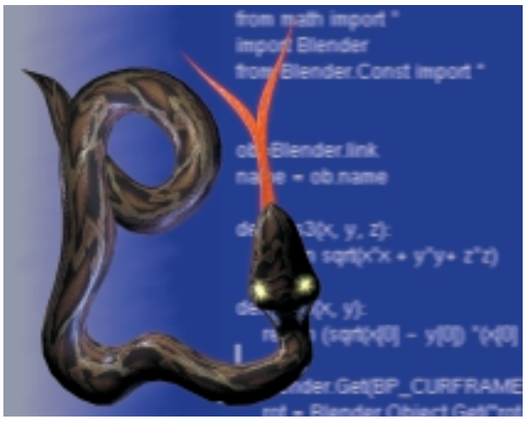
Writing Blender scripts in Python

SNAKE CHARMER

MARTIN STRUBEL



In versions greater than 1.67 and with the C-Key extensions the well-known animation package Blender allows users to explicitly manipulate 3D objects and their attributes. The script language it uses to do this, Python, is popular on every platform and very easy to learn. In this article we will make a start writing Blender scripts with Python.

For the time being, the Python extensions are not available in the freeware version of Blender. This means that without a C-Key you cannot try out the examples provided. However, with Blender 2.0 (the Blender games development system), there is a certain dynamism about Python which could help those who do not yet own a C-Key to feel that acquiring one would be worthwhile.

First steps

We do not intend to describe in detail the structure and implementation of Python in this article. At the Python home page you will find a complete user manual on Python, which you do not have to read through right now. Python is actually very simple and you could probably learn it standing on your head. However, we must define some key concepts. As the term "object" is used both in Blender and in Python, we intend to describe a specific Python object as "PyObject" in the rest of the article. However, just to immediately confuse you again, an object in Blender can also be a PyObject – in fact, as soon as you address a Blender object using Python. But, a PyObject can also be a material or a light source in Blender and is not simply a variable or a data record. The best thing to do is to take a look at how we manipulate Blender objects. And the best way to do this is to start the Blender in a window (not full-screen) from the shell:

```
blender -p 0 200 640 480
```

This way you can still view the shell window displaying the current *stdout* and *stderr* of the Blender Python module. Now call up the text editor in Blender using [Shift+F11] and select "Add New" in the menu panel. Now you are ready to start writing. Try out the famous Hello program, somewhat extended:

```
# Everything after the "#" is a comment
a = 1
print a
print "hello"
a = a + 1
print a
```

Now run the script using [Alt+P]. Good, that was easy. However, we can establish straight away that "a" is a PyObject, and one of the simplest at that: an integer value (int). You can use:

```
print type(a)
```

to establish what type of PyObject it is the output in the shell: <type 'int'>. Make a small change to the script by setting "a = 1.0" instead of "a = 1". Check again what type a is. Aha!

Now we take the surface which is usually presented first when we start Blender. Its default name is Plane (OB:Plane) and it is controlled using the EditButtons menu [F9] (see Figure 1). Please note: the relevant Polygon object (Mesh) is also called Plane. However, we are only addressing its entity (i.e. the Blender object itself) and therefore always use the OB name.

Enter the following script and run it using [Alt+P]:

```
import Blender
obj = Blender.Object.Get("Plane")
obj.LocX = obj.LocX + 0.5
obj.RotZ = obj.RotZ + 0.2
Blender.Redraw()
```

Something happened! You can print out the co-ordinates using *print* if you want to check them.

And now a brief explanation. The function Blender.Object.Get() waits for the name of the Blender object as an argument and delivers the pointer to the data record (for C hackers: in the same way as *struct*) for the PyObject concerned as a return value. Thus we have specifically allocated a PyObject to the Blender object: if we change the attributes of the PyObject *obj*, the attributes of the Blender object change in the same way. However, this function is not installed in Python – clever readers have already guessed – it is located in the module called Blender which must first be imported.

Blender.Redraw() allows the objects to be redrawn (so that you can also see the effect immediately). Of course, this is not necessary when you compute an animation.

Obj.LocX is – quite obviously – the X co-ordinate of the plane (strictly speaking of the purple centre point) and obj.RotZ the angle of rotation around the Z axis, where the unit is radians (a circle – i.e. 360 degrees – corresponds to 2*Pi or around 6.28). We will look at how to query Pi as a variable later.

That was the basics, but we will also show you a few tricks so that you can check out all the Python functions in Blender.

Hierarchical society

As you can guess, Python has a similar type of class hierarchy to C++ or Java. The *dir* function provides you with a list of strings which contain the names of the class members (or methods) of the argument. Try out another script ("ADD NEW" in the menu):

```
import Blender
print dir(Blender)
```

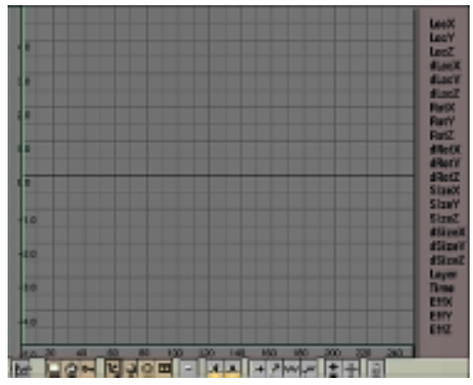
Your output will be something like:

```
['Camera', 'Const', 'Get', 'Lamp', 'Materia
l', 'NMesh', 'Object',
'Redraw', 'World', '__doc__', '__name__', 'b
ylink', 'link'].
```

Now try to move further down the hierarchy, for example using:

```
print dir(Blender.Object)
```

Everything alright? In principle, you can scout out new functions yourself (which you will probably receive with each new version of Blender). The most



[top]
Figure 1: The EditButtons menu [F9]

[left]
Figure 2: The IPO window

important functions are those shown in the following form:

```
Blender.<class>.Get("<Name>")
```

For <class> you can use almost anything you obtained above with *dir*: Camera, Lamp, Material, Object and World. As a return value you always receive a data record object, the type of which corresponds to the class (and, accordingly, its attributes depend on it too). The aforementioned term "method" always stands for a function which is applied to a specific PyObject, e.g. one of the standard methods is the function to attach a PyObject to a list (the method list.append).

Blender.Get() is also a method. But what is a list? If you do not enter anything as an argument for Blender.Object.Get(), you will not receive the data record of an object as a return value but a whole list of all the objects. For example:

```
import Blender
obj = Blender.Object.Get()
print obj
print len(obj)
```

delivers the following output:

```
[[Object Camera at: <0.000000, -8.128851
, 0.000000>],
[Object Plane at: <4.500000, 0.000000, 0.
000000>]].
```

List elements can be addressed in the same way as arrays in C: the X co-ordinate of the object plane can therefore be queried or changed using obj[1].LocX in this instance. The number of elements in the list can be established using len(). In this case print len(obj) produces the result "2".

Now, of course, we ask what else can be manipulated apart from the co-ordinates. Answer: almost all the attributes which can be controlled using the IPO curve. Simply switch to the IPO editor [Shift+F6] and view the attribute names on the right-hand side (see Figure 2).
We want to show you a few examples (see Table 1).

As the last example once again illustrates, the data blocks retrieved using the different functions are

Table 1: Blender scripting with Python

```
cam = Blender.Camera.Get("Camera")
x = cam.Lens
        x = "focal distance" of the
        camera lens
cat = Blender.Object.Get("cat")
cat.SizeZ = cat.SizeZ / 10
        Poor cat (no comment)
mat = Blender.Material.Get("Blue mat")
mat.B = 0.0
mat.R = 1.0
        We have coloured the blue
        mat red…
la = Blender.Lamp.Get("Lamp")
la.Energ = la.Energ - 0.1
ob = Blender.Object.Get("Lamp")
print "co-ordinates:", ob.loc
        We want to dim the light a
        little – but notice: la and ob are
        not the same!
```

not the same. Try to establish the data type (as above with *print type*). If we select the lamp and switch to the EditButtons menu [F9], we see Figure 3:

The variables *la* and *ob* have been used in the example above to give you the gist. *la* is a PyObject for the data record of the lamp parameters and *ob* the parent object for this data – the lamp entity with the name "Lamp". We already know that an object in Blender defines position, rotation, size etc., and refers to a data structure of the corresponding object type (Lamp, Mesh, Surface, Camera, etc.). In the relevant PyObject this happens via the pointer *ob.data*. Again you must note the data type shown by *ob.data*. In the example above with the lamp, *ob.data* points to *la* of type Lamp. We could therefore write the above example differently:

```
ob = Blender.Object.Get("Lamp")
la = ob.data
la.Energ = la.Energ - 0.1
```

What is the point in saving the parameters separately like this? Those of you who have clashed with Blender's object hierarchy will certainly know the difference between "linked copies" and normal "copies" (single user copy). An object can share its parameters with several other objects in other positions, i.e. if I change these parameters, all the partner's parameters change in the same way.

Therefore, you can duplicate ten lamps "linked" (using [ALT-D]) and use *la.Energ* to change their brightness at the same time. To do this, you enter the "LA:" name at Blender.Lamp.Get(). However, if I simply wish to change the position of the individual lamps, I use *ob.loc* and have to enter the "OB:" name at Blender.Object.Get(). And now we should schedule a coffee break before things become too confusing.

Before we go for coffee, though, I would just like to add one thing. Those of you who (with selected lamp) switch to the IPO window and click on the lamp icon will find more attributes of the PyObject *la* of type "Lamp". And those of you who wish to view this whole object hierarchy can do so in the OO window using [Shift-F9].

Complex images

Now it gets serious. The great thing about scripts is that you can program complex animations quite easily instead of having to enter them laboriously by hand as IPO curves. The attributes of an object can depend on the attributes of an object animated using an IPO curve or directly according to time. We have already seen an example of the first case; now need to access a time variable. We get this in the form of the "frame number" using:

```
time = Blender.Get(Blender.Const.BP_CURTIME)
```

The inquisitive reader who immediately tries out

```
print dir(Blender.Const)
```

finds yet another variable BP_CURFRAME. The difference between this and BP_CURTIME is that Blender.Get() provides an integer value (indicating the currently rendered "frame"). In contrast, the time values are not necessarily integer values, e.g. where half images (in PAL format) are rendered, or rendering involves "Motion Blur".

Note: It is usually a good idea to deduct 1.0 from the time variable so that the animation begins with time = 0.0. Now it would be good if the script were retrieved automatically whenever an object was moved or whenever a new frame was rendered. It's all possible! Select the object concerned and switch to the ScriptLink menu (see Figure 4)

On the right-hand side are the scene script links: click on "New" and enter the name of the script in the text field (in the example *taumel.py*). This is retrieved each time the frame is changed. On the left-hand side are more link options. Depending on which type of object was selected you will see the symbols for object, material, lamp, world etc., in the menu panel. However, we will not go into more detail about these link types just now.

Let's try it out. Let's make a virtual lamp sway around and flicker a little. We want to do this using Scene-Link. We begin by adding a lamp using "ADD NEW->Lamp". However, we have only one light source. We want to see the lamp properly and so we add another "Plane", delete three vertices from it in EditMode and move the one vertex to the position of the lamp, which we make the parent of the vertex so that it moves with the lamp too. As material we set a red halo. Our script can be seen in Listing 1.

In conclusion, the script is linked to the scene via the ScriptLink menu so that it is retrieved when the animation is played (Alt+A) and during rendering, as described above. A small snapshot, rendered using Motion Blur, can be seen in Figure 5:

A little bit of math and a little bit of chance

Without any further explanation, we have imported the math module. It contains the functions of the standard C library, as you will see if you use print

Figure 3: The EditButtons menu [F9]

Figure 4: The ScriptLink menu

dir(math). Here we also find the value of Pi promised earlier: *math.pi*. We used the statement:

```
from math import *
```

so that we do not always have to type in the module prefix *math*. The math module functions used to calculate the circular movement or the sway in the orbit are *sin()* and *cos()*, best known in the context of trigonometry. In addition, we would like to have a greater element of chance. For this there is an extra module by the name of *whrandom* in the standard Python directory (under Linux usually */usr/lib/python1.5* or */usr/local/lib/python1.5*.) However, the environment variable $PYTHONPATH is not set everywhere and so the standard system path may not be found. Using *import sys*, the system path can be replaced with the assignment

```
sys.path = ['/usr/lib/python1.5','...', ...]
```

(or extended with *sys.path.append()*) ? or, alternatively, via the environment variable $PYTHONPATH. The random function *whrandom.random()* always provides a floating decimal point value between 0.0 and 1.0. You can read everything else from the script or simply try it out in the example file.

A few pearls of wisdom

We can really do something with the methods described. However, things become very interesting – but more time-consuming – as soon as we begin to simulate functions which are no longer as easily predictable but depend on the position of other objects (e.g. collisions, chaotic functions etc.) This is a topic for another time.

Empties are very useful for establishing the starting position of an object. You can also use these as a kind of Slider without having to enter variables in the script. If you wish to simulate cannon fire, for example, you can stipulate the starting position and firing direction of the canon ball using two Empties.

The fact that variables and modules are not deleted after the script is retrieved but are always available in the memory – and globally too – is very important. Therefore, if script A sets a variable, script B can read it again. Of course, this can be very useful, but it can also be somewhat confusing at times. When experimenting with scripts, you should test your script to establish whether it is foolproof by saving the work, restarting Blender, reloading the file and running the script again.

There are still a few things we haven't discussed yet. Since version 1.69 we have been able to manipulate or query the vertex data of a mesh and the original text co-ordinates directly. This is particularly interesting for the export and import of models (e.g. for Quake2 etc.). It also enables you to generate complex objects easily – Lindenmayer systems or genetic algorithms used to create plant-like objects and trees come to mind. In addition, modules can be developed in C which can be loaded quite easily as dynamic libraries (like the Blender plug-ins) using *import <module>*. So anyone who still thought that Python was too slow as an interpreter language has hopefully been convinced otherwise.

The rapid development of Blender allows us to dream. In future versions there will be built-in collision detection, and work is under way on extensions allowing users to use the Blender GUI from Python. There are likely to be more new features by the time you read this. Therefore, stay on the ball and allow yourself to be surprised. ■

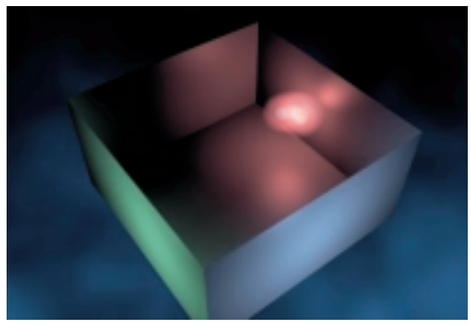


Figure 5:
The sway script in action

Listing 1: The dance of the lamps

```
# sway.py by ms, 11.1999

from Blender import *
from math import *
import whrandom

# Number of frames for "once around" -
# the higher the number, the more slowly th
e lamp sways
speed = 100
pi2 = pi * 2

lamp = Object.Get("Lamp")
box = Object.Get("Box")

t = Get(Const.BP_CURTIME) - 1.0    # Start a
t 0.0

# Make the lamp sway, taking into considerat
ion the size of the
# box  - change the size of the
# box in order to test it and press Alt-A again

# the radius of the orbit should oscillate so
mewhat
r = box.SizeX* (0.7 +  0.1 * sin(10* t * pi2 /
speed))

lamp.LocX = r * cos(t * pi2 / speed)
lamp.LocY = r * sin(t * pi2 / speed)

# Make the lamp flicker:
lampdata = Lamp.Get("Lamp")
r = whrandom.random()
lampdata.Energ = 1.0 + 0.5 * r

# Also make the halo size flicker:
mat = Material.Get("Halo")
mat.HaSize = 0.10 * (1.0 + 0.5 * r)
```

Info

Python home page:
http://www.python.org/
Brief Python documentation on Blender:
http://www.blender.nl/complete/index.html
Blender
http://www.blender.nl/shop:

■