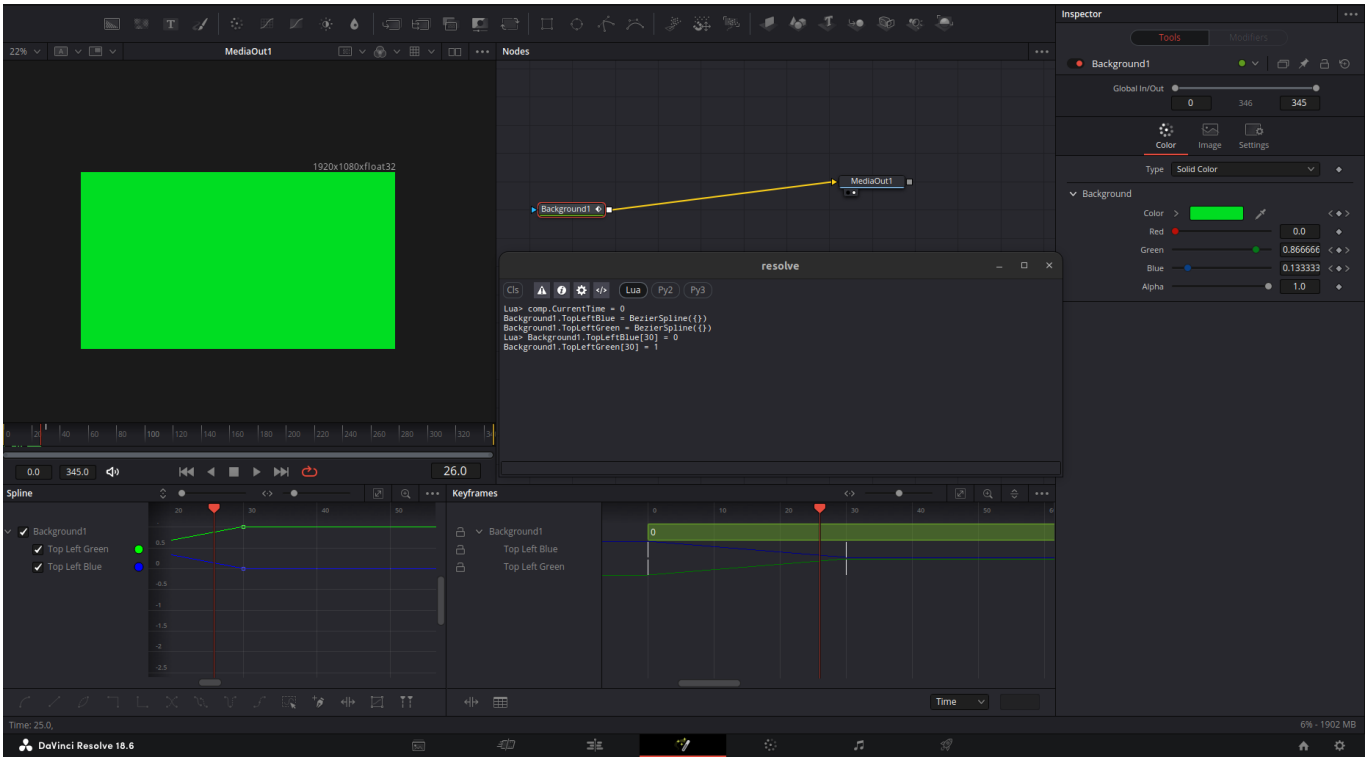
To animate an input, a `BezierSpline` must be passed to the input's value. When you do this, a keyframe is created at the current composition time, so we will se our composition's `CurrentTime` value to the frame `0`. We can then create the `BezierSpline`

```
comp.CurrentTime = 0
Background1.TopLeftBlue = BezierSpline({})
Background1.TopLeftGreen = BezierSpline({})
```



Now, it is simply a matter of changing the values after one second, when a property is animated, you can create a new keyframe by setting a new value at a certain time using the frame as an index of the property, if your composition is 30fps, then frame 30 is at one second in the composition:

```
Background1.TopLeftBlue[30] = 0
Background1.TopLeftGreen[30] = 1
```



And there it is! We have now animated a color change using only scripts.

(as a bonus, you can use `comp:Play()` and `comp:Stop()` to play the animation).

This should give you a general idea of how you can use Fusion scripts for miscellaneous simple tasks.

Putting it all together

This block of text can be copy-pasted in the console to achieve everything we've discussed in this demonstration.

```
comp:AddTool("Background")
input = Mediaout1:FindMainInput(1)
bgOutput = Background1:FindMainOutput(1)
input:ConnectTo(bgOutput)
Background1.TopLeftBlue = 1
comp.CurrentTime = 0
Background1.TopLeftBlue = BezierSpline({})
Background1.TopLeftGreen = BezierSpline({})
Background1.TopLeftBlue[30] = 0
Background1.TopLeftGreen[30] = 1
```

Conclusion

In this article, we tackled the fundamentals of LUA, before learning the basics of Davinci Fusion Scripting. We demonstrated how a basic task can be done via scripting using examples.

In the next section, I will finally break down every bit of code of my ease-copy script into explainable chunks, and go through the logic of the script.