Custom Nodes in Maya

Autodesk Maya

Introduction

This document presents the concepts for implementing custom nodes using Maya's C++ Application Programming Interface (API). An outline of Maya nodes and Maya's architecture will be presented first, followed by the concepts for creating three types of nodes: a Dependency Graph (DG) node, a locator, and a manipulator.

The Dependency Graph

At the heart of Maya's architecture is the Dependency Graph, or the DG. It is the control and data storage system of Maya. The DG is what allows seemingly disparate operations to work together seamlessly to produce the final output. Those operations are implemented as nodes, which store data through their attributes and pass those data using connections.

Nodes

Nodes are the fundamental building blocks of the DG. A node stores its data through its attributes. Typically, a node's job is to take data from its input attributes to produce data for its output attributes. This is done via a compute() function.

Maya's architecture does not distinguish between nodes built into Maya or those implemented from plug-ins, adding to the power and flexibility of Maya.

Attributes

Nodes store their data through attributes. An attribute can represent simple data types like int, float, double, string, boolean, etc., or complex types like mesh, curve or surface data, etc.

An attribute can be designated as an input or an output. An output attribute is computed from one or more inputs, so it is not normally stored in a Maya scene file and is not allowed to have incoming connections from other attributes. Input and output attributes also have a relationship that indicates which inputs are used to compute which outputs. This is called an "attributeAffects" relationship.

Plugs

At a high level, attributes and plugs appear to refer to the same thing. However, when programming with Maya, careful distinction needs to be made between them. An attribute merely describes the data that belongs to nodes of a given type, i.e. the type of data it can store and other properties, like whether it's an input or an output, etc.

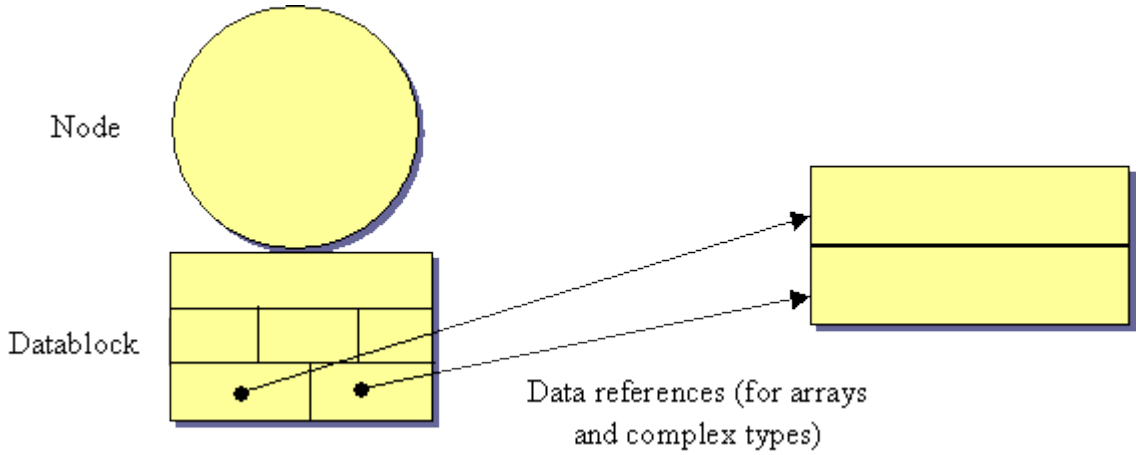
On the other hand, a plug points to a specific attribute on a specific node, and is used for getting/setting data or making/breaking/validating connections.

In the API, objects representing attributes are manipulated using MFnAttribute or one of its more specialized child classes, while those representing plugs are handled with the MPlug class.

Datablock

The datablock is the central location where all the node's data from its attributes is stored. A node will have one datablock for the current time context. When needed, the node can also have other transient datablocks to represent other time contexts, for example when the node is animated.

For non-array attributes, the datablock not only stores the value but also the dirty/clean status, which is used during the DG's evaluation process and determines whether or not the value needs to be recomputed. For array attributes and those of a variable size complex type, the datablock stores a reference to another area of memory. This allows complex types to implement their own efficient storage methods, and keeps the size of the datablock constant for faster allocation.



Access to the datablock is provided inside a node's compute() function, via an MDataBlock object.

Datahandles

Datahandles are pointers to the data residing in the datablock. A datahandle is similar to a plug in that they are both used to set or retrieve data. However, there are crucial differences:

- A datahandle is used within a node's compute() function, while a plug is used outside.
- A datahandle's get/set methods are more efficient since it intimately knows the layout of the datablock.
- Setting data via a plug propagates the dirty/clean status, while doing so via a datahandle does not.

Datahandles are obtained from the MDataBlock object as MDataHandle objects.

Since data for array attributes are stored separately, they are accessed using more specialized array datahandles, which are represented by MArrayDataHandle objects in the API.

DG Evaluation

The DG's evaluation process can be characterized as a lazy, 2-stage push-pull model. It is lazy in that it only evaluates when necessary.

During the first stage, when an attribute's value is set, a "dirty" message instead of actual data is "pushed" to every affected attribute via attributeAffects relationships or connections.

During the second stage, only when a dirty attribute's value is requested will the data be "pulled" by causing the necessary nodes to compute their dirty attributes. Note that at this stage, not all dirty attributes will be generally evaluated at the same time.

Creating Custom Nodes

To implement a custom node in the Maya API, you would create a class derived from MPxNode, or one of its child classes when a more specialized node type is needed. For example, derive from MPxLocatorNode to create a locator, or derive from MPxDeformerNode to implement a deformer, etc.

Custom nodes must have a unique type ID (MTypeID). The type ID is used to identify the node type in a Maya binary (.mb) file. A unique ID prevents your node from potentially clashing with another node type when Maya is saving it to or loading it from the file.

You can use node IDs from the range 0 - 0x7ffff for internal testing purposes. However, to ensure global uniqueness, even if the nodes will only be used internally, you must contact Alias Technical Support to obtain a range of IDs.

It is important to settle on a node's ID as early as possible. If there are scene files that contain the node, and then the ID is changed, Maya won't be able to load the node's data on those files.

Most custom nodes are Dependency Graph or DG nodes derived from MPxNode. Here's the class declaration of a typical DG custom node:

```
#include <maya/MPxNode.h>

class myNode : public MPxNode {
public:
    myNode();
    virtual ~myNode();

    static void* creator();
    static MStatus initialize();

    virtual MStatus compute(const MPlug &plug, MDataBlock &block);

    static MTypeId id; // type ID

    static MObject input; // input attribute
    static MObject output; // output attribute

};
```

The type ID and the MObjects representing the node's attributes are redeclared below the class declaration, with the type ID being assigned a value:

```
    MTypeId myNode::id(0x0ffff);

    MObject myNode::input;
    MObject myNode::output;
```

The constructor is called whenever an instance of the node is created, for example via the createNode MEL command or MFnDependencyNode::create(), etc.

The destructor is called only when the node is truly destroyed. If undo is enabled, and the node is deleted, the node is sent to the undo queue so that its deletion can be undone, but at this point, the destructor is not yet called. If the undo queue is flushed, then the destructor will be invoked.

The creator() method is called by Maya to create instances of the node, and can be named to anything you want. It simply returns a pointer to your node class:

```
static void* myNode::creator() {
    return new myNode;
}
```

In some cases, you would want to implement a postConstructor() method, which is called after the constructor. Internally, Maya creates two objects when a custom node is created: the internal MObject and the user-derived object. The association between these two is not made until after the MPxNode constructor is called. You will not have access to MPxNode member functions or your node's attributes/plugs until the postConstructor is being called.

The initialize() method is where the node's attributes are created and where the attributeAffects relationships are established. The method is called only once when the node is registered in Maya during the loading of the plug-in. This is another method that can be named to anything you want.

To create the attributes, use the appropriate attribute function set class, e.g. MFnNumericAttribute for numeric attributes, MFnTypedAttribute for more complex types like strings, matrices, mesh data, etc.:

```
MFnNumericAttribute nAttrFn;

input = nAttrFn.create("input", "in", MFnNumericData::kFloat, 0.0);

// input can't be the source of a connection
nAttrFn.setReadable(false);

output = nAttrFn.create("output", "out",MFnNumericData::kFloat,0.0);

// output can't be the destination of a connection…
nAttrFn.setWritable(false);

// …nor does it need to be saved in the scene file
nAttrFn.setStorable(false);
```

Once the attributes have been created, add them to the node:

```
addAttribute(input);
addAttribute(output);
```

Finally, set up the necessary attributeAffects relationships:

```
attributeAffects(input, output);
```

The node is registered in Maya when the plug-in is loaded using the initializePlugin() method:

```
    MStatus initializePlugin(MObject obj)
    {

        MStatus stat;
        MFnPlugin plugin(obj, "Alias - Example", "1.0", "Any");

        stat = plugin.registerNode("myNode",
myNode::id,             myNode::creator, myNode::initialize,
            MPxNode::kDependNode, NULL);

        return stat;

    }
```

The first parameter to registerNode() is the type name, which is used to identify the node type in some MEL commands (createNode, ls, etc.) and in Maya ASCII (.ma) files. The second is the node's type ID. The third is the creator function while the fourth is the initialize function.

The fifth parameter is the type of node being registered. For DG nodes, you can leave it blank or specify MPxNode::kDependNode. It is mandatory if you are registering more specialized nodes, for example MPxNode::kLocatorNode for locators or MPxNode::kDeformerNode for deformers, etc.

The last parameter is the classification string, which is required only for custom render nodes, for example "shaders/surface" for shading nodes, "texture/2d" for 2D textures, etc.

When the plug-in is being unloaded using the uninitializePlugin() function, the node must also be deregistered:

```
    MStatus uninitializePlugin(MObject obj)
    {
        MStatus stat;
        MFnPlugin plugin(obj);

        stat = plugin.deregisterNode(myNode::id);

        return stat;
    }
```

The deregisterNode() method only requires the node's type ID.

```
    compute()
```

The compute() method processes data from the node's input attributes to produce results for its output attributes. It is called during the second stage of the DG evaluation process, when the value of an attribute of the node is requested. The method receives a plug object, representing the attribute whose value is being requested, and the node's datablock:

```
   MStatus myNode::compute(const MPlug &plug, MDataBlock &block) {
```

The compute() method will typically have the following logic, expressed in pseudocode:

```
if plug == an attribute belonging to this node  get the value(s)
    from the necessary input(s) calculate output
    from the input(s)
    return appropriate MStatus (kSuccess if OK)

else if plug == another attribute of this node
    do the same steps as above
else
    return MS::kUnknownParameter
```

You must always test that the plug object represents an attribute created by your node. Otherwise, compute() may execute unnecessarily. Also, since compute() is one of many things that happens during the DG evaluation process, in order to ensure that the DG's states remain correct, you must not call it manually.

Going back to C++ code, i f an attribute of this node is being requested, get its input via a datahandle from the datablock and get the value from the datahandle:

```
if (plug == output) {
    MDataHandle inHandle = block.inputValue(input);
    float inputVal = inHandle.asFloat();
```

Calculate the output and set the result on the output attribute's datahandle:

```
    float result = 2 * inputVal + 1;

    MDataHandle outHandle = block.outputValue(output);
    outHandle.set(result);
```

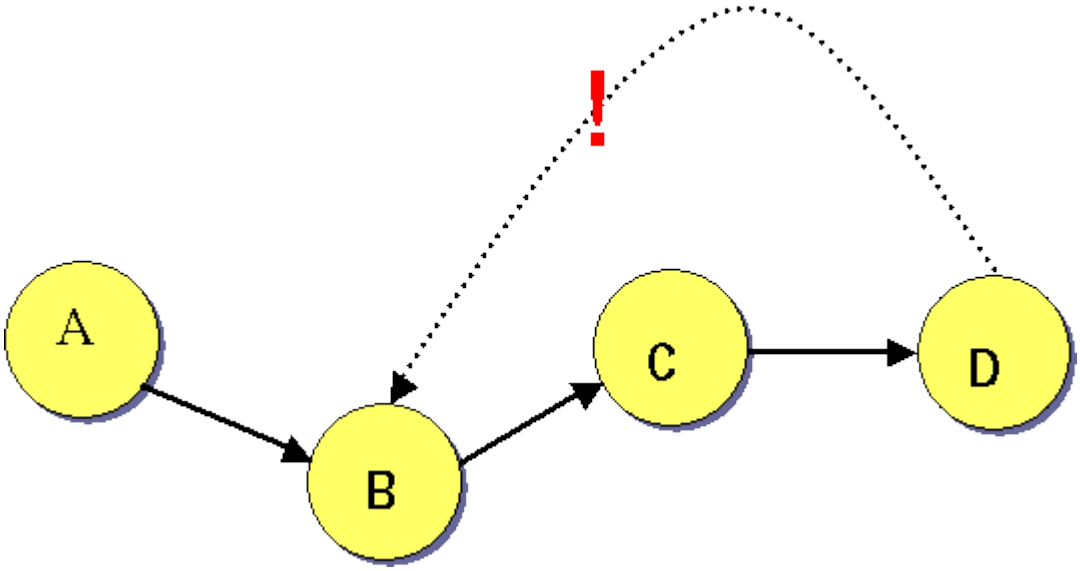
Don't forget to call setClean() for the plug on the datablock, so the DG knows that this attribute is now up-to-date:

```
    block.setClean(output);
    return MS::kSuccess;
}
```

The difference between using inputValue() and outputValue() to obtain a datahandle is that the former will trigger an evaluation, if needed, while the latter does not.

Black-Box Operation

The most important tenet that must be followed by nodes is what's sometimes called the "black-box" principle: a node must compute its outputs using ONLY data from its inputs or incoming connections, i.e. it must NOT directly grab data from outside sources, like other nodes.



Suppose we have the above example, in which B tries to break this rule by getting data from D. When B's compute() is executing, D is not clean yet. By breaking the black-box principle, B created a cycle without the DG's knowledge, modified the state of data without waiting for the state of the graph itself to update, and finally used invalid data since D has not fully computed yet.

Also, a node must NOT know its place in the DG, e.g. it can't make connections to neighboring nodes (use a separate MEL script, plug-in command, or callbacks to do it for the node).

Custom Locators

A class used to implement a custom locator will be derived from MPxLocatorNode. The methods that are required to be overridden are draw(), isBounded(), and boundingBox(). In this case, compute() is not required.

When a locator node is registered in Maya, MPxNode::kLocatorNode is specified as the node type (fifth) parameter in MFnPlugin::registerNode():

```
plugin.registerNode("myLocator", myLocator::id,
    myLocator::creator, myLocator::initialize,
    MPxNode::kLocatorNode);
```

The draw() method draws the locator in the current viewport, using OpenGL code. It is a lso called when Maya is determining whether an object is in the selection region or where the user has clicked with the mouse.

```
void myLocator::draw(M3dView &view, const MDagPath &path,
                     M3dView::DisplayStyle style,
                     M3dView::DisplayStatus status) {
```

You can assume the node is drawn in local object space, so you don't need to take any transformation information into account. Also, color is set automatically based on the locator's current state (selected, live, dormant, etc.), but it's possible to define a custom color by overriding the color() and colorRGB() methods.

Before any OpenGL drawing code is used, you must call M3dView::beginGL() and push the current OpenGL state:

```
view.beginGL();
glPushAttrib(GL_CURRENT_BIT);
```

Now you can call any OpenGL drawing code, e.g. glVertex3f, etc. You can create variations depending on the drawing style and node status (which are passed into the method).

Once drawing is finished, pop the previous OpenGL state and call M3dView::endGL():

```
glPopAttrib();
view.endGL();
```

If boundingBox() is overridden, isBounded() must return true:

```
bool myLocator::isBounded() const
{
    return true;
}
```

It is highly recommended to implement those two methods. Without them, Maya will have difficulty determining the exact size of the locator, e.g. Frame All and Frame Selected will not zoom correctly. The boundingBox() method must be also efficient since it is called often.

Custom Manipulators

A custom manipulator (or informally, manip) is derived from MPxManipContainer. The methods that are required to be overridden are createChildren() and connectToDependNode(). For custom manips, the draw() method is optional.

A custom manipulator is a node that can contain one or more base manip types, which are:

- freePointTriad (like Maya's Move Tool)
- direction (vector from start point to manip position)
- distance (point on a straight line)
- disc
- circleSweep (point along a circle)
- toggle (2 modes)
- state (more than 2 modes)
- pointOnCurve
- pointOnSurface
- curveSegment (2 points on a curve)
- rotate (like Maya's Rotate Tool)
- scale (like Maya's Scale Tool)

For more details on these types, consult the Maya documentation or the examples in the Developer Kit (e.g. swissArmyManip, etc.).

Custom manips in Maya can be implemented two ways. The first is to attach it to a single custom node, and the second is to attach it to a custom context/tool (so it will work with a variety of nodes). In this document, we will cover only the first approach.

To register a manipulator node in Maya, use MPxNode::kManipContainer. Also, if the manip is attached to a single custom node, then it is VERY important to name it based on the node's, i.e. it must be the node's name followed by "Manip":

```
plugin.registerNode("myLocator", myLocator::id,
myLocator::creator, myLocator::initialize,
MPxNode::kLocatorNode);

plugin.registerNode("myLocatorManip", myLocatorManip::id,
myLocatorManip::creator, myLocatorManip::initialize,
MPxNode::kManipContainer);
```

If you need to define an initialize function for a custom manip, you must call MPxManipContainer::initialize().

```
MStatus myLocatorManip::initialize()
{
    // . . .
    return MPxManipContainer::initialize();
}
```

In the initialize function of the custom node the manip will operate on, call addToManipConnectTable(), passing in the node's type ID:

```
MStatus myLocator::initialize()
{
    // . . .
    MPxManipContainer::addToManipConnectTable(myLocator::id);

    return MS::kSuccess;
}
```

The createChildren() method adds one or more base manip types to the manip container node. MPxManipContainer provides several methods to add a specific manip type, e.g. addFreePointTriadManip(), addDistanceManip(), etc. Also, use the appropriate function set (derived from MFnManip3D) to set a manip's properties, e.g. MFnFreePointTriadManip, MFnDistanceManip, etc.

```
MStatus myLocatorManip::createChildren()
{
    stateManipDp = addStateManip("myStateManip", "mode");
    togManipDp = addToggleManip("myToggleManip", "onY");

    MFnStateManip stateManipFn(stateManipDp);
    stateManipFn.setInitialState(0);
    stateManipFn.setMaxStates(4);

    MFnToggleManip togManipFn(togManipDp);
```

```
    togManipFn.setToggle(true);
    togManipFn.setDirection(MVector(0.0, 0.0, 1.0));

    return MS::kSuccess;
}
```

The connectToDependNode() method is invoked when “Show Manipulator Tool” is used in Maya. It creates the association between a given base manip and the plug of the node it will affect, using the appropriate function set.

```
MStatus myLocatorManip::connectToDependNode(const MObject &node)
{
    MFnDagNode dagFn(node);
    dagFn.getPath(nodeDp);

    MPlug modePlug = dagFn.findPlug("mode");
    MPlug onYPlug = dagFn.findPlug("onY");

    MFnStateManip stateManipFn(stateManipDp);
    stateManipFn.connectToStatePlug(modePlug);

    MFnToggleManip togManipFn(togManipDp);
    togManipFn.connectToTogglePlug(onYPlug);

    finishAddingManips();

    return MPxManipContainer::connectToDependNode(node);
}
```

It is required to call finishAddingManips() and MPxManipContainer::connectToDependNode() at the end of connectToDependNode().

You can override the draw() method only if you need to draw additional objects, as the manips themselves will be drawn by MPxManipContainer::draw(). If you need to override draw(), you must call MPxManipContainer::draw() before doing your own drawing. The rules covered in the draw() method of custom locators also apply to custom manips.

Additional Resources

The Maya documentation contains more details about implementing custom nodes as well as information on writing other types of node, like shading nodes (software and hardware), custom shapes, etc. These are covered in the Maya API Guide section.

For details on API classes and methods, consult the API Class Reference section.

For details on Maya’s built-in nodes, consult the Nodes and Attributes Reference section. You can look up the data type or structure of particular attributes of one of Maya’s nodes, so you can implement attributes on your node that can be connected to Maya’s nodes.

Several examples of custom nodes can be found in the Developer Kit (the devkit folder in the Maya installation directory), ranging from simple DG nodes, locators, and manipulators to more complex node/command combinations.

The following book contains sections on implementing nodes, and is highly recommended to those who are new to the Maya API or those who wish to have a reference on the foundations of programming with the API:

Gould, David A. D. *Complete Maya Programming: An Extensive Guide to MEL and the C++ API*. Morgan Kaufmann Publishers, 2003.