Exploration des modules Material et Texture de l'API Python de Blender

Merci à Diamond Editions pour son aimable autorisation pour la mise en ligne de cet article, initialement publié dans Linux Magazine N°75

Olivier Saraja - olivier.saraja@linuxgraphic.org

Dans cet article, nous continuons l'exploration de l'API Python de Blender, en nous attachant particulièrement aux sous-modules Material et Texture et leur usage au quotidien. Nous verrons donc, principalement, comment créer des matériaux et bien sûr des textures procédurales, puis nous verrons comment associer des indices matériau à vos objets.

Pour bien comprendre la gestions des matériaux via Python, il est nécessaire de bien comprendre celle, plus directe, de Blender même. En effet, Blender permet d'associer à un objet jusqu'à seize matériaux différents, chacun repéré par un indice matériau distinct. Il convient alors de déclarer pour chaque indice de matériau la liste des faces concernées, mais nous y reviendrons beaucoup plus tardivement dans l'article. En outre, il est possible de déterminer, en plus de ses caractéristique propres, jusqu'à dix textures différentes pour un matériau donné. Chaque texture peut alors être associée à un (ou plusieurs!) canaux de texture parmi les treize disponibles.

Compliqué? Pas tant que cela, mais pas forcément facile à mettre en oeuvre avec Python. Cet article vise donc à nous donner les bases de la création de matériaux avec l'API Python de Blender. Pour nous y aider, nous effectuerons souvent le parallèle entre une scène créée dans Blender, et la même décrite en Python.

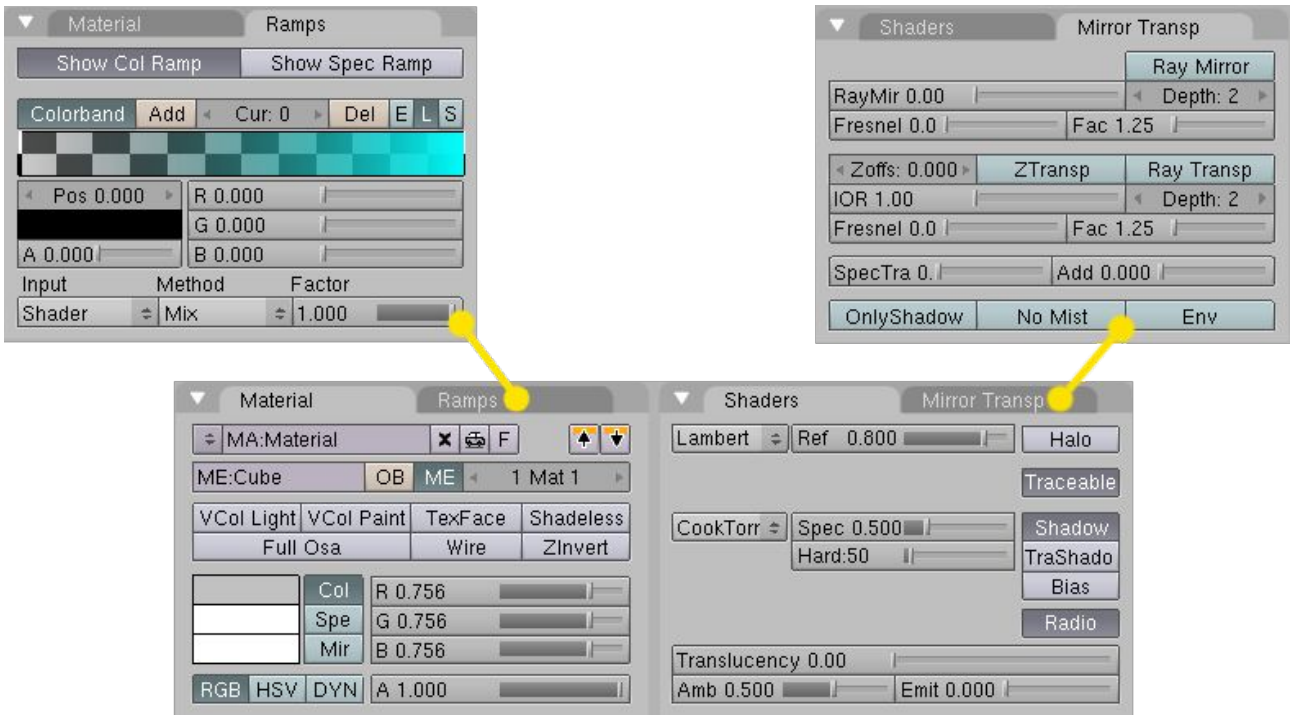

Figure 1: vue éclatée des différents panneaux d'options associées aux matériaux de Blender (hors options des textures)

1. Création d'un nouveau matériau

La définition d'un matériau via l'API Python de Blender est une opération très simple à réaliser, à condition de respecter quelques étapes essentielles. Tout d'abord, il vous sera nécessaire d'importer le module **`Material`** dans Python. Comme nous l'avons déjà vu dans les précédents articles, la première ligne de notre code va consister à importer dans Python le module **`Blender`**,

par la ligne suivante:

```
import Blender
```

la seconde consistant à importer depuis le module **Blender** les sous-modules souhaités, en l'occurrence **Material**:

```
from Blender import Material
```

La création d'un nouveau matériau vierge (ou plutôt avec toutes les propriétés par défaut) se fait simplement grâce à la définition d'une commande du type **Material.New()**. Cette variable sera identifiée sous python sous un nom unique, et le nouveau matériau se voit également attribuer un nom unique.

> **Syntaxe élémentaire de la création d'un nouveau Materiau:**
>
> **[nom variable matériau] = Material.New('[Nom Blender]')**
>
> **[nom variable matériau]**: il s'agit du nom de la variable python correspondant au nouveau matériau. Evitez l'usage du point **'.'** ou des caractères spéciaux au risque de rencontrer des erreurs lors de l'exécution de votre script
>
> **[nom Blender]**: il s'agit du nom sous lequel le matériau apparaîtra dans Blender

Bien sûr, vous ne souhaitez certainement pas créer via l'API Python toujours le même matériau de base, vous souhaitez pouvoir le définir finement. Nous allons voir ci-après comment spécifier, en observant les options de Blender, onglet par onglet, la plupart des options les plus intéressantes.

Supposons que nous venons de créer un nouveau matériau, simplement nommé 'Material ', à l'aide de la ligne suivante:

```
mat = Material.New('Material')
```

Nous allons maintenant explorer les options possibles, mais il sera utile de se rappeler que dans ce qui suit, mat est le nom d'une variable librement définie par l'utilisateur, et non une fonction au nom imposé par l'API.

> **Remarque**
>
> Dans l'usage des méthodes qui vont suivre, vous avez deux moyens à votre disposition pour déclarer des valeurs. La première est de type **mat.[propriété Blender] = [valeur]**, la seconde de type **mat.set[propriété Blender]([valeur])**. Par exemple:
>
> **mat.ref = 1.0** et **mat.spec = 1.55**
>
> sont strictement équivalent à:
>
> **mat.setRef(0.95)** et **mat.setSpec(1.55)**
>
> A noter que dans les deux cas, la méthode citée après « **mat.** » commence toujours par une minuscule.

1.1 Les options de l'onglet Material

Une petite vue de l'onglet en question n'est pas superflue, la figure 2 est là pour nous aider à visualiser ce qui est possible sous Blender, et de voir comment arriver au même résultat avec Python.

*Figure 2: les options de l'onglet **Material** des boutons **Shading** (F5) de Blender*

L'une des opérations les plus courantes consiste à définir la couleur de base du matériau. Cela se fait par la définition des trois composantes que sont le rouge (R, *Red*), le vert (G, *Green*) et le bleu (B, *Blue*). Blender tolère l'usage de d'autres méthodes de codage de couleur (HSV et DYN, pour être précis) mais l'API Python ne permet que d'accéder au codage RGB, au moyen de la syntaxe suivante, pou définir les couleurs de base du matériau (**Col**), de ses reflets spéculaires (**Spe**) et de ses reflets (**Mir**), boutons observables sur la figure 2:

Syntaxe élémentaire des options du Material:

`[nom variable matériau].setRGBCol([composante rouge],[composante vert], [composante bleu])` pour la détermination de la couleur de base du matériau (Col)
`[nom variable matériau].setSpecCol([composante rouge],[composante vert], [composante bleu])` pour la détermination de la couleur des tâches spéculaires (Spe)
`[nom variable matériau].setMirCol([composante rouge],[composante vert], [composante bleu])` pour la détermination de la couleur des reflets (Mir)

Il est également possible de définir la composante Alpha (**A**) de l'objet, c'est à dire celle qui va permettre de gérer sa transparence.

Syntaxe élémentaire des options du Material (suite):

`[nom variable matériau].setAlpha([valeur])` pour la détermination de la transparence (A)

Par exemple, le bloc de lignes suivantes vont permettre de créer un matériau de couleur dorée; malheureusement, il restera de nombreux autres paramètres à définir avant d'obtenir un résultat visuellement attrayant, mais c'est un bon début:

```
mat.setRGBCol([1.0,1.0,0.4)
mat.setSpecCol([1.0,1.0,0.8)
mat.setMirCol([0.8,0.9,0.7)
mat.setAlpha(1.0)
```

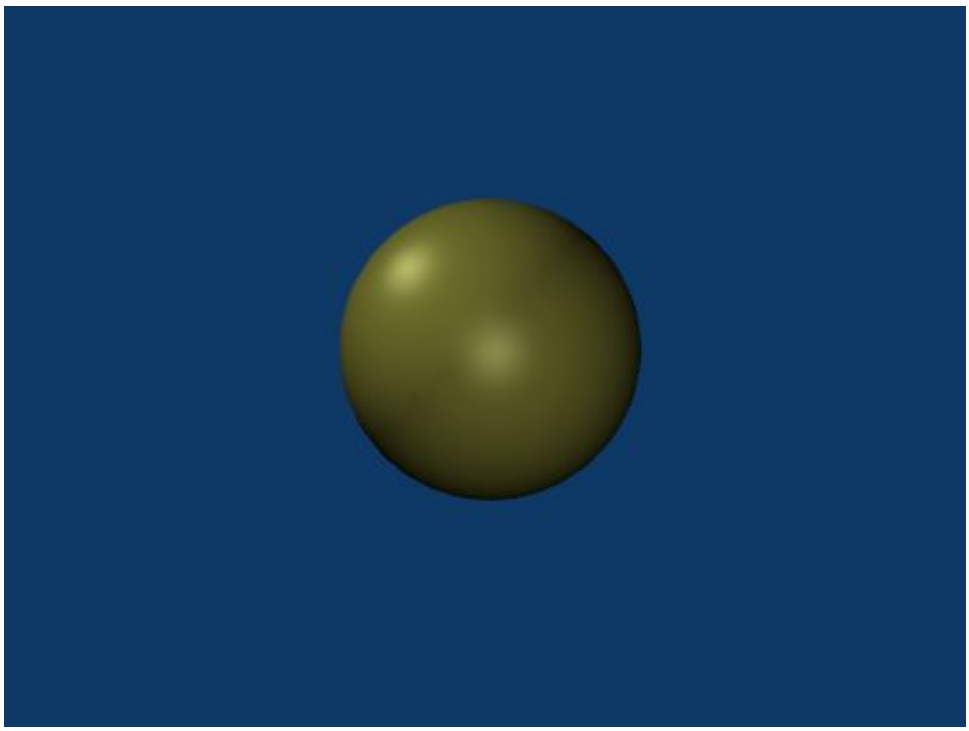

Fig 3: la couleur de base de notre or est là, mais il manque encore tellement de détail pour le rendre crédible!

Il y a également, dans cet onglet, des boutons poussoirs que nous pourrions avoir besoin d'activer ou de désactiver au rythme de nos expérimentations (en particulier plus tardivement, lorsque nous nous intéresserons au *vertex painting*, ou, en français: la peinture sur sommets). Ce sont les boutons **VCol Light**, **VCol Paint**, **TexFace**, **Shadeless**, **Wire**, et **Zinvert**.

Pour les activer, il suffit de spécifier pour le matériau un mode d'affichage, par l'usage très simple de la commande **`setMode()`** et en spécifiant en guise d'argument le mode qui nous intéresse.

> **Syntaxe élémentaire de la définition d'un Mode:**
> **`setMode('[mode1]','[mode2]',...)`**
>
> Relativement à cet onglet, il est possible d'employer les arguments suivants pour activer les options correspondantes (d'autres sont disponibles avec le même code, mais nous y reviendrons lorsque nous explorerons les onglets appropriés): '**`Shadeless`**', '**`Wire`**', '**`VColLight`**', '**`VColPaint`**', '**`Zinvert`**', '**`TexFace`**'.
>
> Il est possible de passer à la commande **`setMode()`** plusieurs arguments, séparés par des virgules. Vous avez d'ailleurs tout intérêt à procéder ainsi, puisque tous les modes non spécifiés seront considérés désactivés. Par conséquent, on peut facilement imaginer que la même commande, sans aucun argument de mode précisé, sert à désactiver par défaut toutes les options. Enfin, ultime remarque, pour préciser un mode, respectez rigoureusement la casse de l'argument (par exemple, **`VColLight`** et pas **`Vcollight`**, par exemple) et n'oubliez pas d'encadrer chaque chaîne de caractères par des apostrophes simples: `'`.

Ainsi, si nous ajoutons la ligne suivante à notre bout de code précédent, nous obtenons une jolie structure en fil de fer doré sur lequel aucune ombre ne se pose:

```
mat.setMode('Wire','Shadeless')
```

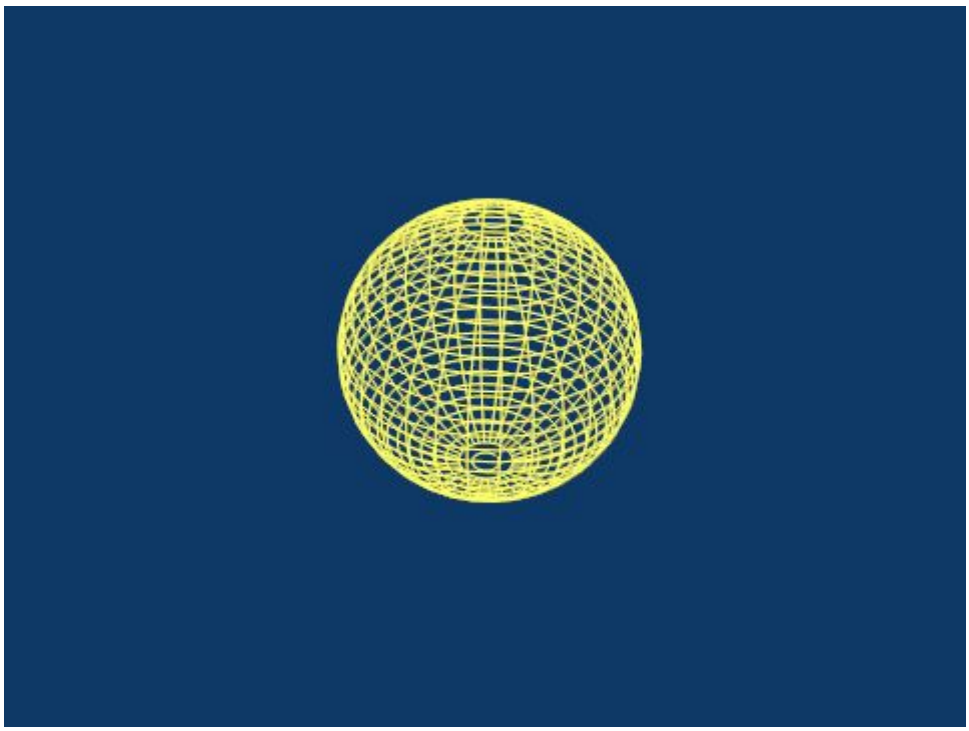

Figure 4: le même matériau, avec les modes Wire et Shadeless spécifiés

1.2 Les options de l'onglet Shaders

Les options découvertes jusqu'à présent nous permettent de définir la couleur de base de notre matériau, mais pas la façon dont il interagit avec l'éclairage ambiant. Pour l'essentiel, c'est l'objectif de cet onglet de Blender: définir les reflets lumineux à la surface des objets pour suggérer une finition (un objet mat ou brillant, avec des reflets durs ou brillants, etc.).

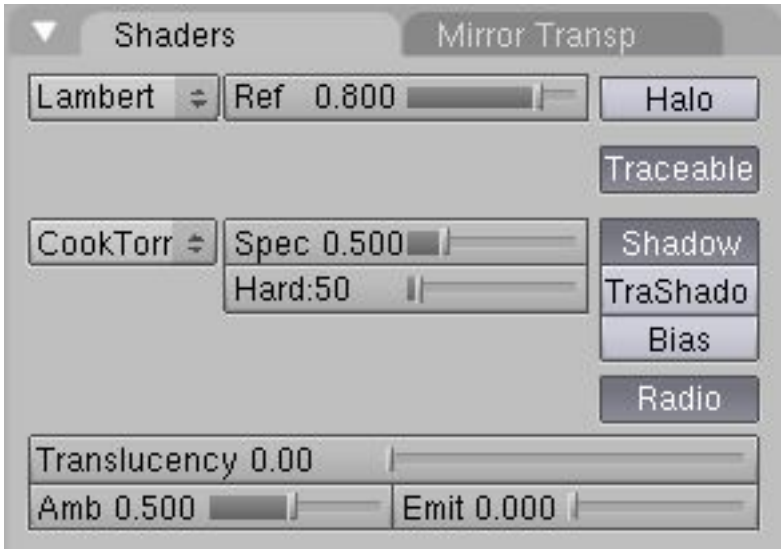

Figure 5: les options de l'onglet Shaders au grand complet

A noter que Blender propose l'usage de plusieurs algorithmes pour les *shaders* de diffusion (*diffuse shaders*: Minnaert, Toon, Oren-Nayar, Lambert) et de spéculaires (*specular shaders*: WardIso, Toon, Blinn, Phong, CookTorr) mais malheureusement seuls ceux par défaut sont manipulables au-travers de l'API Python: Lambert pour le *diffuse shader* et CookTorr pour le *specular shader*. Il est dommage que l'accès aux différents algorithmes soit ainsi (pour l'instant, l'API étant en constante évolution) limité mais cela simplifie considérablement la description des méthodes de l'API relatives à cet onglet, chaque *shader* pouvant avoir ses propres paramètres supplémentaires.

Les paramètres finalement accessibles par l'API sont ceux présentés sur la Figure 5 ci-dessus. Les

trois premiers (**Ref**, **Spec** et **Hard**) suggèrent justement la finition d'un matériau, dans la façon dont celui capte la lumière: **Ref** (*Reflection*) détermine la quantité de lumière reflétée par le matériau; proche de 1, et les conditions d'illumination de la scène mises à part, l'objet sera très clair, tandis que proche de 0, il sera très sombre. **Spec** (*Specular*) détermine la taille de la tâche spéculaire; la valeur associée peut varier de 0 (pas de tâche spéculaire, convient à des matériaux totalement mats) à 2 (tâche d'intensité très supérieure, convient à des matériaux très brillants). Enfin **Hard** (*Hardness*)détermine la « dureté » de la tâche spéculaire; sa valeur peut varier de 0 (bords de la tâche extrêmement flous, convient à des matériaux poreux ou à la surface sale ou irrégulière, comme le caoutchouc) à 255 (bords extrêmement durs et nettement dessinés, convient à des surfaces très lisses, vernies ou très dures, comme le verre).

Syntaxe élémentaire des options des Shaders:

`[nom variable matériau].setRef([valeurRef])` pour la détermination de la réflection (Ref)

`[nom variable matériau].setSpec([valeurSpec])` pour la détermination du spéculaire (Spec)

`[nom variable matériau].setHardness([valeurHard])` pour la détermination de la dureté du spéculaire (Hard)

Il y a trois autres paramètres dans cet onglet que nous pourrions souhaiter donner aux matériaux créés à l'aide de Python. Il s'agit de la translucidité (**Translucency**), de la susceptibilité à la couleur ambiante (**Amb**) et de l'émittance (**Emit**). Le premier détermine la proportion de luminosité qui traverse la paroi externe du matériau, un peu comme s'il s'agissait d'une fine membrane, le tissu d'un abat-jour, ou tout simplement celui d'une robe (jamais entendu parlé du soleil vénitien?). Le second permet de mêler à la couleur propre de l'objet une couleur définie comme étant celle de l'environnement (il s'agit d'une ancienne technique pour simuler, de façon très primitive il faut l'avouer, l'illumination globale). Le dernier permet de déterminer avec quelle puissance le matériau a la particularité de s'auto-illuminer même en l'absence d'éclairage (particulièrement utile dans la détermination d'une solution de radiosité). Malheureusement, de ces trois propriétés, seules deux sont couvertes par l'API Python, la translucidité n'étant à ce jour pas accessible par ce moyen.

Syntaxe élémentaire des options des Shaders (suite):

`[nom variable matériau].setAmb([valeurRef])` pour la détermination du facteur ambiant (Amb)

`[nom variable matériau].setEmit([valeurSpec])` pour la détermination de l'émittance (Emit)

Ces deux méthodes admettent des paramètres variant de 0.0 à 1.0.

Il y a également des boutons poussoirs qu'il est possible d'activer: **Halo**, **Traceable**, **Shadow**, **TraShado**, **Bias**, **Radio** grâce à la méthode **`setMode()`** comme nous l'avons vu lors de l'exploration de l'onglet Material, en spécifiant simplement, en guise d'argument, le mode qui nous intéresse: '**`Traceable`**', '**`Shadow`**', '**`Halo`**', '**`Radio`**'. Il n'y a toutefois pas d'argument permettant d'activer les boutons TraShado et Bias, l'API Python étant toujours en cours de développement.

Pour reprendre notre exemple de tout à l'heure, le bloc de lignes suivant va nous permettre de donner à notre matériau de couleur dorée une finition d'aspect plus métallique:

```
mat.setRef(0.8)
mat.setSpec(1.4)
mat.setHard(20)
```

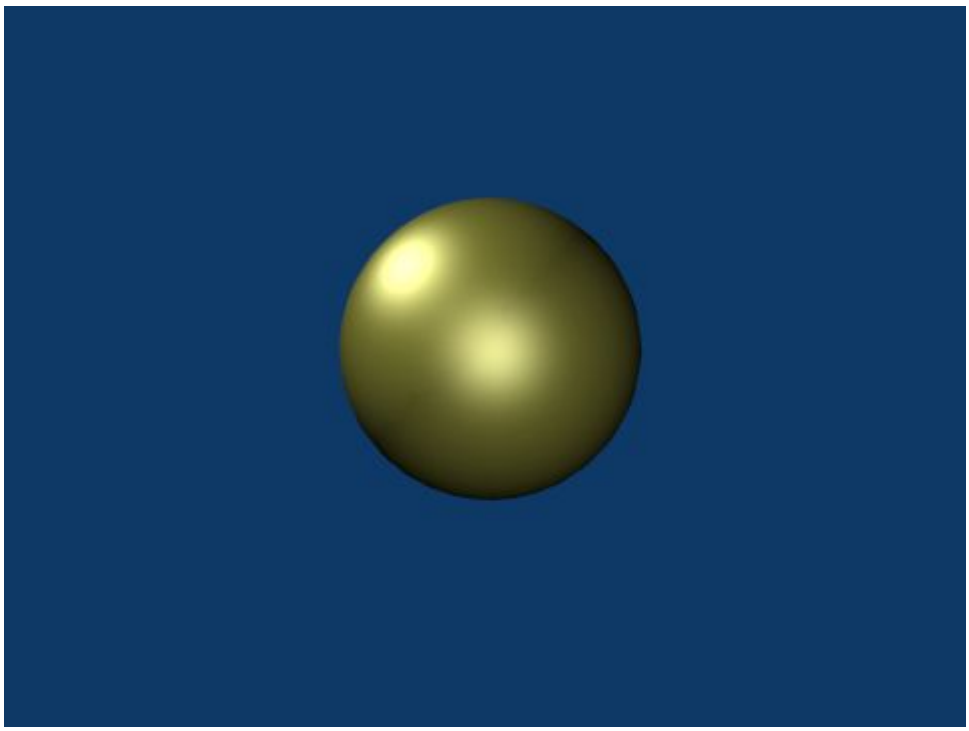

Figure 6: les tâches spéculaires sont là, ainsi que seul un éclairage suffisamment riche peut le révéler

A noter toutefois que dans Blender, lorsque vous activez le bouton **Halo**, vous accédez à de nouveaux paramètres. Ceux-ci sont également accessibles au-travers de l'API Python. L'activation du bouton poussoir **Flare** entraîne lui-même l'apparition de nouveaux paramètres, de sorte que la configuration la plus complexe est celle de la figure 7.

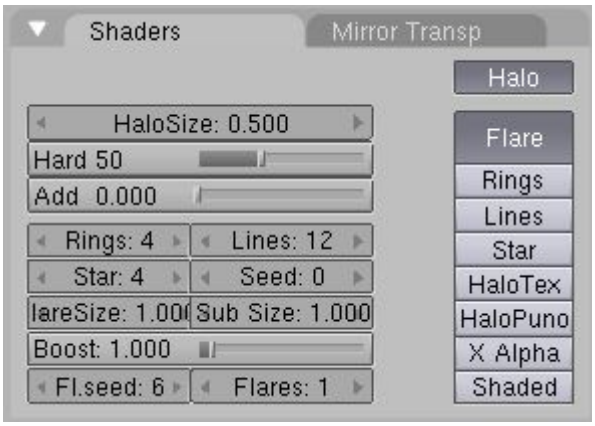

Figure 7: les options relatives aux halos

Les méthodes accessibles via l'API Python sont résumées dans l'encadré suivant. Par manque de place dans cet article, nous vous renvoyons à la documentation de Blender pour connaître leurs effets.

Syntaxe élémentaire des options des Halos:

`[nom variable matériau].setHaloSize([valeurHaloSize)` pour la détermination de la taille du Halo (**HaloSize**)

`[nom variable matériau].setAdd([valeurAdd])` pour la détermination du facteur d'éblouissement (**Add**)

`[nom variable matériau].setNRings([valeurNRings])` pour la détermination du nombre d'anneaux (**Rings**) (nécessite l'activation du mode '`Rings`')

`[nom variable matériau].setNLines([valeurNLines])` pour la détermination du nombre de lignes (**Lines**) (nécessite l'activation du mode '`Lines`')

`[nom variable matériau].setNStars([valeurNStars])` pour la détermination du nombre d'étoiles (**Star**) (nécessite l'activation du mode '`Star`')

`[nom variable matériau].setHaloSeed([valeurHaloSeed])` pour la détermination de la graine pseudo-aléatoire des halos (**Seed**)
`[nom variable matériau].setFlareSize([valeurFlareSize])` pour la détermination de la taille de l'effet lumineux (**FlareSize**) (nécessite l'activation du mode '`Flare`')
`[nom variable matériau].setSubSize([valeurSubSize])` pour la détermination de la taille des points, cercles et effets lumineux secondaires (**Sub Size**) (nécessite l'activation du mode '`Flare`')
`[nom variable matériau].setFlareBoost([valeurFlareBoost])` pour la détermination de la puissance de l'effet lumineux (**Boost**) (nécessite l'activation du mode '`Flare`')
`[nom variable matériau].setFlareSeed([valeurFlareSeed])` pour la détermination de la graine pseudo-aléatoire des effets lumineux (**Fl.seed**) (nécessite l'activation du mode '`Flare`')
`[nom variable matériau].setNFlares([valeurNFlares)` pour la détermination du nombre d'effets lumineux (**Flares**) (nécessite l'activation du mode '`Flare`')

Les boutons poussoirs qu'il est possible d'activer sont les suivants: **Halo**, **Flare**, **Rings**, **Lines**, **Star**, **HaloTex**, **HaloPuno**, **X Alpha**, **Shaded**. Comme précédemment, référez-vous à la documentation de Blender pour une description complète. Il est possible de les activer grâce à la méthode `setMode()` comme nous l'avons déjà vu à deux reprises, en spécifiant cette fois, en guise d'arguments, les modes qui nous intéressent parmi: '`Halo`', '`HaloFlare`', '`HaloRings`', '`HaloLines`', '`HaloStar`','`HaloTex`','`HaloPuno`','`HaloXAlpha`','`HaloShaded`'.

1.3 Les options de l'onglet Mirror Transp

C'est ici que se cachent les principales options ayant trait au *raytracing* dans Blender: la gestion des effets de reflet et de transparence. A noter toutefois que la transparence peut être gérée à la fois par le *raytracer* et par le *scanliner* tous deux intégrés à Blender. Il en résulte la possibilité d'utiliser soit le **Ztransp** (*scanliner*) soit le **Ray Transp** (*raytracer*). Dans le premier cas, les paramètres optionnels (**IOR**, **Depth**, **Fresnel** et **Fac**) sont totalement inutiles car non pris en compte.

Figure 8: les options relatives au raytracing

Nous allons commencer par étudier la mise en place de reflets. Cela passe dans un premier temps par la déclaration d'un mode supplémentaire, figuré dans l'onglet **Mirror Transp** par un bouton **Ray Mirror**. Pour la commande `setmode()`, il est nécessaire de lui passer, comme argument, le paramètre `'RayMirr'`. Nous avons ensuite quatre méthodes Python avec lesquelles jouer, ainsi que le précise l'encadré suivant.

Syntaxe élémentaire des options de Reflets:

`[nom variable matériau].setRayMirr([valeurRayMirr)` pour la détermination du taux de réflection (**RayMir**)
`[nom variable matériau].setRayMirrDepth([valeurRayMirrDepth])` pour la détermination de la « profondeur » de *raytracing* (**Depth**), c'est à dire le niveau de

récursivité des reflets multiples
[nom variable matériau].setFresnelMirr([valeurFresnelMirr]) pour la détermination de la puissance de l'effet Fresnel pour les reflets (**Fresnel**)
[nom variable matériau].setFresnelMirrFac([valeurFresnelMirrFac]) pour la détermination du facteur de Fresnel (**Fac**)

La mise en place de la transparence ne présente guère plus de difficulté, à part que vous avez le choix entre activer la transparence en mode *scanline* (bouton poussoir **ZTransp**) ou en mode *raytracing* (bouton poussoir **Ray Transp**). Le choix entre les deux versions se fait par l'usage de l'un ou l'autre argument avec la méthode **setMode()**: '**ZTransp**' ou '**RayTransp**'. Les méthodes python qui suivent ne sont utiles que pour la version *raytracing* de la transparence.

Syntaxe élémentaire des options de Transparence:

[nom variable matériau].setIOR([valeurIOR) pour la détermination de l'indice de réfraction du matériau (**IOR**)
[nom variable matériau].setTransDepth([valeurTransDepth]) pour la détermination de la « profondeur » de *raytracing* (**Depth**), c'est à dire le niveau de récursivité des objets transparents multiples
[nom variable matériau].setFresnelTrans([valeurFresnelTrans]) pour la détermination de la puissance de l'effet Fresnel pour la transparence (**Fresnel**)
[nom variable matériau].setFresnelTransFac([valeurFresnelTransFac]) pour la détermination du facteur de Fresnel (**Fac**)

Cet onglet propose également, optionnellement, l'usage d'une méthode supplémentaire, pour l'usage de laquelle nous vous renvoyons à la documentation:

[nom variable matériau].setSpecTransp([valeurSpecTransp) pour la détermination de la transparence spéculaire (**SpecTra**)
Enfin, trois modes supplémentaires permettent d'activer les boutons poussoirs **OnlyShadow**, **No Mist** et **Env** grâce à la méthode **setMode()** et l'un ou l'autre des arguments suivants: '**OnlyShadow**', '**NoMist**', '**Env**'.

Pour reprendre notre exemple de tout à l'heure, le bloc de lignes suivant va nous permettre de donner à notre matériau les reflets qu'il mérite pour ressembler à de l'or digne de ce nom. A noter que pour cette image finale, nous avons dupliqué les sphères faisant usage du matériau ainsi créé, et appliqué tous les modes par défaut de Blender en plus de celui activant le calcul des reflets par *raytracing*:

```
mat.setRayMirr(0.40)
mat.setMirrDepth(3)
mat.setMode('Traceable','Shadow','Radio','RayMirr')
```

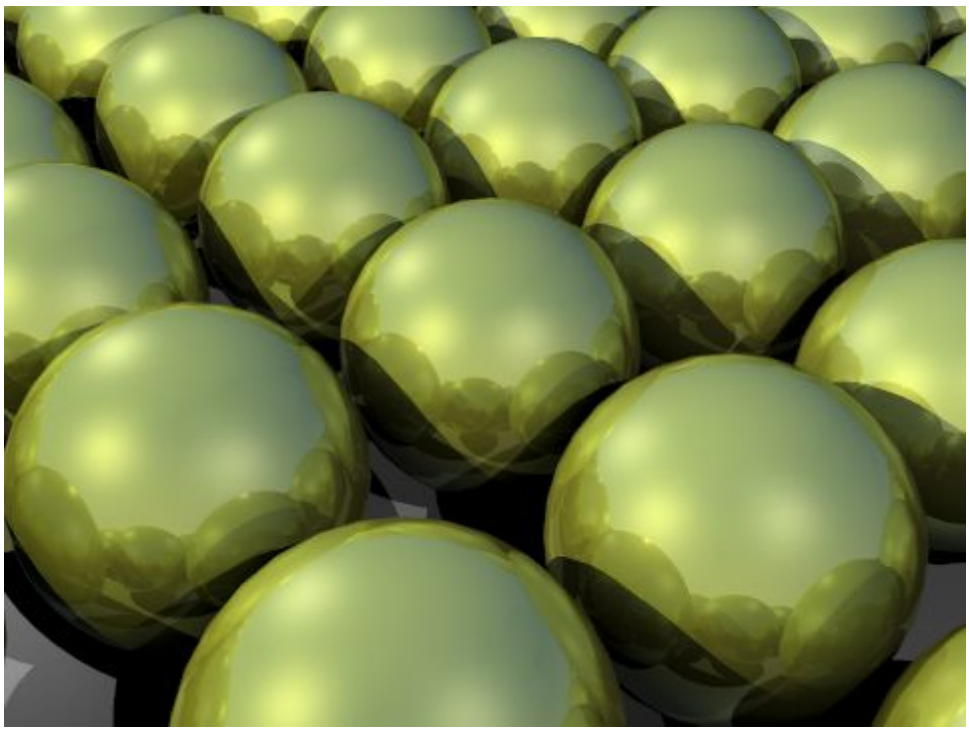

Figure 9: un élevage d'oeufs dorés ou de boules de Noël?

Attribution du nouveau matériau à un objet

Supposons que vous avez créé un maillage en faisant usage de l'API Python, comme nous l'avions vu dans Lmag #73 (article en ligne sur **http://www.linuxgraphic.org**), et que cet objet est référencé sous la variable python `pyObj` (vous pouvez bien sûr avoir spécifié tout autre nom). Supposons également que le matériau que nous créons au cours de cet article est appelé `pyMat`. Vous attribuerez le matériau `pyMat` à l'objet `pyObj` simplement à l'aide de la ligne suivante:

```
pyObj.materials.append(pyMat)
```

2. Création d'une nouvelle texture

La définition d'une texture via l'API Python n'est guère plus difficile que celle d'un matériau, par les mêmes moyens. Il faut cependant au préalable prendre la précaution d'importer depuis le module `Blender` des sous-modules supplémentaires: au minimum `Texture`, et éventuellement `Image` si vous souhaitez faire usage d'images *bitmap* pour habiller vos objets. Le résultat serait donc une ligne d'import ressemblant à:

```
from Blender import Material, Texture, Image
```

Pour créer une nouvelle texture, il suffit de faire appel à la fonction `New()` du module `Texture`, avec le nom Blender de la Texture comme argument de la fonction. Il faut ensuite déterminer le type de texture souhaité, à l'aide de la méthode `setType()`. Parfois, un type de texture admet un sous-type de texture, comme par exemple **Stucci** qui admet les sous-types **Plastic**, **Wall in** et **Wall out**. Le cas échéant, le sous-type est déterminé par l'usage de la méthode `setSType()`. Ensuite, des fonctions optionnelles, dépendant de chaque type et sous-type, doivent être spécifiées pour paramétrer finement les textures.

Syntaxe élémentaire de la création d'une nouvelle texture:

```
from Blender import Material, Texture, Image
[nom variable texture] = Texture.New('[nom Blender]')
```

[nom variable texture].setType('[Type]')

[nom variable texture]: il s'agit du nom de la variable python correspondant à la nouvelle texture. Evitez l'usage du point **'.'** ou des caractères spéciaux au risque de rencontrer des erreurs lors de l'exécution de votre script

[nom Blender]: il s'agit du nom sous lequel la texture apparaîtra dans Blender

Nous n'explorerons pas tous les types et sous-types possibles, mais nous nous attacherons au moins à décrire deux types de texture: la texture **Image**, et une texture procédurale admettant des sous-types: **Stucci**. Nous devrions ainsi explorer la quasi-totalité des options possibles offertes par les onglets restant à étudier dans les menus **Material buttons** et **Texture buttons**.

2.1 La texture Image

Il s'agit d'un grand classique lors de l'usage de Blender, intimement lié aux onglets **Map Input** et **Map To** du menu **Material buttons**. Mais nous allons voir tout cela; supposons le bout de code suivant:

```
tex = Texture.New('Texture')
tex.setType('Image')
```

Il nous faut maintenant charger une image en mémoire. Cela se réalise aisément au travers d'une méthode issue du sous-module **Image**, qu'il nous faut impérativement avoir importé au préalable:

```
from Blender import Material, Texture, Image
```

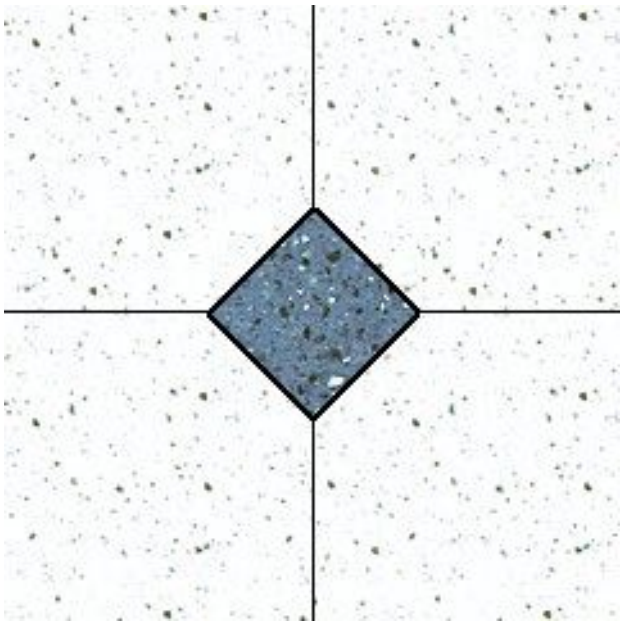

Figure 10: notre photo de carrelage

Supposons simplement qu'une image intitulée '**carrelage.png**' se trouve dans un certain répertoire, et que vous souhaitez simplement nommer '**img**' la variable image dans Python. Vous obtenez alors la ligne suivante:

```
img = Image.Load('/chemin/vers/votre/image/carrelage.png')
```

Il ne vous reste alors plus qu'à lier l'image en mémoire à la texture grâce à la commande suivante:

```
tex.image = img
```

Dès lors, il vous est alors possible de spécifier des options, exactement comme sous Blender, ainsi qu'en témoigne la Figure 11 ci-après. En particulier nous pouvons déterminer les options de l'image grâce à la méthode **`setFlags()`** pour activer certaines des boutons de l'onglet Image: **`'InterPol'`**, **`'UseAlpha'`**, **`'MipMap'`**, **`'Fields'`**, **`'Rot90'`**, **`'CalcAlpha'`**, **`'StField'`**, **`'Movie'`** et **`'Cyclic'`**. Comme nous en avons pris l'habitude avec les **`Modes`**, lors de l'exploration du sous-module **`Material`**, la casse et la présence des apostrophes simples **`'`** sont tous deux importantes. Nous aurons donc recours à une ligne de type:

```
tex.setImageFlags('InterPol','MipMap')
```

Vous noterez que toutes les options ne sont pas encore activables au travers de l'API: **NegAlpha** et **Anti** brillent pour l'instant par leur absence.

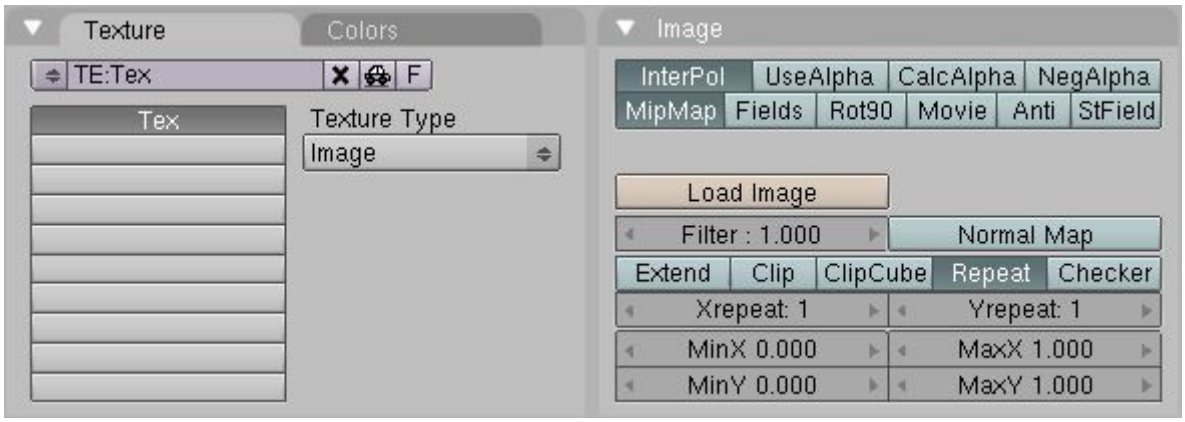

Figure 11: une texture de type Image sous Blender

Nous pouvons également définir le comportement de l'image lorsqu'elle n'a pas la taille suffisante pour recouvrir l'objet entier. Il s'agit du mode d'extension de l'image, et il nous est possible d'activer l'une ou l'autre des options suivantes: **`'Extend'`**, **`'Clip'`**, **`'ClipCube'`**, **`'Repeat'`**. Nous vous invitons à consulter la documentation pour la signification de ces différentes options; précisons seulement que l'option **`'Repeat'`** permet de répéter l'image sur l'objet en juxtaposant l'image à elle-même dans toutes les directions, tandis que l'option **`'Extend'`** « étire » les bords de l'image à l'infini pour terminer de couvrir l'objet si l'image n'est pas assez grande pour l'objet. Dans le cadre de notre exemple, nous aurons recours, avec les mêmes contraintes de casse et d'apostrophes que précédemment, à une ligne de type:

```
tex.setExtend('Repeat')
```

S'agissant d'une image, il reste à définir l'origine de la texture par rapport à l'objet (par exemple, par rapport au centre de l'objet, ou par rapport au centre de la fenêtre, etc.) ainsi que le canal de la texture affecté par l'image (par exemple la couleur, la spécularité, la transparence, etc.). Les différentes options accessibles dans Blender sont illustrées sur la figure 12 suivante.

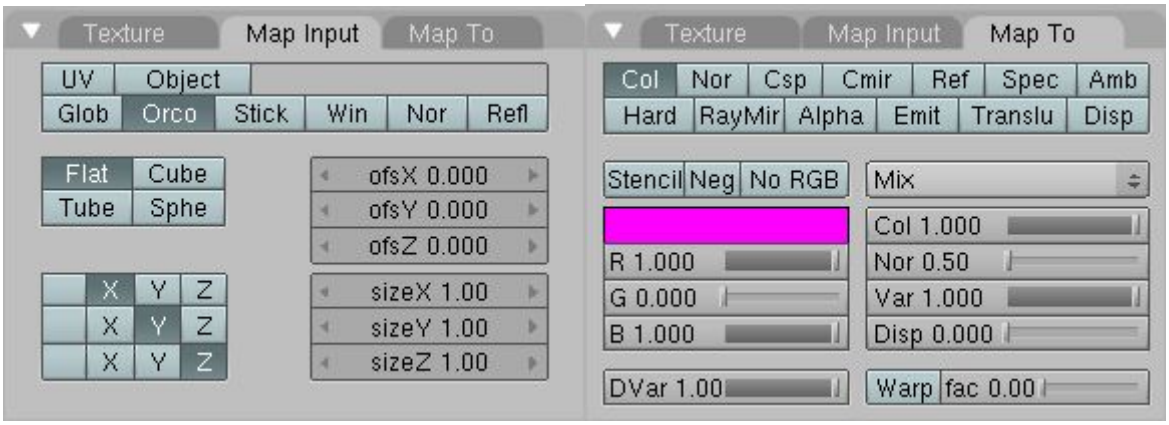

Figure 12: illustration dans Blender des options texco et mapto

Tout cela se réalisera au travers de la méthode **`setTexture()`** qui accepte jusqu'à quatre argument: le premier définit l'index de texture (la valeur peut varier de 0 à 9, sachant que Blender accepte désormais jusqu'à 10 textures au total pour un même matériau, au lieu des 8 accessibles dans le passé); le deuxième définit le nom de la texture (précédemment définie) qui est « collée » dans l'indice donné par le premier argument; le troisième détermine les

coordonnées de la texture à employer parmi `ORCO`, `REFL`, `NOR`, `GLOB`, `UV`, `OBJECT`, `WIN`, `VIEW`, et `STICK` (se reporter à la documentation pour l'usage de chacune de ces options), à faire précéder par la chaîne de caractères: '`Texture.TexCo.`'; enfin, la quatrième et dernière permet de déterminer le canal qui sera « affecté » par la texture: `COL`, `NOR`, `CSP`, `CMIR`, `REF`, `SPEC`, `HARD`, `ALPHA`, et `EMIT` (idem, reportez-vous à la documentation) à faire précéder par la chaîne de caractères '`Texture.MapTo.`'. Ainsi, pour associer la texture `tex` à l'indice 0 du matériau `mat`, avec les coordonnées de texture de type **Orco** et une application aux canaux **Col** du matériau, nous aurions une ligne de type:

```
mat.setTexture(0,tex,Texture.TexCo.ORCO, Texture.MapTo.COL)
```

Le résultat est probant, montrant qu'il est également possible de manipuler des textures de type Image au-travers de l'API Python de Blender, ainsi qu'en témoigne la figure 13 ci-après.

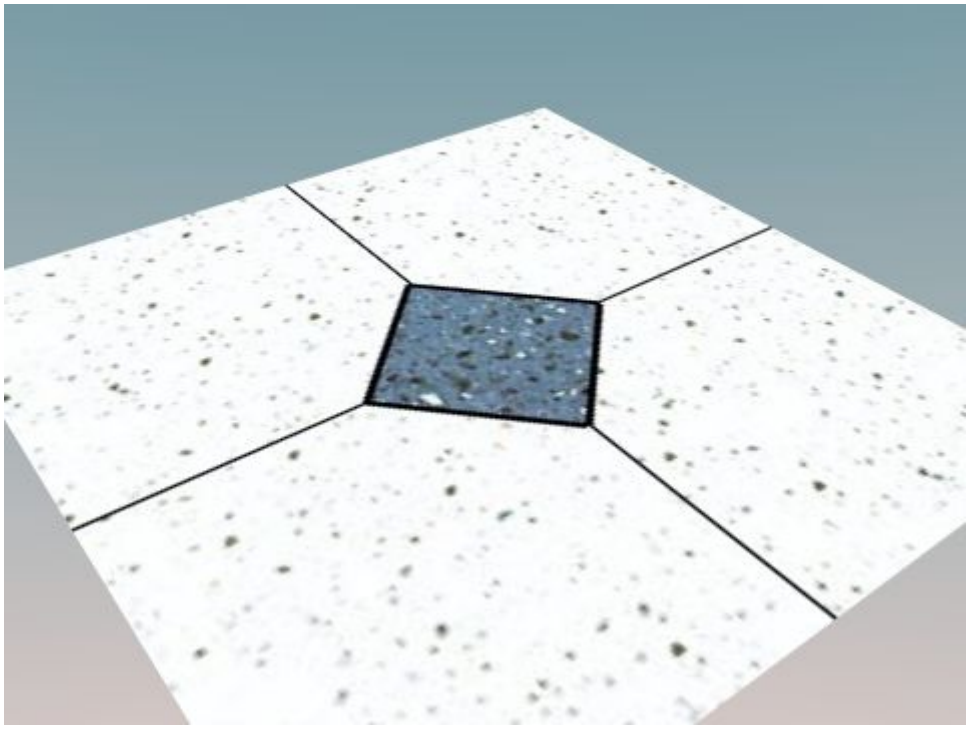

Figure 13: Python servirait donc également au carrelage de votre domicile...???

Toutefois, il est probable que vous éprouviez le besoin d'activer plusieurs canaux à la fois. Par exemple, **Spec** et **Nor**. Malheureusement, la méthode `setTexture()` n'admet en tout et pour tout qu'un seul mode `MapTo`, et active par défaut le mode **Col** si rien n'est mentionné. Il existe toutefois une astuce qui permet d'activer les canaux nominativement. Elle est pleinement explicitée sur l'incontournable site de JM Soler (voir Liens en fin d'article) et s'appuie sur la manipulation des bits. Dans notre cas, nous aurions à insérer ce bout de code pour l'activation des canaux **Nor** et **Spec**:

```
mat.setTexture(0,tex,Texture.TexCo.ORCO)
mtextures = mat.getTextures()
       for mtex in mtextures:
              if mtex!=None:
                     mtex.mapto|=Blender.Texture.MapTo['COL']
                     mtex.mapto|=Blender.Texture.MapTo['NOR']
                     mtex.mapto|=Blender.Texture.MapTo['SPEC']
```

Malheureusement, cette opération va fixer le `MapTo` de toutes les textures à la fois du matériau `mat`, y compris celles que vous ne souhaiteriez pas. JM Soler propose tout de même une solution palliative qui tend à alourdir le code, mais qui permet néanmoins de régler l'opération en attendant l'amélioration de l'API Python.

A noter que dans Blender, les canaux admettent trois états: activé, désactivé, inversé. Ce tout dernier (qui correspond à ce que vous obtenez sous Blender en cliquant deux fois consécutives sur un bouton de canal, c'est à dire le bouton enfoncé mais le texte apparaissant en jaune) n'est pas encore accessible au travers de l'API. Vous pouvez toutefois activer ou désactiver à volonté les canaux grâces aux opérations booléens: OU '| = ' pour activer, et ET INVERSE ' & = ~' pour désactiver. Par exemple, pour désactiver le canal COL qui apparaît systématiquement et activer NOR, nous aurions les quelques lignes suivantes en remplacement des précédentes:

```
mat.setTexture(0,tex,Texture.TexCo.ORCO)
mtextures = mat.getTextures()
       for mtex in mtextures:
              if mtex!=None:
                     mtex.mapto&=~Blender.Texture.MapTo['COL']
                     mtex.mapto|=Blender.Texture.MapTo['NOR']
```

Syntaxe élémentaire de la création d'une texture de type Image:

```
import Blender
from Blender import Material, Texture, Image
[nom variable image] = Image.Load('[chemin et nom complet de l'image]')
[nom variable texture].image = [nom variable image]
[nom variable texture].setImageFlags('[option1]','[option2]',...)
[nom variable texture].setExtend('[mode choisi]')
[nom variable matériau].setTexture(0,[nom variable texture],Texture.TexCo.[mode
TexCo], Texture.MapTo.[mode MapTo])
```

2.2 Création d'une texture procédurale

Nous connaissons déjà les bases de la création d'une telle texture, puisqu'elles ne diffèrent pas de celle de la texture Image. A noter toutefois que les deux seuls sous-modules nécessaires sont **`Material`** et **`Texture`**. La première différence réside dans le fait qu'il nous faut choisir un **`Type`** de texture parmi: **`'None'`**, **`'Clouds'`**, **`'Wood'`**, **`'Marble'`**, **`'Magic'`**, **`'Blend'`**, **`'Stucci'`**, **`'Noise'`**, **`'Image'`**, **`'Plugin'`** et **`'EnvMap'`**. Bien sûr, nous avons déjà exploré le cas de l'Image, nous choisirons donc, à fins d'expérimentation, de nous amuser avec la texture de type **`'Stucci'`**. Malheureusement, ainsi qu'en témoigne la Figure 14, plusieurs types de texture ne sont pour l'instant pas encore accessibles: **Distorted Noise**, **Voronoi**, et **Musgrave**.

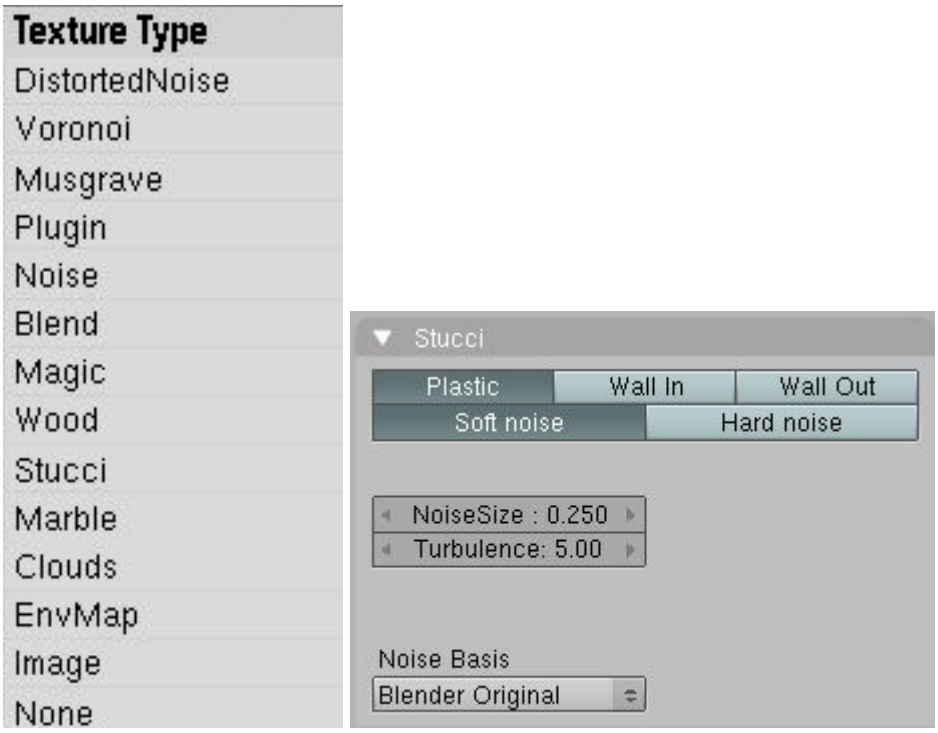

Figure 14: les types de texture accessibles depuis Blender et un zoom sur la texture Stucci

Pour créer une nouvelle texture, de type **Stucci**, nous allons commencer simplement par déclarer la nouvelle texture (grâce à la méthode **`New`** du module **`Texture`**) et ensuite définir son type grâce à la méthode **`setType()`**. Uniquement avec ces deux lignes, nous avons déjà une texture valide, avec tous les paramètres par défaut.

```
tex = Texture.New('Texture')
tex.setType('Stucci')
```

Nous pouvons maintenant affiner le paramétrage de la texture, grâce à la méthode **`setSType()`**

qui permet de définir un sous-type de texture. Pour une texture de type **Stucci**, nous pouvons accéder aux sous-types suivants: `'StucciPlastic'` pour activer l'option **Plastic**, `'StucciWallIn'` pour **Wall In**, et `'StucciWallOut'` pour **Wall Out**. Par exemple:

```
tex.setSType('StucciWallIn')
```

L'observation de la Figure 14 montre également qu'il est possible de choisir entre deux types de *Noise* (Bruit): **Soft Noise** et **Hard Noise**. Il faut pour cela passer par la variable `noiseType`. Celle-ci n'admettant pas de « variante » de forme `[var texture].set[méthode]([valeur])`, nous déclarerons donc le type de bruit par une ligne du type `[var texture].[méthode] = [valeur]`:

```
tex.noiseType = 'hard'
```

De la même façon, nous pouvons définir la taille du bruit, c'est à dire plus ou moins son facteur de répétition, grâce à la variable `noiseSize`. Par exemple:

```
tex.noiseSize = 0.55
```

Certaines autres textures procédurales admettent une autre variable, définissant leur niveau de détail: `noiseDepth`. C'est le cas de `'Clouds'`, `'Marble'`, et `'Magic'`, en particulier. Par exemple, nous aurions pu ajouter pour l'une ou l'autre de celles-ci une ligne:

```
tex.noiseDepth = 3
```

mais ça ne sera pas notre cas pour la texture de type Stucci. Nous pouvons donc résumer le code permettant l'obtention de la texture visible sur la Figure 15 par les lignes suivantes:

```
tex = Texture.New('Texture')
tex.setType('Stucci')
tex.setSType('StucciWallIn')
tex.noiseType = 'hard'
tex.noiseSize = 0.55
```

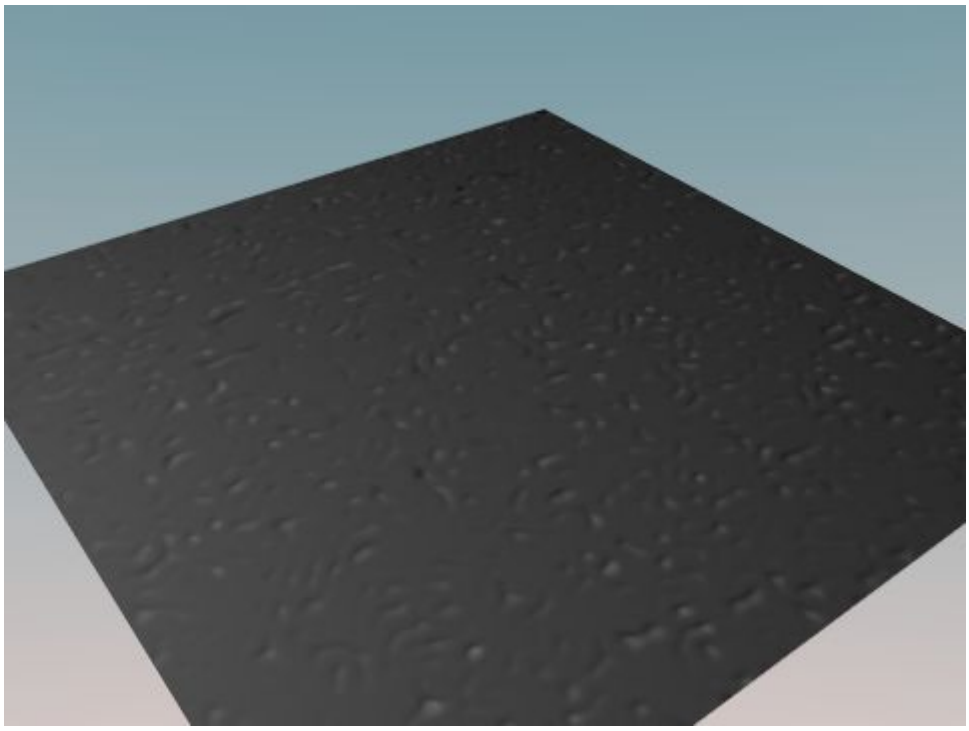

Figure 15: déclaration d'une texture de type Stucci par l'API Python

3. Usage des indices matériau

Lorsque vous modélisez dans Blender un objet de la vie courante, celui-ci peut-être d'apparence fortement composite, avec par exemple des vis chromées, une coque plastique, des courroies en caoutchouc et le tout posé sur un socle en bois ou en marbre. Selon ce qui vous arrange le plus, vous pouvez effectuer votre modélisation en autant que d'éléments que vous le souhaitez, puis « parenter » ([Ctrl]+[P]) le tout à un objet **Empty** pour le manipuler (lors de la composition de la scène ou au cours d'une animation) dans sa globalité. Ou alors vous pouvez décider de « joindre »

([Ctrl]+[J]) tous ces éléments en un seul maillage unique, bien plus facile à manipuler dans sa globalité. Vous pourriez dans ce dernier cas craindre de voir tous les matériaux individuels remplacés par un matériau unique et tout votre travail sur les *shaders* de chaque élément perdu, mais il n'en est rien: Blender est capable de gérer, pour un maillage unique, jusqu'à seize matériaux différents, distinguables les uns des autres par un indice.

L'idée est d'associer à toutes les faces qui ont la même « apparence physique » le même indice de matériau. Au-travers de l'API Python, nous utiliserons donc la méthode **`setMaterials()`** de la classe **`Nmesh`** pour déclarer la liste des matériaux accessibles par le maillage, puis enfin la méthode **`materialIndex`** pour associer à chaque face un indice de matériau.

Dans la pratique, nous allons commencer par lister les matériaux qui vont nous intéresser et les placer dans une variable python au nom le plus simple et explicite possible, grâce à la méthode **`Get()`** du sous-module **`Material`**. En supposons que nous avons trois matériaux à récupérer ('Or', 'Carrelage' et 'Granit'), cela nous donnerait les lignes suivantes:

```
mat_or = Material.Get('Or')
mat_carrelage = Material.Get('Carrelage')
mat_granit = Material.Get('granit')
```

Supposons que nous souhaitions attribuer ces matériaux au cube que nous avons créé au cours des articles précédents:

```
#### Definition du cube:
# Definition des points de controle:
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]
]

# Definition des faces:
liste_des_faces=[
	[0,1,2,3],   #face horizontale basse
	[4,5,6,7],   #face horizontale haute
	[0,4,7,3],   #face veticale avant
	[1,2,6,5],   #face verticale arriere
	[0,1,5,4],   #face verticale gauche
	[3,7,6,2]    #face verticale droite
]

CubeMeshData=NMesh.GetRaw()

# Description du cube:
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)
```

Notre **`Cube`** s'appelle **`CubeMeshData`**, mais c'est un peu long et mal commode à gérer. Appelons-le plus simplement **`m`** grâce à la ligne:

```
m = CubeMeshData
```

Les trois matériaux de notre cube ayant déjà été rapatrié sous les variables **`mat_granit`**, **`mat_or`** et **`mat_carrelage`**, nous pouvons tout simplement les attribuer à notre cube grâce à la ligne suivante:

```
m.setMaterials([mat_granit, mat_or, mat_carrelage])
```

L'ordre de déclaration des matériaux dans la liste passée par la méthode **`setMaterials()`** est importante: **`Granit`** correspond désormais à l'indice 0, **`Or`** à l'indice 1 et **`Carrelage`** à l'indice 2. Nous pouvons désormais attribuer individuellement, à chaque face, un numéro d'indice:

```
m.faces[0].materialIndex = 2
m.faces[1].materialIndex = 2
```

```
m.faces[2].materialIndex = 0
m.faces[3].materialIndex = 0
m.faces[4].materialIndex = 1
m.faces[5].materialIndex = 1
```

Enfin, nous pouvons ordonner à l'API Python d'insérer le maillage nouvellement constitué, paré de toutes ses couleurs, dans la scène courante de Blender, et observer le résultat dans la Figure 16.

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

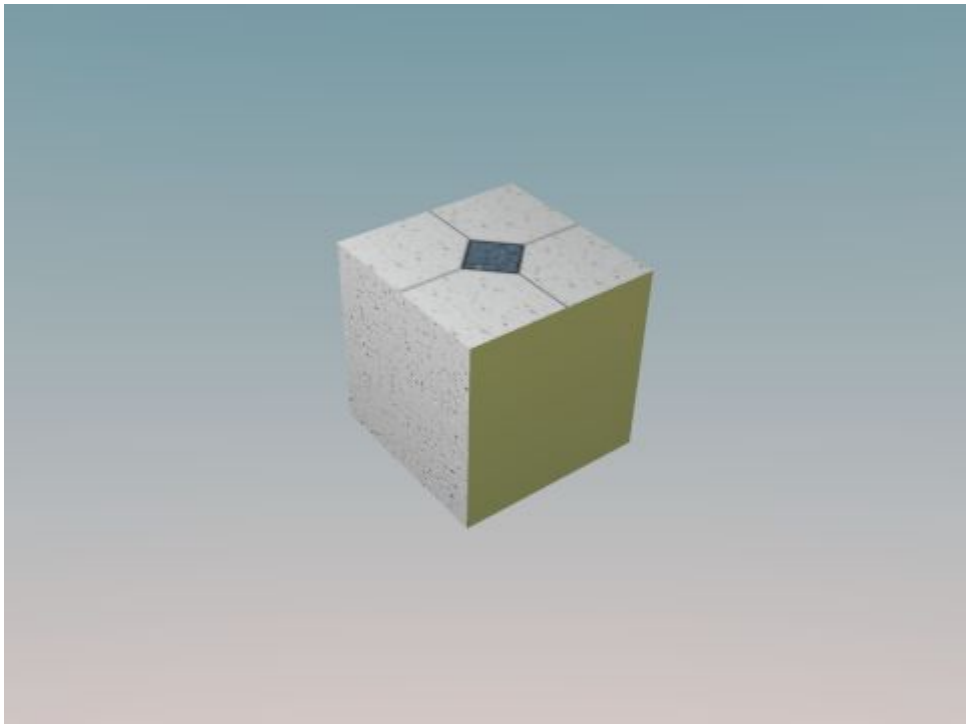

Figure 16: matériau simple, image plaquée et texture procédurale, mêlés au sein d'un même maillage par la magie des indices matériau

4. Conclusion

Vous êtes désormais certainement plus à l'aise avec les notions de Matériau, de texture image et de texture procédurale, et vous savez comment les mettre en oeuvre au-travers de l'API Python. Vous savez également jongler avec les indices matériau, et il n'en faut pas beaucoup plus pour faire des choses intéressantes. Vous avez également noté, une fois de plus, les limites de l'API Python actuelle, mais heureusement vous avez pu voir qu'il reste possible, grâce à une utilisation astucieuse de Python, d'obtenir la plupart des résultats souhaités. Je profite de ce constat pour rappeler que vous pouvez à tout moment intégrer l'équipe des codeurs de Blender, et ainsi, par exemple, contribuer à améliorer l'API Python jusqu'à ce que son usage soit aussi simple que celui de Blender lui-même.

Pour cet article, mes remerciements vont à Jean-Michel Soler (astuce du MapTo multiple) et à Yves Bailly (débogage de mes scripts et conseil pédagogique sur les indices matériau) pour leur aide discrète mais efficace . Enfin, une petite note personnelle: j'ai le plaisir de vous annoncer que je suis, depuis le 30 juin dernier, l'heureux papa d'une petite Chloé. Bien évidemment, la maman se porte à merveille même si les nuits ne sont plus aussi tranquilles qu'avant!

5. Résumé du code

```
import Blender

from Blender import NMesh, Material, Texture, Image

#### Definition des materiaux:
```

```
# Definition du granit:

matGranit = Material.New('Granit')
tex = Texture.New('Granit.Nor')
tex.setType('Stucci')
tex.setSType('StucciWallIn')
tex.noiseType = 'hard'
tex.noiseSize = 0.05
matGranit.setTexture(0,tex,Texture.TexCo.ORCO, Texture.MapTo.NOR)

# Definition de l'or:

matOr = Material.New('Or')
matOr.setRGBCol([1.0,1.0,0.4])
matOr.setSpecCol([1.0,1.0,0.8])
matOr.setMirCol([0.8,0.9,0.7])
matOr.setAlpha(1.0)
matOr.setRef(0.8)
matOr.setSpec(1.4)
matOr.setHardness(20)
matOr.setRayMirr(0.40)
matOr.setMirrDepth(3)
matOr.setMode('Traceable','Shadow','Radio','RayMirr')

# Definition du carrelage:

mat = Material.New('Carrelage')
mat.setSpecCol(1.0,1.0,1.0)
mat.setMirCol(0.9, 0.9, 0.9)
mat.setAlpha(1.0)
mat.setMode('Traceable','Shadow')
tex = Texture.New('Carrelage')
tex.setType('Image')
img = Image.Load('/home/olivier/Documents/Articles/Linux Magazine/blender-lm-
03/files/tile1.jpg')
tex.image = img
tex.setImageFlags('InterPol','MipMap')
tex.setExtend('Repeat')
mat.setTexture(0, tex, Texture.TexCo.ORCO, Texture.MapTo.COL)

#### Definition du cube:

# Definition des points de controle:
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]
]
# Definition des faces:
liste_des_faces=[
       [0,1,2,3],   #face horizontale basse
       [4,5,6,7],   #face horizontale haute
       [0,4,7,3],   #face veticale avant
       [1,2,6,5],   #face verticale arriere
       [0,1,5,4],   #face verticale gauche
       [3,7,6,2]    #face verticale droite
]

CubeMeshData=NMesh.GetRaw()

# Description du cube:

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)

# Definition des materiaux:

m = CubeMeshData
mat_granit = Material.Get('Granit')
mat_or = Material.Get('Or')
mat_carrelage = Material.Get('Carrelage')
m.setMaterials([mat_granit, mat_or, mat_carrelage])
m.faces[0].materialIndex = 2
m.faces[1].materialIndex = 2
m.faces[2].materialIndex = 0
m.faces[3].materialIndex = 0
m.faces[4].materialIndex = 1
m.faces[5].materialIndex = 1

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

Blender.Redraw()
```

6. Liens

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

La documentation officielle de python pour Blender:
http://www.blender.org/documentation/237PythonDoc/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**

La page de JM Soler sur l'activation sélective des canaux:
http://jmsoler.free.fr/didacticiel/blender/tutor/cpl_matcanauxmapto.htm