Introduction au script MEL sous Maya

lionel.reveret@inria.fr

2017-18

Rappel sur l'architecture interne de Maya

Avant toute considération sur le scripting, on rappelle que l'architecture interne de Maya est construite autour d'un graphe de nœuds connectés (nodes). Deux types de graphes coexistent : le "Directed Acyclic Graph (DAG)" et le "Dependency Graph (DG)". Le DAG correspond au graphe de scène : il lie les hiérarchies entre les objets 3D, les repères et les transformations géométriques. Chaque objet 3D comporte un nœud désignant sa forme (nœud de type mesh, NURBS, etc) et sa position dans la scène via un nœud de type Transform. Le DG correspond à un flot de calcul aboutissant à la génération d'objets. Le DG est composé de nœuds désignant un calcul ou un objet (géométrie, shaders, etc). Chaque nœud comporte des paramètres d'entrées et des paramètres de sortie.

Tous les nœuds peuvent être connectés entre eux, à condition que les entrées et sortie soient compatibles, c'est-à-dire de même type. On peut visualiser le DAG via la fenêtre Hypergraph (mode hiérarchie) ou la fenêtre Outliner. On peut visualiser le DG via la fenêtre Hypergraph (mode connections) ou la fenêtre Node Editor. Lorsqu'une entrée d'un nœud du DG est modifiée, le calcul est propagé le long de tous les nœuds en sortie. Un plug-in peut être, entre autre, le développement d'un nœud.

Les deux exemples suivant illustrent les caractéristiques et différences entre DG et DAG.

Note : LMB et RMB désignent un clic sur le bouton gauche de la souris, respectivement droit (Left Mouse Button, Right Mouse Button).

Ex1. Duplication d'objet

1. créer un cube polygonal simple, observer depuis la fenêtre Hypergraph la liaison entre le node polyCube et le mesh généré.

2. dupliquer (edit>duplicate) le cube en simple copie : seul le mesh résultat est dupliqué

3. dupliquer le cube en copiant les "input connections" : les deux mesh partagent un même nœud polyCube

4. dupliquer le cube en copiant "l'input graph" : deux nouveaux meshes et deux polyCube sont créés.

Ex2. Skinning par DAG et par DG

1. créer deux cylindres polygonaux comme métaphore d'un bras et d'un avant-bras.

2. créer une chaîne articulée (joint) avec trois articulations pour l'épaule, le coude et le poignet

3. parenter (edit>parent) le cylindre du bras à l'épaule, et celui de l'avant-bras au coude. Afficher les repères locaux des cylindres (Display>Transform Display>Local Rotation Axis). On voit que la liaison squelette/maillage se fait par le DAG (à voir aussi via les fenêtres Hypergraph ou Ouliner)

4. créer un autre cylindre unique de taille double pour tout le bras et une chaîne articulaire identique à la précédente

5. Lier le squelette et le cylindre par "smooth skin" (Contexte de menu : Rigging, Menu : Skin>Bind Skin)

6. Il n'y a plus de repère local pour la partie avant-bras. La liaison se fait par le DG.

Ex3. Identifier un type de Nœud

La documentation sur "Nodes and Attributes" donne toutes les caractéristiques des nœuds internes: elle est au coeur de Maya et est déjà en jeu avant même une programmation en script ou API. Il ne faut donc pas la confondre avec la documentation des langages script MEL/Python, ni celle de l'API. Que l'on crée une scène par l'interface graphique, une commande MEL ou un plug-in, tout repose en interne sur ces nœuds du DG. Identifier ces sources dans le doc en ligne obtenue à partir de la touche F1.

Caractéristiques générales du script MEL

Le script MEL est un langage interprété, c'est-à-dire qu'il n'y a pas de compilation. Historiquement, il correspond à un langage propriétaire de Maya dont la syntaxe est un mélange de C et de Perl. MEL signifie Maya Embeding Language. Une syntaxe Python a été introduite depuis la version 2011 afin de permettre une programmation plus riche et plus souple. L'ensemble des commandes MEL a été traduit en un module Python (maya.cmds), en conservant la même lexicographie. On verra plus loin les détails de cette équivalence. Il reste donc deux éditeurs de script sous Maya : un en syntaxe MEL historique, l'autre en syntaxe Python via le module maya.cmds. Vis-à-vis des fonctionnalités de Maya atteignables par ligne de commande script, les deux syntaxes sont identiques. La syntaxe MEL a été conservée en raison du très grand nombre de scripts déjà développés dans la communauté, à commencer par les fichiers de configuration de Maya. Les exemples suivants de ce document sont présentés en syntaxe MEL, ils sont bien sûr traduisible en syntaxe Python. Par la suite, on parlera de script et commandes MEL pour désigner de la même manière les fonctionnalités avec la syntaxe MEL ou la syntaxe Python du module maya.cmds. On parlera explicitement de syntaxes lorsqu'il sera nécessaire de différencier les deux.

Le script MEL a trois utilisations principales :

- construire des objets de tous les types, les lister, lire leurs attributs et les modifier;

- implémenter des algorithmes. En syntaxe MEL, on dispose de variables, de types, de flots de contrôle et de procédures. La syntaxe est simplifiée comme peut l'être PERL par exemple : pas besoin de déclarations de variable, extension automatique des tableaux. L'objectif est la sécurité : il n'y a pas de pointeurs. En syntaxe Python, on retrouve tous les avantages du typage dynamique fort, auquel s'ajoute la programmation objet qui n'est pas présente en syntaxe MEL.

- créer des interfaces utilisateur avec fenêtres, boutons, etc. La définition des GUI par script permet de s'affranchir des spécificités de plateforme. Un formalisme propre de création de contrôles graphiques est disponible en script MEL, identiques dans les deux syntaxes MEL et Python. En syntaxe Python, il est aussi possible d'utiliser un portage Python de Qt via le module PySide disponible. A noter que toute l'interface graphique de Maya est en Qt.

Le meilleur moyen d'apprendre les commandes de script MEL est de voir ce qui est généré dans la fenêtre du Script Editor par echo quand on manipule l'interface. La documentation utilisateur présente les caractéristiques générales de MEL. La documentation référence fait la liste de toutes les fonctions disponibles. Il y a bien sûr une documentation séparée pour la syntaxe MEL et la syntaxe Python (menu Help > Maya Scripting Reference).

Utilisation du script MEL

1. Un premier exemple procédural

Les commandes MEL peuvent être entrées via la ligne de commande en bas gauche, le Script Editor, le Command Shell ou un lien dans une "shelf". En syntaxe MEL, un script dans un fichier .mel est évalué par la commande *source*, en syntaxe Python par la commande *execfile* pour un fichier .py. Attention, évaluer un script contenant la déclaration d'une procédure charge celle-ci, elle devient une commande exécutable, mais ne l'évalue pas. L'erreur typique est de modifier une procédure sous un éditeur de texte, exécuter la procédure à nouveau et s'apercevoir qu'aucun changement n'a eu lieu. Il faut d'abord évaluer le script pour modifier en mémoire de Maya les changements du code.

Ex4. Commandes et procédures Comme premier exemple simple, ouvrir le programme serie.mel. Son exécution permet de générer une série de cubes disposés régulièrement. Lorsque le code est déclaré comme une procédure globale, une nouvelle commande est disponible sous Maya. Un bug d'une rare complexité figure dans le code. Faire le changement nécessaire.

2. Les commandes de base

Voici quelques commandes et principes de base pour démarrer :

- *ls* donne la liste des objets,
- *nodeType* donne le type d'un nœud,
- *listAttr, getAttr setAttr*, editent directement les attributs d'un objet,
- *listRelatives* pour explorer les relations dans le DAG,
- *listConnections* pour explorer les relations dans le DG,
- en syntaxe MEL, le backquote ` permet de récupérer le nom renvoyé en sortie par une commande, typiquement lors de la création, ex: *$objs = `ls`;* en syntaxe Python, *objs=cm.ls()*
- la plupart des commandes admettent trois modes create/query/edit via les flags -c, -q ou -e.

Ex5. Commandes de base sur un polygone cube

Tester toutes ces commandes sur un objet avec *polyCube*. Notamment, essayer de récupérer le nom de l'objet créé via *polyCube* par backquote dans une variable *$obj* et essayer de modifier l'attribut width par exemple, via *polyCube* et via *setAttr*.

3. Construire une interface utilisateur en script MEL

On peut voir un exemple de création de GUI avec le script jointedit.mel en syntaxe MEL (charger préalablement un modèle avec squelette, olaf+pose.mb), et son équivalent jointedit.py en syntaxe Python. Cet exemple montre comment bâtir une interface graphique en script MEL uniquement. Dans ce contexte de programmation, il est aussi possible d'utiliser l'outil Designer de Qt pour créer graphiquement des interfaces. Le principe est de charger le fichier .ui créé par le designer de Qt via la commande MEL *loadGUI*. Les fichiers jointedit_qt.mel et joinedit.ui illustre cette possibilité. Pour fonctionner, le fichier jointedit.ui doit être copié dans votre répertoire local maya/2018/scripts.

4. Exercice Avancé: Editer les connections entre nœuds

Cet exemple a pour but d'explorer les fonctionnalités permettant de consulter, créer et détruire les connections entre nœuds. A noter que tout reste valide, que les nœuds soient de type prédéfinis dans Maya, ou créés via un plug-in.

1. Charger walk.mb et sélectionner l'articulation du genou droit, RightLowLeg

2. La commande *help listConnections* donne la liste des arguments possibles pour consulter les connexions: on va s'intéresser en particuliers aux connections et plugs.

On rappelle que *help -doc cmd* ouvre la page html du manuel pour une commande *cmd*.

- *listConnections -c off -p off* : donne la liste des nœuds connectés (à confirmer avec l'Hypergraph)
- *listConnections -c on -p off* : donne en plus les attributs connectés du nœud considéré.
- *listConnections -c off -p on* : donne la liste des attributs des nœuds connectés.
Rq: si le nœud n'est pas sélectionné, il suffit d'ajouter son nom en fin de commande pour que celle-ci s'applique à ce nœud.

3. La commande *disconnectAttr* casse une liaison, sans pour autant détruire les nœuds connectés
ex: *disconnectAttr RightLowLeg_rotateZ.output RightLowLeg.rotateZ*

4. La commande *connectAttr* crée une liaison
connectAttr RightLowLeg_rotateZ.output LeftLowLeg.rotateZ provoque une erreur
connectAttr RightLowLeg_rotateZ.output RightLowLeg.rotateZ rétablit la connection

5. La commande *addAttr* crée un attribut au nœud considéré. Faire un *help addAttr* pour la liste des options.
addAttr -sn input -at "float" crée un attribut pour ce nœud. Il est connectable en entrée et en sortie : on peut en faire l'exemple sur l'hypergraph.
En retournant dans l'attribute editor, il est visualisable comme "extra attribute" et on remarque qu'il hérite de toutes les fonctionnalités de Maya (animation, expression, etc).

On peut ajouter des types plus complexes avec l'option -dt (voir la doc du manuel) : 3 floats, mesh entier, etc.

5. Exercice avancé: Editer le graphe de scène

1. Charger walk.mb et sélectionner l'articulation du genou droit, RightLowLeg

2. La commande *listRelatives* donne les relations (voir toutes les options avec *help*)
listRelatives -c donne les enfants directs
listRelatives -ad donne toute la descendance
listRelatives -p donne le parent

3. L'option -fullPath ajoute une syntaxe pour donner un nom "absolu" dans le graphe de scène
listRelatives -c -f retourne le chemin, chaque nœud étant séparé par un |

4. La commande syntaxique *tokenize* permet de séparer une chaîne de caractères :
```
string $objs[] = `listRelatives -c -f RightLowLeg`;
string $toks[];
tokenize($objs, "|", $toks);
```

En syntaxe Python on obtient la même fonctionnalité avec la commande interne *split*.

6. Editer les positions et les orientations d'un objet 3D

1. En syntaxe MEL, des types vector et matrix existent mais restent d'utilisation limitée.

Le type vecteur est un groupe de trois réels
vector $v = << 1, 2, 3 >>;
On accède aux éléments via un suffixe .x .y ou .z
A noter que *print $v.x* n'est pas autorisé, il faut utiliser *print ($v.x)*
Ce besoin d'un recours au parenthésage se retrouve souvent.
Le type matrix peut être utilisé avec des tailles variables :
matrix $m[4][3];
print($m);

Malheureusement, les matrices ne se multiplient pas avec les vecteurs et ont très peu d'interactions avec les commandes et attributs.

En syntaxe Python, les listes et t-uples sont disponibles comme type de base. Cependant, le module numpy n'est pas disponible dans l'environnement Python fourni par Maya. Il est possible de trouver des versions de ce module compatibles avec la version 2.7.11 utilisé par Maya et le runtime de compilation (sous windows, maya est compilé en visual studio 2015 alors que le runtime habituel des packages python 2.7 compilés comme numpy est généralement sous windows en visual studio 2012).

2. Les attributs à considérer pour éditer les positions et orientations d'un objet 3D sont portés par le nœud transform.

Les commandes sont alors *move/rotate* (inspiré de l'éditeur) ou plus directes comme *setAttr/getAttr*

A noter qu'un objet hérite de tous les attributs de la hiérarchie de nœud. Hiérarchie est à prendre ici au sens de la "spécialisation" d'un nœud. On voit ici la structuration sous-jacente sous forme de classes type C++. A titre d'exemple, un nœud joint est issu d'un nœud transform, issu d'un nœud dagNode, etc : voir avec la documentation Nodes and Attributes.

Rq: les matrices sous Maya sont considérées comme post-multipliées, les vecteurs sont donc des vecteurs lignes.

3. Une commande plus complète pour éditer position et orientation est la commande xform (voir les options avec *help xform*).

On a notamment une sortie qui donne toute la matrice de transformation :
xform -q -matrix

Si l'on affecte la sortie à une matrice, on obtient une erreur. En effet, cette sortie est un tableau de 16 floats, qu'il faut convertir via une routine à écrire soi-même.

Remarques spécifiques sur l'utilisation de Python sous Maya

On a vu une première utilisation de Python comme syntaxe de script MEL en tant que traduction exacte de la syntaxe MEL via le module maya.cmds. L'usage de Python en tant qu'interface script sous Maya se fait selon deux autres modalités :

- un module Python appelé pyMEL qui reprend le principe des commandes MEL mais avec une philosophie "Orientée Objet", plus en rapport avec les principes de Python,

- un "lieur" de classes de l'API qui peut être utilisé comme script et aussi pour créer des plug-ins. On commence là à sortir du champs d'un langage interprété pour aller vers le domaine des API compilée.

L'exemple simple suivant décrit les manières d'utiliser Python avec maya, par rapport à la commande en syntaxe MEL équivalente.

On part de la création d'une sphère de rayon 5, que l'on translate de 5 selon l'axe Y.

En script MEL, syntaxe MEL:
```
$objs = `polySphere -r 5`
setAttr ($objs[0]+".translate") 0 5 0
```

En script MEL, syntaxe Python,
```
import maya.cmds as cm
objs = cm.polySphere(r=5)
cm.setAttr(objs[0]+'.translate',0,5,0)
```

En pyMEL
```
import pymel.core as pm
pm.polySphere(r=5)[0].translate.set(0,5,0)
```

En API, lieur Python
```
import maya.cmds as cm
import maya.api.OpenMaya as om
objs = cm.polySphere(r=5)
obj = MGlobal.getSelectionListByName(objs[0]) .getDependNode(0)
objPose = om.MFnTransform(obj)
objPose.setTranslation(om.MVector(0,5,0),om.MSpace.kObject)
```

On voit sur ce dernier exemple que l'API n'est pas forcément plus « simple » que le développement de script MEL ou pyMEL. Par exemple, il n'y a pas d'équivalent à la commande *polySphere* qui reste ici en script MEL. L'API donne accès à des fonctionnalités plus avancées, plus efficaces qui sont propres à l'approche de développement par plug-ins.