Blackmagicdesign

FUSION 8

SCRIPTING GUIDE
AND REFERENCE MANUAL

February 2016

Fusion 8

Fusion 8

Fusion 8

About this Document

This document is divided into two sections: The Scripting Guide and the Scripting Reference. The first section, the Scripting Guide, explains the scripting application programming interface (API) of Fusion called FusionScript. It can be accessed via Lua or the Python programming language. FusionScript can be utilized to automate repetitive or complex tasks, customize the application behavior, extend Fusion's functionality, or exchange data with third-party applications.

This guide contains information on how to get started, the differences of scripting languages, how the API is laid out to represent the application model, and how to deal with it in practice.

The second section, the Scripting Reference, assumes you have an understanding of the scripting concepts and the fundamentals of scripting from the first section. The Scripting Reference describes the common API, its objects, members, arguments and usage.

In order to write custom tools or extend Fusion's core functionality, refer to the C++ SDK or Fuse documentation. For regular customization and Macros, read the corresponding chapter in the Fusion User Guide.

Target Audience

This document is intended for developers, technical directors, and users knowledgeable in programming. It was by no means written to teach programming concepts and does not act as a reference for programming languages. Please refer to the documentation of the respective language as advised in the chapter "Scripting Languages." However, when possible practical examples will be given and complete tutorials show the most common applications of FusionScript.

Requirements

In order to follow this guide, you will need to have a copy of Blackmagic Design Fusion 8 installed.

A few features only available in Fusion 8 Studio are highlighted, while every other sample will work with the regular version of Fusion 8. In order to utilize Python, the C-based version of Python needs to be installed as explained in detail in the chapter Scripting Languages.

The source code of both scripting languages needs to be stored as plain text, which can be written in any non-formatting text processor like Notepad or TextEdit. It is recommended to make use of a dedicated code editor to benefit from syntax highlighting and language-specific features.

Conventions

Important notes will be featured in text boxes like this:

Note

Read the Introduction chapters before continuing with the guide.

Code is introduced in boxes with a monospaced font like this:

```
print(“Hello World from Fusion!”)   -- Writes text to the console
```

Regular text may refer to code statements inline, which is also represented by a monospaced font, e.g., the statement ‘print’ in this sentence:

```
The statement print writes text to the console.
```

Most examples shown in the guide are only excerpts of the full source code and may not be able to work on their own. This helps to make the guide more readable. However, all passages marked as Tutorial will contain full source code.

Most code examples are shown in Lua. Inline statements show the Lua implementation of the particular statement; as with Lua, it is easier to identify properties and methods. In order to not mix up Lua tuples with Python tuples, the generic term collection is used to describe tuples, list, dictionaries, etc.

The code here is written for teaching purposes. Sometimes things that could be refactored into separate methods are written explicitly or in a non-optimized way. Please do not hesitate to add your own talent to the code after the fundamental concepts of the API are known.

For consistency reasons naming convention follows roughly the naming of the API (cameCase) for both Lua and Python. Feel free to adapt to PEP8 or your own convention instead.

Scripting Guide

1

Content

Introduction

What is scripting? Scripting is interpreting the specific programming language—in theory—line by line or in the form of compiled bytecode as opposed to executing precompiled machine code directly. Without going too deep into implementation details, it can be concluded that due to its nature, a complex application like Fusion can act as host and provide access to its functionality through a dedicated scripting API. The scripting environment wraps the underlying API and is less likely to crash the whole application if third-party code is defective. Code can be changed on the fly without restarting the host application. Additionally, a garbage collector does most of the memory management in common scripting environments. All this results in slower evaluation compared to native compiled code, but the performance is still beyond what can be done by a user with the regular graphical user interface. The JustInTime (JIT) flavor of Lua that is utilized in Fusion is especially known to perform almost as fast as native code in many cases.

Ultimately, scripting allows for any programmer to mix the language features and libraries of the scripting language with the functionality of the host application. This allows an integration of third-party data or applications.

Let's examine practical uses of scripting within Fusion by example. Scripting in production may help with:

- Automation: For example, read all media files from a given folder, for each of these, load them into a composition, add a watermark, scale them, and render them to a specific location.
- Repetitive tasks: For example find all savers in a composition and set their state to pass-through.
- Maintaining conventions: For example making sure the paths of the savers always point at a specific location on the server, and follow a specific naming convention.
- Tasks prone to human error: For example, verify that certain settings are set before sending a composition to rendering.
- Extending core features in the application: For example, importing animation data from a third-party application.
- Behavior that needs customization for specific pipeline: For example, override what happens when certain events occur. It may enforce certain tools to show up when a specific tool was created.
- Communication with a third-party application: For example, not only exchange data but also share events. When a specific pipeline tool triggers to create a shot, create the corresponding composition.

These are just examples of common applications. Some scripts may require an interface in order to adapt its behavior to a particular need. This may be a configuration file or information derived from the applications state (maybe the current selected tool in the composition). But in many cases, a graphical user interface with a custom dialog that shows all the possible options for the behavior is needed. The latter will be examined in detail in the Graphical User Interfaces chapter.

In Fusion, the scripting API called FusionScript gives access to the most required functionality from the application. In order to fully utilize FusionScript, a basic understanding of how Fusion works is needed. Once this model is known, it will be easier to travel through the Scripting Reference in order to find a needed functionality.

With FusionScript, almost any aspect of Fusion can be accessed and controlled, whether it be the composition and its tools, rendering, metadata, settings, and attributes or the interface.

As FusionScript is only an abstract API, it allows access via different scripting languages—most notably the Lua Programming Language, which is embedded in Fusion or, if installed separately, the Python Programming Language. Although these languages and their features differ greatly, the FusionScript access from both languages is very similar as it accesses the same API. Differences and limitations are explained in the following chapter.

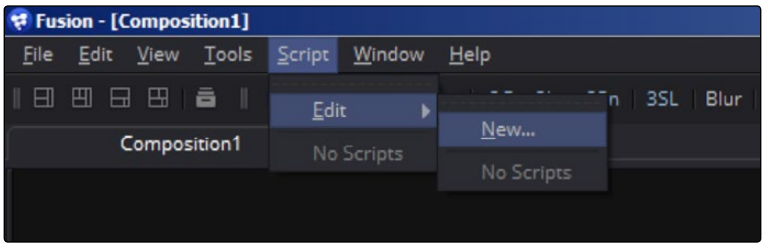
Quick Start Tutorial

Without further ado, let's jump right into a working example.

As proposed earlier, we will create a Lua script that will pass through all but the currently selected Saver. If no Saver is selected, then basically all Savers will be passed through. This is very handy when you have a huge composition but need to prerender only a specific saver.

First Steps

To start, we need to create a new script by accessing the Menu at **Script->Edit->New** …



In the FileDialog, store the script under the name Disable Unselected Savers.lua under the Script folder.

In composition scripts, the filename is used as label to execute the script from the menu. A meaningful name should be chosen.

By default, Fusion will open the default application if nothing else was set in the preferences. You can manually edit the script by invoking **Script->Edit->Disable Unselected Savers**.

In the text processor, write the following line:

```
print(“Hello World from Fusion!”)
```

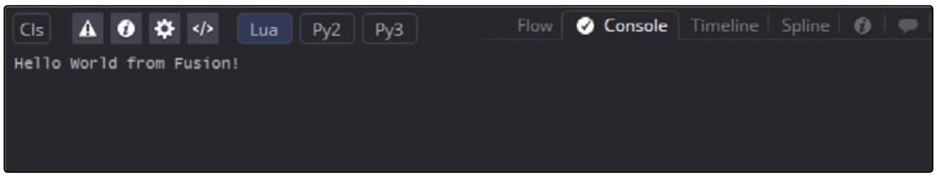
Save the script and execute it with **Script->Disable Unselected Savers.**

Note

Please note that you edit the script with the **Script->Edit->** submenu but execute it directly under **Script->Name of your script**.

All scripts in the composition script folder will be listed here, including subfolders.

Switch to the Console tab in the interface. If everything was set up correctly, the console will show the following text:



```
Hello World from Fusion!
```

All standard output like print will be piped to the console.

The Real Script

Breaking down our intended script in steps the following functionality needs to be implemented:

1. Get and store the current selected tool, if it is a Saver.
2. Iterate through all Savers in the composition.
3. Set these to PassedThrough if they do not match our initial selection.

In Scripts that are executed directly within Fusion two variables are accessible by default:

`fusion` and `composition`. In order to save typing, you can also use the short form `fu` and `comp`.

As their names indicate with `fusion`, you gain access to the applications properties and methods, while `composition` represents anything in the composition.

As all the tasks in this particular script concern the Composition, all required methods are to be found in this object or its members. First of all:

```
comp:GetToolList(bool selected, string type = nil)
```

Returns all tools in the composition, or only the selected ones if the argument is set to true. The type argument is optional. It can be used to filter only specific types of tools.

The tool itself is in fact an object of type Tool or Operator. As you can see, Fusion's application model follows the object-oriented programming concept, which will be examined in detail in the following chapters.

A tool has various properties and methods. But what we are looking for is an Attribute.

Note

Most of the objects in the Scripting API have a base class called Object. Objects may have common properties, one of them being the storage of Attributes. Attributes represent a serializable state of the tool beyond its actual Inputs.

The common attribute to read and write the PassThrough state of a tool is a boolean called **TOOLB_PassThrough**.

Since in this case we will only be setting it, all we need is:

```
tools:SetAttrs( { TOOLB_PassThrough = True } )
```

Note that we pass in a tuple, hence the curly brackets, as we could pass in multiple attributes to be set at once.

With these two commands, we can accomplish all the tasks needed for this script.

Source File: 01 Disable Unselected Savers

```
comp:Lock()
local selectedSavers = comp:GetToolList(true, "Saver")
local allSavers = comp:GetToolList(false, "Saver")
for i, currentSaver in pairs(allSavers) do
    local isSelected = false

    for j, currentSelectedSaver in pairs(selectedSavers) do
    if(currentSaver == currentSelectedSaver) then
```

```
            isSelected = true
        end
        end
```

```
        if isSelected == false then
        currentSaver:SetAttrs( { TOOLB_PassThrough = true } )
        end
end
comp:Unlock()
```

The first and last statement have not been introduced yet.

```
comp:Lock()
comp:Unlock()
```

Whenever the composition needs to change its objects or data, you should Lock the composition, and Unlock it at the end. This guarantees to prevent race conditions, unnecessary redraws but also suppresses Dialogs, e.g., when a Loader or Saver is added to the Flow.

The following two lines simply return a tuple of all selected Saver and all Savers respectively.

```
selectedSavers = comp:GetToolList(true, “Saver”)
allSavers = comp:GetToolList(false, “Saver”)
```

The first loop iterates over all Savers.

The next iteration over each selected Saver compares all the selectedSavers with the currentSaver of the iteration. Since all the selected Savers are also within the collection of allSavers, we can tell for sure if the currentSaver has been selected or not.

If it has not been selected, then we set the currentSaver to PassThrough, which is equivalent to setting the tool to PassThrough in the FlowView.

At the end, we Unlock the composition as mentioned before.

Save the script. Switch to Fusion, create a bunch of Savers. Select few of them and run the script. All but the selected Savers should be set to PassThrough now.

Scripting Languages

Fusion has two scripting Languages to choose from: Lua and Python. Both access the same API through FusionScript so it is up to you which language to choose.

Scripting differs from other APIs available in Fusion. Namely Fuses, Lua scripted plugins that also may contain OpenCL kernels for GPU based evaluation. Fuses allow creation of tools and filters, a feature that was originally only possible through the C++ SDK.

Scripting through FusionScript leaves us with two options:

Lua

The Lua programming language is known for its efficiency, speed, and small memory footprint. Therefore it has been used widely in science and video games.

Fusion ships with Lua 5.1, with some additional libraries build in:

- IUP - for Graphical user interfaces (compare the chapter Graphical User Interfaces)
- bmd.scriptlib - A library with common Fusion related helper functions

Lua is a first class citizen in Fusion as it ships with the install. All preferences and compositions are stored in a Lua table. Fuses are written in Lua and Simple Expressions also consist of a subset of Lua. Additionally, Fusion uses the LuaJIT (JustInTime) flavour of Lua, which outperforms CPython. While in regular scripts this may not matter, it is one reason why Fuses can only be written in Lua.

For a complete reference of the language, see the Lua documentation at: http://www.lua.org/manual/5.1/

Here is the difference of Lua and Python in a nutshell:

- Member properties are accessed with a dot . Methods are invoked with a colon:

For example:

```
print(comp.ActiveTool)

print(comp:GetToolList(true))
```

- Boolean types are lowercase in Lua (`true, false`)
- functions, loops and conditions etc. are closed with an `end` statement.
- Lua only knows one collection type called tuple. It can be used like a Python tuple, list, or dictionary.
- Fusion has a function buildin called `dump()` which can be seen as an extension to `print()`. It formats the output of tables to be more readable. In the console you may also start the line with == as short from for dump, e.g., `==comp:GetAttrs()`

Libraries

One of Lua's benefits is its light weight; Lua does not come with a big standard library. Instead, libraries and Lua files can be added. Since the FusionScript Lua interpreter is a custom version of Lua not all native Lua libraries are guaranteed to work with fusion.

Python

Introduction to Python

Python has been adopted quickly for its efficient syntax and language features. Particularly in the Visual Effects industry, Python resembles a standard for scripting. Most post-production applications today make use of Python, which is especially beneficial if your goal is to streamline the production with scripting. Beyond VFX literally thousands of libraries offer Python bindings, making it possible to access a broad range of tools with a common language.

In order to work in FusionScript the official C-based implementation of Python, sometimes referred to as CPython, needs to be installed on your system as shown below.

Choice of Version

Python comes as Python version 2 or version 3. The latter was introduced to resolve core issues of Python, for the cost of backwards compatibility in syntax and features. Compare:

https://wiki.python.org/moin/Python2orPython3

In Fusion, you have the choice to either use Python 2.7 or 3.3. Depending on your task, either use 2.7 (widest range of applications supported) or 3.3 if your pipeline depends on it.

At the time of writing, the recommended VFX reference platform suggests the latest Python 2.7 version, so many facilities may depend on this version.

Documentation

Official documentation of python can be found here:

https://docs.python.org/2.7/

https://docs.python.org/3.3/

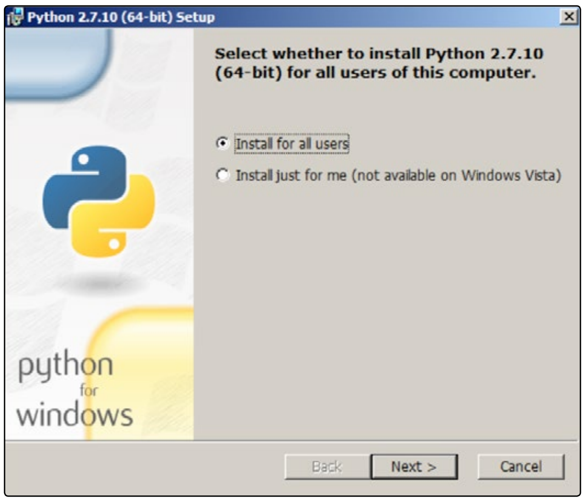
Installation

Windows

You need to have the latest Python 2.7 or Python 3.3 installed on your system in order to be usable with Fusion. To match Fusion it needs to be the 64 bit compile.

https://www.python.org/downloads/windows/

During installation, the install option needs to be set to "`Install for all users`" as shown below:



Python 2.7.10 (64-bit) Setup

Select whether to install Python 2.7.10 (64-bit) for all users of this computer.

[✓ Install for all users]

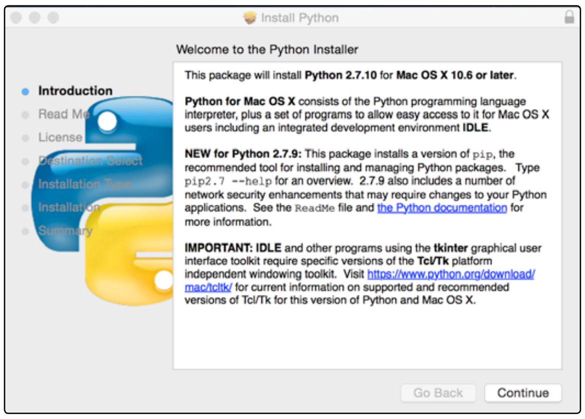
Install just for me (not available on Windows Vista)

This way the Python library is installed so that Fusion is able to pick it up during startup. Continue with the setup below.

Mac OS X

You need to have the latest Python 2.7 or Python 3.3 installed on your system in order to be usable with Fusion. To match Fusion it needs to be the 64 bit compile.

https://www.python.org/downloads/release/python-2710/



Install Python

Welcome to the Python Installer

This package will install **Python 2.7.10** for **Mac OS X 10.6 or later**.

Python for Mac OS X consists of the Python programming language interpreter, plus a set of programs to allow easy access to it for Mac OS X users including an integrated development environment **IDLE**.

NEW for Python 2.7.9: This package installs a version of `pip`, the recommended tool for installing and managing Python packages. Type `pip2.7 --help` for an overview. 2.7.9 also includes a number of network security enhancements that may require changes to your Python applications. See the `ReadMe` file and the Python documentation for more information.

IMPORTANT: IDLE and other programs using the **tkinter** graphical user interface toolkit require specific versions of the **Tcl/Tk** platform independent windowing toolkit. Visit https://www.python.org/download/mac/tcltk/ for current information on supported and recommended versions of Tcl/Tk for this version of Python and Mac OS X.

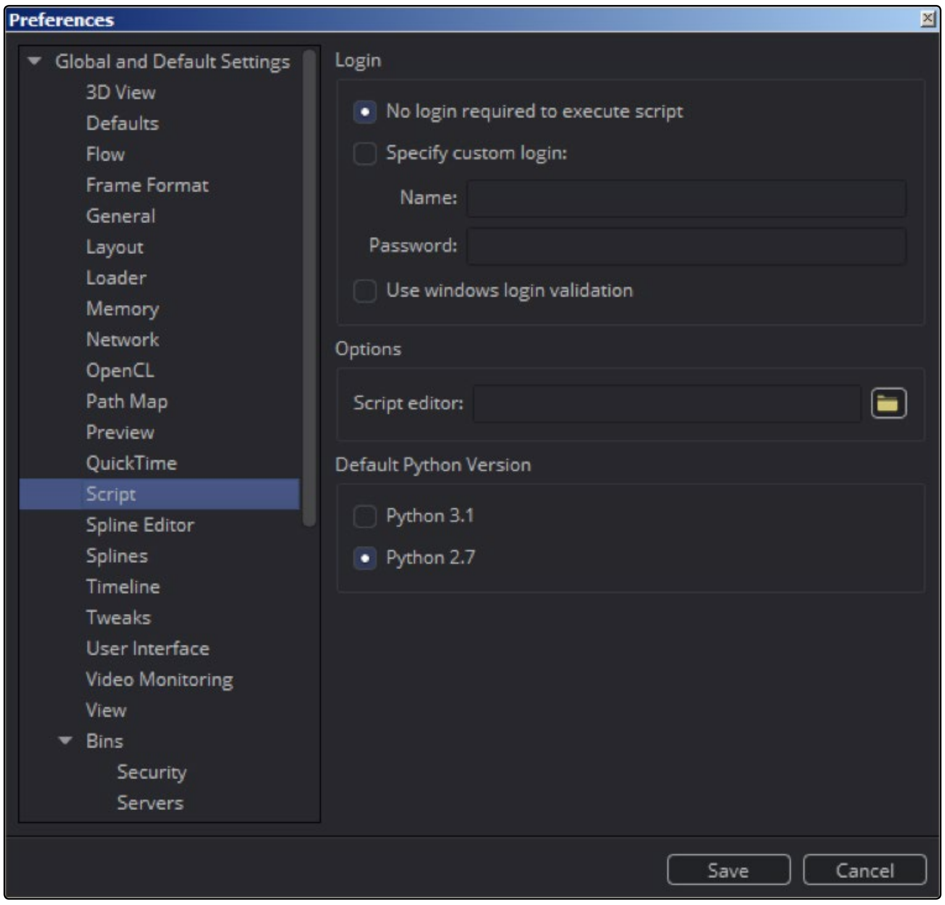
Setup

After the installation of Python, Fusion needs to be restarted.

As you could have both versions of Python installed, you need to specify the preferred version in your preferences.

Set the default Version for .py Files and default console at:

→ File Preferences...->Global and Default Settings->Script->Default Python Version



Login

[✓ No login required to execute script]
Specify custom login:
Name:
Password:
Use windows login validation

Options

Script editor:

Default Python Version

Python 3.1
[✓ Python 2.7]

Note

If you need to make sure that your script is run with either Python 2 or Python 3, you can set the file extension of the script to either .py2 or .py3, respectively.

Note this is a non-standard behavior and will only work within Fusion.

Libraries

In contrast to Lua, Python comes with a complete standard library. As a quick overview, here is a list of important modules.

- os (os & file system access)
- shutil (file system access)
- glob (file system matching & listing)
- os.path (os independent path handling)
- sys (system access)

For a complete list refer to:

https://docs.python.org/2/library/

https://docs.python.org/3.3/library/

Additionally, you can install external libraries either manually or by the eco system accessible through pip or easyinstall. Some libraries that are useful with Fusion are:

- slpp (Lua data parser for python)
  https://github.com/SirAnthony/slpp
  This library makes it easy to parse Lua tables, which most of the data in Fusion consists of.
- Pillow (Python Imaging Library Fork)
  Image manipulation framework
- Numpy
  Mathematical framework

Differences with FusionScript

As already noted Fuses cannot be written in Python.

Also EventScripts, callback scripts for certain events are also only possible with Lua.

Since historically FusionScript was Lua only, some methods that return multiple statements have a special Table() suffix variant to return the proper table for use in Python.

As the Lua collection is a tuple, you will need to pass a dictionary to the API in many cases, even when it seems to be treated like a list.

So each Value needs to have a key in the order of the entry.

For example a list like:

```
l = [“a”, “b”, “c”]
```

needs to map to a dictionary

```
d = {1: “a”, 2: “b”, 3:”c” }
```

Please note that Lua uses 1 as the first index key of its tuples, not 0. Python dictionaries do not have a particular order. Only the key indicates their order in this case.

Similarly, all Lua tuples result in dictionaries in Python that need to be parsed into Lists. If order does not matter, it can be simply done by:

```
l = d.values()
```

If order is important their values need to be sorted by their keys before conversion to a list. This can be achieved with a list comprehension:

```
l = [item[1] for item in sorted(d.items())]
```

Choice of Scripting Language

The following list compiles reasons for the use of one or the other language.

Pro Lua:

- Batteries included - No setup needed
- Therefore shared scripts will guarantee to work in Fusion without setup
- More features in Fusion
- Easier to parse Fusion Tables
- Lighter and faster
- Fusion is shipping with many scripts in Lua that can act as examples

Pro Python:

- Utilization of other Python scripts/apps in the pipeline
- Most major VFX apps use Python
- Allows external scripting for cross-app communication (Studio only)

- Strong standard library
- Higher usage & more third party libraries, scripts, and bindings
- Comes pre-installed in Linux & OSX

The recommendation should always be to stick with the one you know. It makes no sense to learn a completely new language in most cases if you are already familiar with either Lua or Python, especially when scripts and libraries exist that you can rely on.

If you are just starting with scripting, you should stick to Lua if all you care for is Fusion, and you want to make it possible for other artist to utilize your scripts without prior setup. Also the knowledge gained in scripting will be beneficial for writing custom Fuses.

If you are using other VFX applications that eventually also support Python this might be the better choice for Fusion as well. The choice can also depend on the standard libraries or a particular third party library. Research your required environment before making a choice will save you time in the long run.

Regardless Fusion with its FusionScript API will respect your choice.

Cross-Language Evaluation

Sometimes it is necessary or useful to call in from one language to the other to access certain features, e.g., you might want to access the Lua function dump from within Python.

With the console set to Py2 execute:

```
composition.Execute(“dump(comp:GetAttrs())”)
```

To execute the string as Python from within Lua use:

```
composition:Execute(“!Py: print(comp.GetAttrs())”)
```

To target a specific Python version use !Py2: or !Py3:

You may also want to run complete Lua or Python scripts. Use:

```
composition:RunScript(filePath)
```

Use either .lua, .py, .py2 or .py3 as file extension for the corresponding interpreter. Similar to the script menu, .py will execute in the Python interpreter that is installed and set in the preferences. As RunScript is also available in Python you may run .lua scripts from within Python.

Note

The shown scripts are executed in the context of the currently open composition. Hence, all the evaluation methods are members of the `composition` object.

```
composition:Execute(command)
composition:RunScript(filePath)
```

If you want to execute the scripts in the context of the application, use fusion instead.

```
fusion:Execute(command)
fusion:RunScript(filePath)
```

Please note that it is not possible to pass return objects from one language to the other.

Scripting and Debugging

Console

Fusion has a console build-in that outputs print statements in scripts. This is useful for scripts without a GUI, or as tool for simple debugging.

For example:

→ Lua

```
print("Hello World.")
```

→ Python

```
print("Hello World.")
print "Hello World." # This only works with Python 2.x
```

In all cases, the console will show `"Hello World."` If executed from the console, the command will be mirrored in the console preceding the interpreter: Lua>, Py2> or Py3>

When used with a collection, print will only output the reference to the collection. To display its content in a preformatted way, use:

→ Lua

```
dump(comp:GetAttrs())
```

Tip

If used in the console, FusionScript offers a short form of dump for Lua and Python:

```
==comp:GetAttrs() -- Same as the command above
```

The same can be achieved in Python with a module called `"Data pretty printer"` (`pprint`).

→ Python

```
from pprint import pprint # Needs to be loaded once
pprint(comp.GetAttrs())
```

Please note that all the collections coming from FusionScript are essentially Lua tuples. Compare to the chapter Scripting Languages.

Types of Scripts

Fusion supports different types of scripts based on the context, e.g., you might have a script that makes changes to the composition, while another script might only act on a certain tool.

Some of these contexts supply different sets of predefined objects. Like a Tool script will expose the tool it has been applied on as variable.

For better understanding, let us examine important script-contexts:

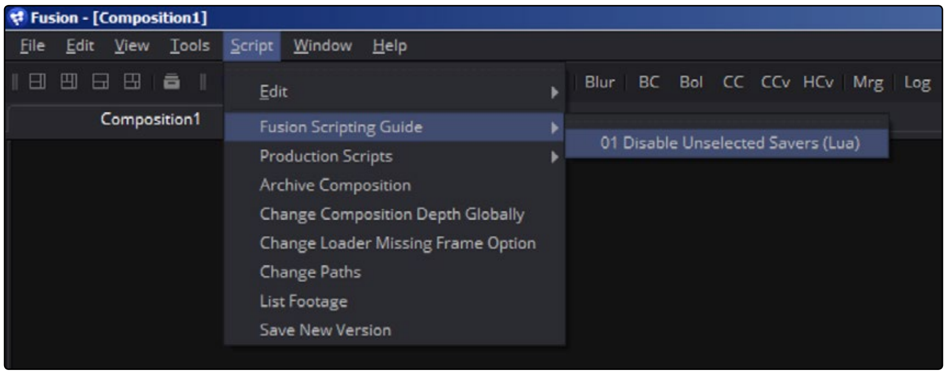
Interactive Scripts

Interactive scripts are all scripts within Fusion that require a user interaction to run. Most of these scripts are invoked by the user from the menu.

The contexts available are:

Composition Script

Compositing scripts are the most common type of scripts. They are stored inside the Scripts:/ Comp folder and run from the Scripts Menu. As their name implies their intended context is the Composition. Therefore, access to the fusion and the composition object is given. Nothing stops you from implementing functionality that acts on a single selected tool within a Composition script, but you should consider using a tool script instead. The Menu understands subfolders, so when a script is placed inside a subfolder, a submenu for that folder will be created.



Tool Script

Tool scripts act on a single tool. They are stored inside the Scripts:/Tool folder and are accessible and editable from the right-click context menu of the tool's properties. When invoked, the fusion, composition, and particular tool objects are available as variable.

Bin Script

Bin scripts are special scripts that act on the contents of a bin. They are stored inside the Scripts:/ Bin folder and are invoked through the context menu of the bin.

For more information about bins, refer to the Fusion User Manual.

Utility Script

Utility scripts are those that act on Fusion itself, rather than on a particular Composition. They are stored inside the Scripts:/Utility folder and can be accessed through the File > Script Menu. The fusion variable is available by default.

Script Libraries

A scriptlib is a file containing a library of functions that can be used in multiple scripts. Included with the default installation of Fusion is the bmd.scriptlib, which contains common useful functions. The scriptlib could have additions in it such as variable declarations (added to the globals table, for instance). Script Libraries are installed in the root of the scripts directory (by default Scripts:). In that directory, anything with a .scriptlib extension will be run whenever Fusion is started. In order to execute a scriptlib when a composition is created or opened, put the scriptlib in the Scripts:/Comp folder instead. The added benefit of the scriptlib is that you can instruct Fusion to run a set of code every time a composition is created or opened. The downside to this is that Fusion will execute the files in the scripts directory in an arbitrary order. This means that any code you write in the script libraries that is reliant upon other libraries may not work. To get around this, try inserting the functions that are needed at the top of the scriptlib.

Beyond passing functions into the global environment of the composition, the scriptlib also can be set up to perform default actions on a composition. It can also be used to create custom events set up in event suites.

External Scripts

External scripts are run from outside of Fusion but can still access the Fusion instance.

Commandline Scripts

In the install directory of Fusion an application called FuScript is available, which allows to run scripts directly from the command line.

The mac version is to be found inside the app bundle at Fusion.app/Contents/MacOS/fuscript.

FuScript can execute a .lua script file directly:

```
FuScript <script> [args]
```

The passed args can be accessed by the script via arg[1], arg[2] … arg[n], while arg[0] is reserved to point at the path of the script being executed.

FuScript also has an interactive shell which can be started with:

```
FuScript -i
```

For other uses of FuScript run it without any argument. A list of possible arguments will be printed to the console.

To connect FuScript to a running instance of Fusion use the following snippet:

```
fusion = Fusion()
fu = fusion
composition = fu.CurrentComp
comp = composition
SetActiveComp(comp)
```

From now on, the interactive shell will act like the build in shell in Fusion. By calling

SetActiveComp(comp), the global scope will accept calls to the composition. For example, the creation of tools like this:

```
blur = Blur()
```

This command will create a blur tool on the FlowView of the current open composition.

To run python version 2 or 3 you can specify the language like this:

```
FuScript <script> [args] -l python2
FuScript <script> [args] -l python3
```

Events & Callbacks

Events and callbacks get triggered when a certain event has occurred. A predefined callback will be invoked.

Event Suites

Event suites are installed as callbacks to certain events in Fusion. Install them like a regular scriptlib. Inside the scriptlib, add the following variable:

```
ev = AddEventSuite(“Composition”)
```

This variable has now access to events that occur when certain events are triggered.

Possible events are:

- OnOpen() -- Triggers every time a file is opened
- OnSave() -- Triggers every time a comp is saved
- OnSaveAs() -- Whenever save as is called
- OnStartRender() -- Whenever a render starts
- OnEndRender() -- Whenever a render ends
- OnFrameRendered() -- Whenever a frame is rendered
- OnTimeChange() -- Whenever the time changes
- OnActivateTool() -- Whenever a tool is made active

For example: Create a file called PrintSaverPathsOnRender.scriptlib in the Scripts:/Comp folder. Enter the following content:

```
globals.ev = AddEventSuite(“Composition”)
function ev:OnStartRender(event)
    local toollist=comp:GetToolList(“Saver”)
    for i, tool in pairs(toollist) do
    print(tool:GetInput(“Clip”))
    end
    self:Default(event)
end
```

Now start a render with a composition that has at least one Saver with a valid path defined. The console will print all the paths of the Savers. Although this sample does not add much value, it could easily be modified to check and manipulate the paths.

Note

Always use self:Default(event) to call the base implementation of the event.

This will allow you to create multiple events with different scriptlibs.

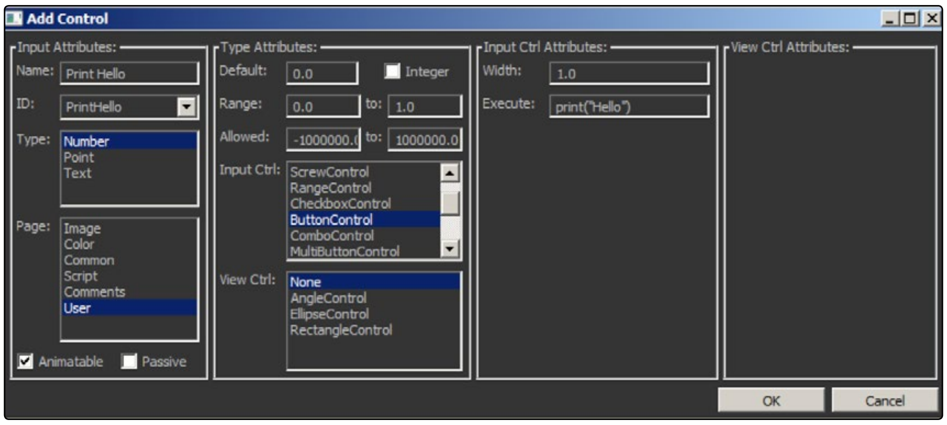
Removing an event suite is accomplished by running the RemoveEventSuite(suite) function. In the example scenario, the syntax would be:

```
RemoveEventSuite(ev)
```

Button Callbacks

Button callbacks are invoked when custom Button Controls within a tool are clicked.

Internally, the Attribute called `BTNCS_Execute` needs to be set. The easiest way to accomplish this is by using the UserControls ToolScript. When adding a Button control, a field labeled Execute can be used to call Lua commands.


Add Control

Input Attributes:
Name: Print Hello
ID: PrintHello
Type: Number (selected), Point, Text
Page: Image, Color, Common, Script, Comments, User (selected)
Animatable [✓], Passive

Type Attributes:
Default: 0.0 Integer
Range: 0.0 to: 1.0
Allowed: -1000000.(to: 1000000.0
Input Ctrl: ScrewControl, RangeControl, CheckboxControl, ButtonControl (selected), ComboControl, MultiButtonControl
View Ctrl: None (selected), AngleControl, EllipseControl, RectangleControl

Input Ctrl Attributes:
Width: 1.0
Execute: print("Hello")

View Ctrl Attributes:

The generated button control will end up in the composition as:

```
UserControls = ordered() {
            PrintHello = {
                  LINKID_DataType = “Number”,
                  INP_Default = 0,
                  BTNCS_Execute = “print(\”Hello\”)”,
                  LINKS_Name = “Print Hello”,
                  INPID_InputControl = “ButtonControl”,
            },
        },
```

When clicked `“Hello”` will be printed in the console.

Hotkey Script

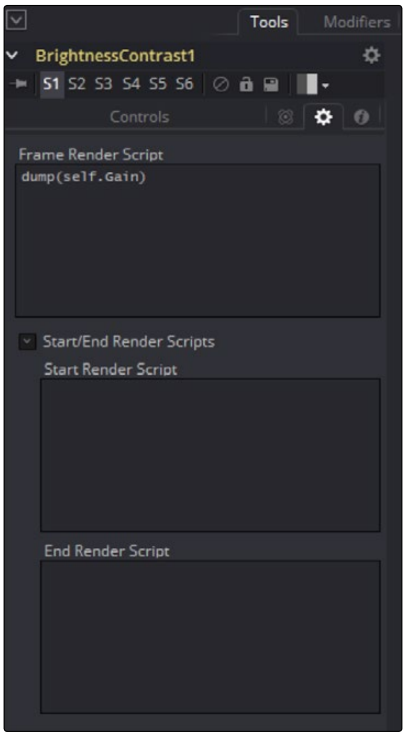
Hotkey scripts are scripts that can be attached to keyboard shortcuts in a particular context.

By default they are stored in a file called Fusion.hotkeys in the Profile: folder.

InTool Scripts

InTool scripts are special scripts that run on the tool during evaluation of each frame, at the start of the render or at the end. They are defined directly within the tool and have read access to a limited set of data through the input’s name—self, composition or comp, and fusion or fu. The limitation is supposed to prevent infinite loops, race conditions, and performance problems.

For example, you cannot call or change Inputs. If you want change Inputs based on a logic, use modifiers, expressions, or simple expressions. Also note that changing most of the Inputs in a tool will trigger a re-rendering and therefore the InTool Frame Render Script is evaluated again.



```
dump(self.Gain)
```

Simple Expressions

Simple Expressions are a limited subset of the scripting environment directly within each Input of a tool. They can be used as replacements for the expression modifier, to directly connect and change incoming Inputs based on calculations.

Fuses

Fuses are Lua scripted plugins that act as regular Tool. They may be multithreaded and contain OpenCL kernels to process on the GPU. Refer to the dedicated Fuses documentation and reference.

Fusion's Object Model

For a better understanding of FusionScript, it is worth looking under the hood of Fusion's object model. Although FusionScript and the following overview is a great simplification of the real application, it will help us to navigate around the application within the scripting API.

Overview

Fusion is composed of different objects with individual types. One possible object type is an Operator, also known as Tool. Each Operator might have a couple of Links, being Input or Output objects, that may be represented in GUI inside the properties view. The reference to the Composition is also a special object type, as is is Fusion itself. Even FileTypes, which represent file formats that can be read by a Loader, are objects.

Most objects contain a set of Attributes that represent the state of the object and it capabilities. Additionally they may contain Data, a special form of metadata.

Each object must be registered in an internal registry with its particular type and function. This way information about every object or tool can be read from the registry before an instance has been created.

While we do can access most of the information from the registry, FusionScript deals most of the time with the instances in the Application, Composition, Tool, Inputs etc.

Common Object Dependencies

This chapter pictures the common object dependencies in Fusion. This means that the users experiences the relations of objects similar to these dependencies, while the underlying implementation and exposed object hierarchy may look different.

This is only an excerpt of the most common objects a user is likely to use and may help to picture the interaction with Fusion from a user's point of view:

Fusion

→ Composition (collection)

→ Tool (collection)

→ Inputs (collection)

Some being MainInputs = Input connections on the FlowView

→ Type: Sets and gets type

Text

Number

Image

Data3D

- → Outputs (collection)

  Some being MainOutputs = Output connections on the FlowView

  - → Type: Sets and gets type

    Text

    Number

    Image

    Data3D

    other

While this view is highly simplified and neglects many aspects and features of the interface—like LUT, Viewers etc.—it is at the core the data that a user deals with most of the time.

Fusion Instance

The starting point for all access is a Fusion object. A Fusion object represents a running Fusion instance. It can create, open, and close compositions, stores application wide settings and preferences or persistent metadata. Fusion is able to open and manage multiple compositions from one Fusion instance. The graphical user interface represents these with a Tab-Layout. In scripting all currently loaded compositions are accessible with `fu:GetCompList()`. The currently active Composition can be accessed via `fu.CurrentComp` or `fu:GetCurrentComp()`. To load a composition use `fu:LoadComp(path, locked)` or create an empty composition using `fu:NewComp(locked, auto-close, hidden)`. You can also quit the Fusion instance by using `fu:Quit()`. If you are running the script from within Fusion it still will be executed. In reality the script is not bound to the Fusion instance. Instead a FuScript application is spawned that evaluates the scripts and communicates to the running Fusion instance. If your script exits, eventually the FuScript instance will also be stopped. This obviously also applies if running scripts from an external scripting environment as explained in the earlier chapter.

Composition Instance

A Composition may also store settings, attributes, and persistent metadata. While the Fusion instance holds Global Settings, each composition may have an individual set of settings. This behaviour is mimicked in the preferences dialog, where either global settings for each new composition, or individual settings of currently opened compositions can be changed. Most of the time the composition settings should be accessed to include the overrides for the current composition. This includes the PathMapping, which is used to identify paths from Fusion's relative path system.

The composition can be Saved and Closed, create Undos, Undo actions, and Redo them and Clear Undos altogether. Also, playback and rendering can be invoked from a composition.

Please note that you can `comp:Lock()` a composition, which prevents re-rendering due to changes and dialog pop-ups until the composition is unlocked again with `comp:Unlock()`. Use locking whenever possible if you manipulate the composition. You can query the lock state of a composition via `comp:IsLocked()`.

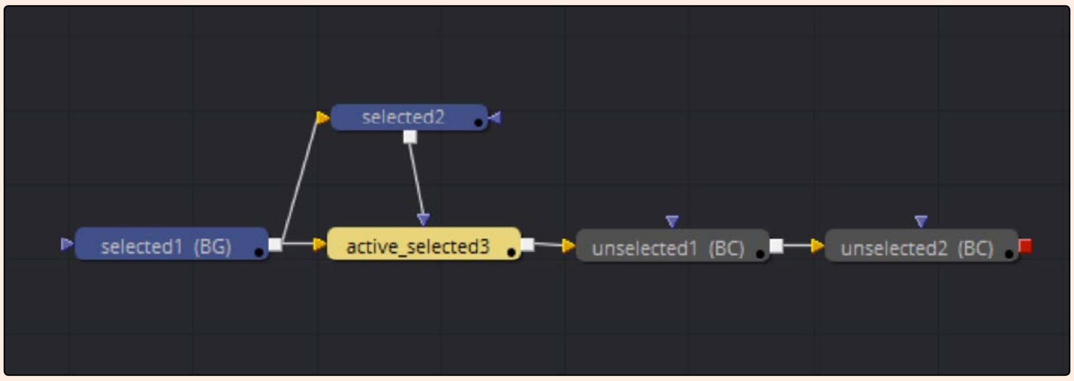
Tools on the composition can be queried. A composition can get and set the currently active tool via `comp.ActiveTool` and `comp:SetActiveTool(tool)`. All tools within the composition are queried with `comp:GetToolList()` while only the selected tools a queried with `comp:GetToolList(true)`.

Note

Fusion tools can have three selection states: unselected, selected and active & selected .

While the selected tools are the ones drag-selected (indicated by a blue color), the active tool is the last clicked tool (indicated by a yellow color). Still, an active tool is also automatically selected.

This behaviors enables a finer selection, e.g., when you want to copy one tool's settings to other tools, you can drag select all the target tools and then activate the source tool by clicking on it.



Selection of tools is part of the FlowView and can be triggered like this:

```
flow = comp.CurrentFrame.FlowView

flow:Select(Blur1, true)             -- Adds blur1 to the selection

flow:Select(Blur2, false)     -- Removes blur2 from the selection

flow:Select()                 -- Deselects all
```

Both `composition` and `fusion` have `GetPrefs()` and `SetPrefs()`, which store the preferences of Fusion and the local copy of the composition. If you cannot find a particular setting in there, take a look inside the Attributes as described later on.

Tool Instances

Tools are uniquely named operators of a particular type. Internally a tool is a subset of an Operator that is visible on the flow. It can be a Creator or Filter, 3D Tool, etc. Another example of Operator is a Modifier. It is a like a Tool but deals with Number or Text data instead of Image data. Still you can connect it to Inputs and other Modifiers.

For simplicity, we will talk about Tools most of the time while the techniques may also apply to different Operator types.

Read access to the name and its type is given with `tool.Name` and `tool.ID.` For read and write access of the name, use the attribute called `TOOLS_Name`. Note that the attribute `TOOLB_NameSet` indicates if the name was manually changed. If not, some tools will show additional information on the tile next to its name. For example, the loader will show the clip's filename.

Other important attributes are its PassThrough-State with `TOOLB_PassThrough` and Lock-State with `TOOLB_Locked`.

Similar to the selection state the position of the tool on the FlowView is not part of the tool instance but of the flow.

```
flow = comp.CurrentFrame.FlowView
==flow:GetPos(Blur1)         -- prints the position of Blur1
==flow:SetPos(Blur1, 5, 1)   -- sets the position of Blur1 to x = 5 y = 1
```

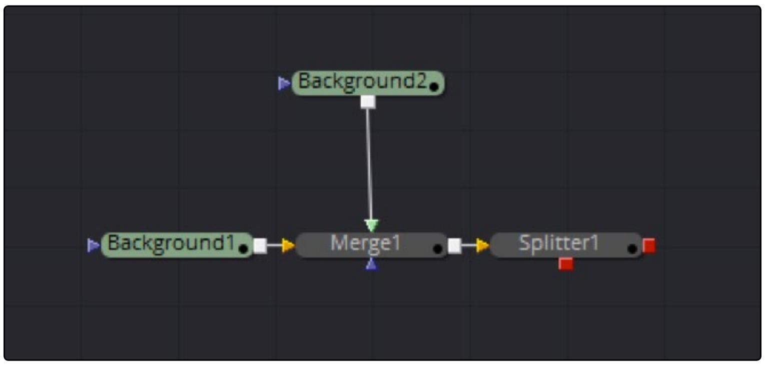
If many tools are repositioned, the shown method will be slow. You can queue re-positioning of multiple tools and apply it in one batch like this:

```
-- Repositions all tools in a column
flow = comp.CurrentFrame.FlowView
flow:QueueSetPos()
for i, tool in ipairs(comp:GetToolList()) do
    flow:QueueSetPos(tool, 0, i)
end
flow:FlushSetPosQueue()
```

Tools have Inputs and Outputs that are discussed in detail next.

MainInputs and MainOutputs

In general, tools have Inputs and Outputs. Property Inputs—being represented by controls in the properties view (e.g. the Gain slider in a ColorCorrector)—or the Inputs on the flow view that connect one tool to the other, so called MainInputs. Outputs are very similar although most of the time tools only have one MainOutput on the FlowView. An exception being the Stereo Splitter (Fusion Studio) as shown in the figure.



The distinction if an Input or Output is on the flow is made by defining them as MainInput and MainOutput during the development of the Plugin or Fuse.

Visible MainInputs can be queried by using `tool:FindMainInput(i)`, while MainOutputs are available with `tool:FindMainOutput(i)`. As there can be more than one MainInput or MainOutput, these methods require an argument `i` starting with 1.

If there is no result for the given index, the method returns `nil`. The following snippet shows how to query all MainInputs and MainOutputs of the active tool:

```
tool = comp.ActiveTool

if(tool ~= nil) then
    print (tool.Name)

    local i = 1
    while(true) do
        out = (tool:FindMainInput(i))
        if out == nil then break end
        print(string.format(“\tMainInput %d: %s”, i, out.Name))
        i = i + 1
    end
```

```
    i = 1
    while(true) do
        out = (tool:FindMainOutput(i))
        if out == nil then break end
        print(string.format(“\tMainOutput %d: %s”, i, out.Name))
        i = i + 1
    end
end
```

Inputs and Outputs

Next to the MainInput and MainOutputs there are other Inputs and Outputs. If Inputs are not hidden they can be represented as an Input control in the properties view. Still the underlying DataType might be the same. For example a Number DataType might be accessible through a slider control, a Checkbox, a DropdownList, a Multibutton etc.

To query the underlying DataType of an Input, use `inp:GetAttrs(“INPS_DataType”).`

To query the underlying DataType of an Output use `outp:GetAttrs(“OUTS_DataType”).`

A control allows users to change the corresponding value of the underlying DataType in the properties view. An optional preview control allows to change the value directly in the Viewer.

In scripting the value on an Input can be changed directly with an assignment, by using an index that represents a specific time or by using `tool:SetInput(“InputName”, value, [time]).`

Specifying the time only makes sense the input is animated as shown later. Only simple DataTypes like integers, float, and strings are supported.

Consider the following example:

```
Merge1.Angle = 10                          -- Sets Angle to 10
Merge1.Angle[5] = 20                   -- Sets Angle to 20 on frame 5
Merge1:SetInput("Angle", 20, 5)            -- Same as above
```

```
To get a value of a given Input use:
print(Merge1.Angle)                        -- Gets the Angle input handle
print(Merge1.Angle[TIME_UNDEFINED])        -- Gets the Angle value
print(Merge1.Angle[5])                             -- Gets Angle on frame 5
Merge1:GetInput("Angle", 5)                -- Same as above
```

Please note that you cannot use Merge1.Angle to retrieve a value as this will return the Input handle.

Querying Inputs

Like MainInputs on the Flow, Inputs can be connected to other Outputs like Published Inputs, Animated Inputs, or Modifiers. Although not represented with a FlowView, a similar connection flow is possible with all Inputs. The main difference being that MainInputs deal with Image data, Masks, Data3D, Particle Streams while regular Inputs deal with Numbers, Points, and Text etc.

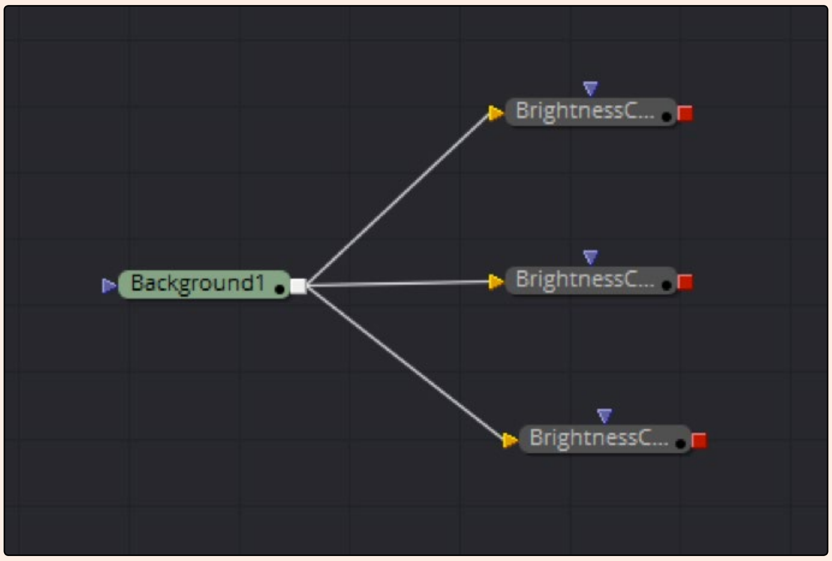
All Inputs, regardless of being MainInputs or not can be listed via tool:GetInputList(). All Outputs can be listed with tool:GetOutpuList(). In both cases, an optional filter of the DataType can be specified. Additionally if the name is known the Input and Output can be accessed directly as property of the tool. If you mouse hover over an Input, the status bar will show the name. E.g. to access the Gain Input of a BrightnessContrast tool use: BrightnessContrast1.Gain

Connections

An Input can be connected to an Output via the inp:ConnectTo(output) method. It can be disconnected via inp:ConnectTo(). To get the connected output of an Input use inp:GetConnectedOutput(). Similarly you can get all connected Inputs of an Output with outp:GetConnectedOutputs(). Please note the plural form of the latter command.

Note

By design, one Output might be connected to multiple Inputs, but one Input can only have one incoming Output connection.



Please note that both Output and Input share the same parent class called Link, which allows them to access `GetTool()` to refer to the Tool containing the Input or Output.

Inputs can be connected directly to their underlying DataType but they can also be connected to other inputs of the same DataType, a modifier or animation. As all of this internally is implemented as connection, once the upstream Input needs to be evaluated, all of it connected downstream outputs get queried. This allows for a complex connection scheme with many cross dependencies.

Animation

To animate an Input via script, the first step is to add a BezierSpline. A Bezier Spline is an animation curve that can be viewed in the spline editor. It is a storehouse for the information contained in the animated properties of a tool. To do this for a Merge's blend property, the following code could be employed:

```
Merge1.Blend = BezierSpline({})
```

By setting the input's value at a specific time, keyframes will be created.

If the property is a Point DataType, use the Path{} function instead to add a bezier-based path.

If the desire was to then animate the blend from 1 to 0 over the period of 100 frames, one could use the following code:

```
Merge1.Blend[1] = 1
Merge1.Blend[100] = 0
```

You can request a collection of all keyframes on a BezierSpline:

```
local spline, splineout, splinedata

-- gets the spline output that is connected to the Blend input,
splineout = Merge1.Blend:GetConnectedOutput()

-- then uses GetTool() to get the Bezier Spline modifier itself, and
if splineout then
    spline = splineout:GetTool()

    -- then uses GetKeyFrames() to get a table of a spline data.  This
    splinedata = spline:GetKeyFrames()

    -- data is then dumped.
    dump(splinedata)
end
```

The data returned consists of a nested table, one for each keyframe and with a key value of the keyframe's time. The subtables contain an entry for the keyframe's value, and optionally, subtables for the left and/or right handles, called `"LH"` and `"RH."` The handle subtables contain two entries, for the handle's X & Y position.

To remove key frames from an animated spline, set the value to nil.

```
Merge1.Blend[composition.CurrentTime] = nil
```

In the above case, the key frame was removed from the comp's current frame. However, if that key frame was the only point on an animation spline, the point would not be deleted, as splines must have at least one point at all times.

The animation can be deleted completely, reverting the Input to a static value.

Instead of specifying the time set the whole Input to nil.

```
Merge1.Blend = nil
```

Attributes

Attributes store information about the capabilities of a certain type, as well as some common flags that contribute to the object's state.

For example, in the case of a Tool the attributes may include the typename of the object, its name in the composition, its abbreviation shown in the Toolbar, its PassThrough and selection state, etc.

Attributes have read access but it is not guaranteed that you can change all Attributes. So while it is possible to change the PassThrough-State of a tool, it makes no sense to change its type.

Each Operator Type will have a different set of Attributes depending on its type. You cannot add your own Attributes. Instead, use a mechanism like Image stream metadata or Object Data.

In order to access the Attributes, the `GetAttrs()` method can be used. As it is provided by the Object superclass, pretty much all objects can have Attributes. So `GetAttrs()` is a good place to look for functionality or data within an object.

If no argument is given, all Attributes are returned. It is also possible to supply a single tag string to narrow down the search.

```
==Merge1:GetAttrs()                    -- dump all Tool Attributes
==Merge1.Blend:GetAttrs()                     -- Inputs also have attributes
==Merge1:GetAttrs("TOOLB_Locked")      -- Only show the Locked status
```

The tags consist of a Type prefix, a character for the type, an underline and the Name of the Attribute. So for example most Attributes within a Tool have a TOOL prefix, Inputs INP, Compositions COMP etc.

The type character stands for:

S	String
B	Boolean
N	Number (float)
I	Integer
H	Handle
NT	Number Table
IT	Integer Table
ST	String Table
BT	Boolean Table

In our example, TOOLB_Locked stands for a Tool Attribute of type boolean with the name `“Locked.”`

Attributes can be changed by using SetAttrs({}). The supplied table is required to have the Tag as key and the new value as value. Multiple attributes can be changed at a time, however not all Attributes can be changed at all. The following example renames `“Merge1”` to `“MyMerge”` and locks the tool in one call:

```
Merge1:SetAttrs({TOOLS_Name = “MyMerge”, TOOLB_Locked = true})
```

Object Data

Data is a special type of Metadata that is stored within the application preferences or composition.

As opposed to Metadata that is read from an image data stream (e.g., OpenEXR) and is passed from tool to tool, the Object Data is not passed with the data stream. Instead, it is consistent for the current state of the application, composition, or tool.

This makes it a perfect candidate for reliably storing states of custom scripts with the composition.

For example, let’s say a custom script with a GUI needs to store its last used path so that the user does not have to change the path each time the script is being used.

One option is to create a global variable and check if it is set on each run:

```
if globals.mytool_lastpath then
    path = mytool_lastpath
else
    path = “default/path”
end

-- ... Dialog with the path

globals.mytool_lastpath = path
```

However, once Fusion is closed the variable is gone. This strategy only makes sense for data that is not likely to change from session to session, like a cached list of currently loaded Tools.

But for our scenario, it may be wiser to store each latest path with the fusion preferences so that each new composition can reference the last used path, even when Fusion is closed and reopened.

```
local last_path = fusion:GetData(“mytool.lastpath”)
if last_path then
    path = last_path
else
    path = “default/path”
end

-- ... Dialog with the path

fusion:SetData(“mytool.lastpath”, path)
```

However, another strategy might be to store the data with the composition, so each composition can have its own path. Simply replace fusion with composition or any other context that makes sense if your case.

Please note that the dot notation is not random. Dots represent a subtable. So you can put multiple variables or even other nested tables inside of "mytool." Use this to your advantage, e.g., to define a domain wide root name that represents your studio, a sub table with the tools and their individual settings:

```
fusion:SetData("MyStudioInc.MyCompTool.DoMagic", true)
fusion:SetData("MyStudioInc.MyRenderSettings.RemoteNames", "clients")
...
```

Where is the actual ObjectData stored?

In the case of the fusion you will find the data in the Fusion8.prefs, at Global.Script.GlobalData.

With Compositions, tools etc. the ObjectData is stored with the respective object in the Composition file. As all these are Lua-Tables, go ahead and open the .comp file with a text processor. You should find the Object data you specified.

Note

A big benefit of Tool ObjectData is that it is stored directly within the Tool. It will be passed on if the tool is copied and pasted into another composition. However, a newly created tool will not have any ObjectData, so make sure to catch this default case by an EventSuite or similar.

Metadata

ObjectData is easily confused with regular Image Metadata. However, Image Metadata can only be read with scripts, but not changed, as it is tied to the Image Stream itself. You will need Fuses or Plugins to change the Image Stream and its Metadata directly. In order to access it, you will need to evaluate the Output up to the point where the Metadata was queried.

This is not needed in the case of ObjectData, as it depends on the Object instance and not on its underlying data stream.

In a Loader with a valid input access is Metadata like this:

```
==Loader1.Output[comp.CurrentTime].Metadata.Filename
```

In SimpleExpression, the evaluation is not needed, as it is automatically evaluated at the current time. For example put this in a text field's Expression field:

```
Loader1.Output.Metadata.Filename
```

Graphical User Interfaces

Although scripts can run in the background and output text to the Console often a graphical user interface is required. This way the logic of a script can be changed based on options set by the user. There are two options. For more complex user interfaces, Lua ships with the iup GUI library. Please refer to the documentation of the library, as its usage is beyond the scope of this document:

http://webserver2.tecgraf.puc-rio.br/iup/

The other option is a build-in dialog called AskUser.

Ask User

A simple way to build and evaluate a dialog is called: comp:AskUser(name, {table of inputs}).

Each input is a table structured as follows :

```
{Input Name, Input Type, Options ...}
```

Input Name (string, required)

This name is the index value for the controls value as set by the user (i.e., dialog.Control or dialog[`“Control Name”`]). It is also the label shown next to the control in the dialog, unless the Name option is also provided for the control.

Input Type (string, required)

A string value describing the type of control to display. Valid strings are FileBrowse,PathBrowse, Position, Slider, Screw, Checkbox, Dropdown, and Text. Each Input type has its own properties and optional values.

Options (misc)

Different control types accept different options that determine how that control appears and behaves in the dialog.

All script execution stops until the user responds to the dialog by selecting OK or Cancel.

The returned table contains the responses from the user, or nil if the user canceled the dialog.

Note

This function can only be called interactively, command line scripts cannot use this function.

For example, if you wanted to display a dialog that requested a path from a user, you might use the following script:

```
ret = composition:AskUser("A Sample Dialog", { {"Select a Directory", "PathBrowse"} } )

dump(ret)
```

Several of the Options are common to several controls. For example, the name option can be used with any type of control, and the DisplayedPrecision option can be used with any control that displays and returns numeric values. The commonly used options for controls are:

- Name (string)
  This option can be used to specify a more reasonable name for this inputs index in the returned table than the one used as a label for the control.
- Default (string)
  The default value displayed when the control is first shown.
- Min (integer)
  Sets the minimum value allowed by the slider or screw control.
- Max (numeric)
  Sets the maximum value allowed by the slider or screw control.
- DisplayedPrecision (numeric)
  Use this option to set how much precision is used for numeric controls like sliders, screws and position controls. A value of 2 would allow two decimal places of precision - i.e., 2.10 instead of 2.105
- Integer (boolean)
  If true the slider or screw control will only allow integer (non decimal) values, otherwise the slider will provide full precision. Defaults to false if not specified.

Control Types

The following table indicate types of control.

Text	Displays the Fusion textedit control, which is used to enter large amounts of text into a control.	Linear (integer) A number specifying how many lines of text to display in the control. Wrap (boolean) A true or false value that determines whether the text entered into the control will wrap to the next line when it reaches the end of the line. ReadOnly (boolean) If this option is set to true, the control will not allow any editing of the text within the control. Used for displaying non-editable information. FontName (string) The name of a truetype font to use when displaying text in this control. FontSize (numeric) A number specifying the font size used to display the text in this control.
FileBrowse PathBrowse ClipBrowse	The FileBrowse input allows you to browse to select a file on disk, while the PathBrowse input allows you to select a directory. ClipBrowse is used to get sequences with their appropriate filters.	Save (boolean) Set this option to true if the dialog is used to select a path or file which does not yet exist (i.e.when selecting a filae to save to)
Slider	Displays a standard Fusion slider control. Labels can be set for the high and low ends of the slider using the following options.	LowName (string) The text label used for the low (left) end of the slider. HighName (string) The text label used for the high (right) end of the slider.
Checkbox	Displays a standard Fusion checkbox control. You can display several of these controls, next to each other using the NumAcross option	Default (numeric) The default state of the checkbox, use 0 to leave the checkbox deselected, or 1 to enabled the checkbox. Defaults to 0 if not specified. NumAcross (numeric) If the NumAcross value is set, the dialog will reserve space to display two or more checkboxes next to each other. The NumAcross value must be set for all checkboxes to be displayed on the same row. See examples below for more information.

Position	Displays a pair of edit boxes used to enter X & Y coordinates for a center control or other position value. The default value of this control is a table with two values, one for the X value and one for the Y. The control returns a table of values.	Default (table {x,y}) A table with two numeric entries specifying the value for the x and y coordinates.
Screw	Displays the standard Fusion thumbnail or screw control. This control is almost identical to a slider in almost all respects except that its range is infinite, and so it is well suited for angle controls and other values without practical limits.	
Dropdown	Displays the standard Fusion drop down menu for selecting from a list of options. This control exposes and option call Options, which takes a table containing the values for the drop down menu. Note that the index for the Options table starts at 0, not 1 like is common in most FusionScript tables. So, if you wish to set a default for the first entry in a list, you would use Default=0, for the second Default=1, and so on	Default (num) A number specifying the index of the options table (below) to use as a default value for the drop down box when it is created. Default (table {string, string, string…}) A table of strings describing the values displayed by the drop down box.
Multibutton	Displays a Multibutton, where each option is drawn as a button. The same options are used like in a Dropdown.	Default (num) A number specifying the index of the options table (below) to use as a default value for the drop down box when it is created. Options (table {string, string, string…}) A table of strings describing the values displayed as buttons.

This example shows a dialog that contains most of the various control types:

```
composition_path = composition:GetAttrs().COMPS_FileName
```

```
msg = "This dialog is only an example. It does not actually do anything, "..
      "so you should not expect to see a useful result from running this script."
```

```
d    = {}
d[1] = {"File", Name = "Select A Source File", "FileBrowse", Default = composition_path}
d[2] = {"Path", Name = "New Destination", "PathBrowse" }
d[3] = {"Copies", Name = "Number of Copies", "Slider", Default = 1.0, Integer = true, Min = 1, Max = 5 }
d[4] = {"Angle", Name = "Angle", "Screw", Default = 180, Min = 0, Max = 360}
d[5] = {"Menu", Name = "Select One", "Dropdown", Options = {"Good", "Better", "Best"}, Default = 1}
d[6] = {"Center", Name = "Center", "Position", Default = {0.5, 0.5} }
d[7] = {"Invert", Name = "Invert", "Checkbox", NumAcross = 2 }
d[8] = {"Save", Name = "Save Settings", "Checkbox", NumAcross = 2, Default = 1 }
d[9] = {"Msg", Name = "Warning", "Text", ReadOnly = true, Lines = 5, Wrap = true, Default = msg}
dialog = composition:AskUser("A Sample Dialog", d)
if dialog == nil then
 print("You cancelled the dialog!")
else
 dump(dialog)
end
```

Note

In Python, make sure to create a dictionary with proper indices starting with 1 as explained in the Chapter about Python. For Example:

```
dialog = {1: {1: "dlgDir", "Name": "Select a Directory", 2: "PathBrowse"},
          2: {1: "dlgDir", "Name": "A Check Box", 2: "Checkbox", "Default": 1}}
ret = composition.AskUser("A simple dialog", dialog)
```

Scripting Reference

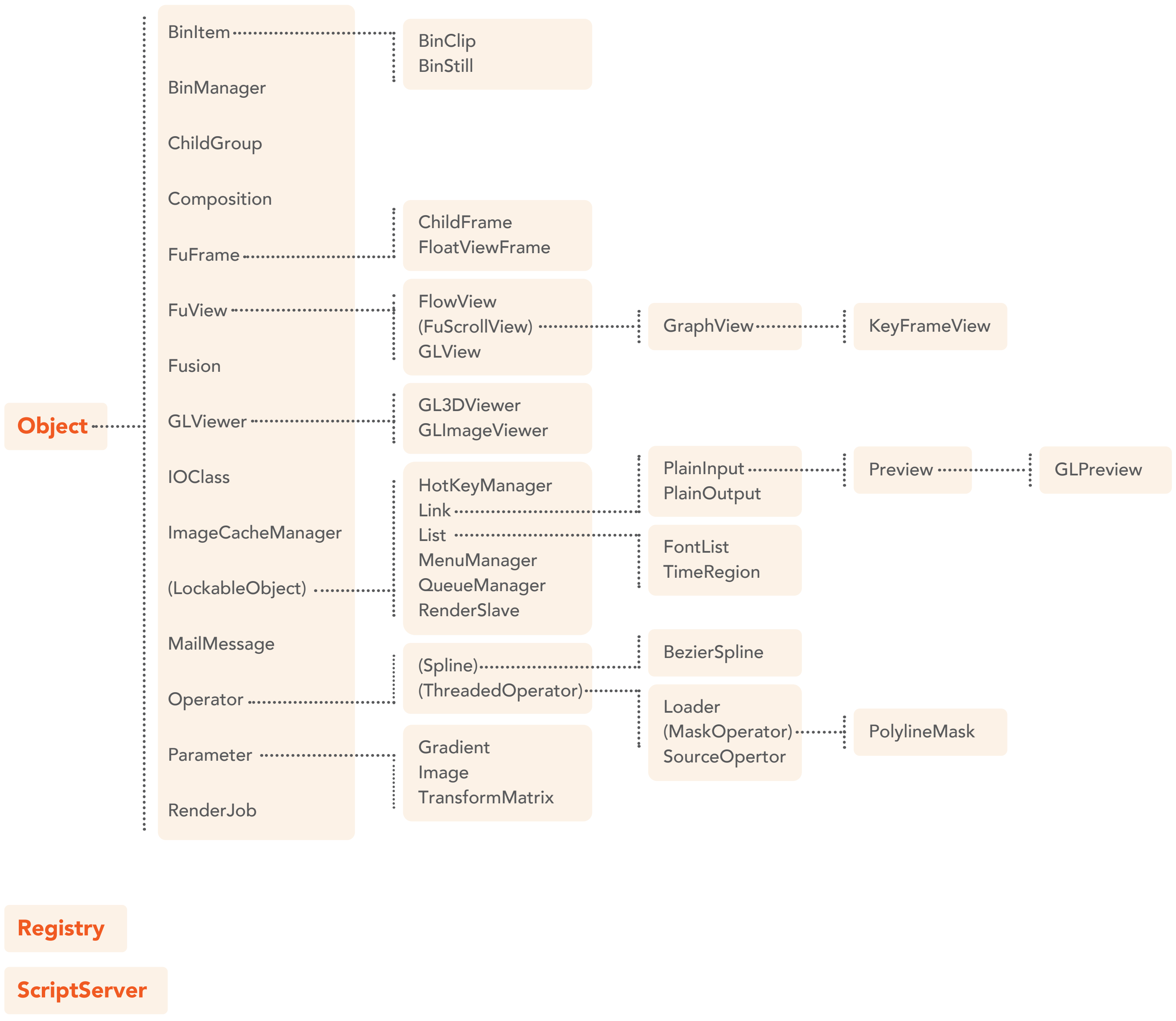
2

Content

Class Hierarchy

The following table shows Fusion's class hierarchy.

Undocumented intermediate objects are indicated by parenthesis.



Reference

BezierSpline

class BezierSpline

Parent class: Spline

Modifier that represents animation on a number value input.

Keyframes are interpolated with a bezier spline.

To animate Points use a Path instead.

→ Python usage:

```
comp.Merge1.Blend= comp.BezierSpline()
comp.Merge1.Blend[1] = 1
comp.Merge1.Blend[50] = 0
```

→ Lua usage:

```
Merge1.Blend = BezierSpline()
Merge1.Blend[1] = 1 -- Add keyframe at frame 1
Merge1.Blend[50] = 0        -- Add keyframe at frame 50
```

Methods

BezierSpline.AdjustKeyFrames (*start*, *end*, *x*, *y*, *operation*[, *pivotx*][, *pivoty*])

Set, Offset or Scale a range of key frames.

start, end Time range of keypoints to adjust (inclusive)

x, y Time and Value offsets/factors

operation Can be "set", "offset" or "scale" (case sensitive)

pivotx, pivoty optional values to scale around. Default is zero.

→ Parameters:

start (*number*) – start

end (*number*) – end

x (*number*) – x

y (*number*) – y

operation (*string*) – operation

pivotx (*number*) – pivotx

pivoty (*number*) – pivoty

`BezierSpline.DeleteKeyFrames(`*start[, end]*`)`

Delete key frames.

→ **Parameters**:

start (*number*) – start

end (*number*) – end

`BezierSpline.GetKeyFrames()`

Get a table of keyframes.

While Operator:GetKeyFrames() returns a table of the tool's valid extent, and Input:GetKeyFrames() returns a table of the keyframe times for any animation, when GetKeyFrames() is called from a BezierSpline modifier, it will return a table fully describing the spline's curvature.

The data returned consists of a table of subtables, one for each keyframe and with a key value of the keyframe's time. The subtables contain an entry for the keyframe's value, and optionally, subtables for the left and/or right handles, keyed by "LH" and "RH". The handle subtables contain two entries, for the handle's X & Y position.

Returns a table containing information about a spline control's animation keyframes.

An example table for a spline with three keyframes follows:

```
{
    [0.0] = { 2.0, RH = { 12.666667, 2.0 } },
    [38.0] = { 3.86, LH = { 25.333333, 3.666667 }, RH = { 46.0, 4.0 } },
    [62.0] = { 2.5, LH = { 54.0, 2.5 } }
}
```

→ **Python usage**:

```
from pprint import pprint
# gets the spline output that is connected to the Blend input
splineout = comp.Merge1.Blend.GetConnectedOutput()

# then uses GetTool() to get the Bezier Spline modifier itself, and
```

```
if splineout:
    spline = splineout.GetTool()

# then uses GetKeyFrames() to get a table of a spline data.
    splinedata = spline.GetKeyFrames()
    pprint(splinedata)
```

→ Lua usage:

```
-- gets the spline output that is connected to the Blend input
splineout = Merge1.Blend:GetConnectedOutput()

-- then uses GetTool() to get the Bezier Spline modifier itself, and
if splineout then
    spline = splineout:GetTool()

-- then uses GetKeyFrames() to get a table of a spline data.
    splinedata = spline:GetKeyFrames()
    dump(splinedata)
end
```

→ **Returns**: keyframes

→ **Return type**: table

`BezierSpline.SetKeyFrames(`*keyframes[, replace]*`)`

Set a table of keyframes.

This function allows you to set a spline's keyframes as well as its curvature. The table should consist of a table of subtables, one for each keyframe, each with a key value of the keyframe's time. The subtables should contain an entry for the keyframe's value, and may optionally contain subtables for the left and/or right handles, keyed by "LH" and "RH". The handle subtables should contain two entries, for the handle's X & Y position.

An example table for a spline with three keyframes follows:

```
{
    [0.0] = { 2.0, RH = { 12.666667, 2.0 } },
    [38.0] = { 3.86, LH = { 25.333333, 3.666667 }, RH = { 46.0, 4.0 } },
    [62.0] = { 2.5, LH = { 54.0, 2.5 } }
}
```

→ Parameters:

keyframes (*table*) – keyframes

replace (*boolean*) – replace

BinClip

`class BinClip`

Parent class: `BinItem`

Methods

`BinClip.CreateStamp()`

Create a stamp for this BinClip.

`BinClip.Defragment()`

Defragment this clip.

`BinClip.DeleteStamp()`

Delete the stamp for this BinClip.

BinItem

`class BinItem`

Parent class: `Object`

Methods

`BinItem.Delete()`

Delete the BinItem.

`BinItem.GetData([`*name*`])`

Get custom persistent data.

See Composition:GetData().

→ **Parameters**:

name (*string*) – name

→ **Returns**: Value

→ **Return type**: (number|string|boolean|table)

`BinItem.SetData(`*name, value*`)`

Set custom persistent data.

See Composition:SetData().

→ **Parameters**:

name (*string*) – name

value ((*number|string|boolean|table*)) – value

BinManager

`class BinManager`

Parent class: `Object`

Methods

`BinManager.Close()`

Close

`BinManager.DeleteSelected()`

DeleteSelected

`BinManager.GetRootID()`

GetRootID

`BinManager.GetRootLibraryInfo()`

GetRootLibraryInfo

`BinManager.GetSelectedIDs()`

GetSelectedIDs

`BinManager.IsOpen()`

IsOpen

`BinManager.Open()`

Open

`BinManager.PlaySelected()`

PlaySelected

`BinManager.Refresh()`

Refresh

`BinManager.RenameSelected()`

RenameSelected

`BinManager.SetLibraryRoot()`

SetLibraryRoot

`BinManager._Library_AddItem()`

_Library_AddItem

`BinManager._Library_DeleteItem()`

_Library_DeleteItem

`BinManager._Library_Reload()`

_Library_Reload

`BinManager._Library_UpdateItem()`

_Library_UpdateItem

BinStill

`class BinStill`

Parent class: `BinItem`

Methods

`BinStill.Defragment()`

Defragment this clip.

ChildFrame

`class ChildFrame`

Parent class: `FuFrame`

Represents the context of the frame window, that contains all the views.

Usually, there's just one ChildFrame object for each comp and you can retrieve it via comp.CurrentFrame.

Methods

`ChildFrame.ActivateFrame()`

Activates this frame window.

`ChildFrame.ActivateNextFrame()`

Activates the next frame window.

`ChildFrame.ActivatePrevFrame()`

Activates the previous frame window.

`ChildFrame.GetControlViewList()`

Returns the list of views from the Controls tabs.

- **Python usage**:

```
list = comp.CurrentFrame.GetControlViewList()
```

- **Lua usage**:

```
list = comp.CurrentFrame:GetControlViewList()
```

- **Returns**: views
- **Return type**: table

`ChildFrame.GetMainViewList()`

Returns the list of views from the Main tabs.

- **Returns**: views
- **Return type**: table

`ChildFrame.GetViewLayout()`

Retrieves the current view layout.

- **Returns**: layout
- **Return type**: table

`ChildFrame.SetViewLayout(`*layout*`)`

Sets the current view layout from a table.

- **Parameters**:

layout (*table*) – layout

- **Returns**: success
- **Return type**: boolean

`ChildFrame.SwitchControlView(`*id*`)`

Displays a given view from the Control tabs.

- **Parameters**:

id (*string*) – id

`ChildFrame.SwitchMainView(`*id*`)`

Displays a given view from the Main tabs.

→ **Parameters:**

id (*string*) – id

ChildGroup

`class ChildGroup`

Parent class: `Object`

Methods

`ChildGroup.GetID()`

GetID

`ChildGroup.GetOwner()`

GetOwner

Composition

`class Composition`

Parent class: `Object`

Represents an composition.

The Composition object's methods and members are directly available in the console and in comp scripts written in Lua. This means that you can simply type ==CurrentTime or call AddTool("Blur") without the need to prefix the command with comp. Python scripts have to use the full name.

Composition Attributes

Attribute Name	Type	Description
COMPN_CurrentTime	number	This is the current time that the composition is at. This is the time that the user will see, and any modifications that do not specify a time will set a keyframe at this time.
COMPB_HiQ	boolean	Indicates if the composition is currently in 'HiQ' mode or not.

Attribute Name	Type	Description
COMPB_Proxy	boolean	Indicates if the composition is currently in 'Proxy' mode or not.
COMPB_Rendering	integer	Indicates if the composition is currently rendering.
COMPN_RenderStart	number	The render start time of the composition. A render with no start specified will begin from this time.
COMPN_RenderEnd	number	The render end time of the composition. A render with no end specified will render this frame last.
COMPN_GlobalStart	number	The global start time of the comp. This is the start of time at which the composition is valid. Anything before this cannot be rendered or evaluated.
COMPN_GlobalEnd	number	The global end time of the composition. This is the end of time at which the comp is valid. Anything after this cannot be rendered or evaluated.
COMPN_LastFrameRendered	number	The most recent frame that has been successfully completed during a render.
COMPN_LastFrameTime	number	The amount of time taken to render the most recently completed frame, in seconds.
COMPN_AverageFrameTime	number	The average amount of time taken to render each frame to this point of the render, in seconds.
COMPN_TimeRemaining	number	An estimation of how much more time will be needed to complete this render, in seconds.
COMPS_FileName	string	The full path and name of the composition file.
COMPS_Name	string	The name of the composition.
COMPI_RenderFlags	integer	The flags specified for the current render.
COMPI_RenderStep	integer	The step value being used for the current render.
COMPB_Locked	boolean	This indicates if the composition is currently locked.

Members

`Composition.ActiveTool`

Represents the currently active tool on this comp (read-only).

→ **Getting**:

tool = Composition.ActiveTool – (Tool)

`Composition.AutoPos`

Enable autoupdating of XPos/YPos when adding tools.

→ **Getting**:

val = Composition.AutoPos – (boolean)

→ **Setting**:

Composition.AutoPos = val – (boolean)

`Composition.CurrentFrame`

Represents the currently active frame for this composition (read-only).

Do not confuse with CurrentTime.

→ **Getting**:

frame = Composition.CurrentFrame – (FuFrame)

`Composition.CurrentTime`

The current time position for this composition.

→ **Getting**:

val = Composition.CurrentTime – (number)

→ **Setting**:

Composition.CurrentTime = val – (number)

`Composition.UpdateMode()`

Represents the Some/All/None mode.

`Composition.XPos`

The X coordinate on the flow of the next added tool.

→ **Getting**:

val = Composition.XPos – (number)

→ **Setting**:

Composition.XPos = val – (number)

`Composition.YPos`

The Y coordinate on the flow of the next added tool.

→ Getting:

val = Composition.YPos – (number)

→ Setting:

Composition.YPos = val – (number)

Methods

`Composition.AbortRender()`

Stops any current rendering.

`Composition.AbortRenderUI()`

Asks the user before aborting the render.

`Composition.AddTool`*(id[, defsettings][, xpos][, ypos]*)

Adds a tool type at a specified position.

id the RegID of the tool to add.

defsettings specifies whether user-modified default settings should be applied for the new tool (true) or not (false, default).

xpos the X position of the tool in the flow view.

ypos the Y position of the tool in the flow view.

You can use the number -32768 (the smallest negative value of a 16-bit integer) for both x and y position. This will cause Fusion to add the tool as if you had clicked on one of the toolbar icons. The tool will be positioned next to the currently selected one and a connection will automatically be made if possible. If no tool is selected then the last clicked position on the flow will be used. The same behaviour can be achieved with the comp:AddToolAction method.

Returns a tool handle that can be used to control the newly added tool.

→ Python usage:

```
bg = comp.AddTool(“Background”, 1, 1)

mg = comp.AddTool(“Merge”, -32768, -32768)
```

→ Lua usage:

```
bg = comp:AddTool(“Background”, 1, 1)

mg = comp:AddTool(“Merge”, -32768, -32768)
```

- **Parameters**:

  id (*string*) – id

  defsettings (*boolean*) – defsettings

  xpos (*number*) – xpos

  ypos (*number*) – ypos

- **Returns**: tool
- **Return type**: Tool

`Composition.AddToolAction(`*id[, xpos][, ypos]*`)`

Adds a tool to the comp.

If no positions are given it will cause Fusion to add the tool as if you had clicked on one of the toolbar icons. The tool will be positioned next to the currently selected one and a connection will automatically be made if possible. If no tool is selected then the last clicked position on the flow will be used.

- **Parameters**:

  id (*string*) – id

  xpos (*number*) – xpos

  ypos (*number*) – ypos

- **Returns**: tool
- **Return type**: Tool

`Composition.AskRenderSettings()`

Show the Render Settings dialog.

`Composition.AskUser(`*title, controls*`)`

Present a custom dialog to the user, and return selected values.

The AskUser function displays a dialog to the user, requesting input using a variety of common fusion controls such as sliders, menus and textboxes. All script execution stops until the user responds to the dialog by selecting OK or Cancel. This function can only be called interactively, command line scripts cannot use this function.

The second argument of this function recieves a table of inputs describing which controls to display. Each entry in the table is another table describing the control and its options. For example, if you wanted to display a dialog that requested a path from a user, you might use the following script.

Returns a table containing the responses from the user, or nil if the user cancels the dialog.

Input Name (string, required)

This name is the index value for the controls value as set by the user (i.e. dialog.Control or dialog["Control Name"]). It is also the label shown next to the control in the dialog, unless the Name option is also provided for the control.

Input Type (string, required)

A string value describing the type of control to display. Valid strings are FileBrowse, PathBrowse, Position, Slider, Screw, Checkbox, Dropdown, and Text. Each Input type has its own properties and optional values, which are described below.

Options (misc)

Different control types accept different options that determine how that control appears and behaves in the dialog.

AskUser Inputs

Input Type	Description	Options
	Several of the Options are common to several controls. For example, the name option can be used with any type of control, and the DisplayedPrecision option can be used with any control that displays and returns numeric values.	Name (string) This option can be used to specify a more reasonable name for this inputs index in the returned table than the one used as a label for the control. Default (various) The default value displayed when the control is first shown. Min (integer) Sets the minimum value allowed by the slider or screw control. Max (numeric) Sets the maximum value allowed by the slider or screw control. DisplayedPrecision (numeric) Use this option to set how much precision is used for numeric controls like sliders, screws and position controls. A value of 2 would allow two decimal places of precision - i.e. 2.10 instead of 2.105 Integer (boolean) If true the slider or screw control will only allow integer (non decimal) values, otherwise the slider will provide full precision. Defaults to false if not specified.
FileBrowse PathBrowse ClipBrowse	The FileBrowse input allows you to browse to select a file on disk, while the PathBrowse input allows you to select a directory.	Save (boolean) Set this option to true if the dialog is used to select a path or file which does not yet exist (i.e. when selecting a file to save to)

Input Type	Description	Options
Screw	Displays the standard Fusion thumbwheel or screw control. This control is identical to a slider in almost all respects except that its range is infinite, and so it is well suited for angle controls and other values without practical limits.	
Text	Displays the Fusion textedit control, which is used to enter large amounts of Text into a control.	**Lines (integer)** A number specifying how many lines of text to display in the control. **Wrap (boolean)** A true or false value that determines whether the text entered into this control will wrap to the next line when it reaches the end of the line. **ReadOnly (boolean)** If this option is set to true the control will not allow any editing of the text within the control. Use for displaying non-editable information. **FontName (string)** The name of a true type font to use when displaying text in this control. **FontSize (numeric)** A number specifying the font size used to display the text in this control.
Slider	Displays a standard Fusion slider control. Labels can be set for the high and low ends of the slider using the following options.	**LowName (string)** The text label used for the low (left) end of the slider. **HighName (string)** The text label used for the high (right) end of the slider.

Input Type	Description	Options
Checkbox	Displays a Fusion standard checkbox control. You can display several of these controls next to each other using the NumAcross option.	Default (numeric) The default state for the checkbox, use 0 to leave the checkbox deselected, or 1 to enable the checkbox. Defaults to 0 if not specified. NumAcross (numeric) If the NumAcross value is set the dialog will reserve space to display two or more checkboxes next to each other. The NumAcross value must be set for all checkboxes to be displayed on the same row. See examples below for more information.
Position	Displays a pair of edit boxes used to enter X & Y co-ordinates for a center control or other positional value. The default value for this control is a table with two values, one for the X value and one for the Y value. This control returns a table of values.	Default (table {x,y}) A table with two numeric entries specifying the value for the x and y co-ordinates.
Dropdown Multibutton	Displays the standard Fusion drop down menu for selecting from a list of options. This control exposes an option called Options which takes a table containing the values for the drop down menu. Note that the index for the Options table starts at 0, not 1 like is common in most tables. So if you wish to set a default for the first entry in a list, you would use Default = 0, for the second Default = 1, and so on.	Default (num) A number specifying the index of the options table (below) to use as a default value for the drop down box when it is created. Options (table {string, string, string,...}) A table of strings describing the values displayed by the drop down box.

Python usage:

```
# In Python make sure to create a dictionary with proper indices starting with 1
dialog = {1: {1: “dlgDir”,   “Name”: “Select a Directory”, 2: “PathBrowse”},
          2: {1: “dlgCheck”, “Name”: “A Check Box”, 2: “Checkbox”, “Default”: 1}}
ret = composition.AskUser(“A sample dialog”, dialog)
```

Lua usage:

```
composition_path = composition:GetAttrs().COMPS_FileName

msg = “This dialog is only an example. It does not actually do anything, “..
      “so you should not expect to see a useful result from running this script.”

d    = {}
d[1] = {“File”, Name = “Select A Source File”, “FileBrowse”, Default = composition_path}
d[2] = {“Path”, Name = “New Destination”, “PathBrowse” }
d[3] = {“Copies”,Name = “Number of Copies”, “Slider”, Default = 1.0, Integer = true,
         Min = 1, Max = 5 }
d[4] = {“Angle”, Name = “Angle”, “Screw”, Default = 180, Min = 0, Max = 360}
d[5] = {“Menu”, Name = “Select One”, “Dropdown”, Options = {“Good”, “Better”, “Best”},
         Default = 1}
d[6] = {“Center”,Name = “Center”, “Position”, Default = {0.5, 0.5} }
d[7] = {“Invert”,Name = “Invert”, “Checkbox”, NumAcross = 2 }
d[8] = {“Save”, Name = “Save Settings”, “Checkbox”, NumAcross = 2, Default = 1 }
d[9] = {“Msg”, Name = “Warning”, “Text”, ReadOnly = true, Lines = 5, Wrap = true,
        Default = msg}
dialog = composition:AskUser(“A Sample Dialog”, d)
if dialog == nil then
 print(“You cancelled the dialog!”)
else
 dump(dialog)
end
```

- **Parameters**:

  title (*string*) – title

  controls (*table*) – controls

- **Returns**: results
- **Return** type: table

`Composition.ChooseTool(`*path*`)`

Displays a dialog with a list of selectable tools.

Returns the RegID of the selected tool or nil if the dialog was canceled.

- **Parameters**:

  path (*string*) – path

- **Returns**: ID
- **Return** type: string

`Composition.ClearUndo()`

Clears the undo/redo history for the composition.

`Composition.Close()`

The Close function is used to close a composition. The Fusion Composition object that calls the function will then be set to nil.

If the comp is in locked mode, then the Close function will not attempt to save the comp, whether the comp has been modified or not since its last save. If modifications have been made that should be kept, call the Save() function first.

If the comp is unlocked, it will ask if the comp should be saved before closing.

Returns true if the composition was successfully closed, nil if the composition failed to close.

`Composition.Copy()`

Note: This method is overloaded and has alternative parameters. See other definitions.

Copy tools to the Clipboard.

Accepts no parameters (currently selected tools), a tool or a list of tools.

Returns true if successful, else false.

- **Returns**: success
- **Return** type: boolean

`Composition.Copy(`*tool*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Copy tools to the Clipboard.

Accepts no parameters (currently selected tools), a tool or a list of tools.

Returns true if successful, else false.

→ **Parameters**:

tool (*Tool*) – tool

→ **Returns**: success

→ **Return type**: boolean

`Composition.Copy(`*toollist*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Copy tools to the Clipboard.

Accepts no parameters (currently selected tools), a tool or a list of tools.

Returns true if successful, else false.

→ **Parameters**:

toollist (*table*) – toollist

→ **Returns**: success

→ **Return type**: boolean

`Composition.CopySettings()`

Note: This method is overloaded and has alternative parameters. See other definitions.

Copy a tools to a settings table.

Accepts no parameters (currently selected tools), a tool or a list of tools.

Returns the toollist as settings table.

→ **Returns**: ettings

→ **Return type**: table

`Composition.CopySettings(`*tool*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Copy a tools to a settings table.

Accepts no parameters (currently selected tools), a tool or a list of tools.

Returns the toollist as settings table.

→ **Parameters**:

tool (*Tool*) – tool

→ **Returns**: settings

→ **Return type**: table

`Composition.CopySettings(`*toollist*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Copy a tools to a settings table.

Accepts no parameters (currently selected tools), a tool or a list of tools.

Returns the toollist as settings table.

- **Parameters**:

  toollist (*table*) – toollist

- **Returns**: settings
- **Return type**: table

`Composition.DisableSelectedTools()`

Pass-through the selected tools.

`Composition.EndUndo(`*keep*`)`

The StartUndo() function is always paired with an EndUndo() function. Any changes made to the composition by the lines of script between StartUndo() and EndUndo() are stored as a single Undo event.

Changes captured in the undo event can be undone from the GUI using CTRL-Z, or the Edit menu. They can also be undone from script, by calling the Undo function. keep determines whether the captured undo event is to kept or discarded. Specifying 'true' results in the undo event being added to the undo stack, and appearing in the appropriate menu. Specifying 'false' will result in no undo event being created. This should be used sparingly, as the user (or script) will have no way to undo the preceding commands.

If the script exits before the EndUndo() is called Fusion will automatically close the undo event.

- **Python usage**:

```
composition.StartUndo("Add some tools")
bg1 = comp.Background()
pl1 = comp.Plasma()
mg1 = comp.Merge({ "Background": bg1, "Foreground": pl1 })
composition.EndUndo(True)
```

- **Lua usage**:

```
composition:StartUndo("Add some tools")
bg1 = Background{}
pl1 = Plasma{}
mg1 = Merge{ Background = bg1, Foreground = pl1 }
composition:EndUndo(true)
```

→ **Parameters**:

keep (*boolean*) – keep

`Composition.Execute()`

Executes a script string for the composition. To execute a script in the context of fusion use fusion:Execute(...) instead.

By default Lua is used as interpreter. To use python prepend the following prefix:

!Py: default Python version. !Py2: Python version 2. !Py3: Python version 3.

→ **Python usage**:

```
comp.Execute("print('Hello from Lua!')")

comp.Execute("!Py: print('Hello from default Python!')")

comp.Execute("!Py2: print 'Hello from Python 2!'")

comp.Execute("!Py3: print ('Hello from Python 3!')")
```

→ **Lua usage**:

```
comp:Execute("print('Hello from Lua!')")

comp:Execute("!Py: print('Hello from default Python!')")

comp:Execute("!Py2: print 'Hello from Python 2!'")

comp:Execute("!Py3: print ('Hello from Python 3!')")
```

`Composition.FindTool(`*name*`)`

Finds first tool by name.

→ **Parameters**:

name (*string*) – name

→ **Returns**: tool

→ **Return type**: Tool

`Composition.FindToolByID(`*id*`[, `*prev*`])`

Finds first tool of a given type.

Returns only the first found tool.

To find the next tool use the prev parameter to supply the previous tool.

→ Python usage:

```
# Create three Blur tools
blur1 = comp.Blur()
blur2 = comp.Blur()
blur3 = comp.Blur()

print (comp.FindToolByID(“Blur”).Name)
# Prints: Blur1
print (comp.FindToolByID(“Blur”, blur1).Name)
# Prints: Blur2
print (comp.FindToolByID(“Blur”, blur2).Name)
# Prints: Blur3
```

→ Lua usage:

```
-- Create three Blur tools
blur1 = Blur
blur2 = Blur
blur3 = Blur

print (comp:FindToolByID(“Blur”).Name)
-- Prints: Blur1
print (comp:FindToolByID(“Blur”, blur1).Name)
-- Prints: Blur2
print (comp:FindToolByID(“Blur”, blur2).Name)
-- Prints: Blur3
```

→ Parameters:

id (*string*) – id

prev (*Tool*) – prev

→ Returns: tool

→ Return type: Tool

`Composition.GetCompPathMap([`*built_ins*`][, `*defaults*`])`

Returns a table of all Composition path maps.

build_ins If set build-in (read-only) PathMaps will be returned.

defaults If set the default PathMaps will be returned, else excluded.

→ **Python usage**:

```
# Returns custom PathMaps
==comp.GetCompPathMap(False, False)
# Show all, same as true, true
==comp.GetCompPathMap()
```

→ **Lua usage**:

```
-- Returns custom PathMaps
==comp:GetCompPathMap(false, false)

-- Show all, same as true, true
==comp:GetCompPathMap()
```

→ **Parameters**:

built_ins (*boolean*) – built_ins

defaults (*boolean*) – defaults

→ **Returns**: map

→ **Return type**: table

`Composition.GetConsoleHistory()`

This function is useful for getting all information displayed in the console between two points. Could be used to search for warnings or errors generated by previous scripts.

Returns a table with the history of the console between two points. If endSeq is omitted, the startSeq the console sequence number that the script will start reading from.

endSeq the console sequence number that the script will stop reading at.

script gets all history starting from the variable passed into startSeq. If both values are omitted, returns a general table about the history of the console (number of lines, etc.) If no parameters are given the total number of lines will be returned in the Total key.

→ Lua usage:

```
-- Get the total number of console lines
dump(composition:GetConsoleHistory().Total)

-- Get the console history lines 1 and 2
dump(composition:GetConsoleHistory(1, 2))
```

`Composition.GetData([`*name*`])`

Get custom persistent data.

name name of the data. This name can be in "table.subtable" format, to allow persistent data to be stored within subtables.

Persistent data is a very useful way to store names, dates, filenames, notes, flags, or anything else, in such a way that they are permanently associated with this instance of the object, and are stored along with the object using SetData(), and can be retrieved at any time with GetData().

The method of storage varies by object: SetData() called on the Fusion app itself will save its data in the Fusion.prefs file, and will be available whenever that copy of Fusion is running. Calling SetData() on any object associated with a Composition will cause the data to be saved in the .comp file, or in any settings files that may be saved directly from that object. Some ephemeral objects that are not associated with any composition and are not otherwise saved in any way, may not have their data permanently stored at all, and the data will only persist as long as the object itself does.

Returns a value that has been fetched from the object's persistent data. It can be of almost any type.

→ Python usage:

```
from datetime import datetime
tool = comp.ActiveTool
tool.SetData("Modified.Author", fusion.GetEnv("USERNAME"))
tool.SetData("Modified.Date", str(datetime.now()))
author = tool.GetData("Modified.Author")
dt = tool.GetData("Modified.Date")
print("Last modified by {0} on {1}".format(author, dt))
```

→ Lua usage:

```
tool = tool or comp.ActiveTool
tool:SetData("Modified.Author", fusion:GetEnv("USERNAME"))
tool:SetData("Modified.Date", os.date())

author = tool:GetData("Modified.Author")
dt = tool:GetData("Modified.Date")

print("Last modified by" ..author.. " on " ..dt)
```

→ Parameters:

name (*string*) – name

→ Returns: value

→ Return type: (*number|string|boolean|table*)

`Composition.GetFrameList()`

Retrieves a table of the comp's ChildFrames.

ChildFrames are the windowed workspace of Fusion. This function allows the user to access each of the available ChildFrame window objects, and their views.

→ Python usage:

```
windowlist = composition.GetFrameList()
tool = comp.ActiveTool
for window in windowlist.values():
    window.ViewOn(tool, 1)
```

→ Lua usage:

```
windowlist = composition:GetFrameList()

tool = comp.ActiveTool
for i, window in pairs(windowlist) do
    window:ViewOn(tool, 1)
end
```

`Composition.GetNextKeyTime([`*time*`][, `*tool*`])`

Returns the keyframe time of the next keyframe.

It can be used to either check for a keyframe among all tools in the composition, or for a specified tool only.

time The source time for the search.

tool If set keyframes only for the tool will be returned.

→ **Parameters**:

time (*number*) – time

tool (*Tool*) – tool

→ **Returns**: time

→ **Return type**: number

`Composition.GetPrefs([`*prefname*`][, `*exclude-defaults*`])`

Retrieves a table of comp-specific preferences, or a single value.

prefname The name of the specific preference to be retrieved. Use dots to indicate subtables. If no prefname or nil is specified, a table of all the preferences is returned.

exclude-defaults Do not include preferences with defaults if true

This function is useful for getting the full table of preferences for a Composition, or a subtable, or a specific value.

→ **Python usage**:

```
from pprint import pprint

# All preferences
pprint(comp.GetPrefs())

# A sepcific preference
pprint(comp.GetPrefs("Comp.AutoSave.Enabled"))

# All but default preferences
pprint(comp.GetPrefs(None, False))
```

- Lua usage:

```
-- All preferences
dump(comp:GetPrefs())

-- A sepcific preference
dump(comp:GetPrefs("Comp.AutoSave.Enabled"))

-- All but default preferences
dump(comp:GetPrefs(nil, false))
```

- **Parameters**:

  prefname (*string*) – prefname

  exclude-defaults (*boolean*) – exclude-defaults

- **Returns**: prefs
- **Return type**: table

`Composition.GetPrevKeyTime([`*time*`][, `*tool*`])`

Returns the keyframe time of the previous keyframe.

It can be used to either check for a keyframe among all tools in the composition, or for a specified tool only.

time The source time for the search.

tool If set keyframes only for the tool will be returned.

- **Parameters**:

  time (*number*) – time

  tool (*Tool*) – tool

- **Returns**: time
- **Return type**: number

`Composition.GetPreviewList([`*include_globals*`])`

Retrieves a table of previews.

The GetPreviewList function is used to determine what views are available for a flow or for Fusion. The object itself is a View object that can then be passed on to the various functions that affect views in Fusion.

Returns a table of all available views for a composition. For floating views use the fusion:GetPreviewList function instead.

- **Parameters**:

  include_globals (*boolean*) – include_globals

- **Returns**: previews
- **Return type**: table

`Composition.GetToolList([`*selected*`][, `*regid*`])`

Returns a table of all tools or selected tools.

selected If the selected argument is set to true then GetToolList returns a list of handles to the selected tools in the composition. If no tools are selected then the table returned is nil. If the selected argument is false, or empty then a table with handles to all tools in the composition are returned.

regid This string value will limit the return of the GetToolList function to tools of a specific type (this type is related to the TOOLS_RegID attribute).

- **Python usage**:

```
from pprint import pprint
# outputs the name of every tool in the composition

pprint(composition.GetToolList())

# Get all selected tools
pprint(composition.GetToolList(True))

# Get all loaders
pprint(comp.GetToolList(False, “Loader”))
```

- **Lua usage**:

```
-- outputs the name of every tool in the composition
dump(composition:GetToolList())

-- Get all selected tools
dump(composition:GetToolList(true))

-- Get all loaders
dump(comp:GetToolList(false, “Loader”))
```

- **Parameters**:

  selected (*boolean*) – selected

  regid (*string*) – regid
- **Returns**: tools
- **Return type**: table

`Composition.GetViewList()`

Returns all the view in the composition.

`Composition.Heartbeat()`

Heartbeat

`Composition.IsLocked()`

Returns true if popups and updates are disabled.

Use this function to see whether a composition is locked or not.

Returns a boolean with the locked status of the comp.

- **Returns**: locked
- **Return type**: boolean

`Composition.IsPlaying()`

Returns true if the comp is being played.

- **Returns**: playing
- **Return type**: boolean

`Composition.IsRendering()`

Returns true if the comp is busy rendering.

It will return true if it is playing, rendering, or just rendering a tool after trying to view it.

This is equal to the state of COMPB_Rendering composition attribute.

- **Returns**: rendering
- **Return type**: boolean

`Composition.Lock()`

Lock the composition from updating.

The Lock() function sets a composition to non-interactive ("batch", or locked) mode. This makes Fusion suppress any dialog boxes which may appear, and additionally prevents any re-rendering in response to changes to the controls. A locked composition can be unlocked with the Unlock() function, which returns the composition to interactive mode.

It is often useful to surround a script with Lock() and Unlock(), especially when adding tools or modifying a composition. Doing this ensures Fusion won't pop up a dialog to ask for user input, e.g. when adding a Loader, and can also speed up the operation of the script since no time will be spent rendering until the comp is unlocked.

→ Python usage:

```
comp.Lock()
# Will not open the file dialog, since the composition is locked

my_loader = comp.Loader()
comp.Unlock()
```

→ Lua usage:

```
comp:Lock()
-- Will not open the file dialog, since the composition is locked
my_loader = Loader()
comp:Unlock()
```

`Composition.Loop(`*enable*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Enables looping interactive playback.

This function is used to turn on the loop control in the playback controls of the composition.

→ Parameters:

enable (*boolean*) – enable

`Composition.Loop(`*mode*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Enables looping interactive playback.

This function is used to turn on the loop control in the playback controls of the composition.

→ Parameters:

mode (*string*) – mode

`Composition.MapPath(`*path*`)`

Expands path mappings in a path string.

Retruns a file or directory path with all path maps expanded into their literal path equivalents.

There are a number of default and user-specified path maps within Fusion that are intended to provide convenient ways to access common locations, or for flexibility in scripting. These can be any string, but often end in a colon, e.g. Fusion:, Comp:. They are expanded into a literal path as specified by the Path Maps preferences page.

However, many Fusion functions (and all Lua functions) require strictly literal paths. MapPath() can be used to easily convert any path map into a fully-expanded literal path. If there is no path map at the beginning of the path, MapPath() will just return the unchanged path.

In addition to expanding all global path maps like Fusion:MapPath(), Composition:MapPath() will also expand any path maps listed in the composition's Path Map preferences, and the following built-in defaults.

For multiple directories use MapPathSegments().

→ **Python usage**:

```
print(composition.MapPath("Comp:footage\\file0000.tga"))
```

→ **Lua usage**:

```
print(composition:MapPath("Comp:footage\\file0000.tga"))
```

→ **Parameters**:

path (*string*) – path

→ **Returns**: mapped

→ **Return type**: string

`Composition.MapPathSegments(`*path*`)`

Expands all path mappings in a multipath.

MapPathSegments is similar to MapPath but works with strings that contain multiple directories. The return value is a table with all expanded paths while MapPath only expands the first segment and discards the rest.

→ **Python usage**:

```
from pprint import pprint

pprint(comp.MapPathSegments("AllDocs:Settings;Fusion:Settings"))
```

```
# Returns
# {1.0:  'C:\\Users\\Public\\Documents\\Blackmagic Design\\Fusion\\Settings',
# 2.0:  'C:\\Program Files\\Blackmagic Design\\Fusion 8\\Settings'}
```

→ **Lua usage**:

```
dump(comp:MapPathSegments("AllDocs:Settings;Fusion:Settings"))

-- Returns table: 0x03800440
--   1 = C:\Users\Public\Documents\Blackmagic Design\Fusion\Settings
--   2 = C:\Program Files\Blackmagic Design\Fusion 8\Settings
```

→ **Parameters**:

path (*string*) – path

→ **Returns**: mapped

→ **Return type**: table

`Composition.NetRenderAbort()`

NetRenderAbort

`Composition.NetRenderEnd()`

NetRenderEnd

`Composition.NetRenderStart()`

NetRenderStart

`Composition.NetRenderTime()`

NetRenderTime

`Composition.Paste([`*settings*`])`

Pastes a tool from the Clipboard or a settings table.

settings if not supplied the Clipboard will be used.

→ **Parameters**:

settings (*table*) – settings

→ **Returns**: success

→ **Return type**: boolean

`Composition.Play([`*reverse*`])`

Starts interactive playback.

This function is used to turn on the play control in the playback controls of the composition.

reverse Play in reverse direction.

→ Parameters:

reverse (*boolean*) – reverse

`Composition.Print()`

Print in the context of the composition.

Useful to print to a console of a different composition.

→ Python usage:

```
new_comp = fu.NewComp()

new_comp.Print("Hello World")
```

→ Lua usage:

```
new_comp = fu:NewComp()

new_comp:Print("Hello World")
```

`Composition.Redo(`*count*`)`

Redo one or more changes to the composition.

The Redo function reverses the last undo event in Fusion.

Note that the value of count can be negative, in which case Redo will behave as an Undo, acting exactly as the Undo() function does.

count specifies how many redo events to trigger.

→ Parameters:

count (*number*) – count

`Composition.Render([`*wait*`][, `*start*`][, `*end*`][, `*proxy*`][, `*hiq*`][, `*motionblur*`])`

Note: This method is overloaded and has alternative parameters. See other definitions.

Start a render.

The Render function starts rendering the current composition. There are two forms for calling this function, one where the arguments are passed directly, and a second form where all the arguments are passed in a table. The table format is useful for declaring non-contiguos render ranges, such as the following one.

Returns true if the composition rendered successfully, nil if it failed to start or complete the render.

Arguments

wait_for_render a true or false value indicating whether the script should wait for the render to complete, or continue processing once the render has begun.

renderstart the frame to start rendering at.

renderend the frame to stop rendering at.

step render 1 out of x frames. For example, a value of 2 here would render every second frame.

proxy scale all frames down by this factor, for faster rendering.

hiQ do high-quality rendering (defaults to true, if not specified).

mblur calculate motion-blur when rendering (defaults to true, if not specified)

Table form

The table entries should be one or more of the following:

Start First frame to render. Default: Comp's render end setting.

End Final frame to render (inclusive). Default: Comp's render end setting.

HiQ Render in HiQ. Default true.

RenderAll Render all tools, even if not required by a saver. Default false.

MotionBlur Do motion blur in render, where specified in tools. Default true.

SizeType Resizes the output:

-1	Custom (only used by PreviewSavers during a preview render)
0	Use prefs setting
1	Full Size (default)
2	Half Size
3	Third Size
4	Quarter Size

Width Width of result when doing a Custom preview (defaults to pref).

Height Height of result when doing a Custom preview (defaults to pref).

KeepAspect Maintains the frame aspect when doing a Custom preview. Defaults to Preview prefs setting.

StepRender Render only 1 out of every X frames ("shoot on X frames") or render every frame, default false.

Steps If step rendering, how many to step. Default 5.

UseNetwork Enables rendering with the network. Default false.

Groups Use these network slave groups to render on (when net rendering). Default "all".

Flags Number specifying render flags, usually 0 (the default). Most flags are specified by other means, but a value of 262144 is used for preview renders.

Tool Handle to a tool to specifically render. If this is specified only sections of the comp up to this tool will be rendered. eg you could specify comp.Saver1 to only render up to Saver1, ignoring any tools (including savers) after it. default nil.

FrameRange Describes which frames to render. (eg "1..100,150..180"), defaults to "Start".."End" (above).

Wait Whether the script command will wait for the render to complete, or return immediately, default false.

→ Python usage:

```
# Render explicit render range, wait for the render.
composition.Render(True, 1, 100, 1) # wait, specify the render range

# Renders a non-contiguous frame range, and returns once the render has completed.
comp.Render({ “FrameRange”: “1..10,20,30,40..50”, “Wait”: True })

# Render up to the Saver1 tool, but nothing further downstream.
comp.Render({“Tool”: comp.Saver1})
```

→ Lua usage:

```
-- Render explicit render range, wait for the render.
composition:Render(true, 1, 100, 1) -- wait, specify the render range

-- Renders a non-contiguous frame range, and returns once the render has completed.
comp:Render({ FrameRange = “1..10,20,30,40..50”, Wait = true })

-- Render up to the Saver1 tool, but nothing further downstream.
comp:Render({Tool = comp.Saver1})
```

- **Parameters**:

  wait (*boolean*) – wait

  start (*number*) – start

  end (*number*) – end

  proxy (*number*) – proxy

  hiq (*boolean*) – hiq

  motionblur (*boolean*) – motionblur

- **Returns**: success
- **Return type**: boolean

`Composition.Render(`*settings*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Start a render.

The Render function starts rendering the current composition. There are two forms for calling this function, one where the arguments are passed directly, and a second form where all the arguments are passed in a table. The table format is useful for declaring non-contiguos render ranges, such as the following one.

Returns true if the composition rendered successfully, nil if it failed to start or complete the render.

Arguments

wait_for_render a true or false value indicating whether the script should wait for the render to complete, or continue processing once the render has begun.

renderstart the frame to start rendering at.

renderend the frame to stop rendering at.

step render 1 out of x frames. For example, a value of 2 here would render every second frame.

proxy scale all frames down by this factor, for faster rendering.

hiQ do high-quality rendering (defaults to true, if not specified).

mblur calculate motion-blur when rendering (defaults to true, if not specified)

Table form

The table entries should be one or more of the following:

Start First frame to render. Default: Comp's render end setting.

End Final frame to render (inclusive). Default: Comp's render end setting.

HiQ Render in HiQ. Default true.

RenderAll Render all tools, even if not required by a saver. Default false.

MotionBlur Do motion blur in render, where specified in tools. Default true.

SizeType Resizes the output:

-1	Custom (only used by PreviewSavers during a preview render)
0	Use prefs setting
1	Full Size (default)
2	Half Size
3	Third Size
4	Quarter Size

Width Width of result when doing a Custom preview (defaults to pref).

Height Height of result when doing a Custom preview (defaults to pref).

KeepAspect Maintains the frame aspect when doing a Custom preview. Defaults to Preview prefs setting.

StepRender Render only 1 out of every X frames ("shoot on X frames") or render every frame, default false.

Steps If step rendering, how many to step. Default 5.

UseNetwork Enables rendering with the network. Default false.

Groups Use these network slave groups to render on (when net rendering). Default "all".

Flags Number specifying render flags, usually 0 (the default). Most flags are specified by other means, but a value of 262144 is used for preview renders.

Tool Handle to a tool to specifically render. If this is specified only sections of the comp up to this tool will be rendered. eg you could specify comp.Saver1 to only render up to Saver1, ignoring any tools (including savers) after it. default nil.

FrameRange Describes which frames to render. (eg "1..100,150..180"), defaults to "Start".."End" (above).

Wait Whether the script command will wait for the render to complete, or return immediately, default false

→ **Python usage**:

```
# Render explicit render range, wait for the render.
composition.Render(True, 1, 100, 1) # wait, specify the render range
```

```
# Renders a non-contiguous frame range, and returns once the render has completed.
comp.Render({ "FrameRange": "1..10,20,30,40..50", "Wait": True })

# Render up to the Saver1 tool, but nothing further downstream.
comp.Render({"Tool": comp.Saver1})
```

→ Lua usage:

```
-- Render explicit render range, wait for the render.
composition:Render(true, 1, 100, 1) -- wait, specify the render range

-- Renders a non-contiguous frame range, and returns once the render has completed.
comp:Render({ FrameRange = "1..10,20,30,40..50", Wait = true })

-- Render up to the Saver1 tool, but nothing further downstream.
comp:Render({Tool = comp.Saver1})
```

→ **Parameters**:

settings (*table*) – settings

→ **Returns**: success

→ **Return type**: boolean

`Composition.ReverseMapPath(`*mapped*`)`

Collapses a path into best-matching path map.

Whereas MapPath() is used to expand any Fusion path maps within a pathname to get an ordinary literal path, ReverseMapPath() will perform the opposite process, and re-insert those path maps.

This is often useful if the path is to be stored for later usage (within a comp or script, for example). It allows the path to be used with the same meaning for another system or situation, where the literal location of the path may be different.

In addition to handling all the global path maps like Fusion:ReverseMapPath(), Composition:ReverseMapPath() also handles any path maps listed in the composition's Path Maps preferences page, as well as the built-in Comp: path map (see MapPath()).

Returns a path with the Fusion path map handles re-inserted wherever possible.

- **Parameters**:

  mapped (*string*) – mapped
- **Returns**: path
- **Return type**: string

`Composition.RunScript(`*filename*`)`

Run a script within the composition's script context.

Use this function to run a script in the composition environment. This is similar to launching a script from the comp's Scripts menu.

The script will be started with 'fusion' and 'composition' variables set to the Fusion and currently active Composition objects. The filename given may be fully specified, or may be relative to the comp's Scripts: path.

Fusion supports .py .py2 and .py3 extensions to differentiate python script versions.

- **Parameters**:

  filename (*string*) – filename

`Composition.Save(`*filename*`)`

Save the composition

This function causes the composition to be saved to disk. The compname argument must specify a path relative to the filesystem of the Fusion which is saving the composition. In other words - if system 'a' is using the Save() function to instruct a Fusion on system 'b' to save a composition, the path provided must be valid from the perspective of system 'b'.

filename is the complete path and name of the composition to be saved.

- **Parameters**:

  filename (*string*) – filename
- **Returns**: success
- **Return type**: boolean

`Composition.SaveAs()`

Prompt user with a Save As dialog box to save the composition.

`Composition.SaveCopyAs()`

Prompt user with a Save As dialog box to save the composition as copy.

`Composition.SetActiveTool(`*tool*`)`

Set the currently active tool.

This function will set the currently active tool to one specified by script. It can be read with ActiveTool.

To deselect all tools, omit the parameter or use nil.

Note that ActiveTool also means the tool is selected, while selected tools are not automativally Active. Only one tool can be Active at a time. To select tools use FlowView:Select().

→ Parameters:

tool (*Tool*) – tool

`Composition.SetData(`*name*, *value*`)`

Set custom persistent data.

name name of the data. This name can be in "table.subtable" format, to allow persistent data to be stored within subtables.

value to be recorded in the object's persistent data. It can be of almost any type.

Persistent data is a very useful way to store names, dates, filenames, notes, flags, or anything else, in such a way that they are permanently associated with this instance of the object, and are stored along with the object using SetData(), and can be retrieved at any time with GetData().

The method of storage varies by object: SetData() called on the Fusion app itself will save its data in the Fusion.prefs file, and will be available whenever that copy of Fusion is running. Calling SetData() on any object associated with a Composition will cause the data to be saved in the .comp file, or in any settings files that may be saved directly from that object. Some ephemeral objects that are not associated with any composition and are not otherwise saved in any way, may not have their data permanently stored at all, and the data will only persist as long as the object itself does.

→ Python usage:

```
from pprint import pprint
from datetime import datetime
tool = comp.ActiveTool
tool.SetData(“Modified.Author”, fusion.GetEnv(“USERNAME”))
tool.SetData(“Modified.Date”, str(datetime.now()))
pprint(tool.GetData(“Modified”))
```

→ Lua usage:

```
tool:SetData(“Modified.Author”, fusion:GetEnv(“USERNAME”))
tool:SetData(“Modified.Date”, os.date())
dump(tool:GetData(“Modified”))
```

→ Parameters:

name (*string*) – name

value ((*number|string|boolean|table*)) – value

`Composition.SetPrefs(`*prefname*`, `*val*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Set preferences from a table of attributes.

The SetPrefs function can be used to specify the values of virtually all preferences in Fusion. Its can take a table of values, identified by name, or a single name and value.

The table provided as an argument should have the format [prefs_name] = value. Subtables are allowed.

It is possible to set a preference that does not exist. For example, setting fusion:SetPrefs ({Comp.FrameFormat.Stuff = "Bob"}) will create a new preference which will be thereafter preserved in the Fusion preferences file.

Returns false if any of the arguments provided to it are invalid, and true otherwise. Note that the function will still return true if an attempt is made to set a preference to an invalid value. For example, attempting to setting the FPS to "Bob" will fail, but the function will still return true.

→ Python usage:

```
comp.SetPrefs({ "Comp.Transport.FrameStep":5, "Comp.FrameFormat.AspectX":2 })

comp.SetPrefs("Comp.Interactive.BackgroundRender", True)
```

→ Lua usage:

```
comp:SetPrefs({ ["Comp.Unsorted.GlobalStart"]=0, ["Comp.Unsorted.GlobalEnd"]=100 })

comp:SetPref("Comp.Interactive.BackgroundRender", true)
```

→ Parameters:

prefname (*string*) – prefname

val (*value*) – val

`Composition.SetPrefs(`*prefs*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Set preferences from a table of attributes.

The SetPrefs function can be used to specify the values of virtually all preferences in Fusion. Its can take a table of values, identified by name, or a single name and value.

The table provided as an argument should have the format [prefs_name] = value. Subtables are allowed.

It is possible to set a preference that does not exist. For example, setting fusion: SetPrefs({Comp.FrameFormat.Stuff = "Bob"}) will create a new preference which will be thereafter preserved in the Fusion preferences file.

Returns false if any of the arguments provided to it are invalid, and true otherwise.

Note that the function will still return true if an attempt is made to set a preference to an invalid value. For example, attempting to setting the FPS to "Bob" will fail, but the function will still return true.

→ Python usage:

```
comp.SetPrefs({ "Comp.Transport.FrameStep":5, "Comp.FrameFormat.AspectX":2 })

comp.SetPrefs("Comp.Interactive.BackgroundRender", True)
```

→ Lua usage:

```
comp:SetPrefs({ ["Comp.Unsorted.GlobalStart"]=0, ["Comp.Unsorted.GlobalEnd"]=100 })

comp:SetPref("Comp.Interactive.BackgroundRender", true)
```

→ Parameters:

prefs (*table*) – prefs

`Composition.StartUndo(`*name*`)`

Start an undo event.

The StartUndo() function is always paired with an EndUndo() function. Any changes made to the composition by the lines of script between StartUndo() and EndUndo() are stored as a single Undo event.

Changes captured in the undo event can be undone from the GUI using CTRL-Z, or the Edit menu. They can also be undone from script, by calling the Undo function.

Should be used sparingly, as the user (or script) will have no way to undo the preceding commands.

name specifies the name displayed in the Edit/Undo menu of the Fusion GUI a string containing the complete path and name of the composition to be saved.

Actual changes must be made to the composition (forcing a "dirty" event) before the undo will be added to the stack.

→ Python usage:

```
composition.StartUndo("Add some tools")
bg1 = comp.Background()
pl1 = comp.Plasma()
mg1 = comp.Merge({ "Background": bg1, "Foreground": pl1 })
composition.EndUndo(True)
```

→ Lua usage:

```
composition:StartUndo("Add some tools")
bg1 = Background{}
pl1 = Plasma{}
mg1 = Merge{ Background = bg1, Foreground = pl1 }
composition:EndUndo(true)
```

→ Parameters:

name (*string*) – name

`Composition.Stop()`

Stops interactive playback.

Use this function in the same way that you would use the Stop button in the composition's playback controls.

`Composition.Undo(`*count*`)`

Undo one or more changes to the composition.

The Undo function triggers an undo event in Fusion. The count argument determines how many undo events are triggered.

Note that the value of count can be negative, in which case Undo will behave as a Redo, acting exactly as the Redo() function does.

count specifies how many undo events to trigger.

→ Parameters:

count (*number*) – count

`Composition.Unlock()`

Unlock the composition.

The Unlock() function returns a composition to interactive mode. This allows Fusion to show dialog boxes to the user, and allows re-rendering in response to changes to the controls. Calling Unlock() will have no effect unless the composition has been locked with the Lock() function first.

It is often useful to surround a script with Lock() and Unlock(), especially when adding tools or modifying a composition. Doing this ensures Fusion won't pop up a dialog to ask for user input, e.g. when adding a Loader, and can also speed up the operation of the script since no time will be spent rendering until the comp is unlocked.

→ Python usage:

```
comp.Lock()
# Will not open the file dialog, since the composition is locked
my_loader = comp.Loader()
comp.Unlock()
```

→ Lua usage:

```
comp:Lock()
-- Will not open the file dialog, since the composition is locked
my_loader = Loader()
comp:Unlock()
```

`Composition.UpdateViews()`

UpdateViews

FloatViewFrame

`class FloatViewFrame`

Parent class: `FuFrame`

Methods

`FloatViewFrame.ActivateFrame()`

Activates this frame window.

`FloatViewFrame.ActivateNextFrame()`

Activates the next frame window.

`FloatViewFrame.ActivatePrevFrame()`

Activates the previous frame window.

FlowView

`class FlowView`

Parent class: `FuView`

The FlowView represents the flow with all the tools.

Positions of tools, their selection state and the views zoom level are controlled with this object.

→ Python usage:

```
# Get the current FlowView
flow = composition.CurrentFrame.FlowView
```

→ Lua usage:

```
-- Get the current FlowView
flow = composition.CurrentFrame.FlowView
```

Methods

`FlowView.FlushSetPosQueue()`

Moves all tools queued for positioning with QueueSetPos.

`FlowView.FrameAll()`

Rescale and reposition the FlowView to contain all tools.

`FlowView.GetPos()`

Returns the position of a tool.

This function returns two numeric values containing the X and Y co-ordinates of the tool. In Python use GetPosTable instead.

- **Python usage**:

```
flow = comp.CurrentFrame.FlowView

x, y = flow.GetPosTable(comp.Background1).values()
```

- **Lua usage**:

```
flow = comp.CurrentFrame.FlowView

x, y = flow:GetPos(tool)
```

- **Returns**: x
- **Return type**: number

`FlowView.GetPosTable(`*tool*`)`

Returns the position of a tool as a table.

Use this in Python to get the X and Y value.

- **Python usage**:

```
flow = comp.CurrentFrame.FlowView
x, y = flow.GetPosTable(comp.Background1).values()
```

- **Lua usage**:

```
flow = comp.CurrentFrame.FlowView
x, y = flow:GetPos(tool)
```

- **Parameters**:

tool (*object*) – tool

- **Returns**: pos
- **Return type**: table

`FlowView.GetScale()`

Returns the current scale of the contents.

This function returns a numeric value indicating the current scale of the FlowView. 1 means 100%, while 0.1 means 10% of the default scale.

- **Returns**: scale
- **Return type**: number

`FlowView.QueueSetPos(`*tool, x, y*`)`

Queues the moving of a tool to a new position.

All queued moves will be evaluated once FlushSetPosQueue() has been called.

→ Parameters:

tool (*object*) – tool

x (*number*) – x

y (*number*) – y

`FlowView.Select(`*tool*`[, `*select*`])`

Selects or deselects a tool.

This function will add or remove the tool specified in it's first argument from the current tool selection set. The second argument should be set to false to remove the tool from the selection, or to true to add it.

tool should contain the tool that will be selected or deselected in the FlowView.

select setting this to false will deselect the tool specified in the first argument. Otherwise the default value of true is used, which selects the tool.

If called with no arguments, the function will clear all tools from the current selection.

→ Parameters:

tool (*object*) – tool

select (*boolean*) – select

`FlowView.SetPos(`*tool, x, y*`)`

Moves a tool to a new position.

→ Python usage:

```
# Align all selected tools to x co-ordinate of the ActiveTool
flow = comp.CurrentFrame.FlowView
x, y = flow.GetPosTable(comp.ActiveTool)
for t in comp.GetToolList(True).values():
    cur_x, cur_y = flow.GetPosTable(t)
    flow.SetPos(t, x, cur_y)
```

→ Lua usage:

```
-- Align all selected tools to x co-ordinate of the ActiveTool
local flow = comp.CurrentFrame.FlowView
local x, y = flow:GetPos(comp.ActiveTool)
```

```
for i, t in pairs(comp:GetToolList(true)) do
    cur_x, cur_y = flow:GetPos(t)
    flow:SetPos(t, x, cur_y)
end
```

→ **Parameters**:

tool (*object*) – tool

x (*number*) – x

y (*number*) – y

`FlowView.SetScale(`*scale*`)`

Change the scale of the contents.

This function rescales the FlowView to the amount specified. A value of 1 for the scale argument would set the FlowView to 100%, while a value of 0.1 would set it to 10% of the default scale.

→ **Parameters**:

scale (*number*) – scale

FontList

`class FontList`

Parent class: `List`

Methods

`FontList.AddFont(`*fontfile*`)`

Adds the specified font to the global font list.

→ **Parameters**:

fontfile (*string*) – fontfile

→ **Returns**: success

→ **Return type**: boolean

`FontList.Clear()`

Empties the global font list.

`FontList.GetFontList()`

Returns all font files in the global font list.

→ **Returns**: fonts

→ **Return type**: table

`FontList.ScanDir([dirname])`

Adds the specified dir to the global font list.

→ **Parameters**:

dirname (*string*) – dirname

FuFrame

`class FuFrame`

Parent class: `Object`

Members

`FuFrame.Composition`

Represents this frame window's Composition (read-only).

→ **Setting**:

FuFrame.Composition = comp – (Composition)

`FuFrame.ConsoleView`

Represents this frame window's console (read-only).

→ **Setting**:

FuFrame.ConsoleView = view – (FuView)

`FuFrame.CurrentView`

Represents the currently active view for this frame window (read-only).

→ **Setting**:

FuFrame.CurrentView = view – (FuView)

`FuFrame.FlowView`

Represents this frame window's Flow view (read-only).

→ **Setting**:

FuFrame.FlowView = view – (FuView)

`FuFrame.InfoView`

Represents this frame window's Info view (read-only).

→ Setting:

FuFrame.InfoView = view – (FuView)

`FuFrame.LeftView`

Represents this frame window's left display view (read-only).

→ Setting:

FuFrame.LeftView = view – (GLView)

`FuFrame.ModifierView`

Represents this frame window's Modifier controls view (read-only).

→ Setting:

FuFrame.ModifierView = view – (FuView)

`FuFrame.RightView`

Represents this frame window's right display view (read-only).

→ Setting:

FuFrame.RightView = view – (GLView)

`FuFrame.SplineView`

Represents this frame window's spline editor view (read-only).

→ Setting:

FuFrame.SplineView = view – (FuView)

`FuFrame.TimeRulerView`

Represents this frame window's time ruler (read-only).

→ Setting:

FuFrame.TimeRulerView = view – (FuView)

`FuFrame.TimelineView`

Represents this frame window's Timeline view (read-only).

→ Setting:

FuFrame.TimelineView = view – (FuView)

`FuFrame.ToolView`

Represents this frame window's Tool controls view (read-only).

→ Setting:

FuFrame.ToolView = view – (FuView)

`FuFrame.TransportView`

Represents this frame window's transport controls view (read-only).

→ **Setting**:

FuFrame.TransportView = view – (FuView)

Methods

`FuFrame.GetPreviewList([`*include_globals*`])`

Retrieves a table of previews.

→ **Parameters**:

include_globals (*boolean*) – include_globals

→ **Returns**: previews

→ **Return type**: table

`FuFrame.GetViewList()`

Returns the list of views within this frame.

→ **Returns**: views

→ **Return type**: table

`FuFrame.SwitchView(id)`

Displays a given view within this frame.

→ **Parameters**:

id (*string*) – id

`FuFrame.ViewOn([`*tool*`][, `*view*`])`

Displays a tool on a numbered view.

→ **Python usage**:

```
comp.CurrentFrame.ViewOn(tool, 1)
```

→ **Lua usage**:

```
comp.CurrentFrame:ViewOn(tool, 1)
```

→ **Parameters**:

tool (*Tool*) – tool

view (*number*) – view

Fusion

`class Fusion`

Parent class: `Object`

Handle to the application.

Fusion Attributes

Attribute Name	Type	Description
FUSIONS_FileName	string	The path to the Fusion.exe file.
FUSIONS_Version	string	The version of FUSION that we are connected to, in either string (FUSION_Version) or numeric (FUSIONI_VersionHi, FUSIONI_VersionLo) format.
FUSIONI_SerialHi FUSIONI_SerialLo	integer	The serial number of the Fusion license that we are connected to.
FUSIONS_MachineType	string	The type (OS and CPU) of machine.
FUSIONI_NumProcessors	integer	The number of processors present in the machine running Fusion.
FUSIONB_IsManager	boolean	Indicates if this Fusion is currently a render master.
FUSIONI_MemoryLoad	integer	The current Memory load percentage of the machine, from 0 to 100.
FUSIONI_PhysicalRAMTotalMB	integer	The total amount of physical RAM, in MB.
FUSIONI_PhysicalRAMFreeMB	integer	The amount of physical RAM free, in MB.
FUSIONI_VirtualRAMTotalMB	integer	The total amount of virtual RAM, in MB.
FUSIONI_VirtualRAMUsedMB	integer	The total amount of virtual RAM in use, in MB.
FUSIONB_IsPost	boolean	Indicates if this Fusion is a Post license.
FUSIONB_IsDemo	boolean	Indicates if this Fusion is a Demo license.
FUSIONB_IsRenderNode	boolean	Indicates if this Fusion is a Render Node license.
FUSIONH_CurrentComp	handle	Returns a handle to the current composition that has the focus in Fusion.
FUSIONI_VersionHi FUSIONI_VersionLo	integer	

- Python usage:

```
# Get basic connection to fusion.
fu = bmd.scriptapp(“Fusion”)
```

- Lua usage:

```
-- Get basic connection to fusion.
fu = fu or Fusion()
```

Members

`Fusion.Bins`

Bins (read-only).

- Getting:

bins = Fusion.Bins – (Bins)

`Fusion.Build`

Returns the build number of the current Fusion instance.

- Getting:

build = Fusion.Build – (number)

`Fusion.CacheManager`

The Global Cache Manager (read-only).

- Getting:

cm = Fusion.CacheManager – (CacheManager)

`Fusion.CurrentComp`

Represents the currently active composition (read-only).

- Getting:

comp = Fusion.CurrentComp – (Composition)

`Fusion.FileLogging()`

Are Fusion logs enabled.

Returns true if Fusion was started with a /log filepath argument.

`Fusion.FontManager`

The Global Font Manager (read-only).

- Getting:

fm = Fusion.FontManager – (FontList)

`Fusion.HotkeyManager`

The Global Hotkey Manager (read-only).

→ Getting:

hkm = Fusion.HotkeyManager – (HotkeyManager)

`Fusion.MenuManager`

The Global Menu Manager (read-only).

→ Getting:

mm = Fusion.MenuManager – (MenuManager)

`Fusion.QueueManager`

The global render manager for this instance of Fusion (read-only).

→ Getting:

qm = Fusion.QueueManager – (QueueManager)

`Fusion.RenderManager`

The global render manager for this instance of Fusion (read-only).

→ Getting:

qm = Fusion.RenderManager – (QueueManager)

`Fusion.Version`

Returns the version of the current Fusion instance.

→ Getting:

ver = Fusion.Version – (number)

Methods

`Fusion.AllowNetwork()`

AllowNetwork

`Fusion.ClearFileLog()`

Clears the log if started with the /log argument.

`Fusion.CreateFloatingView()`

Creates a new FloatView.

`Fusion.CreateMail()`

Returns an object handle that can be manipulated with other mail related functions.

Within Fusion there are a number of scripts that can be used to send information to people through email. This could be utilized to notify a user when their render is complete, or if any errors have occurred with a render.

- Python usage:

```
mail = fusion.CreateMail()

mail.AddRecipients(“vfx@studio.com, myself@studio.com”)
mail.SetSubject(“Render Completed”)
mail.SetBody(“The job completed.”)

ok,errmsg = mail.SendTable().values()
print(ok)
print(errmsg)
```

- Lua usage:

```
mail = fusion:CreateMail()

mail:AddRecipients(“vfx@studio.com, myself@studio.com”)
mail:SetSubject(“Render Completed”)
mail:SetBody(“The job completed.”)

ok,errmsg = mail:Send()
print(ok)
print(errmsg)
```

- Returns: mail
- Return type: `MailMessage`

`Fusion.DumpCgObjects`(*filename*)

Writes the state of all current Cg shaders to the given file.

- Parameters:

  filename (*string*) – filename

- **Returns**: success
- **Return type**: boolean

`Fusion.DumpGLObjects(`*filename*`)`

Writes the state of all current OpenGL objects to the given file.

- **Parameters**:

  filename (*string*) – filename

- **Returns**: success
- **Return type**: boolean

`Fusion.DumpGraphicsHardwareInfo(`*filename*`)`

Writes the information of the graphics hardware to the given file.

- **Parameters**:

  filename (*string*) – filename

- **Returns**: success
- **Return type**: boolean

`Fusion.DumpOpenCLDeviceInfo(`*filename*`)`

Writes the information of the OpenCL device to the given file.

- **Parameters**:

  filename (*string*) – filename

- **Returns**: success
- **Return type**: boolean

`Fusion.Execute()`

Executes a script string for the fusion instance.

See Composition:Execute().

`Fusion.FindReg(`*id*`[, `*type*`])`

Finds a registry object by name.

An optional type restricts the search. Some valid type constants include

- CT_Tool
- CT_Filter
- CT_FilterSource
- CT_ParticleTool
- CT_ImageFormat

Returns nil / None if no match is found.

- **Python usage**:

```
from pprint import pprint
reg = fusion.FindReg("Loader")
pprint(reg.GetAttrs())
Lua usage:
```

- **Lua usage**:

```
reg = fusion:FindReg("Loader")
dump(reg:GetAttrs())
```

- **Parameters**:

  id (*string*) – id

  type (*number*) – type

- **Returns**: reg
- **Return type**: `Registry`

`Fusion.GetAppInfo()`

Returns a table containing information about the current application's name, executable, version, and build number.

`Fusion.GetArgs()`

Get command line arguments.

Returns Fusion's command line arguments as a table.

- **Returns**: args
- **Return type**: table

`Fusion.GetCPULoad()`

Retrieves the current CPU load of the system.

Returns the current CPU load as a percentage between 0 and 100.

`Fusion.GetClipboard()`

Retrieves the tool(s) on the clipboard, as tables and as ASCII text.

Returns a string or table of the current contents of the clipboard, or nil if empty.

- **Returns**: cliptbl
- **Return type**: table

`Fusion.GetCompList()`

Retrieves a table of all compositions currently present.

- **Returns**: complist
- **Return type**: table

`Fusion.GetCurrentComp()`

Returns the currently active composition.

- **Returns**: comp
- **Return type**: `Composition`

`Fusion.GetData([`*name*`])`

Get custom persistent data.

See Composition:GetData().

- **Parameters**:

  name (*string*) – name

- **Returns:** value
- **Return type**: (number|string|boolean|table)

`Fusion.GetEnv(`*name*`)`

Retrieve the value of an environment variable.

Returns the value of an environment variable on the machine running Fusion. This function is identical to the global os.getenv() function, except that it runs in the context of the Fusion instance, so if the Fusion instance points to a remote copy of Fusion the environment variable will come from the remote machine.

- **Parameters**:

  name (*string*) – name

- **Returns**: value
- **Return type:** string

`Fusion.GetGlobalPathMap([`*built_ins*`][, `*defaults*`])`

Returns a table of all global path maps.

- **Parameters**:

  built_ins (*boolean*) – built_ins

  defaults (*boolean*) – defaults

- **Returns**: map
- **Return type:** table

`Fusion.GetMainWindow()`

Get the window handle for fusion.

`Fusion.GetPrefs([`*prefname*`][, `*exclude-defaults*`])`

Retrieve a table of preferences.

This function is useful for getting the full table of global preferences, or a subtable, or a specific value.

If the argument is omitted all preferences will be returned.

Returns a table of preferences, or a specific preference value.

→ **Python usage**:

```
from pprint import pprint
pprint(fusion.GetPrefs(“Global.Paths.Map”))
print(fusion.GetPrefs(“Global.Controls.GrabDistance”))
```

→ **Lua usage**:

```
dump(fusion:GetPrefs(“Global.Paths.Map”))
print(fusion:GetPrefs(“Global.Controls.GrabDistance”))
```

→ **Parameters**:

prefname (*string*) – prefname

exclude-defaults (*boolean*) – exclude-defaults

→ **Returns**: prefs

→ **Return type**: table

`Fusion.GetPreviewList()`

Retrieves a table of global previews.

This function returns a list of preview objects currently available to the Fusion object. The Composition:GetPreviewList function is similar, but it will not return floating views, like this function does.

→ **Returns**: previewlist

→ **Return type**: table

`Fusion.GetRegAttrs(`*id*`[, `*type*`])`

Retrieve information about a registry ID.

The GetRegAttrs() function will return a table with the attributes of a specific individual registry entry in Fusion. The only argument is the ID, a unique numeric identifier possessed by each entry in the registry. The ID identifiers for each registry item can be obtained from fusion:GetRegList(), fusion:FindRegID(), and tool:GetID() functions.

Registry attributes are strictly read only, and cannot be modified in any way.

- **Python usage**:

```
from pprint import pprint

# Dump RegAttrs for the Active tool,
# or prints message if nothing is Active.
pprint(comp.ActiveTool and
    fusion.GetRegAttrs(comp.ActiveTool.ID) or
    “Please set an ActiveTool first.”)
```

- **Lua usage**:

```
-- Dump RegAttrs for the Active tool,
-- or prints message if nothing is Active.
dump(comp.ActiveTool and
    fusion:GetRegAttrs(comp.ActiveTool.ID) or
    “Please set an ActiveTool first.”)
```

- **Parameters**:

  id (*string*) – id

  type (*number*) – type

- **Returns**: attrs
- **Return type**: table

`Fusion.GetRegList(`*typemask*`)`

Retrieve a list of all registry objects known to the system.

The Fusion registry stores information about the configuration and capabilities of a particular installation of Fusion. Details like which file formats are supported, and which tools are installed are found in the registry. Note that this is NOT the same thing as the operating system registry, the registry accessed by this function is unique to Fusion.

The only argument accepted by GetRegAttrs is a mask constant, which is used to filter the registry for specific registry types. The constants represent a particular type of registry entry, for example CT_Any will return all entries in the registry, while CT_Source will only return entries describing tools from the source category of tools (Loader, Plasma, Text...). A complete list of valid constants can be found here.

Returns a table, which contains a list of the Numeric ID values for each registry entry. The numeric ID is constant from machine to machine, e.g. the numeric ID for the QuickTime format would be 1297371438, regardless of the installation or version of Fusion.

These ID's are used as arguments to the GetRegAttrs() function, which provides access to the actual values stored in the specific registry setting.

typemask a predefined constant that determines the type of registry entry returned by the function.

Some valid Mask constants:

CT_Tool all tools

CT_Mask mask tools only

CT_SourceTool creator tools (images/3D/particles) all
of which don't require an input image

CT_ParticleTool Particle tools

CT_Modifier Modifiers

CT_ImageFormat ImageFormats

CT_View Different sections of the interface

CT_GLViewer All kinds of viewers

CT_PreviewControl PreviewControls in the viewer

CT_InputControl Input controls

CT_BinItem Bin items

→ **Python usage**:

```
from pprint import pprint

# this example will print out all of the
# image formats supported by this copy
# of Fusion

```

```
reg = fusion.GetRegList(comp.CT_ImageFormat)
reg["Attrs"] = {}

for i in range(1, len(reg)):
    reg["Attrs"][i] = fusion.GetRegAttrs(reg[i].ID)
    name = reg["Attrs"][i]["REGS_MediaFormat_FormatName"]

    if name == None:
        name = reg["Attrs"][i]["REGS_Name"]

    if reg["Attrs"][i]["REGB_MediaFormat_CanSave"] == True:
        print(name)
    else:
        print(name + " (Cannot Save)")
```

→ Lua usage:

```
-- this example will print out all of the
-- image formats supported by this copy
-- of Fusion

reg = fusion:GetRegList(CT_ImageFormat)
reg.Attrs = {}

for i = 1, #reg do
        reg.Attrs[i] = fusion:GetRegAttrs(reg[i].ID)
        name = reg.Attrs[i].REGS_MediaFormat_FormatName

        if name == nil then
            name = reg.Attrs[i].REGS_Name
        end
```

```
    --dump(reg.Attrs[i])
        if reg.Attrs[i].REGB_MediaFormat_CanSave == true then
            print(name)
        else
            print(name .. " (Cannot Save)")
        end
end
```

- **Parameters**:

  typemask (*number*) – typemask

- **Returns**: reglist
- **Return type**: table

`Fusion.GetRegSummary(`*typemask*`[, `*hidden*`])`

Retrieve a list of basic info for all registry objects known to the system.

This function is useful for getting the full table of global preferences, or a subtable, or a specific value.

Returns a table containing a summary of the Name, ID, ClassType, and OpIconString of every item in the registry. Useful for returning a lightweight version of the information presented by Fusion:GetRegList.

- **Parameters**:

  typemask (*number*) – typemask

  hidden (*boolean*) – hidden

- **Returns**: regattrs
- **Return type**: table

`Fusion.LoadComp(`*filename*`[, `*quiet*`][, `*autoclose*`][, `*hidden*`])`

Note: This method is overloaded and has alternative parameters. See other definitions.

Loads an existing composition.

auto-close a true or false value to determine if the composition will close automatically when the script exits. Defaults to false.

hidden if this value is true, the comp will be created invisibly, and no UI will be available to the user. Defaults to false.

Returns a handle to the opened composition.

- **Parameters**:
  - **filename** (*string*) – filename
  - **quiet** (*boolean*) – quiet
  - **autoclose** (*boolean*) – autoclose
  - **hidden** (*boolean*) – hidden
- **Returns**: comp
- **Return type**: `Composition`

`Fusion.LoadComp(`*filename*`,` *options*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Loads an existing composition.

auto-close a true or false value to determine if the composition will close automatically when the script exits. Defaults to false.

hidden if this value is true, the comp will be created invisibly, and no UI will be available to the user. Defaults to false.

Returns a handle to the opened composition.

- **Parameters**:
  - **filename** (*string*) – filename
  - **options** (*table*) – options
- **Returns**: comp
- **Return type**: `Composition`

`Fusion.LoadComp(`*savedcomp*`,` *options*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Loads an existing composition.

auto-close a true or false value to determine if the composition will close automatically when the script exits. Defaults to false.

hidden if this value is true, the comp will be created invisibly, and no UI will be available to the user. Defaults to false.

Returns a handle to the opened composition.

- **Parameters**:
  - **savedcomp** (*MemBlock*) – savedcomp
  - **options** (*table*) – options
- **Returns**: comp
- **Return type**: `Composition`

`Fusion.LoadPrefs([`*filename*`][, `*mastername*`])`

Reloads all current global preferences.

Reloads all global preferences from the specified file and (optionally) an overriding master prefs file.

→ **Parameters**:

filename (*string*) – filename

mastername (*string*) – mastername

→ **Returns**: success

→ **Return type**: boolean

`Fusion.LoadRecentComp(`*index*`[, `*quiet*`][, `*autoclose*`][, `*hidden*`])`

Loads an composition from the recent file list.

index the most recent composition is 1. The index is the same as in the Recent Files menu.

auto-close a true or false value to determine if the composition will close automatically when the script exits. Defaults to false.

hidden if this value is true, the comp will be created invisibly, and no UI will be available to the user. Defaults to false.

→ **Parameters**:

index (*integer*) – index

quiet (*boolean*) – quiet

autoclose (*boolean*) – autoclose

hidden (*boolean*) – hidden

→ **Returns**: comp

→ **Return type**: `Composition`

`Fusion.MapPath(`*path*`)`

Expands path mappings in a path string.

See Comp:MapPath().

→ **Python usage:**

```
print(comp.MapPath("Fusion:"))
```

→ **Lua usage**:

```
print(MapPath("Fusion:"))
```

- **Parameters**:

  path (*string*) – path

- **Returns**: mapped
- **Return type**: string

`Fusion.MapPathSegments(`*path*`)`

Expands all path mappings in a multipath.

See Comp:MapPathSegments().

- **Parameters**:

  path (*string*) – path

- **Returns**: mapped
- **Return type**: table

`Fusion.NewComp([`*quiet*`][, `*autoclose*`][, `*hidden*`])`

Creates a new composition.

auto-close a true or false value to determine if the composition will close automatically when the script exits. Defaults to false.

hidden if this value is true, the comp will be created invisibly, and no UI will be available to the user. Defaults to false.

- **Parameters**:

  quiet (*boolean*) – quiet

  autoclose (*boolean*) – autoclose

  hidden (*boolean*) – hidden

- **Returns**: comp
- **Return type**: `Composition`

`Fusion.OpenFile(`*filename*`, `*mode*`)`

Open a file.

filename specifies the full path and name of the file to open

mode specifies the mode(s) of file access required, from a combination of the following constants:

FILE_MODE_READ Read access **FILE_MODE_WRITE** Write access **FILE_MODE_UNBUFFERED** Unbuffered access **FILE_MODE_SHARED** Shared access

Returns a file object or nil if the open fails.

- **Lua usage**:

```
fusion:OpenFile([[c:\\fusion.log]], FILE_MODE_READ)


line = f:ReadLine()
while line do
    print(line)
    line = f:ReadLine()
end
```

- **Parameters**:

  filename (*string*) – filename

  mode (*number*) – mode
- **Returns**: file
- **Return type**: File

`Fusion.OpenLibrary()`

OpenLibrary

`Fusion.QueueComp(`*filename*`[, `*start*`][, `*end*`][, `*group*`])`

Note: This method is overloaded and has alternative parameters. See other definitions.

Queue a composition to be rendered locally.

The QueueComp function submits a composition from disk to the render manager. If the render start and end are not provided then the render manager will render the range saved with the composition. Otherwise these arguments will override the saved range.

Returns true if it succeeds in adding the composition to the Queue, and false if it fails.

filename a string describing the full path to the composition which is to be queued.

start a number which describes the first frame in the render range.

end a number which describes the last frame in the render range.

group specifies the slave group to use for this job.

Table form

Specifies the slave group to use for this job. The following keys are valid:

FileName The Comp to queue QueuedBy Who queued this comp Groups Slave groups to render on Start Render Start End Render End FrameRange Frame range string, used in place of start/end above RenderStep Render Step ProxyScale Proxy Scale to render at TimeOut Frame timeout

→ Python usage:

```
# QueueComp with additional options
fusion.QueueComp({
"FileName": "c:\\example.comp",
"QueuedBy": "Bob Lloblaw",
"Start": 1,
"End": 25,
"Step": 5,
"ProxyScale": 2
})

# Specify a non-sequential frame range
fusion.QueueComp({
    "FileName": "c:\\example.comp",
    "FrameRange": "1..10,20,30,40..50"
})
```

→ Lua usage:

```
-- QueueComp with additional options
fusion:QueueComp({
  FileName = [[c:\example.comp]],
  QueuedBy = "Bob Lloblaw",
  Start     = 1,
  End       = 25,
  Step      = 5,
  ProxyScale = 2
})
```

```
-- Specify a non-sequential frame range
fusion:QueueComp({
    FileName=[[c:\example.comp]],
    FrameRange = "1..10,20,30,40..50"
})
```

- Parameters:
  lename (*string*) – filename
  start (*number*) – start
  end (*number*) – end
  group (*string*) – group
- **Returns**: job
- **Return type**: `RenderJob`

`Fusion.QueueComp(args)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Queue a composition to be rendered locally.

The QueueComp function submits a composition from disk to the render manager. If the render start and end are not provided then the render manager will render the range saved with the composition. Otherwise these arguments will override the saved range.

Returns true if it succeeds in adding the composition to the Queue, and false if it fails.

filename a string describing the full path to the composition which is to be queued.

start a number which describes the first frame in the render range.

end a number which describes the last frame in the render range.

group specifies the slave group to use for this job.

Table form

Specifies the slave group to use for this job. The following keys are valid:

FileName The Comp to queue **QueuedBy** Who queued this comp **Groups** Slave groups to render on **Start** Render Start **End** Render End **FrameRange** Frame range string, used in place of start/end above **RenderStep** Render Step **ProxyScale** Proxy Scale to render at **TimeOut** Frame timeout

→ Python usage:

```
# QueueComp with additional options
fusion.QueueComp({
    “FileName”: “c:\\example.comp”,
    “QueuedBy”: “Bob Lloblaw”,
    “Start”: 1,
    “End”: 25,
    “Step”: 5,
    “ProxyScale”: 2
})

# Specify a non-sequential frame range
fusion.QueueComp({
    “FileName”: “c:\\example.comp”,
    “FrameRange”: “1..10,20,30,40..50”
})
```

→ Lua usage:

```
-- QueueComp with additional options
fusion:QueueComp({
  FileName = [[c:\example.comp]],
  QueuedBy = “Bob Lloblaw”,
  Start     = 1,
  End       = 25,
  Step      = 5,
  ProxyScale = 2
})
```

```
-- Specify a non-sequential frame range
fusion:QueueComp({
    FileName=[[c:\example.comp]],
    FrameRange = "1..10,20,30,40..50"
})
```

- **Parameters**:
  args (*table*) – args
- **Returns**: job
- **Return type**: `RenderJob`

`Fusion.Quit(`*exitcode*`)`

Quit Fusion.

The Quit command will cause the copy of Fusion referenced by the Fusion instance object to exit. The Fusion instance object will then be set to nil.

- **Parameters**:
  exitcode (*number*) – exitcode

`Fusion.ReverseMapPath(`*mapped*`)`

Collapses a path into best-matching path map.

See Composition:ReverseMapPath().

- **Parameters**:
  mapped (*string*) – mapped
- **Returns**: path
- **Return type**: string

`Fusion.RunScript(`*filename*`)`

Run a script within the Fusion's script context.

See Composition:RunScript().

- **Parameters**:
  filename (*string*) – filename

`Fusion.SavePrefs([`*filename*`])`

Saves all current global preferences.

- Python usage:

```
fusion.SetPrefs("Comp.AutoSave.Enabled", True)
fusion.SavePrefs()
```

- Lua usage:

```
fusion:SetPrefs("Comp.AutoSave.Enabled", true)
fusion:SavePrefs()
```

- **Parameters**:

  filename (*string*) – filename

`Fusion.SetBatch()`

SetBatch

`Fusion.SetClipboard()`

Note: This method is overloaded and has alternative parameters. See other definitions.

Sets the clipboard to contain the tool(s) specifed by a table or as ASCII text.

Sets the system clipboard to contain the ASCII for tool(s) specifed by a table or sets the clipboard to the text specified.

- **Returns**: success
- **Return type**: boolean

`Fusion.SetClipboard()`

Note: This method is overloaded and has alternative parameters. See other definitions.

Sets the clipboard to contain the tool(s) specifed by a table or as ASCII text.

Sets the system clipboard to contain the ASCII for tool(s) specifed by a table or sets the clipboard to the text specified.

- **Returns**: success
- **Return type**: boolean

`Fusion.SetData(`*name*`,` *value*`)`

Set custom persistent data.

See Composition:SetData().

- **Parameters**:

  name (*string*) – name

  value ((*number|string|boolean|table*)) – value

`Fusion.SetPrefs(`*prefname*`,` *val*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Set preferences from a table of attributes.

The SetPrefs function can be used to specify the values of virtually all preferences in Fusion. Its can take a table of values, identified by name, or a single name and value.

The table provided as an argument should have the format [prefs_name] = value. Subtables are allowed.

→ Python usage:

```
fusion.SetPrefs({
    "Global.Network.Mail.OnJobFailure": True,
    "Global.Network.Mail.Recipients": "admin@studio.com"
})

fusion.SetPrefs("Global.Controls.AutoClose", False)
```

→ Lua usage:

```
fusion:SetPrefs({
    ["Global.Network.Mail.OnJobFailure"]=true,
    ["Global.Network.Mail.Recipients"]="admin@studio.com"
})

fusion:SetPrefs("Global.Controls.AutoClose", false)
```

→ Parameters:

prefname (*string*) – prefname

val (*value*) – val

`Fusion.SetPrefs(prefs)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Set preferences from a table of attributes.

The SetPrefs function can be used to specify the values of virtually all preferences in Fusion. Its can take a table of values, identified by name, or a single name and value.

The table provided as an argument should have the format [prefs_name] = value. Subtables are allowed.

→ Python usage:

```
fusion.SetPrefs({
    “Global.Network.Mail.OnJobFailure”: True,
    “Global.Network.Mail.Recipients”: “admin@studio.com”
})
fusion.SetPrefs(“Global.Controls.AutoClose”, False)
```

→ Lua usage:

```
fusion:SetPrefs({
    [“Global.Network.Mail.OnJobFailure”]=true,
    [“Global.Network.Mail.Recipients”]=”admin@studio.com”
})

fusion:SetPrefs(“Global.Controls.AutoClose”, false)
```

→ Parameters:

prefs (*table*) – prefs

`Fusion.ShowAbout()`

Display the About dialog.

`Fusion.ShowPrefs([`*pageid*`][, `*showall*`][, `*comp*`])`

Display the Preferences dialog.

→ Parameters:

pageid (*string*) – pageid

showall (*boolean*) – showall

comp (*Composition*) – comp

`Fusion.ShowWindow(mode)`

Show or Hide main window.

This function will show or hide the main window of Fusion. Note that you can only reshow the window after hiding it if you are using the command prompt to control Fusion.

→ **Parameters:**

mode (*number*) – mode

`Fusion.Test()`

Test

`Fusion.ToggleBins()`

Shows or hides the Bins window.

The ShowPrefs function will display the Preferences dialog. Optional arguments can be used to specify which page or panel of the preferences will be opened.

prefname name of the specific page (or panel) of the preferences to show. The name should be chosen from one of the following:

- PrefsGeneral
- Prefs3D
- PrefsBinSecurity
- PrefsBinServers
- PrefsBins
- PrefsDefaults
- PrefsFlow
- PrefsFrameFormat
- PrefsEDLImport
- PrefsLayout
- PrefsLoader
- PrefsMemory
- PrefsNetwork
- PrefsOpenCL
- PrefsPathMap
- PrefsPreview
- PrefsQuickTime
- PrefsScript
- PrefsSplineViews
- PrefsSplines
- PrefsTimeline
- PrefsTweaks
- PrefsUI
- PrefsDeckLink
- PrefsView

→ Python usage:

```
# Open Preferences at the view page
fu.ShowPrefs(“PrefsView”)

# Print possible prefname for the current Fusion version
for v in fu.GetRegList(comp.CT_Prefs).values():
    print(v.GetAttrs()[“REGS_ID”])
```

→ Lua usage:

```
-- Open Preferences at the view page
fu:ShowPrefs(“PrefsView”)

-- Print possible prefname for the current Fusion version
for i,v in ipairs(fu:GetRegList(CT_Prefs)) do
    print(v:GetAttrs().REGS_ID)
end
```

`Fusion.ToggleRenderManager()`

Shows or hides the Render Manager.

`Fusion.ToggleUtility(id)`

Shows or hides a Utility plugin.

→ Parameters:

id (*string*) – id

FuView

`class FuView`

Parent class: `Object`

Members

`FuView.ID()`

ID of this View (read-only).

Methods

`FuView.Refresh()`

Redraw this view.

GL3DViewer

`class GL3DViewer`

Parent class: `GLViewer`

Methods

`GL3DViewer.CenterSelected()`

Centers this view on the selected object.

`GL3DViewer.FitAll()`

Fits this view to the entire scene.

`GL3DViewer.FitSelected()`

Fits this view to the selected object.

GLImageViewer

`class GLImageViewer`

Parent class: `GLViewer`

Methods

`GLImageViewer.DragRoI()`

Lets the user drag out an RoI rectangle.

`GLImageViewer.EnableLUT(`[*enable*]`)`

Enables or disables the current View LUT.

→ Parameters:

enable (*boolean*) – enable

`GLImageViewer.EnableRoI(`[*enable*]`)`

Enables or disables the current View RoI.

→ Parameters:

enable (*boolean*) – enable

`GLImageViewer.ExportTo3DLUT()`

Exports the current LUTs to a 3D LUT file.

- **Returns**: success
- **Return type**: boolean

`GLImageViewer.IsLUTEnabled()`

Returns true if the current View LUT is enabled.

- **Returns**: enabled
- **Return type**: boolean

`GLImageViewer.LoadLUTFile([pathname])`

Loads a LUT file, setting or LUT plugin ID into the View LUT.

- **Parameters**:

pathname (*string*) – pathname

- **Returns**: success
- **Return type**: boolean

`GLImageViewer.LockRoI([`*enable*`])`

Locks or unlocks the View RoI.

- **Parameters**:

enable (*boolean*) – enable

`GLImageViewer.SaveLUTFile([pathname])`

Saves current LUTs into a .viewlut file.

- **Parameters**:

pathname (*string*) – pathname

- **Returns**: success
- **Return type**: boolean

`GLImageViewer.SetRoI()`

Note: This method is overloaded and has alternative parameters. See other definitions.

Sets the current View RoI region.

`GLImageViewer.SetRoI(`*rect*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Sets the current View RoI region.

- **Parameters**:

rect (*table*) – rect

`GLImageViewer.SetRoI(auto)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Sets the current View RoI region.

→ **Parameters**:

auto (*boolean*) – auto

`GLImageViewer.ShowDoD([enable])`

Enables or disables drawing the current View DoD rectangle.

→ **Parameters**:

enable (*boolean*) – enable

`GLImageViewer.ShowLUTEditor()`

Pops up the Editor window for the current View LUT.

`GLImageViewer.ShowRoI([enable])`

Enables or disables drawing the current View RoI rectangle.

→ **Parameters**:

enable (*boolean*) – enable

GLPreview

`class GLPreview`

Parent class: `Preview`

Members

`GLPreview.View`

Represents the display GLView for this Preview (read-only).

→ **Getting**:

view = GLPreview.View – (GLView)

GLView

`class GLView`

Parent class: `FuView`

→ Python usage:

```
# Reach Left GLView of Fusion instance
left = comp.GetPreviewList()[“Left”][“View”]
left.SetBuffer(0)
```

→ Lua usage:

```
-- Reach Left GLView of Fusion instance
left = comp:GetPreviewList().Left.View
left:SetBuffer(0)
```

Members

`GLView.CurrentViewer`

Returns the current viewer.

→ Getting:

viewer = GLView.CurrentViewer – (GLViewer)

Methods

`GLView.DisableCurrentTools()`

Pass-through the currently selected tools.

`GLView.DisableSelectedTools()`

Pass-through the selected tools.

`GLView.EnableLUT(`*enable*`)`

Enables or disables the current Monitor LUT.

→ Parameters:

enable (*boolean*) – enable

`GLView.EnableStereo(`*enable*`)`

Enables or disables 3D stereo display.

→ Parameters:

enable (*boolean*) – enable

`GLView.GetBuffer()`

Returns which buffer is shown.

- **Returns**: buffer
- **Return type**: number

`GLView.GetLocked()`

Returns true if the display is locked.

- **Returns**: enabled
- **Return type**: boolean

`GLView.GetPos()`

Returns the position of the display.

In Python use GetPosTable.

- **Returns**: x
- **Return type**: number

`GLView.GetPosTable()`

Returns the position of the display as a table.

- **Returns**: pos
- **Return type**: table

`GLView.GetPrefs()`

Retrieve a table of preferences for this view.

- **Returns**: prefs
- **Return type**: table

`GLView.GetPreview([`*buffer*`])`

Returns the buffer's Preview.

- **Parameters**:

buffer (*number*) – buffer

`GLView.GetRot()`

Returns the x,y,z rotation of the display in degrees.

In Python use GetRotTable.

- **Returns**: x
- **Return type**: number

`GLView.GetRotTable()`

Returns the x,y,z rotation of the display in degrees as a table.

- **Returns**: rot
- **Return type**: table

`GLView.GetScale()`

Returns the scale of the display.

- **Returns**: scale
- **Return type**: number

`GLView.GetSplit()`

Get the split position of the view.

In Python use GetSplitTable.

- **Returns**: x
- **Return type**: number

`GLView.GetSplitTable()`

Get the split position of the view as a table.

- **Returns**: split
- **Return type**: table

`GLView.GetStereoMethod()`

Returns the method and options being used for stereo display.

- **Returns:** method
- **Return type**: string

`GLView.GetStereoSource()`

Returns the source being used for stereo display.

- **Returns**: ABsource
- **Return type**: boolean

`GLView.GetViewerList()`

Returns a list of available viewers.

- **Returns**: viewers
- **Return type**: table

`GLView.IsLUTEnabled()`

Returns true if the current Monitor LUT is enabled.

- **Returns**: enabled
- **Return type**: boolean

`GLView.IsStereoEnabled()`

Indicates if stereo display is currently enabled.

→ **Returns**: enabled

→ **Return type**: boolean

`GLView.IsStereoSwapped()`

Indicates if the left & right stereo eyes are currently swapped.

→ **Returns**: enable

→ **Return type**: boolean

`GLView.LoadLUTFile(`*pathname*`)`

Loads a LUT file, setting or LUT plugin ID into the Monitor LUT.

→ **Parameters**:

pathname (*string*) – pathname

→ **Returns**: success

→ **Return type**: boolean

`GLView.LoadPrefs()`

Note: This method is overloaded and has alternative parameters. See other definitions.

Saves the current view prefs to a named configuration.

`GLView.LoadPrefs(`*configname*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Saves the current view prefs to a named configuration.

→ **Parameters**:

configname (*string*) – configname

`GLView.ResetView()`

Resets the display to default position etc.

`GLView.SavePrefs()`

Note: This method is overloaded and has alternative parameters. See other definitions.

Saves the current view prefs to a named configuration.

`GLView.SavePrefs(`*configname*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Saves the current view prefs to a named configuration.

→ **Parameters**:

configname (*string*) – configname

`GLView.SetBuffer(`*buffer*`)`

Show a particular buffer.

The SetBuffer function is used to display a specific one of the three possible view options for the A/B subviews in a view in Fusion. As stated above, 0 = the buffer view that the function is being run on, 1 = the buffer view that the function is not being run on, 2 = A/B view. So if the preview window that the function was being run on was the Left B view, the function would set the display viewer to B if the integer value was 0.

buffer buffer integer that the view will be set to. Buffer 0 = The Buffer view that is the currently selected on, 1 = The buffer view that is not the current one, 2 = A/B.

→ Python usage:

```
# Set the buffer to A/B with a 45 degree split at the center
left = comp.GetPreviewList()[“Left”][“View”]
left.SetBuffer(2)
left.SetSplit(0.5, 0.5, 45)
```

→ Lua usage:

```
-- Set the buffer to A/B with a 45 degree split at the center
left = comp:GetPreviewList().Left.View
left:SetBuffer(2)
left:SetSplit(.5, .5, 45)
```

→ Parameters:

buffer (*number*) – buffer

`GLView.SetLocked(`*enable*`)`

→ Parameters:

enable (*boolean*) – enable

`GLView.SetPos(`*x*, *y*`[, `*z*`])`

Set the position of the display.

Sets the position of the display relative to the center (0, 0). In a 3D GLView the view position can be set in 3D space.

x X coordinate in pixels (2D) or unity (3D)

Y Y coordinate in pixels (2D) or unity (3D)

Z Z coordinate in unity (3D only)

- **Parameters**:

  x (*number*) – x

  y (*number*) – y

  z (*number*) – z

- **Returns**: success
- **Return type**: boolean

`GLView.SetRot(`*x, y, z*`)`

Set the x,y,z rotation of the display in degrees.

- **Parameters**:

  x (*number*) – x

  y (*number*) – y

  z (*number*) – z

`GLView.SetScale(`*scale*`)`

Set the scale of the display.

The SetScale function is used to set the scale of a view.

scale the percentage, expressed as a numerical value, that the image in the view will be scaled by. Percentages are translated to numerical values (50% = .5, 200% = 2.0) with 0 being the view's "Fit" option.

- **Python usage**:

```
# Fit the left view
left = comp.GetPreviewList()[“Left”][“View”]
left.SetScale(0)
```

- **Lua usage**:

```
-- Fit the left view
left = comp:GetPreviewList().Left.View
left:SetScale(0)
```

- **Parameters**:

  scale (*number*) – scale

`GLView.SetSplit(`*x, y, angle*`)`

Set the split position of the view.

Sets the A/B view split based on the x, y, coordinates and the angle.

x the coordinate along the x axis of the A/B Split view's center.

y the coordinate along the y axis of the A/B Split view's center.

angle the angle of the A/B Split view line.

→ Python usage:

```
# Set the buffer to A/B with a 45 degree split at the center
left = comp.GetPreviewList()["Left"]["View"]
left.SetBuffer(2)
left.SetSplit(.5, .5, 45)
```

→ Lua usage:

```
-- Set the buffer to A/B with a 45 degree split at the center
left = comp:GetPreviewList().Left.View
left:SetBuffer(2)
left:SetSplit(.5, .5, 45)
```

→ Parameters:

x (*number*) – x

y (*number*) – y

angle (*number*) – angle

`GLView.SetStereoMethod(`*method*`[, `*option1*`][, `*option2*`])`

Sets the method for stereo display.

→ Parameters:

method (*string*) – method

option1 – option1

option2 – option2

`GLView.SetStereoSource(`*ABsource*`, `*stacked*`[, `*stackmethod*`])`

Sets the source for the left & right stereo images.

→ Parameters:

ABsource (*boolean*) – ABsource

stacked (*boolean*) – stacked

stackmethod (*string*) – stackmethod

`GLView.ShowLUTEditor()`

Pops up the Editor window for the current Monitor LUT.

`GLView.ShowQuadView(enable)`

Splits the view into four subviews.

→ **Parameters**:

enable (*boolean*) – enable

`GLView.ShowSubView(`*enable*`)`

Enables the inset SubView display.

→ **Parameters**:

enable (*boolean*) – enable

`GLView.ShowingQuadView()`

Returns true if the view is split into four.

→ **Returns**: enabled

→ **Return type**: boolean

`GLView.ShowingSubView()`

→ Returns true if the inset SubView is currently being displayed.

→ **Returns**: enabled

→ **Return type**: boolean

`GLView.SwapStereo()`

Note: This method is overloaded and has alternative parameters. See other definitions.

Swaps left & right stereo eye views.

`GLView.SwapStereo(`*enable*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Swaps left & right stereo eye views.

→ **Parameters**:

enable (*boolean*) – enable

`GLView.SwapSubView()`

Swaps the SubView with the Main View.

→ **Returns**: enabled

→ **Return type**: boolean

GLViewer

class `GLViewer`

Parent class: `Object`

Parent class for 2D and 3D viewers.

2D image viewers are instances of the GLImageViewer subclass and have additional methods to set and show the DoD, RoI or LUT.

Please note that most Set-methods need to be followed by a Redraw() call.

→ Python usage:

```
# Reach the Left GLViewer
left = comp.GetPreviewList()[“Left”][“View”]
left_viewer = left.CurrentViewer
if left_viewer != None:
    left_viewer.SetChannel(0)
    left_viewer.Redraw()
```

→ Lua usage:

```
-- Reach the Left GLViewer
left = comp:GetPreviewList().Left.View
left_viewer = left.CurrentViewer
if left_viewer ~= nil then
    left_viewer:SetChannel(0)
    left_viewer:Redraw()
end
```

Methods

`GLViewer.AreControlsShown()`

Returns true if controls are being displayed on the view.

- **Returns**: enabled
- **Return type**: boolean

`GLViewer.AreGuidesShown()`

Returns true if image guides are being displayed on the view.

- **Returns**: enabled
- **Return type**: boolean

`GLViewer.GetAlphaOverlayColor()`

Return which alpha overlay is being used.

- **Returns:** color
- **Return type**: number

`GLViewer.GetAspectCorrection()`

Returns true if the viewer is correcting the aspect of images.

- **Returns:** enabled
- **Return type**: boolean

`GLViewer.GetChannel()`

Return which channel is shown.

- **Returns:** channel
- **Return type**: number

`GLViewer.GetPos()`

Get the position of the viewer.

In Python use GetPosTable.

- **Returns**: x
- **Return type**: number

`GLViewer.GetPosTable()`

Get the position of the viewer as a table.

- **Returns**: pos
- **Return type**: table

`GLViewer.GetRot()`

Get the rotation angles of the view.
In Python use GetRotTable.

- **Returns**: x
- **Return type**: number

`GLViewer.GetRotTable()`

Get the rotation angles of the view as a table.

→ **Returns**: rot

→ **Return type**: table

`GLViewer.GetScale()`

Get the scale (zoom) of the view.

→ **Returns**: scale

→ **Return type**: number

`GLViewer.LoadFile(`*filename*`)`

Load and display the contents of a file.

→ **Parameters:**

filename (*string*) – filename

`GLViewer.Redraw()`

Refreshes the viewer.

`GLViewer.ResetView()`

Resets the display to default position etc.

`GLViewer.SaveFile(`*filename*`)`

Save the currently displayed parameter.

→ **Parameters:**

filename (*string*) – filename

`GLViewer.SetAlphaOverlayColor(`*color*`)`

Select which alpha overlay to use.

→ **Parameters:**

color (*number*) – color

`GLViewer.SetAspectCorrection(`*enable*`)`

Enables or disables aspect correction.

→ **Parameters:**

enable (*boolean*) – enable

`GLViewer.SetChannel(`*channel*`,` *toggle*`)`

Select which channel to show.

→ **Parameters:**

channel (*number*) – channel

toggle (*boolean*) – toggle

`GLViewer.SetPos(`*x*`,` *y*`[,` *z*`])`

Set the position of the viewer.

→ **Parameters:**

x (*number*) – x

y (*number*) – y

z (*number*) – z

→ **Returns**: success

→ **Return type**: boolean

`GLViewer.SetRot(`*x*`,` *y*`,` *z*`)`

Set the rotation of the view.

→ **Parameters**:

x (*number*) – x

y (*number*) – y

z (*number*) – z

`GLViewer.SetScale(`*scale*`)`

Set the scale (zoom) of the view.

→ **Parameters:**

scale (*number*) – scale

`GLViewer.ShowControls(`*enable*`)`

Shows or hides controls on the view.

→ **Parameters:**

enable (*boolean*) – enable

`GLViewer.ShowGuides(`*enable*`)`

Shows or hides guides on the view.

→ **Parameters:**

enable (*boolean*) – enable

Gradient

`class Gradient`

Parent class: `Parameter`

Members

`Gradient.Value`

The gradient in table form.

→ **Getting**:

gradient = Gradient.Value – (table)

→ **Setting**:

Gradient.Value = gradient – (table)

GraphView

`class GraphView`

Parent class: `FuScrollView`

Methods

`GraphView.DeleteGuides([`*start*`][, `*end*`])`

Deletes guides between start and end.

→ **Parameters:**

start (*number*) – start

end (*number*) – end

`GraphView.GetClipboard()`

Retrieves the tool(s) on the clipboard, as tables and as ASCII text..

→ **Returns**: clipboard

→ **Return type**: table

`GraphView.GetGuides([`*start*`][, `*end*`])`

Returns a table of snapguide times & names.

→ **Parameters:**

start (*number*) – start

end (*number*) – end

→ **Returns**: guides

→ **Return type**: table

`GraphView.GoNextKeyTime()`

Jumps to next key frame of the active spline.

`GraphView.GoPrevKeyTime()`

Jumps to previous key frame of the active spline.

`GraphView.Paste(`*desttime*`, `*spline1*`[, `*spline2...*`][, `*points*`])`

Paste points to given splines at given time from the Clipboard.

→ Parameters:

desttime (*number*) – desttime

spline1 (*object*) – spline1

spline2... (*object*) – spline2...

points (*table*) – points

→ **Returns**: success

→ **Return type**: boolean

`GraphView.SetGuides([`*guides*`][, `*rem_prev*`])`

Sets snapguide.

→ Parameters:

guides (*table*) – *guides*

rem_prev (*boolean*) – rem_prev

`GraphView.ZoomFit()`

Changes scale to fit all displayed splines within the view.

`GraphView.ZoomIn()`

Increases the scale (zoom) of the view.

`GraphView.ZoomOut()`

Decreases the scale (zoom) of the view.

`GraphView.ZoomRectangle()`

Note: This method is overloaded and has alternative parameters. See other definitions.

Fill the view with the specified rectangle.

`GraphView.ZoomRectangle(`*x1*`, `*y1*`, `*x2*`, `*y2*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Fill the view with the specified rectangle.

→ Parameters:

x1 (*number*) – x1

y1 (*number*) – y1

x2 (*number*) – x2

y2 (*number*) – y2

HotkeyManager

`class HotkeyManager`

Parent class: `LockableObject`

Methods

`HotkeyManager.GetDefaults()`

GetDefaults

`HotkeyManager.GetHotkeys()`

GetHotkeys

`HotkeyManager.GetKeyNames()`

GetKeyNames

`HotkeyManager.GetModifierNames()`

GetModifierNames

`HotkeyManager.LoadHotkeys()`

LoadHotkeys

`HotkeyManager.SaveHotkeys()`

SaveHotkeys

`HotkeyManager.SetHotkey()`

SetHotkey

`HotkeyManager.SetHotkeys()`

SetHotkeys

Image

`class Image`

Parent class: `Parameter`

Members

`Image.DataWindow`

Rectangle of valid data pixels, in a table (read-only).

→ Getting:

rect = Image.DataWindow – (table)

`Image.Depth`

Image depth indicator (not in bits) (read-only).

→ Getting:

val = Image.Depth – (number)

`Image.Field`

Field indicator (read-only).

→ Getting:

val = Image.Field – (number)

`Image.Height`

Actual image height, in pixels (read-only).

→ Getting:

val = Image.Height – (number)

`Image.OriginalHeight`

Unproxied image height, in pixels (read-only).

→ Getting:

val = Image.OriginalHeight – (number)

`Image.OriginalWidth`

Unproxied image width, in pixels (read-only).

→ Getting:

val = Image.OriginalWidth – (number)

`Image.OriginalXScale`

Unproxied pixel X Aspect (read-only).

→ Getting:

val = Image.OriginalXScale – (number)

`Image.OriginalYScale`

Unproxied pixel Y Aspect (read-only).

→ Getting:

val = Image.OriginalYScale – (number)

`Image.ProxyScale`

Image proxy scale multiplier (read-only).

→ Getting:

val = Image.ProxyScale – (number)

`Image.Width`

Actual image width, in pixels (read-only).

→ Getting:

val = Image.Width – (number)

`Image.XOffset`

Image X Offset (read-only).

→ Getting:

val = Image.XOffset – (number)

`Image.XScale`

Pixel X Aspect (read-only).

→ Getting:

val = Image.XScale – (number)

`Image.YOffset`

Image X Offset (read-only).

→ Getting:

val = Image.YOffset – (number)

`Image.YScale`

Pixel Y Aspect (read-only).

→ Getting:

val = Image.YScale – (number)

ImageCacheManager

class `ImageCacheManager`

Parent class: `Object`

Methods

`ImageCacheManager.FreeSpace()`

FreeSpace

`ImageCacheManager.GetSize()`

GetSize

`ImageCacheManager.IsRoom()`

This is useful to see how much room there currently is in the cache manager by checking to see if a certain number of bytes will fit without needing to purge/flush.

bytes The number of bytes to check.

Returns a boolean indicating whether or not there is room in the cache manager for the number of bytes passed as an argument.

`ImageCacheManager.Purge()`

This function allows the cache to be purged exactly as if doing so interactively in Fusion.

IOClass

`class IOClass`

Parent class: `Object`

Methods

`IOClass.Close()`

Close

`IOClass.Flush()`

Flush

`IOClass.GetFilePos()`

GetFilePos

`IOClass.GetFileSize()`

GetFileSize

`IOClass.Read()`

Read

`IOClass.ReadLine()`

ReadLine

`IOClass.Seek()`

Seek

`IOClass.Write()`

Write

`IOClass.WriteLine()`

WriteLine

KeyFrameView

`class KeyFrameView`

Parent class: `GraphView`

Methods

`KeyFrameView.GoNextKeyTime()`

Jumps to next key frame of the active spline.

`KeyFrameView.GoPrevKeyTime()`

Jumps to previous key frame of the active spline.

Link

`class Link`

Parent class: `LockableObject`

Represents the parent class of Input and Outputs.

Members

`Link.ID`

ID of this Link (read-only).

→ **Getting**:

id = Link.ID – (string)

`Link.Name`

Friendly name of this Link (read-only).

→ **Getting**:

name = Link.Name – (string)

Methods

`Link.GetData([`*name*`])`

Get custom persistent data.

`See Composition:GetData().`

→ **Parameters**:

name (*string*) – name

→ **Returns**: value

→ **Return type**: (number|string|boolean|table)

`Link.GetTool()`

Returns the Tool object that owns this Link.

- **Returns**: tool
- **Return type**: Tool

`Link.SetData(`*name*`, `*value*`)`

Set custom persistent data.

`See Composition:SetData().`

- **Parameters**:

  name (*string*) – name

  value ((*number|string|boolean|table*)) – value

List

`class List`

Parent class: `LockableObject`

Loader

`class Loader`

Parent class: `ThreadedOperator`

Methods

`Loader.SetMultiClip(`*filename*`[, `*startframe*`][, `*trimin*`][, `*trimout*`])`

Gives Loader a MultiClip clip list.

- **Parameters**:

  startframe (*number*) – startframe

  trimin (*number*) – trimin

  trimout (*number*) – trimout

MailMessage

`class MailMessage`

Parent class: `Object`

Represents an email message.

Please note that if no explicit server settings are set with
the SetServer, SetLogin and SetPassword

methods, the default Preferences (Globals -> Network -> Server Settings ...) are used. If these are not set the recipient server is tried to be reached.

→ Python usage:

```
mail = fusion.CreateMail()

mail.AddRecipients("vfx@studio.com, myself@studio.com")
mail.SetSubject("Render Completed")
mail.SetBody("The job completed.")

print(mail.SendTable())
status = mail.SendTable().values()
print (status[0]) # success boolean
if len(status) > 1:
    print(status[1]) # error message
```

→ Lua usage:

```
mail = fusion:CreateMail()

mail:AddRecipients("vfx@studio.com, myself@studio.com")
mail:SetSubject("Render Completed")
mail:SetBody("The job completed.")

ok,errmsg = mail:Send()
print(ok)
print(errmsg)
```

Methods

`MailMessage.AddAttachment(`*filename*`)`

Attaches a filename to the body.

- → **Parameters**:

  filename (*string*) – filename

- → **Returns:** success
- → **Return type**: boolean

`MailMessage.AddRecipients(`*addresses*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Adds a recipient to the To: list.

- → **Parameters**:

  addresses (*string*) – addresses

`MailMessage.AddRecipients(`*addresses*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Adds a recipient to the To: list.

- → **Parameters**:

  addresses (*table*) – addresses

`MailMessage.GetTable()`

Returns the message in the form of a table.

- → **Returns**: msg
- → **Return type**: table

`MailMessage.RemoveAllAttachments()`

Removes all attachments from the message.

`MailMessage.RemoveAllRecipients()`

Removes all recipients from the To: field.

`MailMessage.Send()`

Sends the message.

Return the success as bool and the message.

Note there is a SendTable method for Python.

- → **Returns**: success
- → **Return type**: boolean

`MailMessage.SetBody(bodytext)`

Sets the message body.

- → **Parameters**:

  bodytext (*string*) – bodytext

`MailMessage.SetHTMLBody(`*bodyhtml*`)`

Sets the message body using HTML.

→ Parameters:

bodyhtml (*string*) – bodyhtml

`MailMessage.SetLogin(`*login*`)`

Sets the login to use for authentication.

→ Parameters:

login (*string*) – login

`MailMessage.SetPassword(`*password*`)`

Sets the password to use for authentication.

→ Parameters:

password (*string*) – password

`MailMessage.SetSender(`*sender*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Sets the From: field.

→ Parameters:

sender (*string*) – sender

`MailMessage.SetSender(`*sender*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Sets the From: field.

→ Parameters:

sender (*table*) – sender

`MailMessage.SetServer(`*servername*`)`

Sets the outgoing mail server to use.

→ Parameters:

servername (*string*) – servername

`MailMessage.SetSubject(`*subject*`)`

Sets the Subject: field.

→ Parameters:

subject (*string*) – subject

MenuManager

`class MenuManager`

Parent class: `LockableObject`

Methods

`MenuManager.GetMenus()`

GetMenus

`MenuManager.LoadMenus()`

LoadMenus

`MenuManager.SaveMenus()`

SaveMenus

Object

`class Object`

Operator

`class Operator`

Parent class: `Object`

Base class for all Tools, Modifiers etc.

Operator Attributes

Attribute Name	Type	Description
TOOLS_Name	string	The full name of this tool
TOOLS_Name	string	The full name of this tool
TOOLB_Visible	integer	Indicates if this tool is visible on the flow, or a non-visible tool, such as a modifier.
TOOLB_Locked	boolean	Indicates if this tool is locked.
TOOLB_PassThrough	boolean	Indicates if this tool is set to pass-through.
TOOLB_HoldOutput	boolean	Indicates if this tool is set to hold its output (not update).
TOOLB_CtrlWZoom	integer	Indicates if this tool's control window is open or closed.

Attribute Name	Type	Description
TOOLB_NameSet	boolean	Indicates if this tool's name has been set (by the user) or is the default name.
TOOLB_CacheToDisk	integer	Indicates if this tool is set to cache itself to disk.
TOOLS_RegID	string	The RegID of this tool.
TOOLH_GroupParent	group userdata	The associated group object
TOOLNT_EnabledRegion_Start	number	The point (frame) at which this tool is enabled, and will start to take effect.
TOOLNT_EnabledRegion_End	number	The point (frame) at which this tool is disabled, and will cease to have an effect..
TOOLNT_Region_Start	number	The point at which this tool can start providing results.
TOOLNT_Region_End	composition userdata	The point at which this tool stops providing results.
TOOLN_LastFrameTime	number	The amount of time (in seconds) taken to process the most recently rendered frame by this tool.
TOOLI_Number_o_Inputs	number	Useful for determining the number of inputs a tool has (implemented for 3D merges).
TOOLI_ImageWidth	integer	For image-based tools, these represent the format of the image most recently processed by this tool.
TOOLI_ImageHeight	integer	
TOOLI_ImageField	integer	
TOOLI_ImageDepth	integer	
TOOLN_ImageAspectX	number	
TOOLN_ImageAspectY	number	
TOOLST_Clip_Name	string	For clip-based tools (Loader and Saver), one or more entries for these may be present in tables to define information on the clip(s) currently selected into this tool. Note that these attributes actually return a table of values of the type indicated in parenthesis. Each index in the table represents a clip in the cliplist.

Attribute Name	Type	Description
TOOLIT_Clip_Width	integer	
TOOLIT_Clip_Height	integer	
TOOLIT_Clip_StartFrame	integer	
TOOLIT_Clip_Length	integer	
TOOLBT_Clip_IsMultiFrame	boolean	
TOOLST_Clip_FormatName	string	
TOOLST_Clip_FormatID	string	
TOOLNT_Clip_Start	number	
TOOLNT_Clip_End	number	
TOOLBT_Clip_Reverse	boolean	
TOOLBT_Clip_Saving	boolean	
TOOLBT_Clip_Loop	boolean	
TOOLIT_Clip_TrimIn	integer	
TOOLIT_Clip_TrimOut	integer	
TOOLIT_Clip_ExtendFirst	integer	
TOOLIT_Clip_ExtendLast	integer	
TOOLIT_Clip_ImportMode	integer	
TOOLIT_Clip_PullOffset	integer	
TOOLIT_Clip_InitialFrame	integer	
TOOLIT_Clip_AspectMode	integer	
TOOLIT_Clip_TimeCode	integer	
TOOLST_Clip_KeyCode	string	
TOOLST_AltClip_Name	string	
TOOLIT_AltClip_Width	integer	
TOOLIT_AltClip_Height	integer	
TOOLIT_AltClip_StartFrame	integer	
TOOLIT_AltClip_Length	integer	
TOOLBT_AltClip_IsMultiFrame	boolean	
TOOLST_AltClip_FormatName	string	
TOOLST_AltClip_FormatID	string	

Members

`Operator.Composition`

The composition that this tool belongs to (read-only).

→ **Getting**:

comp = Operator.Composition – (Composition)

`Operator.FillColor`

→ **Getting**:

color = Operator.FillColor – (table)

→ **Setting**:

Operator.FillColor = color – (table)

`Operator.ID`

Registry ID of this tool (read-only).

→ **Getting**:

id = Operator.ID – (string)

`Operator.Name`

Friendly name of this tool (read-only).

→ **Getting**:

name = Operator.Name – (string)

`Operator.ParentTool`

The parent tool of this tool (read-only).

That is a group parent if the tool is inside a group or macro.

→ **Getting**:

parent = Operator.ParentTool – (Tool)

`Operator.TextColor`

Color of a tool's icon text in the Flow view.

→ **Getting**:

color = Operator.TextColor – (table)

→ **Setting**:

Operator.TextColor = color – (table)

`Operator.TileColor`

Color of a tool's icon in the Flow view.

→ **Getting**:

color = Operator.TileColor – (table)

→ Setting:

Operator.TileColor = color – (table)

`Operator.UserControls`

Table of user-control definitions.

→ Getting:

controls = Operator.UserControls – (table)

→ Setting:

Operator.UserControls = controls – (table)

Methods

`Operator.AddModifier(`*input*`,` *modifier*`)`

Creates a modifier and connects it to an input.

This provides an easy way to animate the controls of a tool.

input ID of the tool's Input to be connected to.

modifier ID of the modifier to be created.

Returns a boolean value indicating success.

→ Python usage:

```
myBlur = comp.Blur()
if myBlur.AddModifier("Blend", "BezierSpline"):
    myBlur.Blend[0] = 1.0
    myBlur.Blend[100] = 0.0
```

→ Lua usage:

```
myBlur = Blur()
if myBlur:AddModifier("Blend", "BezierSpline") then
    myBlur.Blend[0] = 1.0
    myBlur.Blend[100] = 0.0
end
```

- **Parameters**:

  input (*string*) – input

  modifier (*string*) – modifier

- **Returns**: success
- **Return type**: boolean

`Operator.ConnectInput(`*input*, *target*`)`

Connect or disconnect an Input.

The input can be connected to an Output or an Operator, or to nil, which disconnects the input.

If the target given is an Operator, the Input will be connected to that Operator's main Output.

input the ID of an Input to connect

target an Output or Operator object to connect the input to, or nil to disconnect

- **Python usage**:

```
# Find a Loader, and connect it to Merge1.Foreground
ldr = comp.FindToolByID("Loader")
if ldr and comp.Merge1:

    comp.Merge1.ConnectInput("Foreground", ldr)
```

- **Lua usage**:

```
-- Find a Loader, and connect it to Merge1.Foreground
ldr = comp:FindToolByID("Loader")

if ldr and Merge1 then
    print(comp.ActiveTool)
    Merge1:ConnectInput("Foreground", ldr)
end
```

- **Parameters**:

  input (*string*) – input

  target ((*Tool|Output|nil*)) – target
- **Returns**: success
- **Return type**: boolean

`Operator.Delete()`

Delete this tool.

Removes the tool from the composition. This also releases the handle to the Fusion Tool object, setting it to nil.

`Operator.FindMainInput(`*index*`)`

Returns the tool's main (visible) input.

index integer value of 1 or greater.

- **Python usage**:

```python
# Loop through all main inputs.

i = 1
while True:
    inp = tool.FindMainInput(i)
    if inp is None:
        break

    # Got input
    print(inp.GetAttrs()["INPS_Name"])
    i+=1
```

- **Lua usage**:

```lua
-- Loop through all main inputs.
tool = comp.ActiveTool
i = 1
while true do
    inp = (tool:FindMainInput(i))
    if inp == nil then
        break
    end

    -- Got input
    print (inp:GetAttrs().INPS_Name)
    i = i + 1
end
```

- **Parameters**:
  index (*number*) – index
- **Returns**: inp
- **Return type**: Input

`Operator.FindMainOutput(`*index*`)`

Returns the tool's main (visible) output.

index integer value of 1 or greater.

- **Python usage**:

```
# Loop through all main outputs.

i = 1
while True:
    outp = tool.FindMainOutput(i)
    if outp is None:
        break

    # Got output
    print(outp.GetAttrs()["OUTS_Name"])
    i+=1
```

- **Lua usage**:

```
-- Loop through all main outputs.
tool = comp.ActiveTool
i = 1
while true do
    outp = (tool:FindMainOutput(i))
    if outp == nil then
        break
    end

    -- Got output
    print (outp:GetAttrs().OUTS_Name)
    i = i + 1
end
```

- **Parameters**:

  index (*number*) – index

- **Returns**: out
- **Return type**: Output

`Operator.GetChildrenList([`*selected*`][, `*regid*`])`

Returns a list of all children tools, or selected children tools.

This function is useful for finding members of Macro or Group tools.

selected Pass true to get only selected child tools.

regid pass a Registry ID string to get only child tools of that type.

Returns a table of tool objects.

- Python usage:

```
# list all tools in a group or macro
for t in comp.ActiveTool.GetChildrenList().values():
    print(t.Name)
```

- Lua usage:

```
-- list all tools in a group or macro
for i,t in pairs(comp.ActiveTool:GetChildrenList()) do
    print(t.Name)
end
```

- Parameters:

  selected (*boolean*) – selected

  regid (*string*) – regid

- **Returns**: tools
- **Return type**: table

`Operator.GetControlPageNames()`

Returns a table of control page names, indexed by page number.

- **Returns**: names
- **Return type**: table

`Operator.GetCurrentSettings()`

Returns the index of the tool's current settings slot.

A tool has 6 different collections/slots of settings. By default, it uses slot 1.

Returns a numerical index of 1 or greater.

- **Returns**: index
- **Return type**: number

`Operator.GetData([`*name*`])`

Get custom persistent data.

See Composition:GetData().

- Parameters:

  name (*string*) – name

- **Returns**: value
- **Return type**: (number|string|boolean|table)

`Operator.GetInput(`*id*`[, `*time*`])`

Fetches the value of an input at a given time.

The time argument may be omitted, if the input is not animated.

A similar result may be obtained by simply indexing the input with the desired time.

id the ID of the input to be queried.

time the keyframe time to be queried.

Returns a number, string or other Parameter object, depending on the DataType of the queried Input.

- **Python usage**:

```
# these lines: the same thing
print(tool:GetInput("Blend", 30.0))
print(tool.Blend[30])
```

- **Lua usage**:

```
-- these lines do the same thing
print(tool:GetInput("Blend", 30.0)
print(tool.Blend[30.0]
```

- **Parameters**:

  id (*string*) – id

  time (*number*) – time

- **Returns**: value
- **Return type**: (number|string|Parameter)

`Operator.GetInputList([`*type*`])`

Return a table of all inputs on this tool.

type can be used to filter the results to return only a specific datatype. Valid values include "Image", "Number", "Point", "Gradient" and "Text".

Returns a table containing handles all the Inputs available for the tool.

- Python usage:

```
# this Tool script prints out the name
# of every control on the selected tool
tool = comp.ActiveTool
x = tool.GetInputList().values()
for inp in x:
    print(inp.GetAttrs()["INPS_Name"])
```

- Lua usage:

```
-- this Tool script prints out the name
-- of every control on the selected tool
tool = tool or comp.ActiveTool
x = tool:GetInputList()

for i, inp in pairs(x) do
  print(inp:GetAttrs().INPS_Name)
end
```

- **Parameters**:

  type (*string*) – type

- **Returns**: inputs
- **Return type**: table

`Operator.GetKeyFrames()`

Return a table of all keyframe times for this tool.

Returns a table containing a list of keyframe times, in order, for the tool only. Any animation splines or modifiers attached to the tool's inputs are not considered.

- **Returns**: keyframes
- **Return type**: table

`Operator.GetOutputList([`*type*`])`

Return a table of all outputs on this tool.

type can be used to filter the results to return only a specific datatype. Valid values include "Image", "Number", "Point", "Gradient" and "Text".

Returns a table containing handles all the Outputs available for the tool.

→ Python usage:

```
# this Tool script prints out the name
# of every output on the selected tool
tool = comp.ActiveTool
x = tool.GetOutputList().values()

for outp in x:

    print(outp.GetAttrs()["OUTS_Name"])
```

→ Lua usage:

```
-- this Tool script prints out the name
-- of every output on the selected tool
tool = tool or comp.ActiveTool
x = tool:GetOutputList()

for i,out in pairs(x) do

   print(out:GetAttrs().OUTS_Name)

end
```

- **Parameters**:

  type (*string*) – type
- **Returns**: outputs
- **Return type**: table

`Operator.LoadSettings`(*filename*)

Note: This method is overloaded and has alternative parameters. See other definitions.

Load the tools's settings from a file or table.

Used to load .setting files or tables into a tool. This is potentially useful for any number of applications, such as loading curve data into fusion or to synch updates to tools over project management systems.

- **Python usage**:

```
settingtable = bmd.readfile("fusion:\\settings\\ccv_project1.setting")
comp.ColorCurve1.LoadSettings(settingtable)

# Same as
comp.ColorCurve1.LoadSettings("fusion:\\settings\\ccv_project1.setting")
```

- **Lua usage**:

```
settingtable = bmd.readfile("fusion:\\settings\\ccv_project1.setting")
ColorCurve1:LoadSettings(settingtable)

-- Same as
ColorCurve1:LoadSettings("fusion:\\settings\\ccv_project1.setting")
```

- **Parameters**:

  filename (*string*) – filename
- **Returns**: success
- **Return type**: boolean

`Operator.LoadSettings(`*settings*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Load the tools's settings from a file or table.

Used to load .setting files or tables into a tool. This is potentially useful for any number of applications, such as loading curve data into fusion or to synch updates to tools over project management systems.

→ **Python usage**:

```
settingtable = bmd.readfile("fusion:\\settings\\ccv_project1.setting")
comp.ColorCurve1.LoadSettings(settingtable)
# Same as
comp.ColorCurve1.LoadSettings("fusion:\\settings\\ccv_project1.setting")
```

→ **Lua usage**:

```
settingtable = rbmd.readfile("fusion:\\settings\\ccv_project1.setting")
ColorCurve1:LoadSettings(settingtable)

-- Same as
ColorCurve1:LoadSettings("fusion:\\settings\\ccv_project1.setting")
```

→ **Parameters**:

settings (*table*) – settings

→ **Returns**: success

→ **Return type**: boolean

`Operator.Refresh()`

Refreshes the tool, showing updated user controls.

Calling Refresh will invalidate the handle to the tool. A new handle is returned and can be stored.

Returns a new handle to the refreshed tool.

`Operator.SaveSettings(`*filename*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Save the tool's current settings to a file or table.

If a path is given, the tool's settings will be saved to that file, and a boolean is returned to indicate success.

If no path is given, SaveSettings() will return a table of the tool's settings instead.

→ **Parameters**:

filename (*string*) – filename

→ **Returns**:	success
→ **Return type**:	boolean

`Operator.SaveSettings(`*customdata*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Save the tool's current settings to a file or table.

If a path is given, the tool's settings will be saved to that file, and a boolean is returned to indicate success.

If no path is given, SaveSettings() will return a table of the tool's settings instead.

→ **Parameters**:

customdata (*boolean*) – customdata

→ **Returns**:	settings
→ **Return type**:	table

`Operator.SetCurrentSettings()`

Sets the tool's current settings slot.

If the slot is not empty, the function will change all the tool's Inputs to the settings stored in that slot.

A tool has 6 different collections ("slots") of settings. By default, it uses slot 1. Changing the current settings slot may change any or all of the tool's Inputs to new values, or new animations, stored in the new slot (if any).

All of the tool's previous settings are stored in the old slot, before changing to a new slot.

index numerical index of 1 or greater.

→ **Python usage**:

```
import time


tool = comp.ActiveTool

slot = tool.GetCurrentSettings()
```

```

# change to new slot, and turn off the effect
tool.SetCurrentSettings(slot + 1)
tool.Blend[comp.CurrentTime] = 0.0
print(tool.Name + “. Before...”)

# wait(a few seconds)
time.sleep(3)

# change back to the old slot, and turn the effect back on
tool.SetCurrentSettings(slot)
tool.Blend[comp.CurrentTime] = 1.0
print(tool.Name + “. After!”)
```

→ Lua usage:

```
local clock = os.clock
function sleep(n)  -- seconds
  local t0 = clock()
  while clock() - t0 <= n do end
end

tool = tool or comp.ActiveTool
slot = tool:GetCurrentSettings()

-- change to new slot, and turn off the effect
tool:SetCurrentSettings(slot + 1)
tool.Blend[comp.CurrentTime] = 0.0
print(tool.Name .. “: Before...”)
```

```

-- wait(a few seconds)
sleep(3)

-- change back to the old slot, and turn the effect back on
tool:SetCurrentSettings(slot)
tool.Blend[comp.CurrentTime] = 1.0
print(tool.Name .. ": After!")
```

- **Returns**: index
- **Return type**: number

`Operator.SetData(`*name*`,` *value*`)`

Set custom persistent data.

See Composition:SetData().

- **Parameters**:

name (*string*) – name

value ((*number|string|boolean|table*)) – value

`Operator.SetInput(`*id*`,` *value*`,` *time*`)`

Sets the value of an input at a given time.

The time argument may be omitted, if the input is not animated.

A similar result may be obtained by simply indexing the input with the desired time, and assigning to that.

- **Parameters**:

id (*string*) – id

value ((*number|string|Parameter*)) – value

time (*number*) – time

`Operator.ShowControlPage(`*name*`)`

Makes the specified control page visible.

Valid ControlPageNames for the tool can be queried with GetControlPageNames().

- **Parameters**:

name (*string*) – name

Parameter

`class Parameter`

Parent class: `Object`

Base class for Parameters like Image, Number etc.

Members

`Parameter.ID`

ID of this Parameter (read-only).

→ **Getting**:

id = Parameter.ID – (string)

`Parameter.Metadata()`

Get or set metadata tables.

Note that setting a Metadata from a regular script will be reset once the Loader re-evaluates the Output.

→ **Python usage**:

```
metadata = comp.Loader1.Output.GetValue().Metadata
print("Image was loaded from " + metadata["Filename"])
```

→ **Lua usage**:

```
metadata = Loader1.Output:GetValue().Metadata
print("Image was loaded from " .. metadata.Filename)
```

`Parameter.Name`

Friendly name of this Parameter (read-only).

→ **Getting**:

name = Parameter.Name – (string)

Methods

`Parameter.GetData([`*name*`])`

Get custom persistent data.

See Composition:GetData().

- **Parameters**:

  name (*string*) – name

- **Returns**: value
- **Return type**: (number|string|boolean|table)

`Parameter.SetData(`*name*, *value*`)`

Set custom persistent data.

See Composition:SetData().

- **Parameters**:

  name (*string*) – name

  value ((*number|string|boolean|table*)) – value

PlainInput

`class PlainInput`

Parent class: `Link`

Represents an Input.

PlainInput Attributes

Attribute Name	Type	Description
INPS_Name	string	The full name of this input.
INPS_ID	string	The script ID of this input.
INPS_DataType	string	The type of Parameter (e.g. Number, Point, Text, Image) this input accepts.
INPS_StatusText	string	The text shown on the status bar on mouse hover.
INPB_External	boolean	Whether this input can be animated or connected to a tool or modifier.
INPB_Active	boolean	This input's value is used in rendering.
INPB_Required	boolean	The tool's result requires a valid Parameter from this input.
INPB_Connected	boolean	The input is connected to another tool's Output.
INPI_Priority	integer	Used to determine the order in which the tool's inputs are fetched.
INPID_InputControl	string	The ID of the type of tool window control used by the input.

Attribute Name	Type	Description
INPID_PreviewControl	string	The ID of the type of display view control used by the input.
INPB_Disabled	boolean	The input will not accept new values.
INPB_DoNotifyChanged	boolean	The tool is notified of changes to the value of the input.
INPB_Integer	boolean	The input rounds all numbers to the nearest integer.
INPI_NumSlots	integer	The number of values from different times that this input can fetch at once.
INPB_ForceNotify	boolean	The tool is notified whenever a new parameter arrives, even if it is the same value.
INPB_InitialNotify	boolean	The tool is notified at creation time of the initial value of the input.
INPB_Passive	boolean	The value of this input will not affect the rendered result, and does not create an Undo event if changed.
INPB_InteractivePassive	boolean	The value of this input will not affect the rendered result, but it can be Undone if changed.
INPN_MinAllowed	number	Minimum allowed value - any numbers lower than this value are clipped.
INPN_MaxAllowed	number	Maximum allowed value - any numbers higher than this value are clipped.
INPN_MinScale	number	The lowest value that the input's control will normally display.
INPN_MaxScale	number	The highest value that the input's control will normally display.
INPI_IC_ControlPage	integer	Determines which tab of a tool's control window that the input's control is displayed on.
INPI_IC_ControlGroup	integer	When multiple inputs share a single compound window control, they must all have the same Control Group value.
INPI_IC_ControlID	integer	When multiple inputs share a single compound window control, they must all have different Control ID values.

Methods

`PlainInput.ConnectTo()`

Note: This method is overloaded and has alternative parameters. See other definitions.

Connect the Input to an Output.

Note that ConnectTo is not needed to connect inputs and outputs. Setting an input equal to an output behaves the same.

out is equal to an output of some sort that will be connected to the input that the function is run on. Will disconnect the input from any outputs if connected to a nil value.

→ **Python usage**:

```
mybg = comp.Background()
myblur = comp.Blur()

# Connect
myblur.Input.ConnectTo(mybg.Output)
# Disconnect
myblur.Input.ConnectTo()

# Now the same with the = operator
# Connect
myblur.Input = mybg.Output
# Disconnect
myblur.Input = None
```

→ **Lua usage**:

```
mybg = Background()
myblur = Blur()

-- Connect
myblur.Input:ConnectTo(mybg.Output)
-- Disconnect
myblur.Input:ConnectTo()
```

```

-- Now the same with the = operator
-- Connect
myblur.Input = mybg.Output
-- Disconnect
myblur.Input = nil
```

- **Returns**: success
- **Return type**: boolean

`PlainInput.ConnectTo(`*out*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Connect the Input to an Output.

Note that ConnectTo is not needed to connect inputs and outputs. Setting an input equal to an output behaves the same.

out is equal to an output of some sort that will be connected to the input that the function is run on. Will disconnect the input from any outputs if connected to a nil value.

- **Python usage**:

```
mybg = comp.Background()
myblur = comp.Blur()

# Connect
myblur.Input.ConnectTo(mybg.Output)
# Disconnect
myblur.Input.ConnectTo()

# Now the same with the = operator
# Connect
myblur.Input = mybg.Output
# Disconnect
myblur.Input = None
```

- **Lua usage**:

```
mybg = Background()
myblur = Blur()

-- Connect
myblur.Input:ConnectTo(mybg.Output)
-- Disconnect
myblur.Input:ConnectTo()

-- Now the same with the = operator
-- Connect
myblur.Input = mybg.Output
-- Disconnect
myblur.Input = nil
```

- **Parameters**:

  out (*Output*) – out

- **Returns**: success
- **Return type**: boolean

`PlainInput.GetConnectedOutput()`

Returns the output that this input is connected to.

Note by design an Input can only be connected to a single Output, while an Output might be branched and connected to multiple Inputs.

- **Returns**: out
- **Return type**: Output

`PlainInput.GetExpression()`

Returns the expression string shown within the Input's Expression field, if any, or nil if not.

Simple expressions can be very useful for automating the relationship between controls, especially in macros and commonly-used comps.

`PlainInput.GetKeyFrames()`

Return a table of all keyframe times for this input. If a tool control is not animated with a spline this function will return nil.

The GetKeyFrames() function is used to determine what frames of an input have been keyframed on a spline. It returns a table that shows at what frames the user has defined key frames for the input.

- **Returns**: keyframes
- **Return type**: table

`PlainInput.HideViewControls(`*hide*`)`

Hides or shows the view controls for this input.

Use this function to hide or expose a view control in the display view.

hide if set or true then hide the controls else show them.

- **Python usage**:

```
# Hide Center position transform controls
comp.Transform1.Center.HideViewControls()

# Show Center position transform controls
comp.Transform1.Center.HideViewControls(False)
```

- **Lua usage**:

```
-- Hide Center position transform controls
Transform1.Center:HideViewControls()

-- Show Center position transform controls
Transform1.Center:HideViewControls(false)
```

- **Parameters**:

hide (*boolean*) – hide

`PlainInput.HideWindowControls(`*hide*`)`

Hides or shows the window controls for this input.

Use this function to hide or expose a window control in the tool properties window. For instance, this could be used to hide all gamma controls on Brightness / Contrasts to prevent user manipulation.

hide if set or true then hide the controls else show them.

→ Python usage:

```
# Hide Center from properties
comp.Transform1.Center.HideWindowControls()

# Show Center in properties
comp.Transform1.Center.HideWindowControls(False)
```

→ Lua usage:

```
-- Hide Center from properties
Transform1.Center:HideWindowControls()

-- Show Center in properties
Transform1.Center:HideWindowControls(false)
```

→ Parameters:

hide (*boolean*) – hide

`PlainInput.SetExpression()`

This function reveals the expression field for the Input, and sets it to the given string.

Simple expressions can be very useful for automating the relationship between controls, especially in macros and commonly-used comps.

→ Python usage:

```
# Make Lift and Gamma relate to Gain
comp.BrightnessContrast1.Lift.SetExpression("Gain * 0.7")
comp.BrightnessContrast1.Gamma.SetExpression("Gain * 0.4")
```

→ Lua usage:

```
-- Make Lift and Gamma relate to Gain
BrightnessContrast1.Lift:SetExpression("Gain * 0.7")
BrightnessContrast1.Gamma:SetExpression("Gain * 0.4")
```

`PlainInput.ViewControlsVisible()`

Returns the visible state of the view controls for this input.

→ **Returns**: hidden

→ **Return type**: boolean

`PlainInput.WindowControlsVisible()`

Returns the visible state of the window controls for this input.

→ **Returns**: hidden

→ **Return type**: boolean

PlainOutput

`class PlainOutput`

Parent class: `Link`

Represents an Output.

PlainOutput Attributes

Attribute Name	Type	Description
OUTS_Name	string	The name of the Output
OUTS_ID	string	The Output's unique ID string
OUTS_DataType	string	The type of Parameter that this Output uses

Methods

`PlainOutput.ClearDiskCache(`*start*`, `*end*`)`

Clears frames from the disk cache.

start the frame to start purging the cache at (inclusive).

end the last frame to be purged (inclusive).

→ **Parameters**:

start (*number*) – start

end (*number*) – end

→ **Returns**: success

→ **Return type**: boolean

`PlainOutput.EnableDiskCache()`

Controls disk-based caching.

Enable Enables or disables the cache.

Path A valid path to cache the files at.

LockCache Locks the cache, preventing invalidation of existing cache files when upstream tools are modified. Use with extreme caution, as cache files may become out of date.

LockBranch Locks all upstream tools (defaults to false).

Delete Deletes the cache that might already exist at Path (defaults to false).

PreRender Render now to create the cache (defaults to true).

UseNetwork Use network rendering when prerendering (defaults to false).

Returns boolean if successful as well as a string to the path of the cache.

→ **Python usage**:

```
comp.BC1.Output.EnableDiskCache(True,“c:\\temp\\BC.0000.raw”)
```

→ **Lua usage**:

```
BC1.Output:EnableDiskCache(true,“c:\\temp\\BC.0000.raw”)
```

→ **Returns**: success

→ **Return type**: boolean

`PlainOutput.GetConnectedInputs()`

Returns a table of Inputs connected to this Output.

The GetConnectedInputs function is used to determine what inputs are using a given output.

Note by design an Input can only be connected to a single Output, while an Output might be branched and connected to multiple Inputs.

`PlainOutput.GetDoD([`*time*`][, `*flags*`][, `*proxy*`])`

Returns the Domain of Definition at the given time.

time The frame to fetch the value for (default is the current time).

reqflags Quality flags (default is final quality).

proxy Proxy level (default is no proxy).

The returned table has four integers containing the DoD of the tool’s output in the order left, bottom, right, top.

- **Parameters**:

  time (*number*) – time

  flags (*number*) – flags

  proxy (*number*) – proxy
- **Returns**: dod
- **Return type**: table

`PlainOutput.GetValue()`

Returns the value at the given time.

Useful for retrieving the result of a chain of tools. It does this by triggering a render (if cached values are not found) of all tools upstream of the Output.

time The frame to fetch the value for (default is the current time).

reqflags Quality flags (default is final quality).

proxy Proxy level (default is no proxy).

Returned value may be nil, or a variety of different types:

Number returns a number Point returns a table with X and Y members Text returns a string Clip returns the filename string Image returns an Image object

attrs is a table with the following entries:

Valid table with numeric Start and End entries DataType string ID for the parameter type TimeCost time take to render this parameter

- **Returns**: value
- **Return type**: `Parameter`

`PlainOutput.ShowDiskCacheDlg()`

Displays Cache-To-Disk dialog for user interaction.

Note this is a modal dialog. The script execution waits for the user to dismiss the dialog.

Return false if canceled, else true.

- **Returns**: success
- **Return type**: boolean

PolylineMask

`class PolylineMask`

Parent class: `MaskOperator`

Methods

`PolylineMask.ConvertToBSpline()`

Converts to b-spline polyline.

`PolylineMask.ConvertToBezier()`

Converts to Bezier polyline.

`PolylineMask.GetBezierPolyline(`*time*`[, `*which*`])`

Get a table of bezier polyline.

→ **Parameters**:

time (*number*) – time

which (*string*) – which

→ **Returns**: poly

→ **Return type**: table

Preview

`class Preview`

Parent class: `PlainInput`

Methods

`Preview.Close()`

Closes the current clip.

`Preview.Create(`*tool*`[, `*filename*`])`

Renders a new preview clip.

→ **Parameters**:

tool (*Tool*) – tool

filename (*string*) – filename

→ **Returns**: success

→ **Return type**: boolean

`Preview.DisplayImage(`*img*`)`

Displays an Image object.

→ **Parameters**:

img (*Image*) – img

→ **Returns**: success

→ **Return type**: boolean

`Preview.IsPlaying()`

Indicates if the preview is currently playing.

- **Returns**: playing
- **Return type**: boolean

`Preview.Open(`*filename*`)`

Opens a filename for seeking and playback.

- **Parameters**:
  filename (*string*) – filename
- **Returns**: success
- **Return type**: boolean

`Preview.Play([`*reverse*`])`

Plays the current clip.

- **Parameters**:
  reverse (*boolean*) – reverse

`Preview.Seek(frame)`

Seeks to specified frame.

- **Parameters**:
  frame (*number*) – frame

`Preview.Stop()`

Stops playback.

`Preview.ViewOn(tool)`

Attaches a Preview to a Tool to display its output.

- **Parameters**:
  tool (*Tool*) – tool
- **Returns**: success
- **Return type**: boolean

QueueManager

`class QueueManager`

Parent class: `LockableObject`

Represents the QueueManager.

QueueManager Attributes

Attribute Name	Type	Description
RQUEUEB_Paused	boolean	True if rendering is currently paused, and no jobs are being rendered.
RQUEUEB_Verbose	boolean	True if Verbose Logging is currently enabled.
RQUEUES_QueueName	string	The name of the file the queue has been loaded from, or saved to, if any.

→ Python usage:

```
# Access to the QueueManager
qm = fusion.RenderManager
```

→ Lua usage:

```
-- Access to the QueueManager
qm = fusion.RenderManager
```

Methods

`QueueManager.AddItem()`

AddItem

`QueueManager.AddJob(`*filename*`[, `*groups*`][, `*frames*`][, `*endscript*`])`

Note: This method is overloaded and has alternative parameters. See other definitions.

Adds a job to the list.

This function allows a user to add jobs remotely to the Fusion Render Manager, either through a standalone script or through the Fusion interface. This is potentially useful for the batch adding of jobs.

filename A valid path for a job to be added to the render manager.

groups A string listing the slave groups (comma separated) to render this job on. Defaults to "all".

frames The set of frames to render, e.g. "1..150,155,160". If nil or unspecified, the comp's saved frame range will be used.

endscript Full pathname of a script to be executed when this job has completed (available from the RenderJob object as the RJOBS_CompEndScript attribute).

Returns the RenderJob object just created in the queue manager.

- **Parameters**:

  filename (*string*) – filename

  groups (*string*) – groups

  frames (*string*) – frames

  endscript (*string*) – endscript

- **Returns**: job
- **Return type**: `RenderJob`

`QueueManager.AddJob(`*args*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Adds a job to the list.

This function allows a user to add jobs remotely to the Fusion Render Manager, either through a standalone script or through the Fusion interface. This is potentially useful for the batch adding of jobs.

filename A valid path for a job to be added to the render manager.

groups A string listing the slave groups (comma separated)
to render this job on. Defaults to "all".

frames The set of frames to render, e.g. "1..150,155,160". If nil or
unspecified, the comp's saved frame range will be used.

endscript Full pathname of a script to be executed when this job has completed
(available from the RenderJob object as the RJOBS_CompEndScript attribute).

Returns the RenderJob object just created in the queue manager.

- **Parameters**:

  args (*table*) – args

- **Returns**: job
- **Return type**: `RenderJob`

`QueueManager.AddSlave(`*name*`[, `*groups*`][, `*unused*`])`

Adds a slave to the slave list.

This function allows a user to add jobs remotely to the Fusion Render Manager, either through a standalone script or through the Fusion interface. This is potentially useful for the batch adding of jobs.

name the slave's hostname or IP address.

groups the render groups to join (this defaults to "all").

The RenderSlave object just created in the queue manager.

→ **Parameters**:

name (*string*) – name

groups (*string*) – groups

unused (*boolean*) – unused

→ **Returns**: slave

→ **Return type**: `RenderSlave`

`QueueManager.AddWatch()`

AddWatch

`QueueManager.DeleteItem()`

DeleteItem

`QueueManager.GetGroupList()`

Get a list of all slave groups.

Returns a table of all the various groups used by the slaves within this QueueManager.

→ **Returns**: groups

→ **Return type**: table

`QueueManager.GetID()`

GetID

`QueueManager.GetItemList()`

GetItemList

`QueueManager.GetJobFromID()`

GetJobFromID

`QueueManager.GetJobList()`

Get the list of jobs to render.

Returns a table with RenderJob objects that represent the jobs currently in the queue manager. Like any other object within Fusion, these objects have attributes that indicate information about the status of the object, and functions that can query or manipulate the object.

- Python usage:

```
# Print all RenderJobs in Queue.
qm = fusion.RenderManager
joblist = qm.GetJobList().values()
for job in joblist:
    print(job.GetAttrs()[“RJOBS_Name”])
```

- Lua usage:

```
-- Print all RenderJobs in Queue.
qm = fusion.RenderManager
joblist = qm:GetJobList()
for i, job in pairs(joblist) do
    print(job:GetAttrs().RJOBS_Name)
end
```

- **Returns**: jobs
- **Return type**: table

`QueueManager.GetJobs()`

Get tables with current RenderJob information.

`QueueManager.GetRootData()`

GetRootData

`QueueManager.GetSchemaList()`

GetSchemaList

`QueueManager.GetSlaveFromID()`

GetSlaveFromID

`QueueManager.GetSlaveList()`

Get the list of available slaves.

This function returns a table with RenderSlave objects that represent the slaves currently listed in the queue manager.

→ Python usage:

```
# Print all RenderSlaves in Queue.
qm = fusion.RenderManager
slavelist = qm.GetSlaveList().values()
for slave in slavelist:
    print(slave.GetAttrs()["RSLVS_Name"])
```

→ Lua usage:

```
-- Print all RenderSlaves in Queue.
qm = fusion.RenderManager
slavelist = qm:GetSlaveList()
for i, slave in pairs(slavelist) do
    print(slave:GetAttrs().RSLVS_Name)
end
```

→ **Returns**: slaves

→ **Return type**: table

`QueueManager.GetSlaves()`

Get tables with current RenderSlave information.

`QueueManager.LoadQueue(`*filename*`)`

Loads a list of jobs to do.

This function allows a script to load a Fusion Studio Render Queue file, containing a list of jobs to complete, into the queue manager.

filename path to the queue to load.

→ **Parameters**:

filename (*string*) – filename

`QueueManager.LoadSlaveList([`*filename*`])`

Loads a list of slaves to use.

- **Parameters**:

  filename (*string*) – filename

- **Returns**: success
- **Return type**: boolean

`QueueManager.Log(`*message*`)`

Writes a message to the Render Log.

Write messages to the render manager's log. This is useful for triggering custom notes for compositions submitted to the manager.

- **Parameters**:

  message (*string*) – message

`QueueManager.MoveJob(`*job*`, `*offset*`)`

Moves a job up or down the list.

Changes the priority of jobs in the render manager by an offset.

job the RenderJob to move.

offset how far up or down the job list to move it (negative numbers will move it upwards).

- **Python usage**:

```
# Moves all jobs called "master" to the top of the queue
# or at least up one hundred entries.
qm = fusion.RenderManager
jl = qm.GetJobList().values()

for job in jl:
    if "master" in job.GetAttrs()["RJOBS_Name"]:
        qm.MoveJob(job,-100)
```

- **Lua usage**:

```
-- Moves all jobs called "master" to the top of the queue
-- or at least up one hundred entries.

```

```
qm = fusion.RenderManager

jl = qm:GetJobList()

for i, job in pairs(jl) do
    if job:GetAttrs().RJOBS_Name:find(“master”) then
        qm:MoveJob(job,-100)
    end
end
```

→ **Parameters**:

job (*RenderJob*) – job

offset (*number*) – offset

`QueueManager.NetJoinRender()`

NetJoinRender

`QueueManager.RemoveJob(`*job*`)`

Removes a job from the list.

→ **Parameters**:

job (*RenderJob*) – job

`QueueManager.RemoveSlave(`*slave*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Removes a slave from the slave list.

→ **Parameters**:

slave (*RenderSlave*) – slave

`QueueManager.RemoveSlave(`*slave*`)`

Note: This method is overloaded and has alternative parameters. See other definitions.

Removes a slave from the slave list.

→ **Parameters**:

slave (*string*) – slave

`QueueManager.RemoveWatch()`

RemoveWatch

`QueueManager.SaveQueue(`*filename*`)`

Saves the current list of jobs.

filename the location to save the queue in.

This function save the currently loaded queue in the render manager to a file.

→ **Parameters**:

filename (*string*) – filename

`QueueManager.SaveSlaveList([`*filename*`])`

Saves the current list of slaves.

→ **Parameters**:

filename (*string*) – filename

→ **Returns**: success

→ **Return type**: boolean

`QueueManager.ScanForSlaves()`

Scans local network for new slaves.

This function locates all machines on the local network (local subnet only), queries each to find out if they are currently running a copy of Fusion then adds them to the manager's Slaves list.

`QueueManager.Start()`

Start

`QueueManager.Stop()`

Stop

`QueueManager.UpdateItem()`

UpdateItem

Registry

`class Registry`

Represents the registry.

Registry Attributes

Attribute Name	Type	Description
REGS_Name	string	Specifies the full name of the class represented by this registry entry.
REGS_ScriptName	boolean	Specifies the scripting name of the class represented by this registry entry. If not specified, the full name defined by REGS_ Name is used.
REGS_HelpFile	string	The help file and ID for the class.
REGI_HelpID	integer	The help file and ID for the class.
REGI_HelpTopicID	integer	The help file and ID for the class.
REGS_OpIconString	boolean	Specifies the toolbar icon text used to represent the class.
REGS_OpDescription	integer	Specifies a description of the class.
REGS_OpToolTip	boolean	Specifies a tooltip for the class to provide a longer name or description.
REGS_Category	integer	Specifies the category for the class, defining a position in the Tools menu for tool classes.
REGI_ClassType REGI_ClassType2	integer	Specifies the type of this class, based on the classtype constants.
REGI_ID	string	A unique ID for this class.
REGI_OpIconID	string	A resource ID for a bitmap to be used for toolbar images for this class.
REGB_OpNoMask	integer	Indicates if this Tool class cannot deal with being masked.

Attribute Name	Type	Description
REGI_DataType	string table	Specifies a data type RegID dealt with by this class.
REGI_TileID	number	Specifies a resource ID used for the tile image by this class.
REGB_CreateStaticPreview	integer	Indicates that a preview object is to be created at startup of this type.
REGB_CreateFramePreview	boolean	Indicates that a preview object is to be created for each new frame window.
REGB_Preview_CanDisplayImage REGB_Preview_CanCreateAnim REGB_Preview_CanPlayAnim REGB_Preview_CanSaveImage REGB_Preview_CanSaveAnim REGB_Preview_CanCopyImage REGB_Preview_CanCopyAnim REGB_Preview_CanRecord REGB_Preview_UsesFilenames REGB_Preview_CanNetRender	boolean	Defines various capabilities of a preview class.
REGI_Version	integer	Defines the version number of this class or plugin.
REGI_PI_DataSize	number	Defines a custom data size for AEPlugin classes.
REGB_Unpredictable	string	Indicates if this tool class is predictable or not. Predictable tools will generate the same result given the same set of input values, regardless of time.
REGI_InputDataType	integer	Specifies a data type RegID dealt with by the main inputs of this class.

Attribute Name	Type	Description
REGB_OperatorControl	integer	Indicates if this tool class provides custom overlay control handling.
REGB_Source_GlobalCtrls	number	Indicates if this source tool class has global range controls.
REGB_Source_SizeCtrls	integer	Indicates if this source tool class has image resolution controls.
REGB_Source_AspectCtrls	integer	Indicates if this source tool class has image aspect controls..
REGB_NoAutoProxy	boolean	Indicates if this tool class does not want things to be auto-proxied when it is adjusted.
REGI_Logo	boolean	Specifies a resource ID of a company logo for this class.
REGI_Priority	boolean	Specifies the priority of this class on the registry list.
REGB_NoBlendCtrls	boolean	Indicates if this tool class does not have blend controls.
REGB_NoObjMatCtrls	boolean	Indicates if this tool class does not have Object/Material selection controls.
REGB_NoMotionBlurCtrls	boolean	Indicates if this tool class does not have Motion Blur controls.
REGB_NoAuxChannels	boolean	Indicates if this tool class cannot deal with being given Auxiliary channels (such as Z, ObjID, etc)
REGB_EightBitOnly	boolean	Indicates if this tool class cannot deal with being given greater than 8 bit per channel images.
REGB_ControlView	boolean	Indicates if this class is a control view class.

Attribute Name	Type	Description
REGB_NoSplineAnimation	boolean	Specifies that this data type (parameter class) cannot be animated using a spline.
REGI_MergeDataType	integer	Specifies what type of data this merge tool class is capable of merging.
REGB_ForceCommonCtrls	boolean	Forces the tool to have common controls like motion blur, blend etc, even on modifiers.
REGB_Particle_ProbabilityCtrls REGB_Particle_SetCtrls REGB_Particle_AgeRangeCtrls REGB_Particle_RegionCtrls REGB_Particle_RegionModeCtrls REGB_Particle_StyleCtrls REGB_Particle_EmitterCtrls REGB_Particle_RandomSeedCtrls	boolean	Specifies that particle tools should have (or not have) various standard sets of controls.
REGI_Particle_DefaultRegion	integer	Specifies the RegID of a default Region for this particle tool class.
REGI_Particle_DefaultStyle	integer	Specifies the RegID of a default Style for this particle tool class.
REGI_MediaFormat_Priority	integer	Specifies the priority of a media format class.
REGS_MediaFormat_FormatName	string	Specifies the name of a media format class
REGST_MediaFormat_Extension	string	Specifies the extensions supported by a media format class

Attribute Name	Type	Description
REGB_MediaFormat_CanLoad REGB_MediaFormat_CanSave REGB_MediaFormat_CanLoadMulti REGB_MediaFormat_CanSaveMulti REGB_MediaFormat_WantsIOClass REGB_MediaFormat_LoadLinearOnly REGB_MediaFormat_SaveLinearOnly REGB_MediaFormat_CanSaveCompressed REGB_MediaFormat_OneShotLoad REGB_MediaFormat_OneShotSave REGB_MediaFormat_CanLoadImages REGB_MediaFormat_CanSaveImages REGB_MediaFormat_CanLoadAudio REGB_MediaFormat_CanSaveAudio REGB_MediaFormat_CanLoadText REGB_MediaFormat_CanSaveText REGB_MediaFormat_CanLoadMIDI REGB_MediaFormat_CanSaveMIDI REGB_MediaFormat_ClipSpecificInputValues REGB_MediaFormat_WantsUnbufferedIOClass	boolean	Specify various capabilities of a media format class
REGB_ImageFormat_CanLoadFields REGB_ImageFormat_CanSaveField REGB_ImageFormat_CanScale REGB_ImageFormat_CanSave8bit REGB_ImageFormat_CanSave24bit REGB_ImageFormat_CanSave32bi	boolean	Specify various capabilities of an image format class

Members

`Registry.ID`

ID of this Registry node (read-only).

→ **Getting**:

id = Registry.ID – (string)

`Registry.Name`

Friendly name of this Registry node (read-only).

→ **Getting**:

name = Registry.Name – (string)

`Registry.Parent`

Parent of this Registry node (read-only).

→ **Getting**:

parent = Registry.Parent – (Registry)

Methods

`Registry.IsClassType()`

Returns whether a tool's ID or any of its parent's IDs is a particular Registry ID.

→ **Returns**: matched

→ **Return type**: boolean

`Registry.IsRegClassType()`

Returns whether a tool is a particular Registry ClassType.

→ **Returns**: matched

→ **Return type**: boolean

RenderJob

`class RenderJob`

Parent class: `Object`

Represents a RenderJob.

RenderJob Attributes

Attribute Name	Type	Description
RJOBS_Status	string	The current status of the job as String.
RJOBB_Resumable	boolean	
RJOBS_CompEndScript	string	
RJOBN_CompID	number	
RJOBS_QueuedBy	string	
RJOBB_IsRemoving	boolean	
RJOBB_Paused	boolean	Indicates if the Job is paused.
RJOBS_Name	string	The filename of the Job.
RJOBB_DontClose	boolean	
RJOBN_TimeOut	number	The timeout of the job in minutes.
RJOBN_Status	number	Legacy status indicator for scripts that were reliant on the old numeric index for job status. 0. Not Rendered 1. Incomplete 2. Done 3. Failed 4. Paused 5. Submitted 6. Rendering 7. Aborting
RJOBN_RenderingFrames	number	The number of currently rendering frames.
RJOBN_RenderedFrames	number	The number of frames rendered in the job.
RJOBID_ID	string	The UUID of the job for Fusion's internal tracking.

→ Python usage:

```
# Adds the current composition as new job
# and print all RenderJobs in Queue.
qm = fusion.RenderManager

qm.AddJob(comp.GetAttrs()[“COMPS_FileName”])
joblist = qm.GetJobList().values()
for job in joblist:
    print(job.GetAttrs()[“RJOBS_Name”])
```

→ Lua usage:

```
-- Adds the current composition as new job
-- and print all RenderJobs in Queue.
qm = fusion.RenderManager

qm:AddJob(comp:GetAttrs().COMPS_FileName)

joblist = qm:GetJobList()
for i, job in pairs(joblist) do
    print(job:GetAttrs().RJOBS_Name)
end
```

Methods

`RenderJob.ClearCompletedFrames()`

Clears the list of completed frames, restarting the render.

`RenderJob.GetFailedSlaves()`

Lists all slaves that failed this job.

This function returns a table containing all slaves that were assigned to this job but have been unable to load the comp, or to render a frame that was assigned to them.

These slaves are no longer participating in the job, but can be added back to the job by using RetrySlave().

- **Returns**: failedslaves
- **Return type**: table

`RenderJob.GetFrames()`

Returns the total set of frames to be rendered.

- **Returns**: frames
- **Return type**: string

`RenderJob.GetRenderReport()`

GetRenderReport

`RenderJob.GetSlaveList()`

Gets a table of slaves assigned to this job.

- **Returns**: slaves
- **Return type**: table

`RenderJob.GetUnrenderedFrames()`

Returns the remaining frames to be rendered.

The frames in the returned string is separated by commas. Contiguous frames are given as a range in the form <first>..<last>.

- **Returns**: frames
- **Return type**: string

`RenderJob.IsRendering()`

Returns true if job is currently rendering.

- **Returns**: rendering
- **Return type**: boolean

`RenderJob.RetrySlave([slave])`

Attempts to reuse slaves that have previously failed.

The job manager will place them back on the active list for the job, and attempt to assign frames to them again.

slave a RenderSlave object, assigned to this job, that has previously failed to render a frame assigned to it. If slave is not specified, all failed slaves will be retried.

- **Parameters**:

slave (*RenderSlave*) – slave

RenderJob.SetFrames(*frames*)

Specifies the set of frames to render.

frames a string with valid formatting for frames to be rendered by the job. Frame numbers should be separated by commas, without spaces, and ranges of frames are denoted by <first>..<last>.

→ Python usage:

```
# Set the frames to render on the first job in queue
job = fusion.RenderManager.GetJobList()[1]

job.SetFrames("1..50,55,60,75,80..100")
```

→ Lua usage:

```
-- Set the frames to render on the first job in queue
job = fusion.RenderManager:GetJobList()[1]

job:SetFrames("1..50,55,60,75,80..100")
```

→ Parameters:

frames (*string*) – frames

RenderJob._Heartbeat()

_Heartbeat

RenderSlave

class RenderSlave

Parent class: `LockableObject`

Represents a RenderSlave.

RenderSlave Attributes

Attribute Name	Type	Description
RSLVS_Status	string	The current status of the slave.
RSLVN_Status	number	The current status of the slave as number. 0. Scanning 1. Idle 2. Failed 3. Busy 4. Assigning Job 5. Connecting 6. Checking Settings 7. Loading Comp 8. Starting Render 9. Rendering 10. Ending Render 11. Disconnecting 12. Offline 13. Disabled 14. Unused
RSLVS_IP	string	The IP address of the slave machine.
RSLVID_ID	string	The ID of the job.
RSLVS_Name	string	The network name of the slave being used.
RSLVB_IsUnused	boolean	Indicates if the slave is unused.
RSLVS_Version	string	The version number of the slave.
RSLVS_Groups	string	The assigned group of the slave.
RSLVN_RenderingComp	number	The comp ID number that it's currently rendering.
RSLVB_IsRemoving	boolean	If the slave is being removed from the queue.
RSLVB_IsFailed	boolean	If the slave has failed enough times to remove it from further jobs.

- Python usage:

```
# Print all RenderSlaves in Queue.
qm = fusion.RenderManager
slavelist = qm.GetSlaveList().values()
for slave in slavelist:
    print(slave.GetAttrs()
```

- Lua usage:

```
-- Print all RenderSlaves in Queue.
qm = fusion.RenderManager
slavelist = qm:GetSlaveList()
for i, slave in pairs(slavelist) do
    print(slave:GetAttrs().RSVLS_Name)
end
```

Methods

`RenderSlave.Abort()`

Cease rendering, and quit the current job.

`RenderSlave.GetJob()`

Return the slave's current RenderJob object, if any.

`RenderSlave.IsDisconnecting()`

True if slave is disconnecting from a job.

Sometimes when a slave is disconnecting from the render manager object, it will take a few seconds to actually disconnect. During this time, it will not show up interactively in the Render Manager's slave list, however, it will show up in the table returned by GetSlaveList(). As such, this function was added to easily tell if a RenderSlave is currently disconnecting.

Returns a boolean value indicating whether the slave's RSLVB_IsDisconnecting attribute is currently set to false.

`RenderSlave.IsIdle()`

True if slave has no job and nothing to do.

Returns a boolean value indicating whether the slave's RSLVB_IsIdle attribute is currently set to false.

`RenderSlave.IsProcessing()`

True if slave is busy.

Returns a boolean value indicating whether the slave is currently processing a frame.

ScriptServer

`class ScriptServer`

Methods

`ScriptServer.AddHost()`

AddHost

`ScriptServer.Connect()`

Connect

`ScriptServer.FindHost()`

FindHost

`ScriptServer.GetHostList()`

GetHostList

`ScriptServer.RemoveHost()`

RemoveHost

`ScriptServer.StartHost()`

StartHost

SourceOperator

`class SourceOperator`

Parent class: `ThreadedOperator`

TimeRegion

`class TimeRegion`

Parent class: `List`

Members

`TimeRegion.End`

- → **Getting**:

  val = TimeRegion.End – (number)

`TimeRegion.Start`

- → **Getting**:

  val = TimeRegion.Start – (number)

Methods

`TimeRegion.FromFrameString(`*frames*`)`

Reads a string description.

- → **Parameters**:

  frames (*string*) – frames

`TimeRegion.FromTable(`*frames*`)`

Reads a table of {start, end} pairs.

- → **Parameters**:

  frames (*table*) – frames

`TimeRegion.ToFrameString()`

Returns a string description.

- → **Returns**: frames
- → **Return type**: string

`TimeRegion.ToTable()`

Returns a table of {start, end} pairs.

- → **Returns**: frames
- → **Return type**: table

TransformMatrix

class `TransformMatrix`

Parent class: `Parameter`

Members

`TransformMatrix.Depth`

Image depth indicator (not in bits) (read-only).

→ Getting:

val = TransformMatrix.Depth – (number)

`TransformMatrix.Field`

Field indicator (read-only).

→ Getting:

val = TransformMatrix.Field – (number)

`TransformMatrix.Height`

Actual image height, in pixels (read-only).

→ Getting:

val = TransformMatrix.Height – (number)

`TransformMatrix.OriginalHeight`

Unproxied image height, in pixels (read-only).

→ Getting:

val = TransformMatrix.OriginalHeight – (number)

`TransformMatrix.OriginalWidth`

Unproxied image width, in pixels (read-only).

→ Getting:

val = TransformMatrix.OriginalWidth – (number)

`TransformMatrix.OriginalXScale`

Unproxied pixel X Aspect (read-only).

→ Getting:

val = TransformMatrix.OriginalXScale – (number)

`TransformMatrix.OriginalYScale`

Unproxied pixel Y Aspect (read-only).

→ Getting:

val = TransformMatrix.OriginalYScale – (number)

`TransformMatrix.ProxyScale`

Image proxy scale multiplier (read-only).

→ Getting:

val = TransformMatrix.ProxyScale – (number)

`TransformMatrix.Width`

Actual image width, in pixels (read-only).

→ Getting:

val = TransformMatrix.Width – (number)

`TransformMatrix.XOffset`

Image X Offset (read-only).

→ Getting:

val = TransformMatrix.XOffset – (number)

`TransformMatrix.XScale`

Pixel X Aspect (read-only).

→ Getting:

val = TransformMatrix.XScale – (number)

`TransformMatrix.YOffset`

Image X Offset (read-only).

→ Getting:

val = TransformMatrix.YOffset – (number)

`TransformMatrix.YScale`

Pixel Y Aspect (read-only).

→ Getting:

val = TransformMatrix.YScale – (number)

***

Index

3