



# Scripting Manual

Version 1.1

# Table of Contents

Scripting Manual.....	1
Introduction:.....	5
What is eyeonScript?.....	5
Examples of useful scripts.....	5
eyeonScript filenames.....	5
Principles:.....	6
eyeonScript : eyeonScript and Lua.....	6
eyeonScript : eyeonScript and SciTe.....	6
eyeonScript in Generation.....	7
Conventions.....	7
Navigational Hierarchy Chart.....	8
Working with Scripts.....	9
Script Editor.....	9
Scripts Menu.....	10
Language Reference.....	11
Application object.....	11
App:Project().....	11
App:ProjectLoad().....	11
App:ProjectActivate().....	11
App:ProjectCreate().....	11
App:ProjectDelete().....	11
App:LoadPlayer().....	12
App:LoadSub().....	12
App:PlayControl().....	12
App:RenderControl().....	12
App:FarmControl().....	12
App:OpenView().....	12
App:Log().....	13
App:DropMousePattern().....	13
App:DoubleSlashes().....	13
Project object.....	14
Project:Name().....	14
Project:FPS().....	14
Project:Width().....	14
Project:Height().....	15
Project:Path().....	15
Project:PathData().....	15
Project:MediaPath().....	15
Project:DropFiles().....	15
Project:DropSequence().....	15
Project:DropFolder().....	15
Project:DropPattern().....	16
Project:DropPatternFusion().....	16
Project:SubCreate().....	16
Project:SubCount().....	16
Project:SubGet().....	16
Project:SubDelete().....	17
Project:ItemCreate().....	17
Project:ItemGet().....	17
Project:AudioLoad().....	17
Project:AudioSlip().....	17
Project:SelectionClear().....	17
Project:SelectionAdd().....	17
Project:SelectedThumbs().....	17

Project:ThumbToItem()	18
Project:ThumbClipIndex()	18
Project:ThumbVersionIndex()	18
Project:PlayerLayoutControl()	18
Sub object	19
Sub:SubGet()	19
Sub:Length()	19
Sub:TrackCreate()	20
Sub:TrackCount()	20
Sub:TrackGet()	20
Sub:TrackDelete()	20
Track object	21
Track:Name()	21
Track:Length()	21
Track:ItemInsert()	21
Track:ClipCount()	21
Track:ClipGet()	22
Track:ClipDelete()	22
Clip object	23
Clip:Length()	23
Clip:InPoint()	23
Clip:OutPoint()	23
Clip:VersionInPoint()	23
Clip:VersionCount()	23
Clip:VersionInsert()	24
Clip:VersionDelete()	24
Clip:VersionGet()	24
Clip:VersionUserFrame()	24
Clip:GlobalFrame2ClipFrame()	24
Item object	25
Item:Name()	25
Item:Filename()	25
Item:CreationInfo()	25
Item:CreationUser()	25
Item:Settings()	26
Item:LUT()	26
Item:RefCount()	26
Item:RefGet()	26
Item:RefFilename()	26
Item:RefAudioFilename()	26
RenderControl object	27
RenderControl:QueueCreate()	27
RenderControl:QueueCount()	27
RenderControl:QueueGet()	27
RenderControl:QueueDelete()	27
RenderControl:Render()	27
RenderControl:ProxyGroupGet()	27
Queue object	28
Queue:SourceSet()	28
Queue:OutputCreate()	28
Queue:OutputCount()	28
Queue:OutputGet()	28
QueueOutput object	29
QueueOutput:Width()	29
QueueOutput:Height()	29
QueueOutput:Quality()	29
QueueOutput:InterFace()	29

QueueOutput:Format()	29
QueueOutput:FrameRate()	29
QueueOutput:ProcessAudio()	29
QueueOutput:RescaleMode()	30
QueueOutput:FrameStart()	30
QueueOutput:FrameEnd()	30
QueueOutput:Filename()	30
ProxyGroup object	31
ProxyGroup:ProxyPathDepth()	31
ProxyGroup:ProxyPathClear()	31
ProxyGroup:ProxyPathAdd()	31
ProxyGroup:ProxyPathCount()	31
ProxyGroup:ProxyGroup:ProxyPathGet()	31
RenderQueue Example Script	32
FarmControl object	33
FarmControl:ManagerOpen()	33
FarmControl:Submit()	33
PlayerLayoutControl object	34
PlayerLayoutControl:OpenTool()	34
PlayerLayoutControl:LayoutActivate()	34
PlayerLayoutControl:LayoutCreate()	34
PlayerLayoutControl:LayoutGet()	34
PlayerLayoutControl:LayoutDelete()	34
PlayerLayoutControl:LayoutCount()	34
Layout Object	35
Layout:Name()	35
Layout:ViewerCreate()	35
Layout:ViewerGet()	35
Layout:ViewerCount()	35
Viewer Object	36
Viewer:Position()	36
Viewer:SourceReferenceName()	36
Viewer:SourceItem()	36
Viewer:FrameOffset()	36
Appendix	37
Lua License Agreement	37
SciTe License Agreement	37

# Introduction:

## What is eyeonScript?

As in Fusion, eyeon's Flagship compositing system, Generation offers a scripting environment as a way of rapidly creating new capabilities and for automating repetitive time consuming tasks. Scripts can be written to help solve the problems unique to your own projects and workflow. Properly used, scripting can dramatically change the way you work.

Scripts can be used to automate simple tasks, like moving all selected items into a new sub project. Or it can be used for far more complex tasks, such as building a dailies list of nightly renders for review or automatically creating a new sub project each monday which has all shots and versions that are due for delivery that week.

Creating and writing scripts is not a necessity. An artist using Generation does not ever need to become proficient in scripting. Indeed, it is hardly necessary to know that Generation even has scripting capabilities to use the software to its full extent.

Nevertheless, scripts provided with the software, obtained from eyeon's website, written internally by programmers or other Generation operators can be run from within the interface without the user ever becoming aware of the detailed logic behind the script.

An artist or programmer who embraces scripting, however, can quickly and easily create solutions to complex issues that arise in the post production world with a minimum of set up time or fuss.

## Examples of useful scripts

The following are examples drawn from real world existing usage of eyeonScript in production environments:

- Create a new sub project and move all selected items into there.
- Assign selected items to the renderqueue.
- Create a Fusion composition from a selected filmclip and launch Fusion to work on that shot.
- Automatically build a dailies list of nightly renders for review.
- From the project view render individual quicktime movies for each shot.
- Create a subproject for the VFX supervisor each Monday which hosts all shots and versions due for delivery that week.

## eyeonScript filenames

Any text file containing eyeonScript commands can be executed from the command line with Scriptsend.exe or interactively in Generation. Files with scripting commands for use with Generation are identified by the use of the .lua extension in the filename, for example, scriptname.lua.

Scripts without the .lua extension can still be run from the command line using Scriptsend.exe, but are not available from within the Generation interface.

# Principles:

## eyeonScript : eyeonScript and Lua

Structurally and syntactically, eyeonScript is similar to Visual Basic, C++ and other object oriented languages, without much of the additional complexity that object oriented languages like these impose. The similarities between eyeonScript and other commonly used languages makes it easy for anyone who knows the basics of another modern programming languages to quickly pick up on the basic syntax, and to read existing sample scripts with little to no training.

eyeonScript is in fact based on the Lua language, a powerful light-weight programming language designed by TecGraf for extending applications. More information on Lua, and TecGraf, can be obtained at [www.lua.org](http://www.lua.org). Lua was chosen as the basis for eyeonScript because it is powerful but small, because it was designed from the ground up to extend existing applications in a simple and straightforward manner, and because it completely platform independent.

One of the advantages to using Lua as the base for eyeonScript is that Lua can easily be embedded into the other applications used in your facility, particularly in house applications critical to your production pipeline. If you believe that Lua may be of additional use to you, visit the Lua website for more details. You can also contact [sdk@eyeonline.com](mailto:sdk@eyeonline.com) for additional details on incorporating eyeonScript into your own application.

As per the requirements of the Lua license, the license agreement is attached at the end of this document.

## eyeonScript : eyeonScript and SciTe

While developing eyeonScript it quickly became apparent that a more mature text editor for creating and modifying scripts than Notepad.exe would be useful to have. We have chosen to include the SciTe text editor with eyeonScript, because of its flexibility and broad feature set. This editor is pre-installed with Generation 5.0 if you perform a default installation or select it as an option in a custom installation. eyeon Software has made several modifications to the versions of SciTe included with Generation, enabling support for Lua and eyeonScript syntax. Source code for the changes made can be requested by sending email to [sdk@eyeonline.com](mailto:sdk@eyeonline.com).

You can find the executables and help files for SciTe under the Generation \ Utilities \ SciTe directory. A shortcut for SciTe will also be found in the Start > Programs > Generation menu if this editor has been installed. If you chose to use this editor for working with your scripts open the global scripting preferences in Generation and change the default editor from notepad.exe to the path for the SciTe executable. SciTe is included under license (see Appendix). For more information about SciTe please visit [www.scintilla.org](http://www.scintilla.org).

## eyeonScript in Generation

In Generation you can address specific objects. These are organized in a very straight forward hierarchy. The hierarchy to access an item can be imagined like so:

**App()** - The application. Generation itself.

**Project()** - A list of projects in the application. This can hold multiple Subs.

**Sub()** - A list of subs in the project. This can hold multiple Tracks.

**Track()** - A list of tracks in the Sub. This can hold multiple Clips.

**Clip()** - A list of time slots in a track. This can hold multiple Items for versioning.

**Item()** - An item in a time slot.

Please see the navigational hierarchy chart on the next page as well.

Since each object with lower hierarchy than App():Project() can hold multiple content (e.g. a Project can hold multiple Subs, each Sub can hold multiple Tracks, etc.), the methodology to create handles to those Objects uses the Get function.

So to establish a handle to a specific item you have to use

**MyItem = App():Project():SubGet():TrackGet():ClipGet():ItemGet()**

Of course you can build your own handles for the objects.

A Project handle could look like this:

MyPrj=App():Project()

A handle to the current track could look like this:

MyTrack=App():Project():SubGet():TrackGet()

And you can freely combine any of the above:

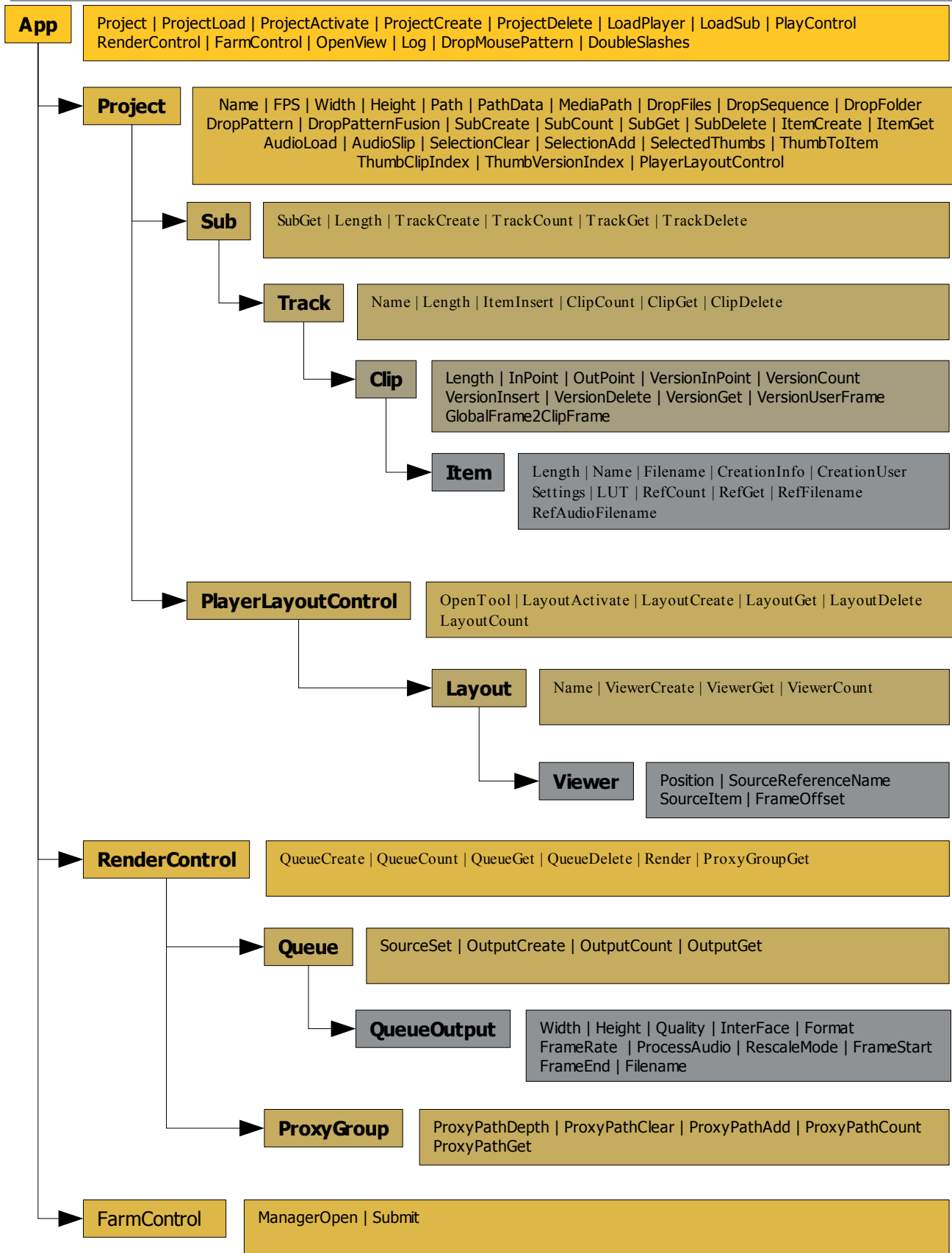
MyPrj=App():Project()

MyTrack=MyPrj:SubGet():TrackGet()

## Conventions

- [] Square brackets specify an optional value. This can be given to set a property. If it is left out the property is read rather than set.
- <> Pointed brackets specify an obligatory value. This value must be given.
- My The "My" suffix is used throughout this document to clearly distinguish user-definable variables from system variables, objects or members.
- | The pipe character indicates optional values. Basically read this as "either/or".

# Navigational Hierarchy Chart

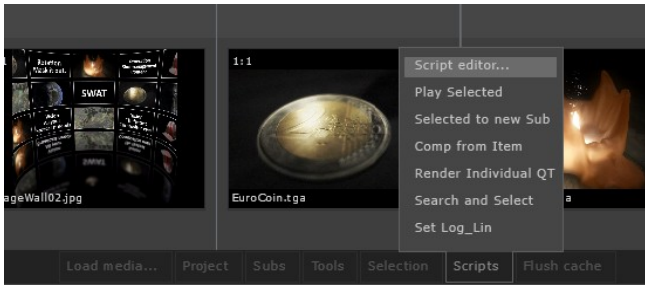


## Working with Scripts

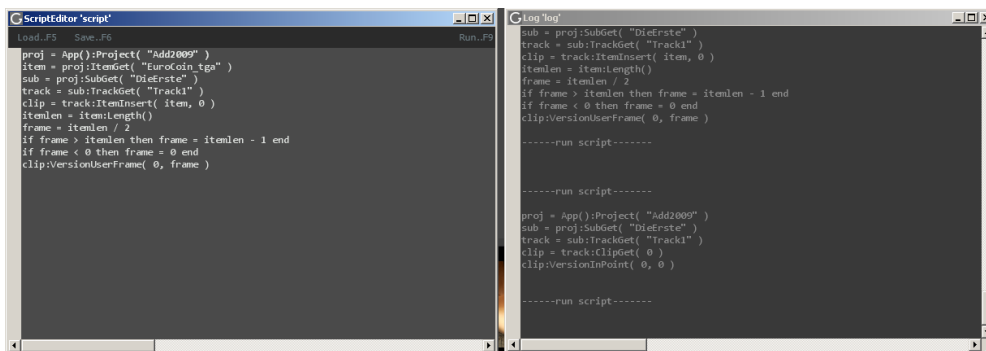
There are multiple ways to write, evaluate and apply scripts in Generation. Namely the build in Script editor and Log window, an external text editor like SciTe and adding scripts directly to the scripts menu.

### Script Editor

To open the Script editor and Log windows you can either press Ctrl-Shift-L or click on the [Scripts] button and select Script editor as shown in the screenshot below.



In either case two windows will open in your Generation interface. In the top window, the script editor, you can write, load, evaluate, modify and save scripts directly. The log window can be used to output messages using the `App():Log("message")` funktion. Anything you do in the Generation interface, like moving clips around, adding new versions etc. is also reflected in the Log window as script commands. You can copy these lines and use them in your own scripts.



### Example:

Open Script editor and Log window and copy a clip in your project. Depending on the name of your project, clip, etc. you will see something similar to this in your log-window:

```
proj = App():Project( "Add2009" )
item = proj:ItemGet( "EuroCoin_tga" )
sub = proj:SubGet( "Main" )
track = sub:TrackGet( "Track1" )
clip = track:ItemInsert( item, 0 )
itemlen = item:Length()
frame = itemlen / 2
if frame > itemlen then frame = itemlen - 1 end
if frame < 0 then frame = 0 end
clip:VersionUserFrame( 0, frame )
```

Press Ctrl-Z to undo your last action, copy the lines from the Log window and paste them in the Script editor. Now run the script by pressing F9 or clicking on the [Run] button in the Script editor's title bar. The very same action you performed with your mouse earlier is now carried out by the script. This is a huge help for creating your own scripts, as it allows you to perform your desired action first, then evaluate and modify the scripting commands to roll your own scripts out of that.

### Scripts Menu

To add or remove Scripts to or from the scripts menu you have to edit the "custom.cfg" file which can be found in your Generation install folder. The lines for the Scripts menu look like this:

```
SCRIPT_MENU=<"name">,<"PathToScript">
```

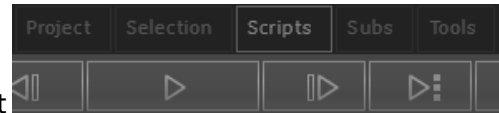
For example

```
SCRIPT_MENU="Play Selected", "scripts/generation/Play_Selection.lua"
```

The first string defines the name that is displayed in the menu, the second string defines the path to the actual script. Note that forward slashes are used here for compatibility reasons. Relative paths can be used as in the example above, where the folder "custom.cfg" sits in is used as the master folder.

### Script Buttons

As of Generation II scripts can also be organized and called using the toolbar. The scripts in here are defined by copying them into the respective folders of your Generation install. So for Scripts that would be [Generation]\Scripts\Generation\toolbar\Scripts\.



New buttons can be created by making a new folder in [Generation]\Scripts\Generation\toolbar\.

For example to add a button labelled "[custom]", create a folder named "[custom]" in said directory and put scripts in there for easy access.

The Scripts are alphabetically sorted.

# Language Reference

## Application object

The Application Object represents Generation itself.

Establish a handle to the Application:

```
MyApp = App()
```

### **App:Project()**

Get the active or find a specific project by name

*Example 1: Get the active Projekt:*

```
MyProject = App():Project()
```

*Example 2: Find the Project with the name of Bob:*

```
MyProject = App():Project("Bob")
```

### **App:ProjectLoad()**

Load a project with a given filename.

*Example: Load the Project with the name of Bob:*

```
project = App():ProjectLoad( "C:\\MyProjects\\Bob" )
```

### **App:ProjectActivate()**

Activates a loaded Project.

*Example: Activate the Project with the name of Bob:*

```
project = App():ProjectActivate( "Bob" )
```

### **App:ProjectCreate()**

Create a new project with a given name.

*Example: Create a new Project with the name of Bob:*

```
project = App():ProjectCreate( "Bob" )
```

### **App:ProjectDelete()**

Deletes an existing project

*Example: Delete the Project with the name of Bob:*

```
App():ProjectDelete("Bob")
```

### **App:LoadPlayer()**

Loads a specified item, sub, track or layout into the player and switches to playview.

Usage: App():LoadPlayer( [item]|[sub]][track]|[layout] )

*Example: Play the Sub named Barbara:*

```
MyProj=App():Project()
MySub=MyProj:SubGet( "Barbara" )
App():LoadPlayer(MySub)
```

### **App:LoadSub()**

Loads a track or a Sub with default track to the player. This will switch to the previous Sub editor if no parameter is given.

Usage: App():LoadSub([name])

*Example: Select and load the Sub named Barbara:*

```
MyProj = App():Project()
MySub = MyProj:SubGet( "Barbara" )
App():LoadSub( MySub )
```

### **App:PlayControl()**

Allows to control the playback controls via Script.

Usage: App():PlayControl( <"play"|"stop"|"set">, [frame] )

*Example: Set the current time to Frame 75:*

```
App():PlayControl("set", 75 )
```

*Example: Play the current Track:*

```
App():PlayControl("play")
```

### **App:RenderControl()**

Establishes a handle to the RenderControl Object.

Please see the chapter [RenderControl](#) in this manual.

```
Usage: MyRC = App():RenderControl()
```

### **App:FarmControl()**

Establishes a handle to the farming control.

Please see the chapter [FarmControl](#) in this manual.

```
Usage: MyFC = FarmControl()
```

### **App:OpenView()**

Switch to storyboard or playview

```
Usage: OpenView( <"storyboard"> | <"player"> )
```

### **App:Log()**

Print to the Log console.

Usage:     App():Log( <string>, ... )

*Example: Print "Hello World" to the Log console:*

```
App():Log( "Hello World" )
```

*Example: Concatenate a variable and print to the Log console:*

```
MyValue=75  
App():Log( "The Value has a mightyness of: "..MyValue )
```

*Example: Concatenating will fail if a variable is NIL.*

*We can use a comma instead to print warning messages e.g.*

```
App():Log( "The Value has a mightyness of: ",MyValue )
```

### **App:DropMousePattern()**

Switches to Cut editor and attaches the specified pattern to the mousepointer, so that it can be dropped on the timeline. This is particularly useful when the script is called by external applications (like Fusion e.g.) to send a new comp to Generation and allow the user to place it where he needs it.

"Pattern" in this case refers to a filename pattern like: "C:\\Projects\\Bob\\seq\\sequence\_1-100#.tif"

1-100# defines the frame range pattern, in this case frames 1-100 are loaded.

You can also define specific frames like 1-10,55,78,90-100#.tif, which then loads frames 1 to 10, 55, 78 and 90 to 100.

Make sure to define the path with double backslashes for Generation to understand it.

Usage:     App():DropMousePattern( <string with pattern> )

### **App:DoubleSlashes()**

Fixes paths to have double backslashes so it can be used as a LUA string.

The backslash "\" character used in Windows paths is a special command in almost all languages.

Therefore you have to define paths in LUA like "C:\\Projects\\Bob\\" whereas Windows expects

"C:\Projects\Bob\".

To help converting between these two standards you can use the DoubleSlashes function.

Usage:     LUAstring = DoubleSlashes( <string> )

*Try the following example and watch the Log output:*

```
MyPath=App():Project():Path()  
App():Log(MyPath)  
LuaPath=App():DoubleSlashes(MyPath)  
App():Log(LuaPath)
```

## Project object

The Project Object is the Generation Project. This can either be already active or loaded from disk using the appropriate commands. A Project can hold multiple Subprojects, so-called Subs.

Establish a handle to the Project:

**MyPrj = App():Project()**

**NOTE:**

*In the following examples we will refer to MyPrj as the handle above.*

*So instead of MyPrj:Name() you could also use App():Project():Name() if you have not set the handle for convenience.*

### **Project:Name()**

Get or set the Name for the Project

Read: MyName = MyPrj:Name()

Write: MyPrj:Name( [new name] )

*Example: Print the project's name to the console:*

```
MyPrj = App():Project()
MyName = MyPrj:Name()
App():Log(MyName)
```

### **Project:FPS()**

Get or set project's playback speed in Frames per Second

Read: fps = MyPrj:FPS()

Write: MyPrj:FPS( [fps] )

*Example: Print the project's playback speed to the console:*

```
MySpeed = MyPrj:FPS()
App():Log(MySpeed)
```

*Example: Set the project's playback speed to 30 FPS:*

```
MyPrj:FPS(30)
```

### **Project:Width()**

Get or set the project's default output width in pixels.

Read: MyWidth = MyPrj:Width()

Write: MyPrj:Width( [width] )

*Example: Print the project's default Width to the console:*

```
MyWidth = MyPrj:Width()
App():Log(MyWidth)
```

*Example: Set the project's default Width to 2048 pixels:*

```
MyPrj:Width(2048)
```

### **Project:Height()**

Get or set the project's default output height in pixels.

Read: MyHeight = MyPrj:Height()

Write: MyPrj:Height( [height] )

### **Project:Path()**

Get the path where the project file is stored on disk.

Usage: MyPath = MyPrj:Path()

### **Project:PathData()**

Get path to where the project .data folder is stored on disk.

Usage: MyPath = MyPrj:PathData()

### **Project:MediaPath()**

Get or set the media folder for the project. All media used from these paths will use a relative filenames.

Read: MyMediapath = MyPrj:MediaPath()

Write: MyPrj:MediaPath( [media path] )

### **Project:DropFiles()**

Drop file(s) as item(s) to the timeline

This is similar to DropMousePattern but does not attach the files to the mousepointer but gives you an item object. You can then use VersionInsert() to actually drop the item on the timeline.

Usage: MyItem = MyPrj:DropFiles( <filename>|<string array of filenames> )

*Example: Get an Item object and drop it on the timeline:*

```
MyFile=App():DoubleSlashes([[C:\Projects\Bob\Seq\Image01.tif]])
```

```
-- you could also put in the path like this to avoid using DoubleSlashes():
```

```
-- MyFile="C:\\Projects\\Bob\\Seq\\Image01.tif"
```

```
MyItem=MyPrj:DropFiles(MyFile)
```

```
MyPrj:SubGet():TrackGet():ClipGet(0):VersionInsert(MyItem)
```

### **Project:DropSequence()**

Drop a single file and find a sequence for it from the same folder. As with DropFiles() you have to use VersionInsert() to put the item on the timeline. Please refer to the example given in DropFiles().

Usage: MyItem = MyPrj:DropSequence( <"filename"> )

### **Project:DropFolder()**

Drops an entire folder as an item. Currently this only returns the first sequence found in the folder.

As with DropFiles() you have to use VersionInsert() to put the item on the timeline. Please refer to the example given in DropFiles().

Usage: MyItem = MyPrj:DropFolder( <folder path> )

### **Project:DropPattern()**

Drops a filename pattern as an item.

As with DropFiles() you have to use an additional function to put the item on the timeline. Please refer to the example given in DropFiles().

Usage: MyItem = MyPrj:DropPattern( <string with pattern> )

There are different pattern formats:

1-100#	loads frames 1 to 100 from the sequence
1,20,55	loads the three frames 1, 20 and 55 from the sequence. The order can be changed as well.
45-200@@@@@	loads frames 45-200, automatically expanding the pattern to 5 digits. So that the actual filename would be Image00045-Image00200.

It is also possible to use any combination of the above like 1-50#,67,75,90-105#

### **Project:DropPatternFusion()**

Drops a Fusion formatted filename pattern as item. The .comp filename is used to fill the COMP: string in the filename.

Usage: MyItem = MyPrj:DropPatternFusion( <.comp Filename>, <string with pattern>, [first frame] )

## Sub functions

### **Project:SubCreate()**

Creates a new Sub.

Usage: MySub = MyPrj:SubCreate( <"name"> )

Note that each Sub needs at least one Track. So creating the Sub only does not show it in the Sub menu. You will have to create a Track in that Sub as well in order for it to show up. Please see the example.

*Example: Create a new Sub and Track:*

```
MyPrj=App():Project()
MySub = MyPrj:SubCreate( "Bob" )
MyTrack = MySub:TrackCreate( "Bob" )
```

### **Project:SubCount()**

Returns the amount of Subs in the Project.

Usage: MyCount = MyPrj:SubCount()

### **Project:SubGet()**

Establishes a handle to a Sub by index or name. It will take the current Sub from the editor if no parameter is given.

Usage: MySub = MyPrj:SubGet( [index][["name"]] )

### **Project:SubDelete()**

Delete a Sub by index, name or Sub object.

Usage: MyPrj:SubDelete( <index>|<"name">|<sub object> )

## Item functions

### **Project:ItemCreate()**

Creates an empty item.

Usage: MyItem = MyPrj:ItemCreate()

### **Project:ItemGet()**

Finds an item by index or name.

Usage: MyItem = MyPrj:ItemGet( <index>|<"name"> )

## Audio functions

### **Project:AudioLoad()**

Adds an audio file to the project.

Usage: MyPrj:AudioLoad( <filename> )

### **Project:AudioSlip()**

Gets or sets the audio slip in seconds

Usage: MySecs = MyPrj:AudioSlip( [seconds] )

## Selection functions

### **Project:SelectionClear()**

Clears all user selections.

Usage: MyPrj:SelectionClear()

### **Project:SelectionAdd()**

Add media item to the current selection. Creates a new selection if nothing is selected yet.

Usage: MyPrj:SelectionAdd( <cut index>, <version index> )

### **Project:SelectedThumbs()**

Returns a table of all selected thumbs. Note that these are no item objects. Use ThumbToItem() to convert a thumb to an Item Object.

Usage: MyThumbs = MyPrj:SelectedThumbs()

**Project:ThumbToItem()**

Convert a thumbnail into a referenced Item Object.

Usage:      MyItem = MyPrj:ThumbToItem( <thumb> )

**Project:ThumbClipIndex()**

Returns the clip index for a thumb.

Usage:      MyClipIndex = MyPrj:ThumbClipIndex()

**Project:ThumbVersionIndex()**

Returns the version index for a selected thumb.

Usage:      MyVersionIndex = MyPrj:ThumbVersionIndex()

Player Functions

**Project:PlayerLayoutControl()**

Gets the Player layout controls for the current project.

Usage:      MyPLC= MyPrj:PlayerLayoutControl()

## Sub object

A Sub is a Subproject of a Generation Project. It can host multiple Tracks.

### NOTE:

*Each object with lower hierarchy than App():Project() can be multiple.  
e.g. the Application can only hold one Project, but the Project can hold multiple Subs.  
Each Sub can hold multiple Tracks, each Track multiple Clips etc.  
That's why the Sub object uses the SubGet() methodology to address subobjects.  
The same counts for TrackGet(), ClipGet() and VersionGet().*

Establish a handle to the current Sub:

**MySub=App():Project():SubGet()**

### NOTE:

*In the following examples we will refer to MySub as the handle above.  
So instead of MyPrj:Name() you could also use App():Project():SubGet():Name() if you have not set the handle for convenience.*

### **Sub:SubGet()**

Returns or sets the name of the current Sub.

Read: MyName = MySub:Name()

Write: MySub:Name( [new name] )

*Example 1: Print the current Sub's name to the Log*

```
MySub=App():Project():SubGet()
MyName = MySub:Name()
App():Log(MyName)
```

*Example 2: Rename the current Sub to "Bob"*

```
MySub=App():Project():SubGet()
MySub:Name("Boob")
```

### **Sub:Length()**

Returns the length of the current sub in frames.

Usage: MyLength=MySub:Length()

### **Sub:TrackCreate()**

Creates a new track. Note that each Sub needs at least one Track. So creating the Sub only does not show it in the Sub menu. You will have to create a Track in that Sub as well in order for it to show up. Please see the example.

Usage:      MyTrack = MySub:TrackCreate( [name] )

*Example: Create a new Sub and Track:*

```
MyPrj=App():Project()
MySub = MyPrj:SubCreate( "Bob" )
MyTrack = MySub:TrackCreate( "Bob" )
```

### **Sub:TrackCount()**

Returns the number of tracks in the current Sub.

Usage:      MyCunt = MySub:TrackCount()

### **Sub:TrackGet()**

Establishes a handle to a Track by index or name. It will take the current Track from the editor if no parameter is given.

Usage:      MyTrack = MySub:TrackGet( [index][name] )

### **Sub:TrackDelete()**

Deletes track by index, name or track object --

Usage:      MySub:TrackDelete( <index>|<"name">|<track> )

## Track object

A Track is a subsidiary of a Sub. It can host multiple Clips.

*NOTE:*

*Each object with lower hierarchy than App():Project() can be multiple.  
e.g. the Application can only hold one Project, but the Project can hold multiple Subs.  
Each Sub can hold multiple Tracks, each Track multiple Clips etc.  
That's why the Track object uses the TrackGet() methodology to address subobjects.  
The same counts for TrackGet(), ClipGet() and VersionGet().*

Establish a handle to the current Track:

**MyTrack=App():Project():SubGet():TrackGet()**

*NOTE:*

*In the following examples we will refer to MyTrack as the handle above.  
So instead of MyTrack:Name() you could also use App():Project():SubGet():TrackGet():Name() if you have not set the handle for convenience.*

### **Track:Name()**

Returns or sets the name of the Track.

Read: MyName = MyTrack:Name()  
Write: MyTrack:Name( [new name] )

*Example 1: Print the current Track's name to the Log*

```
MyTrack=App():Project():SubGet():TrackGet()
MyName = MyTrack:Name()
App():Log(MyName)
```

*Example 2: Rename the current Track to "Bob"*

```
MyTrack=App():Project():SubGet():TrackGet()
MyTrack:Name("Bob")
```

### **Track:Length()**

Returns the Track's length in frames

Usage: MyFrames = MyTrack:Length()

### **Track:ItemInsert()**

Inserts an item or cut as a new clip. The default index is default index is -1, which will add the item after the last clip in the current track.

Usage: MyClip = MyTrack:ItemInsert( <item>, [index] )

### **Track:ClipCount()**

Returns the number of Clips.

Usage: MyCount = MyTrack:ClipCount()

**Track:ClipGet()**

Gets a clip by it's index.

Usage:      MyClip = MyTrack:ClipGet( <index> )

**Track:ClipDelete()**

Deletes a clip.

Usage:      MyClip:ClipDelete( <index> )

## Clip object

A Clip is a timeslot in a Track. It can host multiple Items or versions. Though the name might be misleading this is not a filmclip. The filmclips are called "item" in Generation.

*NOTE:*

*Each object with lower hierarchy than App():Project() can be multiple.  
e.g. the Application can only hold one Project, but the Project can hold multiple Subs.  
Each Sub can hold multiple Tracks, each Track multiple Clips etc.  
That's why the Clip object uses the ClipGet() methodology to address subobjects.  
The same counts for TrackGet(), ClipGet() and VersionGet().*

Establish a handle to the current Clip:

**MyClip=App():Project():SubGet():TrackGet():ClipGet()**

For the currently selected Clip you have to use the ClipIndex:

MyClipIndex=proj:ThumbClipIndex()

MyClip=App():Project():SubGet():TrackGet():ClipGet(MyClipIndex)

*NOTE:*

*In the following examples we will refer to MyClip as the handle above.  
So instead of MyClip:Name() you could also use App():Project():SubGet():TrackGet():ClipGet() if you have not set the handle for convenience.*

### **Clip:Length()**

Returns the length of the clip in frames.

As opposed to Item:Length() it returns the trimmed length of the clip.

Usage: MyFrames = MyClip:Length()

### **Clip:InPoint()**

Returns or sets the in point of the clip. The second parameter defines the edit mode "roll". By default the edit mode is "ripple".

Read: MyFrame = MyClip:InPoint()

Write: MyClip:InPoint( [frame], ["roll"] )

### **Clip:OutPoint()**

Returns or sets the out point. The second parameter defines the edit mode "roll". By default the edit mode is "ripple".

Read: MyFrame = MyClip:OutPoint()

Write: MyClip:OutPoint( [frame], ["roll"] )

### **Clip:VersionInPoint()**

Returns or sets the media in point for the specified version.

Read: MyFrame = MyClip:VersionInPoint()

Write: MyClip:VersionInPoint( <version>, [frame] )

### **Clip:VersionCount()**

Returns the number of versions in the Clip.

Usage:      MyCount = MyClip:VersionCount()

### **Clip:VersionInsert()**

Inserts an item as a version to an existing timeslot. By default the index is -1, which is the latest version. 0 is the first version.

Usage:      MyClip:VersionInsert( <item>, [version index] )

### **Clip:VersionDelete()**

Deletes a version based on the index.

Usage:      MyClip:VersionDelete( <index> )

### **Clip:VersionGet()**

Establishes a handle to a version in the clip.

Can be used to acquire a handle to an actual item. Also see item:Length as an example.

Usage:      MyItem = MyClip:VersionGet( <index> )

### **Clip:VersionUserFrame()**

Sets the default media frame to be shown in the thumbnail while the item is outside the current time.

Usage:      MyClip:VersionUserFrame( <frame> )

### **Clip:GlobalFrame2ClipFrame()**

Converts the frame at the global time to the clip frame.

Usage:      clipframe = MyClip:GlobalFrame2ClipFrame( <global frame> )

## Item object

An Item is an actual filmclip in Generation. It can also refer to a Fusion comp.

### *NOTE:*

*Each object with lower hierarchy than App():Project() can be multiple.  
e.g. the Application can only hold one Project, but the Project can hold multiple Subs.  
Each Sub can hold multiple Tracks, each Track multiple Clips etc.  
That's why the Item object uses the ItemGet() methodology to address subobjects.*

Establish a handle to the current Item:

**MyItem=App():Project():SubGet():TrackGet():ClipGet():VersionGet()**

### *NOTE:*

*In the following examples we will refer to MyItem as the handle above.  
So instead of MyClip:Name() you could also use  
App():Project():SubGet():TrackGet():ClipGet():VersionGet():Name if you have not set the handle for  
convenience.*

### **Item:Length()**

Returns the item's length in frames.

As opposed to Clip:Length() this returns the full length of the clip, disregarding trim in/out values.

Usage: MyFrames = MyItem:Length()

### **Item:Name()**

Returns or sets the name of the item.

Read: MyName = MyItem:Name()

Write: MyItem:Name( <"new name"> )

### **Item:Filename()**

Returns the filename at the given frame number. It will use the first frame if [frame] is zero.

Usage: MyFilename = MyItem:Filename( [frame] )

### **Item:CreationInfo()**

Sets the creation time and user for the item.

Usage: MyItem:CreationInfo( <user name>, <time> )

### **Item:CreationTime()**

Returns or sets the creation time for the item.

Read: MyTime = MyItem:CreationTime()

Write: MyItem:CreationTime( [time] )

### **Item:CreationUser()**

Returns or sets the creation username for the item.

Read: MyUsername = CreationUser()

Write: CreationUser( [username] )

### **Item:Conversion()**

Sets the log/lin conversion settings for the item.

Usage: MyItem:Conversion( <black>, <white>, <gamma 1>, <gamma 2> )

### **Item:Settings()**

Sets aspect and speed settings for the item.

Usage: MyItem:Settings( <aspect x>, <aspect y>, <speed> )

### **Item:LUT()**

Sets the LUT for the item.

Usage: MyItem:LUT( <"lut filename"> )

### **Item:RefCount()**

Returns the referenced file count for a fusion comp for the item.

Usage: MyCount=MyItem:RefCount()

### **Item:RefGet()**

Returns the referenced filename for the item.

Usage: MyFilename = MyItem:RefGet( <index> )

### **Item:RefFilename()**

Add a fusion comp reference to an item.

Usage: refFilename = MyItem:RefFilename( <"name">, [saver], [range] )

<"name"> is the Fusion Comp name.

[saver] is the saver index if there is more than one saver in the comp.

[range] can be 0 or 1. 0 means to use the comp range, 1 means to use the cut range.

### **Item:FusionActive()**

Sets the status of the item rendering.

Usage: active = MyItem:FusionActive( [1] | [0] )

### **Item:RefAudioFilename()**

Defines the audio comment audio file for the item.

Usage: MyItem:RefAudioFilename( <"wav audio filename"> )

## RenderControl object

Establish a handle to the Rendercontrol:

**MyRC=App():RenderControl()**

*NOTE:*

*In the following examples we will refer to MyRC as the handle above.  
So instead of MyRC:QueueCreate() you could also use App():RenderControl():QueueCreate() if you have not set the handle for convenience.*

### **RenderControl:QueueCreate()**

Creates a new render queue object.

Usage: MyQueue = MyRC:QueueCreate( [name] )

### **RenderControl:QueueCount()**

Returns the number of Queues.

Usage: MyCount = MyRC:QueueCount()

### **RenderControl:QueueGet()**

Establishes a handle to the queue by index or name.

Usage: MyQueue = MyRC:QueueGet( <index>|<"name"> )

### **RenderControl:QueueDelete()**

Deletes a queue by index or name..

Usage: MyRC:QueueDelete( <index>|<"name"> )

### **RenderControl:Render()**

Starts rendering of a queue object.

Usage: MyRC:Render( <queue> )

### **RenderControl:ProxyGroupGet()**

Get and edit proxy group settings.

Usage: proxygroups = MyRC:ProxyGroupGet()

## Queue object

A QueueObject in Generation is an entry in the actual render queue. There can be only one render queue, but it can host multiple QueueObjects.

Establish a handle to the QueueObject:

**MyQ=App():RenderControl():Queue()**

*NOTE:*

*In the following examples we will refer to MyQ as the handle above.*

*So instead of MyQ:SourceSet() you could also use App():RenderControl():Queue():SourceSet() if you have not set the handle for convenience.*

### **Queue:SourceSet()**

Defines the source Item, Sub, Clip or Track.

Usage:      MyQ:SourceSet( <item>|<sub>|<clip>|<track> )

### **Queue:OutputCreate()**

Creates a queue output object. This can then be further defined in terms of resolution, codec, etc.

Usage:      queueoutput = MyQ:OutputCreate( [name] )

### **Queue:OutputCount()**

Returns how many output objects are in the queue.

Usage:      count = MyQ:OutputCount()

### **Queue:OutputGet()**

Establishes a handle to a specific output.

Usage:      queueoutput = MyQ:OutputGet( <index>|<"name"> )

## QueueOutput object

Establish a handle to the QueueOutput:

**MyOut=App():RenderControl():Queue():QueueOutput()**

*NOTE:*

*In the following examples we will refer to MyOut as the handle above.*

*So instead of MyOut:Width() you could also use App():RenderControl():Queue():QueueOutput():Width() if you have not set the handle for convenience.*

### **QueueOutput:Width()**

Returns or sets the Width of the output.

Read: width = MyOut:Width()

Write: MyOut:Width( [new width in pixels] )

### **QueueOutput:Height()**

Returns or sets the Height of the output.

Read: height = MyOut:Height()

Write: MyOut:Height( [new height in pixels] )

### **QueueOutput:Quality()**

Returns or sets the quality of compressor if that property is supported by the codec.

Accepts integer values from 0 – 100.

Read: quality = MyOut:Quality()

Write: MyOut:Quality( [new quality] )

### **QueueOutput:InterFace()**

Sets the video interface. The default is SEQUENCE.

Usage: MyOut:InterFace( <"QUICKTIME"|"AVI"|"SEQUENCE"> )

### **QueueOutput:Format()**

Sets the video codec for quicktime. For AVI a codec dialog will open.

Usage: MyOut:Format( <"TIF"|"JPG"|"PNG"|"H264"|"SOR3"> )

### **QueueOutput:FrameRate()**

Sets the video frame rate. The default is 25.0 FPS

Usage: MyOut:FrameRate( <frames per second> )

### **QueueOutput:ProcessAudio()**

Enable audio encoding for quicktime. Only Quicktime and .wav audio from project is supported.

Usage: MyOut:ProcessAudio( [1] )

### **QueueOutput:RescaleMode()**

Defines the rescale mode that is used when Width() and or Height() do not match the original dimensions of the footage.

Usage: MyOut:RescaleMode( <"ASPECT"|"FIT"|"NOSCALE"> )

"ASPECT" preserves the source aspect ratio when scaling to output resolution.

"FIT" scales the source to fill the whole output image. Aspect distortions might occur.

"NOSCALE" Centers the source image inside the output dimensions. Source images large than the output image will be cropped, source images smaller than the output image will get a black border to fill the dimensions of the output.

### **QueueOutput:FrameStart()**

Returns or sets the starting frame. It defaults to "reset", which uses the value from the source material.

Read: frame = MyOut:FrameStart()

Write: MyOut:FrameStart( [new start frame] )

### **QueueOutput:FrameEnd()**

Returns or sets the end frame. It defaults to "reset", which uses the value from the source material.

Read: frame = MyOut:FrameEnd()

Write: MyOut:FrameEnd( [new end frame] )

### **QueueOutput:Filename()**

Defines the filename for the output.

Usage: MyOut:Filename( <"new filename"> )

For filesequences use # to define the frame number and include the correct file extension.

For example MyOut:Filename( "c:\\temp\\frames\_#.tif" ). Also observe the double backslashes.

Since some video format settings generate a default filename, make sure to use Filename() as the last command after all the interface and codecs settings are done.

You can use @ to set the number of digits used in frame numbering. @@@@ evaluates to 4 digits, including leading zeros if necessary. @@@@ evaluates to 6 digits, respectively. If you use

"c:\\temp\\Frames\_@.tif" instead of "c:\\temp\\Frames\_#.tif" you will get a 2-digit frame numbering for the first 99 frames. Any frame above 99 will of course have a 3-digit numbering.

## ProxyGroup object

Establish a handle to the ProxyGroup:

**MyPG = App():RenderControl():ProxyGroupGet( "local" )**

*NOTE:*

*In the following examples we will refer to MyPG as the handle above.  
So instead of MyPG:ProxyPathDepth() you could also use  
App():RenderControl():ProxyGroupGet():ProxyPathDepth()  
if you have not set the handle for convenience.*

### **ProxyGroup:ProxyPathDepth()**

Returns or sets the proxy path depth parameter

Read: MyDepth = MyPG:ProxyPathDepth()

Write: MyPG:ProxyPathDepth( [depth] )

### **ProxyGroup:ProxyPathClear()**

Clears all defined proxy paths.

Usage: MyPG:ProxyPathClear()

### **ProxyGroup:ProxyPathAdd()**

Adds a new proxy path to this group.

Usage: MyPG:ProxyPathAdd( <folder path> )

### **ProxyGroup:ProxyPathCount()**

Returns the number of Proxy paths defined.

Usage: MyCount = MyPG:ProxyPathCount()

### **ProxyGroup:ProxyGroup:ProxyPathGet()**

Returns the path string for a specific index in the ProxyPath table.

Usage: MyPath = MyPG:ProxyPathGet( <index> )

## RenderQueue Example Script

*Render each Item in the current Track to an individual Quicktime mov.*

*This Script can be found in [Generation]\Scripts\Generation\Render Individual Quicktime Movies.lua*

```
-- renders all clips in track to separate folders with additional frames

extraFrames = 0          -- lets render 10 frames as handles
pathOut = "c:\\\         -- where do we render to?
                        -- Observe the double backslashes.

proj = App():Project()   -- The Project handle
rc = App():RenderControl() -- The RenderControl handle

sub = proj:SubGet()      -- Handle to the active Sub
track = sub:TrackGet()   -- Handle to the active Track

clipCount = track:ClipCount() -- How many Clips are in the Track?

for i = 1, clipCount do  -- Do this for every Clip.
    clip = track:ClipGet( i - 1 ) -- indices start with 0. Get a clip.
    item = clip:VersionGet(0)     -- Handle to the item with the latest version.
    name = item:Name()           -- What's the Clip's name?

    queue = rc:QueueCreate()      -- make a new QueueObject
    queue:SourceSet( track )      -- track is the source
    out = queue:OutputCreate()    -- add a new output for our queue
    -- Now we define the properties of the output. What format shall be rendered to etc.
    out:RescaleMode( "ASPECT" )   -- rescaling mode "ASPECT", "FIT", "NOSCALE"
    out:Interface( "QUICKTIME" )  -- what video interface we are using
    out:Format( "SOR3" )          -- and what video format
    out:Quality( 85 )            -- on some encoders you can set the quality
    -- change the output resolution (not active in this example)
    -- out:Width( 512 )
    -- out:Height( 256 )
    -- Do not change the resolution, use the original resolution.
    out:Width( 0 )
    out:Height( 0 )

    -- set our render start frame -10 frames of clip start
    out:FrameStart( clip:InPoint() - extraFrames )

    -- render end at +10 frames of clip end
    out:FrameEnd( clip:OutPoint() + extraFrames - 1 )

    -- lets generate the name of the file based on the Item's name and print that to the console.
    out:Filename( pathOut .. name .. ".mov" )
    App():Log(pathOut .. name .. ".mov")

    rc:Render( queue )           -- put the Object in the RenderQueue
end
```

*--NOTE: rendering does not block the execution of this script !*

## FarmControl object

Establish a handle to the FarmControl:

**MyFC = App():FarmControl()**

*NOTE:*

*In the following examples we will refer to MyFC as the handle above.*

*So instead of MyFC:ManagerOpen() you could also use App():FarmControl():ManagerOpen() if you have not set the handle for convenience.*

### **FarmControl:ManagerOpen()**

Opens the generation internal farm manager tool.

Usage: MyFC:ManagerOpen()

### **FarmControl:Submit()**

Submits a selected thumbnail to farm.

Usage: MyFC:Submit( <thumb> )

## PlayerLayoutControl object

Establish a handle to the PlayerLayoutControl:

**MyPLC=App():Project():PlayerLayoutControl()**

*NOTE:*

*In the following examples we will refer to MyPLC as the handle above.  
So instead of MyPLC:OpenTool() you could also use App():Project():PlayerLayoutControl():OpenTool()  
if you have not set the handle for convenience.*

### **PlayerLayoutControl:OpenTool()**

Opens the tool window.

Usage: MyPLC:OpenTool()

### **PlayerLayoutControl:LayoutActivate()**

Activates an existing Layout.

Usage: MyPLC:LayoutActivate( <layout name> )

### **PlayerLayoutControl:LayoutCreate()**

Creates and names a new layout page. If the name is already in the list, it will create a unique name.

Usage: MyLayout = MyPLC:LayoutCreate( [name] )

### **PlayerLayoutControl:LayoutGet()**

Get a layout with name or index.

Usage: MyLayout = MyPLC:LayoutGet( <index>|<"name"> )

### **PlayerLayoutControl:LayoutDelete()**

Deletes a layout.

Usage: MyPLC:LayoutDelete( <layout> )

### **PlayerLayoutControl:LayoutCount()**

Returns the number of Layouts in the Project.

Usage: MyCount = MyPLC:LayoutCount()

## Layout Object

Establish a handle to the Layout:

**MyLay=App():Project():PlayerLayoutControl():LayoutGet()**

*NOTE:*

*In the following examples we will refer to MyLay as the handle above.*

*So instead of MyLay:Name() you could also use App():Project():PlayerLayoutControl():LayoutGet():Name() if you have not set the handle for convenience.*

### **Layout:Name()**

Returns or sets the name of the active layout.

Read: MyName = MyLay:Name()

Write: MyLay:Name( [name] )

### **Layout:ViewerCreate()**

Creates a new viewer in the active Layout.

Usage: MyViewer = MyLay:ViewerCreate()

### **Layout:ViewerGet()**

Establishes a handle to a viewer via an index.

Usage: MyViewer = MyLay:ViewerGet( <index> )

### **Layout:ViewerDelete()**

Deletes a viewer by viewer, index or name.

Usage: MyLay:ViewerDelete( <viewer>|<index>|<"name"> )

### **Layout:ViewerCount()**

Returns the number of Viewers in the active Layout.

Usage: MyCount = MyLay:ViewerCount()

## Viewer Object

Establish a handle to the Viewer:

**MyView=App():Project():PlayerLayoutControl():LayoutGet():ViewerGet()**

*NOTE:*

*In the following examples we will refer to MyView as the handle above.  
So instead of MyView:Position() you could also use  
App():Project():PlayerLayoutControl():LayoutGet():ViewerGet():Position()  
if you have not set the handle for convenience.*

### **Viewer:Position()**

Returns or sets the position of a viewer.

When reading the position it returns a table of two elements, representing the X and Y coordinates of the viewer.

Read: MyPosition[ 2 ] = MyView:Position()

Write: MyView:Position( <x>, <y> )

### **Viewer:SourceReferenceName()**

Attaches media with its referenced name to the active Viewer.

Usage: MyView:SourceReferenceName( <ref item name> )

### **Viewer:SourceItem()**

Attaches media with an item object

Usage: MyView:SourceItem( <item> )

### **Viewer:FrameOffset()**

Returns or sets the time alignment or offset for the viewer.

Read: MyFrame = MyView:FrameOffset()

Write: MyView:FrameOffset( [frame] )

# Appendix

## Lua License Agreement

Copyright © 2002 Tecgraf, PUC-Rio.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sub-license, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## SciTe License Agreement

License for Scintilla and SciTE

Copyright 1998-2001 by Neil Hodgson

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

NEIL HODGSON DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL NEIL HODGSON BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.