API

Version 6

Alias®

ALIAS ■ 210 KING STREET EAST ■ TORONTO, CANADA M5A 1J7

Table of Contents

1 Plug-in API introduction

Developer

Plug-in API

Overview of plug-in API

Maya is an open product, meaning anyone outside of Alias can change existing features or add entirely new features. There are two ways to modify Maya:

- **MEL—**(Maya Embedded Language) is a powerful and easy to learn scripting language. Most common operations can be done using MEL.
- **API—**(Application Programmer Interface) provides better performance than MEL. You can add new objects to Maya using API, and code executes approximately ten times faster than when you perform the same task using MEL.

Note Maya does not provide binary compatibility for plug-ins. With each new release, the source must be recompiled. However, it is our goal to maintain source-code compatibility, and in the majority of cases, no changes to the source will be required. At some point, we may identify a core API that will provide source and binary compatibility going forward.

The Maya API on IRIX, Windows, Linux, and Mac OS X

The Maya API is a C++ API that provides internal access to Maya. The API is packaged as a set of libraries corresponding to different functional areas of Maya. These libraries are:

OpenMaya—Contains fundamental classes for defining nodes and commands and for assembling them into a plug-in.

OpenMayaUI—Contains classes necessary for creating new user interface elements such as manipulators, contexts, and locators.

OpenMayaAnim—Contains classes for animation, including deformers and inverse kinematics.

OpenMayaFX—Contains classes for dynamics.

OpenMayaRender—Contains classes for performing rendering functions.

Loading a Plug-in

There are two ways to load and unload a plug-in. The easiest way to load a plug-in is to use the Plug-in Manager.

To load a plug-in from the Plug-in manager:

1 Choose Window > Settings/Preferences > Plug-in Manager to open the Plug-in Manager window and display the list of all known plug-ins.
2 Find the plug-in you need and either click the loaded or auto load check box to load the plug-in.

The Plug-in Manager uses the `MAYA_PLUG_IN_PATH` environment variable to locate available plug-ins to load.

To load a plug-in from the command line:

Use MEL's `loadPlugin` command.

```
loadPlugin “hello”;
```

This searches `MAYA_PLUG_IN_PATH` looking for a file named `hello.so` on IRIX and Linux platforms, a file named `hello.mll` on Windows platforms, and a file named `hello.lib` on Mac OS X. Once found, it will be loaded into Maya as a plug-in.

Unloading a plug-in

Unloading a plug-in through MEL is simple—you use the `unloadPlugin` command and supply the plug-in name.

Notes
- A plug-in must be unloaded before it is recompiled. Failure to do so causes Maya to crash.
- Before you can unload a plug-in, you must first remove all references to it from the Maya scene. Along with deleting nodes from the scene that are defined in plug-ins, it is also necessary to flush references to deleted nodes and executed commands from the undo queue. Even though the artifacts are no longer in the scene, they are still there for undo purposes.
- If you force the unload of a plug-in while it is in use, it will not be possible to reload node plug-ins. This is because existing nodes in the scene will have to be converted to “Unknown” nodes, and on plug-in reload, you will not be allowed to change the type of these existing nodes.

Writing a simple plug-in

The following shows how to write a simple plug-in called Hello World.

Writing your first plug-in

When learning a new computer language, the first program you are likely to see is a “Hello World” program. Following this tradition, the first plug-in we create is a “Hello World” plug-in, which simply outputs “Hello World” to the window from which Maya was launched.

```
#include <maya/MSimple.h>
DeclareSimpleCommand( helloWorld, "Alias", "6.0");
MStatus helloWorld::doIt( const MArgList& )
{
    printf(“Hello World\n”);
    return MS::kSuccess;
}
```

IRIX:

If this is put into a file called `helloWorld.cpp` it can be compiled with:

```
CC -c -I/usr/aw/maya/include -g -mips3 -n32 \
      -DSGI  helloCmd.cpp
CC -shared -no_transitive_link -g -mips3 -n32 \
      -DSGI  -o helloCmd.so helloCmd.o \
      -L/usr/aw/maya/lib -lOpenMaya
```

Linux:

If this is put into a file called `helloWorld.cpp` it can be compiled with:

```
g++332 -c -I/usr/aw/maya6.0/include \
      -I/usr/X11R6/include -O3 -pipe \
      -D_BOOL -DLINUX  -mcpu=pentium4  \
      -Wno-deprecated -fno-gnu-keywords helloCmd.cpp

g++332 -shared  -O3 -pipe -D_BOOL -DLINUX  \
      -mcpu=pentium4  -Wno-deprecated \
      -fno-gnu-keywords -o helloCmd.so helloCmd.o \
      -L/usr/aw/maya6.0/lib -lOpenMaya
```

Windows and Mac OS X:

If this is put into a file called `helloWorld.cpp`, it can be compiled by following the platform-specific instructions in “Creating your own plug-in build file” in Chapter 10, “Setting up your plug-in build environment”.

Once the file has compiled, you can load it into Maya (see “Loading a Plug-in”). Type “helloWorld” into the command window (press Enter to invoke the command), and “Hello World” displays in the output window.

Important plug-in features

The “Hello World” plug-in introduces a number of important features, outlined below.

MSimple.h

A special header file used for simple command plug-ins. It takes care of all the work necessary to register the new command with Maya through the DeclareSimpleCommand macro. However, you can only create a plug-in that contains a single command.

Notes

- It is quite possible, and even common, to write a plug-in that implements several features, such as dependency graph nodes and commands. For such plug-ins, MSimple.h cannot be used. You must write custom registration code to tell Maya the plug-ins’ capabilities.
- A major limitation of this macro is that you can only create non-undoable commands. The following sections describe this limitation in detail.

MStatus

Indicates the success or failure of a method. Most methods in API classes return a status code through MStatus and the documentation for each method details the possible status codes returned. To avoid possible name space collisions with other status codes, all MStatus values are scoped with “MS”. For example, `MS::kSuccess` is the success status code. The complete list is in MStatus.h.

Note API uses few status codes, but if error logging is enabled in the API through `MGlobal::startErrorLogging()` additional detailed error messages are output to the error log file when a method returns something other than `MS::kSuccess`. This is described in more detail in “Error checking”, “MStatus class”, and “Error logging” in this chapter.

DeclareSimpleCommand

The DeclareSimpleCommand macro requires three parameters: the name of a class that will be used to implement the command, the name of the vendor (or author) of the command, and the command’s version number.

As in MSimple.h, the DeclareSimpleCommand() macro saves you from writing the registration code necessary for Maya to properly recognize your file as a plug-in. To keep it simple, though, you cannot specify an undo method for the command, so you cannot create truly undoable commands using this macro.

Note Commands that do not support undo must not change the state of the scene in any way. They can be used to query different aspects of the scene, but not to change anything. If an non-undoable command does in fact change anything, Maya's undo capability will break.

Writing a plug-in that interacts with Maya

There are only a few changes between this plug-in and the Hello World plug-in (see "Writing your first plug-in". Since the "helloWorld" plug-in always prints the same thing, you may want to write a plug-in that interacts with Maya. (One way is through command line arguments in MEL).

The following is another simple program which prints "Hello" followed by its input.

```
#include <maya/MSimple.h>
DeclareSimpleCommand( hello, "Alias", "6.0");
MStatus hello::doIt( const MArgList& args )
{
    printf("Hello %s\n", args.asString( 0 ).asChar() );
    return MS::kSuccess;
}
```

This is put in a file called `hello.cpp` and compiled. Once loaded, typing "hello neighbor" into the command window outputs "Hello neighbor".

MArgList

The MArgList class provides functionality similar to the argc/argv parameters of the entry point of a C or C++ program, which provides a list of arguments to your function. The class provides methods to retrieve the arguments as various types, such as including an integer, a double, a string, or a vector.

In the next example, the helix command is defined to allow several arguments to be passed in via the MArgList object. The two arguments are pitch and radius.

```
#include <math.h>

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

```
#include <maya/MFnNurbsCurve.h>
#include <maya/MPointArray.h>
#include <maya/MDoubleArray.h>
#include <maya/MPoint.h>

DeclareSimpleCommand( helix, "Alias - Example", "3.0");

MStatus helix::doIt( const MArgList& args )
{
        MStatus stat;

        const unsigneddeg = 3;               // Curve Degree
        const unsignedncvs = 20;             // Number of CVs
        const unsignedspans = ncvs - deg;// Number of spans
        const unsignednknots= spans+2*deg-1;// Number of knots
        double  radius                = 4.0;                // Helix radius
        double  pitch                 = 0.5;                // Helix pitch
        unsignedi;

        // Parse the arguments.
        for ( i = 0; i < args.length(); i++ )
                if ( MString( "-p" ) == args.asString( i, &stat )
                                && MS::kSuccess == stat)
                {
                        double tmp = args.asDouble( ++i, &stat );
                        if ( MS::kSuccess == stat )
                                pitch = tmp;
                }
                else if ( MString( "-r" ) == args.asString( i, &stat )
                                && MS::kSuccess == stat)
                {
                        double tmp = args.asDouble( ++i, &stat );
                        if ( MS::kSuccess == stat )
                                radius = tmp;
                }

        MPointArray controlVertices;
        MDoubleArray knotSequences;

        // Set up cvs and knots for the helix
        //
        for (i = 0; i < ncvs; i++)
                controlVertices.append( MPoint( radius * cos( (double)i ),
                        pitch * (double)i, radius * sin( (double)i ) ) );

        for (i = 0; i < nknots; i++)
                knotSequences.append( (double)i );

        // Now create the curve
        //
        MFnNurbsCurve curveFn;

        MObject curve = curveFn.create( controlVertices,
                                                                  knotSequences,
deg,

MFnNurbsCurve::kOpen,
                                                                  false, false,

MObject::kNullObj,
                                                                  &stat );

        if ( MS::kSuccess != stat )
               printf("Error creating curve.\n");

        return stat;
}
```

Tip One important difference between using argc/argv and an MArgList is that the zeroth element of an MArgList is the first argument to the command and not the command name, as in a C or C++ program.

Building a curve using a plug-in

The following is a plug-in that builds a helical curve. See “Appendix A: NURBS Geometry” for a brief explanation of NURBS geometry.

```
#include <math.h>
#include <maya/MSimple.h>
#include <maya/MPoint.h>
#include <maya/MPointArray.h>
#include <maya/MDoubleArray.h>
#include <maya/MFnNurbsCurve.h>

DeclareSimpleCommand( doHelix, "Alias - Example", "6.0");

MStatus doHelix::doIt( const MArgList& )
{
    MStatus stat;

    const unsigned     deg       = 3;                 // Curve Degree
    const unsigned     ncvs      = 20;                // Number of CVs
    const unsigned     spans     = ncvs - deg;        // Number of spans
    const unsigned     nknots    = spans+2*deg-1;     // Number of knots
    double     radius            = 4.0;               // Helix radius
    double     pitch             = 0.5;               // Helix pitch
    unsigned     i;
    MPointArray     controlVertices;
    MDoubleArray    knotSequences;

    // Set up cvs and knots for the helix
    //
    for (i = 0; i < ncvs; i++)
        controlVertices.append( MPoint( radius * cos( (double)i ),
            pitch * (double)i, radius * sin( (double)i ) ) );

    for (i = 0; i < nknots; i++)
        knotSequences.append( (double)i );

    // Now create the curve
    //
    MFnNurbsCurve curveFn;

    MObject curve = curveFn.create( controlVertices,
        knotSequences, deg, MFnNurbsCurve::kOpen, false, false,
MObject::kNullObj, &stat );

    if ( MS::kSuccess != stat )
        printf("Error creating curve.\n");

    return stat;
}
```

Compile this plug-in, load it, then enter "doHelix" at the prompt. A helical curve displays in the Maya views.

Interacting with Maya

Maya's API contains four types of C++ objects which you use in the code to interact with Maya. These are *wrappers*, *objects*, *function sets*, and *proxies*.

Object ownership in the API

The combination of an object and a function set is similar to a wrapper, but the distinction is necessary to simplify ownership. Object ownership in the API is important. If not properly defined, you may delete something the system needs, or use something the system has just deleted. API wrappers, objects, and function sets remove any question of ownership. Therefore, the potential for using an object at an inconvenient time, like right after the system has deleted it, is removed.

MObject

Access to all Maya objects (curves, surfaces, DAG nodes, dependency graph nodes, lights, shaders, textures, etc.) is done through a handle object called an *MObject*. This handle provides a few simple methods to help determine the object type (see the MObject class documentation for a complete description). The MObject destructor does not delete the Maya object that it references—calling the destructor only deletes the handle object, thereby maintaining ownership.

Important Tip You should never keep a pointer to an MObject between "runs" of the plug-in.

Instead you may want to use an MObjectHandle since this object contains information on the validity of the MObject.

Wrappers

Wrappers exist for simple objects such as math classes (like vectors or matrices). They are generally fully implemented C++ classes with public constructors and destructors. API methods may return a wrapper, which you are then responsible for—leaving scope will usually be adequate for deleting the wrapper. You are also free to allocate and deallocate them as necessary. In the previous example ("Building a curve using a plug-in"), MPointArray and MDoubleArray are wrappers. You always own the wrapper that you reference.

Objects and Function Sets

Objects and function sets are always used together. They are separate which easily establishes ownership—objects are always owned by Maya, and function sets are always owned by you.

Function sets

Function sets are C++ classes which operate on objects. In the example, Building a curve using a plug-in, MFnNurbsCurve is a function set (the MFn prefix indicates this).

These two lines create a new curve.

```
MFnNurbsCurve curveFn;
MObject curve = curveFn.create( ... );
```

- `MFnNurbsCurve curveFn;` creates a new curve function set which contains methods for operating on curve objects, and the create method in particular is used to create new curves.
- `MObject curve = curveFn.create( ... );`creates a new Maya curve object which you can then use however you like.

If you add a third line:

```
curve = curveFn.create( ... );
```

a second curve is created and the curve MObject now references the new curve. The first curve still exists—it simply is no longer referenced by the MObject handle.

Proxies

The Maya API uses proxy objects to create new types of Maya objects. Proxies are objects that you create but Maya owns.

A common misunderstanding is that you can create new types of objects by deriving from an existing function set. For example, MFnNurbsSurface derives from MFnDagNode. You might conclude that if you derive MySurface from MFnDagNode and provide all the methods to operate on a special surface type, you would have added a new surface type to Maya. Unfortunately this doesn't work. What you would have is simply a new function set which operates on existing objects using the new methods. Remember that function sets are entirely owned by you. Maya never sees or uses them; Maya only uses the objects underlying the MObject handle.

Typelessness

An interesting consequence of the separation of objects and function sets is that the API can operate in a typeless manner. For example:

```
MFnNurbsCurve curveFn;
MObject curve = curveFn.create( ... );
MFnNurbsSurface surface( curve );
```

This code creates a curve and passes it to a surface function set on which to operate. Since MFnNurbsSurface only operates on surface objects, the above example does not do anything, but you may not know it. The error checking code in the API checks for such erroneous initializations.

Function sets accept MObjects of any type and if they do not recognize them, ignores them and returns error values whenever you try to operate on them. See “Error checking”, “MStatus class”, and “Error logging” in this chapter.

Naming Conventions

The Maya API uses a convention of prefixes on its classes to distinguish the various types of C++ objects that it uses.

MFn

Any class with this prefix is a function set used to operate on MObjects of a particular type.

MIt

These classes are iterators and work on MObjects much the way a function set does. For example, MItCurveCV is used to operate on an individual NURBS curve CV (there is no MFnNurbsCurveCV), or iteratively, on all the CVs of a curve.

MPx

Classes with this prefix are all “Proxies”, API classes designed for you to derive from and create your own object types.

M classes

Most of these classes are “Wrappers”, though there are others. For example, “Function sets” is a handle on Maya’s internal objects, and MGlobal is a class of static methods that operate globally and do not require an MObject on which to operate. (See Chapter 2, “Selecting with the API” for information on MGlobal.)

Adding arguments

The helix plug-in generates a simple curve, but it always produces the same curve (see “Building a curve using a plug-in”).

Adding arguments to the curve example

You can make a few changes so you can specify the radius and pitch of the curve. Change the function definition by adding the **args** parameter name:

```
MStatus doHelix::doIt( const MArgList& args )
```

Add the following lines after the variable declarations:

```
// Parse the arguments.
for ( i = 0; i < args.length(); i++ )
    if ( MString( “-p” ) == args.asString( i ) )
        pitch = args.asDouble( ++i );
    else if ( MString( “-r” ) == args.asString( i ) )
        radius = args.asDouble( ++i );
```

This code fragment reads arguments so you can change the pitch and radius of the helix you generate. This change is quite simple:

The for-loop walks through all arguments in the MArgList wrapper. The two if-statements convert the current argument (referenced by the index variable “i”) to an MString (the Maya API’s string wrapper class) and compare them with the two possible argument flags.

If there is a match, the next argument is converted to a double and assigned to the appropriate variable. For example:

```
doHelix -p 0.5 -r 5
```

produces a helix with a pitch of 0.5 units and a radius of 5 units.

Error checking

In the examples presented so far you have not been prompted to do much error checking. This is usually fine for examples, but when producing a production plug-in you really want to check for errors.

Most methods take an optional final argument, which is a pointer to an “MStatus” variable into which the status return value is put.

If you replace the argument parsing code with the following fragment in the helix example, the example will be checking for, and handling, most possible errors.

```
// Parse the arguments.
for ( i = 0; i < args.length(); i++ )
    if ( MString( “-p” ) == args.asString( i, &stat )
         && MS::kSuccess == stat )
    {
         double tmp = args.asDouble( ++i, &stat );
         // argument can be retrieved as a double
         if ( MS::kSuccess == stat )
              pitch = tmp;
    }
    else if ( MString( “-r” ) == args.asString( i, &stat )
         && MS::kSuccess == stat )
    {
         double tmp = args.asDouble( ++i, &stat );
         // argument can be retrieved as a double
         if ( MS::kSuccess == stat )
              radius = tmp;
    }
```

The addition of `&stat` to the asString() and asDouble() methods allows you to check if the casting operation succeeds.

For example, `args.asString(i, &stat)` could return `MS::kFailure` if the index is greater than the number of arguments,

or

`args.asDouble(++i, &stat)` could fail if the argument could not be converted to a double.

MStatus class

The MStatus class can determine if a method has failed.

Many API methods either return an instance of the MStatus class, or fill in an instance passed as an optional parameter. The MStatus class contains the methods *error*, and an overloaded operator *bool*, both of which return false if the instance is holding an error status. This means you can check the success of a call quickly, for example:

```
MStatus stat = MGlobal::clearSelectionList;
    if (!stat) {
        // Do error handling
    ...
    }
```

If the MStatus instance contains an error, you can do one of several things:

- Use the **statusCode** method to retrieve an element of the MStatusCode enum that indicates the reason for the failure.
- Use the **errorString** method to retrieve an MString containing a detailed description of the error.
- Use the **perror** method to print the detailed description of the error to standard error, optionally pre-pended by a string you provide.
- Use the overloaded **equality** and **inequality** operators to compare the instance to a specific MStatusCode.
- Reset the instance to the successful state with the **clear** method.

Error logging

As well as using the MStatus class, you can check for failures in API methods using *error logging*.

To enable and disable error logging:

1 From MEL, use the **-errlog** flag of the openMayaPref command.

or

2 From the plug-in, use the methods `MGlobal::startErrorLogging()` and `MGlobal::stopErrorLogging()`.

Once you enable error logging, Maya creates a log file and each time an API method fails, Maya writes a description of the error to the file along with a mini stack trace that shows where the call to the routine was made.

The default name of this file is *OpenMayaErrorLog* located in the current directory. This can be changed, however, by calling:

`MGlobal::setErrorLogPathName.`

Tip Plug-ins can also use the method, `MGlobal::doErrorLogEntry()` to add their own messages to the error log.

2 Selecting with the API

Developer

Plug-in API

Selecting with the API

Overview of selecting with the API

A command usually gets input from the selection list. The result of the MGlobal::getActiveSelectionList() method contains all selected objects and can easily be checked through **“MSelectionList”** and **“MItSelectionList”**—two API classes you can use to edit selection lists.

MGlobal::setActiveSelectionList()

The global active selection list can be copied through:

```
MGlobal::getActiveSelectionList()
```

This returns an MSelectionList and makes a copy of the list.

Any changes you might make through MSelectionList methods will not affect the global list unless you use:

```
MGlobal::setActiveSelectionList()
```

You can also create your own lists using MSelectionList to merge with other lists, including the global list. You can use this list to create sets of objects (see “setObject() method”).

MSelectionList

MSelectionList provides you with methods to add and remove objects from the selection list, as well as walk through the objects on the list.

For example, the following simple plug-in prints the names of all selected DAG nodes. If you create geometry and then select it, this plug-in prints the name of each selected object.

Simple plug-in example

```
#include <maya/MSimple.h>
#include <maya/MGlobal.h>
#include <maya/MString.h>
#include <maya/MDagPath.h>
#include <maya/MFnDagNode.h>
#include <maya/MSelectionList.h>

MStatus pickExample::doIt( const MArgList& )
```

```
{
    MDagPath            node;
    MObject             component;
    MSelectionList      list;
    MFnDagNode          nodeFn;

    MGlobal::getActiveSelectionList( list );
    for ( unsigned int index = 0; index < list.length();
index++ )
    {
        list.getDagPath( index, node, component );
        nodeFn.setObject( node );
        printf("%s is selected\n", nodeFn.name().asChar() );
    }

    return MS::kSuccess;
}
DeclareSimpleCommand( pickExample, "Alias", "1.0" );
```

Walking through the list of selected objects is quite simple. If you create geometry and then select it, this plug-in prints the name of each selected object.

The setObject() method on MFnDagNode is inherited by all function sets from MFnBase and is used to set the object the function set will operate on. Usually this is done through the function set's constructor, but if the function set has already been created and you want to change the object the function set operates on, you use setObject(). This is far more efficient than creating and destroying function sets each time you need one.

Try selecting a few CVs and then calling this plug-in. Notice that you do not get the name of the CV output, but rather the name of the parent object (the curve, surface, or mesh). Also notice that the number of objects selected is not the same as the number of names printed.

The Maya selection architecture simplifies the selection of object components such as CVs. Rather than putting each component object (for instance, each CV) onto the selection list, the parent object is put on the list and the components are grouped together.

For example, if several CVs of nurbSphereShape1 were selected, the list.getDagPath() call above would return an MDagPath to nurbSphereShape1 and an MObject grouping all the selected CVs together. The MDagPath and the MObject could then be passed to an MItSurfaceCV iterator to examine the selected CVs.

As long as you continue to select components of only one object, the object appears on the selection list only once. However, if you pick some components of one object, and then some components of another object, and then more components of the first object, the first object would actually appear on the selection list twice. This is so that you can determine the order in which objects were selected. Within each component MObject the individual components are listed in the order they were selected.

MItSelectionList

MltSelectionList is a wrapper class containing selected objects. This can either be a copy of the global active selection list, or a list you build yourself.

MItSelectionList lets you filter the objects on the selection list to only see objects of a particular type (MSelectionList does not let you filter selected objects).

```
MGlobal::getActiveSelectionList( list );
for ( MItSelectionList listIter( list ); !listIter.isDone();
listIter.next() )
{
    listIter.getDagPath( node, component );
    nodeFn.setObject( node );
    printf("%s is selected\n", nodeFn.name().asChar() );
}
```

The “MSelectionList” example can be changed with this code fragment to use MItSelectionList instead to iterate through the selection list. This produces exactly the same results as before when selecting objects.

You can easily change the code to only look for objects of a particular type. For example, changing the constructor of the selection list iterator to:

```
MItSelectionList listIter( list, MFn::kNurbsSurface )
```

causes the loop to only iterate across selected NURB surfaces—it also ignores surface CVs. If, however, you wanted to just iterate across selected surface CVs, you would change the iterator to:

```
MItSelectionList listIter( list, MFn::kSurfaceCVComponent )
```

which would only iterate across surfaces with selected CVs.

setObject() method

The setObject() method on MFnDagNode is inherited by all function sets from MFnBase. It sets the object on which the function set operates. This is usually done through the function set’s constructor, but if the function set already exists and you want to change the object, use setObject(). This is more efficient than creating and destroying function sets each time you need one.

Example

Try selecting a few CVs then calling this plug-in. Instead of the name of the CV output, you get the name of the parent object (the curve, surface, or mesh). You may also notice the number of selected objects is not the same as the number of printed names.

The Maya selection architecture simplifies the selection of object components such as CVs. Instead of putting each component object (for instance, each CV) onto the selection list, the parent object is put on the list and the components are grouped together.

So, if several CVs of nurbSphereShape1 are selected, the `list.getDagPath()`call in the "Simple plug-in example" returns an MDagPath to nurbSphereShape1, and an MObject groups all selected CVs. The MDagPath and the MObject can then be passed to an MItSurfaceCV iterator to examine the selected CVs.

As long as you continue to select components of only one object, the object appears on the selection list only once. If you pick components of one object, components of another object, then more components of the first object, the first object appears on the selection list twice. This is so you can determine the order in which objects were selected. Within each component MObject, the individual components are listed in the order they were selected.

Limitation—Mesh vertices, faces, or edges are not returned in the order selected.

MFn::Type enumeration

The MFn::Type enumeration is used throughout the API to indicate item types.

- The "Function sets" class has an apiType() method which you can use to determine the type of object the MObject is referencing. Each function set also has a type() method which can be used to determine the function set type.
- The MGlobal::getFunctionSetList() also returns an array of strings representing the types of function sets that will accept an object.

See also "Objects and Function Sets" in Chapter 1.

MGlobal::selectByName()

The add() method on MSelectionList combined with "MGlobal::setActiveSelectionList()" provides one method for a plug-in to modify the active selection list.

Another way is to use `MGlobal::selectByName()`. This finds all objects matching a pattern and adds them to the active selection list. For example:

```
MGlobal::selectByName( “*Sphere*” );
```

selects everything with Sphere in the name.

Tip You can also use MGlobal::select() to add an object to the global selection list without creating an MSelectionList first.

3 Command plug-ins

Developer Plug-in API

Add commands to Maya

Overview of adding commands to Maya

Plug-ins

The API supports several different types of plug-ins:

- Command plug-ins—commands that extend the MEL scripting language.
- Tool commands—plug-ins that take mouse input.
- “Dependency graph plug-ins”—plug-ins that add new operations, such as dependency graph nodes.
- Device plug-ins—plug-ins that allow new devices to interact with Maya.

Command plug-ins

This chapter describes the following command plug-in topics:

- ”MFnPlugin” on page 31
- ”initializePlugin()” on page 32
- ”uninitializePlugin()” on page 33
- ”MPxCommand” on page 34
- ”MPxContext” on page 44
- ”MPxContextCommand” on page 49
- ”MPxToolCommand” on page 51

Registering commands

Before you can write more complex commands, you have to know how to properly register them with Maya. The MFnPlugin is a class used for registering the command with Maya.

MFnPlugin

The following is a new version of the hello command which uses MFnPlugin to register itself instead of using the macros in “MSimple.h”.

```
#include <stdio.h>
#include <maya/MString.h>
#include <maya/MArgList.h>
#include <maya/MFnPlugin.h>
#include <maya/MPxCommand.h>

class hello : public MPxCommand
{
public:
    MStatus        doIt( const MArgList& args );
    static void*   creator();
};

MStatus hello::doIt( const MArgList& args ) {
    printf("Hello %s\n", args.asString( 0 ).asChar() );
    return MS::kSuccess;
}

void* hello::creator() {
    return new hello;
}

MStatus initializePlugin( MObject obj ) {
    MFnPlugin plugin( obj, "Alias", "1.0", "Any" );
    plugin.registerCommand( "hello", hello::creator );
    return MS::kSuccess;
}

MStatus uninitializePlugin( MObject obj ) {
    MFnPlugin plugin( obj );
    plugin.deregisterCommand( "hello" );

    return MS::kSuccess;
}
```

Note! initializePlugin() and uninitializePlugin() must be present in all plug-ins. If both or either is absent the plug-in will not be loaded and the creator is necessary to allow Maya to create instances of the class. See the following for details.

initializePlugin()

The initializePlugin() function can be defined as either a C or C++ function. **If you do not define this function, the plug-in will not be loaded**.

initializePlugin() contains the code to register any commands, tools, devices, and so on, defined by the plug-in with Maya. It is called only once—immediately after the plug-in is loaded.

For example, commands and tools are registered by instantiating an MFnPlugin function set to operate on the MObject passed in. This MObject contains Maya private information such as the name of the plug-in file and the directory it was loaded from. It is passed in to the MFnPlugin constructor, along with the vendor name which defaults to "Unknown" if not specified, the version number of the plug-in as a string which defaults to "1.0", and the version of the API required for the plug-in to operate properly, which defaults to "Any".

Once constructed, the MFnPlugin function set is used to register the contents of the plug-in file. In the example ("MFnPlugin"), the MFnPlugin::registerCommand() is called to register the "hello" command, along with the creator (see "Creator methods") for the command. Once done, the function returns a status code indicating whether or not it succeeded. After an unsuccessful initialization, the plug-in is unloaded automatically.

uninitializePlugin()

uninitializePlugin(), like initializePlugin(), can be a C or C++ function. If you neglect to declare this function, your plug-in will not be loaded.

The uninitializePlugin() function contains the code necessary to de-register from Maya whatever was registered through initializePlugin(). It is called once only—when the plug-in is unloaded.

This function should be used for a few quick clean-up operations, such as closing files. It is not necessary for you to delete those commands, or nodes created by your plug-in when it exits since Maya takes care of them. You should therefore *not* be keeping a list of the Maya objects allocated by your plug-in nor freeing them when uninitializePlugin() is called.

Creator methods

The items, such as commands, tools, or devices, registered by a plug-in are not available when Maya is compiled. As a result , Maya has no way of determining the size of a new command object (or any other plug-in defined C++ object). This makes it impossible for Maya to allocate these objects without some help.

The creator methods on API objects provide a mechanism for Maya to allocate an object of unknown characteristics. When you register a new object, you are actually registering its creator method which Maya can then call to allocate a new instance of an object.

MPxCommand

The new hello command introduced earlier uses a proxy object to add new functionality to Maya (see “MFnPlugin”). This proxy object is derived from MPxCommand which provides all the functionality necessary for Maya to use the command as if it were built in.

A minimum of two methods must be defined. These are the *doIt()* method and the *creator*.

```
class hello : public MPxCommand
{
public:
    MStatus         doIt( const MArgList& args );
    static void*    creator();
};
```

doIt() and redoIt() methods

The doIt() method is a pure virtual method, and since there is no creator defined in the base class, you must define both doIt() and creator.

For simple commands, the doIt() method performs the actions of the command. In more complex commands, the doIt() method parses the argument list, the selection list, and whatever else may be necessary. It then uses this information to set data internal to the command before calling the redoIt() method, which does the bulk of the work. This avoids code duplication between the doIt() and redoIt() methods.

Verifying when methods are called

The following is a simple plug-in that outputs a string when any of its methods are called by Maya. You can use it to see when the methods are called.

```
#include <stdio.h>
#include <maya/MString.h>
#include <maya/MArgList.h>
#include <maya/MFnPlugin.h>
#include <maya/MPxCommand.h>

class commandExample : public MPxCommand
{
public:
                    commandExample();
    virtual         ~commandExample();
    MStatus         doIt( const MArgList& );
    MStatus         redoIt();
    MStatus         undoIt();
    bool            isUndoable() const;
    static void*    creator();
};

commandExample::commandExample() {
    printf("In commandExample::commandExample()\n");
}
commandExample::~commandExample() {
    printf("In commandExample::~commandExample()\n");
}
MStatus commandExample::doIt( const MArgList& ) {
    printf("In commandExample::doIt()\n");
    return MS::kSuccess;
}
MStatus commandExample::redoIt() {
    printf("In commandExample::redoIt()\n");
    return MS::kSuccess;
}
MStatus commandExample::undoIt() {
    printf("In commandExample::undoIt()\n");
    return MS::kSuccess;
}
bool commandExample::isUndoable() const {
    printf("In commandExample::isUndoable()\n");
    return true;
}
void* commandExample::creator() {
    printf("In commandExample::creator()\n");
    return new commandExample();
}

MStatus initializePlugin( MObject obj )
{
    MFnPlugin plugin( obj, "My plug-in", "1.0", "Any" );
    plugin.registerCommand( "commandExample",
commandExample::creator );
    printf("In initializePlugin()\n");
    return MS::kSuccess;
}

MStatus uninitializePlugin( MObject obj )
{
    MFnPlugin plugin( obj );
    plugin.deregisterCommand( "commandExample" );
    printf("In uninitializePlugin()\n");
    return MS::kSuccess;
}
```

When you first load this plug-in, notice that “In initializePlugin()” is printed immediately. If you then type “commandExample” in the command window you will see:

```
In commandExample::creator()
In commandExample::commandExample()
In commandExample::doIt()
In commandExample::isUndoable()
```

Note that the destructor is not called. This is because the command object remains indefinitely so that it can be undone, or redone (after being undone).

This is how Maya's undo mechanism works. Command objects maintain information which allows them to undo themselves when necessary. The destructor is called when the command falls off the end of the undo queue, it is undone and not redone, or the plug-in is unloaded.

If you now use Edit > Undo (or you use the MEL undo command) and Edit > Redo (or you use the MEL redo command), the undoIt() and redoIt() methods of the command get called when these menu items are invoked.

If you modify this example so that the isUndoable() method returns false rather than true (remember to unload the plug-in before recompiling) when you run it, the output becomes:

```
In commandExample::creator()
In commandExample::commandExample()
In commandExample::doIt()
In commandExample::isUndoable()
In commandExample::~commandExample()
```

In this case the destructor is called immediately since the command cannot be undone. Maya treats a non-undoable command as an *action* that does not affect the scene in any way. This means that no information needs to be saved after the command executes, and when undoing and redoing commands, it does not need to be executed since it does not change anything.

Helix example with undo and redo

The following example is another implementation of the helix plug-in. This version is implemented as a full command with undo and redo. It works by taking a selected curve and turning it into a helix.

```
#include <stdio.h>
#include <math.h>

#include <maya/MFnPlugin.h>
#include <maya/MFnNurbsCurve.h>
#include <maya/MPointArray.h>
#include <maya/MDoubleArray.h>
#include <maya/MPoint.h>
#include <maya/MSelectionList.h>
#include <maya/MItSelectionList.h>
#include <maya/MItCurveCV.h>
```

```
#include <maya/MGlobal.h>
#include <maya/MDagPath.h>
#include <maya/MString.h>
#include <maya/MPxCommand.h>
#include <maya/MArgList.h>

class helix2 : public MPxCommand {
public:
                   helix2();
    virtual        ~helix2();
    MStatus        doIt( const MArgList& );
    MStatus        redoIt();
    MStatus        undoIt();
    bool           isUndoable() const;
    static         void* creator();
```

The command starts out as the previous example, declaring the methods it will be defining.

```
private:
    MDagPath        fDagPath;
    MPointArray     fCVs;
    double          radius;
    double          pitch;
};
```

This command will be modifying the model. So that it will be able to undo the changes it makes, it allocates space to store the original definition of the curve. It also stores the description of the helix so that it can reproduce it if the redoIt method is called.

It is important to notice that the command does not store a pointer to an MObject, but rather uses an MDagPath to reference the curve for undo and redo. An MObject is not guaranteed to be valid the next time your command is executed. As a result, if you had used an MObject, Maya would likely core dump when performing your undoIt() or redoIt(). An MDagPath however, being simply a description of the path to the curve, is guaranteed to be correct whenever your command is executed.

```
void* helix2::creator() {
    return new helix2;
}
```

The creator simply returns an instance of the object.

```
helix2::helix2() : radius( 4.0 ), pitch( 0.5 ) {}
```

The constructor initializes the radius and pitch.

```
helix2::~helix2() {}
```

The destructor does not need to do anything since the private data will be cleaned up automatically.

Note Data owned by Maya should not be deleted.

```
MStatus helix2::doIt( const MArgList& args ) {
    MStatus status;

    // Parse the arguments.
    for ( int i = 0; i < args.length(); i++ )
        if ( MString( "-p" ) == args.asString( i, &status )
            && MS::kSuccess == status )
        {
            double tmp = args.asDouble( ++i, &status );
            if ( MS::kSuccess == status )
                pitch = tmp;
        }
        else if ( MString( "-r" ) == args.asString( i,
&status )
            && MS::kSuccess == status )
        {
            double tmp = args.asDouble( ++i, &status );
            if ( MS::kSuccess == status )
                radius = tmp;
        }
        else
        {
            MString msg = "Invalid flag: ";
            msg += args.asString( i );
            displayError( msg );
            return MS::kFailure;
        }
```

As before, this simply parses the arguments passed into the doIt() method and uses them to set the internal radius and pitch fields which will be used by the redoIt() method. The doIt() method is the only one that receives arguments. The undoIt() and redoIt() methods each take their data from internal data of the command itself.

In the final else-clause, the displayError() method, inherited from MPxCommand, outputs the message in the command window and in the command output window. Messages output with displayError() are prefixed with "Error:". Another option is displayWarning() which would prefix the message with "Warning:".

```
    // Get the first selected curve from the selection list.
    MSelectionList slist;
    MGlobal::getActiveSelectionList( slist );
    MItSelectionList list( slist, MFn::kNurbsCurve, &status
);
    if (MS::kSuccess != status) {
        displayError( "Could not create selection list 
iterator" );
        return status;
    }

    if (list.isDone()) {
        displayError( "No curve selected" );
        return MS::kFailure;
    }

    MObject component;
    list.getDagPath( fDagPath, component );
```

This code gets the first curve object off the selection list. The fDagPath field of the command is set to the selected object selection is covered in detail in Chapter 2, "Selecting with the API").

```
    return redoIt();
}
```

Once the internal data of the command is set , the redoIt() method is called. The doIt() method could perform the necessary actions itself, but these actions are always identical to those performed by redoIt() so, having doIt() call redoIt() reduces code duplication.

You might wonder why doIt() calls redoIt() and not the other way around. Although this is possible—the redoIt() method could take the cached data and turn it into an MArgList which it could then pass to doIt()—it would be far less efficient.

```
MStatus helix2::redoIt()
{
    unsigned            i, numCVs;
    MStatus             status;
    MFnNurbsCurve       curveFn( fDagPath );

    numCVs = curveFn.numCVs();
    status = curveFn.getCVs( fCVs );
    if ( MS::kSuccess != status )
    {
        displayError( "Could not get curve's CVs" );
        return MS::kFailure;
    }
```

This code gets the CVs from the selected curve and stores them in the command's internal MPointArray. These stored CV positions could then be used if the undoIt() method is called to return the curve to its original shape.

```
    MPointArray        points(fCVs);
    for (i = 0; i < numCVs; i++)
        points[i] = MPoint( radius * cos( (double)i ),
            pitch * (double)i, radius * sin( (double)i ) );
    status = curveFn.setCVs( points );
    if ( MS::kSuccess != status )
    {
        displayError( "Could not set new CV information" );
        fCVs.clear();
        return status;
    }
```

As with the earlier helix examples, this code sets the position of the curve's CVs so that the curve forms a helix.

```
    status = curveFn.updateCurve();
    if ( MS::kSuccess != status )
    {
        displayError( "Could not update curve" );
        return status;
    }
```

The updateCurve() method is used to inform Maya that the geometry of the curve has changed. Failing to call this method after modifying geometry causes the display of the object to remain unchanged.

```
    return MS::kSuccess;
}
```

Returning MS::kSuccess at the completion of a function indicates to Maya that the operation completed successfully.

```
MStatus helix2::undoIt()
{
    MStatus        status;

    MFnNurbsCurve  curveFn( fDagPath );
    status = curveFn.setCVs( fCVs );
    if ( MS::kSuccess != status)
    {
        displayError( "Could not set old CV information" );
        return status;
    }
```

These few lines take the stored CV positions (the original positions of the curve's CVs) and resets them.

Note You needn't worry about the number of CVs changing, or the curve being deleted in an undo function. You assume that anything done after your command has been undone before your undoIt() is called. As a result the model is in the same state as it was immediately after your command finished.

```
    status = curveFn.updateCurve();
    if ( MS::kSuccess != status )
    {
        displayError( “Could not update curve” );
        return status;
    }

    fCVs.clear();
    return MS::kSuccess;
}
```

The MPointArray is cleared here just as a precaution.

```
bool helix2::isUndoable() const
{
    return true;
}
```

This command is undoable. It modified the model, but an undoIt() method has been provided which returns the model to the state it was in before the command was run.

```
MStatus initializePlugin( MObject obj )
{
    MFnPlugin plugin( obj, “Alias”, “1.0”, “Any”);
    plugin.registerCommand( “helix2”, helix2::creator );

    return MS::kSuccess;
}

MStatus uninitializePlugin( MObject obj )
{
    MFnPlugin plugin( obj );
    plugin.deregisterCommand( “helix2” );

    return MS::kSuccess;
}
```

The plug-in is completed with the usual initialize and uninitialize functions.

Returning results to MEL

Commands can also return results to MEL. This is done using the set of overloaded “setResult” and “appendToResult” methods inherited from MPxCommand. For example, if the command needs to return an integer value of 4, it can be done using code that looks like the following:

```
int result =4;
clearResult();
setResult( result );
```

You can return arrays by making multiple calls to the appendToResult method. For example, to return three doubles indicating the position of apoint in three-space, you could do something like the following:

```
MPoint result (1.0, 2.0, 3.0);
...
clearResult();
appendToResult( result.x );
appendToResult( result.y );
appendToResult( result.z );
```

Or, this can also be done by returning an array.

```
MDoubleArray result;
MPoint point (1.0, 2.0, 3.0);

result.append( point.x );
result.append( point.y );
result.append( point.z );

clearResult();
setResult( result );
```

Syntax objects

The classes you need to work with when writing syntax objects are MSyntax and MArgDatabase. These classes are required for defining and handling command flag input.

MSyntax—Used to specify flags and arguments passed to commands.

MArgDatabase—Class used to parse and store all flags, arguments, and objects passed to a command. The MArgDatabase accepts an MSyntax object, which describes the format for a command, and uses it to parse the command arguments into a form which is easy to query .

Note MArgParser is similar to MArgDatabase except that it is used for context commands rather than commands.

Flags

Syntax objects require flags. You need to define both short flags and long flags. Short flags are three letters or less; long flags are four letters or more.

Define these flags in one place using the `#define` declaration. For example, `scanDagSyntax` uses the following flags:

```
#define kBreadthFlag     "-b"
```

```
#define kBreadthFlagLong "-breadthFirst"
#define kDepthFlag       "-d"
#define kDepthFlagLong   "-depthFirst"
```

Creating the Syntax Object

In your command class, you need to write a newSyntax method in which the syntax for your command is set up. This method needs to be a static method that returns the syntax object, MSyntax.

Inside your newSyntax method, you need to add the necessary flags to a syntax object and then return it.

The scanDagSyntax class's newSyntax is defined in the following way:

```
class scanDagSyntax: public MPxCommand
{
public:
    ...
    static MSyntax newSyntax();
    ...
};

MSyntax scanDagSyntax::newSyntax()
{
    MSyntax syntax;

    syntax.addFlag(kBreadthFlag, kBreadthFlagLong);
    syntax.addFlag(kDepthFlag, kDepthFlagLong);
    ...
    return syntax;
}
```

Parsing the Arguments

By convention, the arguments to your command are typically parsed in a parseArgs method which is called from doIt. This parseArgs method creates a local MArgDatabase which is initialized with a syntax object and the arguments to the command. MArgDatabase has convenient methods which enable you to determine which flags are set.

Note Unless otherwise specified, all API methods use Maya internal units—cm and radians.

```
MStatus scanDagSyntax::parseArgs(const MArgList &args,
                                 MItDag::TraversalType &
                                     traversalType,
                                 MFn::Type &filter,
                                 bool &quiet)
```

```
{
    MArgDatabase  argData(syntax(), args);

    if (argData.isFlagSet(kBreadthFlag))
        traversalType = MItDag::kBreadthFirst;
    else if (argData.isFlagSet(kDepthFlag))
        traversalType = MItDag::kDepthFirst;
    ...
    return MS::kSuccess;
}
```

Registration

The method that creates the syntax object is registered with the command in the initializePlugin method.

```
MStatus initializePlugin( MObject obj )
{
    MStatus status;
    MFnPlugin plugin(obj, "Alias - Example",
                     "2.0", "Any");
    status = plugin.registerCommand("scanDagSyntax",
                                    scanDagSyntax::creator,

scanDagSyntax::newSyntax);
    return status;
}
```

Contexts

Contexts in Maya are modes which define how mouse interaction will be interpreted. A context can execute commands, modify the current selection, or perform drawing operations, etc. In addition, the context can draw the cursor differently denoting the context. In Maya, contexts are presented with the name “tool.”

MPxContext

The MPxContext class allows you to create your own context.

The realization of a context in the Maya application is done through a special command which creates the context. In this regard, a context is similar to a shape—it is created and modified by a command and has state which defines its behavior or appearance. When you write a context by subclassing MPxContext, you must also define a command for it by subclassing from MPxContextCommand described in the following.

The following is the marqueeTool example which does simple selection using a user drawn OpenGL selection box. For the purposes of brevity, the header files have been left out. See the example code in the devkit/plug-ins directory for the complete example.

```
const char helpString[] =
    "Click with left button or drag with middle button to
select";

class marqueeContext : public MPxContext
{
public:
                            marqueeContext();
    virtual     void        toolOnSetup( MEvent & event );
    virtual MStatus         doPress( MEvent & event );
    virtual MStatus         doDrag( MEvent & event );
    virtual MStatus         doRelease( MEvent & event );
    virtual MStatus         doEnterRegion( MEvent & event );
```

The methods on MPxContext provide default actions if they are not overridden so you need only define those methods which are necessary for the proper functioning of your context. What each of these methods does is described below.

```
private:
    short                   start_x, start_y;
    short                   last_x, last_y;
    MGlobal::ListAdjustment listAdjustment
    M3dView                 view;
};

marqueeContext::marqueeContext()
{
    setTitleString ( "Marquee Tool" );
}
```

The constructor sets the title that will appear in the UI when this tool is selected.

```
void marqueeContext::toolOnSetup ( MEvent & )
{
    setHelpString( helpString );
}
```

When the tool is selected this method is called to put user help information on the prompt line and so that you can do any initialization that may be required.

```
MStatus marqueeContext::doPress( MEvent & event )
{
```

This method is called after the tool has been selected and you have pressed a mouse button. The MEvent object contains the relevant information to the user's mouse down event, such as the co-ordinates the user clicked on.

```
    if (event.isModifierShift() || event.isModifierControl()
) {
        if ( event.isModifierShift() ) {
            if ( event.isModifierControl() ) {
                // both shift and control pressed, merge new
selections
                listAdjustment = MGlobal::kAddToList;
            } else {
                // shift only, xor new selections with
previous ones
                listAdjustment = MGlobal::kXORWithList;
            }
        } else if ( event.isModifierControl() ) {
            // control only, remove new selections from the
previous list
            listAdjustment = MGlobal::kRemoveFromList;
        }
    }
    else
        listAdjustment = MGlobal::kReplaceList;
```

Since the mode of selection can be varied by what modifier keys are held down, the mouse up event is checked to see what if any modifiers were down, and then adjusts the type of selection accordingly.

```
    event.getPosition( start_x, start_y );
```

The positions of the selection are determined. This method returns screen co-ordinates.

```
    view = M3dView::active3dView();
    view.beginGL();
```

This determines the active view and enables OpenGl rendering into it.

```
    view.beginOverlayDrawing();
    return MS::kSuccess;
}

MStatus marqueeContext::doDrag( MEvent & event )
{
```

This method is called while the mouse button is down and you drag the cursor around the screen.

```
    event.getPosition( last_x, last_y );
    view.clearOverlayPlane();
```

Each time this method is called the overlay planes are cleared before rendering the new selection box.

```
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluOrtho2D(
            0.0, (GLdouble) view.portWidth(),
            0.0, (GLdouble) view.portHeight()
    );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glTranslatef(0.375, 0.375, 0.0);
```

This sets up the view transformations to make sure that the rendering correctly appears in the active view.

```
    glLineStipple( 1, 0x5555 );
    glLineWidth( 1.0 );
    glEnable( GL_LINE_STIPPLE );
    glIndexi( 2 );
```

Next the line style is selected.

```
    // Draw marquee
    //
    glBegin( GL_LINE_LOOP );
        glVertex2i( start_x, start_y );
        glVertex2i( last_x, start_y );
        glVertex2i( last_x, last_y );
        glVertex2i( start_x, last_y );
    glEnd();
```

The selection box is drawn.

```
#ifndef _WIN32
     glXSwapBuffers(view.display(), view.window() );
#else
     SwapBuffers(view.deviceContext() );
#endif
```

The buffers are swapped.

```
    glDisable( GL_LINE_STIPPLE );
```

Finally the special draw mode for the lines is disabled.

```
    return MS::kSuccess;
}

MStatus marqueeContext::doRelease( MEvent & event )
{
```

This method is called when the mouse button is released.

```
    MSelectionList              incomingList, marqueeList;
    MGlobal::ListAdjustment     listAdjustment;

    view.clearOverlayPlane();
    view.endOverlayDrawing();
    view.endGL();
```

All OpenGL rendering is done so the overlay planes are cleared and OpenGL rendering is turned off for the active view.

```
    event.getPosition( last_x, last_y );
```

This determines the co-ordinates where the mouse button was released.

```
    MGlobal::getActiveSelectionList(incomingList);
```

This gets the current selection list and saves a copy for later use.

```
    if ( abs(start_x - last_x) < 2 && abs(start_y - last_y)
< 2 )
        MGlobal::selectFromScreen( start_x, start_y,
MGlobal::kReplaceList );
```

If the co-ordinates are the same at the beginning and end, then a click-pick was done rather than a bounding box pick.

```
    else
        // Select all the objects or components within the
marquee.
        MGlobal::selectFromScreen( start_x, start_y, last_x,
last_y,
            MGlobal::kReplaceList );
```

Do a bounding box pick.

```
    // Get the list of selected items
    MGlobal::getActiveSelectionList(marqueeList);
```

This gets the list of objects just selected.

```
    MGlobal::setActiveSelectionList(incomingList, \
        MGlobal::kReplaceList);
```

Restore the original selection list.

```
    MGlobal::selectCommand(marqueeList, listAdjustment);
```

Modify the original selection list using your modifier and the selected object’s.

```
    return MS::kSuccess;
}

MStatus marqueeContext::doEnterRegion( MEvent & )
{
    return setHelpString( helpString );
}
```

This method is called whenever you move the mouse on top of one of the modeling views.

```
class marqueeContextCmd : public MPxContextCommand
{
public:
                           marqueeContextCmd();
    virtual MPxContext*    makeObj();
    static  void*          creator();
};
```

MPxContextCommand

The MPxContextCommand is the class used to define the special command for creating contexts. Context commands are similar to regular commands in that they can be executed from the command line and put into MEL scripts. They can have edit and query options which modify the properties of the context. They create an instance of the context and give it to Maya. Context commands are not undoable.

Creating a context command

The following is the implementation of the context command used to create the marquee context described previously.

```
marqueeContextCmd::marqueeContextCmd() {}
```

This method is used by Maya to create an instance of the context.

```
MPxContext* marqueeContextCmd::makeObj()
{
    return new marqueeContext();
}

void* marqueeContextCmd::creator()
{
    return new marqueeContextCmd;
}

MStatus initializePlugin( MObject obj )
{
    MStatus status;
    MFnPlugin plugin( obj, “Alias”, “1.0”, “Any”);
    status = plugin.registerContextCommand( \
           “marqueeToolContext”, marqueeContextCmd::creator
);
```

The context command must be registered, but in this case MFnPlugin::registerContextCommand() is used rather than MFnPlugin::registerCommand().

```
    return status;
}

MStatus uninitializePlugin( MObject obj )
{
    MStatus status;
    MFnPlugin plugin( obj );
    status = plugin.deregisterContextCommand( \
           "marqueeToolContext" );
```

MFnPlugin::deregisterContextCommand() is likewise used to deregister the context command.

```
    return status;
}
```

And that's all that's necessary to create a simple context.

Adding a context command to the Maya shelf

There are two ways to "activate" or make your context the current context in Maya. The first is through the use of the setToolTo command. This command takes the name of a context (tool) and makes it the current context.

A second method is by making an icon to represent your context and putting it in the Maya tool shelf. The Maya tool shelf can store two kinds of buttons, command buttons and tool buttons. When the tool is activated, its icon is displayed next to the standard Maya tools in the toolbar.

The following is a set of MEL commands you can use to create a context and tool button for the context.

```
marqueeToolContext marqueeToolContext1;
setParent Shelf1;
toolButton  -cl toolCluster
            -t marqueeToolContext1
            -i1 "marqueeTool.xpm" marqueeTool1;
```

This MEL code instantiates an instance of the marqueeToolContext and adds it to the "Common" tools shelf.

marqueeTool.xpm, the icon for the tool, must be in the XBMLANGPATH to be found and added to the UI. If it is not found, a blank spot will appear on the shelf, but the tool will still be usable.

This code could either be sourced by hand from the MEL command window, or it could be invoked with MGlobal::sourceFile() in the initializePlugin() method of the plug-in.

Tool property sheets

Tool property sheets are interactive editors for displaying and modifying the properties of a context. They are similar to attribute editors for modifying properties of a dependency graph node. They execute the context command following user actions in the editor to perform editing operations on the activated context. The tool property sheet for the activated context is displayed by double-clicking on the tool's icon.

- Implementation of a tool property sheet for your context entails writing two MEL files, one for editing the context and one for querying the context.
- The files must be named <yourContextName>Properties.mel and <yourContextNameValues.mel where <yourContextName> is the name of your context as returned by the getClassName() method of your context.
- The <>Properties.mel file defines the layout of the editor and the actions to be taken by widgets in the editor.
- The <>Values.mel file is used to retrieve values from the context with the editor is initialized.

Refer to the helixProperties.mel and helixValues.mel files in the example plug-in directory for a sample implementation of a property sheet.

To effectively implement a tool property sheet for your context, you must implement sufficient edit and query options in your context command as well as sufficient access methods in your MPxContext class for setting and retrieving its internal properties.

MPxToolCommand

The MPxToolCommand is the base class for creating commands that can be executed from within a context. Tool commands are similar to regular commands in that they are defined with command flags and can be executed from the Maya command line. However, they must perform extra duties, as the actions taken by a context do not come from the normal Maya command mechanism but from inside the methods of the MPxContext class. These duties are to alert Maya of the execution of the command so that the undo/redo and journalling mechanisms operate correctly on the command. The MPxToolCommand is a subclass of MPxCommand with the additional methods.

If a context wants to perform its own command, it must register the command when the context and context command are registered. A context can only have one tool command associated with it.

The following is an example of a tool command, the helixTool. As with the marqueeTool, the list of include files is omitted for brevity. See the helixTool.cpp file in devkit/plug-ins for the complete example.

```
#define         NUMBER_OF_CVS         20

class helixTool : public MPxToolCommand
{
public:
                        helixTool();
    virtual             ~helixTool();
    static void*        creator();

    MStatus             doIt( const MArgList& args );
    MStatus             redoIt();
    MStatus             undoIt();
    bool                isUndoable() const;
    MStatus             finalize();
    static MSyntax      newSyntax();
```

The set of methods on MPxToolCommand are similar to those on MPxCommand but with the addition of finalize(). The finalize method is used to create a string representing the command and its arguments.

```
    void                setRadius( double newRadius );
    void                setPitch( double newPitch );
    void                setNumCVs( unsigned newNumCVs );
    void                setUpsideDown( bool newUpsideDown );
```

These methods are necessary since the properties of the helix will be set from the context object.

```
private:
    double              radius;     // Helix radius
    double              pitch;      // Helix pitch
    unsigned            numCV;      // Helix number of CVs
    bool                upDown;     // Helis upsideDown
    MDagPath            path;       // Dag path to the curve.
                                    // Don't save the
pointer!
};

void* helixTool::creator()
{
    return new helixTool;
}

helixTool::~helixTool() {}
```

These first two methods are identical to the earlier “helix2” example.

```
helixTool::helixTool()
{
    numCV = NUMBER_OF_CVS;
    upDown = false;
    setCommandString( “helixToolCmd” );
}
```

The constructor saves away the name of the MEL command for later use in the finalize() method.

```
MSyntax helixTool::newSyntax()
{
    MSyntax syntax;

    syntax.addFlag(kPitchFlag, kPitchFlagLong,
            MSyntax::kDouble);
    syntax.addFlag(kRadiusFlag, kRadiusFlagLong,
            MSyntax::kDouble);
    syntax.addFlag(kNumberCVsFlag, kNumberCVsFlagLong,
            MSyntax::kUnsigned);
    syntax.addFlag(kUpsideDownFlag, kUpsideDownFlagLong,
            MSyntax::kBoolean);

    return syntax;
}

MStatus helixTool::doIt( const MArgList &args )
{
    MStatus status;

    status = parseArgs(args);

    if (MS::kSuccess != status)
        return status;

    return redoIt();
}

MStatus helixTool::parseArgs(const MArgList &args)
{
    MStatus status;
    MArgDatabase argData(syntax(), args);

    if (argData.isFlagSet(kPitchFlag)) {
        double tmp;
        status = argData.getFlagArgument(kPitchFlag, 0,
tmp);
        if (!status) {
            status.perror("pitch flag parsing failed.");
            return status;
        }
        pitch = tmp;
    }
    if (argData.isFlagSet(kRadiusFlag)) {
        double tmp;
        status = argData.getFlagArgument(kRadiusFlag, 0,
tmp);
        if (!status) {
            status.perror("radius flag parsing failed.");
            return status;
        }
        radius = tmp;
    }

    if (argData.isFlagSet(kNumberCVsFlag)) {
        unsigned tmp;
        status = argData.getFlagArgument(kNumberCVsFlag,
                0, tmp);
        if (!status) {
            status.perror("numCVs flag parsing failed.");
            return status;
        }
        numCV = tmp;
    }

    if (argData.isFlagSet(kUpsideDownFlag)) {
        bool tmp;
        status = argData.getFlagArgument(kUpsideDownFlag,
                0, tmp);
        if (!status) {
            status.perror("upside down flag parsing
failed.");
            return status;
        }
        upDown = tmp;
    }

    return MS::kSuccess;
}
```

This method is similar to the earlier helix example—it parses the arguments and uses them to set its internal state. In general, this command will be used through the UI, but since it is still a MEL command, it can be invoked from the MEL command shell.

```
MStatus helixTool::redoIt()
{
    MStatus stat;

    const unsigned  deg     = 3;            // Curve Degree
    const unsigned  ncvs    = NUMBER_OF_CVS;// Number of CVs
    const unsigned  spans   = ncvs - deg;   // Number of
spans
```

```
    const unsigned  nknots  = spans+2*deg-1;// Number of
knots
    unsigned        i;
    MPointArray     controlVertices;
    MDoubleArray    knotSequences;

    int upFactor;
    if (upDown) upFactor = -1;
    else upFactor = 1;

    // Set up cvs and knots for the helix
    //
    for (i = 0; i < ncvs; i++)
        controlVertices.append( MPoint(
                radius * cos( (double)i ),
                upFactor * pitch * (double)i,
                radius * sin( (double)i ) ) );

    for (i = 0; i < nknots; i++)
        knotSequences.append( (double)i );

    // Now create the curve
    //
    MFnNurbsCurve curveFn;

    MObject curve = curveFn.create( controlVertices,
            knotSequences, deg, MFnNurbsCurve::kOpen,
            false, false, MObject::kNullObj, &stat );

    if ( !stat )
    {
        stat.perror("Error creating curve");
        return stat;
    }

    stat = curveFn.getPath( path );

    return stat;
}
```

This is essentially the same as the earlier helix example.

```
MStatus helixTool::undoIt()
{
    MStatus stat;
    MObject transform = path.transform();
    stat = MGlobal::removeFromModel( transform );
    return stat;
}
```

Again this is essentially the same as the earlier helix example. You should be noticing a pattern developing. It is quite easy to change a command into a tool. Little of the command has to be changed, additions just have to be made to hook the tool up to the UI.

```
bool helixTool::isUndoable() const
{
    return true;
}
```

This tool is undoable.

```
MStatus helixTool::finalize()
{
    MArgList command;
    command.addArg( commandString() );
    command.addArg( MString(kRadiusFlag) );
    command.addArg( radius );
    command.addArg( MString(kPitchFlag) );
    command.addArg( pitch );
    command.addArg( MString(kNumberCVsFlag) );
    command.addArg( (int)numCV );
    command.addArg( MString(kUpsideDownFlag) );
    command.addArg( upDown );
    return MPxToolCommand::doFinalize( command );
}
```

This method is the one noticeable addition to the tool which wasn't necessary in the command. When a command is typed in it is easy to take it and print it out to a journal file. Since a tool is not typed in, but created through mouse input, no text string exists to be output to a journal file. The finalize() method solves this by outputting a string when the tool has completed. It is necessary for you to call MPxToolCommand::doFinalize() to have the command output to the journal file.

```
void helixTool::setRadius( double newRadius )
{
    radius = newRadius;
}

void helixTool::setPitch( double newPitch )
{
    pitch = newPitch;
}

void helixTool::setNumCVs( unsigned newNumCVs )
{
    numCV = newNumCVs;
}

void helixTool::setUpsideDown( double newUpsideDown )
{
    upDown = newUpsideDown;
}

const char helpString[] = "Click and drag to draw helix";

class helixContext : public MPxContext
{
```

This is the context which will be executing the helixTool command..

```
public:
                            helixContext();
    virtual void            toolOnSetup( MEvent & event );
    virtual MStatus         doPress( MEvent & event );
    virtual MStatus         doDrag( MEvent & event );
    virtual MStatus         doRelease( MEvent & event );
    virtual MStatus         doEnterRegion( MEvent & event );
```

The set of methods are the same as for the marqueeTool example.

```
private:
    short                   startPos_x, startPos_y;
    short                   endPos_x, endPos_y;
    unsigned                numCV;
    bool                    upDown;
    M3dView                 view;
    GLdouble                height,radius;
};

helixContext::helixContext()
{
    setTitleString( "Helix Tool" );
}

void helixContext::toolOnSetup( MEvent & )
{
    setHelpString( helpString );
}

MStatus helixContext::doPress( MEvent & event )
{
    event.getPosition( startPos_x, startPos_y );
    view = MGlobal::active3dView();
    view.beginGL();
    view.beginOverlayDrawing();
    return MS::kSuccess;
}
```

These three methods are essentially the same as the marqueeTool examples, the only difference being that doPress() for the helixTool does not need to determine the modifier key state.

```
MStatus helixContext::doDrag( MEvent & event )
```

```
{
    event.getPosition( endPos_x, endPos_y );
    view.clearOverlayPlane();
    glIndexi( 2 );

    int upFactor;
    if (upDown) upFactor = 1;
    else upFactor = -1;

    // Draw the guide cylinder
    //
    glMatrixMode( GL_MODELVIEW );
    glPushMatrix();
        glRotatef( upFactor * 90.0, 1.0f, 0.0f, 0.0f );
        GLUquadricObj *qobj = gluNewQuadric();
        gluQuadricDrawStyle(qobj, GLU_LINE);
        GLdouble factor = (GLdouble)numCV;
        radius = fabs(endPos_x - startPos_x)/factor + 0.1;
        height = fabs(endPos_y - startPos_y)/factor + 0.1;
        gluCylinder( qobj, radius, radius, height, 8, 1 );
    glPopMatrix();
```

This code draws a cylinder in the current view defining the outlines of the helix that will be generated.

```
    #ifndef _WIN32
        glXSwapBuffers(view.display(), view.window() );
    #else
        SwapBuffers(view.deviceContext() );
    #endif

    return MS::kSuccess;
}

MStatus helixContext::doRelease( MEvent & )
{
    // Clear the overlay plane & restore from overlay
drawing
    //
    view.clearOverlayPlane();
    view.endOverlayDrawing();
    view.endGL();
```

The user has released the mouse so this code cleans up the OpenGL drawing.

```
    helixTool * cmd = (helixTool*)newToolCommand();
    cmd->setPitch( height/NumCVs );
    cmd->setRadius( radius );
    cmd->setNumCVs( numCV );
    cmd->setUpsideDown( upDown );
    cmd->redoIt();
    cmd->finalize();
```

This code creates the actual helixTool command, by calling the helixTool::creator method that you will register later, sets the radius and pitch, and then calls the redoIt() method to generate the data. As a last step, the finalize() method is called to ensure that this command is written out to the journal file.

```
    return MS::kSuccess;
}

MStatus helixContext::doEnterRegion( MEvent & )
{
    return setHelpString( helpString );
}

void helixContex::getClassName( MString &name ) const
{
    name.set("helix");
}
```

The next four methods are used in the interaction between the context and the contextCommand's edit and query methods. These will be called by the tool property sheet for the context. The MToolsInfo::setDirtyFlag() method alerts the tool property sheet so it can redraw itself with the new property values for the context.

```
void helixContext::setNumCVs( unsigned newNumCVs )
{
    numCV = newNumCVs;
    MToolsInfo::setDirtyFlag(*this);
}

void helixContext::setUpsideDown( bool newUpsideDown )
{
    upDown = newUpsideDown;
    MToolsInfo::setDirtyFlag(*this);
}

unsigned helixContext::numCVs()
{
    return numCV;
}

bool helixContext::upsideDown()
{
    return upDown;
}
```

The next class and implementation repeats the code from the marqueeTool example. This class is necessary to create instances of the tool context.

```
class helixContextCmd : public MPxContextCommand
{
public:
                            helixContextCmd();
    virtual MStatus         doEditFlags();
    virtual MStatus         doQueryFlags();
    virtual MPxContext*     makeObj();
    virtual MStatus         appendSyntax();
    static void*            creator();

protected:
    helixContext *          fHelixContext;
};

helixContextCmd::helixContextCmd() {}

MPxContext* helixContextCmd::makeObj()
{
    fHelixContext = new helixContext();
    return fHelixContext;
}

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

The next two methods handle the argument parsing for the command. There are two types of arguments—those which make modifications to the properties of a context, and those which query the properties of a context.

Note Argument parsing is done through the MPxContextCommand::parser() method which returns an MArgParser. This class is analogous to the MArgDatabase class that is used with the MPxCommand class.

```
MStatus helixContextCmd::doEditFlags()
{
    MArgParser argData = parser();

    if (argData.isFlagSet(kNumberCVsFlag)) {
        unsigned numCVs;
        status = argData.getFlagArgument(kNumberCVsFlag,
                0, numCVs);
        if (!status) {
            status.perror("numCVs flag parsing failed.");
            return status;
        }
        fHelixContext->setNumCVs(numCVs);
    }

    if (argData.isFlagSet(kUpsideDownFlag)) {
        bool upsideDown;
        status = argData.getFlagArgument(kUpsideDownFlag,
                0, upsideDown);
        if (!status) {
            status.perror("upsideDown flag parsing
failed.");
            return status;
        }
        fHelixContext->setUpsideDown(upsideDown);
    }

    return MS::kSuccess;
}

MStatus helixContextCmd::doQueryFlags()
{
    MArgParser argData = parser();

    if (argData.isFlagSet(kNumberCVsFlag)) {
        setResult((int) fHelixContext->numCVs());
    }
    if (argData.isFlagSet(kUpsideDownFlag)) {
        setResult(fHelixContext->upsideDown());
    }

    return MS::kSuccess;
}

MStatus helixContextCmd::appendSyntax()
{
    MStatus status;

    MSyntax mySyntax = syntax();

    if (MS::kSuccess != mySyntax.addFlag(kNumberCVsFlag,
            kNumberCVsFlagLong, MSyntax::kUnsigned)) {
        return MS::kFailure;
    }

    if (MS::kSuccess != mySyntax.addFlag(kUpsideDownFlag,
            kUpsideDownFlagLong, MSyntax::kBoolean)) {
```

```
        return MS::kFailure;
    }

    return MS::kSuccess;
}

MStatus initializePlugin( MObject obj )
{
    MStatus status;

    MFnPlugin plugin( obj, “Alias”, “1.0”, “Any”);


    // Register the context creation command and the tool
    // command that the helixContext will use.
    //
    status = plugin.registerContextCommand(
            “helixToolContext”, helixContextCmd::creator,
            “helixToolCmd”, helixTool::creator,
            helixTool::newSyntax);
    if (!status) {
        status.perror(“registerContextCommand”);
        return status;
    }

    return status;
}
```

The initializePlugin() method registers both the helix command and the context via a single register call.

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

    // Deregister the tool command and the context
    // creation command.
    //
    status = plugin.deregisterContextCommand(
            “helixToolContext” “helixToolCmd”);
    if (!status) {
        status.perror(“deregisterContextCommand”);
        return status;
    }

    return status;
}
```

MEL code similar to the marqueeTool example’s is necessary to attach the helixTool to the UI.

4 DAG Hierarchy

Developer

Plug-in API

DAG Hierarchy

Overview of the DAG Hierarchy

In Maya, a *directed acyclic graph* (DAG), defines elements such as the position, orientation, and scale of geometry. The DAG is composed of two types of DAG nodes, transforms and shapes.

Transform nodes—Maintain transformation information (position, rotation, scale, etc.) as well as parenting information. For example, if you model a hand, you would like to apply a single transformation to rotate the palm and fingers, rather than rotating each individually—in this case the palm and fingers would share a common parent transformation node.

Shape nodes—Reference geometry and do not provide parenting or transformation information.

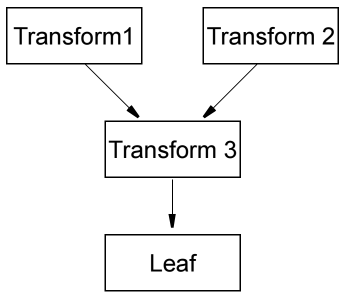
In the simplest case, the DAG describes how an instance of an object is constructed from a piece of geometry. For example, when you create a sphere, you create both a shape node (the sphere) and a transformation node that allows you to specify the sphere's position, scale, or rotation. The transformation node's shape node is its *child*.

Nodes

Transform nodes can have multiple child nodes—the child nodes are "grouped" beneath the transformation node. Node grouping allows them to share transformation information and be treated as a unit.

Instancing

Transform nodes and shape nodes can also have multiple parent nodes—these nodes are "instanced". Instancing can be useful to reduce the amount of geometry for a model. For example, if you model a tree, you could create a thousand unique leaves to populate the tree. This would make for a very data heavy model, since each leaf would have it's own transformation nodes, shape nodes, and NURBS or polygon data. Instead, you can create a single leaf and instance it a thousand times to create a thousand identical leaves and position them independently around the branches of the tree. This way the shape node and NURBS or polygon data is shared.



This DAG hierarchy has three transform nodes (Transform1, Transform2, Transform3) and one shape node (Leaf). This DAG hierarchy would cause two leaves to be displayed since Transform3 and the Leaf is instanced (it has two parents).

Transforms and shapes

A DAG node is simply an entity in the DAG. It may have and know about parents, siblings, and children, but it does not necessarily know about transformations or geometry. Transforms and Shapes are two types of nodes derived from a DAG node. A transform node is a type of DAG node which handles transformations (translate, rotate, and scale), while a shape node is a type of DAG node which handles geometry. A shape node does not maintain transformation information, and geometry cannot be hung below a transform node. This means that any piece of geometry requires two DAG nodes above it, a shape node immediately above it, and a transform node above the shape node.

For example:

MFnDagNode—has methods for determining the number of parents, and the parents of a node.

MFnTransform—is the function set for operating on transform nodes (derived from MFnDagNode) and has methods to get and set transformation, such as rotation, translation, or scale.

MFnNurbsSurface—is one of many types of function sets which operate on the many types of shape nodes (also derived from MFnDagNode, but not derived from MFnTransform) and has methods to get and set the CVs of the surface, etc.

DAG paths

A path through the DAG is a set of nodes which uniquely identifies the location of a particular node or instance of a node in the graph. The path represents a graph ancestry beginning with the root node of the graph and containing, in succession, a particular child of the root node followed by a particular child of this child, etc., down to the node identified by the path. For instanced nodes, there are multiple paths which lead from the root node to the instanced node, one path for each instance. Paths are displayed in Maya by naming each node in the path starting with the root node and separated by the vertical line character, “|”.

DAG paths and worldspace operations in the API

It is important to note that because the DAG path represents how a shape is inserted into the scene, a DAG path must be used when attempting any world space operation via the API. If one simply gets an “MObject” handle to a node and asks for the world space position of a component of that node, the API operation will fail. This is because without the DAG path Maya has no idea where in world space the object is. In fact, in the case of instanced objects, there can be multiple answers to that question, and only the DAG path will uniquely identify the particular instance of the node. Almost all of the classes that contains methods that will return an MObject for a node also contain methods that will return DAG paths so you can get an MDagPath handle to the desired node. As well, all the MFn classes can be constructed with either an MObject or an MDagPath. If an MObject is used, all world space operations attempted using methods of the MFn class will fail, if an MDagPath is used, they will succeed.

Adding or removing nodes from the representation

The MDagPath class builds a representation of a path and lets you examine and modify this representation. The MDagPath represents paths as a stack of nodes with the root node being on the bottom of the stack. The push() and pop() methods allow nodes along the path to be added or removed from the representation.

Note These methods do not add and remove nodes from the actual DAG but only from the representation constructed by the MDagPath class. The comparison and assignment operators operate on the representations constructed by the MDagPath class and not on the graph itself. So assigning one path to another will not modify the graph but only the contents of the destination MDagPath instance.

Inclusive and exclusive matrices

Since the nodes in a path exist at different levels of the DAG hierarchy, there is a different transformation that may have accumulated at each node in the path. The MDagPath class allows these transformations to be returned using the inclusiveMatrix() and exclusiveMatrix() classes.

- An "inclusive matrix" represents the accumulated transformation to the last node stored in the DAG path taking into account the last node.
- An "exclusive matrix" represents the same accumulation with the exception that it does not take into account any transformation from the last node.

For example, if a path is defined as:

|RootTransform|Transform1|Transform2|Shape

the inclusive matrix down to Transform2 would be the accumulation of RootTransform, Transform1, and Transform2. The exclusive matrix would contain the accumulation of only RootTransform and Transform1.

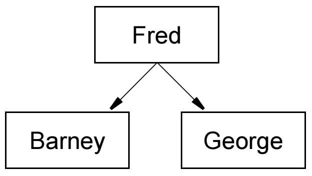
Why add the shape node to a DAG path

In Maya, selection at the object level results in the selection of the transform node that is the parent node of the shape actually selected. When querying the selection using MGlobal::getActiveSelectionList(), the MDagPath returned to the caller only specifies the path to this transform and not down to the actual shape that was picked on the screen. A convenience method on MDagPath called extendToShape() can be called to add the shape node below the last transform to the path.

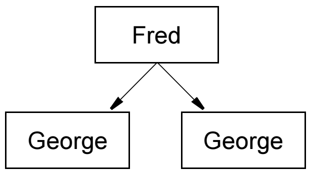
The valid function sets applicable to a particular MDagPath are determined by the last node on the path. If the last node is a transform node, then the function sets that can operate on transform nodes can be applied to the MDagPath instance. If a shape node is the last node of the path, then the applicable function sets for the MDagPath instance are those sets which can operate on the shape.

Unique Names

The use of the DAG path allows for object names to be reused. Object names can be reused as long as the same name does not appear on more than one DAG node with a common parent.



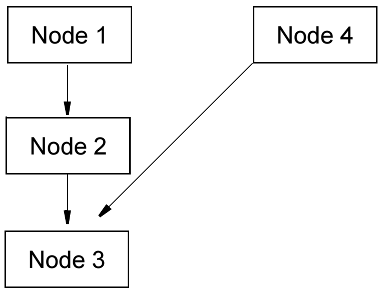
This is legal.



This is illegal.

Generalized instancing

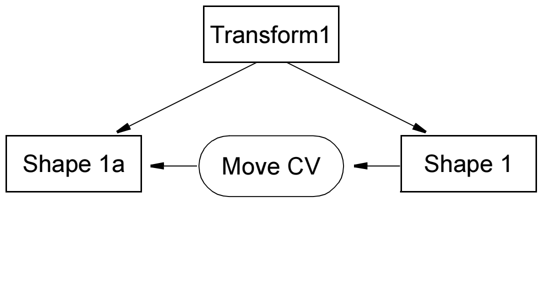
Maya supports generalized instancing. Generalized instancing means nodes which instance another node do not have to be siblings.



Node 2 and Node 4 are not siblings yet they each instance Node 3. More complex hierarchies can be created, so long as a reference is not made back up the hierarchy. Doing so could create a cycle which would break the acyclic nature of the DAG (remember that a DAG is a directed *acyclic* graph).

Transforms with multiple shapes

A transform node can have any number of transform nodes as children. In general, a transform node can only have a single shape node as a child, and when viewing the DAG through an interactive window this will always be the case. However when examining the DAG through the API you will find that transforms may have multiple shape nodes as children. This happens when the original shape under the transform has been modified by the dependency graph. To maintain the transformations on the result of the dependency graph, the result is placed under the same transform as the original node. The new node would have the same DAG transforms applied as the original, but would be modified in some way (for example, its CVs could have been moved). When this happens, only the final product is visible in an interactive window, and the original nodes are historical.



|Transform1|Shape1 is the original historical object while |Transform1|Shape1a is the actual object visible in any interactive window. |Transform1|Shape1 is also called an intermediate object.This is important later when working with the dependency graph.

WARNING If you use the MDagPath::extendToShape() method on a path whose last node is a transform that contains multiple shapes, the first child shape node will be the node that is added onto the end of the path. If this is not the desired node, it is recommended that you not use the extendToShape() method. Instead, use the MDagPath::child() and MDagPath::childCount() methods to help examine and access the desired shape node.

The Underworld

The "underworld" is a name given to the parameter space of a shape node, such as the UV space of a NURBS surface. Nodes and whole subgraphs of nodes may be defined in this underworld space.

For example, the transform and shape nodes that define a curve on a NURBS surface. The control points defining the curve are in the UV space of the surface. The paths that uniquely identify the nodes of an underworld are rooted inside the shape node which defines the parameter space of the underworld. The first node of an underworld path is the first node that is defined in the parameter space of the containing shape. Most likely this first node is a transform node.

Underworld paths are specified in Maya like regular paths, using the "|" character to separate the node names in the path. The extra nomenclature is the use of the "->" characters to specify the transition between the shape node and the root node of the underworld path.

For example, the complete specification of a path to a curve on surface node of a NURBS surface could be listed as |SurfaceTransform|NURBSSurface->UnderworldTransform|CurvesShape. Underworlds may be recursively defined on the shapes in the underworld as long as the shapes have some parameter space which defines them.

The MDagPath contains methods for accessing the different paths from a shape down through its underworld. The methods MDagPath::pathCount() method returns the total number of paths represented by the given MDagPath instance. In the above curve on surface example, if the MDagPath instance represents the path down to the curve shape in the underworld, the pathCount would be 2. The MDagPath::getPath() method returns a path, either in the underworld or in the 3D space. Path 0 always specifies the 3D path. Path 1 specifies the path in the underworld directly inside the shape at the end of path 0. Path 2 specifies the path in the underworld inside the shape at the end of path 1, and so forth.

DAG walking example

The following example is the scanDagSyntaxCmd example. It demonstrates iterating through the DAG in either a depth first, or a breadth first manner. This code makes a good basis for many DAG walking plug-ins, in particular those written as file translators.

As with previous examples, the list of include files is omitted for brevity. See the scanDagSyntaxCmd.cpp file in *devkit/plug-ins* for the complete example.

```
class scanDagSyntax: public MPxCommand
{
public:
                    scanDagSyntax() {};
    virtual         ~scanDagSyntax();
    static void*    creator();
    static MSyntax  newSyntax();
    virtual MStatus doIt( const MArgList& );

This is a simple example so the undoIt() and redoIt() methods are not
implemented.
private:
    MStatus         parseArgs( const MArgList& args,
                               MItDag::TraversalType& traversalType,
                               MFn::Type& filter, bool & quiet);
    MStatus         doScan( const MItDag::TraversalType traversalType,
                            MFn::Type filter, bool quiet);
    void            printTransformData(const MDagPath& dagPath, bool quiet);
};

scanDagSyntax::~scanDagSyntax() {}

void* scanDagSyntax::creator()
{
    return new scanDagSyntax;
}

    MSyntax scanDagSyntax::newSyntax()
{
    MSyntax syntax;

    syntax.addFlag(kBreadthFlag, kBreadthFlagLong);
    syntax.addFlag(kDepthFlag, kDepthFlagLong);
    syntax.addFlag(kCameraFlag, kCameraFlagLong);
    syntax.addFlag(kLightFlag, kLightFlagLong);
    syntax.addFlag(kNurbsSurfaceFlag, kNurbsSurfaceFlagLong);
    syntax.addFlag(kQuietFlag, kQuietFlagLong);

    return syntax;
}


MStatus scanDagSyntax::doIt( const MArgList& args )
{
    MItDag::TraversalType   traversalType = MItDag::kDepthFirst;
    MFn::Type               filter        = MFn::kInvalid;
    MStatus                 status;
    bool                    quiet = false;
```

The DAG iterator being used later can be set to only iterate across objects of a particular type (for example cameras). If the filter mode is set to MFn::kInvalid, no filtering will be done and all DAG nodes will be iterated across.

```
    status = parseArgs ( args, traversalType, filter, quiet );
    if (!status)
        return status;

    return doScan( traversalType, filter, quiet);
};
The doIt() method is simply calling a few auxiliary methods which do the real
work.
MStatus scanDagSyntax::parseArgs( const MArgList& args,
                                  MItDag::TraversalType& traversalType,
                                  MFn::Type& filter,
                                  bool & quiet)
{
    MStatus         stat;
    MArgDatabase    argData(syntax(), args);

    MString         arg;

    if (argData.isFlagSet(kBreadthFlag))
        traversalType = MItDag::kBreadthFirst;
    else if (argData.isFlagSet(kDepthFlag))
        traversalType = MItDag::kDepthFirst;

    if (argData.isFlagSet(kCameraFlag))
        filter = MFn::kCamera;
    else if (argData.isFlagSet(kLightFlag))
        filter = MFn::kLight;
    else if (argData.isFlagSet(kNurbsSurfaceFlag))
        filter = MFn::kNurbsSurface;

    if (argData.isFlagSet(kQuietFlag))
```

```
        quiet = true;

    return stat;
}
```

The DAG iterator can either iterate across the DAG depth first or breadth first. This simple example only filters on cameras, lights, and NURBS surfaces, but it is possible to iterate across any type in MFn::Type.

```
MStatus scanDagSyntax::doScan( const MItDag::TraversalType traversalType,
                              MFn::Type filter,
                              bool quiet)
{
```

This method will do all the real work of this command. It uses the traversal type (depth or breadth first) and the filter type to initialize an MItDag (a DAG iterator) to walk across the DAG.

```
    MStatus status;

    MItDag dagIterator( traversalType, filter, &status);
```

The DAG iterator is initialized looking at the DAG. It will walk the DAG downwards.

```
    if ( !status) {
        status.perror("MItDag constructor");
        return status;
    }

    //  Scan the entire DAG and output the name and depth of each node

    if (traversalType == MItDag::kBreadthFirst)
        if (!quiet)
            cout << endl << "Starting Breadth First scan of the Dag";
    else
        if (!quiet)
            cout << endl << "Starting Depth First scan of the Dag";
```

Breadth first walking of the DAG means that siblings will be visited before children, while depth first means that children will be visited before siblings.

```
    switch (filter) {
        case MFn::kCamera:
            if (!quiet)
                cout << ": Filtering for Cameras\n";
            break;
        case MFn::kLight:
            if (!quiet)
                cout << ": Filtering for Lights\n";
            break;
        case MFn::kNurbsSurface:
            if (!quiet)
                cout << ": Filtering for Nurbs Surfaces\n";
            break;
        default:
            cout << endl;
    }

    int objectCount = 0;
    for ( ; !dagIterator.isDone(); dagIterator.next() ) {

        MDagPath dagPath;

        status = dagIterator.getPath(dagPath);
        if ( !status ) {
            status.perror("MItDag::getPath");
            continue;
        }
```

MItDag::getPath() gets the reference to the object that the iterator is currently on. This DAG path can then be used in a function set to operate on the object. In general it is not a good idea to rearrange the DAG from with an iterator.

```
        MFnDagNode dagNode(dagPath, &status);
        if ( !status ) {
            status.perror("MFnDagNode constructor");
            continue;
        }

        if (!quiet)
            cout << dagNode.name() << ": " << dagNode.typeName() << endl;

        if (!quiet)
            cout << "  dagPath: " << dagPath.fullPathName() << endl;

        objectCount += 1;
        if (dagPath.hasFn(MFn::kCamera)) {
```

This determines if the object the iterator is currently visiting is a camera or not, and if it is, the following code outputs camera specific information.

```
            MFnCamera camera (dagPath, &status);
            if ( !status ) {
                status.perror("MFnCamera constructor");
                continue;
            }

            // Get the translation/rotation/scale data
            printTransformData(dagPath, quiet);
```

```
    // Extract some interesting Camera data
    if (!quiet)
    {
        cout << "  eyePoint: "
             << camera.eyePoint(MSpace::kWorld) << endl;
        cout << "  upDirection: "
             << camera.upDirection(MSpace::kWorld) << endl;
        cout << "  viewDirection: "
             << camera.viewDirection(MSpace::kWorld) << endl;
        cout << "  aspectRatio: " << camera.aspectRatio() << endl;
        cout << "  horizontalFilmAperture: "
             << camera.horizontalFilmAperture() << endl;
        cout << "  verticalFilmAperture: "
             << camera.verticalFilmAperture() << endl;
    }
} else if (dagPath.hasFn(MFn::kLight)) {
```

If the object is a light, this code outputs light specific information.

```
    MFnLight light (dagPath, &status);
    if ( !status ) {
        status.perror("MFnLight constructor");
        continue;
    }

    // Get the translation/rotation/scale data
    printTransformData(dagPath, quiet);

    // Extract some interesting Light data
    MColor color;

    color = light.color();
    if (!quiet)
    {
        cout << "  color: ["
             << color.r << ", "
             << color.g << ", "
             << color.b << "]\n";
    }
    color = light.shadowColor();
    if (!quiet)
    {
        cout << "  shadowColor: ["
             << color.r << ", "
             << color.g << ", "
             << color.b << "]\n";

        cout << "  intensity: " << light.intensity() << endl;
    }
} else if (dagPath.hasFn(MFn::kNurbsSurface)) {
```

```
Finally, if the object is a NURBS surface, surface specific information is
output.
            MFnNurbsSurface surface (dagPath, &status);
            if ( !status ) {
                status.perror("MFnNurbsSurface constructor");
                continue;
            }

            // Get the translation/rotation/scale data
            printTransformData(dagPath, quiet);

            // Extract some interesting Surface data
            if (!quiet)
            {
                cout << "  numCVs: "
                     << surface.numCVsInU()
                     << " * "
                     << surface.numCVsInV()
                     << endl;
                cout << "  numKnots: "
                     << surface.numKnotsInU()
                     << " * "
                     << surface.numKnotsInV()
                     << endl;
                cout << "  numSpans: "
                     << surface.numSpansInU()
                     << " * "
                     << surface.numSpansInV()
                     << endl;
            }
        } else {
```

For any other type of DAG node, just the transformation information is printed.

```
            // Get the translation/rotation/scale data
            printTransformData(dagPath, quiet);
        }
    }

    if (!quiet)
    {
        cout.flush();
    }
    setResult(objectCount);
    return MS::kSuccess;
}

void scanDagSyntax::printTransformData(const MDagPath& dagPath, bool quiet)
{
```

```
This method simply determines the transformation information on the DAG node
and prints it out.
    MStatus     status;
    MObject     transformNode = dagPath.transform(&status);
    // This node has no transform - i.e., it's the world node
    if (!status && status.statusCode () == MStatus::kInvalidParameter)
        return;
    MFnDagNode  transform (transformNode, &status);
    if (!status) {
        status.perror("MFnDagNode constructor");
        return;
    }
    MTransformationMatrix   matrix (transform.transformationMatrix());

    if (!quiet)
    {
        cout << "  translation: " << matrix.translation(MSpace::kWorld)
             << endl;
    }
    double                                  threeDoubles[3];
    MTransformationMatrix::RotationOrder    rOrder;

    matrix.getRotation (threeDoubles, rOrder, MSpace::kWorld);
    if (!quiet)
    {
        cout << "  rotation: ["
             << threeDoubles[0] << ", "
             << threeDoubles[1] << ", "
             << threeDoubles[2] << "]\n";
    }
    matrix.getScale (threeDoubles, MSpace::kWorld);
    if (!quiet)
    {
        cout << "  scale: ["
             << threeDoubles[0] << ", "
             << threeDoubles[1] << ", "
             << threeDoubles[2] << "]\n";
    }
}

MStatus initializePlugin( MObject obj )
{
    MStatus status;

    MFnPlugin plugin ( obj, "Alias - Example", "2.0", "Any" );
    status = plugin.registerCommand( "scanDagSyntax",
                                     scanDagSyntax::creator,
                                     scanDagSyntax::newSyntax );

    return status;
```

```
}

MStatus uninitializePlugin( MObject obj )
{
    MStatus status;

    MFnPlugin plugin( obj );
    status = plugin.deregisterCommand( "scanDagSyntax" );

    return status;
}
```

The plug-in finishes with the usual initializePlugin and uninitializePlugin methods.

This plug-in can easily be modified for use as a file translator, or any other type of plug-in which needs to visit the DAG nodes in the model.

5 Dependency graph plug-ins

Developer

Plug-in API

Dependency graph plug-ins

Overview of dependency graph plug-ins

The dependency graph is the heart of Maya. It is used for animation and construction history. You can add new nodes to this graph to allow for entirely new operations.

Parent class descriptions

Twelve different parent classes are defined from which you can subclass a new node. Each parent class specializes in a different functional area of Maya. The parent classes are:

Name	Description
MPxNode	Allows the creation of a new dependency node. This is derived from the most basic DG node in Maya and has no inherited behavior.
MPxLocatorNode	Allows the creation of a new locator node. This is a DAG object that does not render, but which is allowed to draw into the 3d views.
MPxIkSolverNode	Allows the creation of a new type of IK solver.
MPxDeformerNode	Allows the creation of a new deformer.
MPxFieldNode	Allows the creation of a new type of dynamic field.
MPxEmitterNode	Allows the creation of a new type of dynamic emitter.
MPxSpringNode	Allows the creation of a new type of dynamic spring.
MPxManipContainer	Allows the creation of a new type of manipulator.
MPxSurfaceShape	Allows the creation of a new DAG object. This is often used to create a new type of *shape* (i.e. something other than a NURBS or mesh surface), but can also be used in many other ways.

Name	Description
MPxObjectSet	Allows for the creation of a new type of set.
MPxHwShaderNode	Allows the creation of a new hardware shader.
MPxTransform	Allows the creation of new types of transformation matricies.

The basics

The following is a simple user-defined dependency graph node subclasses from the MPxNode parent class which takes a floating point number as input, takes the sine of it, and outputs the result.

```
#include <string.h>
#include <iostream.h>
#include <math.h>

#include <maya/MString.h>
#include <maya/MFnPlugin.h>
```

This is still a plug-in so you still need MFnPlugin.h. However, you use a different method to register a node than a command.

```
#include <maya/MPxNode.h>
#include <maya/MTypeId.h>
#include <maya/MPlug.h>
#include <maya/MDataBlock.h>
#include <maya/MDataHandle.h>
```

These header files are used by most plug-in dependency graph nodes.

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

There are a number of different types of attributes (which you will be introduced to) and the ones you need are dependent on the type of node you write. For this example, only numeric data is used.

```
class sine : public MPxNode
{
```

User-defined dependency graph nodes are derived from the MPxNode class.

```
public:
                    sine();
```

constructor

The constructor for the node is called whenever an instance of this node is created. This can either be when the createNode command is called, the MFnDependencyNode::create() method is invoked, etc.

```
    virtual         ~sine();
```

destructor

The destructor is only called when the node is truly deleted. Because of the undo queue in Maya, deleting the node does not actually cause the node's destructor to be called, so if the deletion is undone, the node can be returned without recreating it. Generally, a deleted node's destructor is only called when the undo queue is flushed.

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

compute() method

The compute() method is the brains of a node. It does the actual work of the node using the inputs on the node to generate its outputs.

```
    static  void*        creator();
```

creator() method

The creator() method serves the same purpose as the creator method on commands. It allows Maya to instantiate instances of this node. It is called every time a new instance of the node is requested by either the createNode command or the MFnDependencyNode::create() method.

```
    static  MStatus      initialize();
```

The initialize() method is called by the registration mechanism for dependency nodes. As a result it is called once immediately after a plug-in containing a user-defined node is loaded. It is used to define the inputs and outputs of the node (for instance, its attributes).

```
public:
    static  MObject input;
    static  MObject output;
```

These two MObjects are the attributes of the sine node. You are free to use any names for a node's attributes—input and output are just used here for clarity.

```
    static  MTypeId id;
```

Each node requires a unique identifier which is used by MFnDependencyNode::create() to identify which node to create, and by the Maya file format.

For local testing of nodes you can use any identifier between 0x00000000 and 0x0007ffff, but for any node that you plan to use for more permanent purposes, you should get a universally unique id from Alias Technical Support. You will be assigned a unique range that you can manage on your own.

```
};

MTypeId     sine::id( 0x80000 );
```

This initializes the node’s identifier to a unique tag. These tags must be unique between all nodes (the tag is used by the file format to recreate the node) and will be assigned to API users by Alias.

```
MObject     sine::input;
MObject     sine::output;
```

The attributes of the node are initialized to NULL values.

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

As mentioned earlier, the creator() method simply returns new instances of this node. In more complex situations where several nodes may need to be interconnected, it is possible to define a single creator for the connected nodes and have this creator allocate and connect all the nodes together.

```
MStatus sine::initialize() {
```

The initialize method is called only once when the node is first registered with Maya. In this method you define the attributes of the node, what data comes in and goes out of the node that other nodes may want to connect to.

```
    MFnNumericAttribute nAttr;
```

This example only uses numeric data so all it’s attributes are numeric and therefore only MFnNumericAttribute is necessary.

```
    output = nAttr.create( “output”, “out”,
            MFnNumericData::kFloat, 0.0 );
    nAttr.setWritable(false);
    nAttr.setStorable(false);
```

The first of these three lines defines the output attribute for the sine node. When defining an attribute, you must specify a long name (four characters or longer) and a short name (no more than three characters) for the attribute. These names are used in MEL scripts and UI editors to identify particular attributes. While it is not necessary, it is generally a good idea if the long name is the same as the C++ identifier for the attribute—in this example they are both called “output”.

The create method also indicates the type of the attribute, in this case a float (MFnNumericData::kFloat) and sets its default value to zero. The names of attributes need only be unique within a node, different nodes may have similarly named attributes.

The next two lines set specific characteristics of this attribute. Since this is the output of the sine node, it is not writable by other nodes. That means it is not possible for the output attribute of another node to be connected to this attribute. Also, because this is an output, it is not necessary to store the output when writing a file, since the output can be generated from the inputs. (It wouldn't cause problems if you stored an output—it would just waste space.)

When instantiating a new node, Maya sets the following characteristics to true:

- readable (can it be used as an output?)
- writable (can it be used as an input?)
- connectable (can another attribute connect to it?)
- storable (can it be stored to a file?)
- settable (can its value be set?)

and the following characteristics to false:

- multi (an array)
- keyable (can be keyframed)
- indeterminant (attribute may or may not be used -- rendering related)
- hidden (not visible in the Attribute Editor)

```
input = nAttr.create( “input”, “in”,
        MFnNumericData::kFloat, 0.0 );
nAttr.setStorable(true);
```

The initialization of the input attribute is similar to the output attribute, but since the value of the input cannot be calculated by the node, it must be stored when the node is stored.

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

The input attribute is added as an attribute to the sine node. The attributeAffects() method is used to indicate when the input attribute affects the output attribute. This knowledge allows Maya to optimize dependencies in the graph in more complex nodes where there may be several inputs and outputs, but not all the inputs affect all the outputs.

```
addAttribute( output );
```

The output attribute is added to the sine node. Since the output attribute is generated it does not affect the input attribute so no attributeAffects() method is required.

```
    return MS::kSuccess;
```

Success is returned to indicate to Maya that the node was successfully initialized. A failure return would stop the initialization, and since the initialization method is called only once, the node would never be usable in the session. Failure returns should be made whenever a resource is unavailable that the node requires.

}

```
sine::sine()  {}
sine::~sine() {}
```

Since this is a very simple node, the constructor and destructor do not do anything.

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

The compute() method is the brains of a dependency graph node doing all the actual work of the node. It takes two arguments. The first is a reference to the plug for which a compute is being requested, and the second is the data block for the node. Plugs and data blocks will be described in more detail in a later section.

```
    MStatus returnStatus;

    if( plug == output )
    {
```

plugs

A plug can be thought of as the recomputed attribute. This test checks which output attribute the recompute is being requested for.

In this simple example, it will only be the output attribute, but in more complex nodes it could be any of the outputs. You should always test that the plug represents a known attribute. For example, someone may attach a dynamic attribute to your node that might have connections. This could cause your compute method to be called when none of your inputs have changed.

```
        MDataHandle inputData = data.inputValue( input,
                &returnStatus );
```

data blocks

The data block contains all the data for this instance of the node. For efficiency this data is kept as a single block, so the data handles are used to reference a piece of the block. In this case the data handle is being set to reference the input attribute.

```
        if( returnStatus != MS::kSuccess )
            cerr << "ERROR getting data" << endl;
        else
        {
            float result = sin( inputData.asFloat() );
```

The data is retrieved from the data handle through the as*() methods on MFnDataHandle. It is vital that the as*() method matches the declared type of the attribute. The input attribute was declared as MFnNumericAttribute::kFloat so the data must be retrieved with asFloat(). Mixing types and retrieve methods will cause fatal errors. For example declaring an attribute as MFnNumericAttribute::kDouble and retrieving it with asFloat() will result in a fatal error.

```
            MDataHandle outputHandle = data.outputValue(
                    output );
```

A new data handle is allocated to be used to reference the piece of the data block for the output attribute.

```
            outputHandle.set( result );
```

The output attribute is then assigned the result of the calculation.

```
            data.setClean(plug);
```

The plug in the data block which caused the recompute is marked clean indicating that it has been newly recomputed.

```
        }
    }

    return MS::kSuccess;
}
```

A successful status return indicates that the computation completed properly.

```
MStatus initializePlugin( MObject obj ) {
    MStatus status;
    MFnPlugin plugin( obj, "My plug-in", "1.0", "Any");
    status = plugin.registerNode( "sine", sine::id,
sine::creator, sine::initialize );
```

The plug-in requires an initializePlugin() and an uninitializePlugin() as with command plug-ins, but rather than using registerCommand(), registerNode() is used to add the node to Maya's database of nodes. The initialize method for the node is called as a result of the call to registerNode. If the initialize method returns a failure code, registerNode will also fail and your node will fail to load.

```
    return status;
}

MStatus uninitializePlugin( MObject obj) {
    MStatus status;
    MFnPlugin plugin( obj );
    status = plugin.deregisterNode( sine::id );

    return status;
}
```

And that's a simple dependency graph node.

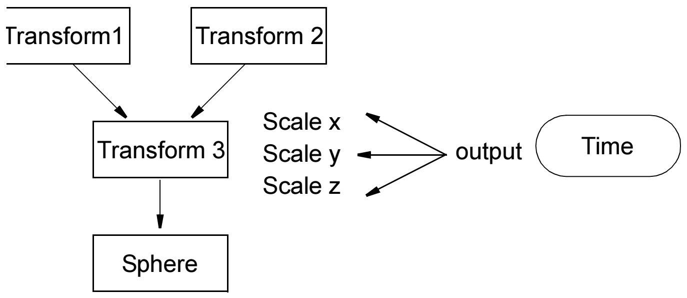
Dependency Graph (DG) nodes

The dependency graph (DG) is a collection of entities connected together. Unlike the DAG, these connections can be cyclic, and do not represent a parenting relationship. Instead, the connections in the graph allow data to move from one entity in the graph to another. The entities in the graph which accept and output data are called dependency graph nodes.

Dependency graph nodes are the part of a dependency graph which perform computations. A node takes in a set of input data (supplied by connections to other nodes, or which are simply supplied to the node) and use them to create a set of output data. Dependency graph nodes are used for almost everything in Maya such as model creation, deformation, animation, simulation, and audio processing.

Most objects in Maya are dependency graph nodes, or networks of nodes (several nodes connected together). For example, DAG nodes are dependency graph nodes, and shaders are networks of nodes.

When dependency graph nodes are connected together they can affect DAG nodes and thus affect what is rendered.



This illustration combines a DAG hierarchy with a dependency graph. Transform1, Transform2, Transform3, and Sphere are all DAG nodes (and also dependency graph nodes) while Time is just a dependency graph node.

The x, y, and z scale parameters of Transform3 are driven by time. Alternatively, you can think of the output of Time being plugged into the x, y, and z scale connections of Transform3. When the animation is played back, the two instances of the sphere increase in size.

The data which flows through the graph can be as simple as numbers, or as complicated as a surface. It can also be a completely user-defined object.

The dependency graph consists of a very complex architecture, and a complete explanation of how it works would require a separate manual. A brief explanation is provided instead.

As noted before, the dependency graph is a directed graph, the edges of the graph connect plugs on different nodes. Data is sent along these edges, and includes basic types such as numbers, vectors, and matrices, and complex types such as curves, surfaces, and user defined types.

As part of the definition of Maya nodes, you are required to specify which input attributes affect which output attributes (via the API this is done with the MPxNode::attributeAffects method).

When an attribute of a node is changed, the dependency graph checks to see if that attribute affects any output. If it does, each of those outputs is marked *dirty*, meaning that its cached value is stale and needs to be recomputed. Then for each of those output attributes, the dependency graph checks to see if they are the source for a connection. If so, then the connection is followed, and the destination attribute is marked dirty also. This process recurs, and at the end all attributes of nodes in the graph which need to be recomputed are marked dirty. It is important to note that at this time *no attributes* have been recomputed, instead the state has just been updated so we know which data is no longer valid. The evaluation and re-computation of those invalid attributes occurs as part of a separate step.

Certain events cause the dependency graph to re-evaluate itself, examples being screen refresh, and animation playback. During a refresh for example, the system will walk down the DAG and for each DAG node check to see whether it needs to be re-evaluated (by checking if any plugs on it are dirty), if so, the compute method of the node affecting the plug will be called. This compute method may be dependent on plugs which may also be dirty, so the affecting nodes’ compute methods would also be called, and in this way the pertinent parts of the DG will be re-evaluated, but only those parts that require re-evaluation.

One optimization is that the DG does not re-evaluate the graph unless it needs to. For example, imagine a revolved surface where there are three nodes, a curve DAG node used as input to the second node, a node which revolves the curve and generates a surface, which is the output to the third node, a DAG node which puts the surface into the DAG. If the input curve was modified, the surface would not be regenerated immediately, it may not happen until the next screen refresh. To make sure that the surface does eventually get rebuilt, modifying the curve would cause all plugs connected to the curve's output plug to be marked dirty, hence the input to the revolve node would be marked dirty (the curve's output plug itself would not be marked dirty since it has just been recomputed). When declaring attributes it's necessary to indicate what attributes affect each other, so in the revolve node, the output attribute is dependent on the input attribute, then marking the input attribute dirty causes the output attribute to be marked dirty. The output of the revolve node is connected to the surface node, marking the revolve node's output dirty marks the surface as being dirty. So, when the DAG is walked during a screen refresh, since the surface is marked dirty, everything that it is dependent on which has also been marked dirty needs to be re-evaluated.

Re-evaluation stops at the first node which doesn't have any dirty inputs. For example if the number of degrees to revolve a curve was changed but the curve itself was not, then rebuilding the revolved surface would cause the revolve node to be recomputed, but the curve would not be affected.

This is a high level description, the actual implementation provides a great deal of intelligence so that unnecessary evaluations are avoided.

The data flowing through the graph is analogous to water flowing through pipes. The pipes themselves are the connections but unless they have data to redirect and modify they are not actually doing anything.

Extending this analogy, the nodes are like taps, showers, fountains, WCs, springs, and water piks. They all do something with the water in their own unique way but they must have water to operate.

An interesting side effect of using the DG is that it can make it difficult to directly affect an object.

For example, consider the sphere in the above figure. If no DG node is connected to Transform3 to affect the sphere's scale, any values set through the UI or API will be the new scale. However, if as in the figure Time affects the scale of the sphere, the effect would be different when Transform3's scale is additionally modified through the UI or API. If you set the scale it would override the scale being set by Time only until the next re-evaluation of the dependency graph, when the scale of the sphere would again be set by Time and the value you set through the UI or API would be lost.

A more complex example would be a revolved surface. What would happen if you tried to move a CV on the generated surface? The CV would be moved to its new position only until the DG is re-evaluated, at which time the CV would be moved back to the position dictated by the revolve node.

However, fine-tuning or “tweaking” a model is a necessary operation for building complex models and scenes. So Maya has designed a mechanism for handling these tweaks. Mesh shapes have an attribute, *pnts*, which stores local changes made to the mesh vertices. Any upstream connection to the mesh shape node which generates a new set of vertices for the mesh will not disturb the pnts attribute. The values in the pnts attribute are added to the coordinates of the mesh. For NURBS surfaces and other control point based nodes, the *controlPoints* attribute stores tweak information. Maya also has implemented a tweak node which will store tweak information for a control point based node. The tweak node is placed between the control point based node and an upstream deforming node which operates on the control points. The tweak node integrates the tweak information in with the deformed control points to generate the final set of control points that is then passed to the shape. Refer to the manual pages for the tweak node as well as the mesh and NURBS surface shape nodes for more information on the attributes which handle tweak information.

Nodes

Nodes are the engines which drive the dependency graph. Data comes in to nodes, they perform an operation on the data, and they make the new data available again. The data comes in through the input plugs (instantiations of the nodes attributes) and goes out through the output plugs. At no time should a node require any additional external data beyond what is available through its plugs.

Attributes and plugs

A node by itself without any means to affect it is not very useful. Modification of a node is done through attributes and plugs.

The attributes of a node define a data interface for the node to the rest of the graph. The only data that nodes pass to one another during evaluation comes and goes through this interface. They indicate what type of data can be accepted, what the name for that data will be (in a long form, or a short form), whether the data can be made into a list, and how the data is allowed to move in and out of the node. Types of data include simple data types such as integer and floating point values, and more complex data types such as points, whole polygonal meshes, and NURBS surfaces.

Plugs are constructs which contain the data for a given attribute. All data access for the attribute is done through the plug. The attribute itself only defines the data type and name for the attribute. Plugs also are the ports for making connections between nodes.

Attribute names must be unique across the entire node hierarchy, that is, across all derived and parent classes, but can be reused between unrelated nodes.

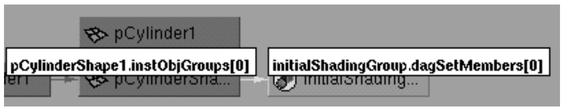
Attributes may have a default value assigned to them. This will be the value that a plug on that attribute will have if its value has not been set. For example, a numeric attribute always has a default value of zero, though this can be changed when creating the attribute. Any of the typed attributes will have a default value equal to the default value of the data type they accept—numeric data defaults to zero, matrix data defaults to the identity matrix, etc.

By default, Maya will automatically arrange the attributes of a node in the attribute editor. If you desire a special arrangement of the attributes of your node, you can write an *attribute editor template* for the node. This is a MEL file on your MAYA_SCRIPT_PATH whose name is of the form AE*{nodeName}*Template.mel that contains a MEL procedure with the name AE*{nodeName}*Template. This procedure contains editorTemplate commands that instruct the attribute editor how to alter the default layout for the attributes in the node.

Complex Attributes

By default, attributes have only one associated plug. These are called simple attributes. An attribute can also be defined as containing an arbitrarily long list of plugs. Attributes of this type are called **array attributes** and the plugs in the array are called **elements**. Each element plug can contain its own value and can have its own connection, and the array can be sparse. The data type of each element is defined to be the type specified by the attribute. Each element in the array is identified by its sparse index into the array.

Maya's Hypergraph and Connection Editor display the index of an element plug in square brackets ([]) after the attribute name.


pCylinder1
pCylinderShape1.instObjGroups[0]
initialShadingGroup.dagSetMembers[0]
pCylinderSha...
initialShading...

Hypergraph

The array of plugs is accessed through another plug called the **array plug**. This plug is returned when asking an attribute for its associated plug (it is not recommended to connect this plug to another array plug). Any attribute can be defined as an array attribute. The specification is made using the MFnAttribute::setArray() method and must be called after calling the create() method for the attribute.

An important distinction must be made between plugs which are defined as array plugs and simple plugs which are defined to contain array data. Both constructs can contain multiple values and are referred to as "arrays."

In the case of a simple array, the data type is defined as an array, such as pointArray or intArray (refer to the reference page for the MFnData class for a description of the allowable array types). The array is treated as a single data item by the plug. When asking for the plug's value, the whole array is returned. If the plug is connected to another plug in the scene, the whole array is passed or retrieved along the connection.

For an array attribute, the data type of the attribute is the data type of a single value, such as an int or a double. The array comes from the list of element plugs, each containing a single data value. Each element is independently connectable. The data items are retrieved by accessing each plug in the array and retrieving the single values stored at the plugs. So there is an advantage and disadvantage to each method. In the single attribute case the data are treated as a single unit and can be moved more efficiently through the dependency graph network, but there is less flexibility in accessing the individual data items. In the array attribute case there is great flexibility as each data item can be accessed and connected, but there is more overhead for the node and the dependency graph network.

Following these rules, one can define an attribute whose data type itself is an array. Such an attribute would contain an array plug whose element plugs each contained a whole array. Each array could be independently accessed and connected. The individual data arrays would each be treated as a single block of data when being retrieved and sent along connections.

Compound attribute

An attribute can also be defined as a collection of other attributes. Such an attribute is called a compound attribute and the members of its collection are called children. Compound attributes are not defined as containing a particular data type—they are defined as the set of attributes which make up the collection.

Child attributes

Each child attribute is treated as any other attribute. Child attributes have names and data types and can be defined as array attributes or compound attributes. A plug is associated with the compound attribute itself and is referred to as the parent plug to the members of the compound attribute.

Each child attribute also follows the same rules of connectability. A child is independently connectable, and if a child attribute is defined as an array attribute, its element plugs are also independently connectable. The parent plug of a compound attribute can be connected to another node's compound parent plug as long as the child attributes of each plug are defined identically. In this case, the data for all of the child plugs is sent along the connection.

If a compound attribute is specified as an array attribute, then each element plug of the array will contain children plugs for each of the members of the compound attribute. The element plug will be the parent plug.

Compound attributes are created using the MFnCompoundAttribute class. This class contains methods for specifying the child attributes which will be contained in the compound attribute. Refer to the reference page of MFnCompoundAttribute for more information.

Dynamic Attributes

Dynamic attributes are used to attach blind data to a node. Every node initially has a set of attributes defined. You may, however, want to add new attributes to either a single node or to all nodes of a given type. These attributes are called dynamic attributes.

A dynamic attribute is treated much like any other attribute. The main difference is that someone is responsible for allocating and deallocating it since it will not be statically created.

The following is a code fragment taken from the blindShortDataCmd example. It creates a simple dynamic attribute to contain a short which it then attaches to a selected dependency graph node. The blindComplexDataCmd example demonstrates adding user-defined data as a dynamic attribute.

```
MFnNumericAttribute fnAttr;
const MString fullName( “blindData” );
const MString briefName( “bd” );
double attrDefault = 99;
MObject newAttr = fnAttr.create( fullName, briefName,
MFnNumericData::kShort,
    attrDefault, &stat );
```

This creates a new numeric attribute called “blindData” with a short name of “bd” which has a default value of 99. When using dynamic attributes as blind data the name of the attribute must be unique so that you and someone else do not create attributes which conflict with each other.

```
stat = fnDN.addAttribute(
        newAttr,MFnDependencyNode::kLocalDynamicAttr);
if ( MS::kSuccess != stat ) {
    cerr << "Error adding dynamic attribute" << endl;
}
```

These few lines add the attribute to the selected dependency graph node (fnDN is an instance of MFnDependencyNode that was initialized elsewhere). Note the use of MFnDependencyNode::kLocalDynamicAttr which indicates that this new attribute is a dynamic attribute.

Data blocks

A data block is the storage object for a node's data. It maintains the received and sent data of the node, and is only valid during the execution of a node's compute function. A pointer to the data block must not be maintained after the compute function exits.

The dependency graph can be a bottleneck when rendering if care is not taken. Since a node can be computed several times per pixel its important that it be as efficient as possible.

It may seem that getting at the data being received or sent by a node is a little complex at first, but this is necessary to provide a fast mechanism.

Data handles

A data handle is a reference into the data block which references a particular piece (attribute or plug) of the node's data. The type used to get or set the data on an attribute or plug must match the type of the attribute. For example, if an attribute is declared as an integer attribute, you should not use the data handle to get or set it as a float, matrix, or even a short—you should only get or set it as an integer.

Data creators

Data creators are classes used to create data to be put into a data block, most likely to an output plug of a node. Data creators are not required for simple data types such as integers and floating point values, but are necessary for heavier data such as mesh shapes and NURBS surfaces.

The classes allow Maya to more efficiently modify and transfer the data along dependency graph connections. The subclasses of MFnData are the data creator classes. For the data creator classes that create heavier data which also corresponds to shape nodes, such as mesh shapes and NURBS surfaces, the MFnDependencyNode subclasses pertaining to the data type are used to fill the data block item with data. For instance, the MFnMeshData class is used to create a new mesh data block item, but the MFnMesh class must be used to fill the item with vertices and polygons.

Some operations may behave differently depending on the data block item. That is because these shape-related dependency node function set classes can be used to access data block items that have come from connections to other nodes in the graph as well as to access data block items that have been created locally within the node.

Note All operations which can return world space information, such as MFnMesh::getPoint(), become invalid when applied to data block items that do not come from shape nodes. The shape node is required to be able to determine the transformation for calculating the world space position. Caution must be used when using these methods.

Compute methods

The implementation of a dependency graph node is little more than assigning attributes and coding a compute method. The DG node is a C++ class, with the attributes being static class members. Instances of the class make up individual instances of the DG node in the graph. Plugs are separate objects which indicate the state of a particular attribute for a node.

The compute method takes its input from the attributes and plugs of an instance of the node (the default value of an attribute if there is no plug for it, or the specific value of the plug). When requesting information from a plug, the plug determines if it is in a proper state (for instance, clean). If it is not, it automatically asks the node to which it is connected to update itself. This can propagate all the way through the DG. Once all the plugs are clean, the node takes the data from the plugs (through the data blocks and data handles), computes its results and sends the output to its output attributes.

A node must not know anything outside of its attributes and plugs. For instance, it should not alter the DAG or other dependency graph nodes. Altering the DAG or another node in the DG could prompt a new DG evaluation which could cause your node to re-evaluate, causing you to alter again, which forces a new DG evaluation putting you in an infinite loop. If for some reason a node needs to know something about its environment, that knowledge should be provided as an attribute that can be connected to. Using any outside data will almost certainly violate one of the following:

- multiprocessing-friendliness
- evaluating at an alternate context (for example, at time T)
- node instancing

- vectorization of computations

The compute() method in every node must operate with only the data that is present in its parameters. If any other data is required then it should be made into an attribute. Even if it is only temporarily cached information to speed up computation, it could be useful as a hidden attribute to speed up the first refresh after file retrieval.

Also, a node must not save a reference to any of the data coming in on a plug. If, for example, surface geometry comes in on a plug, do not save a pointer to this geometry. The data coming in on a plug is transitory and may cease to exist without you knowing it. You are only guaranteed that it will exist during the execution of your node’s compute function.

A more complex example

The following are fragments from a slightly more complex example than the previous example. The code fragments are taken from the simpleLoftNode example which takes a curve as input and generates a surface.

```
MObject        simpleLoft::inputCurve;
MObject        simpleLoft::outputSurface;
```

This example has only two attributes, an input curve and an output surface.

```
MStatus simpleLoft::initialize()
{
    MStatus stat;
    MFnTypedAttribute   typedAttr;
```

The previous example used MFnNumericAttribute since the attributes were simply floating point numbers. Since this example uses more complex data, MFnTypedAttribute is used.

```
    inputCurve = typedAttr.create( “inputCurve”, “in”,
        MFnData::kNurbsCurve, &stat );
    if( !stat )
        return stat;
```

This creates an attribute to hold curve objects. The Type enumeration in MFnData lists the types of data which can be created using a typed attribute. This list includes curves, surfaces, meshes, strings, user-defined data, etc.

```
    outputSurface = typedAttr.create( “outputSurface”, “out”,
        MFnData::kNurbsSurface, &stat );
    if( !stat )
        return stat;
```

This creates an attribute to hold the generated surface object.

```
    typedAttr.setStorable( false );
```

Since the surface is a generated object, it isn’t necessary to store it when storing the node to a file.

```
    addAttribute( inputCurve );
    addAttribute( outputSurface );
    attributeAffects( inputCurve, outputSurface );
```

Finally, the two attributes are added to the node and attributeAffects() is used to indicate that when the input curve is modified the resulting surface will have to be regenerated.

```
    return MS::kSuccess;
}

MStatus simpleLoft::compute( const MPlug& plug, MDataBlock&
data )
{
    MStatus stat;

    if ( plug == outputSurface )
    {
```

This ensures that the computation of the node is only done for the appropriate attribute.

```
        MDataHandle inputData = data.inputValue( inputCurve,
&stat );
        if( !stat )
            return stat;
```

As before, the data block contains all the data for the node in an efficient manner. The data handle is required to access this data.

```
        else
        {
            MObject curve = inputData.asNurbsCurve();
            MFnNurbsCurve curveFn( curve, &stat );
            if( !stat )
                return stat;
```

With the data handle you can then get the input curve which you can then pass on to an MFnNurbsCurve function set to operate on.

```
            else
            {
                MDataHandle surfHandle = data.outputValue(
                    outputSurface );
                if( !stat )
                    return stat;
```

A second data handle is used to access the surface’s portion of the data block.

```
                MFnNurbsSurfaceData dataCreator;
                MObject newSurfData = dataCreator.create(
                        &stat );
                if ( !stat )
                    return stat;
```

Notice that you don’t use MFnNurbsSurface in this example, but rather MFnNurbsSurfaceData. This is necessary since you want to create a data object to pass through the dependency graph and not a DAG object. This will always be the case when creating objects in dependency graph nodes. There are special nodes used which connect surface data into the DAG, and therefore any node that you create which creates geometry need not also create a DAG node for it, just the data.

```
                MObject newSurf = loft( curve, newSurfData,
                       stat );
                if( !stat )
                    return stat;
```

This calls some user-written code which creates a lofted surface from the curve. The method would use MFnNurbsSurface to operate on the surface data object. (MFnNurbsSurface determines whether it is operating on a surface in the DAG or not. If not, some of it’s methods will not succeed. For example, since the surface data object isn’t in the DAG, determining the world position of it does not make sense, so that method would fail.)

```
                surfHandle.set( newSurfData );
```

Add the new surface to the data block so that the output changes.

```
                stat = data.setClean( plug );
                if( !stat )
                    return stat;
```

Tell the system that the plug has been successfully recomputed and is now clean.

```
            }
        }
    }
else
        {
            cerr << "unknown plug\n";
            return MS::kUnknownParameter;
        }
```

You must return MS::kUnknownParameter if the plug is not recognized or if there is no computation occurring on a given keyable plug. This causes the compute of the base class to be called which will implement default data handling and cause:

```
        return MS::kSuccess;

    }
```

MPxNode and its derived classes

The MPxNode class is the parent class from which the other functionally-specific MPx classes are defined. These derived classes possess specific knowledge corresponding to their functional area for integrating your class into Maya. For example, the MPxLocator class knows that it must draw itself in different colors depending on whether it is selected or not. Each parent class defines a set of attributes which Maya expects and can automatically connect to when your subclass is used in a scene. The compute() method that you will define with your subclass can use these attributes as desired. Refer to the manual pages for the various MPx parent classes for a listing and description of the predefined attributes.

The MPxManipContainer (see Chapter 7, “Manipulators”) and MPxSurfaceShape classes (see Chapter 8, “Shapes”) are complex classes which require more in depth descriptions.

MPxLocatorNode

This parent class is a DAG node which allows you to draw three dimensional graphical elements in the Maya scene. The elements are associated with a location in the scene which can be manipulated using the standard Maya manipulators.

This class can be used for defining entities which have a location in space but no explicit shape, such as a new type of light source, a destination point for the behavior of some other entity, or a construction location for a shape not yet created. The graphical elements drawn by the locator are not rendered. The MPxLocator class itself draws a default graphic, but a draw() method is provided which can be implemented in your subclass to perform more specialized drawing.

MPxDeformerNode

This class allows you to take an input geometry shape and deform it. Maya requires a special protocol for performing deformations, and provides a few special methods which you implement.

The first is the deform() method. The actual deformation does not take place in the compute() method of deformer nodes but through an internal mechanism which calls the deform() method.

Two other methods, accessoryAttribute() and accessoryNodeSetup() are also defined by the parent class. Accessories are the geometry shapes that you select and manipulate to affect the deformation. This can be a set of wireframe lines, a set of NURBS curves, or any other geometry you desire which can intuitively convey the function of your deformer and affect useful deformations.

Accessories are not required by deformers. Your deformer can function solely based on the predefined input attributes of the MPxDeformerNode class and/or any other attributes you define for your subclass. In this case, simply do not implement the two accessory methods.

MPxIkSolverNode

Inverse Kinematics (IK) describes a class of algorithms for animating linkages of rigid bodies based on a set of goals and constraints placed on the linkage. An IK solver is a mathematical procedure for finding a set of rotations and offsets for the links in order to satisfy the goals and constraints.

Solvers can be tailored for different types of linkages or to behave a certain way, such as minimizing the motion of certain joints or keeping certain joint angles between certain ranges. Maya defines a set of solvers which can be used in different situations.

The MPxIkSolverNode allows you to write your own solver and use it on linkages you build in Maya. As with the MPxDeformerNode class, the real computation of the node is not done in the compute() method, but in the doSolve() method. There are several other methods which must be defined when subclassing a new solver. Refer to the MPxIkSolverNode manual page for a description of these methods.

MPxFieldNode

This class lets you define your own dynamic field to affect other geometry in the scene. This node follows the normal dependency graph rules of evaluation in that the work of the node is done in the compute() method.

MPxEmitterNode

Emitters are nodes which emit particles in to a Maya scene. Different emitters emit particles in different ways, such as emitting only in one direction, emitting slowly, or emitting randomly from the surface of a sphere. Once the particle is emitted, it is no longer controlled by the emitter node. The MPxEmitterNode class lets you define the behavior of how particles are emitted. The node follows the normal dependency graph rules of evaluation in that the work of the node is done in the compute() method.

MPxSpringNode

Springs are forces which interact between two endpoints which have mass. Maya defines a default spring force which follows a traditional mathematical model of springs. You can define a new behavior for applying a force between two points in a scene by subclassing from MPxSpringNode. The predefined attributes supply all of the standard spring constants as well as the positions and masses of the endpoints. The work of the node is done through the applySpringLaw() method instead of the compute() method.

MPxObjectSet

This class can be used to implement new kinds of sets within Maya that can have selectable/manipulatable components and behave in a similar manner to the objectSet node included in Maya.

MPxHwShaderNode

MPxHwShaderNode allows the creation of user-defined hwShaders. A hwShader is a node which takes any number of input geometries, deforms them and places the output into the output geometry attribute.

MPxTransform

MPxTransform allows the creation of user defined transform nodes. User defined transform nodes can introduce new transform types or change the transformation order. They are designed to be an extension of the standard Maya transform node and include all of the normal transform attributes. Standard behaviors such as limit enforcement and attribute locking are managed by this class, but may be overridden in derived classes. Although it is not a node, the MPxTransformationMatrix class is used in conjunction with MPxTransform to add custom transformation matrices to Maya

6 Writing a Shading Node

Developer Plug-in API

Write a shading node

Overview of shading node plug-ins

A shading node plug-in is written as a Maya Dependency Graph Node. A basic shading node contains attributes that are treated as inputs and outputs, where each shading node must have an output so that it can be connected in the Dependency Graph.

Writing a shading node plug-in

Custom dependency graph nodes include the header file:

<maya/MPxNode.h>

and then derive from the class MPxNode. To build your own shading node as a plug-in to Maya, you follow the same guidelines for making a Dependency Node (see "Dependency Graph (DG) nodes" in Chapter 5, "Dependency graph plug-ins").

You request specific rendering information for your plug-in through pre-defined attributes provided during the rendering process. (See "Appendix C: Rendering attributes" for a complete list of rendering specific attributes and their corresponding names.)

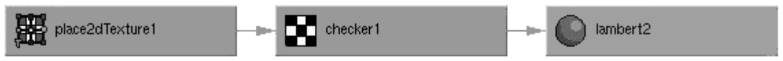
There are currently five different types of shading nodes that can be created with Maya.

- Surface shaders
- Light shaders
- Texture shaders
- Displacement shaders
- Volumetric shaders

Shading nodes in Maya are connected to form shader networks. Any Maya dependency node can be part of a shader network. Instead of hard-coding many effects into each shader, texture and light, the same functionality is available in Maya through Utility nodes which can implement the same effects and more. In addition, shading nodes can contribute to more than one shading network at a time.

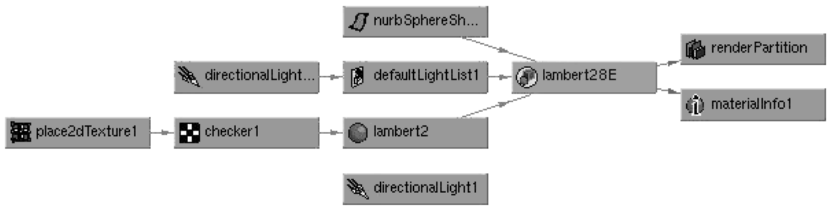
The following example shows Maya's Hypergraph display of a simple Lambert shading node (highlighted) that has a texture node as an input connected to the Lambert node's color attribute.

The illustration also shows a placement node connected to the checker texture for transforming texture coordinates. Each arrow denotes an attribute connection between two dependency nodes.

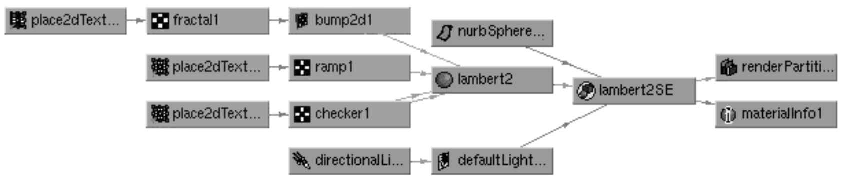


A shading network is not complete until is has been connected to a Shading Group. A Shading Group is a "Renderable Set" which contains a list of objects and/or components of objects which will all be rendered using the same shader network. In addition to its list of objects, the shading group also maintains a connection to a shading network. The shading group is the connection point between objects in the scene, and a shader network which describes how they should be rendered.

The following shows the shader network connected to a shading group (highlighted) and assigned to a default NURBS sphere. Notice the directional light in the scene used for illumination.



As you build a complex shader network with various shading nodes, you can see that the all the connections lead into the Lambert nodes inputs (highlighted) and the computed result of the Lambert shading node's output is connected to a shading group.



Anatomy of a shading node plug-in

Maya Shading node plug-ins include the header file <maya/MPxNode.h> and then derive from the class MPxNode. While this command has a rich set of methods, only a few are actually necessary to create a working shading node.

Constructor

Initializes elements of the new class itself.

Destructor

Deletes anything created by the class.

Creator

This static method is responsible for actually creating instances of your new class (which is derived from MPxNode). When you register a new object you are actually registering its creator() method which Maya can then call to allocate a new instance of an object. In virtually all cases, is should look like:

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

initializePlugin/uninitializePlugin

The first of these methods is called by Maya when the plug-in is loaded. Its purpose is to create an instance of the MFnPlugin class (initialized with the MObject passed to the routine) and call register methods in that class to inform Maya what it is capable of doing.

Important! Both initializePlugin() and uninitializePlugin() must be present in all plug-ins. If both or either is absent the plug-in will not be loaded.

initialize

All the attributes of your new node are declared as static MObject members of the derived class. The initialize method is responsible for making MFnAttribute calls to actually provide the type information on the attributes. In addition, it sets default values, ranges etc. Like the creator function, this is a static method on the class, and will only be called once by Maya.

Id String

One of the required attributes of your node has to be of the type MTypeID. This maps to Maya's internal IFF flag, and must be unique. The value of this attribute is set in the MTypeID constructor.

For local node testing, you can use any identifier between 0x00000000 and 0x0007ffff, but for any node that you plan to use for permanent purposes, you should get a universally unique id from Alias Support. They will assign you a unique range that you can then manage on your own

compute method

This is just like the compute method for an internal dependency node. It is passed a Data Handle to a Data Block and is responsible for extracting its input attributes from the datablock in order to compute the new values of the requested output attributes.

InterpNode example code walkthrough

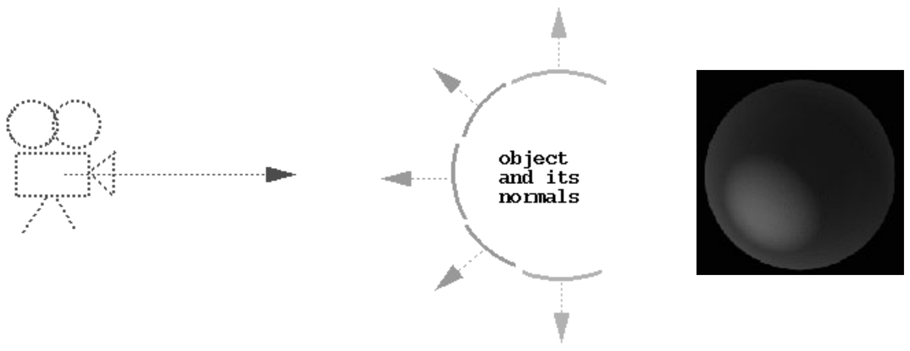
Shading node plug-ins rely on the usage of Compound Attributes and Simple Attributes. The mapping of data between rendering samplers and shading networks is by attribute name. This approach is straightforward and easy to learn and remember, and general enough to work with both the present rendering requirements and future enhancements.

All rendering attributes for which a plug-in is interested has been pre-computed for the current sample being considered. The “datablock” argument that is passed into the plug-in's compute() method contains the rendering attribute information the node has requested. When the plug-in is evaluated by the dependency graph it also passes in a “plug” argument for the specific attribute it wants to evaluate. To optimize evaluations, you need to check for only the output attributes you defined in your plug-in.

This example plug-in node has 20 attributes (aside from its id attribute).

- Two attributes are color input attributes that are built as a compound attribute from three float attributes that represent red, green, and blue (eight total attributes).
- One attribute is a color output attribute that is built as a compound attribute from three float attributes that represent red, green, and blue (four total attributes).
- One attribute is a surface normal that is built as a compound attribute from three float attributes that represent the vector components in x, y, and z (four total attributes).
- One attribute is the position of the geometry in camera space built as a compound attribute from three float attributes that represent the current sample point in x, y, and z (4 total attributes).

The node interpolates between two colors based on the direction of the surface normal it gets from the datablock, and uses the compute() method in that class to derive a result color that is placed into the output color attribute.



Derivation

```
class InterpNode : public MPxNode
{
public:
                InterpNode();
  virtual       ~InterpNode();

  virtual MStatus compute( const MPlug&, MDataBlock& );
  static  void *  creator();
  static  MStatus initialize();
  static  MTypeId id;

protected:
  static MObject InputValue;
  static MObject color1R,color1G,color1B,color1;
  static MObject color2R,color2G,color2B,color2;
  static MObject aNormalCameraX, aNormalCameraY,
    aNormalCameraZ, aNormalCamera;
  static MObject aPointCameraX, aPointCameraY,
    aPointCameraZ, aPointCamera;
  static MObject aOutColorR, aOutColorG, aOutColorB,
    aOutColor;

};
MObject InterpNode::InputValue;
MObject InterpNode::color1R;
MObject InterpNode::color1G;
MObject InterpNode::color1B;
MObject InterpNode::color1;
MObject InterpNode::color2R;
MObject InterpNode::color2G;
MObject InterpNode::color2B;
MObject InterpNode::color2;
MObject InterpNode::aNormalCameraX;
MObject InterpNode::aNormalCameraY;
MObject InterpNode::aNormalCameraZ;
MObject InterpNode::aNormalCamera;
MObject InterpNode::aPointCameraX;
MObject InterpNode::aPointCameraY;
MObject InterpNode::aPointCameraZ;
MObject InterpNode::aPointCamera;
MObject InterpNode::aOutColorR;
MObject InterpNode::aOutColorG;
MObject InterpNode::aOutColorB;
MObject InterpNode::aOutColor;
```

Constructor/Destructor

```
InterpNode::InterpNode() { }
InterpNode::~InterpNode() { }
```

Creator

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

initializePlugin/uninitializePlugin

```
MStatus initializePlugin( MObject obj )
{
   const MString UserClassify( “utility/general” );

    MFnPlugin plugin( obj, “Alias”, “1.0”,
            “Any”);
    plugin.registerNode( “Interp”, InterpNode::id,
            InterpNode::creator,
            InterpNode::initialize,
            MPxNode::kDependNode, &UserClassify);

    return MS::kSuccess;
}

MStatus uninitializePlugin( MObject obj)
{
    MFnPlugin plugin( obj );
    plugin.deregisterNode( InterpNode::id );

    return MS::kSuccess;
}
```

initialize

```
MStatus InterpNode::initialize()
```

```
{
     MFnNumericAttribute nAttr;

// Inputs and Attributes
//
// User defined attributes require a long-name and short-
// name that are required to be unique within the node.
// (See the compound attribute color1 named "Sides".)
//
// Rendering attributes that your node wants to get from
// the sampler require them to be defined given the pre-
// defined unique long-name.(See the compound attribute
// aNormalCamera named "normalCamera".)
//
// User defined Attributes are generally something that you
// want to store in the Maya file. The setStorable(true)
// method enables an attribute to be stored into the Maya
// scene file.
//
// Rendering attributes are primarily data that is
// generated per sample and not something that you want to
// store in a file. To disable an attribute from being
// recorded to the Maya scene file use the
// setStorable(false) method.
//
// Simple attributes that represent a range of values can
// enable a slider on the Attribute Editor by using the
// methods setMin() and setMax().
// (See the simple attribute InputValue named "Power".)
//
// Compound attributes that represent a vector of 3 floats
// can enable a color swatch on the Attribute Editor that
// will launch a color picker tool by using the method
// setUsedAsColor(true).
// (See the compound attribute color1 name "Sides".)
//
// Both Simple and Compound attributes can be initialized
// with a default value using the method setDefault().
//
// Attributes by default show up in the Attribute Editor
// and in the Connection Editor unless they are specified
// as being hidden by using the method setHidden(true).
//
// Attributes by default have both read/write access in the
// dependency graph. To change an attributes behaviour you
// can use the methods setReadable() and setWritable(). The
// method setReadable(true) indicates that the attribute
// can be used as the source in a dependency graph
// connection. The method setWritable(true) indicates that
// the attribute can be used as the destination in a
// dependency graph connection.
// (See the compound attribute aOutColor named “outColor”
// below. It has been marked as a read-only attribute since
// it is the computed result of the node, it is not stored
// in the Maya file since it is always computed, and it is
// marked as hidden to prevent it from being displayed in
// the user interface.)
//
//


// User defined input value

    InputValue = nAttr.create( “Power”, “pow”,
            MFnNumericData::kFloat);
    nAttr.setDefault(1.0f);
    nAttr.setMin(0.0f);
    nAttr.setMax(3.0f);
    nAttr.setStorable(true);

// User defined color attribute
    color1R = nAttr.create( “color1R”, “c1r”,
            MFnNumericData::kFloat);
    color1G = nAttr.create( “color1G”, “c1g”,
            MFnNumericData::kFloat);
    color1B = nAttr.create( “color1B”, “c1b”,
            MFnNumericData::kFloat);

    color1 = nAttr.create( “Sides”, “c1”, color1R, color1G,
            color1B);
    nAttr.setStorable(true);
    nAttr.setUsedAsColor(true);
    nAttr.setDefault(1.0f, 1.0f, 1.0f);

    color2R = nAttr.create( “color2R”, “c2r”,
            MFnNumericData::kFloat);
    color2G = nAttr.create( “color2G”, “c2g”,
            MFnNumericData::kFloat);
    color2B = nAttr.create( “color2B”, “c2b”,
            MFnNumericData::kFloat);

    color2 = nAttr.create( “Facing”, “c2”, color2R,
            color2G, color2B);
    nAttr.setStorable(true);
    nAttr.setUsedAsColor(true);
    nAttr.setDefault(0.0f, 0.0f, 0.0f);


// Surface Normal supplied by the render sampler
```

```
    aNormalCameraX = nAttr.create( “normalCameraX”, “nx”,
            MFnNumericData::kFloat);
    nAttr.setStorable(false);
    nAttr.setDefault(1.0f);

    aNormalCameraY = nAttr.create( “normalCameraY”, “ny”,
            MFnNumericData::kFloat);
    nAttr.setStorable(false);
    nAttr.setDefault(1.0f);

    aNormalCameraZ = nAttr.create( “normalCameraZ”, “nz”,
            MFnNumericData::kFloat);
    nAttr.setStorable(false);
    nAttr.setDefault(1.0f);

    aNormalCamera = nAttr.create( “normalCamera”,”n”,
            aNormalCameraX,
            aNormalCameraY, aNormalCameraZ);
    nAttr.setStorable(false);
    nAttr.setHidden(true);

// Point on surface in camera space, will be used to compute
view vector

    aPointCameraX = nAttr.create( “pointCameraX”, “px”,
            MFnNumericData::kFloat);
    nAttr.setStorable(false);
    nAttr.setDefault(1.0f);

    aPointCameraY = nAttr.create( “pointCameraY”, “py”,
            MFnNumericData::kFloat);
    nAttr.setStorable(false);
    nAttr.setDefault(1.0f);

    aPointCameraZ = nAttr.create( “pointCameraZ”, “pz”,
            MFnNumericData::kFloat);
    nAttr.setStorable(false);
    nAttr.setDefault(1.0f);

    aPointCamera = nAttr.create( “pointCamera”,”p”,
            aPointCameraX,
            aPointCameraY, aPointCameraZ);
    nAttr.setStorable(false);
    nAttr.setHidden(true);

// Outputs

    aOutColorR = nAttr.create( “outColorR”, “ocr”,
            MFnNumericData::kFloat);
    aOutColorG = nAttr.create( “outColorG”, “ocg”,
            MFnNumericData::kFloat);
    aOutColorB = nAttr.create( “outColorB”, “ocb”,
            MFnNumericData::kFloat);
    aOutColor  = nAttr.create( “outColor”,   “oc”,
            aOutColorR, aOutColorG, aOutColorB);
    nAttr.setStorable(false);
    nAttr.setHidden(false);
    nAttr.setReadable(true);
    nAttr.setWritable(false);

    addAttribute(InputValue);
    addAttribute(color1R);
    addAttribute(color1G);
    addAttribute(color1B);
    addAttribute(color1);
    addAttribute(color2R);
    addAttribute(color2G);
    addAttribute(color2B);
    addAttribute(color2);
    addAttribute(aNormalCameraX);
    addAttribute(aNormalCameraY);
    addAttribute(aNormalCameraZ);
    addAttribute(aNormalCamera);
    addAttribute(aPointCameraX);
    addAttribute(aPointCameraY);
    addAttribute(aPointCameraZ);
    addAttribute(aPointCamera);
    addAttribute(aOutColorR);
    addAttribute(aOutColorG);
    addAttribute(aOutColorB);
    addAttribute(aOutColor);

    attributeAffects (InputValue, aOutColor);
    attributeAffects (color1R, color1);
    attributeAffects (color1G, color1);
    attributeAffects (color1B, color1);
    attributeAffects (color1,  aOutColor);
    attributeAffects (color2R, color2);
    attributeAffects (color2G, color2);
    attributeAffects (color2B, color2);
    attributeAffects (color2,  aOutColor);
    attributeAffects (aNormalCameraX, aOutColor);
    attributeAffects (aNormalCameraY, aOutColor);
    attributeAffects (aNormalCameraZ, aOutColor);
    attributeAffects (aNormalCamera,  aOutColor);
    attributeAffects (aPointCameraX, aOutColor);
    attributeAffects (aPointCameraY, aOutColor);
    attributeAffects (aPointCameraZ,  aOutColor);
    attributeAffects (aPointCamera, aOutColor);
```

```
    return MS::kSuccess;
}
```

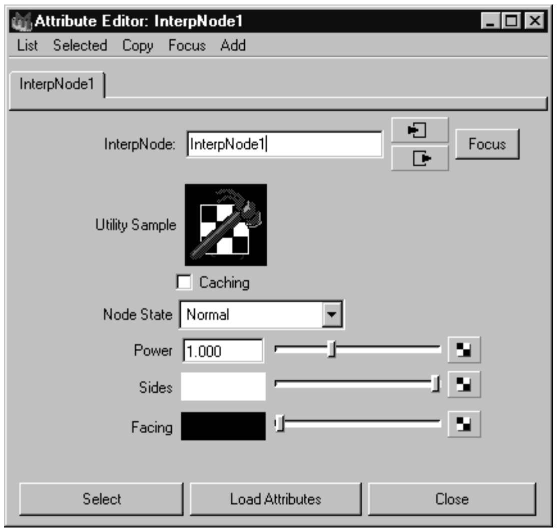
Id String

```
MTypeId InterpNode::id( 0x81005 );
```

compute method

```
MStatus InterpNode::compute( const MPlug& plug, MDataBlock&
    block )
{
    int k=0;
    float gamma,scalar;

    k |= (plug == aOutColor);
    k |= (plug == aOutColorR);
    k |= (plug == aOutColorG);
    k |= (plug == aOutColorB);
    if (!k) return MS::kUnknownParameter;

    MFloatVector resultColor(0.0,0.0,0.0);

    MFloatVector&  Side = block.inputValue( color1 ).
            asFloatVector();
    MFloatVector&  Face = block.inputValue( color2 ).
            asFloatVector();
    MFloatVector&  surfaceNormal = block.
            inputValue( aNormalCamera ).
                asFloatVector();
    MFloatVector&  viewVector = block.
            inputValue( aPointCamera ).
                asFloatVector();
    float power = block.inputValue( InputValue ).asFloat();


    // Normalize the view vector
    double d = sqrt((viewVector[0] * viewVector[0]) +
                    (viewVector[1] * viewVector[1]) +
                    (viewVector[2] * viewVector[2]));

    if (d != (double)0.0) {
        viewVector[0] /= d;
        viewVector[1] /= d;
        viewVector[2] /= d;
    }

    // find dot product
    float scalarNormal = ((viewVector[0]*surfaceNormal[0])
            + (viewVector[1]*surfaceNormal[1])
            + (viewVector[2]*surfaceNormal[2]));
    // take the absolute value
    if (scalarNormal < 0.0) scalarNormal *= -1.0;

    // Use InputValue to change interpolation
    //     power == 1.0 linear
    //     power >= 0.0 use gamma function
    //
    if (power > 0.0) {
        gamma = 1.0 / power;
        scalar = pow(scalarNormal,gamma);
    }
    else { scalar = 0.0; }


    // Interpolate the colors
    MFloatVector interp(0.0,0.0,0.0);
    interp[0] = scalar * (Face[0] - Side[0]);
    interp[1] = scalar * (Face[1] - Side[1]);
    interp[2] = scalar * (Face[2] - Side[2]);

    resultColor[0] = Side[0] + interp[0];
    resultColor[1] = Side[1] + interp[1];
    resultColor[2] = Side[2] + interp[2];

    // set ouput color attribute
    MDataHandle outColorHandle = block.
            outputValue( aOutColor );
    MFloatVector& outColor = outColorHandle.
            asFloatVector();
    outColor = resultColor;
    outColorHandle.setClean();

    return MS::kSuccess;
}
```

Attribute Editor view for InterpNode Example



InterpNode: InterpNode1

Caching

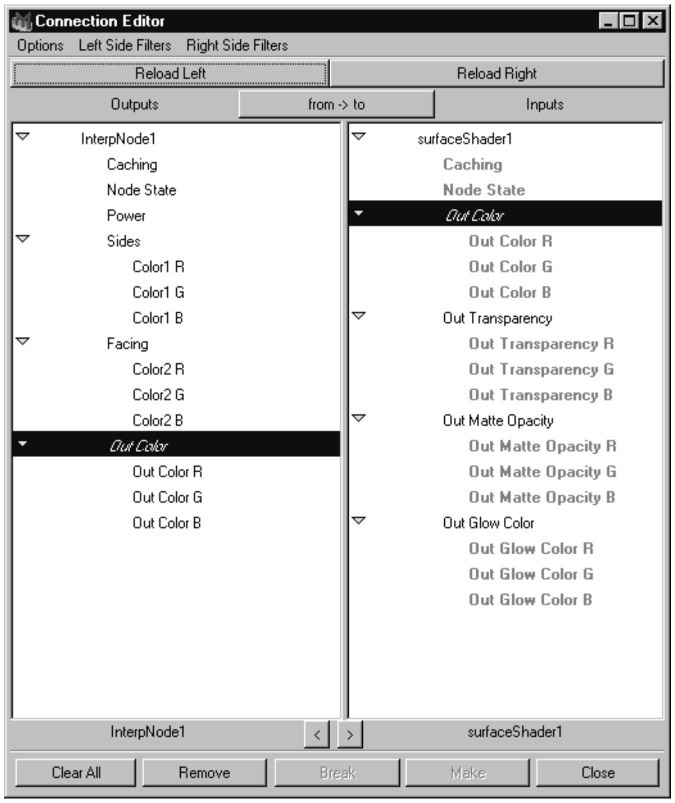
Node State: Normal

Power: 1.000

Sides

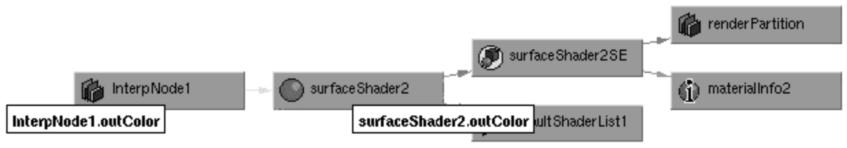
Facing

Connection Editor view of an InterpNode connection



Outputs	Inputs
InterpNode1	surfaceShader1
Caching	Caching
Node State	Node State
Power	Out Color
Sides	Out Color R
Color1 R	Out Color G
Color1 G	Out Color B
Color1 B	Out Transparency
Facing	Out Transparency R
Color2 R	Out Transparency G
Color2 G	Out Transparency B
Color2 B	Out Matte Opacity
Out Color	Out Matte Opacity R
Out Color R	Out Matte Opacity G
Out Color G	Out Matte Opacity B
Out Color B	Out Glow Color
	Out Glow Color R
	Out Glow Color G
	Out Glow Color B
InterpNode1	surfaceShader1

Hypergraph view of an InterpNode connection



Shading nodes classification

When registering your new shading node in Maya, you can assign a classification to your node that will determine where it appears in the Create Render Node interface. Each classification corresponds to a Tab and Frame in which it appears.

The following is a list of classification strings and where they appear in the interface when you use them.

Tab	**Frame**	**Classification String**
Textures	2D Textures 3D Textures Environment Textures	“texture/2d” “texture/3d” “texture/environment”
Materials	Surface Materials Volumetric Materials Displacement Materials	“shader/surface” “shader/volume” “shader/displacement”
Lights	Lights	“light”
Utilities	General Utilities Color Utilities Particle Utilities Image Planes Glow	“utility/general” “utility/color” “utility/particle” “imageplane” “postprocess/opticalFX”

Implicit connections and the Create Render Node window

When you create a rendering node using Create Render Node, you are really executing embedded commands that are used to create a shading node and connect them together. The creation command is "shadingNode" and the connection command is "connectAttr". If you use the commands in the command shell window, no auxiliary nodes are created. All auxiliary nodes are created by the user interface.

The following is a complete description of what the buttons in the Create Render Node window do. The commands are listed by what they execute; this shows you what nodes get created, and how they are connected. For some classifications, the behavior is dependent on the status of check boxes or radio buttons in the window.

Shaders

shader/surface

(for example, blinn)

1. "shadingNode -asShader blinn;"

   The command creates the 'blinn' node, connects its ".message" attribute to the ".shaders" attribute on the defaultShaderList1 node - this lets the multilister know that the node is a shader.

   If the "With Shading Group" check box is checked, then the following also occurs:

2. "sets -renderable true -noSurfaceShader true -empty -name blinn1SG;"

   "connectAttr -f blinn1.outColor blinn1SG.surfaceShader;"

   These command create a new shadingGroup and make the shading group's surface shader the newly created blinn node.

shader/volume

(for example, lightFog)

1. "shadingNode -asShader lightFog;"

   Same as above, used to let the multilister know that the node is a shader.

   If the "With Shading Group" check box is checked, then the following also occurs:

2. "sets -renderable true -noSurfaceShader true -empty -name lightFog1SG;"

   "connectAttr -f lightFog1.outColor lightFog1SG.volumeShader;"

   This will create a new shading group and make the shading group's volume shader the newly created light fog node.

shader/displacement

(for example, displacementShader)

1. "shadingNode -asShader displacementShader;"

Same as above.

If the “With Shading Group” check box is checked, then the following also occurs:

2 “sets -renderable true -noSurfaceShader true -empty -name displacementShader1SG;”

“connectAttr -f displacementShader1.displacement DisplacementShader1SG.displacementShader;”

Same as above, except that the new shader becomes the displacement shader for the new shading group.

Textures

texture/2d

(for example, checker)

1 “shadingNode -asTexture checker;”

Creates the texture node, tells the multilister that it is a texture.

If the “With New Texture Placement” button is checked, then the following are also executed:

2 “shadingNode -asUtility place2dTexture;”

“connectAttr place2dTexture1.outUV checker1.uv;”

Creates a 2d texture placement and connects it to the texture node.

If the “As Projection” button is checked, then the following are also executed:

3 “shadingNode -asTexture projection;”

“shadingNode -asUtility place3dTexture;”

“connectAttr place3dTexture1.wim[0] projection1.pm;”

“connectAttr checker1.outColor projection1.image;”

Creates a projection 3d texture with placement, and connects the newly created 2d texture to its “image” attribute.

If the “As Stencil” button is checked, then the following are also executed:

4 “shadingNode -asTexture stencil;”

“shadingNode -asUtility place2dTexture;”

“connectAttr place2dTexture2.outUV stencil1.uv;”

“connectAttr checker1.outColor stencil1.image;”

Creates a stencil 2d texture with placement, and connects the newly created 2d texture to its “image” attribute.

texture/3d

(for example, brownian)

1. "shadingNode -asTexture brownian;"

   Creates the texture node, tells the multilister that it is a texture.

   If the "With New Texture Placement" button is checked, then the following are also executed:

2. "shadingNode -asUtility place3dTexture;"

   "connectAttr place3dTexture2.wim[0] brownian1.pm;"

   Creates a new 3d texture placement and connects its ".worldInverseMatrix" to the "placementMatrix" attribute of the newly created texture node.

texture/environment

(for example, sphere)

Identical to texture/3d above.

Lights

light

(for example, pointLight)

"shadingNode -asLight pointLight;"

Creates the light node, tells the multilister that it is a light.

Utilities

all utilities

(for example, imagePlane)

"shadingNode -asTexture -asUtility imagePlane;"

Creates the utility node, tells the multilister that it is a utility.

Shading node icons for Hypershade

For Hypershade, create 32x32 icons in xpm format. (You can use imconvert to convert images to xpm.) The icon name must have the preface "render_". For example, for the shader lambertShader.mll, name the icon render_lambertShader.xpm.

Put the icons in one of the directories specified in your XBMLANGPATH. You can extend this path by modifying maya.env. Type "getenv XBMLANGPATH" in the Maya script editor to see the current setting for the path.

Special shading nodes

Maya also contains special shading nodes—surfaceShader, volumeShader and displacementShader. These nodes do nothing except provide appropriately named attributes. You can connect anything you like to these nodes, which then presents that connection to rendering as the appropriate attribute name. Special mechanisms internal to Maya ensure that these special nodes do not impose any execution time overhead, and so can be used with impunity.

For descriptions of shading nodes, see “Shader source code examples” in Chapter , “Example Plug-ins”.

SuperSampling within shading nodes

As explained previously, you can request specific rendering information regarding the current sample position through pre-defined attributes provided during the rendering process. (See Appendix C: Rendering attributes for a complete list of rendering specific attributes and their corresponding names.) However, it is sometimes desirable to describe a hypothetical position and force a shading network evaluation to sample this hypothetical position. Some applications of this technique are bump mapping, and filtering (antialiasing).

A shading node can mark attributes as "render sources" through the API call MFnAttribute::setRenderSource. If the shadingNode then sets the values of the one of these attributes, subsequent calls to request data from the datablock will force the shading network to reevaluate.

The devkit contains an example plug-in, shiftNode.cpp, that demonstrates modifying uvCoord and refPointCamera from within a plug-in texture. The uvCoord and refPointCamera are marked as "renderSource" attributes. The uvCoord and refPointCamera for the current sample position are requested and then subsequently shifted four times. Each time these attributes are modified, the inColor attribute is requested, and because the attributes are render sources, the request for inColor forces a shading evaluation. Thus the 2D or 3D texture connected to inColor will be evaluated four additional times for every point shaded. The inColor values are averaged which produces a blurred result.

Evaluating shading nodes outside of the rendering context

Shading nodes can request rendering information regarding the current sample position through pre-defined attributes (See Appendix C: Rendering attributes for a complete list of rendering specific attributes and their corresponding names.). However, these pre-defined attributes are not supplied in a non-rendering context. Evaluation of shading nodes outside of the rendering context is supported using the call MRenderUtil::sampleShadingNetwork.

```
MStatus MRenderUtil::sampleShadingNetwork(

    MString             shadingNodeName,
    long                numSamples,
    bool                useShadowMaps,
    bool                reuseMaps,

    MFloatPointArray    *points,  // sample points in world
    MFloatArray         *uCoords,
    MFloatArray         *vCoords,
    MFloatVectorArray   *normals, // normals in world
    MFloatPointArray    *refPoints, // refPoints in world
    MFloatVectorArray   *tangentUs,
    MFloatVectorArray   *tangentVs,
    MFloatArray         *filterSizes,

    MFloatVectorArray   &resultColors,
    MFloatVectorArray   &resultTransparencies
);
```

You provide lists of sample points, normals, UVs, etc., and the function will return lists of colors and transparencies calculated based on the supplied sample data.

Shadow calculation can also be done by forcing a test rendering to generate shadow maps before samples are taken.

An example plug-in, sampleCmd.cpp, has been provided. This command takes a particle object and a shading node/shading engine name as input, and creates particles which have the color assigned based on the sampling result.

7 Manipulators

Developer

Plug-in API

Write a manipulator

Overview of creating manipulators

This chapter describes how to create manipulators in Maya.

- “What is a manipulator?”
- “Base manipulators”
- “Writing a manipulator”
- “Manipulator containers”
- “Communication between manipulators and nodes”
- “Connect manipulators to the Show Manipulator Tool”

What is a manipulator?

A manipulator is a node that draws itself using 3D graphical elements that respond to user events. Manipulators translate the events into values which are used to modify attribute values of other nodes in a scene. The attribute values are modified directly by the manipulator and not through the standard plug and connection mechanism used by other dependency graph nodes.

You invoke manipulators through the Show Manipulator Tool or a user-defined context. Manipulators exist in the DAG as subclasses of transform nodes, but they only exist while the Show Manipulator Tool or context is active, and the object that they correspond to is selected. Unlike transforms, they are not visible in the hypergraph or outliner, and they are not added to the Maya selection list. Additionally, their attributes are not accessible from MEL or the attribute editor and they are not written to file.

Manipulators are designed to operate on data types, ranging from integer and floating point values to matrix data and can operate on one or more attribute values at the same time. Maya defines a set of simple manipulators, called *base manipulators*. These operate on a range of data types from a single boolean, integer, or floating point value, to vectors of floating point values of different lengths.

More complex manipulators, called *container manipulators*, can be designed by combining one or more base manipulators. In the API, you construct a manipulator by creating a container manipulator and adding one or more base manipulators to it. The new manipulator can be invokes either through the Show Manipulator Tool or through a user-defined context.

Base manipulators

The OpenMaya API supports 10 base manipulator classes that can be combined to form a composite manipulator.

The following lists these base manipulators and the C++ function sets that correspond to them:

- "FreePointTriadManip": MFnFreePointTriadManip
- "DirectionManip": MFnDirectionManip
- "DistanceManip": MFnDistanceManip
- "PointOnCurveManip": MFnPointOnCurveManip
- "PointOnSurfaceManip": MFnPointOnSurfaceManip
- "DiscManip": MFnDiscManip
- "CircleSweepManip": MFnCircleSweepManip
- "ToggleManip": MFnToggleManip
- "StateManip": MFnStateManip
- "CurveSegmentManip": MFnCurveSegmentManip
- "RotateManip": MFnRotateManip
- "ScaleManip": MFnScaleManip

FreePointTriadManip

The FreePointTriadManip provides a moveable point which can be moved anywhere. It has axes for constrained x, y, and z movement and obeys grid snapping, point snapping, and curve snapping. The FreePointTriadManip generates the 3D position of the point. It is useful for specifying the position of an object in space.

Note The FreePointTriadManip is a subset of the moveManip in Maya.

DirectionManip

The DirectionManip lets you specify a direction as defined by the vector from the start point to the manipulator position. It uses a FreePointTriadManip to specify the end point of a vector relative to a given start point. This manipulator generates a vector from the start point to the end point.

DistanceManip

The DistanceManip lets you manipulate a point that is constrained to move along a line. The distance value is calculated from the start point of the line to the manipulated point. This manipulator generates a single floating point value. Scaling factors can be used to determine how long the manipulator appears when it is drawn.

PointOnCurveManip

The PointOnCurveManip lets you manipulate a point constrained to move along a curve, in order to specify the "u" curve parameter value. This manipulator generates a single floating point value corresponding to the curve parameter.

PointOnSurfaceManip

The PointOnSurfaceManip lets you manipulate a point constrained to move along a surface, in order to specify the (u, v) surface parameter values. This manipulator generates two floating point values corresponding to the surface (u, v) parameters.

DiscManip

The DiscManip lets you rotate a disc in order to specify a rotation about an axis. This manipulator generates a single floating point value corresponding to the rotation.

CircleSweepManip

The CircleSweepManip lets you manipulate a point constrained to move around a circle, in order to specify a sweep angle. This manipulator generates a single floating point value corresponding to the sweep angle.

ToggleManip

The ToggleManip lets you switch between two modes or some on/off state. It is drawn as a circle with or without a dot. When the mode is on, the dot is drawn in the circle; when the mode is off, the circle is drawn without the dot. This manipulator generates a boolean value corresponding to whether or not the mode is on or off.

StateManip

The StateManip lets you switch between multiple states. It is drawn as a circle with a notch. Each click on the circle increments the value of the state (modulo the maximum number of states). This manipulator generates an integer value corresponding to the state of the manip.

CurveSegmentManip

The CurveSegmentManip lets you manipulate two points on a curve to specify a curve segment. This manipulator generates two floating point values, which correspond to the parameters of the start and end of the curve segment.

RotateManip

The RotateManip allows you to manipulate a 3d rotation vector. It supports the 3 rotation modes of the built-in rotate manipulator (object space, global, gimbal) and allows constrained rotation on the x, y, z and viewing axes. The vector generated by the manipulator is an Euler rotation that is suitable for input to a rotation plug.

ScaleManip

The ScaleManip lets you manipulate relative x, y, and z scaling values. The scale manipulator provides a central handle for proportional scaling, as well as x, y, and z axis handles for non-proportional scaling on each axis. The vector generated by the manipulator is a relative scaling vector that is suitable for input to a scale plug.

Writing a manipulator

Writing a manipulator involves defining a subclass of the MPxManipContainer class, adding base manipulators to the container manipulator, and defining associations between the manipulator and the attributes on the nodes they affect. Even if your manipulator consists of only one base manipulator, it is necessary to create a container manipulator and add the base manipulator to it.

Container manipulators are composed of one or more base manipulators. When base manipulators are added to a container manipulator, they are referred to as *children* of the container manipulator, and are added using the createChildren method.

The association between the manipulator and the corresponding plugs on a node must be defined. The nature of the association between the manipulator and the plugs may be simple or complex. For simple associations, there is a direct correspondence between a manipulator value and the corresponding plug. For more complex associations, conversion functions are used. These are described below.

Manipulator containers

MPxManipContainer is the parent class of user-defined container manipulators. User-defined container manipulators are comprised of one or more base manipulators. There are a number of methods on MPxManipContainer that allow you to add a variety of base manipulators to the container. There are also a number of methods you need to implement on your user-defined manipulator derived from MPxManipContainer.

The necessary methods are:

- "creator method"
- "initialize method"
- "createChildren method"
- "connectToDependNode method".

You can also override the draw method to customize the way your container manipulator is drawn.

creator method

The creator method needs to return a new instance of the manipulator, and it is registered in the initializePlugin function with a call to the MFnPlugin::registerNode method.

Note Since this method is static and registered and not derived, the name of the method does not need to be "creator" although by convention the name "creator" is typically used.

initialize method

The initialize method performs any necessary initializations for the manipulator as well as calling the method in the parent class MPxManipContainer::initialize. Like the creator method, the initialize method is static and registered rather than derived.

createChildren method

The createChildren method is where calls to add base manipulators should be called.

For example, in the moveManip::createChildren method:

```
MStatus moveManip::createChildren()
{
    ...
    fDistanceManip = addDistanceManip(manipName,
                                      distanceName);
    fFreePointManip = addFreePointTriadManip(pointManipName,
                                             pointName);
    ...
}
```

connectToDependNode method

The connectToDependNode method is where the association is made between the manipulator and the plug(s) with which it will communicate. This method requires two additional methods to be called after the calls to associate manipulators with plugs have been made. The methods are MPxManipContainer::finishAddingManips and MPxManipContainer::connectToDependNode and must be called in that order. It is important to note that MPxManipContainer::finishAddingManips must be called after all calls to connect to plugs have been made. Furthermore, finishAddingManips must be called only once.

For example, in the moveManip::connectToDependNode method:

```
MStatus moveManip::connectToDependNode(const MObject &node)
{
    ...
    distanceManipFn.connectToDistancePlug(syPlug);
    ...
    freePointTriadManipFn.connectToPointPlug(tPlug);
    ...
    finishAddingManips();
    ...
    MPxManipContainer::connectToDependNode(node);
    ...
}
```

Note Although connectToDependNode is a virtual method, you will usually be responsible for calling this method yourself when you are using the manipulator within a custom context. For more information, see "Connect manipulators to the Show Manipulator Tool" on page 131.

draw method

The draw method is an optional method which can be used to customize the drawing of a container manipulator. If you are overriding the draw method, you should first call MPxManipContainer::draw to draw all the children.

For example, the moveManip::draw method draws a label in addition to the base manipulators.

```
void moveManip::draw(M3dView &view,
                        const MDagPath &path,
                        M3dView::DisplayStyle style,
                        M3dView::DispalyStatus status)
{
    MPxManipContainer::draw(view, path, style, status);
    view.beginGL();
    MPoint textPos(0, 0, 0);
    char str[100];
    sprintf(str, "Stretch Me!");
    MString distanceText(str);
    view.drawText(distanceText, textPos, M3dView::kLeft);
    view.endGL();
}
```

Registration/Deregistration

Because manipulator containers are derived from nodes, user-defined manipulators can be registered and deregistered like any other node, with the exception that the MPxNode::Type argument in MFnPlugin::registerNode is set to MPxNode::kManipContainer instead of the default MPxNode::kDependNode.

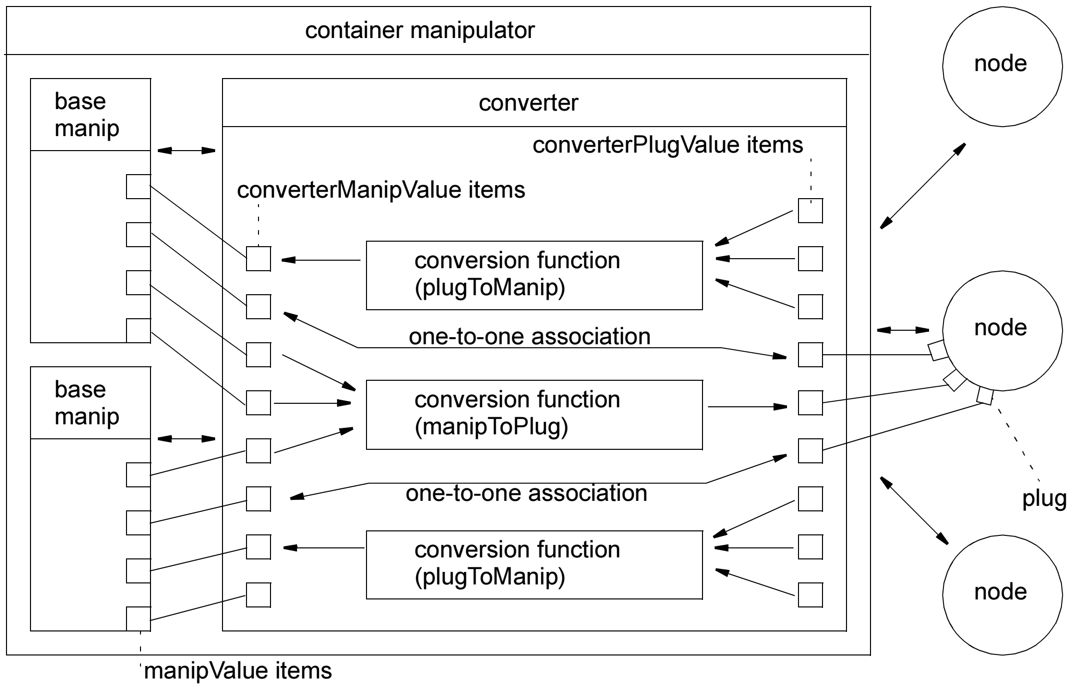
For example, in moveToolManip.cpp:

```
MStatus initializePlugin(MObject obj)
{
    ...
    plugin.registerNode("moveManip",
                        moveManip::id,
                        &moveManip::creator,
                        &moveManip::initialize,
                        MPxNode::kManipContainer);
    ...
}

MStatus uninitializePlugin(MObject obj)
{
    ...
    plugin.deregisterNode(moveManip::id);
    ...
}
```

Communication between manipulators and nodes

Manipulators communicate with plugs on nodes to set the values of those plugs and also to set manipulator values appropriately with respect to the values of the plugs.

The communication between manipulators and nodes can be done in one of two ways: simple one-to-one associations, or through the use of more complex conversion functions. The following diagram illustrates how the communication between nodes and manipulators takes place.



The converter in the diagram is the mechanism that manages the communication between the plugs on the nodes and manipulator values. The arrows indicate the direction the information flows. Each container manipulator has one converter which is the interface between the container's children manipulators and the plugs they affect.

There are a number of data items identified by square boxes on the converter and the base manipulators. The items on the converter that are related to children manipulator values are called *converterManipValue items*, and the items on the converter that are related to the node plug values are called *converterPlugValue items*.

The items on the base manipulators are called *manipValue items*. Some of these manipValue items relate directly to an affordance of the manipulator. For example, the MFnDiscManip::angleIndex relates directly to the rotation affordance of the DiscManip.

Other manipValue items do not relate to an affordance of the manipulator, but provide important information on the position or orientation of the manipulator such as MFnDiscManip::centerIndex and MFnDiscManip::axisIndex.

Each converterManipValue item and each converterPlugValue item has an integer index that uniquely identifies that item. Each manipValue item on a base manipulator also has an integer index that uniquely identifies that item.

Note There is a one-to-one correspondence between a converterManipValue item and a base manipulator’s manipValue item, and these corresponding items share the same integer index. There is also a one-to-one correspondence between a converterPlugValue item and a plug on a node affected by the manipulator.

As seen in the diagram, the one-to-one associations are directly between a converterManipValue item and a converterPlugValue item.

More complex conversions between converterManipValue items and converterPlugValue items are performed through conversion functions. These functions can use any number of converterPlugValue or converterManipValue items to calculate the value of the corresponding converterManipValue or converterPlugValue item.

One-to-one associations

One-to-one assocations between a converterManipValue item and a converterPlugValue item are established through methods on the manipulator classes derived from MFnManip3D. These methods and the data types corresponding to the plugs they connect to are:

- MFnFreePointTriadManip::connectToPointPlug (3 doubles)
- MFnDirectionManip::connectToDirectionPlug (3 doubles)
- MFnDistanceManip::connectToDistancePlug (double)
- MFnPointOnCurveManip::connectToCurvePlug (curve)
- MFnPointOnCurveManip::connectToParamPlug (double)
- MFnPointOnSurfaceManip::connectToSurfacePlug (surface)
- MFnPointOnSurfaceManip::connectToParamPlug (2 doubles)
- MFnDiscManip::connectToAnglePlug (double)
- MFnCircleSweepManip::connectToAnglePlug (double)
- MFnToggleManip::connectToTogglePlug (boolean)

- MFnStateManip::connectToStatePlug (long)
- MFnCurveSegmentManip::connectToCurvePlug (curve)
- MFnCurveSegmentManip::connectToStartParamPlug (double)
- MFnCurveSegmentManip::connectToEndParamPlug (double)
- MFnRotateManip::connectToRotatePlug (3 doubles)
- MFnRotateManip::connectToRotateCenterPlug (3 doubles)
- MFnScaleManip::connectToScalePlug (3 doubles)
- MFnScaleManip::connectToScaleCenterPlug (3 doubles)

These methods should be called from the connectToDependNode method described above. For example, in the footPrintManip:

```
MStatus footPrintLocatorManip::connectToDependNode
                                (const MObject &node)
{
    ...
    MFnDistanceManip distanceManipFn(fDistanceManip);
    MFnDependencyNode nodeFn(node);
    MPlug sizePlug = nodeFn.findPlug("size", &stat);
    if (MStatus::kFailure != stat) {
        distanceManipFn.connectToDistancePlug(sizePlug);
        ...
        finishAddingManips();
        MPxManipContainer::connectToDependNode(node);
    }
    return stat;
}
```

Conversion functions

Conversion functions are used to convert between manipulator values and plug values. They are implemented as callback methods. A simple example of a manipulator that uses conversion functions is a container manipulator with a DiscManip connected to a plug (that is associated with an attribute of type MFnUnitAttribute::kAngle) that takes the rotation of the disc manip and multiplies that rotation by 10. The conversion function uses MPxManipContainer:: getConverterManipValue on the MFnDiscManip::angleIndex and then multiplies that angle by 10.

Conversion functions are very useful when the position of a manipulator has to be affected by the position of an object, or to move a group of manipulators together in a specific way. Without conversion functions, manipulators would not be able to move together as a unit, and certain components of the manipulator would remain either at the origin or a fixed position in space.

Note Conversion functions are not required in situations where a FreePointTriadManip specifies a position or a PointOnCurveManip specifies a parameter along a curve. However, if you have a DiscManip that you want to move along with the PointOnCurveManip, you need a conversion function to give the DiscManip information about its position and normal.

There are two kinds of conversion callback methods: manipToPlug and plugToManip.

plugToManip

A plugToManip conversion callback is used to get the value of a converterManipValue item from various converterPlugValue items. This callback has access to all the converterPlugValue items and returns the value of a converterManipValue item.

manipToPlug

A manipToPlug conversion callback is used to get the value of a converterPlugValue item from various converterManipValue items. This callback has access to all the converterManipValue items and returns the value of a converterPlugValue item.

In general, manipToPlug conversions are less commonly used. In addition to using converterPlugValues and converterManipValues, it is sometimes useful to use class data, such as a DAG path. (See the footPrintManip for an example of how fNodePath is used to calculate the node translation.) For manipulators that operate on components, it may also be useful to store initial component positions (see the componentScaleManip for an example of how this is done).

MManipData

The conversion callback methods return a data type called MManipData. MManipData encapsulates manipulator data which is returned from the manipulator conversion functions. It represents data that is either simple or complex. The simple data methods on MManipData are used to represent bool, short, long, unsigned, float, and double types.

Note Sometimes attributes associated with the simple data types have higher level meanings such as distance, angle, and time (for example, MFnUnitAttribute::kAngle, MFnUnitAttribute::kDistance, and MFnUnitAttribute::kTime).

MManipData is also used to represent complex data types created by MFnData or classes derived from MFnData, such as matrices, curves, and arrays of data.

The footPrintManip example plug-in has an example of a plugToManip conversion callback called startPointCallback. The startPointCallback returns an MManipData which is set to be an MObject created by MFnNumericData.

```
class footPrintLocatorManip : public MPxManipContainer
{
public:
    ...
    MManipData startPointCallback(unsigned index) const;
    MVector nodeTranslation() const;
    MDagPath fDistanceManip;
    ...
};

MManipData footPrintLocatorManip::startPointCallback
    (unsigned index) const
{
    // The index is the startPointIndex that is
    // specified in addPlugToManipConversionCallback,
    // but it is not necessary to use this in the callback.
    MFnNumericData numData;
    MObject numDataObj =
        numData.create(MFnNumericData::k3Double);
    MVector vec = nodeTranslation();
    numData.setData(vec.x, vec.y, vec.z);
    return MManipData(numDataObj);
}

MStatus footPrintLocatorManip::connectToDependNode
                                (const MObject &node)
{
    ...
    unsigned startPointIndex =
        distanceManipFn.startPointIndex();
    addPlugToManipConversionCallback(
        startPointIndex,
        (plugToManipConversionCallback) startPointCallback);
    ...
}
```

Connect manipulators to the Show Manipulator Tool

The Show Manipulator Tool uses Maya manipulators to modify plug values or provide access to the construction history of a node. If you select several objects while in the Show Manipulator Tool, the corresponding manipulator displays for each of the objects selected.

The object that the manipulator is attached to does not have to be a DAG node. Some manipulators, such as the Revolve manipulator, are attached to a dependency node upstream of the final shape. For example, for a sphere called nurbsSphereShape1, the revolve manipulator must be attached to the makeNurbsSphere1 node which is upstream relative to nurbsSphereShape1.

Note makeNurbsSphere1 is not a DAG node, and can be selected either in the Hypergraph or the Channel Box.

Other manipulators, such as the camera manipulator and the light manipulator, can be attached to either the transform or the shape of the light or camera.

The Show Manipulator Tool is explained in detail in the *Basics* guide.

Writing a manipulator to work with the Show Manipulator Tool

The first step when creating a manipulator that works with the Show Manipulator Tool for a given node is to pick a name for the manipulator that corresponds to the node type name. To determine the name of the manipulator, take the name of the node and append “Manip” to the end of the node name. For example, the “Show Manip” for footPrintLocator is footPrintLocatorManip.

```
plugin.registerNode("footPrintLocator", // name of node
                    footPrintLocator::id,
                    &footPrintLocator::creator,
                    &footPrintLocator::initialize,
                    MPxNode::kLocatorNode);

plugin.registerNode("footPrintLocatorManip", // name of manip
                    footPrintLocatorManip::id,
                    &footPrintLocatorManip::creator,
                    &footPrintLocatorManip::initialize,
                    MPxNode::kManipContainer);
```

The second step is to have the initialize method of the node call MPxManipContainer::addToManipConnectTable. For example, in the footPrintLocator::initialize method:

```
MStatus footPrintLocator::initialize()
{
    ...
    MPxManipContainer::addToManipConnectTable(id);
    ...
}
```

Where id is defined and declared as:

```
class footPrintLocator : public MPxLocatorNode
{
    ...
public:
    static MTypeId id;
};
MTypeId footPrintLocator::id(0x81101);
```

Adding the manipulator to a Context

An alternative to invoking a manipulator by the Show Manipulator Tool is to invoke the manipulator from a user-defined context.

The class MPxSelectionContext has two methods to support the management of manipulators: addManipulator and deleteManipulator.

In addition to using these two methods, MPxSelectionContext::toolOnSetup and MPxContext::toolOffCleanup should be overridden so that toolOnSetup adds a callback for manipulators, and toolOffCleanup removes the callback when the active list is modified.

The callback would be an *updateManipulators* function which actually adds and deletes the manipulators.

For example:

```
MCallbackId id1;
void moveContext::toolOnSetup(MEvent &)
{
    ...
    id1 = ModelMessage::addCallback(
          MModelMessage::kActiveListModified,
          updateManipulators, this, &status);
    ....
}

void moveContext::toolOffCleanup()
{
    ...
    status = MModelMessage::removeCallback(id1);
    ...
}

void updateManipulators(void * data)
{
    ...
    moveContext * ctxPtr = (moveContext *) data;
    ctxPtr->deleteManipulators();
    ...
    // for each object selected
    MString manipName("moveManip");
    MObject manipObject;
    ctxPtr->moveM = (moveManip *)
                    moveManip::newManipulator(manipName,
                                              manipObject);
    ...
    ctxPtr->addManipulator(manipObject);
    ...
    ctxPtr->moveM->connectToDependNode(dependNode);
    ...
}
```

Example Manipulators

moveManip

The moveManip.cpp plug-in shows how to create a manipulator from a context. The user-defined manipulator in moveToolManip.cpp is called moveManip and consists of two base manipulators: a FreePointTriadManip and a DistanceManip.

footPrintManip

This plug-in example demonstrates how to use the Show Manipulator Tool with a user-defined node and a user-defined manipulator. The user-defined manipulator consists of a DistanceManip.

Note This manipulator uses a conversion function to place the DistanceManip at the location of the foot. Otherwise, the DistanceManip would appear at the origin.

rotateManip

This plug-in demonstrates the different modes of the rotate base manipulator. The user-defined manipulator in rotateManip.cpp consists of a rotate base manipulator and a state base manipulator. The state manipulator is used to control the mode of the rotate manipulator: object mode, world space mode, gimbal mode, and object mode with snapping.

componentScaleManip

This plug-in demonstrates how to use the scale base manipulator and also demonstrates a method for manipulating components. The plug-in componentScaleManip.cpp consists of a scale base manipulator. The manipulator works by attaching manipToPlug conversion callbacks for every selected vertex. The conversion function computes the new vertex positions using stored initial vertex positions and the scale manipValue.

surfaceBumpManip

The surfaceBumpManip plug-in example demonstrates how the pointOnSurface base manipulator can be used to modify vertices close to the param manipValue. The plug-in uses a manipToPlug conversion function as a callback to update vertex positions on the NURBS surface.

Note Since this plug-in uses the manipToPlug conversion function as a callback that computes a dummy plug, with the vertex positions updated independently, this plug-in will not support undo.

8 Shapes

Developer Plug-in API

Define a shape

Shapes in Maya

Shapes in Maya are selectable DAG objects that display in 3D views. Meshes, NURBS surfaces and curves, and locators are just some examples of shapes. Shapes are also dependency graph (DG) nodes that have attributes which can be connected to other nodes.

Shapes can be thought of as a container for geometry. The role of this container is to present the geometry for interactive display and manipulation.

Shapes which are formed and manipulated by the position of control vertices (or CVs) are called control point shapes. The control points are attributes on the shape which are usually arrays of points. Control points become interactive, i.e. they can be selected and manipulated, by associating the control point attributes with a component.

Components are objects (MObject's) which contain two pieces of information, the type of component and the index values or range. For example, a mesh vertex component. This component represents the vertex (or "pnts") attribute of a mesh shape where "vtx[0]" represents vertex 0 of the mesh and "vtx[0:7]" represents the first eight vertices of the mesh.

Shapes which define a surface such as NURBS surfaces and meshes support hardware display of materials and hardware lighting in interactive views as well as the hardware render buffer.

User-defined shapes

The purpose of a user-defined shape and all the classes associated with shapes is to "wrap" an arbitrary geometry type that you defined as a DAG node (and a DG node), or as data passed through the DG.

The class MPxSurfaceShape lets you write shapes using the Maya API. Writing a shape node is similar to writing a dependency node, in fact, the base class for user defined shapes (MPxSurfaceShape) derives from MPxNode.

The major difference between MPxSurfaceShape and MPxNode is the addition of component handling, drawing, and selection functions. The overridable functions have been split into two classes, the DG node functionality is in MPxSurfaceShape and the drawing and select functions are in MPxSurfaceShapeUI.

User-defined shapes are best suited for control point based shapes such as polygonal meshes or spline-based objects. This does not mean that you cannot write other kinds of shapes, it just means that MPxSurfaceShape is already set up to handle control point based shapes.

Shapes are registered using the MFnPlugin::registersShape function. This function is similar to registerNode but contains an additional argument specifying the creator function for the MPxSurfaceShapeUI class. Unregistering shapes is done using MFnPlugin::deregisterNode in the same way that MPxNode classes are unregistered.

Shape classes

The following lists the proxy classes you will be deriving from:

- “MPxSurfaceShape (required)”: the DG node class
- “MPxSurfaceShapeUI (required)”: drawing and selection
- “MPxGeometryIterator (optional)”: iterator for components (optional)
- “MPxGeometryData (optional)”: used to pass geometry through the DG (optional)

MPxSurfaceShape and MPxSurfaceShapeUI are mandatory. The main functionality for shapes is split between these two classes to separate the drawing and selection code from the evaluation code.

MPxSurfaceShape (required)

This class defines the non UI part of a shape. The goal of this class is to represent some user-defined geometry in Maya’s DAG.

MPxSurfaceShapeUI (required)

This class defines the drawing and selection part of a shape.

MPxGeometryIterator (optional)

This class provides Maya with a control point iterator for your geometry. The iterator queries and sets points in your geometry generically. Iterators are used when your shape has components associated with its attributes.

MPxGeometryData (optional)

This class is used to provide Maya with a container for your geometry that can be passed through DG connections. It is often more efficient to pass all of the information about your geometry as data then as separate attributes. If you want your shape to work with Maya's deformers then you must provide a data class as input and output attributes to your shape.

Writing a shape

The goal of writing a shape is to take a geometry class you defined and integrate that geometry into Maya in the form of a DAG object.

It is a good idea to keep geometry-specific information in your geometry class. This way it will be easier to represent the geometry as dependency graph data. Keeping geometry specific information in your geometry class will also make it easier to handle drawing as you will have to pass geometry-specific drawing information onto Maya's draw request queue as draw data.

Where to start

User-defined shapes are more complex than DG nodes because of the additional drawing, selection, and component functionality. It is a good idea to start by designing the shape node before writing any code. This involves defining your input and output attributes and determining the relationships (or affects) between those attributes.

The simplest shape is one without components or geometry data—for example, a sphere shape that takes an input radius and perhaps x and y divisions and from this computes the geometry for a sphere to draw in openGL.

It is also a good idea to write your shape in stages. The first step is to derive from MPxSurfaceShape and MPxSurfaceShapeUI. Writing the MPxSurfaceShape class is the same as writing any MPxNode class. The additional virtual functions can be overridden as the functionality is needed.

Registering and deregistering shapes

Registering shapes with Maya is similar to registering DG nodes. The only difference is that shapes have a separate UI class that must be registered with the shape node. The MFnPlugin::resgisterShape function is used for shape registration. Deregistering a shape is exactly the same as deregistering a node. Here is an example of shape registration:

```
MStatus initializePlugin( MObject obj )
{
    MFnPlugin plugin( obj,"Alias","2.0", "Any");
    return plugin.registerShape( "yourShape", yourShape::id,
                                 &yourShape::creator,
                                 &yourShape::initialize,
                                 &yourShapeUI::creator );
}

MStatus uninitializePlugin( MObject obj )
{
    return plugin.deregisterNode( yourShape::id );
}
```

Drawing and refresh

Drawing is a two step process—first the geometry and materials are evaluated and all of the information necessary for drawing is placed onto a queue, and secondly, the actual OpenGL drawing.This two step process allows your shapes to take advantage of the multi-threaded drawing capabilities of future versions of Maya.

Drawing occurs whenever the camera changes or the view has to be refreshed. When this happens, the MPxSurfaceShapeUI::getDrawRequest function is called. This is Maya's way of asking the shape what it needs to draw. Inside this function you should query the drawing state, using MDrawInfo, and then construct a draw request (MDrawRequest) to place on the queue. You will often want to place multiple requests on the queue (MDrawRequestQueue) in this function. For instance, in shaded mode if your shape is selected you may want to add a request to draw the shaded object and another request to draw wireframe on top of the shaded object.

The draw data (MDrawData) holds information specific to your shape which Maya does not intrinsically know about. The draw data acts as a container to pass your geometry through the draw request queue. For each draw request you must create and add a draw data object which contains geometry-specific information that you will need in the subsequent call to MPxSurfaceShapeUI::draw.

To create draw data, use the function MPxSurfaceShapeUI::getDrawData and to add the data to a request use MDrawRequest::setDrawData. The following example shows how to create a draw request and draw data for your geometry.

```
void yourShapeUI::getDrawRequests( const MDrawInfo & info,
                     bool objectAndActiveOnly,
                     MDrawRequestQueue & queue )
{
    MDrawData data;
    MDrawRequest request = info.getPrototype( *this );
    yourShape* shapeNode = (yourShape*)surfaceShape();
    yourGeom* geom = shapeNode->geometry();
    getDrawData( geom, data );
    request.setDrawData( data );
...
}
```

The draw request (MDrawRequest) should be created in the overridden MPxSurfaceShapeUI::getDrawRequests method. Once the request is created the appropriate “set” methods of this class should be used to define what is being requested. Then the request should be placed on the draw request queue using MDrawRequestQueue::add.

The draw token is an integer value which you can use to specify what is to be drawn. This is object specific and so you should define an enum with the information you require to decide what is being drawn in your MPxSurfaceShapeUI::draw method. Here is an example of a draw token for a mesh object:

```
    enum {
        kDrawVertices, // component token
        kDrawWireframe,
        kDrawWireframeOnShaded,
        kDrawSmoothShaded,
        kDrawFlatShaded,
        kLastToken
    };
```

Maya processes the draw request queue and for each draw request, calls the corresponding objects draw function. In the case of user defined shapes, your MPxSurfaceShapeUI::draw method is called.

Drawing in shaded mode

Supporting shaded mode display is a two step process. The first step occurs in your getDrawRequests function. Here you must “evaluate” or setup the material so that it can be displayed when your object is drawn. The second step occurs in your draw function where you must setup the view to display the material.

The following code demonstrates how to setup the material in your getDrawRequests function if the request is for shaded mode display.

```
    MDagPath path = info.multiPath();
    M3dView view = info.view();
    MMaterial material = MPxSurfaceShapeUI::material( path
);
    material.evaluateMaterial( view, path );
    if ( material.materialIsTextured() ) {
        material.evaluateTexture( data );
    }
    request.setMaterial( material );
```

In your draw function you will need to setup the viewport so that it can display the material. This is done using MMaterial::setMaterial. To set up the viewport to display textures, use MMaterial::applyTexture. The following code demonstrates this.

```
void yourShapeUI::draw( const MDrawRequest& request,
                        M3dView & view ) const
{
    ...
    MMaterial material = request.material();
    material.setMaterial( request.isTransparent() );
    drawTexture = material.materialIsTextured();
    if ( drawTexture ) glEnable(GL_TEXTURE_2D);
    if ( drawTexture ) {
        material.applyTexture( view, data );
    }
    ...
}
```

Note All OpenGL calls should be enclosed by calls to M3dView::beginGL() and M3dView::endGL().

Selection

When an object is selected in Maya, a selection ray is generated based on the orientation of the camera and the mouse position. Selection happens in two steps—first, all of the object bounding boxes are tested for ray intersection, then the select functions are called for all objects whose bounding boxes were hit.

The shape selection function is defined by overriding the select function of MPxSurfaceShapeUI. You must override this function to add items to the given selection list based on the selection state information provided by MSelectInfo.

See the method apiMeshShapeUI::select and apiMeshShapeUI::selectVertices in the apiMeshShape example plug-in for an example of object and component selection.

Components

Shapes which require visual feedback and manipulation of attributes, such as control point based shapes, must have components associated with those attributes. Shapes such as polygonal meshes or spline surfaces have control vertices which can be selected and manipulated. These control vertices exist as attributes on the shape which are exposed interactively in Maya by representing them as components.

Components are objects (MObject's) which contain two pieces of information, the type of component and the index values or range. An example is a vertex component for a mesh shape where "vtx[0]" represents vertex 0 of the mesh and "vtx[0:7]" represents the first 8 vertices of the mesh.

The classes used for creating, editing, and querying components are:

- MFnComponent
- MFnDoubleIndexedComponent
- MFnSingleIndexedComponent
- MFnTripleIndexedComponent

Components fall into three categories based upon the dimensions of the index. The types are single, double, and triple indexed. Examples of these types are mesh vertices (single indexed), NURBS surface CVs (double indexed), and lattice points (triple indexed).

Components can be marked as complete, meaning the component represents a complete set of indices from 0 to numElements-1.

Mapping attributes to components

In Maya, components are specified as strings. Each type of component has a different string name. In the API components are MObjects distinguished by their API type (see MFn.h). For example, a mesh vertex component can be specified in Maya as "vtx[0]" and in a plug-in it represented as an MObject with apiType "MFn::kMeshVertComponent". The index information can be extracted using an MFnComponent derived class.

To associate (or map) a component with one of your shapes attributes you must choose one of Maya's existing component types and override MPxSurfaceShape::componentToPlugs to convert component types to plugs.

The following is an example of associating a mesh vertex component with the "mControlPoints" attribute of a shape:

```
void yourShape::componentToPlugs( MObject& component,
                    MSelectionList& list )const
{
    if ( component.hasFn(MFn::kMeshVertComponent) ) {
        MFnSingleIndexedComponent fnVtxComp( component );
        MObject thisNode = thisMObject();
        MPlug plug( thisNode, mControlPoints );
        int len = fnVtxComp.elementCount();
        for ( int i = 0; i < len; i++ ) {
            MPlug vtxPlug = plug.elementByLogicalIndex(
                                 fnVtxComp.element(i) );
            list.add( vtxPlug );
        }
    }
}
```

Component matching

Attributes can be specified as strings in MEL. Your shape must be able to validate these strings to ensure that proper names, indices etc. have been given. The method MPxSurfaceShape::matchComponent is used for this purpose.

```
virtual MatchResult matchComponent( const MSelectionList&
item,
                       const MAttributeSpecArray& spec,
                       MSelectionList& list );
```

This method validates component names and indices which are specified as a string and adds the corresponding component to the passed in selection list. Select commands such as “select shape1.vtx[0:7]” are validated with this method and the corresponding component is added to the selection list.

The attribute specification (MAttributeSpec) is a class that provides convenient access to all of the information about how attributes are specified. This includes attribute names, indices, and ranges.

Component iteration

For Maya to get and set the position of components you must define an iterator for your geometry by deriving from the class MPxGeometryIterator.

A geometry iterator is used by the translate/rotate/scale manipulators to determine where to place the manipulator when components are selected.

Deformers also require a geometry iterator with overridden setPoint and point methods in order to deform the points your shape.

In general, you will want to override the following methods from MPxGeometryIterator:

```
    MPxGeometryIterator( void * userGeometry,
                         MObjectArray & components );
    MPxGeometryIterator( void * userGeometry,
                         MObject & components );
    virtual void   reset();
    virtual MPoint point() const;
    virtual void   setPoint( const MPoint & ) const;
```

You must override the following functions of MPxSurfaceShape to associated an iterator with your shape:

```
virtual MPxGeometryIterator*
 geometryIteratorSetup( MObjectArray&, MObject&, bool );
virtual bool acceptsGeometryIterator( bool writeable );
virtual bool acceptsGeometryIterator( MObject&,
                                      bool, bool );
```

Translate, scale, and rotate tools for components

To support the translate, rotate and scale tools, you must override the method MPxSurfaceShape::transformUsing. The function takes a matrix and array of components as arguments. The matrix specifies the transformation that is being applied and the components specify the attribute indices that are being transformed.

For shapes with large numbers of control vertices, it can be prohibitively slow relying on Maya's compute mechanism for setting attribute values. The method MPxNode::forceCache can be used in these situations to gain direct access to a node's datablock and to get/set attribute values directly without going through compute. Special care must be taken to ensure that all attributes that depend on the ones you are changing also get updated. For instance, if the vertices of a mesh are changed, then the normals should also be updated as well as the bounding box. The method, vertexOffsetDirection, must be overridden if the Move tool is to work in move normal mode.

Tweaks and internal attributes

Shapes that have input history, for instance, another node feeding in input geometry, need some way of storing any offsets (or tweaks) applied to vertex positions. Each time the input geometry changes, the shape has to recompute. By storing any tweaks in a separate attribute, the tweaks can be added to the input vertex positions forming the output geometry.

If there is no input history, you do not have to store the tweaks in a separate attribute. Instead, vertex movements can be applied directly to the output surface.

Marking attributes as internal, using MFnAttribute::setInternal, allows you to override the behavior of setAttr and getAttr so you can deal with tweaks in a different manner depending on whether there is input history.

Note Using internal attributes is entirely up to you. This is a design issue and using internal attributes is not necessarily the only way to handle tweaks.

Input history implies that some other node is supplying your shape with input data. Creator nodes are dependency nodes which are responsible for creating specific types of shape data. Typically, a shape will have one or more creator nodes. For instance, a polygonal shape may have creator nodes for generating sphere data and cube data. See apiMeshCreator in the apiMeshShape sample plug-in for an example of a creator node.

Geometry data

To support sets and deformations, user-defined shapes must be able to pass their own data type through attribute connections. To define some data for your shape, you must derive a new class from MPxGeometryData. This class is similar to MPxData but includes methods to support sets (or groups) and component iteration.

The purpose of MPxGeometryData is to provide a wrapper or container for your geometry so that it can be passed through DG connections just like any other Maya data.

If your geometry has an iterator associated with it, then you can associate this iterator with your data to support Maya's deformers.

The following methods must be overridden to accomplish this:

```
    virtual MPxGeometryIterator* iterator( MObjectArray&
                                           MObject&, bool
);
    virtual MPxGeometryIterator* iterator( MObjectArray&,
                                           MObject&,
                                           bool, bool );
```

File IO

Shapes which do not define their own data type do not need to add any additional code to support file IO. By default, attributes on your node are saved if they are marked storable (which is default) and if there is no input connection to that attribute. If you want to selectively decide which attributes get written and which are not then override the “shouldSave” method of MPxNode.

If you define a new type of data then you must override the writeASCII and writeBinary methods of MPxData to support file IO.

For shapes which support tweaking of attribute values (like applying offsets to input history), you may need to do some extra work to specify what should be saved.

Deformers

Deformers in Maya operate on control point based shapes with components defined for the control points attribute.

To support Maya's deformations, you must provide the following:

- An "MPxGeometryData" derived class encapsulating your geometry
- A "localShapeInAttr" function
- A "localShapeOutAttr" function
- A "worldShapeOutAttr"function
- An "MPxGeometryIterator" for your geometry
- A "match" function
- A "createFullVertexGroup" function
- A "geometryData" function

MPxGeometryData

You must provide a geometry data class that encapsulates your geometry in order to for deformers to work.

localShapeInAttr

This function must be overridden to return the attribute corresponding to the input history for your shape. This attribute must be the same type as your geometry data.

localShapeOutAttr

This function must be overridden to return the attribute representing the output geometry for your shape. This attribute must be the same type as your geometry data.

worldShapeOutAttr

This function must be overridden to return the output array attribute representing instances of the output geometry for your shape. This attribute must be the same type as your geometry data. Each element in the array will represent a particular instance of your shape.

MPxGeometryIterator

Points are manipulated by deformers through an iterator which you define and implement.

match

This function must be overridden to check for matches between a selection type / component list, and the type of this shape / or it's components. This is used by sets and deformers to make sure that the selected components fall into the "vertex only" category.

createFullVertexGroup

This method is used by Maya when it needs to create a component containing every vertex (or control point) in the shape. This will get called if you apply some deformer to the whole shape, i.e. select the shape in object mode and add a deformer to it.

geometryData

This function should return the input data object for the surface. This gets called internally by Maya to get at the shapes grouping (set) information.

Example Shapes

quadricShape

The quadricShape plug-in demonstrates how to create a simple shape based around the OpenGL gluQuadric functions. This plug-in registers a new type of shape with Maya called "quadricShape" which can display spheres, cylinders, and disks. This shape supports hardware display of materials including 2d textures.

apiMeshShape

The apiMeshShape plug-in demonstrates how to create a polygonal mesh shape which has selectable, movable, animatable, and deformable vertices. This shape also supports hardware display of materials. This plug-in also creates geometry data and demonstrates how to pass this data between nodes.

The files associate with this plug-in are:

- apiMeshGeom.{h,cpp}
- apiMeshShape.{h,cpp}
- apiMeshShapeUI.{h,cpp}
- apiMeshIterator.{h,cpp}
- apiMeshData.{h,cpp}
- api_macros.h

9 Polygon API

Developer

Polygon API

Overview of Polygon API

The Poly API is the subset of the Maya API that deals with handling polygonal geometry in Maya. This chapter introduces you to the following:

- "How polygons are handled internally" on page 147
- "The five basic polygonal API classes" on page 156
- "Construction History and Tweaks" on page 161
- "polyModifierCmd example" on page 164
- "splitUVCmd example" on page 186
- "Poly exporter plug-ins" on page 208

How polygons are handled internally

Basic data structures are used to contain the components that represent the polygon (faces, edges, vertices). These structures are then further encapsulated in polygonal shape nodes to provide structure within the core of the Maya architecture—the dependency graph. Each of these concepts is crucial for manipulating and interfacing with polygons in Maya.

Polygon components

Polygonal meshes are composed of three basic components:

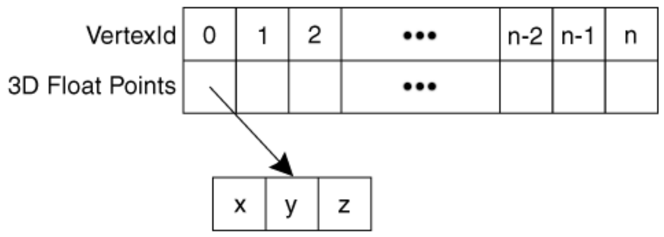
- "Vertices" on page 148
- "Edges" on page 148
- "Faces" on page 148

Beyond the three basic polygon components there are two additional components, which are just as important in understanding how to work with polygons in Maya:

- "Face-Vertices" on page 149
- "UVs" on page 150

Vertices

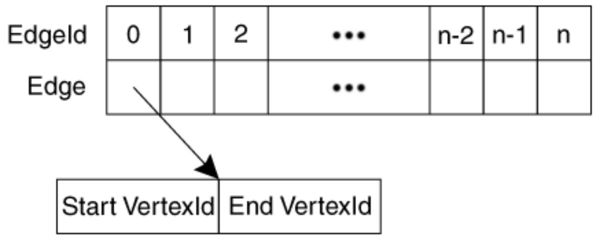
The vertices of a polygonal mesh are stored in a simple array of 3D float points, each point having a vertex id based on the given index in the array. Both edges and faces are based on this array.



Vertex array

Edges

The edges of a polygonal mesh are stored in an edge array. Each edge in the edge array consists of two integers that make up each vertex id. The first integer represents the start vertex of the edge while the second integer represents the end vertex of the edge. This provides edges with vertex composition, direction, and an edge id (represented by the indices of the edge array).

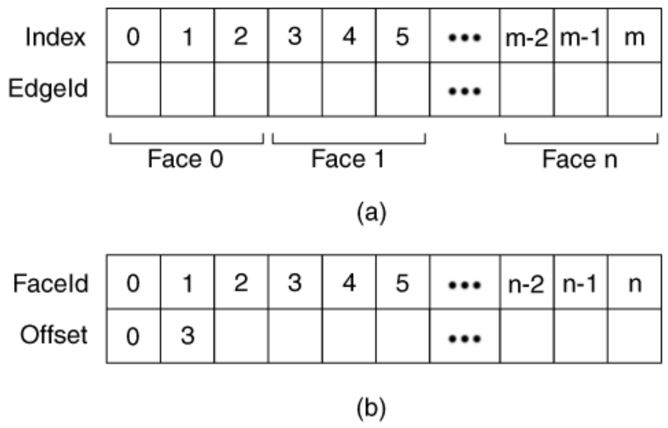


Edge array and edge structure

Faces

The faces of a polygonal mesh are stored in an integer array. Each face is described by a number of sequences of integers—each integer representing an edge id. The first sequence of edges represents the boundary of the face. Any subsequent sequences represent holes in the face. Internal flags mark the start and end of each sequence as well as the end of the face description are marked by internal flags.

A face offset or index array compliments the face array. This array holds the starting positions of each face description in the face array. Since each face can be composed of a series of a number of edges as well as multiple sequences, it can be redundant to traverse the array looking for the beginning of each face. This face index array provides a quick access to information about each face. The index of each face position is referred to as the face id of the given face. In addition to the elements marking the start of each face, an element is appended to the end of the face index array to mark the final index in the face list. This final index figure lets you quickly access the order of each face (number of edges/vertices in the face).



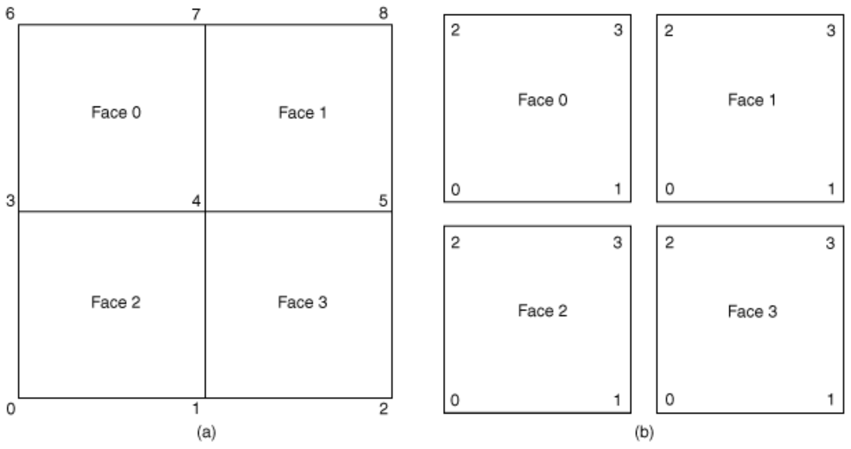
Face array and face index array structure

Face-Vertices

In cases where faces are adjacent to each other, the faces share common vertices. You often need to associate data to a specific vertex of a specific face, while distinguishing that specific vertex from any faces that share it. These are known as face-vertices.

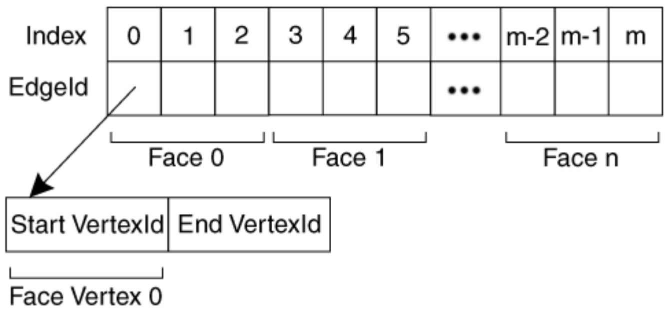
Face-vertices are conceptual components used by polygonal features such as color per vertex and UVs. Face-vertices are represented by an existing data structure—the face array and face index array. Each face vertex is associated with a given face id and vertex id. You can use the face id to find the offset into the face array and subsequently search for the given vertex id in the edge loops by using the start vertices of each edge. Notice that each vertex id in each face is a unique index for the given vertex of a face—a face-vertex index. A shared vertex id appears multiple times across the face array, appearing once in each face description that shares the vertex.

In the following illustration, (a) depicts the topology of a four face polygonal plane while (b) shows the face-vertex view of that four face polygonal plane. In the face-vertex view, each face is separated, holding its own individual vertices. Each individual vertex is labeled using a vertex index local to each face (that is, 0 to 3). Each face-vertex is associated with a UV. However this does not guarantee a unique UV per face-vertex. By default, for face-vertices that represent a shared vertex, such as vertex 4 in (a), each face-vertex is associated with the same UV, thus sharing a UV. “Splitting” a UV provides each face vertex of a shared vertex a unique UV.



Concept of face-vertices

The following figure illustrates how the view of the face array can be changed to interpret it as a representation of face-vertices.



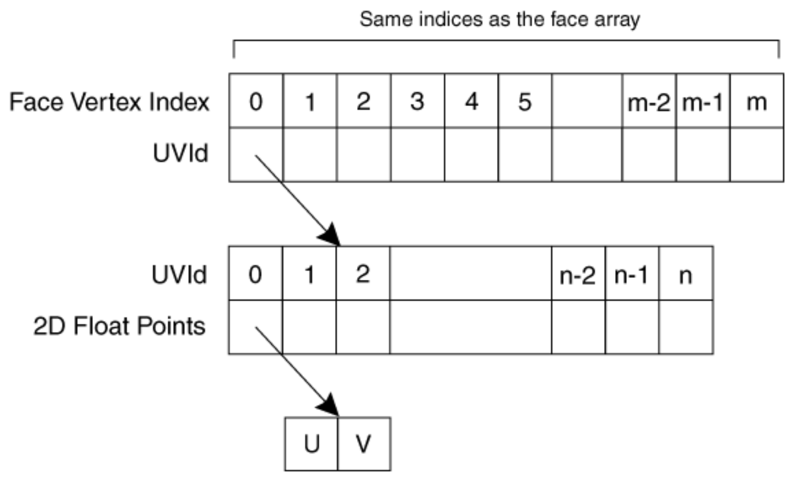
Storage and association of face-vertices

UVs

UVs rely heavily on the concept of face-vertices. UVs correspond to a 2D plane used to map a texture onto a polygonal surface. Texture mapping is done on a face- by-face basis. As a result, UVs are mapped on a face-vertex basis to allow each face its own set of map coordinates, if desired. The structure that holds UVs in Maya consists of two arrays:

- A UV index array that uses the exact same indices of the face array (visualized as a face vertex array).
- A UV array that holds a list of UV points indexed by UVIds.

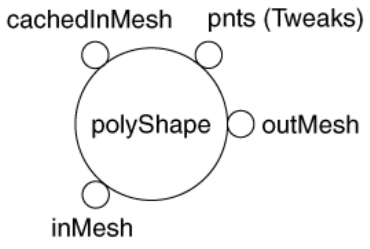
The first array associates each face vertex with a given UVId or none at all if the face that the face-vertex belongs to is not mapped. Each UVId then corresponds to an index in the UV list that holds the 2D point (U and V float values) where the UV is situated on the UV space.



UV array and UV index array

The polygonal shape node

Alone, the components are capable of representing the geometry of a mesh. However to coincide with the flexibility of the Maya architecture, these structures are integrated into the dependency graph architecture in the form of a polygonal shape node. The polygonal shape node holds four fundamental attributes: an inMesh, an outMesh, a cachedInMesh, and pnts (tweaks), as shown in the following illustration.



Polygonal shape node

The basic attributes

The four basic attributes of the polyShape node are explained below. Each of the first three attributes cache their own copy of the mesh for the polyShape node. The differences between each represent the different stage of evaluation during a DG evaluation.

inMesh

The standard input attribute of the polyShape node. This attribute accepts input mesh data from other DG nodes and forwards the data through the node to the outMesh. It stores its own internal copy of the geometry being passed into the node. inMesh is only valid if there is an input connection. Otherwise it is ignored.

outMesh

The standard output attribute of the polyShape node. This attribute receives input mesh data from either the inMesh or cachedInMesh (depending on the node state) and stores it as its own internal copy of the mesh. The outMesh geometry represents the final geometry of the shape and is always valid.

cachedInMesh

The simulated input attribute of the polyShape node. This attribute is only ever initialized and used in the case where the inMesh attribute is invalid (that is, no input connection) and tweaks exist on the mesh. It also stores its own internal copy of the geometry.

pnts

The tweaks attribute. This is an array attribute that stores the position offsets for each vertex in the geometry, representing manual “tweaks” or modifications to these basic components. The presence of tweaks is determined by looking for a non-zero value in the array attribute.

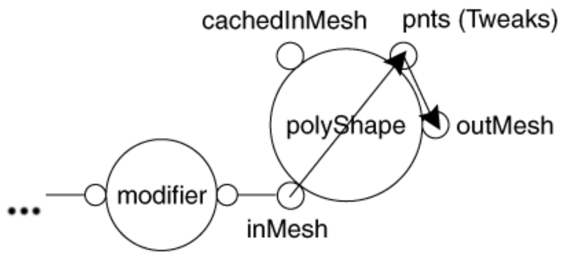
polyShape data flow

The data flow of the polyShape node is dependent on two factors:

- Construction history
- Tweaks

A node with construction history and tweaks implies that there is an input connection present on the node and a non-zero value present in the pnts attribute. In this case the inMesh is valid and upon receiving the mesh data from the upstream history, the pnts attribute is applied to the inMesh data and the resultant mesh stored in the outMesh.

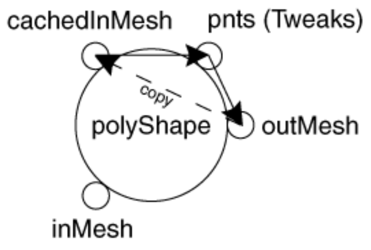
For a node with construction history and no tweaks, the inMesh is redirected to the outMesh. The following illustration shows the data flow for the case where construction history is present as well as tweaks.



Data flow with history and tweaks

A node without construction history and without tweaks implies that there is no input connection present on the node and an all-zero pnts attribute array. In this case the inMesh is invalid. However since there is no need to redirect any data, nor apply any tweaks, the geometry of the polyShape node is the outMesh itself.

A node without construction history and with tweaks implies that there is no input connection present on the node and a non-zero value present in the pnts attribute. In this case you cannot use the outMesh since you need to apply tweaks. Applying tweaks directly on the outMesh will result in losing the former values without the ability to undo. As a result, you need a simulated input—the cachedInMesh—which will store the current state of the outMesh before tweaks are applied and re-evaluate the node. The outMesh geometry is copied to cachedInMesh as soon as tweaks are applied to the node. From there the cachedInMesh behaves just like the inMesh with construction history and tweaks. The pnts attribute is applied to the cachedInMesh and subsequently forwarded to the outMesh.



Data flow without history but with tweaks

Interfacing with the node

Interfacing with a polyShape node involves two basic actions:

- accessing data
- modifying/creating data

Most of the Poly API provides accessor and mutator methods that understand the anatomy of a polyShape node. As a result, you can use these methods to properly interface with the geometry of the polyShape node. However, you may need to interface with the node directly, for example, when backing up the mesh data of a node. These operations involve mostly DG operations such as retrieving plugs, setting plugs and retrieving plug data, etc.

Accessing data

The outMesh always has the most up to date information as it represents the final resultant mesh in the polyShape node. Consequently, the outMesh is where all accessors retrieve their information. In the example of backing up a mesh, it is the outMesh that you would backup as it represents the current state of the node.

Modifying/creating data

The same two factors that affect the data flow of the node also affect how the node should be modified—construction history and tweaks.

In the case where construction history is present (an input connection exists), there is an upstream node from the polyShape node. During a DG evaluation the data that is passed to the polyShape node will overwrite the inMesh of the polyShape node, which in turn updates the outMesh. As a result, setting an attribute on the polyShape node or any other “direct modification” to the node should be avoided if history exists, as the change will be overwritten by the next DG evaluation. To modify the mesh, a modifier node containing the modification needs to be inserted ahead of the polyShape node. This applies regardless of the presence of tweaks, so long as history exists. For more details see ”Construction

History" on page 161. This case requires less direct interaction with the node itself since the DG requires that a modifier node be inserted ahead to avoid overwriting the changes made.

In the case where construction history is not present, you can create history or modify the node directly. This depends on the "Record History" preference in Maya. If the user chooses to create history, then the case is similar to the above case with some minor tweaks and involves little interaction with the node itself other than setting up connections.

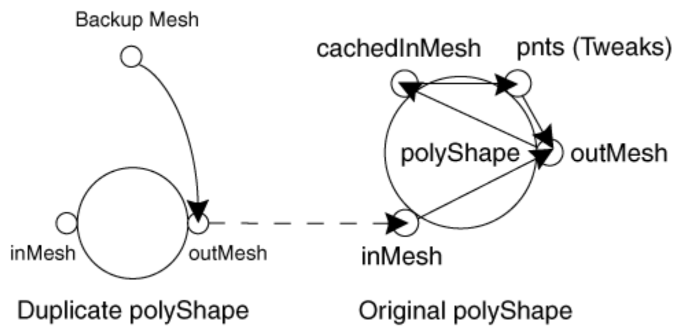
Attempting to write to the node directly where you do not want to create history involves more interaction and understanding of the purpose of the shape's attribute composition. For a node without history and tweaks, the outMesh represents the only geometry of the shape node with all other mesh attributes ignored. Under such a case you can operate directly on the outMesh. So if you obtain a backup mesh, you can reapply the backup mesh to the outMesh and the node will be reverted to its original state.

When the node is without history but has tweaks, a cachedInMesh is generated as a copy of the outMesh and used to apply the tweaks to obtain the final mesh. During the outMesh copy to the cachedInMesh the node performs some synchronization among its attributes which is internal to the node and inaccessible from the API. This means you need to update the outMesh before the cachedInMesh is initialized. (In the example of backing up a mesh, you risk destabilizing the node if you simply copy the backup mesh to the cachedInMesh.) The recommended approach is to do the following:

1. Duplicate the shape node.
2. Copy the backup mesh into the outMesh attribute of the duplicate shape.
3. Connect the backup mesh's outMesh to the shape node's inMesh.
4. Force a DG evaluation.
5. Disconnect and delete the duplicate node.

(You could use the polyDuplicateAndConnect MEL command to perform the first three steps.)

This updates the outMesh, through the inMesh so that once the inMesh is invalidated by the disconnected node, the cachedInMesh will hold the original mesh backup before it applies the tweaks. This is shown in the "polyModifierCmd example" on page 164. The following illustration describes the data flow used to restore the backup of a mesh.



Restoring a backup mesh

The five basic polygonal API classes

The Poly API consists of five main classes:

- MItMeshPolygon
- MItMeshEdge
- MItMeshVertex and MItMeshFaceVertex
- MFnMesh

The first four classes are iterators while the last class is a function set. The polygon iterators are primarily used to navigate or parse a mesh component by component and retrieve component specific information. The polygon function set, MFnMesh, is used to create, modify, and retrieve mesh specific data.

Note Although the iterators contain a few methods that can be used to modify the mesh, it is good practice to rely on the iterators solely for navigating the mesh and accessing component specific data. MFnMesh provides all the necessary methods to perform any other desired operation on the mesh.

MItMeshPolygon

MItMeshPolygon is a polygonal face iterator. Initializing this class to a specific mesh object lets you iterate over all faces in a mesh, in order of face ids. Alternatively, the iterator can be restricted to the faces adjacent to a given component (for example, edge or vertex) by initializing the class to both a DAG path referring to the mesh and an MObject reference to certain components.

This iterator is useful for parsing a mesh as it can traverse the mesh more quickly—there are fewer faces than edges and vertices and less overlap in data retrieval. As a face iterator, MItMeshPolygon provides methods to retrieve face-specific data mostly comprised of:

- Face composition (a number of edges greater than 2)
- Face-vertex data
- Face-edge data
- Adjacent component data
- UV data
- Color per vertex data
- And other miscellaneous data, such as blind data and smoothing information…

MItMeshPolygon is ideal for cases where you would like to quickly search a mesh on a face-by-face basis or when you require face specific data from the mesh. To see how this class is used, refer to the ”splitUVCmd example” on page 186. The following example also illustrates the use of the MItMeshPolygon class. In this sample code, MItMeshPolygon is used to traverse the mesh for a specific face and then retrieve the edges making up the face.

```
MStatus getFaceEdges( MObject mesh,
                      int faceId,
                      MIntArray faceEdges )
{
   MStatus status;

   // Reset the faceEdges array
   //
   faceEdges.clear();

   // Initialize a face iterator and function set
   //
   MItMeshPolygon faceIter( mesh, &status );
   MCheckStatus( status, “MItMeshPolygon constructor failed”
);
   MFnMesh meshFn( mesh, &status );
   MCheckStatus( status, “MFnMesh constructor failed” );

   // Check to make sure that the faceId passed in is valid
   //
   if( faceId >= meshFn.numPolygons() || faceId < 0 )
   {
      cerr << “Invalid faceId.\n”;
      status = MS::kFailure;
   }
   else
   {
      // Now parse the mesh for the given face and
      // return the edges
      //
      for( ; !faceIter.isDone(); faceIter.next() )
      {
         // If we find the matching face, retrieve the
         // edge indices
         //
         if( faceIter.index() == faceId )
         {
            faceIter.getEdges( faceEdges );
            break;
         }
      }
   }

   return status;
}
```

MItMeshEdge

MItMeshEdge iterates over the mesh on an edge-by-edge basis and retrieves edge specific data. This edge iterator iterates over the edges in order of edge ids or it iterates over the edges adjacent to a passed in component. MltMeshEdge can retrieve the following types of data:

- Edge composition (two vertices)
- Edge-face data
- Edge-vertex data
- Edge smoothing
- Adjacent component data

MItMeshEdge is best suited for an edge-by-edge traversal of the mesh and for fetching edge specific data. The following sample code illustrates the use of the edge iterator. This example traverses each edge in the mesh and collects their start vertices, storing them inside an array indexed by edge id.

```
MStatus getEdgeStartVertices( MObject mesh,
                              MPointArray& pointArray )
{
   MStatus status;

   // Clear the output array
   //
   pointArray.clear();

   // Initialize our iterator
   //
   MItMeshEdge edgeIter( mesh, &status );
   MCheckStatus( status, “MItMeshEdge constructor failed” );

   // Now parse the mesh
   //
   for( ; !edgeIter.isDone(); edgeIter.next() )
   {
      // Retrieve the start vertex of each edge and append 
it to
      // our point array. Use the default object coordinate
      // system for our space
      //
      pointArray.append( edgeIter.point(0, MSpace::kObject) 
);
   }

   return status;
}
```

MItMeshVertex and MItMeshFaceVertex

MItMeshVertex iterates over the mesh on a vertex-by-vertex basis in order of vertex ids, retrieving vertex specific data. The vertex iterator is best suited for those two cases and can retrieve vertex specific data such as:

- Vertex composition (a 3D position)
- Vertex-face data
- Vertex-edge data
- Vertex normals
- UV data (specific to a vertex)
- Color data (specific to a vertex)
- Adjacent component data

MItMeshFaceVertex iterates over the mesh on a face vertex-by-face vertex basis in order of face ids, retrieving face-vertex specific data. The face vertex iterator can retrieve data such as:

- Normal data
- UV data
- Color data

MFnMesh

MFnMesh contains several methods for retrieving mesh specific data and modifying a mesh. You could use an iterator to find a particular component and use MFnMesh to perform an operation on that component. This is shown in the ”splitUVCmd example” on page 186 which searches the mesh for a given UV and uses MFnMesh to “split” the UVs.

Although there is some overlap between the methods provided by MFnMesh and the MItMesh* iterators, MFnMesh represents more of a global library of operations for the mesh, while the iterators remain centric around their respective components. The following sample code demonstrates some things you might use MFnMesh for. The example retrieves various data and modifies it. Note that this code cannot be compiled.

```
// The argument list contains a “...” to represent a “Fill
in
// the data you would like here”
//
MStatus getRandomPolyData( MObject mesh, ... )
{
   MStatus status;

   // Initialize a function set to a polygonal mesh
   //
   MFnMesh meshFn( mesh, &status );
   MCheckStatus( status, “MFnMesh constructor failed” );

   // Retrieve topological information
   //
   int faceCount = meshFn.numPolygons();
   int edgeCount = meshFn.numEdges();
   int vertexCount = meshFn.numVertices();
   int faceVertexCount = meshFn.polygonVertexCount();
   int UVCount = meshFn.numUVs();

   MPointArray vertexList;
   meshFn.getPoints( vertexList );

   MFloatArray UArray;
   MFloatArray VArray;
   meshFn.getUVs( UArray, VArray );

   // Modify topological information
   //
   // Add a UV to the UV list - setUV will automatically
grow
   // the UV list, based on the given index
   //
   meshFn.setUV( numUVs, 0.0, 0.0 );

   // Move vertex 0 to the origin of the world
   //
   MPoint origin( 0.0, 0.0. 0.0 );
   meshFn.setPoint( 0, origin, MSpace::kWorld );

   // Can also work with:
   //
   // - Vertex Colors
   // - Blind data
   // - etc.
   //
}
```

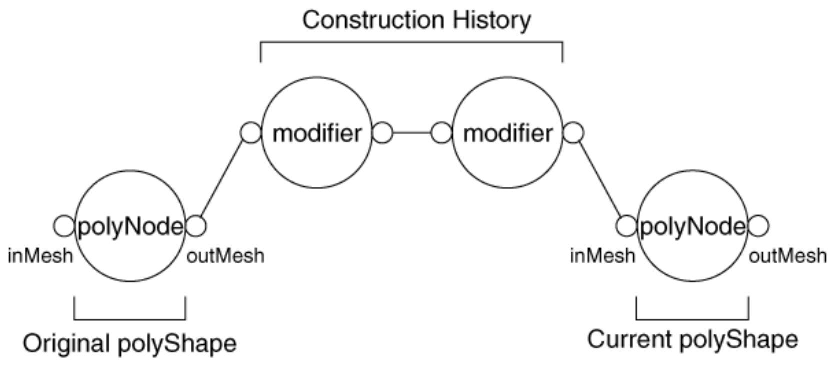
Construction History and Tweaks

This section assumes basic knowledge of the dependency graph. For details, see ″Dependency Graph (DG) nodes″ on page 84.

All the operations that can be performed on a polygonal mesh can be generalized into three basic types: create, edit and query. While both the creating and querying a mesh are straightforward, editing involves complications resulting from construction history and tweaks and how they work within the confines of the dependency graph.

Construction History

Construction history provides a backlog of actions performed on a mesh. The implementation of construction history in the DG makes it unique. For all objects in Maya, a single linear chain of DG nodes can exist upstream from the object’s node. This chain is known as the construction history of an object. The final node of a history chain always represents the object that the history is recording actions for. At the beginning of the chain lies a hidden intermediate node representing the initial state of the node when the history first began to record actions performed on the mesh.



Construction history chain

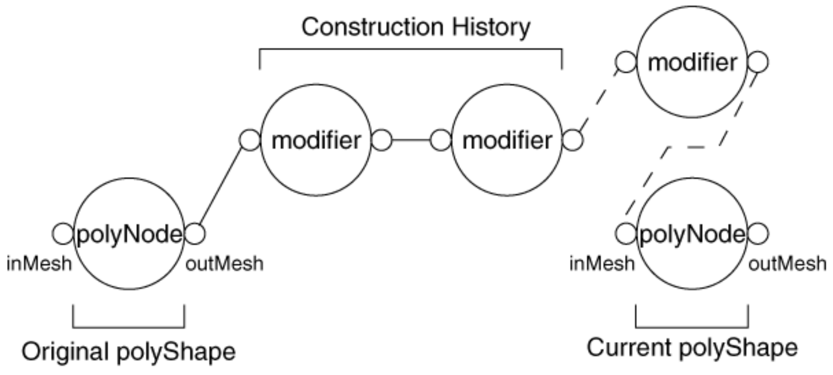
The manipulation of the mesh can be complicated by the following factors, which affect the state of a node:

- Whether or not the mesh has construction history
- Whether or not the user has construction history recording turned on

The presence of construction history indicates that there is a chain of modifier nodes upstream from the mesh node—the history chain. Each modifier node in the history chain is connected via their inMesh/outMesh attributes through which the mesh data flows down the chain. During a DG evaluation, the outMesh at the top of the history chain passes the mesh down to each modifier node, each applying their modification in turn. Once the mesh reaches the actual mesh node and end of the chain, the modified mesh is stored on the node, overwriting any previous data on the node.

Attempting to modify a mesh node via API mutator methods writes the information directly onto the inMesh attribute of the given mesh node. Although some methods are history sensitive, there are many that are not. This, combined with the DG evaluation process presents the problem if history exists. Since a DG evaluation will overwrite the inMesh of the mesh node due to the connection from the outMesh of the modifier node directly upstream from the mesh, any modifications made to the mesh will be discarded.

The solution to this problem is to create a node that performs the modification want and if history exists, to insert your node directly ahead of your mesh node in the history chain as shown below. The node directly upstream from the mesh node always represents the last change made to the mesh.



Inserting a modifier node

Whether the user has construction history turned on or off does not restrict the modification of a node as rigidly as whether the mesh has construction history. However it is a good practice to adhere to the user's construction history preferences and behave similarly to the rest of Maya. This preference will change how the node should be modified as well as what the node will look like in the DG following the operation.

With history turned on, the user has selected to keep a history chain. Following the operation, the resulting mesh would look like the previous diagram (Inserting a modifier node).

With history turned off, the user has selected not to see a history chain. From here there are two possible ways to modify the mesh:

- Operate on the mesh directly.
- Use a modifier node as shown in the previous diagram (Inserting a modifier node), and then collapse the node down into the mesh.

Note If history already exists on the mesh and history is turned off, the preference is ignored and regarded as though history was turned on. This leaves it at the user's discretion to collapse all history down into the mesh node.

Tweaks

Tweaks are manual transformations applied to polygonal components (for example, manually repositioning a vertex through a translate transformation). The presence of tweaks on a shape complicates the interface of a polyShape node because they change the data flow through the node. Although the change isn't drastic it affects how you modify the node. The challenge with tweaks is maintaining the order of operations.

Tweaks are stored locally on the node and applied to the input mesh of the node (inMesh or cachedInMesh). If a node is inserted ahead of the polyShape node (common for a node with history), the order of operations is not kept. This can change the resulting mesh if a modifier node was altering the topology of the mesh. For example, if the modifier node rotated the edge containing a tweaked vertex, the resulting position of the edge would vary based on the order of operations.

The polyTweak node resolves this problem. This node stores tweaks inside a local tweak attribute. Upon receiving a mesh input (inputPolymesh attribute), the node applies its tweaks to the mesh and returns the output through its output attribute. To address the problem with this node, you extract the tweaks into the polyTweak node and insert the polyTweak node ahead of the modifier node. This maintains the order of operations.

polyModifierCmd example

The source code for polyModifierCmd is located in the devkit\plug-ins directory.

polyModifierCmd encapsulates the generic process and interface with the Maya architecture for creating a command that can modify a polygonal mesh. Although it deals entirely with polygons, it can be extended to other object types in Maya as the DG concepts are closely knit.

This process is as follows:

1. Modify the data.

   This part of the process deals only with the geometry or mesh data.

2. Apply it to the polyShape node.

   Common to all poly modifier commands, this part of the process contains all the interaction with the Maya architecture regarding construction history, tweaks, and the polyShape node.

Because this is a single action, to apply a modifier to a polyShape node you can make use of the doIt(), undoIt() and redoIt() class structure to implement the class. However, since polyModifierCmd is a subset of MPxCommand, it can derive itself off of MPxCommand so that any derived polyModifierCmd classes will inherit the full command architecture. The only consequence of this is that you cannot override the doIt(), undoIt() and redoIt() methods as they are required by the actual command class to perform the operation. Instead you define our own group of similar methods: doModifyPoly(), undoModifyPoly() and redoModifyPoly().

A derived class can initialize the polyModifierCmd and proceed to call doModifyPoly() inside the doIt() and can then extend the respective capabilities to the undo and redo. The "splitUVCmd example" on page 186 shows the implementation of a derived polyModifierCmd command. Below is the class interface for the polyModifierCmd.

```
class polyModifierCmd : MPxCommand
{
public:
            polyModifierCmd();
   virtual  ~polyModifierCmd();

// Restrict access to derived classes only
//
protected:

   ////////////////////////////////////
   // polyModifierCmd Initialization //
   ////////////////////////////////////

   // Target polyMesh to modify
   //
   void         setMeshNode( MDagPath mesh );
   MDagPath     getMeshNode() const;

   // Modifier node type
   //
   void         setModifierNodeType( MTypeId type );
   void         setModifierNodeName( MString name );
   MTypeId      getModifierNodeType() const;
   MString      getModifierNodeName() const;

   ///////////////////////////////
   // polyModifierCmd Execution //
   ///////////////////////////////

   virtual MStatus   initModifierNode( MObject modifierNode
);
   virtual MStatus   directModifier( MObject mesh );

   MStatus           doModifyPoly();
   MStatus           redoModifyPoly();
   MStatus           undoModifyPoly();

private:
   ////////////////////////////////////////////////
   // polyModifierCmd Internal Processing Data //
   ////////////////////////////////////////////////

   struct modifyPolyData
   {
      MObject meshNodeTransform;
      MObject meshNodeShape;
      MPlug   meshNodeDestPlug;
      MObject meshNodeDestAttr;

      MObject upstreamNodeTransform;
      MObject upstreamNodeShape;
      MPlug   upstreamNodeSrcPlug;
      MObject upstreamNodeSrcAttr;

      MObject modifierNodeSrcAttr;
      MObject modifierNodeDestAttr;

      MObject tweakNode;
      MObject tweakNodeSrcAttr;
      MObject tweakNodeDestAttr;
   };

   ////////////////////////////////////////
   // polyModifierCmd Internal Methods //
   ////////////////////////////////////////

   // Preprocessing methods
   //
   bool      isCommandDataValid();
   void      collectNodeState();

   // Modifier node methods
   //
   MStatus   createModifierNode( MObject& modifierNode );

   // Node processing methods (need to be executed in this
order)
   //
   MStatus   processMeshNode( modifyPolyData& data );
   MStatus   processUpstreamNode( modifyPolyData& data );
   MStatus   processModifierNode( MObject modifierNode,
                                  modifyPolyData& data );
   MStatus   processTweaks( modifyPolyData& data );

   // Node connection methods
   //
   MStatus   connectNodes( MObject modifierNode );
```

```
// Mesh caching methods
//
MStatus   cacheMeshData();
MStatus   cacheMeshTweaks();

// Undo methods
//
MStatus   undoCachedMesh();
MStatus   undoTweakProcessing();
MStatus   undoDirectModifier();

/////////////////////////////////////
// polyModifierCmd Utility Methods //
/////////////////////////////////////

MStatus   getFloat3PlugValue( MPlug plug,
                              MFloatVector& value );
MStatus   getFloat3asMObject( MFloatVector value,
                              MObject& object );

//////////////////////////
// polyModifierCmd Data //
//////////////////////////

// polyMesh
//
bool      fDagPathInitialized;
MDagPath  fDagPath;
MDagPath  fDuplicateDagPath;

// Modifier node type
//
bool      fModifierNodeTypeInitialized;
bool      fModifierNodeNameInitialized;
MTypeId   fModifierNodeType;
MString   fModifierNodeName;

// Node State Information
//
bool      fHasHistory;
bool      fHasTweaks;
bool      fHasRecordHistory;

// Cached Tweak Data
//
MIntArray           fTweakIndexArray;
MFloatVectorArray   fTweakVectorArray;

// Cached Mesh Data
//
```

```
    MObject   fMeshData;

    // DG and DAG Modifier
    //
    MDGModifier   fDGModifier;
    MDagModifier  fDagModifier;
};
```

In section of the class interface labeled // polyModifierCmd Execution // , notice the corresponding doModifyPoly(), undoModifyPoly() and redoModifyPoly() method definitions. These three methods represent the core of the interface that derived commands will interface with.

polyModifierCmd has three basic stages:

- polyModifierCmd initialization
- polyModifierCmd preprocessing
- polyModifierCmd processing

Of the most importance however is to understand the basic class interface of polyModifierCmd (doModifyPoly(), undoModifyPoly(), redoModifyPoly()).

polyModifierCmd initialization

Before any modifier can be applied to a polyShape node, the polyModifierCmd requires some initialization data to guide the process. This data is distinctive from preprocessing data, since this is the required input to get the ball rolling, while preprocessing data is data extracted from our initial data. There are only two pieces of initialization data necessary to perform the operation, through which all other data can be extracted:

- A polyShape node—you need a mesh to apply the modifier to.
- A modifier—you need a modifier to apply to the mesh.

(“A modifier”, means the actual modification made to the mesh data, exclusive of the manner through which it is applied. For example, the modifier in the case where construction history exists would be applied as a modifier node.)

The polyShape node can be stored in the form of a DAG path. It is recommended that you use a DAG path rather than an MObject since the DAG path is absolute and guaranteed to be pointing to the proper object, whereas the MObject is a handle to an object owned by Maya which could change between calls to a plug-in. The polyShape input is represented in the class interface as:

```
// Prototypes
//
void      setMeshNode( MDagPath mesh );
MDagPath  getMeshNode() const;

MDagPath  fDagPath;

// Implementations
//
void polyModifierCmd::setMeshNode( MDagPath mesh )
{
   fDagPath = mesh;
}

MDagPath  polyModifierCmd::getMeshNode() const
{
   return fDagPath;
}
```

The modifier can be applied in two forms:

- Through a modifier node
- Directly on the mesh

Applying the modifier through a modifier node requires a DG node type that you can create and connect to the polyShape. Since it provides something tangible to work with, we provide an interface for you to initialize polyModifierCmd with a node type or node name:

```
// Prototypes
//
void      setModifierNodeType( MTypeId type );
void      setModifierNodeName( MString name );
MTypeId   getModifierNodeType() const;
MString   getModifierNodeName() const;

virtual MStatus   initModifierNode( MObject modifierNode );

bool      fModifierNodeTypeInitialized;
bool      fModifierNodeNameInitialized;
MTypeId   fModifierNodeType;
MString   fModifierNodeName;

// Implementations
//
void polyModifierCmd::setModifierNodeType( MTypeId type )
{
   fModifierNodeType = type;
   fModifierNodeTypeInitialized = true;
}

void polyModifierCmd::setModifierNodeName( MString name )
{
   fModifierNodeName = name;
   fModifierNodeNameInitialized = true;
}

MTypeId polyModifierCmd::getModifierNodeType() const
{
   return fModifierNodeType;
}

MString polyModifierCmd::getModifierNodeName() const
{
   return fModifierNodeName;
}

MStatus polyModifierCmd::initModifierNode( MObject
modifierNode )
{
   return MS::kSuccess;
}
```

The initModifierNode() method does not have any role in the node type that is created but rather the node creation. Often modifier nodes require an absolute input to tell the node how to modify the data. The splitUVNode, for example, requires a list of UVs to split. The problem that arises here is that if polyModifierCmd creates the node, how does the derived command initialize other data on the node? polyModifierCmd cannot do this because it is indifferent to the modifier—it just knows how to connect it. To get past this problem, we provide a callback in the form of a virtual method for derived commands to override and expect to be executed prior to the use of a modifier node.

In contrast to the modifier node, applying the modifier by direct means provides nothing tangible to store and apply. That is, while a modifier node contains the modification inside an object thereby separating the modification from the polyShape node, a direct modifier does not. Since a direct modifier works directly on the polyShape node and polyModifierCmd needs to be independent of the modification process, we provide a callback to derived commands. This callback would be executed when there is a need for a direct modifier (that is, in the case with no construction history and construction history is turned off). Below, the code for the directModifier callback is present in the form of a virtual method that is called when the polyModifierCmd deems it appropriate.

```
// Prototypes
//
virtual MStatus   directModifier( MObject mesh );

// Implementations
//
MStatus polyModifierCmd::directModifier( MObject /* mesh */ )
{
   return MS::kSuccess;
}
```

polyModifierCmd preprocessing

Once you have our initialization data you can extract the rest of the data necessary to apply the modifier. In addition to our initialization data you need to know if:

- Initialization data is valid
- Construction history is present
- Tweaks are present
- Construction history is turned on

The first piece of information is a check to ensure that you can continue on with the given data. It consists of a check of the initialization of a polyShape node and modifier node information. If the data is invalid, polyModifierCmd cannot continue to execute and returns a failure:

```
// Prototypes
//
bool   isCommandDataValid()

// Implementations
//
bool polyModifierCmd::isCommandDataValid()
{
   bool valid = true;

   // Check the validity of the DAG path
   //
   if( fDagPathInitialized )
   {
      // Ensure we are pointing to a shape node
      //
      fDagPath.extendToShape();
      if( !fDagPath.isValid() || fDagPath.apiType !=
MFn::kMesh )
      {
         valid = false;
```

```
            }
        }
        else
        {
            valid = false;
        }

        // Check the validity of the modifier node type/name.
        // Can only check that it has been set.
        //
        if( !fModifierNodeTypeInitialized &&
            !fModifierNodeNameInitialized )
        {
            valid = false;
        }
    }
```

The next three pieces of information relate to the node state of the polyShape node. Since these three pieces of data require the polyShape node in order to extract out their data, the validity of the polyShape node is required. The data extraction is straightforward and compiled into a single method, collectNodeState:

```
// Prototypes
//
void   collectNodeState();

// Implementations
//
void polyModifierCmd::collectNodeState()
{
    MStatus status;

    // Collect the node state information on the given mesh
    //
    // -HasHistory
    // -HasTweaks
    // -HasRecordHistory
    //
    fDagPath.extendToShape();
    MObject meshNodeShape = fDagPath.node();

    MFnDependencyNode depNodeFn( meshNodeShape );

    // If the inMesh is connected, we have history
    //
    MPlug inMeshPlug = depNodeFn.findPlug( "inMesh" );
    fHasHistory = inMeshPlug.isConnected();
```

```
   // Tweaks exist only if the multi “pnts” attribute
contains
   // plugs that contain non-zero tweak values. Use false,
   // until proven true search pattern.
   //
   fHasTweaks = false;
   MPlug tweakPlug = depNodeFn.findPlug( “pnts” );
   if( !tweakPlug.isNull() )
   {
      // ASSERT : tweakPlug should be an array plug
      //
      MAssert( tweakPlug.isArray(), “tweakPlug.isArray()” );

      MPlug tweak;
      MFloatVector tweakData;
      int i;
      int numElements = tweakPlug.numElements;

      for( i = 0; i < numElements; i++ )
      {
         tweak = tweakPlug.elementByPhysicalIndex( i,
&status );
         if( status == MS::kSuccess && !tweak.isNull() )
         {
            // Retrieve the float vector from the plug
            //
            getFloat3PlugValue( tweak, tweakData );
            if( 0 != tweakData.x ||
                0 != tweakData.y ||
                0 != tweakData.z )
            {
               fHasTweaks = true;
               break;
            }
         }
      }
   }

   // Query the constructionHistory command for the
preference
   // of recording history
   //
   int result;
   MGlobal::executeCommand( “constructionHistory -q -tgl”,
                            result );
   fHasRecordHistory = (0 != result );
}
```

polyModifierCmd processing

The points of entry are:

- doModifyPoly()
- redoModifyPoly()
- undoModifyPoly()

doModifyPoly()

doModifyPoly() is the most complex piece and the core of the polyModifierCmd. In here all the data is parsed and the appropriate action given. Rather than implementing the entire process in a single method, doModifyPoly() only focuses on quickly scanning the node states and passes control over the appropriate method based on the state:

```
MStatus polyModifierCmd::doModifyPoly()
{
   MStatus status = MS::kFailure;

   if( isCommandDataValid() )
   {
      // Get the state of the polyMesh
      //
      collectNodeState();

      if( !fHasHistory && !fHasRecordHistory )
      {
         MObject meshNode = fDagPath.node();

         // Pre-process the mesh - Cache the old mesh
         //
         cacheMeshData();
         cacheMeshTweaks();

         // Call the directModifier
         //
         status = directModifier( meshNode );
      }
      else
      {
         MObject modifierNode;
         createModifierNode( modifierNode );
         initModifierNode( modifierNode );
         connectNodes( modifierNode );
      }
   }
}
```

The skeleton of doModifyPoly() shows the concepts of how to interface with a polyShape node given its state. If you have no history and recording history is turned off, you cache the mesh data for undo purposes (explained with undoModifyPoly()) and proceed to call the directModifier callback. Otherwise the modifierNode approach is taken. Calls are made to create the modifier node, initialize it (through the callback) and subsequently passed on to attempt to connect the nodes.

Note that doModifyPoly() processes only the construction history states. Though tweaks do play a role in the process, they are considered further on in the process separately since they are independent from construction history. The following table shows the code path that is followed based on the node’s construction history state:

	Record History – On	**Record History – Off**
History – Exists	connectNodes()	connectNodes()
History – Does Not Exist	connectNodes()	directModifier()

The directModifier line of code is implemented by the derived command and as such has nothing complex to worry about. It passes the derived command the mesh node to operate on directly—all the control rests with the derived command.

In contrast, the three other cases regarding history are more involved. Beginning with the creation of the modifierNode:

```
MStatus polyModifierCmd::createModifierNode( MObject&
modifier )
{
   MStatus status = MS::kFailure;

   if( fModifierNodeTypeInitialized ||
       fModifierNodeNameInitialized )
   {
      if( fModifierNodeTypeInitialized )
      {
         modifier =
fDGModifier.createNode(fModifierNodeType,
                                             &status);
      }
      else if( fModifierNodeNameInitialized )
      {
         modifier =
fDGModifier.createNode(fModifierNodeName,
                                                &status);
      }

      // Check to make sure that we have a modifier node of
the
      // appropriate type. Requires an inMesh and outMesh
      // attribute.
      //
      MFnDependencyNode depNodeFn( modifier );
      MObject inMeshAttr;
      MObject outMeshAttr;
      inMeshAttr = depNodeFn.attribute( "inMesh" );
      outMeshAttr = depNodeFn.attribute( "outMesh" );

      if( inMeshAttr.isNull() || outMeshAttr.isNull() )
      {
         displayError("Invalid Modifier: inMesh/outMesh
needed");
         status = MS::kFailure;
      }
   }

   return status;
}
```

createModifierNode() uses the initialized data for the modifier node type or modifier node name to create the modifier node via the local DG modifier. Use of the DG modifier is essential to keep the undo/redo relatively simple. Though the DG modifier has not created the node yet, it allows you to queue up actions for that node such as connections, so that once the DG modifier's doIt() is called everything is executed in order. This helps alleviate rollback issues in case of errors. createModifierNode() also does a few checks to ensure that you have an inMesh and outMesh attribute on the node. Though the names may seem restrictive they keep a standard in the chains of modifier nodes. A helper class named polyModifierNode (discussed in a later section) automatically generates these two key attributes.

Following the createModifierNode() the callback to the initModifierNode() is made through which a derived class can initialize node data. This also lies under the control of the derived command. From there you enter the final stage of the process—connectNodes().

connectNodes()

connectNodes() is a larger method which processes all the variables of the polyShape and modifier nodes and connects them. Look at the connectNodes() implementation for a higher level view of what it controls:

```
MStatus polyModifierCmd::connectNodes( MObject& modifierNode
)
{
   MStatus status;

   // Declare our internal processing data structure
   //
   modifyPolyData data;

   // Get the mesh node attrs and plugs
   //
   status = processMeshNode( data );
   MCheckStatus( status, "processMeshNode" );

   // Get the upstream node attrs and plugs
   //
   status = processUpstreamNode( data );
   MCheckStatus( status, "processUpstreamNode" );

   // Get the modifierNode attributes
   //
   status = processModifierNode( modifierNode, data );
   MCheckStatus( status, "processModifierNode" );

   // Process the tweaks on the meshNode
   //
   status = processTweaks( data );
   MCheckStatus( status, "processTweaks" );

   // Connect the nodes
   //
   if( fHasTweaks )
   {
      status = fDGModifier.connect( data.upstreamNodeShape,

data.upstreamNodeSrcAttr,
                                    data.tweakNode,
                                    data.tweakNodeDestAttr
);
      MCheckStatus( status, "upstream-tweak connect" );

      status = fDGModifier.connect( data.tweakNode,
                                    data.tweakNodeSrcAttr,
                                    modifierNode,

data.modifierNodeDestAttr );
      MCheckStatus( status, "tweak-modifier connect" );
   }
   else
   {
      status = fDGModifier.connect( data.upstreamNodeShape,

data.upstreamNodeSrcAttr,
                                    modifierNode,

data.modifierNodeDestAttr );
      MCheckStatus( status, "upstream-modifier connect" );
   }

   status = fDGModifier.connect( modifierNode,
                                 data.modifierNodeSrcAttr,
                                 data.meshNodeShape,
                                 data.meshNodeDestAttr );
   MCheckStatus( status, "modifier-mesh connect" );

   status = fDGModifier.doIt();
   return status;
}
```

connectNodes() is broken down into several subsections. First a general processing data structure is constructed. This contains all of the processing variables required between each of the processing methods. The structure was created to reduce the amount of argument passing between processing methods. From there a set of processing methods are called to collect the necessary data to connect the nodes and also to process node state specific intricacies. Following that it uses the collected data to connect the nodes.

For details on each of the processing methods that follow, refer to the source code in polyModifierCmd.cpp, which is documented thoroughly.

processMeshNode()

The first process method that the connectNodes() runs through is processMeshNode(). It processes all that's necessary of the polyShape node, which is comprised of the node shape, node transform and connection data (the inMesh plug and attribute). This data is stored inside the passed in modifyPolyData data structure to be used further on in the other process methods. The order that these process methods are run is important.

processUpstreamNode()

For the second process method, processUpstreamNode(), there is a differance between the history exists case and the history does not exist case. ConnectNodes() is only called in the case where history exists or the case where history does not exist but history recording is turned on. The reason for this is that in each of those cases, the addition of history is permissible and more stable and flexible so we process it that way.

If history already exists, processUpstreamNode() uses the inMesh plug obtained in processMeshNode() to retrieve the node directly upstream from the polyShape node. Once you've obtained the node, you disconnect the node from the polyShape node so that you're prepared to insert the modifier node further on. Since all DG connections take place between the mesh node and the upstream node you only retrieve the outMesh plug and attribute for connections further on.

If history does not exist, you need to create the history chain. At the start of each history chain there is a hidden intermediate node that represents the initial state of the polyShape node at the time history was created. processUpstreamNode() does this by calling an MFnDagNode::duplicate() on the polyShape node. Since the duplicate() method also creates a new transform for the duplicate shape, you reparent the shape node under the same transform as the original shape and delete the transform through the DAG modifier. To the DG, this duplicate shape node behaves the same as the upstream node processed in the case where history exists. That is, all connections take place between this duplicate shape node and the original shape node. Thus you set the upstream node information to the duplicate shape and retrieve the outMesh plug and attribute for connections further on.

processModifierNode()

Like processMeshNode() the third process method, processModifierNode(), retrieves the inMesh and outMesh attributes of the modifier node for the connection of the nodes at the base of connectNodes().

processTweaks()

Because of history, tweaks need to be extracted prior to the addition of the modifier node to maintain the order of operations. So if tweaks exist, you go through two stages:

- Tweak extraction
- Tweak application

Begin by creating a tweak node (polyTweak) to store the tweaks and begin to parse the pnts attribute for tweaks. The tweaks are individually cached into a class array member for undo purposes as well as in a local MObject array for the purposes of transferring over to the tweak node. Although tweaks are composed of the vector arrays that you just extracted, that is not all that you must account for during the extraction. Tweaks are stored on a pnts attribute. As an attribute in the DG, it can also contain connections. These connections must be preserved and transferred over to the tweak node to maintain the DG structure.

In addition to each vector array, you extract a plug array of each node that is connected to the pnts attribute as well as each plug in the pnts attribute that is connected. Note that each attribute in the pnts array attribute is a compound attribute. Each attribute in the compound attribute is associated with a single axial translation for the given vertex tweak vector (that is, x, y, z).

Once you've extracted the data, you apply the tweak vectors on the polyTweak node and reconnect any connections on each tweak vector. Now you have a tweak node that is ready to be connected to the history stream. But if you connect it as is, you would expect the resulting mesh to have double the tweaks. This is because the tweaks have not been removed from the mesh node. So the tweaks would be applied twice, once from the tweak node and once from the mesh node. However, recall that you used an MObject array to retrieve the plugs. These retrieved an MObject reference to the compound plug that was contained in the array attribute. Setting these back onto the tweak node moved them over to the tweak node. There is a large segment of code that is commented out in processTweaks() that removes the tweaks from the mesh node. Executing this code gives the same result, however it slows down the performance with no apparent benefit.

Like the other process methods, processTweaks() also retrieves connection data for the tweak node to allow connectNodes() to connect all the nodes.

connectNodes() revisited

All the necessary nodes have been set up and the connection data extracted. The node connections are set up via MDGModifier::connect calls(). And all the operations are executed via a final MDGModifier::doIt() call. From there you can implement undo support by calling the undoIt() in the opposite order that the DG modifiers were executed in, and implement redo support as the doIt() calls in the same order as before. This is accomplished since the DG modifier caches all the executions that it made. Likewise, polyModifierCmd caches all the necessary data through the doModifyPoly() call. This leaves the undoModifyPoly() and redoModifyPoly() relatively simple.

undoModifyPoly()

undoModifyPoly is very similar to the doModifyPoly() method in structure and level of abstraction. It looks at the scenario from which it is being called (that is, node states) and caters the control over to the appropriate methods:

```
MStatus polyModifierCmd::undoModifyPoly()
{
   MStatus status;
   if( !fHasHistory && !fHasRecordHistory )
   {
      status = undoDirectModifier();
   }
   else
   {
      fDGModifier.undoIt();

      // undoCachedMesh() must be called prior to
      // undoTweakProcessing() because undoCachedMesh()
      // copies the original mesh without tweaks back
      // onto the existing mesh. Any changes made before
      // the copy will be lost.
      //
      if( !fHasHistory )
      {
         status = undoCachedMesh();
         fDagModifier.undoIt();
      }

      status = undoTweakProcessing();
   }

   return status;
}
```

undoDirectModifier()

The undoDirectModifier is not as simple as directModifier. Since polyModifierCmd is not aware of what the directModifier does, it must revert the entire mesh back into its original state. It does this through a unique use of MObject handles, making use of the knowledge of how MObjects work.

MObjects are said to be handles to internal objects owned by Maya. That is partly true. MObjects are handles, but Maya does not always own them. There are certain objects in Maya that are reference counted. That is, for each reference to the object a count is incremented. Once each reference is deleted the count is decremented until it reaches zero where the object is deleted. It's like an object that lives only if it is being used. MObjects may refer to such reference counted objects that increment the reference count. Thus even though Maya owns these objects, you can have some control over the lifetime of that object by holding onto the MObject handle to such an object. For other types of objects the general tip for not hanging onto an MObject is valid and still highly recommended as the data can change between calls to a plug-in (for example, a deleted node).

The types of data that are reference counted are objects classified under the MFnData and MFnComponent class hierarchy. Fortunately the MFnData object type includes the entire contents of a mesh. Using this concept as a foundation for backing up a mesh we can properly undo the direct modifier.

cacheMeshData()

cacheMeshData() caches the data on the use of the mesh prior to the directModifier(). Caching the mesh data makes use of the MObject concept (see undoDirectModifier()). To make a backup of the data you must be careful not to retrieve the reference to the current object, as you will then be holding onto a reference of possibly dirtied data. To get around this duplicate the current mesh and retrieve an MFnData object by extracting an MObject off the outMesh attribute of the duplicate mesh. The MObject that you retrieved is a reference to the original mesh data and thus it increases the reference count. To make this transparent to the user finish up by deleting the nodes created by the duplicate. Note that this does not delete the mesh data since you now have a reference to it.

cacheMeshTweaks()

This method is similar to the processTweaks() method except it does not deal with tweak nodes. Instead it parses the pnts attribute and extracts the tweak vectors into the tweak cache data members.

undoDirectModifier() revisited

You can put the caching of data required to backup our mesh to use in undoDirectModifier(). There is a distinct difference in the data flow of a polyShape node with tweaks and without. This directly affects the way the backup mesh is reapplied.

Tweaks affect the point of reapplication of the backup mesh for reasons of the data flow inside the node. For a node without tweaks you can set the value of the outMesh to the backup mesh.

For a node with tweaks you have to use a trick to force the copy of the backup mesh to the cachedInMesh to keep the node in sync. In this case you duplicate the polyShape node, set the outMesh of the duplicate shape to the backup mesh, and then connect the duplicate shape node to the original node. After forcing an evaluation, disconnect the duplicate shape node and delete it. This implicitly forces the outMesh to hold the backup mesh. Then reapply the initial tweaks to the node via undoTweaksProcessing, forcing the node to copy the backup mesh to the cachedInMesh and perform the internal node synchronization.

undoModifyPoly() revisited

For the case where connectNodes() needs to be undone, the MDGModifier::undoIt() method recalls all the connections and nodes created by connectNodes(). Recall that processUpstreamNode() also made use of a DAG modifier in the case where there was no history initially. In this case you need to perform some extra operations prior to calling the MDagModifier::undoIt().

The reason for this is that since you created history and you're undoing the operation you need to remove history. Removing is straightforward, however the node's outMesh attribute still holds the last known evaluation of the node—it still contains the modifier. As a result you need to undo the "cached" mesh on the node. To do this, call the method undoCachedMesh(). Following the restoration of the mesh data, restore the tweaks leaving ~ with the original mesh.

undoCachedMesh()

Similar to the undoDirectModifier() case, the operations contained in undoCachedMesh() are dependent on the presence of tweaks due to the change in data flow through the node. This is because you are reverting to a node without history and you need to restore the mesh by directly interfacing with the node.

For the case where tweaks do not exist, you only need to restore the outMesh since it becomes the geometry that represents the node. To do this, use the duplicate shape node that was created to start a history chain (recall that this is only done if history was not initially present). Retrieve the outMesh of the duplicate shape node and copy that mesh data over to the outMesh of the shape node, restoring the node to its initial state.

For the case where tweaks do exist, you need to access the node in a similar manner to the way undoDirectModifier handles the tweaks case, however this time you don't need to duplicate the shape node since you already have one. Similarly reconnect the outMesh of the duplicate shape to the inMesh of the original shape through a local DG modifier and force a DG evaluation. Then undo the connection via the same DG modifier. The original mesh data is subsequently forced into the outMesh prior to the reapplication of tweaks whereupon the outMesh will be copied to the cachedInMesh.

redoModifyPoly()

redoModifyPoly() is straightforward because doModifyPoly() initialized and set up all that was necessary for the operation. redoModifyPoly() holds a similar structure to its counterparts. For the directModifier() case, the doModifyPoly() recalls that method without caching any of the mesh data again, since the class already has it. Otherwise, in the connectNodes() case, it recalls the MDGModifier::doIt() to redo the operations previously set up by doModifyPoly():

```
MStatus polyModifierCmd::redoModifyPoly()
{
   MStatus status = MS::kSuccess;

   if( !fHasHistory && !fHasRecordHistory )
   {
      MObject meshNode = fDagPath.node();

      // Call the directModifier - No need to pre-process 
the
      //                           mesh data again.
      //
      status = directModifier( meshNode );
   }
   else
   {
      // Call the redo on the DG and DAG modifiers
      //
      if( !fHasHistory )
      {
         status = fDagModifier.doIt();
      }
      status = fDGModifier.doIt();
   }

   return status;
}
```

Implementing a polyModifierCmd command

With a general understanding of what polyModifierCmd is capable of, we are faced with the issue of how to implement a command based on it. The rules are fairly straightforward. polyModifierCmd structures its process in a similar manner as Maya treats its nodes. It is here where the concept of a factory is introduced. The following sections introduce the polyModifierFty and polyModifierNode in the context of implementing a command based on polyModifierCmd. For further details on how these work, please see the respective source file.

polyModifierFty

From the inner workings of polyModifierCmd, you see that there are two spots where we could potentially have redundant code: directModifier() and inside the modifier node. Effectively they do the same thing, except have a different means of retrieving their inputs. To reduce code duplication, the concept of a factory is introduced. The factory exists solely as a class structure which implements a modification to a mesh. It possesses a basic interface through which the modifier can be called.

To help guide you, a base factory class named polyModiferFty is provided from which a factory can be derived. Though it serves no functional purpose, it provides an outline through which you can implement your modification:

```
class polyModifierFty
{
public:
            polyModifierFty();
   virtual  ~polyModifierFty();

   // Pure virtual doIt()
   //
   virtual MStatus   doIt() = 0;
};
```

polyModifierNode

Similar to polyModifierFty, another guidance class is provided to give a framework for all modifier nodes to be used in association with polyModifierCmd. This class suggests things such as, the modifier node requires an inMesh and outMesh attribute to work:

```
class polyModifierNode
{
public:
            polyModifierNode();
   virtual  ~polyModifierNode();

   // There needs to be an MObject handle declared for
   // each attribute that the node will have. These handles
   // are needed for getting and setting the attribute
   // values later.
   //
   static MObject inMesh;
   static MObject outMesh;
};
```

For further details on implementing a polyModifierCmd command, refer to both the splitUVCmd example (next) and the source code. Inside the source code a general set of guidelines for each class is provided.

splitUVCmd example

(Source code is located in the devkit plugins directory.)

The splitUV command is a MEL command used to unshare or “split” the selected UVs of a polygonal mesh. A UV is a point on a 2D texture plane that is used to map textures onto a mesh. To properly associate these UVs with a mesh, each face can either be mapped or unmapped. If a face is mapped, each vertex belonging to that face is associated with a UV. This relationship is known as a face-vertex relationship. (For more information, see ”Face-Vertices” on page 149.)

For a better understanding of what splitting a UV means, try the following steps in Maya:

1. Create a square 3x3 face polygonal plane.
2. With the plane selected, open the UV Texture Editor (Windows > UV Texture Editor).
3. Change the selection mode to UVs.
4. Select a UV inside the plane (not along the border).
5. Select the Move tool.
6. Drag the UV around and notice that there is only a single UV.
7. Change the selection mode to Edges.
8. Select all four edges surrounding the UV you moved.
9. Select Edit Polygons > Texture menu > Map Cut.
10. Change the selection mode back to UVs.
11. Click once on the selected UV. (Make sure you do not drag over the UV or you will select them all.)
12. Drag the UV around and notice that it is no longer shared among the faces but is a single UV with its own face.
13. [Optional] Turn on texture borders in the Custom Polygon Display window (Display > Custom Polygon Display) to reveal the new borders introduced by the map cut operation.

The Map Cut command operates on edges of a mesh while splitUV operates on a UV.

UVs are stored in a single linear array, indexed by face vertices. Each face vertex indexes a specific UVId, and in turn each UVId represents a single UV point on a texture map. Thus the same number of faces that share the UV would index a shared UVId. (For more information, see ”UVs” on page 150.)

You need to add a number of UVs (at the same 2D coordinates) equal to the total number of faces that share the UV minus one. Subtract one knowing you already have at least one UV already associated to a shared face, which leaves one less face and UV to create and associate. Now associate each of the new UVs to one of the faces that shared the original UV, leaving one of the faces indexing the original UV itself. This leaves each face with an unshared UV at their respective face-vertex location.

Initial implementation

First gather your input—in this case the selected UVs. Obtain the selection list and filter it for the first object you find with selected UVs. For simplicity, only the first object encountered with selected UVs is taken. You can easily extend this to operate on multiple objects.

```
// Get the active selection list and filter for poly
objects.
// Also create a selection list iterator to parse the list
//
MSelectionList selList;
MGlobal::getActiveSelectionList( selList );
MItSelectionList selListIter( selList );
selListIter.setFilter( MFn::kMesh );

// Declare a dagPath and component to store a reference to
the
// mesh object and selected components
//
MDagPath dagPath;
MObject component;

// Now parse the selection list for poly objects with
selected
// UVs
//
for( ; !selListIter.isDone(); selListIter.next() )
{
   selListIter.getDagPath( dagPath, component );

   // Check for selected UVs
   //
   if( component.apiType() == MFn::kMeshMapComponent )
   {
      break;
   }
}

// Now we break down the component object into an int array
// of UVIds
```

```
//
MFnSingleIndexedComponent compFn( component );
MIntArray selUVs;
compFn.getElements( selUVs );
```

Now you have the object to operate on and the selected UVs. Before you can perform the operation, you need to collect some preprocessing data—which faces share each selected UV and the UV vertex associations. Finally, you must cover for the possible appearance of multiple UV sets. Since UVs from only one UV set can be selected at any particular time, you only need to access the current active UV set and operate on that set.

```
// Declare our processing variables
//
MObject mesh;
MString selUVSet;
MIntArray selUVFaceIdMap;
MIntArray selUVFaceOffsetMap;
MIntArray selUVLocalVertIdMap;

// Make sure our dagPath points to a shape node. That is
where
// the topological/geometry data is stored (not on the
transform)
//
dagPath.extendToShape();
mesh = dagPath.node();

// Initialize a mesh function set to our mesh
//
MFnMesh meshFn( mesh );

// Get the current UV set name
//
meshFn.getCurrentUVSetName( selUVSet);

// Now parse the mesh for face and vertex UV associations
//
MItMeshPolygon polyIter( mesh );
int i;
int j;
int offset = 0;
int selUVsCount = selUVs.length();

for( i = 0; i < selUVsCount; i++ )
{
   // Append the current offset in the faceId map to the
   // faceOffset map so we have an index reference into the
   // faceId map for each selected UV. In other words,
   // for each offset value, we have a number of faces equal
   // to (faceOffsetMap[i+1] - faceOffsetMap[i]) that share
   // the UV that that offset value represents.
   //
   selUVFaceOffsetMap.append( offset );

   // Parse the mesh for faces which share the current UV
   //
   for( ; !polyIter.isDone(); polyIter.next() )
   {
      // Only continue if the face is mapped
      //
      if( polyIter.hasUVs() )
      {
         // Now parse the vertices of each face and check
for
         // the current UV
         //
         int polyVertCount = polyIter.polygonVertexCount();

         for( j = 0; j < polyVertCount; j++ )
         {
            int UVIndex = 0;
            polyIter.getUVIndex( j, UVIndex );

            // If we find the UV on this face, append the
faceId,
            // append the local vertex (relative to the
current
            // face) and increment our offset.
            //
            if( UVIndex == selUVs[i] )
            {
               selUVFaceIdMap.append( polyIter.index() );
               selUVLocalVertIdMap.append(j);
               offset++;
               break;
            }
         }
      }
   }
}

// Finally append the last offset value so we can obtain the
// number of faces that share the last UV in the list.
//
selUVFaceOffsetMap.append( offset );
```

The face and face-vertex associations are stored using a similar technique used by Maya to store topological data for polygons. This technique is apparent in the code where the list of faces shared by each selected UV is accumulated. Rather than creating a multi dimensional array to store the list of faces shared by each selected UV, store it all in a single dimensional array (a *data* array). To do this you create another single dimensional array (an *index* array) to store the index values of the data array where each selected UV begins its list of faces.

For each UV, parse the mesh for any faces sharing that particular UV. This is accomplished by parsing the face-vertices of each face in the mesh, looking at the associated UVId and comparing the UVId with the current UV. Now store the face Id and the local vertex Id (relative to the current face, enumerating from 0 to (faceVertexCount – 1)). Make note of the local vertex Id rather than the global or mesh vertex Id because UVs are assigned on a face-vertex basis.

```
// Declare UV count variables so we can keep create and
// keep track of the indices of the new UVs
//
int currentUVCount = meshFn.numUVs( selUVSet );

// For each UV in our list of selected UVs, split the UV.
//
for( i = 0; i < selUVsCount; i++ )
{
   // Get the current faceId map offset
   //
   offset = selUVFaceOffsetMap[i];

   // Get the U and V values of the current UV
   //
   float u;
   float v;
   int UVId = selUVs[i];

   meshFn.getUV( uvId, u, v, &selUVSet );

   // Get the number of faces sharing the current UV
   //
   int faceCount = selUVFaceOffsetMap[i+1]-
selUVFaceOffsetMap[i];

   // Arbitrarily choose that the last faceId in the list
   // of faces sharing this UV will keep the original UV
   //
   for( j = 0; j < faceCount - 1; j++ )
   {
      // Create a new UV (setUV dynamically sizes the UV
array
      // if the index value passed in exceeds the length of
the
      // UV array) with the same 2D coordinates as our UV.
      //
      meshFn.setUV( currentUVCount, u, v, &selUVSet );

      // Get the face and face-vertex info so we can assign
      // our newly created UV to one of the faces in the
list
      // of faces sharing this UV
      //
      int localVertId = selUVLocalVertIdMap[offset];
      int faceId = selUVFaceIdMap[offset];

      // Associate the UV with the particular face vertex
      //
      meshFn.assignUV( faceId,
                       localVertId,
                       currentUVCount,
                       &selUVSet );

      // Increment our counters so we can create another new
UV
      // at the currentUVCount index. Increment the offset,
so we
      // can associate the next new UV with the next face in
the
      // the list of faces sharing this UV
      //
      currentUVCount++;
      offset++;
   }
}
```

There are two primary methods which are called to perform the actual split:

- MFnMesh::setUV(…)
- MFnMesh::assignUV(…)

Call the first method, setUV(), to create a new UVId. The method automatically grows the UV array to accommodate the index passed into the method. Thus in the code you can see a variable named currentUVCount which is continuously incremented after each new UV. currentUVCount keeps track of the index that is one element greater than the highest element in the UV array. Incrementing it after each iteration through the loop allows you to create a new UV, one at a time.

Call the second and last method, assignUV(), to associate a given UVId with a face and face-vertex.

Integrating into the Maya architecture

There are many intricacies involving modifying a polygonal mesh, including construction history and tweaks. If the mesh does not have hisory, you could attempt to unshare the UVs directly on the mesh itself. If the mesh has history, any DG evaluation from a node upstream in the construction history overwrites the mesh on the mesh node and the modifications made directly to the mesh would be lost. Even if that were the case, the existence of tweaks would change the appropriate place to write the modifications on the mesh.

You need to look at the mesh node, analyze its state, and apply your operation accordingly. The MPxCommand class polyModifierCmd (see polyModifierCmd example) was developed with the splitUV command to aid in abstracting the handling of construction history and tweaks. polyModifierCmd is a mid level command class designed for commands which intend to modify a polygonal mesh. It provides an outlet through which a poly command can take its code to modify a mesh directly and seamlessly integrate it into the Maya framework, accounting for both construction history and tweaks.

polyModifierCmd is a good example of using the API and demonstrates how construction history and tweaks work.

polyModifierCmd enhanced splitUV

Before you proceed with this section, read the ”polyModifierCmd example” on page 164. This section steps through the implementation of a command based on polyModifierCmd.

There are three main pieces of the polyModifierCmd that must be handled:

- A splitUV command
- A splitUV node
- A splitUV factory

splitUV factory

The factory is the lowest level of the splitUVCmd, which performs the operation given a set of inputs. The inputs are an integer array of UV Ids and a reference to the mesh you are about to modify. The rest fits inside the factory. The splitUVFty factory class interface is shown below.

```
class splitUVFty : polyModifierFty
{
```

```
public:
            splitUVFty();
   virtual  ~splitUVFty();

   void     setMesh( MObject mesh );
   void     setUVIds( MIntArray uvIds );

   // polyModifierFty inherited methods
   //
   MStatus  doIt();

private:
   // Mesh Node
   //
   MObject fMesh;

   // Selected UVs
   //
   MIntArray fSelUVs;
};
```

splitUV node

There are two methods of deploying the factory. One is through the splitUV node and the other is directly through the command for certain exception cases where the node is not applicable. The splitUV node is used for cases where you want to build or add to an existing history chain in the DG.

When a DG evaluation is propagated via a dirtied node, the DG evaluates from the top of the history chain where a copy of the original mesh (original referring to the node's state denoting the start of this history) is located. It then takes a copy of that mesh and passes it in through each node in order, where the mesh is altered through each node evaluation. Once it reaches the final shape, you have a mesh placed onto the shape which holds all the modifications stored in the history chain. The splitUV node needs to take in a mesh input as well as an input of which UVs to modify, and pass that data down to an instance of the factory. The resulting mesh is then passed out through a mesh output attribute.

```
class splitUVNode : polyModifierNode
{
public:
                   splitUVNode();
   virtual         ~splitUVNode();

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

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

private:
   // Note: There needs to be an MObject handle for each
   // attribute on the node. These handles are needed for
   // setting and getting values later. The standard inMesh
   // and outMesh attributes are already inherited from
   // polyModifierNode, thus we only need to declare
splitUVNode
   // specific attributes
   //
   static MObject  uvList;

   // Node type identifier
   //
   static MTypeId  id;

   // We instantiate a copy of our splitUV factory on the
node
   // for it to perform the splitUV operation
   //
   splitUVFty      fSplitUVFactory
};
```

The standard node interface is in the above class declaration. The only differences to note reside in the private members. From the class hierarchy, splitUVNode inherits an inMesh and outMesh attribute from polyModifierNode. We add yet another attribute to the node, specific to the splitUVNode, which consists of our only other input—the list of UVs to operate on.

Notice that the node class has an instance of a splitUV factory. You create a distinct factory for each node so that the splitUVFty implementation would require no foreknowledge of which node is calling the operation. Continuing with the basic node setup, implement the basic methods in the above interface, creating and associating attributes, assigning a type id, etc.:

```
MTypeId splitUVNode::id( 0x34567 );

MStatus splitUVNode::creator()
{
   return new splitUVNode();
}

MStatus splitUVNode::initialize()
{
   MStatus status;
   MFnTypedAttribute attrFn;

   uvList = attrFn.create("inputComponents",
                          "ics",

MFnComponentListData::kComponentList);
   // To be stored during file-save
   attrFn.setStorable(true);

   inMesh = attrFn.create("inMesh",
                          "im",
                          MFnMeshData::kMesh);
   // To be stored during file-save
   attrFn.setStorable(true);

   // outMesh is read-only because it is an output attribute
   //
   outMesh = attrFn.create("outMesh",
                             "om",
                             MFnMeshData::kMesh);
   attrFn.setStorable(false);
   attrFn.setWritable(false);

   // Add the attributes we have created for the node
   //
   status = addAttribute( uvList );
   if( !status )
   {
      status.perror("addAttribute");
      return status;
   }
   status = addAttribute( inMesh );
   if( !status )
   {
      status.perror("addAttribute");
      return status;
   }
   status = addAttribute( outMesh );
   if( !status )
   {
      status.perror("addAttribute");
      return status;
   }

   // Set up a dependency between the inputs and the output.
   // This will cause the output to be marked dirty when the
   // input changes. The output will then be recomputed the
   // next time it is requested.
   //
   status = attributeAffects( inMesh, outMesh );
   if( !status )
   {
      status.perror("attributeAffects");
      return status;
   }
   status = attributeAffects( uvList, outMesh );
   if( !status )
   {
      status.perror("attributeAffects");
      return status;
   }

   return MS::kSuccess;
}
```

Finally, we turn towards the implementation of our compute() method. The compute method is not overly complex. Since we have the factory to perform the operation, all the compute method needs to do is provide the factory with the references to the proper mesh to modify.

Start as all plug-in nodes should and look to handle the 'state' attribute, inherited from MPxNode, on the node. The 'state' attribute holds a short integer representing how the DG treats the node. In a sense it is an override mechanism to alter how the node is treated during a DG evaluation. With respect to plug-in nodes, the only state of concern is the 'HasNoEffect' or 'PassThrough' state, where the node is ignored entirely. For the node to behave as though it were transparent, you need to redirect the inMesh to the outMesh without altering the mesh passing through. Otherwise the node behaves normally.

Following the node state check, grab the UVs from the component list and the input mesh, assign the input mesh to the output mesh, and pass in these references to the factory. Assigning the input mesh to the output mesh allows you to operate directly on the output mesh, so that the output mesh will hold the modified mesh. From there, let the factory take care of the rest of the operation.

```
MStatus splitUVNode::compute(const MPlug& plug, MDataBlock&
data)
{
   MStatus status = MS::kSuccess;

   // Retrieve our state attribute value
   //
   MDataHandle stateData = data.outputValue(state,&status);
   MCheckStatus( status, "ERROR getting state" );

   // Check for the HasNoEffect/PassThrough state
   //
   // (stateData is stored as a short)
   //
   // (0 = Normal)
   // (1 = HasNoEffect/PassThrough)
   // (2 = Blocking)
   // ..
   //
   if( stateData.asShort() == 1 )
   {
      MDataHandle inputData =
data.inputValue(inMesh,&status);
      MCheckStatus(status, "ERROR getting inMesh");
      MDataHandle outputData =
data.outputValue(outMesh,&status);
      MCheckStatus(status, "ERROR getting outMesh");

      // Simply redirect the inMesh to the outMesh
      //
      outputData.set(inputData.asMesh());
   }
   else
   {
      // Check which output value we have been asked to
compute.
      // If the node doesn't know how to compute it, return
      // MS::kUnknownParameter.
      //
      if( plug == outMesh )
      {
         MDataHandle inputData = data.inputValue( inMesh,
                                                  &status
);
         MCheckStatus(status, "ERROR getting inMesh");

         MDataHandle outputData = data.outputValue( outMesh,

&status );
         MCheckStatus(status, "ERROR getting outMesh");

         // Now, we retrieve the input UV list
         //
         MDataHandle inputUVs = data.inputValue( uvList,
                                                 &status
);
         MCheckStatus(status, "ERROR getting uvList");

         // Copy the inMesh to the outMesh so we can operate
         // directly on the outMesh
         //
         outputData.set(inputData.asMesh());
```

```
            MObject mesh = outputData.asMesh();

            // Retrieve the UV list from the component list
            //
            // Note, we use a component list to store the 
components
            // because it is more compact memory wise. (ie.
            // comp[81:85] is smaller than 
comp[81],...,comp[85])
            //
            MObject compList = inputUVs.data();
            MFnComponentListData compListFn( compList );

            unsigned i;
            int j;
            MIntArray uvIds;

            for( i = 0; i < compListFn.length(); i++ )
            {
               MObject comp = compListFn[i];
               if( comp.apiType() == MFn::kMeshMapComponent )
               {
                  MFnSingleIndexedComponent uvComp(comp);
                  for( j = 0; j < uvComp.elementCount(); j++ )
                  {
                     int uvId = uvComp.element(j);
                     uvIds.append(uvId);
                  }
               }
            }

            // Set the mesh object and uvList on the factory
            //
            fSplitUVFactory.setMesh( mesh );
            fSplitUVFactory.setUVIds( uvIds );

            // Now, call the factory to perform the splitUV
            //
            status = fSplitUVFactory.doIt();

            // Mark the outputMesh as clean
            //
            outputData.setClean();
         }
         else
         {
            status = MS::kUnknownParameter;
         }
      }
```

```
   return status;
}
```

splitUV command

Now that we have a splitUVNode, the last thing left on the list to do is the splitUV command. This is the piece that ties everything together. It is from here that the method for modifying the mesh is chosen (although implicitly through polyModifierCmd). The command manages the operation and is the highest level of interfacing with the user.

As a child class of polyModifierCmd, the splitUVCmd does not have much work to do other than override some specific polyModifierCmd methods and retrieve the input.

```
class splitUV : polyModifierCmd
{
public:
                  splitUV();
   virtual        ~splitUV();

   static void*   creator();

   bool           isUndoable();

   // MPxCommand inherited methods
   //
   MStatus        doIt( const MArgList& );
   MStatus        redoIt();
   MStatus        undoIt();

   // polyModifierCmd inherited methods
   //
   MStatus        initModifierNode( MObject modifierNode );
   MStatus        directModifier( MObject mesh );

private:
   // Private methods
   //
   bool           validateUVs();
   MStatus        pruneUVs();

   // Private members
   //

   // Selected UVs
   //
   // We store two copies of the UVs, one that is passed
down to
```

```
   // the node and another kept locally for the
directModifier.
   // Note, the MObject member, fComponentList, is only ever
   // accessed during a single call of a plugin, never
between
   // calls where its validity is not guaranteed.
   //
   MObject        fComponentList;
   MIntArray      fSelUVs;

   // splitUV Factory
   //
   // This factory is for the directModifier to have access
to
   // operate directly on the mesh.
   //
   splitUVFty     fSplitUVFactory;
};
```

This looks much like the standard MPxCommand class interface. However there are a few differences due to the polyModifierCmd inheritance as well as some performance enhancing methods. Two methods need to be overridden:

- initModifierNode()
- directModifier()

initModifierNode() is the chance for a command to initialize any inputs aside from the inMesh on the modifier node, which in our case is the splitUVNode. This is not restricted to input initialization, but can be catered towards custom node initialization if desired. This method is called before the modifier node is placed in the history chain, if the creation of history is permissible. For example, in our case we'd like to initialize the uvList input on our splitUVNode:

```
MStatus splitUV::initModifierNode( MObject modifierNode )
{
   MStatus status;

   // Tell the splitUVNode which UVs to operate on.
   //
   MFnDependencyNode depNodeFn( modifierNode );
   MObject uvListAttr;
   uvListAttr = depNodeFn.attribute( "inputComponents" );

   // Pass the component list down to the node.
   // To do so, we create/retrieve the current plug
   // from the uvList attribute on the node and simply
   // set the value to our component list.
   //
   MPlug uvListPlug( modifierNode, uvListAttr );
   status = uvListPlug.setValue( fComponentList );

   return status;
}
```

directModifier() is a method called only in a specific exception case where the mesh node has no history and the preference to record history is turned off. The consequence of this state is that the user does not wish to have any history chain at all. So in effect, the polyModifierCmd is forbidden to use the DG. As a result we modify the mesh directly. The polyModifierCmd description discusses the implications of this state in more detail as well as alternative approaches. However all we need to know is that we need to provide a method to operate on the mesh directly, which if you recall, we completed in the concepts section. It is for this reason that the command also holds an instance of the factory as well as a copy of the UVs to modify in an integer array format (as opposed to a component list for the splitUVNode).

You might wonder why we store two copies of the selected UVs in different formats. The reason for this is that an MObject is never guaranteed to be valid between plug-in calls (including redoIt() calls). Since the directModifier() would be called in a redoIt() case, it would rely on the validity of the MObject component list between calls. As such we’ve stored two copies on the command. Alternatively, one could choose to modify the node so that it receives an integer array as a node input rather than a component list to streamline the operation, however it’s a balancing issue of performance vs. storage.

Using these inputs we have the following simple directModifier() method:

```
MStatus splitUV::directModifier( MObject mesh )
{
   MStatus status;

   fSplitUVFactory.setMesh( mesh );
   fSplitUVFactory.setUVIds( fSelUVs );

   // Now, call the factory to perform the splitUV
   //
   status = fSplitUVFactory.doIt();

   return status;
}
```

Before we look at the performance enhancing methods, let's take a peek at the MPxCommand inherited methods. These methods will give us a better appreciation of how the performance of the command can be slightly tweaked:

- doIt()
- undoIt()
- redoIt()

The doIt() method is our main method which retrieves the input and caters the rest of the operation to the various parts, overseeing the entire operation. The doIt() method is the method used to initialize the command and perform the operation as the name would suggest. And much to that effect, the splitUV's doIt() method does exactly that.

We begin by parsing the selection list for any objects where UVs are selected, just the same as we did in our original implementation. Following that we initialize the polyModifierCmd settings, call our performance enhancing methods and issue the doModifyPoly() method, which can be viewed as the polyModifierCmd's version of doIt(). Additionally we scatter the appropriate error messages in the code to inform the user of improper use of the command.

```
MStatus splitUV::doIt( const MArgList& )
{
   MStatus status;

   MSelectionList selList;
   MGlobal::getActiveSelectionList( selList );
   MItSelectionList selListIter( selList );
   selListIter.setFilter( MFn::kMesh );

   // Initialize our component list
   //
   MFnComponentListData compListFn;
   compListFn.create();

   // Parse the selection list
   //
   bool found = false;
   bool foundMultiple = false;
   for( ; !selListIter.isDone(); selListIter.next() )
   {
      MDagPath dagPath;
      MObject component;
      selListIter.getDagPath( dagPath, component );

      if( component.apiType() == MFn::kMeshMapComponent )
      {
         if( !found )
         {
            // Add the components to our component list.
            // ‘component’ holds all selected components
            // on the given object, so only a single call
            // to add is needed.
            //
            compListFn.add( component );
            fComponentList = compListFn.object();

            // Locally store the selected UVIds in the
command
            // int array member, fSelUVs
            //
            MFnSingleIndexedComponent compFn( component );
            compFn.getElements( fSelUVs );

            // Ensure that our DAG path is pointing to a
            // shape node. Set the DAG path for
polyModifierCmd.
            //
            dagPath.extendToShape();
            setMeshNode( dagPath );
            found = true;
         }
         else
         {
            // Since we are only looking for whether or not
there
            // are multiple objects with selected UVs, break
out
            // once we have found one other object.
            //
            foundMultiple = true;
            break;
         }
      }
   }

   if( foundMultiple )
   {
      displayWarning( “Only operates on first found mesh” );
   }

   // Set the modifier node type for polyModifierCmd
   //
   setModifierNodeType( splitUVNode::id );

   if( found )
   {
      if( validateUVs() )
      {
         // Now, pass control over to polyModifierCmd
         //
         status = doModifyPoly();
         if( status == MS::kSuccess )
         {
            setResult( "splitUV command succeeded!" );
         }
         else
         {
            setResult( "splitUV command failed!" );
         }
      }
      else
      {
         displayError( "Selected UVs are not splittable" );
         status = MS::kFailure;
      }
   }
   else
   {
      displayError( "Unable to find selected UVs" );
      status = MS::kFailure;
   }

   return status;
}
```

The undo/redo mechanism is supported by the undoIt() and redoIt() methods inherited from MPxCommand. These methods often use cached data from the first doIt() to execute. This is what splitUV does as well as polyModifierCmd, which supports its own undo/redo in the form of undoModifyPoly() and redoModifyPoly(). Since splitUV relies on polyModifierCmd to perform the operation, the undo/redo redirects the undo/redo to polyModifierCmd. As a result, the splitUV's undoIt() and redoIt() methods are very straightforward:

```
MStatus splitUV::redoIt()
{
   MStatus status;

   status = redoModifyPoly();
   if( status == MS::kSuccess )
   {
      setResult( "splitUV command succeeded!" );
   }
   else
   {
      setResult( “splitUV command failed!” );
   }

   return status;
}

MStatus splitUV::undoIt()
{
   MStatus status;

   status = undoModifyPoly();
   if( status == MS::kSuccess )
   {
      setResult( “splitUV undo succeeded!” );
   }
   else
   {
      setResult( “splitUV undo failed!” );
   }

   return status;
}
```

The call to validateUVs() in the doIt() method is a performance enhancing method. Though this method is primarily a pre-condition check on the selected UVs, it increases the optimal performance of the command by pruning the selected UV list of UVs that cannot be split. This potentially saves the operation unnecessary loops. However, an extra pass of the mesh is required to check for UVs that cannot be split, but only on the very first call to the command. Any successive redoIt() calls or node evaluations are faster.

To define when a UV is invalid and unable to be split, look at the definition of the operation. splitUV provides each face which shares a particular UV with it’s own unique and unshared UV. Thus a UV can only be split if it is shared by more than one face. Subsequently, the validateUVs method parses the mesh and retrieves the face sharing information for each UV, marking valid UVs. The valid UV list is sent to the pruneUVs() method which replaces the component list and locally stored integer array of selected UVs.

```
bool splitUV::validateUVs()
{
   // Get the mesh that we are operating on
   //
   MDagPath dagPath = getMeshNode();
   MObject mesh = dagPath.node();
   // Get the number of faces sharing each UV
   //
   MFnMesh meshFn( mesh );
   MItMeshPolygon polyIter( mesh );
   MIntArray selUVFaceCountArray;

   int i;
   int j;
   int count = 0;
   selUVsCount = fSelUVs.length();

   for( i = 0; i < selUVsCount; i++ )
   {
      for( ; !polyIter.isDone(); polyIter.next() )
      {
         if( polyIter.hasUVs() )
         {
            int UVIndex = 0;
            polyIter.getUVIndex( j, UVIndex );

            // If we have a matching UVId, then we have a
            // face which shares this UV, so increment the
            // count.
            //
            if( UVIndex == fSelUVs[i] )
            {
               count++;
               break;
            }
         }
      }
      selUVFaceCountArray.append( count );
   }

   // Now, check to make sure that at least one UV has more
than
   // one face sharing it. So long as we have at least one
valid
   // UV, we should proceed with the operation by returning
true
   //
   bool isValid = false;
   MIntArray validUVIndices;

   for( i = 0; i < selUVsCount; i++ )
   {
      if( selUVFaceCountArray > 1 )
      {
         isValid = true;
         validUVIndices.append(i);
      }
   }

   if( isValid )
   {
      pruneUVs( validUVIndices );
   }

   return isValid;
}

MStatus splitUV::pruneUVs( MIntArray& validUVIndices )
{
   MStatus status = MS::kSuccess;

   unsigned i;
   MIntArray validUVIds;

   for( i = 0; i < validUVIndices.length(); i++ )
   {
      int uvIndex = validUVIndices[i];
      validUVIds.append( fSelUVs[uvIndex] );
   }

   // Replace our local int array of UVIds
   //
   fSelUVs.clear();
   fSelUVs = validUVIds;

   // Build our list of valid components
   //
   MFnSingleIndexedComponent compFn;
   compFn.create( MFn::kMeshMapComponent );
   compFn.addElements( validUVIds );
   MObject component = compFn.object();

   // Replace the component list
   //
   MFnComponentListData compListFn;
   compListFn.create();
   compListFn.add( component );
   fComponentList = compListFn.object();

   return status;
}
```

For further details on the implementation of the splitUV command, look at the source code provided in the plug-ins directory of the developer's kit.

Poly exporter plug-ins

These are two new example exporter plugins that demonstrate how to extract polygonal data from Maya using the Poly API. The two exporters write out data in raw text and Extensible 3D (X3D) formats respectively. The functionality is split into two components, one that extracts the data from Maya and stores it in intermediate data structures, and the second that writes the data to a file in the required format. These two components are implemented as pure virtual base classes from which the user derives to implement export to a specific format. More detailed comments are in the source code.

Classes

- polyExporter – base class for code that extracts poly information from Maya.
- polyWriter – base class for code that writes data to a disk file.
- polyRawExporter, polyRawWriter – Derived classes implementing raw text output.
- polyX3DExporter, polyRawWriter – Derived classes implementing X3D output.

polyX3DExporter

Adds the ability to export polygonal meshes from a Maya scene to the Extensible 3D (X3D) file format. Once this plug-in is loaded, the new file format is listed as an output format for export.

Polygonal meshes are exported by using the File > Export All menu item (or by selected specific meshes and using the File > Export Selection option), choosing X3D as the file type, and providing a filename. The resulting file will be in X3D compliant format.

This plugin example demonstrates how to utilize the Maya Poly API for extracting polygonal geometry data, in conjunction with the Maya MPxFileTranslator class to create a file exporter plug-in. Currently, data that is extracted includes:

- faces and their vertex components
- vertex coordinates
- colors per vertex
- normals per vertex
- current uv set and coordinates (X3D did not support multiple UV sets when this plugin was written.)
- component sets
- file textures (for the current UV set)

polyRawExporter

This exporter is the same as polyX3DExporter except that the output data is in raw text format rather than X3D. Also this plugin exports all UV sets and coordinates.

10 Setting up your plug-in build environment

Developer

Plug-in API

Setting up a build area

IRIX and Linux environments

Maya plug-ins

The Maya Development Kit product contains a number of example plug-ins located in `/usr/aw/maya/devkit/plug-ins`.

Before you can use these plug-ins, you need to build them. You first have to create a working directory somewhere, recursively copy the directory and run make. For example,

```
mkdir -p $HOME/devkit/
cd $HOME/devkit/
cp -r /usr/aw/maya/devkit/plug-ins .
make Clean
make
```

Also, to attach your plug-in development area to the rest of Maya, you need to set a number of variables. These are:

- `MAYA_LOCATION`
- `MAYA_SCRIPT_PATH`
- `MAYA_PLUG_IN_PATH`
- `XBMLANGPATH`

These variables can be defined in a file called Maya.env. Maya lets you define these variables in a file so that you can easily set up the same runtime environment on another system by simply copying the file. You can still use variables in the environment and they will either override the corresponding variable in the Maya.env file or be prepended to the variable for variables which represent search paths.

The environment variable, `MAYA_APP_DIR`, can be used to help find the `Maya.env` file. If this variable is not set, Maya looks in your `$HOME/maya` directory. In addition, if you have multiple versions of Maya installed on your system, you can put your `Maya.env` file in a subdirectory of either the directory pointed to by the `MAYA_APP_DIR` environment variable or `$HOME/maya`.

The subdirectory must be named to be the version number of the Maya application that will be executed. For example, if you have set `MAYA_APP_DIR` to be `/usr/mydir`, you can create a version specific Maya.env file in the directory `/usr/mydir/6.0`. that will be used when the 6.0 version of Maya is run. If you do not set `MAYA_APP_DIR`, you can put your version 6.0 tailored `Maya.env` file in `$HOME/maya/6.0`.

The following assumes that Maya is installed in `/usr/aw/maya` and that you have set up your plug-in development area in `$HOME/devkit/plug-ins`. If your installation is different, you will have to modify the lines that set `MAYA_LOCATION` in the examples below.

Your Maya.env file should contain the following:

```
MAYA_SCRIPT_PATH     = $HOME/devkit/plug-ins
MAYA_PLUG_IN_PATH    = $HOME/devkit/plug-ins
XBMLANGPATH          = $HOME/devkit/plug-ins/%B
```

Users of `/bin/sh` or `/bin/ksh` need to add the following lines to `$HOME/.profile`.

```
# Location of installed maya.
MAYA_LOCATION=/usr/aw/maya
export MAYA_LOCATION
```

Users of `/bin/csh` or `/bin/tcsh` need to add the following lines to `$HOME/.cshrc`.

```
# Location of installed maya
setenv MAYA_LOCATION /usr/aw/maya
```

If you now start Maya and open the Plug-in Manager window, you should see a list of all the pre-compiled plug-ins you copied to your `$HOME/devkit/plug-ins` directory.

Maya API applications

To build the supplied stand-alone application examples, you need to do the following:

```
mkdir $HOME/devkit/applications
cd $HOME/devkit/applications
cp /usr/aw/maya/devkit/applications/* .
make Clean
make
```

Important note

The shell script mayald is used to link these applications to isolate you from the exact set of Maya shared libraries necessary for the link.

On IRIX and Linux, you must set the LD_LIBRARYN32_PATH environment variable before you try to execute one of these applications so the runtime linker can find the Maya shared libraries.

On Linux, the variable you must set is LD_LIBRARY_PATH.

The recommended procedure to prepare for building and running stand-alone apps is to set the following environment variables:

IRIX:

```
setenv MAYA_LOCATION /usr/aw/maya
setenv LD_LIBRARYN32_PATH $MAYA_LOCATION/lib
```

Linux:

```
setenv MAYA_LOCATION /usr/aw/maya
setenv LD_LIBRARY_PATH $MAYA_LOCATION/lib
```

When linking your plug-in, make sure to list all of the OpenMaya libraries containing the API classes you have used. The reference pages for each class specify the particular OpenMaya library containing the class.

Note Stand-alone apps can't read files with IK. Specifically, if you try to read a file with IK using the devkit “readAndWrite” app, you get an error about a failed connection and the ik fails in the scene. To counter this, you must force the IK subsystem to load before the file is loaded. After the MLibrary() call is made, add in some command that uses the IK subsystem before the call to MFileIO::open(). The following command works:

```
MGlobal::executeCommand( "ikSystemInfo -q qsh" );
```

Linux compiler requirement

To compile plug-ins and standalone plug-ins for Maya 6 for Linux, use the gcc compiler 3.3.2. Maya is built with this compiler under RedHat 7.3. Plug-ins built with any other compiler will not work because the C++ ABI (Application Binary Interface) must match between Maya and plug-ins.

Note that the compiler should be renamed with the “332” extension to avoid conflict with the default version of gcc on the system.

Please consult the section “Additional Linux Notes” in the Installing Linux chapter of the *Installation and Licensing* guide for information on how to build the gcc 3.3.2 compiler.

Using a debugger to debug your plug-ins

To start Maya under the control of a debugger, you use the “-d” flag of the `maya` shell script. The syntax for this is:

```
maya -d debuggerName
```

or, for example:

```
maya -d cvd
```

This launches the debugger given as an argument to the -d, and then starts Maya under control of this debugger. Once you have started Maya and loaded your plug-in, you can:

- set breakpoints in the plug-in code
- single step through the plug-in
- perform any of the operations supported by the debugger

By default, Maya catches several signals generated by programming faults, in particular: SIGSEGV, SIGILL, SIGBUS and SIGABRT. When debugging a plug-in with a debugger, it is likely that you will want to suppress this behavior of Maya, and instead let the debugger catch the signals. This can be accomplished by setting an environment variable, as shown below:

```
setenv MAYA_DEBUG_NO_SIGNAL_HANDLERS 1
```

This variable can be set either in the environment or the Maya.env file. If you use cvd as your debugger, beware of a potential conflict. The Maya shell script sets the values of two environment variables that tell the Maya binary where to find things. These variables are:

- MAYA_LOCATION

IRIX:

- LD_LIBRARYN32_PATH

Linux:

- LD_LIBRARY_PATH

If you have set either of these in your shell’s start-up file (.cshrc or.profile), you must protect them so shells started by the debugger (after the Maya shell script has started that debugger) do not *undo* these modifications.

IRIX:

If you use *csh* or *tcsh* use the following construct:

```
if ( ! ${?MAYA_LOCATION} )
    setenv MAYA_LOCATION /usr/aw/maya
if ( ! ${?LD_LIBRARYN32_PATH} )
    setenv LD_LIBRARYN32_PATH whatEver
```

and if you use *sh* or *ksh* use this construct:

```
MAYA_LOCATION=${MAYA_LOCATION:=/usr/aw/maya}
export MAYA_LOCATION
LD_LIBRARYN32_PATH=${LD_LIBRARYN32_PATH:=whatEver}
export LD_LIBRARYN32_PATH
```

Linux:

If you use *csh* or *tcsh* use the following construct:

```
if ( ! ${?MAYA_LOCATION} )
    setenv MAYA_LOCATION /usr/aw/maya
if ( ! ${?LD_LIBRARY_PATH} )
    setenv LD_LIBRARY_PATH whatEver
```

and if you use *sh* or *ksh* use this construct:

```
MAYA_LOCATION=${MAYA_LOCATION:=/usr/aw/maya}
export MAYA_LOCATION
LD_LIBRARY_PATH=${LD_LIBRARY_PATH:=whatEver}
```

export LD_LIBRARY_PATH

Windows environment

The Maya Development Kit product contains a number of example plug-ins located in `C:\Program Files\Alias\Maya6.0\devkit\plug-ins`. The development kit also contains several Maya API applications, located in `C:\Program Files\Alias\Maya6.0\devkit\applications`.

Maya plug-ins

Before you can use the example plug-ins, you need to build them. You can choose to build the plug-ins in the directory to which they were installed or you can copy the plug-ins to your own working directory.

To build an individual plug-in, you need to load the corresponding solution file (the .sln file) into Microsoft Visual Studio .NET 2003 Visual C++.

The easiest way to do this is to open Visual C++ and drag and drop the .sln file onto it. When the workspace is loaded, you can select *Build Solution* from the Build menu. Visual C++ will build your plug-in and place the resulting .mll file in the *current directory*.

To build all of the example plug-ins, you need to load the Plugins.sln workspace file into Visual C++. As above, the easiest way to do this is to open Visual C++ and drag and drop the Plugins.sln file onto it. When the workspace is loaded, you can select *Rebuild Solution* from the Build menu.

To load your plug-in into Maya, open the Plug-in Manager window and browse to the directory containing your plug-in. If you want the Plug-in Manager to automatically find your directory, you can build and put the plug-in into a directory defined by the MAYA_PLUG_IN_PATH variable.

Maya API Applications

You can choose to build the applications in the directory to which they were installed or you can copy the applications to your own working directory.

To build an individual application, you need to load the corresponding workspace file (the .sln file) into Microsoft Visual C++. The easiest way to do this is to open Visual C++ and drag and drop the .sln file onto it.

When the solution is loaded, you can select *Build Solution* from the Build menu. Visual C++ will build your application and place the resulting executable file in the `C:\Program Files\Alias\Maya6.0\devkit\applications` directory.

To build all of the example Maya API applications, you need to load the AllApplications.sln workspace file into Visual C++. As above, the easiest way to do this is to open Visual C++ and drag and drop the AllApplications.sln file onto it. When the workspace is loaded, you can select *Rebuild Solution* from the Build menu.

If, during installation, you added the Maya executable directory to your path, you can run the application immediately. If you did not, you will need to copy your application to the Maya executable directory to run it.

Creating your own plug-in build file

The instructions in the previous section enable you to build and use the example plug-ins included with Maya, but you still need information on creating your own plug-ins. On Windows, the process for creating the source code files is the same as it is on IRIX and Linux, but in addition you must create Microsoft “Project” files so that Developer Studio knows how to build the plug-in. You can do this using the plug-in wizard, described in the following sections.

Using the Maya Plug-in Wizard for Developer Studio

Maya contains a “Maya Plug-in Wizard” for Microsoft Visual Studio .NET 2003 Visual C++ that makes it very easy to create project files for your plug-in. It is highly recommended that you use this wizard.

To install this wizard, please follow the instructions contained in the `$MAYA_LOCATION/devkit/pluginwizard` directory.

To use the wizard, select File > New Project in NET 20003, then select Visual C++ Projects from Project Types and then finally select "Maya Plug-in Wizard" from the Templates area. The wizard will prompt you for the name of the plug-in, the type of plug-in (e.g. Command, Node, Tool, etc.) and the list of libraries the plug-in requires for linking. When you have answered all the questions, the wizard will create a complete project that contains the needed .sln and .vcproj files, and a complete template of the code needed to create your Command, Node, Tool, etc. This plug-in will compile without any changes, and will be a "do nothing" version of the type of plug-in you specified to the wizard. You just need to edit the .h and .cpp files and add the logic for your plug-in.

Mac OSX environment

The Maya Development Kit is located in the `/Applications/Alias/maya6.0/devkit/plug-ins` directory in a standard install of Maya.

Maya Plug-ins

Building with Makefiles

There are two ways of building plug-ins on Mac OS X. Your choices will be based on that level of Mac OS X that you are developing on. If you are on Mac OS X 10.2.8, we only provide a Makefile based build solution with our release. If you are on Mac OS X 10.3 then you can either use the Makefile or Xcode project files we supply to build plug-ins.

The Maya Development Kit product contains a number of example plug-ins located in `/Applications/Alias/maya6.0/devkit/plug-ins`.

Before you can use these plug-ins, you need to build them. You first have to create a working directory somewhere, recursively copy the directory and run make. For example,

```
mkdir -p $HOME/devkit
cd $HOME/devkit
cp -r /Applications/Alias/maya6.0/devkit/plug-ins .
cd plug-ins
make Clean
make
```

Also, to attach your plug-in development area to the rest of Maya, you need to set a number of variables. These are:

- `MAYA_LOCATION`
- `MAYA_SCRIPT_PATH`
- `MAYA_PLUG_IN_PATH`
- `XBMLANGPATH`

These variables can be defined in a file called `Maya.env`. Maya lets you define these variables in a file so that you can easily set up the same runtime environment on another system by simply copying the file. You can still use variables in the environment and they will either override the corresponding variable in the Maya.env file or be prepended to the variable for variables which represent search paths.

The environment variable, `MAYA_APP_DIR`, can be used to help find the `Maya.env` file. If this variable is not set, Maya looks in your `$HOME/maya` directory. In addition, if you have multiple versions of Maya installed on your system, you can put your `Maya.env` file in a subdirectory of either the directory pointed to by the `MAYA_APP_DIR` environment variable or `$HOME/maya`.

The subdirectory must be named to be the version number of the Maya application that will be executed. For example, if you have set `MAYA_APP_DIR` to be `/usr/mydir`, you can create a version specific `Maya.env` file in the directory `/usr/mydir/6.0.` that will be used when the 6.0 version of Maya is run. If you do not set `MAYA_APP_DIR`, you can put your version 6.0 tailored `Maya.env` file in `$HOME/maya/6.0`.

The following assumes that Maya is installed in `/Applications/Alias/maya6.0/devkit/plug-ins` and that you have set up your plug-in development area in `$HOME/devkit/plug-ins`. If your installation is different, you will have to modify the lines that set `MAYA_LOCATION` in the examples below.

Your Maya.env file should contain the following:

```
MAYA_SCRIPT_PATH = $HOME/devkit/plug-ins
MAYA_PLUG_IN_PATH = $HOME/devkit/plug-ins
XBMLANGPATH = $HOME/devkit/plug-ins/%B
```

Either set the following on the command line or add the equivalent for the shell environment you are using. The following can be used placed into your .tcshrc if this is your default shell.

```
# Location of installed maya
setenv MAYA_LOCATION /Applications/Alias/maya6.0/Maya.app/
Contents
```

If you now start Maya and open the Plug-in Manager window, you should see a list of all the pre-compiled plug-ins you copied to your

```
$HOME/devkit/plug-ins directory.
```

Building with Xcode project files

To build one of our example plug-ins using the supplied Xcode project file you must be using Mac OS X 10.3 and have the Xcode application installed. Do the following:

1. Browse to the `/Applications/Alias/maya6.0/devkit/plug-ins` directory.
2. Double-click on an Xcode project file such as `cirlceNode.xcode`.
3. Select the Build option from the Build menu.

It may take a few minutes to build but once complete you will have a plug-in created in the same directory. The plug-in will have a .lib extension. This plug-in can now be loaded into Maya via the Plug-in Manager.

Maya API applications

For building API applications, we only provide a Makefile solution. To build the supplied stand-alone application examples, you need to do the following:

```
mkdir $HOME/devkit/applications
cd $HOME/devkit/applications
cp MAYA_LOCATION/Applications/Alias/maya6.0/devkit/
applications/* .
make Clean
make
```

Note The shell script mayald is used to link these applications to isolate you from the exact set of Maya shared libraries necessary for the link.

On Mac OS X, you must set the DYLD_LIBRARY_PATH environment variable before you try to execute one of these applications so the runtime linker can find the Maya shared libraries.

The recommended procedure to prepare for building and running stand-alone apps is to set the following environment variables:

```
setenv MAYA_LOCATION /Applications/Alias/maya6.0/Maya.app/
Contents
setenv DYLD_LIBRARY_PATH $MAYA_LOCATION/MacOS
```

11 Appendices

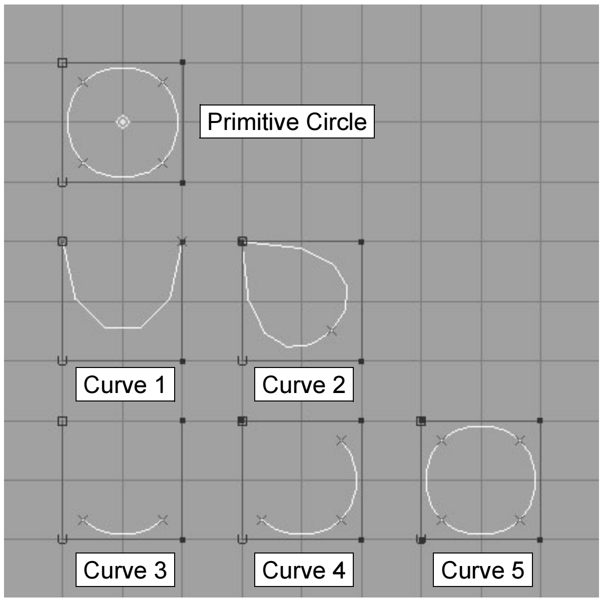
Developer

Plug-in API

Appendices

Appendix A: NURBS Geometry

There are quite a few really good books on spline geometry and NURBS geometry. This appendix will not try to teach you everything about NURBS but will try to give you a general overview of the Maya particulars.



Of the six curves in the illustration, the first is a circle primitive with four spans. Notice how the circle does not touch the hull. The next five curves are attempts at replicating this geometry using the curve building tools in Maya.

The circle appears to have four CVs using the default options of the curve tool (Create > CV Curve Tool), with the following options set:

Multiple End Knots	ON
Curve Degree	3 Cubic

If you place four CVs, you produce something that looks like “Curve 1”. This is an Open curve. It’s open because the curve has a gap between the first and last CVs. It also has only one span, where the circle has four. (A span connects two edit points.)

If you try to close the curve by placing another CV on top of the first CV, you produce something that looks like Curve 2. Curve 2 is closer to the primitive circle, first because it is Closed, that is, there is no gap in the curve between the first and last CVs, and second, because it has two spans. However, you will notice that the curve does not look like a circle, it intersects the hull at the first CV and there is a discontinuity in the curve’s tangent at this point. If you were to place another CV overlapping the second CV, you would find that the curve is now open and looks even less like a circle. A third and fourth CV don’t help either, all because the first and last CVs are always on the hull.

If you go into the curve tool’s option box and turn off Multiple End Knots and place four CVs, you produce a curve which looks like Curve 3. This looks more promising than Curve 1 since the curve does not intersect the hull. If you now place a fifth CV on top of the first CV you produce Curve 4. This still looks promising. If you go further and add a sixth and seventh CV on top of the second and third CV, you produce Curve 5. This looks exactly like the circle. You’ve done it.

Well, not quite. If you were now to pull one of these additional CVs away from the CV under it, you would pull the curve apart, producing an open curve again. However, no matter how you pull the CVs on the circle you cannot pull it apart, it remains a closed curve. So there is still a small difference between these two curves. The difference is the form of the curve. Curves 1, 3 and 4 are all Open, that is, their form is open. Curves 2 and 5 are closed, that is, their form is closed. The circle is neither open nor closed, it’s form is a third type, called periodic. Periodic implies that the last degree CVs of the curve overlap the first degree CVs, and all operations on the CVs ensure that they stay together and cannot be pulled apart.

A periodic curve generally has tangency continuity on the whole curve while a closed curve will not.

Examples

The following plug-in creates Curve 1. To ensure that the CVs interpolate the end points of the curve, the knots of the curve are duplicated (piled up) at each end.

```
#include <maya/MSimple.h>
#include <maya/MPointArray.h>
#include <maya/MDoubleArray.h>
#include <maya/MFnNurbsCurve.h>
```

```
MStatus curveTest( const MArgList& )
{
    MFnNurbsCurve    curveFn;

    const double cvs[][4] = {
        { -1, 0, -1, 1 },
        { -1, 0,  1, 1 },
        {  1, 0,  1, 1 },
        {  1, 0, -1, 1 }
    };
    const double knots[] = { 0, 0, 0, 1, 1, 1 };

    MPointArray cvArray( cvs, unsigned( sizeof(cvs) / (4*sizeof(double)) ) );
    MDoubleArray knotArray( knots, unsigned(sizeof( knots )/sizeof( double ))
);
    MStatus status;
    MObject    curve = curveFn.create( cvArray, knotArray, 3,
MFnNurbsCurve::kOpen,
               false, false, MObject::kNullObj, &status );
    if ( MS::kSuccess != status )
    {
              printf( "Failed to create curve\n" );
              return status;
    }

    return MS::kSuccess;
}

DeclareSingle( curveTest );
```

The only change necessary to produce Curve 3 from Curve 1 is simply to change the knot vector so that the knots are not piled up at the ends of the curve:

```
const double knots[] = { 0, 1, 2, 3, 4, 5 };
```

Changing Curve 1 to Curve 2 is a little more involved. A CV, a duplicate of the first, has to be added to the end of the CV array, and a new knot must be inserted into the knot array.

```
const double cvs[][4] = {
    { -1, 0, -1, 1 },
    { -1, 0,  1, 1 },
    {  1, 0,  1, 1 },
    {  1, 0, -1, 1 },
    { -1, 0, -1, 1 }
};
const double knots[] = { 0, 0, 0, 1, 2, 2, 2 };
```

Curve 4 is to Curve 3 what Curve 2 is to Curve 1, that is, just an additional CV (a duplicate of the first) and an additional knot.

```
const double cvs[][4] = {
    { -1, 0, -1, 1 },
    { -1, 0,  1, 1 },
    {  1, 0,  1, 1 },
    {  1, 0, -1, 1 },
    { -1, 0, -1, 1 }
};
const double knots[] = { 0, 1, 2, 3, 4, 5, 6 };
```

Curve 5 continues on from Curve 4 with an additional two CVs (duplicating the second and third) and two knots.

```
const double cvs[][4] = {
    { -1, 0, -1, 1 },
    { -1, 0,  1, 1 },
    {  1, 0,  1, 1 },
    {  1, 0, -1, 1 },
    { -1, 0, -1, 1 },
    { -1, 0,  1, 1 },
    {  1, 0,  1, 1 }
};
const double knots[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8 };
```

This will produce a curve identical in shape to the circle primitive, however if you were to start pulling on the CVs of this curve you would find that you could pull the curve apart. To keep the curve from separating one additional change is required. When creating the curve, rather than specifying it be open (MFnNurbsCurve::kOpen) specify that it should be periodic (MFnNurbsCurve::kPeriodic).

```
MObject curve = curveFn.create( cvArray, knotArray, 3,
    MFnNurbsCurve::kPeriodic, false, false, MObject::kNullObj, &status );
```

So long as there is the proper number of overlapping CVs (one for each degree of the curve - these curves are degree three, so there should be three overlapping CVs) you can create a periodic curve. If there are insufficient overlapping CVs, the create() method will fail.

Appendix B: Dependency graph rendering nodes

Node Name	**Classification**
ambientLight	light
blendColors	utility/color
blinn	shader/surface

Node Name	**Classification**
brownian	texture/3d
bulge	texture/2d
bump2d	utility/general/bump
bump3d	utility/general/bump
camera	camera
cameraView	none
checker	texture/2d
clamp	utility/color
cloth	texture/2d
cloud	texture/3d
condition	utility/general
contrast	utility/color
crater	texture/3d
defaultLightList	none
defaultRenderUtilityList	none
defaultShaderList	none
defaultTextureList	none
directionalLight	light
displacementShader	shader/displacement
distanceBetween	none
envBall	texture/environment
envChrome	texture/environment
envCube	texture/environment
envFog	shader/volume/fog

Node Name	**Classification**
envSky	texture/environment
envSphere	texture/environment
environmentFog	none
file	texture/2d
fractal	texture/2d
gammaCorrect	utility/color
geometryShape	none
granite	texture/3d
grid	texture/2d
hsvToRgb	utility/color
imagePlane	imageplane
implicitCone	none
lambert	shader/surface
layeredShader	shader/surface
leather	texture/3d
light	none
lightFog	shader/volume/fog
lightInfo	utility/general
lightList	none
luminance	utility/color
marble	texture/3d
materialInfo	none
mountain	texture/2d
multilisterLight	none

Node Name	Classification
multiplyDivide	utility/general
nonAmbientLightShapeNode	none
nonExtendedLightShapeNode	none
opticalFX	postprocess/opticalFX
particleAgeMapper	utility/particle/mapper
particleCloud	shader/volume/particle
particleColorMapper	utility/particle/mapper
particleIncandMapper	utility/particle/mapper
particleTranspMapper	utility/particle/mapper
partition	none
phong	shader/surface
phongE	shader/surface
place2dTexture	utility/general/placement/2d
place3dTexture	utility/general/placement/3d
plusMinusAverage	utility/general
pointLight	light
pointMatrixMult	none
postProcessList	none
projection	utility/general
ramp	texture/2d
reflect	none
renderCone	none
renderGlobals	none
renderGlobalsList	none

Node Name	Classification
renderQuality	renderGlobal/quality
renderSphere	none
resolution	renderGlobal/resolution
reverse	utility/general
rgbToHsv	utility/color
rock	texture/3d
samplerInfo	utility/general
setRange	utility/general
shaderGlow	postprocess/glow
shadingMap	shader/surface
simpleVolumeShader	none
snow	texture/3d
solidFractal	texture/3d
spotLight	light
stencil	utility/general
stucco	texture/3d
surfaceLuminance	utility/color
surfaceShader	shader/surface/utility
texture2d	none
texture3d	none
textureEnv	none
useBackground	shader/surface
vectorProduct	utility/general
volumeShader	shader/volume/utility

Node Name	**Classification**
water	texture/2d
wood	texture/3d

Appendix C: Rendering attributes

Note The shading process uses the long names for attributes, so it doesn't matter what you use for short names.

Output Attributes requested by Shading Groups

Name Long (short)	**Data Type**	**Description**
displacement (d)	float	Output surface displacement distance along the surface normal
outColor (oc)	float3	Output color
outColorR (ocr)	float	
outColorG (ocg)	float	
outColorB (ocb)	float	
outGlowColor (ogc)	float3	Output glow color
outGlowColorR (ogr)	float	
outGlowColorG (ogg)	float	
outGlowColorB (ogb)	float	
outMatteOpacity (omo)	float3	Output matte
outMatteOpacityR (omor)	float	
outMatteOpacityG (omog)	float	
outMatteOpacityB (omob)	float	
outTransparency (ot)	float3	Output transparency
outTransparencyR (otr)	float	
outTransparencyG (otg)	float	
outTransparencyB (otb)	float	

Rendering Attributes available per sample

Name Long (short)	Data Type	Description
farPointCamera (fc)	float3	used for volume, the far point of the visible interval in camera space
farPointCameraX (fcx)	float	
farPointCameraY (fcy)	float	
farPointCameraZ (fcz)	float	
farPointObj (fo)	float3	used for volume, the far point of the visible interval in object space
farPointObjX (fox)	float	
farPointObjY (foy)	float	
farPointObjZ (foz)	float	
farPointWorld (fw)	float3	used for volume, the far point of the visible interval in world space
farPointWorldX (fwx)	float	
farPointWorldY (fwy)	float	
farPointWorldZ (fwz)	float	
filterSize (fs)	float3	Filter size in (u,v, w) with which to filter textures
filterSizeX (fsx)	float	
filterSizeY (fsy)	float	
filterSizeZ (fsz)	float	
infoBits (ib)	32 bit unsigned integer	Passes information from one node that may be needed by another node. Using this field, a file texture node with advanced filtering turned on (such as Quadratic filtering) can be used simultaneously as both a color map and a bump map. When rendering, Maya computes the color map using advanced filtering, but computes the bump map without it since advanced filtering is incompatible with bump mapping.

Name Long (short)	Data Type	Description
lightDataArray (ltd)	lightData	Multi-attribute representing all lights linked to the shading group
lightDirection (ld)	float3	The light direction
lightDirectionX (ldx)	float	
lightDirectionY (ldy)	float	
lightDirectionZ (ldz)	float	
lightIntensity (li)	float3	The light intensity
lightIntensityR (lir)	float	
lightIntensityG (lig)	float	
lightIntensityB (lib)	float	
lightAmbient (la)	boolean	Flag for ambient component
lightDiffuse (ldf)	boolean	Flag for diffuse component
lightSpecular (ls)	boolean	Flag for specular component
lightShadowFraction (lsf)	float	Percentage shadowing of the current light, provided shadows are enabled on the given light
matrixObjectToWorld (mow)	floatMatrix	Transformation from object space into world space
matrixWorldToObject (mwo)	floatMatrix	Transformation from world space into object space
mediumRefractiveIndex (mrfi)	float	refractive index of the medium through which the incident ray was travelling before it hit the point being shaded
normalCamera (n)	float3	Surface normal in camera space
normalCameraX (nx)	float	
normalCameraY (ny)	float	
normalCameraZ (nz)	float	
numShadingSamples (ns)	char	Number of shading samples to take for this surface

Name Long (short)	Data Type	Description
objectId (oi)	int	unique ID for the object being shaded, may not be the same ID across frames
objectType (ot)	char	the rendering type (0=unknown, 1=surface, 2=volume(not particles), 3=blobby surface, 4=particle system, 5=image plane)
particleAge (pa)	float	age of the particle currently being shaded
particleColor (pc)	float3	per-particle color as provided by a particle color mapper
particleColorR (pcr)	float	
particleColorG (pcg)	float	
particleColorB (pcb)	float	
particleId (pid)	int	unique identifier for the particle being shaded
particleIncandescence (pi)	float3	per-particle incandescence as provided by a particle incandescence mapper
particleIncandescenceR (pir)	float	
particleIncandescenceG (pig)	float	
particleIncandescenceB (pib)	float	
particleLifespan (pls)	float	life-span of the current particle
particleTransparency (pt)	float3	per-particle transparency as provided by a particle transparency mapper
particleTransparencyR (ptr)	float	
particleTransparencyG (ptg)	float	
particleTransparencyB (ptb)	float	
particleWeight (w)	float	weight of the current particle
pixelCenter (pc)	float2	center of the pixel currently being shaded in screen space
pixelCenterX (pcx)	float	
pixelCenterY (pcy)	float	

Name Long (short)	Data Type	Description
pointCamera (p)	float3	xyz location of geometry in camera space
pointCameraX (px)	float	
pointCameraY (py)	float	
pointCameraZ (pz)	float	
pointObj (po)	float3	xyz location of geometry in object space
pointObjX (pox)	float	
pointObjY (poy)	float	
pointObjZ (poz)	float	
pointWorld (pw)	float3	xyz location of geometry in world space
pointWorldX (pwx)	float	
pointWorldY (pwy)	float	
pointWorldZ (pwz)	float	
rayDepth (rd)	int	during raytracing, the depth of the current ray (the primary ray has a depth of 0)
rayDirection (rad)	float3	The direction of the current intersection ray in camera space
rayDirectionX (rdx)	float	
rayDirectionY (rdy)	float	
rayDirectionZ (rdz)	float	
rayOrigin (ro)	float3	The origin of the current intersection ray in camera space
rayOriginX (rox)	float	
rayOriginY (roy)	float	
rayOriginZ (roz)	float	
refPointCamera (rpc)	float3	The current reference sample point that has to be shaded. Used in conjunction with reference objects.
refPointCameraX (rcx)	float	
refPointCameraY (rcy)	float	
refPointCamearZ (rcz)	float	
refPointObject (rpo)	float3	The current reference sample point that has to be shaded. Used in conjunction with reference objects.
refPointObjectX (rox)	float	
refPointObjectY (roy)	float	
refPointObjectZ (roz)	float	

Name Long (short)	Data Type	Description
refPointWorld (rpw)	float3	The current reference sample point that has to be shaded. Used in conjunction with reference objects.
refPointWorldX (rwx)	float	
refPointWorldY (rwy)	float	
refPointWorldZ (rwz)	float	
tangentUCamera(tu)	float3	The U tangent of the surface in camera space
tangentUx (tux)	float	
tangentUy (tuy)	float	
tangentUz (tuz)	float	
tangentVCamera (tv)	float3	The V tangent of the surface in camera space
tangentVx (tvx)	float	
tangentVy (tvy)	float	
tangentVz (tvz)	float	
triangleNormalCamera (tnc)	float3	Normal of the visible triangle in camera space.
triangleNormalCameraX (tnx)	float	
triangleNormalCameraY (tny)	float	
triangleNormalCameraZ (tnz)	float	
uvCoord (uv)	float2	texture UV coordinates in surface parametric space
uCoord (u)	float	
vCoord (v)	float	
uvFilterSize (uf)	float3	The sample (filter) size
uvFilterSizeX (ufx)	float	
uvFilterSizeY (ufy)	float	
vertexCameraOne (vc1)	float3	vertex one of the triangle currently being shaded in camera space
vertexCameraOneX (c1x)	float	
vertexCameraOneY (c1y)	float	
vertexCameraOneZ (c1z)	float	
vertexCameraTwo (vc2)	float3	vertex two of the triangle currently being shaded in camera space
vertexCameraTwoX (c2x)	float	
vertexCameraTwoY (c2y)	float	
vertexCameraTwoZ (c2z)	float	

Name Long (short)	Data Type	Description
vertexCameraThree (vc3) vertexCameraThreeX (c3x) vertexCameraThreeY (c3y) vertexCameraThreeZ (c3z)	float3 float float float	vertex three of the triangle currently being shaded in camera space
vertexUvOne (vt1) vertexUvOneU (t1u) vertexUvOneV (t1v)	float2 float float	texture coordinate of the triangle currently being shaded
vertexUvTwo (vt2) vertexUvTwoU (t2u) vertexUvTwoV (t2v)	float2 float float	texture coordinate of the triangle currently being shaded
vertexUvThree (vt3) vertexUvThreeU (t3u) vertexUvThreeV (t3v)	float2 float float	texture coordinate of the triangle currently being shaded

Rendering Attributes available per frame

Name	Data Type	Description
cameraFarClipPlane (fcp)	float	Far clipping plane distance for camera view
cameraNearClipPlane (ncp)	float	Near clipping plane for camera view
hFilmAperture (hfa)	float	width of the film (inches) from the camera being currently rendered
hFilmOffset (hfo)	float	horizontal offsets of the film (inches)
isPerspCamera (ipc)	boolean	If TRUE (non-zero), camera is perspective projection, else it is orthographic
lensSqueezeRatio (lsr)	float	squeeze ratio of the camera being currently rendered

Name	Data Type	Description
matrixEyeToNormPersp (etp)	floatMatrix	Transformation from camera (eye) space to normalized perspective space
matrixEyeToWorld (e2w or etw)	floatMatrix	Transformation from camera (eye) space to world space
matrixNormPerspToEye (pte)	floatMatrix	Transformation from normalized perspective space to camera (eye) space
matrixWorldToEye (wte)	floatMatrix	Transformation from world space to camera (eye) space
vFilmAperture (vfa)	float	height of the film (inches) from the camera being currently rendered
vFilmOffset (vfo)	float	vertical offsets of the film (inches)
xHighRenderRegion (hrx)	int	resolution of the image in x
yHighRenderRegion (hry)	int	resolution of the image in y
xLowRenderRegion (lrx)	int	always 0
yLowRenderRegion (lry)	int	always 0
xPixelAngle (xpa)	float	The maximum angle subtended in X or Y by a single pixel

Appendix D: Frequently asked questions

This list is a compilation of questions received from programmers using the Maya API. A set of categories has been defined (see the list below), and the questions have been organized into these categories to make the presentation more logical.

The current list of Categories is:

- "General Questions"
- "Documentation Questions"
- "Dependency Graph Questions"

- "GUI Questions"
- "Animation Questions"
- "Windows Questions"

General Questions

Q:

How do I know what units an API method returns?

A:

Unless otherwise specified all API methods use Maya internal units: cm and radians.

Q:

Is there a stand-alone mode similar to OpenModel? Or, would a stand-alone be similar to Softimage where the basic software package (and license) is required?

A:

Yes. Chapter 10, "Setting up your plug-in build environment," and the documentation on the MLibrary class describe how to set this up and use it. There are also several example stand-alone applications. Descriptions of these can be found in Chapter , "Example Plug-ins."

Q:

While OpenModel presents the same API interface as OpenAlias, many of the function calls (even the non-UI ones) work differently or not at all in OpenModel. The lack of render control is a prime example, where some parts of the same function set work and others don't. This is extremely frustrating, especially since the renderer is a very likely candidate for being done in batch. I cannot tell if the same disparities will show up with Maya in batch and interactive modes, but I will be quite disappointed in they do.

A:

This problem should not exist in the Maya API. Of course, UI calls will not work when run in library mode, but all other calls should behave identically.

Q:

The use of angular units is inconsistent. In Maya's attribute window, transform rotations are given in degrees. However, in the API, attaching to the rotation attributes requires writing out radians. When writing out values for keyframes, we want to be consistent with what we see in the UI.

A:

When dealing with the UI, the API uses the MAngle class. This class contains a method “uiUnit” that returns the unit the user has chosen in the UI. This value is a user preference that can be changed at any time. The MAngle class defaults to radians in all cases so that plug-in code does not have to adjust to the UI preference currently in effect. You can adjust to the units that have been set by the UI user with the following code:

```
MAngle foo;
foo.setInternalUnit( foo.uiUnit() );
```

After this, all new MAngle instances will operate in the same units as the UI, until, of course, the UI user changes his/her preferences again.

Additionally, the rotation functions in the transform class deal in doubles that represent radians for efficiency reasons, as that is what the underlying implementation demands. If you are acquiring angular data from the UI, you should access it via the asMAngle method in MArgList and use the asRadians method to extract the value required by the transform class.

Q:

We want to modify the blind_data example to add a multidimensional array as a dynamic attribute, but don’t know which MFnAttribute class is the most appropriate for this operation.

A:

However, you can make an attribute, of whatever type, and turn it into an array by calling the setArray method of the MFnAttribute class. If you need a multidimensional array, you will have to build it on top of this using some form of index conversion.

If this is not sufficient, another option is to derive a whole new data type off MPxData that can directly store a multidimensional array. This is more work however, and you will need to implement the readASCII, writeASCII, readBinary, and writeBinary virtual methods derived from MPxData in order for the new data type to save and restore correctly. There are 3 example plugins provided that demonstrate how this is accomplished: blindShortDataCmd.cc, blindDoubleDataCmd.cc and blindComplexDataCmd.cc.

Q:

How can we get notified when an attribute of a node changes?

A:

There is a hierarchy of classes rooted at the MMessage class in the API that provide a way for you do register callback functions that will be invoked when a particular Maya event occurs. There is a large set of message that, among other things, allow you to find out when an attribute changes.

Q:

How to we bake data via the API?

A:

The “bakeResults” command can be used to bake animation data. All expressions, motionPaths, animCurves, etc., are replaced in the dependency graph with a single animCurve the will produce the same motion. You can access this functionality via the “MGlobal::executeCommand” method.

As well, the MEL command delete (-ch option) removes construction history for an object. From the API, you will also have to access this via the MGlobal::executeCommand method. Access to this functionality is also available from the UI under Edit > Delete > Construction History menu item.

Q:

The MFnNurbsSurface::cv() method returns an MObject, but it is unclear what function set can be used to access the returned information. A function set that operates on a class that represents CVs or that operates on MPoints does not exist.

A:

Use the MItSurfaceCV function set. While this might be a little counter-intuitive, the MObject returned by the CV method actually returns a component structure that can contain multiple CVs of an object, and so the CV iterator is required to unpack it and get at the CV data.

Q:

When a group node is selected, all objects in the group are highlighted as if they are selected, but the global selection list only has the group node in it. Is this the way it’s supposed to be? It seems to me that everything in the group would be in the list, since they are all selected. (I’m constructing an MItSelectionList from the global active selection list and using MFn::kInvalid as the filter...)

A:

This is indeed the way it works. This is not an issue with the API or MItSelectionList, but rather the way Maya works. You can see identical behavior by starting the Hypergraph (Window > Hypergraph) and then:

- create two or three primitives
- select them all
- select Edit > Group

Notice in the Hypergraph that only the group transform, "group1" is "selected", even though all the primitives in the group are "highlighted". As far as Maya is concerned, only the one node is selected, and so that single node is the only one returned by MItSelectionList. You certainly can select the individual primitives in the Hypergraph, or by name using MEL, but the UI only selects the group transform.

However, you can get the list of objects that are "highlighted" when the transform is selected via the API call MGlobal::getHiliteList.

Q:

This question concerns simple API array structures, like the MPointArray. Is the data stored in an MPointArray contiguous, or is it stored as a linked list?

In other words, does the append() method just add another element on (as in a linked list) or is it doing the equivalent of a realloc() function (allocates a new contiguous block of data plus one element, and then copies the old data over)?

A:

It is not a linked list, however, neither does it do a realloc on each append (or insert either). Instead, it manages a logical/physical space model, and expands the physical space by a user-configurable number of elements when more is needed.

All the "*Array" should contain the methods "sizeIncrement", and "setSizeIncrement". The former tells you by how many elements the array will grow when it needs to, and the later allows you to change that value.

As of Maya 2.0, the constructors for all the array classes accept an initial size parameter. So if you know the size of your array in advance, you can completely avoid any growing/copying overhead in the array.

Q:

Why is a new instance of a command created every time it is invoked from the command window? Is this somehow related to Maya's undo capability? When do these instances of the command object get deleted?

A:

Maya implements its infinite undo capability as follows:

When a command or tool is invoked, the creator function for that object will be called to create a new instance. That instance must contain local data members sufficient to retain state so that when its doIt method is called, it can save enough state to:

- undo what it is about to do
- redo what it is about to do

Typically, a command's doIt method will just save the current state of what it is about to change for undo, then cache the parameters of the "about to be performed" operation and call redoIt.

redoIt operates off the cached parameters, and if called from the undo manager, can "redo" the operation without any further user interaction.

undo also operates off the cached data, and can also work without any further user interaction.

Additionally, when a tool is "finished", its virtual method "finalize" (that is provided in the MPxToolCommand base class) will be called. This routine is responsible for constructing an MArgList containing a command that will "redo" the operation. This command string is written in to the Maya Journal to record all the operations that have taken place.

If the virtual method MPxCommand::isUndoable is overridden and made to return "false" (it defaults in the base class to "true"), then right after the doIt method is called, Maya will call the destructor for the command instance. Otherwise, the instance is passed to the undo manager which will call its undoIt and redoIt members to implement undo and redo requests. When the undo queue is flushed, all the instances of the commands or tools are destroyed, thus freeing the local memory that is caching the parameters needed for undo or redo.

Q:

Regardless of whether a single CV or multiple CVs are selected via the UI, the MItSelectionList iterator will only return one selection item. If multiple CVs were selected, how can I find which ones?

A:

If you select a multiple CVs, and then use the MItSelectionList iterator class of the API, all the selected CVs will be returned in a single component.

To access the individual CVs you must use the getDagPath method of MItSelectionList, which returns both an MDagPath and a MObject, then pass these as arguments to the constructor of an iterator. For NURBS surfaces, the MItSurfaceCV class would be used to extract the individual surface CVs. The iterators: MItCurveCV, MItMeshVertex, MItMeshEdge and MItMeshPolygon can be used to perform similar operations on NURBS curve, and the various components of polygonal objects. The lassoTool plug-in provides a good example of how this is done.

Q:

What is the meaning of the value that MItSurfaceCV::index() returns?

A:

One of the components of a NURBS surface is an array of CVs. The index method returns the position of the given CV in the array maintained by the surface. The UI represents this as a 2D array of CVs with rows and columns of CVs corresponding to U and V indices. Internally this is stored as a 1D array (row1, row2, row3, etc.) and index returns the position of the CV in this data structure. (Incidentally, if you created the surface via MFnNurbsSurface::create, this is the way you had to provide the CV array). You can convert this to a pair of 2D indices via:

```
sizeInV = MFnNurbsSurfaceInstance.numCVsInV();
indexU = index() / sizeInV;
indexV = index() % sizeInV;
```

The method getIndex of the MItSurfaceCV class returns the indexU and indexV values using exactly this calculation

Q:

How can I compare two components of a object to see if they are the same? Specifically, I need to compare two CVs on a NURBS surface, but this problem appears to apply to all types components of both NURBS and polygonal objects.

A:

There is no simple mechanism for doing this. Components are identical if the are members of the same Dag path (the MDagPath class defines an == operator to perform this comparison), and if their indices, returned by the index methods of the various component iterators, are also the same.

Q:

I created an instance of an MFnNurbsSurface function set, and got good data out of it, however, I then called MGlobal::viewFrame, to move the animation to another frame. I know the surface moved, but I got the same data out of the function set as I did the first time. How can I make this work?

A:

You will have to restructure your code a little to make this work.

After a call to MGlobal::viewFrame, is it necessary to *rebind* your function sets to the objects they are accessing. This can be handled by code that looks like the following:

```
    MDagPath        path;
    // initialize path somehow to refer to the object in
question
    MFnNurbsSurface surf;

    for (int i = 1; i <= maxFrames; ++i) {
        MGlobal::viewFrame(i);
        surf.setObject(path);
        }
```

The MDagPath will remain valid across frames, and thus can be used to rebind the function set to the object in each frame.

Documentation Questions

Q:

We would like to have full documentation on all the transformations on a CV as it is positioned in world space. For example, CVs end up in their global position via a number of matrices, clusters, functions, animations etc. Documentation of these transforms explicitly and exactly would be very useful.

A:

We believe that the set of possible transforms is both too complex and do dynamic to document in the general case. Take clusters for example. Clusters in Maya are implemented as deformers. This means that a deformer is put between the original surface and the new output surface.

It is therefore possible to get the world transformation information from the deformed CV up to the world through the DAG shape that holds the deformation result. So, it is trivial to query the world space location of a point. We get that for free from our current implementation.

However, the local to world transformation of any point is arbitrarily complex. Nodes can easily implement procedural transformations that don't involve matrices at all. Conceptually, the architecture is one in which a point in local space is passed through a series of "black boxes" each of which affect its position and we simply don't know what is in all of the boxes.

This implies that we can't set the position of a point (or CV) exactly in world space either. To do so would require computing the inverse of the local to world space transformation, and I have just been busy telling you we don't know how to define that transform in general.

That being said, if you are just interested in the order in which Maya's transform node applies scale, rotation, translation and other transformations from its attributes to an object, this is described in detail in the documentation for the transform node in the Commands online documentation in xform.html.

Dependency Graph Questions

Q:

Can we derive our own custom classes/nodes from the standard classes/nodes such that they will be correctly processed in the DG? This capability has been inferred to in the past and we just want to get the most recent confirmation on this capability.

A:

Maya maintains ownership of the MObjects which it presents as opaque data. So it is not possible to derive from these objects.

Additionally it isn't possible to derive from Maya's internal nodes. For example, let's say you wanted to derive something from the internal revolve node. The revolve node, like all nodes, is simply a compute function on a set of attributes. To make modifications to this node you would require the source code to the compute method - which we can't give you. Instead what can be done is to connect new (user-written) nodes (see the next paragraph) to the attributes of the Maya revolve node and use these new nodes to modify the input and output of that node. Alternatively, in this case, a new revolve node could be written and used in place of the system defined one.

Maya Proxy objects are designed expressly for derivation, and allow new user nodes to be added to Maya.

As well, it is possible to derive from the function sets to create new operations on the MObjects (limited only by the fact that the MObjects are opaque, and the source for the implementation of the function set members is unavailable).

So, in summary, you can't derive directly from Maya nodes, but you can create your own nodes, insert them into the dependency graph and have them either replace an existing Maya node, or modify the input or output parameters of Maya nodes.

Q:

It looks like the user-defined nodes are fundamentally different from the already existing nodes. If you look at something like MFnNurbsCurve, you see that eventually, it is derived from MFnBase, but if you look in the circle example, you see that the "circle" is derived from MPxNode.

A:

There is a fundamental difference between "function sets" and "maya objects". Maya internal objects (which include dependency nodes) are encapsulated in MObjects and function sets, which are indeed derived from MFnBase, are initialized "with" an MObject and then act upon it. This is sort of an "outward > in" kind of paradigm in which user written code is allowed to affect the internals of Maya objects.

To create a user-defined dependency node, we have to do something completely different, which is why the MPxNode classes are necessary. Effectively, what we do is create a new internal Maya node, and "hook up" its methods to the ones defined in the customer generated node. For example, if during the evaluation of the dependency graph it is necessary to "recompute" a user defined node, what happens is:

- dependency graph evaluator calls the compute method for our "internal node"
- its compute function calls out to the compute function of the user written node, gets the result, and
- passes the result on.

This is repeated for any of the attributes of the node that require recomputation. This is sort of an "inward > out" kind of paradigm in which internal Maya objects have to call a user written compute function.

So, yes there are fundamental differences, but that is intentional and caused by the fact that the problems are fundamentally different.

Q:

It is also unclear if we can accomplish a "persistent" effect through the API. That is, if the CV gets altered, the arclen will change. So anything that is attached to our arclen attribute would need to be moved or sized accordingly.

A:

As long as the propagation of values is done through connections in the dependency graph, this is taken care of automatically. For example:

MayaNodeA.cvSet > customerNode.input > CustomerNode.output(computes scale from arclen) > MayaNodeB.scale

A change in a CV in MayaNodeA automatically forces a recompute in CustomerNode and MayaNodeB, and the object is moved or sized accordingly.

Q:

We want to be able to drive an attribute of one object by a derivable value of another object. For example, we may want to drive the scale of one object by the arclen of a curve. Or we may want to translate an object according to the evaluated value of a curve at a particular parametric value. The examples show how to instantiate a dependency node that allows us to tie together attributes of objects, but can we take it one step further and have the driving value be an evaluated value, e.g. myNurbsCurve.arclen() or myNurbsCurve.point(0.5)?

Additionally when creating your own node, can you create an input attribute that takes a node or MObject as its input, rather than a float or a string? This would allow us to jump back into the MFnNurbsCurve and use whatever derived methods we want.

A:

Such a construct is fairly easily handed in the Maya architecture by writing a node that has an MFnTypedAttribute. One parameter of the declaration of such an attribute is the “type” it accepts as input. It is quite possible to specify it as taking a nurbsCurve by using the type “kNurbsCurve”. Since your node will then have the actual curve, you can compute anything you want based upon it.

So the node I think you want to write would have a nurbsCurve input attribute, and three double (scale) output parameters. All you need to do is connect this node to a node that produces a curve in the input, connect its outputs to the scale inputs of the node you want to animate, and the dependency graph will do the rest. Any change in the curve node will automatically propagate through the graph and update the scale of the final object. The example plug-in called arcLenNode demonstrates how to do this.

Furthermore, inside your node, you will really have a nurbsCurve object, and thus you can attach a MFnNurbsCurve function set to that object. Then you can use any of its methods that compute values you need. The “length” function will compute the arclen of the curve, and the “pointAtParm” method will return a point at a particular parameter value.

As well, a similar result can be obtained by simply hooking together existing dependency nodes. For example, the subCurve and curveInfo nodes allow you compute the arclen of a subcurve as shown below:

```
global float $arclen;

// Create a curve
curve -p -5 0 8 -p -9 0 2 -p -3 0 5 -p -6 0 -2
    -p 1 0 3 -p -4 0 -5 -p 4 0 1;

    // Create a node to extract part of a curve,
    // set the parts to keep, and attach it to
     // the curve created above.
    createNode -n subCurve1 subCurve;
    setAttr subCurve1.minValue 0.4;
    setAttr subCurve1.maxValue 1.4;
    connectAttr curveShape1.local subCurve1.inputCurve;
```

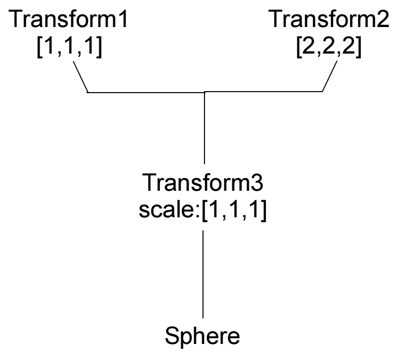
```
 // Create a curveInfo node, and connect
// it to the output of the subcurve.
createNode -n curveInfo1 curveInfo;
connectAttr subCurve1.outputCurve curveInfo1.inputCurve;

// Get the arclen of the subcurve from the
// curveInfo node and print it.
$arclen =`getAttr curveInfo1.arcLength`;
print("curve[0.4:1.4] has arclen " + $arclen + "\n");

// Change the part of the curve extracted by the
// subCurve node  and print the new arclen
setAttr subCurve1.minValue 0.0;
setAttr subCurve1.maxValue 5.0;
$arclen =`getAttr curveInfo1.arcLength`;
print("curve[0.0:5.0] has arclen " + $arclen + "\n");
```

Q:

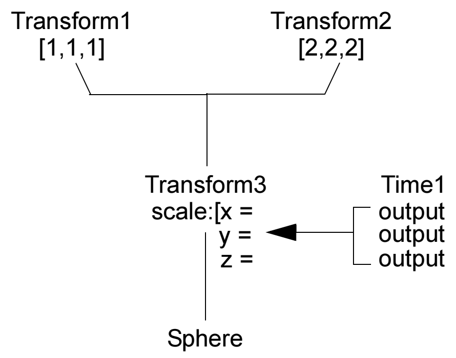
I need a better understanding of the differences between DagNodes and Dependency Nodes and how these relate to the API object classes. In particular, how do I know when to use the getDagPath method from the MItSelectionList iterator, and when do I use getDependNode?

A:

A DAG nodes describe how an instance of an object is constructed from a piece of geometry. For example, when you create a sphere, you create both a geometry node (the sphere itself) and a Transform Node that allows you to specify where the sphere is located, its scaling, etc. It is quite possible to have multiple transform nodes attached to the same piece of geometry. For example:



The dependency graph, however, is something new. All DAG nodes are also dependency nodes, but not vice-versa. For example there is a "time1" dependency node that can produce the frame number of the current animation. The "circleNode" and "sineNode" types created by the "circle.cc" and "sine.cc" plug-in examples are dependency nodes, however, they are not part of the DAG. Instead dependency nodes can be wired together to provide a dynamic evaluation graph that can end up affecting DAG nodes (and thus affecting what is drawn). For example,



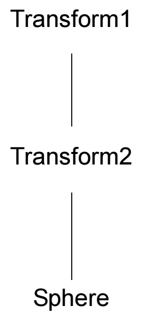
In this example the x, y, and z scale parameters of Transform3 are driven by the frame number. Thus as the animation is run, the 2 instances of the sphere will grow.

So, now from the API, how do you know when to used getDagPath and when to use getDependNode?

Well, if you pick something on the screen with the mouse then you will always be picking an instance and thus you will always have a DAG node available, and you should use getDagPath.

If you pick something by name, then you might or might not get a DAG node. The right thing to do in this case is ask. The MItSelectionList iterator's "itemType" method will return an element of an enum that will differentiate between the two node types, you can then call getDagPath or getDependNode as appropriate.

Yet another important thing to understand in Maya is that geometry DAG nodes do not have transformation matrices. They rely on the transform nodes above them for their transformation information. Because of this, selection of geometry in 3D views always causes the transform node above the geometry be selected rather than the actual geometry node. This allows all of the transformation tools to work properly.



So, in the above diagram, clicking on Sphere1 in a 3D view will cause Transform2 to be selected.

For instance, if you are iterating through the selection list looking for the sphere and you want to perform an operation upon the sphere's CVs, in the iteration, you will eventually come to Transform2.

If you get Transform2 as a dependency node (via getDependNode), then you have a transform node. From this transform node, you will not be able to find either the sphere or its CVs. Additionally, if your object is instanced (as in the first diagram), then you will have lost the information about which instance was selected.

If you get Transform2 as a DAG node, you will get a DAG path object. The DAG path object is more intelligent. It knows where the transform resides in the DAG. If you give the DAG path to the NURBS surface function set (or the iterator), then the sphere node under the transform will be found automatically and the CVs will be available for modification.

Q:

I can't figure out how to get the transform matrix of an object. I have played around with attaching the MFnTransform function set to dependency node and dag paths, but can't quite seem to get it right.

A:

You were close. To accomplish this you must first get a DAG path structure for the object, then attach the MFnTransform function set to it. You can use that to get a MTransformationMatrix object, which can access and update the transform for the object in numerous ways, including returning the entire matrix. To solve your problem you would include code like:

```
MDagPath                mdagPath;
MStatus                 status;
MTransformationMatrix   transform;
MMatrix                 matrix;
```

```
if ( mdagPath.hasFn(MFn::kTransform) ) {

// Get the transform matrix via the Dag path.
MFnTransform transformNode(mdagPath,&status);

// Get the transform matrix via the function set
transform = transformNode.transformation(&status);
matrix = transform.asMatrix();
...
    }
```

Q:

How can I create a revolve-like plug-in, and have it take a curve, create a surface from it, and when the curve is modified, regenerate the surface?

A:

In order to implement this "history" functionality, you must write your own "revolve" node. It will take the curve as input (using the MFnTypedAttribute - see the arcLenNode plug-in) and output a surface. The output attribute of this node should then be connected to the create attribute of a nurbsSurface node which will draw it. In MEL the typical way to hook this up would be:

```
createNode transform -n revolvedSurface1;
createNode nurbsSurface -n revolvedSurfaceShape1 -p revolvedSurface1;
createNode yourRevolveNode -n yourRevolveNode1;
connectAttr yourRevolveNode1.outputSurface revolvedSurfaceShape1.create
```

The simpleLoftNode example plug-in and provides a good example of how to do this.

Q:

If I set up a deformation in Maya, and then traverse the DAG tree from a plug-in, under the transform node for the deformed surface I see two shape nodes, both of which can be interpreted as MFnNurbsSurface - one is the deformed shape, and one is the shape in some neutral position. I need a flag to indicate which is the neutral position, so I know that it's not really there, and don't need to operate on it.

A:

The 2 surfaces are differentiated by their boolean intermediateObject attributes. If value of the attribute is TRUE, then this node is the input surface for the deformation and can be ignored.

You can check the value of this attribute by creating a plug for this attribute on for each of the two nodes, and then get the value of the attribute from the plug. Alternatively, the convenience routine isIntermediateObject in the MFnDagNode function set performs this operation.

Q:

How can one used Maya multi attributes to implement an array of a user defined data type?

A:

This is not how you implement arrays of a user defined data type. When dealing with user defined data, Maya doesn't know or care what the data looks like. As you point out, it is tempting to create one data type and then attempt to create multiple instances of it via multi-plugs, but multi-plugs were designed for multiple connections in the DG rather than data storage, so this won't work.

Instead, to implement an array of data, one must create the array inside a user defined data type and use the method outlined in the blindComplexDataCmd example to access that array. For example,

```
class blindComplexData : class MPxData {
    public:
    // override methods like readBinary()....
    // define any data you want, for example, an array of
integers
    int a[12];
};
```

After adding the above user-defined data as a dynamic attribute, access the data by attaching a plug to it the usual way and do a getValue() to get a handle to the data. Once you have the handle, convert it to a pointer to an instance of your custom data type.

Q:

Given a NURBS surface that is implemented via a cluster, moving the CVs of this surface via the MFnNurbsSurface function set seems to have no effect. Why is this, and how can this be done?

A:

If clusters are present, then you have a deformer network which is computing the shape of the surface that appears in the Dag. If you move a CV on that "final & visible" surface, the deformer will simply move it back since the deformer is creating that surface. There are two ways to cope with this. First, find the "intermediate object" that is the input surface to the deformer and move the CV there. The problem with this approach is that it is difficult to predict what effect moving an input CV will have on its position on the output surface.

The other approach is via tweaks which provide the capability to move (or tweak) a CV after a deformer has determined its position. There is no specific API in MFnNurbsSurface for handling tweaks (and there won't be for Maya 1.0) but you can create a tweak by directly modifying the attributes of the nurbsSurface dependency node. To do a tweak you must:

- set the boolean attribute "tweak" to true.
- Create a CV array with the same dimensions as the input surface to the nurbsSurface node, and
- initialize all the CVs in that array to [0,0,0].
- For the CVs you want to tweak, set the corresponding element in the CV data structure and then use setCVs to update the array.

If the boolean attribute "relativeTweak" is true, the values in the CV array are used to move the corresponding CV relative to the position in which the deformer puts it, otherwise they are absolute positions of the CVs.

GUI Questions

Q:

Can a plug-in open its own graphics window (via winopen(), for example) and do graphics without interfering with the rest of Maya? If so, should it use OpenGL? How does it handle events (such as mouse movements within its own window) and still work with the rest of Maya? Does Maya provide an event handling mechanism that the plug-in can use and be compatible with the rest of Maya?

A:

Yes. In fact the helixMotifCmd example plug-in (not available on Windows) shows how to open a Motif window. You can also use OpenGL in such a window.There is one restriction however, Maya must retain control of the event loop. This should not be a problem however as your window simply needs to register callbacks for its UI elements, and Maya will happily invoke them for you.

A question for you however is "why do you really want to do this?" As there are several other ways to create windows that you might prefer.

First of all, the MEL scripting language in Maya allows you to create new windows and access virtually all Maya features. If you need a window for a dialog box, doing this in MEL is both easy and far and away the most efficient method of implementing such a feature.

Maya also contains a new class called MPxLocatorNode that allows you to create DAG objects and provide a draw routine for them implemented with OpenGL calls. These objects are called locators because they do not render. So, you can use them for screen feedback, but not to create renderable objects. The example plug-ins footPrintNode and cvColorNode provide examples of how to create and use locators.

As for input events, we currently supply an API class called MPxContext that allows you to handle such events. The example plug-ins marqueeTool, helixTool and lassoTool all demonstrate how to implement this.

Q:

How does one tell when both left and mid buttons are pressed at the same time? Right now only one is reported (I think).

A:

When you press a second button it does not generate an event but instead it is stuffed into the modifier for the hold event for the first button. For example, say the user presses the left mouse button. To see if the user has pressed the middle mouse button while the left is still down, check the modifier for the doHold event using MEvent::isModifierMiddleMouseButton().

Q:

How can I tell which 3D window is active? I thought of using the camera name to do this, but this will fail if the user changes the camera name. I need a function returning which window the 3dview is (XY, XZ, YZ, pers).

A:

This is a little difficult. Because any view can be arbitrarily tumbled or changed into a perspective view, a general solution to this problem requires a bit of work.

M3dView::active3dView will give you the active view from which you can get the camera. You can then use MFnCamera methods upDirection and rightDirection to get the respective vectors, and compare them against MVector::xAxis, MVector::xNegAxis, etc. in order to determine the view that the user is seeing.

Q:

The viewToWorld method in M3dView correctly maps 2D coordinates to 3D coordinates in orthographic windows, but returns bogus value in its cursor argument in the perspective window.

A:

For the perspective view, you must use the version of viewToWorld which returns points on both the near and far clipping planes given a point in the 2D view. Any point on the line segment connecting those points is a valid solution to the mapping, and you will have to determine on your own which of these points you wish to use.

Animation Questions

Q:

Animation “created” in the DG via a user-defined node does not show up in the animation curve graphs, i.e. there is no way to see the results of procedurally generated animation.

Only animation generated via keyframe animation shows up in the animation curve graphs.

A:

Unfortunately, this is unlikely to change. For a keyframed animation, we need only to check whether the node connected to an attribute is an animCurve node, if so, it is quite simple to extract the keyframed attribute values for display in the graph.

For any other type of animation, we only know if it is an animation if somewhere in the graph connected to a particular attribute we find a time node.

For a node which performs a procedural animation, we would actually have to run the entire animation, and save the output attribute values at each frame for display. This has the potential to be extremely computationally intensive. As well, the resulting curves would not be editable as they would be displaying only output values with no access to the knowledge on how they are computed.

Q:

I need in some way to be able to query any data at any animation frame. In OM/OA I do this by doing a viewFrame(x) and then checking the data. This is very slow however, and really what I might want to know is the position of this particular node at time x. In OM if you do a viewframe on a sub node it usually returns an incorrect value depending on how the animation has been set up.

A:

This should be better in Maya as the dependency graph will make sure that the “subnode” kind of information is always accurate.

You can query the data you are interested in a manner quite similar to that in OM/OA: perform a viewframe, to set the time, then query the attributes (like tx, ty, tz, etc.) of the node you are interested in via the getValue methods of the MPlug class. As you point out however, this is somewhat slow, since viewframe changes the global time.

In the Maya API you have another option however. You can create an instance of the MDGContext class initialized to the time you are interested in. This can be passed to the getValue method of the MPlug class and the attribute you are interested in will be evaluated at the specified time. As much or as little of the dependencies as necessary will be reevaluated in order to ensure you get an accurate answer. Lots of dependencies on other nodes will make the evaluation slow. Otherwise it should be fast.

You should also be aware that Maya does not maintain the state of animated objects at all possible keyframes so the only way to determine an object's animation state at any particular time is by querying it at a particular time using one of the two methods described above.

Q:

Can time be set randomly with MGlobal::viewFrame()? Are particles and IK stuff properly updated in all cases?

A:

You can set the time randomly, and everything will always be updated properly. However, the underlying mechanism is highly optimized towards monotonically increasing time. You can incur a large performance penalty when jumping time around randomly.

Q:

The method by which MFnMotionPath does its movement is unclear. In particular, how it interacts with the DAG tree. Suppose you have a Transform parenting a child shape. You can set keyframes and animate the translate channels and they turn green. Or you can do MotionPath and it doesn't affect translate channels.

Note that if you do have both MotionPath and translate, the MotionPath overrides the translate, and translate is ignored.

What is the mechanism by which MotionPath affects the movement?

Can you read the value at any given time? How?

If I'm traversing a DAG tree, how can I tell an object's position?

A:

Keyframes are provided by anim curve dependency nodes, and similarly, motion paths are implemented as dependency nodes. These nodes function because they are connected to the transformation attributes of the parent transform in the DAG. When the transform needs a value, it gets it from the anim curve or motion path node.

So, when you connect the motion path node, the anim curve node gets disconnected, and so no longer affects the transform. You can only have one of these positional nodes connected to a transform at a time, and the last one connected wins.

Regardless of whether or not the transform is connect to an anim curve or motion path node, you can always ask the transform node for its transformation information and get the right values.

Q:

I haven't been able to find any documentation or sample code describing the interpolation system used by Maya for the quaternion curves. Could you tell me where I can find the details on it?

A:

You can find the code for the spherical linear interpolation (called "slerp") in the book *Graphic Gems 3*. For the spline interpolation, we use squad which is described in the first two references below.

- Shoemake, K. "Animating Rotation with Quaternion Curves." SIGGRAPH 85 Proceedings, pp 245-254, 1985.
- Shoemake, K. "Quaternion Calculus for Animation." SIGGRAPH 91 course notes for "Math for SIGGRAPH".
- Eberly, David. "Quaternion Algebra and Calculus." (This paper contains a derivation for squad.)
- Ramamoorthi, Ravi. "Fast Construction of Accurate Quaternion Splines." SIGGRAPH 97 Proceedings, 1997. (This paper discusses an alternate interpolation method).

Windows Questions

Q:

How do I add Windows-specific code to my plug-in?

A:

Use `#ifdef _WIN32` around the Windows-specific code.

Q:

How do I debug my plug-in?

A:

Select Project > Settings, then select the Debug tab. In the Executable for debug session field, type the full pathname to the Maya executable, for example:

C:\Program Files\Alias\Maya6.0\bin\Maya.exe

Use the F9 function key to toggle breakpoints in your plug-in source code. When you are ready to begin debugging, select Build > Start Debug > Go.

Q:

How do I get the handle to the application instance (the HINSTANCE) for my plug-in?

A:

We have saved the HINSTANCE for the plug-in in a global variable, MhInstPlugin, which should be available if you have included the standard set of plug-in include files. Specifically, the variable is defined in the MfnPlugin.h include file.

Q:

What do I do if I get the following warning?

```
warning C4190: 'initializePlugin' has C-linkage specified,
but returns UDT 'MStatus' which is incompatible with C
```

A:

Nothing. The compiler will complain about this, but it will do the right thing. The warning is harmless.

Q:

Why do I get compiler errors when I use the variables “near” and “far”?

A:

These are reserved keywords in the Microsoft compiler. You will need to change the variable names to, for example, nearClip and farClip.

Q:

What do I do if I get the following error?

```
error C2065: 'uint' : undeclared identifier
```

A:

The Windows equivalent for this is UINT. We have added a define to MTypes.h to solve this problem.

Q:

What do I do if I get the following error?

```
error C2065: 'alloca' : undeclared identifier
```

A:

Add the following to your source code.

```
#ifdef _WIN32
#include "malloc.h"
#endif
```

A Example Plug-ins

Developer

Plug-in API

Example plug-ins

Overview of example plug-ins

There are a large number of example plug-ins supplied with the Maya Development Tool Kit. These are described in this chapter to help you find one that demonstrates the operation you are trying to accomplish. The following naming convention is used by the examples so you can tell what kind of plug-in the example is based on its file name.

Suffix	Description
Cmd	Plug-ins that create new commands.
Tool	Plug-ins that create new interactive tools.
Node	Plug-ins that create new node types.
Translator	Plug-ins that create new file translators.
Shader	Plug-ins that create new shading nodes.
Device	Plug-ins that create new devices. Currently only MIDI input devices are support, and those are only supported on the IRIX platform.
Manip	Plug-ins that create new manipulators.
Field	Plug-ins that create new dynamic fields.
Emitter	Plug-ins that create new dynamic emitters.
Spring	Plug-ins that create new dynamic springs.
Shape	Plug-ins that create new shapes.

MEL command plug-ins

blastCmd	This example plug-in has been improved to demonstrate the use of the new off-screen rendering capabilities on the Linux platform.
blindComplexDataCmd	command which demonstrates adding more complex blind data, as a user defined data type, to an object
blindDoubleDataCmd	command which demonstrates adding blind data, as a user-defined data type, to an object
blindShortDataCmd	command which demonstrates adding blind data to an object
closestPointOnCurve	command that sets the weights of the CVs of a cluster according to a mathematical function
closestPointOnMesh	both a MEL command and a DG node that computes the closest point on a mesh from a worldspace position
convertBumpCmd	command that demonstrates the steps necessary to create a non-linear animation clip using the API
convertEdgesToContainedFacesCmd	command that converts a selection of edges into a selection of faces that interconnect
convertVerticesToContainedEdgesCmd	command that converts a selection of vertices into a selection of edges that interconnect the original vertices
convertVerticesToContainedFacesCmd	command that converts a selection of vertices into a selection of faces that interconnet the original vertices
cvExpandCmd	command that splits NURBS CV selections up into one string per selected CV
cvPosCmd	return the world or local space position of a NURBS CV or a poly vertex.
dagPoseInfoCmd	command that demonstrates how to extract DAG pose info for a skeleton’s bind pose, or for other poses created using the “dagPose” command.

deletedMsgCmd	A new example that demonstrates each of the node deletion callbacks available.
deletedMsgCmd	command that demonstrates each of the node deletion callbacks that are available in the API. The command registers callbacks on selected nodes that will trigger messages in the console when the command is run
dynExprFieldTest	A new example that demonstrates the per particle field attributes support that has been added to the class MPxFieldNode.
exportJointClusterDataCmd	command that demonstrates how to find all joint cluster nodes and uses the MFnWeightGeometryFilter function set and MItGeometry iterator to export weights per CV for each geometry deformed by each joint cluster.
exportSkinClusterDataCmd	command that exports smooth skin data to an alternate format
findFileTexturesCmd	locate the file texture nodes in a scene
findTexturesPerPolygonCmd	locate the file texture nodes assigned to each polygon
flipUVCmd	demonstration the use of the MPxPolyTweakUVCommand class to manipulate UVs
helix2Cmd	command which implements undo and redo
helixTool	tool which uses OpenGL to draw out guidelines
helloCmd	trivial command that takes arguments
helloWorldCmd	first simple command
idleTest	display the attribute dependencies within a node
instanceCallbackCmd	A new example that demonstrates listening to instance add and remove messages.
listLightLinksCmd	command to query light linking information

listPolyHolesCmd	command that produces a list of all the holes in each selected polymesh
lockEvent	demonstrates the API callbacks for node and plug locking
marqueeTool	command which implements mouse selection
meshOpCmd	demonstrates the use of the high-level polygon API methods that have been added to MFnMesh
motionPathCmd	command to animate an object along a motion path
motionTraceCmd	command that evaluates the position of a keyframed object over time and draws a motion path curve
moveCurveCVsCmd	command that moves CVs to the origin
nodeInfoCmd	command which demonstrates walking the dependency graph
nodeMessageCmd	command that adds a callback for all the nodes on the active selection list
particlePathsCmd	command that uses particle ID information from the API to derive a set of NURBS curves from the position of particles over time
particleSystemInfoCmd	demonstrates the use of the new MFnParticleSystem class for retrieving particle information
pickCmd	command to pick objects by name
pointOnMeshInfo	both a MEL command and a DG node that computes the worldspace position and normal on a poly mesh
polyMessageCmd	A new example that demonstrates the use of the MPolyMessage class to listen to vertex, edge and face component id changes.
polyPrimitiveCmd	command to create polygons
progressWindowCmd	A new example that demonstrates the use of the MProgressWindow class.
referenceQueryCmd	command to find useful information about the referenced files in a scene

renderViewInteractiveRenderCmd	A new example that demonstrates the immediate feedback setting that has been added to the startRender() methods of the MRenderView class.
scanDagCmd	command which demonstrates walking the DAG
scanDagSyntax	command which demonstrates walking the DAG as well as using syntax objects to parse the arguments to the command
spiralAnimCurveCmd	command to move objects in a spiral
splitUVCmd	command to unshare or “split” select UVs on a poly mesh
surfaceCreateCmd	command that creates a NURBS surface from CVs and knots using the MFnNurbsSurface function set
surfaceTwistCmd	command which modifies the CV positions of NURBS surfaces or the vertex positions of polygons in order to twist the surface around the y-axis
translateCmd	command to translate objects
undoRedoMsgCmd	A new example that demonstrates the use of the new Undo and Redo events that have been added to the MEventMessage class.
userMsgCmd	A new example that demonstrates the use of the MUserEventMessage class. This example allows the creation, removal and posting of user-defined events identified by strings.
userMsgCmd	command that demonstrates how user-defined messages can be used. The command supports options to register, deregister, and post named events through the API. This example uses callbacks that simply print a message when they are entered.
volumeLightCmd	demonstrates the use of the MFnVolumeLight class
whatisCmd	command that prints API type information about objects
zoomCameraCmd	command to zoom the view through a camera

Dependency Graph Node Plug-ins

affectsNode	a new example that demonstrates the use of the MPxNode::setDependentsDirty() method. This new method allows for attributeAffects() relationships involving dynamic and non-dynamic (i.e. static) Maya attributes.
animCubeNode	an example of a dependency node that creates a polygonal mesh from scratch and outputs that mesh in a dependency graph attribute.
apiMeshShape	an example of a shape node that registers a new kind of polygonal mesh, as well as geometry data specific for the shape and a node that can create this new shape type
arcLenNode	a simple example of a node that takes geometry as input
buildRotationNode	a simple node which performs an algebraic computation
circleNode	a more complex procedural animation example
closestPointOnCurve	command that sets the weights of the CVs of a cluster according to a mathematical function
closestPointOnMesh	both a MEL command and a DG node that computes the closest point on a mesh from a worldspace position
componentScaleManip	a complex example that demonstrates how to use conversion functions with a scale manipulator to control vertex positions.
cvColorNode	an example of a locator dependency node that draws colored points on top of each CV of a NURBS surface.
cvColorShader	allows vertex color(CPV) to be software rendered
footPrintManip	an example of a locator dependency node that has a corresponding manipulator. The Show Manip Tool can be used when the footPrint locator is selected to show the footPrint manipulator.
footPrintNode	an example of a locator dependency node. Locators are actually DAG nodes that have draw methods that the user may override. This particular locator draws a foot print.
fullLoftNode	an example of a real loft dependency node that builds a NURBS surface from an array of NURBS curves.

jitterNode	a simple multi-purpose procedural node
latticeNoise	a complex example of geometry modification in a dependency node, along with a command that does low level dependency graph access to hook up the node
multiCurveNode	an example of a dependency node that uses the MArrayDataBuilder class.
NodeMonitor	class that monitors a given node
offsetNode	an example of a deformer dependency node that offsets vertices according to the CV's weights.
pnTrianglesNode	ATI Radeon specific hardware shader plug-in
pointOnMeshInfo	both a MEL command and a DG node that computes the worldspace position and normal on a poly mesh
pointOnSubdNode	an example of how to query a subdivision surface as an input to a dependency node
polyTrgNode	an example of how to add user defined triangulation for meshes using the poly API class, MPxPolyTrg
quadricShape	an example of a simple shape node that implements a quadric shape using the OpenGL gluQuadric functions.
rotateManip	an example that demonstrates the different settings for the rotate manipulator.
shellNode	a dependency graph node that procedurally generates sea shells and outputs them as meshes.
simpleLoftNode	a node that implements a user-defined loft function on a curve with construction history. Also demonstrates how to pass geometry (a NURBS surface) to an internal dependency node.
sineNode	a simple procedural animation example
surfaceBumpManip	this example demonstrates how to use the pointOnSurface manipulator to modify vertices near the manipulator position on the surface
swissArmyManip	a contrived example that attaches all existing user-defined manipulators to a node.

transCircleNode	an example of a dependency node that uses the translate attribute as both an input and output. Demonstrates how to transfer all of X, Y, and Z attributes via a single dependency graph connections. It also contains an example attribute editor template.
yTwistNode	an example of a deformer dependency node that twists the deformed vertices around the y-axis.

User-defined dependency graph nodes—creating dynamics nodes

These user-defined dependency graph nodes create dynamics nodes derived from MPxEmitterNode, MPxSpringNode, and MPxFieldNode.

ownerEmitter	an example particle emitter node that emits in a direction from multiple points defined by a particle shape.
simpleEmitter	an example particle emitter node that emits in a direction from a single point.
simpleSpring	an example spring node that defines the spring law that is used in a simulation.
sweptEmitter	an example particle emitter node that emits in a direction from points on a curve or surface.
torusField	an example field node that implements an attract-repel field between itself and a distance.

Rendering plug-ins

blastCmd	demonstrates how to use the off screen rendering API extension.
blindDataMesh	demonstrates the use of blind data to provide color information to a hardware shading node.
CgFx Shader	a hardware shader plug-in with many advanced possibilities.
renderAccessNode	demonstrates how to work with render callbacks.

renderViewRenderCmd	demonstrates how to render a full image to the Render View window using the MRenderView class.
renderViewRenderRegion Cmd	demonstrates how to use the MRenderView class to update the currently selected Render Region in Maya's Render View.
sampleCmd	demonstrates how to sample shading groups or nodes using MRenderUtil::sampleShadingNetwork().
sampleParticles	demonstrates how to sample shading groups or nodes using MRenderUtil::sampleShadingNetwork() to assign colors to a particle object.
ShadingConnection	a class that stores useful information about a shader's attribute, including what's connected upstream of it.
ShapeMonitor	a class that watches shape or texture nodes and keeps track of changes since the last export.
shiftNode	demonstrate modifying uvCoord and refPointCamera from within a plug-in texture

Miscellaneous plug-ins

The following lists several miscellaneous plug-ins.

conditionTest	display which "conditions" are being changed inside Maya
eventTest	display which "events" are being changed inside Maya
helixMotifCmd	create a new Motif window containing a button that creates a helix when pressed. This plug-in is only available on IRIX and Linux.
idleTest	using both idle messages and UI deleted messages in a plug-in
iffInfoCmd	extracts information from an IFF image file.
iffPixelCmd	extracts a pixel value from an IFF image file.
iffPpmCmd	converts an IFF image file to a PPM image file.
jlcVcrDevice	creates a user defined midi input device for the JL-Cooper midi VCR control box. This plug-in is only available on IRIX and Linux.
lepTranslator	an example file translator that defines its own magic number and reads MEL commands

maTranslator	an example of a file translator that approximates the MayaAscii file
moveNumericTool	Selection-action tool that performs translations in orthographic views as well as allowing the user to type in precise translation values to while in the move tool.
moveTool	Selection-action tool that performs translations in orthographic views.
moveManip	A simple manipulator that demonstrates how the freePointTriad and distance manipulators can be used with an user-defined context.
pnTrianglesNode	This node is a simple example of how to query a subdivision surface as an input to a dependency node.
simpleSolverNode	an example of a simple user defined single-bone IK-solver in the x-y plane that is registered through createNode. By registering the plug-in solver through createNode, the registration mechanism is the same as non-default IK solvers such as the ikMCsolver.
viewCaptureCmd	uses OpenGL to capture the current 3D view and write it into a PPM file

Shader source code examples

Shader source code examples are provided in “Shader source code examples”.

System plug-ins

The following table lists the source for several of the system plug-ins shipped with Maya. Unlike most of the other examples, these are complex samples of real production plug-ins.

Detailed documentation for each of these can be found in the *Translators* guide as compiled versions of these are shipped as part of the Maya package. The following shows the current list.

animImportExport	A translator that allows you to move animation information between scenes.
objExport	A Wavefront OBJ export translator
ribExport	A Renderman RIB export translator

Example stand-alone applications

Stand-alone applications are those that contain the main routine for the application and make API calls to access Maya in batch mode. They are compiled in a slightly different manner than plug-ins (as covered in ”Using a debugger to debug your plug-ins” on page 214), but utilize the same API as plug-ins, and in the same way.

Note (IRIX and Linux) It is important to note that in order to run one of these applications it is necessary to set the environment variables MAYA_LOCATION and LD_LIBRARY_PATH. The former should be set to “/usr/aw/maya” (or the alternative location into which Maya was installed), and the later should be set to “$MAYA_LOCATION/lib”.

If these variables are not set, you will get errors when you attempt to start a stand-alone application.

The following sample applications are provided to demonstrate how to create and use stand-alone applications.

helloWorld	the required hello world example.
surfaceCreate	the surfaceCreateCmd plug-in converted to a stand-alone application.
surfaceTwist	the surfaceTwistCmd plug-in converted to a stand-alone application.
readAndWrite	an application that reads in a Maya scene file and writes it out again in different file. This application is quite useful for upgrading Maya scene files from a previous version.
asciiToBinary	an application that reads in a Maya scene file in ascii format and write is out again as a binary file.

Example plug-in descriptions

Below are brief descriptions and usage instructions for the plug-ins provided.

affectsNode

This plug-in creates a node called "affects". Add two dynamic attributes called "A" and "B". When you change the value on A, note that B will recompute.

The following sequence of commands will demonstrate how to use this plug-in:

```
//  Create an "affects" node by typing the MEL command:
createNode affects;
// Add two integer dynamic attributes to the newly created  affects node by
typing the MEL command:
addAttr -ln A -at long  affects1;
addAttr -ln B -at long  affects1;
// Change the value of "A" to 10 by typing the MEL command:
setAttr affects1.A 10;
// At this point, the affectsNode::setDependentsDirty() method gets called
which causes "B" to be marked dirty.
// Compute the value on "B" by doing a getAttr:
getAttr affects1.B;
// The affectsNode::compute() method is entered which copies the value from "A"
(i.e. 10) to "B".
```

arcLenNode

Produces dependency graph node arcLen

This node is a simple example of how to take geometry as an input to a dependency node. This node takes a NURBS curve as an input and outputs its arc length, using the MFnNurbsCurve function to perform the calculation.

The input for this node is a NURBS curve attribute called “inputCurve”. NURBS curve shapes nodes have two compatible output attributes that you can use as inputs for the arcLen node - “local” and “worldSpace”.

The output attribute of the arcLen node is just called “output”. It is a double value that represents the total length of the curve with an epsilon value of 0.001.

The following MEL code shows how to hook up the node to a curve.

```
createNode -n arcLen1 arcLen;
connectAttr curveShape1.local arcLen1.inputCurve;
```

From here, you can connect the “output” attribute to whatever you wish to drive, or just read it using the following command:

```
getAttr arcLen1.output;
```

This node is provided as an example of how to take geometry as an input. It should be noted that there is already a node in Maya that performs the same service called “curveInfo”.

animCubeNode

Produces dependency graph node animCube

This plug-in demonstrates how to take time as an input, and create polygonal geometry for output. The compute method of the node constructs a polygonal cube whose size depends on the current frame number. The resulting mesh is passed to an internal Maya node which displays it and allows it to be positioned.

To use this node, execute the MEL command "animCubeNode.mel" that contains the following commands:

```
createNode transform -n animCube1;
createNode mesh -n animCubeShape1 -p animCube1;
createNode animCube -n animCubeNode1;
connectAttr time1.outTime animCubeNode1.time;
connectAttr animCubeNode1.outputMesh animCubeShape1.inMesh;
```

This creates a *mesh* node under a *transform* node which is hooked into the world for display. It then creates an *animCube* node, and connects its input to the *time* node, and its output to the *mesh* node.

A cube will now appear on the screen. If the play button on the time slider is pressed, the displayed cube will grow and shrink as the frame number changes.

apiMeshShape

Produces shape node apiMesh, *dependency graph node* apiMeshCreator, *and data type* apiMeshData

This plug-in demonstrates how to create a polygonal mesh shape which has vertices that can be selected, moved, animated, and deformed. This shape also supports OpenGL display of materials.

This plug-in also registers a new kind of geometry data, apiMeshData, and demonstrates how to pass this data between nodes.

The apiMeshCreator node can create two types of apiMeshData, a cube and a sphere. The "shapeType" attribute is used to specify the type of shape to create. This node also takes normal mesh data as an input and converts it to apiMeshData. If there is no input mesh then the output is based on the shapeType attribute.

To create an apiMesh shape, you must first create the apiMesh node, then create an apiMeshCreator node and connect the two nodes as follows:

```
createNode apiMesh -n m1;
createNode apiMeshCreator -n c1;
connectAttr c1.outputSurface m1.inputSurface;
```

blastCmd

Produces MEL command blast

This plug-in adds a "blast" command to Maya. This command exercises the API code to allow Maya to draw to an OpenGL off-screen buffer. This command will draw the refresh the current viewport either on or off screen and then grab a snapshot of the pixels and store them on disk.

This example plug-in only works on SGI IRIX hardware. However, it is possible for a plug-in to create its own off-screen OpenGL buffer on different hardware and have Maya draw into it. The command takes two flags:

-f filenamet	Specifies the output file name. If it is not specified, then it defaults to blast.rgb.
-o	On-screen operation. Refresh and grab the pixels from the on-screen buffer rather than using an off-screen buffer. If part of the on-screen viewport is obscured, that part may not be correct in the output file. The off-screen buffer is never obscured.

blindComplexDataCmd

Produces MEL command blindComplexData *and user defined data type* blindComplexData

This plug-in demonstrates how to create blind data (dynamic attributes) based on user defined data types. The plug-in uses an array of structures in which each element contains both a double and an int as the user data type.

The use of the MPlug class to set and retrieve the value of the attribute is demonstrated, as are read and write routines that implement the storage and retrieval of the data in both Maya ASCII and Maya Binary file formats.

To use this plug-in, select a dependency node, and then issue the command *blindComplexData*. A dynamic attribute containing a five element array will be attached to each selected dependency node. If the scene is saved in Maya ASCII format, you will be able to see the MEL commands that save the value of the dynamic attribute. If the scene is reloaded, the dynamic attribute will be reattached to the applicable nodes.

blindDataMesh

Produces two dependency graph nodes: blindDataShader *and* blindDataMesh

This plug-in demonstrates the use of blind data to provide color information to a hardware shading node. It contains two parts, blindDataMesh and blindDataShader.

The blindDataMesh node builds a mesh and populates its blind data with color information. The blindDataShader node is a hardware shader which picks up the color information for drawing. The shader part of the plug-in picks uses the vertex IDs of the MPxHwShaderNode::geometry() method to acquire the blind data color information.

To use this plug-in: Open the blindDataShader.mel file and execute its contents. The results can be viewed by using the shading menu to smooth shade and then turn on hardware texturing.

blindDoubleDataCmd

Produces MEL command blindDoubleData *and user defined data type* blindDoubleData

This plug-in demonstrates how to create blind data (dynamic attributes) based on user defined data types. The plug-in uses a simple double value as the user data type. The use of the MPlug class to set and retrieve the value of the attribute is demonstrated, as are read and write routines that implement the storage and retrieval of the data in both Maya ASCII and Maya Binary file formats.

To use this plug-in, select a dependency node, and then issue the command *blindDoubleData*. A dynamic attribute containing the double value, 3.2, will be attached to each selected dependency node. If the scene is saved in Maya ASCII format, you will be able to see the MEL commands that save the value of the dynamic attribute. If the scene is reloaded, the dynamic attribute will be reattached to the applicable nodes.

blindShortDataCmd

Produces MEL command blindShortData

This example adds a dynamic attribute to the dependency nodes of each of the currently selected items. The attribute is a short and its default value is set to 99. To use this plug-in, select an object, bring up the Attribute Editor (select Window > Attributes), and then click on the “Extras” tab. The attribute editor should display no extra attributes. Then execute *blindShortData* in the command window. An attribute will appear in the editor set to the value of 99. The value can be modified in the editor, or with the MEL commands setAttr and getAttr. When the scene is saved the new attribute and value for each selected item are also saved. This example demonstrates how simple blind data can be attached to an object.

buildRotationNode

Produces dependency graph node buildRotation

This example demonstrates performing a linear algebra calculation based on inputs and outputting the result.

The node takes an up vector and a forward vector as inputs and outputs the rotation that takes the y-axis and the z-axis and rotates them to be the up and forward vectors respectively.

A sample use of this node is to align an object to another surface based on a normal and a tangent vector. This example uses the pointOnSurfaceNode which is a standard node provided with Maya.

```
sphere -radius 4;
cone -ax 0 1 0;
createNode pointOnSurfaceInfo;
connectAttr nurbsSphereShape1.worldSpace pointOnSurfaceInfo1.inputSurface;
connectAttr pointOnSurfaceInfo1.position nurbsCone1.translate;
setAttr pointOnSurfaceInfo1.u 0.01;
setAttr pointOnSurfaceInfo1.v 0.01;
```

At this point, the cone will be constrained to a point on the surface of the sphere. As the u and v attributes of the pointOnSurfaceNode are changed, the cone will move around the surface. The next step is to align the cone so that the tip points in the direction of the normal. The buildRotation node will be used to do this.

```
createNode buildRotation;
connectAttr pointOnSurfaceInfo1.normal buildRotation1.up;
connectAttr pointOnSurfaceInfo1.tangentU buildRotation1.forward;
connectAttr buildRotation1.rotate nurbsCone1.rotate;
```

Now the cone will be constrained to the surface of the sphere and will also be aligned with the normal of the sphere at the point of constraint.

CgFx Shader

Windows-only.

The plug-in is named cgfxShader.mll. It defines one node, named cgfxShader, and one command, also named cgfxShader. The command is used to manipulate the node.

This is very similar to the expression command and node.

The cgfxShader node is a hardware shader (derived from MPxHwShaderNode). This means that it has many limitations compared to normal shader nodes. Specifically, it only affects drawing in the main viewport (including playblast) and the hardware render buffer. Hardware shaders also only apply to polygonal objects. If you try to apply one to a NURBS or SubD surface, nothing interesting will happen.

By itself, the shader or s attribute is the only thing visible on a cgfxShader node. You set this attribute to the name of a .fx file and all sorts of interesting things happen. A .fx files hold CgFX effect definitions. This effect completely controls how the drawing is done. So if you load a different .fx file, you get a completely different effect on the screen. In theory, the cgfxShader node could do anything that any other hardware shader node could do.

Along with the effect, the CgFX file provides a set of parameters that can be modified to change the effect in controlled ways. For example, a glow effect may allow you to set the color, intensity, and size of the glow. A bumpy, shiny effect may allow you to change the bump texture map. All of these parameters are exposed as dynamic attributes on the shader node. If the effect has a glowColor parameter, the cgfxShader node would have a dynamic color attribute named glowColor. Changes to the attribute affect the parameter.

You specify the .fx file using the cgfxShader command. You cannot simply set the attribute because of all the changes that occur on the node when the .fx file is changed. All the dynamic attributes that no longer apply are removed and all the attributes that are needed by the new effect are added. You can see all the changes in the Attribute Editor.

The syntax of the command is:

```
cgfxShader [-e] [-fx fileName] [-n name] [nodeName]
```

-e

Edit an existing shader. If -e is not specified, a new cgfxShader node is created.

-fx filename

Set the .fx file for the cgfxShader node.

-n name

Sets the name of the shader node to create. (Create only).

nodeName

The name of the node to edit. You do not need to type this if the shader node is currently selected.

Tthe easiest way to use the shaders is via the Hypershade window. Load the plug-in before opening the HyperShade window and then simply invoke Create > Material > Cgfx Shader. You can then simply drag the material onto a shape to assign it.

Limitations

- HW shaders show up with black swatches in the Hypershade and the Attribute Editor.
- HW shaders only work on polygonal objects.

- The shaders do not save correctly. When they load from a file, the attributes are not connected back up correctly. You can save a file with cgfxShaders in them, you will just have to specify the .fx file name again when you load them.
- You have to type .fx file names by hand using the command: `cgfxShader -e -fx "file.fx" cgfxShader1;` . You could always put this command on a shelf button.
- You have to type texture file names by hand. There is no browser for entering this information.
- Currently, only NVIDIA .dds textures are supported.
- Normal maps are not yet supported.
- There appear to be some bugs in the CgFX code. There appears to be a memory bug that is causing Maya to crash on exit. It may crash on File > New.

circleNode

Produces dependency graph node circle

This plug-in is an example of a user-defined dependency graph node. It takes a number as input (such as time) and generates two output numbers one which describes a sine curve as the input varies and one that generates a cosine curve. If these two are hooked up to the x and z translation attributes of an object the object will describe move through a circle in the xz plane as time is changed.

Executing the command “source circleNode” will create a new “Circle” menu with a single item. Selecting this will build a simple model (a sphere which follows a circular path) which can be played back, by clicking on the “play” icon on the time slider.

The node has two additional attributes which can be changed to affect the animation, “scale” which defines the size of the circular path, and “frames” which defines the number of frames required for a complete circuit of the path. Either of these can be hooked up to other nodes, or can be simply set via the MEL command “setAttr” operating on the circle node “circleNode1” created by the MEL script. For example, “setAttr circleNode1.scale #” will change the size of the circle and “setAttr circleNode1.frames #” will cause objects to complete a circle in indicated number of frames.

closestPointOnCurve

Computes closestPointOnCurve (*NURBS curve) from the input position*

This plug-in defines both a MEL command and a DG node which takes in as input, a NURBS curve and a worldspace position, then computes the closest point on the input curve from the input position.

In addition to the worldspace "position" at the closest point on the curve, also returned are the "normal", "tangent", "U-parameter" and "closest distance from the input position", at the closest point on the curve.

closestPointOnMesh

Computes closestPointOnMesh *from the input position*

This plug-in defines both a MEL command and a DG node which takes in as input, a mesh and a worldspace position, then computes the closest point on the input mesh from the input position.

In addition to the worldspace "position" at the closest point on the mesh, also returned are the "normal", "U-parameter", "V-parameter" and the "closest face index" at the closest point on the mesh.

clusterWeightFunction

Produces Mel command clusterWeightFunction

This example demonstrates how to use the MFnWeightGeometryFilter::setWeight method to set the weights of the CVs of a cluster according to a mathematical function.

Note A call to MFnWeightGeometryFilter::setWeight cannot be made inside the geometry iterator as this would result in a fatal error.

To use this plug-in, select the cluster and a geometry affected by the cluster.

For example:

```
select cluster1 nurbsSphereShape1;
```

Then execute the MEL command clusterWeightFunction with the flag to specify which mathematical function to use:

For example:

```
clusterWeightFunction -sine;
```

componentScaleManip

The componentScaleManip is a manipulator plug-in that demonstrates how components can be manipulated using a scale manipulator using the manipulator API. This example produces the MEL command componentScaleContext to create the tool context for the component scale manipulator.

To create the tool button for the plug-in, create a new shelf named "Shelf1" and execute the following MEL commands to create the tool button in this shelf:

```
componentScaleContext;
setParent Shelf1;
toolButton -cl toolCluster -t componentScaleContext1 -i1 "moveManip.xpm";
```

To use the manipulator, create a nurbs surface and select some CVs. Then click on the tool button created above to activate the plug-in context. The tool should work much like the built-in scale tool, except that the plug-in will only operate on NURBS CVs.

conditionTest

Registers conditions callbacks

The conditionTest plug-in is a simple plug-in that displays which “conditions” are being changed inside Maya. A condition in Maya is something of interest to the Maya internals or plug-ins that has a true or false value. For example, “SomethingSelected” and “playingBack” are two available conditions. These conditions can be tracked at the MEL level with the -conditionTrue, -conditionFalse, or -conditionChanged flags to the scriptJob command.

The plug-in adds a “conditionTest” command that lets you see which conditions are available, track specific conditions, and to see which conditions are being tracked. The basic command syntax is:

```
conditionTest [-m on|off] [conditionName ...]
```

The conditionName arguments should be the names of conditions. If no names are specified, the command will operate on all available conditions.

If you use the -m flag, you can specify whether the plug-in should track messages for the specified conditions or not. If you specify the -m flag without any condition names, it will turn on or off tracking for all conditions.

Example:

```
mel: conditionTest -m 1 SomethingSelected
Condition Name        State  Msgs On
--------------------  -----  -------
SomethingSelected     false  yes
```

After this example, the plug-in will display the line:

```
condition SomethingSelected changed to true
```

or

```
condition SomethingSelected changed to false
```

every time the selection list becomes empty or not empty. You can disable all condition tracking with the command: conditionTest -m off;

convertBumpCmd

Produces MEL command convertBump

This plug-in command can be used to convert a bump file texture from a grey-scale height field format (used by Maya) to a normal map format that is typically used for real-time, hardware-based rendering.

This code also demonstrates how to use the MImage class to load, manipulate and save image files on disk.

Note	The MImage class is new as of Maya 4.0.1, but that the saving and filtering capability was only introduced in Maya 4.5.

To test this plug-in, first compile and load it using the plug-in manager, then type the following in the script editor:

```
convertBump "C:/bump.tga" "C:/bump_norm.tga" "tga" 1.0
```

This would convert the input texture (c:/bump.tga) into an output normal map (c:/bump_norm.tga) using the default bumpScale ratio. The bumpScale parameter can be used to increase or decrease the bumpiness of the resulting normal map.

See the documentation of *MImage::saveToFile()* for a complete list of the available file formats supported.

convertEdgesToContainedFacesCmd

Produces MEL command convertEdgesToContainedFaces

This plug-in creates a MEL command that converts a selection of edges into a selection of faces that interconnect the original edges (that is, only faces whose composite edges are contained in the original edge selection).

The command's return value is a string array that contains the names of all of the new contained faces. This MEL command has no flags, returns a string array, and operates on selected edges.

Example MEL usage:

```
select -r pCube1.e[1:2] pCube1.e[6:7];
string $convertedFaces[] = `convertEdgesToContainedFaces`;
// Result: pCube1.f[1] //
```

convertVerticesToContainedEdgesCmd

Produces MEL command convertVerticesToContainedEdges

This plug-in creates a MEL command that converts a selection of vertices into a selection of edges that interconnect the original vertices (that is, only edges whose composite vertices are contained in the original vertex selection). The command's return value is a string array that contains the names of all of the new contained edges.

This MEL command has no flags, returns a string array, and operates on selected vertices.

Example MEL usage:

```
select -r pCube1.vtx[2:5];
string $convertedEdges[] = `convertVerticesToContainedEdges`;
// Result: pCube1.e[1:2] pCube1.e[6:7] //
```

convertVerticesToContainedFacesCmd

Produces MEL command convertVerticesToContainedFaces

This plug-in creates a MEL command that converts a selection of vertices into a selection of faces that interconnect the original vertices (that is, only faces whose composite vertices are contained in the original vertex selection). The command's return value is a string array that contains the names of all of the new contained faces.

This MEL command has no flags, returns a string array, and operates on selected vertices.

Example MEL usage:

```
select -r pCube1.vtx[0:5];
string $convertedFaces[] = `convertVerticesToContainedFaces`;
// Result: pCube1.f[0:1] //
```

createClipCmd

Produces MEL command createClip

This example demonstrates the steps used to create a nonlinear animation clip using the API. The clip will be created either on the selected items, or on a character set node that is supplied using the -c/-char flag. The plug-in creates the animation curves that make up the clip, places them in a source clip, then instances the source clip twice.

To use this plug-in, create a sphere and select it. Then execute the command:

```
createClip
```

Open the Trax Editor to see the clips in place on the timeline.

customAttrManip

This plug-in demonstrates how to create user-defined manipulators from a user-defined context.

This is the script for running this plug-in:

```
source "customAttrManip.mel";
sphere;
move 5 0 0;
cone;
move -5 0 0;
select -cl;
```

Now click on the customAttrManip on Shelf1!

cvColorNode

Produces dependency graph node cvColor

This example provides an example of how to color the CVs of a NURBS surface by drawing OpenGL points on top of them. This node implements a locator that is intended to be made a sibling of a NURBS surface shape, and sit under the same transform. If the “local” space output attribute of the shape is connected to the “inputSurface” attributed of the locator node, then the later will draw colored points at each CV location. The current algorithm in the node will color the CVs in one of four different colors based on their XY location:

```
x < 0 && y < 0: Red
x < 0 && y >= 0: Cyan
x >= 0 && y < 0: Blue
x >= 0 && y >= 0: Yellow
```

To use this plug-in, first load it then execute the command “source cvColorNode” to define the MEL command *attachColorNode*. This command iterates across selected objects, and attaches a cvColor node, as described above, to each NURBS surface it encounters. Moving the objects, or its CVs, after the node is attached will cause the colors of the CVs to change. The *pointSize* attribute of the node controls the size of the point that is drawn. The *drawingEnabled* attribute, if set to false, will disable the display of the colored points.

cvColorShader

Produces dependency graph node cvColorShader

This plug-in creates a node that allows vertex color(CPV) to be software rendered. Once the plug-in is loaded, the node will appear as a Color Utility in the Hypershade window. Connect this node to a shader's Color or Incandescence attribute.

cvExpandCmd

Produces MEL command cvExpand

This example demonstrates how to handle selection lists and return the contents in a string form that the scripting language will understand. The *cvExpand* command goes through the current selection list and splits ranges of CVs that are selected into individual strings for each CV, so if the selection list looked like this:

```
ls -selection;
// Result: curveShape1.cv[1:3] //
```

Then the *cvExpand* command will return this instead:

```
cvExpand;
// Result: curveShape1.cv[1] curveShape1.cv[2] curveShape1.cv[3] //
```

cvPosCmd

Produces MEL command cvPos

This example demonstrates how to obtain the world or local space position of a NURBS CV or a polygonal vertex.

The command accepts the flags *-l/-local* or *-w/-world*, where *world* is the default, to indicate whether a local or world space location of the CV is required. The command can handle at most 1 CV per invocation since it returns the coordinate as 3 element a MEL float array. If no arguments are provided to the command, it checks the active selection list. If exactly one CV or vertex is present in that list, it returns its location. If more than one is selected, an error is produced. Alternatively, a single component can be specified on the command line using MEL syntax. For example:

```
cvPos nurbsSphereShape1.cv[0][0];
```

will return the world space position of the specified CV.

dagPoseInfoCmd

Produces MEL command dagPoseInfo

This example demonstrates how to extract DAG pose info for a skeleton's bind pose, or for other poses created using the "dagPose" command. It demonstrates using an MItSelectionList iterator to determine the selected joints, and using the MPlug class to traverse plug connections and get matrix data from the graph.

To use this plug-in, build a skeleton and bind geometry using either the "Smooth" or "Rigid" Bind feature. Then select the joints for which you want to find the bindPose, and execute the command:

```
dagPoseInfo -f filename;
```

Please refer to the example code that describes the output format.

deletedMsgCmd

This plug-in demonstrates each of the node deletion callbacks available in Maya API. This plug-in can be used to distinguish when each type of callback should be used. Callbacks are added to nodes by invoking the command:

```
deletedMessage <node 1> [<node 2> ...]
```

This command will register 3 callbacks on the nodes.

1) MNodeMessage::addNodeAboutToDeleteCallback is used to register an about to delete callback on the nodes.

2) MNodeMessage::addNodePreRemovalCallback is used to register a pre-removal callback on the nodes.

3) MDGMessage::addNodeRemovedCallback is used to register a callback that called when the node is removed.

For example, to register the callbacks for a nurbs sphere:

```
sphere;
deletedMessage nurbsSphere1;
```

Now delete the sphere:

```
delete nurbsSphere1;
// Removal callback node: makeNurbSphere1
// Removal callback node: nurbsSphereShape1
// About to delete callback for node: nurbsSphere1
// Pre-removal callback for node: nurbsSphere1
// Removal callback node: nurbsSphere1
```

dynExprFieldTest

This plug-in implements a dynExprField node for a uniform field. This allows the per particle attributes to drive the field's attributes.

To run the plug-in, do the following:

```
source dynExprFieldTest.mel
dynExprFieldTest;
```

eventTest

Produces MEL command eventTest

The eventTest plug-in is a simple plug-in that displays which “events” are occurring inside Maya. An event in Maya is something of interest to the Maya internals or plug-ins that occurs at a specific point in time. For example, “SelectionChanged” and “timeChanged” are two available events. These events can be tracked at the MEL level with the -event flag to the scriptJob command.

The plug-in adds an “eventTest” command that lets you see which events are available, track specific events, and to see which events are being tracked. The basic command syntax is:

```
eventTest [-m on|off] [eventName ...]
```

The eventName arguments should be the names of events. If no names are specified, the command will operate on all available events.

If you use the -m flag, you can specify whether the plug-in should track messages for the specified events or not. If you specify the -m flag without any event names, it will turn on or off tracking for all events.

Example:

```
mel: eventTest -m 1 timeChanged
Event Name            Msgs On
--------------------  -------
timeChanged           yes
```

After this example, the plug-in will display the line:

```
event timeChanged occurred
```

every time the current time changes. You can disable all event tracking with the command: eventTest -m off;

exportJointClusterDataCmd

Produces MEL command exportJointClusterData

This example demonstrates how to find all joint cluster nodes and uses the MFnWeightGeometryFilter function set and MItGeometry iterator to export weights per CV for each geometry deformed by each joint cluster.

To use this plug-in, build a skeleton and bind geometry using the “Rigid Bind” feature. Then type the command:

```
exportJointClusterData -f filename;
```

For example:

```
exportJointClusterData -f "C:/temp/skinData"
```

The output format used is:

```
jointName <skinCount>
     skin_1 <weightCount>
     <skin_1_component_index1> <skin_1_wt1>
     <skin_1_component_index2> <skin_1_wt2>
     <skin_1_component_index3> <skin_1_wt3>
     ...
     skin_2 <weightCount>
     <skin_2_component_index1> <skin_2_wt1>
     <skin_2_component_index2> <skin_2_wt2>
     <skin_2_component_index3> <skin_2_wt3>
```

```
...
```

exportSkinClusterDataCmd

Produces MEL command exportSkinClusterData

This example demonstrates how to find all skin cluster nodes and uses the MFnSkinCluster function set and MItGeometry iterator to export weights per CV for all geometry that are bound as a skin to a skeleton.

To use this plug-in, build a skeleton and bind geometry using the "Smooth Bind" feature and execute the command:

```
exportSkinClusterData -f filename;
```

Please refer to the example code that describes the output format.

findFileTexturesCmd

Produces MEL command findFileTextures

This example demonstrates how to navigate the dependency graph both manually and using the DG iterator. The command searches the dependency graph looking for file texture nodes that are attached to the shading engine. This example also illustrates the use of filters when navigating the DG. As file texture nodes are found, information about their attributes is extracted and printed on standard error. More extensive documentation is in the source code.

To use this plug-in, load a scene that contains some texture information, and find a node (or nodes) that are being shaded by a file texture. Then execute "findFileTextures nodeName1 nodeName2 ..." to find the file textures.

findTexturesPerPolygonCmd

Produces MEL command findTexturesPerPolygon

This example is a variation of the findFileTexturesCmd example. When a file texture node is connected to the color attribute of a shader, the file name will be extracted and printed along with the polygonal index. More extensive documentation is in the source code.

To use this plug-in, apply a shader that has a file texture node applied to the color attribute on a selected object and execute the command "findTexturesPerPolygon".

flipUVCmd

Produces a command: flipUV

This is a simple plug-in for demonstration the use of the new MPxPolyTweakUVCommand class to manipulate UVs.

The syntax of the command is:

```
flipUV [-es on|off] [-fg on|off] [-h on|off]
```

Flags:

-es -extendToShell	on | off
-fg -flipGlobal	on | off
-h -horizontal	on | off

To use this plug-in:

- Select UVs from a mesh
- Invoke the flipUV command

Depending on the type of texture used, results may not show up in the modelling windows. To see the result, use the UV Editor.

footPrintManip

Produces dependency graph node footPrintLocator *and* footPrintLocatorManip

This example demonstrates how to use the Show Manip Tool with a user-defined manipulator. The user-defined manipulator corresponds to the foot print locator.

To use this plug-in, just type “createNode footPrintLocator” to create a foot print locator, select the foot print, and then click on the Show Manip Tool.

footPrintNode

Produces dependency graph node footPrint

This example demonstrates how to create a user-defined locator. A locator is a dag object that is drawn in the 3D views but that does not render. This example plug-in defines a new locator node that draws a foot print. The foot print can be selected and moved using the regular manipulators.

To use this plug-in, just type “createNode footPrint” to create a foot print locator.

fullLoftNode

Produces dependency graph node fullLoft

This plug-in demonstrates how to take an array of NURBS curves as input and produce a NURBS surface as output. The input curves must have the same number of knots, and both form and degree are ignored.

To use this command, draw two or more curves. Select the curves and execute the MEL command “source fullLoftNode.mel”. This creates the fullLoft node and an output surface, as well as several rebuildCurve nodes to provide a uniform number of CVs for the fullLoft node.

Note This is a simplified version of the internal Maya loft node.

getAttrAffectsCmd

Produces MEL command getAttrAffects

This command takes the name of a node as an argument. It then iterates over each attribute of the node and prints a list of the attributes that it affects and the ones that affect it. To use it issue the command "getAttrAffects nodeName", where nodeName is the name of the node whose attributes you want to display. If invoked with no arguments, getAttrAffects will display the attribute info regarding all selected nodes.

helixCmd

Produces MEL command helix

This is the helixCmd example from the documentation. It is a demonstration of building a simple command which does not have undo. The command accepts two arguments, "-r" to set the radius of the helix, and "-p" to set the pitch of the helix. So, to create a helix, execute the command "helix [-r #] [-p #]" in the command window.

A MEL script is also provided with this example which can be run by executing the command "source helixCmd".

This script creates a new "Plug-ins" menu under which the "helix" command can be found. Selecting this menu item will bring up a window that allows you to set the radius and pitch for the new helix. This is a good example of hooking up a command to the UI.

helix2Cmd

Produces MEL command helix2

This example takes a selected curve and turns it into a helix. That in and of itself isn't very interesting, but the plug-in implements undo and redo functions so that the change can be undone and then redone. This plug-in is then a simple example of implementing a command which supports do, undo, and redo. To use it, create a curve, then execute "helix2" in the command window. The curve will change into a helix. Select "Undo" from the "Edit" menu, and the change will be undone. Select "Redo" and it will be redone.

helixMotifCmd

Produces MEL command helixMotif

This plug-in is only available on IRIX and Linux.

This example demonstrates how to create a separate Motif window that uses the same application shell as Maya.

The new window contains a single Motif button widget labelled Create Helix that when pressed creates a helix exactly as is done in the getProjectedFacesCmd example. Maya still controls the X event loop, but arbitrary X widgets and callbacks can be registered, and they will be dispatched by Maya when their event occurs.

helixTool

Produces MEL commands helixToolCmd *and* helixToolContext

This example takes the helix example one large step forward by wrapping the command in a context. This allows you to drag out the region in which you want the helix drawn. To use it, you should first execute the command "source helixTool". This will create a new entry in the "Shelf1" tab of the tool shelf called "Helix Tool". Click on the new icon, then move the cursor into the perspective window and drag out a cylinder which defines the volume in which the helix will be generated. This plug-in is an excellent example of building a context around a command.

helloCmd

Produces MEL command hello

This is the "hello" example from the documentation. You simply type a "hello" in the command window and "Hello" will be output to the window from which you started Maya.

helloWorldCmd

Produces MEL command helloWorld

This is the first example from the documentation. When "helloWorld" is typed into the command window "Hello World" is output to the window from which you started Maya.

idleTest

Produces MEL command idleTest

The idleTest plug-in shows an example of using both the idle messages and UI deleted messages in a simple plug-in. These messages correspond to the -idleEvent and -uiDeleted flags for the scriptJob command.

When you load the plug-in, it adds the command "idleTest". To run it, type "idleTest n" where n is some positive number. Idle test will then create a window and start filling it with a list of prime numbers. The test will compute one new prime number for each idle message that it receives. You will notice that idle messages stop during playback or while dragging an object.

If you type a large enough number for n, you will also notice that the idle messages will use up *all* available CPU cycles. For this reason, plug-ins should generally cancel their requests for idle messages once they are no longer needed.

When you delete the window that idleTest has created, the plug-in will receive a “UI deleted” message and cancel any outstanding idle message callbacks.

iffInfoCmd

Produces MEL command iffInfo

This command takes as an argument the name of an IFF file. The file is opened and read and general information about the file is returned as a result. For example: “iffInfo sphere.iff”

iffPixelCmd

Produces MEL command iffPixel

This command takes as arguments the name of an IFF file and the x and y co-ordinates of a pixel in the image. It returns the r/g/b/a values at that pixel. For example: “iffPixel sphere.iff 100 210"

iffPpmCmd

Produces MEL command iffPpm

This command takes as arguments the names of an existing IFF file and the name of a PPM (portable pixmap) file that it should create. The IFF image is read and written out in PPM format to the second file.

For example: “iffPpm sphere.iff sphere.ppm”.

instanceCallbackCmd

This plug-in demonstrates the functionality of the MDagMessage class which allows the listening of instance related messages. The messages support listening to:

1) Instance Added for a specified node(and its instances)

2) Instance Removed for a specified node(and its instances)

3) Instance Added for any node

4) Instance Removed for any node

This plug-in works in the following manner:

i. Draws a circle,

ii. Gets its dagPath using an iterator

iii. Adds callback for instance added and removed for this circle.

The callback functions just displays a message indicating the invocation of the registered callback function.

To execute this plug-in, do the following:

```
instCallbackCmd;
```

jitterNode

Produces dependency graph node jitter

This plug-in is an example of a user-defined procedural dependency graph node. It is primarily oriented toward animation, but can be used to add *noise* in any connection between two float attributes. It takes a float value as input, adds a pseudo-random value to the input and outputs the *noisy* float value. For example, if the output of a parameter curve node is connected to the input of the jitter node and the output of the jitter node is connected to the translateX attribute of a surface, the motion of the surface will “jitter” parallel to the X axis.

The node has one other input, “time”. The output of the time slider node, “time1.outTime”, should be connected to the time attribute on the jitter node if the “jittered” animation is to be repeatable. The attribute “scale” can be used to increase or decrease the magnitude of the random offset applied to the input of the jitter node.

Once the plug-in is loaded, the MEL command:

```
jitter "jitter1" "someNode1.outFloat" "someNode2.inFloat";
```

will create the jitter node, jitter1, attach the attribute someNode1.outFloat to jitter1.input and attach jitter1.output to someNode2.inFloat. It also attaches the time slider output, time1.outTime to jitter1.time and jitter2.time.

Additionally, it creates two windows with sliders for adjusting the scale.

The jitter node can be easily demonstrated in conjunction with the circleNode plug-in. Load the circleNode and jitterNode plug-ins. Then execute the following commands:

```
source circleNode
source jitterNode
createSphereAndAttachCircleNode;
jitter "jitter1" "circleNode1.sineOutput"
"sphere1.translateX";
jitter "jitter2" "circleNode1.cosineOutput"
"sphere1.translateZ";
```

Clicking the “play” icon on the time slider will cause the sphere to move along a “jittered” circle. The amount of jitter can be varied by adjusting the scale sliders in the windows “jitter1 Scale Editor” and “jitter2 Scale Editor”.

jlcVcrDevice

Registers new midi input device jlcVcrDevice

This plug-in is only available on IRIX and Linux.

Loading this plug-in will register the JL-Cooper midi VCR input device with Maya as type *jlcVcrDevice*.

To use the device, execute the MEL script "jlcVcrDevice.mel". This will attach the buttons on the midi device to the animation playback commands in Maya as follows:

- The *play* button will start animation playback.
- The *stop* button will stop animation playback.
- The *rewind* button sets the current time to the start time in the range control.
- The *forward* button sets the current time to the end time in the range control.
- The *rec* button executes the setKeyframe command.
- The *scrub* wheel moves the time forward and back.

To get a list of all input devices currently registered in Maya, use the command *listInputDevices*.

latticeNoise

Produces dependency graph node latticeNoise *and MEL command* latticeNoise

This plug-in is an example of the following:

- how to have node attributes input and output geometry
- how to modify dependency graph connections using the API
- how to take Maya objects as arguments to a user defined MEL command

The latticeNoise command creates a new lattice deformer around the currently selected geometry, or around the objects specified on the command line. The command also inserts a latticeNoise node in between the lattice shape in the DAG and the node that performs the deformation.

The end effect of the latticeNoise command is that the objects inside the lattice deform with respect to the lattice, but they also wobble about randomly as noise is applied to the lattice points. The latticeNoise node has attributes for amplitude and frequency that control the amount of noise applied.

An example of using the command is:

```
latticeNoise nurbsSphereShape1 nurbsConeShape1;
```

lepTranslator

Adds the new file format Lep *to the file manipulation dialogs*

As soon as this plug-in is loaded, the new file format will be available in the “Open”, “Import, and “Export” dialogs.

The icon that is displayed in the file selection boxes is the one that is contained in the file lepTranslator.rgb that is also located in the example plug-in directory. Maya will find this icon as long as the path to the directory that contains it is included in FILE_ICON_PATH environment variable.

An “Lep” file is an ASCII file with a first line of “<LEP>”. The remainder of the file contains MEL commands that create one of the primitives: *nurbsSphere*, *nurbsCone* and *nurbsCylinder*, as well as *move* commands to position them.

When writing the file, only primitives of these three types will be created along with their positions in 3D space. The reader routine will actually handle more MEL commands than these, but only this limited set of types will be written.

As well, this example demonstrates how to utilize file options. When saving a file, if you click on the option box beside the File > Export menu item, a dialog will pop up that contains two radio boxes asking whether to “Write Positions”. The default is true, and if false is selected, then the move commands for primitives will not be written to the output file. This dialog is implemented by the MEL script, *lepTranslatorOpts.mel* which is also located in the plug-in directory.

An sample input file is supplied in the example plug-in directory as *lepTranslator.lep*.

listLightLinksCmd

Produces MEL command listLightLinks.

This example demonstrates how to use the MLightLinks class to query Maya's light linking information. The command takes no arguments.If the currently selected object is a light, then the command will select all objects illuminated by that light. If the currently selected object is a piece of geometry, then the command will select all the lights that illuminate that geometry.

listPolyHolesCmd

Produces MEL command listPolyHoles.

This example demonstrates how to use the MFnMesh::getPolyHoles() function to describe all the holes in a polygon mesh. The command takes no arguments. When invoked, the command outputs a description of any holes in the currently selected mesh to the Output Window (on Windows), or to the console (on IRIX and Linux platforms).

lockEvent

Produces MEL command lockEvent

This plug-in demonstrates the API callbacks for node and plug locking. These callbacks allow you to receive notification when the locked status of a plug or node is queried internally. The API programmer has the option, upon receipt of the callback, to override/(-o) the lock state of node or plug. This override is controlled via a decision variable passed into the callback function. The variable can hold two values

1. decision = true --> You want to accept the lock state and do whatever the internal default behavior is.

2. decision = false --> You want to deny the lock state and do the opposite of what Maya would usually do.

The flow of execution would be as follows:

1. Received a callback from Maya.

2. What kind of event is this?

3. Do I want to allow this event?

    4. Yes, I do not want to OVERRIDE this event. decision = true.

    4. No, I want to OVERRIDE this event. decision = false.

5. Return from callback.

Example usage:

```
sphere ;
// Watch the translateX plug on the sphere we just created
lockEvent -a 3 nurbsSphere1.translateX;
// Do not allow any changes to the plug.
lockEvent -o true;
// Now you can try changes nurbsSphere1.translateX 's value
// but you will not be allowed to do so.
//
setAttr "nurbsSphere1.translateX" 22;
```

maTranslator

Produces file translator Maya ASCII(via plug-in)

This plugin is an example of a file translator. Although this is not the actual code used by Maya when it creates files in MayaAscii format, it nonetheless produces a very close approximation of the same format. Close enough that Maya can load the resulting files as if they were MayaAscii.

Currently, the plugin does not support the following:

- Export Selection. The plugin will only export entire scenes.
- Referencing files into the default namespace, or using a renaming prefix. It only supports referencing files into a separate namespace.
- MEL reference files.
- Size hints for multi plugs.

To use this plug-in, load it and then invoke it through the Export All menu item.

marqueeTool

Produces MEL command marqueeToolContext

This is another context example, except that this example does not have an associated command. To use it, you must execute the command “source marqueeTool” in the command window. This will create a new entry in the “Shelf1” tab of the tool shelf called “Marquee Tool”. When this tool is active, you can select objects in the 3D windows in the same way that you would with the selection tool, that is it can be used for either click selection, drag selection. Both will also work with the shift key held down in the same manner as the selection tool.

meshOpCmd

Produces command meshOp. *Produces dependency node* meshOpNode.

Demonstrates the use of the new high level polygon API methods that have been added to MFnMesh.

Syntax: `meshOp $operationType`

$operationType is one of:

0 - Subdivide edge(s).

1 - Subdivide face(s).

2 - Extrude edge(s).

3 - Extrude face(s).

4 - Collapse edge(s).

5 - Collapse face(s).

6 - Duplicate face(s).

7 - Extract face(s).

8 - Split face(s).

Example usage:

- Select the appropriate component(edge, face)
- meshOp 2;

Note: this plug-in re-uses the following files from the splitUVCmd

- polyModifierNode.cpp
- polyModifierCmd.cpp
- polyModifierFty.cpp

motionPathCmd

Produces MEL command motionPath

This plug-in assigns a curve as a motion path to an object. To use it, create an object and draw a curve. Clear all selected items, then pick the object followed by the curve (order is important). Once both are selected, execute "motionPath" in the command window, then click the play button. The object will move along the curve. This is a simple example of how to use the motion path function set.

motionTraceCmd

Produces MEL command motionTrace

In order to use this plug-in you must first create an object and animate it by setting keyframes. An easy way to do this is to just create a primitive, then set a bunch of keyframes for it with the spiralAnimCurveCmd plug-in.

Once this is done, select the animated object and invoke "motionTrace". The object will move along its animated path under control of the plug-in and when the animation is complete, the plug-in will draw a curve into the scene that represents the motion path of the object.

The plug-in accepts *-s*, *-e*, and *-b* parameters to control the *startFrame*, *endFrame* and *byFrame* values that it uses in running the animation.

moveCurveCVsCmd

Produces MEL command moveCurveCVs

This is a simple little plug-in which takes the selected CVs of a NURBS curves and moves them to the origin. Of itself it's not a very practical plug-in, but it demonstrates retrieving CVs from a selection list and the use of the Curve CV iterator.

To use it, you must draw a curve, switch Maya from Object selection mode to Component selection mode, the pick some or all of the CVs of the curve. Then, type the command “moveCurveCVs” in the command window to move the CVs.

moveManip

Produces MEL command moveManipContext to create the example context.

To use this plug-in, execute the command "source moveManip". This creates a new entry in the "Shelf1" tab of the tool shelf, called "moveManip". Create a sphere and click on the moveManip icon on the shelf. A free point triad manipulator will appear when the object is selected.

moveNumericTool

Produces MEL commands moveNumericToolCmd *and* moveNumericToolContext

This is an example of a selection-action tool that allows the user to type in precise translation values to while in the move tool. Once an object is selected, the tool turns into a translation tool. In this mode, the user can type in numeric values in the numeric input field to translate the object.

This tool only supports the translation of transforms, and will only perform translation in orthographic views. Undo, redo, and journaling are supported by this tool.

To use this plug-in, execute the command “source moveNumericTool”. This creates a new entry in the “Shelf1” tab of the tool shelf, called “moveNumericTool”. Click on the new icon, then select an object and drag it around in an orthographic view. With the object still selected, type in the numeric input field to enter a specific translation. You can specify whether you want absolute or relative translation values by clicking on the button to the left of the numeric input field.

moveTool

Produces MEL commands moveToolCmd *and* moveToolContext

This is an example of a selection-action tool. When nothing is selected, this tool behaves in exactly the same way as Maya’s selection tool. Once an object is selected, the tool turns into a translation tool.

Note that at this time, the plug-in can translate:

- transforms
- NURBS curve CVs
- NURBS surface CVs

- polygonal vertices

This plug-in will only perform translation in orthographic views. Undo, redo, and journaling are supported by this tool.

To use this plug-in, execute the command “source moveTool”. This creates a new entry in the “Shelf1” tab of the tool shelf, called “moveTool”. Click on the new icon, then select an object and drag it around in an orthographic view. The left mouse button allows movement in two directions, while the middle mouse button constrains the movement to a single direction.

moveManip

Produces MEL command moveManipContext *to create the example context.*

To use this plug-in, execute the command "source moveManip". This creates a new entry in the "Shelf1" tab of the tool shelf, called "moveManip". Create a sphere and click on the moveManip icon on the shelf. A free point triad manipulator will appear when the object is selected.

multiCurveNode

Produces dependency graph node multiCurve

This plug-in demonstrates how to use the MArrayDataBuilder class to create an array attribute in a compute function, the number of elements of which change on each compute cycle.

The node accepts a nurbsCurve as input, and outputs an array of nurbsCurves as outputs. The number of curves is controlled by the attribute *numCurves* and the spacing between each of the output curves is controlled by the attribute *curveOffset*. Both *numCurves* and *curveOffset* are keyable.

To use this plug-in, load it then execute the MEL command “source multiCurveNode.mel”. This will create a curve, hook it up to an instance of a multiCurve node, keyframe the *numCurves* and *curveOffset* attributes, and then hook the output of the multiCurve node to a curveVarGroup node which will display all the output curves. Once this script has been run, push play and as the animation progresses, new curves will be created and the spacing between them will increase.

nodeInfoCmd

Produces MEL command nodeInfo

This plug-in demonstrates how to query a dependency node for its type and for the plugs connected to it. It iterates over the selected items and for each item it prints the following:

- the type of the current selection item’s dependency node

- the number of plugs connected to the node
- name/attribute information for each connected plug
- the node type(s) that each connected plug is a destination of

To use it, select some objects, then execute “nodeInfo” in the command window. The node information will be printed in the window from which you started Maya.

nodeMessageCmd

Produces MEL command nodeMessage

This plug-in that demonstrates how to register/de-register a callback with the MNodeMessage class. This plug-in will register a new command in Maya called “nodeMessage” which adds a callback for the all nodes on the active selection list. A message is printed to stdout whenever a connection is made or broken for those nodes.

NodeMonitor

This class monitors a given node.

offsetNode

Produces dependency graph node offsetNode

This plug-in demonstrates how to create a user-defined weighted deformer with an associated shape. A deformer is a node which takes any number of input geometries, deforms them, and places the output into the output geometry attribute. This example plug-in defines a new deformer node that offsets vertices according to their CV’s weights.

To use this node:

- create a plane or some other object
- type: “deformer -type offset”
- a locator is created by the command, and you can use this locator to control the direction of the offset. The object’s CV’s will be offset by the value of the weights of the CV’s (the default will be the weight * some constant) in the direction of the y-vector of the locator
- you can edit the weights using either the component editor or by using the percent command (eg. percent -v .5 offset1;)

ownerEmitter

Produces dependency graph node ownerEmitter

This node is an example of a particle emitter that emits in a direction from multiple points defined by a particle shape.

There is an example MEL script “ownerEmitter.mel” that shows how to create the node and appropriate connections to correctly establish a user defined particle emitter.

particlePathsCmd

This example plug-in produces the MEL command “particlePaths” that demonstrates how particle ID information can be used to trace out curves from particle positions over time.

The following sequence of commands will create a set of curves for a particle explosion:

```
emitter -type omni -r 15 -sro 0 -nuv 0 -cye none -cyi 1 -spd 1 -srn 0 -nsp 1 -
tsp 0 -mxd 0 -mnd 0 -dx 1 -dy 0 -dz 0 -sp 0;
particle ;
connectDynamic -em emitter1 particle1;
select -r particleShape1;
gravity -pos 0 0 0 -m 0.5 -att 0 -dx 0 -dy -1 -dz 0  -mxd -1  -vsh none -vex 0
-vof 0 0 0 -vsw 360 -tsr 0.5;
connectDynamic -f gravityField1  particle1;
particlePaths -s 0 -f 4 -i 0.5 particleShape1;
```

The command uses the function set MFnParticleSystem to sample particle positions and identifiers at discrete points in time.

The command supports the options –s (start time), -f (finish time), and –i (increment period). The particle positions will be sampled starting from the start time through to the finish time in increments of the increment time. The accumulated particle positions will be passed to the MFnNurbsCurve function set to create curves from the accumulated data.

particleSystemInfoCmd

Produces command particleSystemInfo

Demonstrates the use of the new MFnParticleSystem class for retrieving particle information. The number of particle positions followed by the positions are printed to the script editor.

Syntax:

```
particleSystemInfo $particleNodeName;
```

Example:

```
particleSystemInfo particleShape2;
```

pickCmd

Produces MEL command pick

This simple plug-in demonstrates the pick-by-name functionality. Simply execute "pick <object_name>" in the command window. Also some pattern matching is possible, i.e. "pick surface*" which will pick all objects whose name begins with "surface".

pnTrianglesNode

This plug-in is ATI Radeon specific. It is a hardware shader plug-in to perform:

- ATI PN triangle tessellation on geometry, if the extension ATI_pn_triangles is supported in hardware.
- A simple vertex program using EXT_vertex_program, if the extension is supported.
- A simple fragment program using AT_fragment_program, if the extension is supported.

This is an excerpt from the PN triangle extension specification:

"When PN Triangle generation is enabled, each triangle-based geometric primitive is replaced with a new curved surface using the primitive vertices as control points. The intent of PN Triangles are to take a set of triangle-based geometry and algorithmically tessellate it into a more organic shape with a smoother silhouette. The new surface can either linearly or quadratically interpolate the normals across the patch. The vertices can be either linearly or cubically interpolated across the patch. Linear interpolation of the points would be useful for getting more sample points for lighting on the same geometric shape. All other vertex information (colors, texture coordinates, fog coordinates, and vertex weights) are interpolated linearly across the patch."

pointOnMeshInfo

This plug-in defines both a MEL command and a DG node which computes the worldspace position and normal on a poly mesh given a face index, a U-parameter and a V-parameter as input.

The concept of this plug-in is based on the pointOnSurface MEL command and pointOnSurfaceInfo node. The pointOnSubdNode.cpp plug-in example from the Maya API Devkit was also used for reference.

The MEL script AEpointOnSurfaceInfoTemplate.mel was referred to for the AE template MEL script that accompanies the pointOnMeshInfo node.

pointOnSubdNode

Produces dependency graph command pointOnSubd

This node is a simple example of how to query a subdivision surface as an input to a dependency node. This node takes a subdivision surface and a parameter point on subdivision surface and outputs the position and the normal of the surface at that point. The MEL script "connectObjectToPointOnSubd.mel" that accompanies this plug-in contains detailed documentation on how to use the node and itself provides a demonstration of that use.

polyMessageCmd

This plug-in demonstrates how to register/de-register a callback with the MPolyMessage class.

This plug-in will register a new command in Maya called "polyMessage" which adds a callback for the all nodes on the active selection list. A message is printed to stdout whenever component ids for those nodes are changed.

To run this plug-in, do the following:

```
// create a poly plane
// open the outliner and select the poly shape
// Run the plug-in
polyMessage
// select some vertices of the poly shape
// hit the delete key to see the remapped ids of the edges,
vertices and faces written out
```

polyPrimitiveCmd

Produces MEL command polyPrimitive

This plug-in creates several types of polygon primitives and demonstrates the creation of polygonal data. Once it is loaded, executing "polyPrimitive #", where "#" is an integer between 1 and 7, in the command window will cause a polygon to be created at the origin.

If you execute the command "source polyPrimitiveCmd", the script polyPrimitiveCmd.mel will be run. After this, if the command "polyPrimitiveMenu" is executed, it will bring up a window which allows you to create these objects simply by clicking a button.

polyTrgNode

This plug-in demonstrates how to add user defined triangulation for meshes using the new poly API class, *MPxPolyTrg*.

The node registers a user defined face triangulation function. After the function is registered it can be used by any mesh in the scene to do the triangulation (replace the mesh native triangulation). In order for the mesh to use this function, an attribute on the mesh 'userTrg' has to be set to the name of the function.

Different meshes may use different functions. Each of them needs to be registered. The same node can provide more than one function.

Example:

```
createNode polyTrgNode -n ptrg;
polyPlane -w 1 -h 1 -sx 10 -sy 10 -ax 0 1 0 -tx 0 -ch 1 -n
pp1;
select  -r pp1Shape;
setAttr pp1Shape.userTrg  -type "string" "triangulate";
```

progressWindowCmd

This plug-in demonstrates how to use the MProgressWindow class. The command "progressWindowPlugin" displays a simple progress window which updates every second. The progress window can be terminated by hitting escape.

To run this plug-in, do the following:

```
progressWindowCmd;
```

A progress window will be displayed. It should be noted that Maya can only display one progress window at a time. MEL also supports creating a progress window. Program errors may occur if the progress windows of MEL and the API are being called at the same time.

quadricShape

Produces shape node quadricShape

This plug-in registers a new type of shape with Maya called “quadricShape”. This shape will display spheres, cylinders, disks, and partial disks using the OpenGL gluQuadric functions.

For example, to create a sphere:

createNode quadricShape -n qSphere;

setAttr qSphere.shapeType 3;

It should be noted that there are no output attributes for this shape.

The following input attributes define the type of shape to draw.

- shapeType : 0=cylinder, 1=disk, 2=partialdisk, 3=sphere
- radius1 : cylinder base radius, disk inner radius, sphere radius
- radius2 : cylinder top radius, disk outer radius
- height : cylinder height
- startAngle : partial disk start angle
- sweepAngle : partial disk sweep angle
- slices : cylinder, disk, sphere slices
- loops : disk loops
- stacks : cylinder, sphere stacks

referenceQueryCmd

Produces MEL command referenceQuery

This example provides useful information about referenced files in the main scene. For each referenced file, the output format is as follows:

```
Referenced File: filename1
    Connections Made
      sourceAttribute -> destinationAttribute
      ...
Connections Broken
      sourceAttribute -> destinationAttribute
      ...
Attributes Changed Since File Referenced
      attribute1
      attribute2
      ...
```

To use the plug-in, open a scene file that contains file references. Execute the MEL command “referenceQuery”. The reference information is written to standard output

renderAccessNode

Produces dependency graph node renderAccessNode

This example demonstrates how to work with render callbacks. The plug-in will register a render callback, a shadow cast callback, and a post-process callback. When a render starts, render callback will be invoked, providing info related to the render's size. Then if shadow maps exist in the render, shadow cast callback will be invoked after shadow maps have been calculated, providing data to the shadow maps. Finally after geometry's have been rendered, post-render callback will be invoked, providing pointers to the rendered pixels.

The plug-in will modify the rendered image in post-process callback to demonstrate how to manipulate the pixels. The attribute pointWorld will be converted to screen space and back to test MRenderData::worldToScreen() and MRenderData::screenToWorld().

renderViewInteractiveRenderCmd

A new example that demonstrates the immediate feedback setting that has been added to the startRender() methods of the MRenderView class. After loading this plug-in, execute the following to see what options it supports:

```
help renderViewInteractiveRender;
```

renderViewRenderCmd

Produces MEL command renderViewRender

This example demonstrates how to render a full image to the Render View window using the MRenderView class. The command takes no arguments. It renders a 640x480 image tiled with a red and white circular pattern to the Render View.

renderViewRenderRegionCmd

Produces MEL command renderViewRenderRegion

This example demonstrates how to use the MRenderView class to update the currently selected Render Region in Maya's Render View. The command takes no arguments, and always updates the Render Region with a blue and white circular pattern. The command assumes that the Render View is currently displaying a 640x480 image (such as the one generated by the 'renderViewRender' example command).

rotateManip

The rotateManip is a manipulator plug-in that demonstrates the rotate base manipulator function set. This example produces the MEL command rotateContext to create the tool context for this manipulator.

To create the tool button for the plug-in, create a new shelf named “Shelf1” and execute the following MEL commands to create the tool button in this shelf:

```
rotateContext;
setParent Shelf1;
toolButton -cl toolCluster -t rotateContext1 -i1 "moveManip.xpm";
```

To use the manipulator, select an object and click on the new tool button. A rotate manipulator should appear on the object along with a state manipulator nearby. The plug-in rotate manipulator is configured to behave similarly to the built-in rotate manipulator. The state manipulator can be used to choose the mode for the rotate manipulator, which can be one of object space mode, world space mode, gimbal mode, and object space mode with snapping enabled.

sampleCmd

Produces MEL command sampleCmd

This example demonstrates how to sample shading group/node using MRenderUtil::sampleShadingNetwork().

The command takes lists of shading info such as pointCamera and UVs, and samples a given shading group/node and returns the result colors and transparencies. Lighting will be calculated if a shading group is sampled. Shadows can be calculated as well if the -shadow flag is specified. For example:

```
sampleCmd file1.outColor 4 -uvs 0 0 1 0 1 1 0 1;
sampleCmd creater1.outColor 4 -refPoints 0 0 0 1 0 0 1 0 1 0
0 1;
```

sampleCmd -h provides more details on how to use the command.

sampleParticles

Produces MEL command sampleParticles

This example demonstrates how to sample shading groups or nodes using MRenderUtil::sampleShadingNetwork() to assign colors to a particle object.

The command takes lists of shading info such as pointCamera and UVs, and samples a given shading group or node, then assigns the sampled colors and transparencies to a grid of particles using the emit command. Lighting will be calculated if a shading group is sampled. Shadows can be calculated if the -shadow flag is specified.

- sampleParticles -h provides more details on how to use the command.
- The MEL script, sampleParticles.mel demonstrates how to use this command.

scanDagCmd

Produces MEL command scanDag

This plug-in demonstrates walking the DAG using the DAG iterator class. To use it, create a number of objects anywhere in the scene, then execute the command *scanDag*. This will traverse the DAG printing information about each node it finds to the window from which you started Maya. The plug-in contains specific knowledge of cameras, lights, and NURBS surfaces, and will print object type specific information for each of these.

As well, the command accepts several flags:

-b/-breadthFirst	Perform breadth first search
-d/-depthFirst	Perform depth first search
-c/-cameras	Limit the scan to cameras
-l/-lights	Limit the scan to lights
-n/-nurbsSurfaces	Limit the scan to NURBS surfaces

scanDagSyntax

Produces MEL command scanDagSyntax

This command plug-in provides the same functionality as scanDagCmd except that the syntax parsing is performed via syntax objects. The command accepts several flags:

-b/-breadthFirst	Perform breadth first search
-d/-depthFirst	Perform depth first search
-c/-cameras	Limit the scan to cameras
-l/-lights	Limit the scan to lights
-n/-nurbsSurfaces	Limit the scan to NURBS surfaces

ShadingConnection

This class stores useful information about a shader's attribute, including what's connected upstream of it. It also automatically passes through shader switches.

ShapeMonitor

The ShapeMonitor is a singleton class that watches shape or texture nodes, and keep track of which one changed since the last export. It is used to keep the IFX scenegraph up-to-date in respect to textures.

Client code can:

- Ask for a pointer to the ShapeMonitor. (using the instance() function) If it doesn't already exist, it is created.
- Ask the ShapeMonitor to watch a specific Maya node name (specifying whether it's a texture, or a shape) and give a unique texture name. Doing so creates some callbacks on the specified node, so that we know when it changes. When a specific node changes, it's unique name is appended to the list (actually, set) of dirty textures.
- Ask for the list of dirty textures. Those should be removed from the IFX scenegraph, and will possibly be regenerated, if they still exist.
- Clear the list of dirty textures.

Additionally, once the ShapeMonitor is no longer necessary (for example, the scene is being closed), it can be destroyed using the destroy() function. Finally, all callbacks and data structures can be cleared by calling the initialize() function.

shellNode

Produces dependency graph node shell

This plug-in demonstrates how to generate a complex mesh object procedurally. It also demonstrates how to customize the attribute editor for a node.

To use the plug-in, enter the MEL command “shell” to create a new shell node. A mesh object will also be created to display the output. Open the attribute editor for the new shell node to see custom controls for picking various sea shell presets. The attribute editor layout is created by the file AEShellTemplate.mel and provides a complex example of how to create an attribute editor template.

shiftNode

Produces dependency graph node shiftNode

This plug-in demonstrates modifying uvCoord and refPointCamera from within a plug-in texture. The uvCoord and refPointCamera are marked as “renderSource” attributes. The uvCoord and refPointCamera for the current sample position are requested and then subsequently shifted four times. Each time these attributes are modified, the inColor attribute is requested, and because the attributes are render sources, the request for inColor forces a shading evaluation. Thus the 2D or 3D texture connected to inColor will be evaluated four additional times for every point shaded. The inColor values are averaged which produces a blurred result.

simpleEmitter

Produces dependency graph node simpleEmitter

This node is an example of a particle emitter that emits in a direction from a single position.

There is an example MEL script “simpleEmitter.mel” that shows how to create the node and appropriate connections to correctly establish a user defined particle emitter.

simpleLoftNode

Produces dependency graph node userLoft

This plug-in demonstrates how to accept geometry as an input, and create geometry for output. A NURBS curve is input to the node, and from it a NURBS surface is created. The resulting geometry is passed to an internal Maya node which displays it and allows it to be positioned.

To use this node, first draw a curve in the X-Y plane. Then execute the MEL command “simpleLoftNode.mel” that contains the following commands:

```
createNode transform -n simpleLoft1;
createNode nurbsSurface -n simpleLoftShape1 -p simpleLoft1;
createNode simpleLoft -n simpleLoftNode1;
connectAttr curveShape1.local simpleLoftNode1.inputCurve;
connectAttr simpleLoftNode1.outputSurface
simpleLoftShape1.create;
```

This creates a *nurbsSurface* node and hooks the result into the world for display. It then creates a *simpleLoft* node, and connects its input to *curveShape1* (the geometry from the first curve drawn), and connects its output to the NURBS surface node.

A surface will now appear on the screen. If the CVs of the original curve are selected and moved, the surface will be reconstructed to match.

simpleSolverNode

Registers IK solver simpleSolverNode

Loading this plug-in will register a new IK solver with Maya as type *simpleSolverNode*. To use the solver, create a single IK bone with 2 joints using the Joint tool, then enter the following command in the command window to create an IK handle which uses the new solver:

```
ikHandle -sol simpleSolverNode -sj joint1 -ee joint2
```

Moving the handle in the x-y plane will rotate the joint.

The command:

```
ikHandle -q -sol handleName
```

can be used to determine which solver a handle is using.

simpleSpring

Produces dependency graph node simpleSpring

This node is an example of a spring node that calculates the spring behavior that Maya will use in a simulation.

There is an example MEL script “simpleSpring.mel” that shows how to create the node and appropriate connections to correctly establish a user defined spring law.

sineNode

Produces dependency graph node sine

This plug-in is a simpler version of circleNode. It takes a single input value and outputs the sine of this multiplied by 10. It’s a simple example of creating a procedural animation. Below are a simple set of commands to attach this node to a sphere:

```
sphere -n sphere;
createNode sine -n sine1;
connectAttr sine1.output sphere.tx;
connectAttr time1.outTime sine1.input;
```

Once this is done, as the time changes the sphere will roughly move along the X axis in a periodic manner.

(Roughly because the input is normally 1 through 48 and the compute method of the sine node just takes the sine of the input, which should be radians.)

spiralAnimCurveCmd

Produces MEL command spiralAnimCurve

This plug-in attaches anim curves to the X, Y, and Z translation attributes of the selected objects, and then animates the object along a spiral path. This is a good example of attaching a anim curve to the attributes of an object.

To use it, select an object (or objects) in the Maya perspective window, the execute the command "spiralAnimCurve". Next click on the "play" button on the time slider at the bottom right of the screen. The selected objects will move in a spiral as the animation plays.

splitUVCmd

The splitUV command unshares or "splits" the selected UVs of a polygonal mesh. It is also a good example of how to write poly operation nodes that properly deal with history, tweaks, etc. For a thorough explanation of creating this command, refer to the splitUVCmd example in the "Working with the polyAPI" chapter of the <PalItal>Maya Developer's Tool Kit.

surfaceBumpManip

The surfaceBumpManip is a manipulator plug-in that demonstrates how the pointOnSurface base manipulator can be used. This example produces the MEL command surfaceBumpContext to create the tool context for this manipulator.

To create the tool button for the plug-in, create a new shelf named "Shelf1" and execute the following MEL commands to create the tool button in this shelf:

```
surfaceBumpContext;
setParent Shelf1;
toolButton -cl toolCluster -t surfaceBumpContext1 -i1 "moveManip.xpm";
```

To use the manipulator, create a NURBS sphere and select the tool button. A manipulator should appear on the surface of the sphere. Moving the manipulator over the surface of the sphere will cause the surface to be modified near the manipulator by moving a CV out along the sphere normal.

surfaceCreateCmd

Produces MEL command surfaceCreate

This plug-in creates a NURBS surface by supplying its own CVs and knots. To use it, just enter the command “surfaceCreate”.

surfaceTwistCmd

Produces MEL command surfaceTwist

To use this command, you must first create and select a number of NURBS surfaces or polygons (a fun surface to twist is the one created by the surfaceCreateCmd command). Then enter the command “surfaceTwist” and all selected surfaces will be twisted around the y-axis.

This command demonstrates how to access and modify the CVs of a NURBS surface or the vertices of a polygon.

sweptEmitter

Produces dependency graph node sweptEmitter

This node is an example of a particle emitter that emits in a direction from a curve or surface.

There is an example MEL script “sweptEmitter.mel” that shows how to create the node and appropriate connections to correctly establish a user defined particle emitter.

swissArmyManip

*Produces dependency graph nodes swissArmy*Locator *and* swissArmyLocatorManip

This is a contrived example plug-in that attaches one of every kind of user-defined manipulator to a node. It is a good example of the source code required to create each user-defined manipulator. To use it, issue the command:

```
createNode swissArmyLocator
```

then click the *showManip* icon on the toolbar. The locator on the screen will be overlaid with one of every kind of user-defined manipulator.

torusField

Produces dependency graph node torusField

This node is an example of a dynamics field that creates a attract-repel field between itself and a distance.

There is an example MEL script “torusField.mel” that shows how to create the node and appropriate connections to correctly establish a user defined field.

transCircleNode

Produces dependency graph node transCircle

This plug-in demonstrates how to use an attribute that contains multiple values (a *compound* attribute). The *translate* attribute of the *transform* node is used which is composed of the elements: *translateX*, *translateY*, and *translateZ*, all of which are communicated over a single dependency graph connection.

To use this node, execute the MEL command "transCircleNode.mel" that contains the following commands:

```
createNode transCircle -n circleNode1;
sphere -n sphere1 -r 1;
sphere -n sphere2 -r 2;
connectAttr sphere2.translate circleNode1.inputTranslate;
connectAttr circleNode1.outputTranslate sphere1.translate;
connectAttr time1.outTime circleNode1.input;
```

This creates two spheres and a *transCircle* node. The translate attribute of *sphere1* is connected to the input of the *transCircle* node, and its output is connected to the translate attribute of *sphere2*.

If the play button is pressed, the second sphere will circle around the first. If the first sphere is moved, the second will also move such that the first sphere always remains at the center of the circle.

This plug-in also comes with a sample of an *attribute editor template*. This example suppresses the display of all attributes except *scale* and *frames*, and also provides an extra *quick set* control for the scale attribute that uses radio buttons to update the attribute value.

translateCmd

Produces MEL command translate

This plug-in translates the CVs of selected curves, surfaces, or polygonal objects by a user specified amount. It is a good example of manipulating CVs on these three data types. To use it, select an object then execute "translate 1.0 2.0 3.0" in the command window.

It should be noted that this command will not work on NURBS primitives that have construction history. The history forces the CVs to return to their original position immediately after the translate command has moved them. To allow the translate command to work on such surfaces, you must either delete the construction history on the object (Edit > Delete By Type > Construction History), or open the Tool Settings window before creating the surface and turn off History.

undoRedoMsgCmd

This plug-in example demonstrates how to listen to undo and redo message events using the MEventMessage class.

The syntax of the command is:

```
undoRedoMsg add;
undoRedoMsg remove;
```

The add argument causes listening to undo/redo to be turned on. The remove argument causes undo/redo listening to be removed.

userMsgCmd

This example plug-in produces the MEL command "userMessage" that demonstrates how a plug-in can use user-defined messages.

Here is an example of how the command can be used:

```
// Register a user-defined event named "test".  The plug-in will internally
register
// callbacks for the event.
userMessage -r test;
// Post to the user-defined event.  The plug-in prints info messages from the
callbacks.
userMessage -p test;
// Entered userMessage::userCallback2
// Received data: Sample Client Data (an MString object)
// Entered userMessage::userCallback1
// Received data: Sample Client Data (an MString object)
// Deregister the user-defined event
userMessage -d test;
// Trying to post a message after the event has been removed will fail.
userMessage -p test;
```

viewCaptureCmd

Produces MEL command viewCapture

This plug-in uses OpenGL to capture the current 3D view and write it into a PPM file. To use it, give it a filename as an argument into which the PPM image of the current view should be written.

Limitations

Any parts of other X windows that are obscuring the view will be captured rather than the view underneath. This is an effect of the OpenGL buffer system on SGIs.

Color index mode buffers cannot be read by this plug-in, so the view should be set to shaded or rgb mode before doing the capture.

volumeLightCmd

Produces command volumeLight

Demonstrates the use of the MFnVolumeLight class. This example creates a volume light then queries and sets a number of its attributes.

Example usage:

```
volumeLight

volumeLight -a $arc -c $coneEndRadius -e
$emitAmbient
```

whatisCmd

Produces MEL command whatis

This simple command prints a message to standard out describing the API types of Maya objects. If no Maya objects are passed to the command then it lists the types of all of the currently selected objects.

For each object, the following information will be printed:

- name of the object
- API type for the object
- API function sets that could be used on the object. Note that not every function set listed actually exists.

This list is essentially the class derivation list containing all parent classes of this object.

For example, the command

```
whatis nurbsSphereShape1
```

results in the following output:

```
Name: nurbsSphereShape1
Type: kNurbsSurface
Function Sets: kBase, kNamedObject, kDependencyNode, kDagNode, kShape,
kGeometric, kSurface, kNurbsSurface, kNurbsSurfaceGeom
```

This is a good example of taking Maya objects as arguments to a command.

yTwistNode

Produces dependency graph node yTwist

This plug-in demonstrates how to create a user-defined deformer. A deformer is a node which takes any number of input geometries, deforms them, and places the output into the output geometry attribute. This example plug-in defines a new deformer node that twists the deformed vertices of the input around the y-axis.

To use this node:

- create a sphere or some other object
- select the sphere
- type: “deformer -type yTwist”
- bring up the channel box
- select the yTwist input
- change the Angle value of the yTwist input in the channel box

zoomCameraCmd

Produces MEL command zoomCamera

This is a simple plug-in which divides the horizontal field of view for the current active camera by 2.0. It is a good example of getting the current active view, and of modifying the camera. To use this plug-in, first create a camera by selecting the menu item
Rendering > Navigation >Create Camera, then position the camera to “look at” an object in the scene. Then, either,

- Select the menu item View > Perspective > Camera Name

or

- Hold down the Ctrl key, press the right mouse button, and select Perspective > Camera Name from the menu that pops up.

You will now be looking through the camera. Execute “zoomCamera” in the command window, and your view through the camera will zoom-in by a factor of 2.

Example stand-alone application descriptions

asciiToBinary

This command takes a list of Maya scene files as arguments. For each one, the file is loaded and if the file was in Maya Ascii format, it is written in Maya Binary format to a file with the same name that the extension is replaced with string ".mb" (or at the end if the filename has no extension).

helloWorld

Running this application will load the Maya DSOs, initialize everything, then simply print "hello World" on standard output. This example now uses the Mlibrary::initialize() method that allows the displaying of command console output.

readAndWrite

This command takes a list of Maya scene files as arguments. For each one, the file is loaded and written without changes to a file with the same name except that the string ".updated" is inserted just before the extension in the filename (or at the end if the filename has no extension).

This command is actually useful because as a side effect of the "read then write" operation, the given scene files will be upgraded to the latest Maya file format.

surfaceCreate

This command takes no arguments. It creates a a NURBS surface by supplying its own CVs and knots, then writes the result out to a file called *surf1.ma*.

surfaceTwist

This command takes no arguments. It opens a files called *surf1.ma* and applies the *twist* function to all surfaces in that file. The modified scene is then written out to a file called *surf2.ma*.

Shader source code examples

Example shader	Classification	Description
anisotropicShader	shader/surface	example shader that modifies specular highlights
backfillShader	shader/surface	modified Phong surface shader that fills non-diffuse illuminated areas with color

Example shader	**Classification**	**Description**
brickShader	texture/2d	brick texture node
cellShader	texture/3d	Solid texture of cells
checkerShader	texture/2d	node that uses texture UV coordinates to make a checker pattern
compositingShader	texture/2d	node for compositing anti-aliased mask and rgb textures
contrastShader	utility/color	node that manipulates color with contrast + bias
depthShader	shader/surface	example surface shader that colors objects based on distance from the camera
displacementShader	shader/ displacement	example displacement shader
flameShader	texture/3d	Solid texture of flames
gammaShader	utility/color	node that manipulates color with gamma correction
geomShader	utility/general	node that outputs geometry xyz position
hwAnisotropicShader_NV20	shader/surface/ utility	NV20-specific (Geforce3) sample shader to produce an anisotropic shading effect
hwPhongShader	shader/surface/ utility	example of using a cube-environment map to perform per pixel phong shading
hwToonShader_NV20	hader/surface/ utility	NV20-specific (Geforce3) sample shader for cartoon-like effects
interpShader	utility/general	node that interpolates two colors based on the surface normal
lambertShader	shader/surface	example Lambert surface shader
lavaShader	texture/3d	Solid texture of lava

Example shader	**Classification**	**Description**
lightShader	light	example light shader
mixtureShader	utility/color	node that takes two color inputs and two mask inputs and mixes them into a resulting color
noiseShader	texture/3d	node that applies a noise function to the output color
phongShader	shader/surface	example Phong surface shader
shadowMatteShader	shader/surface	example shader that outputs mask/ alpha in shadowed areas only
slopeShader	utility/color	colors faces of a mesh based on their slope or angle relative to a user defined threshold
solidCheckerShader	texture/3d	example solid texture
vertexColorShader (cvColorShader)	utility/color	node that allows vertex color (CPV) to be software rendered
volumeShader	shader/volume	example volume shader

anisotropicShader

Produces dependency graph node AnisotropicShader

This node modifies the specular highlight of a surface shader.

The output attributes of the AnisotropicShader node are called “outColor” and “outTransparency”. To use this shader, create a AnisotropicShader node with Shading Group or connect it’s output to a Shading Group’s “SurfaceShader” attribute.

backfillShader

Produces dependency graph node BackFillShader

This node is an example of a surface shader that fills non-diffuse illuminated areas with color.

The inputs for this node are can be found in the Maya UI on the Attribute Editor for the node.

The output attribute of the node is called “outColor”. It is a 3 float value that represents the resulting color produced by the shader. To use this shader, create a BackFillShader with Shading Group or connect its output to a Shading Group’s “SurfaceShader” attribute.

brickShader

Produces dependency graph node brickTexture

This node is an example of brick 2d texture.

The output attribute of the BrickTexture node is called “outColor”. To use this shader, create a BrickTexture and connect it’s output to an input of a surface/shader node such as Color.

cellShader

Produces dependency graph node Cells

This node is an example of a solid texture that divides 3D space into cells.

The output attributes of this node are called “outColor” and “outAlpha.”

To use this shader, create a Cell node and connect the output to an input of a surface/shader node such as Color.

checkerShader

Produces dependency graph node CheckerNode

This node is an example of evaluating texture UV coordinates and produces a checkerboard pattern. It also can take advantage of the 2d texture placement node for transforming UV coordinates with a manipulator.

The inputs for this node are two colors and two bias values, where each bias value is used to determine the pattern.

The output attribute of the CheckerNode node is called “outColor”. It is a 3 float value that represents the resulting color derived from the object’s UV texture coordinates. To use this shader, create a CheckerNode and connect its output to an input of a surface/shader node such as Color.

compositingShader

Produces dependency graph node CompositeNode

This node is an example of a compositing shading node.

The inputs for this node are foreground, background, back color, and a mask value.

The output attributes of the CompositeNode node are called “outColor” and “outAlpha”. To use this shader, create a CompositeNode and connect it’s output to an input of a surface/shader node such as Color.

contrastShader

Produces dependency graph node ContrastNode

This node is an example of manipulating color.

The inputs for this node are input color, contrast and bias. The contrast and bias inputs are 3 float values so that it can be applied to each rgb channel of an input color separately.

The output attribute of the ContrastNode node is called “outColor”. It is a 3 float value that represents the resulting color derived from the effective contrast and bias settings. To use this shader, create a ContrastNode and connect its output to an input of a surface/shader node such as Color.

depthShader

Produces dependency graph node DepthShader

This node is an example of a surface shader that colors objects based on distance from the camera.

The inputs for this node are can be found in the Maya UI on the Attribute Editor for the node.

The output attribute of the node is called “outColor”. It is a 3 float value that represents the resulting color produced by the node. To use this shader, create a DepthShader with Shading Group or connect its output to a Shading Group’s “SurfaceShader” attribute.

displacementShader

Produces dependency graph node DispNodeExample

This node is an example of displacement shader.

The inputs for this node are input color and an input multiplier.

The output attribute of the DispNodeExample node is called “displacement”. To use this shader, create a DispNodeExample node and connect it’s output to the “displacementShader” input of a Shading Group.

flameShader

Produces dependency graph node Flame

This node is an example of a solid texture that uses turbulence and an axis to animate the texture’s movement.

The output attributes of this node are called “outColor” and “outAlpha.”

To use this shader, create a Flame node and connect the output to an input of a surface/shader node such as Color.

gammaShader

Produces dependency graph node GammaNode

This node is an example of manipulating color with gamma correction.

The inputs for this node are input color and gamma values. The gamma input has 3 float values so that it can be applied to each rgb channel of an input color separately.

The output attribute of the GammaNode node is called “outColor”. To use this shader, create a GammaNode and connect its output to an input of a surface/shader node such as Color.

geomShader

Produces dependency graph node GeomNode

This node is an example of a evaluating the geometric xyz position on an object.

The inputs for this node are scale and offsets depicted as sliders in the Attribute Editor for the node.

The output attribute of the GeomNode node is called “outColor”. It is a 3 float value that represents the xyz position on the object. To use this shader, create a GeomNode and connect its output to an input of a surface/shader node such as Color.

hwAnisotropicShader_NV20

NV20-specific (Geforce3) sample shader.

This shader produces an anisotropic shading effect. It allows the user to change the parameters of an anisotropic lookup table.

This shader builds on the foundation demonstrated in the hwUnlitShader.

hwPhongShader

This is an example of a using a cube-environment map to perform per pixel Phong shading.

The light direction is currently fixed at the eye position. This could be changed to track an actual light but has not been coded for this example.

If multiple lights are to be supported, than the environment map would need to be looked up for each light either using multitexturing or multipass.

hwToonShader_NV20

NV20-specific (Geforce3) sample shader.

This shader allows for cartoon-like effects.

It allows the user to specify a base decal texture, and a lighting look-up texture.

This shader builds on the foundation demonstrated in the hwUnlitShader.

interpShader

Produces dependency graph node InterpNode

This node is an example of evaluating the surface normal on an object.

The inputs for this node are two colors and an input value that will modify the interpolation of color.

The output attribute of the InterpNode node is called “outColor”. It is a 3 float value that represents the interpolated color as a result of the surface normal. To use this shader, create a IntepNode and connect its output to an input of a surface/shader node such as Color or Transparency.

lambertShader

Produces dependency graph node LambertShader

This node is an example of a Lambert shader and how to build a dependency node as a surface shader in Maya.

The inputs for this node are many, and can be found in the Maya UI on the Attribute Editor for the node.

The output attributes for the node are “outColor” and “outTransparency”. To use this shader, create a lambertShader with Shading Group or connect the outputs to a Shading Group’s “SurfaceShader” attribute.

lavaShader

Produces dependency graph node Lava

This node is an example of a solid texture that uses turbulence.

The output attributes of this node are called “outColor” and “outAlpha”.

To use this shader, create a Lava node and connect the output to an input of a surface/shader node such as Color.

lightShader

Produces dependency graph node lightNode

This node is an example of a directional light.

The inputs for this node are direction, color, and some boolean flags to indicate if the light is contributing to ambient, diffuse, and specular components. There is also a position input that can receive a connection from a 3d manipulator for placement within the scene.

The output attribute of the LightNode node is compound attribute called “lightData”. To use this shader, create a LightNode and modify it’s inputs to see the different illumination results.

mixtureShader

Produces dependency graph node MixtureNode

This node is an example of evaluating multiple inputs and produces a resulting color.

The inputs for this node are two colors and two masks, where each color has a corresponding mask associated with it and the result color is the mixture of both colors with masks.

The output attribute of the MixtureNode node is called “outColor”. It is a 3 float value that represents the resulting color mixture based on the mask values. To use this shader, create a MixtureNode and connect it’s output to an input of a surface/shader node such as Color.

noiseShader

Produces dependency graph node SolidNoise

This node is an example of a 3d texture.

The output attribute of the SolidNoise node is called “outColor”.

shadowMatteShader

Produces dependency graph node ShadowMatte

This node will create a matte/mask only in areas that are in a lights shadow through the transparency output.

You can optionally use a boolean input to see where the shadow areas are in the color channels instead of the mask.

The output attributes of the ShadowMatte node are called “outColor” and “outTransparency”. To use this shader, create a ShadowMatte node with Shading Group or connect it’s output to a Shading Group’s “SurfaceShader” attribute.

slopeShader

Produces dependency graph node slopeShader

The slopeShape is comprised of three components: slopeShader, slopeShader Behavior and slopeShaderNode.

The slopeShader colors faces of a mesh based on their slope or angle relative to a user defined threshold.

solidCheckerShader

Produces dependency graph node SolidChecker

This node is an example of a 3d checker texture.

The output attributes of the SolidChecker node are called “outColor” and “outAlpha”. To use this shader, create a SolidChecker node and connect its output to an input of a surface/shader node such as Color.

phongShader

Produces dependency graph node PhongNode

This node is an example of a Phong shader and how to build a dependency node as a surface shader in Maya.

The inputs for this node are can be found in the Maya UI on the Attribute Editor for the node.

The output attribute of the node is called “outColor”. It is a 3 float value that represents the resulting color produced by the node. To use this shader, create a phongNode with Shading Group or connect its output to a Shading Group’s “SurfaceShader” attribute.

vertexColorShader (cvColorShader)

This plug-in creates a node that allows vertex color (CPV) to be software rendered.

volumeShader

Produces dependency graph node VolumeNode

This node is an example of a volume shader. Volume shaders are used to apply color to specialized shapes associated with light sources called “light shapes”. One such set of shapes is created by the Light Fog effect. The Light Fog effect can be assigned to any point or spot light through the attribute editors for the point and spot lights.

The output attribute of the VolumeNode node is called “outColor”. To use this shader, create a VolumeNode node and connect its output to the “volumeShader” input of a Shading Group. The shading group must be connected to a light shape.

Index

Index