Title of Seminar: Advanced Maya Python

Presenter: Dean Edmonds, Autodesk

Introduction

With the power of the Maya API and the scriptability of MEL, Maya Python hits a sweet spot in Maya's suite of development tools. Being an interpreted language it allows for quick turnaround, making it perfect for rapid prototyping and with thousands or dedicated users around the world contributing modules to Python, Maya inherits a large and growing array of new capabilities.

In this seminar we will explore some of those new capabilties and show how they can be used to build faster, more sophisticated tools for binding together the disparate elements of the typical production pipeline. We'll discuss how the dynamic nature of Python's runtime environment allows us customize standard tools and datatypes to meet special needs. We'll take a look at some standard Python modules and novel ways in which they can be used. At the end of the seminar we will talk about the benefits of moving heavy computation into C++ for improved performance and step through the process of doing so.

A Note About the Examples

To simplify the examples we will assume that the following `import` statements have already been executed:

```
import maya.OpenMaya as om
import maya.OpenMayaMPx as ompx
import maya.cmds as cmds
```

Getting a Python Object From a Proxy Object Pointer

The Maya Python API documentation contains a section entitled *Need self but only API object is available* which describes how to convert a pointer to a proxy object into a Python object so that its methods and attributes can be accessed. We have received several requests for help in this area so it seems appropriate to cover it in more detail.

The basic approach is to have the proxy class keep track of its objects in a dictionary which other classes can then use to retrieve them. The key to this dictionary is provided by the `maya.OpenMayaMPx.asHashable()` method which will take either a proxy object or a pointer to a proxy object and return a value which is unique to that object.

The example given in the documentation is of an MPxContext-derived class which calls its `_newToolCommand()` method and gets back a pointer to an MPxToolCommand, but the technique can be successfully used in most areas where the Maya Python API returns a pointer to an object. For example, another common situation is when `MFnDependencyNode::userNode()` returns a pointer to the node's proxy object. Before any of the proxy object's methods can be called the pointer must be converted into a Python object.

A Python class can have two special methods named `__init__` and `__del__`. `__init__` is known as the class's *constructor* and is called whenever a new object of that class type is created. `__del__` is the class's *destructor* and is called whenever an object of that class type is destroyed. These methods can be used to maintain a dictionary of objects for the class:

```
class sineNode(ompx.MPxNode):
     nodeTypeId = om.MTypeId(0x8700)
     objDict = {}

     def __init__(self):
          ompx.MPxNode.__init__(self)
          nodeKey = ompx.asHashable(self)
          sineNode.objDict[nodeKey] = self

     def __del__(self):
          nodeKey = ompx.asHashable(self)
          del objDict[nodeKey]
```

Whenever a new object is created the `__init__` method creates a dictionary entry for it and when the object is destroyed the `__del__` method deletes its dictionary entry.

The dictionary can then be used by other classes to convert a returned pointer into the corresponding proxy object:

```
nodeFn = om.MFnDependencyNode(nodeObject)
if nodeFn.typeId() == sineNode.nodeTypeId:
     nodePtr = nodeFn.userNode()
     nodeKey = ompx.asHashable(nodePtr)
     node = sineNode.objDict[nodeKey]

     # Now we can call the proxy object's methods.
     node.someMethod()
```

Overriding __str__() For Better Runtime Information

Some Maya API types display their values nicely when printed. MIntArray is an example of this. It displays its array of values as a Python list:

```
ints = om.MIntArray()
ints.append(5)
ints.append(-7)
ints.append(12)

print ints
[5, -7, 12]
```

However there are some API types which display their values is a less friendly manner. Consider MVector:

```
v = om.MVector(1, 2, 3)

print v
<maya.OpenMaya.MVector; proxy of <Swig Object of type 'MVector
*' at 0x40c2bf0> >
```

When Python displays the value of an object it first converts the object to a string by calling the object's `__str__()` method. We can customize an object's string formatting by replacing its class's `__str__()` method with one of our own:

```
# Define a method to format an MVector into a string.
def myMVector_str(self):
     return “(%g, %g, %g)” % (self.x, self.y, self.z)

# Save MVector's existing __str__() method.
oldMVector_str = om.MVector.__str__

# Replace MVector's __str__() method with our own.
om.MVector.__str__ = myMVector_str

# Try printing the vector again.
print v
(1, 2, 3)

# Restore the original __str__() method.
om.MVector.__str__ = oldMVector_str
```

It's a good idea to restore the original `__str__()` method when done, just in case other modules are relying on it.

Even those API types which already display their values in a user-friendly way may benefit from a custom implementation of `__str__()` in some situations. For example, let's say that we have a sparse array of integers:

```
sparse = om.MIntArray(100)

sparse[7] = -3
sparse[20] = 13
sparse[82] = 6

print sparse
[0, 0, 0, 0, 0, 0, 0, -3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
13, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The output would be more readable if the repetitious sequences of zeros were compressed into a more concise format. We can do that as follows:

```
# Replacement for MIntArray's __str__() method
# which compresses long sequences of identical values.
def compressedMIntArray_str(self):
     s = "["
     count = 0
     prev = None
     for i in self:
          if i == prev:
               count += 1
          else:
               if prev != None:
                    s += compressSeq(count, prev) + ", "
               count = 1
               prev = i

     s += compressSeq(count, prev) + "]"
     return s

# Helper function which returns a string containing
# 'count' copies of 'value'.
# If 'count' is three or less then they are returned as
# a comma-separated list, otherwise they are returned
# in 'value * count' notation. E.g. '3*24' means a
# sequence of 24 values of 3.
def compressSeq(count, value):
     if count > 0:
          if count > 3:
               s = str(value) + "*" + str(count)
          else:
               s = str(value)
               for i in range(1, count):
                    s += ", " + str(value)
     else:
          s = ""
     return s

# Replace MIntArray's __str__() method with our own.
om.MIntArray.__str__ = compressedMIntArray_str
```

Now when we print out the sparse array we get a much more compact representation:

```
print sparse
[0*7, -3, 0*12, 13, 0*61, 6, 0*17]
```

See the alt_str.py file on the conference CD/DVD.

Create Your Own Debugger Interface

The pdb debugger is a standard part of every Python implementation. You can use it to step through the execution of a Python function one statement at a time. For example, let's say that we have the following code in a module named *dist.py*:

```
def distance(p1, p2):
     dist = 0
     if len(p1) == len(p2):
          for i in range(0, len(p1)):
               diff = p1[i] - p2[i]
               dist += diff*diff
          dist = dist**0.5
     return dist
```

If we were to execute pdb within a command-line Python interpreter, then our debug session might look something like this:

```
python
>>> import dist
>>> import pdb
>>> pdb.run("dist.distance((3,4), (5, 7))")
> <string>(1)?()
(Pdb) s
--Call--
> /home/deane/dist.py(1)distance()
-> def distance(p1, p2):
(Pdb) n
> /home/deane/dist.py(2)distance()
-> dist = 0
(Pdb) n
> /home/deane/dist.py(3)distance()
-> if len(p1) == len(p2):
(Pdb) n
> /home/deane/dist.py(4)distance()
-> for i in range(0, len(p1)):
(Pdb) n
> /home/deane/zot.py(5)distance()
-> diff = p1[i] - p2[i]
(Pdb) n
> /home/deane/dist.py(6)distance()
-> dist += diff*diff
```

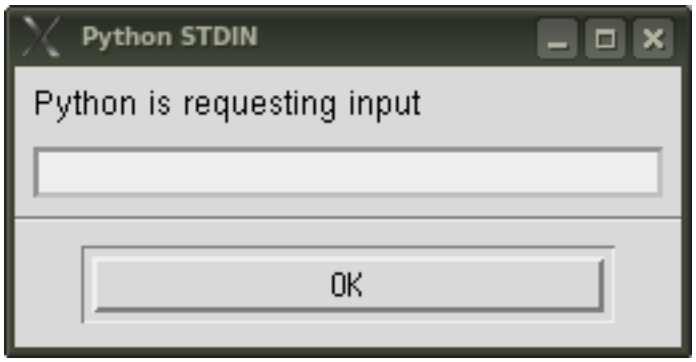
and so on.

Each time you see `(Pdb)` above that is the debugger prompting for a command. The command

set, which is fully documented in the Python Debugger section of the Python Library Reference, is similar to that of `gdb`, the GNU debugger. For example, if the next statement contains a function call then the 's' ('`step`') command will step into the function while the 'n' ('`next`') command will step over it.

After each command is executed pdb prints a line giving the module and line number of the next statement, as well as the name of the function that it is in, followed by the statement itself, then a prompt for the next command.

When run from within Maya the behaviour is slightly different. Because Maya's Script Editor does not have a mechanism for responding to prompts within the Script Editor window, it instead displays a separate input window each time pdb prompts for a new command:



You would, for example, enter your 's' or 'n' command into the input field, then press Enter or click on the OK button to have pdb execute it. On Linux this is particularly cumbersome because each time the window appears the focus is on the OK button, so you must first change the focus to the input field, then enter your command and press Enter.

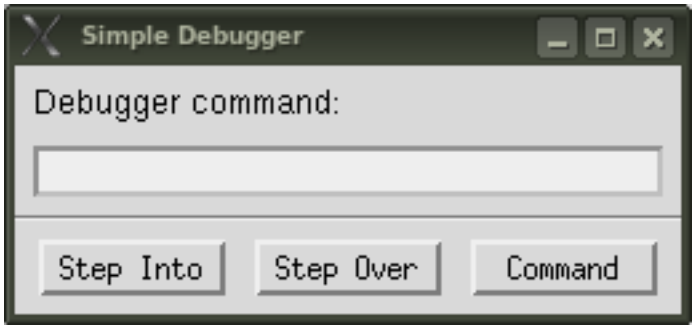
We can make this interaction smoother by replacing Maya's prompt window with one of our own devising which is better suited to the needs of the debugger. We do this by replacing the `sys.stdin` object used by Maya with our own. Assume that the following code resides in module *guipdb.py*:

```
import pdb
import sys
import os
import maya.cmds as cmds

class DebuggerStandardInput:
     def readline( self ):
          return self._getUserDebugResponse()

     def read( self ):
          return self._getUserDebugResponse()

     def _getUserDebugResponse(self):
          result = cmds.promptDialog(
               title='Simple Debugger',
               message='Debugger command:',
               button=['Step Into', 'Step Over','Command'],
               defaultButton='Command')

          if result == 'Command':
               result = cmds.promptDialog(query=True, text=True)
          elif result == 'Step Over':
               result = 'n'
          elif result == 'Step Into':
               result = 's'
          return result

def run(command):
     # Save Maya's current stdin.
     orig_stdin = sys.stdin

     # Replace stdin with an instance of our own input class.
     sys.stdin = DebuggerStandardInput()

     # Run pdb.
     pdb.run(command)

     # Restore Maya's original stdin.
     sys.stdin = orig_stdin
```

Now we can run the debugger as follows:

```
import guipdb
import dist
guipdb.run("dist.distance((3,4), (5, 7))")
```

Whenever the debugger prompts for a command the following window will appear:



The 'Step Into' and 'Step Over' buttons provide fast access to the two most common debugger commands.

If you look at the output being sent to the Script Editor you'll see that while our commands are no longer being echoed the '`(Pdb)`' prompt is:

```
guipdb.run("dist.distance((3,4), (5, 7))")

> <string>(1)<module>()->None
(Pdb) --Call--
> /home/deane/maya/scripts/dist.py(1)distance()
-> def distance(p1, p2):
(Pdb) > /home/deane/maya/scripts/dist.py(2)distance()
-> dist = 0
(Pdb) > /home/deane/maya/scripts/dist.py(3)distance()
-> if len(p1) == len(p2):
(Pdb) > /home/deane/maya/scripts/dist.py(4)distance()
-> for i in range(0, len(p1)):
(Pdb) > /home/deane/maya/scripts/dist.py(5)distance()
-> diff = p1[i] - p2[i]
```

We can eliminate that clutter by replacing Maya's stdout object with one of our own which filters out those prompts. To do this we add the following to *guipdb.py*:

```
class DebuggerStandardOutput:
     def __init__(self, orig_stdout):
          self.orig_stdout = orig_stdout

     def write(self, msgoutput):
          line = msgoutput.strip()
          if line[:5] == "(Pdb)":
               line = "\n"
          self.orig_stdout.write(line)

     def writelines(self, msgSeq):
          for l in msgSeq:
               self.write(l)

     def flush(self):
          return None
```

and modify its `run()` function as follows:

```
def run(command):
     # Save Maya's current stdin and stdout.
     orig_stdin = sys.stdin
     orig_stdout = sys.stdout

     # Replace stdin and stdout with instances of
     # our own classes.
     sys.stdin = DebuggerStandardInput()
     sys.stdout = DebuggerStandardOutput(orig_stdout)

     # Run pdb.
     pdb.run(command)

     # Restore Maya's original stdin and stdout.
     sys.stdin = orig_stdin
     sys.stdout = orig_stdout
```

Now when we execute `guipdb.run()` the Script Editor output is a bit cleaner:

```
guipdb.run("dist.distance((3,4), (5, 7))")
><string>(1)<module>()->None
--Call-->/home/deane/maya/scripts/dist.py(1)distance()
-> def distance(p1, p2):
>/home/deane/maya/scripts/dist.py(2)distance()
-> dist = 0
>/home/deane/maya/scripts/dist.py(3)distance()
-> if len(p1) == len(p2):
>/home/deane/maya/scripts/dist.py(4)distance()
-> for i in range(0, len(p1)):
```

```
>/home/deane/maya/scripts/dist.py(5)distance()
-> diff = p1[i] - p2[i]
>/home/deane/maya/scripts/dist.py(6)distance()
-> dist += diff*diff
```

By extending the `DebuggerStandardInput` and `DebuggerStandardOutput` classes it is possible to build a fairly sophisticated GUI interface to the Python debugger. For example, `DebuggerStandardOutput` could parse the current line information and make it available to `DebuggerStandardInput` to display in a separate field in its prompt window.

One note of caution when hijacking standard input and output in this way: if the code that you are debugging is also using the standard input or output then you could run into problems. It may be possible to work around those difficulties in many cases, but it is important to be aware that the potential for error exists.

See the guidpdb.py file on the session CD/DVD.

Data Transfer Using The cPickle Module

It is often necessary to transfer data between different applications or between different sessions of the same application. Indeed the very idea of a film or game development “pipeline” is that of data flowing from one stage of the pipeline to the next.

In an ideal world Maya would meet every production need and transferring data along a pipeline would be a simple matter of saving out a scene file and passing it on. But the reality is that there may be many points in the pipeline where data will have to pass between Maya and external tools, not least of which are the in-house tools which every studio needs to bind its pipeline together and harness it to their particular needs.

Every one of those data transfer points requires a definition of the format of the data to be transferred, code to write the data in that format, and code to read it. When transferring data between commercial applications this functionality may already be in place in the form of file translators, but when dealing with tools written in-house the burden of creating those formats, of writing the code to read and write, and of debugging the whole shebang, falls squarely on the shoulders of in-house developers. Throughout the industry there are people who spend most, if not all, of their time working on data transfer tools.

However, if your in-house tools are written in Python then there are ways of significantly lightening that burden.

Python provides a pair of modules called `pickle` and `cPickle` which can be used to automatically read and write most Python objects. No data format needs to be defined because the structure of the object itself provides the format.

`pickle` and `cPickle` are identical in functionality. In `cPickle` all the heavy lifting is done by C code rather than Python. That makes it much faster than `pickle` at the cost of some degree of customation which is rarely needed. For the rest of this discussion we'll be focussing on `cPickle`.

Since the most common use of `cPickle` is to write data to files and read it back in, the module has facilities to simplify that workflow. To write data you open a file for writing, create a `Pickler` object, then pass the `Pickler` object whatever Python objects you want it to dump to the file.

Let's walk through a simple example. Assume that we need to interface with a tool which keeps track of the shot information being worked on by each user. The information required by this tool is defined by the `Shot` class in the `ShotData` module:

```
class Shot:
     def __init__(self):
          self.user = None
          self.production = None
          self.sequence = None
          self.scene = None
          self.shot = None
```

In Maya a shelf button brings up a window which allows the user to select the shot to work on and then loads the corresponding scene. It also populates a `Shot` object called `currentShot` with the shot information and calls the module's saveShotDefaults function to store it in a filed called *shotDefaults* the user's home directory:

```
import cPickle
import os

def saveShotDefaults(shotInfo):
     try:
          home = os.environ['HOME']
          fileName = os.path.join(home, 'shotDefaults')
          file = open(fileName, 'wb')
          pickler = cPickle.Pickler(file, 2)
          pickler.dump(shotInfo)
          file.close()
     except:
          pass
```

Going through that bit-by-bit, we first find the user's home directory and append to that the string `'shotDefaults'` to give us the full pathname of the file we will be writing to. Then we call `open` to open the file. The `'wb'` as the second argument to the `open` call means that we're opening the file for writing and will be writing binary (i.e. not human-readable) data to it.

Next, we create a `Pickler`, passing it the open file and telling it to use protocol 2 which means to write the data as binary. Then we tell the `Pickler` to dump the `shotInfo` object to the file, close the file, and we're done. Notice that there was no need to specify a file format because the `Pickler` used the layout of the `shotInfo` object's data as the file format.

Some elements of the user's current shot information, such as the production and scene, will change rarely from session to session and the user's name won't likely change at all. So rather than having the user re-enter that data every time he wants to work on a shot, it makes sense to reuse the default information, if there is any. So each time Maya starts up, it imports the `ShotData` module and calls its `loadShotDefaults` function:

```
def loadShotDefaults():
     try:
          home = os.environ['HOME']
          fileName = os.path.join(home, 'shotDefaults')
          file = open(fileName, 'rb')
          unpickler = cPickle.Unpickler(file)
          shotInfo = unpickler.load()
          file.close()
     except:
          # We couldn't get the user's defaults so
          # let's just create an empty Shot.
          shotInfo = Shot()

     return shotInfo
```

Once again we build the pathname of the file where the defaults are stored but this time we open it for binary reading. We create an `Unpickler`, passing it the open file. We don't have to tell the `Unpickler` what protocol to use because it will be able to determine that itself from the contents of the file. We next call the unpickler's load method which will create a `Shot` object from the contents of the file and return it to us. We then close the file and return the `Shot` object to the caller. If anything fails then we just return an empty `Shot` object.

With a very small amount of code we've managed to provide a persistent state for the user's current shot information.

The `cPickle` module doesn't just read and write to files, it will read from any object which provides `read` and `readline` methods and will write to any object which provides a `write` method. That means that it can read and write to strings via the `StringIO` class. This is such a common usage that `cPickle` provides convenience methods for reading and writing strings:

```
# Dump the object to a string
s = cPickle.dumps(obj)

# Load the object from a string.
obj = cPickle.loads(s)
```

This makes it easy to pass data across a network. For example, the shot selection window may communicate with a central server using a TCP/IP socket. When requesting a new shot it will pass the server a copy of the user's current shot information so that the server can pick out the next nearest shot which is still available. For that we will need a class which contains the request type and current shot information:

```
class ShotRequest:
```

```
        def __init__(self):
             self.type = None
             self.shot = None
```

Now we can create a function to get a new shot from the server:

```
SERVER = 'localhost'
PORT = 50507

def requestNewShot(oldShot):
     # Create the request object.
     request = ShotRequest()
     request.type = 'NewShot'
     request.shot = oldShot

     # Create a socket and connect it to the server.
     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
     host = SERVER
     port = PORT
     s.connect((host, port))

     # Pickle the request into a string and send it
     # to the server.
     requestStr = cPickle.dumps(request, 2)
     s.send(requestStr)

     # Read the server's response into a string and
     # unpickle it into a Shot object.
     responseStr = s.recv(4096)
     newShot = cPickle.loads(responseStr)

     s.close()

     return newShot
```

The corresponding code for the server might look something like this:

```
def startServer():
     # Create a socket for listening for requests.
     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

     # Server always runs on the host it's on, so 'localhost'
     host = 'localhost'
     port = PORT
     s.bind((host, port))
     s.listen(5)

     while True:
          # Wait for a sender to connect.
          conn, sender = s.accept()

          # Get the request string and unpickle it.
          requestStr = s.recv(4096)
          request = cPickle.loads(requestStr)

          # Handle the request.
          if request.type == 'Stop':
               conn.close()
               break
          elif request.type == 'NewShot':
               newShot = getNextShot(request.shot)

               # Pickle the new shot into a string
               # and send it back to the requestor.
               responseStr = cPickle.dumps(newShot)
               conn.send(responseStr)

          conn.close()

     s.close()
```

See ShotServer.py on the session CD/DVD.

Using lambda Functions To Simplify Callbacks

If the action of a function can be expressed as a single expression then rather than going through the full syntax of a function definition, Python allows the function to be written in a more concise *lambda form*.

A lambda form function consists fo the keyword `lambda` followed by a comma-separated list of the function's formal parameters, followed by a colon and then the expression which forms the body of the function. For example, the following code using a lambda function:

```
lenSquared = lambda a, b: a**2 + b**2

print lenSquared(3, 4)
```

is equivalent to the following standard function form:

```
def lenSquared(a, b):
     return a**2 + b**2

print lenSquared(3, 4)
```

A lambda function can be a handy replacement for a callback function which performs a very simple task. For example, consider the following code which uses callbacks to maintain a list of all the transform nodes in the scene:

```
def transformAdded(node, l):
     l.append(om.MObjectHandle(node))

def transformRemoved(node, l):
     l.remove(node)

xforms = []

id1 = om.MDGMessage.addNodeAddedCallback(
          transformAdded, “transform”, xforms)
id2 = om.MDGMessage.addNodeRemovedCallback(
          transformRemoved, “transform”, xforms)
```

Each of those last two calls takes three args. The first arg is the callback function, the second is the string `“transform”`, indicating that we only want the callback to fire for transform nodes, and the third arg is the `xforms` list which will be passed to the callback function as its second argument. (The first argument passed to the callback function will be the node which is being added or removed.)

We can eliminate the two function definitions by using lambda forms to inline them:

```
xforms = []

id1 = om.MDGMessage.addNodeAddedCallback(
          lambda node, l: l.append(om.MObjectHandle(node)),
          “transform”, xforms)
id2 = om.MDGMessage.addNodeRemovedCallback(
          lambda node, l: l.remove(node), “transform”, xforms)
```

Another advantage of a lambda function is that it has access to variables from the scope in which it was defined. In our example above the lambda forms have direct access to the xforms list, so they don't need to have it passed to them as an argument. That allows us to simplify the code even further:

```
xforms = []

id1 = om.MDGMessage.addNodeAddedCallback(
          lambda node, x: xforms.add(om.MObjectHandle(node)),
          “transform”)
id2 = om.MDMessage.addNodeRemovedCallback(
          lambda node, x: xforms.remove(node), “transform”)
```

The primary constraint on the use of lambda functions is that the function body must consist of a single expression. For example, let's say we just wanted to keep a count of the number of transforms in the scene. It would be nice to do be able to do the following:

```
class Counter:
     pass

counter = Counter()
counter.count = 0

id1 = om.MDGMessage.addNodeAddedCallback(
          lambda node, x: x.count += 1, “transform”, counter)
id2 = om.MDMessage.addNodeRemovedCallback(
          lambda node, x: x.count -= 1, “transform”, counter)
```

Unfortunately this will produce a syntax error because assignment (i.e. `x.count += 1` and `x.count -= 1`) is not an expression and is therefore not allowed in a lambda function. We could get around this by giving our `Counter` class methods to increment and decrement a stored count, as follows:

```
class Counter:
     def __init__(self):
          self.count = 0

     def inc(self):
          self.count += 1
          return self.count

     def dec(self):
          self.count -= 1
          return self.count

counter = Counter()
id1 = om.MDGMessage.addNodeAddedCallback(
          lambda node, x: x.inc(), “transform”, counter)
id2 = om.MDMessage.addNodeRemovedCallback(
          lambda node, x: x.dec(), “transform”, counter)
```

but this defeats our original reason for using lambda functions because it leaves us with even more code than if we'd just use normal, non-lambda callback functions to handle the increment and decrement.

Writing C++ Python Modules for Maya

There is very little which can be done in C++ which cannot also be done in Python. Indeed, because of its syntax and the easy integration of MEL commands, Python solutions generally end up being more concise and easier to understand than their equivalents in C++.

There are still reasons why you might want to push certainly functionality into C++, though. For example, you may want to make use of existing C++ tools and libraries. However the most common reason to use C++ is to improve performance. Because Python is an interpreted language it is slower to execute than compiled C++ code, and the wrapper code which interfaces Maya's C++ API to Python introduces some overhead of its own. Shifting heavily-used or time-critical computation into C++ can produce a significant boost in performance.

The example below demonstrates the performance gains to be had. It is a simple Python module with a single function which takes the name of a mesh and returns a tuple containing its centroid:

```
def mesh(meshName):
     # Get the mesh's dag path.
     sList = om.MSelectionList()
     sList.add(meshName)
     path = om.MDagPath()
     sList.getDagPath(0, path)

     if not path.hasFn(om.MFn.kMesh):
          raise ValueError, "'" + meshName + "' is not a mesh."

     # Get the mesh's vertex positions.
     meshFn = om.MFnMesh(path)
     verts = om.MPointArray()
     meshFn.getPoints(verts, om.MSpace.kWorld)
     numVerts = verts.length()

     if numVerts == 0:
          raise ValueError, "Mesh '" + meshName + "' has no
vertices."

     # Add all the vertex positions together.
     x = 0.0
     y = 0.0
     z = 0.0

     for i in range(0, numVerts):
          pt = verts[i]
          x += pt.x
          y += pt.y
```

```
            z += pt.z

        # Calculate the average position.
        divisor = 1.0 / numVerts
        x *= divisor
        y *= divisor
        z *= divisor

        return (x, y, z)
```

The module is called `pyCentroid`. We also have a module called `cppCentroid` which implements the identical algorithm in C++ code, which we'll take a look at shortly.

To test the relative performance of these two implementations, we create a polyPlane with a quarter of a million vertices and then use the timeX command to see how long it takes each implementation of our centroid algorithm to return a result:

```
import pyCentroid
import cppCentroid

cmds.polyPlane(h=10, w=10, sx=498, sy=498)

t = cmds.timerX()
pyCentroid.mesh('pPlane1')
print cmds.timerX(st=t)
1.48

t = cmds.timerX()
cppCentroid.mesh('pPlane1')
print cmds.timerX(st=t)
0.11
```

As you can see, the Python version completed in 1.48 seconds while the C++ version completed in 0.11 seconds, making it over 14 times faster. So the performance gains to be had from a C++ implementation are quite compelling.

The `python.org` web-site has a document entitled *Extending and Embedding the Python Interpreter* which describes how to create a C++ module for use with Python. Adding Maya to the mix only requires a few small changes. We'll go over the process now, using the cppCentroid module as our example.

The first step is to create a source file containing the C++ code. You can give the source file any name you like but it simplifies matters if you give it the same name as you want your resulting module to be named, but with an extension of `.cpp`. We want our module to be called `cppCentroid` so we named our source file `cppCentroid.cpp`.

At the top of the file we must include the `Python.h` header file, followed by any other header files we may need. It is important that `Python.h` be first:

```
#include <Python.h>

#include <maya/MDagPath.h>
#include <maya/MFnMesh.h>
#include <maya/MPoint.h>
#include <maya/MPointArray.h>
#include <maya/MSelectionList.h>
```

Next, create the 'mesh' function which will do the centroid calculation:

```
static PyObject* cppCentroid_mesh(PyObject*, PyObject* args)
{
    ...
}
```

We can name the function whatever we like but it must be a static function which takes two `PyObject` pointers in its parameter list and returns a `PyObject` pointer as its result. The two parameters are the `self` pointer and a tuple containing the arguments which were passed to the function when it was called from within Python. The `self` pointer is only used when creating methods for classes. Since we are creating a standalone function we won't be using it and have left its name out of the function declaration to avoid the warning messages that some C++ compilers would generate about “unused” parameters.

The first thing our method must do is extract the name of the mesh from the `args` tuple. This is easily done using Python's `PyArg_ParseTuple` function:

```
const char*    meshName;
if (!PyArg_ParseTuple(args, "s:mesh", &meshName))
{
    return NULL;
}
```

`PyArg_ParseTuple` takes three arguments: the args tuple which was passed to our function, a format string which describes the types of arguments which we are expecting the tuple to hold, and a list of one or more pointers to variables to hold the arguments which are extracted from the tuple.

In our case the format string is quite simple, just an 's' indicating that we are expecting the tuple to contain a single string, followed by ':mesh' at the end of the format string to tell `PyArg_ParseTuple` to use 'mesh' as the name of our function in any error messages it may

display. For the third argument we pass a pointer to the `meshName` variable which will receive the name of the mesh.

Now that we have the name of the mesh, the body of the function is fairly straightforward:

```
// Get the mesh's dag path.
MSelectionList slist;
slist.add(meshName);

MDagPath       path;
slist.getDagPath(0, path);

if (!path.hasFn(MFn::kMesh))
{
     return PyErr_Format(PyExc_ValueError,
               "'%s' is not a mesh.", meshName);
}

// Get the mesh's vertex positions.
MFnMesh        meshFn(path);
MPointArray    verts;
meshFn.getPoints(verts, MSpace::kWorld);

unsigned int numVerts = verts.length();
if (numVerts == 0)
{
     return PyErr_Format(PyExc_ValueError,
               "Mesh '%s' has no vertices.", meshName);
}

// Add all the vertex positions together.
double x = 0.0;
double y = 0.0;
double z = 0.0;

for (unsigned int i = 0; i < numVerts; ++i)
{
     MPoint& p = verts[i];
     x += p.x;
     y += p.y;
     z += p.z;
}

// Calculate the average position.
double divisor = 1.0 / (double)numVerts;

x *= divisor;
y *= divisor;
z *= divisor;
```

This is basically the same as the Python version except for the error handling. Whenever an error occurs an appropriate Python exception must be raised and the function must return NULL. To do that we call `PyErr_Format` telling it to raise a `ValueError` exception and providing a description of the error. The error description is handled the same as a standard `printf` call, with a format string followed by the values to be substituted into the format string.

The last thing that our function must do is return the centroid as an (x, y, z) tuple. For that we use the `Py_BuildValue` function:

```
return Py_BuildValue("(ddd)", x, y, z);
```

Py_BuildValue takes a format string which describes the type of Python object to be built, followed by a list of the values to be used in building it. The three 'd's indicate that we will be supplying three double values. Enclosing the 'd's within parentheses indicates that we want them returned as a tuple.

Now that we have our function we need to create the “method table” which lists of all the methods and functions in our module. In our case we only have the one function so the table contains just two entries: one for our mesh function, and an entry full of NULLs to mark the end of the list.

```
static PyMethodDef cppCentroidMethods[] =
{
     { "mesh", cppCentroid_mesh, METH_VARARGS,
          "Find the centroid of a mesh." },
     { NULL, NULL, 0, NULL }
};
```

The table is a static array of `PyMethodDef` elements and can have any name we like.

Each entry in the table consists of four elements: the name by which the function (or method) will be known inside Python, a pointer to the actual C++ function to be called, a flag giving the calling convention to be used, and a string describing the purpose of the function.

Because our function takes a single argument without the use of keyword/value pairs, we use the `METH_VARARGS` calling convention. If we wanted to allow keyword/value pairs then we would include the `METH_KEYWORDS` flag using the C++ binary-or operator. E.g. `METH_VARARGS|METH_KEYWORDS`.

The last piece of code we need to complete our module is the initialization function which Python will call when loading the module. The name of this function *must* be 'init' followed by the name of the module and it must be a non-static function so that it will be visible to Python:

```
extern "C" PyMODINIT_FUNC initcppCentroid()
{
     Py_InitModule("cppCentroid", cppCentroidMethods);
}
```

C and C++ can differ in the conventions they use when calling functions. Since Python is expecting to call a C function we have to add `extern "C"` to the start of our function declaration to tell the C++ compiler to use the C calling convention for this function.

The body of the function is simply a call to `Py_InitModule`, passing it the name of our module and a pointer to the module's method table.

Compiling and linking a C++ Python module for Maya is done in the same way as a C++ plugin for Maya, with a couple of minor changes. Below we discuss each supported platform in turn.

Building The Module On Linux

On Linux we can use the existing `buildconfig` file provided in the devkit. We can also use the `Makefile` from the devkit but we need to add the following line immediately after `buildconfig` has been included:

```
INCLUDES += -I$(MAYA_LOCATION)/include/python2.5
```

At the end of the Makefile we add a build rule for the module the same as we would for a plugin:

```
CPPCENTROIDOBJS = cppCentroid.o

cppCentroid.$(EXT): $(CPPCENTROIDOBJS)
     -rm -f $@
     $(LD) -o $@ $(CPPCENTROIDOBJS) $(LIBS)
```

We can then build the module from the command line in the usual way:

```
make cppCentroid.so
```

The built module must be copied into Maya's *lib/python2.5/site-packages* directory. For example, if we were building for Maya 2008 and had it installed in the usual place, then directory would be */usr/autodesk/maya2008/lib/python2.5/site-packages*.

Building The Module On OS X

On OS X we can use the existing `buildconfig` file provided in the devkit. We can also use the `Makefile` from the devkit but we need to add the following lines immediately after

`buildconfig` has been included:

```
PYVERSION = 2.5
PYLOC = $(MAYA_LOCATION)/Frameworks/Python.framework/Versions/Current
INCLUDES += -I$(PYLOC)/include/Python$(PYVERSION)
LFLAGS += -L$(PYLOC)/lib/python$(PYVERSION)/config -lpython$(PYVERSION)
```

At the end of the Makefile we add a build rule for the module the same as we would for a plugin except instead of use `$(EXT)` for the extension we hard-code it to `.so` since that's what Python is expecting:

```
CPPCENTROIDOBJS = cppCentroid.o

cppCentroid.so:     $(CPPCENTROIDOBJS)
     -rm -f $@
     $(LD) -o $@ $(CPPCENTROIDOBJS) $(LIBS)
```

We can then build the module from the command line in the usual way:

```
make cppCentroid.so
```

The built module must be copied into Maya's *lib/python2.5/site-packages* directory, which is located in the Python framework deep within Maya's installation tree. For example, if we were building for Maya 2008 and had it installed in the usual place, then the directory would be */Applications/Autodesk/maya2008/Maya.app/Contents/Frameworks/Python.framework/Versions/Current/lib/python2.5/site-packages*

Building The Module On Microsoft Windows

For Microsoft Windows we will have to separately download and install the same version of Python as Maya uses. We can get the version number by executing the following Python command from the Script Editor:

```
print sys.version
```

Maya 2008 uses version 2.5.1 of Python and for the remainder of this discussion we will assume that it is installed in its default location of *C:\Python2.5*.

We create a Visual C++ project for the module the same as we would for a Maya C++ plug-in, then make the following changes to the project's properties:

- In the **General** section of the **C/C++** configuration properties, add to the end of the **Additional Include Directories** field the path to Maya's python2.5 include directory.

E.g. *C:\Program Files\Autodesk\maya2008\include\python2.5*

- In the **General** section of the **Linker** configuration properties, add to the end of the **Additional Library Directives** field the path to Python's 'libs' director. E.g: *C:\Python2.5\libs*
- Also in the **General** section of the **Linker** configuration properties, change the extension of the **Output File** to *.pyd*
- In the **Command Line** section of the **Linker** configuration properties, remove from the **Additional Options** field the two /**export** flags listed there and replace them with an export of your module's initialization function. E.g: /export:initcppCentroid

Once we have built the module, we copy it into Maya's *Python\lib\site-packages* directory. For example, if we have Maya 2008 installed in its default location, then the directory would be *C:\Program Files\Autodesk\maya2008\Python\lib\site-packages*.

Limitations

As we saw in the example, C++ Python modules can make use of Maya's API. However Maya API objects cannot be passed from Python to the module's functions, nor may they be returned to Python as a result of the module's functions. In our example we passed the cppCentroid.mesh function the name of the mesh, which is a simple string. If we had wanted instead to pass the mesh as an MObject or MDagPath we wouldn not have been able to do so.

The reason for this restriction has to do with the manner in which the Python interface to the API has been derived. A tool called SWIG automatically generates the code to convert the Maya API types between their Python and C++ representations. SWIG stores these conversion functions as a set of internal macros and table entries and we don't currently expose that functionality in OpenMaya.

Often it is possible unbundle the pertinent data from an API type into standard Python types, such as passing a string for an MDagPath or a list of integers for an MIntArray. Although this unbundling will incur its own overhead, the sheer magnitude of the performance gains to be had by shifting heavy computations into C++ will generally still make it worth doing.

See the pyCentroid.py and cppCentroid.cpp files on the session CD/DVD.