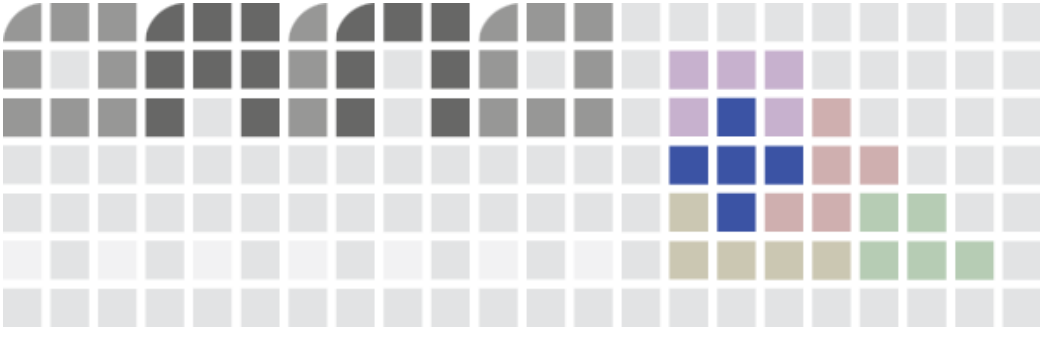


OMINO PYTHON for After Effects

from omino.com, 2010

Contents

Introduction

After Effects is my favorite application! Animations and visual effects and all manner of video can be assembled in a paradigm that's powerful, consistent, and fun. But… I cut my computer graphics teeth, if you will, writing funny little programs on the Apple II, the Commodore 64, and the Atari 400. The joy of typing a for loop in BASIC that draws a spirograph or a moiré or a field of colored boxes is hard to beat.

Lots of people now like to write code to make art. In fact, writing code to make art has even acquired a fancy name: "Generative Art". And there are plenty of exciting apps that let you write code to – that is, to produce generative art. Processing and Nodebox are two of them. And many artists use Adobe Flash and ActionScript for this as well.

But I wanted to work within the After Effects platform.

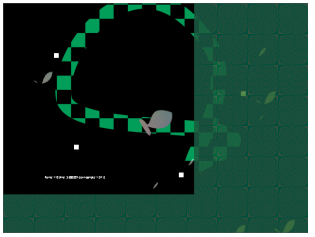
Omino Python for After Effects is an effect plug-in that runs a Python script which can perform drawing commands. Your script is executed once for each frame. Your script can draw something different at each frame based on the time, the frame number, and changes to its parameters. Why Python? I'd heard it was a neat language, and, technically, it lends itself very well to embedding into a plug-in like this.

Host	After Effects on Mac OS X only for now
Required App Verson	After Effects CS 5
Omino Python Version	1.0, January 2011
Required OS Version	Mac OS X 10.6.5
Useful Skills	A familiarity with Python or other scripting language

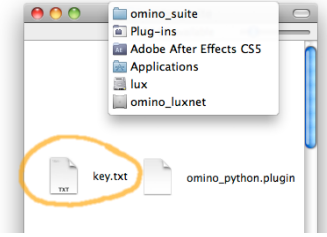
Installation & Licensing

Omino Python.pkg

Omino Python is delivered by a standard Mac OS X installer which walks you through the several steps. Adobe After Effects CS5 must be installed in the standard location.



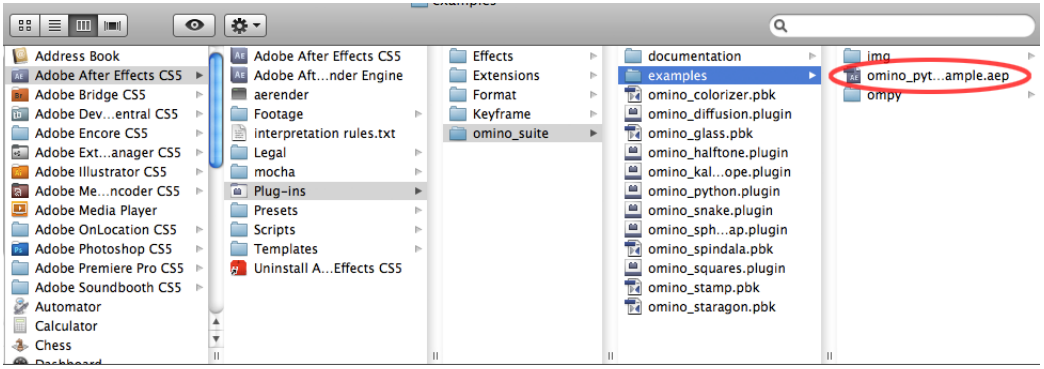
The free version adds a semitransparent imprint on your renders outside of a reduced area, and several minutes duration. The paid version allows rendering at any resolution and duration.



When you purchase a full license key (http://omino.com/store), you can place it in the same folder as the plug-in to disable the limitations.

Quick Start! Instant Gratification

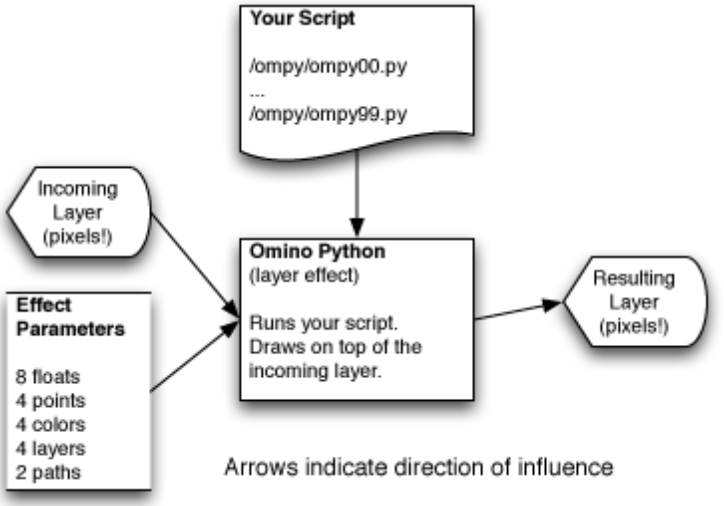
For immediate demonstration, open up the example project, found installed right next to the Omino plug-ins:



It has several compositions which use the example Python scripts in the adjacent ompy/ folder.

The Workflow

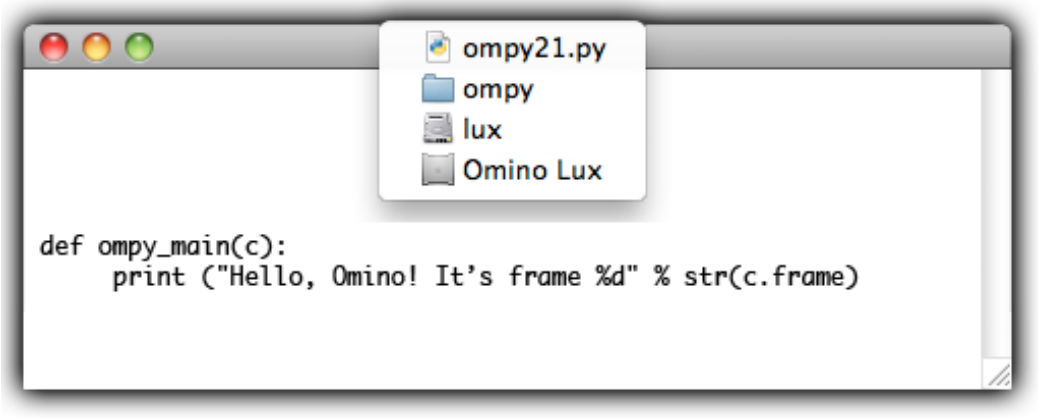
Omino Python is used just like every other After Effects effect plug-in. You attach it to a layer in your composition, and adjust its parameters. Unlike most other effects, Omino Python then reads, and executes, an external script.



This is quite powerful and useful. It also adds a small maintenance burden. After Effects doesn't know about your script file. If you need to hand-off or archive a project which uses Omino Python, you must make sure to include the relevant scripts, too. (And, as usual, any nonstandard plug-ins, including Omino Python.)

A Script

Let's begin. Here is a simple script.

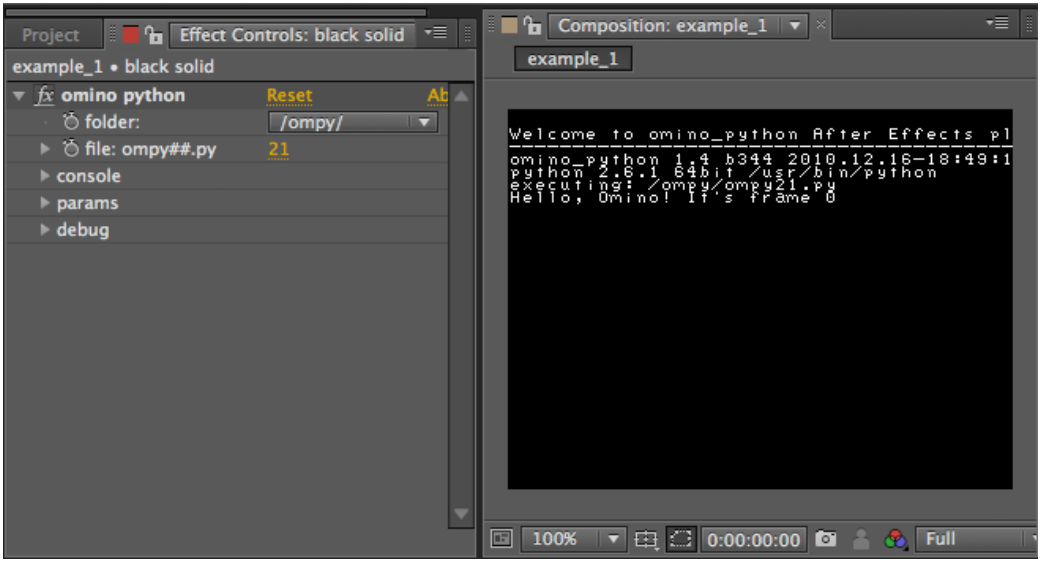


It's shown here in TextEdit. Be sure to choose Format:Make Plain Text, and save as UTF-8, with the exact name, ompy21.py, in a root-level folder named ompy. (Also, make sure that TextEdit doesn't change (c) to a © copyright symbol.) (And plain quotes for plain-text.)

Alternatively, you can use vi from the command-line.

Next, let's apply that script to a 320 x 240 composition with a black solid, like so. We set the file parameter to 21, so that it executes /ompy/ompy21.py.

Here is a screen shot of the script running.



Look! The script has printed some information onto the black solid, ending with the result of our Python print statement. If we preview the animation, the printed frame number will increment.

A Script That Draws

Printing to the console is nice enough, but it's hardly why we fired up After Effects, is it? Let's continue by drawing a rectangle.

Here is the script:

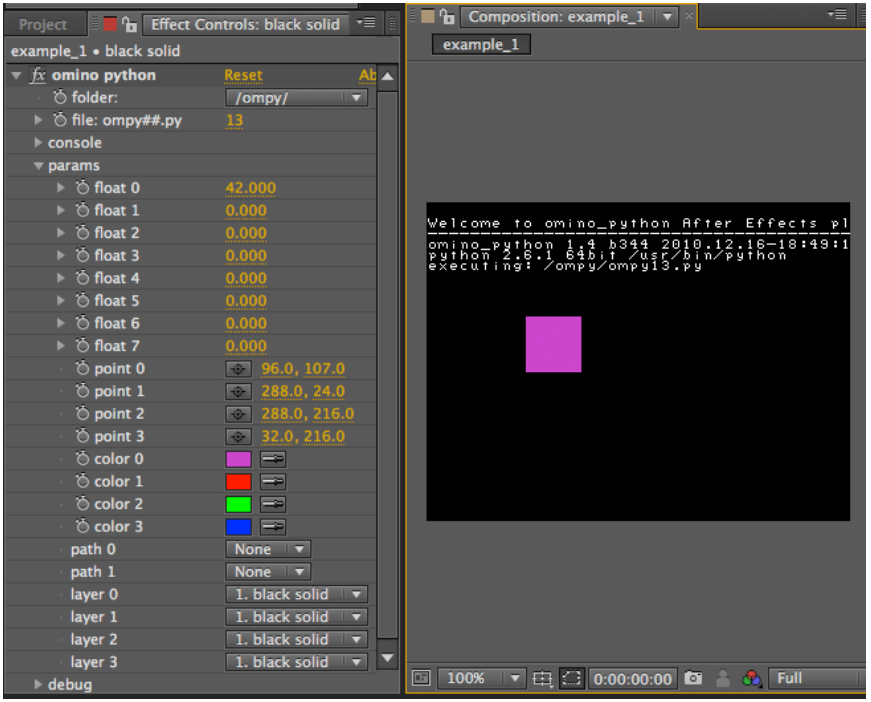
```
#
# Draw a rectangle
# file: /ompy/ompy13.py
#

def ompy_main(c):
    ctx = c.layer_out_context
    color = c.colors[0]
    center_point = c.points[0]
    size = c.floats[0]

    cX = center_point
    cY = center_point
    ctx.rectangle(cX - size / 2,cY - size / 2,size,size)
    ctx.set_source_rgb(color.r,color.g,color.b)
    ctx.fill()
```

Omino Python lets you name your scripts `ompy00` to `ompy99`. This one is number 13. Every Omino Python script must include a function named `ompy_main`. That's what gets called each frame. It gets one argument, `c`, which references a handful of other useful and interesting values.

The first few lines of this script extract a drawing context for the output canvas (`ctx`), a color parameter, a point parameter, and a float parameter.

The latter half of the script draws a rectangle centered on the point.

Here is a screen shot with the running script.



Hopefully this is beginning to make sense. This screen shot shows the "params" parameter group twirled open to reveal a bag of general purpose parameters. This script uses float 0 for the rectangle size, point 0 for the rectangle location, and color 0 for the fill color. But you can use any these parameters for whatever you need. Some paths and layers are available, too. And, since this is After Effects, you can animate them.

The Python Context

Your script's method ompy_main(c) is called for each frame. Here's the names of the parameters and other fields you can access as fields of c in your script.

Field in c	Range	Description
c.floats[n]	0 to 7	A numeric value
c.points[n]	0 to 3	Instance of ompy_point. A point parameter. Each point has fields .x and .y measured in pixels.
c.colors[n]	0 to 3	Instance of ompy_color. A color parameter. Each color has fields .r, .g, and .b, ranging from 0.0 to 1.0.
c.paths[n]	0 or 1	Instance of ompy_path. Each has field .is_closed, and .path_points, Bezier points. Use len(c.path[n].path_points) to find the number of points. Each Bezier point has fields .x_in, .y_in, .x, .y, .x_out, and .y_out, which describes each mask point and its curve-handles.
c.layers[n]	0 to 3	Instances of ompy_layer. Each has field .surface (the Cairo surface) and methods .get_pixel(x,y) and .draw (ctx,x,y,scale,rotation).
c.layer_out_context		The Cairo drawing context for the output bitmap
c.time	0 to dur	The current layer time.
c.frame	0, 1, …	The current frame number of the layer.
c.downsample_x, c.downsample_y		The downsampling resolution of the display. Generally you can ignore this; the drawing context will have an appropriate transformation matrix already applied.
c.width, c.height		The width and height of the current drawing context. This is the full-resolution size, not the current display size.
c.script_path		The absolute path to the current script, like "/ompy/ompy23.py".

The Python module import path always includes the directory containing your script.

The Graphics Library: Cairo Graphics & PyCairo

The graphics package used here is called Cairo, a powerful open source library. It's pretty nifty! Its features are bound to Python with PyCairo.

Here's an introduction to some of the commands in PyCairo. All of these examples assume that `ctx = c.layer_out_context` was run. The general approach is to build up a set of shapes, and then draw them.

ctx.move_to(x,y)	Without drawing, move the "current point" to pixel position (x,y)
ctx.line_to(x,y)	Create a line segment between the current point and (x,y), and add this segment to the "current path". This doesn't draw, yet!
ctx.set_line_width()	Set the width for any lines that will soon be drawn.
ctx.set_source_rgb(r,g,b)	Set the color for any lines or shapes or text that will soon be drawn.
ctx.stroke()	Stroke the current path, with the current width and color, and clear the path.
ctx.rectangle(x,y,width,height)	Add a rectangle to the path, with one corner at (x,y) and the other at (x + width,y + height).
ctx.fill()	Fill the current path with the current color, and clear the path.
ctx.stroke_preserve(), ctx.fill_preserve()	Stroke or fill the current path, but leave it as is. You can further modify it, or draw it again with different settings.
ctx.close_path()	Add a segment from the current point to the first point.

The full PyCairo API is described at http://cairographics.org/documentation/pycairo/2/.

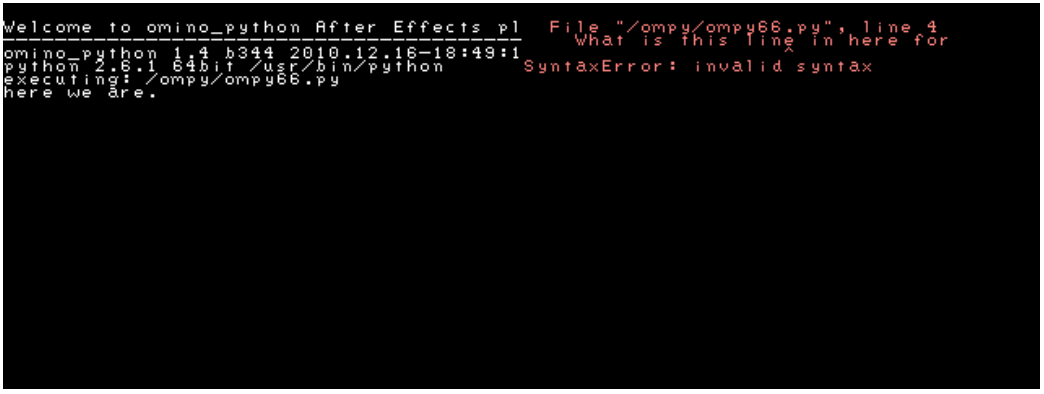
Debugging Your Script

Since computers behave predictably, it is possible, always, to write perfect code, without bugs. Alas, although computers are predictable, we are imperfect, and bugs are a fact of life.

Python has pretty good error messages, and Omino Python displays them for you. Here's a script with a syntax error.

```
def ompy_main(c):
    print("here w are.")
    What is this line in here for???
```

And here's a screen shot of its execution.



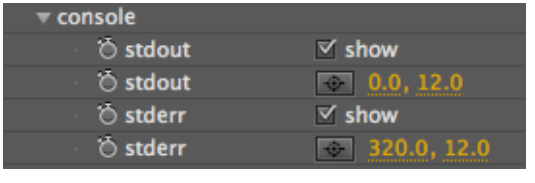
The text in red shows the errors.

After you edit the file to fix the error (and introduce new ones) you can force After Effects to retry your script by either changing any parameter on the effect, or pressing control-clear. (The clear key is on the numeric keyboard, and not visible on laptop keyboards.)

To help debug the functionality of your script -- apart from the syntax errors shown in red -- you can print messages using the Python print statement.

But eventually, it seems likely that you'll get your script working. At that point, the visual clutter of the Welcome text and the error messages will just be in the way.

The "console" section of the parameters can be twirled open, and from here you can move the two text imprints (the white one is stdout and shows print() output, the red one is stderr and shows error messages) or disable them entirely.



Tips & Notes

Composition Display Resolution & Aspect Ratio

In many cases, a pixel spans 1 unit of space, and is square... but not if the layer uses a non-square aspect ratio (such as for certain video and film formats) or is displayed at reduced resolution during preview or scrubbing.

Omino Python applies an appropriate transform to c.layer_out_context so that basic drawing commands are well-behaved, regardless of pixel aspect ratio or reduced resolution. For example, a 640x480 composition at half resolution only has 320x240 pixels. Omino Python will set up the drawing context to reduce all actions by one half; a line from (100,100) to (400,300) at width 5 will hit the pixels from (50,50) to (200,150), with width 2.5.

All is well, for basic drawing.

However, Cairo's bitmap drawing is not affected by the drawing context's transform. As you set up the source pattern, you should scale it by c.downsample_y.

Known Bugs

A Python script can hang After Effects, if it enters an infinite loop. (We're working on this.) The only remedy for now is to Force Quit the application. Be sure to save often...

Support & Questions

This product is still being developed. Please send any questions, suggestions, bug reports, and other feedback to python@omino.com, subject should start with “plugin”.

And if you do something neat with it, let me know!

Happy Pixeling.

Appendix A: Installation Details

The Omino Python installer installs several libraries. For most users, this should be transparent and unobtrusive. If you are a software developer, or otherwise use lots of libraries, this information may be useful.

Python Cairo bindings	`/Library/Python/2.6/site-packages/cairo/`
Cairo libraries	`/usr/local/lib/libcairo*`

Support libraries	`/opt/local/lib/`
	`libexpat.1.dylib*`
	`libfontconfig.1.dylib`
	`libfreetype.6.dylib`
	`libiconv.2.dylib`
	`libpixman-1.0.dylib`
	`libpng12.0.dylib`
	`libz.1.dylib`

Omino Plug-ins	/Applications/Adobe After Effects CS5/Plug-ins/omino_suite/
Documentation	<Omino Plug-ins>/documentation
Examples	<Omino Plug-ins>/examples
Python Scripts	`/ompy/ompy00.py` is provided as a starting point