

Making an ease script for Davinci Fusion: Part 3

Table of contents

- [Making an ease script for Davinci Fusion: Part 3](#)
 - [Table of contents](#)
 - [Intro](#)
 - [Making the ease-copy script](#)
 - [Finding adjacent keyframes](#)
 - [Getting a input's keyframes](#)
 - [Finding every eligible input when applying a preset](#)
 - [Saving an ease](#)
 - [Applying an ease](#)
 - [Fusion's UI Manager](#)
 - [Bugs that were fixed after release](#)
 - [Bugs that could not be fixed](#)
 - [Conclusion](#)

Intro

This is the final part of this article series on Davinci Fusion scripting. In the previous part, we had an introductory course on LUA and the fundamentals of Fusion scripting, plus a simple practical example. In This part, We will go over all the functionalities implemented by the ease-copy script, each responding to a specific technical need. This should both help you understand how this specific script work, but also give you a wide breadth of possible solutions to specific, more advanced, technical problems.

Making the ease-copy script

Finding adjacent keyframes

One of the very first problems that has to be solved while making such a script is "how exactly do we query the keyframes we want to save/apply a preset to?". This is when the first, and probably most annoying limitation of the scripting API presents itself:

You cannot query selected keyframes

Because of this, a solution that works like After Effect's Flow extension can immediately be thrown out the window.

We must find another way to query keyframes, one that is both simple and intuitive. Here is the solution that I picked:

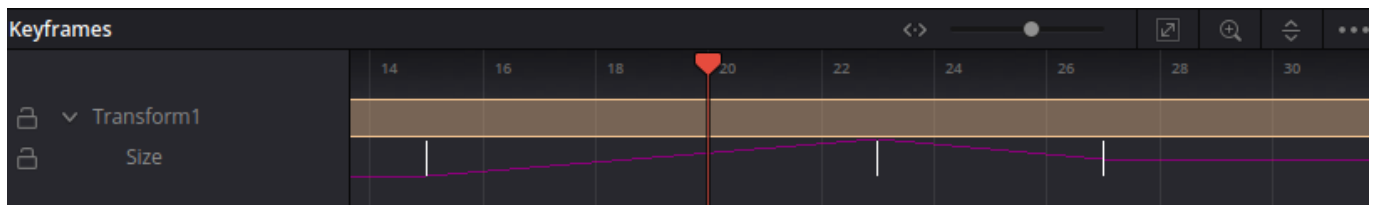
- When applying a preset, the script will act on every input that has a pair of surrounding keyframes;
- When saving a preset, the script will find the first input that has a pair of surrounding keyframes.

We will define "surrounding keyframes" as a pair where:

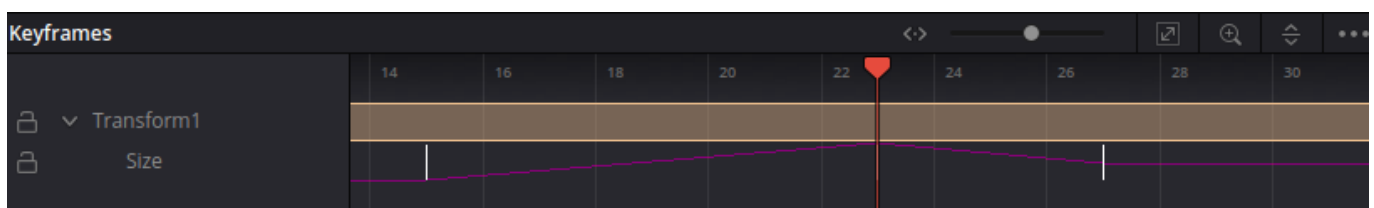
- The first keyframe is less than or equal to the current time;
- The second keyframe is strictly greater than the current time;

As such:

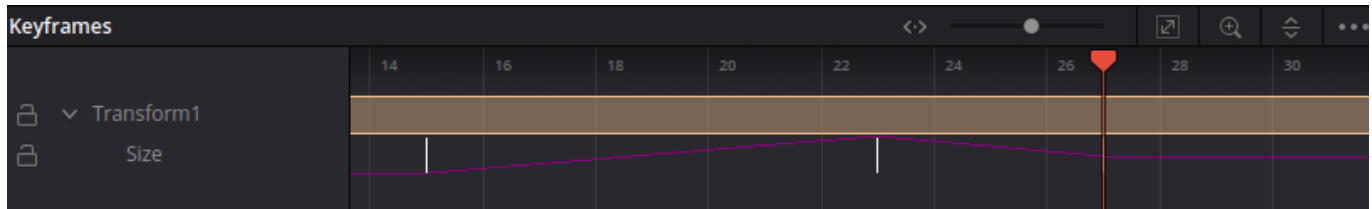
This should query keyframes 1 and 2;



This should query keyframes 2 and 3;



This should query nothing;



Once we have [obtained the keyframes of an input](#), we can find the `CurrentTime`'s adjacent keyframes with the following implementation:

```
function GetAdjacentKeyframes(keyframes)

    local closestLeft = nil
    local closestRight = nil

    for k,v in pairs(keyframes) do
        if k <= currentComp.CurrentTime and (closestLeft == nil or k >
closestLeft) then
            closestLeft = k
        end
        if k > currentComp.CurrentTime and (closestRight == nil or k <
closestRight) then
            closestRight = k
        end
    end

    if (closestLeft and closestRight) then
        return {[closestLeft] = keyframes[closestLeft], [closestRight] =
keyframes[closestRight]}
    end
end
```

We use a loop with no break statement so that the method can deal with unordered tables (ordering is not guaranteed as the indices are not sequential, instead, each index is the frame at which the keyframe is placed). If a valid pair is found, it is then returned.

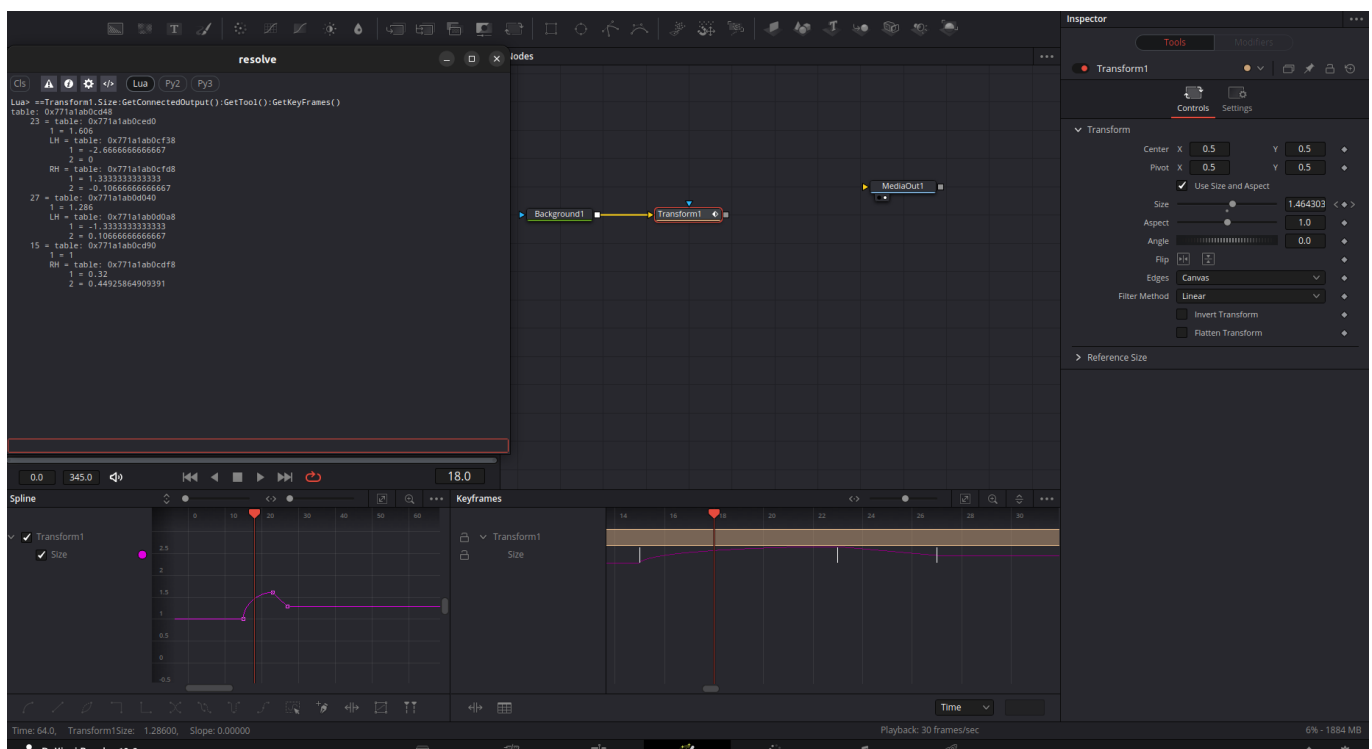
Getting a input's keyframes

Because this script doesn't just work with values, but also easings. We cannot query properties the usual way.

As seen in the Fusion Scripting Primer, for an input to have keyframes, it needs to have a `BezierSpline` as its value. But as it turns out, a `BezierSpline` is internally treated as a modifier. Because modifiers are treated like Tools in the scripting engine, it means that the `BezierSpline` is now a separate entity, with an output that is connected to the property's input. So in order to get the keyframes, we have to do the following:

```
input = Transform1.Size
-- To get the BezierSpline, we must first get its output
splineOutput = input:GetConnectedOutput()
-- From the output, we can then get the tool
splineTool = splineOutput:GetTool()
-- And finally, we can get the keyframes
keyframes = splineTool:GetKeyFrames()

-- As a one-liner
dump(Transform1.Size:GetConnectedOutput():GetTool():GetKeyFrames())
```



Looking at this, you can notice that, as seen in the LUA primer, the indices are arbitrarily ordered.

Because such an operation is very verbose, it can be wrapped in a helper:

```
function GetTool(input)
  local output = input:GetConnectedOutput()
  if (output ~= nil) then
    return output:GetTool()
  end
end
```

We do not call `GetKeyframes` in this method, as the tool itself can have other uses beyond just getting the keyframes.

Finding every eligible input when applying a preset

Let's now imagine that we have an easing preset we want to apply, how are we to figure out which nodes/tools are selected, and which input of these nodes contain animated properties (properties that have keyframes)?

First of all, `comp:GetToolList()` contains an optional argument, passing `comp:GetToolList(true)` will make it return only the nodes that we have selected.

Then, we can iterate over those tools.

```
function EaseCopy(presetName, targetProp, copy)
  currentComp:StartUndo("EaseCopy")
  currentComp:Lock()
  for k,v in pairs(currentComp:GetToolList(true)) do
    local endExecutionEarly = EaseCopyTool(v, presetName, targetProp, copy)
    if (endExecutionEarly) then
      currentComp:Unlock()
      currentComp:EndUndo(true)
      return true
    end
  end
  currentComp:Unlock()
  currentComp:EndUndo(true)
end
```

Tip: you can lock and unlock compositions when doing automated scripting actions in order to avoid unnecessary UI updates.

Next, we need to iterate over all the properties of a tool. To that end, we use `tool:GetInputList()`

```
function EaseCopyTool(tool, presetName, targetProp, copy)
  for k,v in pairs(tool:GetInputList()) do
    local endExecutionEarly = EaseCopyInput(v, presetName, targetProp,
copy)
    if (endExecutionEarly) then
      return true
    end
  end
end
```

Once we are iterating over inputs, we need to find one which input is eligible, as in, which input contains keyframes.

```
function EaseCopyInput(input, presetName, targetProp, copy)
  if not IsViableInput(input) then return end

  local inputTool = GetTool(input)

  if not inputTool then; return; end

  if (IsModifier(inputTool) and not IsBezierSpline(inputTool)) then
    return EaseCopyTool(inputTool, presetName, targetProp, copy)
  end

  if not IsTargetInput(input, targetProp) then return end

  local keyframes = inputTool:GetKeyFrames()

  if not keyframes then; return; end

  -- ...
  -- Rest omitted
  -- ...

end
```

This method makes use of numerous helpers to keep the code clean:

```
function IsViableInput(input)
  return input:GetAttrs("INPB_Connected") and
input:GetAttrs("INPS_DataType") ~= "LookupTable"
end
```

First, we make sure that the input is a connected input (meaning, a modifier's output is connected to it and it is not just a plain value). We also make sure to ignore Lookup Tables (LUTs), which are also treated like a connected input, but cannot be keyframed (the `CustomTool` node, for example, contains LUT inputs).

Because we want easings applied to every modifier (as modifiers can sometimes also be animated), we need to recursively go over every input that is a modifier, except when the modifier is a `BezierSpline` (because it being a bezier spline means that it is our target to apply the ease to), for that, we use the following helpers.

```
function IsModifier(tool)
    local regModifiers = fusion:GetRegList(fusion.CT_Modifier)
    local toolAttrs = tool:GetAttrs()

    for _,v in pairs(regModifiers) do
        if v:GetAttrs().REGS_ID == toolAttrs.TOOLS_RegID then
            return true
        end
    end
    return false
end

function IsBezierSpline(tool)
    return tool:GetAttrs().TOOLS_RegID == "BezierSpline"
end
```

To know whether a tool is a modifier, we need to make use of the fusion registry: we query every modifier type from the registry, and then check if the current tool is any of such types.

As for checking whether it's a `BezierSpline` or not, this is easily done using just the tool's attributes.

This was not mentioned in the Fusion Scripting primer, but `Attributes` are the metadata of many fusion object, they use a specific syntax that includes the object's type, the attribute type, and the attribute name: ex. `REGS_ID => Registry - String - ID`. Attributes are very handy for these kinds of use cases. Read more about attributes on [page 39 of the scripting manual](#).

When the script is called with a `targetInput`, it means that the user only wants to save/apply a preset targeting one single property.

```
function IsTargetInput(input, targetProp)
    if targetProp == 'ALL' then return true end

    t = Split(targetProp, ':')

    return input:GetTool():GetAttrs().TOOLS_Name == t[1] and
input:GetAttrs().INPS_ID == t[2]
end

function Split (inputstr, sep)
    if sep == nil then
        sep = "%s"
    end
    local t={}
    for str in string.gmatch(inputstr, "([^\"..sep.."]+)") do
        table.insert(t, str)
    end
    return t
end
```

Splitting a string is not part of the LUA standard library, it needs to be implemented manually.

Once all those checks have passed, we know that our input is a valid input that has keyframes, all that remains is [checking if the input has adjacent keyframes](#) and [saving/applying](#) an ease preset.

Saving an ease

Once we have our [adjacent keyframes](#) we can save their easing curves as a new preset.

First, we need to understand how a keyframe is structured:



This is how the keyframe appears in the keyframe table.

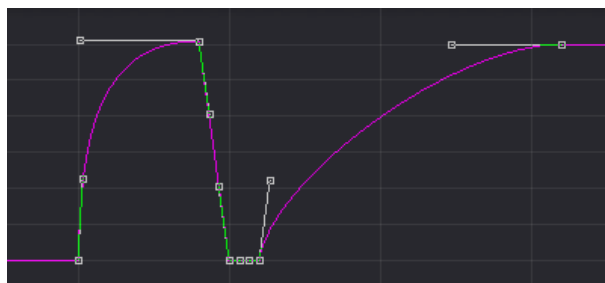
```
==Transform1.Size:GetConnectedOutput():GetTool():GetKeyFrames()
-- table: 0x771a1ab11860
-- 23 = table: 0x771a1ab119a8
--     1 = 1.606
--     LH = table: 0x771a1ab11a10
--         1 = -2.66666666666667
--         2 = 0
--     RH = table: 0x771a1ab11ab0
--         1 = 1.33333333333333
--         2 = -0.106666666666667
-- ... rest omitted
```

The index is the frame at which the keyframe is placed.

The keyframe itself is also a table:

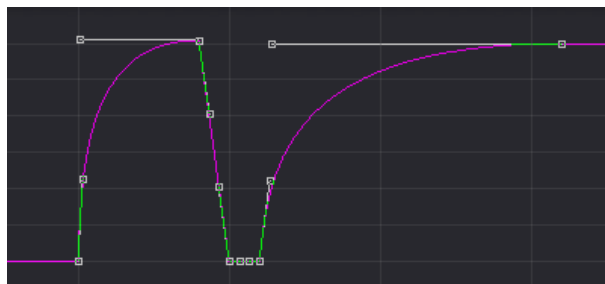
- Its first property represents the keyframe value: a size of 1.6;
- The second and third parameters are the handles, each a table;
 - The first value represents the X offset, in frames, compared to the keyframe;
 - The second value is the Y offset compared to the keyframe too.

Because the handles are dependant on the X and Y units, we cannot simply save them as they currently are. Or the easing curves would not be copied properly:



As you can see here, the handle offsets are the same for the first and second curves, but the resulting curves look wildly different.

This would be a more desirable result:



Because of that, we must normalize the keyframes before saving them:

```
function NormalizeKeyframePairHandles(adjacentKeyframes)
    local tLeft, hLeft, tRight, hRight =
SortAdjacentFrames(adjacentKeyframes)

    local timeDiff = tRight - tLeft
    local valueDiff = hRight[1] - hLeft[1]

    if valueDiff == 0 then; return nil; end

    local RH = hLeft.RH
    local LH = hRight.LH

    return {
        RH = { RH[1] / timeDiff, RH[2] / valueDiff },
        LH = { LH[1] / timeDiff, LH[2] / valueDiff },
    }
end
```

This function transforms the unit-relative handle offsets to values that are normalized between -1 and 1.

Finally, to save and persist a preset, we use the `SetData` to save the data inside of fusion's preferences:

```
function CopyEase(presetName, adjacentKeyframes)
    local normalized = NormalizeKeyframePairHandles(adjacentKeyframes)
    print("copying ease as " .. presetName)
    fusion:SetData("easeCopy.presets." .. presetName, normalized)
end
```

Applying an ease

To paste an easing preset, the process is the same, just in reverse, once we have our [adjacent keyframes](#), we get the normalized handle from fusion's preferences, before denormalizing them, patching the existing keyframes with the new ones, and replacing all keyframes with the patched result.

```
function PasteEase(tool, presetName, adjacentKeyframes, hardReplace)
    local ease = fusion:GetData("easeCopy.presets." .. presetName)
    if ease then
        print("pasting ease preset " .. presetName)
        local denormalized = DenormalizeKeyframePairHandles(adjacentKeyframes,
ease)
        local oldKf = tool:GetKeyFrames()
        local newKf = PatchExistingKeyFrames(oldKf, denormalized)

        tool:SetKeyFrames(newKf, false)
    end
end
```

```
end

function DenormalizeKeyframePairHandles(adjacentKeyframes, normalized)
    local tLeft, hLeft, tRight, hRight =
SortAdjacentFrames(adjacentKeyframes)

    local timeDiff = tRight - tLeft
    local valueDiff = hRight[1] - hLeft[1]

    local RH = normalized.RH
    local LH = normalized.LH

    adjacentKeyframes[tLeft].RH = { RH[1] * timeDiff, RH[2] * valueDiff }
    adjacentKeyframes[tRight].LH = { LH[1] * timeDiff, LH[2] * valueDiff }
    adjacentKeyframes[tLeft].Flags = { RH[1] * timeDiff, RH[2] * valueDiff }
    adjacentKeyframes[tRight].Flags = { LH[1] * timeDiff, LH[2] * valueDiff }

    return adjacentKeyframes
end

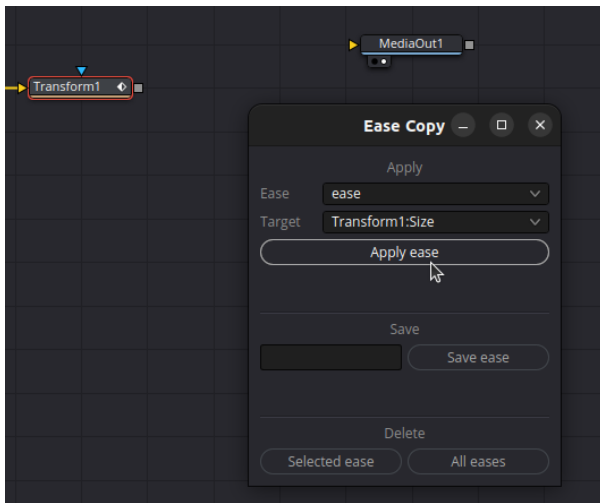
function PatchExistingKeyFrames(keyframes, denormalized)
    local k1, v1 = next(denormalized)
    local k2, v2 = next(denormalized, k1)

    keyframes[k1] = v1
    keyframes[k2] = v2

    return keyframes
end
```

Fusion's UI Manager

The script window looks like this:



Depending on the selected node, different target properties are available for applying and saving eases. Setting the parameter to "ALL" applies to every eligible keyframed property.

The `UIManager` is an utility class that is used to create parametrized user interfaces for scripts.

Because it is based on QT, the scripting documentation contains no information on it, but the following links provide a decent introduction to its functionalities:

- [A quick article by Yanru Mu](#)
- [A thread on the weSuckLess forum, containing a multitude of usage examples](#)
- [The official QT Documentation](#)

The `AddWindow` method is only used to define all the UI components making up the script window:

```
local ui = fu.UIManager
local disp = bmd.UIDispatcher(ui)
local width,height = 275,300
local positionX,positionY = 800,400
local currentComp = fu:GetCurrentComp()

win = disp:AddWindow({
    ID = 'MyWin',
    TargetID = 'MyWin',
    WindowTitle = 'Ease Copy',
    Geometry = {positionX, positionY, width, height},
    Spacing = 0,
    ui:VGroup{
        ID = 'root',
        ui:Label{
            Weight = 0,
            Text = 'Apply',
            Alignment = {AlignHCenter = true},
        },
    },
},
```

```

    ui:HGroup{
        Weight = 0,
        ui:Label    { Weight = 0.5, Text = 'Ease', },
        ui:ComboBox { Weight = 2, ID = 'qEase', Text = '', },
    },
    ui:HGroup{
        Weight = 0,
        ui:Label    { Weight = 0.5, Text = 'Target', },
        ui:ComboBox { Weight = 2, ID = 'qTargetProp', Text = '', },
    },
    ui:Button { Weight=0, ID = 'qApplyBtn', Text = 'Apply ease' },
    -- 30 extra lines omitted
}
})

```

After initializing the `UIManager` and defining parameters like window size and position, widgets are created and placed in the flow, the `ID` properties of each widget are used for event binding.

The following events are implemented:

```

-- EVENT BINDING --

notify = ui:AddNotify('Comp_Activate_Tool')

function win.On.MyWin.Close(ev)
    disp:ExitLoop()
end

function disp.On.Comp_Activate_Tool(ev)
    ReloadTargetComboBox()
end

function win.On.qApplyBtn.Clicked(ev)
    local presetName = itm.qEase.CurrentText
    local targetProp = itm.qTargetProp.CurrentText

    if (presetName ~= '') then
        EaseCopy(presetName, targetProp)
    end
end

function win.On.qSaveBtn.Clicked(ev)
    local presetName = itm.qSaveEaseText.Text
    local targetProp = itm.qTargetProp.CurrentText

    if (presetName ~= '') then
        if EaseCopy(presetName, targetProp, true) then
            itm.qSaveEaseText.Text = ''
            ReloadEaseComboBox(presetName)
        end
    end
end

```

```

end

function win.On.qDeleteOne.Clicked(ev)
    local presetName = itm.qEase.CurrentText
    fusion:SetData("easeCopy.presets." .. presetName, nil)
    ReloadEaseComboBox()
    print('Ease: ' .. presetName .. ' deleted.')
end

function win.On.qDeleteAll.Clicked(ev)
    local confirmClear = currentComp:AskUser("Delete all eases?", {})
    if not confirmClear then return end
    fusion:SetData("easeCopy", nil)
    ReloadEaseComboBox()
    print('All eases have been deleted.')
end

```

With this, all the interactivity of the UI is done, we also added a special event `Comp_Activate_Tool`, so that, every time the user clicks on a new node, the list of available target properties is updated.

Finally, we need functions that populate dynamic dropdown menus (the menus for selecting an ease preset, and the menu for selecting eligible inputs):

```

function ReloadEaseComboBox(newSelected)
    dump('reloading')
    local savedEases = fusion:GetData("easeCopy.presets")
    local presets = {}

    if savedEases then;
        presets = GetKeys(fusion:GetData("easeCopy.presets"));
    end

    itm.qEase:Clear()
    for _, preset in pairs(presets) do
        itm.qEase:AddItem(preset)
    end

    if newSelected ~= '' then
        itm.qEase:SetCurrentText(newSelected)
    end
end

function ReloadTargetComboBox()
    currentComp = fu:GetCurrentComp()

    itm.qTargetProp:Clear()
    itm.qTargetProp:AddItem('ALL')
    for _, target in pairs(FindEligibleInputs(currentComp:GetToolList(true)))

```

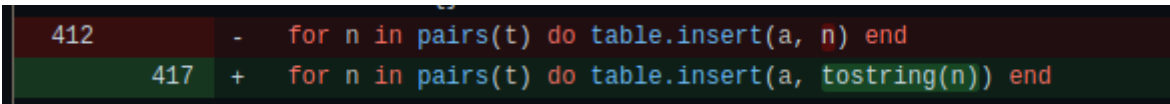
```
do
    itm.qTargetProp.AddItem(target)
end
end
```

With this, the script is ready and working.

Bugs that were fixed after release

Once I uploaded this script, I got notified of a few bugs that I failed to notice in testing due to differences in workflows.

The first bug that I was notified of was an issue where the script would break for no apparent reason, after a bit of troubleshooting, I realized that some people named their presets with just numbers, which broke some internal logic, the fix itself was easy enough:



```
412 - for n in pairs(t) do table.insert(a, n) end
417 + for n in pairs(t) do table.insert(a, tostring(n)) end
```

The second bug was found by people who switch compositions often, each time they did, they needed to close and reopen the script for it to work. This was caused by the script using `comp` to access the current composition, but this value is only set when the script first launches. In order to fix that, I used another variable called `currentComp`, which I updated based on `fusion:GetCurrentComp()`, which always stays accurate.

Bugs that could not be fixed

Some bugs I encountered while doing this script were actually bugs in the scripting API. I was only able to reduce their effect. One such bug was when applying keyframes on two-dimensional properties:

The `PasteEase` implementation shown earlier in this article looked like this:

```
function PasteEase(tool, presetName, adjacentKeyframes, hardReplace)
    local ease = fusion:GetData("easeCopy.presets." .. presetName)
    if ease then
        print("pasting ease preset " .. presetName)
        local denormalized = DenormalizeKeyframePairHandles(adjacentKeyframes,
ease)
        local oldKf = tool:GetKeyFrames()
        local newKf = PatchExistingKeyFrames(oldKf, denormalized)

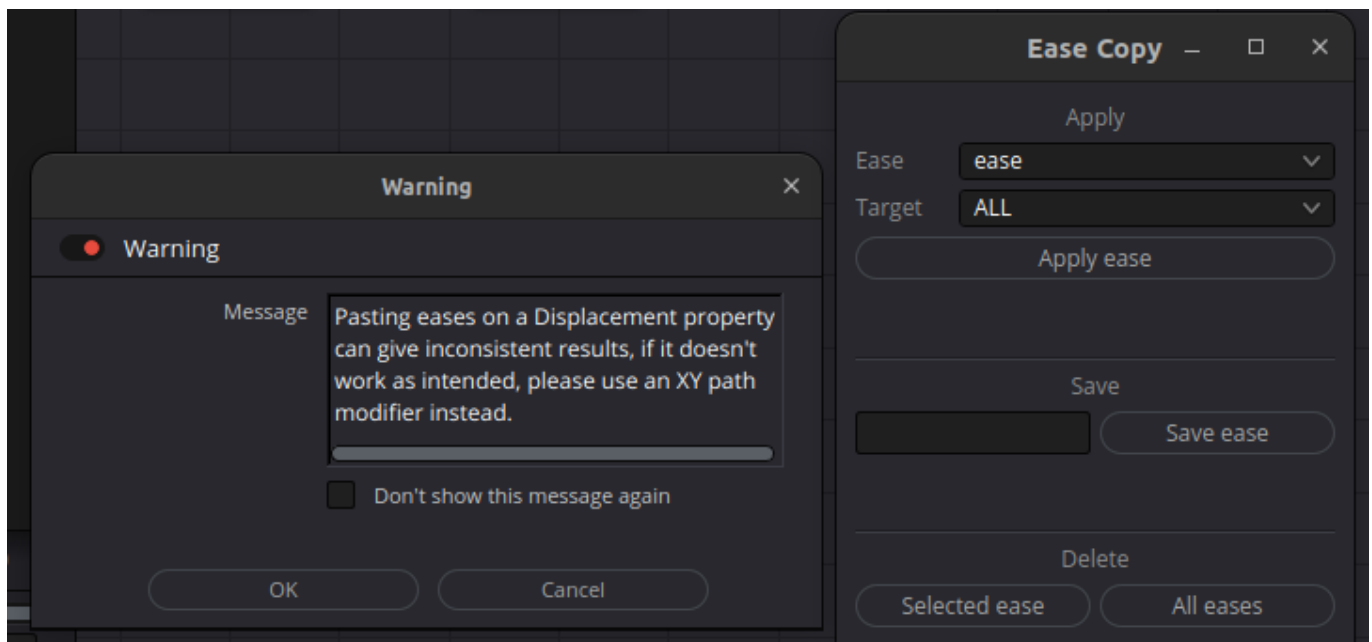
        tool:SetKeyFrames(newKf, false)
    end
end
```

This was a simplified version for the sake of demonstration, the actual implementation looks like this:

```
function PasteEase(tool, presetName, adjacentKeyframes, hardReplace)
    local ease = fusion:GetData("easeCopy.presets." .. presetName)
    if ease then
        print("pasting ease preset " .. presetName)
        local denormalized = DenormalizeKeyframePairHandles(adjacentKeyframes,
ease)
        local oldKf = tool:GetKeyFrames()
        local newKf = PatchExistingKeyFrames(oldKf, denormalized)

        if hardReplace then
            tool:DeleteKeyFrames(currentComp:GetAttrs().COMPN_GlobalStart,
currentComp:GetAttrs().COMPN_GlobalEnd)
            tool:SetKeyFrames(newKf, false)
        else
            ShowDisplacementWarning()
            tool:SetKeyFrames(newKf, false)
            -- This is not a mistake, for some reason, running this twice on
Displacement properties
            -- gives better (though still inconsistent) results
            tool:SetKeyFrames(newKf, false)
        end
    end
end
```

An additional precaution that was taken was to show the user a warning when such a situation was encountered:



Conclusion

In this article we tackled every relevant aspect of the ease-copy script and explained the logic behind them, as well as the implementation choices. Hopefully this would have given you a better idea of more complex use case scenarios when it comes to Fusion scripting.

This finally conclude our series of articles about scripting in Davinci Fusion. Thanks for reading, [the full script can be found here](#).