CIS 6600: Advanced Topics in Computer Graphics and Animation

Homework Assignment 3 (Spring 2025)

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

Due: Wed., Feb. 12, 2025

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:

1. **Maya 2022:** Maya 2022 upgraded the Python version from 2.7 to 3.7. Any Python version later than 3.7.7 will work. Python 3.7.9 downloads can be found here: https://www.python.org/downloads/release/python-379/

2. Download the HW3 base code Visual Studio project from the CIS6600 Canvas site. This project contains a slightly modified L-System framework, some SWIG base code, along with an example Python script.

3. Download the SWIG executable from the SWIG website. The executable is available from http://www.swig.org/download.html by accessing the **swigwin-4.0.1** link. Once downloaded, copy the inner "**swigwin-4.0.1**" directory to your Visual Studio project directory.

4. **Other versions of Maya**: To find out what version of Python your Maya install is using, go to the bin folder in your Maya folder, open the command terminal, and run "mayapy.exe". This will open Maya's version of python in your command line and will print the version. Please download this version to use when setting up your project as per the instructions in Steps 3 & 4 of Appendix A.

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:

1) As a first step, read through the SWIG tutorial located at: http://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…".
2) 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.
3) 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.
4) 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\Maya2022\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

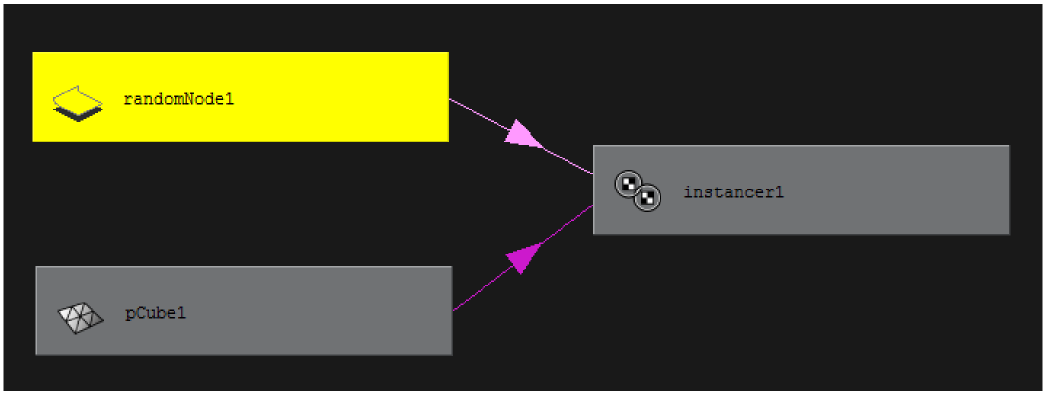
1) Update the processPy function in LSystem.cpp so that it can output branches with flowers at the end. Make sure you look at all of the sections marked LOOK in the LSystem.cpp file.
2) Use the symbol "*" to represent a flower. Create a couple of new grammars that include flowers in the plants/ directory.

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

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 *(20 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.

1) Create and add an input MFnNumericAttribute for number of random points.
2) 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).
3) Create and add an MFnTypedAttribute for the generated random points. The type of this attribute should be MFnArrayAttrsData.

4) 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.

5) 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 *(30 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).

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

2) 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.

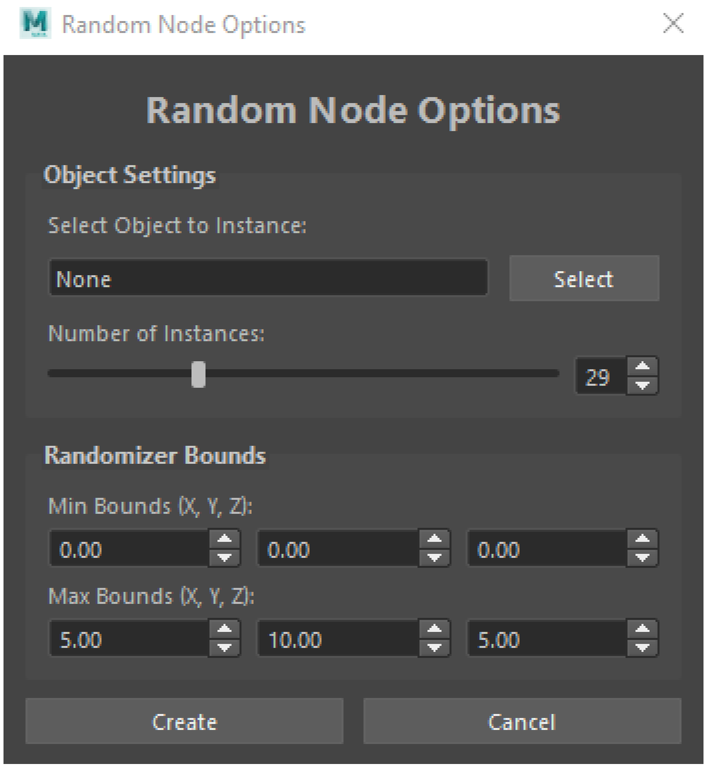
3) Create and add an MFnTypedAttribute for the flowers. Again, the type should be MFnArrayAttrsData, and for each flower of the L-System, there will be an instance.

4) 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!

4. PyQt GUI & Connecting Nodes

4.1 Random Node Maya UI

We will next create a GUI for the user to create a Random Node. The goal of this part of the assignment is to use *ChatGPT* (https://chat.openai.com/), *Claude* (https://claude.ai/), *Deepseek* (https://chat.deepseek.com/), or another LLM software to help you generate **Python** Script(s) that create something *similar* to the Maya UI panel shown below:



Your implementation should include the following GUI features :

1) A way for the user to select the object to instance. The simplest way to do this is to add a button that records the currently selected object, but feel free to get creative with how you want to approach this!
2) A Slider/Text Field for the user to input the amount of objects to instance
3) A series of Text Fields for the user to input the max/min bounds for the node.
4) A button that creates the Randomizer Node with the parameters inputted into your GUI.

4.2 *(15 pts)* GUI Generation w/ LLM

We will be using the PyQt GUI Framework for Maya to create a GUI fully within Python, able to use the Maya Python API natively. Check out **Appendix C** for more information to get started with the PyQt framework!

1) Use your chosen LLM to create your GUI scripts! Note that the GUI does not have to look exactly like the example, just that it has the same essential features. Make sure to tell it your essential features, as well as your Maya Version to ensure that the right modules are imported.
    a) Here are some general guidelines/tips for generating your GUI with the LLM:
        i) When starting out, have the LLM generate one feature at a time. Although it is tempting to have it generate the whole GUI at once, bug fixing (as well as understanding the API) will be easier in small increments.
        ii) The LLM in some cases can be prone to hallucinations and make up a command that does not exist in the API. If you suspect this, look at the documentation (linked in Appendix C) to confirm and inform the LLM so it can avoid this and find an alternative.
        iii) Sometimes the LLM will not create the feature as you intended. For ChatGPT, tell it to ***edit*** the code instead of regenerating it all.
        iv) Save the encoding of the node connection logic (i.e. when the user clicks the "Create" button) for last. Until then, have the button have a print statement as a placeholder.
            (1) To link the node connections with Python, the commands are extremely similar to the MEL Commands in Appendix B. You can find the command here.
            (2) Alternatively, you can run MEL Commands in Python using `maya.mel.eval('command')`

4.3 *(5 pts)* LLM Generation Post Mortem

After you finish your GUI, complete the following Post Mortem and include it in a README.txt file with your submission.

1) In general, do you often use LLMs, or tools like CoPilot for other coding tasks (for other classes or personal projects)?
2) Which LLM/Version did you use? Why?
3) What roadblocks did you hit while working with the LLM? Were there any specific features that were especially hard to generate?
4) Was there anything that worked well for you? List any tips/tricks you would recommend to a classmate when using a LLM for generating Maya Code.
5) Did you feel that using the LLM(s) improved your efficiency or learning compared to

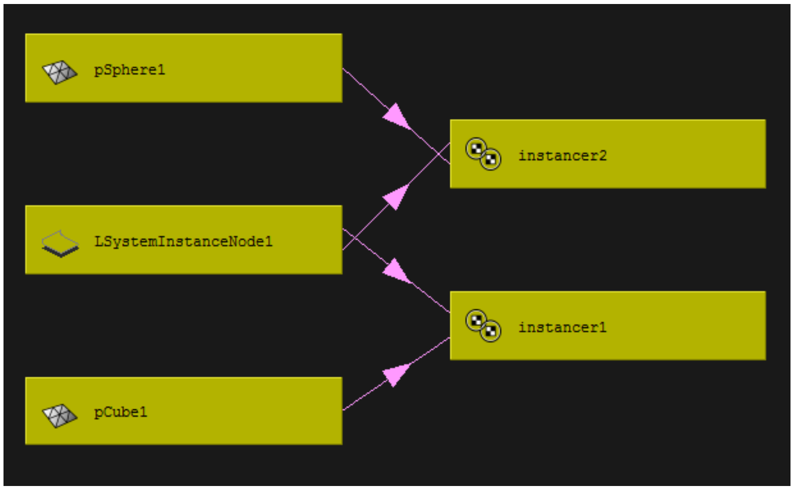
developing without it. Specify which one (if you tried different LLMs) and why?

4.4 *(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 three options:

1) An option to create a randomNode network. This option should call the Python Script that creates your GUI from the previous section.
2) An option to create an LSystemInstanceNode network. This option should call a MEL or Python script that creates a polygon cube, a polygon sphere, two instancer nodes, and an LSystemInstanceNode, and then connects their attributes to create the network (As listed in Appendix B). The cube should be the geometry for the branches, and the sphere should be the geometry for the flowers.
3) 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

5. 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. Make sure to include your LLM post-mortem there as well. 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.

(c) **Include with your submission screenshots of your L-System output and a video showing your Python plugin in action. Include your randomizer node GUI functionality as well**.

6. *(Extra Credit, max 20 pts)* Extend your L-System further!

For maximum extra credit, make sure you 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.

(d) Use PyQt or MEL to create a GUI for your LSystemInstancerNode. Feel free to build upon your existing GUI from the previous assignment.

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 then -> Add -> Existing Item, and choose the .i file.
2. Right click the project, go to Properties and set Configuration at top of page to “All Configurations”.
3. Under Configuration Properties select -> C/C++ -> General and add the path of your Python37\include directory to Additional Include Directories. On some machines the default for this might be “C:\Python37\include”.
4. Under Configuration Properties select -> Linker -> General and add the path of your Python37\libs directory to Additional Library Directories. On some machines the default for this might be “C:\Python37\libs”.
5. Under Configuration Properties ->General, change Output Directory to “$(ProjectDir)\bin\” and Target Name to “_$(ProjectName)”,
6. Under Configuration Properties ->Advanced, change Target Extension to “.pyd”.
7. Right click the .i file, select Properties and set Configuration at top of page to “All Configurations”.
8. Under Configuration Properties select -> General and change Item Type to “Custom Build Tool” , then click Apply.
9. Under Configuration Properties select “Custom Build Tool” and for the Command Line field enter: “swigwin-4.0.1\swig.exe -c++ -python -outdir $(Outdir) %(Identity)”. This assumes that the swigwin-4.0.1 folder is in your Project directory.
10. In Outputs enter “%(Filename)_wrap.cpp;$(Outdir)%(Filename).py” and click OK.
11. Right click the .i file, and select Compile.
12. Right click the project, press Add -> New Filter, and name this filter “Generated Files”.
13. Right click Generated Files, press Add -> Existing Item, and select the “LSystem_wrap.cxx” file that was generated by the .i file compile. Right click on the LSystem_wrap.cxx file and rename it to “LSystem_wrap.cpp”.
14. Build a “Release” version of the LSystem project. You can’t build the Debug version unless you build a debug version of Python itself.
15. As a test, open the command console, go to the bin directory of the LSystem project, run the python command (assuming the Python exe is on the system path) and try to import your LSystem 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
# the 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.

APPENDIX C: Getting Started with PyQt

PyQt is a commonly used framework in the Games & Animation industry for building GUIs for tools in Maya. Since it can easily interface with Maya's native Python bindings, it is commonly used as an alternative to MEL for more complex GUIs. PyQt itself is a Python Wrapper built over the Qt C++ Framework, similar to what we have done in this assignment with SWIG and the LSystem Library.

To Start with PyQt, you must import the PySide Library, which comes with your Maya Install. For Maya Versions 2022-2024, you can use:

```
from PySide2 import QtWidgets, QtCore
```

For Maya 2025, use:

```
from PySide6 import QtWidgets, QtCore
```

To create your first window, use the script:

```
class SimpleWindow(QtWidgets.QDialog):
    def __init__(self, parent=None):
        super(SimpleWindow, self).__init__(parent)
        self.setWindowTitle("PyQt Example")
        self.setMinimumSize(300, 150)

        layout = QtWidgets.QVBoxLayout(self)
        self.button = QtWidgets.QPushButton("Click Me")
        layout.addWidget(self.button)

        # Connecting button signal to slot
        self.button.clicked.connect(self.on_button_click)

    def on_button_click(self):
        print("Button clicked!")

# Show the window
win = SimpleWindow()
win.show()
```

Like in MEL, Components/Widgets in PyQt are placed within Layouts. There are several layouts, including `QHBoxLayout, QVBoxLayout, QGridLayout, QStackedLayout`. Widgets can have functionality connected to them through slots and signals.

For more information, feel free to check out these resources (in addition to generating code examples from LLMs):

- https://www.pythonguis.com/tutorials/pyside-creating-your-first-window/
- https://www.pythonguis.com/pyside6-tutorial/
- https://doc.qt.io/qtforpython-5/PySide2/QtWidgets/QWidget.html