CHAPTER 4

C++ API

4.1 INTRODUCTION

MEL clearly provides a very powerful and effective means for both automating and simplifying tasks in Maya. It may be that MEL provides all the programmable functionality you'll ever need. If however you need further access and control, you can use the C++ API. Through this API you can create your own custom Dependency Graph nodes. These can be integrated directly into Maya and work seamlessly with all other nodes. This allows you to directly implant your own functionality into the very core of Maya.

Another nonnegligible advantage of the C++ API is that the plugins you develop will often run much faster than the equivalent MEL scripts. Since the plugins are compiled and linked, the result is fully optimized for the target platform. There is no need for on-the-fly interpretation as with MEL, thereby making it much faster.

Admittedly, to exploit the C++ API fully requires a good understanding of the C++ language, though it is possible to begin creating some Maya plugins with just a minimum understanding of C++.

Following is a nonexhaustive list of all the areas and functionality in Maya that can be extended using this API.

- **Commands**

  Custom commands can be created that have all the functionality of Maya's own built-in commands, including undo/redo, help, complete access to the scene, and any combination of command arguments. Since the command is written in C++, you are free to use any C++ functionality, including external libraries, from within your command. This is in sharp contrast with MEL procedures, which

allow calls only to other MEL procedures and commands. Your own commands are treated exactly like Maya's built-in commands. As such, they can be called in any MEL statement.

- **Dependency Graph Nodes**

  The C++ API allows you to create your own custom DG nodes. You can create a simple DG node that provides basic functionality, such as adding two points together, or a very complex one that animates an entire character. Once your node is registered with Maya, it can be created, deleted, and edited like any standard node. Connections can be made to it from other nodes, and it can connect its outputs to other nodes. Your nodes can be completely and seamlessly integrated into Maya.

  In addition to the basic DG nodes, you can extend on more specialized DG nodes, such as manipulators, locators, shapes, and so on to add your own functionality. By deriving your own nodes from these specialized nodes you can reuse a lot of their existing functionality, thereby enabling you to add just the minimum necessary for your purposes.

- **Tools/Contexts**

  There is often a need to perform a given set of interactive operations in order to complete a task. To split an edge, for example, you must select two edges then press **Enter**. This series of interactive steps is performed using a context. You are given quite a lot of freedom as to what your context can do, including how the mouse clicks and the mouse movements are interpreted. Creating your own contexts allows for the implementation of specific modeling and animation operations.

- **File Translators**

  In order to support a wide variety of known, and unknown, file formats, Maya provides for custom file translators. These file translators control the loading and saving of Maya data. They can use any means necessary to convert the data into their own formats. There are even specific extensions for doing game translators.

- **Deformers**

  A deformer provides a mechanism for moving a series of points. The way in which it does its deformation can be as simple or as complex as necessary. With the C++ API, you can create your own deformers that are completely free to deform many different types of objects.

- **Shaders**

  Maya includes an extensive set of **Shading Network** nodes. These nodes can be combined in different ways to create complex shading of objects. Using the C++ API, you can create your own custom shading nodes. Like Maya's nodes, they have complete access to all the relevant shading information, including normals, positions, textures, and so on. You can even implement your very own super-sampling mechanisms.

  Custom hardware shaders can also be created.

- **Manipulators**

  While is it possible to change a node's attribute values using the Channel Box, Graph Editor, and so on, Maya does provide a visual means of changing values: manipulators. Manipulators provide a series of visual controls with which you interact to change a node's attribute. For instance the sphere manipulator allows you to move one of the manipulator points to increase the radius of the sphere. Manipulators can sometimes be more intuitive for users, since with them users can interactively move or adjust a visual control, which is often easier than typing in a specific value. While they are interactively changing the value, they will get immediate feedback as to the result.

- **Locators**

  A locator is used to give the user a visual reference point. Locators operate exactly like any Maya object, except that they won't show up in the final render. Custom locators can be created using the C++ API. You are free to draw the locator using almost any OpenGL method, so locators can be very simple or complex in appearance.

- **Fields**

  A field provides a volume in which various forces can be applied. A field is typically applied to particles to have them move in a particular way. Some examples of fields include gravity, drag, and turbulence. By creating your own fields you control the forces being applied to particles.

- **Emitters**

  The purpose of an emitter is to determine when and how a series of particles is generated. Emitters control how many particles are created and in what direction and at what velocity they move. While Maya comes with an extensive set of different emitters, using the C++ API you can create your own custom emitters.

- **Shapes**

  Shape nodes hold the actual geometry data. Some examples include NURBS curves and surfaces, polygonal meshes, particles, and so on. Maya allows you to create your own custom shapes. These shapes can be completely integrated into Maya, so all the standard modeling and animation tools will work with them. Your custom shapes can be created, deleted, and edited like Maya's built-in shapes.

- **Solvers**

  Even though Maya already provides an extensive set of inverse kinematic (IK) solvers, it is possible to create your own. A custom IK solver can be created and integrated into Maya. It can then be used to control a series of bones such as the standard Maya does.

  It is also possible to define your own *spring laws*. These can be used to get complete control over how soft body dynamics are performed.

4.2 FUNDAMENTAL CONCEPTS

Before discussion of the specifics of the C++ API, it is extremely important to cover some of the fundamental concepts. Since any system provides its own particular levels of freedom and, likewise, constraints, it is important to have a good grasp of the factors that motivated its design. Understanding why the C++ API was designed the way it is gives you insights into how best to design your own plugins so that they work well within Maya's framework.

Since the design of the C++ API differs from the typical object-oriented approach, it is important to learn its differences. An understanding of the differences will help guide you in the design and implementation of your own plugins. In fact, this understanding is absolutely critical to developing efficient and effective plugins.

4.2.1 Abstracted Layer

When using the C++ API, it may seem as if you are directly manipulating Maya's objects and data structures. What you are, in fact, doing is using a layer above the actual Maya core. The API gives the programmer access to Maya's core through a well-defined set of interfaces. Maya's core consists of all the internal functions and data that is developed by Alias | Wavefront. At no time can you *directly* access Maya's core. All communication, whether it be the creation, manipulation, or deletion of

data, must be done through the API. The diagram in Figure 4.1 shows the different programming layers and how they communicate with each other.

Those familar with systems or windows programming will recall that typical APIs provide a set of function calls that give them access to the underlying system. The complete API consists of the entire set of available functions. However, in Maya the API is defined through a set of C++ classes. All access to Maya's core is therefore done through each class's member functions. These member functions can create, retrieve, and manipulate Maya's data.

So why create an API on top of Maya's core? Why not give the developer direct access to Maya's internal functions and data? By creating an API on top of the core, the developer is abstracted from the actual details of Maya's current implementation. By not exposing the current implementation to the developer, Maya's engineers are free to change and improve Maya's core without having to be concerned about "breaking" code developed by external developers. As long as the API doesn't change too radically, external developers don't have continually to update their plugins when a change is made to Maya's core. This safeguards their investment in plugins that have already been developed.

The API also provides a certain level of protection from possible misuse. Since Maya maintains and controls the internal data, it can prevent an errant plugin from deleting critical data. The API can trap potentially bad calls and return an error without actually performing the call. As such, the API acts as a filter and guards against potentially dangerous operations. This isn't to say that you can't crash Maya, though these safeguards reduce the likelihood substantially.

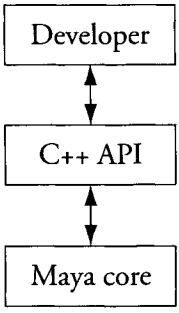


FIGURE 4.1 Programming interface

Given that you are using a layer on top of Maya's core, you may wonder if there is a speed penalty by going through this extra layer. In practice there is little penalty. Many of the C++ classes are, in fact, wrapper classes that translate very quickly into Maya's internal representation. Also, many of the API class methods have direct equivalents in the internal classes, so the translation from one to the other is fast.

4.2.2 Classes

The C++ API consists of a series of C++ classes. The majority of the classes are divided into logical hierarchies based on their type. At this stage, it isn't important to learn what each particular class is and how it works, but it is important to understand the general structure of the hierarchy and where the classes are located in this hierarchy. The complete hierarchy of Maya classes can be found in the Maya documentation:

```
maya_install\docs\en_US\html\DevKit\PlugInsAPI\classDoc\hierarchy.html
```

The Maya C++ hierarchy consists of a lot of classes. Fortunately, only a relatively small portion of them are used for the most common plugins. In practice, a core set of classes is used very often, while many of the more esoteric ones are rarely used. As such, this book doesn't cover each class in detail, but instead focuses on the most important and widely used ones.

NAMING CONVENTION

All the Maya class names start with a capital **M**, for example, **MObject**. Since Maya doesn't use C++ namespaces, this helps prevent any conflicts with other user-defined classes. Maya also differentiates classes by putting them into subclasses that are based on their functionality. While some of the classes don't follow the typical object-oriented parent-child hierarchy, they do contain common functionality and so share a common prefix. For example, you'll notice that there are many classes prefixed with **MPx**. All proxy classes are derived from this subclass. The class name prefixes are presented in Table 4.1.

DESIGN

While the C++ class hierarchy may appear to be similar in design to most object-oriented hierarchies, there are some very important differences that must be understood in order to effectively use the API. In fact, not fully understanding the differences often causes a great deal of confusion when you later design your own plugins.

TABLE 4.1 CLASS NAME PREFIXES

Prefix	Logical Grouping	Examples
M	Maya class	MObject, MPoint, M3dView
MPx	Proxy object	MPxNode
MIt	Iterator class	MItDag, MItMeshEdge
MFn	Function set	MFnMesh, MFnDagNode

Classical Approach

First, take a look at how *typical* C++ class hierarchies are designed and then compare this to the design of the class hierarchy used in the Maya C++ API. The standard texts on object-oriented design state that the most common methodology for designing a class hierarchy is by starting with a base class. This base class defines the root of the hierarchy. While it is possible to have more than one base class, for the purposes of this explanation it is restricted to just one. This root base class is typically abstract and contains just the most common member functions and data that all derived classes share. These functions are often defined to be abstract (pure virtual functions). Classes derived from this base class implement either some or all of these member functions or possibly add their own abstract functions. The number of classes and whether they implement the functions or leave them abstract is up to the designer. Applying this design methodology to the creation of a vehicle class hierarchy could result in the hierarchy shown in Figure 4.2. The hierarchy starts with the root **Vehicle** class and then derives a set of specific vehicle classes (**Motorcycle**, **Car**, **Truck**) from it.

If you use the standard object-oriented approach, each of the classes contains both the data and the functions that operate on it. As such, each vehicle subclass would contain its own specific vehicle data. The **Car** class may, for instance, hold information about what type of wheels and how many gears there are. This data

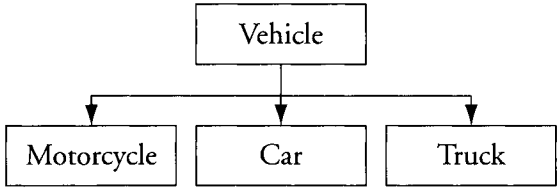


FIGURE 4.2 Vehicle class hierarchy

could be hidden away in a private or protected member, or it could be exposed as a public member. Alternatively, this data could be accessed only through member functions. Whichever approach is used, the class ultimately defines the *interface* to the data. The class decides how you can access the data. So each class, in essence, defines its own API to its functionality and data.

In order to better exploit *polymorphism,* the **Vehicle** class may define the following set of pure virtual functions:

```
virtual void drive() = 0;
virtual int numWheels() = 0;
```

The subclasses, **Motorcycle**, **Car**, and **Truck**, would then implement this function. Once implemented, these classes could then be used as follows:

```
Car speedy;
speedy.drive();
```

Since all the main classes are derived from **Vehicle**, you can use polymorphism to call an object's `drive()` function without knowing the exact type of the object. In the following example, the broom pointer can point to both the **Truck** and **Car** objects and call their `drive()` functions. The actual function called is the class-specific `drive()` function.

```
Truck t;
Car c;

Vehicle *broom = &c;
broom->drive();
broom = &t;
broom->drive();
```

Imagine that the vehicle hierarchy was the API provided to you, the developer. In order to extend the hierarchy to incorporate your own classes, you could derive them from **Vehicle** or any of the other classes. If, for example, you wanted to create a more specific type of car that is used for off-road travel, you could derive a new class, **Offroad**, from the **Car** class. The new class hierarchy is shown in Figure 4.3.

The new **Offroad** class has been seamlessly integrated into the hierarchy. It can be used with the same ease as any of the original vehicle-derived classes. Even

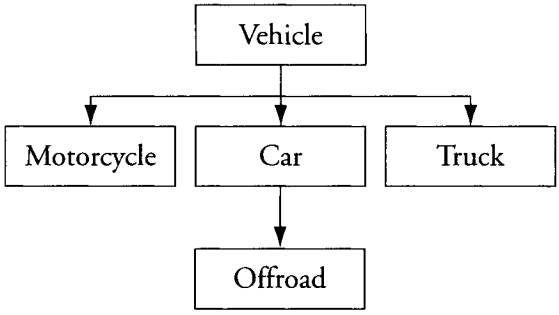


FIGURE 4.3 Extended class hierarchy

though you can add classes, you can't make changes to the original hierarchy. You can add extensions to it by deriving your own classes. It isn't possible, with this design, to add a new virtual function to the **Vehicle** or **Truck** classes. Only the original hierarchy designers can do this. If they decide to make a change to the hierarchy, then your class will be affected. The amount of work you would have to do to your class depends on the pervasiveness of their changes. Any new member functions may also need to be implemented in your class. If, for example, the following virtual function was added to the **Vehicle** class:

```
virtual bool isElectric() = 0;
```

then your **Offroad** class must provide an implementation for this function or else it can't be instantiated, since it will be an abstract class.

Maya Approach

With an understanding of the classical approach to object-oriented hierarchy design, now look at how Maya's approach differs. In order to demonstrate the difference, this same set of vehicle classes is redesigned using the Maya approach. The previous design coupled the data and the functions that operate on that data into the class. The Maya approach makes a separation between the two. So for the vehicle hierarchy, a class, **Object**, for holding data is created. This class is designed to hold many different types of data. The type of data it holds depends on which vehicle class uses it. The hierarchy of data classes is shown in Figure 4.4.

The **Object** class would contain data common to all the classes in the hierarchy, while the derived classes, **MotorcycleObj**, **CarObj**, and **TruckObj**, would contain

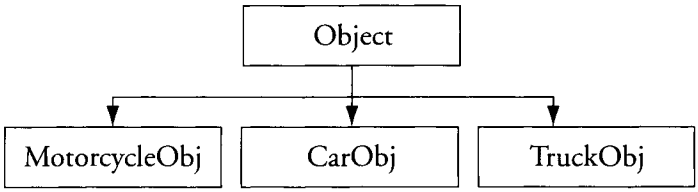


FIGURE 4.4 Data hierarchy

more specific data for a particular type of vehicle. These classes contain the data and also their own member functions to access the data.

```
virtual void drive() = 0;
virtual int numWheels() = 0;
```

Another separate hierarchy of classes is used to access and manipulate the various data classes. Maya refers to classes that operate on the data as *function sets.* In this example, you create classes that don't contain any data but that provide member functions that can operate on an **Object** object. In object-oriented terminology, these classes are known as *functors.* An entire class hierarchy of function sets is created to operate on vehicle data. Figure 4.5 shows this hierarchy.

The root of the hierarchy is the **VehicleFn** class. This class contains the same set of member functions as the original hierarchy.

```
virtual void drive();
virtual int numWheels();
```

The difference is that the class won't also contain the data that it operates on. Instead, the data will be given to the class through the **Object** hierarchy classes. The

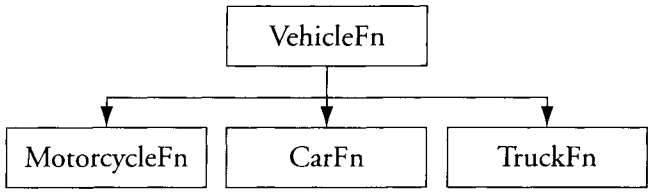


FIGURE 4.5 Function set hierarchy

MotorcycleFn class therefore operates on a **MotorcycleObj** object. The data object is attached to the function set class so that whenever a member function is called it operates on that data.

The **VehicleFn** class contains a private pointer to the data object it operates on.

```
Object *data;
```

Another member function, which allows you to specify which data object the class operates on, is added to the base **VehicleFn** class.

```
virtual void setObject( Object *obj ) { data = obj; }
```

Given this new hierarchy design, how can you perform some of the same operations as the original hierarchy? To make the car drive, the following would now be used:

```
CarObj carData;
VehicleFn speedyFn;
speedyFn.setObject( &carData );
speedyFn.drive();
```

The **CarObj** object is first created. This contains the data for the car. A **VehicleFn** function set is then created and attached to the car data. When the `drive()` function is called, it then operates on the **carData** object.

Since the **VehicleFn** class defines common functions for all derived classes, it can call those functions. It could be applied to the **TruckObj** object.

```
TruckObj t;
CarObj c;

VehicleFn broomFn;
broomFn.setObject( &t );
broomFn.drive();
broomFn.setObject( &c );
broomFn.drive();
```

The astute programmer will notice that the **VehicleFn** class can't know about the **CarFn** or the **TruckFn** class implementations, so calling `drive()` in the preceding

code actually calls the **VehicleFn**'s `drive()` function rather than the **CarFn** or **TruckFn** `drive()` function. The trick is that while the function set classes define an interface for performing operations on the data, it is actually the data classes that do the real work.

As shown earlier, the data classes would define their own internal virtual `drive()` function. Under the covers, the **VehicleFn**'s `drive()` function would be implemented as follows:

```
virtual void VehicleFn::drive()
{
data->drive();
}
```

Depending on what data object you set the function set to using **setObject()**, the data's appropriate `drive()` function is called. As such, the same function set can operate on different types of data objects.

You may then wonder why they even bother having function sets when you can access the data objects directly and call their member functions. In Maya, you are *never* given access to the data class hierarchy. You are given access only to a class called **MObject**. This class knows about the hierarchy. So with the entire data hierarchy hidden from you, you must use the function set class hierarchy to operate on Maya's hidden data objects. It is through these function set classes that you access all of Maya's internal data.

So the major difference between the original hierarchy design and Maya's is that Maya exposes a function hierarchy only through its function set classes. This is unlike the original design, in which the hierarchy exposes both a data and function hierarchy.

4.2.3 MObject

While the previous section described the Maya class hierarchy in abstract terms, its actual specifics are now covered.

In the previous example, the data objects were known to you. You knew that there was an **Object**, **MotorcycleObj**, **CarObj**, and **TruckObj** class. Since you knew about the different objects, you could then access them directly. In Maya only the root **Object** class is exposed to the developer. All the other classes are hidden. In Maya, the equivalent of your example **Object** class is the **MObject** class.

All data is accessed through the **MObject** class, so this class is used for accessing all the different data types in Maya. It may appear that the **MObject** class itself holds the actual data that is being used. In reality the **MObject** is just a *handle* to another

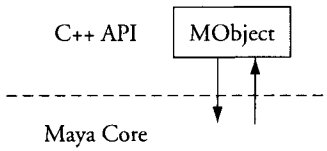


FIGURE 4.6 MObject interface

object inside the core. Since it is just a handle, it can be thought of as a pointer to some other internal data held within the core. Only the core can make use of the pointer. The **MObject** itself doesn't contain any data other than this pointer. This situation is shown in Figure 4.6.

Since the **MObject** effectively contains a *void pointer* (`void *`) to some internal data and the specifics of the internal data are never exposed, you can't convert this pointer into something meaningful in your code. Only Maya's core knows exactly what the pointer refers to. Since the class doesn't hold the data but instead a reference, when you delete or create an **MObject**, you are just deleting and creating a handle. You aren't actually deleting or creating the internal Maya data.

It is very important to understand this, since it can be the source of a lot of confusion. Maya owns the actual internal data and never gives you direct access to it. It instead gives you a handle to the data in the form of an **MObject** object. At no time can you directly delete the internal data, since deleting the **MObject** simply deletes a handle and not the actual data. Maya maintains and controls all of the internal data, whether it be nodes, attributes, or some other Maya data. You are given access to this data through the API, but at no time do you have direct control over it. The data is always accessed and manipulated through the API.

It is important to bear this in mind, since, as with any API, you are just being given an interface to some underlying structure or system. You are never given direct access. The internals of Maya are never directly exposed. An important side effect of this lack of direct access is that you, as a developer, never actually own or have any complete control over any of the Maya nodes or other objects. Maya, in fact, maintains complete and total control over all objects. This immutable fact is so important that it has to be reemphasized:

Maya owns all the data, you own none of it!

4.2.4 MFn Function Sets

With a better understanding of how Maya presents its data, now look at how you can create, edit, and delete that data. A handle is needed to the data before it can be worked on. Maya uses the **MObject** class as this handle. With an **MObject** pointing to the data, the next step is to create a function set that then operates on the data.

As mentioned earlier, all classes prefixed with **MFn** are function sets. Function sets are designed to create, edit, and delete data. To create a **transform** node, the following **MFnTransform** function set would be used:

```
MFnTransform transformFn;
MObject transformObj = transformFn.create();
```

The `transformNodeObj` contains a handle (**MObject**) to the newly created **transform** node. The member functions of the **MFnTransform** object, `dagNodeFn`, can now be called to operate on the **transform** node. Its name can be retrieved using the `name()` function.

```
MString nodeName = transformFn.name();
```

Function sets are organized into a class hierarchy based on what type of data they can operate on. All function set classes are derived from **MFnBase**. The **MFnTransform** function set's ancestors are shown in Figure 4.7.

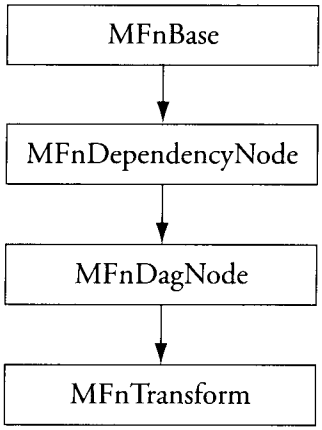


FIGURE 4.7 **MFnTransform** ancestors

Each new derived class adds new functions that can operate on more specific types of data. Since **MFnTransform** is derived from **MFnDagNode**, it can operate on all DAG node objects. Likewise, it is derived from **MFnDependencyNode**, so **MFnTransform** can operate on any dependency node. This hierarchy of functionality allows derived classes to operate on all data that their ancestors can operate on. For instance, the **MFnDagNode** class can operate on the node created by **MFnTransform**, since **MFnDagNode** is a base class of **MFnTransform**.

```
MFnTransform transformFn;
MObject transformObj = transformFn.create();
MFnDagNode dagFn( transformObj );
MString name = dagFn.name();
```

Notice how the `dagFn` function set could be applied to the `transformObj` since the **MObject** that is created points to a **transform** node that is itself derived from a **dagNode**.

So what happens when a function set is asked to operate on an **MObject** that it isn't designed to? The **MFnNurbsSurface** function set is designed to work on NURBS surfaces. The **MFnPointLight** function set operates on point lights. What would happen if the **MFnNurbsSurface** function set were asked to operate on a point light object?

```
MFnPointLight pointLightFn;
MObject pointObj = pointLightFn.create();

MFnNurbsSurface surfaceFn( pointObj );
double area = surfaceFn.area();
```

The `surfaceFn` is asked to operate on the `pointObj` data by having this data passed into its constructor. When the `area()` function is called, Maya checks to see that the data object it is operating on is indeed a NURBS surface. If it isn't, Maya returns an error. Since the object is a point light and not a NURBS surface, an error is returned. In this case, the error isn't checked, so the program continues. The value of the `area` variable is therefore invalid. Error checking and reporting are covered in a later section.

How does Maya know what type of data an **MObject** is referring to? The **MObject** class contains a member function, `apiType()`, that returns the type of object. Each function set class can determine if it is compatible with a given object by calling this function. The **MObject**'s similar function, `hasFn()`, can also be used.

CUSTOM FUNCTION SETS

In the original example, the **Offroad** class was added to the class hierarchy. This was done by simply deriving from the **Car** class. The necessary **Vehicle** and **Car** member functions then had to be implemented in this new class. Since Maya separates the data from the functions, this approach does not work in Maya.

Intuitively, you may consider simply deriving a new function set from an existing one. This new function set would reimplement the functions that it wants to override. Say, for example, you wanted to implement a new NURBS geometric type of your own design. Even though it would be a new type and include its own features, it would have some features in common with Maya's current NURBS implementation. You could potentially derive a new function set class, **MFnMyNurbs**, from the existing **MFnNurbsSurface** class. With this new function set class, it should be possible to operate on your new NURBS shape.

Unfortunately this won't work. Rather than create a new NURBS type, you have in fact simply created a new NURBS function set that can still operate only on existing NURBS surfaces. Recall that an **MObject** points to some data that only Maya knows about. The different **MFn** function sets operate on these **MObjects**. The engineers and designers of Maya know exactly to which data the **MObject** refers and so can manipulate it. Since the exact details of the data pointed to by the **MObject** isn't known to outside developers, there is no way that you can access this data in any meaningful way. Even deriving the new function set class doesn't give you any more access than you had before. As such, the new class can call only the functions that its base class already implemented. You couldn't provide your own functions, since you can't operate on the actual real data. Since the actual Maya data can't be accessed, there is no point in deriving from function sets.

4.3 DEVELOPING PLUGINS

To add your own custom nodes, commands, and so on to Maya, you need to create a plugin. A plugin is a dynamically linked library. Maya loads the plugin (library) at run time to integrate your new functionality.

Under Windows, plugins are standard dynamic link libraries with the special file name extension `.mll` rather than the usual `.dll` extension. Under the various Unix environments, plugins are dynamic shared objects with the file name extension of `.so`. In reality, any file name extension can be used, but using these standard extensions ensures better consistency.

It is assumed that you have installed the Maya development kit. You may recall that this was one of the optional packages available during the installation. The development kit includes all the necessary headers and library files for creating plugins. If it is installed correctly, you should have the following directories under the main `maya` directory:

```
maya_install\include
maya_install\libs
maya_install\devkit
```

If these are missing, you will need to install them before continuing.

When a new version of Maya is released, it may require a change in the plugin development environment. This may mean that a new version of the compiler is needed or that plugins should now be placed in another directory. These and a multitude of other requirements may change from one version to the next. Since the development environment is very specific to a particular version of Maya, any instructions that this book gives may not apply to future versions. Rather than provide obsolete information, the complete instructions for developing plugs for the various platforms are available at the book's companion website, *www.davidgould.com.*

The website includes the most up-to-date instructions for developing plugins for Windows, Irix, Linux, Mac OS X, and any later platforms that run Maya. Having the instructions available online means that they can be quickly updated and amended to ensure that you have the precise set of instructions for the particular version of Maya for which you are developing.

4.3.1 Plugin Development under Windows

For those developing under Windows, this section contains instructions for developing Maya 4.*x* plugins. As mentioned earlier, these instructions may not apply to later versions of Maya, so please consult the companion website for more recent instructions. Maya is supported under Windows NT 4.0, Windows 2000, and Windows XP. Later versions of Windows that are derived from any of these versions should work.

To develop Maya plugins, you'll need a copy of Microsoft Visual C++ 6.0 or later. Be sure to apply the latest Microsoft Visual C++ service packs. If you are familiar with creating Windows DLLs, then creating a plugin should be relatively easy, since it uses exactly the same process.

Be sure to complete the following sections in order, since later sections assume that the previous section has been completed.

SETUP

If Microsoft Visual C++ was already installed when you installed Maya, the **Maya Plug-in Wizard** should have been automatically installed. This wizard does a lot of the plugin setup work for you. It asks you about a series of options, then generates all the necessary workspace, project, and source code files. While it is possible to set up a plugin without it, using the wizard is far more convenient. To create the Hello World plugin, complete the following:

1. Open Microsoft Visual C++.
2. Select **File | New.**
3. In the **New** dialog box, click on the **Projects** tab.
4. Click on the **Maya Plug-in Wizard** from the list of possible project types.
5. Set the plugin name in the **Project name:** field to `HelloWorld`.
6. Set the directory where you'd like to keep the plugin development files in the **Location:** field.
7. Click on **OK.**
8. In the next dialog box, select which version of Maya you'll be targeting.
9. If you've installed the development kit to a custom directory, set the **Location of the Developer Kit** to **custom location.** Ensure that the location is the parent directory of the `devkit` directory. For example, if the `devkit` is a subdirectory under the `C:\Maya4.0` directory, then put `c:\Maya4.0` in the location field and not `c:\Maya4.0\devkit`.
10. Set the **Vendor Name** field to your name.
11. Click on the **Next** button.
12. Leave the plugin type to **Mel Command.**
13. Set the **Maya type name** field to `helloWorld`.
14. Click on the **Finish** button.

    If you clicked on **Next** rather than **Finish**, you could have set the output plugin file name and also specified which other Maya libraries to link in. Most often the default settings are fine, so there is no need to change anything on this page.

15. A listing of all the new project files is then displayed.
16. Click on **OK**.
17. Click on the **Fileview** tab in the **Workspace** area on the left.
18. Expand the **HelloWorld Files** item.
19. Expand the **Source Files** item to reveal the `helloWorldCmd.cpp` file name.
20. Double-click on the `helloWorldCmd.cpp` item to open it.
21. Under the line `#include <maya/MSimple.h>`, add the following:

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

22. Change the following line:

```
setResult( "helloWorld command executed!\n" );
```

to:

```
MGlobal::displayInfo( "Hello World\n" );
```

23. Save the `helloWorldCmd.cpp` file.
24. Build the plugin by selecting **Build | Build helloWorld.mll** from the main menu or by pressing **F7**.

    The resulting plugin file, `helloWorld.mll`, will be located in the `Debug` subdirectory. This subdirectory is under the directory that you specified earlier as the plugin location.

    The plugin file has now been built. It is now ready for use in Maya.

EXECUTION

The next step is to run Maya, then load your new plugin.

1. Open Maya.
2. Select **Windows | Settings/Preferences | Plug-in Manager** from the main menu.

3. Click on the **Browse** button.
4. Locate the directory that contains your plugin file, `helloWorld.mll`.
5. Select the `helloWorld.mll` file.
6. Click on the **Load** button.

   After a short pause Maya will have loaded the plugin. In the **Other Registered Plugins** section you should see a listing with `helloWorld.mll` and the **loaded** check box selected. This indicates that the plugin is currently loaded.
7. Close the **Plug-in Manager** window.
8. Press the back quote (`) key or click in the **Command Line** field.
9. Type in `helloWorld`, then press **Enter**.

   The words `Hello World` are displayed in the Command Feedback line.

   You've now successfully created, loaded, and executed your first Maya plugin.

EDITING

Now that the initial plugin has been created, you'll inevitably want to edit the source code.

1. Return to Visual C++.
2. In the the `helloWorld.cpp` file, change the following line:

```
MGlobal::displayInfo( "Hello World\n" );
```

   to:

```
MGlobal::displayInfo( "Hello Universe \n" );
```

3. Save the `helloWorld.cpp` file.
4. Rebuild the plugin by pressing **F7** or selecting **Build | Build helloWorld.mll** from the main menu.

   During compilation, the following error is displayed.

```
--------------------Configuration: HelloWorld - Win32 Debug-------------
    Compiling...
    helloWorldCmd.cpp
    Linking...
    LINK : fatal error LNK1168: cannot open Debug\helloWorld.mll for
        writing
    Error executing link.exe.

    helloWorld.mll - 1 error(s), 0 warning(s)
```

The plugin file `helloWorld.mll` can't be written to disk. What has happened is that Maya still has the previous `helloWorld.mll` file loaded. As long as this file is loaded in Maya, it can't be overwritten. The plugin must be unloaded prior to recompiling.

5. Return to Maya.
6. Select **Windows | Settings/Preferences | Plug-in Manager...** from the main menu.
7. Click on the **loaded** check box to the right of the `helloWorld.mll` item. This unloads the plugin.
8. Leave the **Plug-in Manager** window open.
9. Return to Visual C++.
10. Build the plugin again by pressing **F7** or selecting **Build | Build helloWorld.mll** from the main menu.

    This time the build should complete successfully.
11. Return to Maya.
12. In the Plug-in Manager window, click on the **loaded** check box next to the `helloWorld.mll` item. This reloads the plugin.
13. Click in the **Command Line**.
14. Type `helloWorld`, then press **Enter**.

    The words, `Hello Universe`, are printed in the **Command Feedback** line.

The process of editing and recompiling a plugin is the same as with any software development. The difference in Maya is that you must ensure that your plugin is unloaded before linking. Once the plugin has been recompiled, you can load it again.

DEBUGGING

Debugging in Visual C++ can be done interactively.

1. Close Maya.
2. Return to Visual C++.

   When the `helloWorld` project was created, the wizard automatically created two configurations: a debug and a release configuration. The debug configuration is used by default. This configuration ensures that the plugin contains debugging information. The steps for setting the active configuration are now covered, even though the project should be already set to the debug configuration.
3. Select **Build | Set Active Configuration...** from the main menu.
4. Select **HelloWorld – Win32 Debug** from the list.
5. Click on the **OK** button.

   Once this configuration is set, you don't have to redo these last steps unless the active configuration is changed.
6. Select **Project | Settings...** from the main menu.
7. Click on the **Debug** tab.
8. Next to the **Executable for debug session:** field, click on the arrow button.
9. Select **Browse...**
10. Locate the Maya executable file, `maya.exe`. This typically is in the `bin` subdirectory under the main Maya directory.
11. Click on the **OK** button.
12. Click on the **Category:** combo box.
13. Select the **Additional DLLs** from the drop-down list.
14. Click in the **Modules...** list.

    An editable field appears.
15. Click on the ... button.

    The **Browse** dialog box is displayed.
16. Set the **Files of type:** to **All Files(*.*)**

17. Locate the `helloWorld.mll` plugin file. It is in the `Debug` subdirectory of the main `helloWorld` project.
18. Click on the **OK** button.
19. Click on the **OK** button to close the **Project Settings** dialog box.

    The project is now set up to begin debugging. These steps have to be completed only once per project.
20. In the `helloWorld.cpp` file, click anywhere in the following line of text:

```
MGlobal::displayInfo( "Hello Universe\n" );
```

21. Press **F9** to set a breakpoint.

    A red dot appears to the left of the line.
22. Press **F5** or select **Build | Start Debug... | Go** from the main menu to start debugging.

    If this is the first time the `maya.exe` program has been run in the debugger, you are told that the executable doesn't contain any debug information.
23. Select the **Do not prompt in the future** check box.
24. Click on the **OK** button.

    Maya then loads. The next steps load the plugin, as before.
25. Select **Windows | Settings/Preferences | Plug-in Manager** from the main menu.
26. Click on the **Browse** button.
27. Locate the directory that contains your `helloWorld.dll` plugin file.
28. Select the `helloWorld.dll` file.
29. Click on the **Load** button.

    After a short pause Maya loads the plugin.
30. Close the **Plug-in Manager** window.
31. Press the back quote (`) key or click in the **Command Line** field.
32. Type in `helloWorld` and then press **Enter**.

The plugin executes and then when it reaches the breakpoint immediately returns you to Visual C++. You can now do the usual debugging tasks of checking variables, stepping through the code, and so on.

RELEASE

As mentioned in the previous section, the `helloWorld` project is automatically created with two configurations: debug and release. When developing and debugging the plugin, the debug configuration should be used. When you decide to finally release the plugin, the release configuration should be used. This configuration ensures that the plugin runs at maximum speed and doesn't contain any unnecessary debugging information. To use the release configuration, complete the following:

1. Select **Build | Set Active Configuration...** from the main menu.
2. Select **HelloWorld – Win32 Release** from the list.
3. Click on the **OK** button.

   The plugin has to be rebuilt using the current configuration.
4. Rebuild the plugin by pressing **F7** or selecting **Build | Build helloWorld.mll** from the main menu.

   The release version of the `helloWorld.mll` file is located in the `Release` subdirectory.

4.3.2 Initialization and Uninitialization

Since a plugin is a dynamic link library, plugins must provide an *entry point* and *exit point*. These are the functions that are called when the plugin is first loaded (entry function) and when it is finally unloaded (exit function), respectively. Under Windows, this is typically handled by the developer's own `DllMain` function. In the `helloWorld` plugin, these entry and exit points were automatically created by using the `DeclareSimpleCommand` macro. This macro automatically creates the command and both the initialization and uninitialization functions for you.

HELLOWORLD2 PLUGIN

Now take a closer look at how plugin initialization and uninitialization happens. In this example, a `helloWorld2` command is created that simply prints out `Hello World`, as before. The complete source code is as follows.

Plugin: HelloWorld2
File: HelloWorld2.cpp

```
#include <maya/MPxCommand.h>
#include <maya/MGlobal.h>
#include <maya/MFnPlugin.h>

class HelloWorld2Cmd : public MPxCommand
{
public:
    virtual MStatus doIt ( const MArgList& )
    { MGlobal::displayInfo( "Hello World\n" ); return MS::kSuccess; }
    static void *creator() { return new HelloWorld2Cmd; }
};

MStatus initializePlugin( MObject obj )
{
    MFnPlugin pluginFn( obj, "David Gould", "1.0" );

    MStatus stat;
    stat = pluginFn.registerCommand( "helloWorld2",
                                     HelloWorld2Cmd::creator );
    if ( !stat )
        stat.perror( "registerCommand failed");

    return stat;
}

MStatus uninitializePlugin( MObject obj )
{
    MFnPlugin pluginFn( obj );

    MStatus stat;
    stat = pluginFn.deregisterCommand( "helloWorld2" );
    if ( !stat )
        stat.perror( "deregisterCommand failed" );

    return stat;
}
```

The first section simply creates the `helloWorld2` command. This section covers some of the basics of creating a command. Section 4.4 covers the creation of commands in far greater detail. The command class contains a simple `doIt()` function that is called when the command is executed. As before, it simply prints out `Hello World` into the Command Feedback line.

```
virtual MStatusdoIt ( const MArgList& )
{ MGlobal::displayInfo( "Hello World\n" ); return MS::kSuccess; }
```

The command also contains a static `creator()` function that allocates a command object and returns it.

```
static void *creator() { return new HelloWorld2Cmd; }
```

Following the definition of the command are the two initialization and uninitialization functions.

```
MStatus initializePlugin( MObject obj )
MStatus uninitializePlugin( MObject obj )
```

Both these functions must be present in any Maya plugin. If they aren't included, the plugin won't link. The `initializePlugin` function takes an **MObject** as input. This **MObject** is a handle to Maya's internal data for plugin types.

```
MStatus initializePlugin( MObject obj )
{
```

The next line creates an **MFnPlugin** object and initializes it with the `obj` variable passed in. Attaching the **MFnPlugin** to the **MObject** allows you to then call the **MFnPlugin** functions that in turn operate on the **MObject**.

```
MFnPlugin pluginFn( obj, "David Gould", "1.0" );
```

The `helloWorld2` command is then registered. Registering the command makes it known to Maya so it can be used. Registration includes giving the name of the command as well as its creator function. The name is the text that you use to call the command, so in this case it is simply `helloWorld2`. The creator function is the static

function that allocates a single instance of the command. This needs to be registered with Maya, since it won't know how to create an instance of your command otherwise.

```
MStatus stat;
stat = plugin.registerCommand( "helloWorld2",
                               HelloWorld2Cmd::creator );
```

The result of the registration is then checked, and if it fails an error is displayed.

```
if ( !stat )
    stat.perror( "registerCommand failed");
```

The initialization function then returns the result:

```
return stat;
}
```

If the return status of the function is not `MS::kSuccess`, then the plugin exits and the dynamic library is automatically unloaded. An error message is also displayed in the Command Feedback line. It is important to note that if the `initializePlugin` function fails, the `uninitializePlugin` function won't be called. As such, all cleanup, in the event `initializePlugin` fails, should be done in the `initializePlugin` function before it returns.

The `uninitializePlugin` function does the reverse of the `initializePlugin` function. It unregisters the command that was registered in the `initializePlugin` function.

```
stat = plugin.deregisterCommand( "helloWorld2" );
```

The function also returns a status indicating whether or not the uninitialization succeeded. If the return status is not `MS::kSuccess`, then the plugin isn't unloaded. It remains loaded in Maya and isn't unloaded until Maya exits.

In this particular example a single command was registered with Maya. In practice there can be an unlimited number of registrations. In later chapters, the registration of other functionality such as custom nodes, and custom data, is covered. However, in all cases, the `initializePlugin` and `uninitializePlugin` functions simply let Maya know what new functionality the plugin provides so that it can be used in the application.

4.3.3 Errors

CHECKING

Checking for errors and consistently handling and reporting them is very important for creating robust and stable applications. Maya provides a consistent error-reporting mechanism through the use of the **MStatus** class. This class defines possible states for the result of a given operation. When an operation fails, the status is set to the appropriate state.

Almost all Maya class functions take an optional pointer to an **MStatus** object. If you supply a pointer to an **MStatus** object, then Maya sets the object to the result of the function call. The complete declaration of the **MFnDependencyNode**'s `name()` function is as follows:

```
MString name( MStatus * ReturnStatus = NULL ) const
```

This function can be called without checking the result as follows:

```
MString dagName;
MFnDagNode dagFn( obj );
dagName = dagFn.name();
```

To do correct error checking, it is important to determine whether the function succeeded or failed. To do this, a pointer to an **MStatus** object is passed to the `name()` function.

```
MStatus stat;
dagName = dagFn.name( &stat );
if( !stat )
    MGlobal::displayError( "Unable to get dag name" );
```

The result of the function call is stored in the **stat** object. It is checked to determine whether it is set to a failure; if so, an appropriate action can be taken. In this example an error message is displayed. This example shows the most commonly used error-checking methodology in plugins.

There are a variety of ways to check the resulting **MStatus**. In addition to the `if( !stat )` used previously, the `error()` function can be used.

```
if( stat.error() )
  ... // error
```

The exact status can be retrieved from the object and compared against a particular status code. To test whether the status is set to the `MS::kSuccess` code, use the following:

```
if( stat.statusCode() != MS::kSuccess )
  ... // error
```

or more simply:

```
if( stat != MS::kSuccess )
  ... // error
```

Unfortunately, the error-reporting mechanism that Maya uses puts a lot of the emphasis on the developer to be vigilent in checking for errors. More often than not, the developer has to check the result of *every* Maya function called. This can become tiresome, and many programmers tend to be sporadic with their checking. However, it is extremely important to maintain consistent error checking throughout your plugin. In a series of function calls that don't check for failed calls, having a single check makes it very difficult to pinpoint where the error occurred. Bugs and runtime issues are resolved a lot faster if every piece of code is checked.

To help make writing error-checking code easier, the following macros can be used:

```
inline MString MyFormatError( const MString &msg, const MString
                              &sourceFile, const int &sourceLine )
{
    MString txt( "[MyPlugin] " );
    txt += msg;
    txt += ", File: ";
    txt += sourceFile;
    txt += " Line: ";
    txt += sourceLine;
    return txt;
}

#define MyError( msg ) \
    { \
    MString __txt = MyFormatError( msg, __FILE__, __LINE__ ); \
    MGlobal::displayError( __txt ); \
    cerr << endl << "Error: " << __txt; \
    } \
```

```
#define MyCheckBool( result ) \
    if( !(result) ) \
      { \
      MyError( #result ); \
      }

#define MyCheckStatus( stat, msg ) \
    if( !stat ) \
      { \
      MyError( msg ); \
      }

#define MyCheckObject( obj, msg ) \
    if( obj.isNull() ) \
      { \
      MyError( msg ); \
      }


#define MyCheckStatusReturn( stat, msg ) \
    if( !stat ) \
      { \
      MyError( msg ); \
      return stat; \
      }
```

You notice that these macros include the source file and line number in the error message. They also automatically output the error message to the standard error stream. The macros can be used as follows:

```
MObject dagObj = dagFn.object();
MyCheckObject( dagObj, "invalid dag object" );
```

Another example includes the following:

```
MStatus stat;
dagName = dagFn.name( &stat );
MyCheckStatusReturn( stat, "Unable to get name" );
```

It is important to note that many of the examples in this book don't do consistent error checking. The reason for this is that this book attempts to not overload or obfuscate the code. With all the error checking removed, the core concepts are presented more clearly to the reader. It is due to this goal of maintaining simplicity and brevity that the example code contains very minimal error checking. When developing plugins, however, the exact opposite should be done. Everything should be checked, and this should be done in a consistent manner. Your plugins will be more robust and stable as a result.

REPORTING

The **MStatus** class also provides some additional functions for error reporting. The `errorString()` function returns a string corresponding to the current error code. The `perror()` function allows you to print an error message to the current `stderr` stream. An example use of these functions is as follows:

```
if( stat.error() )
    stat.perror( MString("Unable to get name. Error: ") +
        stat.errorString() );
```

Like MEL, the C++ API has warning and error-reporting functions. The general method for reporting error messages is to use the **MGlobal**'s static `displayError()` function, as follows:

```
MGlobal::displayError( "object has been deleted" );
```

The error message is displayed in red in the Command Feedback line. To display a warning, use the `displayWarning()` function.

```
MGlobal::displayWarning ( "the selected object is of the wrong type" );
```

The warning message is displayed in magenta in the Command Feedback line. It is also possible to display general information by using the `displayInfo()` function.

```
MGlobal::displayInfo( "select a light, then try again" );
```

While there are a variety of ways of notifying the user of errors and warnings, the `MGlobal::displayError()` and `MGlobal::displayWarning()` are the ones that should be consistently used. They still work when Maya is run in batch mode,

without an interface. If you want to display the result of an **MStatus** comparison, then use the **MStatus**'s `errorString()` function combined with the `MGlobal::displayError()` function.

```
if( !stat )
  {
  MGlobal::displayError( MString("Unable to get name. Error: ") +
                                  stat.errorString() );
  return stat;
  }
```

Don't use pop-up windows to display errors or warnings. You can't know that your command isn't being called many times, so displaying a pop-up window each time may only frustrate the user. Also, the windows won't appear when Maya is being run in batch mode, so there will be no error reporting for the user to see.

Also, outputting errors and warnings to the standard error stream should really be used only for debugging. Users won't be able to see these errors when they are running Maya with the graphical interface. You should design your plugin so that this debugging and testing information isn't displayed in the final release version.

The **MGlobal** class also provides some error-logging features. While they aren't always used, you may find them handy. Refer to the **MGlobal** class for a complete list.

INTEGRATION

When you begin writing your own plugin functions, it is often best to adopt the Maya error-reporting mechanism. Your classes will then work in a consistent manner with other Maya C++ API classes. At its simplest you can provide an additional status pointer in your functions, in the same way that Maya does. The following function demonstrates how to do this:

```
int myNumPoints( points *data, MStatus *returnStatus = NULL );
```

In the function you simply have to check whether the pointer is valid and, if it is then returned, whether the function succeeded or failed.

```
int myNumPoints( points *data, MStatus *returnStatus )
{
int num = 0;
MStatus res = MS::kSuccess;
if( data )
  num = data->nElems();
else
  res = MS::kFailure;
if( returnStatus )
  *returnStatus = res;
return num;
}
```

Some Maya functions don't take a pointer to an **MStatus** object but instead return an **MStatus**. The following example demonstrates this:

```
MStatus myInitialize()
{
bool ok;
... // do initialization
return ok ? MS::kSuccess : MS::kFailure;
}
```

Consistently designing your functions to return **MStatus**, either directly or through a pointer, ensures that your functions behave like Maya's. That way the error-checking and reporting approach you put in place will be consistent throughout your plugin. It also means that other programmers that read or use your code won't have to use different error-checking and reporting schemes.

4.3.4 Loading and Unloading

As mentioned earlier, a plugin must be loaded into Maya before its functionality is made available. Likewise, when a plugin is no longer need, it can be unloaded. Also, when a plugin is being recompiled, it must be unloaded from Maya.

Since the tasks of loading and unloading a plugin are done so often, it is best to try to automate them. The following instructions detail how you can create two shelf buttons for unloading and loading a plugin.

1. Open Maya.
2. In the **Script Editor**, type the following, but don't execute it. Set the `$pluginFile` string to the complete path of the plugin you are developing.

```
{
string $pluginFile = "c:\\helloWorld\\Debug\\helloWorld.mll";
if( `pluginInfo -query -loaded $pluginFile` &&!
    `pluginInfo -query -unloadOk $pluginFile`)
  file -f -new;
unloadPlugin helloWorld.mll;
}
```

   Notice that the MEL statements are enclosed in parentheses. This is to create a temporary block so that any variables created are local and therefore don't pollute the global namespace.

   The `pluginInfo` command is called with `-query -unloadOK` flags to determine whether the plugin can be successfully unloaded. The most common reason that a plugin can't be unloaded is that one of its commands or nodes is still being used in the scene. Even if you delete the nodes, they can still exist in Maya's undo queue. In this case, the easiest solution is to create a new scene. This manually removes all commands and nodes. This is done using the `file -f new` statement. Lastly, the plugin is actually unloaded using the `unloadPlugin` command.

3. Select the text, then drag it to the **Shelf**.

   A shelf button is created. This shelf button is referred to as the **Unload** button.

4. In the **Script Editor**, type the following, but don't execute it. Once again, set the `$pluginFile` variable to the path of the plugin you are developing.

```
{
string $pluginFile = "c:\\helloWorld\\Debug\\helloWorld.mll";
loadPlugin $pluginFile;
}
```

   The plugin is simply loaded using the `loadPlugin` command.

5. Select the text, then drag it to the **Shelf**.

   A shelf button is created. This shelf button is referred to as the **Load** button.

With these two shelf buttons set up, the process of loading and unloading your plugin is a lot faster. They can be used as follows:

- Write the plugin.
- Click on the **Load** button.
- Test the plugin.
- Click on the **Unload** button.
- Edit the source code and recompile.
- Click on the **Load** button.
- Test the plugin.
- .... Repeat.

In addition to the method mentioned previously, you can have your plugin loaded automatically when Maya starts by doing the following:

1. Set the **auto load** check box for the plugin in the **Plug-in Manager**.
2. Include the path to the plugin directory in the `MAYA_PLUG_IN_PATH` environment variable, or use a `maya.env` file to set the `MAYA_PLUG_IN_PATH` environment variable.
3. Restart Maya, and your plugin loads automatically.

4.3.5 Deployment

If you need to make your plugin available to a wider number of users, then you need to decide on a deployment scheme. The method for deploying plugins will most likely be the same as that used for deploying scripts.

1. If you have a work environment in which all the users can access a central server, the task of deployment is greatly simplified. Simply create a shared directory on a server, for example, `\\server\mayaPlugins`. This example uses the UNC (Universal Naming Code) format, but use whatever path format your network requires.

   If you don't have a central server but instead each user has his or her own local machine, then create a directory on each machine, for example, `c:\mayaPlugins`.

This directory is referred to as the *plugin_directory*.

2. Copy the plugin binary files to this directory.
3. On each of the users' machines, set the `MAYA_PLUG_IN_PATH` environment variable to include the path to the server's directory.

```
set MAYA_PLUG_IN_PATH=$MAYA_PLUG_IN_PATH;plugin_directory
```

Alternatively you could update each user's `maya.env` file to contain the environment variable setting:

```
MAYA_PLUG_IN_PATH=$MAYA_PLUG_IN_PATH;plugin_directory
```

While it is possible to store your plugins directly to the *maya_install*`\plugins` directory, this isn't advisable. It is always best to keep your plugins separate from the standard Maya plugins. This prevents confusion and helps to localize and update your plugins.

UPDATING

If a user is already running Maya when you deploy your plugins, the user won't automatically use the latest version. In fact, most operating systems don't allow you to overwrite the old plugin file since Maya is still using it. To correctly update, the user either needs to quit Maya then restart or to unload then reload the plugin. If the scene contains data specific to the plugin, then it won't unload. At that point the easiest thing to do is to restart Maya.

4.4 COMMANDS

When writing MEL scripts, you will most likely be using a lot of Maya's commands. Through the C++ API it is possible to add your own custom commands that can be used exactly like Maya's native commands. Like Maya's commands, they can be called from MEL scripts or anywhere a command can be called. At its barest minimum, a command is simply a function that gets called when the command is executed. In the previous section, the `helloWorld` command was created. When the command was executed, it simply printed out `Hello World`.

This section covers the creation of more-complex commands. Commands can include a lot more features, such as taking multiple inputs (arguments), providing help, and working with Maya's undo and redo mechanism.

4.4.1 Creating Commands

In order to understand how to create more-complex commands, a command named `posts` is covered in detail. It takes a curve and generates a number of posts (cylinders) along the curve. Figure 4.8 shows the original "guide" curve. The second figure, Figure 4.9, shows the result of executing the `posts` command.

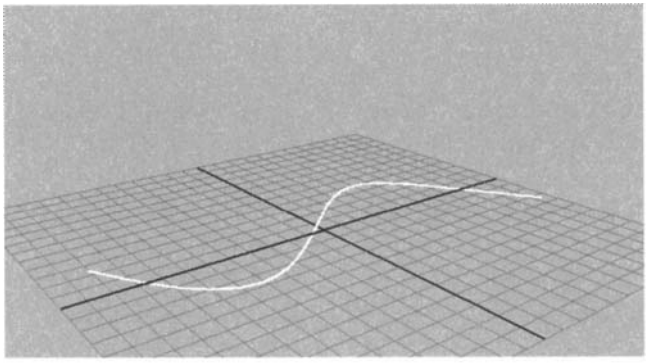
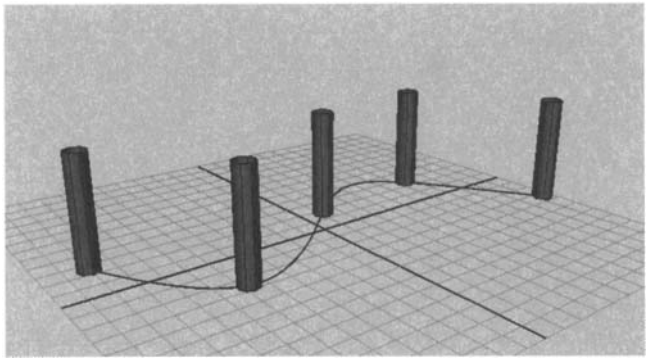


FIGURE 4.8 Guide curve



4.4.2 Posts1 Plugin

This section begins with a simple version of the `posts` command and progressively builds on it. As such, the first version of the command is named `posts1`.

1. Open the **Posts1** workspace.
2. Compile it, and load the resulting `posts1.mll` plugin in Maya.
3. Open the **PostsCurve.ma** scene.
4. Select the curve.
5. In the **Command Line**, type the following, then press **Enter**.

```
posts1
```

Five cylinders are created along the curve.

The source code for the command is now covered in detail.

Plugin: Posts1
File: postsCmd1.cpp

```
class Posts1Cmd : public MPxCommand
{
public:
    virtual MStatus doIt ( const MArgList& );
    static void *creator() { return new Posts1Cmd; }
};

MStatus Posts1Cmd::doIt ( const MArgList & )
{
    const int nPosts = 5;
    const double radius = 0.5;
    const double height = 5.0;

    MSelectionList selection;
    MGlobal::getActiveSelectionList( selection );

    MDagPath dagPath;
    MFnNurbsCurve curveFn;
    double heightRatio = height / radius;
```

```
    MItSelectionList iter( selection, MFn::kNurbsCurve );
    for ( ; !iter.isDone(); iter.next() )
          {
          iter.getDagPath( dagPath );
          curveFn.setObject( dagPath );

          double tStart, tEnd;
          curveFn.getKnotDomain( tStart, tEnd );

          MPoint pt;
          int i;
          double t;
          double tIncr = (tEnd - tStart) / (nPosts - 1);
          for( i=0, t=tStart; i < nPosts; i++, t += tIncr )
               {
               curveFn.getPointAtParam( t, pt, MSpace::kWorld );
               pt.y += 0.5 * height;

               MGlobal::executeCommand( MString( "cylinder -pivot ") +
                    pt.x + " " + pt.y + " " + pt.z + " -radius 0.5
                    -axis 0 1 0 -heightRatio " + heightRatio  );
               }
          }

    return MS::kSuccess;
}

MStatus initializePlugin( MObject obj )
{
    MFnPlugin pluginFn( obj, "David Gould", "1.0" );
    MStatus stat;
    stat = pluginFn.registerCommand( "posts1", Posts1Cmd::creator );
    if ( !stat )
         stat.perror( "registerCommand failed");
    return stat;
}
```

```
MStatus uninitializePlugin( MObject obj )
{
    MFnPlugin pluginFn( obj );
    MStatus stat;
    stat = pluginFn.deregisterCommand( "posts1" );
    if ( !stat )
         stat.perror( "deregisterCommand failed" );
    return stat;
}
```

The first step is to define the new command class, **Post1Cmd**. This class is derived from **MPxCommand**. All commands are derived from this class.

```
class Posts1Cmd : public MPxCommand
{
public:
       virtual MStatus doIt ( const MArgList& );
       static void *creator() { return new Posts1Cmd; }
};
```

The only two functions that need to be implemented are `doIt` and `creator`. The `doIt` function is a virtual function that is called when the command is executed. This function does the real work of the command. It performs whatever operation the command should do to produce its result.

The `creator` function is used to create an instance of the command. You'll notice that it is a static function, so it can be called without needing an instance of the class. In fact, there is no requirement that it be a static member function of the class or even named `creator`. The creation function is simply a standard function that returns an allocated instance of the command.

```
static void *creator() { return new Posts1Cmd; }
```

This function is registered with Maya in the `initializePlugin` function so that Maya knows how to create an instance of the command. When Maya is asked to execute the command, it first allocates an instance of the command using the `creator` function. Maya then calls the instance's `doIt` function. Each time you execute a command, Maya repeats these steps. This means that a new instance of the command is created each time a request is made to execute the command. The

reason for this behavior is related to Maya's undo/redo mechanism. This mechanism is explained shortly.

Since the `doIt` function does all the real work, it is now covered in greater detail. The first section initialized the number of posts and their radius and height.

```
MStatus Posts1Cmd::doIt ( const MArgList & )
{
        const int nPosts = 5;
        const double radius = 0.5;
        const double height = 5.0;
```

A list of currently selected objects is then created. The `selection` object is used to hold the list of objects.

```
MSelectionList selection;
MGlobal::getActiveSelectionList( selection );
```

Unfortunately the **cylinder** primitive doesn't allow you explicitly to set its height. Instead a cylinder's height is based on its `heightRatio` value. This is the ratio of the height of the cylinder to its width. The `heightRadio` is calculated based on the desired height and radius of the cylinder.

```
double heightRatio = height / radius;
```

The list of selected objects is iterated over. This is done using an **MItSelectionList** object. Since the command is interested only in NURBS curves, a *filter* is specified that excludes all other objects that aren't a NURBS curve. The `iter` object is initialized with the `selection` object that you set up earlier.

```
MItSelectionList iter( selection, MFn::kNurbsCurve );
```

Iterating over the NURBS curves is then done with the following loop. Since the iterator traverses all the selected NURBS curves, you can apply the `posts1` command to multiple curves and it creates cylinders for each.

```
for ( ; !iter.isDone(); iter.next() )
      {
```

In order to identify the current NURBS curve, its complete DAG path is retrieved.

```
iter.getDagPath( dagPath );
```

The NURBS curve function set, **MFnNurbsCurve**, is attached to the DAG path. This specifies that all further function set operations will be applied to the object given by the DAG path.

```
curveFn.setObject( dagPath );
```

A NURBS curve is a parametric primitive. The curve can be evaluated at a given parametric value, *t*, resulting in a point that lies on the curve. The value *t* typically ranges from 0 to 1; however, it is possible that the curve has an arbitrary parametric range. The following code gets the start and end of that range:

```
double tStart, tEnd;
curveFn.getKnotDomain( tStart, tEnd );
```

The command creates `nPosts` number of posts along the length of the curve. The parametric range is divided into the number of desired posts. Because there needs to be a post at the start of the curve, as well as its end, the curve is made to have `nPosts-1` divisions.

```
double tIncr = (tEnd - tStart) / (nPosts - 1);
```

The next step is really the core of the function. For each step in the value *t*, a post is planted along the curve.

```
for( i=0, t=tStart; i < nPosts; i++, t += tIncr )
    {
```

The `getPointAtParam` function returns a point on the curve, given a parametric value. In this instance, the point is requested to be in world coordinates (`MSpace::kWorld`) rather than the default object coordinates (`MSpace::kObject`). The reason for this is that the posts should be placed in their final world position, irrespective of the transformation hierarchy of the curve.

```
curveFn.getPointAtParam( t, pt, MSpace::kWorld );
```

By default, the pivot point of a cylinder is at its center. Since the base of the cylinder should rest on the curve, the pivot point needs to be moved up by half the height.

```
pt.y += 0.5 * height;
```

Now with all the various parameters to the command prepared, the necessary MEL commands can be executed to do the actual cylinder creation. The `cylinder` command is used to create the cylinder with the given pivot (`-pivot`), radius (`-radius`), and height ratio (`-heightRatio`). A MEL statement can be executed from C++ by using the `MGlobal::executeCommand` function. It takes a string containing the MEL statements to be executed.

```
MGlobal::executeCommand( MString( "cylinder -pivot ") + pt.x + " " + pt.y
    + " " + pt.z + " -radius " + radius + " -axis 0 1 0 -heightRatio " +
    heightRatio  );
}
```

In fact, it is quite common to see the execution of MEL statements inside a C++ plugin. There are many times when doing so makes more sense than trying to perform the same operation with many C++ API calls. There are also some MEL commands that don't exist in the C++ API, so you'll have no choice but to execute the MEL commands.

4.4.3 Adding Arguments

The `posts1` command is quite good, but it doesn't let you specify a different number of cylinders along the curve or the radius and height of each cylinder. Currently those parameters are set to fixed values inside the `doIt` function. The next command, `posts2`, extends the current command to allow for the various parameters to be set on the command line.

1. Open the **Posts2** workspace.
2. Compile it, and load the resulting `posts2.mll` plugin in Maya.
3. Open the **PostsCurve.ma** scene.
4. Select the curve.
5. In the **Command Line**, type the following, then press **Enter**.

```
posts1 -number 10 -radius 1
```

Ten cylinders are created along the curve. Each of the cylinders is now wider than before.

Notice that the height of the posts is the same as before. Since the height wasn't specified, the command uses a default value. The specific changes to the command are now covered.

4.4.4 Posts2 Plugin

Plugin: Posts2
File: posts2Cmd.cpp

```
...
MStatus Posts2Cmd::doIt ( const MArgList &args )
{
    int nPosts = 5;
    double radius = 0.5;
    double height = 5.0;

    unsigned index;
    index = args.flagIndex( "n", "number" );
    if( MArgList::kInvalidArgIndex != index )
        args.get( index+1, nPosts );

    index = args.flagIndex( "r", "radius" );
    if( MArgList::kInvalidArgIndex != index )
        args.get( index+1, radius );

    index = args.flagIndex( "h", "height" );
    if( MArgList::kInvalidArgIndex != index )
        args.get( index+1, height );
...
```

Since the only major change is at the start of the `doIt` function, just that portion is covered. The `doIt` function itself hasn't changed. It still takes a reference to an **MArgList** as its only input. In the previous command, `posts1`, this function argument was simply ignored. The **MArgList** class used to hold the list of arguments that are passed to a command. From this class the parameter *flags* and *values* can be retrieved. The flags are the parameter names specified with a dash character before

them. The `-radius` flag specifies the radius parameter. The argument following the flag is typically the value to assign to the parameter. In this case it is 1.

Notice that the parameters (`number`, `radius`, `height`) are no longer fixed values. Instead they are initialized to their respective default values. In the event that one of the parameters isn't explicitly set on the command line, its default value is used. In the current example, the height parameter wasn't specified on the command line, so it uses the default value of 5.0.

```
int nPosts = 5;
double radius = 0.5;
double height = 5.0;
```

Since almost all parameters have a default value, specifying a value for them on the command line is optional. As such, the existence of a parameter flag must be checked for. The `flagIndex` function returns the index of the argument containing a given flag. Flags come in two forms: short and long. Either form can be used, so to test for the **number** parameter flag, both `"n"` and `"number"` were given to the `flagIndex` function.

```
index = args.flagIndex( "n", "number" );
```

If the flag hasn't be set on the command line, then the `index` is set to `MArgList::kInvalidArgIndex`.

```
if( MArgList::kInvalidArgIndex != index )
```

If the flag is valid, then the argument following the flag is retrieved. This argument is at `index+1`. The value is stored in the appropriate variable.

```
args.get( index+1, nPosts );
```

The same steps are completed for all the remaining flags. If flags have been set, their values are taken from the command line. Otherwise the default value is used.

Those familiar with the C language constructs `argc` and `argv` will now notice the similarities in functionality provided by the **MArgList** class. It must be noted, however, that the first argument in **MArgList** is the first argument to the command and not the name of the command, as it is with `argv`.

In this example the number of command parameters is relatively small, so it was easier just to check for each parameter flag individually. More sophisticated commands can have many more parameters, so using the **MArgList** class can become rapidly cumbersome. Maya provides another more advanced mechanism for parsing argument flags and getting their values.

4.4.5 Posts3 Plugin

The `posts2` command is now altered to make use of the **MSyntax** and **MArgDatabase** classes. Both these classes give greater flexibility in the number and type of parameters that can be used. Also they provide better argument type checking. In fact, they are the preferred classes to use when writing robust commands. The **MArgList** class is therefore rarely used and, when it is, only for simple commands.

- Open the **Posts3** workspace.

Plugin: Posts3
File: posts3Cmd.cpp

The **Posts3Cmd** class is based on the **Posts2Cmd** class. The following static function was added. Its job is to return an **MSyntax** object for the command.

```
static MSyntax newSyntax();
```

The **MSyntax** class provides a convenient means for specifying all the possible parameters to your command. The following code defines a set of flags, in both short and long form, that the command will accept.

```
const char *numberFlag = "-n", *numberLongFlag = "-number";
const char *radiusFlag = "-r", *radiusLongFlag = "-radius";
const char *heightFlag = "-h", *heightLongFlag = "-height";
```

The data types of the flags' arguments are then specified. Using the `addFlag` function, it is possible to add up to six different types of data that follow the flag. In this example the **number** parameter accepts a single `long`, while both the **radius** and **height** parameters accept a single `double`.

```
MSyntax Posts3Cmd::newSyntax()
{
    MSyntax syntax;
    syntax.addFlag( numberFlag, numberLongFlag, MSyntax::kLong );
    syntax.addFlag( radiusFlag, radiusLongFlag, MSyntax::kDouble );
    syntax.addFlag( heightFlag, heightLongFlag, MSyntax::kDouble );
    return syntax;
}
```

The start of the `doIt` function is as before.

```
MStatus Posts3Cmd::doIt ( const MArgList &args )
{
    int nPosts = 5;
    double radius = 0.5;
    double height = 5.0;
```

The `doIt` function now makes use of the **MArgDatabase** class to parse and separate the different flags and their values. The **MArgDatabase** object is initialized with `syntax()` and the **MArgList** object. The `syntax()` function returns the syntax object for the given command. In this case, it is the prepared **MSyntax** object from the `newSyntax()` function that is returned from the `syntax()` function.

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

Each flag is checked in turn to see if it has been set. If it has, then its value is retrieved from the command line and the associated variable is set.

```
if( argData.isFlagSet( numberFlag ) )
    argData.getFlagArgument( numberFlag, 0, nPosts );

if( argData.isFlagSet( radiusFlag ) )
    argData.getFlagArgument( radiusFlag, 0, radius );

if( argData.isFlagSet( heightFlag ) )
    argData.getFlagArgument( heightFlag, 0, height );
...
```

In order for Maya to know that you will be using your own custom **MSyntax** object, you need to signal this. In `initializePlugin`, the `registerCommand` function is called with an additional parameter, the `newSyntax` function.

```
...
    stat = pluginFn.registerCommand( "posts3",
                                     Posts3Cmd::creator,
                                     Posts3Cmd::newSyntax );
...
```

While it may not be completely apparent from this example that the **MSyntax** and **MArgDatabase** are worth using, in practice they offer a lot more functionality than **MArgList**. The **MSyntax** class allows you to specify that certain object types can be used as parameters. The current selection can be automatically included in the list of command arguments when using the **MSyntax** class. Overall, it is a far more robust and powerful means of parsing, extracting, and verifying arguments to your command.

4.4.6 Providing Help

Almost all of Maya's commands support some form of assistance. This assistance is usually in the form of some help text that explains what the command does and what its parameters are. This basic help functionality is provided through the `help` command. This command prints out a short, concise help description for any given command.

- In the **Script Editor,** type the following, then execute it.

```
help sphere;
```

The result is as follows:

```
// Result:
```

```
Synopsis: sphere [flags] [String...]
Flags:
   -e -edit
   -q -query
  -ax -axis                  Length Length Length
 -cch -caching               on|off
  -ch -constructionHistory   on|off
   -d -degree                Int
 -esw -endSweep              Angle
  -hr -heightRatio           Float
   -n -name                  String
 -nds -nodeState             Int
 -nsp -spans                 Int
   -o -object                on|off
   -p -pivot                 Length Length Length
  -po -polygon               Int
   -r -radius                Length
   -s -sections              Int
 -ssw -startSweep            Angle
 -tol -tolerance             Length
  -ut -useTolerance          on|off
//
```

A complete list of all the command's flags and their acceptable data types is displayed. In addition to working for Maya's built-in command, the `help` command can be used on commands that you write.

AUTOMATIC HELP

As long as you provide an **MSyntax** object, which is the case since the new `newSyntax()` function has been added, the `help` command can determine what flags and values your command accepts. It automatically generates a listing of those flags and values.

1. Open the **Posts3** workspace.
2. Compile it, and load the resulting `posts3.mll` plugin in Maya.

3. In the **Script Editor**, type the following, then execute it.

```
help posts3;
```

The following help text is displayed:

```
// Result:

Synopsis: posts3 [flags]
Flags:
   -h -height  Float
   -n -number  Int
   -r -radius  Float

//
```

POSTS4 PLUGIN

In addition to the quick help that is automatically provided by the `help` command, you can also create and display your own help text. In the following example a help flag, `-h/-help`, is added to the command. The **height** parameter's short form flag was renamed from `-h` to `-he` in order to prevent any conflicts. A parameter's short form flag can have up to three characters.

1. Open the **Posts4** workspace.
2. Compile it, and load the resulting `posts4.mll` plugin in Maya.
3. In the **Script Editor**, type and then execute the following:

```
posts4 -h;
```

The following help text is displayed:

```
// Result:
The posts4 command is used to create a series of posts
(cylinders) along all the selected curves.
It is possible to set the number of posts, as well as
their width and height.
For further details consult the help documentation.
For quick help instructions use: help posts4 //
```

Implementing your own custom help requires just a few simple additions to the existing command.

Plugin: Posts4
File: Posts4Cmd.cpp

A new help flag is added.

```
const char *helpFlag = "-h", *helpLongFlag = "-help";
```

In the `newSyntax()` function, the new flag is added to the **MSyntax** object.

```
syntax.addFlag( helpFlag, helpLongFlag );
```

The help text that will be displayed is then defined:

```
const char *helpText =
"\nThe posts4 command is used to create a series of posts
    (cylinders) along all the selected curves."
"\nIt is possible to set the number of posts, as well as
    their width and height."
"\nFor further details consult the help documentation."
"\nFor quick help use: help posts4";
```

You are free to set the help text to anything you'd like. It is important however to remember that the automatic help should be a succinct, shorthand help, so keep it small. The custom help can include any additional information that you feel is necessary. If the help instructions are much longer or need specialized images or animations, then it is best simply to point the user to the relevant documentation.

The last required change is the addition of several lines to the `doIt()` function. As with the other flags, the help flag is tested to see if it is set. If it is, then the result is set to the help text defined earlier. The command then returns immediately, indicating its success.

```
if( argData.isFlagSet( helpFlag ) )
  {
  setResult( helpText );
  return MS::kSuccess;
  }
```

Commands can return a variety of results. When calculating the length of a curve, a command returns a single distance value. When querying the translation of a transform node, a series of three doubles is returned, one for each of the axes (*x*, *y*, *z*). The number and type of the results are defined by the command. In this example, the `posts4` command returned a string containing the help text.

Since the command returned a string, you can store this in a variable. The following MEL statements store the result of asking the command for help:

```
string $text = `posts4 -h`;
print $text;
```

Since the help text is now stored in the `$text` variable, you could, for example, store it in a file or display it in a window.

4.4.7 Undo and Redo

A very important topic that needs to be understood when you write commands is that your command must be compatible with Maya's undo and redo mechanism. This compatibility is extremely important for your command to work seamlessly in Maya. In fact, a command that changes the scene in some way but doesn't provide a way to undo those changes is actually illegal. Such a command can cause Maya's state to be undetermined if the user attempts to undo it.

1. Open the **Posts4** workspace.
2. Compile it, and load the resulting `posts4.mll` plugin in Maya.
3. Open the **PostsCurve.ma** scene.
4. Select the curve.
5. In the **Command Line**, type and then execute the following:

   ```
   posts4
   ```

6. Select **Edit | Undo** from the main menu.

   Nothing happens. Since the `posts4` command doesn't include support for undo or redo, the command can't be undone.

MAYA'S UNDO/REDO MECHANISM

Maya supports the ability to undo a series of commands through its undo mechanism. When a MEL command is executed, the state of the scene is likely to be changed in some way. Undoing the command is the same as reversing those changes and thereby returning the scene to the scene state before the command was executed. As such, Maya's undo mechanism allows you to undo the operations of a series of commands.

Undo Queue

Maya maintains a queue of the most recently run commands. The size of this queue defines the number of commands that can be undone.

1. Select **Window | Settings/Preferences | Preferences...** from the main menu.
2. Click on the **Undo** item in the **Settings** section.
3. Ensure that **Undo** is set to **On**.
4. Set the **Queue** setting to **Finite** and the **Queue Size** to **30**.

If the **Queue** is set to **Infinite**, then there is no limit to the number of commands that can be undone. However, this setting increases Maya's memory usage since Maya must store every one of the commands executed from the beginning of the session in case you request that they be undone later. Setting the queue to a reasonable, but finite, size is the best compromise between memory usage and the need to undo a series of commands.

Assume that the undo queue was set to hold three commands, that is, the **Queue Size** was set to 3. The queue is initially empty. Figure 4.10 shows the empty queue. The last command points to the last command executed. The queue head is the front

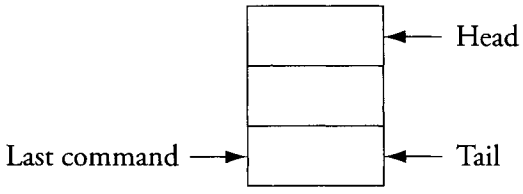


FIGURE 4.10 Empty undo queue

of the queue, while the queue tail is the end of the queue. As with any queue, items are added to the tail of the queue and eventually make their way to the head.

Now see the result of executing a few MEL commands.

```
sphere;
move 10 0 0;
scale 0.5 0.5 0.5;
```

The resulting undo queue is shown in Figure 4.11.

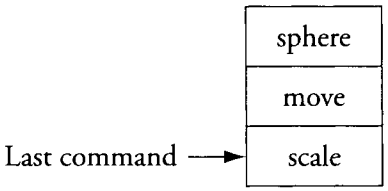


FIGURE 4.11 Full undo queue

The `sphere` command is added to the tail of the queue. The `move` command is added next. The `sphere` command moves up one slot making room for the `move` command. The `scale` command is then added to the tail, pushing the other two items up one slot. The undo queue is now full.

What happens to the undo queue if another command is executed?

```
rotate 45;
```

The `rotate` command is added to the tail of the queue, pushing all the other items up one slot. Since there is no empty space at the head of the queue the `sphere` command is removed from the queue. When the queue is full, the oldest item in the queue is the first to be removed. Figure 4.12 shows the undo queue after the `rotate` command is executed.

A consequence of being removed from the queue is that the command is now deleted. As such, the `sphere` command is deleted. It is now impossible to undo the `sphere` command, since it no longer exists in the undo queue. It may be clear now that setting the size of the undo queue affects how many commands can be undone. It is impossible to undo more commands than are currently available in the undo queue.

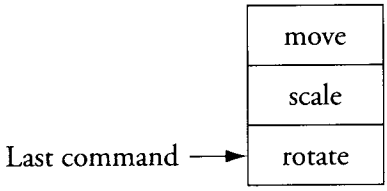


FIGURE 4.12 Undo queue after `rotate` command

What happens if you now select undo? The `rotate` command is undone. The last command is set to the `scale` command. If the user, once again, selects undo, the `scale` command is undone and the last command is set to the `move` command. The resulting undo queue is shown in Figure 4.13.

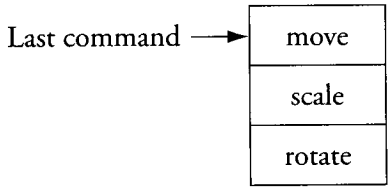


FIGURE 4.13 Undo queue after two undos

The scene is now in the state it was before the `scale` and `rotate` commands were executed. Since you can undo commands, it follows logically that you should also be able to redo commands. If users decided that they didn't want to undo the scale, they would simply select redo. The `scale` command would be redone, and the last command would be set to the `scale` command. Figure 4.14 shows the result.

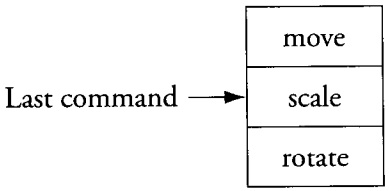


FIGURE 4.14 Undo queue after redo

You can continue redoing commands for as many commands as remain in the queue. Instead of redoing, another command is executed:

```
cylinder;
```

Since the last command, `scale`, is midway in the undo queue and a new command has been executed, all the subsequent commands are removed. Therefore, the `rotate` command is removed. If the queue were larger, all commands after the `rotate` command would also be removed. The `cylinder` command is then added to the end of the queue. The resulting undo queue is shown in Figure 4.15.

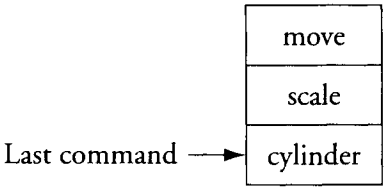


FIGURE 4.15 Undo queue after `cylinder` command

It is important to understand the reason the later commands were deleted. Remember that the undo queue can hold only one possible history path. When several undos were done until you reached the `scale` command, you moved up a history path. If the queue actually allows more than one path, when the `cylinder` command is executed, a branch in the history is created. There would now be two possible paths to follow after the `scale` command. One branch would result in the original `rotate` command, and the other branch would result in the `cylinder` command. Since there is no mechanism to support multiple branching, the previous branch is simply pruned when a new command is executed. The consequence of this is that the earlier commands can no longer be redone. They have been pruned away, and only the last executed command exists.

It is also important to note that the undo queue exists during the current Maya session. The undo queue isn't saved with the scene, so it isn't possible to undo operations to a file that has been saved and loaded again. When a new scene is loaded, the undo queue is completely emptied.

MPXCOMMAND SUPPORT FOR UNDO/REDO

With a better understanding of how the undo queue works, look at how commands can be designed to support undo and redo. When Maya executes a command, an instance of the command is allocated. This is done by calling the command's `creator` function. The command object's `doIt` function is then called. This function performs the real work of the command.

What happens if the user then requests to undo the command? In addition to the `doIt` function, the **MPxCommand** class also contains several other member functions to support undoing. These functions are as follows:

```
virtual MStatus undoIt();
virtual MStatus redoIt();
virtual bool isUndoable() const;
```

In the previous command examples, none of these functions were implemented, since the **MPxCommand** class provided its own default implementation. The default implementation of the `isUndoable` function returns `false`. This effectively means that the command can't be undone. As such, the `undoIt` and `redoIt` functions would never be called.

Since the previous commands weren't undoable, how did they work with Maya's undo mechanism? Here are the series of steps that Maya performed when the command, say `posts1`, was executed.

1. An instance of the `posts1` command is created using the `creator` function.
2. The command object's `doIt` function is then called to do the work of the command.
3. Maya then calls the `isUndoable` function to determine whether the command is undoable. Since the command returns `false`, it isn't undoable, so it isn't placed in the undo queue but is instead deleted immediately.
4. As a result of the deletion, the command object's destructor is then called.

Why did the command object get immediately deleted after it was executed? Since the command can't be undone, it can't be put into the undo queue. What would have happened if the command was undoable? In order for the command to be undoable, the `isUndoable` function would have to be implemeted to return `true`. When the command was then executed, the following would happen:

1. An instance of the `postsl` command is created using the `creator` function.
2. The command object's `doIt` function is called.
3. The `isUndoable()` function is then called to determine if the command is undoable. Since it now returns `true`, the command is placed into the undo queue.

Notice, in this case, that the command object wasn't deleted. The object now sits in the undo queue. What if the user now selects undo? Maya looks at the last command in the undo queue and calls its `undoIt` function. This function should undo any changes the command made. As such, the `undoIt` should restore Maya to exactly the same state it was in prior to the command being called. The scene, for example, will now be the same as if the command had never been executed. If the user continues to select undo, the `undoIt` function is called on each successive command object in the queue. This can be continued until there are no more command objects in the undo queue.

It is important to note that command objects still exist after their `undoIt` function has been called. They aren't deleted, because the user may decide to redo any of them. If redo is selected, the most recently undone command object's `redoIt` function is called. This is the equivalent of reexecuting the command. As a result, the `redoIt` function should provide the same functionality as the `doIt` function. In fact, `doIt` and `redoIt` can be considered synonymous.

UNDOABLE AND NONUNDOABLE COMMANDS

Since it is possible to design a command without support for undoing, how do you decide if you need to support undoing? The answer is quite simple.

If the command changes Maya's state in any way,
it *must* provide undo and redo!

It is possible to write a command that treats the Maya state as read-only. For instance, you could write a command that simply counted the number of spheres in

the scene. This command won't alter Maya's current state, so it doesn't need to provide support for undoing. Maya has a particular name for commands that don't support undoing: *actions*. Actions are commands that when executed don't create, modify, or change the Maya state, so they can be called anytime without concern. Any command whose `isUndoable` function returns `false` is considered an action.

This doesn't mean that you can't write a command that accidently alters Maya's state and that doesn't support undoing. In this case, you must change the command so that it is undoable.

POSTS5 PLUGIN

Since the previous `posts` commands altered the state of Maya, they really should include undo and redo functionality.

1. Open the **Posts5** workspace.
2. Compile it, and load the resulting `posts5.mll` plugin in Maya.
3. Open the **PostsCurve.ma** scene.
4. Select the curve.
5. In the Command Line, type and then execute the following:

   ```
   posts5 -number 7 -radius 1
   ```

   Seven cylinders are created along the curve.
6. Select **Edit | Undo** or press **Ctrl+z** to undo.

   The `posts5` command is undone. As a result, the newly created cylinders are removed.
7. Select **Edit | Redo** or press **Shift+z** to redo.

   The command is reexecuted, which results in seven new cylinders.

Now look at how the `posts` command was changed in order to support undoing and redoing.

Plugin: Posts5
File: posts5Cmd.cpp

The command class is defined as before but with the addition of some new member functions and a new data member.

```
class Posts5Cmd : public MPxCommand
{
public:
        virtual MStatus doIt ( const MArgList& );
        virtual MStatus undoIt();
        virtual MStatus redoIt();
        virtual bool isUndoable() const { return true; }

        static void *creator() { return new Posts5Cmd; }
        static MSyntax newSyntax();

private:
        MDGModifier dgMod;
};
```

The member functions `undoIt`, `redoIt`, and `isUndoable` have been added. You'll notice that the `isUndoable` function has been implemented and now returns `true` to indicate that undoing is supported. The most significant addition is the new member, `dgMod`. It is an **MDGModifier**. The **MDGModifier** class is used to create, remove, and alter nodes in the Dependency Graph. While this can be done using other classes, the biggest benefit of the **MDGModifier** class is that it automatically provides undo and redo for all its operations. This saves you from having to implement this yourself.

As each of the **MDGModifier** functions for editing the Dependency Graph is called, the class keeps a record of it. In fact, when one of these functions is called, the operation it should have performed is simply recorded. The class contains two additional functions, `doIt` and `undoIt`. Only when the `doIt` function is called are all the recorded operations actually executed. Likewise, when the `undoIt` function is called, all the recorded operations are undone. It follows logically that the `undoIt` function can be called only after the `doIt` function; otherwise there would be nothing to undo.

When the `posts5` command is executed, its `doIt` function is called. Its `doIt` function has been changed. The call to `MGlobal::execute` has been replaced with a call to the **MDGModifier**'s `commandToExecute` function.

```
...
dgMod.commandToExecute( MString( "cylinder -pivot ") + pt.x + " " + pt.y +
    " " + pt.z + " -radius " + radius + " -axis 0 1 0 -heightRatio " +
    heightRatio  );
...
```

The series of calls to `commandToExecute` results in a corresponding series of operations being recorded. At this point the commands haven't actually been executed, only recorded. The very last line of the `doIt` function has an important change. The function now calls `redoIt`.

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

The `redoIt` function now does the actual work of the command. Since executing a command (`doIt`) is the same as reexecuting it (`redoIt`), there is no need to write two separate functions that perform the same operation. Instead, the actual command operations are put in the `redoIt` function, and the `doIt` function simply calls it. The `redoIt` function is defined as follows. The function makes a call to the **dgMod**'s `doIt` function. Since all the Dependency Graph operations have been recorded by the **dgMod** in the command's `doIt` function, to actually execute them the **dgMod** 's `doIt` function is called.

```
MStatus Posts5Cmd::redoIt()
{
    return dgMod.doIt();
}
```

Similarly, the `undoIt` function is defined as follows:

```
MStatus Posts5Cmd::undoIt()
{
    return dgMod.undoIt();
}
```

The **MDGModifer** is also responsible for undoing all its recorded operations. This example demonstrates the best approach to designing undoable commands. The `doIt` function should just prepare and record all the operations needed by the command. This information should be stored away in the class. The `doIt` function should simply call the `redoIt` function to do the actual work. The `redoIt` function takes the recorded information and then executes the operations. The `undoIt` function should take the recorded information and undo anything performed during the `redoIt` function.

In this particular example, a series of DG nodes (cylinders) were created, so the undo operation simply had to remove the cylinders. In a more complex command, you may need to record far more information in advance, such as the control points of a mesh before they are deformed or some other relevant information. In fact, all information necessary to return Maya to its previous state before the command was executed must be recorded. For complex operations, the recording of the data that is about to be altered can be quite burdensome. Unfortunately this can't be avoided.

When you design a command that isn't an action, it is very important to consider how you will support undoing and redoing. Often a command that hasn't been designed to handle undoing is difficult to change later into an undoable command. Designing the command with this in mind from the very start often means that you won't have to make large changes to incorporate it later. Once implemented, you can be assured that your command will operate seamlessly with Maya's undo/redo mechanism.

4.4.8 Edit and Query

In a previous section, the `posts` command was extended to include a few custom flags that allowed the specification of such parameters as the height and radius of the posts. Once a command is executed, the current values of the parameters are used. It may be necessary to tweak these parameters later. For many parameters, it is important that the user be able to query their current values as well as to set new values. To enable this, a command can be called in a variety of *modes.* When a command is executed to create something, it is run in *creation mode.* When a command is executed to retrieve the value of a parameter, it is operating in *query mode.* When a command is executed to change an existing parameter, it operates in *edit mode.* Most commands support one or more of these modes.

These various modes aren't actual real states. A command doesn't really change states. A command is free to do anything it wants at any time. The various modes are simply a convention used to describe when certain operations can and can't be

performed. Since the operations of creating, editing, and querying are so common, they have been given their own convention that all commands should follow. The standard convention is that if a command supports querying and editing, it adds the query and edit flags. These are defined in their short and long forms as `-q/query` and `-e/edit`, respectively.

The `sphere` command can be called in a variety of modes.

1. Create a new scene by selecting **File | New Scene.**
2. In the **Script Editor**, execute the following:

```
sphere -radius 2;
```

A NURBS sphere is created with a radius of 2. Since the `sphere` command is neither being queried nor edited, it is being executed in the creation mode. As a result, a new sphere object is created.

The sphere's radius is now increased.

3. Execute the following:

```
sphere -edit -radius 10;
```

The sphere's radius is now larger. In this instance, the `sphere` command was executed in edit mode. The command didn't create a new sphere but instead altered the attributes of the existing sphere.

4. Execute the following to query the current radius of the sphere:

```
sphere -query -radius;
```

The result is then displayed.

```
// Result: 10 //
```

The `sphere` command has been executed in query mode. It neither creates a new sphere nor edits an existing one but instead returns the current value of the requested parameter.

When you call the `sphere` command with the `-query` flag, it returns a result. This result can be stored in a variable and used later. It is important to note that you can't query multiple parameters at the same time. It isn't valid, for example, to ask for the sphere's `radius` and the `startSweep` at the same time, since only one value can be returned at a time from the command.

```
// Could return radius or startSweep?
sphere -query -radius -startSweep;
```

When you want to query multiple parameters, simply break the requests into multiple command calls, each with a single query, as follows:

```
sphere -query -radius;
sphere -query -startSweep;
```

It is possible, however, to edit multiple parameters with the same command call.

```
sphere -edit -radius 2 -startSweep 20;
```

It isn't possible to mix modes in a single command call. Therefore, it isn't possible to run a command in both edit and query modes. Multiple executions of the command must be done, with separate edits and queries.

CLOCK PLUGIN

In order to demonstrate a more complex example of querying and editing, the implementation of the `clock` command is covered. This command sets the hands of a 3D clock based on a provided time. Figure 4.16 shows an example of the `clock` command applied to a 3D clock.

1. Open the **Clock** workspace.
2. Compile it, and load the resulting `clockCmd.mll` plugin in Maya.
3. Open the **Clock.ma** scene.
4. Select the two hand objects, **hour_hand** and **minute_hand**.
5. In the **Script Editor**, execute the following:

```
clock -edit -time 745;
```

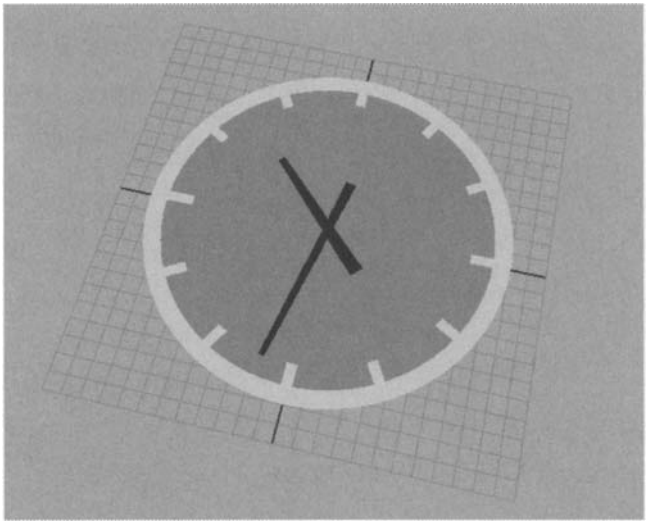


FIGURE 4.16 3D clock controlled by the `clock` command

The hour and minute hands are rotated to the given time, 7:45. Time is specified to the command with the hour first then the minutes: *hhmm.* For example 2:31 is represented as 230 while 3:00 is represented as 300.

6. Execute the following:

```
clock -e -time 1018;
```

The hands rotate to the given time: 10:18. Notice that the short form (`-e`) of the edit flag was used.

Without knowing the time in advance, you may want to query the current time of the clock.

7. Execute the following:

```
clock -query -time;
```

The result of the query is displayed.

```
// Result: 1018 //
```

This command implements both undo and redo.

8. Execute the following:

```
undo;
```

The hands are returned to the previous time.

9. Execute the following:

```
redo;
```

The hands are set, once again, to the new time.

The source code to the `clock` command will now be covered in detail.

Plugin: Clock
File: clockCmd.cpp

The command is implemented in the **ClockCmd** class. As with all Maya commands, it is derived from **MPxCommand**. As demonstrated earlier, it supports both undo and redo. The class also contains some additional members and methods. These are explained as they are used in the command.

```
class ClockCmd : public MPxCommand
{
public:
        virtual MStatus doIt ( const MArgList& );
        virtual MStatus undoIt();
        virtual MStatus redoIt();
        virtual bool isUndoable() const;

        static void *creator() { return new ClockCmd; }
        static MSyntax newSyntax();

private:
        bool isQuery;
        int prevTime, newTime;
        MDagPath hourHandPath, minuteHandPath;

        int getTime();
        void setTime( const int time );
};
```

As before, the command defines which parameters it accepts and what values these parameters can take. The `clock` command has only one parameter flag, `time`, which is given as `-t/-time`.

```
const char *timeFlag = "-t", *timeLongFlag = "-time";

MSyntax ClockCmd::newSyntax()
{
    MSyntax syntax;

    syntax.addFlag( timeFlag, timeLongFlag, MSyntax::kLong );
```

In addition to adding the `time` flag, the syntax object also specifies that it accepts both the `query` and `edit` flags. This is done by calling the **MSyntax**'s `enableQuery` and `enableEdit` functions. The command syntax object now supports three flags: `time`, `query`, and `edit`.

```
    syntax.enableQuery(true);
    syntax.enableEdit(true);

    return syntax;
}
```

The command's `doIt` function is a little more involved, since undoing and redoing have to be handled manually. To perform the undo, you must know the current state of the object that is about to be changed by the command. With this state stored away, the command can restore the object to its original state when the user selects undo. Likewise, exactly which state is going to be changed must be known. This is important for both executing (`doIt`) and reexecuting (`redoIt`) the command.

Since the command really has only one parameter, `time`, this is what is stored. The current time is stored in the `prevTime` class variable. This is the value that is used when the command is undone. The new time value is stored in the `newTime` class variable. This is the value that is used when you execute or reexecute the command.

```
int prevTime, newTime;
```

The `doIt` function takes the command parameters and stores them away. It also records in the `isQuery` class variable whether the command is being run in query or edit mode.

```
bool isQuery;
```

Since the command doesn't create anything, it won't need to support creation mode. The new time is also recorded if the command is in edit mode. Finally the currently selected clock hand objects are also recorded in the `hourHandPath` and `minuteHandPath` class variables.

```
MDagPath hourHandPath, minuteHandPath;
```

The various operations of the `doIt` function will now be explained.

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

The **MArgDatabase** is initialized as before. Notice that if its initialization fails, then the command also fails. Its initialization fails if any of the command arguments are invalid.

```
MArgDatabase argData( syntax(), args, &stat );
if( !stat )
    return stat;
```

The `isQuery` member is then set. The **MArgDatabase** class has a convenience function, `isQuery()`, that returns `true` if the query flag was set on the command line. Setting this effectively puts the command in query mode. Since the **clock** command doesn't create anything, it runs only in two possible modes: query or edit.

```
isQuery = argData.isQuery();
```

If the command is running in edit mode, then it retrieves the value of the `newTime` member.

```
if( argData.isFlagSet( timeFlag ) &&  !isQuery )
    argData.getFlagArgument( timeFlag, 0, newTime );
```

This next section of code iterates over all the selected objects and retrieves their names. When it finds objects named either **hour_hand** or **minute_hand**, their DAG paths are stored.

```
// Get a list of currently selected objects
MSelectionList selection;
MGlobal::getActiveSelectionList( selection );

MDagPath dagPath;
MFnTransform transformFn;
MString name;

// Iterate over the transforms
MItSelectionList iter( selection, MFn::kTransform );
for ( ; !iter.isDone(); iter.next() )
  {
  iter.getDagPath( dagPath );
  transformFn.setObject( dagPath );

  name = transformFn.name();
  if( name == MString("hour_hand") )
    hourHandPath = dagPath;
  else
    {
    if( name == MString("minute_hand") )
      minuteHandPath = dagPath;
    }
  }
```

If the clock hand objects weren't found, then the command emits an error message.

```
// Neither hour nor minute hand selected
if( !hourHandPath.isValid() || !minuteHandPath.isValid() )
  {
  MGlobal::displayError( "Select hour and minute hands" );
  return MS::kFailure;
  }
```

The current time, as represented by the two hand positions, is then determined.

```
prevTime = getTime();
```

If the command is in query mode, then simply set the result to the current time, then return.

```
if( isQuery )
  {
  setResult( prevTime );
  return MS::kSuccess;
  }
```

If the command is in edit mode, then call the `redoIt` function to do the editing work.

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

The `undoIt` and `redoIt` functions simply call the `setTime` member function. The `undoIt` function sets the clock hands to the previous time, while the `redoIt` function sets them to the new time.

```
MStatus ClockCmd::undoIt()
{
  setTime( prevTime );
  return MS::kSuccess;
}

MStatus ClockCmd::redoIt()
{
  setTime( newTime );
  return MS::kSuccess;
}
```

The `getTime` function calculates what time is currently shown by the position of the clock hands. While an exact understanding of the math isn't necessary, it is important to note that the function gets the `hour_hand` object and then determines what rotation it has about the y-axis. The amount of rotation determines what the current time is. The time is then formatted as *hhmm* and returned.

```
int ClockCmd::getTime()
{
    // Get the time from the rotation
    MFnTransform transformFn;
    transformFn.setObject( hourHandPath );
    MEulerRotation rot;
    transformFn.getRotation( rot );

    // Determine the time and format it
    int a = int(-rot.y * (1200.0 / TWOPI));
    int time = (a / 100 * 100) + int( floor( (a % 100) *
        (6.0 / 10.0) + 0.5 ) );

    return time;
}
```

The `setTime` function rotates the **hour_hand** and **minute_hand** objects to reflect the given time.

```
void ClockCmd::setTime( const int time )
{
    MFnTransform transformFn;

    // Calculate the hour and minutes
    int hour = (time / 100) % 12;
    int minutes = time % 100;

    // Rotate the hour hand by the required amount
    transformFn.setObject( hourHandPath );
    transformFn.setRotation( MEulerRotation( MVector( 0.0, hour *
        (-TWOPI / 12) + minutes * (-TWOPI / 720.0), 0 )  ) );

    // Rotate the minute hand by the required amount
    transformFn.setObject( minuteHandPath );
    transformFn.setRotation( MEulerRotation( MVector( 0.0, minutes *
        (-TWOPI / 60.0), 0 ) ) );
}
```

The `isUndoable` function contains an important change. It normally returns `true` for any command that supported undo. Recall from the previous section that a command that isn't undoable is considered an action. An action, when executed, won't change the state of Maya. When the `time` parameter is being queried, this is effectively what is being done. The parameter is only being queried, so Maya's state isn't being changed. As such, the `isUndoable` function returns `false` when the command is operating in query mode; otherwise it returns `true`, since it will then be running in edit mode.

```
bool ClockCmd::isUndoable() const
{
    return isQuery ? false : true;
}
```

What happens if the command doesn't return `false` when in query mode? As with all undoable commands, the command object is added to the undo queue. When the command is undone, nothing happens, since undoing a query doesn't result in anything being restored. Nothing has been changed, so there is nothing to be undone. As such, returning `true` when in query mode can be considered benign. However, since the command object is added to the undo queue, a lot of queries result in the undo stack being filled unnecessarily. In general, it is best that the `isUndoable` function return `true` when it actually has something to undo and `false` otherwise.

4.5 NODES

By creating your own commands and using MEL, you can already access a great deal of Maya's power. However, even these methods don't allow you to add your own pieces to the underlying Maya engine, the Dependency Graph. In order to create your own parts and have them included in this machinery, you'll need to create a node. The node is integrated directly into the Dependency Graph, so it becomes a seamless part of the general scene. It is possible to create a variety of different nodes, each with its own particular way of processing or creating data.

4.5.1 GoRolling Plugin

In order to provide a general understanding of how to write a node and then integrate it into Maya, a simple node will be covered. This node will show you how to control the rotation of a wheel. When animating wheels, it is easiest for the animator

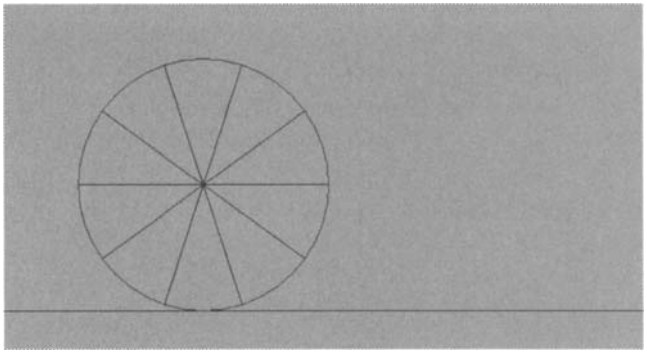


FIGURE 4.17 Wheel that automatically rotates

to define the start and end location of the object. The animator then has to set rotation keys for the wheel so that it rolls as it moves along. As you can imagine, this last step can be completely automated. A custom node is developed that automatically rotates the wheel based on its position. The animator can use this new node by simply moving the wheel, and its rotation happens automatically. In Figure 4.17 the wheel object that automatically rotates as a result of the new node is shown.

The complete project involves creating both a command as well as a node. The command, `goRolling`, will be responsible for creating the node, **RollingNode**, inserting it into the DG, and making all the necessary attribute connections. The node handles the actual rotation of the object.

1. Open the **GoRolling** workspace.
2. Compile it, and load the resulting `GoRolling.mll` plugin in Maya.
3. Open the **GoRolling.ma** scene.
4. Click on **Play**.

   The wheel moves along the ground, but it doesn't rotate.
5. Select the **rim** object.
6. Open the **Attribute Editor**.

   Notice that the rim's translation is currently animated, while its rotation isn't.

7. In the **Script Editor**, execute the following:

```
goRolling;
```

   In the Attribute Editor, you can see that the **Rotate Z** is now animated.

8. Click on **Play**.

   The wheel now rotates as it moves.

9. Click **Stop**, then select the **Move Tool**. With the **rim** object still selected, move it along in the *x* direction (red arrow). Move the rim back and forth to see how the wheel automatically rotates wherever you translate it. Now take a closer look at what is happening under the covers.

10. Open the **Hypergraph**.

11. With the **rim** transform node selected, click on the **Up and Downstream Connections** button.

    The Hypergraph shown in Figure 4.18 is displayed.

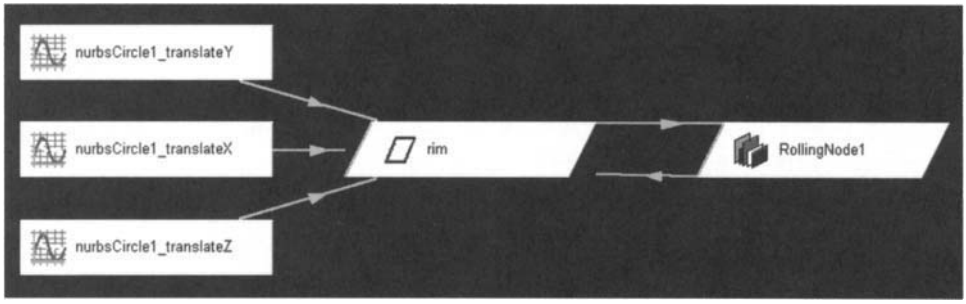


FIGURE 4.18 Hypergraph after executing `goRolling` command

This view shows that the rim's translations, **translateX**, **translateY**, and **translateZ**, are controlled by three animation curves, **nurbsCircle1_translateX**, **nurbsCircle1_translateY**, and **nurbsCircle_translateZ**, respectively. This is the standard setup for animating any attribute.

The new addition is the **RollingNode1** node. There are two connections from the **rim** node to the **RollingNode1** node. It may not be easy to see these, since they

are drawn almost one on top of the other. There is also one connection going back from the **RollingNode1** node to the **rim** node. These connections are shown more clearly in Figure 4.19.

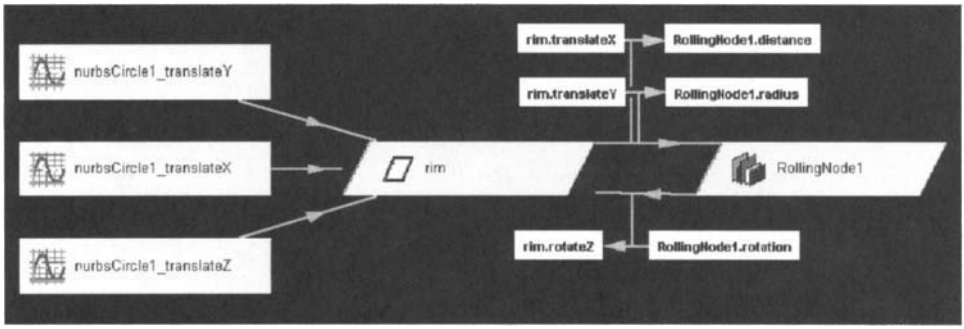


FIGURE 4.19 Node connections

From this diagram, it is easier to deduce what is happening. The value of the **rim**'s *x* translation attribute, **translateX**, is being fed into the **RollingNode1**'s **distance** attribute. The **distance** attribute defines how far the object has traveled. In this instance, this is simply the distance the object has moved along the x-axis, which equates directly to the **rim**'s **translateX** value.

Since the rim shape was created from a circle, its **transform** node, **rim**, is located at its center. Consequently its distance along the y-axis, **translateY**, is equivalent to the radius of the wheel. The **rim**'s **translateY** value is therefore fed into the **RollingNode**'s **radius** attribute. So the **rim**'s **translateX** and **translateY** are being used as the inputs to the **distance** and **radius** values, respectively, in the **RollingNode**. Taking these two input values, the **RollingNode1** node calculates a single output value, **rotation**. The **rotation** output is fed back into the **rim** node's **rotateZ** attribute. The **rim**'s **rotateZ** value is now the result of the **RollingNode1**'s rotation calculation.

When the wheel was moved back and forth, this changed the **translateX** value, which in turn changed the **RollingNode**'s **distance** input value. The node then recomputed a new **rotation** value that was subsequently fed into the **rim**'s **rotateZ** value. The end result is that as the **rim** moves along the x-axis, its *z* rotation changes.

The details of the **RollingNode** node will now be explained. The **GoRolling** project defines a custom command, `goRolling`, and a custom node, **RollingNode**.

Plugin: GoRolling
File: GoRollingCmd.h

The `goRolling` command is used to create a new **RollingNode** and connect it to the currently selected objects. The `posts` command demonstrated how to create cylinder nodes. The `goRolling` command also creates nodes, but it now creates connections between those nodes. The `goRolling` command is designed to handle both undo and redo to ensure that its effects can be removed if requested. The class declaration is straightforward and follows logically from the format of the previous commands.

```
class GoRollingCmd : public MPxCommand
{
public:
        virtual MStatus doIt ( const MArgList& );
        virtual MStatus undoIt();
        virtual MStatus redoIt();
        virtual bool isUndoable() const { return true; }

        static void *creator() { return new GoRollingCmd; }
        static MSyntax newSyntax();

private:
        MDGModifier dgMod;
};
```

The command contains a single **MDGModifier** member, `dgMod`, that is used to create the **RollingNode** node as well as the necessary connections. The **MDGModifier**, as before, simplifies the task of providing undo and redo.

Plugin: GoRolling
File: GoRollingCmd.cpp

The main work is completed in the `doIt` function.

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

A list of the currently selected objects is created.

```
MSelectionList selection;
MGlobal::getActiveSelectionList( selection );
```

All the transform nodes are then iterated over by specifying the `MFn::kTransform` filter.

```
MDagPath dagPath;
MFnTransform transformFn;
MString name;

// Iterate over all the transforms
MItSelectionList iter( selection, MFn::kTransform );
for ( ; !iter.isDone(); iter.next() )
      {
      iter.getDagPath( dagPath );
      transformFn.setObject( dagPath );
```

For each of the **transform** nodes, the following tasks are completed. A new **RollingNode** node is created by using the **MDGModifier**'s `createNode` function.

```
MObject rollNodeObj = dgMod.createNode( "RollingNode" );
```

An **MFnDependencyNode** function set, `depNodeFn`, is attached to the newly created node object. Using `depNodeFn`, the contents of the node can be accessed.

```
MFnDependencyNode depNodeFn( rollNodeObj );
```

A node's attributes are accessed by getting its *plugs*. Plugs are explained in more detail later, but for now they can be considered a mechanism for getting a node's attribute values. The **translateX** plug is obtained from the **transform** node, and the **distance** plug is obtained from the **RollingNode** node. The **MDGModifier**'s `connect` function is then used to make a connection from one plug to the other.

```
dgMod.connect( transformFn.findPlug( "translateX" ),
               depNodeFn.findPlug( "distance") );
```

This same steps are repeated to connect the **translateY** plug to the **radius** plug.

```
dgMod.connect( transformFn.findPlug( "translateY" ),
               depNodeFn.findPlug( "radius" ) );
```

Finally the **RollingNode**'s **rotation** plug is connected to the transform's **rotationX** plug.

```
dgMod.connect( depNodeFn.findPlug( "rotation" ),
               transformFn.findPlug( "rotateZ" ) );
}
```

The `redoIt` function is then called to do the real work now that the **MDGModifier**, `dgMod`, has been set up with all the required Dependency Graph operations.

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

The `redoIt` function simply calls the **MDGModifer**'s `doIt` function, which performs all the requested operations.

```
MStatus GoRollingCmd::redoIt()
{
return dgMod.doIt();
}
```

Undoing the command is, once again, simplified by using the **MDGModifier** class, which provides automatic undoing.

```
MStatus GoRollingCmd::undoIt()
{
return dgMod.undoIt();
}
```

With the command now complete, the custom node is covered.

Plugin: GoRolling
File: RollingNode.h

A custom node is created by deriving from the **MPxNode** class. A node is quite different from a command. It doesn't have any `doIt` function but instead provides a few functions, the most important of which is the `compute` function. This function does all the real work of data processing, and is therefore considered the "brains" of the node.

```
class RollingNode : public MPxNode
{
public:
    virtual MStatus compute( const MPlug& plug, MDataBlock& data );

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

The following static member variables are another important addition to the node. These variables contain definitions for the node's attributes. There is one variable for each of the node's input and output attributes.

```
    static MObject distance;
    static MObject radius;
    static MObject rotation;

    static MTypeId id;
};
```

Plugin: GoRolling
File: RollingNode.cpp

Each type of node in Maya, irrespective of whether it is native or custom, has a unique identifier so that Maya knows how to create it, as well as store and restore it from disk. The **MTypeId** class is used for this identifier.

When Maya needs to reload a node from disk, it needs to know which node stored the data. To identify the node that created the data, the node's identifier is stored with the data. During retrieval Maya loads the node identifier and then creates a new node of the type indicated by the identifier. The node then loads its data from disk.

From this description it is clear that if two different types of nodes have the same identifier, then Maya could create the wrong node when the data is restored from disk. This is particularly important for binary Maya files since they store only the node's identifier and not the node's class name. If a node is stored to a binary file with a given identifier and you later change its identifier, then Maya won't be able to read the restored node data from the file. The node's new identifier won't match the one stored on disk, so Maya won't be able to locate the owner of the original data. Therefore, if you store your nodes to a binary Maya file, make sure never to later change the nodes' identifiers. This isn't necessarily a problem for ASCII files, since the node's type is stored and not its identifier.

It is, therefore, important that you ensure that each new node has a unique identifier. What happens if you get Maya files from another person who happens to use similar identifiers for his or her own custom nodes? This would result in the same data-loading problem mentioned earlier. In order to prevent nodes having similar identifiers, Alias | Wavefront provides developers with a unique set of identifiers that they can use. These numbers are assigned to you so no one else in the world will have the same one. If this protocol is strictly followed, every node ever created will have a unique identifier. As a result, there should never be any problems sharing files from different companies or individuals who use custom nodes.

So before you start development, it is generally a good practice to ask Alias | Wavefront for a set of unique identifiers. For details on becoming a registered developer and getting identifiers, visit the Alias | Wavefront website, *www.aliaswavefront.com.*

If for whatever reason you can't get unique identifiers for your nodes before development starts, you can use some temporary "internal development only" identifiers. These identifiers are to be used only temporarily while you await the final identifiers. Therefore, they should never be used in nodes that will be used outside your development environment. You can use any of the temporary identifiers in the range of 0 to 0x7FFFF. For the **RollingNode** example, the arbitrary temporary identifier, `0x00333`, has been used.

```
MTypeId RollingNode::id( 0x00333 );
```

It is important to replace this number with a correctly allocated identifier as soon as possible. While this may not seem important while you are developing the node, it can cause a great deal of problems if it is stored to a Maya scene file and then the identifier is later changed. This can happen quite easily if testers create Maya files with your node during beta testing and then you change the node identifier when you release the final plugin. All files created during the beta testing now won't load the node's data.

The next section of code defines the node's attribute specifiers. At this point they are just defined. They will be initialized later.

```
MObject  RollingNode::distance;
MObject  RollingNode::radius;
MObject  RollingNode::rotation;
```

The `compute` function is defined next. It takes two arguments; the first is an **MPlug** that specifies which plug (node's attribute) needs to be recomputed, and the second is an **MDataBlock** that holds the current data for the node.

```
const double PI = 3.1415926535;
const double TWOPI = 2.0 * PI;

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

A node can have any number of output attributes. Since only output attributes are computed and they may not all need updating, Maya calls the `compute` function with each output attribute individually. As such, you must check which of the output attributes is being requested to update. Since this node has only one output attribute, **rotation**, only it is checked for.

```
if( plug == rotation )
    {
```

When a request is made to recompute the **rotation** attribute, the set of current input attributes is first retrieved. These values are retrieved from the node by calling the **MDataBlock**'s `inputValue` function. This function treats the node's attribute as read-only. You can then retrieve its values but can't write to it. Similarly there is an `outputValue` function that allows you to write to an attribute but not read it. Even though Maya makes no distinction between input and output attributes, by having separate input and output functions, the data can be retrieved and maintained more efficiently.

You need to specify which of the attributes you want to retrieve. In this case, the **distance** and **radius** attributes are needed. The `inputValue` function returns an **MDataHandle** object. This is used to access the actual attribute data.

```
MDataHandle distData = data.inputValue( distance );
MDataHandle radData = data.inputValue( radius );
```

The steps to getting an attribute's values may seem convoluted, but there are very good reasons, which are explained in detail in the section covering the

MDataBlock class. With the **MDataHandle** object, the input attribute's data can now be retrieved. Since the attributes are known to each hold a `double`, the `asDouble` function is used.

```
double dist = distData.asDouble();
double rad = radData.asDouble();
```

The result of the computation is stored in the **rotation** attribute. A handle to this attribute is needed. This handle is created by calling the **MDataBlock**'s `outputValue` function. This returns a **MDataHandle** object that can be used to write to an attribute.

```
MDataHandle rotData = data.outputValue( rotation );
```

The **rotation** attribute is calculated from the input **distance** and **radius** attributes. The result is stored in the **rotation** attribute by using the **MDataHandle**'s `set` function.

```
rotData.set( -dist / rad );
```

In the next step, the plug is set to be clean. This lets Maya know that the plug has been recomputed and holds the new correct value.

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

The **MPxNode** class itself contains many attributes. Since the **RollingNode** is derived from the **MpxNode**, it automatically inherits these attributes. Rather than have the derived class, **RollingNode**, handle the recomputation of these inherited attributes, the `compute` function simply has to return `MS::kUknownParameter` when it finds an attribute that it doesn't know about. Maya then calls the base class's `compute` function and sees if it can recompute the attribute. This way, derived classes have to compute only the attributes that they explicitly introduce. All other attributes are handled by their direct or indirect base classes.

```
else
    stat  = MS::kUnknownParameter;
```

The `compute` function returns an **MStatus** indicating whether computation was successful.

```
return stat;
}
```

Like commands, nodes also have a function to create an instance of themselves. The `creator` function is a static function that simply creates a new instance of the node.

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

The `initialize` function is also static. It is called only when the node is first registered. Registration of the node is detailed in the next source code file.

```
MStatus RollingNode::initialize()
{
```

The job of the node's `initialize` function is to set up all the node's attributes. Since the node's attributes, **distance**, **radius**, and **rotation**, are all static members, there is one copy of them shared by all instances of the node. The reason for this is that the attributes don't actually hold the data for each individual node. Instead they act as a blueprint for the creation of the attribute data that each node will use. Technically, a node's attribute data is stored in an **MDataBlock** and is retrieved and set using an **MPlug**. In the context of the C++ API, an attribute is just a specification for creating a node attribute. In the C++ API, an attribute is created and edited using an **MAttribute** and its derived classes.

Maya supports a wide variety of attributes including simple numeric types such as `bool`, `float`, and `int`, as well as more complex compound attributes. These attributes use the simple types and combine them in more complex ways. The **RollingNode** node requires only some simple floating-point values as input. The **distance** attribute is defined in the following code. The attribute is created by calling the `create` function. This attribute is stored as a single `double`. This is indicated to the **MFnNumericAttribute** by using the indicator `MFnNumericData::kDouble`. It has

a long and short name, "distance" and "dist", respectively. The attribute's default value is 0.0. The resulting attribute object is stored in the distance static member. The **distance** attribute should be created using the **MFnUnitAttribute** class, but a double suffices for this example.

```
MFnNumericAttribute nAttr;
distance = nAttr.create( "distance", "dist",
                         MFnNumericData::kDouble, 0.0 );
```

The **radius** attribute is created in a similar fashion.

```
radius = nAttr.create( "radius", "rad", MFnNumericData::kDouble, 0.0 );
```

The last attribute is the **rotation** attribute. It is different from the input attributes in that it holds an angle. An angle in Maya can be represented in many different ways. It can for example be given in radians or degrees. In fact, an angle is considered a unit. Since an angle can have multiple representations, it can't be stored in a simple double. Instead, it is specifed using an **MFnUnitAttribute** object. This class is designed to handle different types of Maya units, including time, angles, and so on.

```
MFnUnitAttribute uAttr;
rotation = uAttr.create( "rotation", "rot",
                         MFnUnitAttribute::kAngle, 0.0 );
```

Now that all the attributes have been created, they can be added to the node by calling the **MPxNode**'s addAttribute function.

```
addAttribute( distance );
addAttribute( radius );
addAttribute( rotation );
```

In order for Maya to know how the attributes affect each other, you must explicitly state this using the **MPxNode**'s attributeAffects function. This sets up a dependency between the attributes, thereby defining which attributes are considered input and which output. Changing the input attribute causes a direct reaction in the attributes it affects. When the **distance** or **radius** attribute changes, the **rotation** attribute needs to be recomputed. This relationship is formally expressed by using the attributeAffects function as follows.

```
attributeAffects( distance, rotation );
attributeAffects( radius, rotation );
```

The initialization was successful.

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

Notice that no error checking was done. This is because the example code is designed to be simple and brief. In a real plugin, you would do error checking on each of the function calls, and if any of them fail, an **MStatus** indicating failure would be returned.

With the command and node now completed, you need to notify Maya that they both exist. Once they are registered with Maya, they can be used in the scene.

Plugin: GoRolling
File: pluginMain.cpp

As always, the `initializePlugin` and `uninitializePlugin` functions must be present in order for Maya to load and unload the plugin. These functions have been extended to include better error checking. The registration process is completed as follows:

```
MStatus initializePlugin( MObject obj )
{
    MStatus stat;
    MString errStr;
    MFnPlugin pluginFn( obj, "David Gould", "1.0", "Any");
```

The command is registered, as before, using the **MFnPlugin**'s `registerCommand` function.

```
stat = pluginFn.registerCommand( "goRolling",
                                 GoRollingCmd::creator );
if ( !stat )
    {
    errStr = "registerCommand failed";
    goto error;
    }
```

The registration of the node is slightly different. You must provide a name for the node's type as well as its unique identifier. A creator function also needs to be specified. Last the node's initialize function must also be given.

```
stat = pluginFn.registerNode( "RollingNode",
                              RollingNode::id,
                              RollingNode::creator,
                              RollingNode::initialize );

if ( !stat )
    {
    errStr = "registerNode failed";
    goto error;
    }

return stat;
```

The `initialize` function is called just once to set up the blueprints for all the node's attributes. This is done when the node is first registered.

If any errors occur, an error message is displayed.

```
    error:

    stat.perror( errStr );
    return stat;
}
```

Deregistering the command and node is done in the `unitializePlugin` function.

```
MStatus uninitializePlugin( MObject obj)
{
    MStatus stat;
    MString errStr;
    MFnPlugin pluginFn( obj );
```

As in the previous command examples, the command is deregistered using the **MFnPlugin**'s `deregisterCommand` function. The function is given the name of the command to deregister.

```
stat = pluginFn.deregisterCommand( "goRolling" );
if ( !stat )
    {
    errStr = "deregisterCommand failed";
    goto error;
    }
```

The node is deregistered by passing the node's identifier to the **MFnPlugin**'s `deregisterNode` function. Since each node's identifier is unique and was used earlier to register the node, Maya can deregister the node given its identifier.

```
stat = pluginFn.deregisterNode( RollingNode::id );
if( !stat )
    {
    errStr = "deregisterNode failed";
    goto error;
    }

return stat;
```

If there is an error during the `uninitializePlugin` function, an error message is displayed.

```
    error:

    stat.perror( errStr );
    return stat;
}
```

This example demonstrates that creating a new custom node is relatively simple. Once the node is registered it can be created and inserted into the Dependency Graph like any other node. Maya handles all the data retrieval and storage for the node. It also handles all attribute connections and so on. In fact all the node really has to do is to ensure that it recomputes its output attributes when Maya requests them.

4.5.2 Melt Plugin

A slightly more complex node will now be presented. This node simulates the effect of an object melting. More precisely, it deforms the object so that it gives the impression that the object is being heated from below. It then begins to slowly melt and spread.

The original objects are shown in Figure 4.20. The result of applying the melting is shown in Figure 4.21.

This project creates a command and a node both named **melt**. The command iterates over all the selected NURBS surfaces. A **melt** node is inserted into the shape's

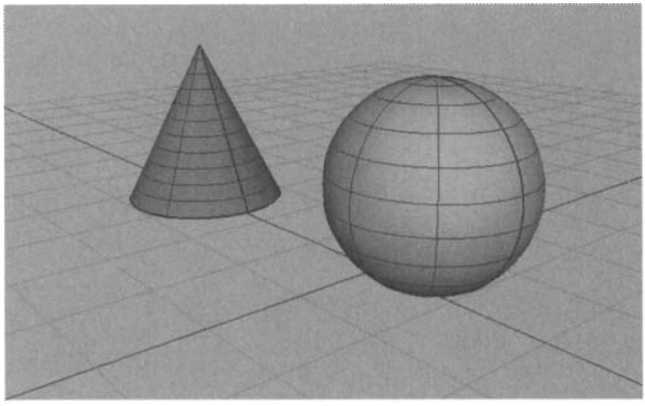


FIGURE 4.20 Original objects

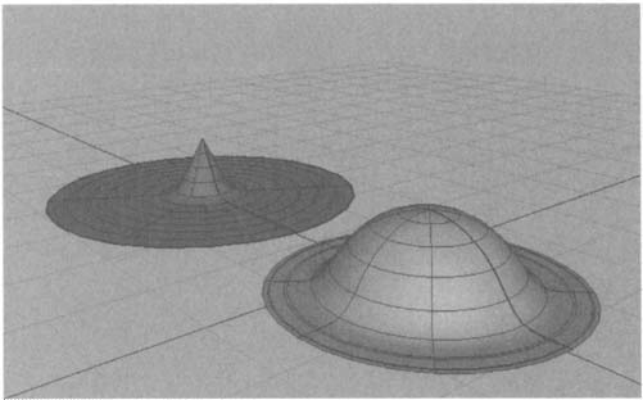


FIGURE 4.21 Objects after melting

construction history. The construction history concept will be explained shortly. The **melt** node's job is to deform the original surface to make it appear as if it were melting. The amount of melting is defined by the **melt** node's **amount** attribute. The **melt** command automatically animates the **melt** node's **amount** attribute so that the object melts over the current animation time range.

1. Open the **Melt** workspace.
2. Compile it, and load the resulting `Melt.mll` plugin in Maya.
3. Open the **Melt.ma** scene.
4. Select both the **nurbsCone1** and **nurbsSphere1** objects.
5. In the **Script Editor**, execute the following:

```
melt;
```

6. Click on the **Play** button.

   Both objects slowly melt downward and then begin spreading outward.

The `melt` command creates a `melt` node and inserts it into each of the object's construction histories. In order to understand what is being done, the **sphere** object is looked at in detail. Before the application of the `melt` command, the sphere appears as in Figure 4.22.

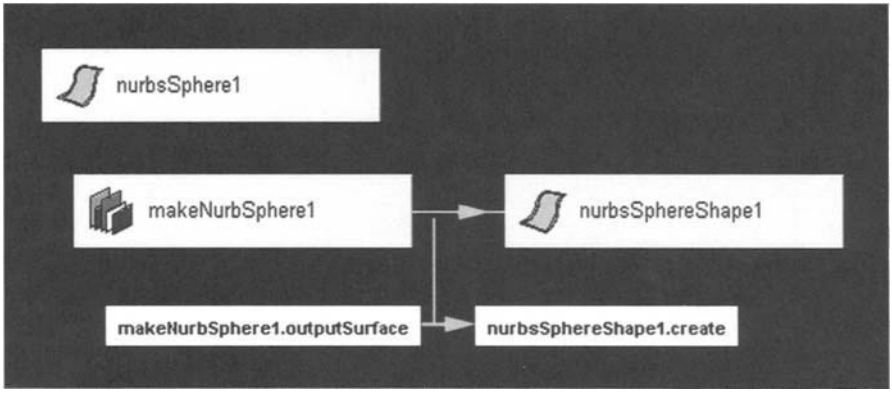


FIGURE 4.22 Standard sphere

The entire sphere consists of both the **shape** node, **nurbSphereShape1**, and the **transform** node, **nurbSphere1**. The actual NURBS surface results from the **makeNurbSphere1** node. This node generates a NURBS surface in the shape of a sphere. This same methodology is used for other NURBS primitives such as the cylinder and torus, which use a **makeNurbCylinder** and **makeNurbTorus** node, respectively.

The surface is output through the **makeNurbsSphere1**'s **outputSurface** attribute. This is then fed into the **nurbsSphereShape1**'s **create** attribute. Therefore, the **nurbsSphereShape1** node holds the final NURBS surface. It is also the responsibility of the **shape** node to display its shape. As such, the **nurbsSphereShape1** node displays the resulting NURBS surface in the scene.

In the previous examples, the node attributes were relatively simple. Many were just single `double` values. This node uses a more complex attribute that holds an entire NURBS surface. This NURBS surface attribute can be stored in the node and be connected to other nodes. The NURBS surface is effectively passed from one node to another through the connection.

When the `melt` command is executed, a **melt** node is created. This node is inserted into the sphere's construction history. The construction history is the series of nodes that together define the final shape. They do this by generating some data and then processing it, each in turn, resulting in the final shape being produced at the very end. The original sphere had only a **makeNurbSphere1** node in its construction history. When the melt node was inserted, it went in between the **makeNurbsSphere1** node and the final **nurbsSphereShape1** node. The new DG is shown in Figure 4.23. Notice the new **melting2** node.

The construction history of the NURBS sphere consists of the **makeNurbSphere1** node followed by the **melting2** node. The **makeNurbSphere1**'s **outputSurface** now feeds into the **melting2**'s **inputSurface** rather than directly into the **nurbsSphereShape1** node. The **melting2** node can now modify the surface before passing it into the **nurbsSphereShape1** node. All that has changed is that the **melting2** node now sits in the middle of the two previous nodes. The surface is fed into the node so it can change the surface before finally passing it into the **shape** node as before.

The **melt** node has an **amount** attribute that defines the amount of melting. The `melt` command automatically animates this attribute for you. In order to do this, an **animCurve** node is created and then connected to the **melting2** node's **amount** attribute. Fortunately this animation curve node doesn't have to be created manually, but instead Maya creates it automatically.

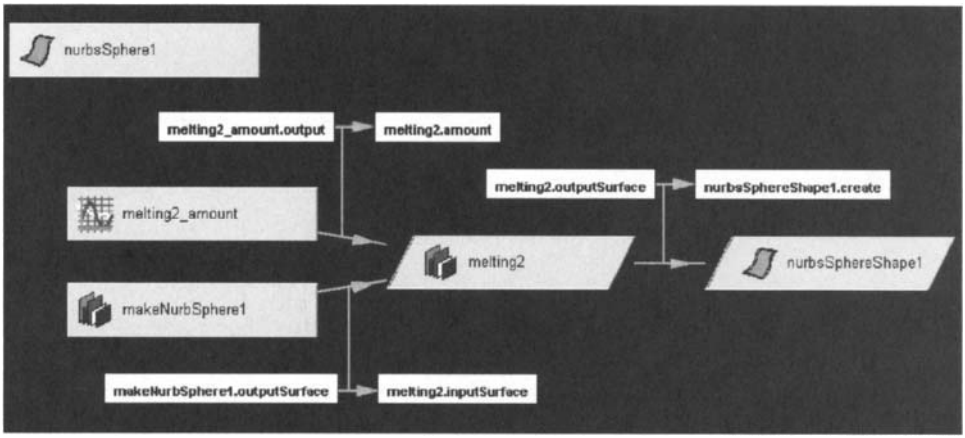


FIGURE 4.23 After inserting the **melt** node

Plugin: Melt
File: MeltCmd.cpp

The `melt` command is responsible for creating the necessary nodes and then connecting them. It also handles the animating of the **amount** attribute. The `doIt` function, once again, is assigned the task of setting up the command, while the `redoIt` function does the actual work.

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

    MSelectionList selection;
    MGlobal::getActiveSelectionList( selection );
```

In order to animate the **amount** attribute, the start and end range of the current animation needs to be known. Keyframes at those times will be created later.

```
MTime startTime = MAnimControl::minTime();
MTime endTime = MAnimControl::maxTime();
```

All the currently selected NURBS surfaces are iterated over. This means that the `melt` command can be applied to any number of NURBS surfaces. To let the **MItSelectionList** know that it should iterate only over NURBS surfaces, the `MFn::kNurbsSurface` filter is used.

```
MItSelectionList iter( selection, MFn::kNurbsSurface );
for ( ; !iter.isDone(); iter.next() )
      {
```

Only the NURBS surface shape nodes are needed from the selected objects.

```
MObject shapeNode;
iter.getDependNode( shapeNode );
```

An **MFnDependencyNode** function set is used to get a plug to the **shape** node's **create** attribute. This attribute holds a NURBS surface.

```
MFnDependencyNode shapeFn( shapeNode );
MPlug createPlug = shapeFn.findPlug( "create" );
```

The `melt` command inserts the new **melt** node between the current **makeNurbsSphere1** node and the **nurbSphereShape1** node, and you need to determine where the incoming connection to the **create** attribute comes from. If you didn't get the source of the connection, you couldn't later connect it into the new **melt** node. To get the source of a connection, the **MPlug**'s `connectedTo` function is used. It generates an array of plugs that are the source of the connection. Since there can be only one input connection to any attribute, why the need for an array? The `connectedTo` function is also used to query all the destination plugs of an attribute. Since an attribute can be fed into multiple destination attributes, an array is needed.

```
MPlugArray srcPlugs;
createPlug.connectedTo( srcPlugs, true, false );
```

Since it is known that there can be only one input connection to the **create** attribute, it must be the first element in the array. A truly robust plugin would check whether there are any elements in the array. This could happen, so it should be checked for, even though this example does not.

```
MPlug srcPlug = srcPlugs[0];
```

A new **melt** node is then created. To create a node, its unique identifier can be used. This is a more robust and error-proof method of specifying a particular node type. It is possible to use a node's type name, but this isn't as robust.

```
MObject meltNode = dgMod.createNode( MeltNode::id );
```

Plugs to the node's **inputSurface** and **outputSurface** attributes are then created.

```
MFnDependencyNode meltFn( meltNode );
MPlug outputSurfacePlug = meltFn.findPlug("outputSurface");
MPlug inputSurfacePlug = meltFn.findPlug("inputSurface");
```

The new **melt** node can now be connected to the existing nodes. Before establishing the new connections, the existing connection must be broken. This means that the connection between the **makeNurbsSphere1** and the **nurbSphereShape1** nodes must be broken. The **MDGModifer**'s `disconnect` function is used to do this. Its first argument is the source plug and its second is the destination plug. Combined, the two plugs uniquely identify a particular connection.

```
dgMod.disconnect( srcPlug, createPlug );
```

The various plugs are now connected.

```
dgMod.connect( srcPlug, inputSurfacePlug );
dgMod.connect( outputSurfacePlug, createPlug );
```

The new **melt** node has now been successfully inserted. An unfortunate consequence of using the **MDGModifier** is that it doesn't actually create nodes until its `doIt` function is called. Before its `doIt` function is called, the DG operations have simply been recorded but not executed. Therefore, it isn't possible to know ahead of time what the exact name of a new node will be. The node's name is determined when it is created. Each node must have a unique name so if a node already exists with the given name, Maya will automatically assign the new node a unique name. A number is appended to the given name to make a unique name. However, in the next section the exact name of the new node is needed in advance. In order to do this, a unique name must be assigned to the new node. Admittedly, the name encoding that is used isn't entirely robust, but it serves the purpose for this example. The **melt** node is renamed to the pregenerated name.

```
static i = 0;
MString name = MString("melting") + i++;
dgMod.renameNode( meltNode, name );
```

The next step is to set a keyframe for the **melt** node's **amount** attribute. At the start of the time range, the attribute's value is set to 0.0, and at the end of the time range, it is set to 1.0. This results in an object that starts as a solid on the first frame and then completely dissolves by the end of the last frame.

It is possible to create an **animCurve** node manually and connect it up to the **amount** attribute. Keyframes can then be set by using an **MFnAnimCurve** function set on the **animCurve** node. However, this is a simpler method. The MEL command, `setKeyframe`, can do all the hard work. The MEL statements are prepared in advance, then passed to the **MDGModifier**'s `commandToExecute` function. This first call to `setKeyframe` sets the first keyframe for the **amount** attribute.

```
MString cmd;
cmd = MString( "setKeyframe -at amount -t " )
                + startTime.value()
                + " -v " + 0.0 + " " + name;
dgMod.commandToExecute( cmd );
```

A similar command is used to generate a keyframe for the last frame.

```
    cmd = MString("setKeyframe -at amount -t ")
                  + endTime.value()
                  + " -v " + 1.0 + " " + name;
    dgMod.commandToExecute( cmd );
}
```

Finally the `redoIt` function is called to do the real work.

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

In the `redoIt` function, all the operations recorded in the **MDGModifier** object are executed by calling the **MDGModifier**'s `doIt` function.

```
MStatus MeltCmd::redoIt()
{
    return dgMod.doIt();
}
```

Similarly, undoing all the operations is handled by the **MDGModifier** in the `undoIt` function.

```
MStatus MeltCmd::undoIt()
{
    return dgMod.undoIt();
}
```

With an understanding of how the command created the **melt** node and connected it up, you are ready to cover the **melt** node. Recall that this node does the actual "melting" of the original surface.

Plugin: Melt
File: MeltNode.cpp

The **MeltNode** class is derived from the **MPxNode** class, as are all custom DG nodes. A temporary node identifier has been assigned to the node class.

```
const MTypeId MeltNode::id( 0x00334 );
```

The node has just three attributes. The **inputSurface** and **outputSurface** attributes are NURBS surfaces, while the **amount** attribute is a floating-point number.

```
MObject MeltNode::inputSurface;
MObject MeltNode::outputSurface;
MObject MeltNode::amount;
```

The `compute` function is where the actual melting takes place. Since the node has only one output attribute, **outputSurface**, it has to check only for this one.

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

    if( plug == outputSurface )
        {
```

Handles to all the attributes are obtained.

```
MDataHandle amountHnd = data.inputValue( amount );
MDataHandle inputSurfaceHnd = data.inputValue( inputSurface );
MDataHandle outputSurfaceHnd = data.outputValue( outputSurface );
```

The **amount** attribute is known to be a `double`, so the **MDataHandle**'s `asDouble` function is used to retrieve it.

```
double amt = amountHnd.asDouble();
```

It is very important to use the correct `as...` function, since using the wrong one could cause Maya to crash. As such, it is important to know in advance the type of data that a given attribute holds.

The **inputSurface** attribute is a NURBS surface. It is retrieved using the **MDataHandle**'s `asNurbsSurface` function. This function returns an **MObject** to the NURBS surface data.

```
MObject inputSurfaceObj = inputSurfaceHnd.asNurbsSurface();
```

Maya stores the more complex data types, such as NURBS surfaces, as special geometry data. In order to be able to create and store this kind of data, you need to use the appropriate **MFnGeometryData** function set. In this case, the appropriate function set is the **MFnNurbsSurfaceData** class. Using this function set, the NURBS surface data is created.

```
MFnNurbsSurfaceData surfaceDataFn;
MObject newSurfaceData = surfaceDataFn.create();
```

With a place to store the new surface, the original input surface can be copied to the new output surface.

```
MFnNurbsSurface surfaceFn;
surfaceFn.copy( inputSurfaceObj, newSurfaceData );
```

The output surface is now an exact duplicate of the original input surface. The output surface can now be deformed, without affecting the input surface. The first step is to attach the **MFnNurbsSurface** function set to the data. This function set can be used to access and change the NURBS surface data.

```
surfaceFn.setObject( newSurfaceData );
```

All the surface control vertices (CVs) are retrieved and stored in an array using the **MFnNurbsSurface**'s `getCVs` function.

```
MPointArray pts;
surfaceFn.getCVs( pts );
```

It is possible to get and set individual surface CVs using the **MFnNurbsSurface**'s `getCV` and `setCV` functions, but it is often easier to put them all into a single array and then operate on that array. This reduces the number of function calls and is thereby faster.

The vertical extents of the NURBS surface is now calculated. After this step, the **minHeight** and **maxHeight** variables hold the minimum (base) and maximum (top) *y* extent of the surface, respectively.

```
double minHeight = DBL_MAX, maxHeight = DBL_MIN, y;
unsigned int i;
for( i=0; i < pts.length(); i++ )
    {
    y = pts[i].y;
    if( y < minHeight )
        minHeight = y;
    if( y > maxHeight )
        maxHeight = y;
    }
```

The distance that all the CVs will move is calculated next. It is computed as a percentage of the vertical range of the NURBS surface. An **amount** of 1 corresponds to the entire vertical range, while 0.5 corresponds to half that distance. This means that when the **amount** is set to 1, all the CVs at the top of the object will be moved to the base. When it is 0.5, they will be halfway down.

```
double dist = amt * (maxHeight - minHeight);
```

All the CVs in the NURBS surfaces are then iterated over.

```
MVector vec;
double d;
for( i=0; i < pts.length(); i++ )
    {
    MPoint &p = pts[i];
```

Each point is moved down vertically by the required distance.

```
p.y -= dist;
```

Those points that are now under the original base are treated specially.

```
if( p.y < minHeight )
    {
```

Their distance under the base is calculated.

```
d = minHeight - p.y;
```

A 2D vector is calculated from the x and z coordinates of the CV. This is the vector from the object's center to the CV, looking from the top viewport.

```
vec = MVector( p.x, 0.0, p.z );
```

The length of this vector is then calculated.

```
double len = vec.length();
```

This length is used to determine how far the CV should spread out. If the CV is very, very close to the center of the object, then it shouldn't move. This ensures that the spreading calculation doesn't result in numerical inaccuracy and divide-by-zero errors. This also ensures that shapes such as spheres, tori, and so on do not have a hole in their underbody, since these points won't be moved.

```
if( len > 1.0e-3 )
    {
```

The new length of the vector is calculated by taking the vertical distance traveled under the base and then moving the CV horizontally by that same distance. This simulates the effect of the CV moving down to the base and then spreading out. The new vector is calculated based on this new length.

```
double newLen = len + d;
vec *= newLen / len;
```

The CV's x and z coordinates are now set to the new position.

```
    p.x = vec.x;
    p.z = vec.z;
}
```

The vertical height of the CV can't ever go below the original base.

```
        p.y = minHeight;
    }
}
```

Now that all the new positions for the CVs have been calculated, the NURBS surface is updated.

```
surfaceFn.setCVs( pts );
```

Since the NURBS surface has been changed, it is very important that Maya be notified of this. The **MFnNurbsSurface**'s `updateSurface` function must always be called after any changes are made to the surface.

```
surfaceFn.updateSurface();
```

The output NURBS surface attribute is now set to the new NURBS surface data.

```
outputSurfaceHnd.set( newSurfaceData );
```

Since the attribute has now been recomputed, the plug is marked as clean.

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

If another attribute that this node didn't define is being asked to recompute, return the `MS::kUnknownParameter` so that the base class can handle it.

```
else
    stat  = MS::kUnknownParameter;
```

Finally, return the success or failure of the function call.

```
    return stat;
}
```

The node's `initialize` function sets up the node's attributes.

```
MStatus MeltNode::initialize()
{
    MFnNumericAttribute nAttr;
    MFnTypedAttribute tAttr;
```

The **amount** attribute is defined to be a single `double` value.

```
amount = nAttr.create( "amount", "amt",
                       MFnNumericData::kDouble, 0.0 );
```

Since the **amount** attribute is animated, it must be made *keyable.* When an attribute is keyable, you can set keyframes for it. Attributes, by default, aren't keyable. Setting an attribute to keyable also makes it visible in the Channel Control.

```
nAttr.setKeyable( true );
```

The **inputSurface** and **outputSurface** attributes use the **MFnTypedAttribute** class for their creation. This class is used to create more complex attributes, such as NURBS surfaces.

```
inputSurface = tAttr.create( "inputSurface", "is",
                             MFnNurbsSurfaceData::kNurbsSurface );
```

Any attribute is visible by default. Since the input surface attribute is manually connected by the `melt` command, it is best to avoid having it displayed. Even if an attribute is hidden, it is still possible to disconnect and reconnect this attribute using the Connection Editor.

```
tAttr.setHidden( true );
```

The **outputSurface** attribute is created in a similar manner to the **inputSurface** attribute.

```
outputSurface = tAttr.create( "outputSurface", "os",
                              MFnNurbsSurfaceData::kNurbsSurface );
```

The **outputSurface** is the direct result of taking the **inputSurface** and **amount** attributes and calculating a new surface. The new surface is derived from the inputs and thus can be re-created with just the input. Since this new surface can always be regenerated from these input attributes, there is no need to store it in the Maya scene file. Not saving the attribute saves disk space, which can be important for scenes with lots of complex surfaces.

```
tAttr.setStorable( false );
```

Like the **inputSurface** attribute, the **outputSurface** attribute should be displayed in any of the main editing windows.

```
tAttr.setHidden( true );
```

All the attributes are then added to the node.

```
addAttribute( amount );
addAttribute( inputSurface );
addAttribute( outputSurface );
```

Both the **amount** and **inputSurface** attributes directly affect the **outputSurface** attribute. This dependency is specified using the **MPxNode**'s `attributeAffects` function.

```
    attributeAffects( amount, outputSurface );
    attributeAffects( inputSurface, outputSurface );

    return MS::kSuccess;
}
```

While this example was more complex than the previous, the basic principles didn't change. The node still specified which attributes it contained and what their types were. It specified that certain attributes affected others. Even when a more complex data type (NURBS surface) was used, there weren't any large changes needed to support it.

4.5.3 GroundShadow Plugin

This project covers the creation of a command and node, named **groundShadow**, that generates interactive ground shadows. Given a point light source and a set of objects, the plugin generates the projected shadows of the objects onto the ground. Since it operates as a node that creates its output dynamically, the light position can be moved, and the resulting shadows are automatically regenerated. Also if the geometry of any of the objects is altered, the projected shadows are automatically updated.

Figure 4.24 shows the object's that will cast shadows. The resulting shadows, from the **groundShadow** plugin, are shown in Figure 4.25.

An important addition to this example is the use of an attribute that can accept different types of geometry. In the previous example, a single type of geometry, NURBS surface, could be operated on. The way the attribute is set up, a polygonal mesh could be used, for instance. This project shows you how to set up an attribute that takes different types of geometry.

1. Open the **GroundShadow** workspace.
2. Compile it, and load the resulting `GroundShadow.mll` plugin in Maya.
3. Open the **GroundShadow.ma** scene.
4. Select **Shading | Smooth Shading All**.
5. Select **Lighting | All Lights**.
6. Click on the **Play** button.

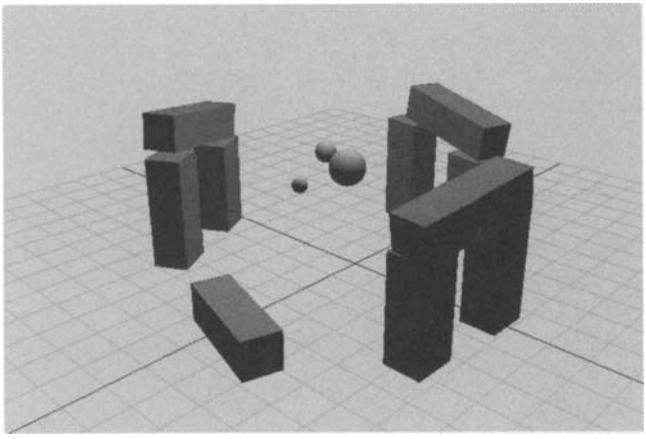


FIGURE 4.24 Original objects

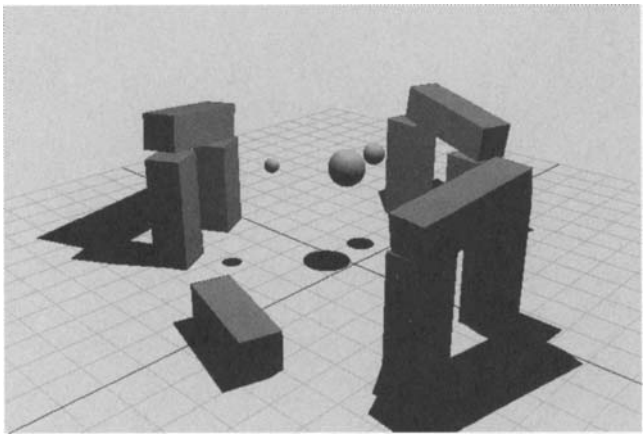


FIGURE 4.25 Shadows cast onto the ground

The single point light moves across the scene while the small spheres rotate around each other.

7. Select all the objects including the point light.
8. In the **Script Editor**, execute the following:

```
groundShadow;
```

Ground shadows are created for all the selected objects.

9. Unselect all the objects.
10. Click on **Play**.

The ground shadows interactively change when the light moves.

The `groundShadow` command is responsible for creating the various geometry and **groundShadow** nodes, as well as connecting their attributes together.

The `groundShadow` command functions by taking all the selected objects and determining which of them are geometry nodes. A geometry node contains some form of geometry, including NURBS surfaces, curves, polygonal meshes, and so on. A shadow node is created for each of the selected geometry objects. A shadow node is an exact duplicate of the original geometry. The only difference is that the geometry is flattened and distorted to create a flat object in the shape of the shadow. The **groundShadow** node is responsible for performing this distortion.

Plugin: GroundShadow
File: GroundShadowCmd.h

The **GroundShadowCmd** class is derived from the **MPxCommand** class.

```
class GroundShadowCmd : public MPxCommand
{
public:
    virtual MStatusdoIt ( const MArgList& );
    virtual MStatus undoIt();
    virtual MStatus redoIt();
    virtual bool isUndoable() const { return true; }

    static void *creator() { return new GroundShadowCmd; }
    static MSyntax newSyntax();
```

The main difference from previous commands is that the class contains an **MDagModifier** rather than the usual **MDGModifier**. The **MDagModifier** is an extension of the **MDGModifier** class to handle DAG nodes. It understands about the DAG hierarchy and so lets you reorganize DAG node parent-child relationships. Since this command will be creating DAG nodes and reparenting them, this class is used instead.

```
private:
    MDagModifier dagMod;
};
```

Plugin: GroundShadow
File: GroundShadowCmd.cpp

The **groundShadow**'s `doIt` function does all the command preparation.

```
MStatus GroundShadowCmd::doIt ( const MArgList &args )
{
MStatus stat;
MSelectionList selection;
```

It is important to note that when a new shape node is created using the C++ API, it isn't automatically assigned the default shading group, **initialShadingGroup**. As such, the new shape must be explicitly added to the group. If this isn't done, the new shape will appear without shading and use a random color, often magenta.

The first step is, therefore, to get the **initialShadingGroup** node.

```
MObject shadingGroupObj;
selection.clear();
MGlobal::getSelectionListByName( "initialShadingGroup", selection );
selection.getDependNode( 0, shadingGroupObj );
```

An **MFnSet** is attached to the shading group, since shading groups are really just sets.

```
MFnSet shadingGroupFn( shadingGroupObj );
```

The currently selected objects are put in the `selection` list.

```
selection.clear();
MGlobal::getActiveSelectionList( selection );
MItSelectionList iter( selection );
```

In order to know where the light is coming from, at least one point light must have been selected. This next step iterates over the selection and determines if a point light has been selected.

```
MDagPath lightTransformPath;
MPlug pointLightTranslatePlug;
iter.setFilter( MFn::kPointLight );
for ( iter.reset(); !iter.isDone(); iter.next() )
      {
      iter.getDagPath( lightTransformPath );
```

When the path for a selected object is requested, the entire DAG path, right down to the **shape** node, is returned. Since only the shape's **transform** node is needed, the last node is popped off the DAG path. This resulting DAG path then includes all the **transform** nodes in the hierarchy above the **shape** node.

```
lightTransformPath.pop();
```

An **MPlug** is created to get the transform's **translate** attribute. This is used to determine the position of the light.

```
MFnDagNode dagNodeFn( lightTransformPath );
pointLightTranslatePlug = dagNodeFn.findPlug( "translate" );
```

Since only the first selected light is taken into consideration, the loop can now stop.

```
break;
}
```

Next, it is determined whether the light's transform path is valid. Even though the **MItSelectionList** would iterate only over point lights, if none were selected, the `lightTransformPath` would never have been set. As such, it is checked here. If it isn't, then no point light must have been selected, and you can exit with an error.

```
if( !lightTransformPath.isValid() )
    {
    MGlobal::displayError( "\nSelect a point light." );
    return MS::kFailure;
    }
```

The light's necessary information has now been retrieved. The next step is to iterate over all the selected geometric shapes and create shadow shapes for them.

```
iter.setFilter( MFn::kGeometric );
for ( iter.reset(), count=0; !iter.isDone(); iter.next(), count++ )
      {
      MDagPath geomShapePath;
      iter.getDagPath( geomShapePath );
```

The complete path to the shape is retrieved. The path is popped to remove the last item, the **shape** node, so that the path includes only the parent **transform** nodes.

```
MDagPath geomTransformPath( geomShapePath );
geomTransformPath.pop();
```

The shadow **shape** node that is created is simply a duplicate of the original shape. The duplicate function is called with the instance argument set to `false` so that a copy of the shape rather than an instance is created. Since this shape is later distorted to create a flat shadow shape, the original shape should be left untouched. If the original **shape** node were instanced, the shadow **shape** node would deform the common shape they would both share, which isn't the intention.

```
MFnDagNode geomShapeFn( geomShapePath );
MObject newGeomTransformObj = geomShapeFn.duplicate( false, false );
```

The **MFnDagNode** function set, `newGeomShapeFn`, is attached to the new **shape** node. The **shape** node is the first child of the parent **transform** node, so the **MFnDagNode**'s `child` function is used.

```
MFnDagNode newGeomShapeFn( newGeomTransformObj );
newGeomShapeFn.setObject( newGeomShapeFn.child(0) );
```

When the new shape is created, it is placed under a new **transform** node. You'd like the new **shape** node to reside under the original shape's parent **transform** node. Setting it up this way ensures that if the original shape moves, the shadow shape also moves, since they share the same parent transform hierarchy.

```
dagMod.reparentNode( newGeomShapeFn.object(), geomTransformPath.node() );
```

As mentioned earlier, Maya doesn't automatically assign a new **shape** node the default shading group, so it must be done manually. Assigning a **shape** node to a shading group consists of simply adding it to the set using the **MFnSet** `addMember` function.

```
shadingGroupFn.addMember( newGeomShapeFn.object() );
```

The **groundShadow** node is created next. It is responsible for distorting the shadow shape to appear as the shadow.

```
MObject shadowNode = dagMod.MDGModifier::createNode(
                                                GroundShadowNode::id );
assert( !shadowNode.isNull() );
MFnDependencyNode shadowNodeFn( shadowNode );
```

Plugs for the various attributes are then set up for the **groundShadow** node.

```
MPlug castingSurfacePlug = shadowNodeFn.findPlug( "castingSurface" );
MPlug shadowSurfacePlug = shadowNodeFn.findPlug( "shadowSurface" );
MPlug lightPositionPlug = shadowNodeFn.findPlug( "lightPosition" );
```

The **groundShadow** node and command are designed to support two different types of geometry: polygonal meshes and NURBS surfaces. Each geometry type has different input and output plugs. For a mesh node, the input geometry comes into the node through its **inMesh** attribute, while a NURBS surface has its input geometry come in through the **create** attribute. To retrieve the right attribute based on the geometry type, the `outGeomPlugName` and `inGeomPlugName` strings are set up. To determine the **shape** node's type, the **MDagPath**'s `apiType` function is used.

```
MString outGeomPlugName, inGeomPlugName;
switch( geomShapePath.apiType() )
        {
        case MFn::kMesh:
            outGeomPlugName = "worldMesh";
            inGeomPlugName = "inMesh";
            break;

        case MFn::kNurbsSurface:
            outGeomPlugName = "worldSpace";
            inGeomPlugName = "create";
            break;
        }
```

A plug is initialized to the geometry shape's associated output attribute.

```
MPlug outGeomPlug = geomShapeFn.findPlug( outGeomPlugName );
```

It is important to remember that the output geometry attribute isn't just a single output but instead an array of outputs. There exists a different output for each instance of the shape, so a shape that has been instanced three times has three elements in its output geometry array. To determine to which of the instances the currently selected object corresponds, the **MDagPath**'s `instanceNumber` function is called. This returns the instance number of the selected object. This number directly corresponds to the instance's index in the output geometry array.

```
unsigned int instanceNum = geomShapePath.instanceNumber();
```

At this point, the `outGeomPlug` plug is pointing to the output geometry's parent attribute, the array. You want it to point to a particular element in the array. This is

done using the **MPlug**'s `selectAncestorLogicalIndex` function. This sets the plug to point to the array element with the given index.

```
outGeomPlug.selectAncestorLogicalIndex( instanceNum );
```

The input geometry plug, `inGeomPlug`, is set the geometry node's input attribute.

```
MPlug inGeomPlug = newGeomShapeFn.findPlug( inGeomPlugName );
```

The **MDagModifier** member is set up to connect the various attributes. The position of the point light is fed into the **shadow** node's light position attribute. The exact attributes of the **GroundShadowNode** are described in the next section. The output geometry is fed into the **shadow** node's casting surface attribute. This is the surface that generates the shadow. The result of the **shadow** node is another surface that is distorted to represent the final flat shadow. This resulting surface is fed into the input geometry attribute of the duplicated shape. This is the shadow shape that is displayed in the scene.

```
dagMod.connect( pointLightTranslatePlug, lightPositionPlug );
dagMod.connect( outGeomPlug, castingSurfacePlug );
dagMod.connect( shadowSurfacePlug, inGeomPlug );
}
```

If none of the objects are either meshes or NURBS surfaces, then the function exits with an error.

```
if( count == 0 )
    {
    MGlobal::displayError("\nSelect one or more geometric objects.");
    return MS::kFailure;
    }
```

With the **dagMod** object now set up, its `redoIt` function can be called to perform the actual work.

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

The `redoIt` and `undoIt` functions are similar in form to the previous commands. The only difference is that the **MDagModfier** class is used in place of the usual **MDGModifier** class.

```
MStatus GroundShadowCmd::redoIt()
{
return dagMod.doIt();
}

MStatus GroundShadowCmd::undoIt()
{
return dagMod.undoIt();
}
```

The **groundShadow** node takes some input geometry and produces some output geometry. The node also takes the light's position as input. Taking into account the light position and input geometry, the node computes where the shadow of the geometry would lie on the ground plane. The resulting output geometry is simply a flat, distorted version of the original input geometry. This flattened geometry is displayed as the object's shadow. An additional extra that hasn't been discussed is the **groundShadow** node's **groundHeight** attribute. You can change this to set the ground plane to a new height.

Plugin: GroundShadow
File: GroundShadowNode.h

The **GroundShadowNode** class is a standard DG node, so it is directly derived from the **MPxNode** class. It contains all the usual member functions. Its attributes are covered in the next section.

```
class GroundShadowNode : public MPxNode
{
public:
    virtual MStatus compute( const MPlug& plug, MDataBlock& data );

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

    static const MTypeId id;

public:
    static MObject lightPosition;
    static MObject castingSurface;
    static MObject shadowSurface;
    static MObject groundHeight;
};
```

Plugin: GroundShadow
File: GroundShadowNode.cpp

The node's attributes are first defined. The **lightPosition** attribute is the position of the light source in the scene. The **castingSurface** attribute is the original object's surface. The **shadowSurface** attribute holds the resulting shadow surface. The **groundHeight** attribute contains a single floating-point value for the height of the ground plane.

```
MObject GroundShadowNode::lightPosition;
MObject GroundShadowNode::castingSurface;
MObject GroundShadowNode::shadowSurface;
MObject GroundShadowNode::groundHeight;
```

The `compute` function is responsible for taking the input attributes (**lightPosition**, **castingSurface**, and **groundHeight**) and creating the final shadow surface in the **shadowSurface** attribute.

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

This node's only output attribute is the **shadowSurface** attribute, which it is checked for.

```
if( plug == shadowSurface )
    {
```

MDataHandle objects to all the input and output attributes are set up.

```
MDataHandle groundHeightHnd = data.inputValue( groundHeight );
MDataHandle lightPositionHnd = data.inputValue( lightPosition );
MDataHandle castingSurfaceHnd = data.inputValue(castingSurface);
MDataHandle shadowSurfaceHnd = data.outputValue( shadowSurface );
```

A copy of the original casting surface is made and put into the shadow surface handle, `shadowSurfaceHnd`. At this point the shadow surface is an exact duplicate of the casting surface.

```
shadowSurfaceHnd.copy( castingSurfaceHnd );
```

The next section of code initializes variables that are used later in the calculation of the final shadow geometry. The height of the ground plane is first retrieved.

```
double gHeight = groundHeightHnd.asDouble();
```

The position of the light source is stored in the **Mvector**'s `lightPoint`.

```
MVector lightPoint( lightPositionHnd.asDouble3() );
```

The *normal* to the ground plane points directly upward, that is, in the direction of the y-axis.

```
MVector planeNormal( 0.0, 1.0, 0.0 );
```

A point is created that is known to lie on the ground plane.

```
MVector planePoint( 0.0, gHeight, 0.0 );
```

The dot product between the normal and the point is calculated.

```
double c = planeNormal * planePoint;
```

Each point in the shadow surface geometry is now distorted. The distortion is the result of projecting the points from the light location onto the ground plane.

```
MPoint surfPoint;
double denom, t;
MItGeometry iter( shadowSurfaceHnd, false );
for( ; !iter.isDone(); iter.next() )
    {
```

The position of the geometry in world space coordinates is retrieved.

```
surfPoint = iter.position( MSpace::kWorld );
```

An imaginary line between the surface point and the light source is then created. The new position of the surface point is where this line intersects the ground plane.

```
denom = planeNormal * (surfPoint - lightPoint);
    if( denom != 0.0 )
        {
        t = (c - (planeNormal * lightPoint)) / denom;
        surfPoint = lightPoint + t * (surfPoint - lightPoint);
        }
```

The surface point's position is updated to the projected position.

```
iter.setPosition( surfPoint, MSpace::kWorld );
}
```

The output surface attribute has now been updated.

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

If the attribute that Maya is requesting to be recomputed isn't known to this node, the appropriate status code is returned.

```
else
    stat  = MS::kUnknownParameter;

return stat;
}
```

The `initialize` function sets up the blueprints for the node's attributes.

```
MStatus GroundShadowNode::initialize()
{
```

The **lightPosition** attribute is simply a position. It is stored as three doubles.

```
MFnNumericAttribute nAttr;
lightPosition = nAttr.create( "lightPosition", "lpos",
                              MFnNumericData::k3Double, 0.0 );
nAttr.setKeyable( true );
```

The **groundHeight** attribute is just a single distance value. Since it is a distance unit, the **MFUnitAttribute** class is used.

```
MFnUnitAttribute uAttr;
groundHeight = uAttr.create( "groundHeight", "grnd",
                             MFnUnitAttribute::kDistance, 0.0 );
uAttr.setKeyable( true );
```

The **castingSurface** attribute is of type **MFnGenericAttribute**. This attribute allows you to define a variety of different data types that it will accept. In this case the attribute should accept both meshes and NURBS surfaces.

```
MFnGenericAttribute gAttr;
castingSurface = gAttr.create( "castingSurface", "csrf" );
gAttr.addAccept( MFnData::kMesh );
gAttr.addAccept( MFnData::kNurbsSurface );
gAttr.setHidden( true );
```

Since the **shadowSurface** attribute is the result of deforming a copy of the **castingSurface** attribute, it must support the same type of data as the **castingSurface** attribute. It is therefore the same attribute type as the **castingSurface**.

```
shadowSurface = gAttr.create( "shadowSurface", "ssrf" );
gAttr.addAccept( MFnData::kMesh );
gAttr.addAccept( MFnData::kNurbsSurface );
gAttr.setHidden( true );
```

Since this is an output attribute, it can be regenerated from the node's inputs, so it doesn't have to be stored in the Maya scene file.

```
gAttr.setStorable( false );
```

All the node's attributes are now added.

```
addAttribute( groundHeight );
addAttribute( lightPosition );
addAttribute( castingSurface );
addAttribute( shadowSurface );
```

When any of the attributes, **groundHeight**, **lightPosition**, or **castingSurface**, change, the **shadowSurface** attribute is affected. This relationship is next defined.

```
attributeAffects( groundHeight, shadowSurface );
attributeAffects( lightPosition, shadowSurface );
attributeAffects( castingSurface, shadowSurface );

return MS::kSuccess;
}
```

4.5.4 Attributes

All node attributes are defined using a class derived from **MFnAttribute**. Depending on what type of attribute you'd like to create, the appropriate **MFnAttribute** derived class is used.

Unfortunately Maya uses the same term, *attribute,* to mean two slightly different things. This creates two different perspectives. From a user's perspective, each node has a series of attributes. The user changes and animates these attributes. From a programmer's perspective, attributes are somewhat different. This section refers to the term *attribute* as it is understood from the programmer's perspective.

In the C++ API, the **MFnAttribute** and all its derived classes can be intuitively thought of as a template or blueprint for how a piece of data in the node should be created. The most important distinction from the user's perspective is that the attribute doesn't actually hold any data. It simply provides a specification for the data. Given this specification, the actual data is created.

For instance, an **MFnAttribute** derived class defines the name and type of the data within the node. It also specifies whether the data should be stored to disk and whether its value can be changed (read/write). An attribute defines the specification for a single piece of data. For instance, an attribute may specify that the node is going to contain a `float` named **amplitude**. When the node is created, it prepares a slot for a `float` and gives it the name **amplitude**. Each node has its own unique **amplitude** data value. If you change the **amplitude** value in one node, it doesn't affect the **amplitude** value in other nodes.

Maya takes the information defined in the attribute and then uses it to actually create the data that exists inside each of the individual nodes. Since an attribute is the blueprint, it follows logically that it is defined only once. By convention attributes are defined in the node's static member function called `initialize`, though you are free to use any function.

CREATING ATTRIBUTES

An attribute is created inside the initialize function. It is then registered with the node by using the **MPxNode**'s `addAttribute` function. The following example creates an attribute, **days**, that is registered with the node.

```
class MyNode : public MPxNode
{
...
  static MStatus initialize();
  static MObject daysAttr;
};

MObject MyNode::daysAttr;

MStatus MyNode::initialize()
{
MFnNumericAttribute numFn;
daysAttr = numFn.create( "days", "d", MFnNumericData::kLong );

addAttribute( daysAttr );

return MS::kSuccess;
}
```

This same basic format is used for all attributes. The main difference is that, depending on the type of attribute, a different **MFn...Attribute** class will be used.

COMPOUND ATTRIBUTES

A compound attribute is used to group other attributes. The grouped attributes are considered *children* of the compound attribute, which is itself considered the *parent.* In this example a compound attribute, **player**, is created with two child attributes, **age** and **homeruns**.

```
class MyNode : public MPxNode
{
...
  static MStatus initialize();
  static MObject playerAttr;
  static MObject ageAttr;
  static MObject homeRunsAttr;
};

MObject MyNode::playerAttr;
MObject MyNode::ageAttr;
MObject MyNode::homeRunsAttr;

MStatus MyNode::initialize()
{
```

The child attributes are defined as usual.

```
MFnNumericAttribute numFn;
ageAttr = numFn.create( "age", "a", MFnNumericData::kLong );
homeRunsAttr = numFn.create( "homeruns", "hr", MFnNumericData::kLong );
```

The compound attribute uses the **MFnCompoundAttribute** class. It is created like any attribute.

```
MFnCompoundAttribute compFn;
playerAttr = compFn.create( "player", "ply" );
```

The child attributes are then added to the compound attribute.

```
compFn.addChild( nameAttr );
compFn.addChild( homeRunsAttr );
```

Last all the attributes, both child and parent, are added to the node. It is important to note that when you add a compound attribute all its children aren't automatically added. The children must be explicitly added.

```
addAttribute( ageAttr );
addAttribute( homeRunsAttr );
addAttribute( playerAttr );

return MS::kSuccess;
}
```

DEFAULT VALUES

Each attribute has a default value. This is the value with which the attribute is initialized. Depending on the particular class, you will have the option of setting the default through the function set's `create` function or using an appropriate default setting function.

For example, for attributes that use enumerated types (**MFnEnumAttribute**), the default is set in the `create` function as follows. This sets the attribute's default to 0.

```
MFnEnumAttribute enumFn;
attr = enumFn.create( "days", "d", 0 );
...
```

The **MFnNumericAttribute** class is used to create numeric attributes. Its default can be set using the `create` function or its `setDefault` member function. The following demonstrates how to use the latter method:

```
MFnNumericAttribute numFn;
attr = numFn.create( "active", "act", MFnNumericData::kBoolean );
numFn.setDefault( false );
```

If the user never changes an attribute's value, it retains its default value. If an attribute's value is the same as its default, it won't be stored to disk. Since the value is no different from the default, there is no need to store it. Its value can be simply initialized to the default rather than to a value on disk. This reduces the size of the scene files stored to disk.

An important consequence of this is that if you later change the default values for your attributes, those attributes that used the previous default values automatically use the new defaults. This may not always be what you intended. This may "break" existing scenes, since those attributes that used the default now have a different value. The best way to avoid this is to decide early in the node's development what an attribute's default will be. By fixing on a default value before any Maya scenes are created using the node, you won't encounter this problem.

ATTRIBUTE PROPERTIES

An attribute has various properties. These properties define, for instance, whether you can change the attribute's value or make connections to it. Table 4.2 lists the common properties that all attributes share.

TABLE 4.2 COMMON ATTRIBUTE PROPERTIES

Property	Description	Default
`Readable`	Can be the source of connections	`true`
`Writable`	Can be the destination of connections	`true`
`Connectable`	Can be connected	`true`
`Storable`	Is stored to scene files	`true`
`Keyable`	Can be animated	`false`
`Hidden`	Is hidden	`false`
`UsedAsColor`	Treat values as colors	`false`
`Cached`	Value is cached	`true`
`Array`	Is an array	`false`
`IndexMatter`	Index shouldn't change	`true`
`ArrayDataBuilder`	Uses array data builder to set value	`false`
`Indeterminant`	Determines if it can be used in an evaluation	`false`
`DisconnectBehavior`	Behavior after disconnection	`kNothing`
`Internal`	Is internal to the node	`false`
`RenderSource`	Overrides rendering sampling information	`false`

The following section covers some of the most important properties in greater detail.

Readable and Writable

An attribute has a *readable* and *writable* flag. By default, both these flags are set to `true`. Interestingly enough an attribute's value can always be retrieved, irrespective of whether its readable flag is set. The readable flag defines whether the attribute can be used as the source of a connection. If it is set to `false`, then you won't be able to create a connection from the attribute to another attribute. The `isReadable` and `setReadable` functions are used to retrieve and set the readable flag, respectively.

An example using these functions is as follows:

```
MFnMessageAttribute msgFn;
attr = msgFn.create( "controller", "ctrl" );
bool isread = msgFn.isReadable(); // Returns true
msgFn.setReadable( false );
```

When an attribute doesn't have its writable flag set to `true`, you can't change its value using the `setAttr` command. It also can't be used as the destination in a connection, which means that you can't create a connection from another attribute to this attribute. The following example demonstrates the various writable flag retrieval and setting functions:

```
MFnMessageAttribute msgFn;
attr = msgFn.create( "controller", "ctrl" );
bool iswrite = msgFn.isWritable (); // Returns true
msgFn.setWritable ( false );
```

Connectable

As mentioned previously, the readable and writable flags determine whether you can use the attribute as the source and destination of a connection, respectively. However, the connectable flag ultimately determines if any connection can be made. If the readable flag is set to `true`, but the connectable flag is set to `false`, then you won't be able to use the attribute as the source of a connection. Both flags must be set to `true` for this to work. As such the connectable flag is used in combination with the readable and writable flags to determine what types of connections are valid.

This example shows how to retrieve and set the connectable flag.

```
MFnMatrixAttribute mtxFn;
attr = mtxFn.create( "xform", "xfm", MFnMatrixAttribute::kDouble );
bool iscon = mtxFn.isConnectable(); // Returns true
mtx.setConnectable( false );
```

Keyable

When an attribute is keyable, you can animate it by setting keyframes. By default, an attribute isn't keyable. Consequently it isn't shown in the Channel Box. The `isKeyable` and `setKeyable` functions are used to retrieve and set the keyable flag, respectively.

```
MFnMatrixAttribute mtxFn;
attr = mtxFn.create( "xform", "xfm", MFnMatrixAttribute::kDouble );
bool canKey = mtxFn.isKeyable(); // Returns false
mtx.setKeyable( true );
```

Storable

The storable flag determines whether an attribute's values are stored to disk. By setting this to `false`, the attribute won't be stored. By default, an attribute is storable.

Certain attributes don't necessarily have to be stored to disk. These attributes include those that have their value computed from other attributes: output attributes. The values of these attributes are derived from other input attributes. Since they are derived, they can be reconstructed from the input attributes. As such, there is no need to store them to disk, since they can be reconstructed. While it isn't illegal to store these attributes, they use disk space unnecessarily. So when you know that an attribute is an output attribute—that is, it is used as the dependent attribute in an `attributeAffects` function call—it doesn't have to be stored to disk. The following example demonstrates this:

```
MFnNumericAttribute numFn;
widthAttr = numFn.create( "width", "w", MFnNumericData::kDouble );

sizeAttr = numFn.create( "size", "s", MFnNumericData::kDouble );
numFn.setStorable( false ); // No need to store it

attributeAffects( widthAttr, sizeAttr );
```

Array

By default, an attribute holds a single instance of its data. An attribute can be made to hold a series of data instances by making it into an array. Array attributes are also referred to as multis. By default, an attribute isn't an array. The `isArray` and `setArray` functions are used to retrieve and set the array property, respectively. In the following example, an attribute is made to hold an array of doubles:

```
MFnNumericAttribute numFn;
sizesAttr = numFn.create( "sizes", "s", MFnNumericData::kDouble );
numFn.setArray( true ); // Can no hold an array of doubles
```

The methods for accessing and adding elements to the array are discussed in a later section.

Cached

An attribute, by default, is cached. This means that a copy of the attribute data is maintained in the node's datablock. By caching an attribute's value, the `compute` function doesn't have to be continually called whenever the attribute's value is requested. This makes it faster to retrieve node attribute values. The downside to caching attributes is that it requires more memory. Caching can be turned on or off using the `setCache` function.

```
MFnNumericAttribute numFn;
sizesAttr = numFn.create( "sizes", "s", MFnNumericData::kDouble );
numFn.setCached( false ); // No longer cached
```

DYNAMIC ATTRIBUTES

The `initialize` function is where you define the various attributes of a node. Each instance of the node will use these attributes. These attributes are known as static attributes since they always exist for each instance of the node. There is no way to remove these attributes from the node.

It is also possible to create *dynamic attributes.* These are attributes that are added later as needed. A dynamic attribute can be shared between all nodes of a given type or be unique to a particular node.

To create a dynamic attribute, an attribute is first defined. It is defined in the same way as static attributes.

```
MFnNumericAttribute numFn;
MObject attr = numFn.create( "size", "s", MFnNumericData::kDouble );
```

This attribute is then added to the node by using the **MFnDependencyNode**'s `addAttribute` function. In this case, the `MFnDependencyNode::kLocalDynamicAttr` value is passed to the `addAttribute` function. This indicates that the attribute should be added to this particular node and not to all nodes of the same type.

```
MFnDependencyNode nodeFn( nodeObj );
nodeFn.addAttribute( attr, MFnDependencyNode::kLocalDynamicAttr );
```

To add an attribute to all instances, both existing and future, of a given node type, the `MFnDependencyNode::kGlobalDynamicAttr` value should be used when calling

the addAttribute function. It is important to use a unique name for the attribute. If the node already has an attribute with the given name, the dynamic attribute won't be added. A dynamic attribute can be deleted later using the removeAttribute function.

```
nodeFn.removeAttribute( attr, MFnDependencyNode::kLocalDynamicAttr );
```

4.5.5 Compute Function

The compute function is where the output attributes are calculated from the input attributes. The compute function takes a reference to the **MdataBlock**, which is used to retrieve and set the various node attributes. It is extremely important to use only the **MDataBlock** to get all the information the node needs to calculate its outputs. No data from outside the **MDataBlock** should be used in the computation.

It is also extremely important to not use the MEL setAttr and getAttr commands inside of the compute function. These commands can indirectly cause the DG to evaluate. If during this evaluation the value of the same plug is requested, either directly or indirectly, an infinite loop results.

Take the following example. Although contrived, it does demonstrate the danger of using getAttr inside the compute function. An instance of the **MyNode** class is created, **myNode1**. Assume that the node has two attributes; **scores**, an array of player scores, and **average**, which gives the average score. If the value of the **average** attribute is requested, the node is asked to compute this value. When the compute function is called and executes the MGlobal::executeCommand part, an infinite loop occurs.

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

if( plug == avgAttr )
    {
    MArrayDataHandle scoresHnd = data.inputArrayValue( scoresAttr );
    MDataHandle avgHnd = data.outputValue( avgAttr );
    MGlobal::executeCommand( "getAttr myNode1.average" );
    data.setClean( plug );
    }
else
    stat  = MS::kUnknownParameter;

return stat;
}
```

This leads to a very important rule that is key to preventing any indirect DG reevaluations when you are inside of a `compute` function.

Get attribute values only from the MDataBlock.

Note that while it isn't illegal to get the value of the plug that is being recomputed, there often isn't any point. Using **MPlug**'s `getValue` and **MDataHandle**'s `asDouble` won't cause an infinite loop.

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

if( plug == avgAttr )
    {
    MArrayDataHandle scoresHnd = data.inputArrayValue( scoresAttr );
    MDataHandle avgHnd = data.outputValue( avgAttr );
    double value;
    plug.getValue( value ); // Retrieve value
    MDataHandle resInHnd = data.inputValue( resAttr );
    value = resInHnd.asDouble (); // Retrieve value
    data.setClean( plug );
    }
else
    stat  = MS::kUnknownParameter;

return stat;
}
```

You should never rely on the previous value of a plug in order to compute its current value. In fact, you can never make any assumption about the current value of a plug. Since plugs can be evaluated at different times, there is no guarantee that the plug will, for instance, hold the value it had from the previous frame.

4.5.6 Plugs

In the context of the C++ API, an attribute is simply a blueprint for creating data within a node. An attribute doesn't hold any data but instead provides a specification for how that data should be created. Given a particular instance of a node, a *plug* is used actually to access the node's data. A plug provides a mechanism for accessing

the actual data for a given node. A plug is created by specifying a particular node and attribute. Using this combination, a plug refers to the actual data of a given node. Plugs are created and accessed using the **MPlug** class.

The following example creates an **MPlug** to the **translateX** attribute of the given **transform** node, `ballObj`. The **MFnDependencyNode**'s `findPlug` function is used to create a plug to a given attribute. The attribute name given can be either the long or short form.

```
MFnDependencyNode nodeFn( ballObj );
MPlug transxPlg = nodeFn.findPlug( "translateX" );
```

The `findPlug` function is also overloaded to take an attribute object registered in the node's `initialize` function. Since a direct reference to the attribute object is often available only in the node's implementation file, this is rarely used.

With the plug now created, the attribute data can now be retrieved using **MPlug**'s `getValue` function.

```
double tx;
transxPlg.getValue( tx );
```

The `getValue` function has been overloaded to retrieve many different types of data. It is therefore important to ensure that the data type being retrieved matches the attribute's data type. Since the **translateX** attribute is of type `double`, it is retrieved into a `double` variable. If you attempt to retrieve an attribute's data but with the wrong type, the result will be unpredictable. It may even cause Maya to crash. The following example demonstrates trying to retrieve the same attribute but as a `short`:

```
short tx;
transxPlg.getValue( tx ); // Unpredictable result
```

This is the most common mistake when using plugs, so be sure that the type of data you are attempting to retrieve matches the attribute's data type.

To set the value of a plug, use the `setValue` function. The following example shows how to set the plug's value to 2.3. Note the use of the explicit type case to ensure that the appropriate overloaded version of the `setValue` function is called.

```
transxPlg.setValue( double(2.3) );
```

By default, the value of an attribute is retrieved at the current time. It is also possible to retrieve its value at another time. To do this, use the **MDGContext** class. First set it to the required evaluation time, then use the call to the `getValue` function. In the following example, the **MTime** variable `t` is set to the time 1.5 seconds. This variable is then used to initialize the **MDGContext** variable `ctx`. This `ctx` variable is passed to the **MPlug**'s `getValue` function to retrieve the attribute value at the given time.

```
MTime t(1.5, MTime::kSeconds );
MDGContext ctx(t);
transxPlug.getValue( tx, ctx );
```

It isn't possible to use the `setValue` function at an alternative time. It always sets the attribute's value at the current time. To change the current time, use the **MGlobal**'s `viewFrame` function. Note that this function often requires the DG to update, so it should be used sparingly.

```
MTime t(1.5, MTime::kSeconds );
MGlobal::viewFrame( t );
```

To create animated values for the attribute, the MEL `keyframe` command should be used.

Compound Plugs

The **MPlug** class is also used to navigate compound attributes. In this next example, a plug is created for a **transform** node's **translate** attribute.

```
MFnDependencyNode nodeFn( transformNodeOj );
MPlug transPlg = nodeFn.findPlug( "translate" );
```

This `transPlg` variable is now referencing the **translate** attribute. This is a compound attribute containing three child attributes: **translateX**, **translateY**, and **translateZ**. To access the child, use the **MPlug**'s `child` function.

```
MObject transxAttr = nodeFn.attribute( "translateX" );
MPlug transxPlg = transPlg.child( transXAttr );
```

The parent of a child plug can be retrieved using the **MPlug**'s `parent` function.

```
MPlug parent = transxPlg.parent();
```

If you don't know the exact number and names of the children, the **MPlug**'s `numChildren` and `child` functions can be used. The `child` function is overloaded to take an index to the *n*th child. For instance, to iterate over all the children in a compound attribute, use the following:

```
const unsigned int nChilds = transPlg.numChildren();
unsigned int i;
for( i=0; i < nChilds; i++ )
     {
     MPlug child = transPlg.child(i);
     ... use the child plug
     }
```

Array Plugs

An array attribute contains a series of other attributes. These attributes are referred to as the elements of the array. A plug that references the array is referred to as an array plug, while a plug that references an element is an element plug.

In the following example, a custom node is created that contains an array attribute, **scores**. This attribute contains a series of `long` integer values.

```
class MyNode : public MPxNode
{
...
  static MStatus initialize();
  static MObject scoresAttr;
};

MObject MyNode::scoredAttr;

MStatus MyNode::initialize()
{
```

The scores attribute is created using the **MFnNumericAttribute** class.

```
MFnNumericAttribute numFn;
scoresAttr = numFn.create( "scores", "scrs", MFnNumericData::kLong );
```

The setArray function is called to indicate that the attribute is an array.

```
numFn.setArray( true );

addAttribute( scoresAttr );

return MS::kSuccess;
}
```

In order to see what elements an array contains, a utility function, printArray, is defined as follows. It prints out the contents of the array to the Script Editor.

```
void printArray( MPlug &arrayPlug )
{
MString txt( "\nArray..." );

unsigned int nElems = arrayPlug.numElements();
unsigned int i=0;
for( i=0; i < nElems; i++ )
    {
    MPlug elemPlg = arrayPlug.elementByPhysicalIndex(i);

    txt += "\nElement #";
    txt += (int)i;
    txt += ": ";
    double value = 0.0;
    elemPlg.getValue( value );
    txt += value;
    txt += " (";
    txt += (int)elemPlg.logicalIndex();
    txt += ")";
    }

MGlobal::displayInfo( txt );
}
```

The significant portions of the function are now covered. The first step in the function is to determine the number of elements in the array. This is done using the **MPlug**'s numElements function.

```
unsigned int nElems = arrayPlug.numElements();
```

The elements are iterated over.

```
for( i=0; i < nElems; i++ )
    {
```

A plug is created for each element by calling the **MPlug**'s `elementByPhysicalIndex` function. This returns a plug to the *i*th array element.

```
MPlug elemPlg = arrayPlug.elementByPhysicalIndex(i);
```

The value of the element plug is retrieved using the **MPlug**'s `getValue` function.

```
double value = 0.0;
elemPlg.getValue( value );
```

In addition to displaying the element's value, the logical index is also displayed. Logical indices are explained shortly. The logical index of the plug is given by the **MPlug**'s `logicalIndex` function.

```
txt += (int)elemPlg.logicalIndex();
```

To demonstrate how elements are added to an array, the following statements are used. They create a **MyNode** node and add elements to the **scores** array.

```
MObject myNodeObj = dgMod.createNode( MyNode::id );
MFnDependencyNode depFn( myNodeObj );
```

A plug to the **scores** attribute is created using the `findPlug` function.

```
MPlug scoresPlg = depFn.findPlug( "scores" );
```

Elements in an array have both a physical and a logical index. The physical index of an element in the array is its index into the actual array. The physical indices of the array range from `0` to `numElements()-1`. When an element is deleted from the array, the physical indices of some of the elements could change. The logical index to an element, on the other hand, never changes. The logical index is a means of giving an element an absolute, unchanging index, irrespective of any additions or deletions to

the physical array. When you refer to an array element in MEL, you use its logical index. There is no way in MEL to get an element's physical index.

The need for physical and logical indices is that connections between attributes are made based on their logical indices. Since these indices don't change, you can be sure that deleting another element in the array won't change an element's logical index. This is unlike their physical index, which could change.

The following examples help further clarify the difference between the two types of indexing. The `scorePlg` plug is set to the element at logical index 0. This is done by using the `elementByLogicalIndex` function.

```
MPlug scorePlg;
scorePlg = scoresPlg.elementByLogicalIndex(0);
```

Its value is then set to 46.

```
scorePlg.setValue( 46 );
```

The contents of the array are then printed out.

```
printArray( scoresPlg );
```

The results are as follows:

```
Array...
Element #0: 46 (0)
```

An element has been added to the array. The `elementByLogicalIndex` is used to create an element at the given index. If an element already exists at the given logical index, the existing element is accessed. The first element has a physical index of 0 and a logical index of 0.

Next, the element at logical index 2 is referenced. Since it doesn't exist yet, it is created.

```
scorePlg = scoresPlg.elementByLogicalIndex(2);
```

Its value is set to 12.

```
scorePlg.setValue( 12 );
```

The array is once again printed out.

```
printArray( scoresPlg );
```

The results are as follows:

```
Array...
Element #0: 46 (0)
Element #1: 12 (2)
```

The physical array now contains two elements. As you would expect, their physical indices are 0 and 1. However the logical indices of the elements are 0 and 2. Since the second element was accessed using an index of 2 to the `elementByLogicalIndex` function, it is assigned the logical index of 2. Notice that there is no logical index 1. Since it hasn't been referenced yet, it hasn't been created. Since the logical indices of the array can have "missing" indices, the array can be considered to be a sparse array. While the logical indices are sparse, the physical indices are contiguous.

In the next step, the element at logical index 1 is referenced.

```
scorePlg = scoresPlg.elementByLogicalIndex(1);
```

The element's value is set to 93.

```
scorePlg.setValue( 93 );
```

The array contents are then printed out.

```
printArray( scoresPlg );
```

The results are as follows:

```
Array...
Element #0: 46 (0)
Element #1: 93 (1)
Element #2: 12 (2)
```

The element is inserted into the array at physical index 1. The logical index it uses is the one passed into the `elementByLogicalIndex` function. Notice that the

element with logical index 2 (value of 12) now has a different physical index. It was index 1 before, and now it is index 2. This shows once again that the physical index of an element can change, while its logical index never will.

To further emphasize that the logical index doesn't necessarily correspond to the physical index, another element is added. This time it uses a logical index that is much larger than any of the others. The element is referenced using logical index 25.

```
scorePlg = scoresPlg.elementByLogicalIndex(25);
```

Its value is set to 57.

```
scorePlg.setValue( 57 );
```

The array is printed.

```
printArray( scoresPlg );
```

The results are as follows:

```
Array...
Element #0: 46 (0)
Element #1: 93 (1)
Element #2: 12 (2)
Element #3: 57 (25)
```

The array now has four elements. The physical indices range from `0` to `numElements()-1`. The logical indices are exactly those chosen. There are currently no elements at logical indices 3 to 24. Since logical indices are unchanging, they are the safest way to refer to array elements.

Elements are retrieved using the logical index with the **MPlug**'s `elementByLogicalIndex` function. As mentioned earlier, if an element exists at a given logical index, it is retrieved. If an element doesn't exist, it is created. In the following example, the element at logical index 1 is retrieved:

```
int value;
scorePlg = scoresPlg.elementByLogicalIndex(1);
scorePlg.getValue( value ); // Value of 93
```

To get a list of all the logical indices of an array, use the **MPlug**'s `getExistingArrayAttributeIndices` function. This returns an array of logical indices for all elements that exist (that are connected or that have their value set).

```
MIntArray logIndices;
scoresPlg.getExistingArrayAttributeIndices( logIndices );
```

The `logIndices` array now contains the values [0, 1, 2, 25]. What happens if you attempt to retrieve a value for an element that doesn't exist? In the following code, the value of the element at logical index 10 is retrieved:

```
MStatus stat;
scorePlg = scoresPlg.elementByLogicalIndex( 10 );
scorePlg.getValue( value );
MGlobal::displayInfo( MString("value: ") + value );
```

The result is as follows:

```
// value: 0
```

When an element doesn't exist at a given logical index, the default value for the element is returned. This indicates that *all* logical indices are valid. When you access an element that doesn't exist, the default value is returned. If an element does exist, its current value is returned.

Given an element plug, it may be necessary to determine its parent array plug. This is done using the **MPlug**'s `array` function.

```
MPlug arrayPlg = scorePlg.array();
```

The `arrayPlg` plug now refers to the **scores** array attribute.

4.5.7 Datablocks

Datablocks are where the actual node data is stored. The data isn't stored within the node but inside one or more datablocks. It isn't necessary to understand their inner workings and how they are networked together. Instead the **MDataBlock**, **MDataHandle**, and **MArrayDataHandle** classes are used to retrieve and set node data.

The **MDataBlock** provides a convenient means of grouping all the node's data together. To access a single node attribute, use the **MDataBlock**'s `inputValue` or `outputValue` functions. If the attribute is being used as input to a computation, then use the `inputValue` function. If it is being used as output in a computation, then use the `outputValue` function. Both these functions return an **MDataHandle** instance that is then used to access the data. Similarly, when you want to access an array attribute, the **MDataBlock**'s `inputArrayValue` and `outputArrayValue` functions should be used. These both return an **MArrayDataHandle** that allows you to access the individual elements of the array.

The `input...Value` functions return a handle that can be used only for reading the data. The data is read-only. You can't write to the data using the handle they return. Similarly, the `output...Value` functions return a handle that is used for writing the data. The data is write-only. You can't read the data using the handle they return. Since attributes are classified into input (read-only) and output (write-only), these restrictions typically aren't a problem.

In this example, the **MyNode** class has two attributes: **scores**, an array of integers, and **average**, which is the average of the scores. The `compute` function therefore needs to access both an array attribute, **scores**, as well as a single attribute, **average**.

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

if( plug == avgAttr )
    {
    MArrayDataHandle scoresHnd = data.inputArrayValue( scoresAttr );
    MDataHandle avgHnd = data.outputValue( avgAttr );
    const unsigned int nElems = scoresHnd.elementCount();
    double sum = 0.0;
    unsigned int i;
    for( i=0; i < nElems; i++ )
         {
         scoresHnd.jumpToElement( i );
         MDataHandle elemHnd = scoresHnd.inputValue();
         sum += elemHnd.asInt();
         }
    sum /= nElems;
    avgHnd.set( sum );
    data.setClean( plug );
    }
```

```
else
    stat  = MS::kUnknownParameter;

return stat;
}
```

Inside the `compute` function, the **scores** data is being used as input in the calculation of the **average** data. The **scores** data is accessed using the **MDataBlock**'s `inputArrayValue` function. This returns an **MArrayDataHandle** that can then be used to access the individual elements of the **scores** array.

```
MArrayDataHandle scoresHnd = data.inputArrayValue( scoresAttr );
```

A handle to the average data is then created by calling the **MDataBlock**'s `outputValue` function.

```
MDataHandle avgHnd = data.outputValue( avgAttr );
```

The number of elements in the **scores** array is determined using the **MArrayDataHandle**'s `elementCount` function.

```
const unsigned int nElems = scoresHnd.elementCount();
```

The individual elements are then iterated over.

```
double sum = 0.0;
unsigned int i;
for( i=0; i < nElems; i++ )
    {
```

To access a particular element, the **MArrayDataHandle**'s `jumpToElement` function is used. It takes the index of the element as its argument.

```
scoresHnd.jumpToElement( i );
```

A handle to the individual element is then created.

```
MDataHandle elemHnd = scoresHnd.inputValue();
```

The value of the element is retrieved and added to the running total.

```
sum += eleHnd.asInt();
}
```

The average is then calculated, and the data set. Finally the plug is marked as being updated.

```
sum /= nElems;
avgHnd.set( sum );
data.setClean( plug );
```

The individual children of a compound attribute can be retrieved using the **MDataHandle**'s `child` function. Since an attribute can contain arbitrary arrays of attributes, and each of these attributes can themselves be arrays, the **MArrayDataHandle** can also return a handle to its subarrays by using the **MArrayDataHandle**'s `inputArrayHandle` or `outputArrayHandle` functions.

The **MDataHandle** and **MArrayDataHandle** classes are lightweight so they can be easily created and deleted as necessary. They simply provide a convenient interface to the underlying data that resides in the various datablocks.

4.5.8 Node Design Guidelines

Having covered how to create custom nodes and integrate them into the DG, some general guidelines for designing your own nodes are now discussed. The following are the cardinal rules of node design:

- Keep the node simple.
- Nodes should never be aware of their environment or context.
- A node should never use data that is external to itself.

SIMPLICITY

As you are designing and implementing nodes, it is quite easy to begin adding more and more features to a given node. As a result, the node can become feature bloated. Recalling that nodes are really building blocks used in a larger system, by keeping them simple and singular in purpose, you can reuse their functionality more easily. If a single node performs two disjointed tasks, it should be split into two separate nodes. This reduces the complexity of your nodes and maximizes their reusability.

Another important consideration is that since a node is simply a C++ class, it is easy to make the mistake of giving the class more features and functionality than is

really necessary. DG nodes fulfill a very precise and specific purpose in Maya. They take input data and produce some outputs. They should never be designed to do anything more than that. So keep in mind that your node is not a general C++ class capable of anything but a specific piece in a particular system: Maya's Dependency Graph.

CONTEXT

A node should never know where it is located in the DG. In fact, the node should never know that it is even used in the DG. A node should know only its input and output attributes and how to generate them when requested.

Unfortunately, it is possible for a node to begin looking outside of itself. The C++ API provides a variety of functions that let you follow the connections from your node to other nodes. These functions let you trace connections upstream and downstream. A node could therefore determine its place in the larger DG network. Doing this, however, is extremely dangerous.

If you design your node so that it knows, for instance, that it is part of an object's construction history, it makes certain assumptions. It assumes that it is being used in a particular configuration of nodes. This is extremely dangerous, since the user can create an instance of your node and use it somewhere else in the DG network. You node will most likely produce incorrect results or may even crash, because it assumes that certain conditions exist; in this case they won't.

A node should never check whether its attributes are connected to other nodes. A well-designed node will never have to do this. Maya handles all node connections without you, the node designer, having to be concerned. Attributes are presented in the C++ API in the same fashion, irrespective of whether they are connected or not. A common mistake is to create nodes that know they are connected to one another. The nodes then attempt to communicate or share data between themselves using pointers. This is effectively trying to circumvent Maya's data passing mechanisms and is extremely dangerous. If one of the nodes is deleted or disconnected, it could cause the other node to dereference a now invalid pointer. This can result in Maya crashing.

A node should never ascertain at which particular time it is being evaluated. If you design a node to cache information based on particular times, there is a very good chance that your node won't work correctly in certain contexts. For instance, it is possible to evaluate a node's attributes at any given time. In fact, a node can be evaluated at multiple times. If you design your node based on the assumption that it being called at a given time, it won't work when other times are used.

As such, making no assumptions about when and how the node is being evaluated ensures that your nodes operate correctly.

LOCALITY

A node must never, under any circumstances, look outside of itself. This is especially important when the node is computing its outputs. A node should never use external data in its computations. It should look only at its input attributes. If there is data that a node needs in order to do its computation that isn't in its input attributes, it should be added as an input attribute. You are free to perform the computation in any way you like.

All data that flows into a node should be done strictly through the DG. Data should not come from external sources. Using external data inside a node breaks its ability to operate without context. The DG will most likely not evaluate to the correct result.

This cardinal rule is so important that it can't be emphasized enough.

A node must never look beyond its own attributes!

NETWORKS OF NODES

Since a node should have a very myopic view of the world, how do you design a system in which many interconnected nodes are needed? Since nodes can't know anything about their context, it should be the task of a custom command to create and maintain the network of nodes. The command should have all the knowledge of the context in which the nodes are created and connected. So the nodes should remain simple and singular in their purpose. The commands should be the ones aware of the context in which the nodes are used.

4.6 LOCATORS

A locator provides users with a 3D visual aid that they can move and manipulate. It is drawn in the Maya viewport but won't appear in the final rendered image. Locators can be used to create a 3D handle that users can manipulate to control some other process. For example, locators in the shape of a footprint could be used to define the steps of a character. The user would simply move the footprint locators to change where the character walked.

Maya comes with a variety of measuring tools that are implemented as locators. The **Distance Tool** creates two locators and then displays the distance between them. Linking each of the end-point locators to separate objects gives you an immediate display of the distance between them, even when they are animated. Additional measuring tools include the **Parameter Tool** and the **Arc Length Tool**, which display the parametric value of a point and the distance along the curve, respectively.

Creating a custom locator is quite simple since you need only to overwrite a few member functions in order to add your new functionality. The main member functions needing changes are `draw`, `isBounded`, and `boundingBox`. If you want to draw the locator in a custom color, then the `color` and `colorRGB` functions should also be implemented. Inside the `draw` function, you are free to draw the locator in any way you like. Almost all OpenGL functions are available for drawing. Last the bounding box functions, `isBounded` and `boundingBox`, are important if you want your locator to work correctly with the various Maya selection and view zooming tools.

4.6.1 BasicLocator Plugin

This example plugin demonstrates how to create a basic locator. This locator can draw itself in a variety of ways. It also allows the user to stretch its width and length. An example of the basic locator is shown in Figure 4.26.

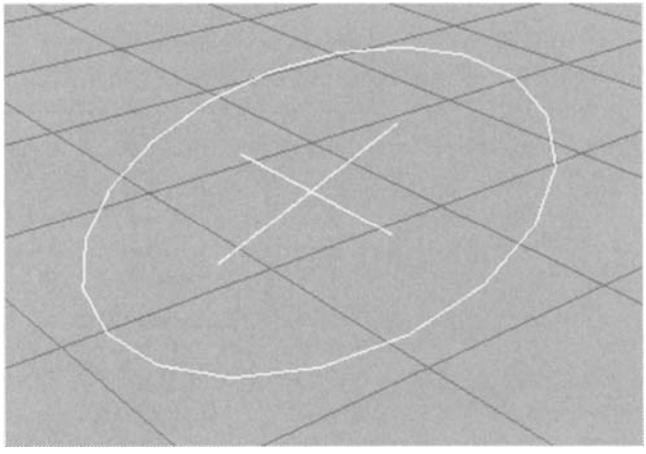


FIGURE 4.26 Basic locator

1. Open the **BasicLocator** workspace.
2. Compile it, and load the resulting `BasicLocator.mll` plugin in Maya.
3. Select **File | New Scene**.
4. Execute the following in the **Script Editor**:

```
createNode basicLocator;
```

   A **basicLocator** node is created and displayed. Since the locator doesn't need to be connected to other nodes or need any particular initialization, as was the case in previous examples, the `createNode` command is a fast and easy method of creating a node instance.

5. Zoom into the locator or press **a** so you can see it more clearly.
6. Open the **Attribute Editor**.

   The keyable attributes for the **basicLocator1** node are displayed.

7. Set the **X Width** to **2**.
8. Set the **Disp Type** to **1**.

   The locator is drawn as a lozenge.

9. Set the **Disp Type** to **2**.

   The locator is drawn as an ellipse, since the **X Width** is twice the **Z Width**. Setting them both to the same size results in a circle being drawn. Since a locator has a parent transform node, like all DAG nodes, you can move, rotate, and scale it.

10. Experiment by moving and rotating the node.

Plugin: BasicLocator
File: BasicLocator.h

A custom locator node is created by first deriving from the **MPxLocatorNode** class. In addition to the usual `creator` and `initialize` functions, the class also implements the `draw`, `isBounded`, and `boundingBox` functions.

It has three attributes, **xWidth**, **zWidth**, and **dispType**, which define the scale of the locator's width, length, and how it will drawn, respectively.

```
class BasicLocator : public MPxLocatorNode
{
public:
    virtual void draw( M3dView & view, const MDagPath & path,
                       M3dView::DisplayStyle style,
                       M3dView::DisplayStatus status );

    virtual bool isBounded() const;
    virtual MBoundingBox boundingBox() const;

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

    static const MTypeId typeId;
    static const MString typeName;

    // Attributes
    static MObject xWidth;
    static MObject zWidth;
    static MObject dispType;

private:
    bool getCirclePoints( MPointArray &pts ) const;
};
```

Plugin: BasicLocator
File: BasicLocator.cpp

The node draws itself in one of three fashions: triangle, lozenge, or circle. Each of these shapes can be defined by a series of interconnected lines. Rather than define different drawing functions for each type, an array of points is created. When the display type is a triangle, only three points are output. When it is a lozenge, then four points are generated, and so on. An additional point that wraps around to the first is also included. The `getCirclePoints` function generates this array of points. It takes into account the **xWidth**, **zWidth**, and **dispType** attributes when calculating the final positions of the points.

```
const double M_2PI = M_PI * 2.0;

bool BasicLocator::getCirclePoints( MPointArray &pts ) const
{
MStatus stat;
MObject thisNode = thisMObject();
MFnDagNode dagFn( thisNode  );

MPlug xWidthPlug = dagFn.findPlug( xWidth, &stat );
float xWidthValue;
xWidthPlug.getValue( xWidthValue );

MPlug zWidthPlug = dagFn.findPlug( zWidth, &stat );
float zWidthValue;
zWidthPlug.getValue( zWidthValue );

MPlug typePlug = dagFn.findPlug( dispType, &stat );
short typeValue;
typePlug.getValue( typeValue );

unsigned int nCirclePts;

switch( typeValue )
    {
    case 0:
        nCirclePts = 4;
        break;
    case 1:
        nCirclePts = 5;
        break;
    default:
        nCirclePts = 20;
        break;
    }

pts.clear();
pts.setSizeIncrement( nCirclePts );
```

```
MPoint pt;
pt.y = 0.0;

const double angleIncr = M_2PI / (nCirclePts - 1);
double angle = 0.0;
unsigned int i=0;
for( ; i < nCirclePts; i++, angle+=angleIncr )
    {
    pt.x = xWidthValue * cos( angle );
    pt.z = zWidthValue * sin( angle );
    pts.append( pt );
    }

return true;
}
```

Often the most complex part of a locator is its `draw` function. The `draw` function serves a dual purpose in Maya. It is called to draw the node in the current viewport. It is also used by Maya to determine whether the node is selected. When Maya is determining whether an object is the selection region or where the user has clicked, the object's `draw` function is called. Maya then uses the result of the drawing to determine if the object is selected. You don't have to be concerned about the second use of the `draw` function, since Maya handles this aspect automatically for you.

The locator can be as simple or as complex as you like. In fact almost all the OpenGL function calls are available to you during drawing. There are some restrictions, however. It is important that the `draw` function leave OpenGL in exactly the same state it was before the function is called. The OpenGL `glPushAttrib` function can make this task easier. It is used for pushing and popping the current graphics state.

When a locator's `draw` function is called, the current transformation has already been applied. As such, the node doesn't need to be concerned with positioning, rotation, or scaling itself. This is all handled automatically. The node simply draws itself in local object space. Also the current color is set automatically based on the current state of the node (selected, live, dormant, and so on). It is possible for the node to define a custom color for any of these states by implementing the `color` and `colorRGB` functions.

The `draw` function is called with several arguments. The first is the current Maya viewport in which the locator will be drawn. This is given as an **M3dView** class. The complete DAG path, **MDagPath**, to this locator node is also provided. The **M3dView ::DisplayStyle** defines the drawing mode in which the locator is to be drawn. The styles include bounding box, shaded, flat shaded, and so on. The **M3dView:: DisplayStatus** defines the current state of the node in the viewport. This status can include active, live, dormant, invisible, and so on.

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

Before any OpenGL drawing functions are called, the `beginGL` function should be called and the current OpenGL state pushed.

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

The vertices of the current display shape are then generated.

```
MPointArray pts;
getCirclePoints( pts );
```

The series of vertices are then drawn as line segments.

```
glBegin(GL_LINE_STRIP);
for( unsigned int i=0; i < pts.length(); i++ )
     glVertex3f( float(pts[i].x), float(pts[i].y), float(pts[i].z) );
glEnd();
```

A small cross is also drawn at the center of the locator.

```
glBegin(GL_LINES);
      glVertex3f( -0.5f, 0.0f, 0.0f );
      glVertex3f( 0.5f, 0.0f, 0.0f );

      glVertex3f( 0.0f, 0.0f, -0.5f );
      glVertex3f( 0.0f, 0.0f, 0.5f );
glEnd();
```

Now that drawing is finished, the current OpenGL state can be popped to restore the previous one. The `endGL` function should also be called to indicate that drawing is now finished.

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

The `isBounded` function is called when Maya needs to determine if the node knows its own bounding extents. The `boundingBox` function is called to retrieve the actual extents of the locator shape. It is highly recommended to implement these bounding functions. Without them, Maya has difficulty determining the exact size of the locator, so the Frame All and Frame Selection operations will result in incorrect zooming.

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

Since the vertices of the locator are known to be the widest and longest parts of the node, finding their bounding box equates to finding the bounding box for the node. This process is greatly simplified by the `getCirclePoint` function, which provides a list of all the vertices in the locator.

```
MBoundingBox BasicLocator::boundingBox() const
{
MPointArray pts;
getCirclePoints( pts );
```

The `bbox` default constructor initializes it to an empty volume.

```
MBoundingBox bbox;
```

Points are added using the **MBoundingBox**'s `expand` function. This increases the bounding box, if necessary, to include the given point.

```
for( unsigned int i=0; i < pts.length(); i++ )
     bbox.expand( pts[i] );
return bbox;
}
```

The `initialize` function creates the three custom attributes and adds them to the node.

```
MStatus BasicLocator::initialize()
{
MFnUnitAttribute unitFn;
MFnNumericAttribute numFn;
MStatus stat;
```

Both the **xWidth** and **zWidth** attributes are distances and so must be created using the **MFnUnitAttribute** rather than a simple `double`. Their minimum and default values are also specified.

```
xWidth = unitFn.create( "xWidth", "xw", MFnUnitAttribute::kDistance );
unitFn.setDefault( MDistance(1.0, MDistance::uiUnit()) );
unitFn.setMin( MDistance(0.0, MDistance::uiUnit()) );
unitFn.setKeyable( true );
stat = addAttribute( xWidth );
if(!stat)
  {
  stat.perror( "Unable to add \"xWidth\" attribute" );
  return stat;
  }

zWidth = unitFn.create( "zWidth", "zw", MFnUnitAttribute::kDistance );
unitFn.setDefault( MDistance(1.0, MDistance::uiUnit()) );
unitFn.setMin( MDistance(0.0, MDistance::uiUnit()) );
unitFn.setKeyable( true );
stat = addAttribute( zWidth );
if(!stat)
  {
  stat.perror( "Unable to add \"zWidth\" attribute" );
  return stat;
  }
```

The **dispType** attribute holds a `short` indicating the current drawing style. It is set up to use a minimum value of 0 and a maximum of 2.

```
dispType = numFn.create( "dispType", "att", MFnNumericData::kShort );
numFn.setDefault( 0);
numFn.setMin( 0 );
numFn.setMax( 2 );
numFn.setKeyable( true );
stat = addAttribute( dispType );
if(!stat)
  {
  stat.perror( "Unable to add \"dispType\" attribute" );
  return stat;
  }

return MS::kSuccess;
}
```

Plugin: BasicLocator
File: PluginMain.cpp

The locator node is like any other node, so it needs to be registered during the initialization of the plugin. Registration of the node is the same as usual, with the exception that the node's type, `MPxNode::kLocatorNode`, needs to be given to the `registerNode` function. If the explicit node type wasn't used, then the default, `MPxNode::kDependNode`, is used.

```
...
stat = plugin.registerNode(
         BasicLocator::typeName,
         BasicLocator::typeId,
         &BasicLocator::creator,
         &BasicLocator::initialize,
         MPxNode::kLocatorNode );
...
```

The `deregisterNode` function is called, as usual, with the node's ID.

```
...
stat = plugin.deregisterNode( BasicLocator::typeId );
...
```

4.7 MANIPULATORS

Maya provides many different ways to adjust and change the attributes of a given node. The Attribute Editor is the most commonly used method for setting an attribute's value. For certain nodes there is a more visual means of achieving this same goal. Through the use of manipulators, the user can visually modify a given set of attributes. Depending on the type of attribute, this can be more intuitive than using the Attribute Editor or other method.

4.7.1 BasicLocator2 Plugin

To actually see a quick example of how manipulators work, the **BasicLocator2** plugin is compiled, then loaded. This plugin extends on the locator node created in the previous example to include manipulators for all its custom attributes: **xWidth**, **zWidth**, and **dispType**. The manipulators are shown in Figure 4.27.

1. Open the **BasicLocator2** workspace.
2. Compile it, and load the resulting `BasicLocator2.mll` plugin in Maya.
3. Select **File | New Scene**.
4. Execute the following in the **Script Editor**:

```
createNode basicLocator;
```

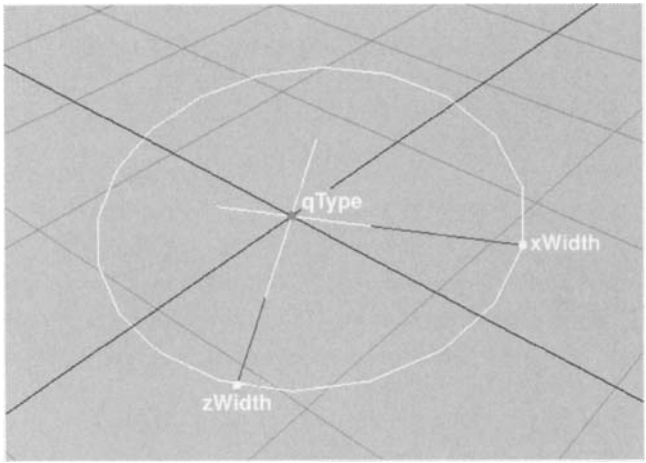


FIGURE 4.27 Basic locator with manipulators shown

A **basicLocator** node is created and displayed.

5. Select **View | Frame Selection** from the viewport panel, or press **f**.
6. Open the **Attribute Editor**.

   The editor is opened so you can see how the attribute's values are changing.
7. Select **Modify | Transformation Tools | Show Manipulator Tool**, or press **t**.

   The manipulators for the locator are displayed.

   If the manipulators aren't shown, ensure that the **basicLocator1** node is selected and not its parent transform node, **transform1**.
8. Click on the circle icon next to **Type**.

   The locator's **dispType** attribute is incremented.
9. Click on the circle icon again.

   The locator is now drawn as a circle.
10. Drag the dot next to the **X Width** text.

    The locator's **xWidth** attribute is updated. Notice in the Attribute Editor that the attribute value changes when you interact with the manipulator.

It is possible to define a manipulator node for any other Dependency Graph node. In the current example, the locator node, **basicLocator**, was defined, as well as its manipulator node, **basicLocatorManip**. The manipulator node is a special node that is created when the user selects a node and then requests its manipulator tool by choosing **Show Manipulator Tool**. Maya then creates an instance of the manipulator node. This node then goes about creating a series of base manipulators with which the user interacts. The manipulator node attaches these base manipulators to the node being edited so that a change in one of the manipulators, for instance dragging a point, results in a change to the node's attribute value. When the user exits the **Show Manipulator Tool**, the manipulator node is automatically deleted. The manipulator node can't be displayed in any of the Maya windows (**Hypergraph**, **Outliner**, and so on) or have any of its own attributes edited. It is, in essence, a temporary worker node that exists only while the node it is editing is selected. It isn't possible to create a manipulator node outside of the **Show Manipulator Tool** or user-defined context.

A manipulator node can be considered a "container" of base manipulators. A base manipulator is one of Maya's predefined interactive methods for changing an attribute. In the current example the **baseLocatorManip** manipulator contains three

base manipulators: distance manipulators for both the **xWidth** and the **zWidth** attributes and a state manipulator for the **dispType** attribute. Maya current supports ten different base manipulators:

- **FreePointManip**
- **DirectionManip**
- **DistanceManip**
- **PointOnCurveManip**
- **PointOnSurfaceManip**
- **DiscManip**
- **CircleSweepManip**
- **ToggleManip**
- **StateManip**
- **CurveSegmentManip**

The complete details of each base manipulator are documented in the Maya class reference documentation. It isn't possible to define your own base manipulators, so you instead build up a more complex manipulator by using a series of base manipulators. These base manipulators are referred to as *children* of the main manipulator node.

The general steps to creating a manipulator node include deriving a new class from **MPxManipContainer**. The node then adds a series of children base manipulators. These children are associated with a given attribute in the target node that the manipulator will operate on. It is possible to define a simple relationship between a child manipulator and its associated attribute or a more complex one. This relationship is defined through the use of conversion functions. These functions define how information is translated from the manipulator to the attribute and vice versa.

Plugin: BasicLocator2
File: BasicLocator.cpp

Before delving into the specifics of the manipulator node, the original locator node must be altered slightly to let Maya know that it now has an associated manipulator node. Maya keeps an internal table of all the nodes and their associated manipulator nodes. At the end of the `initialize` function, the static `addToManipConnectTable` function is called with the ID of the node that now has a manipulator.

```
MStatus BasicLocator::initialize()
{
...
MPxManipContainer::addToManipConnectTable( const_cast<MTypeId &>
                                           ( typeId ) );
...
}
```

This is the only change necessary to the **baseLocator** node in order to have it work with the manipulator.

Plugin: BasicLocator2
File: BasicLocatorManip.h

A manipulator node is derived from the **MPxManipContainer** class.

```
class BasicLocatorManip : public MPxManipContainer
{
public:
    virtual MStatus createChildren();
    virtual MStatus connectToDependNode(const MObject & node);

    virtual void draw( M3dView & view, const MDagPath & path,
                       M3dView::DisplayStyle style,
                       M3dView::DisplayStatus status);

    static void * creator();

    MManipData startPointCallback(unsigned index) const;
    MManipData sideDirectionCallback(unsigned index) const;
    MManipData backDirectionCallback(unsigned index) const;

    MVector nodeTranslation() const;
    MVector worldOffset(MVector vect) const;

    static const MTypeId typeId;
    static const MString typeName;

    MManipData centerPointCallback(unsigned index) const;
```

```
    // Paths to child manipulators
    MDagPath xWidthDagPath;
    MDagPath zWidthDagPath;
    MDagPath typeDagPath;

    // Object that the manipulator will be operating on
       MObject targetObj;
};
```

In addition to defining a variety of member functions, which are covered shortly, the node defines DAG paths to the base manipulators it uses. These are the child manipulators with which the user interacts.

```
MDagPath xWidthDagPath;
MDagPath zWidthDagPath;
MDagPath typeDagPath;
```

The manipulator node also needs to a keep a record of which node it will be manipulating. This is the **baseLocator** node that was currently selected when the user activated the **Show Manipulator Tool**. The node being manipulated is referenced using the `targetObj` member. Notice that it is an **MObject** rather than an **MDagPath**.

```
MObject targetObj;
```

Manipulators can be applied to any dependency node, not just DAG nodes. It is therefore possible to use manipulators on any node that you define.

Plugin: BasicLocator2
File: BasicLocatorManip.cpp

Like all nodes, the **BasicLocatorManip** node must define a unique identifier. In this example, a temporary identifier is used.

```
const MTypeId BasicLocatorManip::typeId( 0x00338 );
```

It is *very* important to name the manipulator node based on the node it operates on. The name of the manipulate node must be the node name followed by **Manip**, so this manipulator node must be named **baseLocatorManip**. It is very important

that the node name match exactly or the manipulator won't be associated correctly. Note that node names are case-sensitive.

```
const MString BasicLocatorManip::typeName( "basicLocatorManip" );
```

It is interesting to note that the **MPxManipContainer** contains its own static `initialize` function. In previous examples, the class defined its own. Since this manipulator doesn't do any special initialization beyond what is done by the base class, the manipulator node doesn't need to override this. Note, however, that if you do define your own `initialize` function, you should be sure to call back to the base class's `initialize` function. The following code shows an example of how you would do this:

```
MStatus BasicLocatorManip::initialize()
{
MStatus stat;
stat = MPxManipContainer::initialize();
... // do any extra initialization here
return stat;
}
```

Since the manipulator node is really just a container for base manipulators, the `createChildren` function is very important. It is used to create and add the base manipulators to the node.

```
MStatus BasicLocatorManip::createChildren()
{
MStatus stat = MStatus::kSuccess;
```

The `addDistanceManip` function is called to create a new distance manipulator node. The DAG path, `xWidthDagPath`, is used to store the path to the newly created node.

```
xWidthDagPath = addDistanceManip( "xWidthManip", "xW");
```

The **MFnDistanceManip** function set is attached to the distance manipulator node so that you can set its various properties.

```
MFnDistanceManip xWidthFn( xWidthDagPath );
```

The distance manipulator is drawn as a start point and end point in a particular direction. The user moves the end point to set the distance. For the **xWidth** manipulator, the start point should begin at the center of the node, and the end point should run along the x-axis.

```
xWidthFn.setStartPoint( MVector(0.0, 0.0, 0.0) );
xWidthFn.setDirection( MVector(1.0, 0.0, 0.0) );
```

Another distance manipulator node is created, this time for the **zWidth** attribute. It also has its starting point at the center of the node. Its end point, however, runs along the z-axis.

```
zWidthDagPath = addDistanceManip( "zWidthManip", "zW");
MFnDistanceManip zWidthFn( zWidthDagPath );
zWidthFn.setStartPoint( MVector(0.0, 0.0, 0.0) );
zWidthFn.setDirection( MVector(0.0, 0.0, 1.0) );
```

A state manipulator is created to interact with the **dispType** attribute. It is initialized to have a maximum of three states, which is the same as the number of states for the **dispType** attribute.

```
typeDagPath = addStateManip( "typeManip", "tM");
MFnStateManip typeFn( typeDagPath );
typeFn.setMaxStates( 3 );
```

All the base manipulators have now been created. Notice that no association has been created between a given base manipulator and the attribute of the node it will affect. The `connectToDependNode` function is used for this purpose. When a node in the scene is selected and its manipulator is requested, usually through the **Show Manipulator Tool**, the `connectToDependNode` function is called with the selected node. It connects the base manipulators to the node's plugs. It also sets up any callbacks for placing and displaying the individual manipulators.

```
MStatus BasicLocatorManip::connectToDependNode( const MObject &node )
{
```

The `targetObj` is set to the node that is being manipulated.

```
targetObj = node;
MFnDependencyNode nodeFn(node);
```

An **MFnDistanceManip** function set is attached to the distance manipulator. A plug is created to the node's **xWidth** attribute.

```
MFnDistanceManip xWidthFn( xWidthDagPath );
MPlug xWidthPlug = nodeFn.findPlug( "xWidth", &stat );
```

The distance manipulator is associated with the **xWidth** plug. Any changes to the distance manipulator now affect the value of the **xWidth** attribute and vice versa.

```
xWidthFn.connectToDistancePlug( xWidthPlug );
```

When a manipulator is drawn, it is done so relative to the world origin. If the node is moved, the manipulator needs to be notified of this so that it can draw itself relative to the node's new location. This notification process is set up by using a callback function. This function will be called any time the node moves. It can then determine the exact location of the center of the node. This center is where the distance manipulator's start point will be drawn.

In order to do this setup, a plug-to-manipulator conversion callback is needed. This callback converts the plug value into its associated manipulator value.

```
addPlugToManipConversionCallback( xWidthFn.startPointIndex(),
                  (plugToManipConversionCallback)centerPointCallback );
```

Similarly, the distance manipulator's direction should follow the node's x-axis direction. Another plug-to-manipulator conversion callback is set up that provides the exact direction of the node's x-axis.

```
addPlugToManipConversionCallback( xWidthFn.directionIndex(),
                  (plugToManipConversionCallback)sideDirectionCallback );
```

These same series of steps are applied to the **zWidth** plug. The start point of its distance manipulator is set to the center of the node, while its direction is set to the node's z-axis.

```
MFnDistanceManip zWidthFn( zWidthDagPath );
MPlug zWidthPlug = nodeFn.findPlug( "zWidth" );
zWidthFn.connectToDistancePlug( zWidthPlug );

addPlugToManipConversionCallback( zWidthFn.startPointIndex(),
                  (plugToManipConversionCallback)centerPointCallback );

addPlugToManipConversionCallback( zWidthFn.directionIndex(),
                  (plugToManipConversionCallback)backDirectionCallback );
```

The state manipulator is associated with the node's **dispType** plug. The position of the state manipulator is defined by the center of the node.

```
MFnStateManip typeFn( typeDagPath );
MPlug typePlug = nodeFn.findPlug( "dispType" );
typeFn.connectToStatePlug( typePlug );

addPlugToManipConversionCallback( typeFn.positionIndex(),
                  (plugToManipConversionCallback)centerPointCallback );
```

It is extremely important that at the end of the function the `finishAddingMaps` function be called. It can be called only once.

```
finishAddingManips();
```

Somewhere following the `finishAddingMaps` function call, there must be a call to the **MPxManipContainer**'s `connectToDependNode` function. This lets the base class add any manipulator associations that it needs.

```
MPxManipContainer::connectToDependNode(node);

return MS::kSuccess;
}
```

Unless you want to add some custom drawing, there is no need to add your own `draw` function. Without it the base manipulators will still draw themselves. Include your own `draw` function when you want to add some additional drawing to the base

manipulators or some other custom display information. In this example, some text labels are displayed next to the base manipulators in order for the user to more easily identify them.

Like the locator's `draw` function, the manipulator node's `draw` function takes an **M3dView** that contains the current view. The **MDagPath** parameter is a path to the manipulator node being drawn. The **M3dView::DisplayStyle** and **M3dView::DisplayStatus** define the appearance and mode the node should be drawn in.

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

Before any drawing is done, the base class's `draw` function must be called. This performs any drawing needed by the base class, which most often includes the drawing of the base manipulators.

```
MPxManipContainer::draw(view, path, style, status);
```

The values of the node's attributes are then retrieved.

```
MFnDependencyNode nodeFn( targetObj );
MPlug xWidthPlug = nodeFn.findPlug( "xWidth" );
float xWidth;
xWidthPlug.getValue( xWidth );

MPlug zWidthPlug = nodeFn.findPlug( "zWidth" );
float zWidth;
zWidthPlug.getValue( zWidth );
```

Preparations are made for the OpenGL drawing.

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

The text for the manipulator labeling is set up.

```
char str[100];
MVector TextVector;
MString distanceText;

Set the text for the manipulator labeling.

strcpy(str, "XWidth");
distanceText = str;
```

The next step is to determine where the label should be drawn. This will be the vector from which the label should be drawn in world coordinates if the node hadn't been moved. This is then offset by the vector to the node's center.

```
MVector xWidthTrans = nodeTranslation();
TextVector = xWidthTrans;
TextVector += worldOffset( MVector(xWidth , 0, 0) );
```

The label is drawn at the calculated position.

```
view.drawText(distanceText, TextVector, M3dView::kLeft);
```

The text label and position are calculated for the **zWidth** and **dispType** attributes. Their labels are also drawn.

```
strcpy(str, "ZWidth");
distanceText = str;
MVector zWidthTrans = nodeTranslation();
TextVector = zWidthTrans;
TextVector += worldOffset( MVector( 0, 0, zWidth ) );
view.drawText(distanceText, TextVector, M3dView::kLeft);

strcpy(str, "Type");
distanceText = str;
TextVector = nodeTranslation();
TextVector += worldOffset( MVector( 0, 0.1, 0 ) );
view.drawText( distanceText, TextVector, M3dView::kLeft );
```

The OpenGL drawing is now finished.

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

The remainder of the class functions are those responsible for determining various positions around the node. They are the plug-to-manipulator conversion callback functions. The `centerPointCallback` function returns the current center position of the node in world coordinates.

```
MManipData BasicLocatorManip::centerPointCallback(unsigned index) const
{
```

A numeric data object is needed to hold the position. This position is stored as three doubles: `k3Double`.

```
MFnNumericData numData;
MObject numDataObj = numData.create( MFnNumericData::k3Double );
```

The `nodeTranslation` function returns the offset of the node's center from the world origin.

```
MVector vec = nodeTranslation();
```

The position is set to this offset.

```
numData.setData( vec.x, vec.y, vec.z );
```

All the callback functions return an **MManipData** object. This object is designed to hold all the different data types that a manipulator could possibly alter.

```
return MManipData( numDataObj );
}
```

The `sideDirectionCallback` function returns the world space position of the vector (1, 0, 0) relative to the node.

```
MManipData BasicLocatorManip::sideDirectionCallback( unsigned index )
    const
{
MFnNumericData numData;
MObject numDataObj = numData.create(MFnNumericData::k3Double);
MVector vec = worldOffset( MVector(1, 0, 0) );
numData.setData( vec.x, vec.y, vec.z );
return MManipData( numDataObj );
}
```

The `nodeTranslation` function is a utility function that returns the node's current center position in world coordinates.

```
MVector BasicLocatorManip::nodeTranslation() const
{
MFnDagNode dagFn( targetObj );
MDagPath path;
dagFn.getPath(path);
path.pop();

MFnTransform transformFn( path );
return transformFn.translation( MSpace::kWorld );
}
```

The `worldOffset` function is a utility function that returns a vector that is the offset between the given vector and its world position.

```
MVector BasicLocatorManip::worldOffset(MVector vect) const
{
MVector axis;
MFnDagNode transform( targetObj );
MDagPath path;
transform.getPath(path);
```

```
MVector pos( path.inclusiveMatrix() * MVector(0, 0, 0) );
axis = vect * path.inclusiveMatrix();
axis = axis - pos;
return axis;
}
```

4.8 DEFORMERS

Deformers are not only conceptually simple to understand, they are also easy to implement. A deformer is a node that take a series of points and moves them to new locations. The deformer can't create or remove points, it can only move them around in space. The deformer is free to use any method to deform the points. The method can be as simple as moving the points by a fixed distance or as complex as using a fluid-dynamics simulation to determine their new positions. A deformer can operate on a wide variety of geometry primitives. At the lowest level, a deformer can modify lattice points, control vertices, and polygonal vertices.

4.8.1 SwirlDeformer Plugin

An example deformer, **SwirlDeformer**, will be covered in detail. The plugin creates a swirl deformer node that deforms an object by rotating the vertices based on their distance from the swirl's center. You can also set the start and end distances between which the swirl takes effect. Figure 4.28 shows the result of applying the **SwirlDeformer** to an object.

1. Open the **SwirlDeformer** workspace.
2. Compile it, and load the resulting `SwirlDeformer.mll` plugin in Maya.
3. Open the **Swirl.ma** scene.
4. Select the **nurbsPlane1** object.
5. Execute the following in the **Script Editor**:

   ```
   deformer -type swirl;
   ```

   The NURBS plane is deformed by the swirl deformer. Notice that the swirl doesn't deform the entire object. This is because the swirl deformer end distance is shorter than the width of the object.

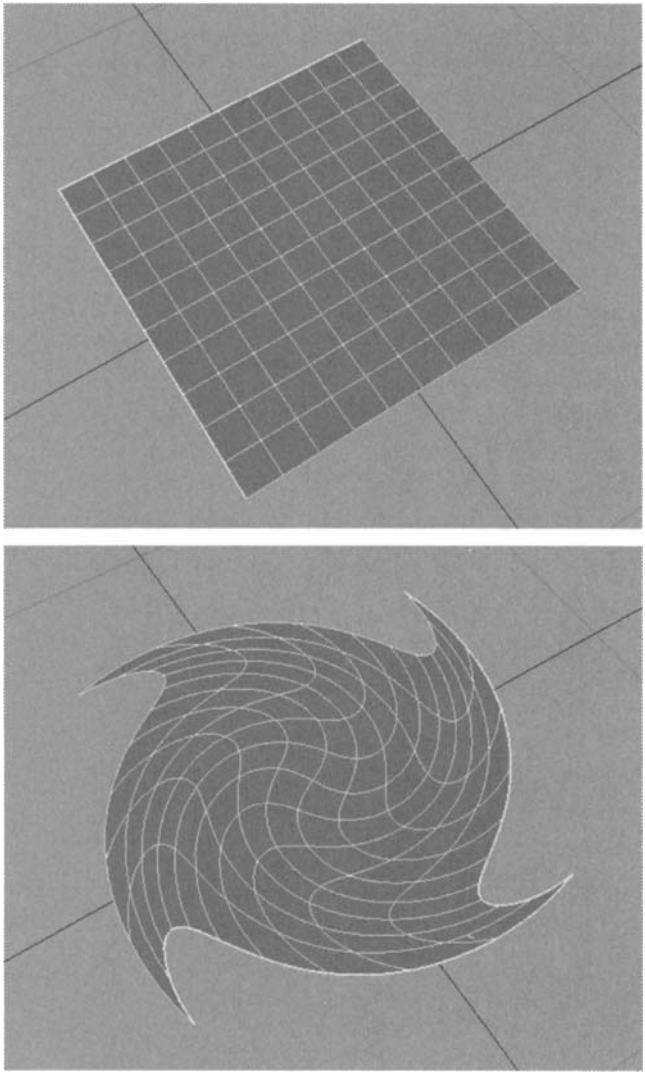


FIGURE 4.28 SwirlDeformer

6. Display the **Channel Box**.
7. Click on the **swirl1** item in the **INPUTS** section.

   The three main parameters are shown, **Envelope**, **Start Dist**, **End Dist**.
8. Set the **Envelope** attribute to **0.5**.

   The swirl is now less pronounced. You can interactively change the **Envelope** value to see the swirl increase and decrease in intensity.
9. Set the **Envelope** attribute back to **1.0**.
10. Set the **End Dist** attribute to **2.0**.

    The swirl now happens to a smaller area of the surface. Experiment by changing the **Start Dist** and **End Dist** to see how these attributes affect the swirl.
11. Set **Start Dist** to **0.0** and the **End Dist** to **3.0**.

    In addition to the general enveloping and distance parameters, you can define the influence of the deformer on individual vertices. This is done by changing the deformer's weighting.
12. Press **F8** to go to the **Control Vertex** level of the NURBS plane.
13. Select some vertices near the center of the plane.
14. Select **Window | General Editors | Component Editor...** from the main menu.
15. Click on the **Weighted Deformers** tab.
16. Drag down the **swirl1** column to select some vertices for editing.
17. Move the value slider at the bottom of the window.

    The weights of the selected vertices are changed, and as a result the effect of the swirl deformer is reduced or increased depending on their values.

When you create a custom deformer, Maya provides certain attributes automatically. The **envelope** attribute is common to all deformers. It defines the amount of deformation to apply.

Plugin: SwirlDeformer
File: SwirlDeformer.h

The **SwirlDeformer** class is created by deriving it from the **MPxDeformerNode** class.

```
class SwirlDeformer : public MPxDeformerNode
{
public:
    static  void *creator();
    static  MStatus initialize();
```

The major difference from other nodes is that a deformer contains a `deform` function and no `compute` function. The `compute` function still exists, but it is implemented in the base class. Unless you need some special computing, it is best just to implement the `deform` function and let the base class handle the `compute` function. The details of the `deform` function are explained in the next section.

```
virtual MStatus deform( MDataBlock &block,
                        MItGeometry &iter,
                        const MMatrix &mat,
                        unsigned int multiIndex );
```

Attributes for the start and end distance are defined. The **envelope** attribute is inherited from the **MPxDeformerNode** class.

```
private:
    // Attributes
    static MObject startDist;
    static MObject endDist;
};
```

Plugin: SwirlDeformer
File: SwirlDeformer.cpp

The major function to implement in a deformer is the `deform` function. It is the function that actually performs the deformation of the geometry. It is passed an **MDataBlock** that holds the datablock for the deformer node. This is the same datablock that you would normally get passed into the `compute` function. The second parameter is the **MitGeometry**, which is an iterator that lets you traverse all the points in the

geometric object. The iterator can iterate a variety of point types including control vertices, lattice points, mesh vertices, and so on. This iterator has already been initialized to traverse only the type of points the deformer should modify.

The third parameter is the local to world transformation **MMatrix**, `localToWorld`. When points are given to the deformer, they are in the local space of the geometry node. If you need to do your deformations in world space, then simply transform the points using this matrix. It is important, however, to return them to local space after the deformation by using the inverse of the `localToWorld` matrix.

The last parameter is the `geomIndex`. It is possible for a deformer to deform multiple geometry nodes as well as multiple components of a single geometry node. Maya keeps track of which section of geometry you are deforming by the `geomIndex`.

```
MStatus SwirlDeformer::deform( MDataBlock& block,
                               MItGeometry &iter,
                               const MMatrix &localToWorld,
                               unsigned int geomIndex )
{
MStatus stat;
```

The value of the **envelope** attribute is retrieved.

```
MDataHandle envData = block.inputValue( envelope );
float env = envData.asFloat();
```

If the **envelope** value is 0, then the deformer won't have any effect on the geometry, so the function can return immediately.

```
if( env == 0.0 )
    return MS::kSuccess;
```

The start and end distance attributes are then retrieved.

```
MDataHandle startDistHnd = block.inputValue( startDist );
double startDist = startDistHnd.asDouble();

MDataHandle endDistHnd = block.inputValue( endDist );
double endDist = endDistHnd.asDouble();
```

The deformer will use the **MItGeometry** geometry iterator to traverse all the points to deform.

```
for( iter.reset(); !iter.isDone(); iter.next() )
    {
```

For each point there can be an associated individual weight. This weight is obtained by using the **MPxDeformerNode**'s `weightValue` function.

```
weight = weightValue( block, geomIndex, iter.index() );
```

If the point has no weight, then the deformer won't affect it, so the point is skipped.

```
if( weight == 0.0f )
    continue;
```

The current point is retrieved.

```
pt = iter.position();
```

The perpendicular distance of the point from the y-axis is calculated.

```
dist = sqrt( pt.x * pt.x + pt.z * pt.z );
```

If the point is closer than the start distance or farther than the end distance, then the deformer won't affect the point.

```
if( dist < startDist || dist > endDist )
    continue;
```

The closer a point is to the center of the deformer, the greater its rotation. The `distFactor` is the result of calculating the strength of this rotation. It has a value between 0 and 1.

```
distFactor = 1 - ((dist - startDist) / (endDist - startDist));
```

The rotation angle for the point is then determined. It is a result of `distFactor` scaled by the **envelope** value and the point's particular `weight`. It is multiplied by the equivalent of a full rotation, expressed in radians.

```
ang = distFactor * M_PI * 2.0 * env * weight;
```

If there is no rotation, then the point is skipped.

```
if( ang == 0.0 )
    continue;
```

The point is rotated about the y-axis by the `ang` amount.

```
cosAng = cos( ang );
sinAng = sin( ang );
x = pt.x * cosAng - pt.z * sinAng;
pt.z = pt.x * sinAng + pt.z * cosAng;
pt.x = x;
```

The point is now updated to use its new deformed position.

```
        iter.setPosition( pt );
    }

return stat;
}
```

The deformer node has two new attributes, **startDist** and **endDist**, in addition to those that come from the base class. The `initialize` function defines these attributes and adds them to the node.

```
MStatus SwirlDeformer::initialize()
{
MFnUnitAttribute unitFn;
startDist = unitFn.create( "startDist", "sd",  MFnUnitAttribute::kDistance );
unitFn.setDefault( MDistance( 0.0, MDistance::uiUnit() ) );
unitFn.setMin( MDistance( 0.0, MDistance::uiUnit() ) );
unitFn.setKeyable( true );

endDist = unitFn.create( "endDist", "ed", MFnUnitAttribute::kDistance );
unitFn.setDefault( MDistance( 3.0, MDistance::uiUnit() ) );
unitFn.setMin( MDistance( 0.0, MDistance::uiUnit() ) );
unitFn.setKeyable( true );

addAttribute( startDist );
addAttribute( endDist );

attributeAffects( startDist, outputGeom );
attributeAffects( endDist, outputGeom );

return MS::kSuccess;
}
```

Plugin: SwirlDeformer
File: PluginMain.cpp

The only minor modification to the `initializePlugin` function is that when you register the deformer node, the type must be set to `MPxNode::kDeformerNode`.

```
...
stat = plugin.registerNode( SwirlDeformer::typeName,
                            SwirlDeformer::typeId,
                            SwirlDeformer::creator,
                            SwirlDeformer::initialize,
                            MPxNode::kDeformerNode );
...
```

The deregistration process in `uninitializePlugin` is the same as for other nodes.

4.8.2 Dependency Graph Changes

As well as understanding how to write a deformer, it is important to understand how Maya uses the deformer node in the DG. When you debug your deformer, it is important to understand where your deformer node sits in the grander scheme of general Maya deformation.

The MEL `deformer` command generated a number of Dependency Graph changes. Before the deformation is applied to the NURBS plane, the DG appears as in Figure 4.29. This is the standard construction history setup for a NURBS plane.

After the `deformer -type swirl` statement is executed, the NURBS plane appears as shown in Figure 4.30. Some parts have been removed for simplicity, but the major nodes and their connections are shown.

The **makeNurbPlane1** node now feeds into the **nurbsPlaneShapeOrg** node. This node is an exact duplicate of the NURBS shape, **nurbsPlaneShape1**, before the deformer was applied. This shape then feeds into a tweak node, **tweak1**. Whenever you first apply a deformer to an object, Maya creates a duplicate and connects it to a new tweak node. This allows you to go back and tweak the object. Any tweaks to the original object are applied and followed by any deformations. The output from the tweak node is a potentially deformed shape. Since no tweaking has been done to the original NURBS plane shape, the tweak node passes the geometry from the shape to the swirl node, **swirl1**, without change. The **swirl1** node is

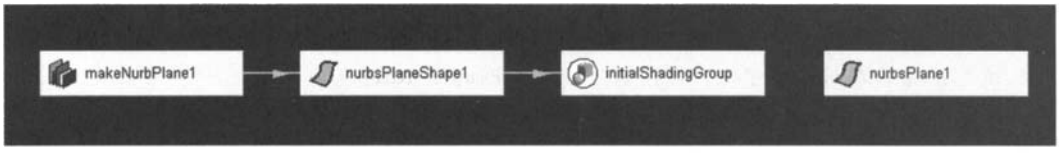


FIGURE 4.29 NURBS plane before deformation

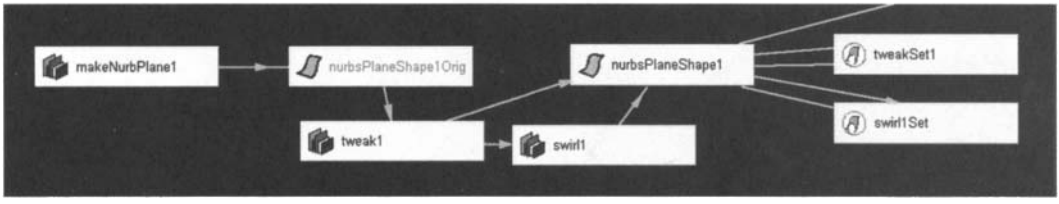


FIGURE 4.30 NURBS plane after deformation

an instance of the **SwirlDeformer** deformer node. It deforms the geometry passed into it. The resulting deformed geometry then is fed into the final NURBS shape, **nurbsPlaneShape1**. This last shape holds the final geometry and displays it on screen.

Whenever a deformer node is created, a **set** node is also generated. The **set** node contains a list of those objects, and possibly their components, to which the deformer should be applied. In this case the **tweakSet1** and **swirl1Set** set nodes have been created for the **tweak1** and **swirl1** deformation nodes, respectively.

Maya automatically handles reordering your deformer node if the user requests it. Also, if the deformer node is deleted, Maya automatically deletes any extraneous nodes, as well as reconnects the deformation chain. Maya also ensures that geometry isn't duplicated unnecessarily from one deformer node to the next. Essentially, one copy of the geometry is successively passed from the first to last deformer. Each deformer passes its deformation to the geometry. The geometry at the end is the result of applying each deformer in turn to the original geometry.

4.8.3 Accessories

For many types of deformers, it is often useful to add one or more accessories. An accessory is a node that the deformer creates either to enable the user to better visualize the deformer attributes or to allow the user to directly manipulate them. For example, the **twist** deformer includes a **twistHandle** node. This allows the user to visualize the various twist parameters, including the start and end angles, as well as the upper and lower bounds.

4.8.4 SwirlDeformer2 Plugin

This plugin demonstrates how to write an accessory that gives further control over the swirl deformer. In particular it creates a locator that is used to define the center and direction of the swirl. Used in this fashion, the accessory locator is commonly referred to as a *handle*. As such the locator is referred to as the **swirlHandle**. Figure 4.31 shows the result of applying the **SwirlDeformer** to a nurbs plane. The **swirlHandle** was then rotated causing the swirl to happen along the rotated direction.

1. Open the **SwirlDeformer2** workspace.
2. Compile it, and load the resulting `SwirlDeformer2.mll` plugin in Maya.
3. Ensure that the `SwirlDeformer.mll` plugin is unloaded.

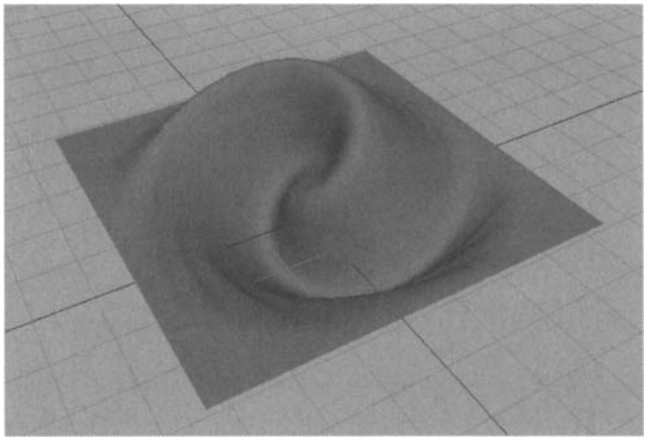


FIGURE 4.31 **SwirlDeformer** with a rotated handle

4. Open the **Swirl.ma** scene.
5. Select the **nurbsPlane1** object.
6. Execute the following in the **Script Editor**:

```
deformer -type swirl;
```

   The swirl deformer is applied to the plane as before.

7. Select the **swirlHandle** object.
8. In the **Channel Box**, set the **swirlHandle**'s **Rotate X** to 25.

   The swirl is now applied at an angle relative to the handle. Experiment by changing the **swirl1** node's **End Dist** attribute to see better how the angle has affected the swirl.

Adding one or more accessories to an existing deformer node is relatively simple. Note that in this example one of Maya's predefined locators was used. It is possible to create your own custom locator to use as a handle. It is also possible to add your own custom manipulators to the deformer node. Many of Maya's standard deformers demonstrate all these features.

Plugin: SwirlDeformer2
File: SwirlDeformer.h

The **SwirlDeformer2** class is defined in the same way as the **SwirlDeformer** class. In order to add accessories to the node, the original **SwirlDeformer** class was modified to include some new functions and a new attribute. The following functions derived from the **MPxDeformerNode** class were implemented. They are explained shortly.

```
virtual MObject &accessoryAttribute() const;
virtual MStatus accessoryNodeSetup( MDagModifier &cmd );
```

An additional matrix attribute, **deformSpace**, that contains the current transformation matrix of the handle object was added.

```
static MObject deformSpace;
```

Plugin: SwirlDeformer2
File: SwirlDeformer.cpp

The new **deformSpace** matrix attribute is added to the node in the `initialize` function.

```
MStatus SwirlDeformer::initialize()
{
MFnMatrixAttribute mAttr;
deformSpace = mAttr.create( "deformSpace", "dSp" );
```

Since the matrix is the transformation matrix of the handle, there is no need to store it.

```
mAttr.setStorable( false );
...
```

The `deform` function is modified to incorporate the new **deformSpace** matrix attribute.

```
MStatus SwirlDeformer::deform( MDataBlock& block,
                               MItGeometry &iter,
                               const MMatrix &localToWorld,
                               unsigned int geomIndex )
{
```

This attribute is retrieved from the data block like the other attributes. The inverse of the matrix is also calculated.

```
...
MDataHandle matData = block.inputValue( deformSpace );
MMatrix mat = matData.asMatrix();
MMatrix invMat = mat.inverse();
...
```

The only major change to the deformation method is that the points are first transformed into the local space of the handle accessory. The deformation is then done in that space before being converted back into the original object space. Converting to the handle space is done by transforming the point by the inverse of the handle's matrix.

```
...
pt = iter.position();
pt *= invMat;
...
```

Once the deformation is complete, the points are converted back to the original local space by transforming the handle's matrix.

```
...
pt *= mat;
iter.setPosition( pt );
...
```

When the deformer node is created, the `accessoryNodeSetup` function is called to create any accessory nodes that the deformer might need. The function is passed an **MDagModifier** so that new nodes can be created and added to the DG.

```
MStatus SwirlDeformer::accessoryNodeSetup( MDagModifier &dagMod )
{
```

A new locator node is created.

```
MObject locObj = dagMod.createNode( "locator", MObject::kNullObj, &stat );
if( !stat )
    return stat;
```

The locator is renamed to something more intuitive.

```
dagMod.renameNode( locObj, "swirlHandle" );
```

The **deformSpace** matrix attribute of the deformer node is controlled by the transformation matrix of the locator node by connecting the **matrix** attribute of the locator transform into the deformer node's **deformSpace** attribute. Whenever the locator is transformed, the **deformSpace** attribute is automatically updated.

```
MFnDependencyNode locFn( locObj );
MObject attrMat = locFn.attribute( "matrix" );
stat = dagMod.connect( locObj, attrMat, thisMObject(), deformSpace );

return stat;
}
```

If the user were to delete the deformer's handle object, the deformer node should also be deleted and vice versa. Fortunately Maya handles this automatically as long as you tell it which of the deformer node's attributes the accessory is affecting. Only one of the affected attributes has to be given. When the deformer's handle object is deleted, the connection to this attribute is deleted, and Maya then deletes the deformer node as well. Likewise, if the deformer node is deleted, the deformer handle object also is deleted.

```
MObject &SwirlDeformer::accessoryAttribute() const
{
return deformSpace;
}
```

4.9 ADVANCED C++ API

This section covers some of the more advanced C++ API topics.

4.9.1 General

REFERENCING NODES

Since an **MObject** is essentially a void pointer to some internal Maya data, it is very important that you don't hold on for too long. In fact, the **MObject** is valid only while the piece of data it is referencing still exists. If that data is deleted for some reason, the **MObject** is not notified and so continues to use what is now an invalid pointer. If the **MObject** is now used, it will most likely cause Maya to crash due to the invalid pointer being dereferenced. If you need to keep a reference to a DAG node, use an **MDagPath** instead. For general DG nodes, use their name. Since a node's name can change, it is important to keep your name reference updated. An **MNodeMessage** can be set up that will notify you when a given node's name changes. This is done by using the **MNodeMessage**'s `addNameChangedCallback` function.

PROXIES

When you define your custom commands and nodes, it may appear that they are the ones actually used in the DG. In reality they are just proxies. All the Maya classes that start with **MPx** define proxy objects.

Say, for instance, you created your own **MyNode** class that is derived from **MPxNode**. When Maya creates an instance of the **MyNode**, it in fact creates two objects. One is an internal Maya object that holds your **MyNode**. Your node isn't used directly in the DG. It is the internal Maya object that sits in the DG, which simply keeps a reference to your MyNode. This is why all the **MPx** derived classes are referred to as proxies. The real node is Maya's internal object.

In fact, you can get a pointer to your node from a DG node by using the **MPxNode**'s `userNode` function. This returns a pointer to the instance of your class that the internal Maya node is using.

Since a user-defined node is composed of two pieces, you have to be careful about what you do in the constructor. In the constructor of the **MyNode** class, you can't call any of the **MPxNode** member functions. This applies to all classes that are directly or indirectly derived from **MPxNode**. The reason for this restriction is that connection between the internal Maya node object and the instance of the **MyNode** isn't made until the instance is fully constructed. So during the construction of

MyNode, the **MPxNode** functionality isn't available. It is available only after the **MyNode** is created, since only then is the connection between the two established.

To make the construction of custom nodes easier, the **MPxNode** defines a virtual function, `postConstructor`, that you can implement in your node. This function is called when the connection between the two objects is created, so you are then free to call any **MPxNode** member function. As such, the **MyNode**'s constructor should be very minimal, and the `postConstructor` function should do most of the initialization work. Of course, if the **MyNode**'s constructor never needs to use any **MPxNode** functions, it won't have to implement the `postConstructor` and so will do all its initialization in its own constructor.

NETWORKED AND NONNETWORKED PLUGS

An often confusing Maya topic is that of networked and nonnetworked plugs. In practice, it isn't necessary to understand the distinction between the two. In fact, how they differ is an implementation issue relating to how Maya keeps connection information between connected attributes. As such, this topic is often redundant for developers. Plugs can be used without the developer ever knowing whether they are networked or nonnetworked. Even though understanding them may not impact your development, they can give you further insight into Maya's internal working.

When a plug is referencing a particular node's attribute, it needs to know only the node and the attribute. From these two pieces of information, it can find the node's specific attribute data. For array attributes, an additional piece of information is needed: the element index. With the addition of the element index, the plug can now locate the node's specific array element data. Since Maya allows attributes to be arbitrarily nested, a plug actually contains a complete path to a given attribute. This path contains indices for array plugs from the root plug to the particular attribute being referenced. To get the complete attribute path for a given plug, use the **MPlug**'s `info` function.

A plug can also serve another purpose in Maya. It can be used to store connection information between connected attributes. It can also store other state information. To make the process of connecting attributes and storing other state information easier, Maya maintains, for each node, an internal plug tree. This tree contains plugs for the node's connected attributes. The DG makes use of this internal plug tree to traverse connections between node attributes. A plug that exists in this internal tree is referred to as a networked plug. A plug that doesn't exist in a plug tree is referred to as a nonnetworked plug. It follows logically that all attributes that are connected will have an associated networked plug.

When an attempt is made to create a plug to an attribute, Maya first looks to see if there is an existing plug to the attribute in the internal plug tree. If there is, this networked plug is returned. If no plug exists, then a nonnetworked plug is created. In both cases, you can use the resulting plug in exactly the same way. The same **MPlug** member functions can be called, irrespective of whether the plug is networked or nonnetworked.

If you need to know if a plug is networked or not, then the **MPlug**'s `isNetworked` function can be used. In the following example, a plug to the **transform** node's **translate** attribute is created. The **MPlug**'s `isNetworked` function returns `true` if the plug is networked or `false` if it isn't.

```
MFnDependencyNode nodeFn( transformObj );
MPlug transPlg = nodeFn.findPlug( "translate" );
bool isNet = transPlg.isNetworked();
```

4.9.2 Dependency Graph

CONTEXTS

A context defines the reason a DG is evaluated. The context of a DG evaluation can be set to many different states. The context can be set to "normal" to indicate that the DG is evaluated at the current time. It can also be set to "at specific time," indicating that the DG should be evaluated at a particular time. The context can also be set to other states, such as "for an instance" or "during inverse kinematics." These later states are internal to Maya, so you can't set or query them directly.

The **MDGContext** is the class used to access and set contexts. It can be initialized with a particular time or another context. In the following example, the context is initialized with a time that is set to frame 12.

```
MTime t( 12, MTime::kFilm );
MDGContext ctx( t );
```

The **MPlug**'s `getValue` function can be used to retrieve the value of an attribute. By default it retrieves the value of the attribute at the current time. The context for the current time is indicated by the **MDGContext**'s `fsNormal` static member. The prototype for the **MPlug**'s `getValue` function for floats is given by the following:

```
MStatus getValue( float&, MDGContext &ctx=MDGContext::fsNormal ) const
```

Notice that it sets the reference to the context to the `fsNormal` object. You can test if a context is the current time by using the **MDGContext**'s `isNormal` function. Unless you explicity give a context, plugs are evaluated at the current time. The following example evaluates a plug at an alternate context.

```
MTime t( 500, MTime::kMilliseconds );
MDGContext ctx( t );
MFnDependencyNode depFn( transformObj );
MPlug transxPlg = depFn.findPlug( "translateX" );
double tx;
tansxPlg.getValue( tx, ctx );
```

DATABLOCKS

Datablocks are where the node's attribute data is stored. A datablock, as its name suggests, is a block of memory. When a node is defined, it also defines all the attributes it contains. The actual amount of memory needed by certain attributes can be worked out in advance. For simple fixed-size attributes, such as `char`, `boolean`, `float`, `2long`, `short`, `3double`, and so on, the memory requirements can be calculated. The memory needed by all these attributes is added up. A chunk of memory of this size is allocated to hold all the datablock data. This is more efficient than allocating separate memory for each of the individual attributes. For all attributes that don't have a fixed size, such as arrays, meshes, and so on, separate memory is allocated. The datablock contains a pointer to the separate memory rather than including it directly inside the datablock memory.

An example will help to further demonstrate. Given is a custom node, **makePyramid**, that generates a polygonal mesh in the shape of pyramid. It has four attrbutes: **width**, **height**, **geom**, and **capBase**. The node is shown in Figure 4.32 with the corresponding data types for each attribute.

A datablock is created for the node as shown in Figure 4.33. The datablock contains a context, which will be explained shortly. All the simple attributes, **width**, **height**, and **capBase**, can be stored in the contiguous memory block. The **geom** attribute has a variable-size data type, `mesh`, so is stored outside of the datablock. A pointer to the mesh is stored in the datablock.

When a node is initially created, it doesn't actually have a datablock. When one of its attributes changes from its default value or a connection is made, the node's datablock is then automatically allocated.

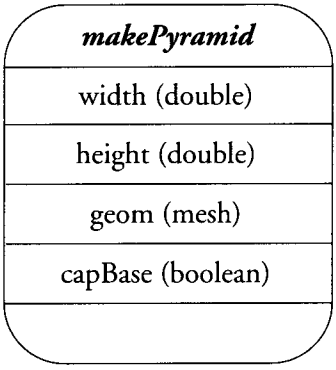


FIGURE 4.32 **makePyramid** node

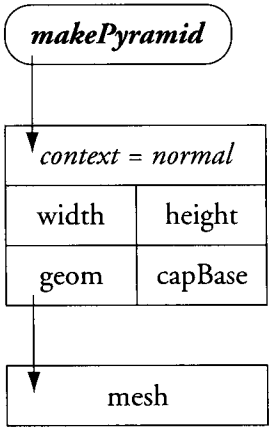


FIGURE 4.33 Datablock for **makePyramid**

A node can actually contain multiple datablocks. Each datablock has associated with it a context. The context defines a particular time instant at which the node is evaluated. Typically there is just a single context, referred to as the *normal context.* This is the context for the current time. A node could be evaluated at different times and under different circumstances and so can have multiple datablocks for each of these individual contexts. Figure 4.34 shows the **makePyramid** node with two datablocks. The first datablock holds the node data when it is evaluated at the current

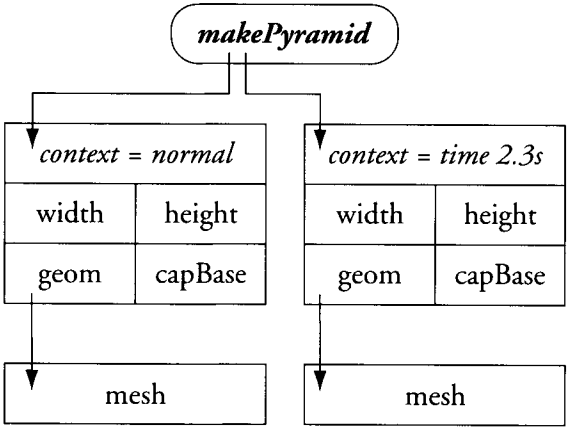


FIGURE 4.34 Multiple datablocks

time (normal). The second datablock holds the node data when the node is evaluated at 2.3 seconds.

Since most nodes are evaluated at the current time, they often contain just a single datablock. The extra datablocks are created and deleted as needed. A node can force the creation of a datablock at a given time by using the **MPxNode**'s `forceCache` function. To get the context for a given datablock, use the **MDataBlock**'s `context` function.

The **MDataHandle** and **MArrayDataHandle** classes are simple objects that understand the memory layout of the datablock. For simple data types that can be stored directly in the datablock, they can efficiently access the data. For data that resides outside the datablock, they use the pointer to dereference the associated data.

PROPAGATION FLAGS

As mentioned in the introductory chapter, when an attribute is changed, all attributes that it affects, including output connections, have their dirty bit set. The dirty bit is propagated to all direct and indirect attributes that the changed attribute affects. In a complex network, even this simple propagation of dirty bits can take some time. To reduce this overhead, a *propagation flag* is associated with each plug. If the propagation flag is set to `true`, then when a plug receives a dirty bit message, it sets its dirty bit and then it propagates this message to other plugs. If the propagation flag is `false`, then it won't pass the dirty bit message onward. The propagation flag effectively prevents propagating of dirty bit messages for plugs that should

already have been set. If a given plug is marked as dirty, it can be assumed that it has propagated this message to all other affected plugs. So if you mark it as dirty again, it won't need to propagate again.

In most cases this methodology works just fine. There may be cases where a plug is not marked dirty, yet the plugs connected upstream are. In this case their dirty bit message hasn't been correctly propagrated to the last plug. It will never recompute, since it is marked as clean. It will never be told that it is dirty, since the propagation flag of the incoming plug is set to `false`.

If this ever occurs, there is fortunately a way to correct this. Use the `dgdirty` command to set all the plugs of a node as either clean or dirty. It will force a propagation of the dirty bit message to all affected nodes, irrespective of their current propagation flag setting.

```
dgdirty $nodeName;
```

To mark all the plugs as clean, use the following:

```
dgdirty -clean $nodeName;
```

HANDLING PASS-THROUGH

All custom nodes are derived, either directly or indirectly, from **MPxNode**. This node contains several attributes, but the one most important to developers is the **nodeState** attribute. This attribute is accessible from Maya's interface by doing the following:

1. Select the node.
2. Open the **Attribute Editor**.
3. Expand the **Node Behavior** frame.
4. Beside the **Node State** prompt, select a new setting from the drop-down list.

Internally, the **nodeState** attribute is an enumerated data type with four values.

- 0 (normal)
- 1 (pass-through)
- 2 (blocking)
- 3 (internally disabled)

The **nodeState** attribute defines whether or not the node should compute its output attributes. Typically, the **nodeState** is set to 0(normal), so it computes its output attributes. By setting the **nodeState** to 1(pass-through), the node passes through its inputs to its outputs without doing any computation on them. This state is listed in the Attribute Editor as `hasNoEffect`.

When developing a node, you need to decide if you'll support the pass-through state. The decision is really based on whether the concept of pass-through is valid for the given type of node. Deformers should support this state. When the **nodeState** is set to pass-through, they should just put the input geometry into the output geometry without doing any deformation on it.

The following code shows how to change the **SwirlDeformer** plugin to support the node state. Before performing the deformation, the function checks whether the **nodeState** attribute is set to 1 (pass-through). If it is, it returns immediately, without doing the deformation.

```
MStatus SwirlDeformer::deform( MDataBlock& block, MItGeometry &iter,
                               const MMatrix &localToWorld,
                               unsigned int geomIndex )
{
MStatus stat;

MDataHandle stateHnd = data.inputValue( state );
int state = stateHdn.asInt();
if( state == 1 ) // Pass through
  return MS::kSuccess;

MDataHandle envData = block.inputValue( envelope );
float env = envData.asFloat();
...
```

CYCLIC DEPENDENCIES

It is possible to create cyclic dependencies. This is when one node feeds into another node and this node then feeds into the first. It is possible to have other intermediate nodes between the two, but the most important property is that if you walk from the first node through its outgoing connections, you eventually return to the same node.

The DG actually handles cyclic dependencies, though the results may not always be what you expect. The result most often depends on which of the nodes is evaluated first. Since the result isn't deterministic, it is best to avoid cyclic dependencies entirely.

This Page Intentionally Left Blank

APPENDIX A

ADDITIONAL RESOURCES

To help you continue your learning of Maya programming, a wide variety of online and offline resources is available.

Online Resources

The Internet contains an enormous variety of computer graphics programming resources. While it is possible to locate information about a given topic using your favorite web search engine, there are some sites that have specific Maya programming information.

COMPANION WEBSITE

The official companion site to this book is located at *www.davidgould.com.* A list, though not exhaustive, of the information available at the site includes:

- MEL scripts and C++ source code for all the examples in the book
- Additional example MEL scripts
- Additional example C++ API source code
- Errata for this book
- Continually updated glossary
- Updated list of other relevant websites and online resources

ADDITIONAL WEBSITES

Of particular note are the following websites that provide specific Maya programming information and forums.

Alias | Wavefront

www.aliaswavefront.com

This is the complete reference site for the Maya product. It contains the latest product development news, example scripts, and plugins. If you intend on creating commercial plugins, then be sure to look at the Alias | Wavefront Conductors program. This is a program by which Alias | Wavefront provides development and marketing support to developers who are going to commercialize their products. There is even support for developers who'd like to distribute their plugins as shareware or freeware.

Highend3D

www.highend3d.com

Best known for its in-depth coverage of the major animation and modeling packages, the Highend3D site is also a great repository of MEL scripts and plugins. It also hosts the Maya Developers Forum, where you can ask questions of other developers.

Bryan Ewert

www.ewertb.com/maya

This site contains a large number of MEL and C++ API tutorials and examples. The MEL basics are covered as well as some of the more advanced topics. The "How To" sections are interesting for developers needing a fix to a particular problem.

Maya Application

The Maya product ships with an extensive collection of Maya programming documentation and examples. Also, don't forget that you can often learn how Maya performs a given set of tasks by turning on **Echo All Commands** in the Script Editor. This can be a great starting guide if you want to do something similar.

DOCUMENTATION

In particular, the MEL and C++ API reference material will be a constant source of important programming information. All the documentation can be accessed from within Maya by pressing **F1** or by selecting **Help** from the main menu. Please note that the following links may depend on which version of Maya and which language you are using.

Learning MEL

maya_install\docs\en_US\html\UserGuide\Mel\Mel.htm

maya_install\docs\en_US\html\InstantMaya\InstantMaya\InstantMaya.htm

(Scroll to the bottom for the Expressions and MEL tutorials.)

Learning C++ API

maya_install\docs\en_US\html\DevKit\PlugInsAPI\PlugInsAPI.htm

For general programming reference material, refer to the **Reference Library** section of the **Master Index**. For specific programming references, visit the following.

MEL Reference

maya_install\docs\en_US\html\Commands\Index\index.html

maya_install\docs\en_US\html\Nodes\Index\indexAlpha.html

C++ API Reference

maya_install\docs\en_US\html\DevKit\PlugInsAPI\classDoc\index.html

maya_install\docs\en_US\html\Nodes\Index\indexAlpha.html

EXAMPLES

The standard Maya product comes with a large number of example MEL scripts and C++ plugins.

MEL Examples

Since Maya's interface is written entirely in MEL, the scripts it uses are provided with the application. There is no better insight into learning how to write professional MEL scripts than by looking at the scripts written by the Maya software engineers. You will find a host of scripts in the following directory, as well its subdirectories:

maya_install\scripts

I strongly recommend perusing them to see how well-designed scripts are written. Please note however that all the scripts provided are the copyright of Alias | Wavefront and cannot be used for derived works. Instead, study them and learn from their example, but don't be tempted to copy and use them as is.

Also be careful when you are reviewing the scripts that you don't accidentally change them. Since they are used by Maya, any changes may cause the system to become unstable and potentially to crash. For this reason, it is best to make a copy of the scripts beforehand.

C++ API Examples

The C++ API example directories include source code for plugins and stand-alone applications as well as motion-capture servers. The examples are located at:

```
maya_install\devkit
```

APPENDIX B

MEL FOR C PROGRAMMERS

For those familiar with C programming, it may come as no surprise that when you first look at a MEL command or script you note a great deal of similarity with the syntax used in the C programming language. In fact MEL is sometimes colloquially referred to as "C with $ signs." While this isn't entirely devoid of truth, there are some important exceptions that differentiate the two languages.

Following is a list of the important differences:

- Since MEL is designed for fast prototyping and also to be more accessible to less-sophisticated programmers, it does away with many of the lower-level system responsibilities that a C programmer is burdened with. One of these responsibilities is the allocation and deallocation of memory. MEL conveniently provides dynamic arrays so that you aren't required to create routines for expanding and shrinking array sizes. MEL handles the allocation and cleanup for you, thereby simplifying the code and reducing the risk of memory-related problems such as segmentation faults or memory leaks.
- MEL does not provide pointers. All variables, except arrays, are passed by value into procedures. All arrays are passed by reference.
- MEL assigns default values to local variables if you don't explicitly initialize them.
- Variables defined in the outermost scope are local by default unless you explicitly define them to be global by using the `global` keyword. This is the inverse of the scoping rules in C, where a variable defined in a unit is global by default unless you explicitly declare it to be `static`.
- The `float` type in MEL is the equivalent of a `double` in C. While the exact precision of this type is machine dependent, it is typically greater than the type used for C's `float`.

- MEL's `int` type is machine dependent, but most likely a signed integer, 32 bits in length.
- MEL comes with a built-in string type, `string`. A variety of operations can be automatically performed on it including concatenation.
- There are no bitwise operators (`|`, `&`, `!`, `~`, and so on) in MEL.
- MEL doesn't support type casting. You can't, for instance, convert an integer to a float using `(float)`. To convert from one type to another, simply assign the type to the destination variable. For example, to convert an integer to a float, use the following:

```
int $intA = 23;
float $fltA = $intA; // Converted to float
... use $fltA in operation
```

  Note that not all types can be assigned to other types.

  Conversion from a string to a number is very simple. Unlike with C, you can simply assign a string to a numeric variable. This works as long as the string contains a valid number.

```
string $v = "123";
int $num = $v; // $num is now 123
```

- Boolean constants include `true`, `false`, `on`, `off`, `yes`, `no`. They have the standard numeric value of 1 for `true` and 0 for `false`. As in C, any statement that doesn't evaluate to 0 is considered `true`.
- Procedures can be defined inside blocks. However, it isn't possible to define a procedure inside of another procedure.
- Procedures can't have default values assigned to their arguments. The following is therefore illegal:

```
proc myScale( string $nodeName = "sphere", float $factor = 1.0 )
{
...
```

- Procedures can't be overloaded. If a procedure with the same name is defined, it overrides, rather than overloads, any previous definitions.
- Unlike C, it is valid to use strings in a `switch` statement. The following example demonstrates this:

```
string $name = "box";
switch( $name )
  {
  case "sphere":
    print "found a sphere";
    break;

  case "box":
    print "found a box";
    break;

  default:
    print "found something else";
    break;
  }
```

- In addition to all the standard flow control constructs (`for`, `while`, `do-while`, `switch`) MEL also has the `for-in` loop construct, which is exactly like a standard `for` loop except that it can be written more succinctly.

  Unlike with C, it isn't possible to define a variable in the `for` loop construct. For instance the following is invalid in MEL:

```
for( int $i=0; ...
```

  The variable has to be defined beforehand as follows:

```
int $i;
for( $i=0; ...
```

This Page Intentionally Left Blank

APPENDIX C

FURTHER READING

Computer graphics encompasses a wide variety of disciplines. However, the foundation of all computer graphics is, undoubtedly, mathematics. In particular, discrete mathematics, linear algebra, and calculus provide the major foundation for almost all computer graphics theory and practice. In addition to mathematics, a solid understanding of programming practices helps you in developing efficient and robust programs.

With a good understanding of both mathematics and programming, you'll have a solid base on which to learn the particular techniques and methods used in computer graphics. Even though the field is continually evolving, there are many computer graphics principles that once learned will hold you in good stead for all future work.

Below is a nonexhaustive list of books that provide a good grounding in their respective areas. Within each section, books are listed in order from basic to advanced.

Mathematics

Selby, Peter, and Steve Slavin. *Practical Algebra.* New York: John Wiley and Sons, 1991.

Thompson, Silvanus P., and Martin Gardner. *Calculus Made Easy.* New York: St. Martin's Press, 1998.

Mortenson, Michael E. *Mathematics for Computer Graphics Applications.* New York: Industrial Press, 1999.

Lengyel, Eric. *Mathematics for 3D Game Programming and Computer Graphics.* Hingham, Mass.: Charles River Media, 2001.

Programming

GENERAL

Deitel, Harvey M., and Paul J. Deitel. *C: How to Program,* 3d ed. Upper Saddle River, N.J.: Prentice Hall, 2000.

Knuth, Donald E. *The Art of Computer Programming,* 3 vols. Boston: Addison-Wesley Publishing Co., 1998.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms,* 2d ed. Cambridge: MIT Press, 2001.

C++ LANGUAGE

Liberty, Jesse. *Sams Teach Yourself C++ in 21 Days Complete Compiler Edition,* 4th ed. Indianapolis: Sams Technical Publishing, 2001.

Deitel, Harvey M., and Paul J. Deitel. *C++: How to Program,* 3d ed. Upper Saddle River, N.J.: Prentice Hall, 2000.

Stroustrup, Bjarne. *The C++ Programming Language,* special 3d ed. Boston: Addison-Wesley Publishing Co., 2000.

Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Design,* 2d ed. Boston: Addison-Wesley Publishing Co., 1997.

Bulka, Dov, and David Mayhew. *Efficient C++: Performance Programming Techniques.* Boston: Addison-Wesley Publishing Co., 1999.

Computer Graphics

GENERAL

Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice in C,* 2d ed. Boston: Addison-Wesley Publishing Co., 1995.

Watt, Alan H. *3D Computer Graphics,* 3d ed. Boston: Addison-Wesley Publishing Co., 1999.

Glassner, Andrew S. *Graphics Gems I.* San Francisco: Morgan Kaufmann Publishers, 1990.

Also see *Graphics Gems II, III, IV, V.*

MODELING

Rogers, David F. *An Introduction to NURBS, with Historial Perspective.* San Francisco: Morgan Kaufmann Publishers, 2001.

Warren, Joe, and Henrik Weimer. *Subdivision Methods for Geometric Design: A Constructive Approach.* San Francisco: Morgan Kaufmann Publishers, 2001.

ANIMATION

Parent, Rick. *Computer Animation: Algorithms and Techniques.* San Francisco: Morgan Kaufmann Publishers, 2002.

IMAGE SYNTHESIS

Watt, Alan, and Mark Watt. *Advanced Animation and Rendering Techniques.* New York: ACM Press, 1992.

Glassner, Andrew S. *Principle of Digital Image Synthesis.* San Francisco: Morgan Kaufmann Publishers, 1995.

Ebert, David S., et al. *Texturing and Modeling.* San Diego: Academic Press, 1998.

Shirley, Peter. *Realistic Ray Tracing.* Natuk, Mass.: A K Peters Ltd., 2000.

Blinn, James. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline.* San Francisco: Morgan Kaufmann Publishers, 1996.

This Page Intentionally Left Blank

GLOSSARY

action An action is a MEL command that doesn't alter or change Maya's state. An action often queries the scene without changing it.

affine transformation A transformation that involves a linear transformation followed by a translation.

animation controller Many 3D packages have specific functionality for animating objects. In 3dsmax they are referred to as controllers. In Softimage, they are fcurves. In Maya, the standard animation controls are the **animCurve** nodes. They allow you to create and edit a curve that then controls a parameter over the range of the animation.

ANSI This an abbreviation of the American National Standards Institute. The institute is involved in defining standards for many computer languages, including C and C++.

API Abbreviation of *application programming interface.* A system provides a programmer with an API. This API defines the complete methods by which a programmer can access and control the given system.

argument An argument to a command or procedure is simply a value given to the command or procedure as input to perform its operation.

array An array is a list of items.

ASCII Abbreviation of American Standard Code for Information Interchange. This is a system for encoding characters using 7 bits.

assignment Assignment consists of storing a value into a variable. The assignment operator (=) is used to store values, for example, `$a = 2`.

attribute This is particular property of a node. For instance, the **makeNurbsSphere** node has a **radius** attribute. When you change this attribute, the sphere changes in size.

axis An axis is a direction. A 3D object has three axes: x, y, z.

black box When the exact operations of a given system aren't known outside the system, it is referred to as a black box. This means that its inner workings can't be seen.

boolean Booleans are used to denote the result of a logical operation. A boolean can be either `true` or `false`.

breakdown key This is a key that depends on keys before and after it. A breakdown key automatically maintains its position relative to the other keys when they are moved.

C++ This is an object-oriented programming language based on the C language.

Cartesian coordinates A coordinate system that defines positions based on their projection onto a series of orthogonal axes.

case sensitive When an operation is case sensitive, it makes a distinction between two names that don't have the same case. For instance, the names *bill* and *Bill* will be considered different in a case-sensitive system.

child This is something that has a parent.

class In C++, a class is the basic construct for defining a self-contained object. Classes have their own member functions and data.

class hierarchy Using standard object-oriented design methods, most complex systems are broken into a hierarchy. At the root (top) is a class with very basic functionality. Other classes (children) are derived from this class to add more specific functionality. As this process continues, you end up with a tree hierarchy of classes.

command A command is used to perform a particular operation. The `sphere` command, for example, is used to create and edit spheres. Commands are used throughout Maya to perform almost all its various operations.

command modes A single command can operate the following variety of modes: creation, edit, and query. When a command is executed in a given mode, it performs a restricted set of operations. When in query mode, it retrieves values. When in creation mode, it creates things.

comment This is some descriptive text that a programmer includes in the source code so that other people can read and understand what the programmer is doing. It is a means of documenting the functionality of a program. A *multiline comment* is a comment that spans more than one line of text.

compile-and-link Compiled languages like C and C++ need to have the source code compiled and linked into machine code in order to run. This is in contrast to scripting languages such as MEL, which interpret instructions and execute them immediately.

component This is the individual values of a vector, point, or other item. A point has three components: x, y, z.

compound attribute This is an attribute that consists of other attributes that are compounded into another more complex attribute.

compute function This is the function in a node that does the calculation of a node's output attributes. The `compute` function takes the input attributes and then calculates the final values for the output attributes.

concatenation This is the process of linking one or more items into a chain.

connection When the value of one attribute feeds into another, a connection is established between the two. It is possible to make and break connections freely.

context When the `compute` function of a node is called, the context defines when, in time, the node is being recalculated.

creation expression An expression that is run when a particle's age is 0, that is, when it is just born.

creation mode This is the mode a command runs in when it is called to create objects or nodes.

cross product The cross product of two vectors is another vector that is perpendicular to the two. It is often used to determine the direction of a vector that is normal to a surface.

curveAnim node These are curve animation nodes. These nodes hold an animation curve that you can edit and modify to animate a particular parameter. These nodes can do standard keyframing and driven-key animation.

DAG Abbreviation for *directed acyclic graph.* This is a technical term for a hierarchy in which none of the children can themselves be their own parents. If you walked from the first node in the hierarchy to the very last, you would never see the same node twice.

DAG path This is the complete path to a given node. The path lists the node and all its ancestor nodes.

data flow model A conceptual model in which data flows through a series of nodes from the first node to the last. Data is modified by each subsequent node.

data type Defines what type of information a variable can hold. Example data types include `string`, `int`, and `float`.

default parent If no explicitly interface element is specified, an element will be added to the default parent. There is a default parent for almost all element types.

deformer A deformer takes one or more points and moves them to new locations.

dependency node A general Dependency Graph node. All Maya nodes are dependency nodes.

dependent attribute This is an output attribute. When an attribute depends on other attributes for its final value, it is considered a dependent attribute. The `MPxNode::attributeEffects` function is used to set this dependent relationship between attributes.

DG Abbreviation for *Dependency Graph.* The DG consists of all the Maya nodes and their connections.

dirty bit This is a flag that an attribute contains to indicate whether it needs to be updated.

dirty bit propagation This is the process whereby a dirty bit message is passed through the DG from one node to another. The message eventually passes from the first attribute to all other attributes that this one affects.

dot product The dot product is simply the result of multiplying all the components of two vectors together and adding the results: `dot product(a,b) = a.x * b.x + a.y * b.y + a.z * b.z`. The dot product is often used to calculate the cosine of the angle between two vectors.

double In C++, this is a data type for storing numbers with decimal digits. It often, though not always, has a higher precision than the `float` data type.

driven key Whereas, in animation, a key is defined by its position in time, a driven key allows you to define a key relative to another attribute.

dynamic attribute This is an attribute that is added to a particular node. This attribute isn't shared by all nodes of the same type but is unique to a given node.

dynamic link library This is a library of programming functionality that is loaded into memory only when needed.

edit mode This is the mode a command runs in when its parameters are being changed.

ELF Abbreviation of *extended layer framework.* This is the framework used in MEL to define interfaces. Interfaces are designed by creating layouts to which controls are then attached. The framework supports arbitrary nesting of layouts and controls.

entry point This is the function that is called when a dynamic link library is loaded into memory.

exit point This is the function that is called when a dynamic link library is unloaded from memory.

Expression In the context of Maya, an Expression is a series of MEL commands that control one or more node attributes. This allows you to programmatically control attributes.

filter Given a large set of items, a filter defines a restricted set.

flag A flag is used to indicate whether something is on or off. It is used as a means of signaling.

float This is the data type for storing numbers with decimal digits. The size of a `float` in MEL is not necessarily the same as in C++.

floating-point A *floating-point number* is used by computers to store numbers with decimal digits. The term refers to the fact that the decimal point can change.

forward kinematics/FK This is the case in which an animator must explicitly define the orientations of all the joints.

function In the C and C++ programming languages, a function defines how to perform a particular operation. A function can, for instance, add two numbers or rotate an object or perform just about any operation. Functions are *called* to execute them.

function set Under Maya's scheme of separating the data from the functions that operate on them, a function set is a C++ class that provides the programmer with access to the data. A function set can be used to create, edit, and query the data without having to know its specific details.

functors A class that implements a set of functions but doesn't provide its own data.

global This defines the scope of variables or procedures. If they are made global, they can be accessed from anywhere.

group In Maya, a group is simply a **transform** node that becomes the parent of all the nodes in the group. The group is all the children of the **transform** node.

GUI Abbreviation of *graphical user interface.* This is the system of windows, dialog boxes, and other user interface elements with which you interact when you use Maya.

handle Something the system gives you in order to later access an object.

hierarchy Any system in which there is a parent-child relationship.

identity matrix A transformation matrix that doesn't have any effect when applied to a point. Technically, the matrix is composed of all 0s with just the diagonal having 1s.

IK solver Maya allows you to write your own inverse kinematic systems. An IK solver determines the orientations of intermediate joints.

initialization This is the value assigned to a variable when it is first defined.

input attribute This is an attribute that provides input to a node. The input attribute's value is often used by the `compute` function to calculate the value of one or more output attributes.

instance An instance of an object is an exact duplicate of the object. An instance is really an object that shares the exact properties of the original. It always stays the same as the original, no matter what changes are made to the original.

int This data type is used to store whole numbers or integers.

in-tangent The in-tangent defines the speed at which the animation curve approaches a key.

interface The specific communication methods through which you communicate with a system. The graphical user interface provides a set of graphical elements that you use to communicate your intentions to the underlying system.

interpreted language This is a computer language in which the source code is interpreted and run immediately. This is different from a compiled language, in which the source code must first be compiled, then linked, before it is run. Interpreted languages tend to be slower than compiled languages, though they are often better for rapid prototyping.

inverse kinematics/IK Through inverse kinematics an animator can control a series of joints by simply placing the last one. All the intermediate joints are calculated by the computer.

joint A joint is like a bone. Joints can be connected to create appendages. Joints are often what the animator moves to control a character.

keyable An attribute that can be animated by keyframing is *keyable.*

keyframe animation In a keyframe animation you define the animation for a parameter by specifying its exact value at a given set of times. The computer can then work out by interpolation what the value should be between the keys.

keystroke This is when a key on the keyboard is pressed.

layout element A layout element is placeholder for other elements. The layout determines the final positioning and sizing of other elements added to it.

library In the context of C++, a library is a repository of functionality that can be used in other programs. A library for handling files allows you to create, open, and edit files. By using a library in your program, you don't have to develop the technology yourself.

local This defines how procedures or variables can be accessed. By making them local, they can be accessed only within the script file or current block.

local space This is the coordinate space in which an object is first defined. In this space no transformations have been applied to the object.

locator A locator is a 3D shape that is displayed in Maya. However, it won't show up in the final render.

loop In a loop an operation is repeated several times.

manipulator This is a visual control with which the user can change an attribute in 3D.

matrix A matrix is a series of rows and columns of numbers. Matrices are used in computer graphics to transform points.

MEL Abbreviation of Maya Embedded Language. This is Maya's interpreted scripting language. It is very close to the C language in syntax but is easier to learn and allows you to write programs to access and control Maya quickly.

mesh A mesh is a series of polygons grouped to form a surface.

modal A modal dialog box prevents you from using the main application until it is dismissed.

namespace A namespace is where a set of names reside. Since they are all in the same set, the names must be different from each other. If you have names that are the same, a *namespace conflict* results.

node A node is the fundamental building block of Maya's Dependency Graph. Nodes contain attributes that a user can change. They also contain a `compute` function that automatically calculates certain attributes.

noise This is a pseudorandom number. For consistent inputs, it generates consistent, though random, numbers.

normal A normal is a vector that is perpendicular to a surface.

NURBS Abbreviation for *nonuniform rational B-spline.* This is a mathematical representation for smooth curves and surfaces.

object space *See* **local space.**

operator An operator is a shorthand method for defining an operation between one or more values. The addition operator is written using the plug sign (+). Other operators include multiply (*), division (/), and so on.

orphaned node A node that was previously connected to other nodes but has since lost all its connections.

output attribute This is an attribute that holds the result of a computation. One or more input attributes are fed into the node's `compute` function that then create an output value that is stored in an output attribute.

out-tangent The out-tangent defines the speed at which the animation curve leaves a key.

parametric space A position defined by parametric coordinates (u, v) rather than explicitly using Cartesian coordinates (x, y, z). This space is relative to the surface of an object and moves when the surface moves.

parent A parent is something that has one or more children. Since a parent can also have a parent, its children can have indirect parents. These are parents (grandparent, great-grandparents, and so on) above their direct parents.

parent attribute In a compound or array attribute, this is the topmost attribute. It is the parent of all the child attributes under it.

particles A particle defines a point in space. Particles are often animated and controlled by applying forces and other physics to them.

per object attribute A single attribute that is used by all particles.

per particle attribute Each particle will receive its own individual attribute.

pipeline In the context of a production studio, the pipeline includes all the various steps that go into making a film. Starting with modeling, then progressing to animation, lighting, and then finally rendering, the pipeline often consists of separate specialized departments.

platform A platform is a particular computer configuration. It includes the operating system and other specific components (CPU and so on). Example platforms include Irix, Linux, and Windows.

plug A plug identifies a particular node's attribute. It is used to access a specific node's attribute values.

plugin A plugin is a program that is integrated into another application. The program *plugs into* the application. Plugins often provide additional functionality that isn't available in the application.

point A point defines a position. In Maya, points are defined in Cartesian coordinates: x, y, z.

polymorphism With regards to an object-oriented programming language, such as C++, polymorphism refers to an object's ability to behave differently depending on its type. This provides a powerful means for making extensions to objects.

postinfinity This is any frame after the last key in an animation curve.

precedence In a programming language, the precedence of an operator determines in what order it is evaluated. An operator with a higher precedence is evaluated before another with a lower precedence. For instance multiplication has a higher precedence than addition.

preinfinity This is any frame before the first key in an animation curve.

procedural animation This is animation that is controlled by a program.

procedure A procedure is a means of defining a particular operation in MEL and is used to perform some operation and often return a result. This is conceptually the same as the C language's function.

propagation flag A flag that determines whether a dirty bit message will be propagated to output connections.

pseudocode Pseudocode is a shorthand way of describing a computer program. Rather than use the specific syntax of a computer language, more general wording is used. Using pseudocode makes it easier for a nonprogrammer to understand the general workings of the program.

push-pull model A conceptual model in which data is both pushed and pulled through a series of nodes. This model is more efficient than the data flow since only nodes that need updating are updated. Maya's Dependency Graph works on this principle.

query mode This is the mode a command runs in when its parameters are being queried.

random number A number that is determined entirely by chance.

redo When an command is undone, it can be reexecuted by choosing to redo it.

render To take the scene information models, lights, camera, and so on and make a final image.

root This is the fictitious node that is the parent of all other nodes in the scene. There is the concept of a root node so that the entire scene can be thought of as a tree, starting at the root.

rotate/rotation Rotating an object is the same as spinning it around. This changes its orientation. The point about which it rotates is called the *rotation pivot.* A wheel has its rotation pivot in its center.

runtime expression An expression that is run when a particle is older than zero.

scale Scaling an object means resizing it. A scale is uniform if the object is resized evenly. With nonuniform scaling, an object can be made wider, higher, or deeper without keeping its original proportions.

scene The scene consists of all the Maya data. It includes all the models, their animation, effects, settings, and so on.

scope The scope of a variable defines whether or not it can be accessed. If a variable has global scope, it can be accessed everywhere. If it has local scope, it can be accessed only in the block in which it is defined and all inner blocks.

script A text file that contains MEL statements.

scripting language A scripting language differentiates itself from other typical languages in that it is usually simpler to learn and use as well as not needing to be compiled. The language is interpreted at runtime, so you can execuate instructions immediately.

seed A number used to initialize a random number generator.

set A set is simply a list of items. When an object is put into a set, it is made a part of the list.

set-driven keys Set-driven keys are used to define a relationship between one parameter and another. Unlike keyframes, which assume that you are using time, a set-driven key can use any parameter (driver) to drive another parameter. The relationship is defined by editing a curve.

shader A shader defines the final surface properties of an object. For example, a shader can define the color, reflectivity, and translucency of a surface.

shape This is the general term for all 3D data that can be displayed in Maya. Shapes include curves, surfaces, and points.

shape node A node that holds a shape, such as a polygonal mesh, curve, NURBS surface, or particles.

sibling A sibling is a child that shares the same parent.

skeleton A hierarchy of joints that define the inner structure of a character.

skinning This is the process whereby a model is wrapped around a skeleton. When the skeleton moves, the model moves correspondingly. The model effectively forms a skin over the skeleton joints.

sourcing This is the process whereby a MEL script is loaded into Maya and then executed.

space A space is a particular frame of reference for an object. Specifically, it defines the transformations that are applied to an object to put it into the frame of reference.

spring A spring provides a means of describing and calculating the elasticity, mass, damping, and so on between two points.

spring laws These define how a set of springs react given a set of forces.

string This is a series of characters; text.

structured programming This is a design approach whereby complex systems are broken down into smaller, more manageable pieces.

tangent In the context of an animation curve key, tangents define how values are interpolated between successive keys. By changing a key's tangents you can make the animation faster or slower between keys.

time node The **time** node is used to store time. It has a single attribute, **outTime,** that holds the time. While the **time1** node holds the current time, it is possible to create additional **time** nodes.

tool A tool defines a set of specific steps that must be completed in order to perform an operation. Tools often require the user to select something with the mouse before the operation can be completed.

transform node A DAG node used to specify the position, orientation, and size of a shape.

transformation hierarchy A single transformation positions, orients, and sizes a given object. By putting the transformations into a hierarchy, you can have a series of transformations applied to an object. An object that has parent transformations is also affected by those transformations.

transformation matrix A transformation matrix is a shorthand way of describing the positioning, rotating, and sizing of an object. When a transformation matrix is applied to an object, it often is in a different place, orientation, and size afterwards. The *inverse* of a transformation matrix restores the object to its original place, orientation, and size.

translate/translation Translating an object is the same as moving it.

translator In Maya, a translator is a piece of software that can translate data from one format into a format that Maya can understand. A translator may, for example, take a Softimage file and convert it to a Maya file. Translators are also referred to as importers and exporters since they can take information into and out of Maya.

tree This is the metaphor used to describe hierarchies.

truncate Truncating is when something is removed to make it compatible with something smaller. A decimal number can often be truncated to an integer by removing the decimal digits.

tweak This refers to the general process of editing something. When you tweak an object, you are simply changing it. Before a scene is final, it often undergoes a lot of tweaking.

tweak node A tweak node is used by Maya to store all the individual moves to some geometry's points.

underworld This is a parametric space (*u, v*) rather than a Cartesian space (*x, y, z*). Parametric positions are guaranteed to always "stick" to the underlying parametric object (NURBS curve or surface).

undo To remove the effect of a command, it can be undone. Undoing a command restores Maya to the state it was before the command was executed.

vector A vector defines a direction. Vectors can also have a magnitude that defines their length. A vector that has a length of 1 is called a *unit vector.*

void pointer A generic pointer that can point to an object of any type.

world space The space in which all the objects in the scene are displayed. The world space is the result of applying all of an object's parent transformations.

This Page Intentionally Left Blank

INDEX