Mocha® 2019 Python Scripting Guide

Table of Contents

Introduction

Welcome to the **Mocha 2019** Python scripting guide.

This User Guide will take you through some of the python functions for **Mocha 2019**. Additionally you can check the mocha Python reference available online.

The guide assumes you already have a basic understanding of Python coding. For guides on how to learn Python, please refer to https://python.org.

> Most code in this guide uses the **Mocha Pro** python package for examples.

Deprecated API in Mocha 2019

These items are now deprecated as of version **Mocha 2019** onwards:

We have moved to Qt5 for **Mocha 2019** and therefore have also upgraded to the supported Python integrations for that release.

- **shiboken is now shiboken2**: If your code used shiboken previously you will now need to update your code to support shiboken2

- **PySide is now PySide2**: Note that PySide2 has structural changes that will affect Mocha scripts written for previous versions. For example: `from PySide.QtGui import QApplication` now becomes `from PySide2.QtWidgets import QApplication`
- **Mocha VR is no longer separate from Mocha Pro:** This means you do not need to define ISL environment variables to run code for Mocha 360 projects. These should be removed.

Installation

Python for mocha installs with the Linux and OS X packages by default. On Windows, Python installs as a separate bundle in the Mocha Pro installer. When installing, you need to make sure to select all features - not just the top level feature.

The mocha Python command tools are available per system in the following default locations in Mocha Pro:

- **OS X:** */Applications/Mocha Pro 2019.app/Contents/MacOS/python*
- **Windows:** *C:\Program Files\Mocha Pro 2019\python\python.exe*
- **Linux:** */opt/isl/mocha2019/python/bin/python*

The mocha package uses Python version 2.7.14+. For version use and compatibility please read the standard Python 2.7.14+ documentation: https://docs.python.org/2/

You can also install the mocha module directly to your own Python installation.

1. Navigate to the mocha Python installation, for example: *C:\Program Files\Mocha Pro 2019\python*
2. Run setup.py install in the directory with your own version of Python, for example: *C:\Python27\python.exe setup.py install*

Now when you run your version of Python you should be able to import the mocha module.

Importing the mocha module into a separate Python installation

```
from mocha.project import *
```

Running mocha Python with Mocha Pro

To run a mocha script with the **Mocha Pro** Python version, call it directly with the script, like so:

- **OS X:** */Applications/Mocha Pro 2019.app/Contents/MacOS/python myMochaScript.py*
- **Windows:** *C:\Program Files\Mocha Pro 2019\python\python.exe myMochaScript.py*
- **Linux:** */opt/isl/mochapro2019/python/bin/python myMochaScript.py*

Python Script Editor

We now have an inbuilt Python script editor to code and run scripts directly in the mocha program. If it is not already open, you can load the Python Script Editor via the View menu.

The script editor is intentionally barebones and is not meant to replace a fully featured IDE, but will color your syntax for easier readability.

The Editor Window

The main edit window lets you type or load scripts into mocha and then run them. You can also save a current script.

The Output Window

The output window will print all script output as well as any errors that may come up. Additional errors are also written to the mocha log file.

The Main mocha Python Structure

The Python API for mocha is broken up into a series of modules.

Top level mocha methods

For the mocha package itself, there are two main functions for locating and running the mocha application.

These are especially useful for when you are running external scripts.

mocha package methods

```
# Return the absolute path of the mocha bin directory.
mocha.get_mocha_exec_dir()

#Run mocha application with given command-line arguments
mocha.run_mocha(app='mochapro', footage_path='/tmp/myfootagepath.png', '--
frame-rate 24 --in 0 --out 100')

# override settings for offscreen buffers using mocha.Settings
overridden_settings = mocha.Settings(override=True, read_overridden=True)
overridden_settings.disable_offscreen_buffers = not
 overridden_settings.disable_offscreen_buffers
```

mocha.run_mocha() is essentially the same command as running mocha from the command line. See the "Command Line" section of the Mocha Pro User Guide for more information on keyword arguments when loading mocha.

mocha.Settings is an extensive way to change preferences with or without directly writing them to disk. See the "Adjusting mocha Preferences using mocha.Settings" section for a deeper example.

The mocha.project Module

The core module you will be using to script python commands for mocha will be the **`mocha.project`** module, which is broken into a number of Classes.

The mocha.project class structure

mocha.project

- **Global functions:** For getting project-level information such as the current project
- **BezierContour:** Provides access to Bezier contours and their control points
- **BezierControlPoint:** Provides access to Bezier contour control points.
- **BezierControlPointData:** Provides access to individual parameters for each Bezier contour control point
- **Clip:** Provides clip manipulation options.
- **ColorParameters:** For colorspace adjustments
- **Layer:** For top-level layer control and information
- **LayerGroup:** For Layer Group control and information

- **Parameter:** Parameter access for data objects in the project
- **ParameterSet:** Access to a set of Parameters for a data object
- **ProgressWatcher:** Progress indication class for different complex operations which might take a long time (e.g. rendering, exporting, etc.)
- **Project:** Main project class. Provides methods and properties for general project management of layers, groups, rendering and output directories
- **RenderOperation:** Base class for rendering operations
- **RenderInsertOperation:** Class for all Insert render operations
- **RenderRemoveOperation:** Class for all Remove render operations
- **RenderStabilizeOperation:** Class for all Stabilize render operations
- **RenderReorientOperation:** Class for all Reorient render operations (Available in 360 mode only)
- **StreamInfo:** Class for accessing stream information in a clip
- **UndoGroup:** Class for setting undoable actions
- **View:** Defines views for clips
- **ViewInfo:** Class representing common view information (name, abbreviation, color).
- **XControlPoint:** Provides access to X-Spline contour control points.
- **XControlPointData:** Provides access to individual parameters for each X-Spline contour control point.
- **XSplineContour:** Provides access to X-Spline contours and their control points.

The mocha.exporters Module

If you want to modify or create your own exporters, you need to use the `mocha.exporters` module.

The mocha.exporters class structure

mocha.exporters

- **AbstractCameraSolveExporter:** Abstract camera solve exporter class. Inherit the class to create your own exporter formats. Inherited by CameraSolveExporter.
- **AbstractShapeDataExporter:** Abstract class for the Shape Data exporter. Inherit the class to create your own custom formats. Inherited by ShapeDataExporter.

- **AbstractTrackingDataExporter:** Abstract tracking data exporter class. Inherit the class to create your own exporter formats. Inherited by TrackingDataExporter.
- **CameraSolveExporter:** Camera data exporter class. Wraps a predefined/ custom mocha exporter inside.
- **ShapeDataExporter:** Class for defining the shape data to export
- **ShapeExportData:** Shape data exporter class. Wraps a predefined/custom mocha exporter inside.
- **TrackingDataExporter:** Tracking data exporter class. Wraps a predefined/ custom mocha exporter inside.

The mocha.tools Module

If you want to create your own tools in the interface, you can use the `mocha.tools` module.

The mocha.tools class structure

mocha.tools

- **Global functions:** Functions for registering and setting tool instances.
- **AbstractTool:** Abstract tool class that provides overridable methods to determine custom tools. Every overridable method must be implemented.
- **InputEvent:** Event handling for mouse interaction and contour data.

The mocha.mediaio Module

If you want to create your own custom formats for reading and writing, you can use the `mocha.mediaio` module.

The mocha.mediaio class structure

mocha.mediaio

- **AbstractImageIOModule:** Abstract image IO class that provides overridable methods to determine custom image input and output operations. Every overridable method must be implemented. The methods should not call each other.
- **ImageData:** Main class for access to image data fields.

The mocha.ui Module

This module is useful for getting values for specific items in the ui or wrapping handlers around actions. Many of these are convenience methods for quickly locating widgets instead of navigating through PySide.

Qt Script Requirements

Some aspects of mocha Python code will require the creation of Qt Objects to handle certain functionality.

QCoreApplication

For external scripts (i.e those not run in the Mocha Python Script Editor), a **QCoreApplication** object must always be created before creating a Project object.

If you don't create the **QCoreApplication** Object, then the internal parameter notification system does not work and you may get unexpected results or errors when dealing with parameter changes.

Assigning a QCoreApplication() object

```
from PySide2.QtCore import QCoreApplication
app = QCoreApplication(sys.argv)
```

to check if you are using an instance of **QCoreApplication** already, you can look for the instance:

Checking for existing QCoreApplication() objects instances

```
from PySide2.QtCore import QCoreApplication
if QCoreApplication.instance():
    print(QCoreApplication.instance().arguments()) #the first argument is
 the path to mocha
```

Creating a New Project

You can generate a new project from python with or without an available clip.

To do this, you first need to import the Clip and Project classes from **mocha.project**:

Importing mocha.project Classes

```
from mocha.project import Project, Clip
```

If you are running the script externally from the Mocha interface, you must also define a **QCoreApplication** object to connect to the Mocha MediaIOServer. This allows you to read in QuickTime-associated media.

Assigning a QCoreApplication() object

```
from PySide2.QtCore import QCoreApplication
app = QCoreApplication(sys.argv)
```

You then create a new Clip object and assign it to a new Project object:

Creating Clip and Project objects

```
clip = Clip('/path/myfile.exr', 'NewClip') # The Clip name is optional
proj = Project(clip)
```

At this point the project is now in memory. You can delete the original Clip object as the project contains a deep copy - the original clip is not part of the project.
To save the project, use the **save_as()** function and define a mocha project file and path.

Saving to a new project file

```
proj.save_as('/path/to/filename.mocha')
```

At any point if you want to save the project again, you can use:

Saving the existing project file

```
proj.save()
```

This will save to the project file you defined with **save_as()**.

Modifying project properties

You can query and set different project properties:

Accessing or modifying project properties

```
#Print the path of the project file
print proj.project_file
```

```
#Set the frame rate of the project
proj.frame_rate = 48

#Add text to the 'Project Notes' panel
proj.notes = 'New Project'

#Set the project output directory
proj.set_output_dir('/tmp/')

#Get the dictionary of clips inside the project.
clip_list = proj.clips

#Get the list of layers inside the project
layer_list = proj.layers
```

Creating a Stereo Project

You can define stereo projects by mapping views to that project.

The **views** property is an array of **`ViewInfo`** objects.

You can define 3 parameters in the **`ViewInfo`**

- The name of the view
- The abbreviated name of the view. This is used for the view buttons as well as for some rendering suffixes
- The color of the view, defined as a tuple for values RGB

Each **`ViewInfo`** entry corresponds to a View index, so:

```
import ViewInfo, View

proj.views = [ViewInfo('left', 'L', (0.1, 0.4, 0.9)),
 ViewInfo('right', 'R', (0.1, 0.0, 0.7))]
```

The above code would map views as follows:

- View(0): Left
- View(1): Right

You can also define the **`default_hero_view`** property:

```
proj.default_hero_view = 0
```

To add new streams to existing clips so you can map them to views, use the **add_stream** method.

The **add_stream** method requires the following parameters:

- The path to the footage
- The **View()** you want to map it to
- The start frame
- The end frame
- Whether you want to validate if the file is valid footage.

```
myClip = Clip('/path/myfile_L.mov', 'NewClip')
myClip.add_stream('/path/myfile_R.mov', View(1), 7, 154, True)
```

You can assign a clip stream to a different project view:

```
myClip.assign_project_view(View(0), View(1))
```

And you can also delete streams:

```
myClip.delete_stream(View(1))
```

Layers and Groups

The mocha module can find and modify layers and groups in a project file, or create new ones. To work with Layers and point data, you will need some additional imports:

Importing layer and point classes

```
from mocha.project import Project, Clip, Layer, LayerGroup,
 XSplineContour, XControlPoint, XControlPointData, BezierContour,
 BezierControlPoint, BezierControlPointData
```

You can then begin to check layer content in projects

Listing layers

```
#Create a Project obj with an existing mocha file
proj = Project('myFile.mocha')

#Get the list of layers inside the project
layer_list = proj.layers

#Print the name of the layer
print layer_list[0].name
```

You can also search for particular layers or groups:

Searching for Layers or Groups

```
group = proj.find_groups('Group 1')
layer = proj.find_layers('Layer 1')
```

Change their order:

Changing layer order

```
#Get the currently open Project
proj = get_current_project()

#Get the list of layers inside the project
layer_list = proj.layers

#Reorder a layer in the list to position 2 in the stack
layer_list[0].z_order = 2
```

Or control their tracking:

Tracking layers

```
#Get the currently open Project
proj = get_current_project()

#Track any layer in the project that has a process cog turned on
proj.track_layers()

#Track layers in the project for a specific frame range (all parameters
 are optional)
proj.track_layers(start_index=5, stop_index=45)

#Track backwards by having a high start index and a low stop index
```

```
proj.track_layers(start_index=100, stop_index=1)
```

To create a new layer, you must assign it to a particular input clip, just as though you were drawing a layer on a clip inside mocha. You can define 4 main properties when creating a layer:

- The input clip you are adding the layer to
- The name of the layer
- The frame number you want to assign the drawing keyframe to (similar to when you draw on a particular frame inside mocha, this generates the first keyframe for that layer)
- The view you want to assign it to, starting from zero. At present you can only assign 0 or 1 (for stereo).

You only need to assign the first property (the input clip you want to assign to the layer)

Creating a layer

```
new_layer = proj.add_layer(proj.clips['My input clip'], name='New Layer',
 frame_number=0, view=0)
```

At this point the layer is empty, so you need to add a shape contour. This is where it starts to get interesting!

Shape Contours

Adding contours to a layer involves setting up the content to draw the layer. You can add a contour to a layer, but first it needs point data. Each point in a contour has a number of important parameters that need to be set.

Bezier Point Data and Contours

To create point data for a Bezier shape you use **`BezierControlPointData`** with the following arguments:

- **corner**: Boolean to set if the Bezier is a corner or smooth type
- **active**: Boolean to set if the point is active
- **x**: The x coordinate of the point (float)
- **y**: The y coordinate of the point (float)

- **edge_width**: The distance of the outer edge from the inner edge to determine feather/falloff (float or None)
- **edge_angle_ratio**: The angle of the out edge point from the inner edge point (float or None)
- **curve_angle**: The tangent angle defining the curve (float or None)
- **handle_offset_backward**: The back offset point of the tangent (tuple or None)
- **handle_offset_forward**: The forward offset point of the tangent (tuple or None)

Creating bezier point data

```
bezier_point = BezierControlPointData(corner=False,
                                      active=True,
                                      x=600.0, y=500.0,
                                      edge_width=0.0,
                                      edge_angle_ratio=0.0,
                                      curve_angle=0.0,
                                      handle_offset_backward=None,
                                      handle_offset_forward=None)
```

Of course, one point is not enough for a shape, so you need to set a tuple of BezierControlPointData objects to define a final contour, using **`add_bezier_Contour()`**. This takes two arguments, the frame you want to start on, and a tuple of point data.

Example of creating a contour from Bezier point data.

```
points = [[546, 234], [806, 377], [546, 520], [286, 377]]
b_point_data = []

for x,y in points:
        b_point = BezierControlPointData(corner=False,
                                         active=True,
                                         x=float(x),
                                         y=float(y),
                                         edge_width=0.0,
                                         edge_angle_ratio=0.0,
                                         relative_angle=0.0,
                                         curve_angle=0.0,
                                         handle_offset_backward=None,
                                         handle_offset_forward=None)
        b_point_data.append(b_point)
```

```
b_contour = new_layer.add_bezier_contour(0.0, tuple(b_point_data))
```

X-Spline Point Data and Contours

To create point data for an X-Spline shape you use **`XControlPointData()`** with the following arguments:

- **corner**: Boolean to set if the X-Spline is a corner or smooth type
- **active**: Boolean to set if the point is active
- **x**: The x coordinate of the point (float or None)
- **y**: The y coordinate of the point (float or None)
- **edge_width**: The distance of the outer edge from the inner edge to determine feather/falloff (float or None)
- **edge_angle_ratio**: The angle of the out edge point from the inner edge point (float or None)
- **weight**: The length of the handle that forms the curve weight of the X-spline point (float or None)

Creating x-spline point data

```
xspline_point = XControlPointData(corner=False,
                                  active=True,
                                  x=600.0,
                                  y=500.0,
                                  edge_width=0.0,
                                  edge_angle_ratio=0.5,
                                  weight=0.25)
```

Of course, one point is not enough for a shape, so you need to set a tuple of XControlPointData objects to define a final contour, using **`add_xpline_Contour()`**. This takes two arguments, the frame you want to start on, and a tuple of point data.

Example of creating a contour from X-Spline point data

```
points = [[546, 234], [806, 377], [546, 520], [286, 377]]
x_point_data = []

for x,y in points:
    x_point = XControlPointData(corner=False,
                                active=True,
                                x=float(x),
```

```
                                y=float(y),
                                edge_width=0.0,
                                edge_angle_ratio=0.5,
                                weight=0.25)
    x_point_data.append(x_point)

x_contour = new_layer.add_xspline_contour(0.0, tuple(x_point_data))
```

Inserting Points

You can also insert points into existing shapes using the **`insert_point()`** function. To do this you just create point data as normal above. You need to insert an XControlPointData point into a X-Spline contour, and of course a BezierControlPointData into a Bezier contour.

The **`insert_point()`** function has three arguments:

- time: The keyframe you want to insert the point on
- data: The point data for the inserting point
- index: Where in the point order you want to place the new point

Example of inserting a point into an x-spline contour

```
layer_contour = proj.layers[0].contours[0]
x_point = XControlPointData(corner=False,
                            active=True,
                            x=400.0,
                            y=300.0,
                            edge_width=0.0,
                            edge_angle_ratio=0.5,
                            weight=0.25)
end_idx = len(layer_contour.control_points)
layer_contour.insert_point(0.0, x_point, end_idx)
```

Obtaining the Current Clip

One very important part of creating or modifying layers can be knowing the right input clip to apply it to. We have a convenient parameter to help with this:

Example of obtaining the current trackable clip

```
from mocha.project import get_current_project
#Get the clip you created the project with
```

```
name = get_current_project().default_trackable_clip.name
print 'Default trackable clip name is', name
```

Getting the matte clip for a layer

If you need to work with the matte clip of a specific layer, you can find it via the **GarbageMatteClipID** parameter. See the section on the Parameter API for more details on accessing project parameters.

Example of getting the matte clip of a layer

```
matte_clip_id = layer.parameter_set()['GarbageMatteClipID'].get()
matte_clip = filter(lambda clip: clip.id == matte_clip_id,
 proj.clips.values())[0]
```

Rendering

Rendering Remove, Insert, Stabilize and Reorient

In addition to creating shapes, we can also render from each module. In the examples below we show Remove, but the same operations are available for Insert and Stabilize.

The key render operation classes are:

- **RenderInsertOperation**
- **RenderRemoveOperation**
- **RenderStabilizeOperation**
- **RenderReorientOperation**

To use the **RenderReorientOperation** class in your python scripts you need to have an Equirectangular 360 project.

To handle removes and exports, you need to have the following mocha classes loaded:

Imported classes for Remove renders

```
from mocha.project import Project, Clip, View, Layer,
 RenderRemoveOperation
```

Rendering removes comes in three parts:

1. Defining a **`RenderRemoveOperation()`**
2. Calling the **`render()`** function
3. Exporting the remove with the **`export()`** function

The **`render()`** function has the following arguments:

- **render_operation (RenderOperation)**: An instance of a render operation.
- **start_index (int)**: The starting frame number.
- **stop_index (int)**: The end frame number.
- **layers (list of Layer instances.)**: The list of layers to render.
- **views (list of View instances)**:The list of views to render.

Exporting a rendered Remove, Insert, Stabilization or Reorient

The **`export()`** function for a render operation object has the following arguments:

- **revert_to_clip (Clip)**: The clip to revert to if a rendered frame does not exist
- **directory (str)**: The output clip directory.
- **extension (str)**: The file extension (.TIF, .DPX,etc.)
- **prefix (str)**: Any prefix you want at the start of the file name
- **suffix (str)**: Any suffix you want at the end of the file name
- **index_start (int)**: The start frame to export
- **index_stop (int)**: The end frame to export
- **index_width (int)**: The index width of your rendered frames
- **views (list of View instances)**: Views to export.

Example of rendering a remove and exporting it

```
from mocha import *
from mocha.project import *
from collections import OrderedDict

render_output_dir = "/var/tmp/exports"

#Assign a project
proj = Project('/myproject.mocha')
```

```
#Assign a clip
clip = proj.clips['my_source_clip']

#define the view
view = clip.views[0]

#define the layer you want to use in the project for the remove
layer = proj.find_layers('Remove Layer')[0]

#define the remove operation
rm = RenderRemoveOperation()

#render the remove, which returns a clip object
remove_clip = proj.render(rm, 1, 15, [layer])

#Define arguments to assign to the clip export, including a render output
 dir
args = OrderedDict((('revert_to_clip', None),
                    ('directory', render_output_dir),
                    ('ext', '.png'),
                    ('prefix', 'Remove'),
                    ('suffix', ''),
                    ('start', 1),
                    ('stop', 7),
                    ('index_width', 0)))

#export the clip
remove_clip.export(*args.values())
```

Rendering and exporting may require write permissions to write to the Cache directory.

Rendering Matte Shapes

Exporting rendered mattes is a little simpler than rendering clips.

To handle shape exports, you require the following mocha classes loaded:

Imported classes for Matte Renders

```
from mocha.project import Project, Clip, View, Layer, ColorizeOutput
```

The **`export_rendered_shapes()`** function has the following arguments:

- **layers (list of Layers)**: Layers which will be exported.
- **colorize_output (ColorizeOutput)**: Colorize output option.
- **directory (unicode)**: Output directory for rendered clip.
- **extension (unicode)**: File extension for rendered clip.
- **prefix (unicode)**: Any prefix you want at the start of the file name
- **suffix (unicode)**: Any suffix you want at the end of the file name
- **index_start (PySide2.QtCore.uint)**: The start frame to export
- **index_finish (PySide2.QtCore.uint)**: The end frame to export
- **index_width (PySide2.QtCore.uint)**: Digits count in clip index.
- **views (list of View)**: Views to export.
- **offset (PySide2.QtCore.uint)**: Frame offset for the exported image sequence.

The **`colorize_output`** option is based on parameters in the **`ColorizeOutput`** object, which defines if you want to export the mattes as *Grayscale*, *Matte Color* in the GUI or by the depth of the layer in the layer stack (i.e *By Layer*)

To illustrate this, here is a dictionary of the **`ColorizeOutput`** parameters:

Example of accessing ColorizeOutput parameters

```
from mocha.project import ColorizeOutput
COLORIZE_OUTPUT = {'grayscale': ColorizeOutput.Grayscale,
                   'matte-color': ColorizeOutput.ByMatteColor,
                   'layer': ColorizeOutput.ByLayer}
```

Example of Exporting Rendered Mattes

```
layer = proj.find_layers('Layer 1')[0]
new_clip = proj.export_rendered_shapes([layer],
                                       ColorizeOutput.Grayscale,
                                       render_output_dir,
                                       '.png',
                                       'Matte',
                                       '',
                                       1,
                                       3,
                                       0)
```

Exporting may require write permissions to write to the Cache directory.

Watching Renders

You can also create watchers for the rendering so that you can trigger events or just keep an eye on progress.

The watcher example below connects to a render process and outputs the render and export progress to the command line.

Example of using the watcher function to output progress of a render and an export

```
from PySide2.QtCore import QCoreApplication
import sys
from mocha.project import *

app = QCoreApplication(sys.argv)
proj = Project('/_clips/Results/Fish_remove.mocha')
rm = RenderRemoveOperation()
layer = proj.find_layers('REMOVE FISHY')[0]

def on_start_rendering():
    sys.stdout.write('Rendering started.\nProgress:\n')
    sys.stdout.write('[ %s ]' % (' ' * 100,))

def on_start_exporting():
    print 'Exporting started'

def on_progress(progress):
    sys.stdout.write('\r')
    sys.stdout.write('[ %s%s ]' % ('#' * progress, ' ' * (100 -
 progress)))

def on_message(message):
    print message


def on_finish():
    print '\n'
    print 'Rendering is finished'

#Watch the remove and show a progress bar
watcher = rm.progress_watcher
watcher.started.connect(on_start_rendering)
watcher.progress_status.connect(on_progress)
watcher.finished.connect(on_finish)
```

```
#Render the remove from frames 0-10
clip = proj.render(rm, 0, 10, [layer])

print 'Exporting!'

# Watch the exporter and print the saved files
watcher = clip.progress_watcher
watcher.started.connect(on_start_exporting)
watcher.progress_message.connect(on_message)
watcher.finished.connect(on_finish)

#Export the clip to a png sequence
clip.export(None,
            '/tmp/exported',
            '.png',
            'prefix_',
            '_suffix',
            0,
            10,
            0)
```

The Parameter API

One of the most powerful aspects of the mocha Python API is the ability to access all project and layer parameters via the **`parameter`** function.

Try the following by pasting the code into the Mocha Python Script Editor:

Example of using the parameter API to get the surface X/Y coordinates of *Layer 1*

```
from mocha.project import get_current_project

proj = get_current_project()
name = 'Layer 1'

scorners = []

for idx in range(0,4):
        sX = proj.parameter([name, u'Surface'+str(idx)+u'X']).get()
        sY = proj.parameter([name, u'Surface'+str(idx)+u'Y']).get()
        scorners.append(sX)
        scorners.append(sY)

print scorners
```

The parameter system opens up a large range of options for users wishing to write tools to modify or create layers with different properties.

In the example code above, once we have the layer's surface coordinates we can then use those positions for various tasks, such as drawing splines that fit the surface, or adjust another layer's surface to match the current one.

Image Access and Creating New Clips

Accessing image data in a clip

Sometimes it may be necessary perform image operations on an existing clip frame. To do this you can combine the built in image API along with third party tools.

In order to access the image on any frame, you simply have to feed the **`Clip.image`** function the frame number.

For example, in the code below, we give the function frame 10 and can print the dimensions of the image.

```
proj = get_current_project()
clip = proj.clips["my_clip"]
frame = 10
image = clip.image(frame)
print image.width, image.height
```

Writing image data to a new clip

In general we don't want to affect the existing source clip when performing image operations, so we have to write image data to a new output clip in order to use it within mocha.

Creating a new output clip is very simple, you have to provide an input clip and a name:

```
proj = get_current_project()
clip = proj.clips["my_clip"]
new_clip = proj.new_output_clip(clip, "my_new_clip")
```

But that only creates the container. You then need to assign the new clip images. To check if a clip has an image on the frame, you can call new_clip.image(frame) and you'll get **`None`** if no image is currently allocated. If the image has already been allocated, the image will be returned.

When you pass **`allocate = True`**, this tells mocha: "if there is no image, allocate a new one and return it please", for example:
`output_image = new_clip.image(frame, allocate = True)`
will return a new image object if this is the first time we've assigned an image to that frame.

The returned image (an **`ImageData`** instance) is a lightweight handle referencing the real image in mocha, hence any changes on its data will immediately apply to the image.

The **`ImageData.pixels`** property returns a weak reference to a python array, which references the actual pixel data.

The setter for **`ImageData.pixels`** accepts a python array instance, deallocates the existing pixel data and pins the appropriate image to the array data, which helps to avoid extra copy operations to the pixel buffer.

So, for example, you could assign a range of pixels to an example like so:

```
pixels = image.pixels()
for x in range(30000):
  pixels[x] = 0
```

This should make the first 10000 pixels black (in case of, say, RGB clip). A black bar should appear at the bottom of the image.

The most efficient way to assign pixels to an image however is via python arrays. The array size must match with the original pixel array size.

Example code of reading and writing pixels

In the code below, we are performing the following tasks:

1. Reading the frames from the entire project length using **`Clip.image`**
2. Creating the a new output clip called "Contrast_my_clip" using **`new_output_clip`**
3. Performing a simple contrast using the Pillow module
4. Writing the resulting pixel data to the new clip via its **`.image`** object

```
import sys
import array
import os
```

```
# if you don't have PIL added to your mocha Python packages,
# you can access it from your system python
sys.path.append('/usr/local/lib/python2.7/dist-packages')

from PIL import Image, ImageEnhance

from PySide2.QtCore import *

from mocha.project import *
from mocha.ui import *

proj = get_current_project()
clip = proj.clips["my_clip"]
contrast_clip = proj.new_output_clip(clip, "Contrast_my_clip")
for frame in range(proj.length):
    image = clip.image(frame)
    pil_image = Image.frombytes('RGB',
                                (image.width, image.height),
                                image.pixels(),
                                decoder_name='raw')
    enhancer = ImageEnhance.Contrast(pil_image)
    factor = 2
    pil_image = enhancer.enhance(factor)
    pil_image_bytes = pil_image.tobytes()
    output_image = contrast_clip.image(frame, allocate=True)
    output_image.pixels = array.array(image.pixels().typecode,
                                      pil_image_bytes)
    print "Rendered frame", frame
```

Any changes made to pixel data will immediately invalidate the image cache.

Using init.py and Initialization Functions

We generate a blank init.py script on the first run of mocha for you to add functionality on startup.

This can be as simple as actions you want to perform when you start mocha, but the real power comes from being able to set up tools in the interface using widgets.

The init.py path

The default init.py path is the Imaginer Systems Scripts directory.

The mocha init.py script is generated per system in the following default locations:

- **OS X:** *~/Library/Application Support/Imagineer Systems Ltd/Scripts/init.py*
- **Windows:** *C:\Users\[username]\AppData\Roaming\Imagineer Systems Ltd\Scripts\init.py*
- **Linux:** *~/.config/Imagineer Systems Ltd/Scripts/init.py*

You can also set the environment variable *MOCHA_INIT_SCRIPT* to control where the path of the init.py initialization script resides.

If the *MOCHA_INIT_SCRIPT* environment variable points to a file, that file will be used, if it points to a directory, it will look specifically for init.py in that directory. If unset, the default locations above will be used.

Using init.py

Below we show a detailed example of using init.py for creating a user-entry tool to prepend a word onto the front of all selected layers.

We also list code at the end to show how to add this to the file menu in mocha and load a dialog for user entry.

Some knowledge of PySide and Qt is helpful here, but if you follow along the script you can see how the widgets are created.

Example of using the init.py script

```
from mocha.project import Project, get_current_project
from collections import OrderedDict

from PySide2.QtWidgets import *
from mocha.project import get_current_project
from mocha.ui import get_widgets

class LayerPrepend():

    def __init__(self):

        self.app = QApplication.instance()
        self.layer_tree = self.get_layer_tree()
        self.layer_prepend()

    def get_layer_tree(self):
```

```
        widgets = get_widgets()
        return widgets['LayerControl']

    def layer_prepend(self):

        selected_layers = self.layer_tree.selectedIndexes()

        if len(selected_layers) > 0:
            dlg = QDialog()
            layout = QFormLayout()
            edt = QLineEdit()
            layout.addRow("Prefix", edt)
            btn_box = QDialogButtonBox(QDialogButtonBox.Ok |
 QDialogButtonBox.Cancel)
            btn_box.accepted.connect(dlg.accept)
            btn_box.rejected.connect(dlg.reject)
            layout.addRow(btn_box)
            dlg.setLayout(layout)
            if dlg.exec_() == QDialog.Accepted:
                self.prepend_selected_layers(edt.text())
                self.layer_tree.update()

    def prepend_selected_layers(self, prefix):

        project = get_current_project()
        selected_layers = self.layer_tree.selectedIndexes()
        for idx in selected_layers:
            layer = project.layer(idx.row())
            layer.name = prefix + layer.name

#grab all widgets
widgets = application.allWidgets()

# Grab all the menu items in mocha
mocha_menus = filter(lambda wgt: isinstance(wgt, QMenu), widgets)

# Locate file menu
file_menu = filter(lambda menu: menu.objectName() == 'MenuFile',
 mocha_menus)[0]

# Create menu action dictionary
actions_dict = {'Layer prepending': (file_menu, LayerPrepend)} #Add more
 menu items to this list as you need them

# Add dictionary of actions to menu
for key, value in actions_dict.iteritems():
```

```
    action = QAction(key, value[0])
    action.triggered.connect(value[1])
    value[0].addAction(action)
```

If you need to check Python error output after loading an init.py script, load the error log from the Help menu, or load mocha via the terminal.

Creating Interfaces

You can create GUI inside mocha using the PySide Qt API. Showing a widget and connecting it to an action or function is very simple:

Example of showing a combo box inside mocha

```
from PySide2.QtWidgets import *

combo = QComboBox()
combo.addItems(['Layer 1', 'Layer 2'])

def nameSelected(name):
    print name

combo.activated[str].connect(nameSelected)
combo.show()
```

You can also create menu items, by locating the menu bar:

Example of creating a new menu inside mocha

```
from PySide2.QtWidgets import *
application = QApplication.instance()
widgets = application.allWidgets()
mocha_menubar = filter(lambda wgt: isinstance(wgt, QMenuBar), widgets)[0]
scripts_menu = mocha_menubar.addMenu('Scripts')
```

Creating Tools

When you want to extend mocha functionality further by using interactive tools, you need to import the mocha.tools API. These set of classes allow you to read mouse events and position, along with registering the necessary tool icon or action in the interface.

Defining the Tool

Tools require the tools module from mocha, along with a couple of PySide modules to be able to create the actions and icons necessary to call the custom tool. We will also use the **find_widget** function from *mocha.ui* to locate widgets in the interface.

Importing the modules for creating a tool

```
from mocha.tools import *
from mocha.ui import find_widget
from PySide2.QtCore import *
from PySide2.QtWidgets import *
```

When defining your tool class, it needs to inherit the **AbstractTool** class from *mocha.tools* to initialize correctly.

Setting up a new tool class

```
class PointHunt(AbstractTool):
    def __init__(self, project):
        action = QAction(None)
        action.setText('Point Hunter Tool')
        action.setIcon(QIcon('/myicons/pointhunt.png'))
        AbstractTool.__init__(self, action)
        action.setParent(self)
        tools_bar = find_widget('ToolsBar', QToolBar)
        tools_bar.addAction(action)
```

Adding an icon using the **setIcon** function command from *PySide2.QtWidget* will still define the icon on the toolbar if your icon file path does not exist.

Once the init class is defined, you can then monitor interaction with the tool using activation and mouse event functions.

Setting up a new tool class

```
def on_mouse_press(self, event):
        print 'Mouse pressed!'

def on_mouse_move(self, event):
        #grab the mouse position on the canvas
    cur_pos = event.pos_on_canvas
    print cur_pos.x(), cur_pos.y()

def on_mouse_release(self, event):
```

```
    sprint 'Mouse released!'

def on_activate(self):
    print 'TOOL ACTIVATED'

def on_deactivate(self):
    print 'TOOL DEACTIVATED'
```

The **on_activate** function is useful for initializing items you only want to occur when the tool has been launched from the toolbar or menu item. A good example of this is to grab the current project on when the tool has become active.

on_activate example

```
def on_activate(self):
    self.proj = get_current_project()
```

The **on_deactivate** function is useful for running items you only want to occur when switching away from the tool by either selecting a different tool or another action.

on_deactivate example

```
def on_deactivate(self):
        release_bees()
```

Example Tool: Spot Cleaner

So great, you can make a tool and monitor mouse position and clicks. What can you do with this? If you know the position of your mouse and can create point data, you can make a lot of useful roto tools.

In the code below we have made a tool that creates a simple 4-point spline to quickly place in a shot for spot removal.

Spot Cleaner Example tool

```
from mocha.tools import *
from mocha.ui import find_widget
from PySide2.QtCore import *
from PySide2.QtWidgets import *

from mocha.project import Layer, LayerGroup, XSplineContour,
 XControlPoint, XControlPointData, get_current_project
class SpotCleaner(AbstractTool):
```

```
    def __init__(self, project):
        action = QAction(None)
        action.setText('Spot Cleaner Tool')
        AbstractTool.__init__(self, action)
        action.setParent(self)
        tools_bar = find_widget('ToolsBar', QToolBar)
        tools_bar.addAction(action)


    def create_spot(self, pos):

        new_layer = self.proj.add_layer(self.proj.clips.values()[0],
name='spot', frame_number=0, view=0)
        points = [[0, 10], [0, 0], [10, 0], [10, 10]]
        x_point_data = []

        for x, y in points:
            x_point = XControlPointData(corner=False,
                                        active=True,
                                        x=float(x) + pos.x(),
                                        y=float(y) + pos.y(),
                                        edge_width=0.0,
                                        edge_angle_ratio=0.5,
                                        weight=0.25)
            x_point_data.append(x_point)

        x_contour = new_layer.add_xspline_contour(0.0,
tuple(x_point_data))
        print "spot created!"
        return x_contour

    def on_mouse_press(self, event):
        cur_pos = event.pos_on_canvas
        self.create_spot(cur_pos)

    def on_mouse_move(self, event):
        pass

    def on_mouse_release(self, event):
        pass

    def on_activate(self):
        self.proj = get_current_project()

    def on_deactivate(self):
        print "All done"
```

```
register_custom_tool_type(SpotCleaner)
```

Custom Exporters

With the ability to dive into most aspects of the project file, it is a lot easier to now write your own tracking, shape and camera solve exporters for your own applications.

To make this more intuitive, we have exporter classes which can register a new export type as part of the standard exporters. In fact two of our new shape exporters, Fusion and Silhouette, have been written entirely in Python.

Creating a Custom Tracking Data Export

The key module you need to import for creating custom exporters is the **`mocha.exporters`** module. This contains the abstract exporter classes necessary to create a new custom export class.

You're also going to need the **`QByteArray`** class from PySide2.QtCore to create the final data output.

Importing the AbstractTrackingDataExporter class

```
from mocha.exporters import AbstractTrackingDataExporter
from PySide2.QtCore import QByteArray
```

As a basic example, let's set up an exporter that will write the x,y coordinates of the surface per frame to a CSV file. First, you need to create a new exporter class that inherits from the **`AbstractTrackingDataExporter`** class. Here we initialize a super class and load the name of the exporter.

Importing the AbstractTrackingDataExporter class

```
class CSVExporter(AbstractTrackingDataExporter):
    '''
    Implementation of the CSV Track exporter.
    '''
    def __init__(self):
        super(CSVExporter, self).__init__('CSV File (*.csv)', '')
```

There are additional parameters you can set to initialize the class:

- **name** (unicode): Name of the exporter which will be displayed in the Export Tracking Data dialog drop-down list. It Should contain a file mask in brackets e.g. *Foo (*.bar)*
- **extension**: Additional extension.
- **number_of_data_streams**: The number of result files required. If it equals to 1 (the default) then the Copy to Clipboard button will be enabled in the GUI.
- **remove_lens_distortion**: Whether the exporter supports removing of a lens distortion.
- **export_multiple_views**: Whether the exporter supports multiple views.
- **export_interlaced**: Whether the exporter supports interlaced footage.

In the example above, we're keeping it simple, so we are leaving the defaults and only setting the export name and file extension.

The brunt of the work is handled in the **do_export** function for the class. This function returns the final data that will go to file or the clipboard.

The do_export function in the final export class

```
from mocha.exporters import AbstractTrackingDataExporter
from PySide2.QtCore import QByteArray


class CSVExporter(AbstractTrackingDataExporter):
    '''
    Implementation of the CSV Track exporter.
    '''

    def __init__(self):
        super(CSVExporter, self).__init__('CSV File (*.csv)', '') #Define
 the CSV exporter
        self._project = None

    def error_string(self):
        return ''

        #Get the corner points of the surface for a given time and layer
    def get_surface_parameters(self, layer, time, view):
        surface_corners = []
        for idx in range(0, 4):
            surface_corners.extend(layer.get_surface_position(idx, time,
 view))
        return surface_corners
```

```
        #Do the actual export
    def do_export(self, project, layer, tracking_file_path, time, view,
options):
        ba = QByteArray()

        in_point = layer.parameter(['Basic', 'In_Point']).get() #Grab the
in point of the layer
        out_point = layer.parameter(['Basic', 'Out_Point']).get() #Grab
the out point of the layer

        for frame in range(in_point,out_point+1):
            surface = self.get_surface_parameters(layer, frame, view)
            result = ', '.join(map(lambda x: str(x), surface))+'\n'
            ba.append(result.encode('utf-8'))
        return {tracking_file_path if
tracking_file_path.lower().endswith('.csv') else tracking_file_path
+ '.csv': ba}
```

The main parameters for the **do_export** function are:

1. project (mocha.project.Project): The mocha project instance you're working with. Usually the currently open project.
2. layer: The layer you want to export the tracking data for.
3. tracking_file_path (unicode): The absolute file path to save which has been chosen by a user in a file dialog.
4. time (PySide2.QtCore.double):The frame index.
5. view (mocha.project.View): The selected view to export.
6. options: A dictionary with keys of type QString and values of type bool. The 3 options for this are Invert, Stabilize and RemoveLensDistortion, which relate to the 3 checkboxes available in the export dialog.

Note that all of the **do_export** parameters will automatically be passed by the interface unless you specifically override them.

In the example above, we define an function **get_surface_parameters** to handle cycling through the position of each corner. Then **do_export** writes the resulting surface [x,y] coordinates to a csv file, one line per frame.

> The use of a **QByteArray** for the actual data is a must in order to properly export.

Customising Existing Exporters

Another advantage of being able to create your own exporters is being able to augment existing exporters to suit your needs.

To do this, you can grab the existing export output and modify it as you require, then assign it to a new exporter.

Example of adding a commented header to a Nuke RotoPaint export

```
from mocha.exporters import AbstractShapeDataExporter


class NukeRotoPaintExtra(AbstractShapeDataExporter):
    def __init__(self):
        super(NukeRotoPaintExtra, self).__init__("Nuke RotoPaint [Basic]
 Extra (*.nk)", "", number_of_data_streams=1,
                                                 export_multiple_shapes=True,
 export_open_splines=True,
                                                 export_multiple_views=True,
 export_interlaced=True)
        self.nuke_exporter =
 AbstractShapeDataExporter.registered_exporters()['Nuke RotoPaint [Basic]
 (*.nk)']

    def error_string(self):
        return ""

    def do_export(self, project, layers, path, views):
        result = self.nuke_exporter.do_export(project, layers, path,
 views)
        header = """#mocha data RotoPaint export
                                        #Version 5.0.0
                                """
        for file_name, contents in result.iteritems():
            result[file_name] = header + contents
        return result


nuke_exporter = NukeRotoPaintExtra()
nuke_exporter.register()
```

Adjusting mocha Preferences using mocha.Settings

The mocha.Settings API provides access to all mocha preferences and control over whether you want the changes to be permanent or just overridden for the session.

This makes it much easier to set up profiles for different users, or define settings based on particular conditions.

For a full list of available setting parameters, see the mocha Python reference.

Below is an example of using the mocha.Settings to override existing saved preferences and then comparing them to the original on disk.

Setting and testing overridden preferences compared to saved preferences

```
import mocha

def settings_diff(settings1, settings2):
    """
    Calculates difference between given PySide2.QtCore.QSettings
 (mocha.Settings) instances.

    :param settings1: left operand
    :type  settings1: mocha.Settings
    :param settings2: right operand
    :type  settings1: mocha.Settings
    :return: difference dict {key: (value1, value2)}
    :rtype: dict
    """
    diff = {}
    assert settings1.group() == settings2.group()
    child_keys = set(settings1.childKeys())
    child_keys.update(settings2.childKeys())
    for key in child_keys:
        value1 = settings1.value(key)
        value2 = settings2.value(key)
        if value1 != value2:
            diff_key_name = "{0}/{1}".format(settings1.group(), key)
            diff[diff_key_name] = (value1, value2)
    child_groups = set(settings1.childGroups())
    child_groups.update(settings2.childGroups())
    for group in child_groups:
        settings1.beginGroup(group)
        settings2.beginGroup(group)
```

```
        diff.update(settings_diff(settings1, settings2))
        settings2.endGroup()
        settings1.endGroup()
    return diff

# Real settings. Changes are immediately written on the disk
real_settings = mocha.Settings(override=False, read_overridden=False)

# Guard to prevent any mocha settings changes
real_settings.setValue = lambda key, value: (_ for _ in
 ()).throw(ValueError("U Can't Touch This"))

# Overridden settings
overridden_settings = mocha.Settings(override=True, read_overridden=True)

# These settings must be synchronized with overridden_settings for reading
 values
checking_settings = mocha.Settings()

# Test that overloads have been not set yet, so the settings must be
 identical
assert not(settings_diff(real_settings, overridden_settings))
assert not(settings_diff(real_settings, checking_settings))
assert not(settings_diff(overridden_settings, checking_settings))

#Toggle an override of the Full Screen setting and test it against the
 saved setting
full_screen = overridden_settings.value('FullScreen')
overridden_settings.setValue('FullScreen', not full_screen)
assert not(settings_diff(overridden_settings, checking_settings))
diff = settings_diff(overridden_settings, real_settings)
assert '/FullScreen' in diff
overridden_settings.setValue('FullScreen', full_screen)

#Toggle an override of the Disable Offscreen buffers setting and test it
 against the saved setting
overridden_settings.disable_offscreen_buffers = not
 overridden_settings.disable_offscreen_buffers
assert not(settings_diff(overridden_settings, checking_settings))
diff = settings_diff(overridden_settings, real_settings)
assert '/DisableFBOs' in diff
overridden_settings.disable_offscreen_buffers = not
 overridden_settings.disable_offscreen_buffers
assert not(settings_diff(overridden_settings, checking_settings))
```

Optimizing threaded Python in mocha

Threading items in mocha Python is possible, however in order to handle the threads, mocha needs to periodically unlock the Global Interpreter Lock(GIL).

There are two parameters to control the GIL locking and unlocking intervals in the root of the mocha settings:

- **Python.VentilateInterval_msec**: This is the interval which is used to run python threads outside of GIL. The default is 300 (Integer), in milliseconds.
- **Python.VentilateSleep_usec**: This is the sleep interval before reenabling GIL. The default is 999 (Integer), in microseconds.

You can time the delay of threads with the following:

```
import threading
import time
import datetime

def test():
    tm0 = datetime.datetime.now()
    for x in range(5):
        time.sleep(1)
    tm1 = datetime.datetime.now()
    print 'Finished:', tm1
    print 'Elapsed :', tm1 - tm0

print 'Started :', datetime.datetime.now()
t = threading.Thread(target=test)
t.start()
```

If you get unexpected delays, you can try tweaking the ventilation parameters to optimize.

For example, try setting **Python.VentilateInterval_msec** to, say, 50 instead of 300 and rerun the script above in the Python Script Editor.

See Adjusting mocha Preferences using mocha.Settings for more details on adjusting mocha settings.

> Negative values of the ventilate settings disable "python ventilation".

Rendering on the Command Line

Since you can use Python to call render operations and export clips, we have written a command line renderer so you don't have to.

Running the mocharender.py tool on OS X

```
'/Applications/Mocha Pro 2019.app/Contents/MacOS/python' '/Applications/
Mocha Pro 2019.app/Contents/MacOS/mocharender.py' [options]
```

Running the mocharender.py tool on Windows

```
'C:\Program Files\Imagineer Systems Ltd\Mocha Pro 2019\python\python.exe'
 'C:\Program Files\Imagineer Systems Ltd\Mocha Pro 2019\python
\mocharender.py' [options]
```

Running the mocharender.py tool on Linux

```
'/opt/isl/mochapro2019/python' '/opt/isl/mochapro2019python/
mocharender.py'
```

Below are the various options to render. After you have typed in your options and pressed Enter, mocha will apply the render to layers with cogs turned on in the saved project file.

In each case you can choose either the abbreviated option (such as -p PROJECT_PATH), or the more descriptive option (--project=PROJECT_PATH):

-p PROJECT_PATH, --project=PROJECT_PATH
Path to the mocha project. E.g. /projects/mochaprojects/Results/myproject.mocha

-g LAYER_NAMES_IN_GROUPS, --group=LAYER_NAMES_IN_GROUPS
Group of layers to render. Specify layer names after the group name to render them only. Duplicated layers will be ignored.

-r RELINK_PATH, --relink-path=RELINK_PATH
Path to the first clip file for relinking. This option is useful if you have sent your project file to a different machine to render and you need to relink the source footage without opening the GUI.

-L MIN_INDEX, --lower-index=MIN_INDEX
Lower clip frame index for relinking. If you are only using a certain frame range for the original clip.

-U MAX_INDEX, --upper-index=MAX_INDEX
Upper clip frame index for relinking. If you are only using a certain frame range for the original clip.

-c CLIP_NAME, --clip-name=CLIP_NAME
Clip name, i.e. the name of the source clip you are using to render with.

-V VIEW_INDEX, --view=VIEW_INDEX
Clip view index. By default this is zero(0), but if you are using a multi-view clip you can set the index here. By default Left and Right views are 0 and 1 respectively.

-D EXPORT_DIR, --export-directory=EXPORT_DIR
Path to the output directory for the rendered export. Note that **`--output-directory`** is now deprecated as it wasn't clear this was the export directory.

-E OUTPUT_EXT, --output-extension=OUTPUT_EXT
Output clip extension. This is where you define your format, such as TIF, DPX, EXR etc.

-P OUTPUT_PREFIX, --prefix=OUTPUT_PREFIX
Output clip prefix. Such as *Remove_*

-S OUTPUT_SUFFIX, --suffix=OUTPUT_SUFFIX
Output clip file name suffix if you require one.

-I FRAME_IN, --frame-in=FRAME_IN
Start frame index. The in point for your render. However, this is deprecated and it is better to use --frames (see below).

-O FRAME_OUT, --frame-out=FRAME_OUT
Stop frame index. The out point for your render. However, this is deprecated and it is better to use --frames (see below).

-R RENDER_TYPE, --render-type=RENDER_TYPE
Rendering operation type (remove, insert, stabilize, reorient). Note the US spelling of stabilize! The *reorient* option is only available in Equirectangular 360 projects.

-v LOG_LEVEL, --verbosity=LOG_LEVEL
Show the render/export operation progress. *-v1* is minimum details, *-v4* is maximum details.

--fbo=FBO
Use offscreen buffers. Use 1 to use frame buffers, 0 to turn them off. If not set, mocha will use the setting in Preferences.

--offset
First file number of the exporting image sequence. If specified with no arguments, the project offset is used.

--frames FRAMES
List of frames and/or frame ranges to render separated a by semicolon. Ranges are presented via python slice notation.

Example Command Argument

```
--frames '0; 2; 3:12:3; 14:17; 19:'
```

Will render frames with the following indices: 0, 2, 3, 6, 9, 12, 14, 15, 16, 17, then 19 onwards until the end frame.

Inverted order is also supported. The ranges must not intersect with each other.

-x, --export-each-frame
Export each frame as soon as it's been rendered. This argument is useful if you do not want to wait for the entire render before you get an export.

-d DEFAULT_OUTPUT_DIR, --default-output-dir DEFAULT_OUTPUT_DIR
Path to the default output render directory (i.e. the *Results* directory). Note this is different from the export output directory where the final files go.

Example mocharender.py usage

I've got a project with two layers (Layer 1, Layer 2).

The following command performs removing contents of Layer 2 on frames 0-1 and saves the resulting clip to the /tmp/rendered directory.

Command

```
$ python mocharender.py --project Markers.mocha "Layer 2"  --export-
directory="/tmp/rendered" --output-extension=png --render-type=remove -v4
 --frames '0:1'
```

Output

```
[DEBUG] 2015-11-05 14:26:14,464 Loading project file: Markers.mocha
[DEBUG] 2015-11-05 14:26:14,749 Project loaded
```

```
[DEBUG] 2015-11-05 14:26:14,750 Preparing for rendering
[INFO]  2015-11-05 14:26:14,752 Rendering started
[DEBUG] 2015-11-05 14:26:14,752 Removing "Layer 2" in Frame 0
[DEBUG] 2015-11-05 14:26:15,528 Removing "Layer 2" in Frame 1
...
[INFO]  2015-11-05 14:26:16,766 Rendering complete
[DEBUG] 2015-11-05 14:26:16,767 Preparing for exporting
[INFO]  2015-11-05 14:26:16,769 Exporting started
[DEBUG] 2015-11-05 14:26:16,769 Saving Clip...
[DEBUG] 2015-11-05 14:26:16,770 Writing /tmp/rendered/0.png
[DEBUG] 2015-11-05 14:26:16,861 Writing /tmp/rendered/1.png
...
[INFO]  2015-11-05 14:26:17,471 Exporting complete
[INFO]  2015-11-05 14:26:17,472 Exported clip has been written to /tmp/
rendered
```

Exporting data on the Command Line

Since you can use Python to export tracking and shape data, we have also written a command line exporter so you don't have to. The mocha exporter also renders matte clips.

Running the mochaexport.py tool on OS X

```
'/Applications/Mocha Pro 2019.app/Contents/MacOS/python' '/Applications/
Mocha Pro 2019.app/Contents/MacOS/mochaexport.py' [options]
```

Running the mochaexport.py tool on Windows

```
'C:\Program Files\Imagineer Systems Ltd\Mocha Pro 2019\python\python.exe'
 'C:\Program Files\Imagineer Systems Ltd\Mocha Pro 2019\python
\mochaexport.py' [options]
```

Running the mochaexport.py tool on Linux

```
'/opt/isl/mochapro2019/python' '/opt/isl/mochapro2019/python/
mochaexport.py' [options]
```

The mochaexport.py script can export layer data to different export formats. There are 4 types of export (see --export-type option):

1. **shapes:** Refers to options in the "Track tab → Export Shape Data…" dialog inside the mocha GUI.

2. **tracking:** Refers to options in the "Track tab → Export Tracking Data…" dialog inside the mocha GUI.
3. **camera-solve:** Refers to options in the "Camera Solve tab → Export Camera Data…" dialog inside the mocha GUI.
4. **rendered-shapes:** Renders layer shapes to file. This is a bit different from other exports. You don't specify --export-type, --exporter-name and --file-path options for rendered shapes. Required options are --output-directory --output-extension, list of layer names.
   Optional inputs are --frame-in, --frame-out, --prefix, --suffix, --index-width.

Below are the various options to export. In each case you can choose either the abbreviated option (such as -p PROJECT_PATH), or the more descriptive option (--project=PROJECT_PATH):

-p PROJECT_PATH, --project=PROJECT_PATH
Path to the mocha project. E.g. /projects/mochaprojects/Results/myproject.mocha

-g LAYER_NAMES_IN_GROUPS, --group=LAYER_NAMES_IN_GROUPS
Group of layers to export. Specify layer names after the group name to export them only. Duplicated layers will be ignored.

-e EXPORT_TYPE, --export-type=EXPORT_TYPE
The Export type. The choices are *rendered-shapes*, *shapes*, *tracking*, or *camera-solve*.

-D EXPORT_DIR, --export-directory=EXPORT_DIR
Path to the output directory for the rendered export. Note that `--output-directory` is now deprecated as it wasn't clear this was the export directory.

-E OUTPUT_EXT, --output-extension=OUTPUT_EXT
Output clip extension. If this option is not set, it will default to a PNG file when exporting rendered shapes.

-P OUTPUT_PREFIX, --prefix=OUTPUT_PREFIX
Output clip prefix. For use if you are exporting rendered shapes. Default is no prefix.

-S OUTPUT_SUFFIX, --suffix=OUTPUT_SUFFIX
Output clip suffix. For use if you are exporting rendered shapes. Default is no suffix.

-V VIEWS, --views=VIEWS
Names or abbreviations of views to export.

-v LOG_LEVEL, --verbosity=LOG_LEVEL
Show the export operation progress

-n EXPORTER_NAME, --exporter-name=EXPORTER_NAME
A name of an exporter. Can accept Regular Expressions as /regexp/. This is required if you are exporting shapes, tracking or camera solves.

-f FILE_NAME, --file-path=FILE_NAME
Exporter output file name.

-t TIME, --time=TIME
Frame time.

-C COLORIZE, --colorize=COLORIZE
Colorize output option. This is used to export the colored version of the mattes. Options are *grayscale*, *matte-color*, or *layer* (for layer id gradient). The default is *grayscale*.

-I FRAME_IN, --frame-in=FRAME_IN
Start frame index. Default is 0.

-O FRAME_OUT, --frame-out=FRAME_OUT
Stop frame index.

-w INDEX_WIDTH, --index-width=INDEX_WIDTH
Output index width. Default is 0.

-L, --exporters-list
If set, the script will output list of all possible exporters grouped by their types.

-i, --invert
Mimes Invert checkbox of the Export Tracking Data dialog.

-R, --remove-lens-distortion
Mimes Remove lens distortion checkbox of the Export Tracking Data dialog.

-s, --stabilize
If set, stabilize data will be exported. Use it together with a tracking exporter type.

--fbo=FBO
Use offscreen buffers. Use 1 to use frame buffers, 0 to turn them off. If not set, mocha will use the setting in Preferences.

--offset

First file number of the exporting image sequence. If specified with no arguments, the project offset is used.

Example mochaexport.py usage

This command exports Layer 1 and Layer 2 shape data from a mocha project to the HitFilm shape format.

Command

```
$ python ./mochaexport.py --project Markers.mocha --export-type="shapes"
 --exporter-name="/HitFilm/" --file-path=/tmp/1.hfcs "Layer 1" "Layer 2" -
v4
```

Output:

```
[DEBUG] 2015-11-05 14:29:41,852 Loading project file: Markers.mocha
[DEBUG] 2015-11-05 14:29:42,137 Project loaded
[DEBUG] 2015-11-05 14:29:42,138 Performing export with 'HitFilm [Transform
 & Shape] (*.hfcs)' exporter...
[DEBUG] 2015-11-05 14:29:42,165 Writing contents to '/tmp/1.hfcs'
[DEBUG] 2015-11-05 14:29:42,165 Done
```

This command exports a rendered shapes clip of Layer 2:

Command

```
$ python .mochaexport.py --project Markers.mocha --output-directory="/tmp/
rendered" --output-extension=png "Layer 2" -v4
```

Output:

```
[DEBUG] 2015-11-05 14:33:40,426 Loading project file: Markers.mocha
[DEBUG] 2015-11-05 14:33:40,713 Project loaded
[DEBUG] 2015-11-05 14:33:40,713 Preparing to export clip...
[DEBUG] 2015-11-05 14:33:40,713 Performing rendered shapes export
[INFO]  2015-11-05 14:33:40,769 Exporting started
[DEBUG] 2015-11-05 14:33:40,769 Saving Clip...
[DEBUG] 2015-11-05 14:33:40,775 Writing /tmp/rendered/0.png
[DEBUG] 2015-11-05 14:33:40,856 Writing /tmp/rendered/1.png
...
[INFO]  2015-11-05 14:33:41,342 Exporting complete
[DEBUG] 2015-11-05 14:33:41,358 Done
```

Updating the GUI

Some Python scripts may require you to update the mocha GUI frequently. Do do this, you can use QCoreApplication.processEvents() in your code:

Using processEvents()

```
from PySide2.QtCore import QCoreApplication
...
QCoreApplication.processEvents()
```

Further Reference

For complete reference of the mocha Python API, see here: https://borisfx.com/support/documentation/mocha/python/