Creation of a GUI (Graphical User Interface) for Blender for your python scripts

Thank you to Diamond Editions for the authorization to place this article online, initially published in Linux Magazine N°74

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

We have seen in the preceding article how to reconstruct the opening default scene of Blender. In order to do this, we have learned how to create in memory some data structures, describing the entities that we want, and how to « import » them into Blender, in order to give them substance. That has been a series of exercises of which the complexity went from simple to moderate, but we have only seen one method of using the script: the combination [ALT]+[P]. However, now we are going to see that we can create a graphical interface to determine, interactively, the functioning of our script.

If one takes another look at the code that we made in the previous issue, we are going to try to give it a graphical interface that will allow us to determine the size of the cube, to specify the location of the geometric center, to give the material a personalized colour, and finally to display the generated object.

![Figure 1: the final result of our efforts](image)

1. Why make a graphical interface?

Certainly, we can envision the existence of a python script which only knows how to do one thing, but does that well; such is the goal of many computer science tools. But we also can envision other forms of scripts, less monolithic, that are able to adapt to our needs. In the first case, we are content to launch the python script and to observe the results. In the second case, we define certain variables (in the form of buttons, sliders, and editable fields, for example) which, according to their value, modify the result (and perhaps even the entire result) of the python script.
In fact, a graphical interface is a sort of loop without end that watches the interventions of the user, and reacts to input. Well conceived, thought out with the ergonomics and spirit of thought of the user, a graphical interface may be a precious tool in mastering a potentially complex script. And that is the aim of computer science: to render simple some complex or very repetitive tasks.

Loops without end (or infinite loops) that we are now going to discover will allow us to watch for the appearance of two types of events: those linked to the usage of clicks of the mouse or keys on the keyboard by the user, and those linked to the use of buttons of the graphical interface that will be programmed. The title of this article could appear mistaken in that we are going to study these two cases.

As a reference, we highly recommend the following script, of which the study will help you to rapidly comprehend the functioning and use of buttons in Blender:

http://infohost.nmt.edu/~jberg/blender/button_demo

1.1 The events linked to the use of entry tools

We are understood by the computer by use of the keyboard or the buttons of the mouse. In particular, we retain the possibility of registering actions using the following keys of the keyboard: A to Z, the events using the names AKEY to ZKEY, the Enter key (RETKKEY), the Escape key (ESCKEY), the function keys F1 to F12 (FKKEY to F12KEY), the Tab key (TABKEY) and the keys of the numeric keypad 0 to 9 (PAD0 to PAD9). Using the mouse, we are able to note the left mouse button (LEFTMOUSE), middle (MIDDLEMOUSE) or right (RIGHTMOUSE), and the use of the wheel button turning up (WHEELUPMOUSE) or turning down (WHEELDOWNMOUSE). Certainly, the movements of the mouse may also be captured thanks to the events MOUSEX and MOUSEY.

1.2 The events linked to the use of the graphical interface

We have at our disposition different types of buttons, each dedicated to a particular use. The illustration which follows illustrates all these possibilities. It is more or less the interface that is launched by the script button_demo.py which we have recommended by the link given above.

![Illustration of buttons in action](image)

**Figure 2:** the principal buttons of Blender in action

**Push button:** it is the first button entitled Push. Its use is simple: while clicking on it, you launch an action, most often it is the execution of a script, or something outside of the graphical interface, for example. It does not allow a general state, it is only active at the time of a click.

**Menu button:** this is the button entitled Number x Slider in the preceeding illustration. While clicking on it, a rolldown menu appears, offering several possible actions. The number of choices is determined by your script.

**Number button:** it appears with the text Number 3 above. Ideal for permitting the user to
choose between different whole numbers which can be used by the script. It is possible to set the default size.

**Slider button:** this is the slider entitled *Slider: 0.500*. The general idea is similar to the number button, except that this one allows fine adjustment of a decimal number. Additionally, the affixed number can be reutilized in the script, and the button has two limits, an upper and a lower.

**String button:** this is the button *String: Hello world!* of Figure 2. It is simply a field where it is possible to capture a text string (of length determined by the script) which can be ultimately used, generally for naming entities.

**Toggle button:** this is the last button, entitled *Toggle*. It has the particularity of allowing two states (active or inactive) and often serves to activate or deactivate some options during the execution of your script, thus realizing two binary choices, often more synthetic than the menu button.

We will now ultimately see how to code most of these buttons.

### 2 Where to begin? Some basics...

As we have underlined during the first part, an interface is a series of loops without end, which watches for certain events or controls the items of the interface itself. It all begins with the import of two modules of the python API of Blender: *Draw* and *BGL*.

**Draw** is the module which is going to allow access to the window interface of Blender and thus the surveillance of the events. For its part, **BGL** is a OpenGL wrapper, which makes available the constants and functions of OpenGL for Blender, which will allow among other things, placing text, colouring pixels, attaching images and these types of things.

<table>
<thead>
<tr>
<th>Elementary Syntax of the interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: import Blender</td>
</tr>
<tr>
<td>02: from Blender import Draw, BGL</td>
</tr>
<tr>
<td>03: def event(evt, val):</td>
</tr>
<tr>
<td>04:     pass</td>
</tr>
<tr>
<td>05: def button_event(evt):</td>
</tr>
<tr>
<td>06:     pass</td>
</tr>
<tr>
<td>07: def draw_gui():</td>
</tr>
<tr>
<td>08:     pass</td>
</tr>
<tr>
<td>09: Draw.register(draw_gui, event, button_event)</td>
</tr>
</tbody>
</table>

Now we define three functions, that we name in an explicit way for us. The first, *event()* which will permit us to watch for events issued by the action of the user on the keyboard or the mouse. The second, *button_event()* is going to allow us to manage the running of the script as a function of the action of the user on the graphical interface buttons. The third and last, *draw_gui()* allows us to design on the screen the buttons and text of the interface and to associate them with the variables of your choice. The programming of the interface concludes by *Draw.Register()* which allows registration and activation of the loops which are passed to them in arguments.

You will note that the definition of a function is of the form:

```
   def [name function]():
```

and that the indentation of the following lines allows us to delimit the code concerning the function. The command *pass*, here, is not used and serves only to produce a code which, if it is correct from the point of view of the syntax, does absolutely nothing.

And voilà! We now have the skeleton of our script and of our graphical interface, now all that remains it to garnish it copiously!

### 2.1 The functions

Certainly, we have two options: put in place the code to execute in response to an event in the corresponding loop, or simply specify a function that the loop will call. This last method has the advantage of allowing us to define your functions in the « upstream » of your program, and to
freely reuse them « downstream », simply by calling the function by its name. To each his own programming habits, but in the cadre of complex programs for which you wish to have the most clearly possible execution (for example in order to conform to a synoptique or a flowchart), functioning of this sort can rapidly become an asset.

Without entering into the detail of the programming of the function, it is sufficient to retain the following elementary syntax:

```
Elementary Syntax of a function:
def [name of the function](variable1, variable2..., variablen)
               ...
#code which must execute the function
return #or return(var1, var2..., varn)
```

The name of the function must be chosen in an explicit way, such that the variables between parentheses are those which the function will apply; it is imperative that, in one function or another, that they are defined somewhere in the program. If the function does not need to return anything, the final command `return` is optional. If it must return something (such as the result of a calculation), the command `return()` becomes obligatory if one wishes to reuse the result of the function somewhere else in the program.

For example the program which follows takes two pairs of variables (var1 and var2, and also cvar3 and var4) defined from lines 03 to 06, and passes them to the function `multiplication()`. This then effects the operations that one expects, and returns the product in the variable `result`, in line 11, that the program prints in the console thanks to the command `print`. In order to be able to use this outside of the function, it will be necessary to define, as mentioned, the variable `result` as being global, as shown in line 09.

```
01: import Blender
02:
03: var1 = 2
04: var2 = 1
05: var3 = 2
06: var4 = 2
07:
08: def multiplication(variable1, variable2):
09:   global result
10:   result = variable1 * variable2
11:   return(result)
12:
13: multiplication(var1, var2)
14:
15: print result
16:
17: multiplication(var3, var4)
18:
19: print result
```

While looking more closely at the function, one realizes that it is a simple multiplication, the two variables are defined at the beginning, being multiplied and returned on exiting from the function in the form of result. In the example, we used, consecutively, two times the function `multiplication()` and we attached the result to the console. In the first case, on line 13, we pass to the function `var1` and `var2`. In the second, on line 17, we pass the variables `var3` and `var4`. All we need to do is launch the script ([ALT]+[P]) and observe the results in the console; attached respectively by lines 15 and 19:

```
2
4
```

2.2 The heart of the cube generator

We are going to keep things simple, and reuse in extenso the code from the preceeding article for the generation of the lamp and the camera. You will find it on the cdrom accompanying the magazine of the preceeding issue, or online at [http://www.linuxgraphic.org](http://www.linuxgraphic.org), under the name `blender-default-scene.py` and `blender-default-scene.blend`.

Like all good python scripts, it begins with the importation of the module Blender and the
submodules necessary for running the script. In this instance, we are going to need Camera, Object, Lamp, NMesh and Material:

```python
import Blender, math
from Blender import Camera, Object, Scene, Lamp, NMesh, Material
```

We are just add the pieces of code which interest us in the appropriate functions, which do not require any variable on entry, and which do not return any value on exiting. Thus, for the camera, we obtain the following code:

#### Definition of the camera

```python
def MakeCamera():
    c = Camera.New('persp','Camera')
    c.lens = 35.0
    cam = Object.New('Camera')
    cam.link(c)
    cur.link(cam)
    cur.setCurrentCamera(cam)
    cam.setEuler(52.928*conv, -1.239*conv, 52.752*conv)
    cam.setLocation(6.283, -5.000, 5.867)
```

Likewise, for the lamp, we have this:

#### Definition of the lamp

```python
def MakeLamp():
    l = Lamp.New('Lamp','Lamp')
    lam = Object.New('Lamp')
    lam.link(l)
    lam.setEuler(47.534*conv,0,0)
    lam.setLocation(0, -10, 7)
    cur.link(lam)
```

These two functions will simply and ultimately be called by the following lines:

```python
MakeCamera()
MakeLamp()
```

but it is necessary to take into account importing the current scene before defining these functions:

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

and also to define the variable `conv` which assures the conversion of radians from Python into the degrees of Blender:

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

This copying does not appear too scary, and we can do the same with the code for the generation of a cube. However, we want to be able to personalize this by means of the graphical interface of our script. As mentioned in the introduction of this article, we wish in particular to determine the length of the sides of the cube, specify the location of the geometric center, give the material a personalized colour, and finally baptize the cube with a name of our choice.

We can then adjust and have already made some choices about the future appearance of our interface. With a little foresight, we reveal the look of our future interface in order to better obtain the parameters that we are going to describe and visualize the associated buttons.
● Name of the cube, determined by a text button, associated variable: stringName.val

● Length of the side, regulated by a slider button, associated parameter: sliderEdge.val

● X coordinate of the center of the cube, regulated by a decimal button, associated variable: numberCentreX.val

● Y coordinate of the center of the cube, regulated by a decimal button, associated variable: numberCentreY.val

● Z coordinate of the center of the cube, regulated by a decimal button, associated variable: numberCentreZ.val

● R component (red) of the colour of the cube, regulated by a slider button, associated variable: sliderR.val

● G component (green) of the colour of the cube, regulated by a slider button, associated variable: sliderG.val

● B component (blue) of the colour of the cube, regulated by a slider button, associated variable: sliderB.val

The variables associated with the buttons reveal their great importance, because we can massively reuse them in the code of the generation of our cube, where we have replaced all of the fixed values that our interface proposes to make them variable. In particular, sliderEdge.val, numberCentreX.val, numberCentreY.val and numberCentreZ.val, which will advantageously replace some variables making them easier to manipulate:

```python
var1 = numberCentreX.val
var2 = numberCentreY.val
var3 = numberCentreZ.val
var4 = sliderEdge.val
```

In the program that we have elaborated in the preceeding article (online at [http://www.linuxgraphic.org](http://www.linuxgraphic.org)), the vertices of our cube were described by the following components:

```python
list_of_vertices=[
    [-1,-1,-1],
    [-1,+1,-1],
    [+1,+1,-1],
    [+1,-1,-1],
```
They now become:

```python
# Definition of control points:
list_of_vertices=
    [-var4+var1,-var4+var2,-var4+var3],
    [-var4+var1,+var4+var2,-var4+var3],
    [+var4+var1,+var4+var2,-var4+var3],
    [+var4+var1,-var4+var2,-var4+var3],
    [-var4+var1,-var4+var2,+var4+var3],
    [-var4+var1,+var4+var2,+var4+var3],
    [+var4+var1,+var4+var2,+var4+var3],
    [+var4+var1,-var4+var2,+var4+var3]
```

Sceptical? There is a very simple way to verify the validity of these new vertices, by substituting the variables with the fixed values of the original program. Thus, supposing a cube of size 1 unit (sliderEdge = var = 4) and origin centered on [0,0,0] (numberCentreX = var1 = 0, numberCentreY = var2 = 0, and numberCentreZ = var3 = 0), we obtain then, for the first line of vertices [-1,-1,-1].

In the same way, the material is only defined as an invariable grey colour, but by the action of the user on three slider buttons which will allow giving each component R, G and B some different values: sliderR.val, sliderG.val et sliderB.val. We then obtain the declaration of the colour for the current material:

```python
mat.rgbCol = [sliderR.val, sliderG.val, sliderB.val]
```

Finally, we have equally attributed to the interface a text button, so that the name of the cube can be freely determined by the script user. The name is take from stringName.val.

Note that MakeLamp() and MakeCamera(), these functions do not need to take any action on entry, and do not return anything either, being content to execute these actions. No line return() concludes these functions, and the function simply presents in the form of:

```python
def MakeCube():
    ...
```

All that remains is that we take the old code for generation of a cube, while respecting the arrangements and substitutions that have been described until now. We then very easily obtain the following final function:

```python
# Definition of a cube:
def MakeCube():
    var1 = numberCentreX.val
    var2 = numberCentreY.val
    var3 = numberCentreZ.val
    var4 = sliderEdge.val
    # Definition of control points:
    list_of_vertices=
        [-var4+var1,-var4+var2,-var4+var3],
        [-var4+var1,+var4+var2,-var4+var3],
        [+var4+var1,+var4+var2,-var4+var3],
        [+var4+var1,-var4+var2,-var4+var3],
        [-var4+var1,-var4+var2,+var4+var3],
        [-var4+var1,+var4+var2,+var4+var3],
        [+var4+var1,+var4+var2,+var4+var3],
        [+var4+var1,-var4+var2,+var4+var3]

    # Definition of faces:
    list_of_faces=[
        [0,1,2,3],
        [4,5,6,7],
        [0,4,7,3],
        [1,2,6,5],
        [0,1,5,4],
        [3,7,6,2]
    ]
```
CubeMeshData=NMesh.GetRaw()

# Definition of material:
mat = Material.New('Material')
CubeMeshData.materials.append(mat)
mat.rgbCol = [sliderR.val, sliderG.val, sliderB.val]
mat.setAlpha(1.0)
mat.setRef(0.8)
mat.setSpec(0.5)
mat.setHardness(50)
for component in list_of_vertices:
    vertex=NMesh.Vert(component[0], component[1], component[2])
    CubeMeshData.verts.append(vertex)
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)
NMesh.PutRaw(CubeMeshData,stringName.val,1)

This function will be called by the script by means of the simple line:

MakeCube()

Now all that remains for us to do is the most fun part, the one to which this article is most specifically consecrated: the creation of a graphical interface.

3 The graphical interface (or the costume of our script)

It is time that we approach the heart of the subject of this article, but you should note that this part is simple, in truth, from the moment that you begin to be ordered and methodical in approach. The creation of a graphical interface passes through several indispensable stages, that we are going to describe and comment.

3.1 To define the imported modules

It is essential to indicate to python which submodules of the module Blender to use during the course of execution. This happens generally by the establishment, at the head of the program, the following two lines:

import Blender
from Blender import ...

where one indicates the submodules that interest us, for example Camera, Object, Scene, Lamp, NMesh, and Material in the program of the previous issue. But the establishment of a graphical interface calls two supplementary submodules: Draw and BGL. The first is relative to support in python a graphical interface integrated in Blender; it furnishes the management of events, the design of buttons on the screen and some other small refinements. The second is relative to the use of interface elements of openGL, such as the placement of text, the colouring or the design of 2D elements and other numerous possibilities.

Our first two lines then, are:

**Elementary syntax**

import Blender
from Blender import ..., Draw, BGL

This then, in the particular case of our program, transforms into the following two lines:

import Blender, math
from Blender import Camera, Object, Scene, Lamp, NMesh, Material, Draw, BGL

3.2 Definition of default values for the buttons

Obviously, the buttons appear on the screen taking default values that we are free to determine.
If we look again at Figure 3, which presented the interface as we have imagined it, we see that each button has a label and a value, according to the situation. We have already defined, from top to bottom and from left to right, the associated variables for each button:

- `StringRef.val`
- `sliderEdge.val`
- `numberCentreX.val`
- `numberCentreY.val`
- `numberCentreZ.val`
- `sliderR.val`
- `sliderG.val`
- `sliderB.val`

and now all that remains is to attribute to them their default value. This is done thanks to the command `Draw.Create()`:

**Elementary syntax:**

```
[name of the button] = Draw.Create([initial value])
```

Pay attention to using the name of the button and not the name of its variable. For example:

```
sliderEdge = Draw.Create(1.00)
```

The values are read directly on the button of Figure 3; the numerical values are taken directly from between parentheses, the character strings are also between quotation marks. For our small program, we then have the following declarations:

```
# Initial values of buttons:
StringRef = Draw.Create("Cube")
sliderEdge = Draw.Create(1.00)
numberCentreX = Draw.Create(0.00)
numberCentreY = Draw.Create(0.00)
numberCentreZ = Draw.Create(0.00)
sliderR = Draw.Create(0.50)
sliderG = Draw.Create(0.50)
sliderB = Draw.Create(0.50)
```

This has nothing to do with the creation of the interface, but it is a question of a small trick to lighten the work of the composition of the interfaces, because it is easy to mix up the lines or values of the numerous parameters of the buttons! For example, in the following, we will not describe the position of each line by means of its value on the screen, but by a pre-established variable. Thus, we name the lines as shown in the following figure:
and we define the height of each line as being equal to 25 and each space between lines as being equal to 5, while defining our first line (line[0]) as beginning at an altitude of 5 pixels from the base of the script window, we can define the following 10 lines:

```python
line = [None, None, None, None, None, None, None, None, None, None]
line[0] = 5
line[1] = 35
line[2] = 65
line[3] = 95
line[4] = 125
line[5] = 155
line[6] = 185
line[7] = 215
line[8] = 245
line[9] = 275
```

### 3.3 Assigning an event number to each button

There is an additional step using this form, because a fixed number may be assigned to each button during its definition. For example:

```python
draw.PushButton("OK", 1, 5, line[0], 60, 25, "Confirm")
```

(but we will come back to the creation of buttons later). We prefer, generally, to assign the name of a variable to an event number outside the definition of a button, and to assign this variable a value outside of the button assignment. For example:

```python
EV_BT_OK = 1
draw.PushButton("OK", EV_BT_OK, 5, line[0], 60, 25, "Confirm")
```

The two results are rigorously identical, except that in the second case, if you build your interface in this manner and reorganize it regularly, without knowing too much about where you are going, you do not risk mixing up things by attributing the same event number to two different buttons.

For example, we will define the following event numbers:

```python
EV_BT_OK = 1
EV_BT_CANCEL = 2
EV_SL_EDGELENGTH = 3
EV_NB_CENTERX = 4
EV_NB_CENTERY = 5
```
Note well that the variable names are left to your attention. I preferred to follow the convention of Yves Bailly during his presentation of python with Blender, in his article presented in GNU/L Mag #68 (or online at http://www.kafka-fr.net), while adapting it to my needs. This means $EV_-$ is used to indicate an event, then a two letter code to indicate the button type used by the event (BT for button, SL for slider, NB for number, ST for string, etc.) and finally a name which seems explicit (CENTERX for the center X coordinate, for example).

4 Definition of the interface drawn on the screen

It is going to be a question of a function, of the same title as MakeCube(), MakeLight() or MakeCamera() except that the call and execution are done very differently.

Elementary syntax:
```python
def draw_gui():
    ...
```
You may well give the function any name, here draw_gui(). Pay attention to the indentation which delimits the content of the function, and be rigorous about this.

4.1 Global definition of interface variables:

The goal of the game being to render the button values available to the rest of the python script, we are quite interested in declaring them in a global way rather than local. This is done simply by the command global, followed by the variables that we wish to render access possible.

```python
def draw_gui():
    global stringName, sliderEdge, numberCentreX, numberCentreY, numberCentreZ, sliderR, sliderG, sliderB
```

4.2 Attaching text

It is a case of a small practical function for garnishing the graphical interface with help instructions or tooltips, such as key shortcuts. To do this, we need to use two instructions. The first is issued from the submodule Bgl, and allows positioning the text as you choose:

Elementary syntax:
```python
BGL.glRasterPos2i(x,y)
x is the position of the distance from the left for placing the element, and y is the position in terms of height position of the element. Note that the placement is in reference to the lower left corner of the screen.
```

The second is issued from the module Draw and allows attaching a character string:

Elementary syntax:
```python
Draw.Text(“[your text]”)```
You can replace the brackets and their content with the text of your choice, as you want it to appear on the screen.
It is then very easy to attach the relevant text as shown in our lines 4, 7 and 9:

```python
def draw_gui():
    ...
    BGL.glRasterPos2i(5,line[9])
    Draw.Text("Choose the name of the cube:")
    BGL.glRasterPos2i(5,line[7])
    Draw.Text("Choose the geometry of the cube:")
    BGL.glRasterPos2i(5,line[4])
    ...
```

4.3 The push buttons

The push buttons, which only allow state, will simply serve us as launching actions. For example, **OK** will launch the script conforming the parameters that will be attached at that moment, and that **Quit** will exit from the graphical interface to return to the code of the script.

### Elementary syntax:

```python
Draw.PushButton("[name]", [event number], [position x], [position y],
[length], [height], "[hint]")
```

- **[name]**: corresponding to the character string which will appear on the button
- **[event number]**: this is the event number passed from the button when it is activated
- **[position x]**: corresponds to the x coordinate (in the sense of the width of the screen) from the bottom left of the button
- **[position y]**: corresponds to the y coordinate (in the sense of the height of the screen) from the bottom left of the button
- **[length]**: this is simply the width (length) of the button
- **[height]**: the height of the button
- **[hint]**: this text appears in an info bubble when the mouse cursor is hovering above the button; ignore this if you do not want the hints

Putting them in place is very easy, we just need to pay attention that the event numbers are replaced by the proper variables (EV_BT_OK and EV_BT_CANCEL) and that the buttons are of the correct dimension. In effect, the line making 25 pixels height and the interspace being of 5 pixels, we will choose to have some buttons of which the height does not exceed 25 pixels.

```python
def draw_gui():
    ...
    Draw.PushButton("OK", EV_BT_OK, 5, line[0], 60, 25, "Confirm")
    Draw.PushButton("Cancel", EV_BT_CANCEL, 80, line[0], 60, 25, "Cancel")
    ...
```

4.4 The slider buttons

The particularity of these buttons is to allow a minimal value, a maximum value, and to vary between these two limits by the action of the mouse by the user on the slider. The position on which one stops the slider will be interpreted as being the value assigned to the associated variable of this **Sliding button**.

### Elementary syntax:

```python
[button name] = Draw.Slider("[name]", [event number], [position x], [position y],
[width], [height], [initial value], [minimal value], [maximum value],
[real time function], "[hint]")
```

- **[name]**: corresponds to the character string that will be attached to the button
4.5 The number buttons

There is only a small fundamental difference between a **Number button** and a **Slider button** as we have just seen. In one case (slider), you must move the slider with the mouse in order to augment or diminish the value, in the second case (number) you use the small arrows on each side of the button to modify the value. In all cases, while clicking on the label (the text) of the button, you have the possibility of entering the exact value that you wish by way of the keyboard. The only true difference resides in the fact that the **Slider button** has a real time capacity that the **Number button** does not have.

**Elementary syntax:**

```python
[button name] = Draw.Number("[name]", [event number], [position x], [position y], [width], [height], [initial value], [minimum value], [maximum value], 
"[hint]"
)
```

- `[button name]`: corresponds to the character string which is attached on the button
- `[event number]`: this is the number passed to the button event when it is activated
- `[position x]`: corresponds to the x coordinate (in the sense of the width of the screen) from the bottom left of the button
- `[position y]`: corresponds to the y coordinate (in the sense of the height of the screen) from the bottom left of the button
- `[width]`: this is simply the width of the button...
- `[height]`: ...and of the height of the button
- `[initial value]`: the value that the button takes on when the script is launched. Preferably use a variable of which the name is derived from the name of the button, such as `[button name].val`
- `[minimal value]`: it is the value below which the slider button cannot go
- `[maximum value]`: likewise, but from the point of view of the maximum value
- `[real time function]`: if you specify a non-null value, the button will emit event signals in real time, that is to say that any change of value will be immediately taken into account by the script and eventually attached by Blender.
- `[hint]`: this is the info bubble text (tooltip) that appears when the mouse cursor hovers above the button; ignore this if you do not want the info bubble
screen) from the bottom left of the button

[position y]: corresponds to the y coordinate (in the sense of the height of the screen) from the bottom left of the button

[width]: this is simply the width of the button...

[height]: ... and the height of the button

[initial value]: the default value that the button has when the script is launched. Preferably use a variable of which the name is derived from the name of the button, such as [button name].val

[minimum value]: this is the value which the slider button cannot go below

[maximum value]: similarly, but from the point of view of the maximum value

[hint]: this is the info bubble text that appears when the mouse cursor hovers above the button; ignore this if you do not want the info bubble

If putting in place a Slider button does not pose any problem for you, then that of a Number button must also be easy, since the syntax is the same, with the exception of the parameter « real time ».

```python
def draw_gui():
    ...
    numberCentreX = Draw.Number("Centre X: ", EV_NB_CENTERX, 5, line[5], 100, 25, numberCentreX.val, -5.00, 5.00, "X coordinate of the Center")
    numberCentreY = Draw.Number("Centre Y: ", EV_NB_CENTERY, 110, line[5], 100, 25, numberCentreY.val, -5.00, 5.00, "Y coordinate of the Center")
    numberCentreZ = Draw.Number("Centre Z: ", EV_NB_CENTERZ, 215, line[5], 100, 25, numberCentreZ.val, -5.00, 5.00, "Z coordinate of the Center")
    ...
```

4.6 The Text button

Always the easiest, the Text button. It collects the character string under the variable name to put it at the disposition of the python script. Very often utilized for attaching some personalized messages, or for naming entities according to the wishes of the user.

**Elementary syntax:**

```python
[button name] = Draw.String("[name]", [event number], [position x], [position y], [width], [height], [initial value], [maximum length of the string], "[hint]"
```

[name]: corresponds to the character string which is attached on the button

[event number]: this is the number passed to the button event when it is activated

[position x]: corresponds to the x coordinate (in the sense of the width of the screen) from the bottom left of the button

[position y]: corresponds to the y coordinate (in the sense of the height of the screen) from the bottom left of the button

[width]: this is simply the width of the button...

[height]: ... and the height of the button

[initial value]: the default value that the button has when the script is launched. Preferably use a variable of which the name is derived from the name of the button, such as [button name].val

[maximum length of the string]: this value defines the maximum number of characters that the user may employ in the definition of the character string
This type of button, the last that we will study today, does not present any particular difficulties.

```python
def draw_gui():
    ...
    stringName = Draw.String("Name: ", EV_ST_NAME, 5, line[8], 310, 25,
    stringName.val, 32, "Name of the object")
    ...
```

4.7 The final appearance

The figure which follows presents then the graphical interface that we have determined, while trying to consider ergonomics and having a rational layout. Certain scripts can be very complex and necessitate a graphical interface. It is necessary to never forget that the objective is that any user can use your script, rather than attach an important collection of buttons and parameters with esoteric names that will always impress the masses.

![Figure 5: our so pretty interface!](image)

The function defining the attachment of the interface is given below:

```python
def draw_gui():
    global stringName, sliderEdge, numberCentreX, numberCentreY, numberCentreZ,
    sliderR, sliderG, sliderB
    BGL.glRasterPos2i(5,line[9])
    Draw.Text("Choose the name of the cube:")
    stringName = Draw.String("Name: ", EV_ST_NAME, 5, line[8], 310, 25,
    stringName.val, 32, "Name of the object")
    BGL.glRasterPos2i(5,line[7])
    Draw.Text("Choose the geometry of the cube:")
    sliderEdge = Draw.Slider("Length of the edge: ", EV_SL_EDGELENGTH, 5, line
    [6], 310, 25, sliderEdge.val, 0.25, 2.00, 1, "Length of edge")
    numberCentreX = Draw.Number("Centre X: ", EV_NB_CENTERX, 5, line[5], 100,
    25, numberCentreX.val, -5.00, 5.00, "X coordinate of the Center")
    numberCentreY = Draw.Number("Centre Y: ", EV_NB_CENTERY, 110, line[5], 100,
    25, numberCentreY.val, -5.00, 5.00, "Y coordinate of the Center")
    numberCentreZ = Draw.Number("Centre Z: ", EV_NB_CENTERZ, 215, line[5], 100,
    25, numberCentreZ.val, -5.00, 5.00, "Z coordinate of the Center")
    BGL.glRasterPos2i(5,line[4])
    Draw.Text("Choose the colour of the cube:")
    sliderR = Draw.Slider("R: ", EV_SL_R, 5, line[3], 100, 25, sliderR.val,
    0.00, 1.00, 1, "Red component of the colour")
    sliderG = Draw.Slider("V: ", EV_SL_G, 5, line[2], 100, 25, sliderG.val,
```
5 Management of events and running of the program

As we have seen in chapter 1 of this article, we must understand how to handle two types of events. This management is intimately linked to the execution of the program, because we wish (or not!) that the user can himself utilize certain parts of the script. This is in fact what will decide then the autonomy of the user in terms of the capacities of the script.

5.1 Events linked to the use of the keyboard

We already know that there are several types of events, such as for example the action of the user on the keyboard or on the mouse. But in the present case, we are going to be interested only in the action of the keys of the keyboard, our goal being of putting in place some key shortcuts allowing the script to launch the script that causes creation of a cube (C key) or of quitting the script (Q key or ESCAPE).

Elementary syntax:

```python
def event(event, val):
    if event == Draw.ESCKEY:
        Draw.Exit()
    if event == ...:
        ...
```

The principle is that of defining a function that is going to capture an event (event) and observe its value (val). While testing this value (if) it is possible to execute some commands or some particular functions of the script. In the elementary syntax above, we see that the action of the user on the key ESCAPE will lead to exiting the graphical interface, commanded by `Draw.Exit()`. The action on another key (not specified) will lead to the execution of another function. As usual, a particular attention must be accorded the indentation of your script, especially when loops or tests are embedded in the function `event()`.

In our case, we decide that the user's action on the keys ESCAPE or Q will lead to stopping the execution of the script, and return to the text window where the code appears. Likewise, we decide that the act of pressing on the C key leads to execution of a script, and traces a cube in the graphic window.

```python
def event(event, val):
    if event == Draw.ESCKEY or event == Draw.QKEY:
        Draw.Exit()
    if event == Draw.CKEY:
        MakeCube()
        Blender.Redraw()
```

But we may allow ourselves some small supplementary refinements. We can in effect make the script ask the user to confirm his action by means of a dialog box which appears for the occasion:

Elementary syntax:

```python
[name of the action] = Draw.PupMenu("[title of the box]%t|[message] %x1")
```

[title of the box]: the name (or the message) which appears in the top border of the box

[message]: this is the message asking for confirmation of the action
If, for example, the action on the Q key asks to stop the script, we can name the action stop and effect a test on the value stop. If it is confirmed (equal to 1), and then the interrupt procedure of the program follows thanks to the command `Draw.Exit()`.

Our first test of the event then becomes:

```python
def event(event, val):
    if event == Draw.ESCKEY or event == Draw.QKEY:
        stop = Draw.PupMenu("OK%t|Stop the script %x1")
        if stop == 1:
            Draw.Exit()
    if event == Draw.CKEY:
        ...
        ...
```

Certainly, nothing prevents us from equally asking for confirmation for the creation of the cube when the user presses on the C key.

```python
def event(event, val):
    if event == Draw.ESCKEY or event == Draw.QKEY:
        ...
    if event == Draw.CKEY:
        make = Draw.PupMenu("Create Cube?%t|Construct the cube %x1")
        if make == 1:
            MakeCube()
            Blender.Redraw()
```

5.2 Events linked to the use of the buttons

The principle here is only of capturing an event (evt) and effecting a test on the captured event number. In view of this, it is then possible to jump to other parts of the script. If you have well understood the preceding paragraph 5.1, this will not pose any particular problems for you.

```
Elementary syntax:
def button_event(evt):
    if evt==[event number]:
        ...
    elif evt==[other event number]:
        ...
```

In our case, we decide that pressing on the button OK launches the creation of a cube and its appearance on the screen, by means of the command `MakeCube()` and `Blender.Redraw()`. Finally, pressing on the button Quit puts a simple end to the script thanks to the command `Draw.Exit()`.

```python
def button_event(evt):
    if evt==EV_BT_OK:
        MakeCube()
        Blender.Redraw()
    elif evt==EV_BT_CANCEL:
        Draw.Exit()
```

5.3 Register

There is a seemingly magical command which must be called to link the three functions to the interface: `draw_gui` which draws to the screen, `event` which watches for the interventions of the user on the keyboard, and finally `button_event` which watches for those of the graphical interface buttons. In other words, this is given the task of emulating the surveillance of the event loop which manages these while giving a hand to different parts of the script.

```python
Draw.Register(draw_gui, event, button_event)
```

6 Conclusion

We have arrived at the end of this long article! It allows us to approach numerous possibilities for
the creation of graphical interfaces in python while using Blender. In referring you to the
documentation, you will notice that there still remains a lot to discuss, such as about the Draw and
BGL modules, but this will suffice in terms of keeping you agreeably occupied (I find the creation
of interfaces particularly intriguing) during all of this summer! Before proposing that you return
to the start, you will find the complete code of this example on the cdrom accompanying the
magazine or online at http://www.linuxgraphic.org.

Recapitulation of the code:
01: import Blender, math
02: from Blender import Camera, Object, Scene, Lamp, NMesh, Material, Draw, BGL
03: conv = 2*math.pi/360
04: # Initial values of buttons:
05: stringName = Draw.Create("Cube")
06: sliderEdge = Draw.Create(1.00)
07: numberCentreX = Draw.Create(0.00)
08: numberCentreY = Draw.Create(0.00)
09: numberCentreZ = Draw.Create(0.00)
10: sliderR = Draw.Create(0.50)
11: sliderG = Draw.Create(0.50)
12: sliderB = Draw.Create(0.50)
13: cur = Scene.getCurrent()
14: def MakeCube():
15:     var1 = numberCentreX.val
16:     var2 = numberCentreY.val
17:     var3 = numberCentreZ.val
18:     var4 = sliderEdge.val
19:     list_of_faces=
20:         [0,1,2,3],
21:         [4,5,6,7],
22:         [0,4,7,3],
23:         [1,2,6,5],
24:         [0,1,5,4],
25:         [3,7,6,2]
26:     list_of_vertices=[
27:         [-var4+var1,-var4+var2,-var4+var3],
28:         [-var4+var1,+var4+var2,-var4+var3],
29:         [+var4+var1,+var4+var2,-var4+var3],
30:         [-var4+var1,-var4+var2,+var4+var3],
31:         [-var4+var1,+var4+var2,+var4+var3],
32:         [+var4+var1,+var4+var2,+var4+var3],
33:         [+var4+var1,-var4+var2,+var4+var3]
34:         ]
35:     # Definition of the cube:
36:     CubeMeshData=NMesh.GetRaw()
37:     # Definition of control points:
38:     # Definition of faces:
39:     for component in list_of_vertices:
40:         vertex=NMesh.Vert(component[0], component[1], component[2])
41:         CubeMeshData.verts.append(vertex)
42:     for face_current in list_of_faces:
43:         face=NMesh.Face()
44:         for vertex_number in face_current:
45:             face.verts.append(CubeMeshData.verts[vertex_number])
46:         mat = Material.New('Material')
47:         CubeMeshData.materials.append(mat)
48:         mat.rgbCol = [sliderR.val, sliderG.val, sliderB.val]
49:         mat.setAlpha(1.0)
50:         mat.setRef(0.8)
51:         mat.setSpec(0.5)
52:         mat.setHardness(50)
53:         for component in list_of_vertices:
54:             vertex=NMesh.Vert(component[0], component[1], component[2])
55:             CubeMeshData.verts.append(vertex)
56:         for face_current in list_of_faces:
57:             face=NMesh.Face()
face.append(CubeMeshData.verts[vertex_number])
CubeMeshData.faces.append(face)
NMesh.PutRaw(CubeMeshData,stringName.val,1)

##### Definition of the camera

def MakeCamera():
c = Camera.New('persp','Camera')
c.lens = 35.0
cam = Object.New('Camera')
cam.link(c)
cur.link(cam)
cam.setCurrentCamera(cam)
cam.setEuler(52.928*conv, -1.239*conv, 52.752*conv)
cam.setLocation(6.283, -5.000, 5.867)

##### Definition of the lamp

def MakeLamp():
l = Lamp.New('Lamp','Lamp')
lam = Object.New('Lamp')
lam.link(l)
lam.setEuler(47.534*conv,0,0)
lam.setLocation(0, -10, 7)
cur.link(lam)

MakeCamera()
MakeLamp()

# Assignment of event numbers to the buttons:
EV_BT_OK = 1
EV_BT_CANCEL = 2
EV_SL_EDGELENGTH = 3
EV_NB_CENTERX = 4
EV_NB_CENTERY = 5
EV_NB_CENTERZ = 6
EV_SL_R = 7
EV_SL_G = 8
EV_SL_B = 9
EV_ST_NAME = 10

# Assignment of line positions (from the bottom):
line = [None,None,None,None,None,None,None,None,None]
line[0] = 5
line[1] = 35
line[2] = 65
line[3] = 95
line[4] = 125
line[5] = 155
line[6] = 185
line[7] = 215
line[8] = 245
line[9] = 275

def draw_gui():
global stringName, sliderEdge, numberCentreX, numberCentreY, numberCentreZ, sliderR, sliderG, sliderB
BGL.glRasterPos2i(5,line[9])
Draw.Text("Choose the name of the cube:")
stringName = Draw.String("Name: ", EV_ST_NAME, 5, line[8], 310, 25, stringName.val, 32, "Name of the object")
BGL.glRasterPos2i(5,line[7])
Draw.Text("Choose the geometry of the cube:")
sliderEdge = Draw.Slider("Length of the edge: ", EV_SL_EDGELENGTH, 5, line[6], 310, 25, sliderEdge.val, 0.25, 2.00, 1, "Length of edges")
numberCentreX = Draw.Number("Center X: ", EV_NB_CENTERX, 5, line [5], 100, 25, numberCentreX.val, -5.00, 5.00, "Coordinate X of the Center")
numberCentreY = Draw.Number("Center Y: ", EV_NB_CENTERY, 110, line [5], 100, 25, numberCentreY.val, -5.00, 5.00, "Coordinate Y of the Center")
numberCentreZ = Draw.Number("Center Z: ", EV_NB_CENTERZ, 215, line [5], 100, 25, numberCentreZ.val, -5.00, 5.00, "Coordinate Z of the Center")
BGL.glRasterPos2i(5,line[4])

Draw.Text("Choose the colour of the cube:")

sliderR = Draw.Slider("R: ", EV_SL_R, 5, line[3], 100, 25, sliderR.val, 0.00, 1.00, 1, "Red component of the colour")

sliderG = Draw.Slider("V: ", EV_SL_G, 5, line[2], 100, 25, sliderG.val, 0.00, 1.00, 1, "Green component of the colour")

sliderB = Draw.Slider("B: ", EV_SL_B, 5, line[1], 100, 25, sliderB.val, 0.00, 1.00, 1, "Blue component of the colour")

Draw.PushButton("OK", EV_BT_OK, 5, line[0], 60, 25, "Validate")

Draw.PushButton("Cancel", EV_BT_CANCEL, 80, line[0], 60, 25, "Quit")

def event(event, val) :
    # functions defining the events linked to the mouse or the keyboard
    if event == Draw.ESCKEY or event == Draw.QKEY:
        stop = Draw.PupMenu("OK?%t|Stop the script %x1")
        if stop == 1:
            Draw.Exit()

    if event == Draw.CKEY:
        make = Draw.PupMenu("Create Cube?%t|Construct the cube %x1")
        if make == 1:
            MakeCube()
            Blender.Redraw()

    def button_event(evt) :
        # functions defining the events linked to the buttons
        if evt==EV_BT_OK:
            MakeCube()
            Draw.Redraw()
        elif evt==EV_BT_CANCEL:
            Draw.Exit()

    Draw.Register(draw_gui, event, button_event)