Blender: premiers pas avec python

Merci à Diamond Editions pour son aimable autorisation pour la mise en ligne de cet article, initialement publié dans Linux Magazine N°73

Saraja Olivier – olivier.saraja@linuxgraphic.org

Vous avez découvert grâce à Yves Bailly dans GNU/Linux Magazine N°68 (article en ligne sur http://www.kafka-fr.net) les possibilités offertes par le couplage de Blender et de Python, dans le cadre du développement de greffons. Nous délaisserons donc quelques temps le langage de description de scènes de POV-ray (largement exploré dans les numéros précédents) pour découvrir comment l'interpréteur Python de Blender nous permet d'approcher le même niveau de puissance et de versatilité que son confère plus éprouvé.

Mais avant toute chose, comme dans tout programme, il convient de se fixer des objectifs. Dans le cadre de cet apprentissage de python, nous allons tout simplement... tenter de reproduire la scène de démarrage par défaut de Blender! Nous allons donc apprendre à créer un cube, lui attribuer un matériau, puis insérer une lampe et une caméra. Chacun de ces objets aura rigoureusement les mêmes positions et orientations (en particulier la caméra et la lampe) que ceux de la scène par défaut.

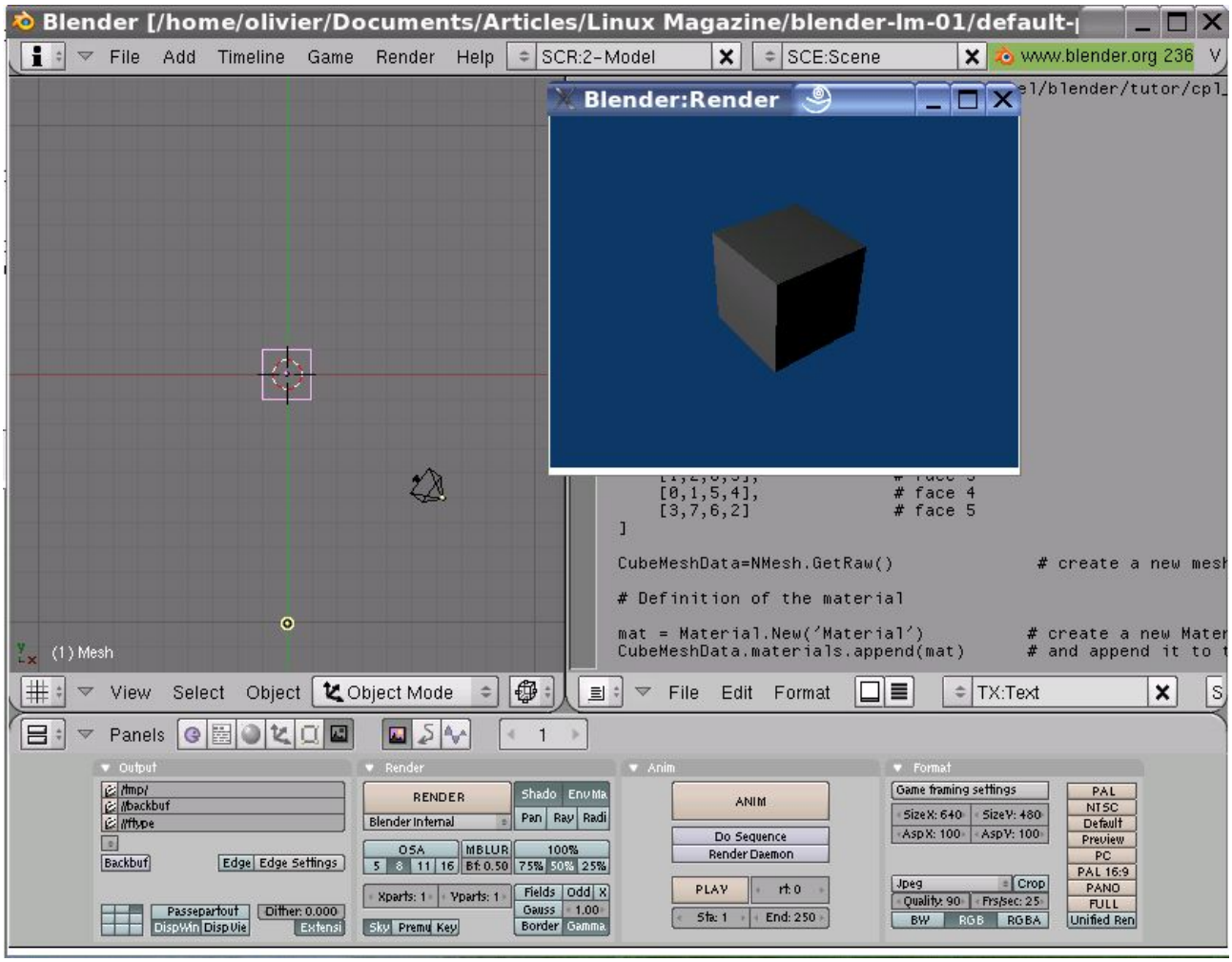

Figure 1: un objectif facile à atteindre? Oui, nous verrons comment...

1.Un python bien configuré

Un grand nombre de fonctions que je qualifierai d'auto-suffisantes sont embarquées dans l'API

python de Blender. Il en résulte qu'il est, avec les scripts actuels, de moins en moins souvent nécessaire d'avoir une installation complète de python sur son disque, voire même d'installation tout court. Il y a toutefois certains modules offerts par Python dont vous pourriez avoir besoin, comme le module **math**, par exemple, dont nous aurons marginalement besoin aujourd'hui.

1.1 Régler le pythonpath

La plupart des distributions actuelles fournissent une version de python, qu'il vous suffit d'installer. Il ne vous restera donc qu'à donner à votre système un chemin vers python. Pour déterminer celui-ci, il y a une procédure simple à accomplir. Ouvrez une console, et taper la commande **python**, avant de valider celle-ci par la touche Entrée.

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

Vous venez d'entrer dans le mode console propre à python. Tapez alors successivement **import sys** puis **print sys.path**, et vous obtiendrez quelque chose comme:

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

Ouvrez maintenant le fichier **.bashrc** à la racine de votre répertoire utilisateur (**/home/moi**). Tout à la fin dudit fichier, ajoutez la ligne suivante:

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

Notez qu'aux séparateurs près (pas d'apostrophe, et chemins séparés par deux points), c'est le contenu exact dicté par la commande **print sys.path**.

1.2 Spécifier le pythonpath dans Blender

Mais cette façon de procéder a de tout temps perdu les utilisateurs de Blender les moins familiers des environnements de programmation (et votre serviteur le premier). Heureusement, l'équipe de développement a eu l'idée d'étendre les possibilités de réglage de Blender de sorte à ce qu'il soit possible de spécifier directement dans celui-ci le chemin vers python. Démarrez Blender, et cliquez sur l'icône, en bas de la vue 3D, indiquant le type d'affichage de la vue. Choisissez **User Preferences**.

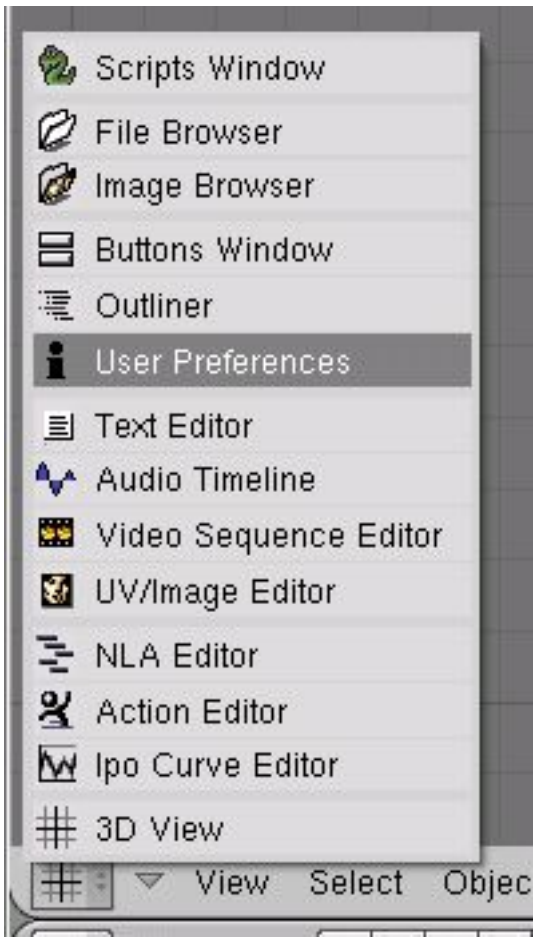

Figure 2: choisissez User Preferences pour régler le chemin vers python

La fenêtre change de physionomie, et il vous est possible de personnaliser le comportement de Blender dans plusieurs catégories différentes: **View & Controls**, **Edit Methods**, **Language & Font**, **Themes**, **Auto Save**, **System & OpenGL** et enfin, la catégorie qui va nous intéresser: **File Paths**. Cliquez sur ce dernier boutons pour découvrir toutes les chemins par défaut de Blender. Dernière ligne, deuxième colonne, repérez le champ réservé à Python. Il ne vous reste plus qu'à cliquer sur le petit répertoire pour ouvrir le mini-navigateur de fichiers et aller sélectionner directement le répertoire approprié de python.

Figure 3: et voilà le chemin vers les librairies python établi!

Il ne vous reste maintenant plus qu'à revenir à la vue 3D régulière, et d'appuyer sur [CTRL]+[U] pour sauvegarder les préférences de l'utilisateur. Ainsi, lorsque vous lancerez Blender la prochaine fois, le chemin vers l'installation de python aura été mémorisé. Autrement plus convivial que la méthode précédemment décrite, non?

2. Préliminaires et conseils de base

Je vous conseille fortement de lancer Blender depuis une console, ceci pour conserver la possibilité de visionner les messages affichés par l'interpréteur python dans la console, au cours de l'exécution de vos scripts. De cette façon, si votre code contient des erreurs de syntaxe ou des commandes inconnues, il vous sera plus facile de comprendre votre problème et de le déboguer.

Une fois Blender lancé, séparez en deux la vue 3D d'un clic droit sur la ligne de séparation horizontale entre la vue 3D et la fenêtre de commande (le curseur change de forme). Choisissez alors **Split Area** et validez d'un clic gauche.

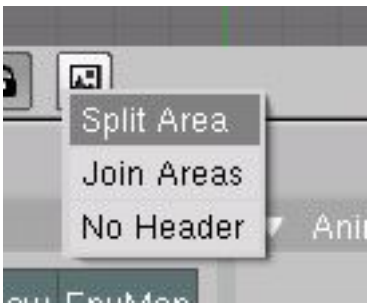

Figure 4: préparez votre espace de travail

Dans la vue de droite, ouvrez la fenêtre de l'éditeur de texte ([SHIFT] + [F11]) et créez un nouveau fichier en utilisant le raccourci [ALT]+[N] ou grâce au menu sous la fenêtre: File > New.

Activez l'affichage du numéro de lignes en cliquant sur l'icône  , cela nous sera bien utile, même si ce n'est pas indispensable.

Pour les premières lignes, nous allons y aller tout doucement. Définissons tout d'abord le « squelette » de notre script:

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

Pour l'instant, ce programme ne fait absolument rien. Vous pouvez toutefois l'obliger à s'exécuter en exécutant la combinaison de touches [ALT]+[P] en prenant soin à ce que le pointeur de la souris soit dans la fenêtre d'édition de texte.

3. Création d'un cube

Quoi, déjà, comme cela, directement? J'aurais bien aimé, mais malheureusement non. Si python nous permettra de réaliser des merveilles, rien d'aussi simple qu'une commande `cube {<x1,y1,z1><x2,y2,z2>}` comme il est de coutume avec POV-ray, toutefois. En effet, nous ne travaillons pas ici avec des primitives, mais avec des maillages. Et la création d'un maillage suit un protocole cohérent mais lourd. Nous allons l'illustrer au travers d'un cas simple, abondamment commenté.

3.1 Création d'une face

D'accord, commençons simplement. Supposons que nous voulions construire la face de la figure 5. Elle est constitué de 4 sommets, qui reliés entre eux, forment une face. Il est alors facile de relever les coordonnées des 4 sommets. Pour plus de commodités, nous allons les numéroter de 1 à 4, ou plutôt de 0 à 3. En effet, les informaticiens ayant des habitudes étranges, ils commencent à numéroter à partir de 0 plutôt que de 1, mais cela ne demande qu'une petite gymnastique à laquelle l'on s'habitue rapidement.

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

Nous pouvons maintenant aisément définir notre face comme étant celle reliant les sommets 0 à 3:

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

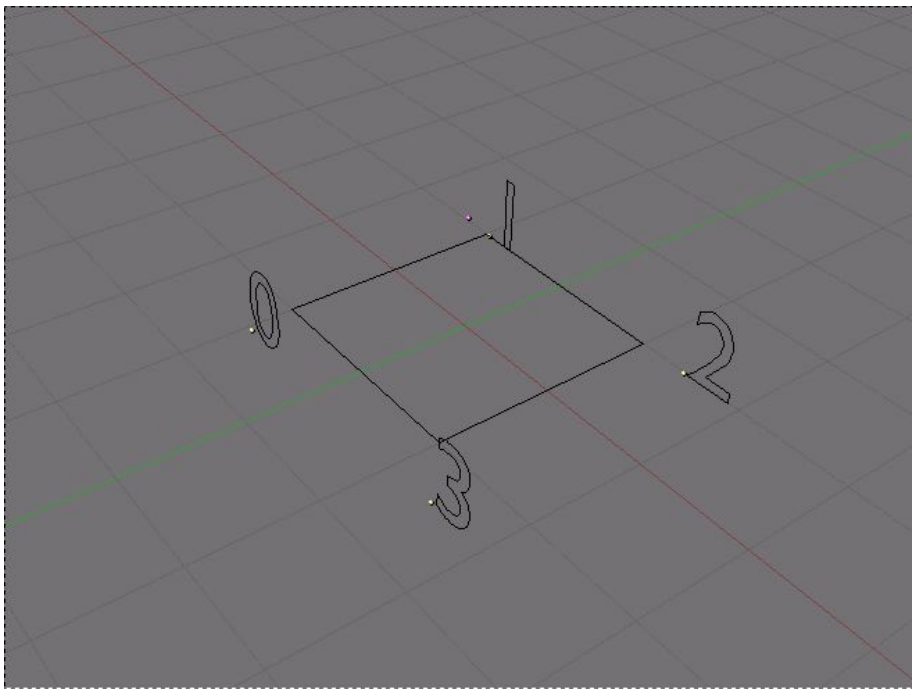

Figure 5: notre face est constituée de 4 sommets numérotés de 0 à 3

Nous pourrions écrire le code suivant. Les deux premières lignes permettent d'initialiser python et le support des modules appropriés de Blender. En l'occurence, comme nous allons pour l'instant nous contenter exclusivement de la génération d'un maillage, nous ne faisons figurer que Nmesh.

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

Par la ligne suivante, nous créons un objet de type maillage en mémoire, mais pour l'instant ils 'agit d'un objet totalement vide. Nous venons en quelque sorte de réserver de l'espace mémoire pour lui, sous le nom de **plandata**. Evitez l'usage de points au milieu des noms que vous spécifiez, si vous voulez éviter des bogues inutiles de python.

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

Nous définissons maintenant les coordonnées des quatre points qui constitueront notre face. **sommet** est un nom arbitraire que nous choisissons, et les coordonnées du vecteur proposé (**Nmesh.Vert**) sont lues sur la figure 5. Sur la ligne suivante, nous demandons à python d'ajouter (**append**) à la liste interne du maillage **plandata** les coordonnées de **sommet** dans la liste des sommets (**Nmesh.Vert**). Nous répéterons ces deux lignes de code pour chaque sommet de notre face, en ne faisant varier que les coordonnées. Dans sa structure interne, les coordonnées du premier sommet sont rangées dans la variable **verts[0]**, celles du second sommet dans **verts[1]** et ainsi de suite jusqu'au nième sommet rangé dans **verts[n-1]**.

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

Nous réservons maintenant l'espace mémoire pour la facette, sous le très simple nom de **face**, grâce à la ligne suivante:

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

Nous pouvons maintenant définir les sommets qui constituent la facette. Cela se fait simplement en ajoutant (**append**) **verts[0]** à **verts[3]** à la liste interne des sommets (**v**) constituant la facette **face** de notre maillage **plandata**. Dans les lignes qui suivent, seuls **face** et **plandata** sont des noms fixés par l'utilisateur, le reste fait référence à des fonctions ou des variables internes au module python de Blender.

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

```
16: face.v.append(plandata.verts[3])
```

Les sommets construisant notre facette sont désormais définies. Il ne nous reste plus qu'à ajouter (**append**) notre facette **face** à la liste des faces de notre maillage **plandata**. A nouveau, seuls **face** et **plandata** sont des noms fixés par l'utilisateur.

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

Ca y est, le maillage est totalement défini et constitué. Il n'existe cependant que de façon virtuelle, dans la mémoire de votre ordinateur. Nous allons donner accès à Blender à sa géométrie grâce à la ligne suivante, où **plandata** est le nom que nous avons donné au maillage en mémoire, `Plan` est le nom que porte l'objet dans Blender (on retrouve ce nom dans le menu **Editing** (touche F9) dans les champs **ME:** et **OB:**, lorsqu'après exécution du script, la face créée est sélectionnée dans la fenêtre 3D interactive).

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

Blender a maintenant bien pris en compte le maillage, il ne lui reste plus qu'à rafraîchir son affichage pour que l'utilisateur soit visuellement informé du succès de son script python.

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

3.2 De l'usage des boucles

Non content d'insulter amicalement mes amis informaticiens en disant qu'ils ont une façon de compter un peu spéciale, j'enfoncerai un peu plus le clou en ajoutant qu'ils sont un peu fainéant. Vous pouvez en effet compter sur un informaticien pour inventer un outil qui fera à sa place les opérations fastidieuses et répétitives. Du coup, si j'avais à décrire un maillage constitué de plusieurs centaines de faces, je n'aurai certainement pas envie d'enchaîner *ad nauseam* les **[nom.sommet]=NMesh.Vert([coordonées]), [nom.data].verts.append[nom.sommet], [nom.face].v.append([nom.data].verts[numéro.sommet])**. Je serais ravi d'avoir un outil le faisant à ma place. Par chance, mes amis informaticiens ont inventé les boucles.

Nous notons en effet qu'il est possible de mettre en place un boucle qui répètera autant de fois que nécessaire la génération des sommets. Pour cela, nous allons créer un vecteur qui contiendra les sommets à générer.

```
# Définition des sommets
liste_des_sommets=[
    [-1,-1,0],              # sommet 0
    [-1,+1,0],              # sommet 1
    [+1,+1,0],              # sommet 2
    [+1,-1,0]               # sommet 3
]
```

Puis au moment de créer les sommets, plutôt que de le faire un par un, nous demandons à python de parcourir le vecteur **liste_des_sommets** que nous avons définis, et pour chaque liste, de récupérer les valeurs notées pour les placer dans des variables nommées **composante[i]**. Nous pouvons ainsi les récupérer au sein d'une ligne permettant la génération du sommet. La ligne suivante, bien sûr, permet d'ajouter le sommet créé à la liste des sommets déjà créés pour l'objet **plandata**.

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

> **Attention**
>
> Les choses importantes à retenir sont:
>
> -ne pas oublier le **:** à la fin de la ligne **for... in...** ; en effet, il indique le début de la boucle.
>
> -conservez une indentation homogène jusqu'à la fin de la boucle; en particulier, il est formellement interdit de mélanger des espaces et des tabulations d'une ligne à l'autre, même si le résultat visuel semble être le même: python ne l'interprétera pas correctement et retournera une erreur!
>
> -vous pouvez imbriquer plusieurs boucles les unes dans les autres, et ce sera

l'indentation qui permettra à python d'interpréter correctement vos souhaits; par exemple, la première boucle sera décrite après une tabulation, la seconde, imbriquée dans la seconde, commencera après deux tabulations. Si une partie de la deuxième boucle est présentée après une tabulation seulement, python l'interprétera donc comme faisant partie de la première boucle.

3.3 Création d'un Cube

Nous savons maintenant définir une liste de sommets, et utiliser une boucle pour en automatiser la génération. Il ne nous reste plus qu'à apprendre à définir une liste de faces, et à utiliser une boucle supplémentaire pour, également, en automatiser la génération.

Observons le cube de la scène par défaut, et attachons-nous à reproduire les coordonnées de ses points. D'après la figure 6, nous pouvons définir les sommets suivants:

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

puis les faces suivantes, comme étant constituées par les numéros de sommets indiqués entre crochets:

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

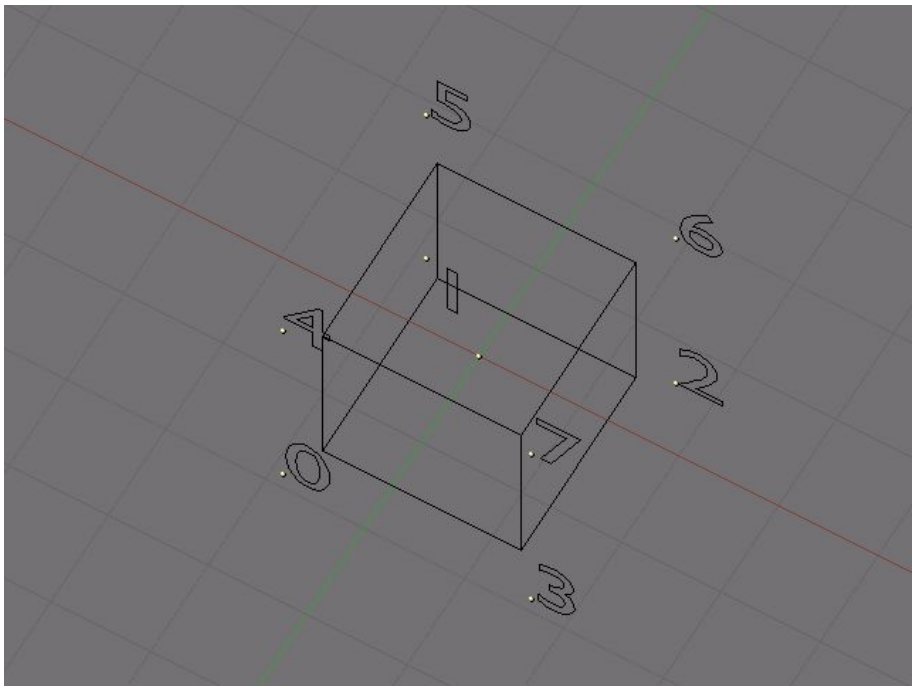

Figure 6: notre cube est constitué de 8 sommets, numérotés de 0 à 7

Nous pouvons envisager la création deux vecteurs pour résumer ces informations: **`liste_des_sommets`** contiendra les coordonnées de tous nos sommets, et **`liste_des_faces`** la constitution des faces à partir des sommets:

```
# Définition des sommets
liste_des_sommets=[
    [-1,-1,-1],             # sommet 0
```

```
    [-1,+1,-1],             # sommet 1
    [+1,+1,-1],             # sommet 2
    [+1,-1,-1],             # sommet 3
    [-1,-1,+1],             # sommet 4
    [-1,+1,+1],             # sommet 5
    [+1,+1,+1],             # sommet 6
    [+1,-1,+1]              # sommet 7
]
# Définition des faces
liste_des_faces=[
    [0,1,2,3],            # face 0
    [4,5,6,7],            # face 1
    [0,4,7,3],            # face 2
    [1,2,6,5],            # face 3
    [0,1,5,4],            # face 4
    [3,7,6,2]             # face 5
]
```

Comme précédemment, nous allons créer un maillage, pour l'instant vide, dans la mémoire de l'ordinateur. Nous l'appellerons `CubeMeshData` (sans point ou caractère spéciaux) pour signifier qu'il s'agit du maillage virtuel d'un cube.

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

Et maintenant, nous allons mettre en oeuvre la magie des boucles. Pour chaque sommet de la `liste_des_sommets`, nous allons créer un nouveau *vertex* en utilisant le vecteur (les coordonnées) indiquant la position du sommet courant. Une fois ceci fait, nous pouvons ajouter ce *vertex* à la liste des *vertice* du maillage.

```
for composante in liste_des_sommets:
        sommet=NMesh.Vert(composante[0], composante[1], composante[2])
        CubeMeshData.verts.append(sommet)
```

Nous allons également mettre en place une boucle pour générer les faces. La première boucle nous permet de créer une face, pour chaque face énumérée dans la `liste_des_faces`; cette face restera en mémoire, telle une donnée vide, jusqu'à ce que nous lui ajoutions (`append`) des *vertice*. C'est justement le rôle de la seconde boucle, incluse dans la première: elle va, pour chaque *vertex* constituant la face courante, ajouter (`append`) le *vertex* à la liste interne de la face courante. Nous quittons alors la boucle interne pour revenir à la boucle principale, où nous ajoutons (`append`) la face nouvellement décrite à la liste interne des faces constituant le maillage.

```
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)
```

Pas si simple, hein? Mais on s'y fait, et puis il ne s'agit encore, pour l'instant, que d'un petit objet avec un nombre ridiculement faible de *vertice* et de faces. Maintenant que notre maillage est parfaitement déterminé, il ne nous reste plus qu'à le transmettre à Blender, afin qu'il en fasse bon usage. C'est le but de la ligne suivante: donner à Blender la main sur le maillage CubeMeshData sous le nom de Cube. Accessoirement, le dernier argument égal à un indique tout simplement que Blender doit recalculer les normales du maillage. Cette opération est facultative, mais prenons l'habitude de la réaliser malgré tout! Cela ne peut faire de mal, à part ralentir l'exécution du script. Nous manipulerions des millions de *vertice*, je dis pas, mais pour huit cela ne devrait pas trop nous handicaper!

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

3.4 Définition d'un matériau

Pour la détermination du matériau, nous allons aller jeter un oeil dans le menu **Shading** (touche F5), et onglet **Material** des *Material buttons*, afin de nous inspirer des valeurs que nous pourrons y observer pour la création du matériau par défaut. Nous essaierons en effet ici d'en reproduire les principales valeurs (la plupart de ses valeurs sont automatiquement initialisées à la valeur par défaut; il nous est théoriquement dispensable d'en définir certaines, mais nous le ferons tout de même, à fins pédagogiques) pour l'attribuer à notre cube.

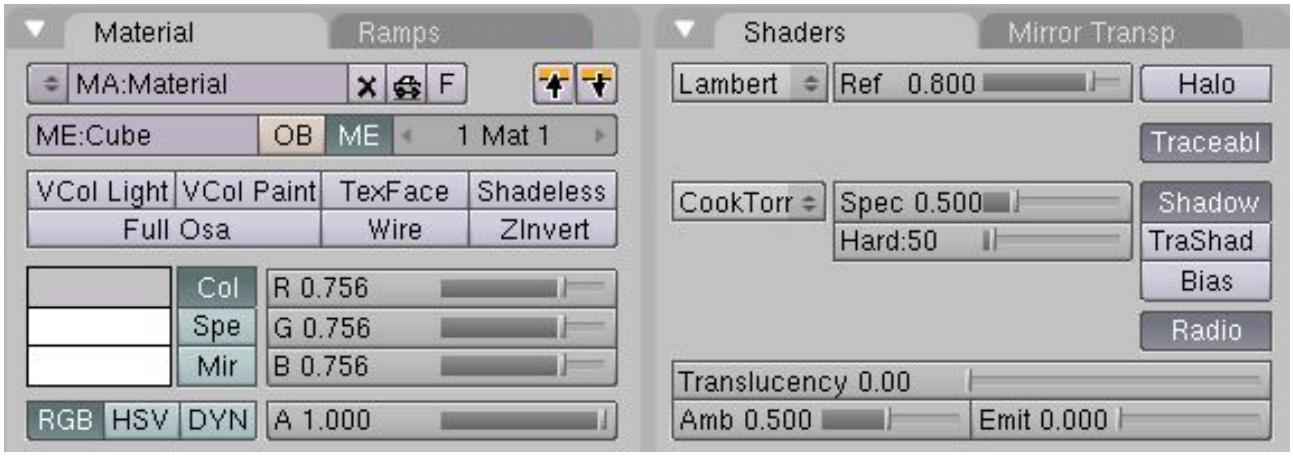

Figure 7: les propriétés du matériau par défaut du cube

Syntaxe élémentaire d'un matériau

```
[nom.data] = Material.New('MA:Blender')
[nom.objet.recevant.le.matériau].materials.append[nom.data]
[nom.objet].link([nom.data])
[nom.scene.courante].link([nom.objet])
```

La création d'un matériau, pour notre maillage, est une opération relativement simple. Il suffit, dans un premier temps, de le créer en mémoire, par exemple sous le nom de **`mat`**, puis d'utiliser la ligne suivante pour le rendre accessible depuis Blender sous le nom de **`'Material'`** (il s'agit du nom du matériau par défaut que l'on peut observer dans le Champ '**`MA:`**' de l'onglet **Material**, sur la Figure 7).

```
# Définition du matériau
mat = Material.New('Material')
```

Si le matériau est disponible, il n'est pour l'instant utilisé par aucun maillage ou objet. Nous allons pouvoir ajouter (**`append`**) le matériau 'mat' dans la liste des matériaux du maillage '**`CubeMeshData`**' (à noter que celui-ci peut en accepter jusqu'à 16 au maximum).

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

Nous allons maintenant définir certaines des propriétés du matériau, en commençant par les composantes de la couleur (R, G et B toutes égales à 0.756 pour obtenir un gris léger), l'Alpha du matériau (A 1.0 pour obtenir un cube totalement opaque), la Réflexion de la lumière par le matériau (Ref 0.8), sa Spécularité (Spec 0.5 pour des reflets spéculaires d'intensité moyenne) et enfin Hardness (la dureté des reflets spéculaires, 50 correspondant à des bords relativement doux).

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

Bien évidemment, il reste un grand nombre de paramètres que l'on peut spécifier pour contrôler, au plus proche de ses souhaits, le shader du matériau. venant de remplir notre objectif premier (l'émulation du matériau par défaut de Blender), nous nous en tiendrons toutefois là.

4. Ajout d'une caméra

Nous allons maintenant voir que l'ajout d'une caméra est une action beaucoup plus simple que la création d'un maillage, mais qu'elle recèle malgré tout quelques particularités, ce qui la rend légèrement plus complexe que la définition d'un matériau. Pour y parvenir malgré tout, il nous faut distinguer le jeu de données de l'objet « matériel », et encore, dans chaque cas, différencier le nom de l'objet en mémoire de python, et celui accessible sous Blender.

Syntaxe élémentaire de la caméra:

```
[nom.data] = Camera.New('[type]','[CA:Blender]')
Définition de la scène courante: [nom.scene.courante] = Scene.getCurrent()
[nom.objet] = Object.New('[OB:Blender]')
[nom.objet].link([nom.data])
[nom.scene.courante].link([nom.objet])
[nom.scene.courante].setCurrentCamera([nom.objet])
```

Quelques options:

`['type']` peut être `'persp'` ou `'ortho'`

`[nom.data].lens = [valeur]` permet de régler la lentille de la caméra (par défaut, 35.00)

`[nom.data].clipStart = [valeur]` permet de régler la valeur indiquant le début du champ de vision de la caméra (par défaut, 0.10)

`[nom.data].clipEnd = [valeur]` permet de régler la valeur indiquant la limite du champ de vision de la caméra (par défaut, 100.00)

Nous allons commencer par créer l'objet en mémoire en lui attribuant un nom: il s'agira tout simplement de la lettre `'c'`, pour *camera*. A l'intérieur de Blender, en revanche, le jeu de données portera le nom `'Camera'`, ce qu'il sera aisé de vérifier dans le menu **Editing** (bouton F9), dans le champ `'CA:'`. Enfin, nous souhaitons une vue en perspective classique, nous déclarons donc le type `'persp'`:

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

Dans la foulée, nous allons définir la valeur de la lentille comme étant égale à 35 (bien que nous aurions pu nous en abstenir, dans la mesure où il s'agit de la valeur par défaut):

```
c.lens = 35.0
```

Nous définissons ensuite le nom du jeu de données de la scène dans python: `'cur'` pour *current*, et nous récupérons la scène courante pour l'y stocker:

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

Nous créons maintenant physiquement l'objet caméra dans Blender. Sous python, il portera simplement le nom `'ob'` pour *object*, et `'Camera'` sous Blender. Dans le menu **Editing** (bouton F9), le nom de la caméra apparaîtra donc dans le champ `'OB:'`. En python, cela se traduit par:

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

L'objet existe désormais sous Blender, mais il s'agit d'un objet vide, jusqu'à ce que nous lui lions un jeu de données, en l'occurrence `'CA:Camera'`. Sous python, il s'agit donc de `'c'`:

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

A son tour, l'objet (`'ob'` sous python) doit être lié à la scène courante (`'cur'` sous python) pour y être pris en compte:

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

Enfin, cette option est spécifique à la déclaration de caméra par un script python: la caméra a été déclarée en tant que donnée, puis en tant qu'objet, et enfin déclarée dans une scène. Malheureusement, une scène peut contenir une infinité de caméra, et il est nécessaire de définir la caméra active. Cela se fait simplement par la ligne suivante, où nous définissons l'objet `'ob'` comme étant la caméra active de la scène courante `'cur'`:

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

Et voilà, nous avons une caméra fonctionnelle et active! Il ne nous reste plus qu'à la définir en position, en rotation et en échelle. Il s'agit de trois transformations courantes, applicables à l'objet `'ob'` que nous venons de définir. Dans le cas présent, nous ne intéresserons qu'à la position et la rotation.

Intéressons-nous à nouveau à la scène par défaut de Blender. Dans la vue 3D, sélectionnez la caméra d'un clic droit de souris puis pressez la touche N. Une fenêtre flottante, intitulée **Transform properties** (propriétés de transformation) fait son apparition. On peut notamment y lire le nom `'OB:'` de l'objet, ainsi que ses propriétés de position, de rotation et d'échelle.

Figure 8: les propriétés de transformation de la caméra de la scène par défaut

Syntaxe élémentaire des transformations:

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

setEuler permet de déterminer les angles de rotation de l'objet dans son repère local (autour de son point pivot). **setLocation** permet de définir sa position dans le repère global (l'origine absolue de la scène). Enfin, **setSize** permet de définir les proportions de l'objet dans les trois directions de son repère local. A noter que **angX**, **angY** et **angZ** doivent être des angles exprimés en radians.

Le nom de l'objet caméra, dans python, est tout simplement **'ob'**, comme nous l'avons défini précédemment. Pour reproduire la sortie du panneau flottant **Transform properties**, nous devons donc écrire une ligne de type:

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

L'idée est la même pour définir les rotations de l'objet. Malheureusement, l'angle, à l'intérieur de python, doit être exprimé en radian, tandis qu'il est exprimé en degrés dans Blender. Qu'à cela ne tienne, il suffit de convertir les degrés en gradians. Si vous ne le savez pas, 360 degrés égalent 2π radians (π = 3.14159...). Il en résulte que 1 degré = (2π/360). Les angles à appliquer sont donc:

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

Dans python, π s'écrit **math.pi**, et nécessite l'importation d'un module supplémentaire: **math**. Par fainéantise et pour épargner nos doigts fatigués par l'usage intensif du clavier, nous définirons:

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

ce qui nous permet d'écrire la ligne de transformation suivant:

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

Le code relatif à la création de la caméra peut donc être résumé aux lignes suivantes:

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

tandis que la première ligne du script devient:

```
import Blender, math
```

5. Ajout d'une Lampe

Si vous avez bien compris l'ajout d'une caméra via un script python, l'ajout d'une lampe ne vous posera aucun problème. La seule complexité pourra provenir de la diversité des lampes qui existent, chacune avec des options différentes. Nous ne chercherons pas ici à être exhaustifs, juste à être démonstratifs sur la méthode et les moyens.

Syntaxe élémentaire de la lampe

```
[nom.data] = Lamp.New('[type]','LA:Blender')
[nom.objet] = Object.New('OB:Blender')
[nom.objet].link([nom.data])
[nom.scene.courante].link([nom.objet])
```

Quelques options:

`['type']` peut être `'Lamp'`, `'Sun'`, `'Spot'`, `'Hemi'`, `'Area'`, ou `'Photon'`

Si vous choisissez le `'type' 'Spot'`, vous pouvez régler des options supplémentaires. Par exemple, la ligne:

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

spécifie un projecteur de section carrée (`'square'`) avec l'option de génération des ombres (`'shadow'`) activée.

Comme précédemment, nous allons commencer par nommer l'objet sous python, de façon à le créer, bien que vide, en mémoire; nous choisissons simplement `'l'` pour *lamp*. A l'intérieur de Blender, le jeu de données portera le nom `'Lamp'`, ce qui sera aisé de vérifier dans le menu **Editing** (bouton F9), dans le champ '`LA:`':

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

La création 'physique' de l'objet lampe est triviale. Sous python, il portera simplement le nom `'ob'` pour *object*, et `'Lampe'` sous Blender. Dans le menu **Editing** (bouton F9), le nom de la lampe apparaîtra donc dans le champ `'OB:'`. En python, cela se traduit par:

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

L'objet `'ob'` existe désormais sous Blender, mais comme précédemment, il s'agit d'un objet vide, jusqu'à ce que nous lui lions un jeu de données, en l'occurence `'LA:Lamp'`. Sous python, il s'agit donc de `'l'`:

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

A son tour, l'objet `'ob'` doit maintenant être lié à la scène courante (`'cur'` sous python) pour y être pris en compte. La scène courante ayant déjà été définie précédemment, nous pouvons cette fois nous contenter d'un simple:

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

Notre lampe est prête, il ne nous reste plus qu'à la placer convenablement. En retournant dans la scène par défaut de Blender, nous sélectionnons maintenant la lampe d'un clic droit de la souris. Normalement, le panneau **Transform properties** est toujours ouvert; si ce n'est pas le cas, pressez à nouveau la touche N. On retrouve pour la lampe les informations suivantes:

Figure 9: les propriétés de transformation de la lampe de la scène par défaut

Vous noterez un détail étonnant: la lampe par défaut est un simple point immatériel de l'espace qui sert d'origine à la lumière. Il n'est donc rigoureusement pas nécessaire de lui attribuer des propriétés de rotation. C'est pourtant ce qui a été fait dans la scène par défaut! Il ne faut rien y voir d'exceptionnel, juste que s'il vous prenait l'envie de transformer la lampe en spot lumineux, le cône serait déjà orienté avec un angle par défaut suffisant pour éclairer le cube sans manipulation supplémentaire de votre part.

Même si nous pourrions nous en passer dans cet exercice, nous allons scrupuleusement respecter les valeurs affichées dans le panneau **Transform properties**.

Comme précédemment, le nom de l'objet lampe, sous python, est un simple `'ob'`. Nous allons d'abord chercher à reproduire les composantes de localisation spécifiée dans le panneau. Nous allons donc écrire une ligne de type:

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

La définition de la rotation de la lampe est également très simple, maintenant que nous avons pris le coup:

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

Résumons le bloc de code qui correspond à la création de la lampe:

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

6. Résumé du code

Nous reprenons ici le code complet que nous avons vu jusqu'à présent, de façon synthétique et en les ordonnant. Vous le retrouverez également sur le cédérom d'accompagnement du magazine ou sur **http://www.linuxgraphic.org**, sous le nom `blender-default-scene.py` et `blender-default-scene.blend`.

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

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

7. Conclusions

Nous voilà arrivés au terme de ce premier article. Nous avons vu des choses moins évoluées que celles proposées dans l'article d'Yves Bailly, mais nous avons en revanche eu l'occasion d'aborder avec un peu plus de profondeur certaines notions, et d'expliciter la synthaxe élémentaire pour l'obtention dans Blender de certains types d'objets. Il apparaît évident, si vous avez suivi la série d'articles sur POV-ray, que le couple Blender + Python est un peu plus difficile à programmer, car il doit en permanence faire appel à des fonctions internes à Blender, traiter les résultats, et les retourner à Blender. Cela n'allège pas les programmes, ni n'améliore de façon fondamentale les temps de traitement. En revanche, les avantages sont significatifs puisqu'ils permettent de puiser

dans un logiciel très riche un en possibilités.

Enfin, comme nous avons pu l'entr'apercevoir, la création d'un maillage est une opération complexe, et dans un futur proche, nous nous y arrêterons un peu plus longuement. La gestion face à face ne facilite en effet pas grandement les choses, et la création d'objets complexes peut devenir vite infernale.

Je vous laisse découvrir les quelques liens qui suivent. Ils vous conduiront vers des ressources diverses sur la programmation Python pour Blender. Je tiens en particulier à souligner l'excellent site de JM Soler, considéré comme étant l'une des références majeures du scripting python pour Blender, tant auprès des communautés d'utilisateurs francophones qu'anglophones. Son site présente en particulier des didacticiels sur la création de maillages complexes; nous aurons également l'occasion de revenir sur le sujet.

Avec Python, c'est un nouveau monde qui s'ouvre à vous, et il est loin d'être hors de portée.

Liens

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

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

La documentation officiel 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**

Le Site de JM Soler: **http://jmsoler.free.fr/didacticiel/blender/tutor/index.htm**