Création d'une interface graphique sous Blender pour vos scripts python

Merci à Diamond Editions pour son aimable autorisation pour la mise en ligne de cet article, initialement publié dans Linux Magazine N°74

Olivier Saraja - olivier.saraja@linuxgraphic.org

Nous avons vu dans l'article précédent comment reconstruire la scène de démarrage par défaut de Blender. Pour ce faire, nous avons appris comment créer en mémoire des ensembles de données, décrivant les entités que nous souhaitons, et comment les « importer » dans Blender, pour leur donner substance. Cela a été une série d'exercices dont la complexité allait de simple à modérée, mais nous n'avons vu qu'une seule façon d'agir sur le script: la combinaison [ALT]+[P]. Nous allons toutefois voir maintenant que nous pouvons créer une interface graphique et déterminer, interactivement, le fonctionnement de notre script.

Si l'on reprend le code que nous avons composé lors du précédent numéro, nous allons essayer de lui attribuer une interface graphique qui nous permettra de déterminer la longueur des arêtes du cube, de spécifier la localisation de son centre géométrique, de donner à son matériau une couleur personnalisée, et enfin de baptiser l'objet généré.

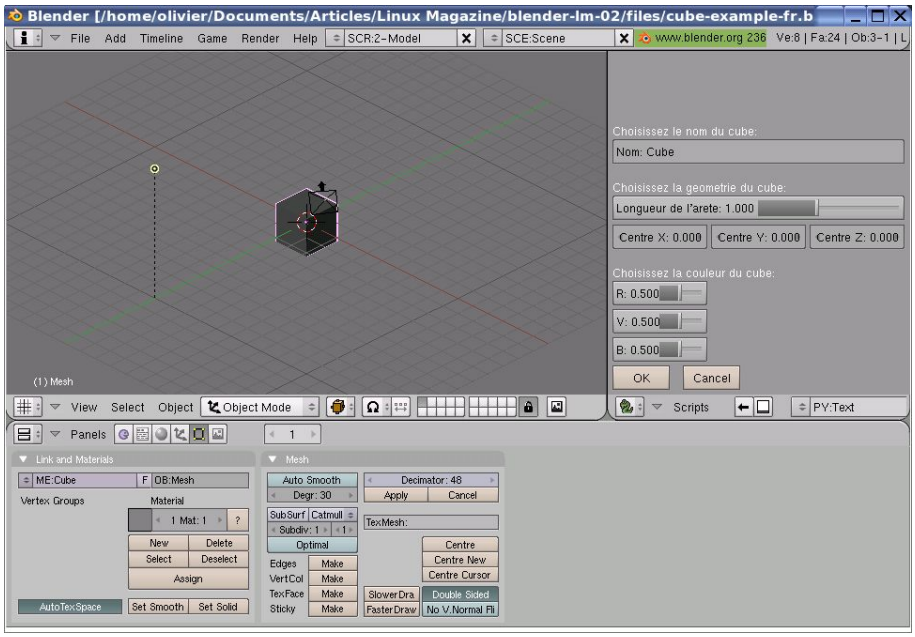

Figure 1: le résultat final de nos efforts

1. Une interface graphique, pourquoi faire?

Nous pouvons bien sûr envisager l'existence d'un script python qui ne sache faire qu'une seule chose, mais qui le fasse bien; c'est d'ailleurs le propre des outils informatiques. Mais nous pouvons aussi envisager d'autres formes de scripts, moins monolithiques, que l'on peut adapter à nos besoins. Dans le premier cas, nous nous contentons de lancer le script python et d'en observer les résultats. Dans le second cas, nous définissons certaines variables (sous forme de boutons, de glissières, de champs à remplir, par exemple) qui, en fonction de leur valeur, modifient le résultat (et peut-être même le déroulement entier) du script python.

En fait, une interface graphique, c'est une sorte de boucle sans fin qui va surveiller les interventions de l'utilisateur, et réagir en fonction. Bien conçue, pensée avec l'ergonomie et

l'utilisateur final à l'esprit, une interface graphique peut être un outil précieux dans la maîtrise d'un script potentiellement complexe. Et c'est bien là le propre de l'informatique: rendre simples des tâches complexes ou très répétitives.

Les boucles sans fin que nous allons découvrir maintenant vont nous permettre de surveiller l'apparition de deux types d'événements: ceux liés à l'usage par l'utilisateur des touches de la souris et/ou du clavier, et ceux liés à l'usage des boutons de l'interface graphique qu'il aura programmé. Le titre de cet article pourra paraître trompeur dans la mesure où nous allons bien étudier les deux cas.

A titre de référence, nous vous recommandons chaudement le script suivant, dont l'étude devrait vous aider à très rapidement comprendre le fonctionnement et l'usage des boutons de Blender: **http://infohost.nmt.edu/~jberg/blender/button_demo**

1.1 Les événements liés à l'utilisation des outils d'entrée

Nous entendons par là l'usage du clavier ou des boutons de la souris. En particulier, nous retiendrons la possibilité d'enregistrer l'action des touches suivantes du clavier: A à Z, les événements portant les noms `AKEY` à `ZKEY`, la touche Entrée (`RETKEY`), la touche Echap (`ESCKEY`), les touches de fonction F1 à F10 (`F1KEY` à **F12KEY**), la touche de Tabulation (`TABKEY`) et les touches du pavé numérique 0 à 9 (`PAD0` à `PAD9`). Côté souris, nous pouvons noter le bouton gauche (`LEFTMOUSE`), milieu (`MIDDLEMOUSE`) ou droit (`RIGHTMOUSE`), l'usage de la molette vers le haut (`WHEELUPMOUSE`) ou vers le bas (`WHEELDOWNMOUSE`). Bien sûr, les mouvements de la souris peuvent aussi être capturés grâce aux événements `MOUSEX` et `MOUSEY`.

1.2 Les événements liés à l'utilisation de l'interface graphique

Nous avons à notre disposition différents types de bouton, chacun dédié à une utilisation particulière. L'illustration qui suit reprend et illustre toutes ces possibilités. Il s'agit ni plus ni moins de l'interface qui se charge au lancement du script `button_demo.py` dont nous vous avons recommandé le lien plus haut.

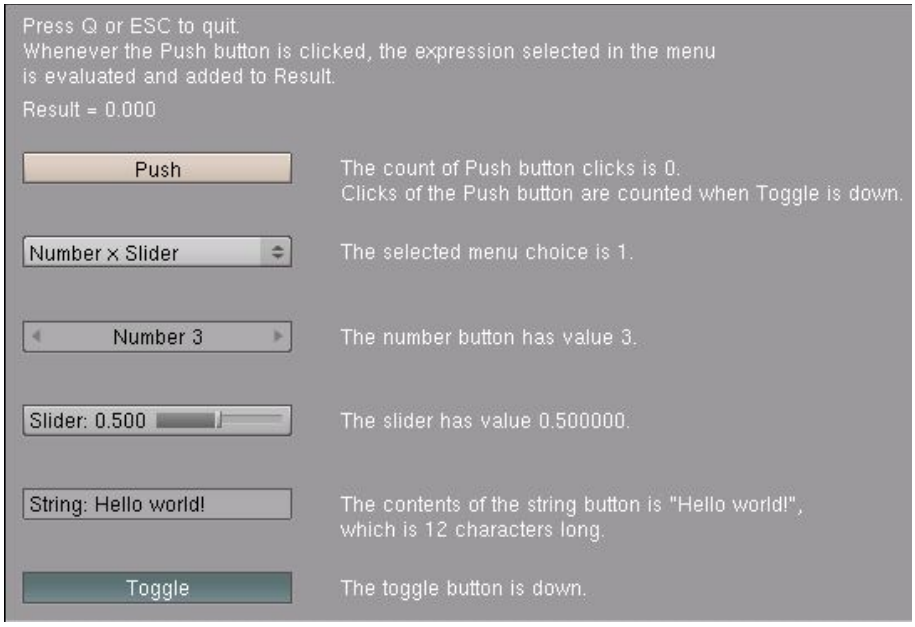

Figure 2: les principaux boutons de Blender en action

Push button, ou bouton poussoir: il s'agit du premier bouton intitulé *Push*. Son utilisation est simple: en cliquant dessus, vous déclenchez une action, le plus souvent l'exécution du script en lui-même, ou la sortie hors de l'interface graphique, par exemple. Il n'admet pas d'état général, il ne s'active que le temps d'un clic.

Menu button, ou bouton menu: c'est le bouton intitulé *Number x Slider* de l'illustration précédente. En cliquant dessus, un menu déroulant apparaît, offrant plusieurs actions possibles. Le nombre de choix est déterminé par votre script.

Number button, ou bouton chiffré: il apparaît avec le texte *Number 3* ci-dessus. Idéal pour permettre à l'utilisateur à choisir entre différents chiffres entiers qui peuvent être utilisés par le script. Il est possible de déterminer la plage possible.

Slider button, ou bouton glissière: c'est la glissière intitulée *Slider: 0.500*. L'idée générale est la mêle que pour le bouton chiffré, à l'exception que celui-ci permet de régler finement un chiffre décimal. A nouveau, le chiffre affiché peut être réutilisé dans le script, et le bouton se voit attribuer deux bornes, une supérieure et une inférieure.

String button, ou bouton texte: il s'agit du bouton *String: Hello world!* de la Figure 02. Il s'agit simplement d'un champ où il est possible de saisir un texte (de longueur déterminée par le script) qui pourra être réutilisé ultérieurement, généralement pour nommer des entités.

Toggle button, ou bouton bascule: c'est le dernier bouton, intitulé *Toggle*. Il a la particularité d'admettre deux états (actif ou inactif) et sert souvent à activer ou désactiver des options dans le déroulement de votre script, ainsi que de réaliser des choix binaires, souvent plus synthétique que le bouton menu.

Nous reverrons ultérieurement comment coder la plupart de ces boutons.

2 Par où commencer? Quelques bases...

Comme nous l'avons souligné dans la première partie, une interface est une série de boucles sans fin, qui vont surveiller certains événements ou contrôler l'affichage de l'interface elle-même. Tout commence toutefois par l'import de deux modules de l'API python de Blender: **`Draw`** et **`BGL`**.

`Draw` est le module qui va permettre de donner accès à l'interface fenêtrée de Blender ainsi que la surveillance des événements. Pour sa part, **`BGL`** est un wrapper OpenGL, qui rend disponible les constantes et fonctions d'OpenGL pour Blender, qui permettront entre autre de placer du texte, colorer des pixels, afficher des images et ce genre de choses.

Syntaxe élémentaire de l'interface

```
01: import Blender
02: from Blender import Draw, BGL
03: def event(evt, val):
04: pass
05: def button_event(evt):
06: pass
07: def draw_gui():
08: pass
09: Draw.register(draw_gui, event, button_event)
```

Nous définissons ensuite trois fonctions, que nous nommons de façon explicite pour nous. La première, **`event()`** va nous permettre de surveiller les événements issus de l'action de l'utilisateur sur le clavier ou la souris. La seconde, **`button_event()`** va nous permettre de gérer le déroulement du script en fonction de l'action de l'utilisateur sur les boutons de l'interface graphique. La troisième et dernière, **`draw_gui()`**, permet de dessiner à l'écran les boutons et le texte de l'interface et de leur associer les variables de votre choix. La programmation de l'interface se conclue par un **`Draw.Register()`** qui permet d'enregistrer et d'activer les boucles qui lui sont passées en arguments.

Vous noterez que la définition d'une fonction est de la forme:

```
def [nom fonction]():
```

et que l'indentation à sa suite permet de délimiter le code la concernant. La commande **`pass`**, ici, est inutile et ne sert qu'à produire un code qui, s'il est correct du point de vue de la syntaxe, ne réalise absolument rien.

Et voilà! Nous avons désormais le squelette à la fois de notre script et de notre interface graphique, il ne nous reste plus qu'à garnir copieusement!

2.1 Les fonctions

Bien sûr, nous avons deux options: soit mettre en place le code à exécuter en réponse à un

événement dans la boucle correspondante, soit simplement spécifier une fonction que la boucle appellera. Cette dernière méthode a pour avantage de vous permettre de définir vos fonctions en « amont » de votre programme, et de réutiliser librement celles-ci en « aval », simplement en appelant la fonction par son nom. A chacun ses habitudes de programmation, mais dans le cadre de programmes complexes pour lesquels vous souhaitez avoir un déroulement le plus clair possible (par exemple pour vous conformer à un synoptique ou un logigramme), fonctionner de la sorte peut rapidement devenir un atout.

Sans rentrer dans le détail de la programmation de fonction, il suffit de retenir la syntaxe élémentaire suivante:

Syntaxe élémentaire d'une fonction:

```
def [nom de la fonction](variable1, variable2..., variablen)
    ... #code que doit exécuter la fonction
    return #ou return(var1, var2..., varn)
```

Le nom de la fonction devra être choisie de façon explicite, tandis que les variables entre parenthèses sont celles sur lesquelles la fonction s'appliquera; il est impératif qu'elle soit, d'une façon ou d'une autre, définies par ailleurs dans le programme. Si la fonction n'a pas besoin de retourner quoi que ce soit, la commande finale **`return`** est optionnelle. Si elle doit retourner quelque chose (comme le résultat d'un calcul), la commande **`return()`** devient obligatoire si l'on souhaite réutiliser le résultat de la fonction par ailleurs dans le programme.

Par exemple, le programme qui suit permet de prendre deux paires de variables (**`var1`** et **`var2`**, ainsi que **`cvar3`** et **`var4`**) définies de la ligne 03 à 06, et de les passer à la fonction **`multiplication()`**. Celle-ci effectue les opérations que l'on attend d'elle, et retourne le produit dans la variable **`resultat`**, en ligne 11, que le programme imprime dans la console grâce à la commande **`print`**. Pour pouvoir être réutilisée en-dehors de la fonction, il sera nécessaire de définir, au préalable, la variable **`resultat`** comme étant globale, grâce à la ligne 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 resultat
10:    resultat = variable1 * variable2
11:    return(resultat)
12:
13: multiplication(var1, var2)
14:
15: print resultat
16:
17: multiplication(var3, var4)
18:
19: print resultat
```

En regardant de plus près la fonction, on se rend compte qu'il s'agit d'une bête multiplication, les deux variables saisies en entrée étant multipliée et retournée en sortie sous forme de résultat. Dans l'exemple, nous utilisons consécutivement deux fois la fonction **`multiplication()`** et en affichons le résultat dans la console. Dans le premier cas, en ligne 13, nous passons à la fonction les variables **`var1`** et **`var2`**. Dans le second, en ligne 17, nous lui passons les variables **`var3`** et **`var4`**. Il ne nous reste plus qu'à lancer le script ([ALT]+[P]) et d'observer les résultats dans la console; affichés respectivement par les lignes 15 et 19:

```
2
4
```

2.2 Le coeur du générateur de cube

Nous allons faire simple, et récupérer *in extenso* le code de l'article précédent pour la génération de la lampe et de la caméra. Vous le retrouverez sur le cédérom d'accompagnement du magazine

ou du numéro précédent, ou en ligne sur **http://www.linuxgraphic.org**, sous le nom `blender-default-scene.py` et `blender-default-scene.blend`.

Comme tout bon script python, il commence par l'importation du module Blender et des sous-modules nécessaires au déroulement du script. En l'occurrence, nous allons avoir besoin de `Camera`, `Object`, `Lamp`, `NMesh` et `Material`:

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

Nous allons juste placer les bouts de code qui nous intéressent dans des fonctions appropriées, qui ne demandent aucune variable en entrée, et ne retournent aucune valeur en sortie. Ainsi, pour la caméra, nous obtenons le code suivant:

```
##### Definition de la camera
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)
```

De même, pour la lampe, nous obtenons ceci:

```
##### Definition de la lampe
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)
```

Ces deux fonctions seront simplement et ultérieurement appelées par les lignes suivantes:

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

mais il faut prendre garde à importer la scène courante avant d'effectuer la définition de ces fonctions:

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

ainsi qu'à définir la variable `conv` qui assure la conversion des radians de Python en degrés de Blender:

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

Le copier/coller ne nous faisant apparemment pas très peur, nous pourrions faire de même avec le code de génération du cube. Toutefois, nous souhaitons pouvoir personnaliser celui-ci au-travers de l'interface graphique de notre script. Comme mentionné dans l'introduction de cet article, nous souhaitons en particulier déterminer la longueur des arêtes du cube, spécifier la localisation de son centre géométrique, donner à son matériau une couleur personnalisée, et enfin baptiser le cube du nom de notre choix.

Nous pouvons donc d'ores et déjà faire des choix quant à l'apparence future de notre interface. Avec un peu d'avance, dévoilons le look de notre future interface pour mieux saisir les paramètres que nous allons décrire et visualiser les boutons associés.

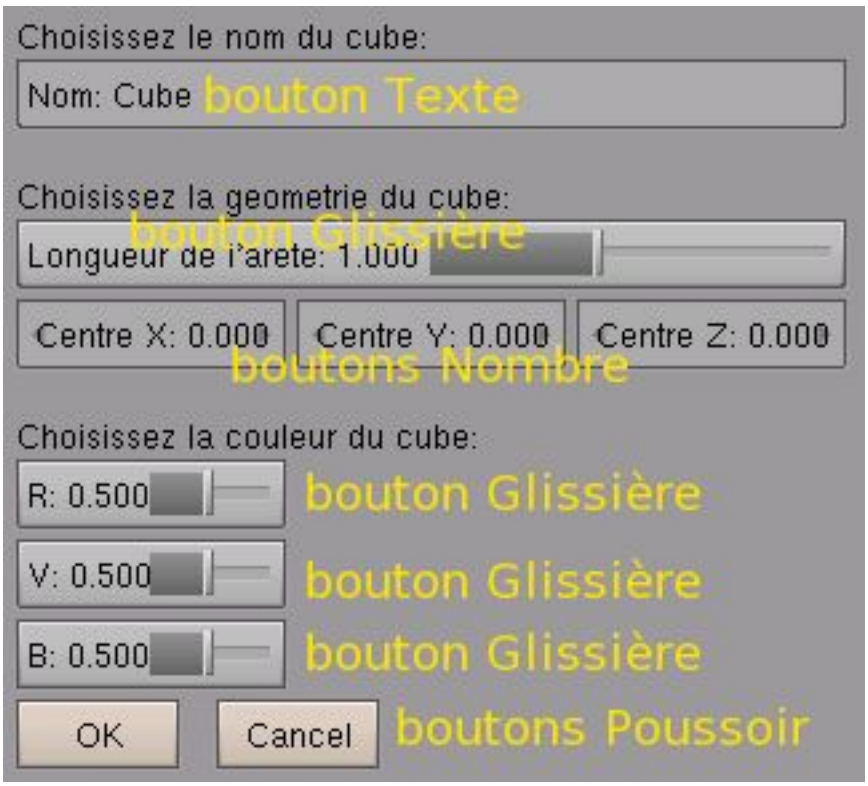

Figure 3: l'interface à venir, pour le choix des boutons et des variables représentatives!

- Nom du cube, déterminé par un **bouton texte**, variable associée: **`stringName.val`**
- Longueur de l'arête, réglée par un **bouton glissière**, paramètre associé: **`sliderEdge.val`**
- Coordonnée X du centre du cube, réglée par un **bouton chiffré**, variable associée: **`numberCentreX.val`**
- Coordonnée Y du centre du cube, réglée par un **bouton chiffré**, variable associée: **`numberCentreY.val`**
- Coordonnée Z du centre du cube, réglée par un **bouton chiffré**, variable associée: **`numberCentreZ.val`**
- Composante R (rouge) de la couleur du cube, réglée par un **bouton glissière**, variable associée: **`sliderR.val`**
- Composante G (verte) de la couleur du cube, réglée par un **bouton glissière**, variable associée: **`sliderG.val`**
- Composante B (bleue) de la couleur du cube, réglée par un **bouton glissière**, variable associée: **`sliderB.val`**

Les variables associées aux boutons revêtent une très grande importance, car nous allons les réutiliser massivement dans le code de génération de notre cube, où elles remplaceront toutes les valeurs fixes que notre interface propose de faire varier. En particulier, **`sliderEdge.val`**, **`numberCentreX.val`**, **`numberCentreY.val`** et **`numberCentreZ.val`**, qui seront avantageusement remplacées par des variables plus faciles à manipuler:

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

Dans le programme que nous avons élaboré dans l'article précédent (en ligne sur **http://www.linuxgraphic.org**), les sommets de notre cube étaient décrits par les composantes suivantes:

```
liste_des_sommets=[
    [-1,-1,-1],
    [-1,+1,-1],
    [+1,+1,-1],
    [+1,-1,-1],
```

```
        [-1,-1,+1],
        [-1,+1,+1],
        [+1,+1,+1],
        [+1,-1,+1]
    ]
```

Ils vont maintenant devenir:

```
        # Definition des points de controle:
        liste_des_sommets=[
               [-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]
        ]
```

Sceptiques? Il y a un moyen très simple de vérifier la validité de ces nouveaux sommets, en substituant les variables par les valeurs fixes du programme d'origine. Ainsi, en supposant un cube d'arête 1 unité (`sliderEdge = var = 4`) et d'origine centrée sur `[0,0,0]` (`numberCentreX = var1 = 0`, `numberCentreY = var2 = 0`, et `numberCentreZ = var3 = 0`), nous obtenons bien, pour la première ligne de sommets `[-1,-1,-1]`.

De même, le matériau ne sera plus défini par une couleur grise invariante, mais par l'action de l'utilisateur sur trois **boutons glissière** qui permettront de donner à chaque composante R, G et B des valeurs différentes: `sliderR.val`, `sliderG.val` et `sliderB.val`. Nous obtenons donc la déclaration de couleur suivante pour le matériau courant:

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

Enfin, nous avons également attribué à l'interface un **bouton texte**, afin que le nom du cube soit librement déterminé par l'utilisateur du script. Le nom saisi est stocké dans la `stringName.val`.

A noter qu'à l'instar de `MakeLamp()` et `MakeCamera()`, cette fonction n'a besoin de prendre aucune fonction en entrée, et ne retourne rien non plus, se contentant d'exécuter des actions. Aucune ligne `return()` ne conclue donc ces fonctions, et la fonction se présente simplement sous la forme:

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

Il ne nous reste plus qu'à reprendre l'ancien code de génération du cube, en respectant les aménagements et substitutions décrits jusque là. Nous obtenons alors très facilement la fonction finale suivante:

```
    #### Definition du cube:
    def MakeCube():
        var1 = numberCentreX.val
        var2 = numberCentreY.val
        var3 = numberCentreZ.val
        var4 = sliderEdge.val
        # Definition des points de controle:
        liste_des_sommets=[
               [-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 des faces:
        liste_des_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 du materiau:
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 composante in liste_des_sommets:
       sommet=NMesh.Vert(composante[0], composante[1], composante[2])
       CubeMeshData.verts.append(sommet)
for face_courante in liste_des_faces:
       face=NMesh.Face()
       for numero_vertex in face_courante:
              face.append(CubeMeshData.verts[numero_vertex])
              CubeMeshData.faces.append(face)
NMesh.PutRaw(CubeMeshData,stringName.val,1)
```

Cette fonction sera appelée par le script au moyen d'une simple ligne:

```
MakeCube()
```

Il ne nous reste plus qu'à nous attaquer à la partie la plus *fun*, à laquelle cet article est plus spécifiquement consacré: la création de l'interface graphique.

3 L'interface graphique (ou l'habillage de notre script)

Il était temps que nous abordions le coeur du sujet de cet article, mais vous constaterez combien cette partie est simple, en vérité, du moment que vous soyez passablement ordonné et méthodique. La création d'une interface graphique passe par quelques étapes indispensables, que nous allons décrire et commenter.

3.1 Définir les bonnes importations

Il est essentiel d'indiquer à python quels sont les sous-modules du module Blender à utiliser au cours de son exécution. Cela se passe généralement par l'établissement, en tête du programme, des deux lignes suivantes:

```
import Blender
from Blender import ...
```

où l'on précise les sous-modules qui nous intéressent, par exemple **`Camera`**, **`Object`**, **`Scene`**, **`Lamp`**, **`NMesh`**, et **`Material`** dans le programme du précédent numéro. Mais l'établissement d'une interface graphique fait appel à deux sous-modules supplémentaires: **`Draw`** et **`BGL`**. Le premier est relatif au support par python d'une interface graphique intégrée à Blender; elle fournit la gestion des événements, le dessin des boutons à l'écran et quelques autres petits raffinements. Le premier est relatif à l'usage dans l'interface d'éléments openGL, comme le placement de texte, la colorisation ou le dessin d'éléments 2D et de nombreuses autres possibilités.

Nos deux premières lignes devront donc ressembler à:

Syntaxe élémentaire

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

Ce qui, dans le cas plus particulier de notre programme, va se transformer en les deux lignes suivantes:

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

3.2 Définition des valeurs par défaut des boutons

Bien évidemment, les boutons qui seront dessinés à l'écran prendront des valeurs de départ que

nous sommes libres de déterminer. Si nous revenons à la Figure 3, qui présentait l'interface telle que nous la visons, nous voyons que chaque bouton porte un label et une valeur, le cas échéant. Nous avons déjà défini, de haut en bas et de gauche à droite, les variables associées à chaque bouton:

```
stringName.val
sliderEdge.val
numberCentreX.val
numberCentreY.val
numberCentreZ.val
sliderR.val
sliderG.val
sliderB.val
```

et il ne nous reste plus maintenant qu'à leur attribuer leurs valeurs par défaut. Cela se fait grâce à la commande `Draw.Create()`:

> **Syntaxe élémentaire:**
>
> `[nom du bouton] = Draw.Create([valeur initiale)]`
>
> Attention à bien utiliser le nom du bouton et pas le nom de sa variable. Par exemple:
>
> `sliderEdge = DrawCreate(1.00)`

Les valeurs sont directement lues sur les boutons de la Figure 3; les valeurs numériques sont directement saisies entre parenthèses, les chaînes de caractères le sont entre guillemets. Pour notre petit programme, nous obtenons donc les déclarations suivantes:

```
# Valeurs initiales des boutons:
stringName = 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)
```

Cela n'a rien à voir avec la création d'interface, mais il s'agit plus d'une petite astuce pour alléger le travail de composition des interfaces, dans la mesure où il est facile de se mélanger dans les lignes ou les valeurs des nombreux paramètres des boutons! Par exemple, dans ce qui suit, nous ne décriront pas la position de chaque ligne par rapport à sa valeur y de l'écran, mais par rapport à une variable pré-établie. Ainsi, si nous nommons les lignes comme sur la figure suivante:

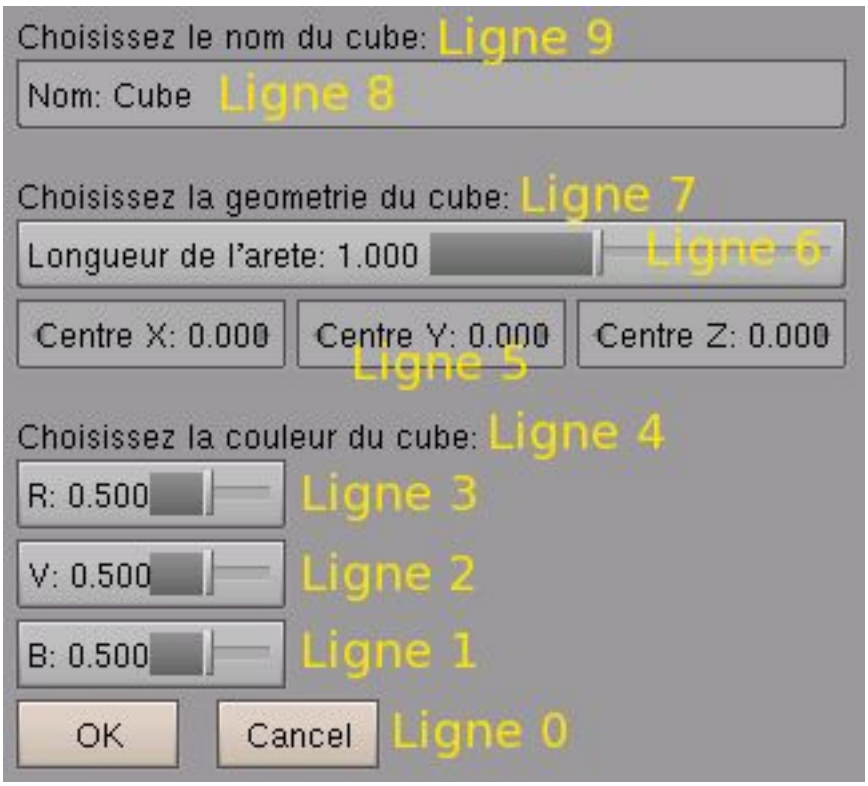

Figure 4: nommer les lignes peut permettre de gagner du temps et simplifier le travail d'ergonomie

et que nous définissons la hauteur d'une ligne comme étant égale à 25 et chaque interligne comme étant égale à 5, en définissant notre première ligne (ligne[0]) comme débutant à une altitude de 5 pixels à partir du bas de la fenêtre du script, nous pouvons définir les 10 lignes suivantes:

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

3.3 Assignation d'un numéro d'événement à chaque bouton

Il s'agit d'une étape facultative sous cette forme, car un numéro fixe peut être assigné à chaque bouton lors de sa définition. Par exemple:

```
Draw.PushButton("OK", 1, 5, ligne[0], 60, 25, "Valider")
```

(mais nous reviendrons sur la création des boutons plus tard). Nous préférons toutefois, généralement, assigner le nom d'une variable en guise de numéro d'événement lors de la définition du bouton, et assigner à cette variable un numéro lors d'une phase préalable d'assignation. Par exemple:

```
EV_BT_OK = 1
Draw.PushButton("OK", EV_BT_OK, 5, ligne[0], 60, 25, "Valider")
```

Les deux résultats sont rigoureusement identiques, sauf que dans le deuxième cas, si vous construisez votre interface au fur et çà mesure et la réorganisez régulièrement, sans trop savoir où vous allez, vous ne risquez pas de vous mélanger les pinceaux en attribuant un même numéro d'événements à deux boutons différents.

Par exemple, nous définirons les numéros d'événements suivants:

```
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
```

Notez bien que le nom des variables est laissé à votre attention. J'ai préféré reprendre la convention tacite de Yves Bailly dans sa présentation de python avec Blender, dans son article présenté dans GNU/L Mag #68 (ou en ligne sur **http://www.kafka-fr.net**), en l'adaptant à mes besoins. C'est à dire `EV_` pour indiquer un événement, puis un code de deux lettres pour indiquer le type de bouton concerné par l'événement (`BT` pour bouton, `SL` pour slider, `NB` pour number, `ST` pour string, etc.) et enfin un nom qui vous paraît un peu explicite (`CENTERX` pour la coordonnée X du centre, par exemple).

4 Définition de l'interface tracée à l'écran

Il va s'agit d'une fonction, au même titre que `MakeCube()`, `MakeLight()` ou `MakeCamera()` sauf que son appel et son exécution seront réalisés très différemment.

Syntaxe élémentaire:

```
def draw_gui():
    ...
```

Vous pouvez bien-sûr donner à la fonction n'importe quel nom, ici `draw_gui()`. Attention à l'indentation qui délimite le contenu de la fonction, soyez donc rigoureux avec celle-ci.

4.1 Définition globale des variables de l'interface:

Le but du jeu étant de rendre les valeurs des boutons disponibles pour le reste du script python, nous avons tout intérêt à les déclarer de façon globale plutôt que locale. Cela se fait tout simplement par la commande `global`, suivie de toutes les variables dont on souhaite rendre possible l'accès.

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

4.2 Afficher du texte

Il s'agit d'une petite fonction bien pratique pour garnir l'interface graphique à l'aide d'instructions ou d'astuces, comme les touches de raccourci. Pour y parvenir, il va nous falloir deux instructions. La première est issue du sous-module `Bgl`, et permet de positionner le texte de votre choix:

Syntaxe élémentaire:

```
BGL.glRasterPos2i(x,y)
```

x est la position en largeur de l'élément à placer, et y la position en hauteur de l'élément. A noter que le placement se fait par rapport au coin inférieur gauche de l'écran.

La seconde est issue du module `Draw` et permet d'afficher une chaîne de caractères:

Syntaxe élémentaire:

```
Draw.Text("[votre texte]")
```

Vous pouvez remplacer les crochets et leur contenu par le texte de votre choix, tel qu'il apparaîtra à l'écran.

Il est donc très facile d'afficher les textes relatifs à nos lignes 4, 7 et 9:

```
def draw_gui():
        ...
        BGL.glRasterPos2i(5,ligne[9])
        Draw.Text("Choisissez le nom du cube:")
        BGL.glRasterPos2i(5,ligne[7])
        Draw.Text("Choisissez la geometrie du cube:")
        BGL.glRasterPos2i(5,ligne[4])
        ...
```

4.3 Les boutons poussoirs

Les **boutons poussoirs**, qui n'admettent pas d'état, vont simplement nous servir à déclencher des actions. Par exemple, **OK** va lancer le script conformément aux paramètres qui seront affichés à cet instant là, tandis que **Annuler** sortira de l'interface graphique pour revenir au code du script.

Syntaxe élémentaire:

```
Draw.PushButton("[nom]", [numéro d'événement], [position x], [position y],
[largeur], [hauteur], "[astuce]")
```

`[nom]`: correspond à la chaîne de caractères qui sera affichée sur le bouton

`[numéro d'événement]`: il s'agit du numéro passé à l'événement du bouton lorsqu'il est activé

`[position x]`: correspond à la coordonnée selon x (dans le sens de la largeur de l'écran) du point inférieur gauche du bouton

`[position y]`: correspond à la coordonnée selon y (dans le sens de la hauteur de l'écran) du point inférieur gauche du bouton

`[largeur]`: il s'agit tout simplement de la largeur du bouton

`[hauteur]`: la hauteur du bouton

`[astuce]`: il s'agit du texte de l'info-bulle qui apparaît lorsque vous promenez le curseur de la souris au-dessus du bouton; ignorer si vous en voulez pas d'info-bulle

Leur mise en place est très simple, nous devons juste faire attention que les numéros d'événements soient remplacées par les bonnes variables (EV_BT_OK et EV_BT_CANCEL) et que les boutons soient de la bonne dimension. En effet, la ligne faisant 25 pixels de haut et l'interligne étant de 5 pixels, nous choisirons d'avoir des boutons dont la hauteur n'excède pas 25 pixels.

```
def draw_gui():
        ...
        Draw.PushButton("OK", EV_BT_OK, 5, ligne[0], 60, 25, "Valider")
        Draw.PushButton("Cancel", EV_BT_CANCEL, 80, ligne[0], 60, 25, "Annuler")
        ...
```

4.4 Les boutons glissière

La particularité de ces boutons est d'admettre une valeur minimale, une valeur maximale, et de varier entre ces deux bornes par l'action de la souris par l'utilisateur sur la glissière. La position sur laquelle sera arrêtée la glissière sera interprétée comme étant la valeur que doit prendre la variable associée à ce **bouton Glissière**.

Syntaxe élémentaire:

```
[nom bouton] = Draw.Slider("[nom]", [numéro d'événement], [position x],
[position y], [largeur], [hauteur], [valeur initiale], [valeur minimale],
[valeur maximale], [fonction temps réel], "[astuce]")
```

`[nom]`: correspond à la chaîne de caractères qui sera affichée sur le bouton

`[numéro d'événement]`: il s'agit du numéro passé à l'événement du bouton lorsqu'il est activé

`[position x]`: correspond à la coordonnée selon x (dans le sens de la largeur de l'écran) du point inférieur gauche du bouton

`[position y]`: correspond à la coordonnée selon y (dans le sens de la hauteur de l'écran) du point inférieur gauche du bouton

`[largeur]`: il s'agit tout simplement de la largeur du bouton...

`[hauteur]`: ... et de la hauteur du bouton

`[valeur initiale]`: la valeur que prend le bouton au démarrage du script. Utilisez de préférence une variable dont le nom est dérivée du nom du bouton, comme [nom bouton].val

`[valeur minimale]`: il s'agit de la valeur en-dessous de laquelle le bouton glissière ne peut aller

`[valeur maximale]`: idem, mais du point de vue de la valeur maximale

`[fonction temps réel]`: si vous spécifiez une valeur non-nulle, le bouton émettra des signaux d'événement en temps réel, c'est à dire que tout changement de valeur sera immédiatement pris en compte par le script et éventuellement affiché par Blender.

`[astuce]`: il s'agit du texte de l'info-bulle qui apparaît lorsque vous promenez le curseur de la souris au-dessus du bouton; ignorer si vous en voulez pas d'info-bulle

A noter la fonction « temps réel » de ce bouton. Par exemple, si nous attribuons à la longueur des arêtes une glissière dont la valeur par défaut est 1.00. Si nous créons le cube (en appuyant sur la touche OK du script) et qu'ensuite nous modifions la longueur de l'arête en usant de la glissière, le cube sera automatiquement et instantanément redimensionné dans la fenêtre 3D de Blender.

```
def draw_gui():
       ...
       sliderEdge = Draw.Slider("Longueur de l'arete: ", EV_SL_EDGELENGTH, 5, ligne
[6], 310, 25, sliderEdge.val, 0.25, 2.00, 1, "Longueur des aretes")
       sliderR = Draw.Slider("R: ", EV_SL_R, 5, ligne[3], 100, 25, sliderR.val,
0.00, 1.00, 1, "Composante Rouge de la couleur")
       sliderG = Draw.Slider("V: ", EV_SL_G, 5, ligne[2], 100, 25, sliderG.val,
0.00, 1.00, 1, "Composante Verte de la couleur")
       sliderB = Draw.Slider("B: ", EV_SL_B, 5, ligne[1], 100, 25, sliderB.val,
0.00, 1.00, 1, "Composante Bleue de la couleur")
       ...
```

4.5 Les boutons Nombre

Il n'y a que peu de différences fondamentales entre un **bouton Nombre** et un **bouton Glissière** comme nous venons de le voir. Dans un cas (glissière), vous devez actionner la glissière avec la souris pour augmenter ou diminuer sa valeur, dans le second cas (nombre) vous vous servez des petites flèches de par et d'autre du bouton pour modifier la valeur. Dans tous les cas, en cliquant sur le label (le texte) du bouton, vous obtenez la possibilité de saisir au clavier la valeur exacte que vous souhaitez. La seule vraie différence réside dans le fait que le **bouton Glissière** a une capacité temp réel que le **bouton Nombre** n'a pas.

Syntaxe élémentaire:

```
[nom bouton] = Draw.Number("[nom]", [numéro d'événement], [position x],
[position y], [largeur], [hauteur], [valeur initiale], [valeur minimale],
[valeur maximale], "[astuce]")
```

`[nom]`: correspond à la chaîne de caractères qui sera affichée sur le bouton

`[numéro d'événement]`: il s'agit du numéro passé à l'événement du bouton lorsqu'il est activé

`[position x]`: correspond à la coordonnée selon x (dans le sens de la largeur de l'écran) du point inférieur gauche du bouton

`[position y]`: correspond à la coordonnée selon y (dans le sens de la hauteur de l'écran) du point inférieur gauche du bouton

`[largeur]`: il s'agit tout simplement de la largeur du bouton...

`[hauteur]`: ... et de la hauteur du bouton

`[valeur initiale]`: la valeur que prend le bouton au démarrage du script. Utilisez de préférence une variable dont le nom est dérivée du nom du bouton, comme [nom bouton].val

`[valeur minimale]`: il s'agit de la valeur en-dessous de laquelle le bouton glissière ne peut aller

`[valeur maximale]`: idem, mais du point de vue de la valeur maximale

`[astuce]`: il s'agit du texte de l'info-bulle qui apparaît lorsque vous promenez le curseur de la souris au-dessus du bouton; ignorer si vous en voulez pas d'info-bulle

Si la mise en place d'un **bouton Glissière** ne vous pose pas de soucis, celle d'un **bouton Nombre** devrait êter tout aussi facile, puisque sa syntaxe est la même, à l'exception du paramètre « temps réel » près.

```
def draw_gui():
      ...
      numberCentreX = Draw.Number("Centre X: ", EV_NB_CENTERX, 5, ligne[5], 100,
25, numberCentreX.val, -5.00, 5.00, "Coordonnee X du Centre")
      numberCentreY = Draw.Number("Centre Y: ", EV_NB_CENTERY, 110, ligne[5], 100,
25, numberCentreY.val, -5.00, 5.00, "Coordonnee Y du Centre")
      numberCentreZ = Draw.Number("Centre Z: ", EV_NB_CENTERZ, 215, ligne[5], 100,
25, numberCentreZ.val, -5.00, 5.00, "Coordonnee Z du Centre")
      ...
```

4.6 Le bouton Texte

Toujours plus simple, le **bouton Texte**. Il collecte des chaînes de caractères sous le nom de variables pour les mettre à disposition du script python. Très souvent utilisé pour afficher des messages personnaliser, ou pour nommer des entités selon les souhaits de l'utilisateur.

Syntaxe élémentaire:

```
[nom bouton] = Draw.String("[nom]", [numéro d'événement], [position x],
[position y], [largeur], [hauteur], [valeur initiale], [longueur max de la
chaîne], "[astuce]")
```

`[nom]`: correspond à la chaîne de caractères qui sera affichée sur le bouton

`[numéro d'événement]`: il s'agit du numéro passé à l'événement du bouton lorsqu'il est activé

`[position x]`: correspond à la coordonnée selon x (dans le sens de la largeur de l'écran) du point inférieur gauche du bouton

[position y]: correspond à la coordonnée selon y (dans le sens de la hauteur de l'écran) du point inférieur gauche du bouton

[largeur]: il s'agit tout simplement de la largeur du bouton...

[hauteur]: ... et de la hauteur du bouton

[valeur initiale]: la valeur que prend le bouton au démarrage du script. Utilisez de préférence une variable dont le nom est dérivée du nom du bouton, comme [nom bouton].val

[longueur max de la chaîne]: cette valeur définit le nombre maximal de caractères que l'utilisateur peut employer dans la définition de sa chaîne de caractères

[astuce]: il s'agit du texte de l'info-bulle qui apparaît lorsque vous promenez le curseur de la souris au-dessus du bouton; ignorer si vous en voulez pas d'info-bulle

Ce type de bouton, le dernier que nous étudierons aujourd'hui, ne présente pas de difficultés particulières.

```
def draw_gui():
        ...
        stringName = Draw.String("Nom: ", EV_ST_NAME, 5, ligne[8], 310, 25,
stringName.val, 32, "Nom de l'objet")
        ...
```

4.7 L'apparence finale

La figure qui suit présente donc l'interface graphique telle que nous l'avons déterminée, en essayant d'avoir l'ergonomie et une disposition rationnelle à l'esprit. Certains scripts peuvent être très complexes et nécessiter une interface graphique à l'avenant. Il ne faut jamais oublier que l'objectif est que n'importe quel utilisateur soit en mesure de faire usage de votre script, bien qu'afficher une collection importante de boutons et de paramètres aux noms ésotériques pourra toujours impressionner les masses.

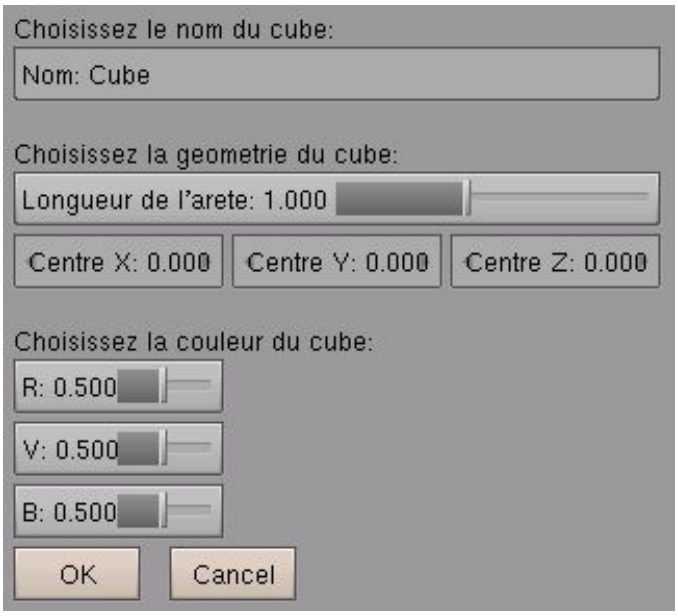

Figure 5: notre si jolie interface!

La fonction définissant l'affichage de l'interface est intégralement reprise ci-dessous:

```
def draw_gui():
        global stringName, sliderEdge, numberCentreX, numberCentreY, numberCentreZ,
sliderR, sliderG, sliderB
        BGL.glRasterPos2i(5,ligne[9])
        Draw.Text("Choisissez le nom du cube:")
        stringName = Draw.String("Nom: ", EV_ST_NAME, 5, ligne[8], 310, 25,
stringName.val, 32, "Nom de l'objet")
```

```
        BGL.glRasterPos2i(5,ligne[7])
        Draw.Text("Choisissez la geometrie du cube:")
        sliderEdge = Draw.Slider("Longueur de l'arete: ", EV_SL_EDGELENGTH, 5, ligne
[6], 310, 25, sliderEdge.val, 0.25, 2.00, 1, "Longueur des aretes")
        numberCentreX = Draw.Number("Centre X: ", EV_NB_CENTERX, 5, ligne[5], 100,
25, numberCentreX.val, -5.00, 5.00, "Coordonnee X du Centre")
        numberCentreY = Draw.Number("Centre Y: ", EV_NB_CENTERY, 110, ligne[5], 100,
25, numberCentreY.val, -5.00, 5.00, "Coordonnee Y du Centre")
        numberCentreZ = Draw.Number("Centre Z: ", EV_NB_CENTERZ, 215, ligne[5], 100,
25, numberCentreZ.val, -5.00, 5.00, "Coordonnee Z du Centre")
        BGL.glRasterPos2i(5,ligne[4])
        Draw.Text("Choisissez la couleur du cube:")
        sliderR = Draw.Slider("R: ", EV_SL_R, 5, ligne[3], 100, 25, sliderR.val,
0.00, 1.00, 1, "Composante Rouge de la couleur")
        sliderG = Draw.Slider("V: ", EV_SL_G, 5, ligne[2], 100, 25, sliderG.val,
0.00, 1.00, 1, "Composante Verte de la couleur")
        sliderB = Draw.Slider("B: ", EV_SL_B, 5, ligne[1], 100, 25, sliderB.val,
0.00, 1.00, 1, "Composante Bleue de la couleur")
        Draw.PushButton("OK", EV_BT_OK, 5, ligne[0], 60, 25, "Valider")
        Draw.PushButton("Cancel", EV_BT_CANCEL, 80, ligne[0], 60, 25, "Annuler")
```

5 Gestion des événements et déroulé du programme

Comme nous l'avons vu dans le chapitre 1 de cet article, nous allons devoir apprendre à gérer deux types d'événements. Cette gestion est intimement liée au déroulement du programme, car nous souhaitons (ou non!) que l'utilisateur puisse piloter lui-même l'exécution de certaines parties du script. C'est en fait ici qu'il sera donc décidé de l'autonomie de l'utilisateur par rapport aux capacités du script.

5.1 Evénements liés à l'usage du clavier

Nous savons déjà qu'il existe plusieurs types d'événements, comme par exemple l'action de l'utilisateur sur le clavier ou sur la souris. Mais dans le cas présent, nous ne nous intéresserons qu'à l'action sur des touches du clavier, notre but étant de mettre en place des raccourcis clavier permettant soit de lancer le script de **C**réation du cube (touche C) ou de **Q**uitter le script (touches Q ou ECHAP).

Syntaxe élémentaire:

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

Le principe est de définir une fonction qui va capter un événement (**event**) et observer sa valeur (`val`). En testant cette valeur (**if**) il est possible d'exécuter des commandes ou des fonctions particulières du script. Dans la syntaxe élémentaire ci-dessus, nous voyons que l'action de l'utilisateur sur la touche ECHAP va entraîner la sortie hors de l'interface graphique, commandée par **Draw.Exit()**. L'action sur une autre touche (non spécifiée) va entraîner le déroulement d'une autre fonction. Comme d'habitude, une attention particulière devra être accordée à l'indentation de votre script, surtout lorsque des boucles ou des tests sont imbriquées dans la fonction **event()**.

Dans notre cas, nous décidons que l'action de l'utilisateur sur les touches ECHAP ou Q conduisent à l'arrêt du script, et le retour dans la fenêtre text où apparaît le code. De même, nous décidons que l'action sur la touche C conduit à l'exécution du script, et au tracé du cube dans la fenêtre graphique.

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

```
MakeCube()
Blender.Redraw()
```

Mais nous pouvons nous permettre quelques petits raffinements supplémentaires. Nous pouvons en effet faire en sorte que le script demande à l'utilisateur de confirmer son action au travers d'une boîte de dialogue apparaissant pour l'occasion:

Syntaxe élémentaire:

```
[nom de l'action] = Draw.PupMenu("[titre de la boîte]%t|[message] %x1")
```

`[titre de la boîte]`: le nom (ou le message) apparaissant dans la bordure supérieure de la boîte

`[message]`: il s'agit du message demandant la confirmation de l'action

Si, par exemple, l'action sur la touche Q demande à stopper le script, nous pouvons nommer l'action `stop` et effectuer un test sur la valeur `stop`. Si elle est confirmée (égale à 1), et bien la procédure d'interruption du programme se poursuit grâce à la commande `Draw.Exit()`.

Notre premier test d'événement devient donc:

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

Bien sûr, rien ne nous empêche de demander également confirmation pour la création du cube lorsque l'utilisateur appuie sur la touche C.

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

5.2 Evénements liés à l'usage des boutons

Le principe est ici de seulement capter un événement (evt) et d'effectuer un test sur le numéro d'événement capturé. En fonction de celui-ci, il est alors possible d'enchaîner telle ou telle autre partie du script. Si vous avez bien compris le paragraphe 5.1 précédent, celui-ci ne devrait pas vous poser de soucis particulier.

Syntaxe élémentaire:

```
def button_event(evt) :
    if evt==[numéro d'événement]:
		...
    elif evt==[autre numéro d'événement]:
		...
```

Dans notre cas, nous décidons que l'action sur le bouton **OK** lance la création du cube et son affichage sur l'écran, au moyen des commande `MakeCube()` et `Blender.Redraw()`. Enfin, l'action sur le bouton **Annuler** met tout simplement fin au script grâce à la commande `Draw.Exit()`.

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

5.3 Register

Il s'agit de la commande magique qui va se charger d'appeler les trois fonctions liées à l'interface: **draw_gui** qui la dessinera à l'écran, **event** qui surveillera les interventions de l'utilisateur sur le clavier, et enfin **button_event** qui surveillera celles sur les boutons de l'interface graphique. En d'autres termes, c'est elle qui se charge d'émuler la boucle de surveillance des événements et qui gère ceux-ci en donnant la main à différents bouts de code du script.

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

6 Conclusion

Nous voilà arrivés au terme de ce copieux article! Il nous a permis d'approcher de nombreuses possibilités de la création d'interfaces graphiques pour python au sein de Blender. En vous référant à la documentation, vous vous rendrez compte qu'il en reste encore beaucoup à dire, tant sur les modules **Draw** que **BGL**, mais cela sera suffisant dans la mesure où cela devrait vous occuper agréablement (je trouve la création d'interfaces particulièrement ludique) tout cet été! Avant de vous proposer de vous retrouver à la rentrée, vous retrouverez le code complet de cet exemple sur le cédérom d'accompagnement du magazine ou en ligne sur **http://www.linuxgraphic.org**.

Récapitulatif du code

```
01: import Blender, math
02: from Blender import Camera, Object, Scene, Lamp, NMesh, Material, Draw, BGL
03:
04: conv = 2*math.pi/360
05:
06: # Valeurs initiales des boutons:
07: stringName = Draw.Create("Cube")
08: sliderEdge = Draw.Create(1.00)
09: numberCentreX = Draw.Create(0.00)
10: numberCentreY = Draw.Create(0.00)
11: numberCentreZ = Draw.Create(0.00)
12: sliderR = Draw.Create(0.50)
13: sliderG = Draw.Create(0.50)
14: sliderB = Draw.Create(0.50)
15:
16: cur = Scene.getCurrent()
17:
18: #### Definition du cube:
19: def MakeCube():
20: var1 = numberCentreX.val
21: var2 = numberCentreY.val
22: var3 = numberCentreZ.val
23: var4 = sliderEdge.val
24: # Definition des points de controle:
25: liste_des_sommets=[
26:         [-var4+var1,-var4+var2,-var4+var3],
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 des faces:
36: liste_des_faces=[
37:         [0,1,2,3],
38:         [4,5,6,7],
39:         [0,4,7,3],
40:         [1,2,6,5],
41:         [0,1,5,4],
42:         [3,7,6,2]
43: ]
44: CubeMeshData=NMesh.GetRaw()
```

```
45: # Definition du materiau:
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 composante in liste_des_sommets:
54:         sommet=NMesh.Vert(composante[0], composante[1], composante[2])
55:         CubeMeshData.verts.append(sommet)
56: for face_courante in liste_des_faces:
57:         face=NMesh.Face()
58:         for numero_vertex in face_courante:
59:                 face.append(CubeMeshData.verts[numero_vertex])
60:                 CubeMeshData.faces.append(face)
61: NMesh.PutRaw(CubeMeshData,stringName.val,1)
62:
63: ##### Definition de la camera
64: def MakeCamera():
65: c = Camera.New('persp','Camera')
66: c.lens = 35.0
67: cam = Object.New('Camera')
68: cam.link(c)
69: cur.link(cam)
70: cur.setCurrentCamera(cam)
71: cam.setEuler(52.928*conv, -1.239*conv, 52.752*conv)
72: cam.setLocation(6.283, -5.000, 5.867)
73:
74: ##### Definition de la lampe
75: def MakeLamp():
76: l = Lamp.New('Lamp','Lamp')
77: lam = Object.New('Lamp')
78: lam.link(l)
79: lam.setEuler(47.534*conv,0,0)
80: lam.setLocation(0, -10, 7)
81: cur.link(lam)
82:
83: MakeCamera()
84: MakeLamp()
85:
86: # Assignation des numeros d'evenement aux boutons:
87: EV_BT_OK = 1
88: EV_BT_CANCEL = 2
89: EV_SL_EDGELENGTH = 3
90: EV_NB_CENTERX = 4
91: EV_NB_CENTERY = 5
92: EV_NB_CENTERZ = 6
93: EV_SL_R = 7
94: EV_SL_G = 8
95: EV_SL_B = 9
96: EV_ST_NAME = 10
97:
98: # Assignation des positions de lignes (a partir du bas):
99: ligne = [None,None,None,None,None,None,None,None,None,None]
100: ligne[0] = 5
101: ligne[1] = 35
102: ligne[2] = 65
103: ligne[3] = 95
104: ligne[4] = 125
105: ligne[5] = 155
106: ligne[6] = 185
107: ligne[7] = 215
108: ligne[8] = 245
109: ligne[9] = 275
110:
111: def draw_gui():
112:         global stringName, sliderEdge, numberCentreX, numberCentreY,
numberCentreZ, sliderR, sliderG, sliderB
113:         BGL.glRasterPos2i(5,ligne[9])
```

```
114:Draw.Text("Choisissez le nom du cube:")
115:	stringName = Draw.String("Nom: ", EV_ST_NAME, 5, ligne[8], 310, 25,
stringName.val, 32, "Nom de l'objet")
116:	BGL.glRasterPos2i(5,ligne[7])
117:	Draw.Text("Choisissez la geometrie du cube:")
118:	sliderEdge = Draw.Slider("Longueur de l'arete: ", EV_SL_EDGELENGTH,
5, ligne[6], 310, 25, sliderEdge.val, 0.25, 2.00, 1, "Longueur des aretes")
119:	numberCentreX = Draw.Number("Centre X: ", EV_NB_CENTERX, 5, ligne
[5], 100, 25, numberCentreX.val, -5.00, 5.00, "Coordonnee X du Centre")
120:	numberCentreY = Draw.Number("Centre Y: ", EV_NB_CENTERY, 110, ligne
[5], 100, 25, numberCentreY.val, -5.00, 5.00, "Coordonnee Y du Centre")
121:	numberCentreZ = Draw.Number("Centre Z: ", EV_NB_CENTERZ, 215, ligne
[5], 100, 25, numberCentreZ.val, -5.00, 5.00, "Coordonnee Z du Centre")
122:	BGL.glRasterPos2i(5,ligne[4])
123:	Draw.Text("Choisissez la couleur du cube:")
124:	sliderR = Draw.Slider("R: ", EV_SL_R, 5, ligne[3], 100, 25,
sliderR.val, 0.00, 1.00, 1, "Composante Rouge de la couleur")
125:	sliderG = Draw.Slider("V: ", EV_SL_G, 5, ligne[2], 100, 25,
sliderG.val, 0.00, 1.00, 1, "Composante Verte de la couleur")
126:	sliderB = Draw.Slider("B: ", EV_SL_B, 5, ligne[1], 100, 25,
sliderB.val, 0.00, 1.00, 1, "Composante Bleue de la couleur")
127:	Draw.PushButton("OK", EV_BT_OK, 5, ligne[0], 60, 25, "Valider")
128:	Draw.PushButton("Cancel", EV_BT_CANCEL, 80, ligne[0], 60, 25,
"Annuler")
129:
130: def event(event, val) :   # fonctions definissant les evenements lies a la
souris ou au clavier
131:	if event == Draw.ESCKEY or event == Draw.QKEY:
132:		stop = Draw.PupMenu("OK?%t|Stopper le script %x1")
133:		if stop == 1:
134:			Draw.Exit()
135:	if event == Draw.CKEY:
136:		make = Draw.PupMenu("Creer Cube?%t|Construire le cube %x1")
137:		if make == 1:
138:			MakeCube()
139:			Blender.Redraw()
140: def button_event(evt) :   # fonctions definissant les evenements lies aux
boutons
141:	if evt==EV_BT_OK:
142:	MakeCube()
143:		Blender.Redraw()
144:	elif evt==EV_BT_CANCEL:
145:		Draw.Exit()
146: Draw.Register(draw_gui, event, button_event)
```

Liens

Le site de développement de Blender: **http://www.blender.org**

Le site officiel de Blender: **http://www.blender3d.org**

La documentation officielle de python pour Blender:
http://www.blender.org/modules/documentation/236PythonDoc/index.html

La documentation de python: **http://www.python.org/doc/2.3.5/lib/lib.html**

Exemple d'usage des boutons: **http://infohost.nmt.edu/~jberg/blender/button_demo**