Introduction langage MEL sous Maya

lionel.reveret@inria.fr

2016-17

L'architecture interne de Maya

L'architecture interne de Maya est construire autour d'un graphe de nœuds connectés (nodes). Deux types de graphes coexistent : le "Dependency Graph ou DG" et le "Directed Acyclic Graph". 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 noeud 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 noeuds 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 visualise le DG et le DAG via les fenêtre Hypergraph, respectivement *connections* et *hierarcy*. 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 pour 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 la liaison entre le node polyCube et le mesh généré depuis la fenêtre *Hypergraph*

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 noeud 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 côte à côte représentant un bras et 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 code. En affichant les repères locaux des cylindres (*Display>Component Display>Local Rotation Axis*), on voit que la liaison squelette/maillage se fait par le DAG (voir fenêtres *Hypergraph* et *Ouliner*)

4. créer un autre cylindre unique de taille double pour tous les bras et une chaîne articulaire identique.

5. Lier le squelette et le cylindre par "smooth skin" (*Animation>Skin>Bind Skin>Smooth Bind*)

6. En affichant le repère local du cylindre, on voit qu'il ne dépend plus de celui du squelette (ni épaule, ni coude). La liaison se fait par le DG.

Ex3. Identifier un type de Noeud

La documentation sur "*Nodes and Attributes*" donne toutes les caractéristiques des nœuds internes: elle est au cœur de Maya et est déjà en jeu avant même une programmation en MEL ou API. Il ne faut donc pas la confondre avec la documentation du langage MEL 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 générale du MEL

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. C'est un langage de programmation interprété. La syntaxe est inspirée de C. On dispose de variables, de types, de flots de contrôle et de procédures. La syntaxe est simplifié 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.

- créer des interfaces utilisateur avec fenêtres, boutons, etc. Les interfaces graphiques peuvent se faire en script MEL, même lors du développement de plug-ins. L'API et le MEL doivent restés indépendants de la plateforme (windows, linux ou mac). L'API et le MEL s'interfacent par appel de procédures.

Le meilleur moyen d'apprendre les commandes MEL est encore de voir ce qui est généré dans la fenêtre du *Script Editor* 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.

Utilisation du MEL

1. Un premier exemple procédurale

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". Un script dans un fichier .mel est évalué par la commande *source*. 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 l'éditeur, exécuter la procédure à nouveau et s'apercevoir qu'aucun changement n'a eu lieu. Il faut d'abord faire un *source* 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. 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 noeud,

- *listAttr*, *getAttr* setAttr, editent directement les attributs d'un objet (les commandes MEL en -query restent plus explicites syntaxiquement),

- *listRelatives* pour explorer les relations dans le DAG,

- *listConnections* pour explorer les relations dans le DG,
- selectedNodes donne la liste des objets sélectionés dans l'interface,
- backquote ` permet de récupérer le nom renvoyé en sortie par une commande, typiquement lors de la création, ex: \$objs = `*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 *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*.

Ex6. Travailler sur une liste d'objet

1. Créer plusieurs polygones (cube, tore, etc)

2. On peut spécialiser ls avec un type d'objet, ex: ls -type mesh; récupérer les objets de type mesh dans une variable \$objs

3. On boucle sur les objets avec une commande for,

for(\$i=0;\$i<size(\$objs);\$i=\$i+1)</pre>

On peut même écrire:

for(\$obj in \$objs)

4. L'affichage se fait avec un simple *print*, les chaînes de caractères se concatènent avec un simple "+". Afficher la liste des objets en mettant le nom entre crochets.

Ex7. Exemples d'animation procédurale

On peut rapidement connecter des "petits" scripts MEL aux attributs des objets via les "expressions". Pour cela on va créer la scène suivante :

1. Créer un cube et un tore.

2. Créer une expression sur l'attribut *translateX* du cube, telle que le cube suive la même corrdonnée en X que le tore quand le tore est dans le demi-espace x<0, et reste sur place avec un mouvement aléatoire sur la coordonnée Z sinon (sous *hypergraph*, observer les connexions crées).

3. Ajouter un cône, qui évite le passage de cube dans le demi-espace x<0 en utilisant des '*driven-key*', contrôlées par la position du cube.

4. Terminer en animant par clés classiques temporelles la position du tore.

3. Construire une interface utilisateur avec MEL

Voir l'exemple de GUI avec le scripts **jointedit.mel** (charger préalablement un modèle avec squelette, **olaf+pose.mb**).

Cet exemple montre comment bâtir un interface graphique avec un langage spécifique MEL. Une autre alternative disponible sous Maya est de pouvoir 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.

A titre d'exercice, reprendre l'interface du script jointedit.mel sous designer et modifier le script pour charger le fichier .ui résultant, tout en gardant le lien sur l'appel des commandes.

4. Exercice Avancé: Editer les connections entre noeuds

Cet exemple a pour but d'explorer les fonctionnalités permettant de consulter, créer et détruire les connections entre noeuds. A noter que tout reste valide que les noeuds 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 la commande <cmd>.

- listConnections -c off -p off : donne la liste des noeuds connectés (à confirmer avec l'*Hypergraph*)

- listConnections -c on -p off : donne en plus les attributs connectés du noeud considéré.

- listConnections -c off -p on : donne la liste des attributs des noeuds connectés.

Rq: si le noeud n'est pas sélectionné, il suffit d'ajouter son nom en fin de commande pour que celle-ci s'applique à ce noeud.

3. La commande **disconnectAttr** casse une liaison, sans pour autant détruire les noeuds 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 noeud considéré. Faire un help addAttr pour la liste des options.

addAttr -sn input -at "float" crée un attribut pour ce noeud. 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)
listRelatives -c donne les enfants directs
listRelatives -ad donne toute la descendance
listRelatives -p donne le parent

2 I 'antion fullDath gioute une suntave neur denner un nem "absolu

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

4. Observer à cette occasion l'utilité de la commande syntaxique tokenize. Elle permet de séparer une chaîne de caractères :
string \$objs[] = `listRelatives -c -f RightLowLeg`;
string \$toks[];
tokenize(\$objs, "|", \$toks);

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

1. Les types vector et matrix existent en MEL mais restent d'utilisation limités.

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, mais 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.

2. Les attributs à considérer pour éditer les positions et orientations d'un objet 3D sont essentiellement portés par le noeud 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 noeud. Hiérarchie est à prendre ici au sens de la "spécialisation" d'un noeud. On voit ici la structuration sous forme de classes type C++. A titre d'exemple, un noeud joint est issu d'un noeud transform, issu d'un noeud dagNode, etc : voir avec la documentation Nodes and Attributes.

Rq: on apprends ici que les matrices sous Maya sont considérés 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 intéressante 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 pour pouvoir acceder à d calcul matriciel. Ceci montre les limites du MEL et incite à passer en API C++.

Maya et Python

Le script MEL est complété par la possibilité d'utiliser Python. Un module Maya est disponible et les scripts Python peuvent être directement exécutés sous Maya. Toutes les commandes spécifiques MEL ont leur contrepartie Python à travers une syntaxe différente, propre à Python. L'utilisation de Python sous Maya se décline de trois manières:

- une réplique de toutes les commandes MEL avec une syntaxe Python,

- 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.

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

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

En MEL: \$objs = `polySphere -r 5`

setAttr (\$objs[0]+".translate") 0 5 0

En commande 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

import maya.api.OpenMaya as om objs = cm.polySphere(r=5) obj = om.MSelectionList().add(objs[0]).getDependNode(0) objPose = om.MFnTransform(obj) obj.Pose.setTranslation(om.MVector(0,5,0),om.MSpace.kObject)