

Blender: first steps with Python

Thanks to Diamond Editions for authorizing online publication of this article, first published in Linux Magazine N°73

Saraja Olivier – olivier.saraja@linuxgraphic.org
English translation: Ed Montgomery

You have discovered thanks to Yves Bailly in GNU/Linux Magazine N°68 (article online at <http://www.kafka-fr.net>) the possibilities offered by the coupling of Blender and Python, in the framework of developing graphics. So we are going to leave for the moment, the scene description language of POV-ray (largely explored in previous issues) in order to discover how the Blender Python interpreter allows us to approach the same level of power and versatility as the standalone version of Python.

But first of all, as in all programs, it is necessary to define objectives. In the framework of learning python, we are going to simply... try to reproduce the opening default scene of Blender! So we are going to learn to create a cube, assign it a material, then add a lamp and a camera. Each of these objects will be in the exact same positions and orientations (in particular the camera and the lamp) as those of the default scene.

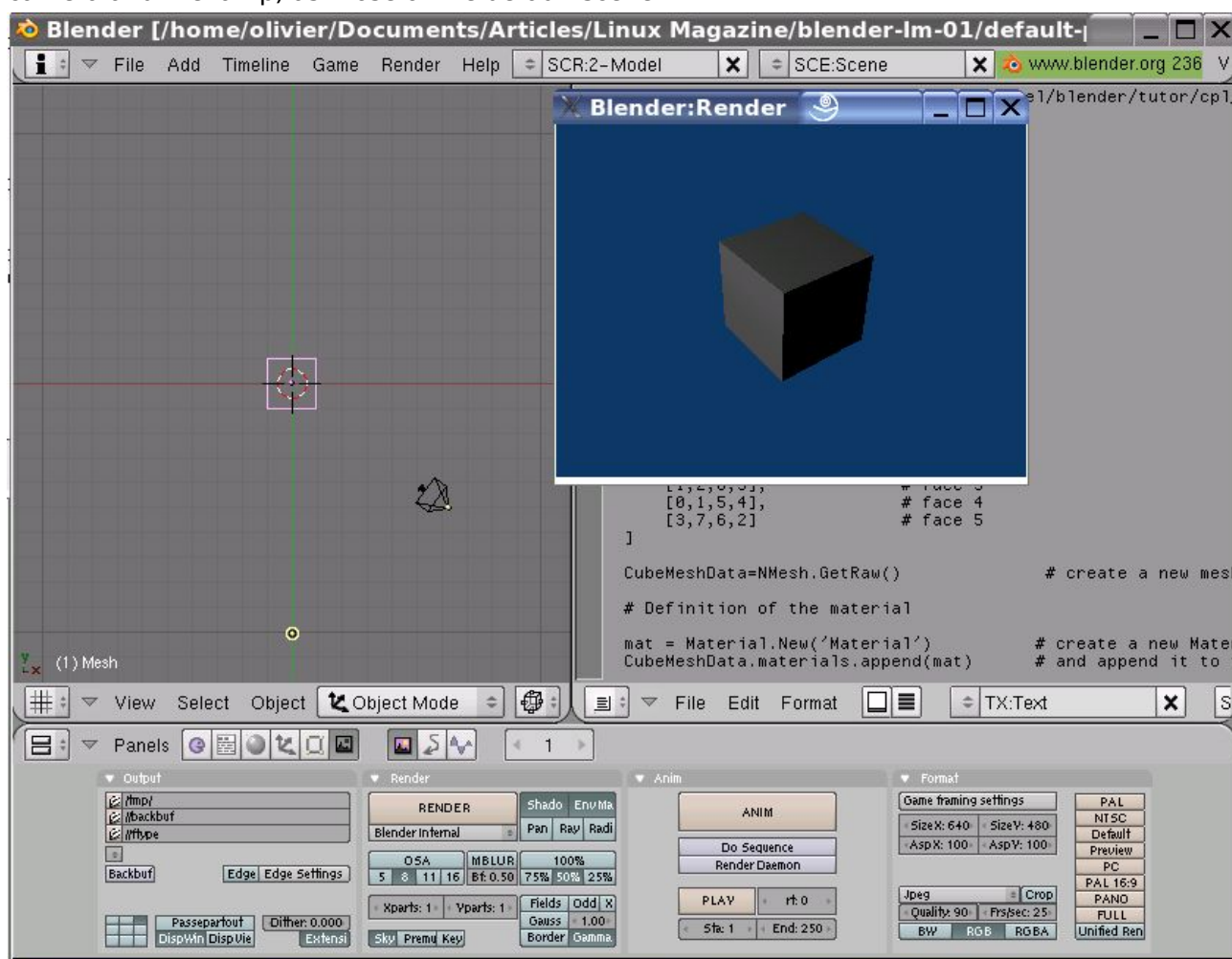


Figure 1: An easy to obtain objective? Yes, we will see how...

1.A well configured python

A large number of functions that I will describe are embedded in the python API of Blender. The result is that, with the actual scripts, it is less often necessary to have a complete installation of

python on the harddrive, perhaps not even a separate python installation at all. There are sometimes certain modules offered by Python that you will need, such as the math module, for example, which we will marginally need today.

1.1 Setting the Pythonpath:

Most distributions actually provide a version of python, probably already installed, and all that is required is to confirm that it is installed. Then all you need to do is give your system the path which points to python. In order to determine the path(s), there is a simple procedure to follow. Open a console or term program, and type the command `python`, and then confirm this by pressing the Enter key.

```
Python 2.3.4 (#1, Feb 7 2005, 15:50:45)
[GCC 3.3.4 (pre 3.3.5 20040809)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You have just entered the console mode for python. Then type `import sys` followed by `print sys.path`, and you will obtain something like:

```
['', '/usr/lib/python23.zip', '/usr/lib/python2.3', '/home/olivier',
'/usr/lib/python2.3/plat-linux2', '/usr/lib/python2.3/lib-tk',
'/usr/lib/python2.3/lib-dynload', '/usr/lib/python2.3/site-packages']
```

Now open the file `.bashrc` from the root of your home directory (`/home/moi`). At the bottom of this file, add the following lines:

```
export PYTHONPATH=/usr/lib/python23.zip:/usr/lib/python2.3: /
usr/lib/python2.3/plat-linux2:/usr/lib/python2.3/lib-tk:/usr/lib/python2.3/lib-
dynload:/usr/lib/python2.3/site-packages
```

Note the above separators: (no apostrophies, and paths separated by colons), this is the exact content dictated by the command `print sys.path`.

1.2 Specifying the default path for scripts

Start Blender, and click on the icon, at the bottom of the 3D view, indicating the window type for the viewport. Choose **User Preferences**.



Figure 2: choose User Preferences in order to set the paths for your scripts

The window changes appearance, and this makes it possible to personalize the behaviour of Blender in several different categories: **View & Controls**, **Edit Methods**, **Language & Font**, **Themes**, **Auto Save**, **System & OpenGL** and finally, the category which interests us: **File Paths**. Click on this last button in order to discover all the default paths of Blender. The last line, second column, sets the directory reserved for Python. Now you only need to click on the small folder to open the mini-navigator for files and directly select the folder where you will keep your python scripts.

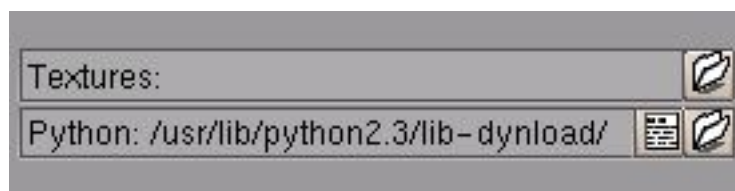


Figure 3: and voila! The python scripts path has been set!

Now all that remains to do is to return to the regular 3D view, and pressing on [CTRL]+[U] in order to save the user's preferences. Thus, the next time you launch Blender, and each time you open a new file, it will be placed in this folder.

2. Preliminaries and basic advice

I strongly advise that you start Blender from a terminal or console, in order to see the messages attached to the console by the python interpreter, in the course of executing your scripts. In this way, if your code contains syntax errors or unknown commands, it will be easier for you to understand the problem and to debug it.

Once Blender has started, separate the 3D view into two parts with a right click on the horizontal

separation line between the 3D view and the command window (the cursor will change shape). Then choose **Split Area** and confirm with a left click.

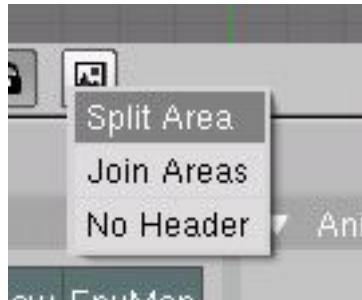



Figure 4: preparing your work space

In the right hand viewport, open the text editor window ([SHIFT] + [F11]) and create a new file by using the combination [ALT]+[N] or use the menu under the window: File > New. Activate the

attachment of line numbers by clicking on the icon , this will prove to be very useful, if not essential.

For the first few lines, we are going to go quite easy. First, we will define the « skeleton » of our script:

```
01: import Blender
02: from Blender import Nmesh, Scene, Object
03: Blender.Redraw()
```

For the moment, it appears that this program does absolutely nothing. However, you may execute the script by pressing the combination of [ALT]+[P] while taking care that the mouse pointer is in the text editor window.

3. Creation of a cube

So, what is the reason for this? If python would permit us to perform some marvels, something as simple as a command `cube {<x1,y1,z1><x2,y2,z2>}` as is the custom with POV-ray, this would be wonderful. In effect, we do not work here with primitives, but with some meshes. And the creation of a mesh follows a coherent but demanding protocol. We are going to illustrate with a simple case, abundantly commented.

3.1 Creation of a face or surface

To begin, we start simply. Let's suppose that we want to construct a plane as shown in Figure 5. It consists of 4 points or vertices, which joined together, form a face. It is then easy to build the coordinates of the 4 vertices. For convenience, we are going to number them from 1 to 4, or actually from 0 to 3. In reality, computer scientists have some strange habits, and they begin numbering from 0 rather than 1, but that just requires some practice to which one becomes rapidly accustomed.

```
vertex 0: [-1,-1,0]
vertex 1: [-1,+1,0]
vertex 2: [+1,+1,0]
vertex 3: [+1,-1,0]
```

Now we are easily able to define our surface as that which relies on the points 0 to 3:

```
face 1: [0, 1, 2, 3]
```

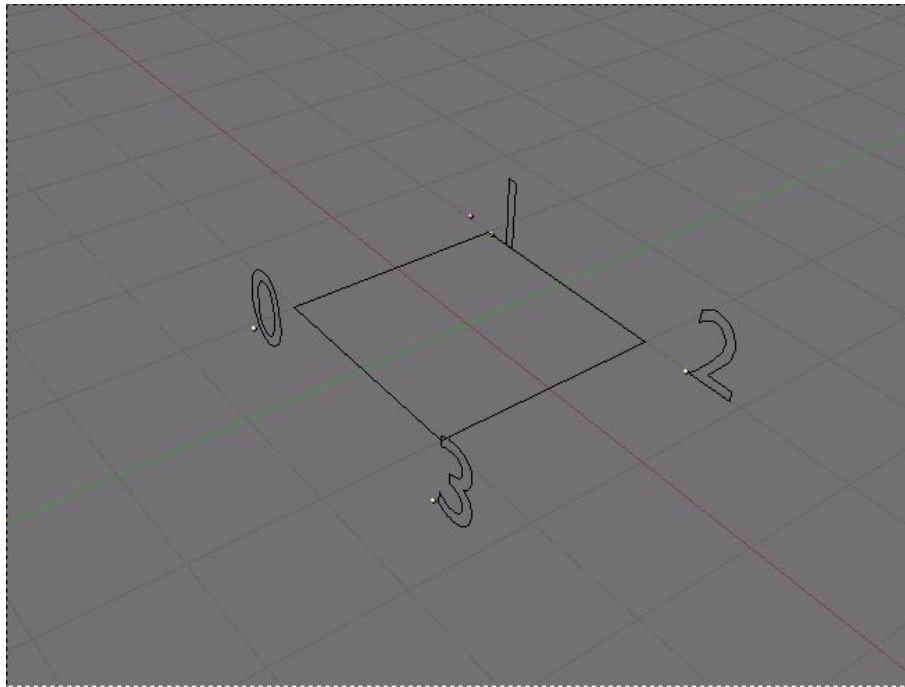


Figure 5: our face consists of 4 vertices numbered from 0 to 3

We can now write the following code. The first two lines permit the initialization of python and the appropriate Blender modules. In these circumstances, as we are going to concentrate on generating a mesh for the moment, we only need to load Nmesh functions.

```
01: import Blender
02: from Blender import Nmesh
```

On the following line, we create an object of type mesh in memory, but for the moment it is a completely empty object. We have just reserved some space in memory for it, under the name of `plandata`. Avoid the use of periods in the middle of names that you specify, if you want to avoid some python bugs.

```
03: plandata=NMesh.GetRaw()
```

Now we will define the coordinates of the four points that will constitute our surface. `vertex` is a name that we have arbitrarily chosen, and the coordinates of the proposed vector (`Nmesh.Vert`) are shown in figure 5. On the following line, we demand that python add (`append`) to the internal mesh list of `plandata` the coordinates of `vertex` in the list of vertices (`Nmesh.Vert`). We will repeat these two lines of code for each vertex of our face, while only changing the coordinates. In the internal structure, the coordinates of the first vertex are arranged in the variable `verts[0]`, those of the second vertex in `verts[1]` and so on just to the n th vertex in the range `verts[n-1]`.

```
04: vertex=NMesh.Vert(-1, -1, 0)
05: plandata.verts.append(vertex)
06: vertex=NMesh.Vert(-1, +1, 0)
07: plandata.verts.append(vertex)
08: vertex=NMesh.Vert(+1, +1, 0)
09: plandata.verts.append(vertex)
10: vertex=NMesh.Vert(+1, -1, 0)
11: plandata.verts.append(vertex)
```

Now we will reserve memory space for the facet, under the very simple name `face`, thanks to the following line:

```
12: face=NMesh.Face()
```

Now we are able to define the vertices that make up the facet. This is done simply by adding (`append`) `verts[0]` to `verts[3]` in the internal list of vertices (`v`) which constitute the facet `face` of our mesh `plandata`. In the lines which follow, only `face` and `plandata` are the names chosen by the user, the rest refer to functions and internal variables of the python module of Blender.

```
13: face.v.append(plandata.verts[0])
14: face.v.append(plandata.verts[1])
15: face.v.append(plandata.verts[2])
16: face.v.append(plandata.verts[3])
```

The vertices which build our face are now defined. We need only add (`append`) our facet `face` to the list of faces of our mesh `plandata`. Once again, only `face` and `plandata` are names chosen by the user.

```
17: plandata.faces.append(face)
```

And there it is, the mesh is totally defined and built. However, it only exists in a virtual fashion, in the memory of your computer. We are going to gain access to Blender's geometry thanks to the following line, where `plandata` is the name that we have given to the mesh in memory, `Plan` is the name which carries the object in Blender (one finds this name in the menu **Editing** (function key F9) in the field `ME:` and `OB:`, when after execution of the script, the face is created and is selected in the interactive 3D window).

```
18: NMesh.PutRaw(plandata, 'Plan', 1)
```

Blender is now ready to render the mesh, it just requires that it refreshes the viewport for the user to be visually informed of the success of his python script.

```
19: Blender.Redraw()
```

3.2 On the use of loops

Not content to insult my computer scientist friends while saying that they have a special way of counting, I will continue by adding that they are a little lazy. We are able to rely on a computer scientist to invent a tool which will do the boring and repetitive operations in his place. Imagine, if I had to describe a mesh consisting of several hundreds of faces, I would certainly not have the desire to chain them together *ad nauseam* the `[name.vertex]=NMesh.Vert([coordinates]), [name.data].verts.append([name.vertex], [name.face].v.append([name.data].verts [number.vertex])).` I would much rather have a tool to do it for me. By chance, my computer scientist friends have invented loops.

We note that it is possible to put in place a loop which will repeat as many times as necessary for the generation of points. For this, we are going to create a vector that will contain the vertices to generate.

```
# Definition of vertices or 3D points
list_of_vertices=[
    [-1,-1,0],          # vertex 0
    [-1,+1,0],          # vertex 1
    [+1,+1,0],          # vertex 2
    [+1,-1,0]           # vertex 3
]
```

Then at the moment of creating the vertices, rather than making them one by one, we demand that python go through the vector `list_of_vertices` that we have defined, and for each one in the list, obtain the noted values in order to place them in the variables named in `composante[i]`. We can then obtain the mesh of a line permitting the generation of a vertex. The following line, of course, allows adding the created vertex to the list of vertices already created for the object `plandata`.

```
for composante in list_of_vertices:
    vertex=NMesh.Vert(composante[0], composante[1], composante[2])
    plandata.verts.append(vertex)
```

Attention

The important things to remember are:

- do not forget the `:` at the end of the line `for... in... ;` in effect, it indicates the start of the loop.

- keep an identical indentation just to the end of the loop; in particular, it is formally forbidden to mix spaces and tabs from one line to another, even if the visual result seems the same: python will not interpret it correctly and will return an error!

- you can nest several loops inside each other, and it will be the indentation that

allows python to correctly interpret your wishes; for example, the first loop will begin with a tab, the second, nested in the second, will begin with two tabs. If one part of the second loop is represented with a single tab, python will interpret that as being part of the first loop.

3.3 Creation of a Cube

We now know how to define a list of vertices, and to use a loop to automate the generation of points. Now all we need to do is to learn to define a list of faces or facets, and how to use a supplementary loop to also automate the generation of faces.

Observe the cube from the default scene, and below we have attached a reproduction of the coordinates of these points. From figure 6, we can define the following points:

```
vertex 0:  [-1,-1,-1]
vertex 1:  [-1,+1,-1]
vertex 2:  [+1,+1,-1]
vertex 3:  [+1,-1,-1]
vertex 4:  [-1,-1,+1]
vertex 5:  [-1,+1,+1]
vertex 6:  [+1,+1,+1]
vertex 7:  [+1,-1,+1]
```

then the following faces, as they are made by the points indicated between brackets:

```
face 0:  [0,1,2,3]
face 1:  [4,5,6,7]
face 2:  [0,4,7,3]
face 3:  [1,2,6,5]
face 4:  [0,1,5,4]
face 5:  [3,7,6,2]
```

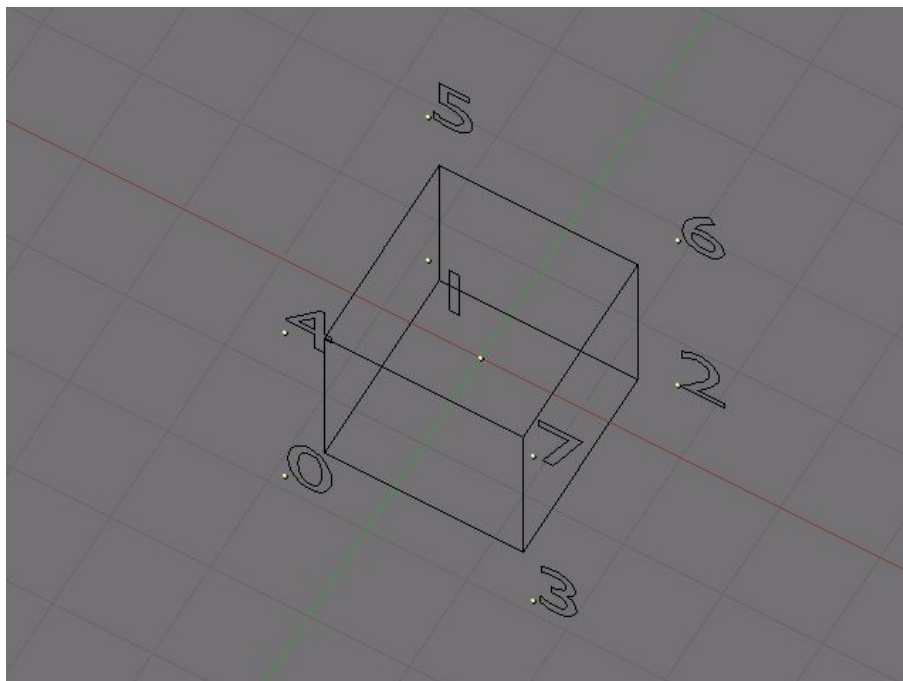


Figure 6: our cube is made of 8 points, numbered from 0 to 7

We can envision the creation of two vectors to summarize this information: `list_of_vertices` will contain the coordinates of all the points, and `list_of_faces` the constitution of the faces from the points:

```
# Definition of vertices
list_of_vertices=[
    [-1,-1,-1],          # vertex 0
    [-1,+1,-1],          # vertex 1
```

```

        [+1,+1,-1],          # vertex 2
        [+1,-1,-1],         # vertex 3
        [-1,-1,+1],         # vertex 4
        [-1,+1,+1],         # vertex 5
        [+1,+1,+1],         # vertex 6
        [+1,-1,+1]          # vertex 7
    ]
    # Definition of faces
    list_of_faces=[
        [0,1,2,3],           # face 0
        [4,5,6,7],           # face 1
        [0,4,7,3],           # face 2
        [1,2,6,5],           # face 3
        [0,1,5,4],           # face 4
        [3,7,6,2]            # face 5
    ]
]

```

As before, we are going to create a mesh, empty for the moment, in the memory of the computer. We will call it `CubeMeshData` (without any period or special characters) in order to signify that it is a case of a virtual mesh of a cube.

```
CubeMeshData=NMesh.GetRaw()
```

And now, we are going to put to work the magic of loops. For each vertex of the `list_of_vertices`, we are going to create a new *vertex* while using the vector (the coordinates) indicating the position of each current vertex. Once this is done, we can add this *vertex* to the list of *vertices* of the mesh.

```

for composante in list_of_vertices:
    vertex=NMesh.Vert(composante[0], composante[1], composante[2])
    CubeMeshData.verts.append(vertex)

```

We are also going to put in place a loop for generating the faces. The first loop allows us to create a face, for each enumerated face in the `list_of_faces`; each face will remain in memory, as an empty piece of data, until we add to it (`append`) some *vertices*. This is exactly the role of the second loop, included in the first: it will, for each *vertex* making up the current face, add (`append`) the *vertex* to the internal list of the current face. We then quit the internal loop in order to return to the principal loop, where we add (`append`) the newly described face in the internal list of the faces that make up the mesh.

```

for face_current in list_of_faces:
    face=NMesh.Face()
    for number_vertex in face_current:
        face.append(CubeMeshData.verts[number_vertex])
    CubeMeshData.faces.append(face)

```

Pretty simple, eh? But in fact, it is still just a case, for the moment, of only a small object with a ridiculously small number of vertices and faces. Now that our mesh is perfectly determined, we only need to transmit the information to Blender, so that it can be used. That is the goal of the following line: to give Blender a handle to the mesh `CubeMeshData` under the name of `Cube`. In addition, the last argument `simple` indicates that Blender must recalculate the normals of the mesh. This operation is optional, but we suggest the habit of including it! That will prevent anything untoward happening, while executing the script. I wouldn't say that we could manipulate millions of *vertices*, but for eight that won't handicap us too much!

```
NMesh.PutRaw(CubeMeshData, 'Cube', 1)
```

3.4 Definition of a material

For the determination of a material, we are going to take a look at the menu **Shading** (function key F5), and the tab **Material** of the *Material buttons*, in order to see the values that we will be able to observe there for the creation of the default material. In effect, we will try to reproduce here the principal values (most of these values are automatically initialized to their default value; it is theoretically possible to define others, but we will make them the same, for pedagogical reasons) in order to attribute them to our cube.



Figure 7: the default material properties of a cube

Elementary syntax of a material

```
[name.data] = Material.New('MA:Blender')
[name.object.receiving.the.material].materials.append([name.data])
[name.object].link([name.data])
[name.scene.current].link([name.object])
```

The creation of a material, for our mesh, is a relatively simple operation. It is sufficient, in the beginning, to create in memory, for example an object under the name of `mat`, then utilize the following line in order to render it accessible to Blender under the name of `'Material'` (one can see the name of the default material by looking at the field `'MA: '` under the tab **Material**, shown in Figure 7).

```
# Definition of a material
mat = Material.New('Material')
```

If the material is available, it is not used for the moment by any mesh or object. We are going to be able to add (`append`) the material `'mat'` to the list of mesh materials `'CubeMeshData'` (note that this is limited to accepting a maximum of 16 characters).

```
CubeMeshData.materials.append(mat)
```

We are now going to define certain properties of the material, beginning with the components of the colour (R, G and B are all equal to 0.756 in order to obtain a light grey), the Alpha of the material (A 1.0 in order to obtain a totally opaque cube), the Reflection of light by the material (Ref 0.8), its Specularity (Spec 0.5 for medium intensity specular reflections) and finally Hardness (the hardness of specular reflections, 50 corresponding to relatively soft highlights).

```
mat.rgbCol = [0.756, 0.756, 0.756]
mat.setAlpha(1.0)
mat.setRef(0.8)
mat.setSpec(0.5)
mat.setHardness(50)
```

And evidently, there remain a great number of parameters that one is able to specify in order to control, as accurately as one possibly wishes, the shade of the material. Having fulfilled our first objective (the emulation of the default material of Blender), we will stop here.

4. Adding a camera

Now we are going to see that adding a camera is a lot more simple than the creation of a mesh, but despite some particularities, it is only slightly more complex than the definition of a material. Despite this, in order to arrive at our goal, it is necessary to distinguish between the data of the object « material », and again, in each case, differentiate the name of the object in memory used by python, and that which is accessible under Blender.

Elementary syntax of the camera:

```
[name.data] = Camera.New('[type]','[CA:Blender]')
Definition of the current scene: [name.scene.current] = Scene.getCurrent()
[name.object] = Object.New('[OB:Blender]')
[name.object].link([name.data])
[name.scene.current].link([name.object])
[name.scene.current].setCurrentCamera([name.object])
```

Some options:

`['type']` may be `'persp'` or `'ortho'`

`[name.data].lens = [value]` allow setting the camera lens size (by default, 35.00)

`[name.data].clipStart = [value]` allow setting the value indicating the field of view of the camera (by default, 0.10)

`[name.data].clipEnd = [value]` allow setting the value indicating the limit of the field of view (by default, 100.00)

We are going to start by creating an object in memory and giving it a name: it will simply be the letter `'c'`, for *camera*. In the internals of Blender, the data file will carry the name `'Camera'`, that will be easy to verify in the menu **Editing** (function key F9), in the field `'CA: '`. Finally, we want a classic perspective view, so we declare the type `'persp'`:

```
c = Camera.New('persp','Camera')
```

Following this, we are going to define the value of the lens as being equal to 35 (although we would be able to leave this out, in that it is the case of being the default value):

```
c.lens = 35.0
```

We then define the name of the data file of the scene in python: `'cur'` for *current*, and we obtain the current scene in order to create the new object:

```
cur = Scene.getCurrent()
```

We have now physically created the camera object in Blender. Under python, it will carry the simple name of `'ob'` for *object*, and `'Camera'` under Blender. In the menu **Editing** (function key F9), the name of the camera will then appear in the field `'OB: '`. In python, that will become:

```
ob = Object.New('Camera')
```

From now on the object exists in Blender, but it is an empty object, just until we link a data file, in this instance `'CA:Camera'`. Under python, it is then simply `'c'`:

```
ob.link(c)
```

And then in its turn, the object (`'ob'` under python) must be linked to the current scene (`'cur'` under python) in order to be taken into account:

```
cur.link(ob)
```

Finally, this option is specific to the camera declaration of a camera in a python script: the camera has been declared as data, then as an object, and finally declared in a scene. Unfortunately, a scene may contain an unlimited number of cameras, and it is necessary to define the active camera. That is simply accomplished by the following line, where we define the object `'ob'` as being the active camera in the current scene `'cur'`:

```
cur.setCurrentCamera(ob)
```

And voilà, we have a functional and active camera! Now all we have to do is define the position, rotation and scale. It is a case of three current transformations, applicable to the object `'ob'` that we have just defined. In the present case, we will only be interested in the position and rotation.

Let's take another look at the default scene in Blender. In the 3D view, we select the camera with a right click of the mouse, then press the N key. A floating window, entitled **Transform properties** (transformation properties) makes its appearance. Notably, one can read there the name `'OB: '` of the object, and also the properties of the position, of rotation and of scale.

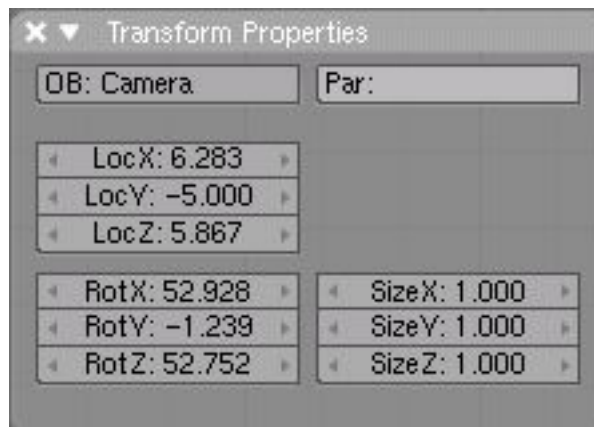


Figure 8: The transformation properties of the camera in the default scene

Elementary syntax of transformations:

```
[name.object].setEuler(angX, angY, angZ)
[name.object].setLocation(posX, posY, posZ)
[name.object].setSize(sizX, sizY, sizZ)
```

`setEuler` allows determining the angles of rotation of the object on its local axis (around its own pivot point). `setLocation` allows defining its position on the global axis (the absolute origin of the scene). Finally, `setSize` allows defining the proportions of the object in the three directions of its local axis. Please note that `angX`, `angY` and `angZ` must be angles expressed in radians.

The name of the camera object, in python, is quite simply 'ob', as we have previously defined. In order to reproduce the data of the floating panel **Transform properties**, we must then write a line of the type:

```
ob.setLocation(6.283, -5.000, 5.867)
```

The idea is the same for defining the rotations of the object. Unfortunately, the angle, in the internals of python, must be expressed in radians, although it is expressed in degrees in Blender. However, that just requires that we convert degrees to radians. If you don't know, 360 degrees equals 2π radians ($\pi = 3.14159\dots$). giving the result that 1 degree = $(2\pi/360)$. The angles to apply then, are:

```
RotX = 52.928*2π/360
RotY = -1.239*2π/360
RotZ = 52.752*2π/360
```

In python, π is written as `math.pi`, and it requires the importation of a supplementary module: `math`. Being lazy and in order to save our tired fingers from intensive use of the keyboard, we define:

```
conv = 2*math.pi/360
```

which will allow us to write the following conversion line:

```
ob.setEuler(52.928*conv, -1.239*conv, 52.752*conv)
```

The code relative to the creation of the camera can then be summarized as the following lines:

```
c = Camera.New('persp','Camera')
c.lens = 35.0
cur = Scene.getCurrent()
ob = Object.New('Camera')
ob.link(c)
cur.link(ob)
cur.setCurrentCamera(ob)
ob.setEuler(52.928*conv, -1.239*conv, 52.752*conv)
ob.setLocation(6.283, -5.000, 5.867)
```

and also that the first line of the script must be:

```
import Blender, math
```

5. Adding a lamp

If you have well understood adding a camera via a python script, then adding a lamp will not pose a problem for you. The only complexity may come from the diversity of lamps which exist, each having different options. We will not try to be exhaustive here, we will just be demonstrative of the means and methods.

Elementary syntax of a lamp

```
[name.data] = Lamp.New('[type]', 'LA:Blender')
[name.object] = Object.New('OB:Blender')
[name.object].link([name.data])
[name.scene.current].link([name.object])
```

Some options:

['type'] may be 'Lamp', 'Sun', 'Spot', 'Hemi', 'Area', or 'Photon'

If you choose the 'type' 'Spot', you will be able to set additional options. For example, the line:

```
l.setMode('square', 'shadow')
```

specifies a projector of a square shape ('square') with the option of generation of shadows ('shadow') activated.

As previously seen, we are going to start by naming the object under python, this way creating it, although empty, in memory; we simply choose 'l' for *lamp*. In the internals of Blender, the data file will carry the name 'Lamp', which will be easy to verify in the menu **Editing** (function key F9), in the field 'LA:':

```
l = Lamp.New('Lamp', 'Lamp')
```

The 'physical' creation of the lamp object is trivial. Under python, it will simply have the name 'ob' for *object*, and 'Lampe' under Blender. In the menu **Editing** (function key F9), the name of the lamp will then appear in the field 'OB:'. In python, that translates to:

```
ob = Object.New('Lamp')
```

The object 'ob' from that point on exists under Blender, but as we saw previously, it is a case of an empty object, just until we link it to a data file, in the occurrence of 'LA:Lamp'. Under python, it is then a case of 'l':

```
ob.link(l)
```

In its turn, the object 'ob' must now be linked to the current scene ('cur' under python) in order to be taken into account. The current scene having been previously defined, we can this time simply use:

```
cur.link(ob)
```

Our lamp is ready, and all that remains is that we conveniently place it in the scene. Returning to the default scene of Blender, We now select the lamp with a right click of the mouse. Normally, the panel **Transform properties** is always open; if this is not the case, press again the key N. One then finds the following information for the lamp:

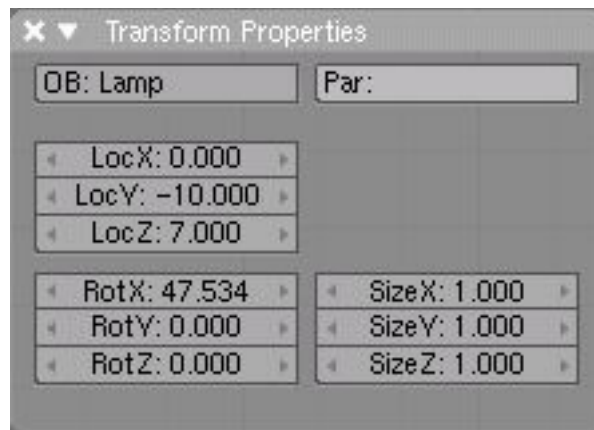


Figure 9: the transformation properties of the lamp in the default scene

You will note a surprising detail: the default lamp is simply an immaterial point in space which serves as the origin of the light. It is then not rigorously required to attribute the properties of rotation to it. However, this has been done in the default scene! There is nothing exceptional to note, just that if you transform the lamp into a luminous spotlight, the cone will already be oriented with a default angle sufficient to illuminate the cube without additional manipulation on your part.

Even if we are able to bypass this exercise, we are going to scrupulously respect the attached values in the panel **Transform properties**.

As previously, the name of the lamp object, under python, is simply 'ob'. We are first going to try to reproduce the components of the specific location of the panel. We are then going to write a line of the type:

```
ob.setLocation(0.000, -10.000, 7.000)
```

The definition of the rotation of the lamp is equally very simple, now that we have take the step:

```
ob.setEuler(47.534*conv, 0.000, 0.000)
```

Summarizing the block of code that corresponds to the creation of the lamp:

```
l = Lamp.New('Lamp','Lamp')
ob = Object.New('Lamp')
ob.link(l)
cur.link(ob)
ob.setLocation(0, -10, 7)
ob.setEuler(47.534*conv,0,0)
```

6. Review of the code

We will review here the complete code that we have seen just up to the present, in a synthetic fashion and while numbering the lines. You will also find it on the cdrom accompanying the magazine or at <http://www.linuxgraphic.org>, under the name `blender-default-scene.py` and `blender-default-scene.blend`.

```
01: import Blender, math
02: from Blender import Camera, Object, Scene, Lamp, NMesh, Material
03:
04: conv = 2*math.pi/360
05:
06: # Définition of vertices
07: list_of_vertices=[
08:     [-1,-1,-1],          # vertex 0
09:     [-1,+1,-1],          # vertex 1
10:     [+1,+1,-1],          # vertex 2
11:     [+1,-1,-1],          # vertex 3
12:     [-1,-1,+1],          # vertex 4
13:     [-1,+1,+1],          # vertex 5
14:     [+1,+1,+1],          # vertex 6
15:     [+1,-1,+1],          # vertex 7
16: ]
17:
```

```

18: # Definition of faces
19: list_of_faces=[
20:     [0,1,2,3],          # face 0
21:     [4,5,6,7],          # face 1
22:     [0,4,7,3],          # face 2
23:     [1,2,6,5],          # face 3
24:     [0,1,5,4],          # face 4
25:     [3,7,6,2]           # face 5
26: ]
27:
28:
29: # Definition of a cube
30: CubeMeshData=NMesh.GetRaw()
31:
32: ## Definition of a material
33: mat = Material.New('Material')
34: CubeMeshData.materials.append(mat)
35: print mat.rgbCol
36: mat.rgbCol = [0.756, 0.756, 0.756]
37: mat.setAlpha(1.0)
38: mat.setRef(0.8)
39: mat.setSpec(0.5)
40: mat.setHardness(50)
41:
42: for component in list_of_vertices:
43:     vertex=NMesh.Vert(component[0], component[1], component[2])
44:     CubeMeshData.verts.append(vertex)
45:
46: for face_current in list_of_faces:
47:     face=NMesh.Face()
48:     for number_vertex in face_current:
49:         face.append(CubeMeshData.verts[number_vertex])
50:     CubeMeshData.faces.append(face)
51:
52: NMesh.PutRaw(CubeMeshData, 'Cube', 1)
53:
54: # Definition of the camera
55: c = Camera.New('persp', 'Camera')
56: c.lens = 35.0
57: cur = Scene.getCurrent()
58: ob = Object.New('Camera')
59: ob.link(c)
60: cur.link(ob)
61: cur.setCurrentCamera(ob)
62: ob.setEuler(52.928*conv, -1.239*conv, 52.752*conv)
63: ob.setLocation(6.283, -5.000, 5.867)
64:
65: # Definition of the lamp
66:
67: l = Lamp.New('Lamp', 'Lamp')
68: ob = Object.New('Lamp')
69: ob.link(l)
70: cur.link(ob)
71: ob.setLocation(0, -10, 7)
72: ob.setEuler(47.534*conv, 0, 0)
73:
74: Blender.Redraw()

```

7. Conclusions

Voila, we have arrived at the end of this first article. We have seen some things less evolved than those proposed in the article by Yves Bailly, but we have also had the occasion to go into a little more depth about certain ideas, and to examine the elementary syntax needed in order to obtain certain types of objects in Blender. It seems evident, that if you have followed the series of articles about POV-ray, that programming Python in Blender is a little more difficult, because it must call internal functions of Blender, get the results and then return them to Blender. That does not lighten the programs, nor improve the fundamental way of treating the data. In review,

the advantages are significant because it allows developing software that is very rich in possibilities.

Finally, as we have been able to perceive, the creation of a mesh is a complex operation, and in a future article, we will spend a little more time there. The control of face and other things is not that difficult, and the creation of complex objects can become quite rapid.

I will leave you to discover the links which follow. They will conduct you towards diverse resources about programming Python in Blender. I recommend in particular the excellent site of JM Soler, considered as being one of the major references for python scripting for Blender, as useful to anglophone users as it is to francophone users. His site in particular presents some tutorials on the creation of complex meshes; we will also have the occasion to return to this subject.

With Python, it is a new world which is open to us, and it is far from being explored.

Links

Blender development site: <http://www.blender.org>

The official Blender site: <http://www.blender3d.org>

The official Python documentation for Blender:

<http://www.blender.org/modules/documentation/236PythonDoc/index.html>

Python documentation: <http://www.python.org/doc/2.3.5/lib/lib.html>

The site of JM Soler: <http://jmsoler.free.fr/didacticiel/blender/tutor/index.htm>