CIS 660: Advanced Topics in Computer Graphics and Animation

Homework Assignment 3 (Spring 2014)

Plug-ins with Python & SWIG and the Maya Dependency Graph

Due: Monday, Feb. 10, 2014

The goals of this assignment are to learn the basics of using SWIG to access C++ functionality from within Python, to gain familiarity with developing Maya plug-ins with Python, and to gain more experience with Maya's Dependency Graph and node networks.

1. SWIG for Python Development

SWIG is a software development tool that can be used to expose functionality written in C and C++ to higher level scripting languages such as Python. Using SWIG, C++ functions can be "wrapped" in Python so that they can be imported and executed directly from within Python code as a module. In fact, the Maya Python API is just a set of SWIG wrappers around the C++ API. The first part of this assignment will focus on wrapping the L-System class developed in the previous assignment with SWIG:

1.1 Getting Started

Make sure your development environment is setup correctly:

a) Make sure that you have Python 2.6 installed on your machine. SWIG can compile to higher versions of Python, but in order to integrate with Maya, you will need Python 2.6. You can download Python 2.6 from www.python.org/download/releases/2.6 and choose the **Windows X86-64 MSI Installer (2.6)** download.

b) Download the HW3 base code Visual Studio project from the blackboard site. This project contains a slightly modified L-System framework, some SWIG base code, along with an example Python script.

c) Download the SWIG executable from the SWIG website. The executable is available from www.swig.org/download.html and accessing the **swigwin-2.0.9** link. Once downloaded, copy the inner "swigwin-2.0.9" directory to your Visual Studio project directory.

1.2 [10 pts] SWIG(ing) your L-System

The next step is to set up Visual Studio so that it will compile the Python wrappers for the L-System:

(a) As a first step, read through the SWIG tutorial located at www.swig.org/tutorial.html. This will give you some good insights into how SWIG functions. Pay particular attention to the section "SWIG for the truly lazy" and "Surely there's more to it…".

(b) Take a look at the LSystem.i file for the general format of SWIG include files. Notice that we have defined two data structures for Python: VecFloat and VectorPyBranch. In

Python you will use something like branches = LSystem.VectorPyBranch() to initialize a vector of branches to be filled by the process function.

(c) Next, setup Visual Studio to build the Python wrappers. This process is rather tedious, and it is explained in **Appendix A** to keep this section concise.

(d) Try importing the LSystem library into Maya in the Python script editor. In order to access your LSystem Python bindings from within Maya, you must copy the files in your project's bin directory to the bin directory of your Maya install. This is usually in the location: C:\Program Files\Autodesk\Maya2012\bin

2. Updating the L-System

Now that you can access the L-System from Python, let's add a little bit of functionality to the L-System to make the output a little more interesting. Recall that L-Systems consist of an input grammar defining the structure of the system and a set of rules for interpreting the grammar. An example grammar:

$$\omega : F - F - F - F$$
$$\rho : F \Rightarrow F - F + F + FF - F - F + F$$

An extension of this basic L-System would be to insert new symbols representing some type of different structure (perhaps leaves or flowers). For example, let's say that the symbol "*" represents a flower in the final L-System. Then the string "F – F – F *" would be three branch segments with a flower at the end.

2.1 [10 pts] Update the L-System to Output Flowers

(a) Update the processPy function in LSystem.cpp so that it can output branches and flowers to Python. Make sure you look at all of the sections marked LOOK in the LSystem.cpp file.

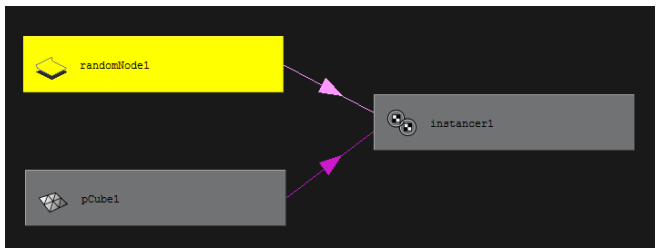
(b) Use the symbol "*" to represent a flower. Create a couple of new grammars that include flowers in the plants/ directory.

3. Maya Plug-ins in Python & the Maya Hypergraph

Perhaps one of the most powerful aspects of Maya is the Hypergraph. Through the Hypergraph's system of nodes with inputs and outputs, the possibilities for what can be connected and created are nearly endless. This section of the assignment is focused on gaining familiarity with integrating custom API nodes with the Maya Hypergraph as well as creating plug-ins in Python.

While there are probably hundreds of Maya nodes that can be integrated into Maya tools, in this assignment, we will be focusing on Maya's "instancer" node as an example and will be creating two nodes that interact with this instancer node. The main idea behind the instancer node is that it creates copies (or instances) of some input geometry so that the changes to the input geometry also appear in all of the instanced geometry. This not only lowers the memory costs of your L-System, but it also allows for quick and easy changes to the geometry.

In more detail, the instancer node is a node that takes in a piece of input geometry and an MFnArrayAttrsData object containing the positions, scales, etc. for the instances. The instancer then creates instances of the input geometry with the transforms described in the MFnArrayAttrsData object.



The image above is the Hypergraph for the instancer. As you can see, the instancer has two inputs: an input geometry (from pCube1), and an MFnArrayAttrsData object (from randomNode1). The two nodes that you will create will each supply the MFnArrayAttrsData object to the instancer. More information on the MFnArrayAttrsData object can be found in **Appendix B**.

3.1 [30 pts] Create a Random Position Node

Using the randomNode.py base code provided, create a node that takes in a minimum and a maximum bound (6 floats) and the number of random points to be generated, then outputs these random points as an MFnArrayAttrsData object. You will load your plug-in from the Plugin Manager window like you did with the .mll files.

(a) Create and add an input MFnNumericAttribute for number of random points.

(b) Create and add input MFnNumericAttributes for the bounds of the random points. This will require 6 floats. (Take a look at cgfxVector.cpp for an example of how to combine 3 floats attributes into a single vector attribute).

(c) Create and add an MFnTypedAttribute for the generated random points. The type of this attribute should be MFnArrayAttrsData.

(d) Complete the compute function. This function should fill the MFnArrayAttrsData output with random points generated within the input bounds. It should contain the "id" and "position" properties. See Appendix B for more details.

(e) Using the MEL script in Appendix B, connect your randomNode and some geometry to an instancer. Try hooking the output mesh of your previous assignment's LSystemNode to the instancer and take some screenshots!

3.2 [40 pts] Create an L-System Instancer Node

Now let's look at a new way to create the geometry for an L-System that makes use of Maya's instancer node. The benefit of creating an L-System using this method is that you can connect any type of geometry you want for the branches and flowers (and even have different geometry for different L-Systems in your scene). The idea here is that each segment of the LSystem will be

an instance of the input geometry. Implement the class LSystemInstanceNode derived from MPxNode (use randomNode as an example).

(a) Create input attributes as you did in the previous assignment for angle, step-size, grammar file, and iterations.

(b) Create and add an MFnTypedAttribute for the branch segments. The type of this attribute should be MFnArrayAttrsData, and for each branch of the L-System, the instancer will create an instance.

(c) Create and add an MFnTypedAttribute for the flowers. Again, the type should be MFnArrayAttrsData, and for each flower of the L-System, there will be an instance.

(d) Make your MFnArrayAttrsData contain not only "id" and "position" but also "scale" and "aimDirection" so that you can orient the branches of your LSystem correctly. Don't forget, you can also change the instancer's input geometry; MFnArrayAttrsData only controls the transformations!

3.3 [10 pts] Create a Menu and MEL scripts for setting up your nodes

Unconnected, these nodes and the instancers are surprisingly not useful. However, by connecting them, you can make some really cool models and images.

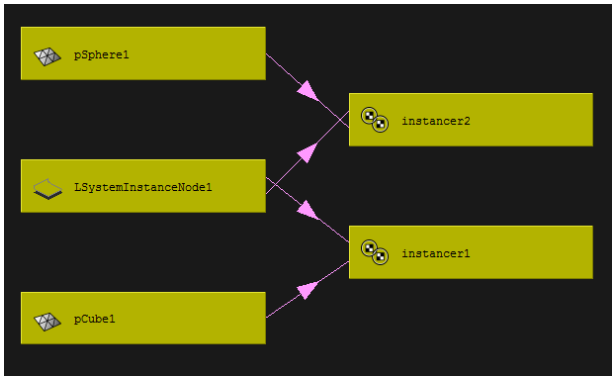
Make a new menu (that loads with your plug-in) with the name LSystemInstance. It should contain at least four options:

1) An option to create a randomNode network. This option should call a MEL script that creates a polygon sphere, an instancer, and a randomNode, and then connects their attributes to create the network. (Consult the script in Appendix B.)

2) An option to create a randomNode network with the user selected object as the input geometry to the instancer. Basically the same as the previous option except uses the user selected geometry as the input geometry instead of a polygon sphere. This option should only execute if a single object is selected.

3) An option to create an LSystemInstanceNode network. This option should call a MEL script that creates a polygon cube, a polygon sphere, two instancer nodes, and an LSystemInstanceNode, and then connects their attributes to create the network. The cube should be the geometry for the branches, and the sphere should be the geometry for the flowers.

4) An option to create an LSystemInstanceNode network with the user selected objects as the branches and flowers, respectively. This option should only execute if two objects are selected; the first should be the branch geometry, and the second should be the flower geometry.



Example node network for LSystemInstanceNode

3.4 Submission

(a) ZIP your Visual Studio project folder with all of your files. Please delete the .SDF file before you submit the project.

(b) Briefly write-up which parts of the assignment you completed in a file called README.txt. Don't forget to point out any extra credit that you attempted. In addition to the readme, please also submit some screenshots or renders of your generated L-Systems.

3.5 [Extra Credit, up to 20 pts] Extend your L-System further!

For maximum extra credit, include screenshots and videos of your best work.

(a) Create a custom procedural shader network for your LSystem. An example would be a shader that changes its output color based on its position in the world. To earn this extra credit you must add a button to your LSystem menu that creates the shader network and applies it to selected geometry.

(b) Update the LSystem so that as branches get farther away from the root, they get thinner.

(c) Add more functionality to your L-System class such as a symbol for leaves or multiple types of flowers. To earn this extra credit you must provide sample grammars and setup the node network in a MEL script accessible from the menu.

APPENDIX A:: Setting up SWIG and Visual Studio

This Appendix will cover the steps required to add SWIG as a build step to Visual Studio so you can generate Python wrappers for your code. This guide is based on a guide by Mark Tolonen from StackOverflow.

1. Open the Visual Studio Project and if the .i file is not already listed in the project, right click Project in Solution Explorer -> Add -> Existing Item, and choose the .i file.
2. Right click the project, go to properties and select Configuration "All Configurations".
3. Select Configuration Properties -> C/C++ -> General and add the path of your Python26\include directory to Additional Include Directories. By default this will be "C:\Python26\include".
4. Select Configuration Properties -> Linker -> General and add the path of your Python26\libs directory to Additional Library Directories. By default this will be "C:\Python26\libs".
5. Select Configuration Properties ->General and change Output Directory to "$(ProjectDir)\bin\", change Target Name to "_$(ProjectName)", and change Target Extension to ".pyd".
6. Right click the .i file, go to Properties and select Configuration "All Configurations".
7. Change Item Type to "Custom Build Tool" and click Apply.
8. Select "Custom Build Tool" and for the Command Line field enter:
   "swigwin-2.0.9\swig.exe -c++ -python –outdir $(Outdir) %Identity"
   (This assumes that the swigwin-2.0.9 directory is in your Project directory)
9. In Outputs enter "%(Filename)_wrap.cpp;$(Outdir)%Filename.py" and click OK.
10. Right click the .i file, and select Compile.
11. Right click the project, press Add -> New Filter, and name this filter "Generated Files".
12. Right click Generated Files, click Properties, and set "SCC Files" to "False".
13. Right click Generated Files, press Add -> Existing Item, and select the _wrap.cpp file that was generated by the .i file compile.
14. Build the "Release" version of the project. You can't build the Debug version unless you build a debug version of Python itself.
15. Open the console, go to the bin directory of the project, run python ("C:\Python26\python.exe"), and try to import your module. (In this case it would be "import LSystem")

APPENDIX B:: More info on the Instancer & MFnArrayAttrsData

An MFnArrayAttrsData object can contain the following data, but only "position" and "id" are required:

id (doubleArray)
position (vectorArray)

age (doubleArray)	scale (vectorArray)	shear (vectorArray)
visibility (doubleArray)	objectIndex (doubleArray)	aimWorldUp (vectorArray)
rotationType (doubleArray)	rotation (vectorArray)	aimDirection (vectorArray)
aimPosition (vectorArray)	aimAxis (vectorArray)	aimUpAxis (vectorArray)

To create an MFnArrayAttrsData attribute you must do something like the following:

```
tAttr·create("outPoints", "op", OpenMaya·MFnArrayAttrsData·kDynArrayAttrs)
```

In order to fill the MFnArrayAttrsData object, you will need to have some code similar to the following:

```
pointsData = data·outputValue(randomNode·outPoints) #the MDataHandle
pointsAAD = OpenMaya·MFnArrayAttrsData()          #the MFnArrayAttrsData
pointsObject = pointsAAD·create()                 #the MObject

# Create the vectors for "position" and "id"· Names and types must match
# table above·
positionArray = pointsAAD·vectorArray("position")
idArray = pointsAAD·doubleArray("id")

# Loop to fill the arrays:
for item in list:
    positionArray·append(<SOME_MVECTOR>)
    idArray·append(<SOME_NUMBER>)

# Finally set the output data handle
pointsData·setMObject(pointsObject)
```

Connecting the inputs and outputs:

In order to get your instancer actually displaying geometry, you will have to make a few connections with MEL. Consider the following example:

```
polyCube;
instancer;
createNode randomNode;
connectAttr pCube1·matrix instancer1·inputHierarchy[0];
connectAttr randomNode1·outPoints instancer1·inputPoints;
```

In this example, a polygon cube, an instancer, and a randomNode are created. The “matrix” attribute of the polygon cube is connected to the inputHierarchy[0] attribute of the instancer, and the “outPoints” (an MFnArrayAttrsData object) attribute of the randomNode is connected to the “inputPoints” attribute of the instancer.

In general any geometry (transform) node can be connected to the instancer with the “matrix” attribute. In addition, any node that outputs an MFnArrayAttrsData attribute can connect to the instancer’s “inputPoints” attribute.