**Introduction:**

Maya’s API is accessible through C++, Python or .NET while the program is scriptable through MEL and Python. The Maya API, unlike scripts, provides low level access to the inner workings of Maya allowing the programmer to create commands, dependency graph nodes and other modifications to Maya’s default functionality. Using the API, especially the C++ version, allows for the fastest execution of commands. This tutorial will focus on the basics of the C++ API, as it is most often used for production plugins and modifications. It is assumed that the reader knows the basics about C++, including virtual functions, class inheritance, stream classes and operator methods.

**Dependency Graph:**

Maya’s graphical information consists of objects called nodes, which exist in the dependency graph. Through the dependency graph connections can be made to and from nodes allowing the flow of information from one part of the scene to another. This is the most basic foundation of Maya, and all API constructions are either dependency graph nodes or commands which affect these nodes. Updates in the dependency graph are handled in a very efficient manner where only the affected nodes are updated, this concept is called dirtying.

For example, if node A (shown below) was updated with a new value node D would be recomputed. This is because the corresponding input of D would be dirtied and require that node to recalculate its value. Nodes B, C and E would be unaffected by this change. If node C were updated, then only node E would need to be updated. Understanding this process is vital, as debugging newly programmed nodes often relies on knowledge of how other nodes are interacting with it.

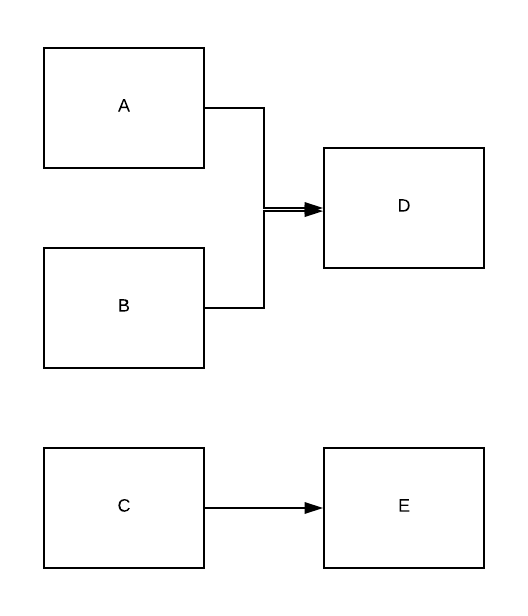


Figure 1 Example Dependency Graph

Inside Maya, the dependency graph is edited and organized through the Node Editor. (located at Window > Node Editor). Each node displays the current connections to and from it, with the top right icon able to show/hide more of the nodes attributes. Nodes can be re-arranged at will and hooked together and disconnected by the user. This is very useful to interact with the inner workings of your scene. Selecting anything in your scene will select its appropriate dependency graph node in the node editor, but be careful since many scene objects have hidden or non-visible nodes that still affect the object. Selecting a node in the node editor also selects it in Maya. Typing tab while no nodes are selected gives the user the option to create a new node from all nodes defined within Maya, we will go over what nodes are available later.

To learn more about the dependency graph and node editor consult these parts of the Maya documentation:

<https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/Maya/files/GUID-23277302-6665-465F-8579-9BC734228F69-htm.html>

<http://help.autodesk.com/view/MAYAUL/2016/CHS/?guid=__files_Dependency_graph_plugins_Dependency_Graph_DG_nodes_htm>

**The MObject:**

MObject is the base class for most classes in the Maya API. This class represents an internal Maya object and allows the API to manipulate it. It has useful methods that output the type of object and if a certain function set is supported. This allows the programmer to readily cast it to the underlying specific object. MObjects cannot be referenced (turned into a pointer). MObjectHandles wrap MObjects and can be used to get added info about the validity of the underlying MObject as well as be referenced.

**Types of Objects in the Maya API:**

**MFn –** function sets, used to operate on an MObject of a particular type

**MPx –** proxies, designed to be inherited and implemented

**Mit –** iterators over MObjects

**M classes –** wrappers on Maya’s internal objects

**MGlobal –** represents global state of Maya, has useful static functions to print to the output window and control the selection of objects in the scene

**Static Enum Attributes:**

In the Maya API, many static attributes of classes are used often. MFn contains static attributes in an enum that represent all function sets, while MFnNumericData contains an enum containing all data representable by MFnNumericAttribute. Be aware of these enumerations and in what ways they can be used, this is a common programming structure in the API.

**MStatus:**

Many functions in the API will have an optional MStatus input argument. It’s advised to create an MStatus object at the initialization of any function that runs such parts of the API and to pass it in whenever requested. Using MStatus will allow you, the programmer, and the eventual end user to see any errors or exceptions your plugin throws. In order to effectively test for errors, running the macro CHECK\_MSTATUS\_AND\_RETURN\_IT on a used MStatus object will display any errors thrown in the Maya GUI. Here is a code example of the MStatus class.

Figure 2 Example of MStatus

// Creating an attribute and showing how the MStatus is parsed

MStatus status;

aBool = attrib.create("test", "test", MFnNumericData::kBoolean, false, &status);

CHECK\_MSTATUS\_AND\_RETURN\_IT(status);

addAttribute(aSwitchBool);

**Matrices in Maya:**

Maya has a lot of support for matrices built right into it. There are pre-built nodes for most matrix operations, including addition/subtraction, multiplication, transpose, decomposing and composing. All Maya matrices are made out of a 4x4 array of double precision floating point numbers, with different indices corresponding to different parts of the transformation. The details of how the Maya transformation matrix is set up can be found here: <http://download.autodesk.com/us/maya/2010help/api/class_m_fn_transform.html>.

Every Maya object in the scene has an internal matrix and world matrix. The matrix describes the transformation of that object from its parent, while the world matrix describes its transformation from the world coordinate system. The world matrix differs from the matrix when the object has parents. When the world matrix is calculated, the different transformations of the object in question are multiplied from the innermost transformation to the outermost. These transformations and more can be performed in the API through MFnTransform and MMatrix.

MFnTransform describes a transformation matrix of a particular object. In particular, it allows the programmer to set the positional attributes of the object like rotation, translation, scale and shear without having to deal with the math. For example, there is a member function called ‘translateBy’ which simply takes in a vector of the X, Y and Z to translate the matrix by and applies the transformation automatically. Keep in mind however that this is a function set and not a wrapper object, in order to receive the wrapper for the actual matrix in question the ‘transformation’ function must be used. Another matrix wrapper is the MMatrix, which gives the programmer access to basic operations that apply to all 4x4 matrices. This wrapper class is useful for when the programmer needs to do a manipulation on the bare matrix itself, for a special transformation in space or in the creation of a deformer node.

**Setting up the Development Environment:**

In order to develop Maya plugins in C++ the Visual Studio environment must be used. The latest community edition of Visual Studio is free to download on Microsoft’s website, make sure the C++ additions are installed. Maya’s API devkit also needs to be installed, either search for your particular Maya’s devkit online for a free download or look at the bottom of this document for the link to the 2018 version. Autodesk provides information on the proper installation of these libraries on their website and in the included readme files.

*<Need to include more information about how to set up the Visual Studio environment>*

Instead of a manual configuration, CMake can be used to automatically configure the Visual Studio environment needed for a new plugin. CMake is an automated and platform independent build tool that can be used to automatically generate IDE specific files allowing the programmer to run their code in the editor/environment of their choice.

**Flow of a Node in the API:**

After the dev environment is set up, we can explore the creation of a new node. In the main class, there should be two functions, initializePlugin and uninitializePlugin. One is called when the plugin is loaded and the other is when it is unloaded from Maya. Initialization of the plugin as a whole takes place here, it needs to be registered with the name of the author, the version number as well as all nodes and commands.

MStatus initializePlugin(MObject obj) {

MStatus status;

// plugin object, name of author, version number, supported Maya versions

MFnPlugin fnPlugin(obj, "Raymond Schade", "1.0", "Any");

status = fnPlugin.registerNode("sampleNode",

SamplePlugin::id,

SamplePlugin::creator,

SamplePlugin::initialize);

CHECK\_MSTATUS\_AND\_RETURN\_IT(status);

return MS::kSuccess;

}

MStatus uninitializePlugin(MObject obj) {

MStatus status;

MFnPlugin fnPlugin(obj);

status = fnPlugin.deregisterNode(SamplePlugin::id);

CHECK\_MSTATUS\_AND\_RETURN\_IT(status);

return MS::kSuccess;

}

Figure 3 Example of Plugin Initialization

Next, another class must be made for the node itself. In order for it to be recognized as a node it must inherit the proxy class MPxNode. This will allow it to override the functions ‘initialize,’ ‘compute,’ ‘creator,’ ‘connectionMade,’ and ‘connectionBroken.’ There are more functions to be implemented from MPxNode visible in the documentation, but the above are a good baseline. An ID for the node must also be specified, between 0x00000000 and 0x0007ffff are reserved for testing usage. Contact Autodesk for a set of unique IDs for production plugin development.

**Setup of a Node:**

The ‘creator’ function is simply responsible for returning a new instance of the node object while the ‘initialize’ method is responsible for hooking up all of the attributes as inputs and outputs as well as telling Maya which inputs affect which outputs. ‘compute’ is called every time this particular instance of the node is dirtied and is in charge of computing the changes brought on by the inputs. ‘connectionMade’ and ‘connectionBroken’ are called by connecting/disconnecting inputs and outputs.

MTypeId TestNode::id(0x0000231);

MObject TestNode::aInput;

MObject TestNode::aOutput;

TestNode:: TestNode () {

}

TestNode::~TestNode () {

}

void\* TestNode::creator() {

return new TestNode();

}

MStatus TestNode::connectionMade(const MPlug &plug, const MPlug &otherPlug, bool asSrc) {

<...>

return MS::kSuccess;

}

MStatus TestNode::connectionBroken(const MPlug &plug, const MPlug &otherPlug, bool asSrc) {

<...>

return MS::kSuccess;

}

MStatus TestNode::compute(const MPlug & plug, MDataBlock & data) {

MStatus status;

<...>

return MS::kSuccess;

}

MStatus TestNode::initialize() {

MStatus status;

<...>

return MS::kSuccess;

}

Figure 4 Example Node Template

**Attributes:**

In order to let data enter our node we must set up the ‘initialize’ function with attributes that control the inputs and outputs of the new node. In order to tell Maya to set up an input or output attribute an appropriate function set must be used. MFnNumericAttribute encompasses most basic types, like Booleans, integers and vectors. MFnMessageAttribute covers the input/output of messages which relate heavily to the callback system within Maya. MFnTypedAttribute encompasses internal Maya types like strings and objects. Other attribute function sets exist and are searchable in the API documentation.

To create a new input, use the appropriate attribute function set and the ‘create’ function. Don’t forget to properly use an MStatus object to catch any errors. After the input has been created set any flags needed on it using the attribute function set. Until the ‘addAttribute’ function is called the current input/output is the thing being edited by the attribute function set. For example, this input is an array which allows the user to connect many of them in the dependency graph.

MFnNumericAttribute attribFnSet;

// create takes the arguments long name, short name, type, where

// all short names must be unique

input = attribFnSet.create("exInput", "exInput", MFnNumericData::kBoolean, &status);

// don't forget the MStatus

CHECK\_MSTATUS\_AND\_RETURN\_IT(status);

attribFnSet.setArray(true);

addAttribute(input);

Figure 5 Example attribute being added

After ‘addAttribute’ is called, it is added to the node. This must be done before any relationships are made. Relationships between node inputs/outputs decide how the node is dirtied, and thus when the compute function is called. Using the ‘attributeAffects’ function an output can be affected by an input.

Figure 6 attributeAffects example

attributeAffects(aAffected, aAffector);

In this example, whenever aAffector experiences an update the ‘compute’ function will be called with the aAffected attribute.

**MPlug and the Compute Function:**

When the ‘compute’ function is called there are two arguments, the MPlug associated with what made the node dirty and the MDataBlock containing information about the node’s current state and connections. The program should check the plug against the desired attribute, if they don’t match then the compute function should call ‘return MS::kUnknownParameter;’ returning an enumeration value representing an unknown parameter to the function. After this check, any computation the node can perform with its inputs can be done. Data can be accessed through the MDataBlock by calling either ‘inputValue’ or ‘outputValue’ and putting in the requested attribute. That will return an MDataHandle which can be unwrapped into the desired data using an appropriate method. For example, if this test attribute contained a float, the following code can be used:

Figure 7 Data retrieval from MDataBlock

float test = data.inputValue(aTestVal, &status).asFloat();

CHECK\_MSTATUS\_AND\_RETURN\_IT(status);

Then the data can be manipulated as the application demands and be put into the appropriate output values.

Figure 8 Example setting a Boolean value to a MDataHandle

MDataHandle testOut = data.outputValue(aTestOut, &status);

CHECK\_MSTATUS\_AND\_RETURN\_IT(status);

testOut.setBool(testBool);

Finally, the data block is set to clean and the ‘compute’ function ends successfully.

Figure 9 Set MDataBlock to clean and return

data.setClean(plug);

return MS::kSuccess;

**On Connect and On Disconnect:**

As mentioned before, when an input is connected or disconnected from a node a special function is called. Like the ‘compute’ method, it gives us access to the plug that triggered it but it unlike that it does not provide an MDataBlock. It intakes the plug of the attribute that is being connected/disconnected on both sides. This data can be used to obtain the node being connected/disconnected, which is very useful in registering/deregistering callback functions.

**Messages and Callbacks:**

Callback functions are called automatically upon a change in state of the target. They can be registered using an object to a node and listen for changes on that node. Callback function registration happens in the MNodeMessage class using static functions and many changes can be listened for. A useful one is the ‘addNodeDirtyCallback’ function, which is fired every time the target node is dirtied. In order to add a callback there must be a node to fire the callback from, a function to execute, and optionally a piece of data that the function has access to. This data, called ‘clientData’ in the API documentation, is useful because the callback functions are not within the scope of the node class, and have no access to their member functions or variables. In order to keep track of these callbacks when one is registered an MCallbackId is returned by the associated add callback function. This unique id represents the callback, and can be used to unregister it using the ‘removeCallback’ function inside MNodeMessage. It’s up to the programmer to keep track of these callback ids, if there is a registered callback that uses data from a node that no longer exists the program will crash when it is called.

**Additional Reference:**

Official Autodesk Intro to Maya API:

<https://help.autodesk.com/view/MAYAUL/2017/ENU/?guid=__files_Maya_API_introduction_htm>

C++ API Reference:

<https://help.autodesk.com/view/MAYAUL/2017/ENU/?guid=__cpp_ref_index_html>

Maya Devkit Download:

https://apps.autodesk.com/MAYA/en/Detail/Index?id=5525491636376351277&os=Win64&appLang=en