Narann

Zeste de savoir

Utiliser Python dans Autodesk Maya

4 septembre 2022

Table des matières

Introduction

Vous connaissez Maya, vous l'utilisez depuis un certain temps, mais vous souhaitez apprendre à automatiser vos tâches? Vous vous sentez un peu à l'aise avec Python, mais vous vous demandez comment l'utiliser dans Maya?

Ce tutoriel est fait pour vous ! 😊

Python dans Maya est une vraie corde à l'arc des *technical artists* (graphistes techniques). Il permet de gagner du temps en automatisant ses tâches ainsi que celle de ses collègues, mais aussi d'organiser son travail pour le rendre plus efficace.

Ce tutoriel vous fournira bon nombre d'informations pratiques ainsi que des TP issus de cas de production pour que vous puissiez à la fois assimiler certaines pratiques, mais aussi en inventer de nouvelles, propres à vos besoins. 😊

Première partie

Prérequis

— Avoir Maya d'installé. On utilise la version 2016, mais le tutoriel est valable pour toute autre version (les évolutions concernant Python sont sporadiques).
— Avoir certaines bases en Python. Savoir ce qu'est une variable, une fonction, un module, une liste, une boucle, etc. Nul besoin d'être expert, simplement être à l'aise avec les concepts de base. Si vous débutez complètement en Python, il vaut mieux faire un tutoriel avant.
— Comme on va se focaliser sur Python dans Maya, on est en Python 2.7.

Il peut être très frustrant pour une personne de l'image/un graphiste de faire un tutoriel de programmation. Dans ce cas, je vous invite à alterner entre ce tutoriel, qui essaie de vous présenter une facette de votre logiciel favori, et un tutoriel Python plus traditionnel. Votre progression sera plus lente, mais plus motivante.

Deuxième partie

Objectifs

Je considère que vous êtes avant tout un graphiste voulant ajouter une corde à son arc et non un développeur en devenir. Je vais principalement montrer des exemples d'utilisation dans Maya sur des cas qui monteront en complexité. Le but est de faire acquérir au graphiste, via l'écriture de scripts, des outils et méthodes qui lui permettront d'aborder les problèmes d'une manière plus large. 🍊

Ce tutoriel peut servir à des développeurs, mais ils iront vite dans la documentation une fois les concepts de base assimilés. 🍊

Troisième partie

Découverte des commandes

Introduction

Cette première partie est une entrée en douceur. Beaucoup de nouveaux concepts, propre à Maya seront abordés. Prenez donc votre temps pour bien les assimiler, car ils reviendront continuellement.

III.1. L'architecture

Introduction

Le logiciel Autodesk Maya (qu'on abrégera Maya dans le reste du tutoriel) est organisé en *couches*. On peut en définir trois grosses que je vais vous présenter dans ce chapitre (gardez toujours à l'esprit que la réalité est souvent plus subtile 🍊).

III.1.1. Le cœur (core en anglais)

Le cœur se compose de trois grosses parties :

- le graphe de dépendance (*dependency graph* en anglais, souvent abrégé *DG*); dorénavant, j'utiliserais l'abréviation *DG*;
- Les commandes;
- L'interface utilisateur (*user interface* en anglais, souvent abrégé *UI*).

Il n'y a aucun moyen de modifier ce code. Ce sont les développeurs de Maya qui le gèrent.

III.1.1.1. Le graphe de dépendance

On peut voir Maya comme une base de donnée de nœuds (*nodes* en anglais). Chaque nœud a des propriétés (appelées «attributs» dans Maya) d'entrée et de sortie. Ces attributs sont typés (nombre entier, nombre flottant, chaîne de caractères, énumération, matrice, géométrie, tableau de nombres entiers, tableau de nombres flottants, on y reviendra…). Les types similaires (ou compatibles) peuvent être connectés entre eux, permettant de passer les valeurs des attributs d'un nœud à l'autre. Cet ensemble de nœuds et de connexions forment le graphe de dépendance (*dependency graph* ou DG).

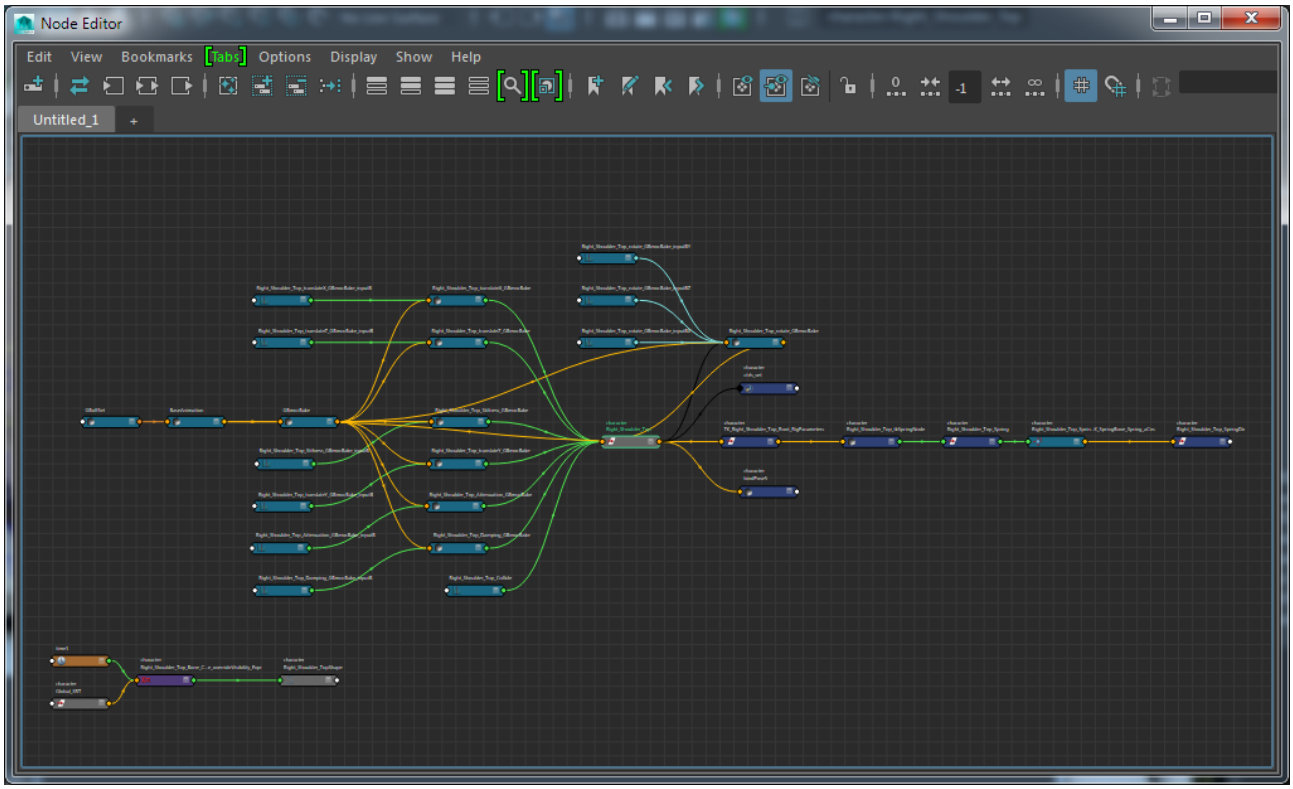


FIGURE III.1.1. – *Dependency Graph* (abrégé *DG*) dans le *Node Editor* de Maya.

Chaque type de nœud est responsable d'informer le DG de quels attributs d'entrée dépendent ses attributs de sortie. Cette information permet au DG de garder une trace claire de qui affect qui, de manière à ne recalculer que le strict nécessaire.

Par exemple, vous pouvez avoir un nœud de type *nurbsCurve* qui a un attribut de sortie contenant une courbe NURBS. Cet attribut peut être connecté à l'attribut d'entrée de courbe d'un nœud de type *revolve*. Le nœud de type *revolve* a un attribut d'entrée qui détermine l'angle de la révolution et l'axe de rotation. L'attribut de sortie du nœud de *revolve* est une surface NURBS. Vous pouvez connecter cet attribut à l'attribut d'entrée d'un nœud de type *nurbsSurface* qui sait comment dessiner la surface. Vous pouvez ensuite placer le nœud *nurbsSurface* en enfant d'un nœud de type *transform* qui sait comment positionner les objets dans l'espace 3D.

La chaîne des nœuds décrite ci-dessus donne ceci:

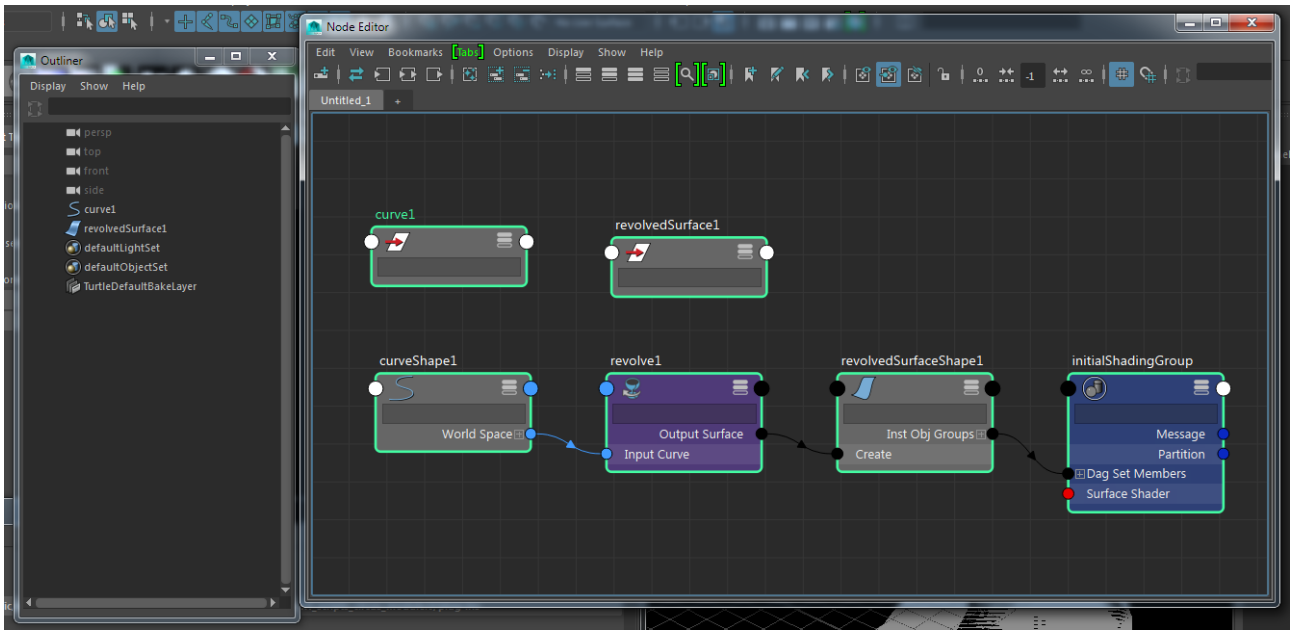


FIGURE III.1.2. – Nœud *revolve* (nommé `revolve1`) dans le *Node Editor* de Maya.

Les attributs du nœud de type *revolve*:

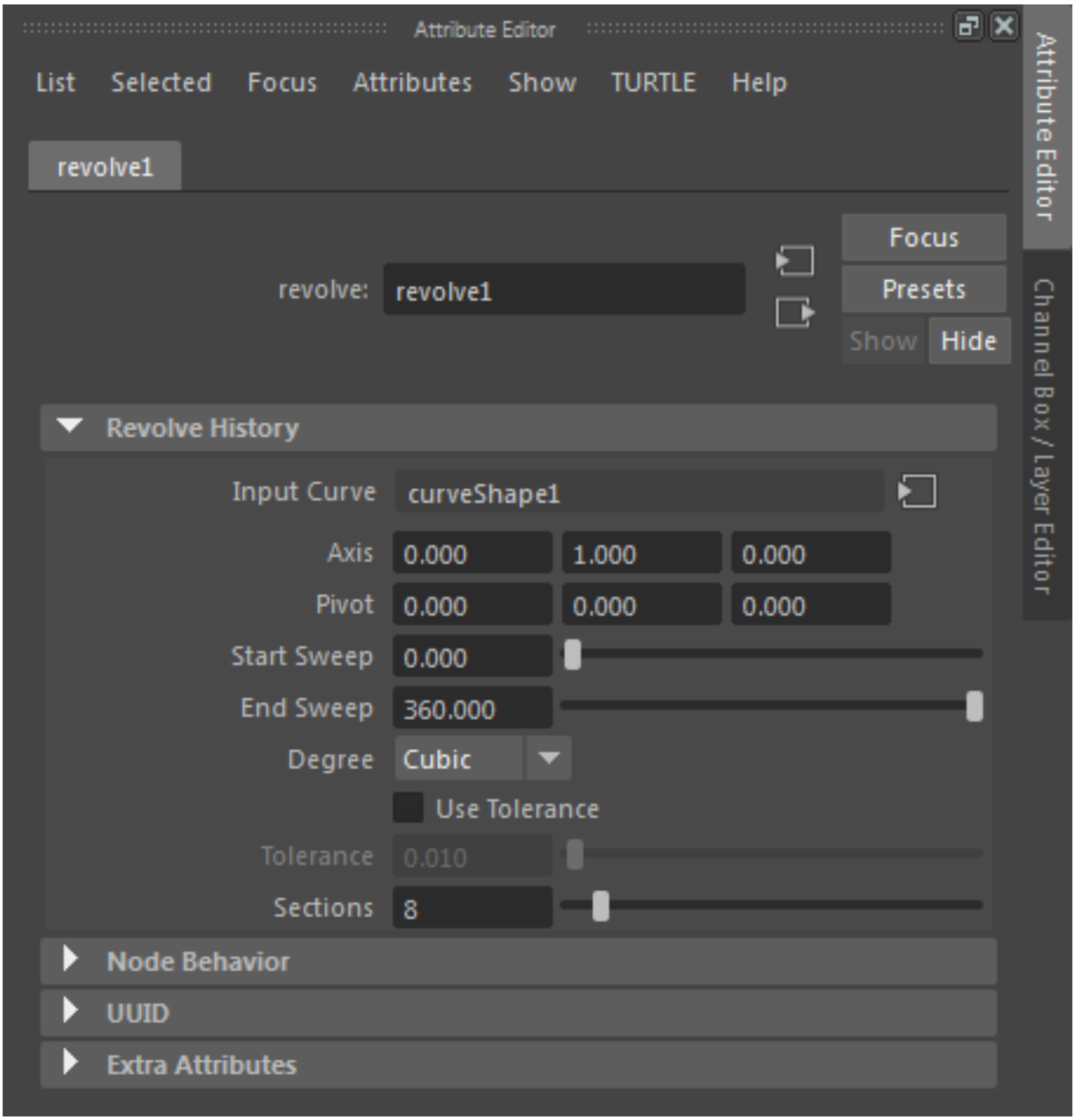


FIGURE III.1.3. – Les attributs d'un nœud de type *revolve*.

Et le résultat dans le *viewport* Maya :

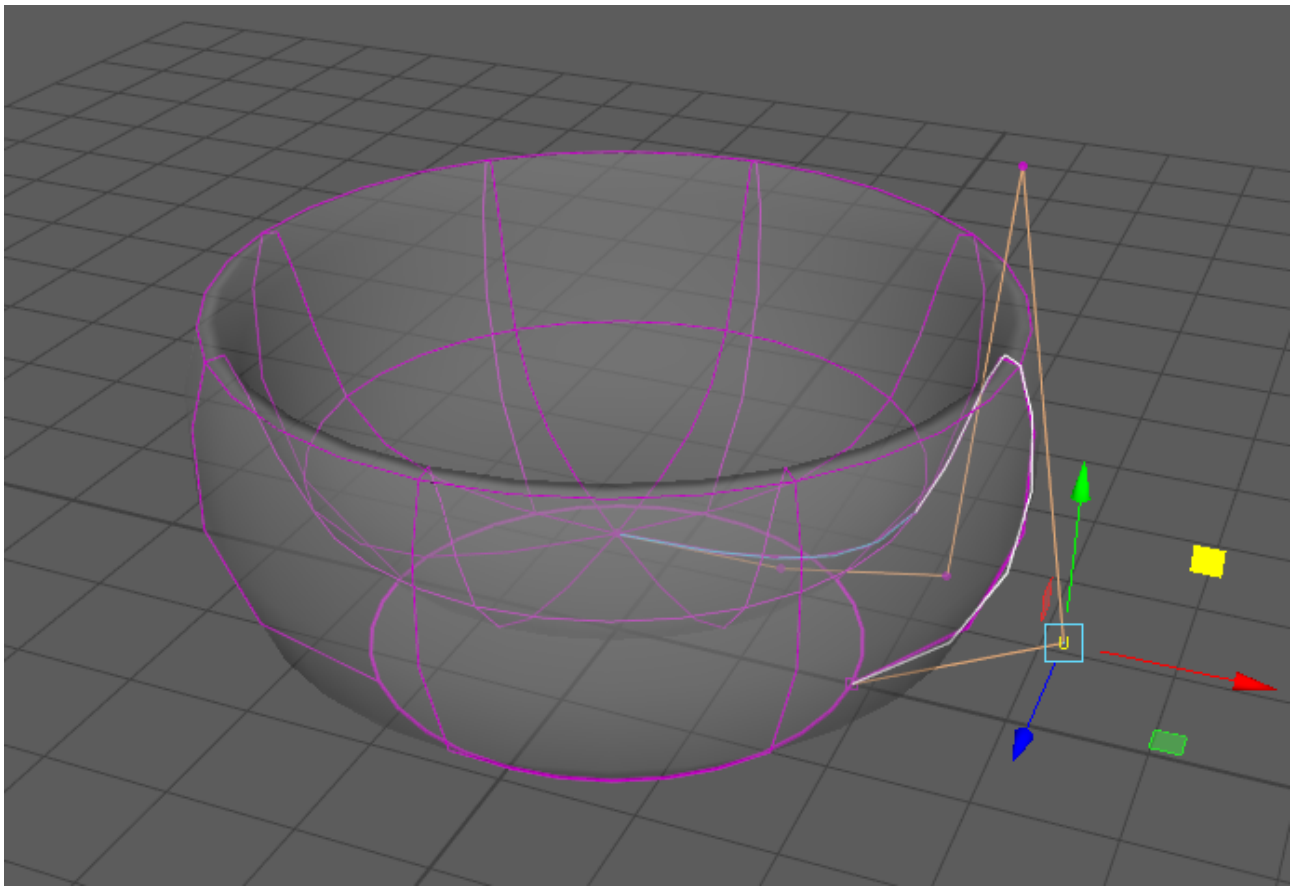


FIGURE III.1.4. – Du talent à en revendre! 🍊

Comme vous pouvez le constater, si on modifie la courbe ou change la valeur d'un des attributs, cela propage l'information dans tout le graphe qui, un à un, met les nœuds à jour.

Maya fournit de base plus de 600 nœuds. La liste des nœuds est disponible dans la documentation officielle (*Technical Documentation/Nodes*)

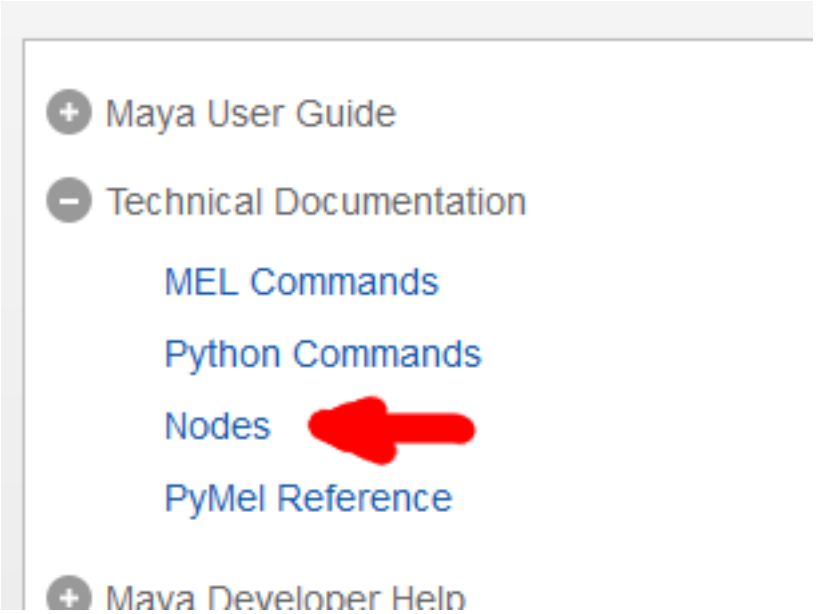


FIGURE III.1.5. – Aide Maya, *Technical Documentation/Nodes*.

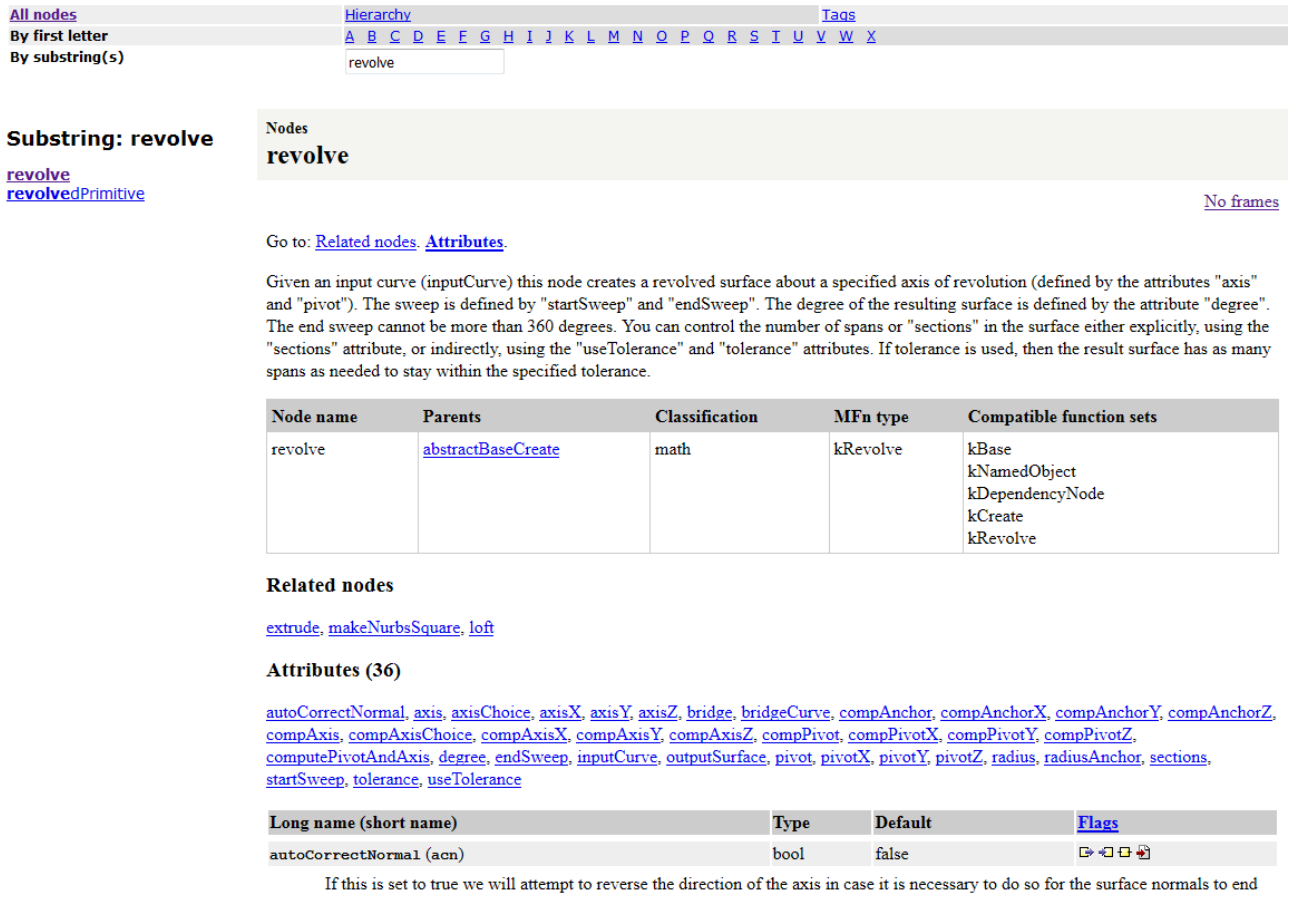


FIGURE III.1.6. – Aide Maya, documentation du nœud *revolve*.

C'est très soviétique me direz-vous mais c'est comme la tête, l'important, c'est ce qu'il y a dedans.

Et si cela ne suffit pas, vous pouvez créer vos propres types de nœud via l'API (on y reviendra)!

III.1.1.2. Les commandes

Manipuler directement les nœuds peut être fastidieux. Maya fournit donc plus de 900 commandes qui manipulent le DG pour vous. Ces commandes créent les nœuds, modifient et connectent leurs attributs et créent les nœuds de transformation qui positionnent les objets.

Par exemple, il existe une commande *revolve* qui prend l'angle et l'axe en argument, et construit le réseau de nœuds présenté précédemment pour la courbe NURBS sélectionnée. Cette commande s'occupe de tous les détails bas niveau pour vous.

La liste des commandes est disponible dans la documentation officielle : MEL Command Reference.

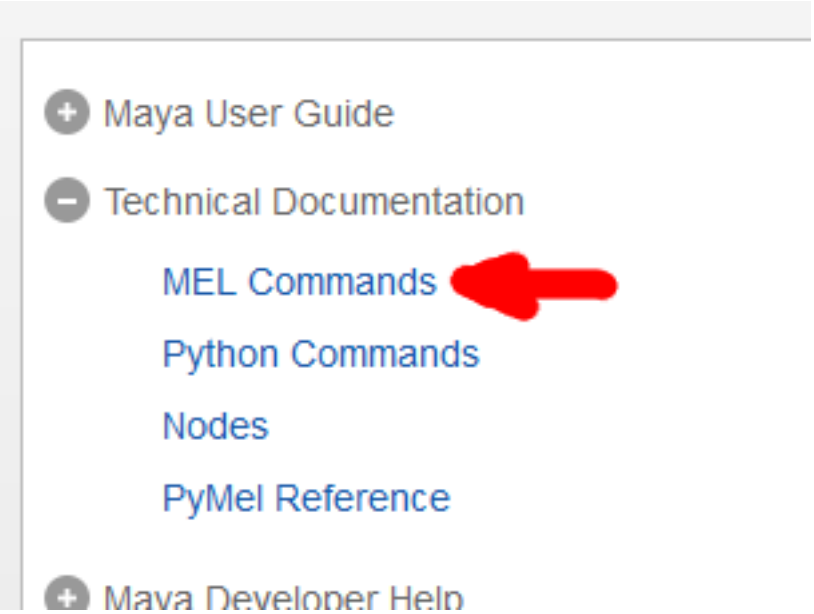


FIGURE III.1.7. – Aide Maya, *Technical Documentation/MEL Commands*.

Une fois encore, si vous n'êtes pas content, vous pouvez créer vos propres commandes.

III.1.1.3. L'interface

C'est tout ce qui fait «tourner» Maya. C'est le code qui détermine dans quel ordre les nœuds vont s'exécuter et comment ces derniers se passent les informations, comment les choses s'affichent dans le *viewport*, comment s'organise la mémoire quand on ajoute/supprime des nœuds/de la géométrie.

Bref, ce sont les données et leur représentation.

Environ 200 des commandes Maya s'apparentent à la création d'UI et vous permettent de créer vos fenêtres, boutons, icônes, etc. Maya utilise également ces commandes pour sa propre UI. En plus du nœud de type *revolve* et de la commande *revolve* existe un menu appelé «Revolve» ainsi qu'une icône «Revolve» dans le *shelf*. Le menu ou l'icône exécute une commande qui, à son tour, crée le réseau de nœud de DG. Le *scripting* est utilisé pour implémenter environ 98 % de l'UI de Maya.

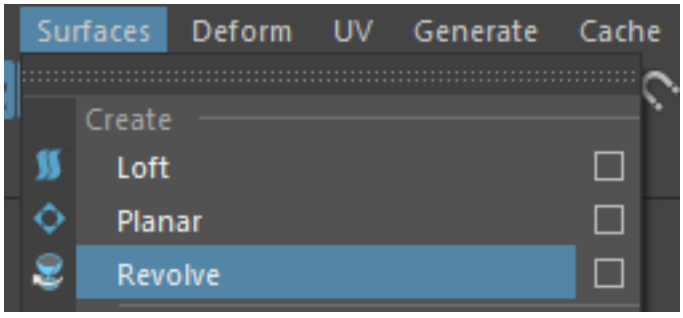


FIGURE III.1.8. – Menu *Surface/Revolve*.

Et une fois encore, si vous n'êtes pas content, vous pouvez créer vos propres UI, ajouter des menus, faire vos fenêtres personnalisées (pour les personnages à animer par exemple).

Vous l'aurez compris, les fondations de Maya sont puissantes et flexibles. Donner la possibilité aux graphistes de pouvoir scripter leurs propres outils et leurs interfaces était révolutionnaire à l'époque où c'était souvent le travail des développeurs. Le fait que n'importe quel TD (*technical director/technical artists*) puisse écrire des morceaux de script (voir des choses beaucoup plus complexes) pour se simplifier la vie a été déterminant pour la réussite de Maya dans les gros studios.

III.1.2. L'API

L'API (pour *Application Programming Interface* en anglais et Interface de programmation en français) est la seconde couche. Elle permet de «communiquer» avec le cœur de Maya et de faire pratiquement tout. Tous les nœuds et commandes Maya s'appuient dessus. Elle se compose de centaines d'outils plus ou moins utiles/interconnectés/spécifiques. Elle permet d'aller au plus bas possible pour le plaisir des bidouilleurs (et des crashs à déboguer :P).

Notez que l'utilisation de l'API implique une bonne maîtrise des concepts de programmation.

Il y a, grosso modo, deux gros blocs de classes dans l'API :

- Les classes *MPx* pour la création de nouveaux types de nœuds (*deformer*, *locator*, *primitive*, etc.) et commandes.
- Les classes *MFn* pour la manipulation des données.

Puis toutes les autres.

L'API s'utilise via trois langages de programmation :

- C++
- Python
- .NET

L'API historique est celle en C++. Celle en Python et .NET sont arrivées plus tard et dérivent directement de celle en C++ (comprendre un peu le C++ permet donc de se familiariser rapidement avec l'API).

Vous l'aurez compris, c'est du lourd.

III.1.3. Le scripting

La dernière grosse couche est celle qui va nous intéresser.

Comme je le disais précédemment, l'API permet de créer des commandes. Mais il faut bien un langage pour les exécuter, ces commandes !

III.1.3.1. Le MEL

Le MEL (*Maya Embedded Langage*) est le langage de script historique de Maya.

III.1.3.1.1. Une révolution dans l'industrie

À l'époque, beaucoup de logiciels 3D proposaient une API C ou C++ permettant aux développeurs *d'augmenter* leur logiciel. Bien qu'il s'agisse d'une forme d'ouverture pour les studios, il fallait maîtriser les C/C++ et la compilation pour pouvoir en tirer quelque chose. Ce n'est pas le genre de choses qu'un graphiste sait faire.

Une des spécificités de Maya (et aussi une grosse révolution en son temps) fut de considérer, dès le début du développement du logiciel, l'utilisation d'un langage de script non compilé (ici, le MEL), pour écrire toute la logique «haut niveau» du logiciel : Les menus de l'interface sont écrits et organisés en MEL. Quand vous cliquez sur un bouton/menu, ça exécute du MEL, vous pouvez donc faire tout ce que fait l'interface graphique en MEL.

Bref, du fait de son histoire, le MEL est partout dans Maya. 🍊

Cette ouverture, comparée aux autres logiciels, permettait à n'importe quel graphiste n'ayant pas peur de mettre les mains dans le cambouis, d'automatiser une grande partie de son travail. C'est sûrement cette approche qui a permis à Maya de devenir le logiciel de référence dans les grands studios d'animation et d'effets spéciaux.

III.1.3.1.2. Aperçu

Comme nous ne parlerons plus du MEL car tout ce tutoriel se focalise sur Python, voici un très bref aperçu.

Le MEL est un langage fonctionnel. Un langage fonctionnel est une manière d'organiser et d'écrire du code. Pour vous donner un exemple, la forme `résultat = action(données)` est typiquement une écriture en langage fonctionnel. En MEL on peut ne pas mettre les parenthèses. Ainsi, l'exemple précédant donnerait `résultat = \`action données“.

Tout est global en MEL. Si vous déclarez une fonction qui porte le même nom qu'une fonction utilisée par Maya, vous pouvez casser Maya. Exemple :

`polyPerformAction` est une fonction utilisée un peu partout dans l'interface :

```
1 whatIs polyPerformAction
```

La commande MEL `whatIs` donne des informations sur une commande donnée.

```
1 // Result: Mel procedure found in: (votre
      installation)/scripts/others/polyPerformAction.mel //
```

On voit ici que la fonction `polyPerformAction` est définie dans un fichier en `.mel` dans le dossier d'installation de Maya.

> Comme vous l'aurez remarqué dans la valeur de retour de `whatIs`, en MEL, on ne parle pas de «fonction» mais de «procédure». Mais c'est un détail. Ne vous embêtez pas avec ça, nous allons nous contenter du terme «fonction». 🍊

Si on essaye de la redéfinir (c.-à-d. créer une nouvelle commande qui porte le même nom que la fonction à redéfinir):

```
1 global proc polyPerformAction()
2 {
3     print("TEST");
4 };
```

Maya nous met en garde :

```
1 // Warning: New procedure definition for "polyPerformAction" has a
      different argument list and/or return type. //
```

Mais on a cassé Maya. Si, par hasard, on fait quelque chose dans l'interface qui appelle la commande `polyPerformAction`, il y a de fortes chance qui votre Maya vous affiche des messages d'erreur. Heureusement, il suffit de redémarrer le logiciel pour repartir de zéro.

Une bonne pratique des studios (et d'une manière générale) était de préfixer leurs fonctions avec un diminutif du studio.

III.1.3.1.3. Le déclin

Au fil des années deux problèmes majeurs apparurent :

- Le MEL commençait à accuser son âge et sa rigidité est devenu problématique alors que d'autres langages (dits *à typage dynamique*) commençaient à percer.
- Le MEL étant fermé et limité à Maya, aucune interopérabilité avec d'autres outils n'était possible. Il fallait donc obligatoirement passer par du C++.

Une alternative était nécessaire…

III.1.3.2. Python à la rescousse !

En 2007, Maya 8.5 embarque un interpréteur Python (version 2.4.3) permettant d'exécuter des commandes de script, mais surtout également, des commandes de l'API C++. C'est une avancée majeure pour les studios de production car Python n'est pas juste un langage de script mais un écosystème de modules intégrés (Python est dit *battery included*) et tiers, là où le MEL est vraiment seul et (quasi) vide.

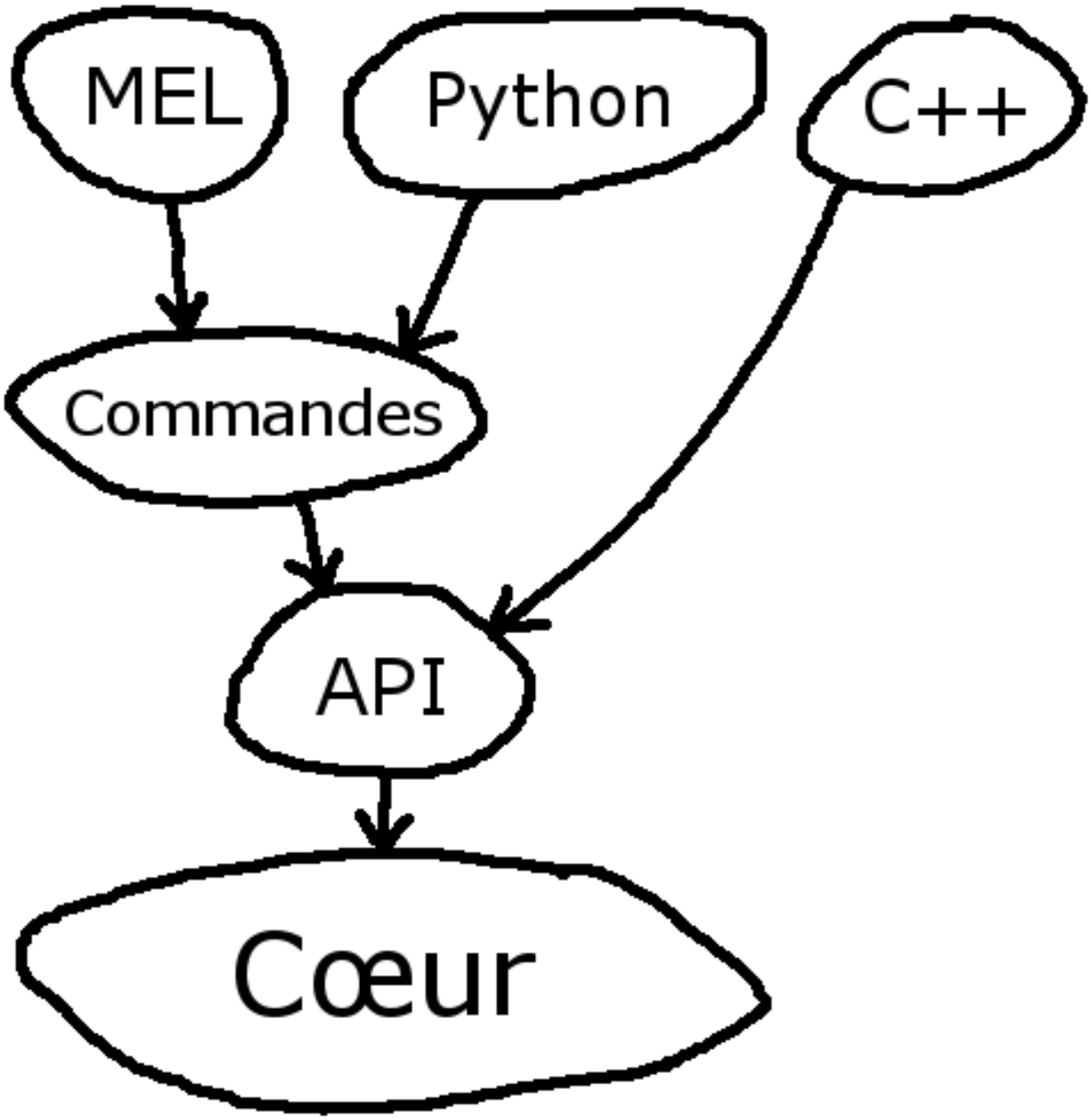


FIGURE III.1.9. – Voici comment MEL, Python et C++ peuvent interagir avec Maya.

Cet ajout a eu une portée quasi historique dans l'histoire de la fabrication des effets spéciaux, car le mouvement a été suivi par tous les autres logiciels du marché.

À l'heure actuelle, Python est le langage de prédilection des *pipelines* des studios d'effets spéciaux.

III.1.3.2.1. Note concernant la version de Python

L'industrie des effets spéciaux est assez paradoxale. D'un côté elle est à la pointe des technologies concernant la représentation et la manipulation artistique de données monstrueuses et complexes (scène, rendu, géométrie, animation, etc.) mais est totalement à la ramasse sur les bibliothèques, chaque logiciel utilisant la version qu'il jugeait la plus pertinente.

Il faut garder à l'esprit que la programmation et le code n'est qu'une partie assez secondaire du travail lié à l'image. La priorité des développeurs des logiciels est mise sur la capacité itérer rapidement sur la qualité d'une image.

La situation est devenue telle que différents développeurs de logiciels concurrents ont décidés de s'organiser pour «synchroniser», avec plus ou moins de succès, les différentes versions des bibliothèques utilisées au fil des années. Cet effort a donné naissance à la VFX Reference Platform dont je vous traduis la phrase d'introduction:

> La Plateforme de Référence VFX est un ensemble de versions d'outils et de bibliothèques communs sur lesquels s'appuyer pour concevoir des logiciels pour l'industrie des effets spéciaux. Son but est d'atténuer les incompatibilités entre les différents logiciels, faciliter la maintenance des pipelines de production utilisant Linux et d'encourager l'adoption de Linux par les éditeurs de logiciels. La Plateforme de Référence est mise à jour annuellement par un groupe d'éditeur de logiciels en collaboration avec le Visual Effects Society Technology Committee .

Ainsi, Python a été intégré pour remplacer le MEL qui ~~devenait~~ était déjà vraiment vieillot et faisait perdre du temps aux équipes. La version a évolué au fil des années de 2.4 à 2.7. Sauf que la version 3 n'était pas rétrocompatible. Ainsi, du code écrit en 2.7 n'était pas directement compatible avec Python 3. Ceci, ajouté au fait que:

— à l'époque Python 3 n'apportait rien de vraiment révolutionnaire par rapport à 2.7,
— certaines librairies ne fonctionnent pas en Python 3 (PySide 2 + Qt 5.6),
— des pipelines entiers ont été montés en Python 2.7,
— le script n'est pas une priorité pour les éditeurs de logiciels,

fait que le passage à Python 3 n'est pas prévu avant 2020, date de fin de support de la branche 2.7 par la communauté Python, qui ne corrigera plus les failles de sécurité.

Comprenez que ce n'est pas pour tout de suite et ça va être un joyeux bordel le jour où ça arrivera!

> Tous les codes que vous verrez seront donc en 2.7.

Conclusion

Nous venons d'avoir un aperçu général du fonctionnement de Maya et quelles ouvertures il laisse à ses utilisateurs suivant le langage qu'ils utilisent. C'est sûrement encore un peu abstrait, mais gardez à l'esprit que nous n'utiliserons que Python.

III.2. Où taper du code ?

Introduction

Après une introduction rapide au fonctionnement de Maya, voici le chapitre concernant l'outil que vous allez utiliser le plus souvent durant ce tutoriel : le *Script Editor*.

III.2.1. La ligne de commande

La ligne de commande (*command line* en anglais) vous permet d'écrire et d'exécuter une simple ligne MEL ou Python sans avoir à ouvrir le *Script Editor* :

FIGURE III.2.1. – La ligne de commande.

Vous pouvez commuter entre MEL et Python en cliquant sur le bouton contenant le nom du langage à gauche de la ligne de commande.

Allez-y, cliquez sur le bouton *MEL* pour qu'il passe en Python puis exécutez ceci :

```
1 print "Bonjour!",
```

Le résultat apparaît dans la sortie à droite de la ligne de commande :

Maintenant, tapez ceci :

```
1 untrucquivapasmarcher
```

Et voyez comment la sortie devient rose méchante ! 🍊

Dernière chose utile avant d'entrer dans le *Script Editor*, placez votre curseur dans la ligne de commande et faites Haut sur votre clavier.

Surprise ! 🍊

Les commandes tapées précédemment (ici `print "Bonjour!",`) apparaissent. On peut ainsi naviguer entre les commandes déjà tapées via Haut et Bas.

Au bout d'un moment, vous deviendrez suffisamment à l'aise et il vous arrivera de taper certaines commandes directement ici. 🍊

III.2.2. Le Script Editor

Le *Script Editor* (l'«éditeur de scripts» en français) est une fenêtre dédiée à l'écriture de scripts dans Maya. Il n'est pas forcément très pratique pour l'écriture de gros scripts, mais c'est largement suffisant pour commencer à se familiariser. 🍊

III.2.2.1. Ouvrir le *Script Editor*

Il y a plusieurs façons d'ouvrir le *Script Editor*. La plus évidente est de cliquer sur l'icône à droite de la sortie de la ligne de commande :

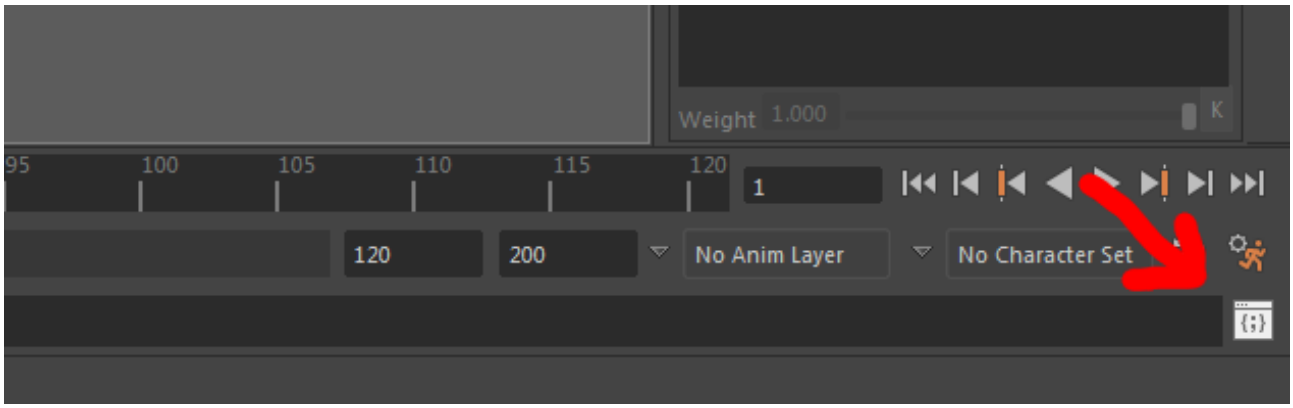
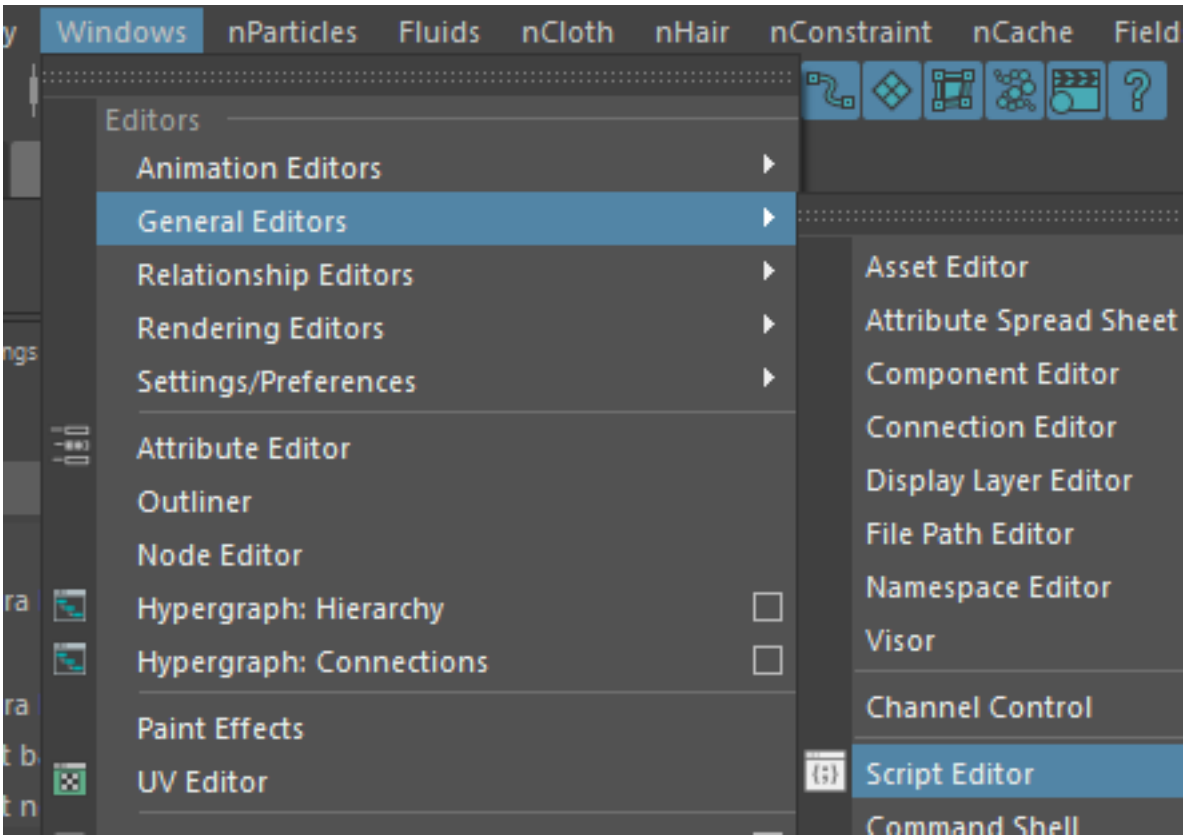
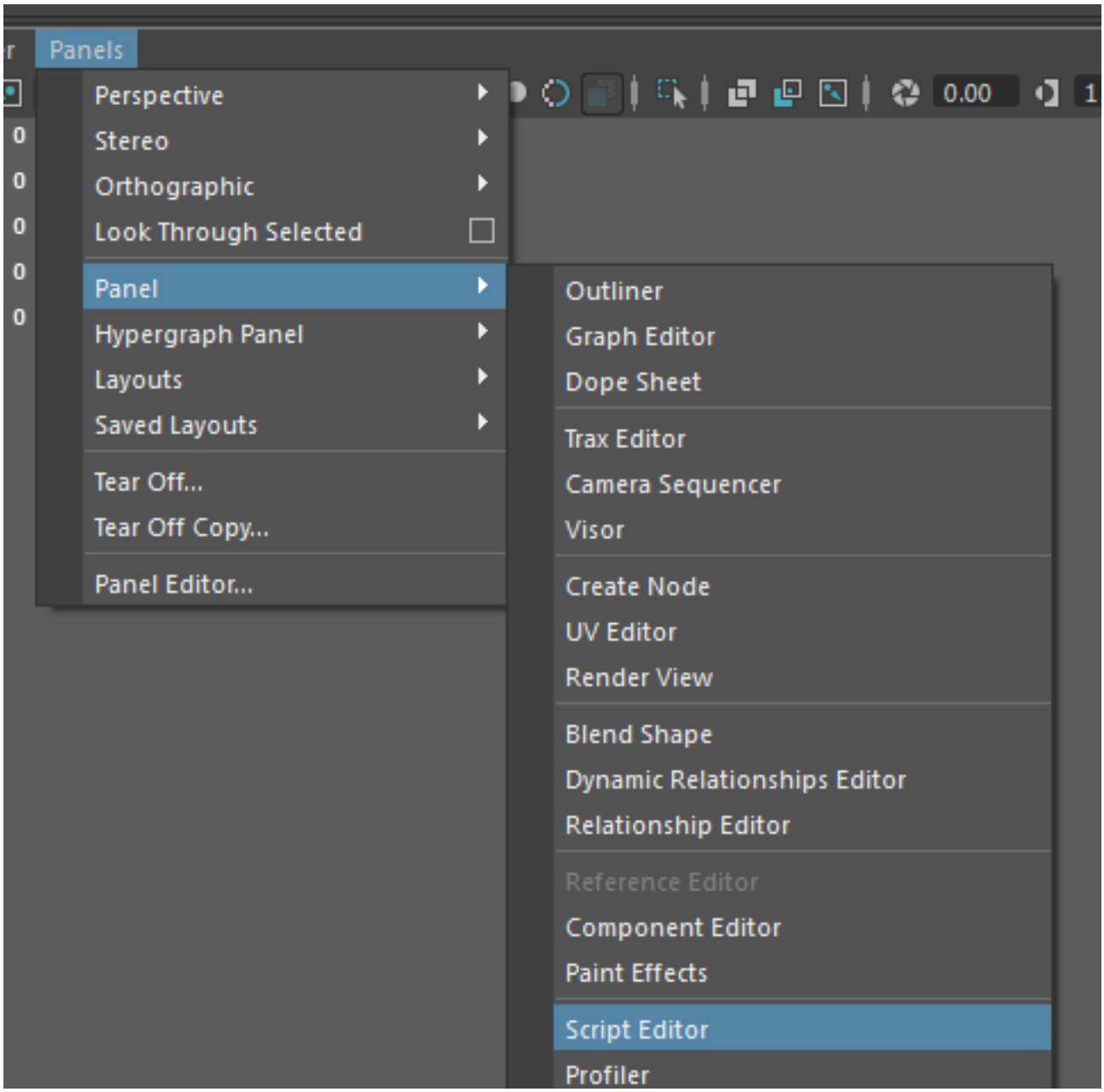


FIGURE III.2.2. – Une icône explicite! 🍊

Mais vous pouvez aussi l'ouvrir depuis le menu principal :



Ou bien encore de convertir un *panel*:



III.2.2.2. Rapide présentation

Quelle que soit la façon dont vous avez ouvert le *Script Editor*, vous devriez vous retrouver face à quelque chose qui ressemble à ça :

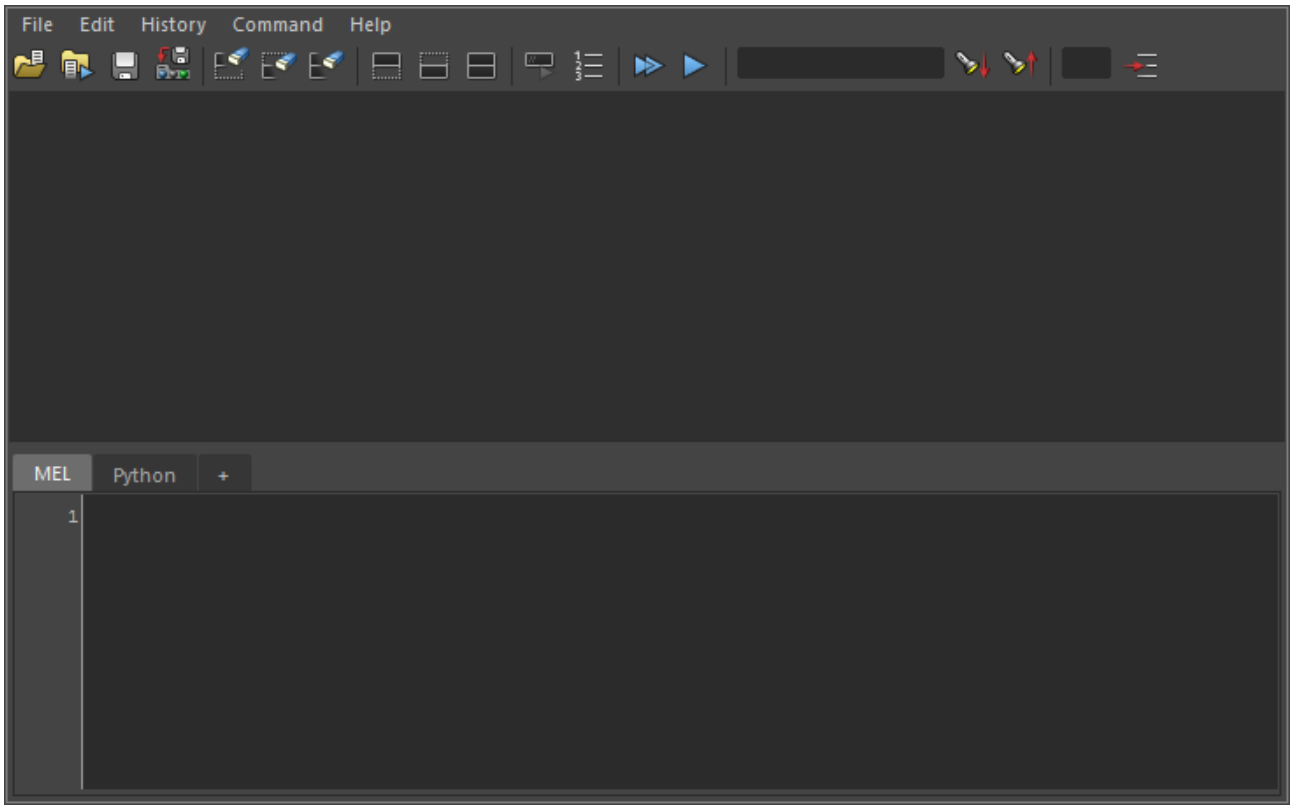


FIGURE III.2.3. – Le *Script Editor* de Maya!

— Tout en haut vous avez le menu (*File*, *Edit*, *History*, *Help*).
— Juste en dessous, toutes les icônes, c'est la barre d'outils (*toolbar* en anglais). Elle contient les entrées du menu les plus souvent utilisées.
— Viens le premier bloc qu'on appelle la sortie (*output* en anglais), c'est ici que tout ce qui sortira de votre code (les appels à `print` entre autres), sera affiché.
— Le second (et dernier) bloc contient des onglets suivant le type de code que vous écrivez. Vous vous en doutez, on va utiliser Python. Le «+» à droite permet de créer un nouvel onglet.

III.2.2.2.1. Un mot sur la sortie

Avant d'entamer ce chapitre un peu indigeste et sans intérêt pratique immédiat, je vais essayer de vous faire saliver un peu.:diable : Créez une sphère :

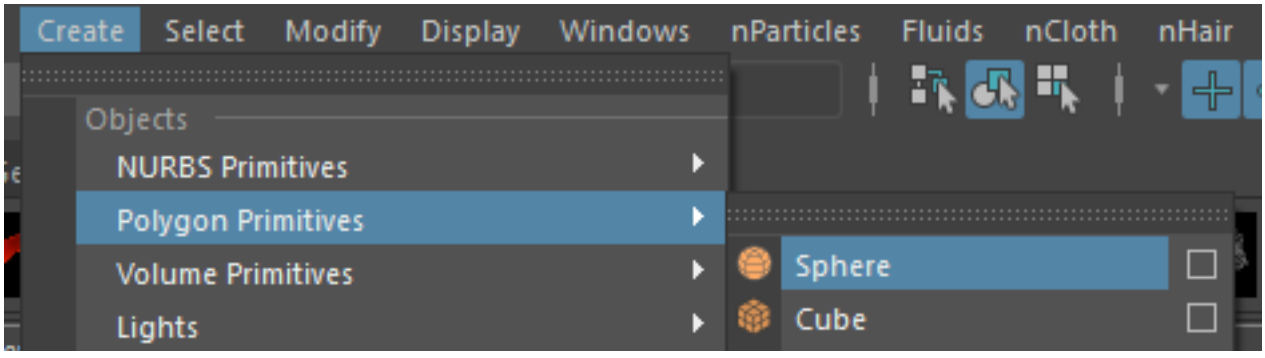


FIGURE III.2.4. – Créer une *polySphere* dans Maya.

Je vous expliquerai au prochain chapitre comment faire ça en script !

Cliquez dessus, déplacez-la, tournez-la, changez ses attributs.

La sortie devrait vous afficher quelque chose comme ça:

```
move -r -3.235522 6.06181 -1.310835 ;
rotate -r -os -fo 15.868418 -21.480475 5.314154 ;
scale -r 0.729335 0.729335 0.729335 ;
setAttr "pSphere1.rotateX" 7234;
setAttr "pSphere1.rotateX" 44;
setAttr "pSphere1.visibility" 0;
select -cl  ;
select -r pSphere1 ;
```

? Ouah ! Tout ce que je fais dans l'interface est affiché dans la sortie ! Comment je peux récupérer ça et en faire un script ? 🍊

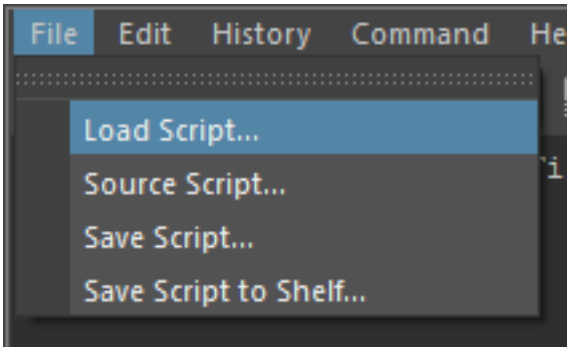
Eh bien nous verrons ça plus tard ! 🍊

Pour l'instant, on va se concentrer sur ce que propose le *Script Editor*.

III.2.2.3. Les menus et icônes

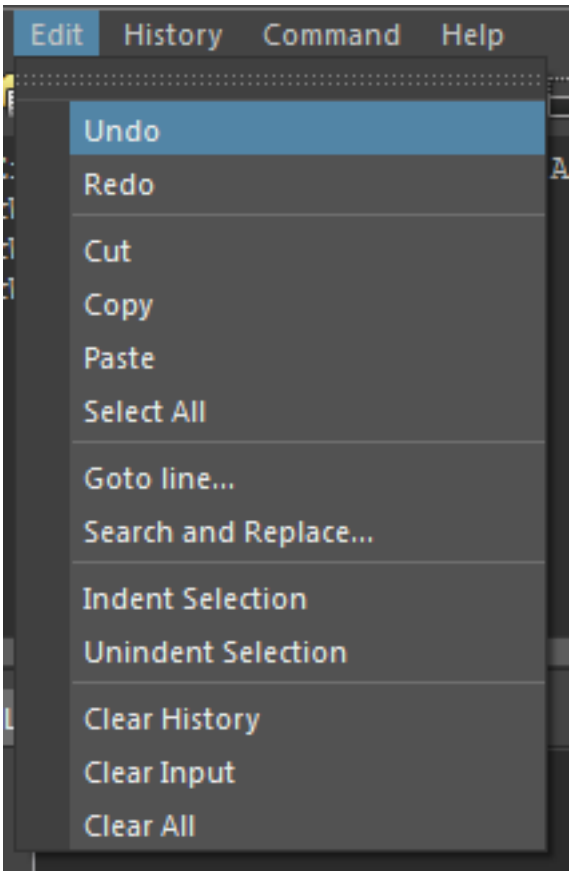
Nous allons rapidement passer en revue les menus du *Script Editor*. Notez que certains de ces menus disposent d'une icône permettant un accès plus rapide. Vous pouvez vous amuser à les retrouver. 🍊

III.2.2.3.1. *File*



— *Load script*: charge le contenu d'un fichier texte dans l'onglet courant.
— *Source script*: exécute le contenu d'un fichier texte.
— *Save script*: sauvegarde le texte sélectionné dans un fichier texte.
— *Save script to shelf*: ajoute un bouton au *shelf* courant exécutant le texte sélectionné. Vous allez voir qu'il y a une manière beaucoup plus classe de le faire. 🍊

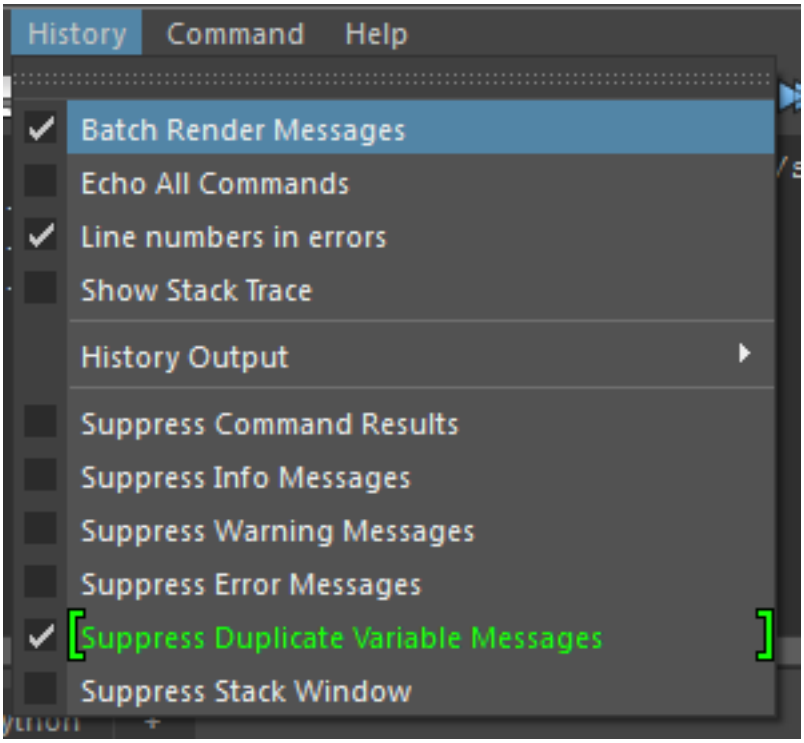
III.2.2.3.2. *Edit*



Je ne vais pas tous les faire, ce sont les fonctions classiques d'un éditeur de texte. Seules les trois dernières sont assez spécifiques a Maya.

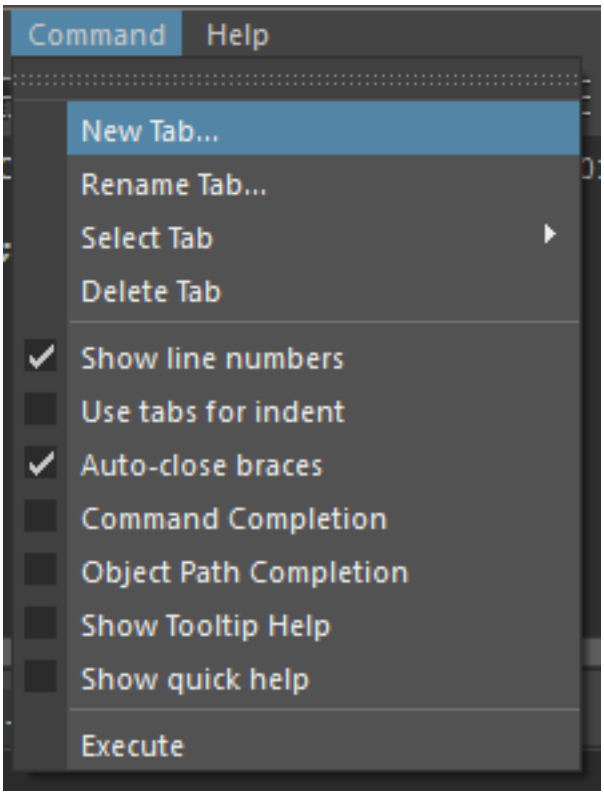
- *Clear History*: vide le contenu de la sortie. On l'utilise souvent quand on exécute un code complexe, qui imprime beaucoup de choses et qu'on souhaite pouvoir relire la sortie.
- *Clear Input*: vide le contenu de l'onglet courant. C'est beaucoup plus rare.
- *Clear All*: vide le contenu de la sortie **et** de l'onglet courant. C'est encore plus rare.

III.2.2.3.3. *History*



— *Batch render messages*: affiche les messages de rendu en *batch*.
— *Echo all commands*: affiche toutes les commandes. Par exemple, si vous faites *Create/Polygon Primitives/Sphere*, la commande MEL correspondante (ici, `polySphere`) s'affichera dans la fenêtre de sortie. On n'active cette option que sporadiquement, sur une seule commande pour essayer de deviner ce que fait l'interface pour nous. Essayez pour voir.
— *Line numbers in errors*: affiche les numéros de la ligne incriminée dans les messages d'erreur. Je vous conseille d'activer cette option pour déboguer plus rapidement un crash dans vos scripts.
— *Show stack trace*: c'est l'option à activer de toute urgence. Quand Python crashe, il peut vous donner la «pile d'exécution» (c.-a-d. «quelle fonction appelle quelle fonction appelle quelle fonction»). C'est vraiment pratique quand un script complexe crashe. Ne vous inquiétez pas, vous aurez tout le plaisir de vous rendre compte de son utilité durant votre apprentissage.
— *Suppress …*: je ne vais pas tous les détailler. Le principe est d'alléger la fenêtre de sortie en masquant des messages dont on n'aurait pas besoin.

III.2.2.3.4. *Command*



Ce menu concerne les onglets ainsi que le bloc d'édition de texte. Beaucoup de ces options sont une affaire de goût. Je vous les présente ici, mais je vous invite a les essayer durant votre apprentissage pour savoir lesquelles correspondent le mieux à votre style.

- *New Tab*: crée un nouvel onglet. Une petite fenêtre s'ouvre alors pour vous demander si vous souhaitez créer un onglet en MEL ou en Python. Notez que vous pouvez également créer un nouvel onglet en cliquant sur le bouton + à droite des onglets déjà présents ou, si vous voulez faire comme les pros, via le raccourci clavier Ctrl+T.
- *Rename Tab*: comme son nom l'indique, sert à renommer l'onglet courant.
- *Select Tab*: un magnifique menu qui vous propose de passer à l'onglet précédant via *Previous* et à l'onglet suivant via *Next*. Sachez que vous pouvez en faire de même en manipulant la molette de votre souris sur les onglets. C'est quand même beaucoup plus pratique.
- *Delete Tab*: pour supprimer l'onglet courant.
- *Show line numbers*: affiche les numéros des lignes à gauche du bloc de texte.
- *Use tabs for indent*: utilise des tabulations pour l'indentation. Quand vous appuierez sur la touche Tab de votre clavier, un caractère de tabulation sera créé plutôt que des espaces. Moi, je n'aime pas les tabulations alors je la désactive.
- *Auto-close braces*: ferme automatiquement les accolades ({ et }) quand vous écrivez du code. Python n'utilisant pas les accolades, le problème est réglé. .
- *Command Completion*: permet l'auto-complétion du nom des commandes. Si vous avez activé *Show Tooltip Help*, le nom des commandes apparaît en même temps que vous les tapez. Sinon, il vous faut faire Ctrl+Barre d'espace pour les faire apparaître.
- *Object Path Completion*: permet l'auto-complétion du nom des objets. Fonctionne de la même façon que *Command Completion*.

— *Show Tooltip Help*: affiche automatiquement la fenêtre l'auto-complétion quand vous tapez.

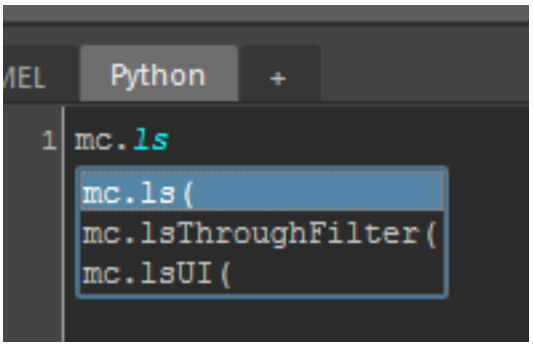


FIGURE III.2.5. – Exemple d'auto-complétion.

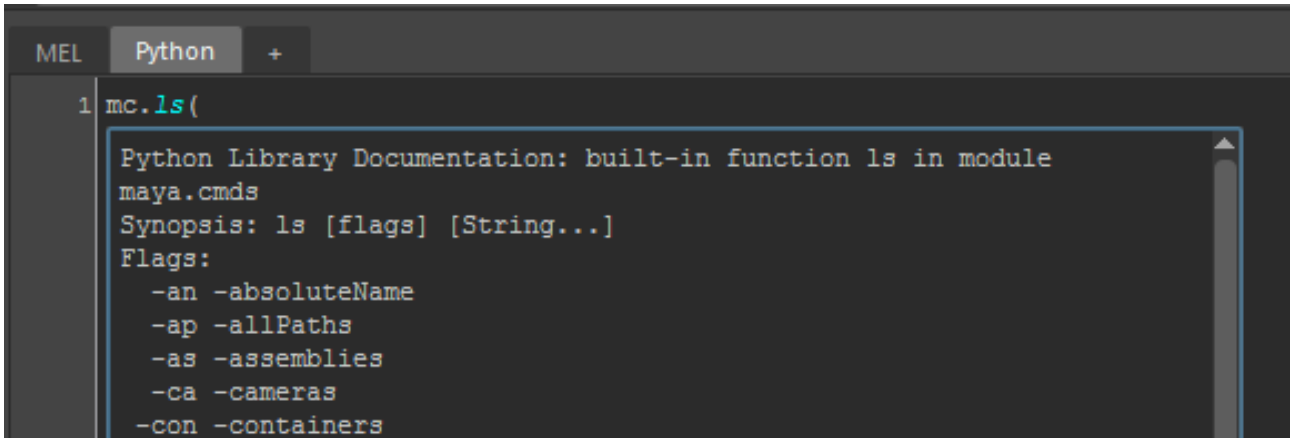


FIGURE III.2.6. – Exemple d'aide.

— *Show Quick Help*: affiche un petit panel d'accès rapide à la liste des arguments d'une commande. On peut double-cliquer sur les arguments et il les écrit dans l'éditeur de texte. Ce n'est toutefois pas très pratique.

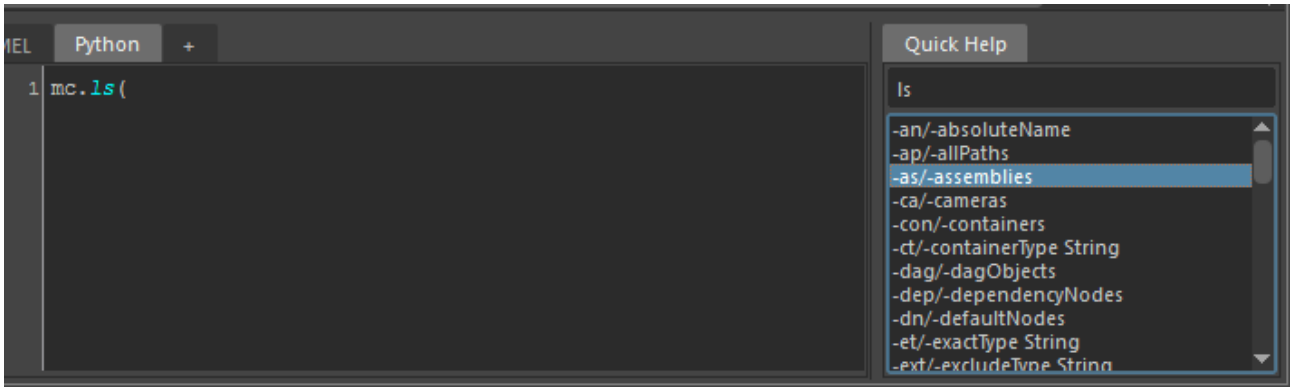
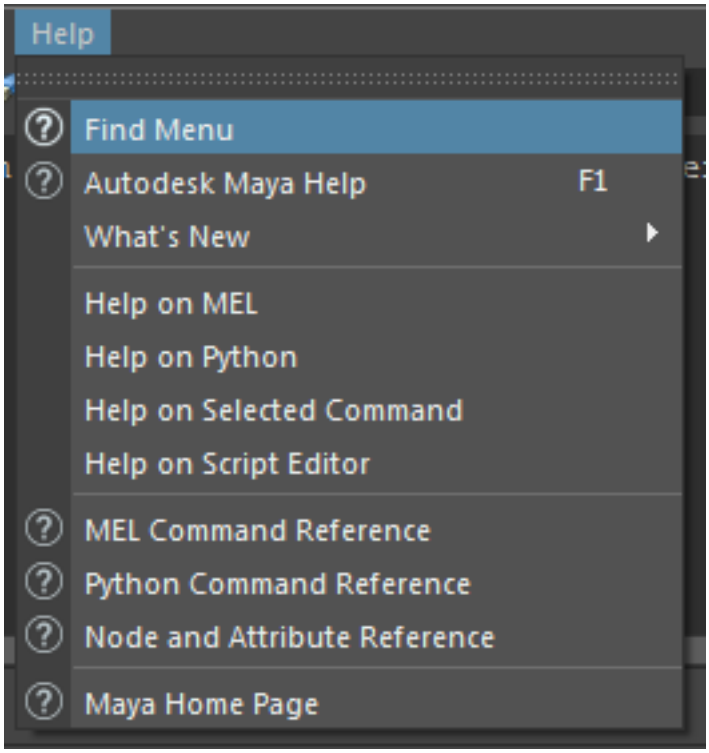


FIGURE III.2.7. – *Quick Help* en action!

— *Execute*: comme vous vous en doutez, permet d'exécuter le contenu de l'éditeur de texte. Je lui préfère la touche Entrée du pavé numérique (ou Ctrl+Entrée du clavier «normal»).

III.2.2.3.5. Help



Ce menu se passe de commentaires.

Pensez à bien activer *History/Show Stack Trace.*

III.2.3. Import du module Maya

C'est bientôt la fin de ce chapitre et je voudrais vous parler d'un truc.

Vous le remarquerez rapidement: partout dans la documentation, le module responsable de l'exécution des commandes maya est appelé sous la forme :

```
1 import maya.cmds as cmds
2
3 cmds.maCommande()
```

J'utilise toutefois cette approche, plus courte :

```python
import maya.cmds as mc

mc.maCommande()
```

Pour éviter d'avoir à taper la ligne d'import à chaque ouverture de Maya, je vous invite à créer un fichier texte nommé `userSetup.py`:

- Sous Windows : `C:\Users\<user>\Documents\maya\scripts\userSetup.py`
- Sous Linux : `~/maya/scripts/userSetup.py`
- Sous Mac : `/users/<user>/Library/Preferences/Autodesk/maya/2018/scripts/userSetup.py`

Et d'y mettre :

```python
import maya.cmds as mc
```

> Le fichier `userSetup.py` est chargé au démarrage de Maya.

Ainsi vous n'aurez plus à importer le module à chaque ouverture.

À partir de maintenant, je partirai du principe que le module Maya est déjà chargé dans votre session et j'appellerai toutes mes commandes via l'espace de nom `mc`.

Conclusion

J'ai bien conscience que ce chapitre était assez lourd à digérer, mais le *Script Editor* étant votre principal outil pour ce tutoriel, il me semblait important de faire le tour de ce qu'il permettait.

Je vous invite à garder cette page sous le coude si vous avez un doute et à tester différentes options.

Nous allons maintenant rentrer dans le vif du sujet, vous allez maintenant apprendre à voir une scène comme un script la voit.

Rendez-vous au prochain chapitre !

III.3. Nos premiers pas

Introduction

Dans ce chapitre, nous allons voir quelques commandes de base qui nous permettent de «regarder», puis de modifier notre scène.

Vous allez voir que le chapitre est un peu long, mais ces quelques commandes sont sûrement les plus utilisées en script, n'hésitez donc pas à prendre votre temps et avancer progressivement. 🍊

À partir de maintenant, je vous invite à toujours avoir un fichier texte d'ouvert sur votre bureau pour garder les précieuses petites lignes de code que vous allez taper. Elles formeront un bloc-note qui vous servira au fil de vos améliorations. Elles vous permettront de trouver et coller rapidement des morceaux pour les adapter à votre usage plutôt que de tout réécrire à chaque fois (ce qui est souvent décourageant). 🍊

III.3.1. nodeType() pour récupérer le type des nœuds

La commande `nodeType()` sert à récupérer le type d'un nœud.

Nous allons partir d'une scène vide et créez une sphère, mais ce coup-ci, en script :

```
1 mc.polySphere()
2 # Result: [u'pSphere1', u'polySphere1'] #
```

Votre première commande, c'est-y pas beau tout ça ? 🍊

Mais nous ne l'expliquerons pas pour l'instant. Gardez juste en tête que cette commande fait la même chose que ce que vous faites manuellement quand vous faites ça :

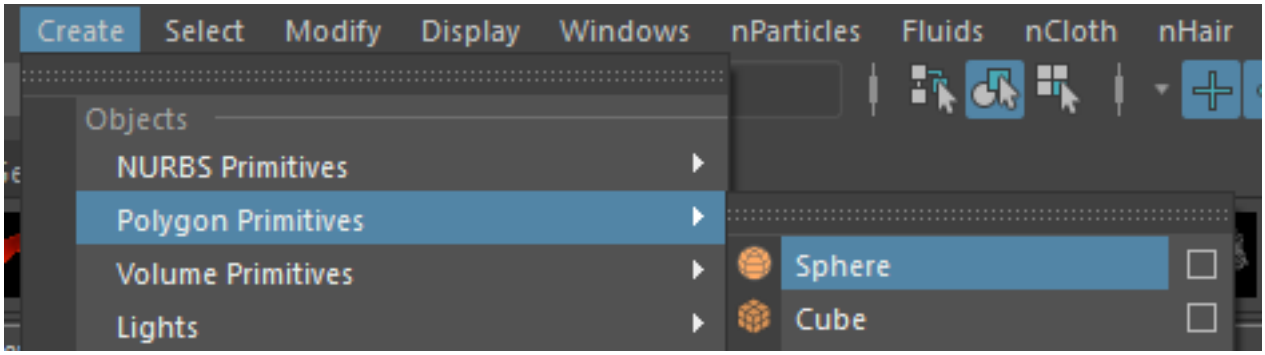


FIGURE III.3.1. – Créer une *polySphere* dans Maya.

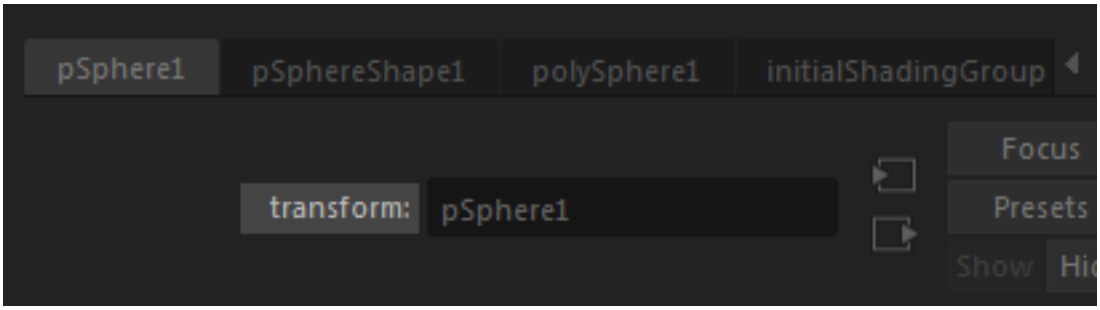
Maintenant exécutez :

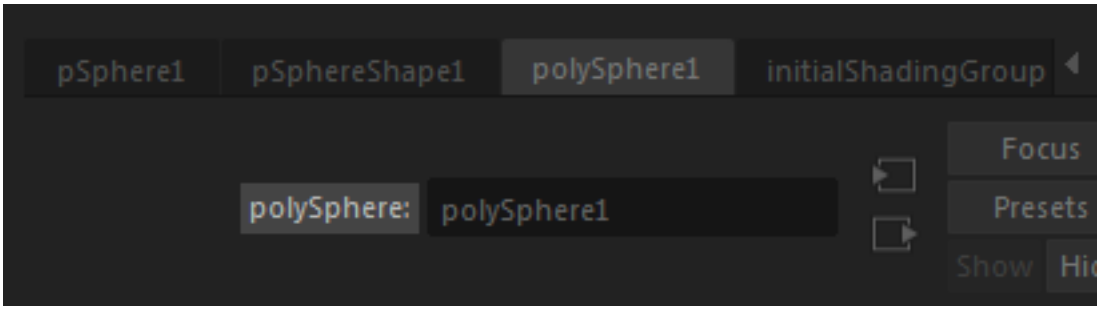
```
mc.nodeType('pSphere1')
# Result: u'transform' #
```

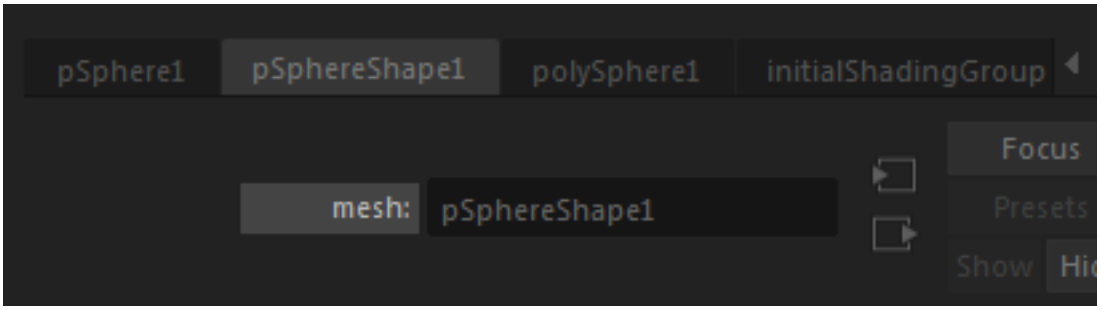
Comme vous pouvez le constater, le nœud s'appelant `pSphere1` est de type `tansform`.

Faites des essais en modifiant le nom du nœud dont vous souhaitez vérifier le type.

Notez que le type des nœuds est accessible dans l'*Attribute Editor* :







C'est à peu près tout ce qu'il y a à dire de cette commande…

Quoi ? Une chapitre pour ça ?

Vous vous doutez qu'il y a anguille sous roche pas vrai ?

Ici, la commande est simpliste… Mais il y a énormément de commandes, chacune disposant de pleins d'arguments possibles, et certains arguments (`edit` et `query` entre autres) changent parfois le sens des autres arguments. Alors comment faire dans le cas des commandes plus compliquées ? Vous pensez que les développeurs ont des cerveaux de fou ? Tapent des commandes super complexes, comme ça, les doigts dans le nez ? 🍊

Certains peut-être… Mais la grande majorité utilise ~~leur clavier~~ la documentation. 🤳

`nodeType()` était une commande simplissime que je vous ai présentée non pas dans le but de savoir s'en servir (vous avez vu, c'est basique…), mais comme prétexte à l'apprentissage de la documentation.

? «l'apprentissage de la documentation» ? Mais c'est si dur que ça d'utiliser la documentation ? 🍊

Ne vous inquiétez pas ! Apprendre à s'en servir ne prend pas plus de quelques minutes ! Si c'est la première fois que vous faites du script, sachez que la documentation est un véritable **outil** de travail. Savoir comment y trouver une information est fondamental pour gagner du temps.

Alors allons-y ! Allez dans *Help/Autodesk Maya Help* :

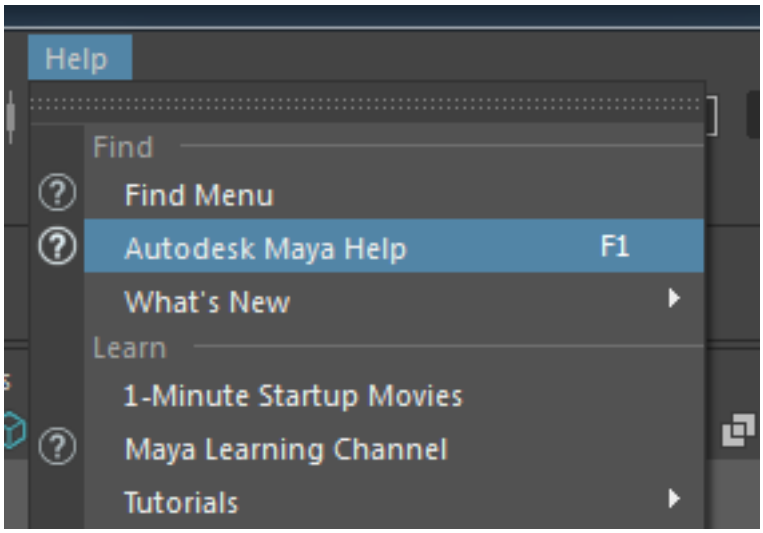


FIGURE III.3.2. – Accéder à l'aide dans Maya.

En bas à gauche, allez dans *Technical Documentation/Python Commands* :

Vous devriez obtenir une page qui ressemble à ça :

En haut, tapez le nom de la commande qui vous intéresse (dans notre cas tapez `nodetype`) :

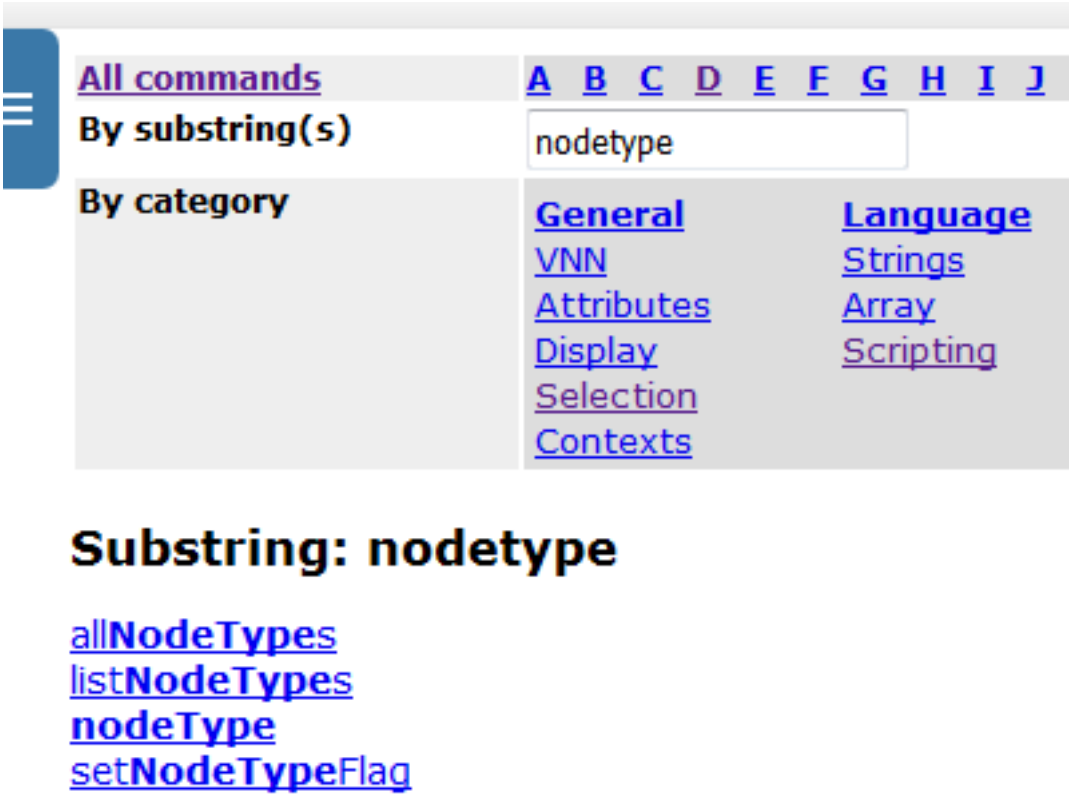


FIGURE III.3.3. – *Un concentré de technologie!*

Cliquez sur la commande et vous devriez retomber sur une page qui ressemble à ça :

command (Python) *MEL version*

nodeType

In categories: General, Attributes

Show frames

Go to: Synopsis. Return value. Related. Flags. Python examples.

Synopsis

```
nodeType( string , [apiType=boolean], [derived=boolean], [inherited=boolean], [isTypeName=boolean])
```

Note: Strings representing object names and arguments must be separated by commas. This is not depicted in the synopsis.

nodeType is undoable, **NOT queryable**, and **NOT editable**.

This command returns a string which identifies the given node's type.

When no flags are used, the unique type name is returned. This can be useful for seeing if two nodes are of the same type.

When the *api* flag is used, the MFn::Type of the node is returned. This can be useful for seeing if a plug-in node belongs to a given class. The *api* flag cannot be used in conjunction with any other flags.

When the *derived* flag is used, the command returns a string array containing the names of all the currently known node types which derive from the node type of the given object.

When the *inherited* flag is used, the command returns a string array containing the names of all the base node types inherited by the the given node.

If the *isTypeName* flag is present then the argument provided to the command is taken to be the name of a node type rather than the name of a specific node. This makes it possible to query the hierarchy of node types without needing to have instances of each node type.

Return value

string
string[]

Related

addAttr, aliasAttr, attributeInfo, deleteAttr, getClassification, objExists, objectType, renameAttr

Flags

apiType, derived, inherited, isTypeName

Long name (short name)	*Argument types*	Properties
apiType (api)	boolean	C

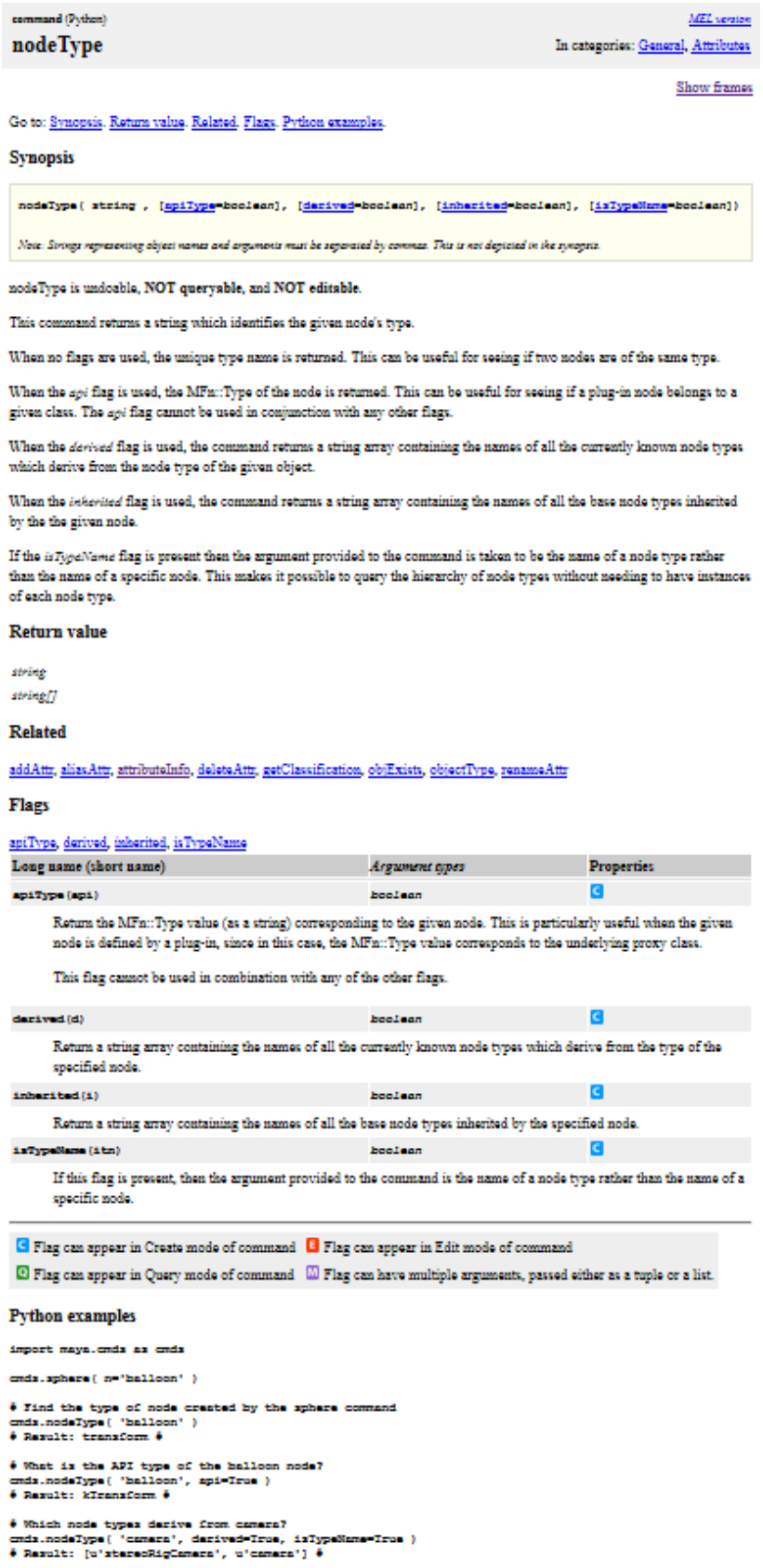
Return the MFn::Type value (as a string) corresponding to the given node. This is particularly useful when the given node is defined by a plug-in, since in this case, the MFn::Type value corresponds to the underlying proxy class.

FIGURE III.3.4. – *Y a des gens qui pense que la doc' elle est mauvaise, qu'elle est méchante, qu'elle ne changera jamais, bref, qu'elle est mauvaise...*

? Fichtre, c'est vraiment à ch...

Tatata ! C'est très vilain d'utiliser ces mots-là !

La documentation de chaque commande est organisé en blocs :



command (Python) — MEL version

nodeType

In categories: General, Attributes

Show frames

Go to: Synopsis, Return value, Related, Flags, Python examples.

Synopsis

```
nodeType( string , [apiType=boolean], [derived=boolean], [inherited=boolean], [isTypeName=boolean])
```

Note: Strings representing object names and arguments must be separated by commas. This is not depicted in the synopsis.

nodeType is undoable, **NOT queryable**, and **NOT editable**.

This command returns a string which identifies the given node's type.

When no flags are used, the unique type name is returned. This can be useful for seeing if two nodes are of the same type.

When the *api* flag is used, the MFn::Type of the node is returned. This can be useful for seeing if a plug-in node belongs to a given class. The *api* flag cannot be used in conjunction with any other flags.

When the *derived* flag is used, the command returns a string array containing the names of all the currently known node types which derive from the node type of the given object.

When the *inherited* flag is used, the command returns a string array containing the names of all the base node types inherited by the the given node.

If the *isTypeName* flag is present then the argument provided to the command is taken to be the name of a node type rather than the name of a specific node. This makes it possible to query the hierarchy of node types without needing to have instances of each node type.

Return value

string
string[]

Related

addAttr, aliasAttr, attributeInfo, deleteAttr, getClassification, objExists, objectType, renameAttr

Flags

apiType, derived, inherited, isTypeName

Long name (short name)	*Argument types*	Properties
apiType(api)	boolean	C
Return the MFn::Type value (as a string) corresponding to the given node. This is particularly useful when the given node is defined by a plug-in, since in this case, the MFn::Type value corresponds to the underlying proxy class. This flag cannot be used in combination with any of the other flags.		
derived(d)	boolean	C
Return a string array containing the names of all the currently known node types which derive from the type of the specified node.		
inherited(i)	boolean	C
Return a string array containing the names of all the base node types inherited by the specified node.		
isTypeName(itn)	boolean	C
If this flag is present, then the argument provided to the command is the name of a node type rather than the name of a specific node.		

C Flag can appear in Create mode of command — E Flag can appear in Edit mode of command
Q Flag can appear in Query mode of command — M Flag can have multiple arguments, passed either as a tuple or a list.

Python examples

```
import maya.cmds as cmds

cmds.sphere( n='balloon' )

# Find the type of node created by the sphere command
cmds.nodeType( 'balloon' )
# Result: transform #

# What is the API type of the balloon node?
cmds.nodeType( 'balloon', api=True )
# Result: kTransform #

# Which node types derive from camera?
cmds.nodeType( 'camera', derived=True, isTypeName=True )
# Result: [u'stereoRigCamera', u'camera'] #
```

En haut à gauche le nom de la commande, en haut à droite un accès vers la version de la documentation en MEL ainsi que les catégories à laquelle elle appartient (le lien `MEL version`).

Gardez bien en tête la position de ce lien, car vous en aurez besoin plus tard, quand nous convertirons les commandes MEL en Python.

III.3.1.1. *Synopsis*

Ici vous trouverez un gros bloc jaune qui liste les arguments et la manière dont on appelle la commande :

Synopsis

```
nodeType( string , [apiType=boolean], [derived=boolean], [inherited=boolean],
[isTypeName=boolean])
```

Note: Strings representing object names and arguments must be separated by commas. This is not depicted in the synopsis.

Juste en dessus il y une explication souvent assez claire, parfois longue, de ce que fait la commande :

nodeType is undoable, **NOT queryable**, and **NOT editable**.

This command returns a string which identifies the given node's type.

When no flags are used, the unique type name is returned. This can be useful for seeing if two nodes are of the same type.

When the *api* flag is used, the MFn::Type of the node is returned. This can be useful for seeing if a plug-in node belongs to a given class. The *api* flag cannot be used in conjunction with any other flags.

When the *derived* flag is used, the command returns a string array containing the names of all the currently known node types which derive from the node type of the given object.

When the *inherited* flag is used, the command returns a string array containing the names of all the base

III.3.1.2. *Return value*

Ce court bloc résume ce que renvoie la commande :

Return value

string

string[]

Le type des valeurs renvoyées dépend d'une certain nombre de choses, souvent expliquées ici.

III.3.1.3. *Related*

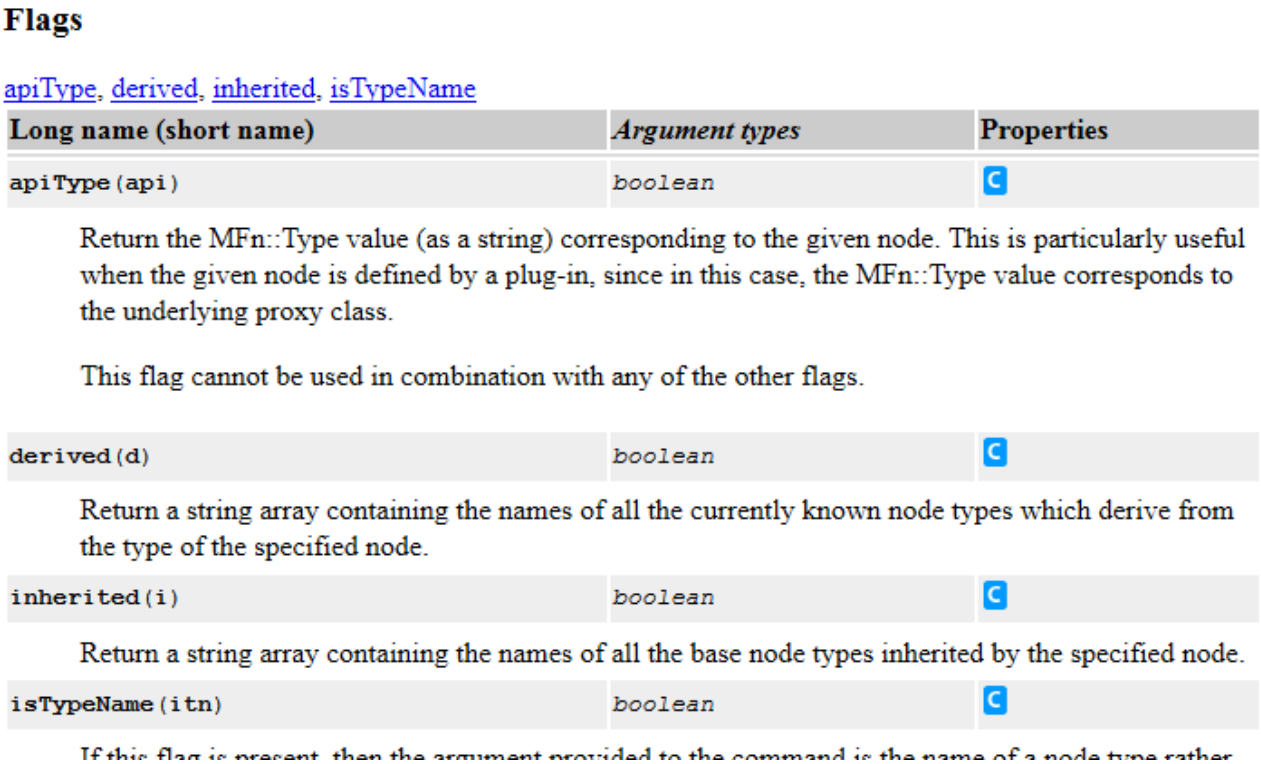
Related

addAttr, aliasAttr, attributeInfo, deleteAttr, getClassification, objExists, objectType, renameAttr

Ce bloc propose une liste d'autres commandes en lien (plus ou moins) direct avec la commande courante.

III.3.1.4. *Flags* (arguments)

Ce bloc contient la documentation de chacun des arguments possibles de la commande. C'est souvent là-dedans qu'on passe pas mal de temps.



Flags

apiType, derived, inherited, isTypeName

Long name (short name)	*Argument types*	Properties
apiType(api)	*boolean*	C

Return the MFn::Type value (as a string) corresponding to the given node. This is particularly useful when the given node is defined by a plug-in, since in this case, the MFn::Type value corresponds to the underlying proxy class.

This flag cannot be used in combination with any of the other flags.

derived(d)	*boolean*	C

Return a string array containing the names of all the currently known node types which derive from the type of the specified node.

inherited(i)	*boolean*	C

Return a string array containing the names of all the base node types inherited by the specified node.

isTypeName(itn)	*boolean*	C

If this flag is present, then the argument provided to the command is the name of a node type rather

FIGURE III.3.5. – Flags de la commande `nodeType`.

Vous avez chacun des arguments sous la forme longue et courte :

`apiType(api)` veut dire que l'argument peut être passé soit sous la forme :

```
1 mc.nodeType('pSphere1', apiType = True)
```

Soit sous la forme :

```
1 mc.nodeType('pSphere1', api = True)
```

Les deux versions faisant la même chose.

Je vous invite à toujours utiliser la version longue d'un argument afin de conserver une idée assez claire de ce que fait la commande, surtout quand vous y revenez après plusieurs semaines/mois.

```
1 mc.nodeType('pSphere1', derived = True)
```

Étant quand même plus clair que :

```
1 mc.nodeType('pSphere1', d = True)
```

Il y a de rares cas ou la version longue d'un argument rentre en conflit avec un mot clef de Python. C'est le cas de `import` de la commande `file()` qu'il est impossible d'utiliser. Dans ce cas, il faut se contenter de la version courte `i`.

Notez, dans le tableau, une colonne `Argument Type` qui détermine le type de la variable que vous devez passer à l'argument de la commande.

Gardez la colonne `Properties` pour plus tard, on ne s'y attarde pas pour le moment.

OK, mais c'est quoi `apiType`, `derived`, `inherited`

Ouh là, malheureux, que vous allez vite en besogne !

Ne vous préoccupez pas de ça pour l'instant. Ce sont des notions qui ne servent qu'avec l'API de Maya.

Je vous invite à essayer ces arguments, mais nous ne les expliquerons pas ici.

III.3.1.5. *Python examples*

Ce dernier bloc est sûrement le premier que vous devez regarder quand vous découvrez une commande :

Python examples

```
import maya.cmds as cmds

cmds.sphere( n='balloon' )

# Find the type of node created by the sphere command
cmds.nodeType( 'balloon' )
# Result: transform #

# What is the API type of the balloon node?
cmds.nodeType( 'balloon', api=True )
# Result: kTransform #

# Which node types derive from camera?
cmds.nodeType( 'camera', derived=True, isTypeName=True )
# Result: [u'stereoRigCamera', u'camera'] #
```

Contenu masqué n°1

Ce sont des exemples très commentés et clairs à copier-coller dans votre *Script Editor* et qui peuvent fonctionner instantanément.

C'est souvent une excellente base pour se familiariser avec une commande, en particulier les commandes concernant l'interface (pour créer des boutons et tout). Je vous invite vraiment à vous en servir, surtout si vous débutez.

III.3.2. ls(), ou comment lister les nœuds de sa scène

La première *vraie* commande que nous allons voir est `ls()` (abréviation de *list*).

Avant de commencer, pensez à garder la page de la documentation officielle sous le coude .

Par défaut, cette commande renvoie tous les nœuds présents dans la scène (en Python : Une `list` de `str`). C'est la combinaison des arguments que vous allez utiliser qui va «filtrer» le résultat.

> Dans une console Linux, `ls` est la commande permettant de lister le contenu d'un dossier. Il y a fort à parier que les développeurs de Maya ont choisi ces deux lettres par habitude.

Allez-y, prenez une scène vide et exécutez ceci :

```
1 mc.ls()
```

La sortie devrait vous afficher quelque chose comme ça :

```
1 [u'time1', u'sequenceManager1', u'renderPartition', ...
```

Ce sont (presque) tous les nœuds présents dans votre scène Maya !

III.3.2.1. La sélection

Voici un des arguments dont vous vous servirez sûrement souvent.

Vous souhaitez lister les nœuds sélectionnés ? Rien de plus simple !

Créez une sphère (*Create/Polygon Primitives/Sphere*):

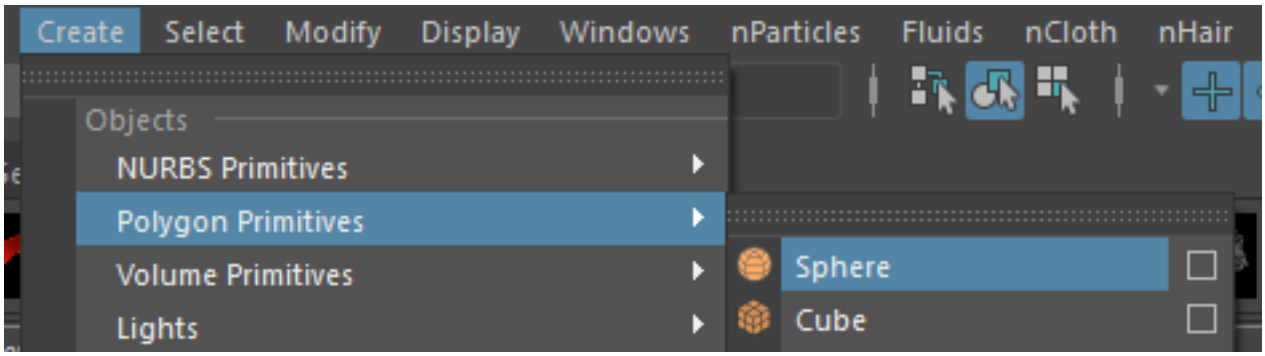


FIGURE III.3.6. – Créer une *polySphere* dans Maya.

Sélectionnez-la puis exécutez :

```
1 mc.ls(selection=True)
2 # Result: [u'pSphere1'] #
```

Bien entendu, vous pouvez sélectionner plusieurs nœuds et l'argument `selection` vous renverra les nœuds dans l'ordre de leur sélection.

Créez plusieurs sphères et essayez.

III.3.2.2. Des étoiles pleins les yeux

Vous pouvez aussi utiliser les étoiles `*` (*wildcard* en anglais) pour essayer de filtrer des nœuds par nom.

Lancez ceci :

```
1 mc.ls("*Sphere*")
2 # Result: [u'pSphere1', u'pSphereShape1', u'polySphere1'] #
```

Une sphère, trois nœuds ? Intéressant… Mais on y reviendra plus tard !

Il est important de noter qu'à partir du moment où vous mettez une étoile `*` dans le nom du nœud, Maya va aller chercher tous les nœuds qui correspondent à l'expression.

À l'inverse, s'il n'y a aucune étoile, Maya considère que vous lui donnez un «nom unique» (notion importante pour la suite).

III.3.2.3. Par type

On peut préciser le `type` des nœuds qu'on veut lister. Voici trois exemples :

III.3.2.3.1. transform

Renverra une liste contenant tous les nœuds de type `transform` de la scène :

```
mc.ls(type="transform")
# Result: [u'front', u'pSphere1', u'persp', u'side', u'top'] #
```

Remarquez comme il vous retourne également les caméras.

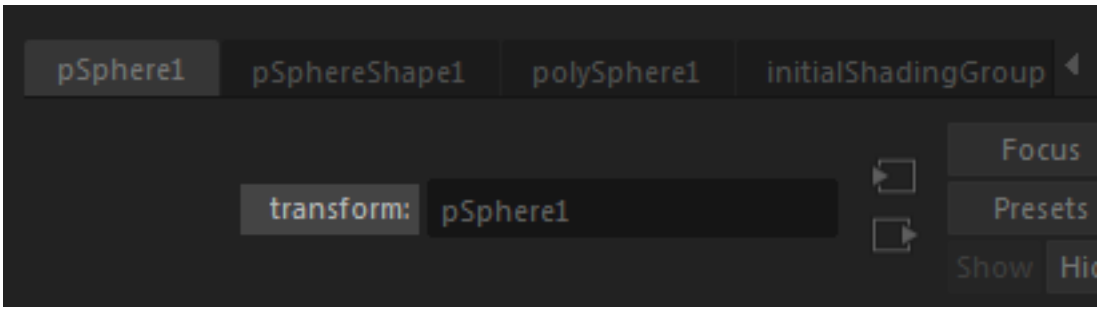
III.3.2.3.2. mesh

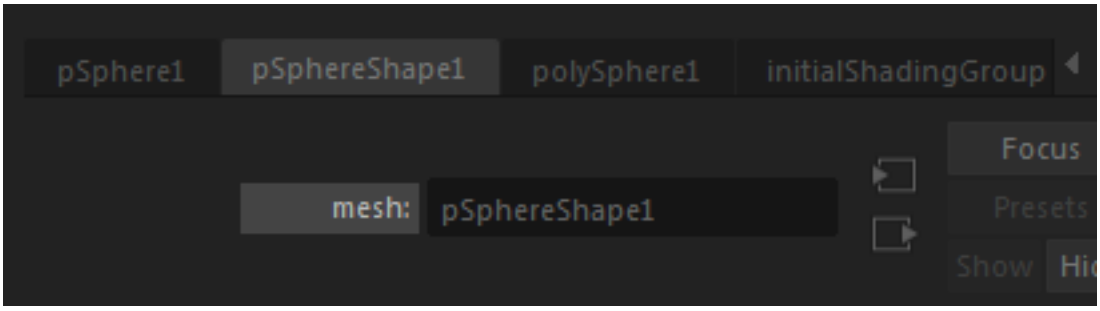
```
mc.ls(type="mesh")
# Result: [u'pSphereShape1'] #
```

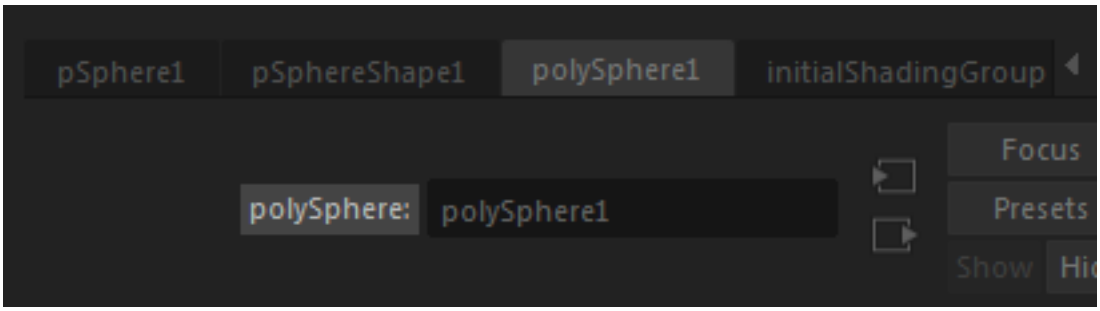
III.3.2.3.3. polySphere

```
mc.ls(type="polySphere")
# Result: [u'polySphere1'] #
```

Au passage, le type d'un nœud est visible dans l'*Attribute Editor*, à côté du nom :







Trois types, trois nœuds, une sphère ? Intéressant… Mais on y reviendra plus tard ! (bis)

III.3.2.3.4. unknown

Un type intéressant est le type `unknown`. Il apparaît souvent quand on ouvre des scènes Maya ASCII (extension *.ma*) contenant des nœuds de plugins externes (moteurs de rendu par exemple) non disponibles dans votre session Maya lors de l'ouverture. Maya découvre un nœud dont il n'a pas le plugin et, ne réussissant pas à l'interpréter, le met de ce type spécial : `unknown`.

Si vous faites un :

```
1 mc.ls(type="unknown")
```

Et que ça vous renvoie quelque chose, vous aurez potentiellement des soucis.

III.3.2.3.5. Cumuler les types

On peut cumuler les types en les passant sous la forme d'une liste !

```
1 mc.ls(type=["transform", "mesh"])
2 # Result: [u'front', u'pSphere1', u'persp', u'side', u'top',
      u'pSphereShape1'] #
```

Ici on demande indistinctement tous les nœuds de type `transform` ainsi que `mesh`.

Deux types, deux nœuds. Coïncidence ? Je ne crois pas… Mais on y… OK, j'arrête !

III.3.2.3.6. Exclure certains types

Si on peut demander certains types particuliers, comment peut-on demander l'inverse, à savoir, demander à ce que certains types ne soient explicitement *pas* renvoyés ?

Pour ceci, on utilise l'argument `excludeType` :

```
1 mc.ls(excludeType="transform")
```

Vous vous en doutez, on peut aussi lui passer une liste :

```
1 mc.ls(excludeType=["transform", "mesh"])
```

III.3.2.3.7. Note sur les différents *types*

Dans Maya, les types sont organisés en hiérarchies. Ainsi, le type `surfaceShape` regroupe `mesh` et `nurbsSurface`. C'est assez anecdotique, mais il peut arriver, sur des scènes ~~crados~~ un peu complexes, que le résultat obtenu ne soit pas nécessairement celui attendu.

L'argument `exactType` est fait pour renvoyer les nœuds du type spécifié sans prendre en compte les types descendants. Si vous voulez éviter les surprises, utilisez `exactType`.

III.3.2.4. Par attribut

On peut également aller chercher tous les nœuds ayant un attribut particulier. Par exemple :

```
1 mc.ls("*.focalLength")
```

renverra tous les nœuds ayant l'attribut `focalLength` (soit les quatre caméras par défaut si vous n'en avez pas créé d'autres).

Notez que l'étoile `*` ne fonctionne pas avec les attributs :

```
1 mc.ls("*.film*")
```

ou même :

```
1 mc.ls("perspShape.film*")
```

renverront tous deux une liste vide, bien que des attributs commençant par `film` existent sur les caméras (exemples : `filmTranslateH`, `filmRollValue`, etc).

Je vous concède que cette utilisation est alléchante, mais je vous déconseille de l'utiliser dans vos scripts. Vous pouvez avoir plusieurs types de nœuds différents ayant un nom

d'attribut similaire. Exemple : les nœuds `polySphere`, `polyTorus`, `makeNurbCylinder` et plein d'autres ont tous un attribut `radius`. Si vous récupérez les nœuds par attribut aveuglément, vous risquez de modifier des choses sans vraiment savoir quoi. Il est souvent plus propre de récupérer un/des nœuds en fonction de leur type qu'en fonction de leurs attributs. La seule exception s'applique aux attributs customs, et là encore, il est préférable de ne créer des attributs spécifiques que sur un type de nœud particulier (en gros, de travailler proprement).

III.3.2.4.1. Le principe de *noms uniques*

Un chouillat plus avancé maintenant :

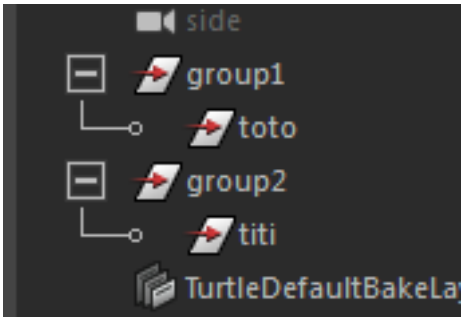
— Repartez d'une scène vide (*Ctrl+n* ou *File/New Scene*).
— Créez un groupe (*Ctrl+g* ou *Edit/Group* ou encore *Create/Empty Group*).
— Appelez-le *toto*.
— Mettez-le dans un groupe (Sélectionnez *toto* puis faites *Ctrl+g*).
— Dupliquez le groupe (*Ctrl+d* ou *Edit/Duplicate*).
— Enfin, renommez-le *toto* du deuxième groupe en *titi*.

Il va falloir vous y habituer tout de suite, mes noms de variable favoris sont *toto tutu* et *titi* (parce que les touches *u*, *i* et *o* se suivent sur le clavier).

En développement, les variables souvent utilisées lors des exemples s'appellent des variables syntaxiques . Les anglo-saxons utilisent beaucoup les variables «foo» et «bar» dans leurs codes d'exemples. Ceci vient de l'acronyme militaire «FUBAR» pour «fucked up beyond all/any recognition/repair/reason/redemption» qu'on pourrait poliment traduire par «un merdier innommable».

Mais continuons ! 🍊

Votre scène devrait ressembler à ceci :



Sélectionnez *toto* et faites :

```
1 mc.ls(selection=True)
```

En toute logique, vous obtenez :

```
1 [u'toto']
```

Et bien oui : vous sélectionnez *toto*, vous demandez la sélection, forcément vous obtenez *toto*. L'inverse serait troublant non ? 🍊

Maintenant, renommez *titi* en *toto*:

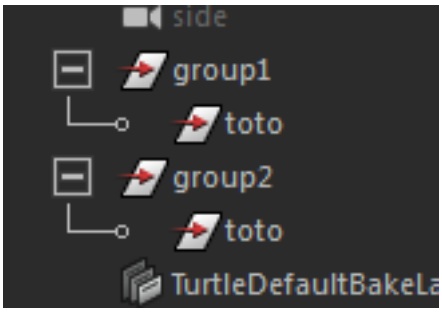


FIGURE III.3.7. – Vous voyez le piège venir? 🍊

Re-sélectionnez le premier *toto* et refaites :

```
1 mc.ls(selection=True)
```

Vous obtenez :

```
1 [u'group1|toto']
```

? Mais ! Mais ! Mon nœud a changé de nom ? Je n'y ai pourtant pas touché! 🍊

Ne paniquez pas ! Tout va bien ! 🍊

Il y a une chose très importante à comprendre : les commandes Maya sont prévues pour communiquer entre elles. Les commandes retournant un ou plusieurs nœuds renvoient le «nom unique le plus court possible». Ici, nous avons deux nœuds nommés *toto*. Si, vous passez *toto* en argument d'une autre commande, Maya ne saura pas auquel des deux nœuds vous faites référence.

Vous ne me croyez pas, essayez par vous-même :

```
1 mc.select('toto')
```

Et :

```
# Error: More than one object matches name: toto
# Traceback (most recent call last):
#   File "<maya console>", line 1, in <module>
# ValueError: More than one object matches name: toto #
```

Le message est explicite : plusieurs nœuds correspondent à *toto* et il ne sait pas quoi faire.

En cela, les développeurs de Maya ont décidé d'utiliser le concept de «nom unique» pour être sûr que les nœuds renvoyés par les commandes sont utilisables par d'autres commandes.

Sauf mention contraire, une commande renvoyant un ou plusieurs nœuds ne renvoient pas son nom, mais son identifiant, à savoir : son «nom unique le plus court possible».

Très bien mais c'est quoi ce trait `|` ?

Ça s'appelle un *pipe* (prononcer à l'anglaise paillp'). Sous Windows, les hiérarchies de dossiers s'écrivent en utilisant le séparateur *back slash* "\" : "C:\Windows\System". Sous Linux, les hiérarchies de dossiers s'écrivent en utilisant le séparateur *slash* "/" : "/usr/share". Et bien sous Maya, les hiérarchies de nœud s'écrivent en utilisant le séparateur *pipe* "|" : "|geo|tete|oeilG".

Il n'y a rien à comprendre de plus !

Mais comment je fais si je souhaite simplement avoir le nom du nœud ?

La commande `ls()` ne le fait pas (pour les raisons évoquées précédemment) mais c'est très simple en Python via la commande split() :

```
for n in mc.ls(selection=True) :
    print n.split("|")[-1]
```

Si ce dernier est «le plus court possible», cela sous-entend qu'il y a un nom «le plus long possible» ?

Vous avez tout compris ! Et c'est l'argument `long` qui s'en charge.

```
mc.ls(selection=True, long=True)
```

Cet argument est supposé renvoyer le *nom long* (*long name* en anglais) mais je trouve qu'il est mal choisi. `fullPath` (*chemin complet* en français) aurait été plus approprié.

```
[u'|group1|toto']
```

Vous remarquerez le pipe `|` devant *group1*. Tout nom de nœud qui commence par `|` est un chemin complet. Un chemin complet est par définition un nom unique.

Je vous recommande d'utiliser l'argument `long` à chaque fois que vous appelez `ls()`. Vous aurez ainsi la garantie d'avoir un chemin complet.

Gardez à l'esprit que tout peut se combiner avec `ls()`. C'est souvent LA commande que vous appellerez en premier dans vos scripts. N'hésitez pas à jeter un œil à ses arguments sur sa page de documentation, notez ceux qui vous semblent intéressants et expérimentez par vous-même. Croyez-moi, ce n'est pas du temps perdu pour vos futurs scripts.

III.3.3. getAttr() pour récupérer la valeur des attributs

Vous savez maintenant comment récupérer et filtrer les nœuds de votre scène.

Ici, nous allons nous servir de la commande `getAttr()` pour récupérer le contenu des *attributs* des nœuds.

Une fois encore (et il va falloir prendre l'habitude), gardez-la page de la documentation sous le coude.

On va commencer par un exemple simple.

Créez une *polySphere* dans votre scène puis faites :

```
mc.getAttr("pSphere1.translateX")
# Result: 6.034336531007678 #
```

Comme vous le voyez, vous récupérez la valeur de l'attribut `translateX` de la sphère. Rien de très compliqué vous en conviendrez.

Maintenant, faites :

```
mc.getAttr("pSphere1.translate")
# Result: [(6.034336531007678, 0.0, 3.0054366242372197)] #
```

Ça renvoie une liste contenant un unique tuple de trois valeurs correspondant respectivement à l'attribut `translateX`, `translateY` et `translateZ`.

? Pourquoi Maya renvoie une liste contenant un unique tuple et pas directement un tuple ?

Sûrement parce que `getAttr()` peut renvoyer plusieurs valeurs grâce à l'étoile.

III.3.3.1. L'étoile (encore !)

Si ça ce n'est pas de la transition…

Vous pouvez donc utiliser l'étoile. Ici, on va récupérer l'attribut *translate* de tous les nœuds en ayant un :

```
1 mc.getAttr("*.translate")
```

Ce qui me renvoie :

```
1 [(0.0, 100.1, 0.0),
2  (100.1, 0.0, 0.0),
3  (28.0, 21.0, 28.0),
4  (6.034336531007678, 0.0, 3.0054366242372197),  # << surement
       notre sphère… ;)
5  (0.0, 0.0, 100.1)]
```

Comme pour la commande `ls()`, ce n'est pas très pratique si vous écrivez des scripts, car il est difficile de savoir à quel nœud appartient quelle valeur. Ça peut être utile pour de l'introspection et du *debug* manuel.

III.3.3.2. Les types

À l'instar des nœuds, les attributs ont des types respectifs. La variable que renvoie `getAttr()` dépend donc du type de l'attribut que vous demandez.

Et des types d'attributs, il y en a un paquet !

Allez sur la page de la documentation de addAttr() pour vous faire une petite idée. Ne soyez pas effrayé pour autant, c'est beaucoup plus simple que ça en a l'air.

Python dispose de moins de types qui englobent pas mal de ceux des attributs Maya. Par exemple, le type d'attributs `long`, `short` et `byte` sont toujours retournés sous la forme d'un `int` Python. Idem pour `float`, `double`, `doubleAngle`, `doubleLinear`, `time`, qui sont tous renvoyé sous la forme d'un `float` Python.

Allez, un tableau non exhaustif :

Type des attributs	Type Python

`long`, `short`, `byte` et `enum`	`int`
`float`, `double`, `doubleAngle`, `doubleLinear` et `time`	`float`
`string`	`string`
`stringArray`	`list` de `string`
`short2` et `long2`	`tuple` de deux `int`
`short3` et `long3`	`tuple` de trois `int`
`float2` et `double2`	`tuple` de deux `float`
`float3` et `double3`	`tuple` de trois `float`
`matrix`	`list` de seize `float`

Vous remarquerez que les types en Python sont simplement une agrégation de types principaux (`bool`, `int`, `float`, `str`, `list` et `tuple`). Ça rend les valeurs moins compliquées à manipuler.

Ajouter à ça qu'en pratique, c'est souvent les mêmes qui sont utilisés ! 🍊

? Intéressant, comment puis-je récupérer les types d'un attribut ?

Avec l'argument `type` pardi ! Reprenons nos exemples précédents :

```
1 mc.getAttr("pSphere1.translateX", type=True)
2 # Result: u'doubleLinear' #
```

```
1 mc.getAttr("pSphere1.translate", type=True)
2 # Result: u'double3' #
```

Et pour le fun : 🍊

```
1 mc.getAttr("pSphere1.rotateOrder", type=True)
2 # Result: u'enum' #
```

```
1 mc.getAttr("pSphere1.matrix", type=True)
2 # Result: u'matrix' #
```

Il est assez rare qu'on *demande* le type d'un attribut directement dans un script, mais on s'en sert souvent à la main, quand on cherche à en savoir plus sur un attribut donné.

Souvent, quand on connaît le type du nœud et le nom de l'attribut, on en déduit implicitement son type.

```
mc.getAttr("pSphere1.matrix")
# Result: [1.0,
 0.0,
 0.0,
 0.0,
 0.0,
 1.0,
 0.0,
 0.0,
 0.0,
 0.0,
 1.0,
 0.0,
 6.034336531007678,
 0.0,
 3.0054366242372197,
 1.0] #
```

? C'est quoi une matrice ?

Vous suivez, c'est bien.

Mais c'est un gros morceau, je ne vais pas vous l'expliquer maintenant. Notez toutefois les valeurs de position de la sphère vers la fin de la liste.

III.3.3.3. Les `enum`

Les attributs énumérés sont assez fréquents dans Maya. On s'en sert quand on souhaite avoir un attribut qui propose un choix parmi une liste établie. En interne, seul un numéro est stocké.

Par exemple, l'attribut `rotateOrder` est un petit menu (une *combo box* dans le jargon) qui propose plusieurs choix (`xyz` par défaut) :

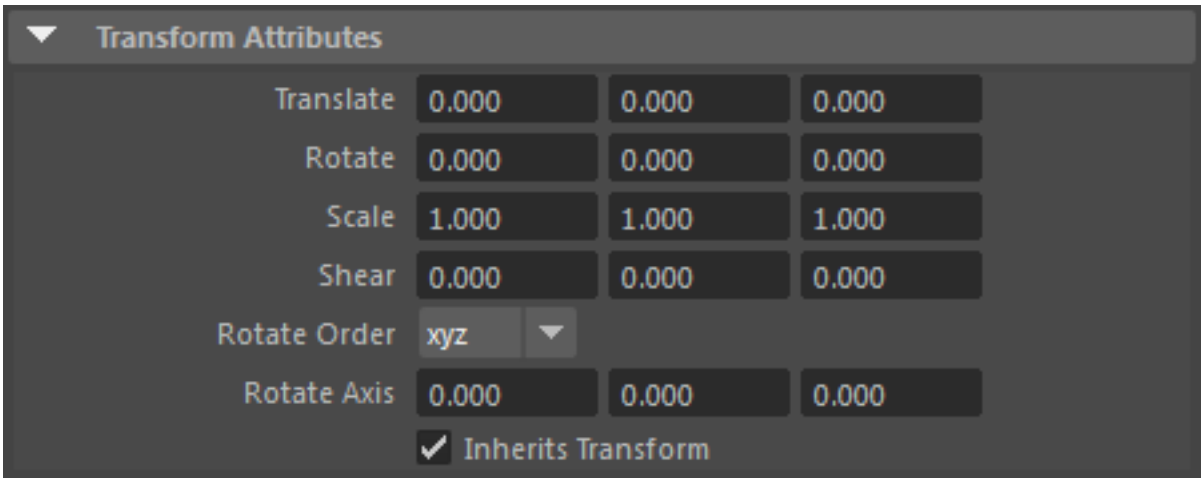


FIGURE III.3.8. – Les attributs de transformation

Essayez de récupérer la valeur de l'attribut `rotateOrder` :

```
1 mc.getAttr("pSphere1.rotateOrder")
2 # Result: 0 #
```

? Mais, `0` ce n'est pas la valeur de la boîte de dialogue... 🍊

On va y venir. 🍊

Essayez d'exécuter la commande plusieurs fois en choisissant différents ordres de rotation et observez le résultat.

? `1`, `2`, `3`, c'est bien gentil tout ça. Mais des petits numéros c'est pas super pratique non ? 🍊

En effet, c'est pourquoi l'argument `asString` existe ! 🍊

III.3.3.3.1. L'argument asString

C'est un argument spécial qui n'a d'effet que sur les attributs de type `enum`. Il renvoie la valeur de l'argument sous sa forme de `string`.

```
1 mc.getAttr("pSphere1.rotateOrder", asString=True)
2 # Result: u'xyz' #
```

Bien plus pratique me direz-vous ! 🍊

En effet. Mais il faut être vigilant :

> Cette `string` n'a de valeur informative que pour l'affichage de l'interface. Vous n'avez aucune garantie qu'elle ne change pas entre les versions de Maya.

En pratique cela pose rarement des soucis, mais il faut le savoir.

III.3.3.4. time

L'argument `time` permet d'évaluer la valeur d'un attribut a un temps donné. Animez le déplacement de votre sphère de l'image 1 a 10 puis faites :

```
for frame in range(1, 11):
    print "Frame:", frame, "| Translate:",
        mc.getAttr("pSphere1.translate", time=frame)
```

> Si vous ne comprenez pas ce qui se passe, vous ne savez pas ce qu'est une boucle ni la commande range() , il est temps de creuser votre tutoriel Python un peu plus.

Chez moi, cela donne :

```
Frame: 1 | Translate: [(0.0, 0.0, 0.0)]
Frame: 2 | Translate: [(2.6587184482341937, 0.853674501465477,
    -0.4781830218152393)]
Frame: 3 | Translate: [(9.784083036035243, 3.1415218913576712,
    -1.759713366780099)]
Frame: 4 | Translate: [(20.09991122219489, 6.453779151945815,
    -3.615063600597007)]
Frame: 5 | Translate: [(32.330012250318575, 10.380680627726392,
    -5.814704811428484)]
Frame: 6 | Translate: [(45.19820996878731, 14.512465352569638,
    -8.129110714174649)]
Frame: 7 | Translate: [(57.42831088281122, 18.43936679171449,
    -10.328751904484744)]
Frame: 8 | Translate: [(67.74414203556556, 21.751625004831574,
    -12.184102671857667)]
Frame: 9 | Translate: [(74.86950399907123, 24.039471552101986,
    -13.465632544830653)]
Frame: 10 | Translate: [(77.52822107810779, 24.893145613938714,
    -13.943815320389263)]
```

Ceci vous évite d'avoir à vous déplacer dans le temps (*spoil* : via la commande `currentTime()` 🍎) avant de faire votre `getAttr()`. C'est à la fois puissant et risqué. Puissant, car cela évite bien des manipulations; risqué, car cela ne marche pas toujours suivant l'organisation de votre graphe de nœuds.

III.3.4. setAttr() pour modifier des attributs

On a réussi à récupérer des nœuds ainsi que les valeurs des attributs qui les composent. Et maintenant ? Et bien on va les modifier ! Mouhahaha ! Et pour ça on va utiliser la commande `setAttr()`.

C'est parti ! 🍹

Repartez d'une scène vide et créez une sphère :

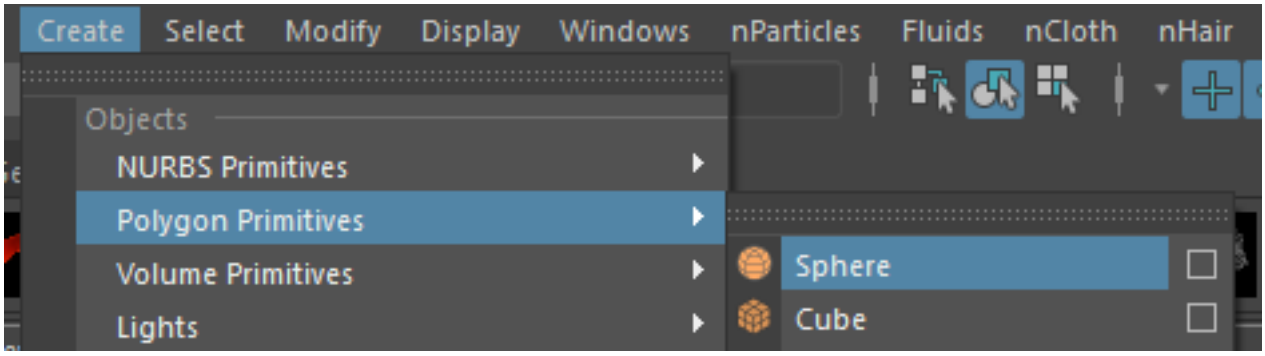


FIGURE III.3.9. – Créer une *polySphere* dans Maya

? Euh… Tu ne nous donnes pas la page de la documentation avant de commencer ? 🍊

Palsambleu ! C'est qu'ils suivent les petits agrumes ! La voici . 🍊

Continuons, ne m'interrompez plus et exécutez-moi ceci sur le champ !

```
1 mc.setAttr("pSphere1.translateX", 2)
```

Comme vous pouvez le voir, la valeur de l'attribut `translateX` s'est mis à 2 et votre sphère a bougé.

Essayez de trouver des attributs et modifiez-les :

```
1 mc.setAttr("polySphere1.radius", 5)
2 mc.setAttr("polySphere1.subdivisionsAxis", 8)
3 mc.setAttr("polySphere1.subdivisionsHeight", 4)
```

```
4 mc.setAttr("pSphere1.visibility", False)
```

Je ne vous explique pas ce que font ces lignes, à vous de travailler un peu.

III.3.4.1. Les attributs en chaîne de caractères

Comme vous pouvez le voir, vous pouvez modifier les attributs numériques facilement. Il existe une petite nuance concernant les attributs en chaîne de caractères (*string attributes* en anglais). Mais faisons un test. Vous voyez l'attribut *Current UVSet* ici :

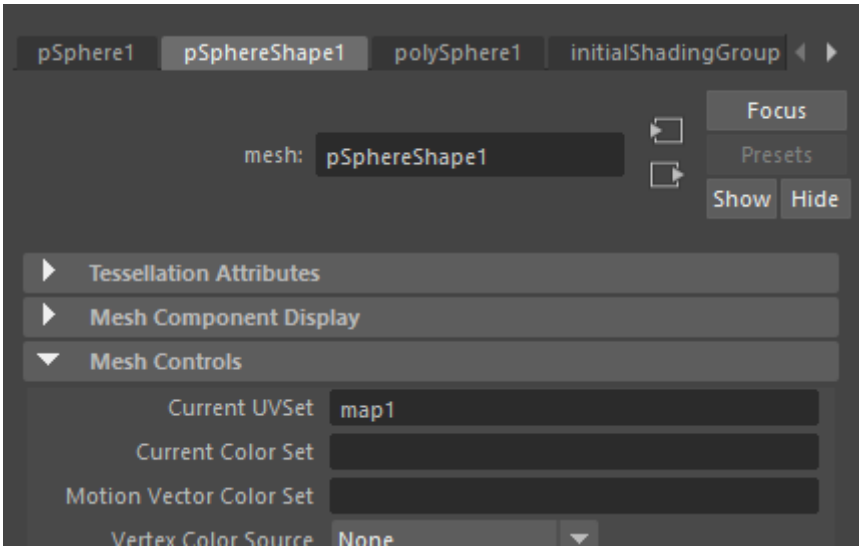


FIGURE III.3.10. – Current UVSet dans Maya

Son nom de code c'est `currentUVSet` (je vous expliquerai plus loin comment je l'ai eu).

Regardez cette ligne puis exécutez-la :

```
1 mc.setAttr("pSphereShape1.currentUVSet", "map2")
```

?

Je ne sais pas pourquoi, mais je sens que ça ne va pas marcher.

Mais pourquoi dis-tu ça ?

```
1 # RuntimeError: setAttr: 'pSphereShape1.currentUVSet' is not a
    simple numeric attribute.  Its values must be set with a -type
    flag. #
```

Tiens, quelle surprise ! 🍊

Plus sérieusement, prenez le temps de lire le message d'erreur, il est très clair et… Oh et puis allez ! Vous avez gagné un bloc bien lourd pour un point important :

> D'une manière générale, les messages d'erreur de Python ou de script sont assez clairs. Ils contiennent de précieuses informations et mettent souvent le doigt sur le problème principal. Si un jour vous avez un message d'erreur que vous ne comprenez pas, cherchez le sur le Net. Dans tous les cas, une fois la solution trouvée, relisez le message d'erreur pour faire le lien entre lui et votre solution, ceci vous permettra d'aller plus vite le jour ou vous re-rencontrerez ce message.

Revenons à notre souci : si la commande `setAttr()` fonctionne sans soucis avec les attributs numériques (`float`, `int`, `bool`, etc.), certains types (dont `str`) nécessitent qu'on spécifie le type via l'argument… `type` :

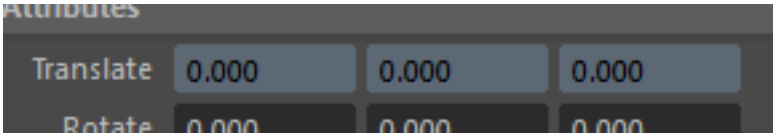
```
1 mc.setAttr("pSphereShape1.currentUVSet", "map2", type="string")
```

Ça marche beaucoup mieux déjà ! 🍊

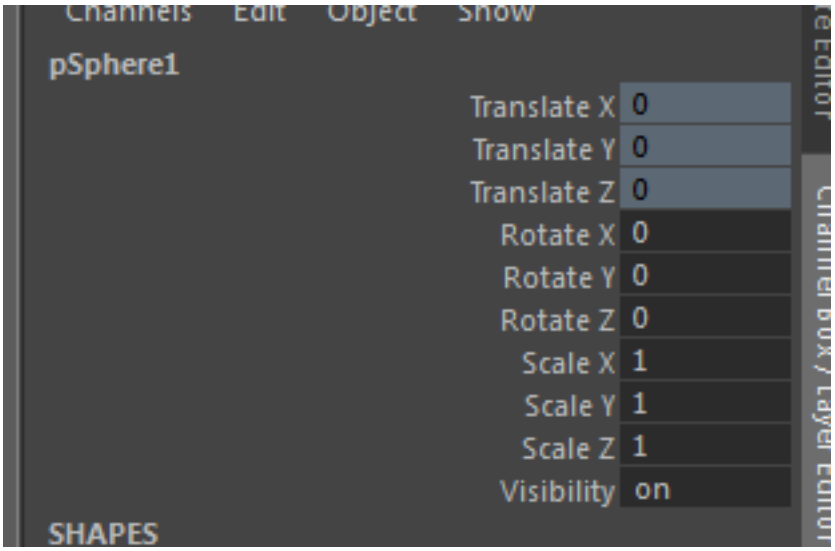
> Quand on souhaite exécuter un `setAttr()` sur un attribut en chaîne de caractères, il faut utiliser l'argument `type="string"`

III.3.4.2. Bloquer un attribut avec *lock*

Peut-être avez-vous déjà rencontré ceci :



ou cela :



Dans les deux cas, l'attribut `transform` est verrouillé (*locked* en anglais). On peut le faire depuis l'interface de Maya (Bouton droit de souris puis *(Un)Lock Attribute*).

Comme je vous en parle, c'est qu'on peut faire de même en script.

Voici comment verrouiller un attribut :

```
1 mc.setAttr("pSphere1.translate", lock=True)
```

Je vous laisse le plaisir de deviner comment déverrouiller l'attribut.

Au passage, vous pouvez également utiliser l'argument `lock` avec `getAttr()` pour savoir si un attribut est verrouillé ou non.

Un aspect intéressant concernant un attribut verrouillé c'est qu'il ne peut pas être déverrouillé dans une scène en référence. Ce qui veut dire que si vous faites un *rig* de personnage et que vous l'importez en référence dans une scène en vu de l'animer, les animateurs ne pourront pas déverrouiller l'attribut. C'est très pratique !

III.3.5. select() pour... Je ne sais plus pour quoi en fait...

Je ne pouvais pas finir ce premier chapitre sans vous montrer cette commande très simple à utiliser et qui vous parlera.

La commande `select()` (doc ici) permet de scripter tout ce qui a trait à la sélection.

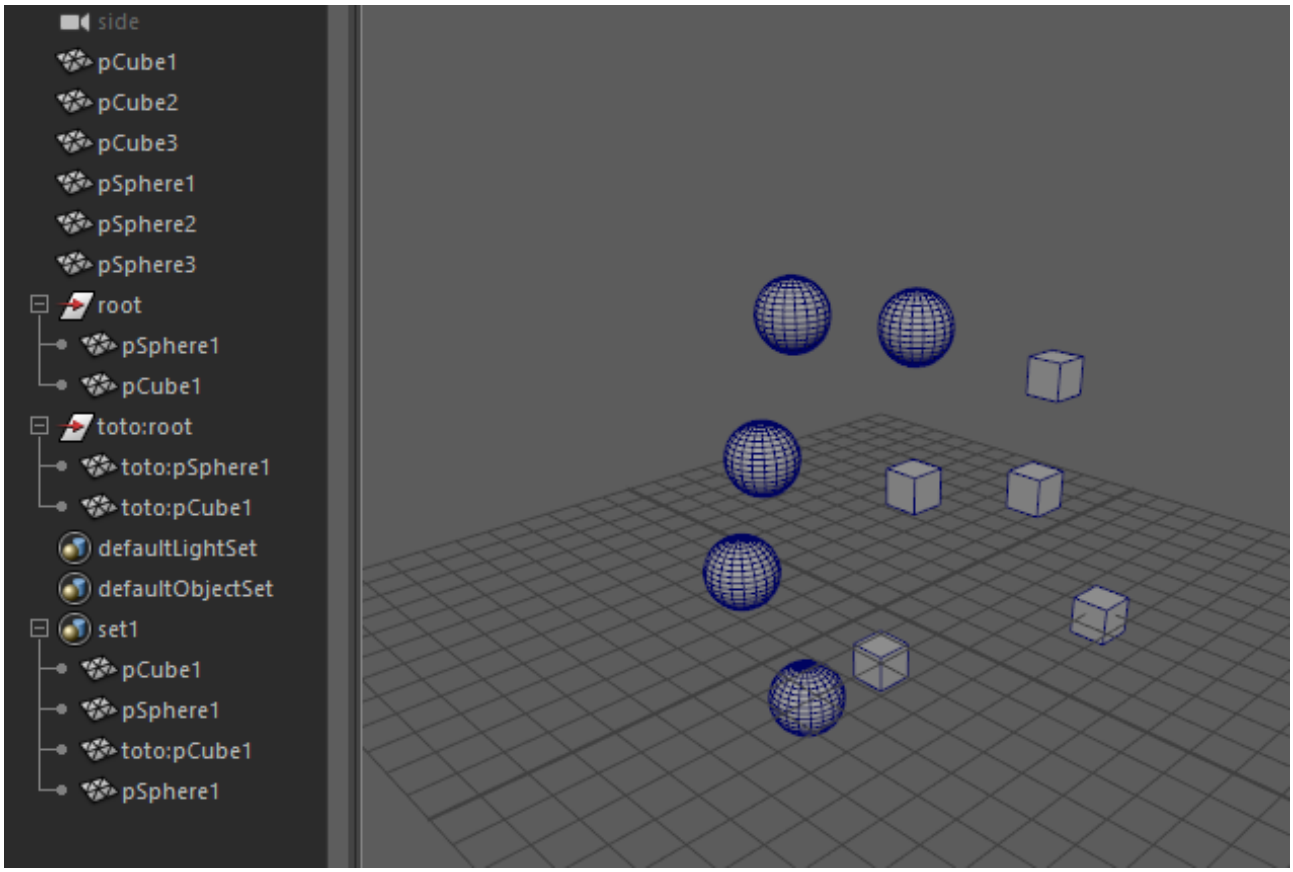


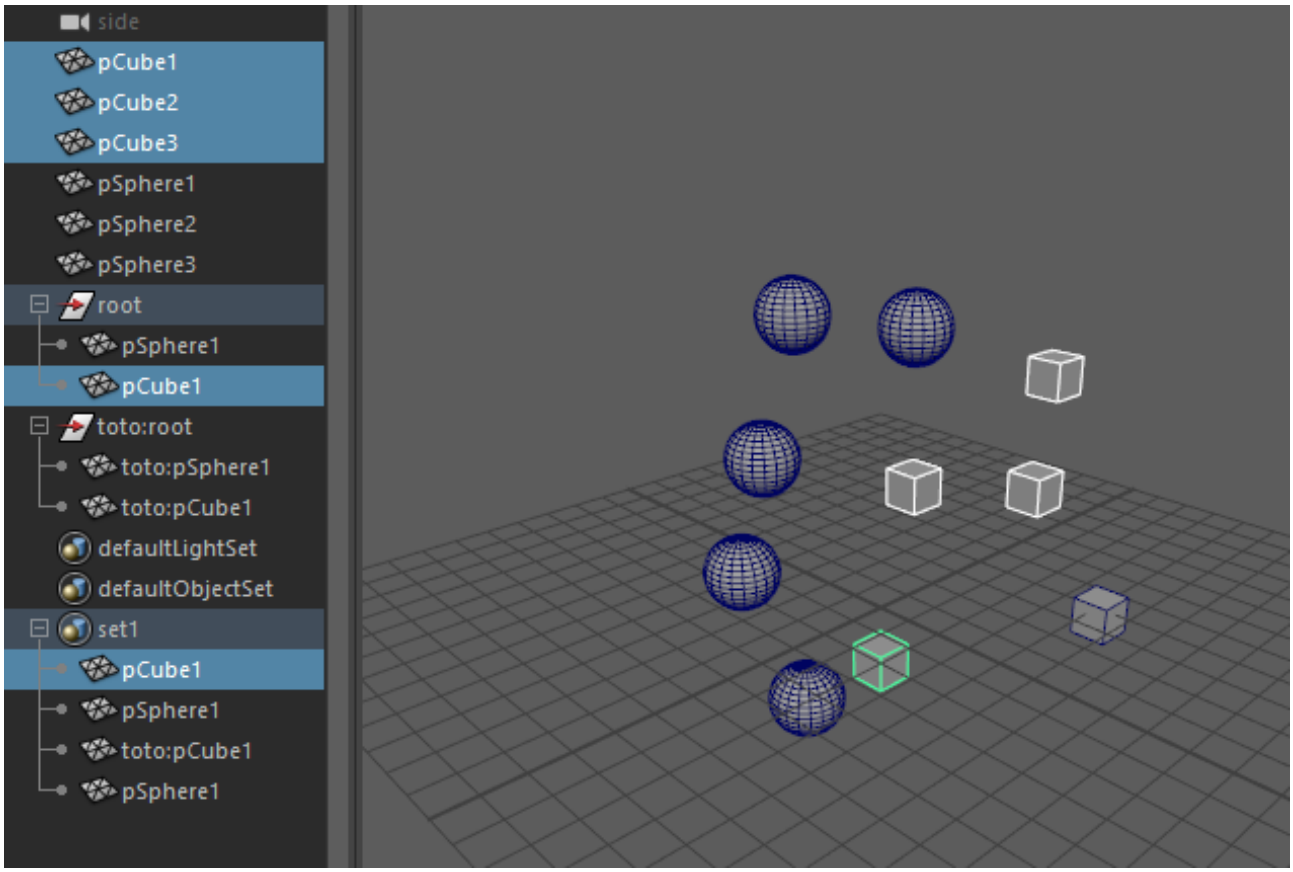
FIGURE III.3.11. – Notre scène avant nos diverses sélections.

III.3.5.1. Mon étoile filante...

Notez qu'elle est compatible avec l'étoile `*`. Vous pouvez donc faire des sélections par rapport à des noms particuliers. Par exemple:

```
1 mc.select('pCube*')
```

vous sélectionnera tous les nœuds commençant par `pCube`:

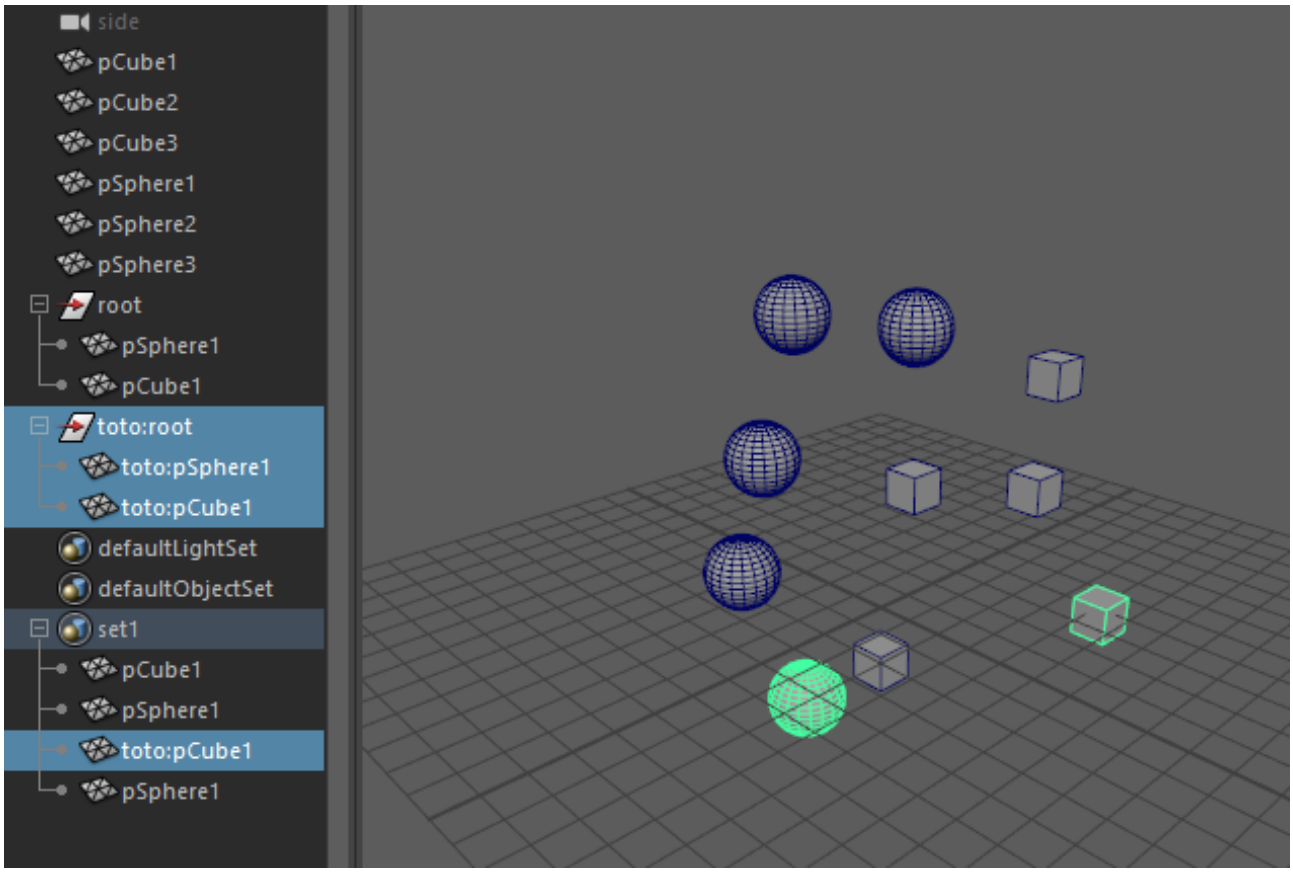


Plus aucun prétexte pour oublier des nœuds mal nommés dans vos scènes. 🍊

Et comme on parle de vraies scènes de production, voici la suite:

```
1 mc.select('toto:*')
```

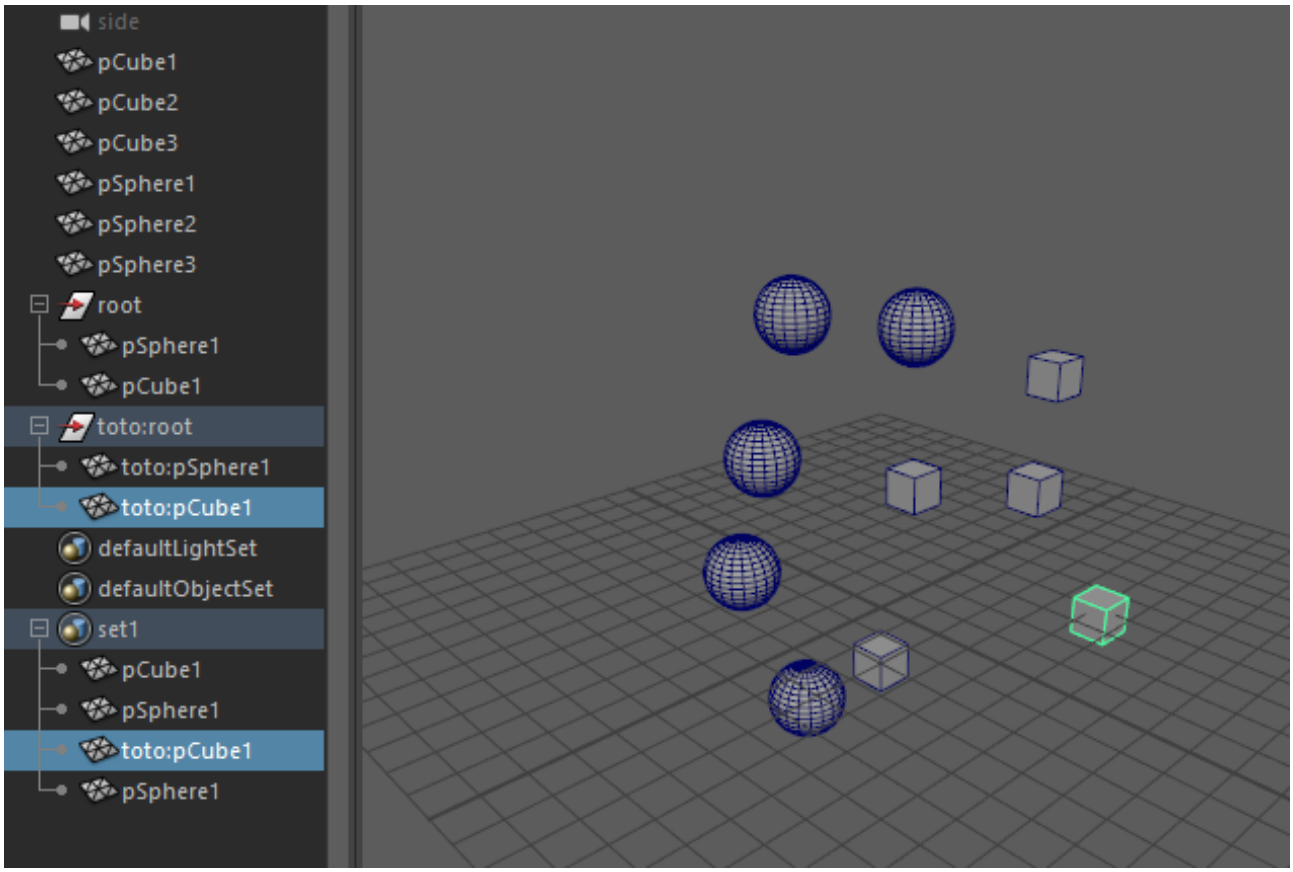
qui sélectionne tout le contenu d'un *namespace* donné:



Et une dernière variation que je n'ai pas voulu vous montrer avec la commande `ls()` pour ne pas trop vous surcharger:

```
1 mc.select('*:pCube*')
```

Ceci sélectionnera tous les nœuds commençant par `pCube` dans n'importe quel *namespace* (ici, il n'y a que `toto`):

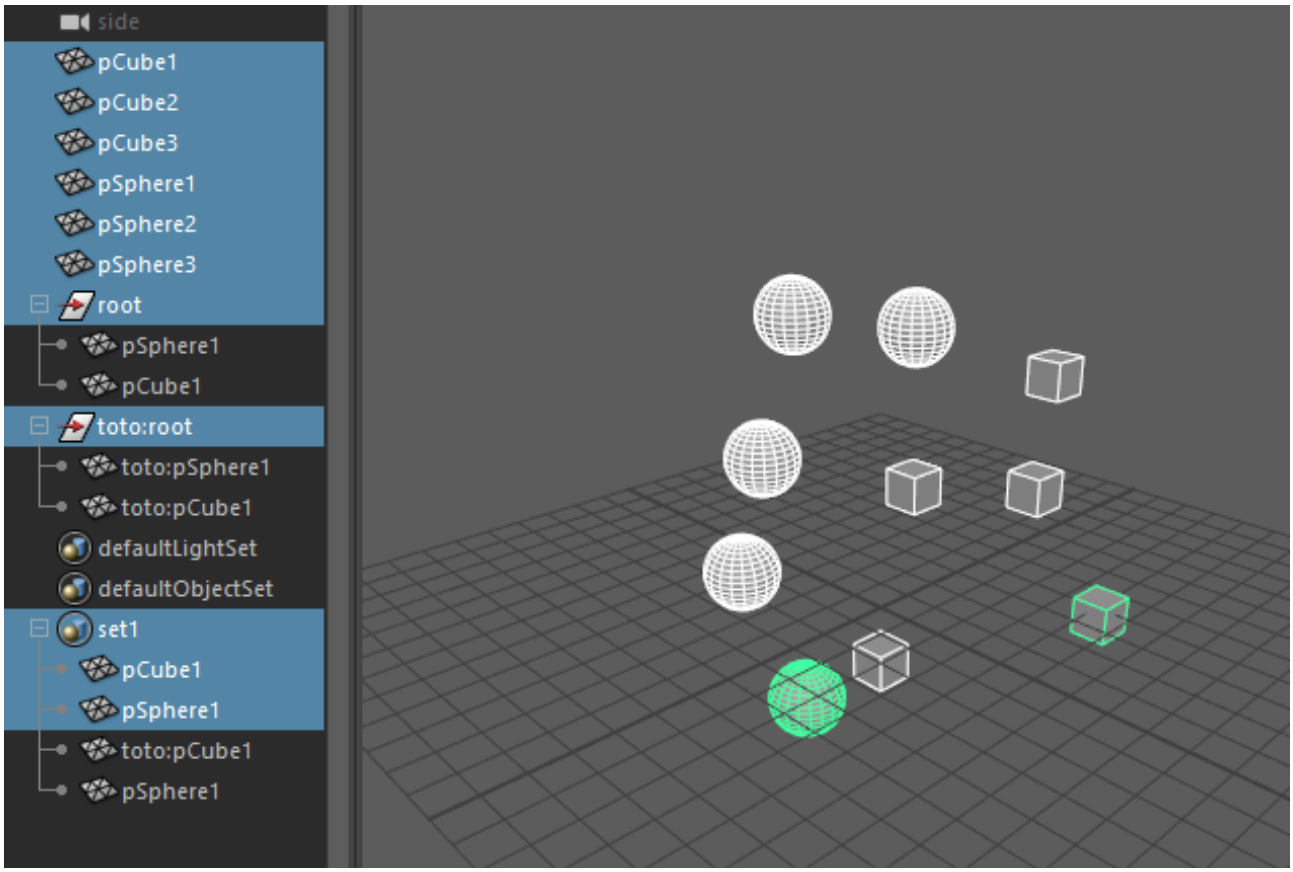


Je suis sûr que vous l'aimez déjà celle-là.

III.3.5.2. Sélectionner tous les nœuds

Celui-là vous vous ne vous en servirez sûrement jamais tout seul:

```
1 mc.select(all=True)
```



Notez comment cette commande ne sélectionne pas le contenu des groupes. 🍊

En revanche, combiné à d'autres options il peut devenir intéressant.

III.3.5.3. Sélectionner tous les nœuds visibles

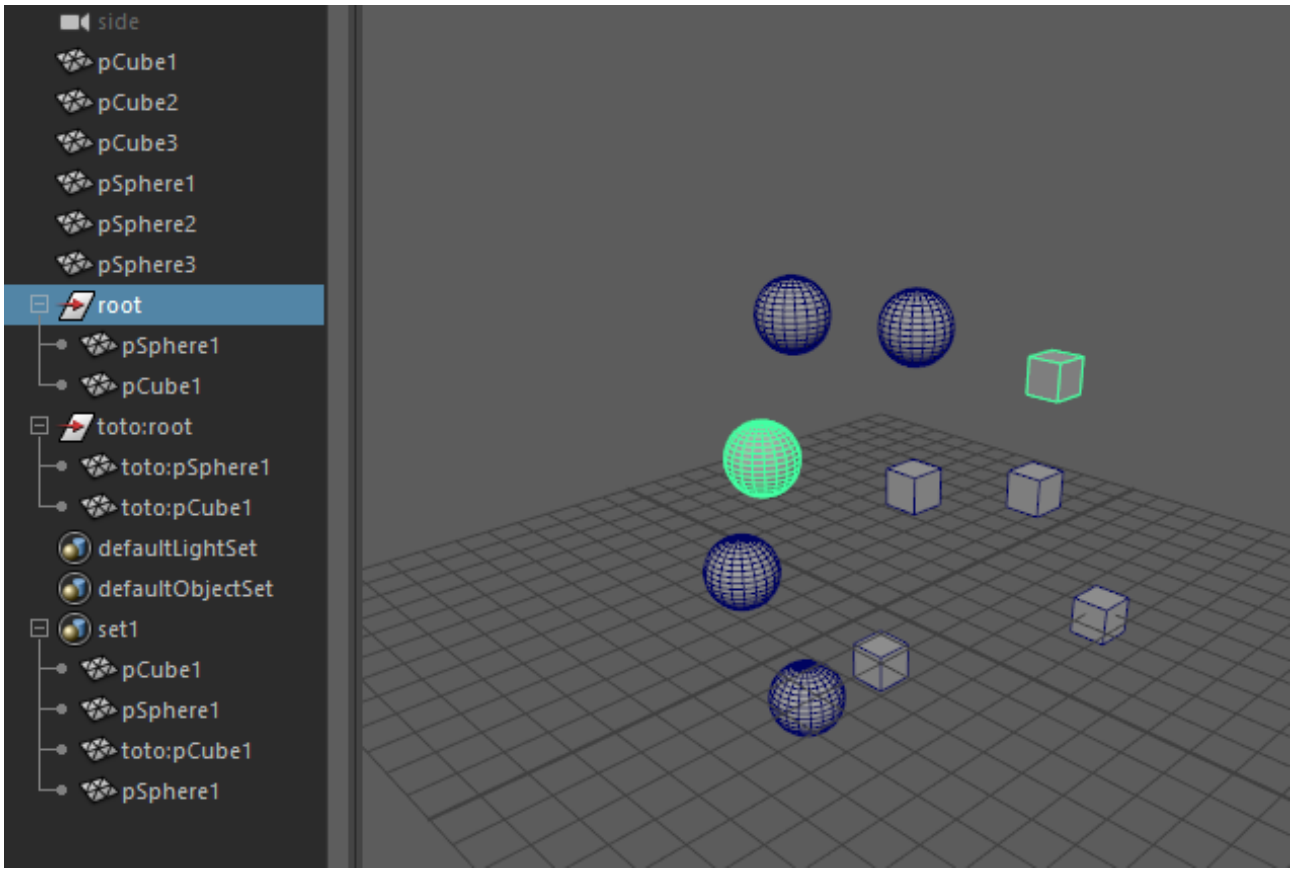
```
1 mc.select(all=True, visible=True)
```

Celui-ci je vous le montre juste pour vous donner un exemple de combinaison entre `all` et un autre argument.

III.3.5.4. Sélectionner le contenu de la hiérarchie

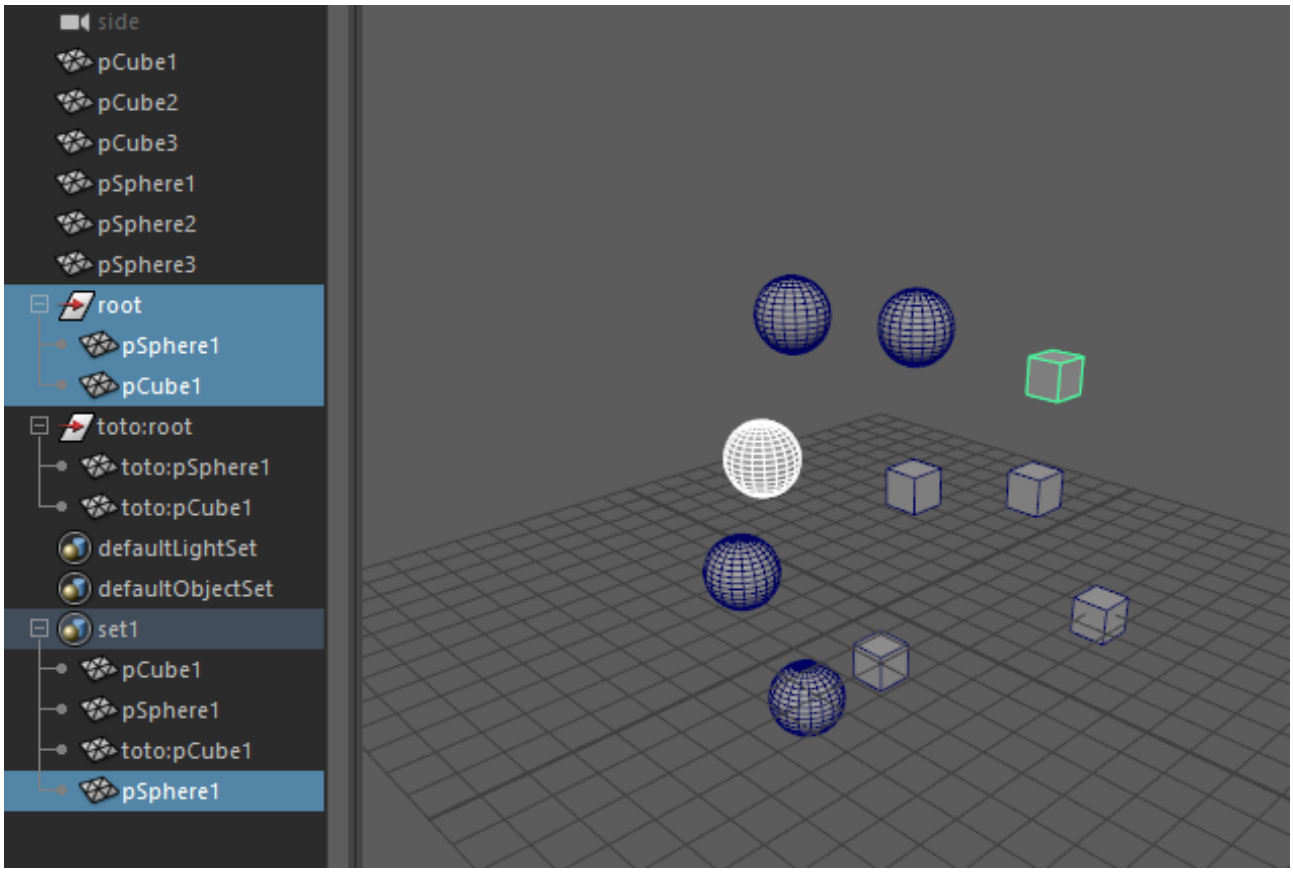
Mais à quoi peut-il bien servir celui-là?

Comme vous le savez sûrement, vous pouvez sélectionner un nœud sans ses enfants:

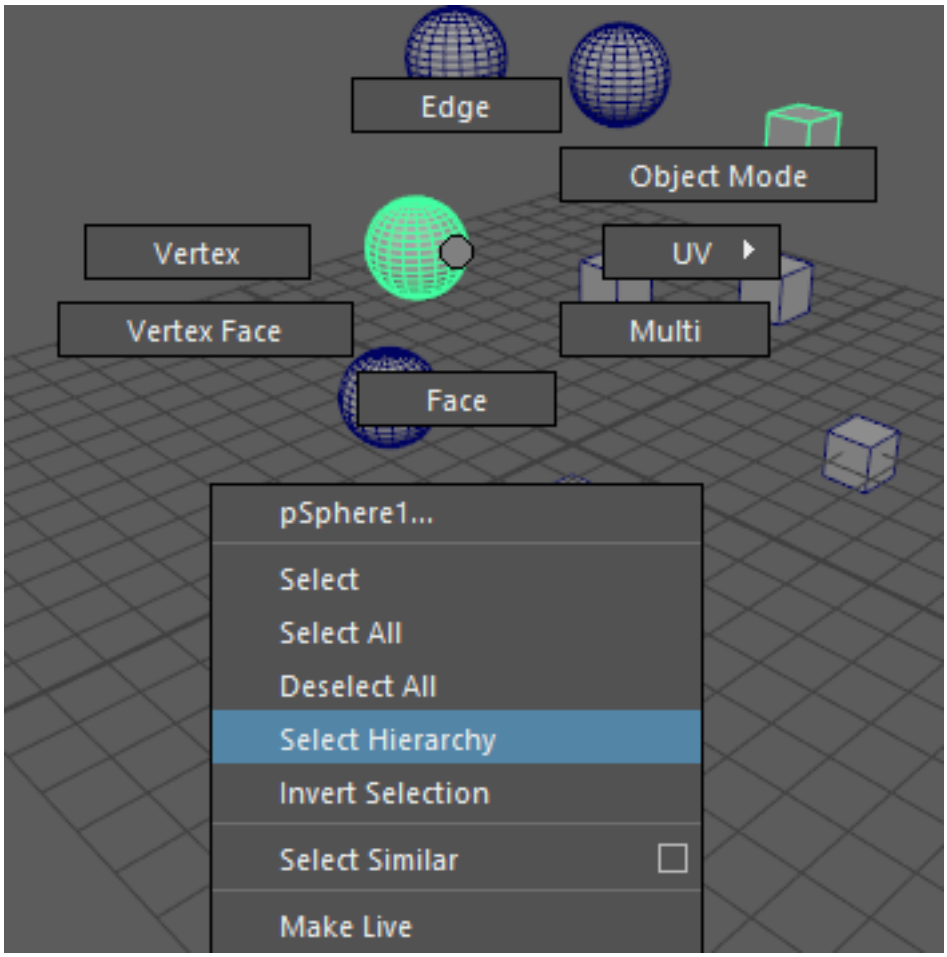


Cette commande s'assure de sélectionner tous les enfants des nœuds sélectionnés.

```
1 mc.select(hierarchy=True)
```



C'est l'équivalent du *Select hierarchy* de Maya:



III.3.5.5. Vider la sélection

Il n'est pas rare de vouloir scripter des choses qui agissent sur la sélection. On peut devoir vider la sélection. Cela se fait comme ceci:

```
1 mc.select(clear=True)
```

III.3.5.6. Alterner une sélection

```
1 mc.select('pCube*', toggle=True)
```

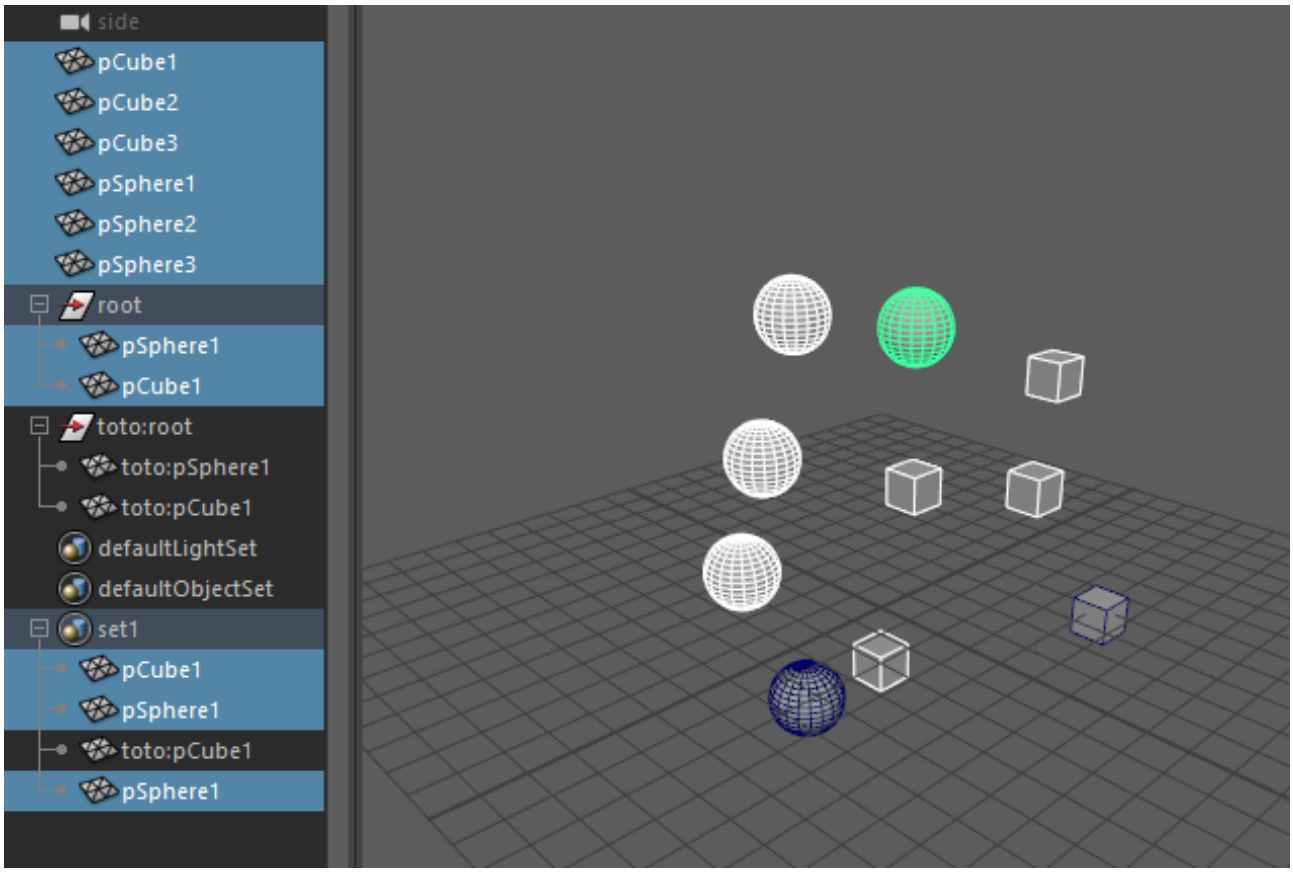
Chaque appel à cette commande sélectionnera puis désélectionnera alternativement les nœuds commençant par `pCube`.

III.3.5.7. Sélectionner et désélectionner un (ou plusieurs) nœud(s)

Aussi évident que ça puisse paraître:

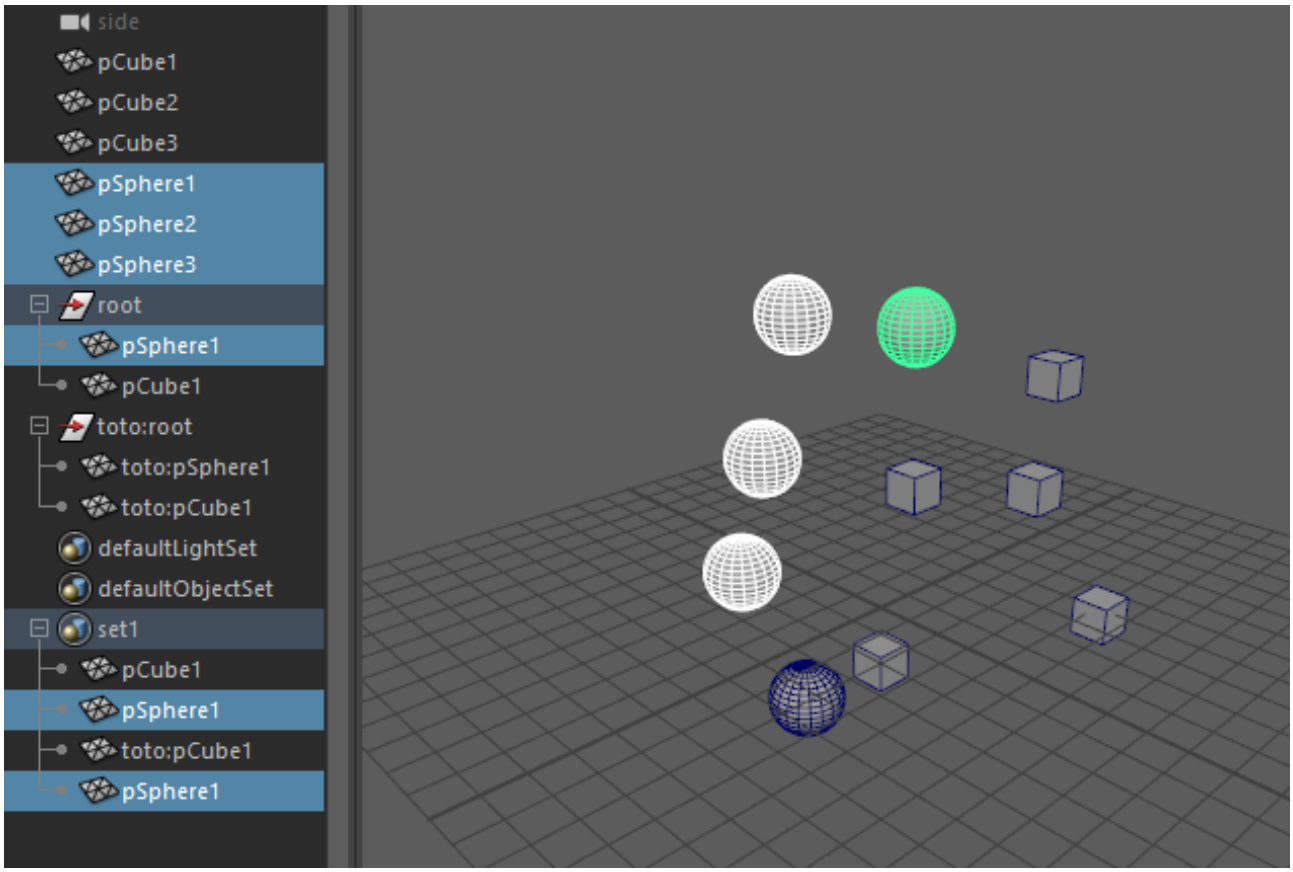
```
1 mc.select('pCube*', add=True)
2 mc.select('pSphere*', add=True)
```

Ici, la commande sélectionne tous les nœuds commençant par `pCube` et dans la seconde commande, l'argument `add` ajoute tous les nœuds commençant par `pCube` à la sélection:




```
1 mc.select('pCube*', deselect=True)
```

Et ici, l'argument `deselect` désélectionnera les nœuds commençant par `pCube`:



Vous auriez presque pu la deviner tout seul celle-là pas vrai? 🍊

III.3.5.8. Remplacer la sélection

L'argument `replace` remplace la sélection quelle qu'elle eût été avant:

```
1 mc.select('pCube*', replace=True)
```

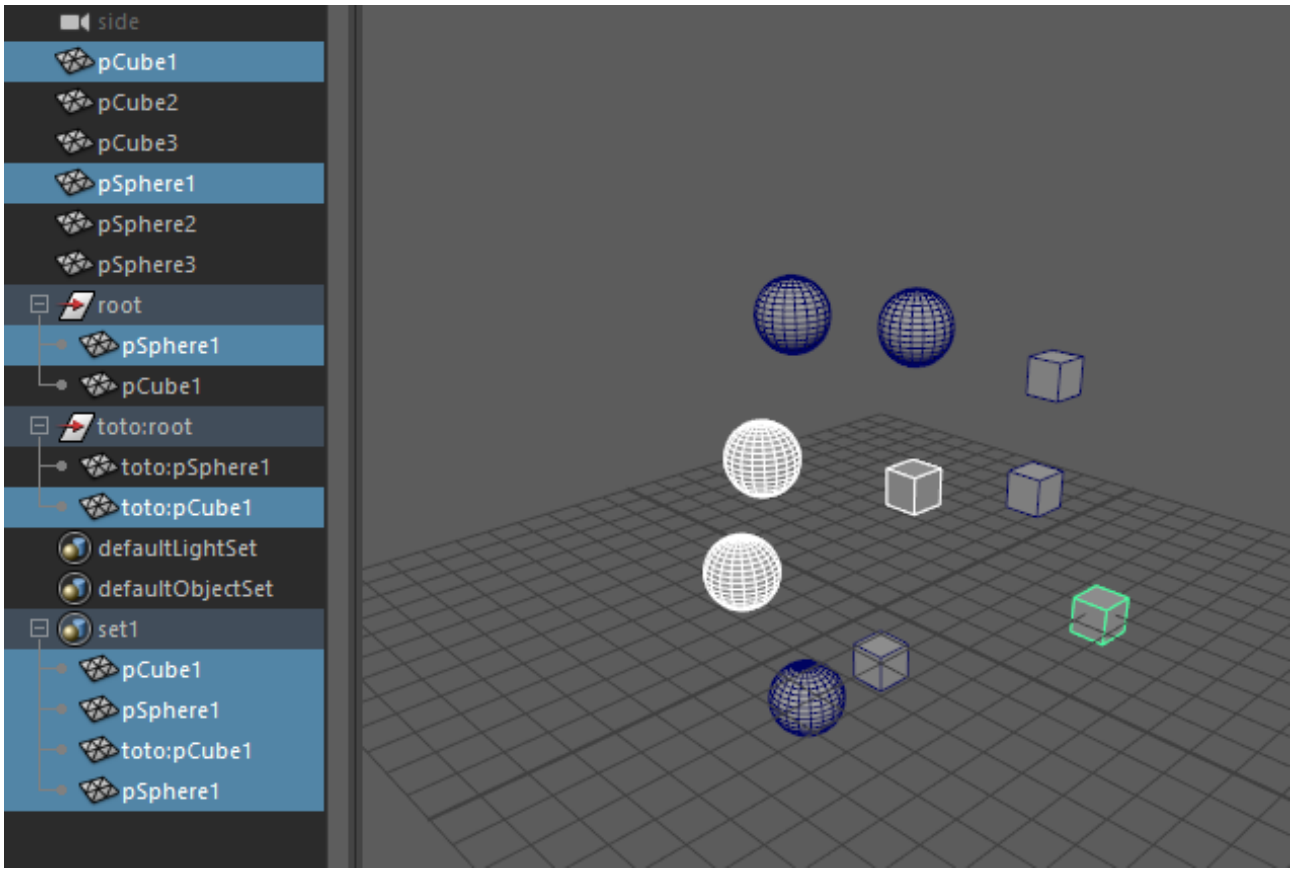
III.3.5.9. Les *sets*

Comme vous le savez peut-être, Maya dispose de nœuds de type `set` qui sont un peu particuliers, car ils permettent de stocker… une sélection.

Donc quand on sélectionne un nœud de set:

```
1 mc.select('set1')
```

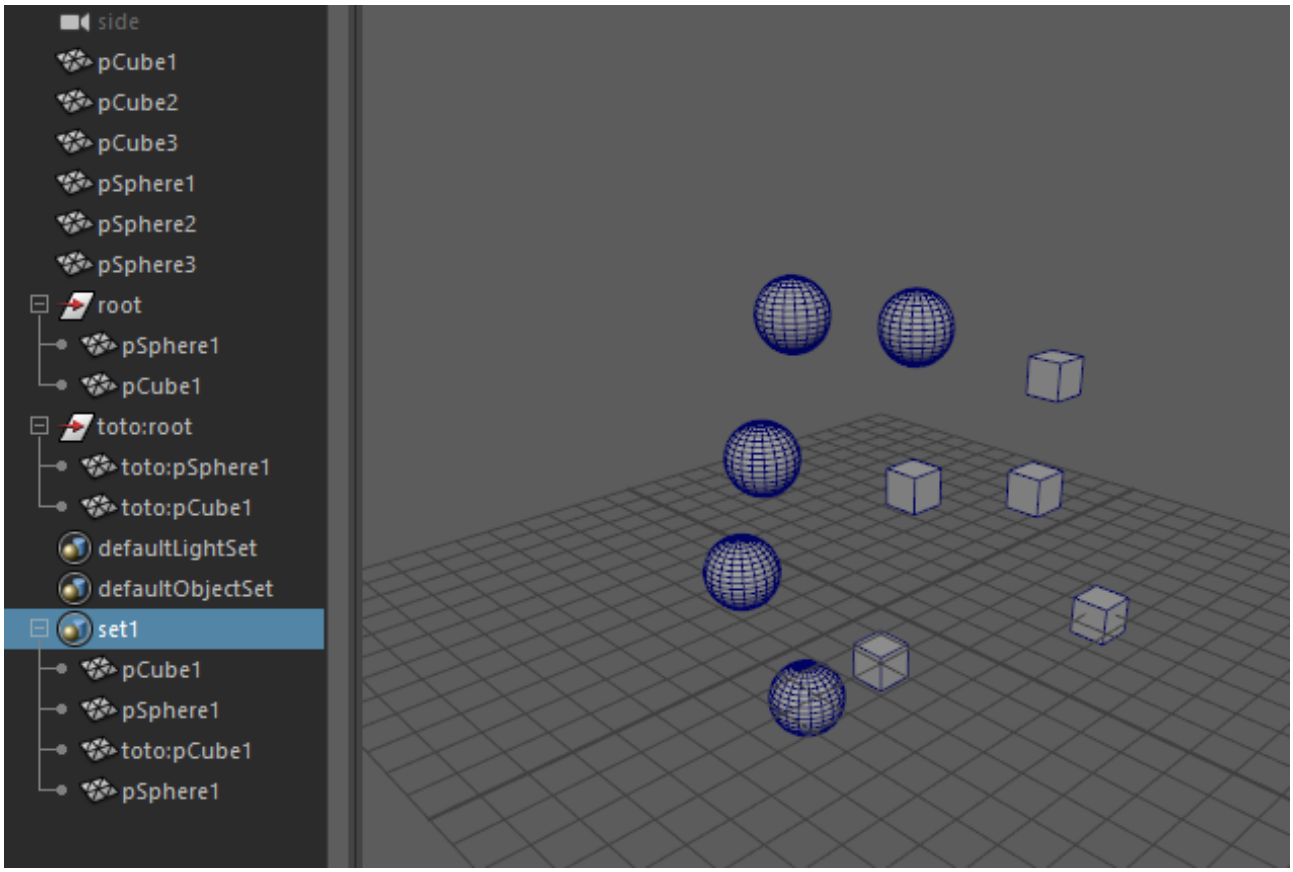
Maya sélectionnera en fait le contenu du set et non le nœud de set lui-même:



Il faut ajouter l'argument `noExpand` :

```
1 mc.select('set1', noExpand=True)
```

Ainsi, c'est bien le nœud qui sera sélectionné et non son contenu:



III.3.5.10. Note importante concernant la commande `select()`

Quand un graphiste travaille au quotidien, il le fait en sélectionnant des nœuds. C'est donc par un jeu de sélection qu'il conçoit son travail dans Maya.

Les graphistes débutants en script dans Maya ont tendance à reproduire ce schéma et abuser de la commande `select()` en s'appuyant sur le mécanisme d'*argument implicite* des commandes (quand aucun nœud n'est fourni en argument d'une commande, Maya utilise le nœud sélectionné).

C'est une très mauvaise pratique, car elle vous incite à saupoudrer votre code de `select()` suivant ce que vous faites. Bien qu'intuitif de prime abord, gérer la sélection est plus compliqué qu'il n'y paraît:

— Toutes les commandes ne fonctionnent pas de la même façon suivant la sélection.
— Le code est difficilement lisible.

Utiliser `select()` **uniquement** pour sélectionner des choses pour l'utilisateur, souvent en fin de script. Mais ne vous appuyez pas dessus pour exécuter une suite de commandes. Toutes les commandes Maya peuvent fonctionner avec des arguments explicites, sans avoir recours à des sélections.

> Privilégiez le passage explicite de variables aux commandes Maya.

Conclusion

Et voilà ! Un chapitre bien costaud, mais vous avez maintenant de solides bases.

? C'est super, mais j'avoue que je me demande comment on utilise tout ça dans un vrai script.

Ne vous inquiétez pas, le chapitre suivant sera composé de petits codes simples utilisant les commandes que vous avez apprises ici, mais en essayant de vous proposer des cas pratiques que j'espère intéressants.

Contenu masqué

Contenu masqué n°1

La version copié-collé:

```python
import maya.cmds as cmds

cmds.sphere( n='balloon' )

# Find the type of node created by the sphere command
cmds.nodeType( 'balloon' )
# Result: transform #

# What is the API type of the balloon node?
cmds.nodeType( 'balloon', api=True )
# Result: kTransform #

# Which node types derive from camera?
cmds.nodeType( 'camera', derived=True, isTypeName=True )
# Result: [u'stereoRigCamera', u'camera'] #
```

Retourner au texte.

III.4. Faire une simple interface

Introduction

Il est certes encore un peu tôt, mais nous allons voir comment faire de petites interfaces pour exécuter nos scripts.

III.4.1. Avant-propos

Avant de commencer ce court chapitre, il me semble important d'aborder certains points.

Nous n'avons vu que peu de commandes jusqu'ici, mais je sais que si vous êtes graphiste, un des premiers trucs que vous allez vouloir faire c'est une petite interface pour pouvoir utiliser vos scripts.

Depuis ses origines, Maya permet d'utiliser les commandes pour faire des interfaces. Bien qu'il s'agisse d'une révolution à l'époque (cf. le premier chapitre : *L'architecture*) l'intégration de Python a pas mal changé la donne en permettant à n'importe quelle bibliothèque d'interface graphique compatible avec Python (et plus particulièrement *Qt* via ses *binding* Python respectifs *PyQt* et *PySide*) de s'exécuter depuis Maya. En pratique il était donc possible de se passer des commandes Maya pour faire des interfaces.

?

Mais pourquoi utiliser une bibliothèque externe si Maya dispose déjà de commandes pour ça ?

La réponse est simple : c'est plus facile de faire une interface simple avec les commandes Maya, mais c'est plus simple de faire une interface évoluée avec *PyQt*/*PySide*.

?

Euh... Tu peux développer ?

PyQt et *PySide* nécessitent d'être familiarisé avec certains concepts un peu plus évolué de Python (programmation orientée objet). Il faut ajouter à ça que *Qt* n'est pas une petite bibliothèque et mériterait un tutoriel dédié.

À l'inverse, les commandes Mayas d'interfaçage gardent la logique des autres commandes, que ce soit au niveau de la documentation ou de leur utilisation. L'idée est donc d'éviter de trop vous dépayser avec de nouveaux concepts, mais de faire de petites interfaces sans pousser trop loin, histoire d'avoir la classe à Dallas.

III.4.2. window() pour ouvrir des fenêtres

Avant de pouvoir mettre des boutons et des cases à cocher partout, il vous faut une fenêtre dans laquelle mettre tout ça.

Exécutez ce code (extrait de... la doc bien entendu !) :

```
1 window = mc.window("zds_window", title="Ma fenetre",
      widthHeight=(200, 55), backgroundColor=(0.925, 0.467, 0.047))
2 mc.columnLayout(adjustableColumn=True)
3 mc.button(label='Do Nothing')
4 mc.button(label='Close',
      command=('mc.deleteUI(\"'+window+'\", window=True)') )
5 mc.setParent('..')
6 mc.showWindow(window)
```

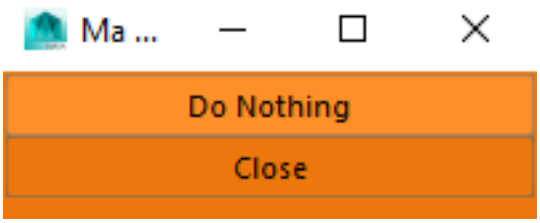


FIGURE III.4.1. – Fenêtre avec des boutons dans Maya.

? Oh ! Quand je clique sur *Close* la fenêtre se ferme, comment on peut faire ça ?

Mais que vous allez vite!

Les commandes qui vont nous intéresser sont `window()` et `showWindow()`. On va garder le reste pour plus tard sinon je vais vous perdre.

III.4.2.1. La commande `window()`

Sur la première ligne, la commande `window()` crée une fenêtre et renvoie son nom interne (oui, parce que *window* ça veut dire *fenêtre* en français).

Penchons-nous un peu sur les arguments :

— Le premier, `"zds_window"`, est le nom interne de la fenêtre. Notez que cet argument est optionnel, si vous ne le spécifiez pas, Maya va donner un nom interne automatiquement. Gardez également à l'esprit que si une fenêtre avec un nom similaire existe, le nom final de votre fenêtre sera suffixé d'un nombre, suivant les règles des noms Maya (et deviendra `"zds_window1"` par exemple).
— L'argument `title` sert à donner un titre à votre fenêtre.

— L'argument `widthHeight` permet de donner la taille de la fenêtre sous la forme d'une `list` (ou un `tuple`) de deux valeurs. Notez que vous pouvez utiliser les arguments `width` et `height` séparément (`width=200, height=55`).
— L'argument `backgroundColor` est mis à titre d'exemple, car on le retrouve dans énormément d'autres commandes d'interface. Il permet de définir une couleur en arrière-plan de ce que l'on crée.

III.4.2.2. Ajustement avec `columnLayout()`

Cette commande crée un *agencement* (un *layout* en anglais) qui agencera ses enfants (les boutons) sous la forme d'une seule colonne ajustable (l'argument `adjustableColumn=True`) :

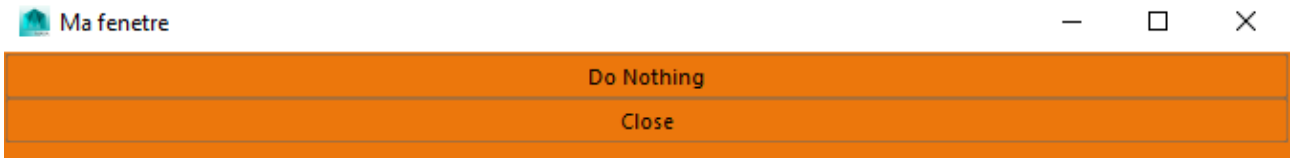


FIGURE III.4.2. – Avec 'adjustableColumn=True'.

N'hésitez pas à modifier et expérimenter avec cette commande pour voir les différents effets possibles. Voire encore mieux, tester les exemples de la doc .

FIGURE III.4.3. – Avec 'adjustableColumn=False'

III.4.2.3. `setParent('..')` ?

Je ne vais pas rentrer dans les détails de cette commande pour l'instant, mais pour faire court:

— Tout objet d'une interface nécessite un agencement parent.
— Vous aurez remarqué que les deux commandes `button()` créent chacune un bouton *dans* l'agencement en colonne.
— Ceci, car Maya définit un *agencement implicite* comme étant la dernière commande ayant généré un agencement. Ici, `columnLayout()`.
— Pour éviter que les prochaines commandes, que nous pourrions taper plus tard, aient notre fenêtre comme *agencement implicite*, `setParent('..')` «remonte» à la fenêtre principale de Maya.

La doc pour plus d'informations.

III.4.2.4. `showWindow()` tout à la fin

Comme son nom l'indique, cette commande permet d'afficher la fenêtre qu'on lui donne en argument. En effet, jusqu'à présent, la fenêtre était créée mais pas affichée.

> Mais… Mais… À quoi ça sert de faire une fenêtre si on ne l'affiche pas ? 🍊

Tout simplement à attendre qu'on y ait tout mis avant de l'afficher. 🍊

Quand une fenêtre est affichée, **à chaque fois** qu'on y ajoute/modifie/supprime un élément de l'interface (bouton, case-à-cocher, agencement, etc.), les parties concernées par ces modifications doivent être rafraîchies.

Pour quatre boutons cela nous donnerait grosso-modo :

- Création du bouton 1
  - Agencement du bouton 1
- Création du bouton 2
  - Réagencement du bouton 1
  - Agencement du bouton 2
- Création du bouton 3
  - Réagencement du bouton 1
  - Réagencement du bouton 2
  - Agencement du bouton 3
- Création du bouton 4
  - Réagencement du bouton 1
  - Réagencement du bouton 2
  - Réagencement du bouton 3
  - Agencement du bouton 4

etc.

Le fait de n'afficher la fenêtre qu'une fois que tout est mis dedans permet d'ajuster le positionnement une seule fois, au dernier moment.

> Prenez l'habitude de n'afficher vos fenêtres qu'une fois que tous les éléments y sont. 🍊

III.4.2.5. Vérifier qu'une fenêtre existe

À l'heure actuelle, rien ne vous empêche de ré-exécuter le script et d'afficher une seconde fenêtre. Il vous faudra alors fermer la fenêtre précédemment ouverte pour éviter d'en avoir deux (trop dur la vie 🍊).

Une technique souvent utilisée pour s'assurer que la fenêtre principale de notre script n'est affichée qu'une seule fois consiste à vérifier qu'elle existe et à la supprimer avant de la recréer.

Pour cela, on utilise la commande `window()` en mode `query` avec l'argument `exists=True` pour savoir si elle existe :

```
1 mc.window("zds_window", query=True, exists=True)
```

Cet appel de commande pourrait se traduire par : «est-ce que la fenêtre `"zds_window"` existe ?».

? C'est quoi ce mode `query`? 🍊

Nous y reviendrons. 🍍 Le but ici c'est que *questionner* (`query` en anglais) l'existence de la fenêtre donnée, ici, `"zds_window"`.

Pour supprimer un élément d'interface, il faut passer par la commande `deleteUI()` avec l'argument `window=True` pour bien spécifier que c'est une fenêtre qu'on cherche à supprimer :

```
1 mc.deleteUI("zds_window", window=True)
```

Cet appel de commande pourrait se traduire par «supprime la fenêtre nommée `"zds_window"`».

Dans un code plus complet cela donnerait :

```
1 # supprime la fenetre si elle existe
2 if mc.window("zds_window", query=True, exists=True):
3     mc.deleteUI("zds_window", window=True)
4
5 # cree ma nouvelle fenetre
6 window = mc.window("zds_window")
7 mc.showWindow(window)
```

Essayez d'exécuter ce code plusieurs fois, vous verrez qu'à chaque fois, la fenêtre se ferme avant d'être recréée. 🍍

Notez que ceci implique que le nom que vous donnez à votre fenêtre (ici `"zds_window"`) est unique pour ne pas entrer en collision avec d'autres noms déjà existants.

? Mais comment je peux m'assurer d'avoir un nom unique ? 🍊

Il y a plusieurs méthodes, mais une approche couramment utilisée consiste à préfixer les noms de ses fenêtres (ici `zds_` pour *Zeste de Savoir*).

III.4.2.6. Conclusion

On sait maintenant créer et manipuler les fenêtres, mais je vois bien que ce qui accapare votre attention depuis le début ce sont les boutons (allez, ne mentez pas).

Et bien c'est ce qu'on va attaquer. Préparez-vous !

III.4.3. La commande Benjamin... button()

Le jeu de mot qui se cache dans ce titre vous est offert par la guilde des codeurs fatigués.

Reprenons le code précédent :

```
window = mc.window("zds_window")
mc.columnLayout(adjustableColumn=True)
mc.button(label='Do Nothing')
mc.button(label='Close',
    command=('mc.deleteUI(\"'+window+'\", window=True)') )
mc.setParent('..')
mc.showWindow(window)
```

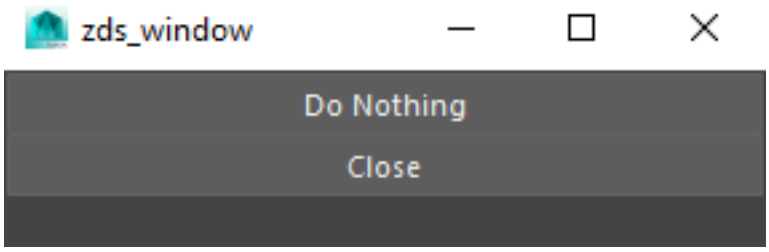


FIGURE III.4.4. – Fenêtre avec des boutons.

On sait maintenant à quoi servent toutes les commandes sauf... `button()`!

Cette commande permet de faire des boutons qui exécutent des scripts quand on clique dessus (vous l'avez deviné tout seul, vous êtes décidément très intelligent).

Voici une liste non exhaustive des arguments qu'on utilise le plus souvent (pour la liste exhaustive, *c'est la même, chan-son !*).

Notez que les commandes d'interface sont très souvent utilisées en mode `edit` pour modifier leurs états (visibilité, grisé/non grisé, etc.) et ainsi donner un peu de dynamique aux comportements de l'interface.

- `label` permet de définir le texte qui s'affichera sur le bouton (exemple : `label="Mon Bouton"`).
- `enable` active ou désactive le bouton pour déterminer si on peut cliquer dessus (exemple : `enable=False`).

— `height` et `width` permettent de définir la taille du bouton (exemple : `height=600`). Par défaut, le bouton s'ajustera au *layout* qui le contient (ici `columnLayout`).
— `visible` permet d'afficher ou de masquer le bouton (exemple : `visible=False` rend le bouton invisible).
— `backgroundColor` comme vu pour la commande `window()`, cet argument permet de faire souffrir les yeux de vos collègues (essayez `backgroundColor=(1.0, 0.0, 1.0)` pour vous donner une idée 🍊).

L'expérimentation vous permettra de découvrir deux trois choses intéressantes, alors prenez quelques minutes et essayez-les tous!

Il en reste un dernier, le plus important. 😈

III.4.3.1. L'argument `command`

Cet argument est particulier, car il y a plusieurs façons de l'utiliser, le but étant de définir ce qui va se passer quand on clique sur le bouton en question.

? C'est l'argument qui est utilisé par le bouton *Close* et qui ferme la fenêtre ? 🍊

Exactement ! Reprenons la ligne en question :

```
mc.button(label='Close',
    command='mc.deleteUI(\"'+window+'\", window=True)')
```

C'est la première façon d'utiliser l'argument `command`: en lui passant une chaîne de caractère qu'il va *évaluer* comme un script python.

Ici, on utilise la variable `window`, contenant le *nom interne* de la fenêtre, pour générer une chaîne de caractère qui ne sera qu'un appel à une commande avec son argument:

```
print 'mc.deleteUI(\"'+window+'\", window=True)'
# mc.deleteUI("zds_window", window=True)
```

Du coup, cliquer sur le bouton revient à exécuter:

```
mc.deleteUI("zds_window", window=True)
```

Bien que facile à comprendre, je ne trouve pas cette approche très élégante. 🍊

III.4.3.2. Les fonctions de rappel (*callbacks* en anglais)

En voilà un bien vilain mot . Ce mécanisme est très connu en programmation et permet de faire mille choses. Dans notre cas, il permet au bouton d'exécuter lui-même une fonction qu'on lui passe (en argument de `command`).

Utiliser une fonction de rappel, c'est passer une fonction qui sera appelée au moment où on clique sur le bouton.

Regardez le code suivant:

```
1 def close_callback(*args):
2   print "Close!", args
3
4 window = mc.window(width=150)
5 mc.columnLayout(adjustableColumn=True)
6 mc.button(label="Close", command=close_callback)
7 mc.showWindow()
```

On définit une fonction, `close_callback()` qu'on passe comme une simple variable (sans l'appeler) en argument de `command`.

Quand on l'appelle, cela écrit `Close !`:

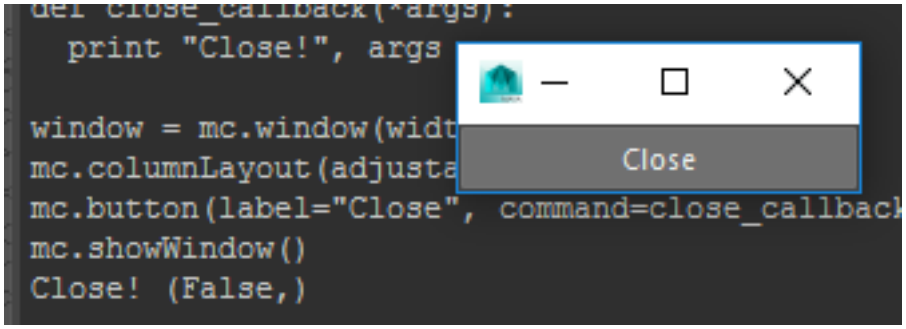


FIGURE III.4.5. – Appel du callback

Comme vous pouvez le voir, nous passons la fonction `close_callback()` en argument de `command` mais sans les `()`, de sorte de ne pas l'appeler. C'est bien la fonction que nous passons en argument, pas son résultat.

Le bouton appellera la fonction `close_callback()` autant de fois qu'on cliquera dessus.

Mais si je souhaite utiliser `mc.deleteUI()` pour faire un bouton *Close* comme précédemment, il faut que je puisse passer le nom de la fenêtre en argument non ?

Bien vu, on ne peut pas le faire avec une simple fonction de rappel, il faut aller plus loin, et c'est ce qu'on va faire ! 🍊

III.4.3.3. `functools.partial()` à la rescousse !

Pour ceci, nous allons utiliser une fonction très intéressant de Python: `functools.partial()` . Cette fonction prend deux arguments:

— La fonction à appeler.
— Les arguments à passer à cette fonction.

Ça va être compliqué quelques minutes (pas plus, promis !), accrochez-vous. 🍊

Prenons un exemple simple:

```
1 import functools
2
3 def print_something(something):
4   print something
5
6 my_function = functools.partial(print_something, "toto")
7 my_function()
8 # toto
```

Ligne a ligne:

```
1 import functools
```

On importe le module functools:

```
1 def print_something(something):
2   print something
```

On définit une fonction `print_something` qui ne fait qu'afficher l'argument `something` qu'on lui donne.

La partie intéressante arrive ici:

```
1 my_function = functools.partial(print_something, "toto")
```

On fabrique une fonction `my_function`. Cette fonction appellera la fonction `print_something` avec l'argument `"toto"`.

La suite vous la connaissez:

```
my_function()
# toto
```

On appelle la fonction qui fait exactement ce qu'on lui a dit: appeler `print_something` avec l'argument `"toto"`.

D'une certaine manière:

```
my_function()
```

revient à appeler:

```
print_something("toto")
```

Si vous souhaitez avoir plus d'informations sur `functools.partial()`, je vous invite à jeter un œil à la documentation officiel .

Vous vous en doutez, on va donc passer ça à notre argument `command`:

```
import functools

def close_callback(window, arg):
  print "Close!", window
  mc.deleteUI(window, window=True)

window = mc.window(width=150)
mc.columnLayout(adjustableColumn=True)
mc.button(label="Close", command=functools.partial(close_callback,
    window))
mc.showWindow()
```

Et ça marche!

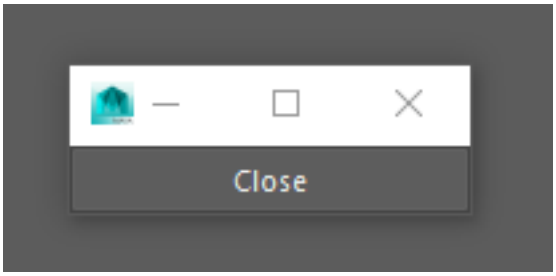


FIGURE III.4.6. – Notre belle fenêtre.

Et quand on clique :

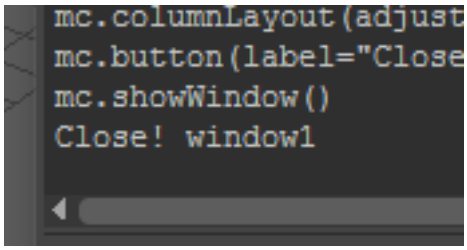


FIGURE III.4.7. – Tadaaa!

? Quel est l'argument `arg` que nous passons après `window`? 🍊

Excellente question et je vais malheureusement devoir botter en touche. 🍊

Maya envoie toujours cet argument et vous devez le mettre dans votre fonction ou vous aurez une erreur de ce genre:

```
# TypeError: close_callback() takes exactly 1 argument (2 given) #
```

Maya vous explique qu'il a mis deux arguments dans la fonction mais que celle-ci ne s'attendait qu'à un seul. 🍊

Une convention, en Python, quand un argument ou une variable doit être présent mais que vous ne considérez pas l'utiliser est de lui donner la valeur `_`, comme ceci:

```
def close_callback(window, _):
```

Ainsi, vous n'aurez pas de message d'erreur et vous informerez le lecteur de votre code que cette variable ne vous intéresse pas. 🍊

Prenez quelques minutes pour bien comprendre ce mécanisme, le modifier et l'expérimenter. Il n'est pas propre à Python et vous servira pour d'autres choses. 🍊

III.4.4. loadUI() pour les flemmards

Maya est fourni avec Qt Designer , un outil de construction d'interface graphique. Allez dans le dossier d'installation de Maya, dans le dossier `bin` et cherchez `designer.exe`.

Sous *Windows* il est placé ici:

```
C:\Program Files\Autodesk\Maya2016.5\bin\designer.exe
```

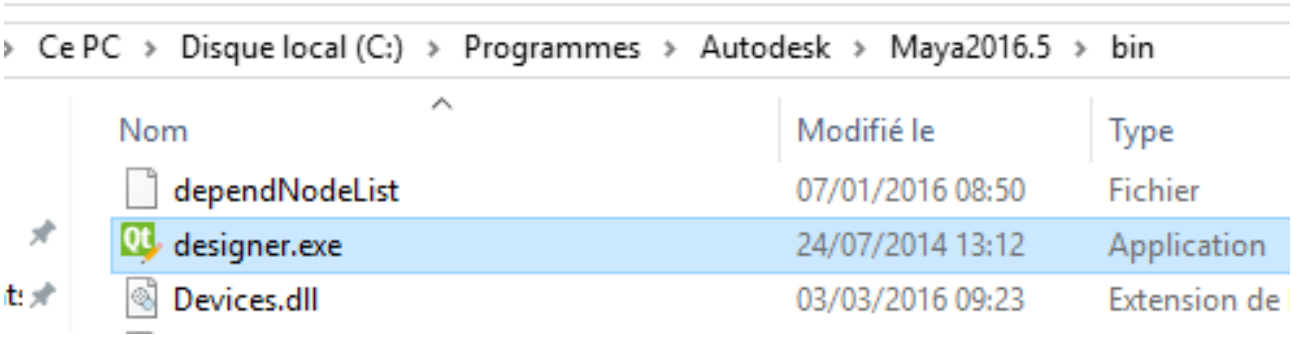


FIGURE III.4.8. – Chemin vers Qt Designer dans Maya.

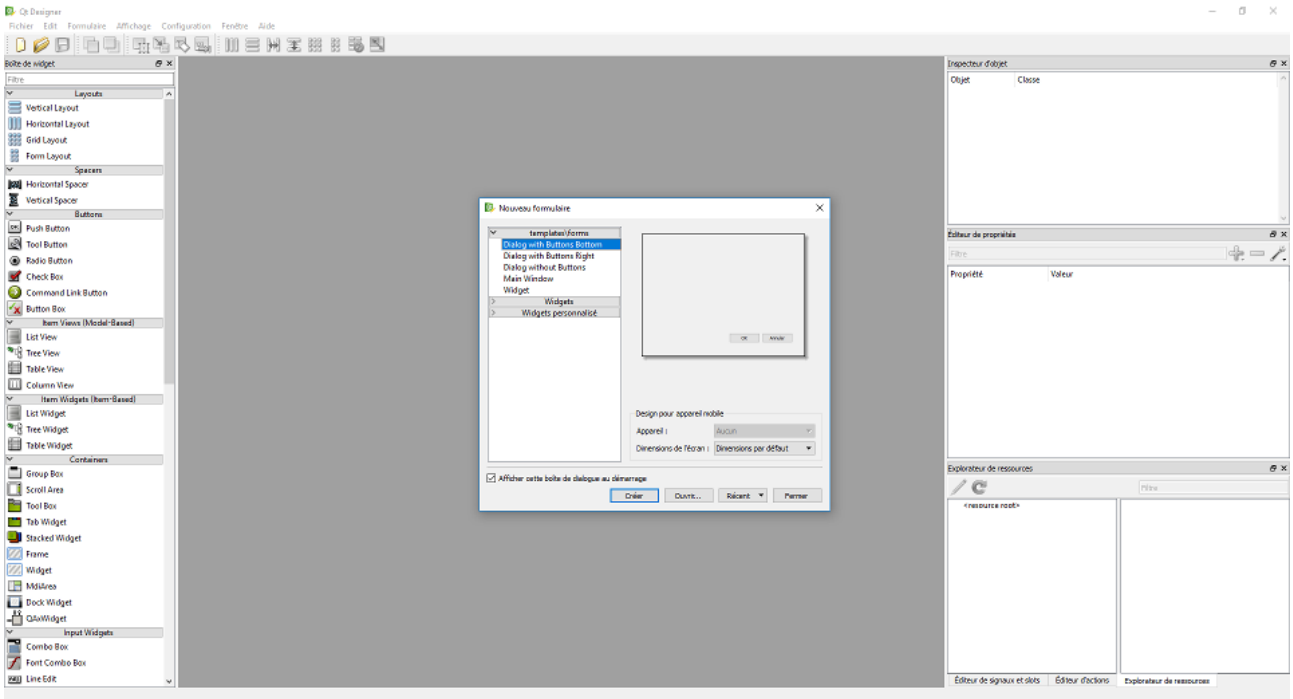


FIGURE III.4.9. – Qt Designer livré avec Maya.

Ne me dis pas que… 🍊

Si, si, vous pouvez construire vos interfaces avec cet outil, les sauvegarder dans un fichier `.ui` et les charger dans Maya en pointant dessus via la commande loadUI :

```python
import maya.cmds as mc

dialog = mc.loadUI(uiFile='C:/users/username/mydialog.ui')
mc.showWindow(dialog)
```

? Mais, pourquoi ne pas l'avoir dit plus tôt? 🍊

Sinon vous n'auriez jamais continué plus loin.

Allez, ne soyez pas déçu, je suis sûr que vous avez appris des choses.

De plus, vous allez voir que nombre des choses vus dans les précédentes sections de ce chapitre sont nécessaires (agencement, fonctions de rappel, etc.). En effet, avec un tel outil, vous êtes dans du *Qt* pur ce qui n'est pas non plus quelque chose qu'on maîtrise en un rien de temps. 🍊

Mais essayez, et vous devriez vite revenir avec pas mal d'interrogations.

? J'ai créé une superbe interface que j'arrive à afficher dans Maya, mais je n'arrive pas à exécuter mes fonctions quand je clique sur les boutons. 🍊

Nous y voilà! 🍊

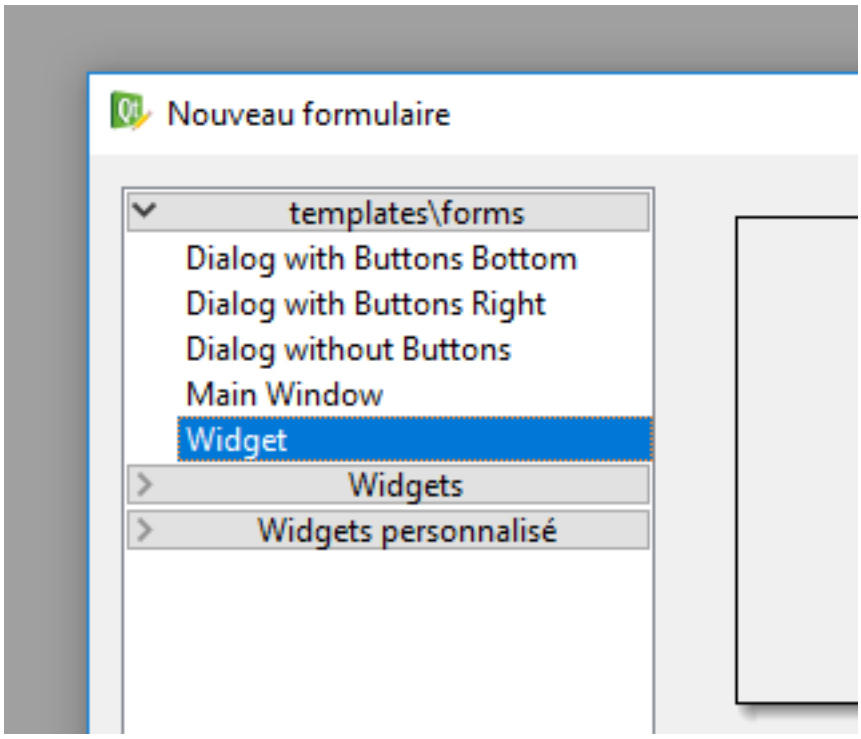
Connecter ses boutons a Maya en passant par ce mécanisme passe par un moyen détourné. Il existe plusieurs mécanismes, mais je vais vous proposer celui que je trouve le plus simple.

III.4.4.1. Connecter les interfaces de Qt Designer avec Maya

Je ne vais pas rentrer dans l'utilisation à proprement parler de Qt Designer (création des *widgets*, agencement, etc.), internet regorge d'information dédiée à ce sujet et je ne ferais que répéter. 🍊

En revanche, nous allons voir comment connecter nos widgets avec Maya, ici, les boutons.

Commencez par créer un *widget* vide:



Cliquez-glissez-y un bouton:

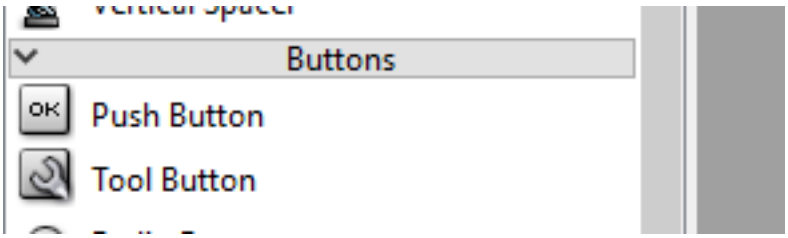


FIGURE III.4.10. – Push Button.

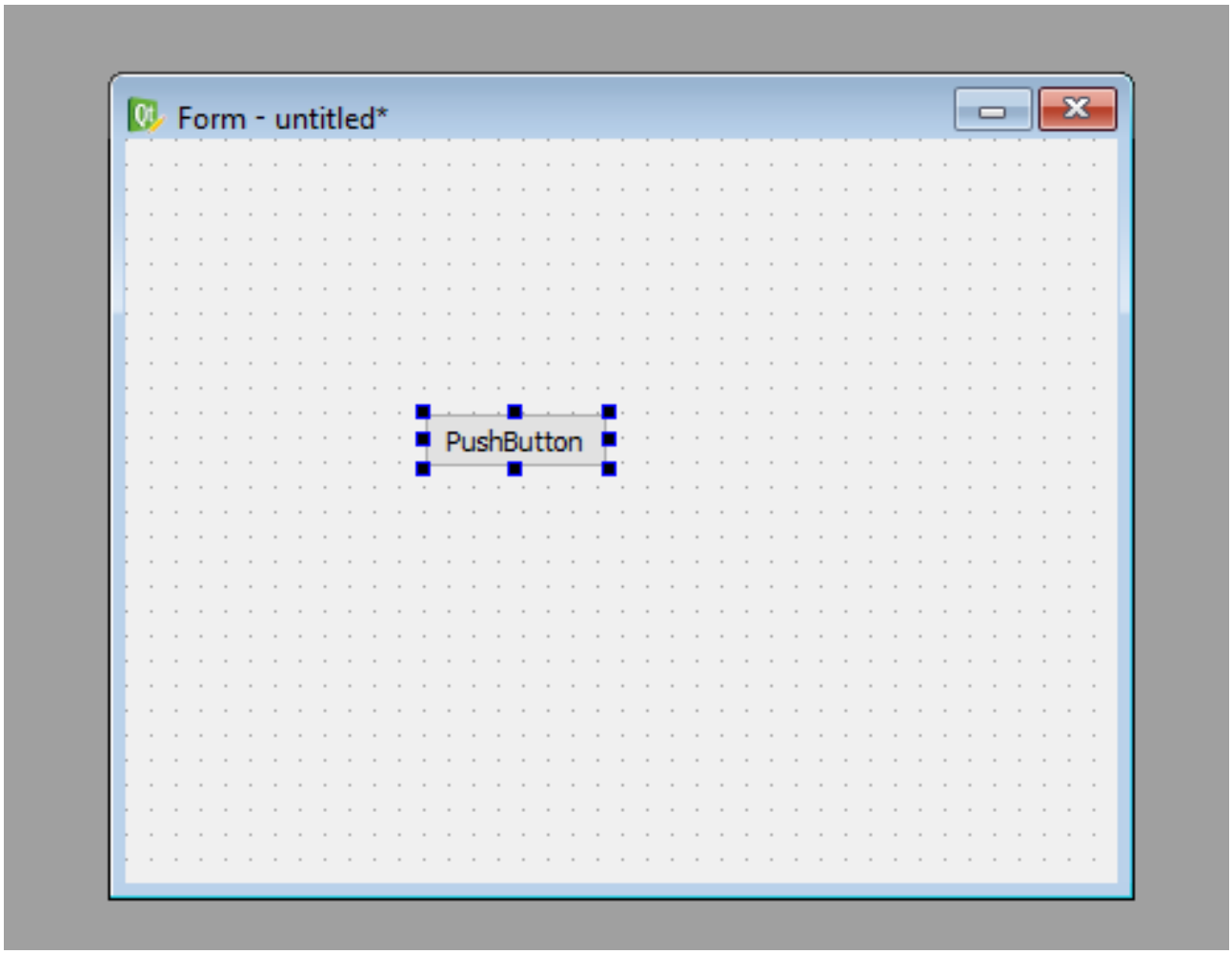


FIGURE III.4.11. – Trop ultime! :-°

Vous pouvez faire Ctrl+r pendant que vous travailler pour visualiser votre interface.

Sauvez votre jolie interface puis ouvrez Maya et chargez-la:

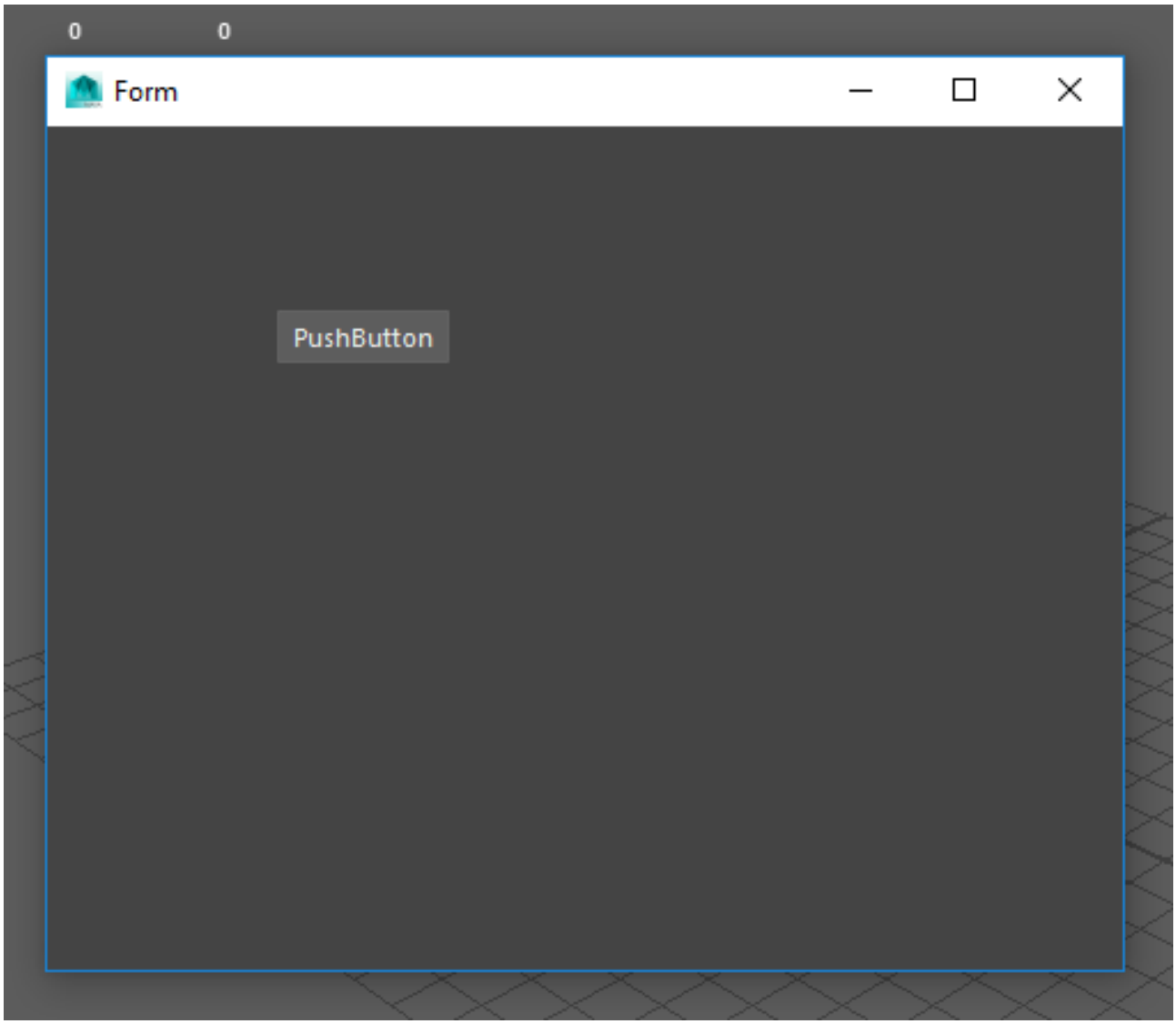


FIGURE III.4.12. – Apple n'a qu'à bien se tenir! :diable :

Vous remarquerez que rien ne se passe quand vous cliquez sur le bouton.

? En même temps, je n'ai rien fait pour qu'il se passe quelque chose donc bon. 🍊

On fait le malin. 🍊

Bref, vous avez compris, on va essayer de connecter une fonction à notre bouton.

III.4.4.1.1. Le nom des *widgets*

Retournez dans Maya puis regardez bien l'*Inspecteur d'objets*:

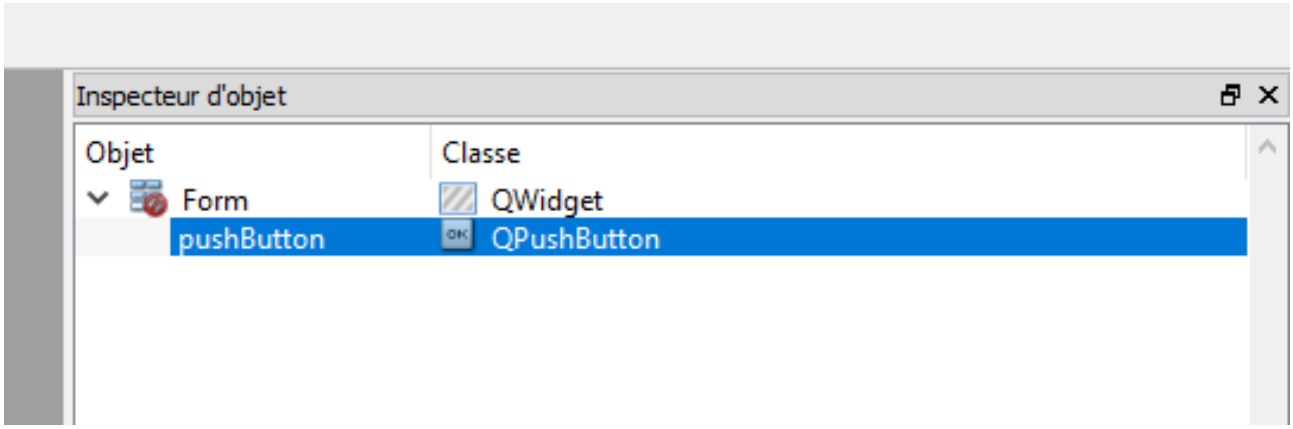


FIGURE III.4.13. – Mais là qui voilà, l'*Inspecteur d'objet*

Il vous affiche comment les choses sont organisées dans votre interface.

La colonne *Classes* affiche la classe Qt utilisée par le *widget*. C'est très pratique pour trouver de la documentation sur un objet qu'on souhaite modifier dans l'*Éditeur de propriétés*. Il suffit de taper le nom de la classe dans votre moteur de recherche favori. Je ne vais pas en parler plus que ça, juste pour vous dire que ces classes sont les noms des *widgets* que Qt utilise.

L'information qui nous intéresse, c'est le nom de chaque *widget*. Ici, nous avons `Form` et `pushButton`.

Exécutez ceci:

```
1 dialog = mc.loadUI(uiFile=r'C:\Users\vous\zds_qt_designer_001.ui')
2 print dialog
```

Comme vous pouvez le voir, on va afficher ce que renvoie la commande `loadUI()`:

```
dialog = mc.loadUI(f=
print dialog
Form
```

Oh! Il renvoie `Form`, comme le nom dans l'*Inspecteur d'objet*!

Exactement!

Et si vous modifiez son nom dans Qt Designer, sauvez puis rechargez, vous verrez que ce nom sera utilisé.

Au même titre que pour la fenêtre, on peut vérifier l'existence d'un bouton.

Vérifions l'existence de… `pushButton` pour voir.

```
mc.button('pushButton', query=True, exists=True)
```

La combinaison de l'argument `query` et `exists` vous rappellera sûrement ce qu'on a utilisé vérifier l'existence d'une fenêtre.

Exécutez cette commande avant de lancer votre fenêtre, elle devrait renvoyer `False`, puis chargez votre interface et ré-exécutez la. Cette fois ci, elle devrait renvoyer `True`.

? Chez moi elle renvoie toujours `True`.

Il est possible qu'une fenêtre soit toujours ouverte. Au pire, exécutez `mc.deleteUI('Form', window=True)` pour forcer Maya à supprimer la fenêtre. Si ça ne fonctionne toujours pas, redémarrez Maya.

III.4.4.1.2. Modifier notre bouton

Bon, il semblerait que nous ayons notre bouton d'accessible. Tiens, si nous lui demandions ce qu'il exécute quand on lui clique dessus. Pour cela nous allons lui demander quel est le contenu de son argument `command`:

```
print mc.button('pushButton', query=True, command=True)
# None
```

Sans surprise, il n'y a rien dans l'argument `command`. Comment puis-je le modifier pour qu'il exécute mon code?

Avec l'argument `edit` bien sur!

? C'est quoi `edit`?

Là où l'argument `query` permet de questionner un argument, l'argument `edit` permet de modifier (*edit* en anglais) des choses.

Je sais, je sais, je vous l'expliquerai en détails plus tard, promis. En attendant on va l'utiliser pour modifier l'argument `command` qui, à l'heure actuelle, est vide.

Chargez votre fenêtre si ce n'est déjà fait (sinon le bouton `pushButton` n'existera pas et la commande plantera) puis exécutez:

```
def my_function(*args):
    print "Hello!"

mc.button('pushButton', edit=True, command=my_function)
```

Ici, on définit une fonction `my_function()` qu'on passe via l'argument `command` de `mc.button()` en mode `edit`. On vient donc modifier le bouton `pushButton` existant.

À partir de maintenant, vous devriez commencer à comprendre un certain nombre de choses:

— Le nom des objets est important. Je vous invite donc à préfixer tout vos objets avec `zds_`.
— Le chargement de l'interface, via la command `loadUI()` doit se faire **avant** la modification, via `edit`, de son contenu.

III.4.4.2. Exemple final

Je vous mets un exemple complet utilisant `functools.partial()` vu précédemment.

J'ai fait quelques modifications de nom dans ma superbe interface:

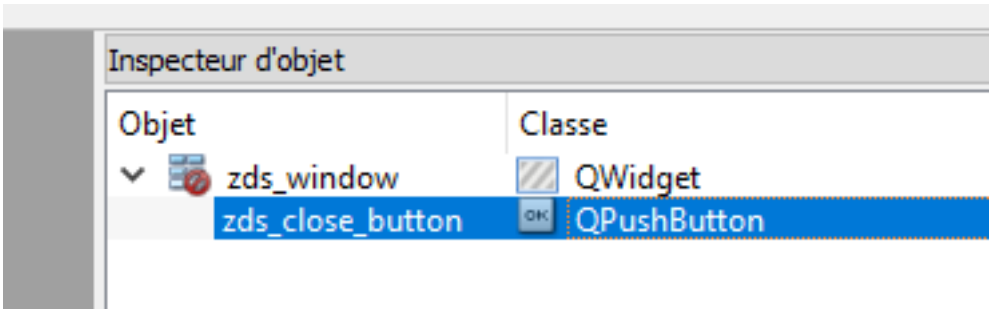


FIGURE III.4.14. – Préfixer ses objets, c'est important. :soleil :

Notez que le bouton s'appelle maintenant `zds_close_button`.

```
import functools

def close_callback(window, *args):
  print "Close!", window
  mc.deleteUI(window, window=True)

# load ui
window = mc.loadUI(uiFile=r'C:\Users\vous\zds_qt_designer_001.ui')

# set callbacks
mc.button('zds_close_button', edit=True,
    command=functools.partial(close_callback, window))
```

```
12
13 mc.showWindow(window)
```

Et quand vous cliquez sur le bouton il affiche `Close! zds_window`, puis ferme la fenêtre.

III.4.4.3. Sous le capot

Nous sommes revenus à notre point de départ mais en utilisant Qt Designer. Mais je ne pouvais clore ce chapitre en expliquant un peu ce qui se passe quand on exécute la commande `loadUI()`. 🍊

En fait, `loadUI()` lit le contenu de votre fichier `.ui` puis exécute les commandes d'interface Maya correspondantes (`mc.window()`, `mc.button()`, etc.). Il fait donc le travail pour vous! 🍊

C'est également pour cette raison que vos éléments d'interface étant de vrais objets Maya, vous pouvez les interroger (via `query`) et les modifier (via `edit`), les supprimer (via `mc.deleteUI()`), etc., comme si vous les aviez créés vous-même.

Durant cette *interprétation*, Maya peut faire d'autres choses, je vous donne la page de la documentation officielle que je vous invite à lire si vous souhaitez allez plus loin. 🍊

III.4.4.3.1. Conversion par `loadUI()`

Comme je sais que vous êtes très curieux, je me doute que vous avez déjà potassé la page de la documentation de la commande `loadUI()` pas vrai? 🍊

? En fait... Euh...

Je m'en doutais. C'est par là !

Si vous exécutez la commande `loadUI()` avec l'argument `verbose`, Maya affichera listera les conversions qu'il exécute. Dans le cas de notre simple bouton:

```
1 mc.loadUI(uiFile=r'C:\Users\vous\zds_qt_designer_001.ui',
      verbose=True)
2 # Creating a QPushButton named "zds_close_button". #
```

Maya ne crée qu'un bouton via le *widget* Qt `QPushButton`.

III.4.4.3.2. Liste des *widgets* supportés

Si vous exécutez la commande `loadUI()` avec l'argument `listTypes`. Maya vous renverra la relation entre la classe Qt qu'il utilise et la commande Maya correspondante.

```
1 mc.loadUI(listTypes=True)
```

```
1  # Result: [u'CharacterizationTool:characterizationToolUICmd',
2   u'QCheckBox:checkBox',
3   u'QComboBox:optionMenu',
4   u'QDialog:window',
5   u'QLabel:text',
6   u'QLineEdit:textField',
7   u'QListWidget:textScrollList',
8   u'QMainWindow:window',
9   u'QMenu:menu',
10  u'QProgressBar:progressBar',
11  u'QPushButton:button',
12  u'QRadioButton:radioButton',
13  u'QSlider:intSlider',
14  u'QTextEdit:scrollField',
15  u'QWidget:control',
16  u'TopLevelQWidget:window'] #
```

À gauche, le nom de la classe Qt, à droite, le nom de la commande Maya.

Vous remarquerez rapidement que tous les *widgets* présents dans Qt Designer n'ont pas d'équivalent dans Maya. C'est le cas notamment de la très utile *Spin Box*. Nous verrons plus tard comment bricoler (on est plus sur de la spéléologie à ce stade) pour la faire fonctionner.

Voici la liste non exhaustive des équivalents Qt en commande Maya (La *Classe Qt* est le nom qui apparaît dans l'*Inspecteur d'objet*):

Nom dans Qt Designer	**Nom de la classe Qt**	**Nom de la commande Maya**
Push Button (Section *Buttons*)	`QPushButton`	`button`
Radio Button (Section *Buttons*)	`QRadioButton`	`radioButton`
Check Box (Section *Buttons*)	`QCheckBox`	`checkBox`
Combo Box (Section *Containers*)	`QComboBox`	`optionMenu`
Line Edit (Section *Input Widgets*)`QLineEdit`	`textField`	
List View (Section *Item Views*)	`QListView`	`textScrollList`
Horizontal Slider (Section *Input Widgets*)	`QSlider`	`intSlider`

Vertical Slider (Section *Input Widgets*)	`QSlider`	`intSlider`
Progress Bar(Section *Display Widgets*)	`QProgressBar`	`progressBar`
Tab Widget (Section *Containers*)	`QTabWidget`	`tabLayout`

Sans oublier la fenêtre principale de classe `QWindow` et généré par la commande Maya `window`.

Les widgets qui n'ont pas d'équivalent en commande Maya sont toutefois modifiables via la commande `control` (documentation ici).

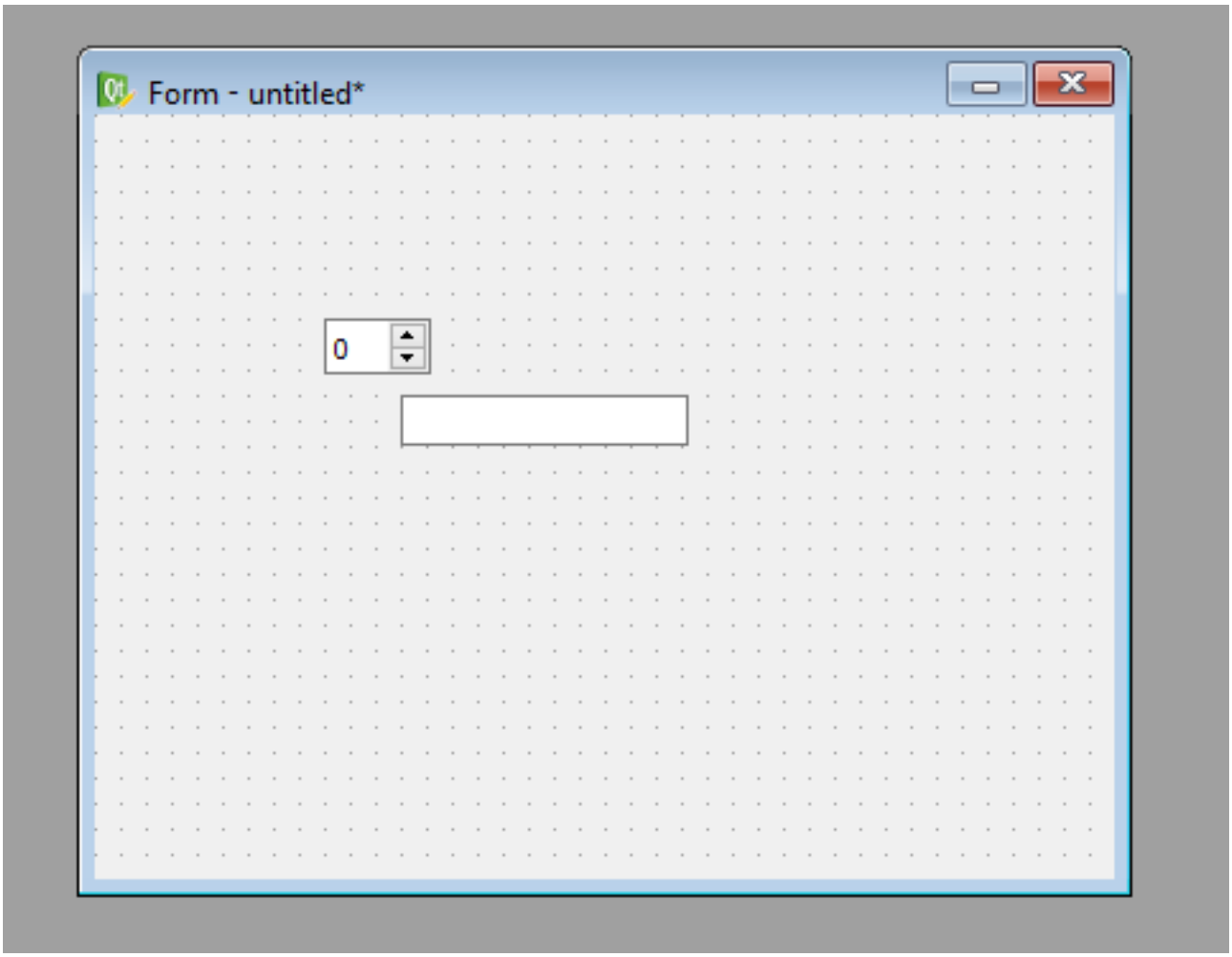
III.4.4.4. Récupérer la valeur d'un *Spin Box* dans Maya

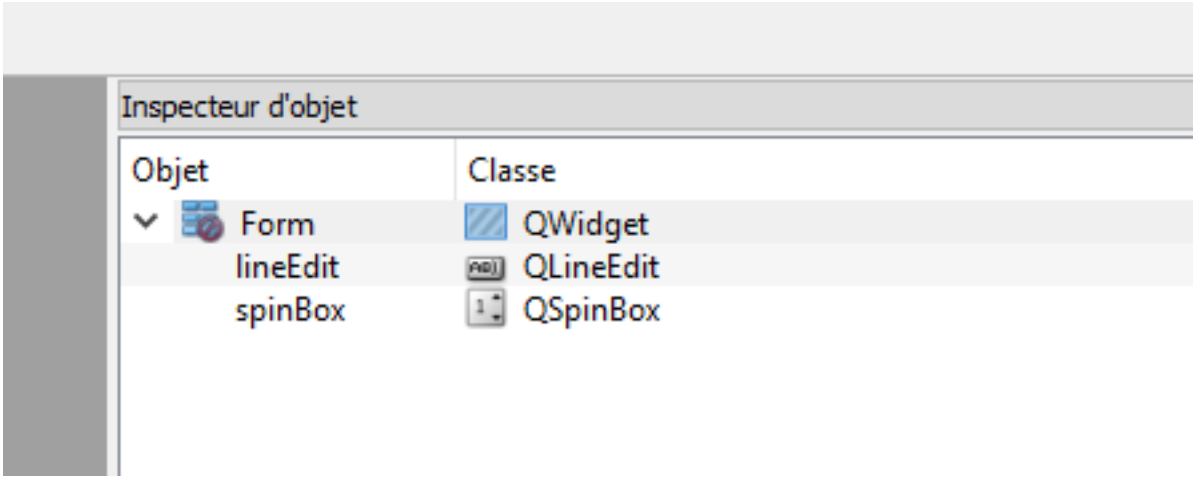
Comme je vous disais, il n'existe pas d'équivalent à la classe `QSpinBox` sous forme de commande Maya.

? Mais comme on est des graphistes, c'est pas ça qui va nous arrêter!

Exactement! Comme tout graphiste qui se respecte, on va faire un truc sale sous le capot pour faire un truc cool en surface.

Faites une *Spin Box* et un *Line Edit*:





? A quoi sert le *Line Edit*? 🍊

Comme on ne peut pas récupérer la valeur de la *Spin Box* directement, que diriez-vous de l'écrire dans un autre *widget* quand elle change puis de récupérer la valeur de cet autre *widget*?

Mais comment écrire la valeur de la *Spin Box* dans le *Line Edit*?

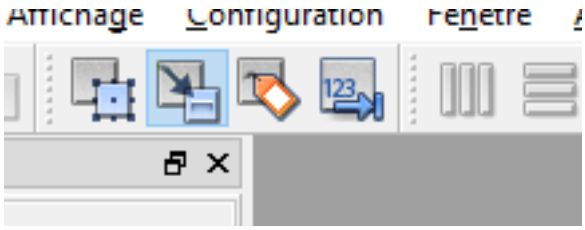
C'est ce que nous allons voir.

III.4.4.4.1. Utiliser les signaux de Qt Designer

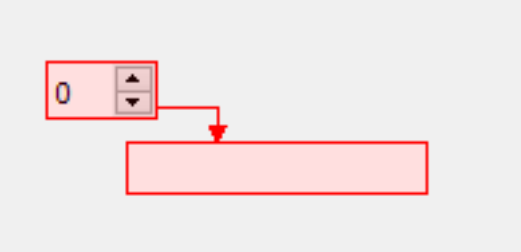
Qt dispose d'un mécanisme de signaux qu'il expose dans Qt Designer. Ce mécanisme permet, entre autre, d'exécuter des fonctions prédéfinies quand un *widget* fait que chose. En l'occurrence, nous allons faire en sorte que le *Spin Box* écrive sa valeur dans le *Line Edit* au moment où on la modifie.

Et tout ça, sans (trop) de code!

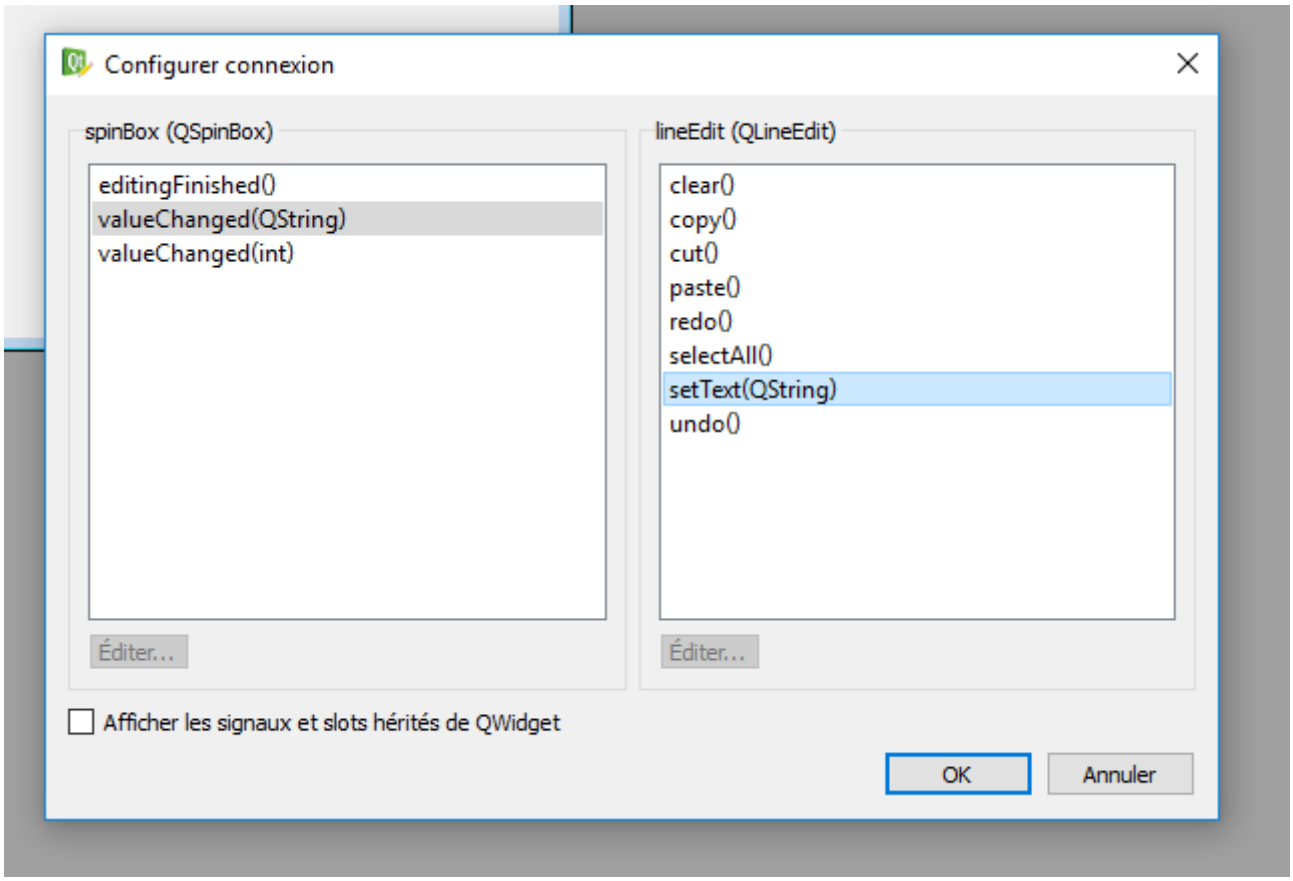
Cliquez sur la seconde icône pour passer en mode *Éditeur de signaux/slots*:



Cliquez-glissez la *Spin Box* sur le *Line Edit*, ceci indique dans quel sens le signal que nous allons créer va s'exécuter:

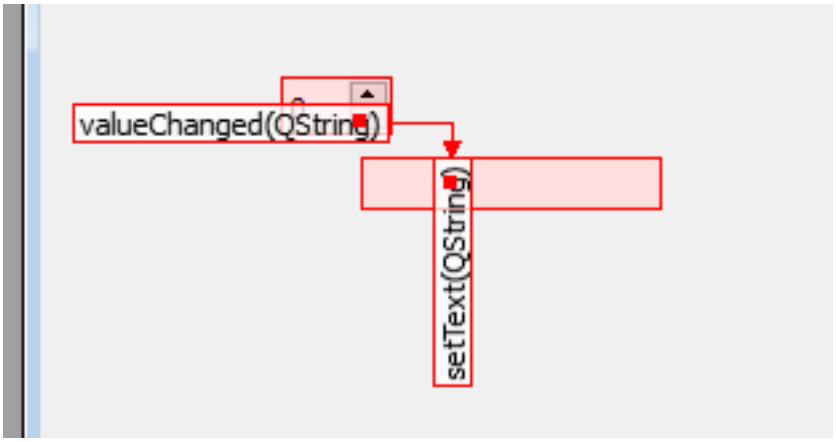


Une fenêtre de configuration de connexion va s'ouvrir:



Connectez le signal `valueChanged(QString)` de la *Spin Box* au `setText(QString)` de la *Line Edit* puis validez.

Une magnifique représentation des signaux s'affichera dans votre interface:



Faites Ctrl+r pour tester votre interface:

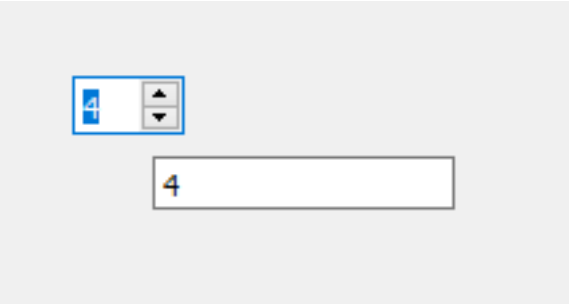


FIGURE III.4.15. – Remarquez comment modifier la *Spin Box* modifie également votre *Line Edit*.

Au démarrage, la *Line Edit* ne contient rien. 🍊

Bien vu, il vaut mieux lui mettre une valeur par défaut. Repassez en mode *Éditeur de widget* (première icône), double-cliquez sur le *Line Edit*, entrez `0` puis validez.

III.4.4.4.2. Retour dans Maya

Une fois cela fait, sauvez puis chargez votre interface dans Maya, jouez avec la *Spin Box* puis exécutez:

```
window = mc.loadUI(uiFile=r'C:\Users\vous\zds_qt_designer_002.ui')
mc.showWindow(window)

print mc.textField('lineEdit', query=True, text=True)
```

La valeur de votre *Line Edit* devrait s'afficher.

Cette commande questionne (via l'argument `query`) la valeur du texte (via l'argument `text`). La page de la documentation est ici .

Au passage, le type de la valeur renvoyée par la commande est, vous vous en doutez, `str`, ce qui veut dire que la valeur n'est pas un nombre (`int` ou `float`) mais un texte. Pour convertir, rien de plus simple:

```
result = int(mc.textField('lineEdit', query=True, text=True))
```

Ceci revient à faire `int('3')` ce que Python est parfaitement capable d'exécuter.

III.4.4.4.3. Suite et fin

Il nous suffit maintenant de cacher notre *widget*:

```
1 mc.textField('lineEdit', edit=True, visible=False)
```

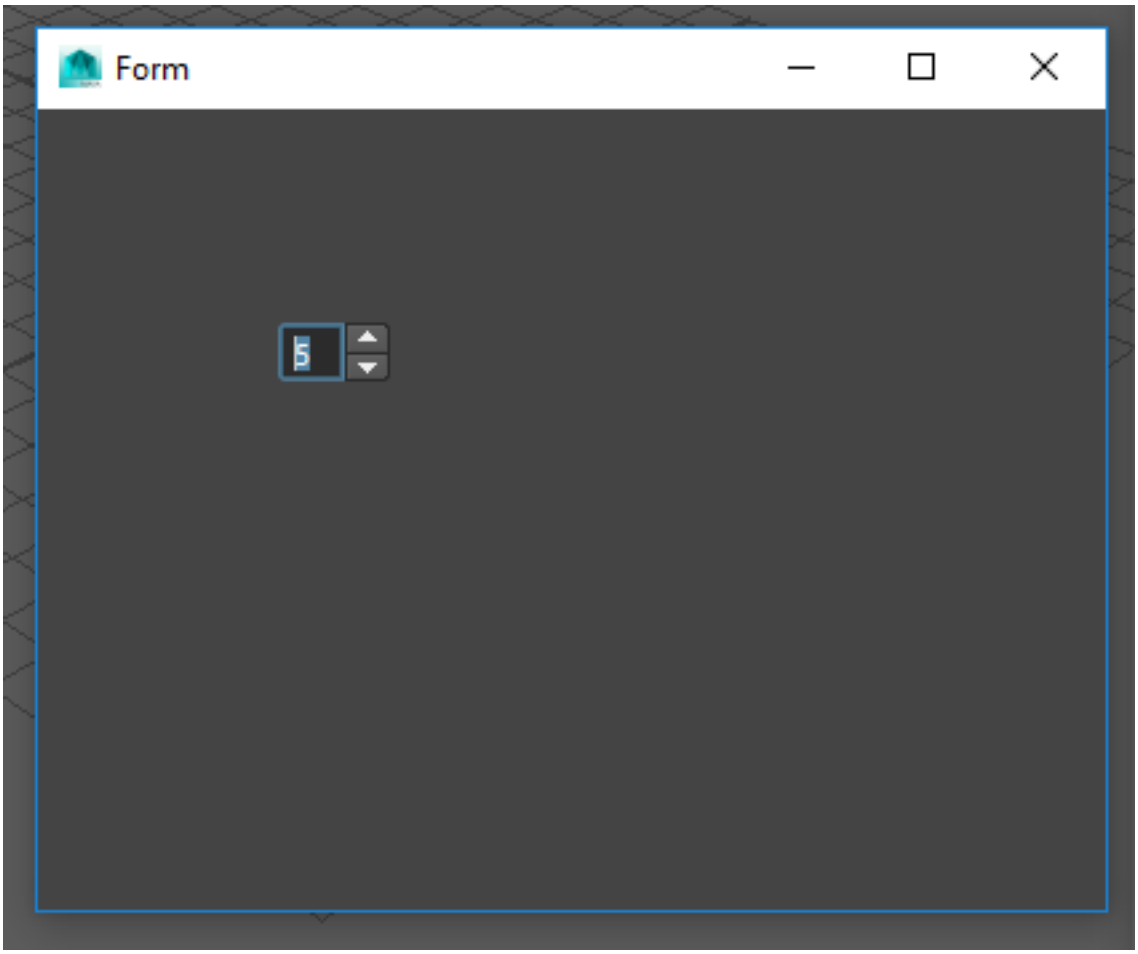


FIGURE III.4.16. – Jouer à cache-cache avec ses *widgets*, un vrai plaisir…

Vous pouvez modifier la *Spin Box* puis réessayer de récupérer sa valeur, vous verrez que vous aurez toujours la bonne. La *Spin Box* n'est pas supprimée, elle est simplement cachée. Mais elle est toujours là!

Notez que du coup, on ne peut faire l'inverse. Ainsi, il est impossible de *setter* une valeur à la *Spin Box* dynamiquement. C'est dommage, je vous l'accorde, mais on a déjà pas mal retourné Maya pour pouvoir faire ça, il ne faut pas trop lui en demander.

III.4.5. Ajouter un menu à Maya

Sans forcément vouloir faire une fenêtre, il peut être intéressant d'ajouter ses propres menus à l'interface Maya.

Pour cela, nous allons devoir récupérer la fenêtre principale, puis lui ajouter des menus.

?

Tu veux dire qu'on va manipuler la fenêtre Maya comme si c'était la nôtre?

Exactement!

III.4.5.1. Récupérer la fenêtre principale de Maya

Comme je vous l'ai expliqué précédemment, l'interface de Maya est codée intégralement via ces commandes, du coup rien ne vous empêche de les manipuler comme vous le feriez avec vos propres fenêtres!

En fait, Maya stocke le nom de la fenêtre principale dans une variable MEL: `$gMainWindow`

?

Mais... Mais... C'est une variable MEL ça, comment je fais pour y accéder en Python?

Pas de panique!

Il existe une solution:

```
1 import maya.mel
2
3 main_window = maya.mel.eval('$tmpVar=$gMainWindow')
```

La première ligne importe le module `maya.mel` qui permet d'exécuter du MEL en Python. L'usage est assez restreint mais dans notre cas, la fenêtre est stockée dans une variable en MEL, donc nous n'avons pas le choix.

La seconde ligne évalue (c'est-à-dire exécute) l'expression MEL `$tmpVar=$gMainWindow` et renvoie le résultat dans la variable `main_window`. Dans notre cas, cela nous donne le nom interne de la fenêtre globale de Maya (souvent, sa valeur est `MayaWindow`).

Faites-vous une petite fonction et stockez ça dans un coin, ça peut toujours servir.

```
1 import maya.mel
2
3 def get_maya_main_window():
4     """Renvoit la fenetre principale de Maya.
5     """
6     return maya.mel.eval('$tmpVar=$gMainWindow')
```

Vous avez maintenant *la* fenêtre Maya à disposition et vous pouvez en faire ce que vous voulez (*ça sent la bavure!*):

```
maya_window = get_maya_main_window()
mc.window(maya_window, edit=True, widthHeight=(900, 777))
```

Comme vous l'aurez deviné, la dernière commande change la taille de votre fenêtre Maya.

?

Super... Mais moi je veux ajouter un menu...

C'est qu'on est pressé...

III.4.5.2. Le code, brut de pomme

Tous les prétextes sont bons pour faire du cidre. Voici le code:

```
import maya.cmds as mc

# get main window menu
maya_window = get_maya_main_window()

# top menu
menu = mc.menu('Coucou!', parent=maya_window)

mc.menuItem(label="Another Manager",
    command="print 'another manager'", parent=menu)

# optionnal: add another entry in the menu with a function instead
    of a string
def do_something(arg):
    print "Do something"

mc.menuItem(label="Another Menu", command=do_something,
    parent=menu)
```

Alors? On fait moins le malin pas vrai?

?

OK, tu peux expliquer?

Avec joie! C'est parti!

Notez que vous pouvez exécuter ce code ligne à ligne pour voir son résultat. Et je vous encourage à le faire.

III.4.5.3. Et les explications

Les premières commandes ont déjà été expliquées plus haut, je ne reviens pas dessus.

```
# top menu
menu = mc.menu('Coucou!', parent=maya_window)
```

Cette commande créée un menu nommé *Coucou !* dans votre interface Maya.

```
mc.menuItem(label="Another Manager",
    command="print 'another manager'", parent=menu)
```

Celle-ci créée, comme son nom l'indique, un élément (*item* en anglais) du menu. Cet élément s'appellera *Another Manager* dans l'interface, aura pour parent le menu qu'on aura créé précédemment et exécutera le code `print 'another manager'` quand on cliquera dessus.

Notez la similitude avec la commande `button()` vu précédemment, notamment concernant l'argument `command`.

```
def do_something(arg):
    print "Do something"
```

Ici, on définit une fonction qui va simplement afficher *Do something* dans votre *script editor*, mais vous pourriez mettre votre code qui ouvre une fenêtre avec vos outils dedans.

```
mc.menuItem(label="Another Menu", command=do_something,
    parent=menu)
```

Et la dernière commande crée, encore `menuItem`, mais cette fois-ci, elle appellera la fonction `do_something()` quand on cliquera dessus.

Encore une fois, rappelez-vous les fonctions de rappel de l'argument `command` de la commande `button()`.

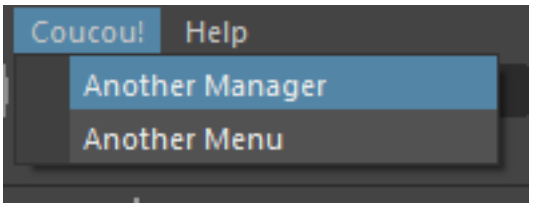


FIGURE III.4.17. – "Coucou !" Notre menu personnalisé.

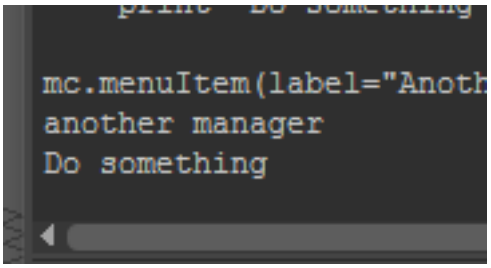


FIGURE III.4.18. – Et son résultat dans le *Script Editor*.

Et voilà ! 🍊

Encore une fois, je vous invite à tester les exemples en fin de documentation de `menu` et `menuItem` .

Conclusion

Nous savons maintenant comment créer de petites interfaces. Nous n'avons qu'effleuré ce qu'il était possible de faire, mais je vous invite à aller plus loin par vous-même et à exécuter les codes d'exemples en fin de chacune des pages de la documentation:

- `text` , qui permet d'afficher de simples textes et de les aligner.
- `checkBox` pour permettre à l'utilisateur de cocher/décocher des cases, et ainsi prendre en compte ses choix (via l'argument `query` dans l'exécution des scripts).
- `floatSlider` pour afficher et récupérer des valeurs de l'utilisateur. Tapez «slider» dans la page de recherche pour trouver ses frères.
- `frameLayout` pour organiser de grosses interfaces.
- `treeLister` si on aime les nuits blanches aspirines.
- `progressWindow` pour faire attendre les gens comme un pro. 🍊
- etc.

Gardez cependant à l'esprit l'existence de *PySide* intégré à Maya. Si vous commencez à passer plusieurs heures sur votre interface, alors peut-être est-il temps de franchir le pas et d'entamer un tutoriel dédié à PySide. 🍊

III.5. Modifier sa scène

Introduction

Dans ce petit chapitre, nous allons créer des nœuds et voir comment on peut interagir avec eux. Vous apprendrez aussi à transformer la sortie MEL de Maya en commande Python.

III.5.1. Créer des nœuds avec des commandes dédiées

C'est bien joli, mais nous n'avons toujours pas créer de nœuds nous-même.

Il existe de très nombreuses façons de créer des nœuds dans Maya :

— soit par une commande dédiée contenant ses propres arguments tel que `polySphere()`, qui s'occupe des connexions pour vous suivant les arguments choisis;
— soit implicitement, par une commande nécessitant des nœuds déjà présents dans votre scène telle que `revolve()`, qui crée une nouvelle géométrie depuis une courbe donnée et s'occupe elle aussi des connexions pour vous;
— soit explicitement via l'utilisation de la commande `createNode()` qui ne crée qu'un seul nœud et vous laisse vous occuper des connexions.

Je vais vous en présenter deux.

III.5.1.1. spaceLocator()

Vous le savez sûrement mais les *locators* sont de petits objets qui s'affichent dans le viewport Maya :

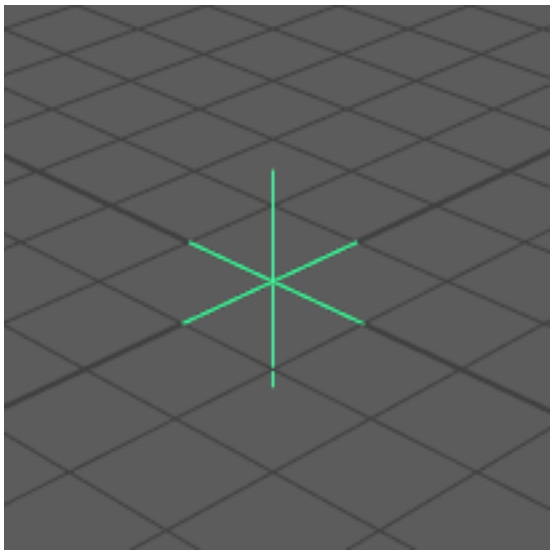


FIGURE III.5.1. – Un locator.

Pour le créer, rien de plus simple :

```
1 mc.spaceLocator()
```

Vous pouvez lui donner un nom :

```
1 mc.spaceLocator(name="monLocator")
```

Il y a quelques autres arguments mais je ne rentre pas dans les détails.

III.5.1.2. polySphere()

On l'a déjà utilisée celle-là, mais on va creuser un peu plus cette fois.

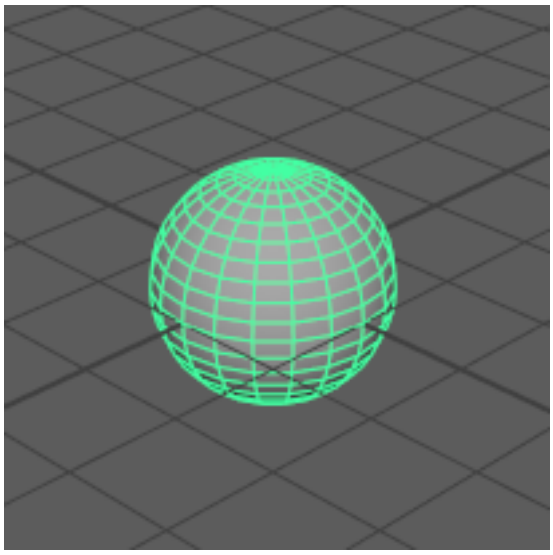


FIGURE III.5.2. – Une sphère.

Pour la créer, rien de plus simple :

```
mc.polySphere()
# Result: [u'pSphere1', u'polySphere1'] #
```

Vous remarquerez que cette commande renvoie deux variables: le nœud de `transform` et le nœud de `polySphere`.

On peut aussi lui donner un rayon via l'argument `radius` :

```
mc.polySphere(radius=5.0)
# Result: [u'pSphere2', u'polySphere2'] #
```

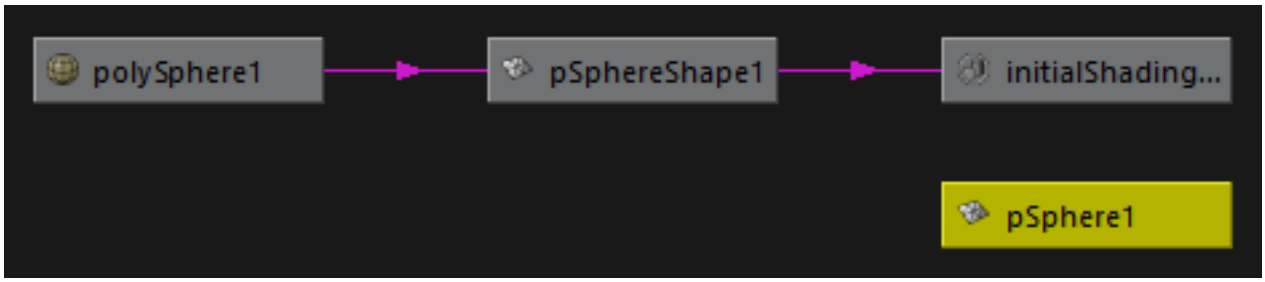
Et une subdivision à notre goût via les arguments `subdivisionsX` et `subdivisionsY` :

```
mc.polySphere(subdivisionsX=10, subdivisionsY=5)
# Result: [u'pSphere3', u'polySphere3'] #
```

Et bien sûr un nom via l'argument `name` :

```
mc.polySphere(name="mySphere")
# Result: [u'mySphere', u'polySphere4'] #
```

Par défaut, `polySphere()` crée un graphe de nœuds pour vous :



L'argument `constructionHistory=False` permet de ne pas conserver ce que Maya appelle le *construction history* (*historique de construction* en français).

```
1 mc.polySphere(constructionHistory=False)
2 # Result: [u'pSphere1'] #
```

Remarquez comment le second nœud n'est pas présent dans la liste.

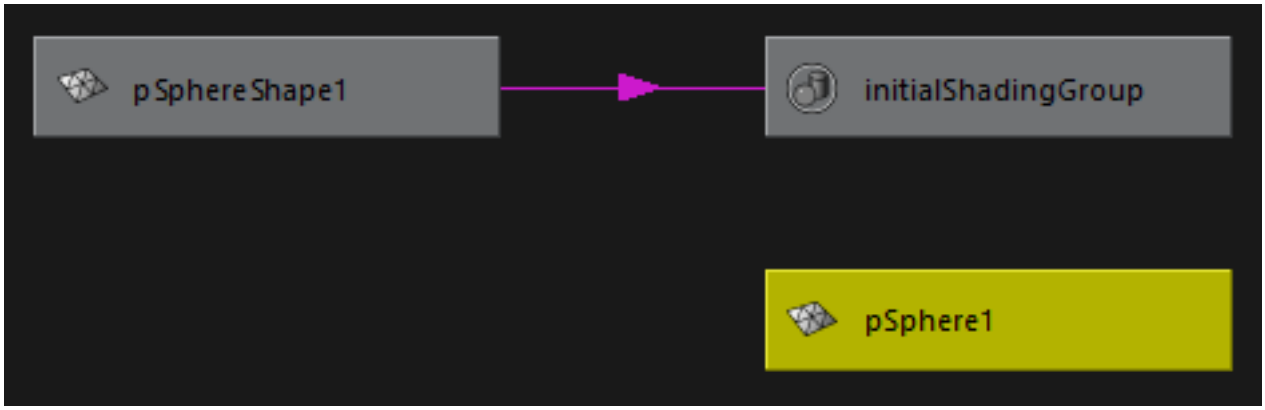


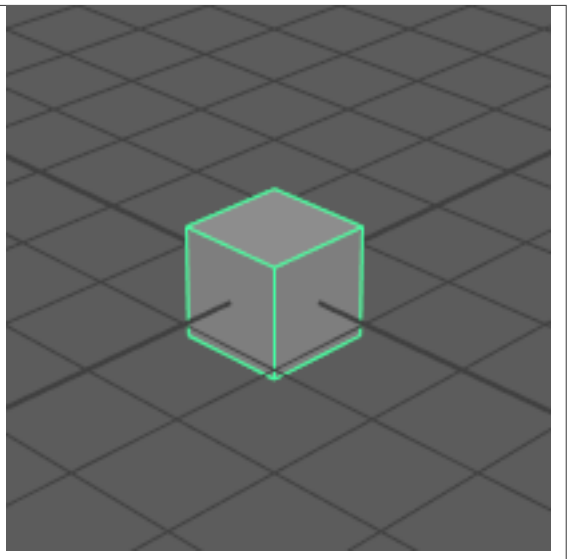
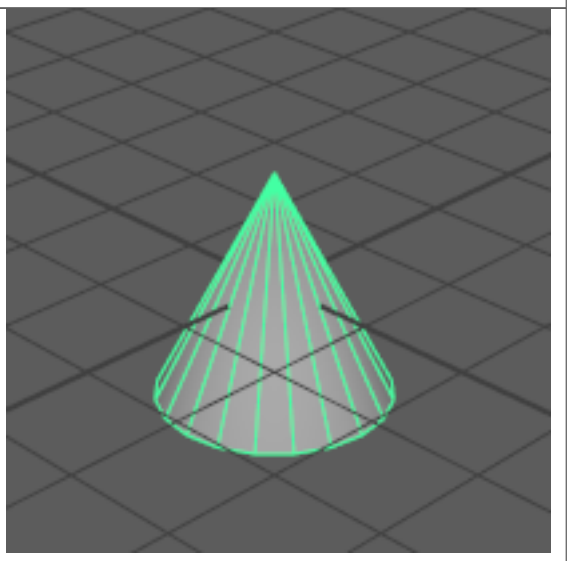
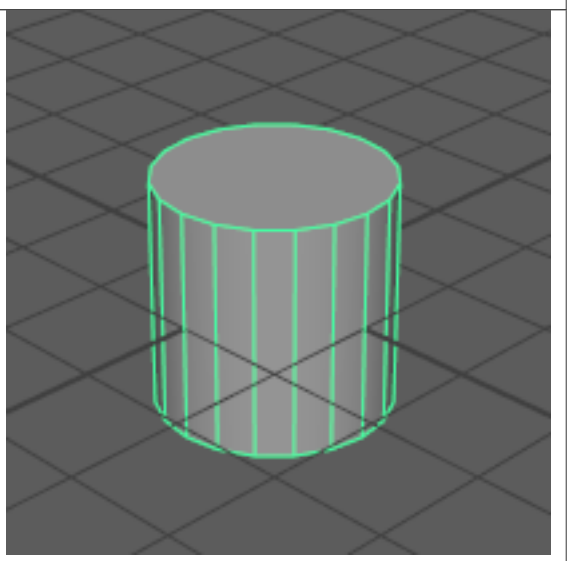
FIGURE III.5.3. – Ni dans le graphe.

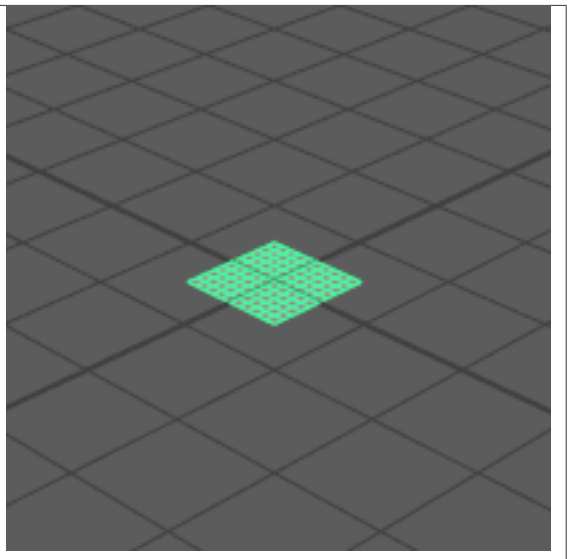
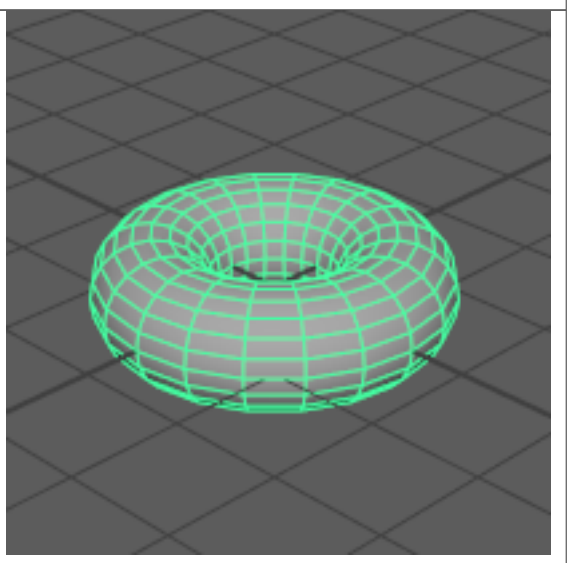
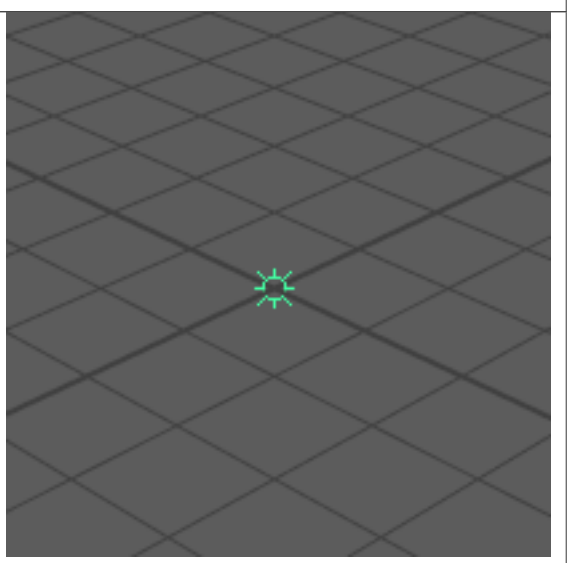
Encore une fois, regardez la documentation si vous souhaitez aller plus loin.

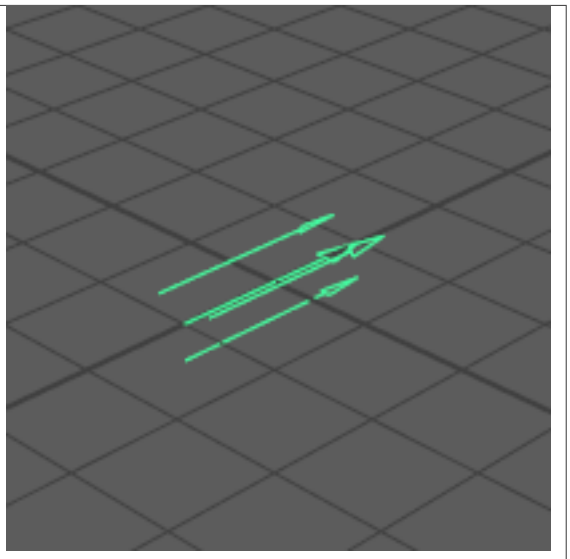
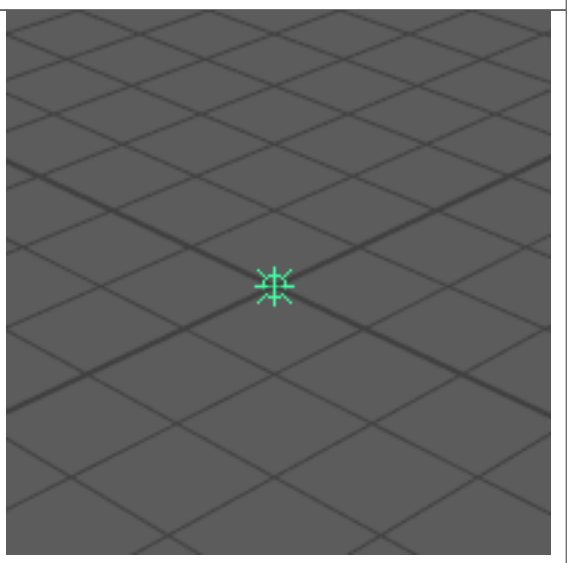
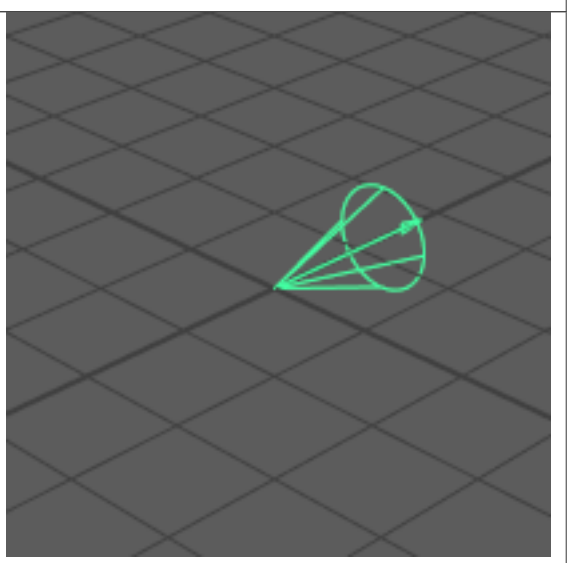
III.5.1.3. Et les autres

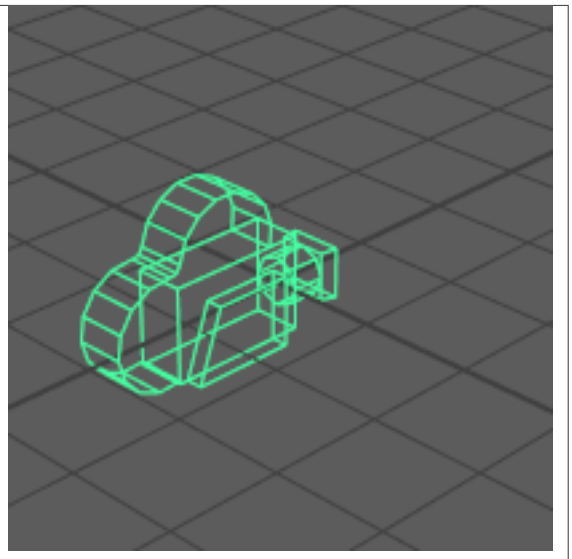
Je ne vais pas toutes les faire, il y a en a plein d'autres, mais je vous invite à jeter un œil sur la documentation de chacune d'elle. Voici une liste non exhaustive, n'hésitez pas à fouiller dans l'index .

Commande	Aperçu

`polyCube()`	
`polyCone()`	
`polyCylinder()`	

`polyPlane()`	
`polyTorus()`	
`ambientLight()`	

directionalLight()	
pointLight()	
spotLight()	

`camera()`	

À cela vous pouvez ajouter les commandes `curve()`, `group()` (avec l'argument `empty=True`), `curveOnSurface()`, `createRenderLayer()`, etc.

En général, quand vous souhaitez créer un truc, il y a une commande pour ça.

III.5.2. edit et query sont sur un bateau...

Je vais vous présenter un concept important des commandes Maya. Il n'est pas particulièrement compliqué mais revient souvent (vous l'avez déjà utilisé pour les interfaces). Il vaut donc mieux s'y être familiarisé.

On va commencer fort. Exécutez-moi ça :

```
1 mc.file(new=True, force=True)
2 mc.polySphere(name='pSphere1')
3 print mc.polySphere('pSphere1', query=True, radius=True)
4 mc.polySphere('pSphere1', edit=True, radius=5)
```

Boum ! Comme ça, sans prévenir.

La première ligne c'est juste la commande `file()` avec l'argument `new` pour partir d'une nouvelle scène et la commande `force` pour ne pas demander si vous voulez sauvegarder votre scène actuelle (la doc ici).

La seconde commande vous la connaissez déjà, elle crée une `polySphere` nommée `pSphere1`:

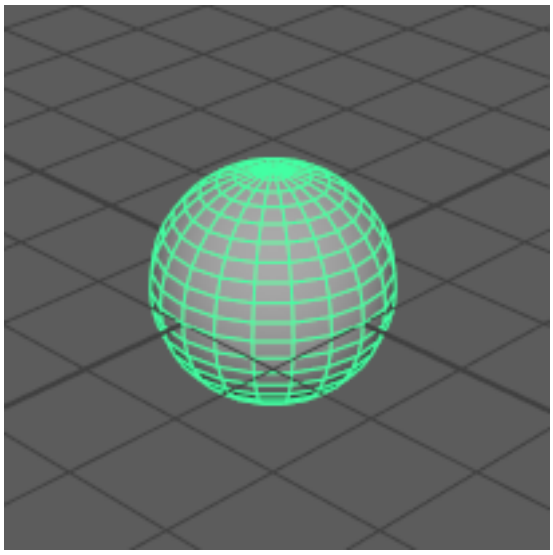


FIGURE III.5.4. – Une polySphere.

Ce sont les deux dernières qui vont nous intéresser :

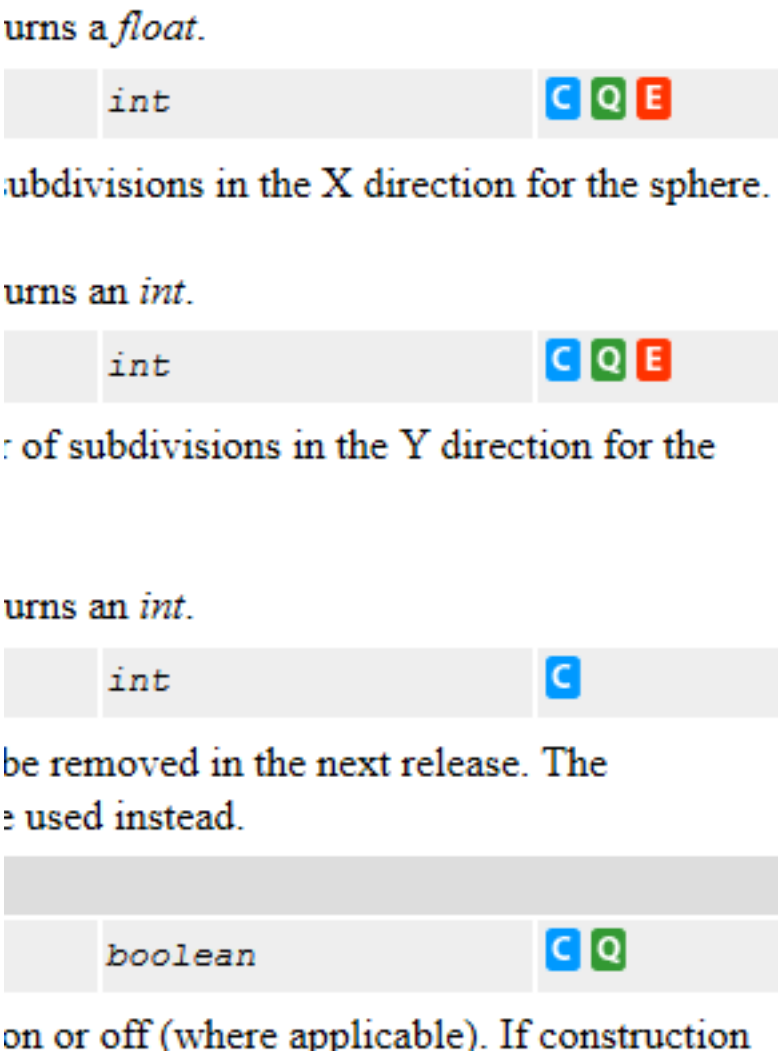
```
print mc.polySphere('pSphere1', query=True, radius=True)
mc.polySphere('pSphere1', edit=True, radius=5)
```

? Quels sont ces arguments `query` et `edit`? Ils n'apparaissent pas dans la documentation… 🍊

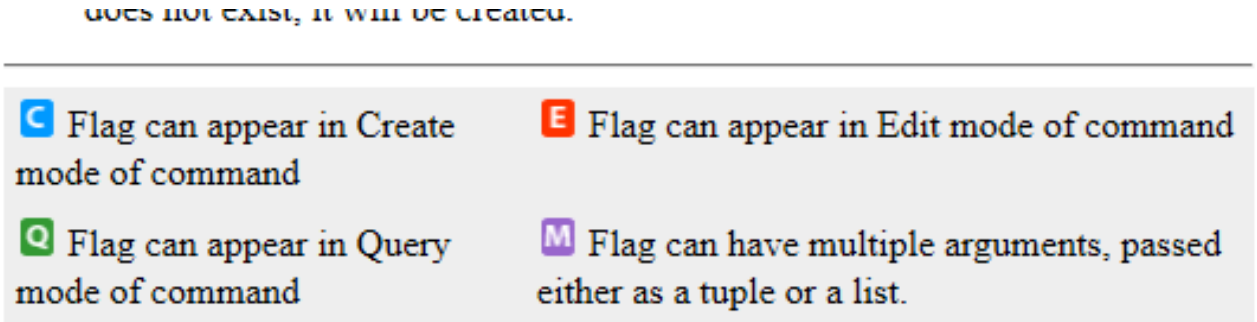
Vous commencez à avoir les bons réflexes, c'est bien. 🍊

En effet, ce sont des arguments spéciaux.

Comme il va de soi que vous vous servez allègrement de la documentation (n'est-ce pas ? 🍊), vous avez sûrement remarqué les trois lettres C, Q et E (et parfois M mais on ne s'en occupe par pour l'instant) à droite des arguments :



Et comme vous êtes attentif, curieux et à toujours chercher plus loin, l'encart présent dans toutes les pages de la documentation, au-dessus des exemples Python ne vous aura pas échappé : 🍍



Ces lettres indiquent que l'argument auquel elles se réfèrent peut s'utiliser suivant le *mode de commande*.

Un *mode de commande*? Mais c'est quoi ?

Ne paniquez pas, je vous l'explique immédiatement.

III.5.2.1. Les modes d'utilisation des commandes

Il existe trois *modes de commande*. Je vais utiliser la commande `polySphere()` avec l'argument `radius` car ce dernier permet une utilisation avec les trois modes.

III.5.2.1.1. *Create* (lettre C)

C'est le mode implicite, il est inutile de le préciser. Ainsi :

```
1 mc.polySphere(radius=5.0)
2 # Result: [u'pSphere1', u'polySphere1'] #
```

Exécute la commande `polySphere()` en mode `create`. C'est le mode que nous utilisons depuis le début.

III.5.2.1.2. *Query* (lettre Q)

Ce mode s'active lorsqu'on ajoute l'argument `query` à la commande. Ce mode *query* (qui peut se traduire par *requête* ou *question*) permet de récupérer des informations sur la valeur d'un argument donné (ici, `radius`):

```
1 mc.polySphere("pSphere1", query=True, radius=True)
2 # Result: 5.0 #
```

Oh! La commande renvoie la valeur flottante `5.0`.

C'est l'idée!

Notez que l'argument `radius` n'est pas un chiffre mais un `bool`. Dans ce contexte, la commande peut se traduire par *donne-moi la valeur de l'argument `radius` pour la sphère `"pSphere1"`*.

Le mode `query` ne peut renvoyer qu'une seule valeur et implique une certaine logique dans la *question* qu'on pose. Ainsi, ceci n'est pas possible :

```
1 mc.polySphere("pSphere1", query=True, radius=True,
      subdivisionsX=True)
```

On ne peut *questionner* qu'une valeur à la fois.

III.5.2.1.3. *Edit* (lettre *E*)

Comme vous le devinez surement l'argument `edit` permet d'éditer/de modifier un objet déjà existant. Je vous laisse deviner ce que fait cette commande :

```
1 mc.polySphere("pSphere1", edit=True, radius=2.0)
```

Bien sur, elle modifie le radius de `"pSphere1"`.

? Quelle est la différence entre cette commande et `mc.setAttr("polySphere1.radius", 2.0)`?

Déjà, le fait que, quand vous avez `"pSphere1"` dans une variable, il faut passer par des commandes subsidiaires (que nous verrons plus tard) pour récupérer `"polySphere1"`, le nœud qui "génère" la géométrie de la sphère (ne croyez pas qu'il suffit de remplacer la lettre *p* par *poly* pour que ça marche à tous les coups).

`edit` est fait pour vous éviter ce genre de manipulations et fait souvent des choses que vous pourriez faire autrement, mais en plus de lignes. Je vous accorde que dans le cas de `polySphere()` l'économie n'est pas si évidente, mais pour certaines commandes vous verrez que c'est quand même plus simple.

Au-delà de ce simple exemple, certaines commandes sont vraiment pensées pour être utilisées avec `query` et `edit` (je pense notamment à `xform()` qui a son chapitre dédié) et nous avons également vu que les interfaces nécessitent obligatoirement de passer par `query` et `edit`.

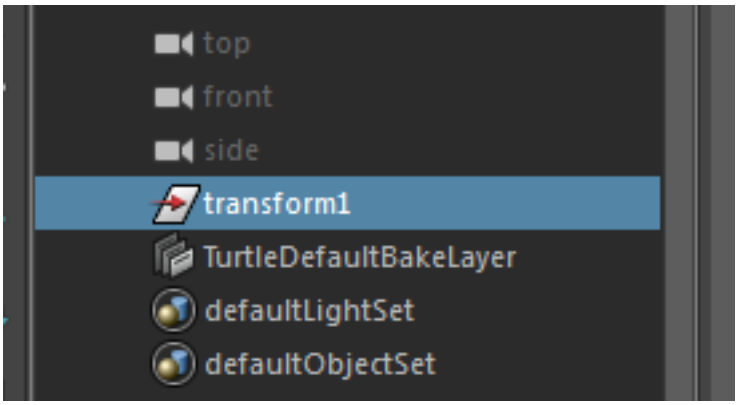
III.5.3. createNode() pour créer des nœuds

La dernière commande du chapitre est la plus générique de toutes. C'est une de celle qu'on utilise le plus.

III.5.3.1. Utilisation de base

Je vous ai montré comment créer des *locators*, des sphères, des *torus*, etc. Et bien ici je vais vous montrer comment créer un simple nœud :

```
mc.createNode("transform")
# Result: u'transform1' #
```



L'argument principal est le type de nœud qu'on souhaite créer. Ici, un nœud de type `transform`.

Mais passons vite à la suite !

III.5.3.2. Une histoire de sélection

Exécutez ceci (après l'avoir lu et compris bien entendu , sinon retournez au chapitre concernant la commande `ls()`) :

```
mc.select("persp")
print "selection avant:", mc.ls(selection=True)
mc.createNode("transform")
print "selection apres:", mc.ls(selection=True)
```

Le résultat de ce bout de code devrait vous afficher :

```
selection avant: [u'persp']
selection apres: [u'transform1']
```

?

Ah, en fait la commande `createNode()` change ma sélection ?

Exactement ! 🍊

Comme vous l'avez sûrement remarqué, le nœud nouvellement créé (ici, `transform1`) est automatiquement sélectionné. Si vous ne souhaitez pas perdre votre sélection, il est possible d'empêcher la sélection automatique en utilisant l'argument `skipSelect`.

Maintenant voici le même code, mais en utilisant l'argument `skipSelect`:

```
1 mc.select("persp")
2 print "selection avant:", mc.ls(selection=True)
3 mc.createNode("transform", skipSelect=True)
4 print "selection apres:", mc.ls(selection=True)
```

Avec le résultat :

```
1 selection avant: [u'persp']
2 selection apres: [u'persp']
```

Magie ! La sélection n'est pas modifiée par la commande `createNode()` ! 🧙

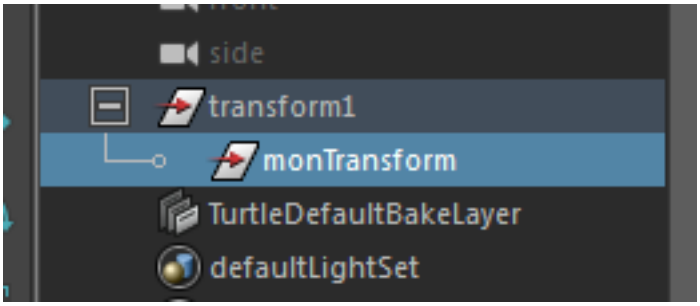
III.5.3.3. Une année « parent »

Des jeux de mot qui tabassent quand la caisse claire fracasse. 🍊

Viens l'argument `parent` qui spécifie un parent sur le nœud qu'on souhaite créer.

```
1 mc.createNode("transform", parent="transform1",
      name="monTransform")
```

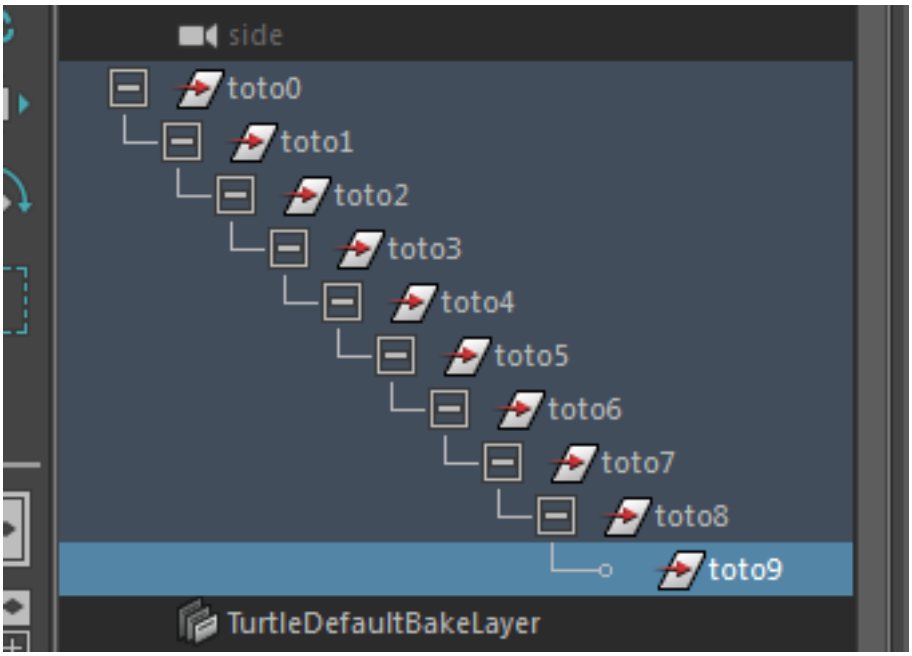
i Je vous ai mis l'argument `name` en bonus, car vous savez déjà à quoi il sert, pas vrai ?



Ici on vient de créer un nœud nommé `monTransform` en tant qu'enfant du nœud créé précédemment (`transform1`).

Mais qu'à cela ne tienne, j'ai envie de faire une boucle pas vous ? Comme ça, pour le fun :

```
current_parent = None
for i in range(10):
    current_parent = mc.createNode("transform",
        parent=current_parent, name="toto"+str(i))
```



? Euh… Tu peux m'expliquer ?

Bien entendu !

La première ligne initialise une variable (`current_parent`) contenant le nom du parent (pour l'instant, `None`).

Puis viens une boucle qui va s'exécuter dix fois, de 0 à 9, via la fonction `range()` (que vous avez sûrement dû voir lors de vos premiers tutoriels sur les boucles en Python):

```
for i in range(10):
    current_parent = mc.createNode("transform",
        parent=current_parent, name="toto"+str(i))
```

Comme vous vous doutez, l'argument `name` (à la fin de l'appel de la fonction) va juste générer un nouveau nom à chaque itération : `toto0`, `toto1`, `toto2`, etc. Rien de bien compliqué pour qui connaît son chapitre sur les boucles en Python.

Rappel : Une itération c'est *un roulement de boucle*. Chaque fois que la boucle est exécutée, on dit qu'une itération a été effectuée. Il s'agit juste de terminologie, mais il est plus simple de dire *une itération* plutôt qu'*un roulement de boucle*.

Ce qui nous intéresse c'est, bien sûr, l'argument `parent`.

La première fois que la boucle va s'exécuter (*la première itération* donc), l'argument `parent` sera mis à `None` (car `current_parent` est à `None`) ce qui veut dire que c'est comme si nous ne passions pas l'argument `parent`. Ainsi :

```
1 mc.createNode("transform", parent=None)  # argument parent a None
```

Est équivalent à :

```
1 mc.createNode("transform")  # pas d'argument parent
```

La première itération va donc créer le premier nœud (*toto0*) sans parent (ce qui a pour effet de créer le nœud à la racine) et modifier la variable `current_parent` qui passe de `None` à *toto0*.

À la seconde itération, le nouveau nœud (*toto1*) aura comme parent le nœud de l'itération précédente (à savoir, *toto0*), ce qui permet de fabriquer cette hiérarchie.

Si on «déroule» la boucle, ça donne la chose suivante :

```
1 mc.createNode("transform", parent=None, name="toto0")  # premiere
    iteration
2 mc.createNode("transform", parent="toto0", name="toto1")  #
    seconde iteration
3 mc.createNode("transform", parent="toto1", name="toto2")  #
    troisieme iteration
4 # etc.
```

J'espère que c'est plus clair. 🍊

Je crois que je comprends un peu mieux, mais ce n'est pas encore ça. 🍊

Ne vous inquiétez pas, il est normal de ne pas comprendre tout ce qu'on peut faire avec une boucle immédiatement. C'est à force de manipuler et d'en faire que ça deviendra une seconde nature.

Dans tous les cas, je vous invite à repasser quelques minutes sur cette boucle. Elle est simple (et un peu inutile je vous le concède) et le but de tout scripter c'est pour justement faire des boucles et automatiser son travail.

Utilisez des `print` partout pour réussir à comprendre ce qui se passe. Par exemple, le script suivant fait exactement la même chose que le script original, j'ai juste ajouté des commandes `print` pour disséquer ce qui se passe quand la boucle s'exécute :

```
1  current_parent = None
2  print "current_parent", current_parent
3  for i in range(10):
4      print "=== iteration", i, "==="
5      print "createNode with parent =", current_parent
6      current_parent = mc.createNode("transform",
           parent=current_parent, name="toto"+str(i))
7      print "next parent will be", current_parent
```

Ce qui nous donne :

```
1  current_parent None
2  === iteration 0 ===
3  createNode with parent = None
4  next parent will be toto0
5  === iteration 1 ===
6  createNode with parent = toto0
7  next parent will be toto1
8  === iteration 2 ===
9  createNode with parent = toto1
10 next parent will be toto2
11 === iteration 3 ===
12 createNode with parent = toto2
13 next parent will be toto3
14 === iteration 4 ===
15 createNode with parent = toto3
16 next parent will be toto4
17 === iteration 5 ===
18 createNode with parent = toto4
19 next parent will be toto5
20 === iteration 6 ===
21 createNode with parent = toto5
22 next parent will be toto6
23 === iteration 7 ===
24 createNode with parent = toto6
```

```
next parent will be toto7
=== iteration 8 ===
createNode with parent = toto7
next parent will be toto8
=== iteration 9 ===
createNode with parent = toto8
next parent will be toto9
```

Essayez de comprendre ça, ce n'est pas du temps perdu croyez-moi.

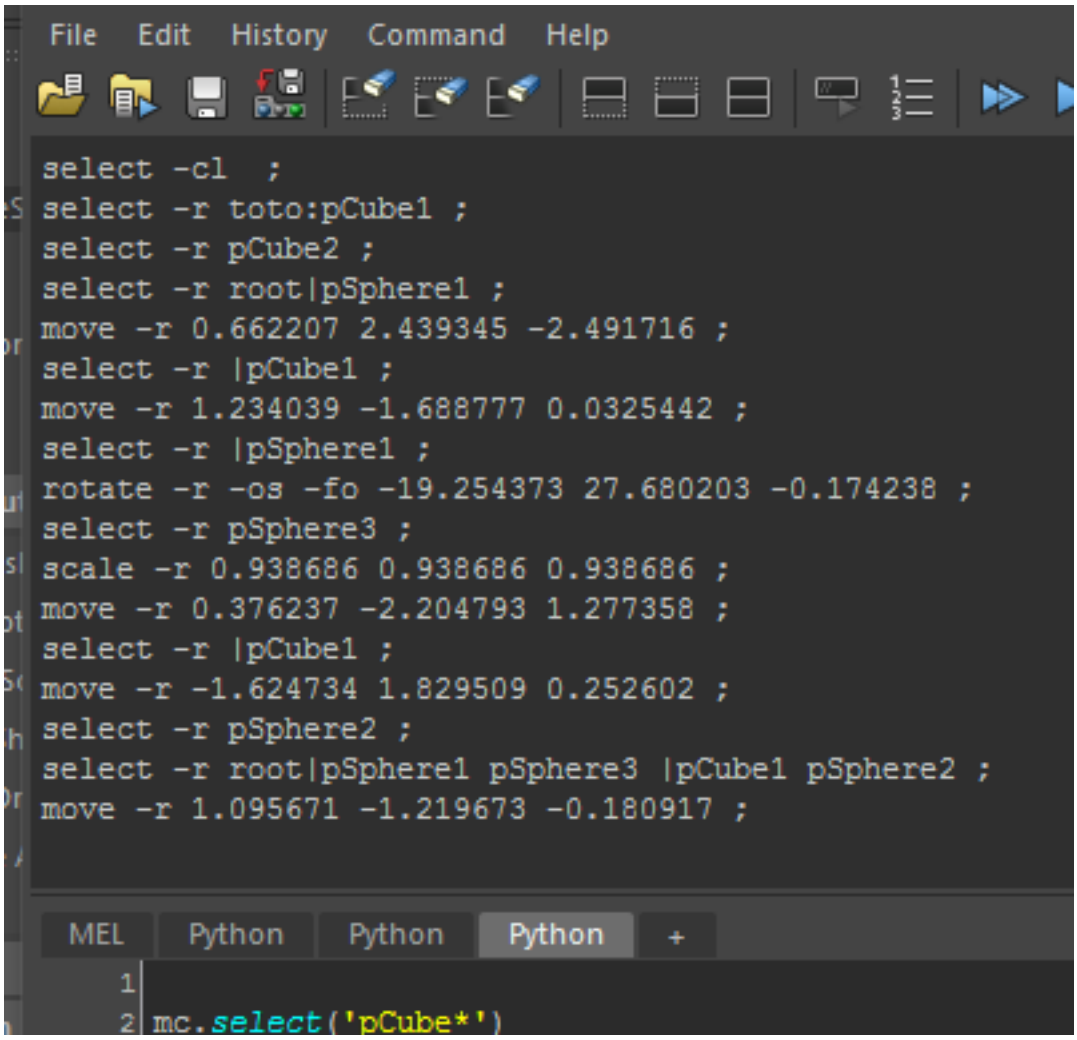
C'est tout pour la commande `createNode()`. Un gros morceau comme vous pouvez le voir.

III.5.4. Convertir une commande MEL en Python

Arrivé ici, vous devriez déjà avoir de quoi jouer un peu. Mais quoi de mieux pour finir un chapitre sur une ouverture?

Quand vous cliquez sur un objet, que vous le déplacez, que vous modifiez la valeur d'un attribut, que vous ouvrez un fichier, que vous créez un objet, bref, que vous travaillez, la sortie de votre *Script Editor* affiche les équivalents, en MEL, des commandes.

Si vous êtes un peu bricoleur, vous avez sûrement déjà dû copier-coller et exécuter ces petits blocs de code qui ressemblent à ça :



En version textuelle, ça donne ça:

```
polySphere -r 1 -sx 20 -sy 20 -ax 0 1 0 -cuv 2 -ch 1;
select -r pSphere1;
select -tgl pSphere2 ;
select -tgl pSphere3 ;
scale -r 0.848105 0.848105 0.848105 ;
rotate -r -os -fo -14.111463 -12.135095 6.09803 ;
move -r -0.52468 0.29218 0.242561 ;
select -cl  ;
select -r pSphere2 ;
setAttr "pSphere2.visibility" 0;
```

Comme je vous le dis juste au-dessus, ces commandes sont en MEL, mais elles sont en fait très facile à convertir en leur équivalent Python, et c'est ce que nous allons faire ensemble.

III.5.4.1. Comparer les commandes MEL et leur équivalent Python

Je viens de reprendre les commandes du dessus et j'ai fait un tableau avec leur équivalent en Python.

J'aimerais que vous les regardiez attentivement pour que vous puissiez distinguer la «forme» et les similitudes qu'il peut y avoir entre les deux versions, l'objectif étant de comprendre comment passer de l'une à l'autre :

MEL	Python
`polySphere -r 1 -sx 20 -sy 20 -ax 0 1 0 -cuv 2 -ch 1 ;`	`mc.polySphere(r=1, sx=20, sy=20, ax=[0, 1, 0], cuv=2, ch=1)`
`select -cl ;`	`mc.select(cl=True)`
`select -r pSphere1 ;`	`mc.select("pSphere1 ", r=True)`
`select -tgl pSphere2 ;`	`mc.select("pSphere2", tgl=True)`
`move -r -0.52468 0.29218 0.242561 ;`	`mc.move(-0.52468, 0.29218, 0.242561, r=True)`
`scale -r 0.848105 0.848105 0.848105 ;`	`mc.scale(0.848105, 0.848105, 0.848105, r=True)`
`rotate -r -os -fo -14.111463 -12.135095 6.09803 ;`	`mc.rotate(-14.111463, -12.135095, 6.09803, r=True, os=True, fo=True)`
`setAttr "pSphere2.visibility" 0;`	`mc.setAttr("pSphere2.visibility", 0)`

La première chose à noter est qu'il n'y a pas de point-virgule `;` à la fin des commandes Python. C'est une des particularités du langage : ses *règles de portées* (*scoping Rules* en anglais) basées sur l'organisation du code (tabulations, `:`, espaces, retours à la ligne, etc.) font qu'il n'est pas nécessaire de faire une séparation entre les commandes. 😎

La seconde particularité, c'est que les `-quelqueChose` en MEL (`-cl`, `-tgl`, `-r`, etc.) deviennent des `quelqueChose=True` en Python (`cl=True`, `tgl=True`, `r=True`, etc.). C'est dû au fait que si vous passez un argument sans valeur, Python croit que c'est une variable. Ainsi, si vous essayez de convertir cette commande MEL :

```
1 uneCommande -quelqueChose
```

...en Python, sous la forme :

```
1 mc.uneCommande(quelqueChose)
```

Python s'attendra à ce que `quelqueChose` soit une variable, et non un argument. Il faut donc lui passer `quelqueChose=True` pour simuler la même chose que le `-quelquesChose` en MEL.

Ensuite viennent les chaînes de caractère. En MEL, `pSphere1` ne peut pas être autre chose qu'une chaîne de caractère car une variable est toujours préfixée d'un `$` (`$maVariable` est une variable, `toto` est une chaîne de caractère). Je ne rentre pas dans les détails, ce n'est pas le MEL qu'on cherche à apprendre ici ^^. En Python, une chaîne de caractère est forcément entre des *guillemets* (*quote* en anglais), à savoir : `"pShere1"` ou `'pShere1'`.

Dernier point un peu plus subtil : la place de l'argument principal. Les commandes MEL tendent à mettre l'argument principal à la fin de la commande. On obtient donc le schéma suivant :

```
maCommand -argument1 -argument2 "node"
```

En Python, cela donnerait plutôt :

```
mc.maCommande("node", argument1=True, argument2=True)
```

Voyez-vous où `"node"` est placé ?

C'est encore une fois dû à des différences entre MEL et Python ; le dernier prenant les arguments «non nommés» (ici, `"node"`) en début de fonction et les arguments «nommés» (`argument1` et `argument2`) ensuite.

La différence d'ordre est particulièrement visible sur la commande `select` (cf : tableau).

Idem pour les positions des commandes `move`, `rotate` et `scale` qui se retrouvent au début de la commande en Python.

? Comment puis-je savoir où exactement placer l'argument ?

Encore et toujours, en comparant les exemples de la doc.

Alternez en cliquant sur le bouton *MEL version* et *Python version* (en haut à droite) et vous verrez immédiatement où se place l'argument.

D'une manière générale, c'est toujours l'opposé de la version MEL, mais il faut vous y faire, *doc rule the world*!

III.5.4.2. À votre tour !

Nettoyez l'historique de la sortie de votre *Script Editor* :

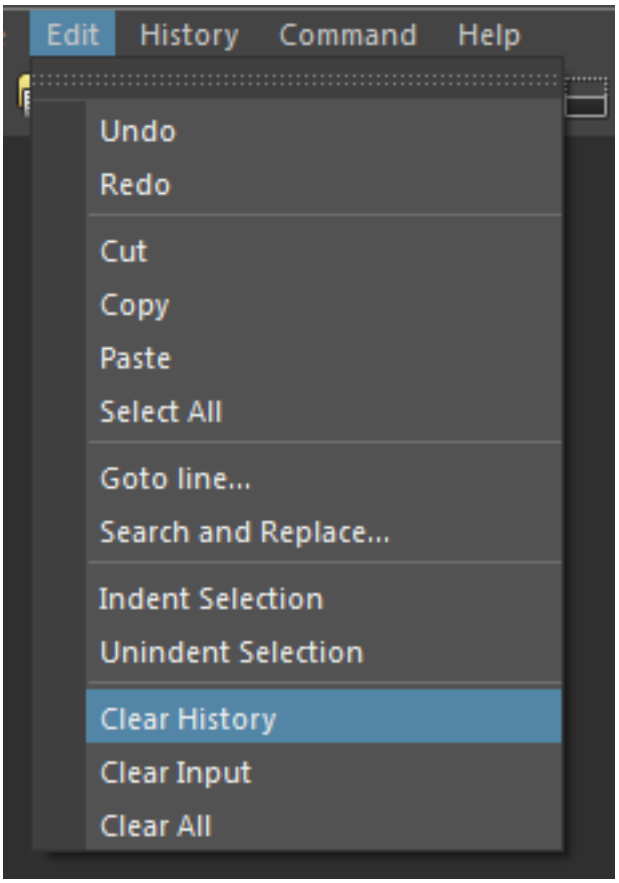


FIGURE III.5.5. – Nettoyez l'historique du *Script Editor* par le menu

Ou :

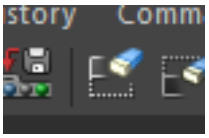


FIGURE III.5.6. – Nettoyez l'historique du *Script Editor* par l'icône

Faites *File/Create Reference...* et amenez un fichier Maya (n'importe lequel) en référence.

Suivant la scène en question, il est possible que la sortie du *Script Editor* vous affiche quelques erreurs.

Remontez l'historique pour chercher la commande qui a fait l'import en référence. Chez moi, j'obtiens ceci :

```
file -r -type "mayaAscii"  -ignoreVersion -gl
    -mergeNamespacesOnClash false -namespace "test" -options
    "v=0;p=17;f=0" "C:/Users/narann/Documents/maya/projects/defaul
    t/scenes/test.ma";
```

Avec ce que je vous ai montré plus haut, vous devriez vous en sortir tout seul. Prenez votre temps, le but n'est pas que vous soyez rapide mais que vous assimiliez la méthode.

Rappelez-vous :

- On trouve la commande (ici `file`) et on ajoute le module Maya devant (ce qui nous donne `mc.file()`).
- On supprime le point virgule `;` à la fin de la commande.
- On passe l'argument principal en début de commande.
- On remplace tous les arguments avec un tiret devant (comme `-gl`) par la forme en Python (`gl=True`).
- Les arguments ayant des valeurs (`-mergeNamespacesOnClash false` par exemple) on les arrange en équivalent Python (`mergeNamespacesOnClash=False`).
- On met des virgules entre chaque argument `,`.
- Dans le doute, on va zieuter les exemples en fin de doc (et oui… toujours la doc.)
- On convertit les arguments en version courte (`-r` par exemple) vers leur équivalent entier (`-reference`)

Bon, je vous ai quand même pas mal mâché le travail bande de feignasses.

Quand vous avez fait ça, vous devriez pouvoir exécuter votre ligne de code dans un onglet Python et que ça fonctionne.

> Avant de regarder la solution, si vous avez une erreur, lisez-la et modifiez votre commande en conséquence. Plus vous apprendrez à faire ce genre de conversion, plus vous serez à l'aise et aurez tendance à le faire instinctivement, à chaque fois que vous aurez des tâches répétitives à faire.

Et la solution, que vous ne devriez ouvrir qu'à titre de comparaison, une fois que votre commande fonctionne.

Contenu masqué n°2

Vous savez maintenant comment importer un fichier en référence avec un *namespace* personnalisé. De quoi, je l'espère, automatiser pas mal de taches redondantes dans votre quotidien.

Conclusion

Voila, ce chapitre n'était peut-être pas le plus excitant, mais il a abordé les derniers concepts importants autour des commandes Maya. À partir de maintenant, nous allons aborder plus de commandes sous forme d'exemples pratiques.

L'aventure ne fait que commencer! 🤠

Contenu masqué

Contenu masqué n°2

```
1 mc.file(
      "C:/Users/narann/Documents/maya/projects/default/scenes/test.ma"
      , reference=True, type="mayaAscii", ignoreVersion=True,
      groupLocator=True, mergeNamespacesOnClash=False,
      namespace="test", options="v=0;p=17;f=0")
```

Retourner au texte.

III.6. TP : Renommer ses nœuds avec classe

Introduction

Quoi de plus fatiguant que de renommer des nœuds en masse dans Maya?

Maya dispose de quelques outils, mais ils ne font jamais exactement ce qu'on souhaite, pas vrai ?

On est tous passés par là, mais c'est dans ces moments qu'on se dit qu'il serait peut-être temps d'apprendre à scripter deux-trois trucs. Peut-être même est-ce la raison pour laquelle vous avez décidé de commencer ce tutoriel?

Eh bien réjouissez-vous, car on va attaquer l'automatisation de ce travail ingrat qu'est le renommage des nœuds.

III.6.1. Retrouver des nœuds avec un nom pas défaut

Notre premier bout de code va se concentrer sur la récupération des nœuds ayant un nom par défaut.

> On va les débusquer ces fripons!

Vous êtes d'attaque, j'aime ça!

III.6.1.1. La commande de base

Sans trop de surprises, la commande qui va nous permettre de trouver les noms par défaut est `ls()`:

```
1 mc.ls('*pSphere*', type='transform')
```

Ici, on cherche tous les nœuds de type `transform` ayant `pSphere` dans leur nom.

Notez que j'ai mis l'argument `type='transform'` car ce que nous souhaitons renommer, ce sont les nœuds de transformation (c'est-à-dire que l'on peut sélectionner).

Sur ma petite scène, cela donne:

```
[u'root|pSphere1', u'|pSphere1', u'pSphere2', u'pSphere3']
```

Elle n'est pas superbement bien nommée cette scène, du travail de sagouin ! 🍊

Mais alors, il suffirait d'avoir une liste de noms par défaut, de leur ajouter `*` avant et après pour avoir notre liste de nœuds mal nommés ? 🍊

Bien vu ! On peut essayer ! 🍊

III.6.1.2. Premier script

```
default_names = ['pSphere', 'pCube', 'pPlane', 'pPyramid',
    'locator']

for default_name in default_names:

    name_expr = '*'+default_name+'*'

    print mc.ls(name_expr, type='transform')
```

Ligne à ligne, cela donne:

```
default_names = ['pSphere', 'pCube', 'pPlane', 'pPyramid',
    'locator']
```

La première chose ici, consiste à lister les noms par défaut des nœuds qu'on retrouve le plus. Pour ce tuto, je vais m'en tenir à ceux-là, mais vous pouvez augmenter cette liste suivant vos besoins. 🍊

Notez qu'on utilise `default_names` (avec un *s*) qui est le nom de la liste. 🍊

```
for default_name in default_names:
```

Nous allons ensuite la parcourir en prenant les noms un à un.

```
    name_expr = '*'+default_name+'*'
```

Puis nous fabriquons une expression en ajoutant les étoiles devant et derrière le nom. Ainsi, `"pSphere"` devient `"*pSphere*"`, `"pCube"` devient `"*pCube*"`, etc.

```
    print mc.ls(name_expr, type='transform')
```

Enfin, nous affichons le résultat de la commande `ls()` pour chacun des noms.

Chez moi, cela donne:

```
[u'root|pSphere1', u'|pSphere1', u'pSphere2', u'pSphere3']
[u'root|pCube1', u'|pCube1', u'pCube2', u'pCube3']
[]
[]
[]
```

Instructif, mais ne serait-il pas plus intéressant de récupérer l'ensemble sous la forme d'une seule et même liste ? Ça tombe bien, la commande `ls()` permet une légèreté syntaxique pour le faire en un seul appel. Vous pouvez en effet passer une liste d'expressions, un peu comme ça:

```
mc.ls(['*pSphere*', '*pCube*', '*pPlane*', '*pPyramid*',
    '*locator*'], type='transform')
```

? Mmmhhh… Si nous avions un moyen de générer une liste depuis `default_names` en ajoutant des étoiles devant et derrière chacun des noms, on pourrait passer cette nouvelle liste en argument de `ls()`.

Vous avez tout compris!

III.6.1.3. Second script

La méthode simple consiste à transformer nos noms en expressions avec des étoiles. En gros, transformer ça:

```
['pSphere', 'pCube', 'pPlane', 'pPyramid', 'locator']
```

En ça:

```
['*pSphere*', '*pCube*', '*pPlane*', '*pPyramid*', '*locator*']
```

Puis à passer cette liste à la commande `ls()`.

Pour cela, une simple boucle suffit:

```
name_exprs = []

for default_name in default_names:

    name_expr = '*'+default_name+'*'

    name_exprs.append(name_expr)

print mc.ls(name_exprs, type='transform')
```

Et bien entendu, je vous propose une explication ligne à ligne 🍊 :

```
name_exprs = []
```

Ici, on prépare la liste que notre boucle va remplir de nos noms sous forme d'expressions.

```
for default_name in default_names:
```

Le début de la boucle, on parcourt un à un tous les noms de notre liste de noms par défaut.

```
    name_expr = '*'+default_name+'*'
```

Comme précédemment, nous ajoutons les étoiles devant et derrière le nom pour en faire une expression.

```
    name_exprs.append(name_expr)
```

Nous ajoutons notre expression fraîchement construite à notre liste finale…

```
print mc.ls(name_exprs, type='transform')
```

…que nous utilisons enfin dans notre commande `ls()` qui va nous sortir tous les nœuds de notre scène correspondant à `*pSphere*`, ou `*pCube*`, ou `*pPlane*`, etc.

Sur ma petite scène faite de cubes et de sphères, cela donne:

```
[u'root|pSphere1', u'|pSphere1', u'pSphere2', u'pSphere3',
    u'root|pCube1', u'|pCube1', u'pCube2', u'pCube3']
```

Pas mal hein ?

III.6.1.4. Script du paresseux

La version pour les flemmards endurcis (les codeurs Python en fait) c'est:

```
mc.ls(['*'+n+'*' for n in default_names], type='transform')
```

Cette méthode utilise une *comprehension list* :

```
['*'+n+'*' for n in default_names]
```

C'est de l'ordre du langage, nous ne l'expliquerons donc pas ici. Je vous invite, en revanche, à creuser un peu le sujet (*list comprehension*, *dict comprehension*, etc.), il est possible que vous appreniez des choses intéressantes.

Vous pouvez lui ajouter un `select()` devant:

```
default_names = ['pSphere', 'pCube', 'pPlane', 'pPyramid',
    'locator']

mc.select(mc.ls(['*'+n+'*' for n in default_names],
    type='transform'))
```

Et bim ! En deux lignes vous sélectionnez tous les nœuds avec un problème de nom par défaut. Avouez que ça envoie du pâté auprès des collègues!

On va mettre ça sous forme de fonction qu'on peut mettre dans un coin et garder pour plus tard.

```
default_names = ['pSphere', 'pCube', 'pPlane', 'pPyramid',
    'locator']

def get_nodes_with_default_names():
    """Return a list of nodes in current scene having default name"""
    return mc.ls(['*'+n+'*' for n in default_names],
        type='transform')
```

III.6.2. Détecter les caractères non-ASCII

Nous allons scripter un moyen de détecter les nœuds ayant un caractère non-ASCII.

?

Debout, couché, *ascii* ?

On prononce *aski* bande de petits malins!

?

Je suppose que ça n'a rien à voir avec ce sport qu'on pratique d'ordinaire à la montagne en hiver ?

Vous avez de l'humour et c'est bien, ça prouve que vous êtes réveillé.

?

Donc *non-askii* c'est en *snowb…*

Le prochain pavé de code risque de vous êtres très désagréable…

Bon, prenons les choses dans l'ordre.

III.6.2.1. Unicode ni reproche

Maya, ainsi qu'énormément d'applications modernes, gèrent très bien la norme de codage de caractère Unicode , qui permet d'écrire des lettres dans toutes les langues (mais pas que), mais si cette dernière est très pratique pour les applications utilisant des caractères riches et, par conséquent, tournées vers l'expression littérale (e.g. le web), elle devient handicapante dans le cas où on cherche, justement, une uniformité et une concision.

? En quoi un moyen de gérer tous les caractères est-il gênant? 🍊

Saviez-vous qu'il existe plusieurs types d'espace en Unicode? Idem pour les traits-d'unions. Il suffit qu'un graphiste copie-colle un mot trouvé sur le net et deux mots *visuellement* identiques ne le seront en fait pas.

Autre exemple: Une fois au *lighting*, vous faites des expressions pour assigner des matériaux et des attributs. Si vos nœuds portent des noms «techniquement» différents, bien que «visuellement» identiques, vous allez passer plus de temps à faire correctement votre travail.

Limiter le nombre de caractères utilisables pour les noms des nœuds (et autres données de votre pipeline) simplifie ce qui passe dans les tuyaux.

En tant que francophone, nous pourrions arguer qu'utiliser une telle limitation de caractère revient à réciter du Corneille la langue greffée de clous rougis au fer. Nous n'aurions qu'à moitié tord. Mais force est de constater que se contraindre à l'utilisation d'un sous-ensemble des caractères de notre langue peut nous rendre de grands services technologiques et il peut être utile de détecter, et modifier, les caractères qui n'y appartiennent pas.

Et c'est ce que nous allons faire! 🤠

III.6.2.2. La norme ASCII

La norme ASCII est la plus simpliste des façons de stocker des caractères en informatique. C'est aussi une des plus vieilles (1960). Elle est composée de 128 caractères (7 bits) dont 95 *imprimables* (c-à-d, lisibles) :

```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~
```

Ils sont tous là ! 🍊

Comme vous pouvez le constater, pas d'accents, un seul type d'espace, pas de *œ*, etc.

III.6.2.3. Curiosité littérale

On va mettre ça en pratique. 🍊

Créez un nœud portant le nom `nœuds` et regardons ce qui se passe:

```
mc.createNode('transform', name='nœuds')
# Result: u'n\u0153uds' #
```

Déjà, `u'n\u0153uds'`, ça calme direct .

Un indice: Le `\u` indique que c'est de l'**U**nicode, puis `0153` parce que «œ» est le caractère Unicode numéro… 153. (Cherchez «caractère Unicode 153» sur internet pour vous en convaincre).

Toutefois, le nœud est correctement nommé dans Maya:

// TODO image

Et si vous tapez la commande en MEL:

```
createNode "transform" -name "nœuds";
// Result: nœuds //
```

Cette fois ci le nom du nœud apparaît correctement. Mais comment cela se fait-il ?

C'est la magie de l'Unicode qui est totalement supporté par le MEL!

Précédemment, je vous ai dis que l'Unicode était géré de manière native par la plupart des applications modernes. C'est le cas pour Maya et son MEL, mais dans le cas de Python 2.x, les chaînes de caractère natives sont en ASCII (comme en langage C). C'est d'ailleurs une des raisons qui fait que Python 2 et 3 ne sont pas (ou très partiellement) compatibles. Dans ce dernier, les chaînes de caractères sont nativement en Unicode.

Il va falloir virer tout ça!

III.6.2.4. La méthode

Il y a plusieurs méthodes pour détecter qu'une chaîne de caractères est valable en ASCII, mais je vais vous proposer la mienne et vous allez voir que c'est assez logique.

On va essayer d'encoder notre chaîne de caractère en ASCII et voir ce qui se passe:

```
>>> 'a'.encode('ascii')
'a'
```

Sans trop de surprise, avec un simple `a`, cela fonctionne, car cette lettre fait partie de la norme ASCII.

Qu'en est-il avec un caractère un peu plus élaboré:

```
>>> 'é'.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec
    can't decode byte 0xc3 in position 0: ordinal not in range(128)
```

Bam ! Pour faire simple, le message d'erreur indique que le caractère `0xc3` (notre `é`) ne peut pas s'encoder en ASCII, car son numéro de caractère, est au-dessus de 128.

III.6.2.4.1. Inspection du plantage

En Python, quand vous voyez un message tel que:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
BlahblahError: une erreur c'est pas bien blah blah
```

C'est que le code a planté quelque part.

Sauf qu'à défaut de faire planter l'application entière (ici, Maya), Python arrête immédiatement l'exécution du script et «crie» à qui veut l'entendre qu'on a eu un problème avec une erreur bien identifiée. Cette «erreur bien identifiée» s'appelle une «exception» (dans notre petit exemple, c'est `BlahblahError`).

Le fait de «crier» une exception porte un nom: «*Lever* une exception».

Comme le précise la documentation ( ), la méthode str.encode() lève une exception `UnicodeDecodeError` si l'encodage échoue.

Voir Python nous ~~cracher~~ lever son exception au visage ne nous aide pas (Venant d'un python c'est même plutôt dangereux), mais que diriez-vous si je vous disais que le principe d'une levée d'exception c'est de pouvoir «l'attraper» et la prendre en compte pour continuer l'exécution de son script?

?

Je dirais que je ne vois pas trop ou tu veux en venir...

Qu'a cela ne tienne, nous allons voir ça rapidement.

III.6.2.4.2. Appropriation du mécanisme

Voici un code qui attrape l'exception et improvise:

```
try:
    'é'.encode('ascii')  # on essait de faire ca
except UnicodeDecodeError:  # si la ligne precedente plante en levant l'exception UnicodeDecodeError
    print 'non ascii string!'  # on affiche le message 'non ascii string!'
```

Si vous exécutez ce bout de code, vous aurez bien évidement le message «non ascii string !».

D'un point de vue logique, nous avons donc un morceau de code qui tente quelque chose, et un autre qui réagit en cas d'échec. Ce mécanisme porte un nom: la «gestion d'exception», et comme vous pouvez le constater en lisant la page Wikipédia , ce n'est pas un *petit* concept… 🍍

Je ne vais pas m'étendre sur le sujet, vous avez compris le principe. On ne va pas vous demander de fabriquer vos propres exceptions. 🍍

On va se contenter de s'appuyer là-dessus et essayer d'encoder tous les noms des nœuds de notre scène et regarder s'ils génèrent une erreur.

III.6.2.5. On prend les mêmes et on recommence.

Voici enfin la fonction renvoyant tous les nœuds disposant d'au moins un caractère non-ASCII:

```
def non_ascii_named_nodes():

    result = []

    for node in mc.ls('*'):

        try:
            node.encode('ascii')
        except UnicodeDecodeError:
            result.append(node)

    return result
```

Avec ce que nous venons de voir plus haut, vous devriez être en mesure de comprendre tout seul, mais expliquons ça ligne à ligne. 🍍

```
    result = []
```

On crée une liste qu'on va remplir dans la boucle `for` qui suit et qui va stocker les nœuds problématiques.

```
    for node in mc.ls('*'):
```

On démarre la boucle qui va itérer sur chacun des nœuds de notre scène courante.

```
        try:
            node.encode('ascii')
```

On entre dans le bloc de gestion d'exception qui essaie de convertir le nom du nœud en ASCII…

```
        except UnicodeDecodeError:
            result.append(node)
```

Et la suite de la gestion d'exception qui «attrape» l'exception `UnicodeDecodeError` si elle se produit (indiquant un souci durant l'encodage en ASCII) et ajoute le nœud à la liste

```
    return result
```

Et enfin, on renvoie la liste remplie des nœuds problématiques.

III.6.2.5.1. La version puriste

Oui, oui, je sais, je chipote mais pour pleins de bonnes raisons, le fait de créer une liste, de la remplir n'est pas une méthode particulièrement recommandée. Il vaut mieux passer par un générateur:

```
def non_ascii_named_nodes():

    for node in mc.ls('*'):

        try:
            node.encode('ascii')
        except UnicodeDecodeError:
            yield node
```

Notez le `yield` à la dernière ligne.

Je ne vais pas vous expliquer ce que c'est car d'autres le font mieux que moi.

III.6.3. Avoir des noms de nœud simples et propres

Il peut être intéressant de limiter les caractères utilisables dans le nom de ses nœuds.

? Pourquoi se limiter?

Cela permet d'éviter de se poser trop de questions.

Si, au *lighting*, on se rend compte qu'un nœud ne passe pas simplement parce que le graphiste a fait une faute d'orthographe (e.g. un accent aigu au lieu d'un accent grave), on peut légitimement se demander ce qu'apporte une multitude de caractères.

On pourrait se limiter:

— aux caractères alphabétiques minuscules (`abcdefghijklmnopqrstuvwxyz`),
— aux caractères alphabétiques majuscules (`ABCDEFGHIJKLMNOPQRSTUVWXYZ`),
— aux nombres (`0123456789`),
— aux tirets bas (`_`).

On pourrait stocker tous ces caractères dans une variable:

```
valid_chars = 
    'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_'
```

Mais tant qu'à faire les choses biens, sachez que Python dispose déjà de quelques variables du genre dans le module `string`.

Faites:

```
import string
help(string)
```

Puis allez en bas. Vous aurez quelque chose comme:

```
DATA
    ascii_letters =
        'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    ascii_lowercase = 'abcdefghijklmnopqrstuvwxyz'
    ascii_uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    digits = '0123456789'
    hexdigits = '0123456789abcdefABCDEF'
    letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuv...
        \xcd\xce...
```

```
    lowercase = 'abcdefghijklmnopqrstuvwxyz\x83\x9a\x9c\x9e\xaa\xb5\xba\xd...
    octdigits = '01234567'
    printable = '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU...
    punctuation = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
    uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ\x8a\x8c\x8e\x9f\xc0\xc1\xc2\xc...
    whitespace = '\t\n\x0b\x0c\r '
```

En fait, le module string contient déjà des variables contenant les caractères de base. Et comme vous pouvez le constater, `ascii_letters` est en fait l'addition de `ascii_lowercase` et `ascii_uppercase`.

Ainsi, nous pouvons faire:

```
>>> string.ascii_letters + string.digits + '_'
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_'
```

? En quoi est-ce mieux de faire ça plutôt que d'avoir une variable contenant déjà toute la chaîne ? 🍊

Parce qu'un code est aussi une *intention*. 🍊

À la lecture du code…

```
>>> string.ascii_letters + string.digits + '_'
```

…peut se traduire par les lettres ASCII + les numéros + tiret bas. À l'inverse…

```
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_'
```

…nécessite que vous lisiez le contenu de la variable pour savoir s'il y a en effet *uniquement* les lettres, les numéros, et un tiret bas.

Nous avons donc notre belle variable `valid_chars` contenant tous les caractères valides que nos nœuds peuvent avoir.

```
import string

valid_chars = string.ascii_letters + string.digits + '_'
```

Il ne manque plus qu'à vérifier que les caractères qui composent le nom des nœuds sont tous présent dans cette variable.

III.6.3.1. La vérification sur un nœud

Nous allons créer une fonction qui va nous indiquer si le nom qu'on lui passe est valide ou non.

Voici le bout de code dont nous allons discuter:

```
import string

valid_chars = string.ascii_letters + string.digits + '_'

def is_valid(node_name):
    for char in node_name:
        if char not in valid_chars:
            return False
    return True
```

La première partie:

```
import string

valid_chars = string.ascii_letters + string.digits + '_'
```

...vous la connaissez déjà.

Passons à la suite.

```
def is_valid(node_name):
```

On définit une fonction `is_valid()` qui prend un nom de nœud en argument et qui renverra `True` ou `False` suivant que le nom du nœud passe en argument est valide ou non.

```
    for char in node_name:
```

C'est parti ! Ici, on rentre dans une boucle qui va itérer sur chacun des caractères de notre nœud (`'p'`, `'S'`, `'p'`, `'h'`, etc.).

```
        if char not in valid_chars:
```

Si le caractère n'est pas dans le paquet des caractères valides...

```
        return False
```

...on renvoie `False`.

```
    return True
```

Si on arrive à la fin de la boucle, c'est que le nom est valide, on renvoie donc `True`.

Vous pouvez essayer:

```
>>> print is_valid('pSphere')
True
>>> print is_valid('pSphère')
False
```

Comme vous pouvez le constater, un nom invalide renvoie `False`.

III.6.3.2. La même chose, mais sur plusieurs nœuds

C'est bien joli mais l'objectif de tout ça c'est quand même de vérifier les noms des nœuds de toute une scène pas vrai ?

Vous allez voir que c'est beaucoup plus facile maintenant que nous avons notre petite fonction `is_valid()`.

```
for node_path in mc.ls('*', recursive=True, long=True):

    node_name = node_path.split('|')[-1]

    node_name = node_name.split(':')[-1]

    if not is_valid(node_name):

        print "Invalid name detected", node_name
```

Vous avez sûrement remarqué quelques bricolages.

Mais trêve de spéculations, sautons les deux pieds joints dans tout ça:

```
for node_path in mc.ls('*', recursive=True, long=True):
```

Ici, nous allons itérer à travers tous les nœuds de notre scène.

Pour rappel, l'argument `recursive` s'assure que l'expression (ici, `'*'`) s'applique à l'intérieur de tous les *namespaces* de notre scène.

L'argument `long` donne le chemin complet vers un nœud. Or, nous ne souhaitons vérifier que son nom. Pourquoi, dans ce cas, utiliser le chemin complet ? 🍊

Bien vu ! 🍊

La réponse est toute simple:

Une fois nos nœuds détectés, il faut bien que nous disposions d'un chemin complet pour pouvoir l'afficher. Si notre boucle affiche:

```
Invalid name detected pSphère
```

Vous serez bien embêté pour le retrouver dans votre scène pas vrai ?

```
    node_name = node_path.split('|')[-1]
```

Ici, on utilise une petite technique pour s'assurer que justement, nous récupérons le nom du nœud.

Prenons un exemple:

```
>>> node_path = '|root|toto|head'
```

La première chose est de couper la chaîne de caractères au niveau du `|`, comme ceci :

```
>>> node_path.split('|')
['root', 'toto', 'head']
```

Puis nous récupérons le dernier élément de la liste via l'index `[-1]` qui, comme vous l'aurez deviné, prend le premier élément en partant de la fin de la liste.

```
>>> node_path.split('|')[-1]
'head'
```

On a bien récupéré le nom du nœud !

Continuons…

```
    node_name = node_name.split(':')[-1]
```

? Encore ?

Et non !

Regardez bien, cette fois ci c'est sur `:` que nous coupons.

? Pourquoi ?

À cause des *namespaces* Maya pardi ! 🍊

Si votre nœud est mis dans un *namespace* et se nomme `'toto_001:head'`, il faut bien couper au niveau du *namespace* et ne conserver que le nom (ici, `'head'`).

C'est donc le même mécanisme que pour le chemin, mais on coupe sur `:` au lieu de `|`.

```
    if not is_valid(node_name):
```

Enfin! Notre petite fonction !

Je ne vais pas vous détailler son fonctionnement, nous l'avons déjà fait précédemment. 🍊

La condition stipule que si le nom du nœud n'est pas valide, on entre dans le bloc de condition…

```
        print "Invalid name detected", node_name
```

…qui ne fait qu'afficher le nom du nœud invalide.

Vous pouvez mettre toute ça dans une fonction et vous obtenez:

```
import string

```

```
valid_chars = string.ascii_letters + string.digits + '_'

def is_valid(node_name):
    for char in node_name:
        if char not in valid_chars:
            return False
    return True

def report_invalid_nodes():

    for node_path in mc.ls('*', recursive=True, long=True):

        node_name = node_path.split('|')[-1]

        node_name = node_name.split(':')[-1]

        if not is_valid(node_name):

            print "Invalid name detected", node_name
```

Et voila, vous avez une jolie fonction qui vous liste tous les nœuds qui pourront vous sauter au visage plus tard.

III.6.3.3. Optimisation et concision

Je me suis retenu pour ne pas vous malmener, mais j'aimerais pousser un peu certains concepts. 

III.6.3.3.1. Utiliser un set plutôt qu'une chaîne de caractère

Ce type de condition peut être très coûteux:

```
if truc in plein_de_trucs:
```

En effet, on va s'assurer que la variable `truc` est présente dans le paquet `plein_de_trucs`. On va donc comparer avec `truc`, chacun des éléments de `plein_de_trucs`, un à un.

Le morceau de code qui nous concerne est dans la fonction `is_valid()`:

```
if char not in valid_chars:
```

Je ne vais pas commencer un cours sur l'algorithmique mais, pour faire simple, plus `plein_de_trucs` (ou, dans notre cas, `valid_chars`) contient d'éléments, plus c'est lent.

Python est assez efficace de base, mais c'est une bonne pratique d'utiliser un `set()` pour stocker des éléments qu'on souhaite comparer régulièrement.

Ainsi, la variable `valid_chars`, originellement définie sous la forme:

```
valid_chars = string.ascii_letters + string.digits + '_'
```

Devient :

```
valid_chars = set(string.ascii_letters + string.digits + '_')
```

Sur mes tests, j'obtiens un joli 25% de vitesse en plus.

III.6.3.3.2. Les générateurs

Autre concision possible, en utilisant des générateurs .

Ces deux fonctions font exactement la même chose:

```
def is_valid(node_name):
    for char in node_name:
        if char not in valid_chars:
            return False
    return True
```

```
def is_valid(node_name):
    return all((c in valid_chars for c in node_name))
```

La première est écrite de manière traditionnelle, la seconde utilise un générateur, plus rapide et concis, combine à `all()` qui renvoie `True` si l'intégralité des éléments du générateur (ici, `c in valid_chars`) sont à `True`.

III.6.4. rename() pour renommer vos nœuds

?

C'est pas que ce tuto est un peu moisi du slip, mais là comme ça je dirais qu'on a toujours rien renommé...

Je sais, je sais, mais avouez qu'avant de renommer quelque chose, il était plus intéressant de savoir quoi.

Ici, on attaque la commande `rename()` (la documentation est ici). Vous allez voir qu'elle n'est pas compliquée.

III.6.4.1. Renommer un nœud

Comme vous vous en doutez, cette commande renomme le nœud que vous lui donnez:

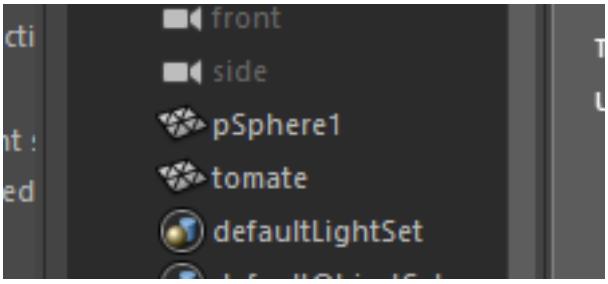
```
1 mc.rename('noeud_a_renommer', 'nouveau nom')
```

Avouez que c'est simple.

Vous ne pouvez pas donner un nom vide, sinon Maya vous enverra bouler.

Notez que si le nouveau nom entre en conflit avec un nom existant, le nœud se verra attribué un nom unique en rapport avec le nom que vous souhaitiez lui donner (souvent, Maya ajoutera un chiffre derrière).

Partons de l'exemple:



Si vous renommez `pSphere1` en `tomate` comme ici:

```
1 mc.rename('pSphere1', 'tomate')
```

Vous obtenez:

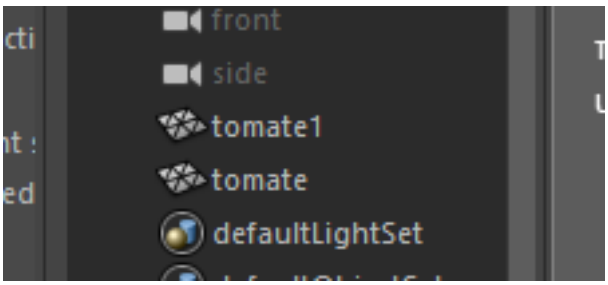


FIGURE III.6.1. – Remarquez comment le nœud a pris le nom `tomate1`.

D'une certaine façon, il est impossible d'être sûr que le nom que vous souhaitez donner est réellement celui que Maya va assigner au nœud.

?

Mais comment je peux être sûr que le nom créé par Maya est bien celui que je lui ai donne?

Les développeurs de Maya ont pensé à vous!

La fonction `rename()` renvoi le nom réel, tel que Maya l'a assigné:

```
1 vrai_nom = mc.rename('noeud_a_renommer', 'nouveau nom')
```

Dans notre exemple précédant, cela aurait donné:

```
1 real_name = mc.rename('pSphere1', 'tomate')
2 print real_name  # 'tomate1'
```

Vous pouvez également ajouter un dièse `#` au nom pour indiquer à Maya que vous souhaitez ajouter un numéro:

```
1 print mc.rename('pSphere1', 'cerise#')
2 # 'cerise1'
```

III.6.4.2. Renommer la sélection

Sachez aussi que vous pouvez renommer le nœud sélectionné en ne passant qu'un seul argument à la fonction `rename()`:

```
1 mc.select('pSphere1')
2 mc.rename('tomate')
```

Je ne suis pas un fervent adepte de cette méthode. Beaucoup de commandes permettent d'agir selon la sélection mais cette approche peut rendre le code difficile à suivre, car il n'est jamais explicitement écrit ce qu'on cherche à faire.

Mais vous êtes grand, faites ce qui vous semble le plus clair.

III.6.4.3. `ignoreShape` pour ne pas renommer la shape

Si vous utilisez Maya depuis un moment, vous aurez sûrement remarqué qu'il a tendance à essayer de renommer les nœuds de *shape* quand vous renommez le *transform*.

Prenons le nœud `pSphere1` :

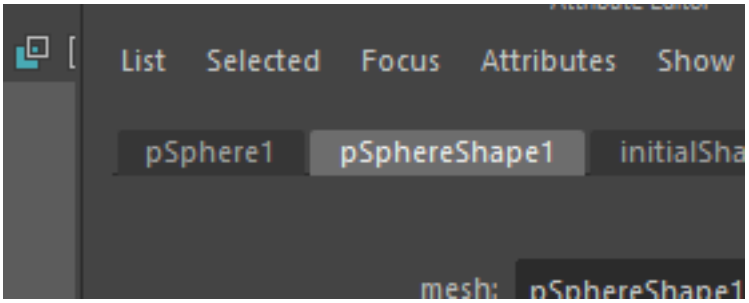
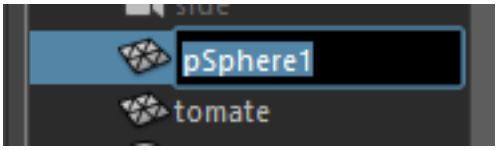
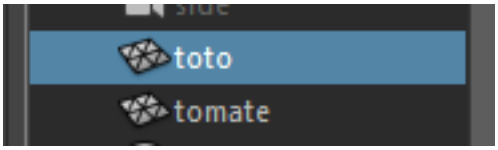


FIGURE III.6.2. – Remarquez comment le nom de du nœud de *transform* (onglet de gauche) et le nœud de *shape* (onglet sélectionné) correspondent.

Renommez le nœud en `toto`:





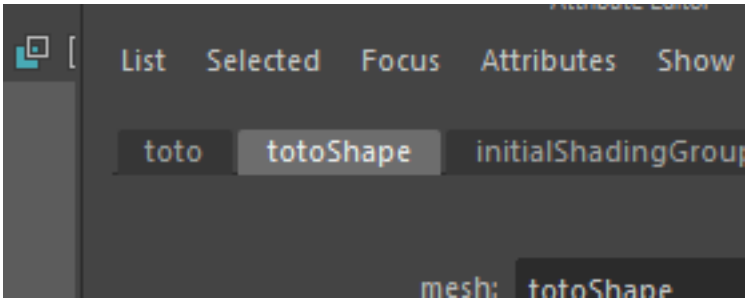


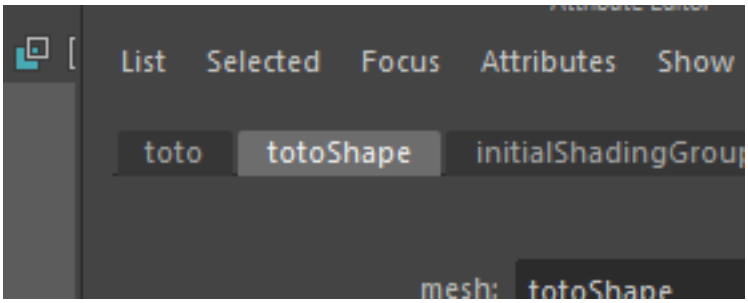
FIGURE III.6.3. – Et observez comment le nœud de *transform* **et** de *shape* ont été renommés.

Par défaut, la commande `rename()` fonctionne également de cette façon.

Si vous annulez et que vous exécutez:

```
1 mc.rename('pSphere1', 'toto')
```

Vous remarquerez le même résultat:



Pourtant, la valeur renvoyée par la commande n'est *que* `toto`.

Donc Maya renomme encore des choses sans me le dire?

Hé oui , mais comme vous l'aurez deviné, vous pouvez désactiver ce mécanisme via l'argument `ignoreShape`.

Annulez une dernière fois puis exécutez:

```
1 mc.rename('pSphere1', 'toto', ignoreShape=True)
```

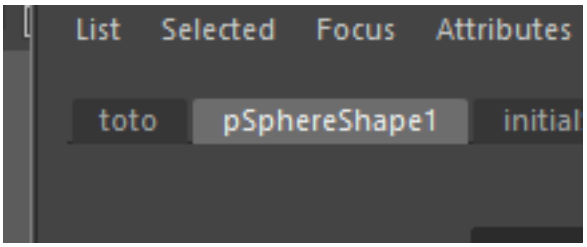


FIGURE III.6.4. – Le nœud de `shape` n'est pas renommé

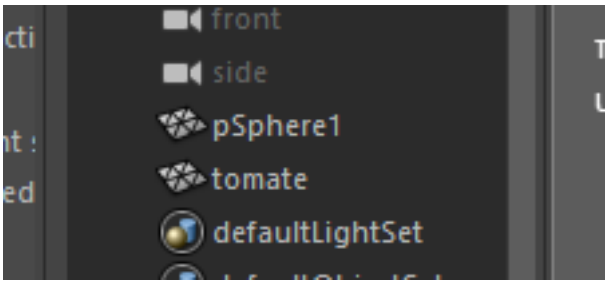
Haha, il fait moins le malin!

III.6.4.4. Créer et modifier des *namespaces*

La commande `rename()` permet également de créer des *namespaces* et de déplacer des nœuds d'un *namespace* à l'autre, simplement en le renommant.

Si vous avez l'habitude de manipuler Maya, vous devriez savoir à quel point les *namespaces* peuvent être casse-pied. Je vous montre cette méthode pour que vous sachiez qu'elle existe. Mais il est peu probable que vous l'utilisiez souvent.

Reprenons notre exemple de départ :



Exécutez:

```
1 mc.rename('pSphere1', ':toto:tomate')
```

Notez qu'on ajoute les deux-points `:` en début de nom pour écrire le *namespace* absolu. Si on ne le fait pas, Maya lève une exception et refuse de crever le *namespace* :

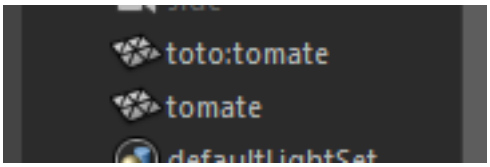
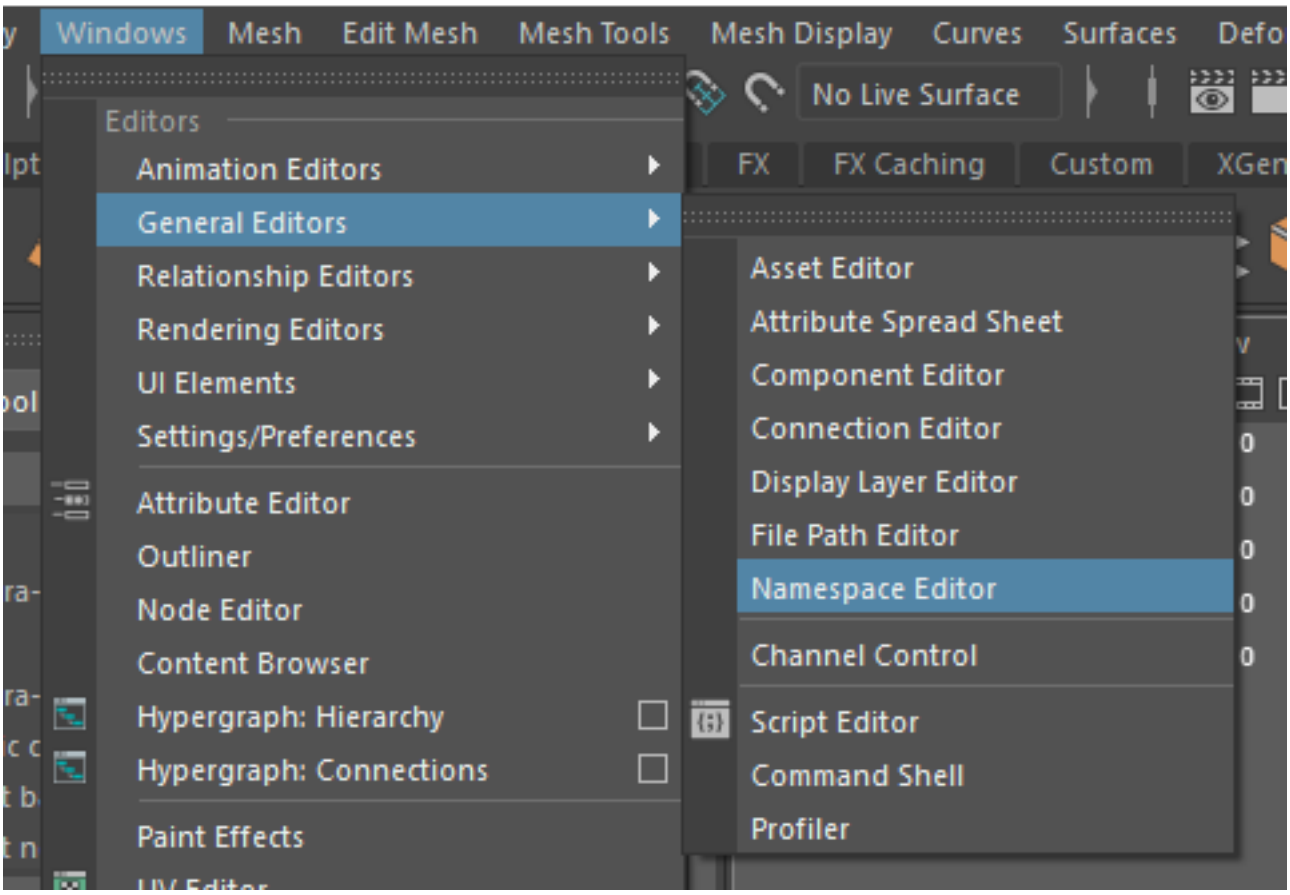


FIGURE III.6.5. – *Tadaaa!*

? Maya a mis le nœud dans un *namespace*! 🍻

Exactement! 🍊

Ouvrez votre *Namespace Editor* et admirez 🍊 :


Windows
Mesh
Edit Mesh
Mesh Tools
Mesh Display
Curves
Surfaces
Defo
No Live Surface
FX
FX Caching
Custom
XGen
Editors
Animation Editors
General Editors
Relationship Editors
Rendering Editors
UI Elements
Settings/Preferences
Attribute Editor
Outliner
Node Editor
Content Browser
Hypergraph: Hierarchy
Hypergraph: Connections
Paint Effects
Asset Editor
Attribute Spread Sheet
Component Editor
Connection Editor
Display Layer Editor
File Path Editor
Namespace Editor
Channel Control
Script Editor
Command Shell
Profiler

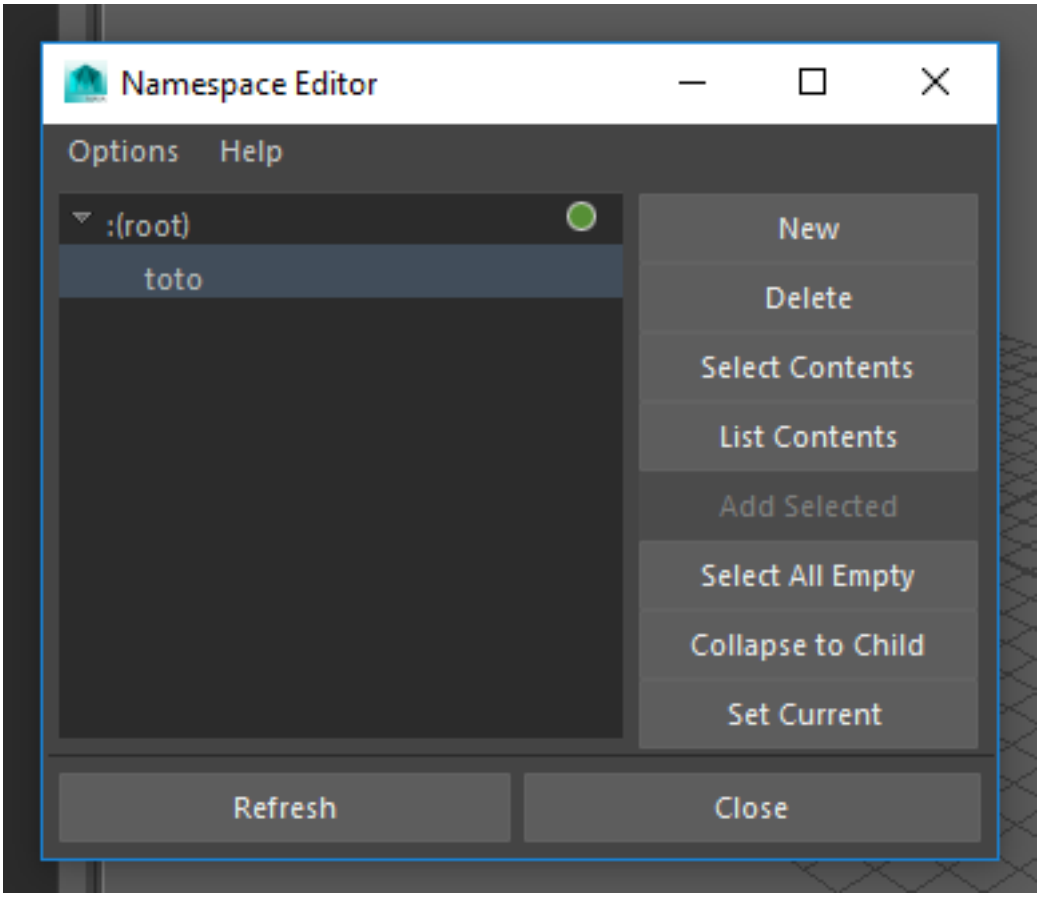

Namespace Editor
Options
Help
:(root)
toto
New
Delete
Select Contents
List Contents
Add Selected
Select All Empty
Collapse to Child
Set Current
Refresh
Close

FIGURE III.6.6. – Un *namespace* tout beau tout frais!

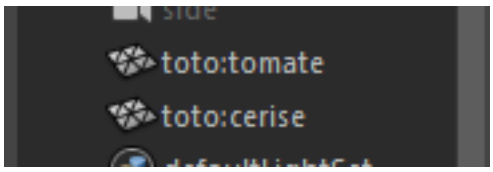
Et c'est pas fini, exécutez:

```
1 mc.rename('tomate', 'toto:cerise')
```

?

Tu as oublié de mettre les deux-points `:`! 🍊

Ah bon?



?

Mais, Maya n'a pas planté? 🦉

Eh non! 🍊 C'est une des bizarreries de Maya. Accrochez-vous:

— quand vous mettez un deux-points `:` devant le *namespace*, vous le passez en mode **absolu** et Maya va le créer;
— quand vous ne mettez pas les deux-points `:` devant le *namespace*, vous le passez en mode **relatif** et Maya va:
— l'ignorer s'il n'existe pas,
— l'utiliser s'il existe.

?

🍊

Ne vous inquiétez pas, comme je vous le disais, je voulais principalement vous faire savoir que vous pouviez créer et changer les nœuds de *namespace* via la commande `rename()`. Il est peu probable que vous n'ayez jamais à le faire donc pas de panique! 🍊

III.6.5. Une ch'tite boucle pour renommer des ch'tits nœuds...

Comme le nom de cette section le laisse présager, je vais vous présenter une petite boucle pour renommer votre sélection. 🍊

Nous allons renommer par rembourrage.

On va peut-être rester poli.

Vous n'en êtes peut-être pas familier, mais c'est le bon terme. Le terme qu'on utilise le plus souvent dans les logiciels 3D est son homologue anglais, *padding*, mais pour des raisons évidentes de ***COCORICO !!!***, eh bien nous allons utiliser le terme rembourrage dans les explications.

Le principe est simple. Nous voulons renommer une sélection avec un modèle (*pattern* en anglais) qui ressemble à `<nom du noeud><trois chiffres>`. Un peu comme `jointure_porte001`, `jointure_porte002`.

Naïvement, on ferait ça:

```
i = 1
name = 'jointure_porte'

# parcours les noeuds de la selection
for node in mc.ls(selection=True, long=True):

    # genere un nouveau nom
    # 'jointure_porte' + '003'
    new_name = name + str(i).zfill(3)

    # et renomme le noeud avec son nouveau nom
    mc.rename(node, new_name)

    # et on incremente la valeur pour passer au suivant
    i += 1
```

Pas vrai?

La seule information qui peut vous manquer pour comprendre cette boucle c'est la méthode `str.zfill()` qui, comme son nom l'indique, remplit la `string` de zéros, je vous renvoie vers la documentation pour plus d'informations.

Ça sent le piège.

Eh bien testons.

Faites une chaîne de groupe (martelez Ctrl+g sur votre clavier), ouvrez la hiérarchie puis sélectionnez aléatoirement (c'est important) des nœuds de la hiérarchie:

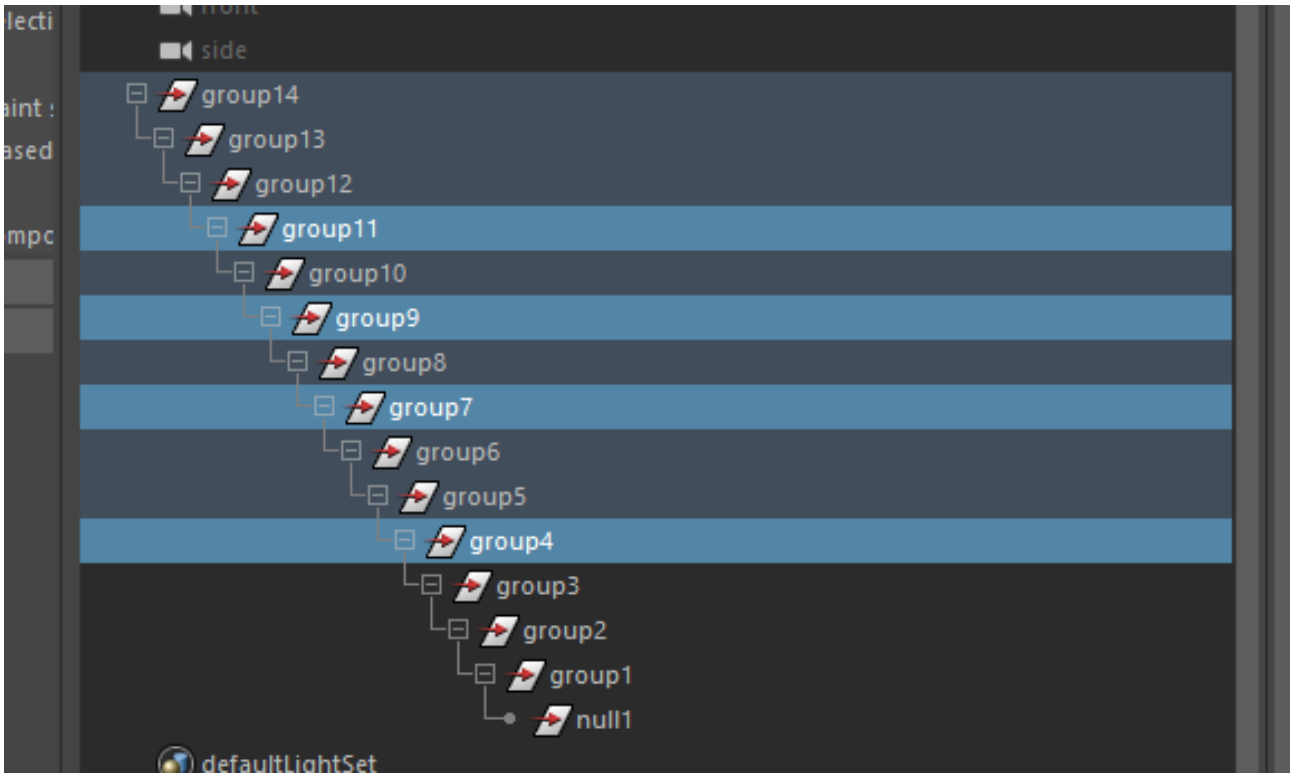
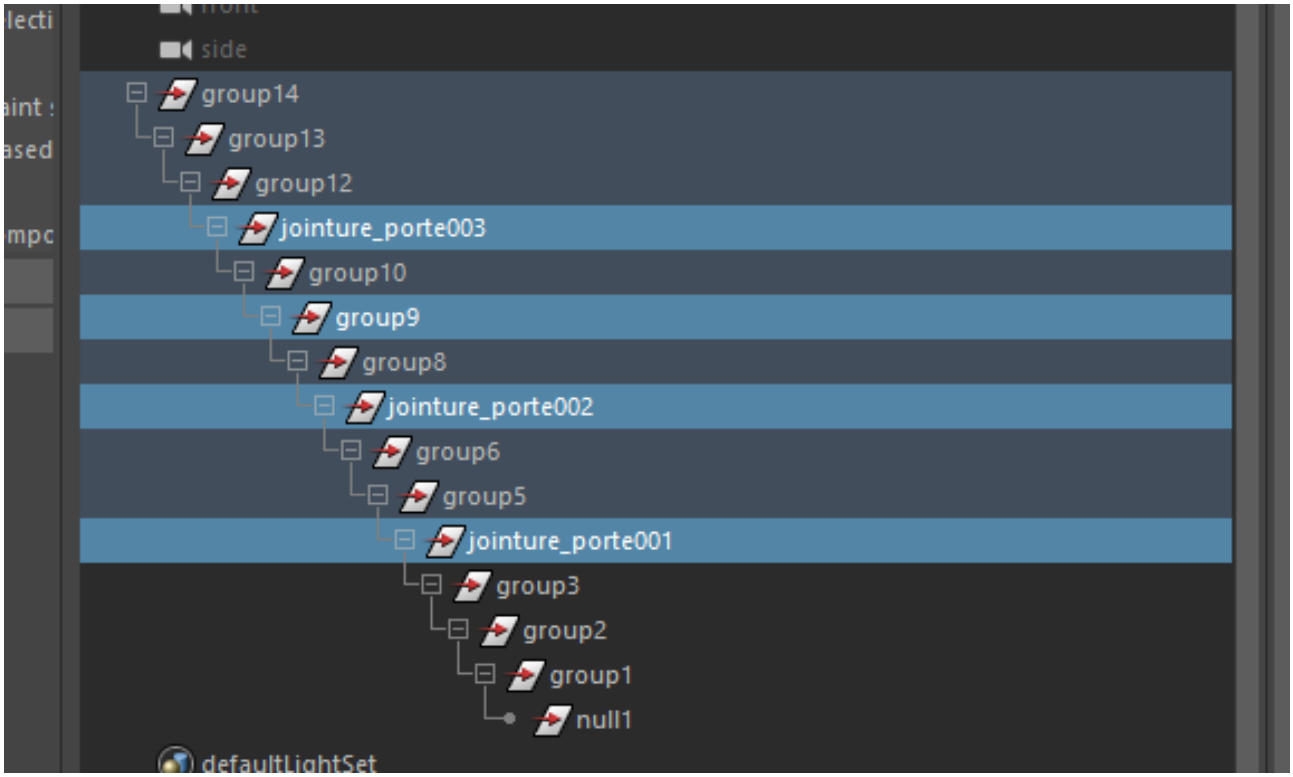


FIGURE III.6.7. – Ouh là que ce n'est pas bien nommé tout ça. *Retake!*

Puis exécutez le code!



Si vous avez sélectionné de manière aléatoire, vous remarquerez que certains nœuds n'ont pas été renommés.

Mais si vous êtes bien vigilant, vous remarquerez que Maya a planté:

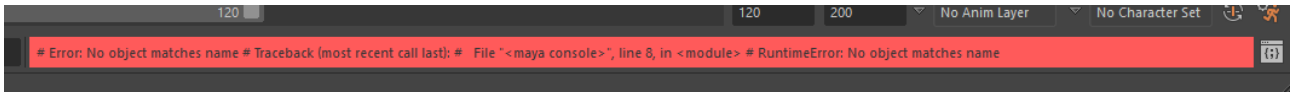


FIGURE III.6.8. – Pas bon...

Ouvrons le *Script Editor* et regardons ça de plus près:

```
1 # Error: No object matches name
2 # Traceback (most recent call last):
3 #   File "<maya console>", line 12, in <module>
4 # RuntimeError: No object matches name #
```

Ligne 12, c'est la ligne qui appelle la commande `rename()`, et le message semble indiquer qu'il ne trouve pas le nœud Maya à renommer.

? Comment ça ce fait? 🍊

C'est à vous de réfléchir un peu! 🍊

Rappelez-vous, j'ai insisté pour sélectionner de manière aléatoire les nœuds du groupe. L'idée était de s'assurer que l'ordre des nœuds n'était pas connu à l'avance.

Si votre sélection impose que la commande `ls(selection=True, long=True)` renvoie quelque chose comme ça:

```
1 ['|group1|enfant1',
2  '|group1',
3  '|group1|enfant2']
```

À la première itération, c'est `|group1|enfant1` qui va être renommé, donc pas de problème. À la seconde, on renomme `|group1`, une fois encore pas de problèmes.

Mais qu'arrive-t-il quand la commande va chercher à renommer `|group1|enfant2`? Hé bien il ne va pas réussir à y accéder, car on a renommé `|group1|` à l'itération précédente.

Le nœud `|group1` n'existe plus! 🍊

Et c'est ce qui c'est la raison pour laquelle on ne peut pas prendre bêtement l'ordre de la sélection.

? Mais comment peut-on faire? 🍊

Comme souvent en programmation, il y a moultes façons d'arriver à un résultat. Je vais vous proposer ma solution qui, cela va de soi, est la meilleure de toute! 🍊

Remarquez comment il suffirait de trier les nœuds renvoyés par `ls()` pour commencer par le bas et remonter. Dans notre micro-exemple, la liste deviendrait:

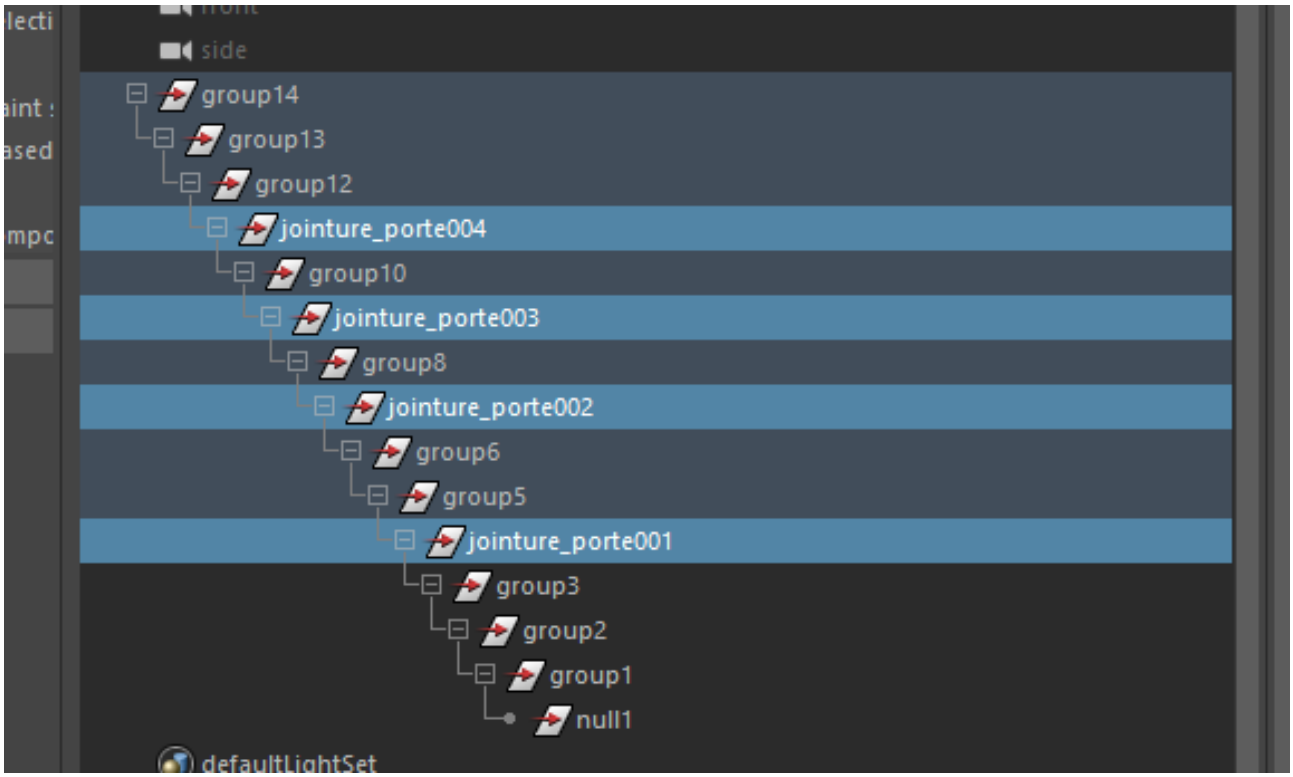
```
['|group1|enfant2',
 '|group1|enfant1',
 '|group1']
```

? Ha! Je me rappelle mes super tutos Python et la fonction `sorted()` (plus d'informations ici). 🍊

C'est ça! On va utiliser cette méthode avec son argument `reverse` pour… inverser le tri. 🍊

```python
i = 1
name = 'jointure_porte'

for node in sorted(mc.ls(selection=True, long=True), reverse=True):

    new_name = name + str(i).zfill(3)

    mc.rename(node, new_name)

    i += 1
```

Et le résultat:



On pourrait s'arrêter là…

? Mouais… Regarde, comme il a commencé par le bas, les noms sont numérotés du bas vers le haut, c'est tout nul.

Notez que j'ai dit on *pourrait*…

Ce qui me permet d'introduire le dernier exercice!

? Ha! Mais je disais rien moi.

Que nenni! Vous allez le faire cet exercice!

Donc, on aimerait que la numérotation se fasse de bas en haut, mais la hiérarchie des nœuds nous impose de renommer de bas en haut… Comment se dépêtrer de tout ça?

Je vais vous donner un indice: Il faut procéder en deux temps.

Allez! Maintenant à vous de jouer! Cherchez un peu et n'ouvrez la solution que quand vous avez fait quelques essais. Ce n'est que comme ça que vous assimilerez les concepts.

Contenu masqué n°3

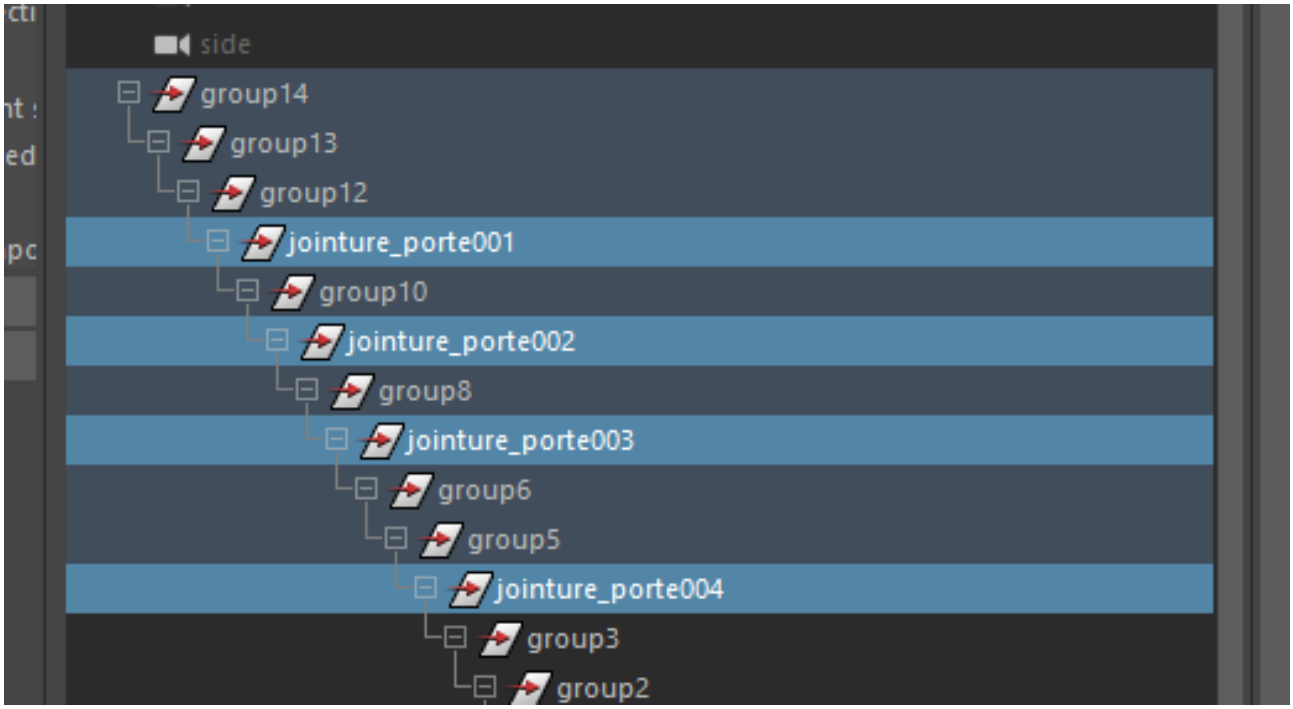


FIGURE III.6.9. – Renommé suivant la hiérarchie.

Je vous explique le code.

```
1 nodes = sorted(mc.ls(selection=True, long=True), reverse=True)
```

Dans la mesure où la boucle de renommage va s'exécuter deux fois, cette ligne stocke les nœuds à renommer, du bas vers le haut, tel qu'on l'a vu précédemment.

```
1 node_names = []
```

Ceci est la liste que nous allons remplir avec les noms générés, en vue de les inverser plus tard.

```
1 for node in nodes:
2
3     new_name = name + str(i).zfill(3)
4
5     node_names.append(new_name)
6
7     i += 1
```

Je mets la boucle en entier, j'suis un ouf!

Bon, le `for in` itère à travers les nœuds de la sélection inversée, générés précédemment.

Remarquez l'autre différence avec la boucle du premier exemple: on n'utilise plus directement `rename()` mais on stocke le `new_name` dans la liste `node_names`.

Arrivé à ce stade, on a deux listes:

— `nodes` contenant les nœuds à renommer, dans le bon ordre.
— `node_names` contenant les noms des nœuds, mais dans le mauvais ordre.

Il nous reste plus qu'à inverser la dernière liste pour s'assurer que les noms sont assignés du plus gros chiffre (en bas de la hiérarchie) au plus petit chiffre (vers le haut). C'est ce que nous faisons en utilisant la méthode `list.reverse()` (documentation ici).

```
1 node_names.reverse()
```

C'est peut-être à partir de là que votre code commence à se différencier du mien :

```
1 for node, node_name in zip(nodes, node_names):
```

Si vous ne comprenez pas cette ligne, vous avez sûrement raté une partie de votre tutoriel Python concernant les boucles.

La commande `zip()` (documentation ici permet d'itérer sur deux listes en même temps, en vous donnant chacun des éléments à la queue leu-leu: `['A', 'B', 'C', 'D']` et `[1, 2, 3, 4]` deviennent `[('A', 1), ('B', 2),` etc.

Les listes `nodes` et `nodes_names` faisant logiquement la même taille, la première contient les nœuds *sources*, à renommer, la seconde comprenant les noms à assigner.

On va *vraiment* s'arrêter là cette fois.

Mettez tout ça dans une fonction que vous pourrez utiliser quand bon vous semblera ou dans une petite interface.

Conclusion

Et voilà!

Vous avez vu que ce n'est pas si compliqué que ça.

Il y a pleins de nuances, de cas particuliers à gérer. Renommer des nœuds n'est pas une tâche dénuée de complexité, mais c'est un exercice motivant, car vous pouvez le pratiquer dans votre travail quotidien, et intéressant, car il permet d'apprendre à manipuler pas mal de concepts liés aux boucles.

Je vais vous laisser quelques pistes pour faire évoluer vos boucles de renommage:

— Creusez un peu le fonctionnement de la commande `ls()` histoire de ne pas vous en tenir qu'à la sélection.
— Utilisez `str.startswith()` et `str.endswith()` pour améliorer votre méthode de recherche de nœuds mal nommés.
— Faites une boucle *chercher et remplacer* qui cherche un morceau du nom d'un nœud pour le modifier. Aidez-vous de la condition `'word' in name` qui permet de trouver un mot dans une chaîne de caractères, puis utilisez `str.replace()` pour remplacer ce mot.
— N'oubliez pas `str.format()` pour les noms un peu tarabiscotés.
— Ras-le-bol de renommer? Faites une petite interface pour présenter tout ça!
— etc.

Faites ce que vous voulez, mais prenez le contrôle sur votre travail et surtout, amusez-vous!

Conclusion

Et voilà! Vous y êtes arrivé! J'espère que vous avez appris pas mal de choses, y compris sur les dessous du fonctionnement de Maya.

Le prochain chapitre aborde toujours plus de commandes, toujours plus de moyens d'automatiser votre travail! Que du bonheur!

Conclusion

Ce tutoriel est loin d'être fini. Je ne mettrai donc pas de conclusion.

Où que vous soyez arrivés, j'espère que ce tutoriel vous aura donné goût au *scripting* et l'envie d'aller plus loin, de comprendre un peu plus ce qui ce passe dans Maya, de prendre un peu plus le contrôle des choses.

Amusez-vous et surtout, prenez soin de vous! 🍊

Contenu masqué

Contenu masqué n°3

Ayez? Vous avez ~~trouvé~~ cherché? 🍊 Voici le code:

```
i = 1
name = 'jointure_porte'
nodes = sorted(mc.ls(selection=True, long=True), reverse=True)
node_names = []

for node in nodes:

    new_name = name + str(i).zfill(3)

    node_names.append(new_name)

    i += 1

node_names.reverse()

for node, node_name in zip(nodes, node_names):

    mc.rename(node, node_name)
```

Retourner au texte.

Liste des abréviations

DG Dependency Graph. 13, 14, 18, 19