Chapter 7

Basic Tools with Maya Commands

CHAPTER OUTLINE

BY THE END OF THIS CHAPTER, YOU WILL BE ABLE TO:

- Describe Maya's GUI management system.
- Create a basic window using Maya commands.
- Design a base class for tool option windows.
- Implement menus, layouts, and controls in windows.
- Compare and contrast different ways to link commands to GUI controls.

Maya Python for Games and Film. DOI: 10.1016/B978-0-12-378578-7.00007-7

© 2012 Elsevier Inc. All rights reserved.

- Extend a base window class to quickly create new option windows.
- Design reusable functions that are separate from your GUI.
- Work with files from a GUI.
- Serialize object data with the cPickle module.

As you delve into the design of custom tools and GUIs, you have a variety of options available. One of the most basic approaches to designing GUIs in Maya is to use the functionality available in the cmds module. Because the cmds module is interacting with Maya's Command Engine, this approach should be immediately familiar. However, because the Command Engine was originally designed with MEL in mind, using commands to create a GUI can be a little cumbersome.

Thankfully, Python's support for object-oriented programming allows you to develop GUIs in ways that would be impossible with MEL. Taking advantage of Python classes in conjunction with basic Maya commands is the easiest way to build and deploy GUI windows. Moreover, working with the cmds module introduces many of the underlying mechanics of Maya's GUI system.

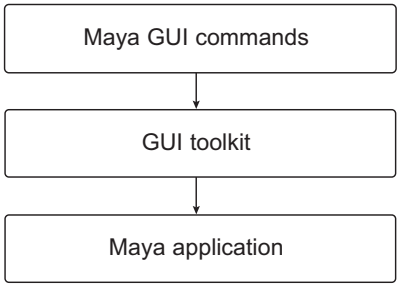
In this chapter, we will first discuss some core technical concepts related to Maya's GUI and then develop a base class for tool option windows. We then explore some of Maya's built-in GUI controls and demonstrate how you can easily extend this class to quickly create new tools. Finally, we discuss some advanced topics related to tool creation, such as serializing data and working with files, by examining a simple pose manager tool.

MAYA COMMANDS AND THE MAYA GUI

We have pointed out many times that the Command Engine is the primary interface for working with Maya. In addition to those commands that manipulate Maya scene objects and operate on files, some commands allow for the creation and manipulation of GUI objects, such as windows and buttons. These commands do not interface with the Maya application directly, but rather communicate with a GUI toolkit (Figure 7.1).

Fundamentally, the Maya GUI consists of a set of controls and windows created using MEL commands. Much like nodes in the Dependency Graph, GUI elements in Maya are accessible via unique string names, which can often become incredibly verbose. The following example illustrates this general concept.

1. In the Script Editor window, select the menu option **History → Echo All Commands**.



■ **FIGURE 7.1** Maya GUI commands interact with a GUI toolkit, which in turn interacts with the Maya application (to display graphics or execute other commands).

2. Click in one of the menus in Maya's main menu bar to open it and then move your cursor left and right to cause other menus to open and close in turn.
3. Look at the output in the Script Editor. You should see a variety of statements executed, the arguments for which show a path to the open menu. For instance, when scrolling over menus in the Polygons menu set, you may see something like the following lines in the History Panel.

```
editMenuUpdate MayaWindow|mainEditMenu;
checkMainFileMenu;
editMenuUpdate("MayaWindow|mainEditMenu");
ModObjectsMenu MayaWindow|mainModifyMenu;
PolygonsSelectMenu MayaWindow|mainPolygonsSelectMenu;
PolygonsNormalsMenu MayaWindow|mainPolygonsNormalsMenu;
PolygonsColorMenu MayaWindow|mainPolygonsColorMenu;
```

Notice that the names of these menu items look similar to **transform** nodes in a hierarchy: they are given unique, pipe-delimited paths. In fact, the menus in Maya are constructed in a similar way. In each of these examples, the `MayaWindow` GUI object is the parent of some menu item, as indicated by the vertical pipe character separating them. In MEL, the "MayaWindow" string is also stored in the global variable `$gMainWindow`.

Just like **transform** nodes in your scene, full GUI object names must be globally unique. Consequently, the names of GUI controls can sometimes be a little unwieldy to maintain uniqueness. Nested controls can be even more frightening to look at. In some versions of Maya, for instance, clearing history in the Script Editor (**Edit → Clear History**) displays something like the following line in the Script Editor's History Panel.

```
// Result: scriptEditorPanel1Window|TearOffPane|
scriptEditorPanel1|formLayout37|formLayout39|paneLayout1|
cmdScrollFieldReporter1 //
```

As you can see, the selected menu option is a child at the end of a very long sequence of GUI objects. Fortunately, when you design GUIs in a module using cmds, the commands you execute will return these names, so you will hopefully never have to concern yourself with them directly. It is important, however, to understand that because these GUI object names must be unique, you should ensure that your own GUIs do not have conflicting names at their top levels.

4. Before proceeding, it is advisable that you disable full command echoing (**History → Echo All Commands**) in the Script Editor, as it can degrade performance.

BASIC GUI COMMANDS

When creating GUI controls with Maya commands, we unfortunately have no visual editor available for creating interfaces. Instead, we must create everything entirely programmatically. Although this process can be a little tedious for complex GUIs, it is easy to test out a new user interface element quickly, since commands respond immediately. That being said, however, Maya GUIs operate in retained mode (in contrast to the immediate mode that many videogames use, for example). The primary consequence as far as we need to be concerned is that our custom GUIs will not update unless we explicitly rerender them somehow. As such, the basic steps you must take are to first define a window and then explicitly render it.

Windows

One of the most common GUI objects is a window, which can house other controls as children. You can create a GUI window using the `window` command, but must execute the `showWindow` command to display it.

1. Execute the following lines in the Script Editor to create a window with the handle "ar_optionsWindow" and then show it. You should see an empty window like that shown in Figure 7.2.

```
import maya.cmds as cmds;
win = cmds.window(
    'ar_optionsWindow',
    title='My First Window',
    widthHeight=(546,350)
);
cmds.showWindow(win);
```

This example first creates a window with a specified (hopefully unique) handle, a title bar string, and a size equal to that of most of Maya's standard

■ **FIGURE 7.2** An empty window.

tools (546 pixels wide by 350 pixels high).[1] After the window is created, it is shown.

Note the handle we assigned to our window: "ar_optionsWindow". One way that many Maya programmers attempt to avoid naming conflicts is to prefix their GUI elements' names with their initials or the initials of their studios. At this point, you can use the window's unique handle to access it further.

2. With your window still up, try to change the title of your window by executing the following code.

```
win = cmds.window(
    'ar_optionsWindow',
    title='My Second Window',
    widthHeight=(546,350)
);
```

You should see an error in the History Panel informing you that the name is already in use.

```
# Error: RuntimeError: file <maya console> line 4: Object's
name 'ar_optionsWindow' is not unique. #
```

To make changes to a GUI, you must destroy it, make your change, and then show it again. You can destroy your window by pressing the close button in its corner or using the `deleteUI` command.

[1]To spare ourselves explanation of how we came up with different dimensions, button sizes, spacing, and so forth, it is worth noting that we refer to the getOptionBox.mel script that ships with Maya. You can find this script in the <Maya application directory>/scripts/others/ folder if you would like more details.

3. Execute the following code to delete the UI, assign a new title, and then show it again.

```
cmds.deleteUI(win, window=True);
win = cmds.window(
    'ar_optionsWindow',
    title='My Second Window',
    widthHeight=(546,350)
);
cmds.showWindow(win);
```

Because of this requirement—as well as a number of other reasons—it is often more convenient to organize your windows into classes.

BUILDING A BASE WINDOW CLASS

While you can simply execute GUI commands in the Script Editor or in a module, as in the previous example, this approach can quickly become tedious. Designing new windows from scratch isn't always fun, especially if you know you will want some basic parts in all of your windows. Fortunately, Python's support for object-oriented programming offers solutions that surpass any organizational strategies available in MEL. In this section, we will make our basic window a class and add a set of controls to create a template that resembles Maya's built-in tool option windows.

1. Execute the following lines to create the **AR_OptionsWindow** class. From this point forward if you are using the Script Editor you will find it useful to highlight all of your code before executing it with **Ctrl + Enter** so it is not cleared from the Input Panel.

```
class AR_OptionsWindow(object):
    def __init__(self):
        self.window = 'ar_optionsWindow';
        self.title = 'Options Window';
        self.size = (546, 350);
    def create(self):
        if cmds.window(self.window, exists=True):
            cmds.deleteUI(self.window, window=True);
        self.window = cmds.window(
            self.window,
            title=self.title,
            widthHeight=self.size
        );
        cmds.showWindow();
```

In the **AR_OptionsWindow** class, the **__init__()** method initializes some data attributes for the window's handle, title, and size. The **create()** method actually (re)draws the window.

2. Execute the following lines in a new Python tab in the Script Editor to create a new instance of the **AR_OptionsWindow** class and call its **create()** method.

```
testWindow = AR_OptionsWindow();
testWindow.create();
```

With our window properly sized and organized in a class, we will want to add a menu and some controls to be more useful. If you are using the Script Editor as opposed to an external IDE, leave the **AR_OptionsWindow** class up in a working Python tab, as we will modify it throughout the rest of this example.

Menus and Menu Items

Adding menus to windows is quite simple. You must enable support for menus by passing a value of True with the `menuBar` flag when you call the `window` command. After that point, you can add menus using the `menu` command, and add children to them using `menuItem` and related commands.

3. Add a new data attribute called `supportsToolAction` in the **__init__()** method and assign a default value of False to it. This data attribute will be used to disable certain menu items by default.

```
def __init__(self):
    self.window = 'ar_optionsWindow';
    self.title = 'Options Window';
    self.size = (546, 350);
    self.supportsToolAction = False;
```

4. Add a new method to the **AR_OptionsWindow** class called **commonMenu()** with the following contents. This method will add common menu items shared across all of Maya's tools.

```
def commonMenu(self):
    self.editMenu = cmds.menu(label='Edit');
    self.editMenuSave = cmds.menuItem(
        label='Save Settings'
    );
    self.editMenuReset = cmds.menuItem(
        label='Reset Settings'
    );
    self.editMenuDiv = cmds.menuItem(d=True);
    self.editMenuRadio = cmds.radioMenuItemCollection();
    self.editMenuTool = cmds.menuItem(
        label='As Tool',
        radioButton=True,
        enable=self.supportsToolAction
    );
```

```
        self.editMenuAction = cmds.menuItem(
            label='As Action',
            radioButton=True,
            enable=self.supportsToolAction
        );
        self.helpMenu = cmds.menu(label='Help');
        self.helpMenuItem = cmds.menuItem(
            label='Help on %s'%self.title
        );
```

The **commonMenu()** method first creates a menu with the label "Edit" and adds Maya's ordinary menu items to it: Save Settings, Reset Settings, a divider, and then two radio buttons to use the window as a tool or as an action.[2]

Note that we call the `radioMenuItemCollection` command to initiate a sequence of items in a radio button group. The subsequent `menuItem` calls then enable the `radioButton` flag to make them members of this group. We also disable these items by default using the `supportsToolAction` data attribute, which we added in the **setupWindowAttributes()** method in the previous step.

We follow these commands with another call to the `menu` command to create the help menu with one item ("Help on Options Window"). As you can see, each successive call to the `menu` command establishes the most recently created menu as the default parent for successive `menuItem` calls.

5. In the **create()** method, add the `menuBar` flag to the `window` call, and then add a call to the **commonMenu()** method before showing the window.

```
    def create(self):
        if cmds.window(self.window, exists=True):
            cmds.deleteUI(self.window, window=True);
        self.window = cmds.window(
            self.window,
            title=self.title,
            widthHeight=self.size,
            menuBar=True
        );
        self.commonMenu();
        cmds.showWindow();
```

[2]Although we tend to use the term *tool* fairly loosely in this text, Maya distinguishes tools and actions in its GUI. A *tool* is something that works continuously by clicking, such as any of the Artisan tools (e.g., Paint Skin Weights, Paint Vertex Colors) or the basic transform tools (e.g., Move, Rotate, Scale). An *action* is a "one-shot" operation, like most of the menu items. Most windows you will create with our template in this chapter will be actions, though some menu items, such as those in the Edit NURBS menu, support both modes. For more information, browse the Maya help in **User Guide → General → Basics → Preferences and customization → Customize how Maya works → Switch operations between actions and tools**.

■ **FIGURE 7.3** A basic window with a menu.

6. If you are working in the Script Editor, execute the **AR_Options Window** class's code again to update it. Remember to highlight all of it before pressing **Ctrl + Enter** so you do not clear the Input Panel.
7. Create a new **AR_OptionsWindow** instance and call its **create()** method. The modifications you made in the previous steps add a menu bar that resembles Maya's default menus for tool options, as shown in Figure 7.3.

```
testWindow = AR_OptionsWindow();
testWindow.create();
```

Although our menu looks pretty good by Autodesk's standards, none of its menu items actually do anything yet. At this point, we should look into adding some commands.

Executing Commands with GUI Objects

Many GUI commands, including `menuItem`, implement a `command` flag, which allows you to specify what happens when the GUI control is used.

Recall that when Maya was first created, the only scripting language available in it was MEL, and that MEL is not object-oriented. Consequently, when using MEL to create a GUI, this flag's argument value contains a string of statements to execute either commands or a MEL procedure. As a Python user, you can pass this flag a pointer to a function or a string of statements to execute.

Passing a Function Pointer

A common and safe approach for menu commands is to pass a function pointer to a method in your class.

8. Add a **helpMenuCmd()** method to the **AR_OptionsWindow** class with the following contents. This method will load the companion web site in a browser.

```
def helpMenuCmd(self, *args):
    cmds.launch(web='http://maya-python.com');
```

9. Change the assignment of the `helpMenuItem` attribute in **commonMenu()** and pass it a pointer to the **helpMenuCmd()** method.

```
self.helpMenuItem = cmds.menuItem(
    label='Help on %s'%self.title,
    command=self.helpMenuCmd
);
```

   If you execute your class definition again to update it in __main__, instantiate the class, and then call its **create()** method, the help menu item will now launch the companion web site. (Try to avoid tormenting your coworkers by sending their help requests to *http://lmgtfy.com/*.)

```
testWindow = AR_OptionsWindow();
testWindow.create();
```

Although this approach is clean, straightforward, and safe, it suffers at least one important limitation. Namely, it does not allow you to pass arguments to the function specified with the `command` flag. If you were to print `args` inside the **helpMenuCmd()** method, you would see that the method is implicitly passed a tuple argument with one item.

```
(False,)
```

While you can largely bypass this problem by putting your GUI into a class and reading data attributes inside the method you call, this issue prevents you from specifying a pointer to any commands in the cmds module (since they will expect different object lists) or from passing a value that you may not have defined as a data attribute (and hence would like to pass as a keyword argument).

Passing a String

Another option you have for issuing commands is to pass a string of statements. Much like Python's built-in **eval()** function, this technique allows you to simply use string formatting operations to pass in arguments, as in the following hypothetical snippet, which creates a personalized polygon sphere.

```
import os;
import maya.cmds as cmds;
class SphereWindow(object):
    def __init__(self):
        self.win = 'arSphereSample';
        if cmds.window(self.win, exists=True):
            cmds.deleteUI(self.win);
        self.win = cmds.window(
            self.win,
            widthHeight=(100,100),
            menuBar=True
        );
        self.menu = cmds.menu(
            label='Create'
        );
        cmds.menuItem(
            label='Personalized Sphere',
            command='import maya.cmds;' +
                'maya.cmds.polySphere(n="%s")'%
                os.getenv('USER')
        );
        cmds.showWindow();
win = SphereWindow();
```

Apart from being an annoyance, this technique also has a big problem associated with it. Namely, the string you pass is executed in __main__. Consequently, you may have problems when trying to access functions in modules. As you saw, because you cannot make assumptions about what modules have been imported into __main__, you may need to include **import** statements if you want to execute Maya commands.

You may find it tedious to include **import** statements for all of your controls to simply ensure that modules you require exist. One technique to combat this problem is to use the **eval()** function in the maya.mel module to invoke the `python` command, allowing you to execute an **import** statement in __main__. The following code snippet illustrates this approach.

```
import os;
import maya.cmds as cmds;
import maya.mel as mel;
mel.eval('python("import maya.cmds");');
class SphereWindow(object):
    def __init__(self):
        self.win = 'arSphereSample';
        if cmds.window(self.win, exists=True):
            cmds.deleteUI(self.win);
        self.win = cmds.window(
            self.win,
```

```
            widthHeight=(100,100),
            menuBar=True
        );
        self.menu = cmds.menu(
            label='Create'
        );
        cmds.menuItem(
            label='Personalized Cube',
            command='maya.cmds.polyCube(n="%s")'%
                os.getenv('USER')
        );
        cmds.menuItem(
            label='Personalized Sphere',
            command='maya.cmds.polySphere(n="%s")'%
                os.getenv('USER')
        );
        cmds.showWindow();
win = SphereWindow();
```

While this approach may be tempting, it is also dangerous and ignores the point of module scope. Thankfully, there is an ideal approach available for most versions of Maya with Python support.

Using the functools Module

Python 2.5 introduced a module called functools, which provides a range of operations for working with higher-order functions. Because it is a complex module we only cover a basic application here. Refer to Section 9.8 of Python Standard Library for more information on the functools module at large.

If you are working in Maya 2008 or later (i.e., not 8.5), you can use the **partial()** function in the functools module to pass a function with arguments to the `command` flag. This technique is documented in the Maya help (**User Guide → Scripting → Python → Tips and tricks for scripters new to Python**). As such, we show only a short hypothetical example here. The following code creates a small window with menu options to create one, two, three, four, or five evenly spaced locators along the *x*-axis.

```
from functools import partial;
import maya.cmds as cmds;
class LocatorWindow(object):
    def __init__(self):
        self.win = cmds.window(
            'ar_locSample',
            widthHeight=(100,100),
            menuBar=True
        );
```

```
        self.menu = cmds.menu(
            label='Make Locators'
        );
        for i in range(5):
            cmds.menuItem(
                l='Make %i'%(i+1),
                command=partial(self.makeLocCmd, i+1)
            );
        cmds.showWindow();
    def makeLocCmd(self, numLocators, *args):
        locs = []
        for i in range(numLocators):
            locs.append(
                cmds.spaceLocator(
                    p=[-(numLocators+1)*0.5+i+1,0,0]
                )[0]
            );
        cmds.select(locs);
win = LocatorWindow();
```

As you can see inside the **for** loop in the **__init__()** method, the **partial()** function lets us pass not only a pointer to a function, but also any number of arguments to correspond to its parameter list. Fortunately, our present menu items are relatively simple, but this trick is essential for creating a more involved GUI.

10. Return to the **AR_OptionsWindow** class and add four placeholder methods for child classes to override: **editMenuSaveCmd()**, **editMenuResetCmd()**, **editMenuToolCmd()**, and **editMenuActionCmd()**. Remember to also specify these methods for their respective controls. Your complete class should currently look like the following example.

```
class AR_OptionsWindow(object):
    def __init__(self):
        self.window = 'ar_optionsWindow';
        self.title = 'Options Window';
        self.size = (546, 350);
        self.supportsToolAction = False;
    def create(self):
        if cmds.window(self.window, exists=True):
            cmds.deleteUI(self.window, window=True);
        self.window = cmds.window(
            self.window,
            title=self.title,
            widthHeight=self.size,
            menuBar=True
        );
```

```
        self.commonMenu();
        cmds.showWindow();
    def commonMenu(self):
        self.editMenu = cmds.menu(label='Edit');
        self.editMenuSave = cmds.menuItem(
            label='Save Settings',
            command=self.editMenuSaveCmd
        );
        self.editMenuReset = cmds.menuItem(
            label='Reset Settings',
            command=self.editMenuResetCmd
        );
        self.editMenuDiv = cmds.menuItem(d=True);
        self.editMenuRadio = cmds.
        radioMenuItemCollection();
        self.editMenuTool = cmds.menuItem(
            label='As Tool',
            radioButton=True,
            enable=self.supportsToolAction,
            command=self.editMenuToolCmd
        );
        self.editMenuAction = cmds.menuItem(
            label='As Action',
            radioButton=True,
            enable=self.supportsToolAction,
            command=self.editMenuActionCmd
        );
        self.helpMenu = cmds.menu(label='Help');
        self.helpMenuItem = cmds.menuItem(
            label='Help on %s'%self.title,
            command=self.helpMenuCmd
        );
    def helpMenuCmd(self, *args):
        cmds.launch(web='http://maya-python.com');
    def editMenuSaveCmd(self, *args): pass
    def editMenuResetCmd(self, *args): pass
    def editMenuToolCmd(self, *args): pass
    def editMenuActionCmd(self, *args): pass
```

Our base class is now almost wrapped up! We will finally add some buttons and then it is ready to go.

Layouts and Controls

In a GUI hierarchy, a window cannot have controls like buttons as its immediate children. Rather, the window must have one or more layouts as children, which may have controls or other layouts as their children. A layout specifies how its children are to be arranged. In the next few steps,

■ **FIGURE 7.4** A basic window with a menu and three buttons.

we will add a layout to the **AR_OptionsWindow** class to create a row of buttons, as shown in Figure 7.4.

Basic Layouts and Buttons

Recall in Chapter 6 that we showed how most actions in the Maya GUI consist of a basic three-button layout. The first button performs the window's action and closes it, the second only performs the action, and the third only closes the window.

11. Add a new data attribute, `actionName`, in the **__init__()** method for the **AR_OptionsWindow** class to set the name that will display on the window's action button. Most windows in Maya default to something like Apply and Close.

```
def __init__(self):
    self.window = 'ar_optionsWindow';
    self.title = 'Options Window';
    self.size = (546, 350);
    self.supportsToolAction = False;
    self.actionName = 'Apply and Close';
```

12. Add methods to the **AR_OptionsWindow** class for the common buttons: **actionBtnCmd()**, **applyBtnCmd()**, and **closeBtnCmd()**.

```
def actionBtnCmd(self, *args):
    self.applyBtnCmd();
    self.closeBtnCmd();
def applyBtnCmd(self, *args): pass
```

```
    def closeBtnCmd(self, *args):
        cmds.deleteUI(self.window, window=True);
```

These methods will be called when the corresponding buttons are pressed. Because of how we have written these methods, the only one that subclasses will need to override is the **applyBtnCmd()** method. The **closeBtnCmd()** method simply closes the window, and the **actionBtnCmd()** method invokes the Apply behavior and then the Close behavior.

13. Add the following **commonButtons()** method to the **AR_Options Window** class right before the **helpMenuCmd()** definition. We will refer back to this method momentarily after some additional changes, so you will be able to compare the code with the visual result.

```
    def commonButtons(self):
        self.commonBtnSize = ((self.size[0]-18)/3, 26);
        self.commonBtnLayout = cmds.rowLayout(
            numberOfColumns=3,
            cw3=(
                self.commonBtnSize[0]+3,
                self.commonBtnSize[0]+3,
                self.commonBtnSize[0]+3
            ),
            ct3=('both','both','both'),
            co3=(2,0,2),
            cl3=('center','center','center')
        );
        self.actionBtn = cmds.button(
            label=self.actionName,
            height=self.commonBtnSize[1],
            command=self.actionBtnCmd
        );
        self.applyBtn = cmds.button(
            label='Apply',
            height=self.commonBtnSize[1],
            command=self.applyBtnCmd
        );
        self.closeBtn = cmds.button(
            label='Close',
            height=self.commonBtnSize[1],
            command=self.closeBtnCmd
        );
```

14. Add a call to **commonButtons()** in the **create()** method right after the call to **commonMenu()**.

```
    def create(self):
        if cmds.window(self.window, exists=True):
            cmds.deleteUI(self.window, window=True);
```

```
self.window = cmds.window(
    self.window,
    title=self.title,
    widthHeight=self.size,
    menuBar=True
);
self.commonMenu();
self.commonButtons();
cmds.showWindow();
```

15. Evaluate all of your updates to the **AR_OptionsWindow** class and then create a new **AR_OptionsWindow** instance. If you are working in the Script Editor, remember to highlight your class definition and press **Ctrl + Enter**.

```
win = AR_OptionsWindow();
win.create();
```

You should now see a window like that shown in Figure 7.5. If you press the buttons, the "Apply and Close" and "Close" buttons simply close the window, and the "Apply" button does nothing yet. Let's look back at the **commonButtons()** method to compare the code with our results. We first specify a size for our buttons as a tuple, `commonBtnSize`, which stores a (width, height) pair.

```
self.commonBtnSize = ((self.size[0]-18)/3, 26);
```

The basic idea for computing the width is that we want 5 pixels of padding on the left and right, and 4 pixels of padding in between the buttons (5 + 4 + 4 + 5 = 18). Hence, we subtract a total of 18 from the window's width before dividing by 3. The height is 26 pixels.

■ **FIGURE 7.5** The intermediate results of the **AR_OptionsWindow** class using a row layout.

Next, we create a row layout, `commonBtnLayout`, which will serve as the parent for our buttons. Remember that controls like buttons must have a layout parent of some kind.

```
self.commonBtnLayout = cmds.rowLayout(
    numberOfColumns=3,
    cw3=(
        self.commonBtnSize[0]+3,
        self.commonBtnSize[0]+3,
        self.commonBtnSize[0]+3
    ),
    ct3=('both','both','both'),
    co3=(2.5,0,2.5),
    cl3=('center','center','center')
);
```

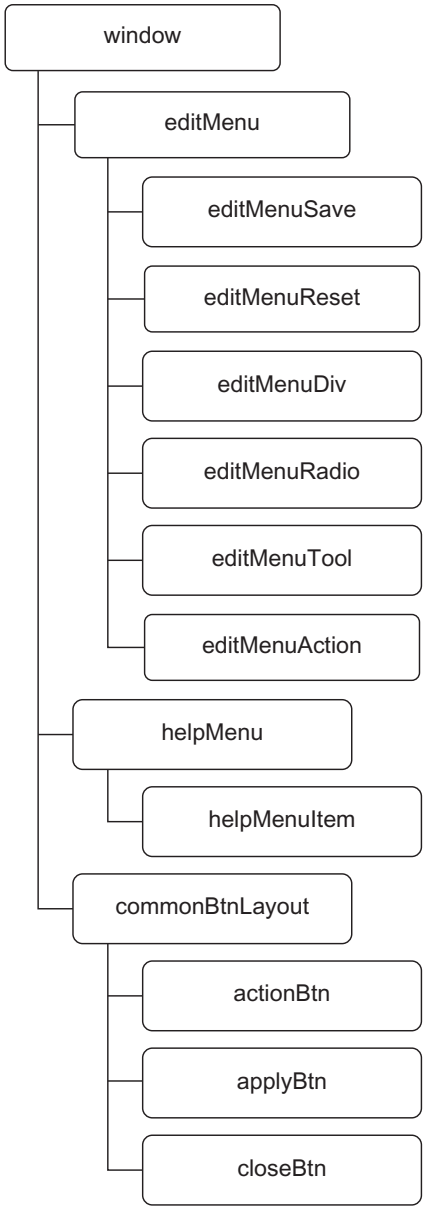
When invoking the `rowLayout` command, we specify that the layout should have three columns and then set properties for column width, attachment, offset, and text alignment. Attachment specifies how the offset values should be interpreted (as being relative to the left, right, or both sides). These flags offer us about as much refinement as we will get from a row layout. The alignment flag, `cl3`, is not strictly necessary for Maya 2011 and later in this case, as all buttons in newer versions of Maya default to centered text.

Finally, we added three buttons with the proper names, using our specified height, pointing to the proper methods that they should invoke.

```
self.actionBtn = cmds.button(
    label=self.actionName,
    height=self.commonBtnSize[1],
    command=self.actionBtnCmd
);
self.applyBtn = cmds.button(
    label='Apply',
    height=self.commonBtnSize[1],
    command=self.applyBtnCmd
);
self.closeBtn = cmds.button(
    label='Close',
    height=self.commonBtnSize[1],
    command=self.closeBtnCmd
);
```

As was the case with the menus, when we create any layout or control, Maya assumes that the most recently created layout is to be the parent. As such, we do not need to manually specify a parent control using the `parent` flag.

You have obviously noticed that although the buttons are sized correctly and appear to function as expected, they are also not at the correct vertical alignment. (Users of Maya 2010 and earlier will see their buttons at the top of the window, rather than centered.) The problem is that `commonBtnLayout` is an immediate child of `window`, and so it is located in the default position under `window`. Figure 7.6 illustrates the current hierarchy for our GUI window.



■ **FIGURE 7.6** The hierarchy of a window created by the **AR_OptionsWindow** class.

Equally as problematic, the controls do not scale when we adjust the size of the window. Scaling in the width of the window simply clips the buttons on the right side. To fix these problems, we need a more advanced layout.

Form Layouts

Although there are a variety of built-in layout commands that can generate quick results, many of them do not offer a great deal of flexibility. The form layout, accessible via the `formLayout` command, allows for a range of precise positioning options, as well as support for interactively scaling controls. In the next few steps we will retool the **AR_OptionsWindow** class to use a hierarchy of form layouts.

16. Add the following line to the **create()** method, right after the call to the `window` command and before the call to **commonMenu()**.

```
self.mainForm = cmds.formLayout(nd=100);
```

If you evaluate the class definition and create a new **AR_OptionsWindow** instance now, the buttons should be at the top of the window, no matter what version of Maya you are using. The `mainForm` attribute will be the parent for the next layout defined in the window (the row layout), which will be positioned at the top left of the form by default.

The `nd` (or `numberOfDivisions`) flag that we passed to the `formLayout` command allows us to use relative coordinates when specifying attachment positions of controls, as we will see soon. The value of 100 that we passed means that a position of (0, 0) refers to the window's upper left corner, and a position of (100, 100) refers to the window's lower right corner.

17. In the **commonButtons()** method, remove the `commonBtnLayout` assignment (`rowLayout` call), and add the following lines after the assignment of `closeBtn`.

```
cmds.formLayout(
    self.mainForm, e=True,
    attachForm=(
        [self.actionBtn,'left',5],
        [self.actionBtn,'bottom',5],
        [self.applyBtn,'bottom',5],
        [self.closeBtn,'bottom',5],
        [self.closeBtn,'right',5]
    ),
    attachPosition=(
        [self.actionBtn,'right',1,33],
        [self.closeBtn,'left',0,67]
    ),
```

```
        attachControl=(
            [self.applyBtn,'left',4,self.actionBtn],
            [self.applyBtn,'right',4,self.closeBtn]
        ),
        attachNone=(
            [self.actionBtn,'top'],
            [self.applyBtn,'top'],
            [self.closeBtn,'top']
        )
    );
```

As you can see, we call the `formLayout` command in edit mode to configure how the buttons need to be attached in `mainForm`. We pass four different (multiuse) flag arguments to accomplish this task.

Attaching a control specifies how it should scale when the form's dimensions are updated (when the user scales the window).[3] The basic pattern for each flag is that we pass lists that specify a control, an edge on the control, and then the values associated with the edge.

- The `attachForm` flag specifies edges on controls that we want to pin to the bounds of the form we passed to the command. In this case, we pin the action button to the left, the close button to the right, and all the buttons to the bottom of the form. We apply an offset of five pixels on each attachment to pad the buttons in from the edge of the window.
- The `attachPosition` flag specifies edges that we want to pin to points in our relative coordinate grid that we defined using the `nd` flag when we first created `mainForm`. The final numbers in each list, 33 for `actionBtn` and 67 for `closeBtn`, correspond to points approximately one-third and approximately two-thirds of the form's width (33/100 and 67/100, respectively). The penultimate number in each list represents a pixel offset for the attachment.
- The `attachControl` flag specifies edges on a control that we want to attach to another control, with an optional offset. In this case, because we pinned the inner edges of the outermost buttons (`actionBtn` and `closeBtn`) using `attachPosition`, we can attach both the left and right edges of the center button (`applyBtn`) to these outer buttons.

[3]Note that if you scale this window it will save its preferences, and thus subsequent calls to **create()** will use the old dimensions. While we won't go into the details of window preferences here, you can consult the companion web site for more information. In the meantime, you can assign a different handle to your class, or call the `window` command in edit mode after the call to `showWindow` to force its size.

- The `attachNone` flag specifies edges on controls that should not attach to anything, and hence should not scale. Because we specified that the bottom edges of our buttons attached to `mainForm`, setting the top edges as none means they will retain the fixed height specified when we called the `button` command to create them, while the buttons remain pinned to the bottom of the window.

At this point, if you evaluate and then instantiate the **AR_OptionsWindow** class, you can see that you are almost done. We have only a few minor adjustments to make this class ready for other classes to inherit from it.

18. Add a placeholder method to the **AR_OptionsWindow** class for **displayOptions()**. This method will be overridden in child classes to actually display the contents in the main part of the options window.

```
def displayOptions(self): pass
```

19. Add the following lines in the **create()** method right after the call to **commonButtons()** and before the call to the `showWindow` command. These lines add a central pane to the window and then call the **displayOptions()** method you just created.

```
self.optionsBorder = cmds.tabLayout(
    scrollable=True,
    tabsVisible=False,
    height=1
);
cmds.formLayout(
    self.mainForm, e=True,
    attachForm=(
        [self.optionsBorder,'top',0],
        [self.optionsBorder,'left',2],
        [self.optionsBorder,'right',2]
    ),
    attachControl=(
        [self.optionsBorder,'bottom',5,self.applyBtn]
    )
);
self.optionsForm = cmds.formLayout(nd=100);
self.displayOptions();
```

As you can see, we do not simply nest our form layout, `optionsForm`, in `mainForm`. Rather, we nest it in a tab layout, `optionsBorder`, which itself is nested in `mainForm`. Although Autodesk has specific reasons for implementing this pattern, the only effect as far as we need to be concerned is that nesting the form layout in the tab layout displays a nice little border for the

options area in Maya 2011 and later.[4] Now, if you evaluate your changes and create an instance of the **AR_OptionsWindow** class, you should see the result we previewed in Figure 7.4.

At this point, you should save the **AR_OptionsWindow** class into a module, optwin.py, as we will use it in further examples in this chapter. Alternatively, you can download the optwin.py module from the companion web site. For your convenience, we have printed the final contents of the **AR_OptionsWindow** class here for you to compare your results. Note that we have also added a class method, **showUI()**, which provides a shortcut for creating and displaying a new window instance. The module on the companion web site additionally contains comments.

Complete AR_OptionsWindow Class

```
class AR_OptionsWindow(object):
    @classmethod
    def showUI(cls):
        win = cls();
        win.create();
        return win;
    def __init__(self):
        self.window = 'ar_optionsWindow';
        self.title = 'Options Window';
        self.size = (546, 350);
        self.supportsToolAction = False;
        self.actionName = 'Apply and Close';
    def create(self):
        if cmds.window(self.window, exists=True):
            cmds.deleteUI(self.window, window=True);
        self.window = cmds.window(
            self.window,
            title=self.title,
            widthHeight=self.size,
            menuBar=True
        );
        self.mainForm = cmds.formLayout(nd=100);
        self.commonMenu();
        self.commonButtons();
        self.optionsBorder = cmds.tabLayout(
            scrollable=True,
```

[4]Autodesk uses this pattern to reduce flickering when changing from one option window to another. For more information, see the comments in the **createOptionBox()** procedure in getOptionBox.mel. See note 1 for more information.

```
            tabsVisible=False,
            height=1
        );
        cmds.formLayout(
            self.mainForm, e=True,
            attachForm=(
                [self.optionsBorder,'top',0],
                [self.optionsBorder,'left',2],
                [self.optionsBorder,'right',2]
            ),
            attachControl=(
                [self.optionsBorder,'bottom',5,
                self.applyBtn]
            )
        );
        self.optionsForm = cmds.formLayout(nd=100);
        self.displayOptions();
        cmds.showWindow();
    def commonMenu(self):
        self.editMenu = cmds.menu(label='Edit');
        self.editMenuSave = cmds.menuItem(
            label='Save Settings',
            command=self.editMenuSaveCmd
        );
        self.editMenuReset = cmds.menuItem(
            label='Reset Settings',
            command=self.editMenuResetCmd
        );
        self.editMenuDiv = cmds.menuItem(d=True);
        self.editMenuRadio = cmds.radioMenuItemCollection();
        self.editMenuTool = cmds.menuItem(
            label='As Tool',
            radioButton=True,
            enable=self.supportsToolAction,
            command=self.editMenuToolCmd
        );
        self.editMenuAction = cmds.menuItem(
            label='As Action',
            radioButton=True,
            enable=self.supportsToolAction,
            command=self.editMenuActionCmd
        );
        self.helpMenu = cmds.menu(label='Help');
        self.helpMenuItem = cmds.menuItem(
            label='Help on %s'%self.title,
            command=self.helpMenuCmd
        );
```

```
def helpMenuCmd(self, *args):
    cmds.launch(web='http://maya-python.com');
def editMenuSaveCmd(self, *args): pass
def editMenuResetCmd(self, *args): pass
def editMenuToolCmd(self, *args): pass
def editMenuActionCmd(self, *args): pass
def actionBtnCmd(self, *args):
    self.applyBtnCmd();
    self.closeBtnCmd();
def applyBtnCmd(self, *args): pass
def closeBtnCmd(self, *args):
    cmds.deleteUI(self.window, window=True);
def commonButtons(self):
    self.commonBtnSize = ((self.size[0]-18)/3, 26);
    self.actionBtn = cmds.button(
        label=self.actionName,
        height=self.commonBtnSize[1],
        command=self.actionBtnCmd
    );
    self.applyBtn = cmds.button(
        label='Apply',
        height=self.commonBtnSize[1],
        command=self.applyBtnCmd
    );
    self.closeBtn = cmds.button(
        label='Close',
        height=self.commonBtnSize[1],
        command=self.closeBtnCmd
    );
    cmds.formLayout(
        self.mainForm, e=True,
        attachForm=(
            [self.actionBtn,'left',5],
            [self.actionBtn,'bottom',5],
            [self.applyBtn,'bottom',5],
            [self.closeBtn,'bottom',5],
            [self.closeBtn,'right',5]
        ),
        attachPosition=(
            [self.actionBtn,'right',1,33],
            [self.closeBtn,'left',0,67]
        ),
        attachControl=(
            [self.applyBtn,'left',4,self.actionBtn],
            [self.applyBtn,'right',4,self.closeBtn]
        ),
```

```
            attachNone=(
                [self.actionBtn,'top'],
                [self.applyBtn,'top'],
                [self.closeBtn,'top']
            )
        );
    def displayOptions(self): pass
```

EXTENDING GUI CLASSES

Now that you have a complete base class for a basic, familiar-looking GUI window, you can easily extend it for use in a variety of scenarios. Maya contains a great many GUI commands, and so we cannot cover them all here, but we can briefly look at an example that implements some common, useful controls by extending the **AR_OptionsWindow** class.

For the example in this section, we assume that you have saved the **AR_OptionsWindow** class in a module called optwin.py, or that you have downloaded the optwin.py module from the companion web site. We will use **AR_OptionsWindow** as a base class for a simple window with some polygon creation options.

1. Execute the following lines to create a subclass of **AR_OptionsWindow** called **AR_PolyOptionsWindow**. If you are working in the Script Editor, highlight this text before pressing **Ctrl + Enter** to execute it, as we will be working on this class through the next several steps.

```
import maya.cmds as cmds
from optwin import AR_OptionsWindow;
class AR_PolyOptionsWindow(AR_OptionsWindow):
    def __init__(self):
        AR_OptionsWindow.__init__(self);
        self.title = 'Polygon Creation Options';
        self.actionName = 'Create';
AR_PolyOptionsWindow.showUI();
```

Planning has clearly paid off! As you can see, it is a trivial matter to start creating a new window right away by using a class. Calling the base class **__init__()** method and then reassigning some of its attributes produces a window with a new title and a new action button label.

Note that we did not override the window's unique handle, `window`, in **__init__()**. While you certainly could do so if you prefer, we have chosen not to in order to give our window similar behavior to Maya's option windows. Namely, when you instantiate a new window derived from this class, it will first clear other windows derived from the same class (unless you overwrite the `window` attribute in **__init__()**).

Although we will not override all of the empty methods in **AR_OptionsWindow**, we will work through enough to explore a few more GUI commands.

Radio Button Groups

2. Override the **displayOptions()** method by adding the following code to the **AR_PolyOptionsWindow** class. This method adds a radio button selector with four different basic polygon object types.

```
def displayOptions(self):
    self.objType = cmds.radioButtonGrp(
        label='Object Type: ',
        labelArray4=[
            'Cube',
            'Cone',
            'Cylinder',
            'Sphere'
        ],
        numberOfRadioButtons=4,
        select=1
    );
```

Note that we set the default selected item to 1. Radio button groups in the GUI use 1-based, rather than 0-based indices. Consequently, if you evaluate your changes and then call the inherited **showUI()** class method, you will see that the Cube option is selected by default.

3. Override the **applyBtnCmd()** method by adding the following code to the **AR_PolyOptionsWindow** class. This method will execute an appropriate command based on the current selection.

```
def applyBtnCmd(self, *args):
    self.objIndAsCmd = {
        1:cmds.polyCube,
        2:cmds.polyCone,
        3:cmds.polyCylinder,
        4:cmds.polySphere
    };
    objIndex = cmds.radioButtonGrp(
        self.objType, q=True,
        select=True
    );
    newObject = self.objIndAsCmd[objIndex]();
```

As you can see, we create a dictionary to map the radio indices to different function pointers. We then query the `objType` radio button group and use

the dictionary to execute the command that corresponds to the currently selected index.

At this point, you should take a look at the following code, which shows the entire **AR_PolyOptionsWindow** class, to make sure yours looks the same as ours.

```
class AR_PolyOptionsWindow(AR_OptionsWindow):
    def __init__(self):
        AR_OptionsWindow.__init__(self);
        self.title = 'Polygon Creation Options';
        self.actionName = 'Create';
    def displayOptions(self):
        self.objType = cmds.radioButtonGrp(
            label='Object Type: ',
            labelArray4=[
                'Cube',
                'Cone',
                'Cylinder',
                'Sphere'
            ],
            numberOfRadioButtons=4,
            select=1
        );
    def applyBtnCmd(self, *args):
        self.objIndAsCmd = {
            1:cmds.polyCube,
            2:cmds.polyCone,
            3:cmds.polyCylinder,
            4:cmds.polySphere
        };
        objIndex = cmds.radioButtonGrp(
            self.objType, q=True,
            select=True
        );
        newObject = self.objIndAsCmd[objIndex]();
```

If you call the **showUI()** class method, you can use the Create and Apply buttons to create different polygon primitives.

```
AR_PolyOptionsWindow.showUI();
```

Frame Layouts and Float Field Groups

4. Add the following lines to the bottom of the **displayOptions()** method to add a group for entering default transformations.

```
self.xformGrp = cmds.frameLayout(
    label='Transformations',
```

```
        collapsable=True
    );
    cmds.formLayout(
        self.optionsForm, e=True,
        attachControl=(
            [self.xformGrp,'top',2,self.objType]
        ),
        attachForm=(
            [self.xformGrp,'left',0],
            [self.xformGrp,'right',0]
        )
    );
    self.xformCol = cmds.columnLayout();
    self.position = cmds.floatFieldGrp(
        label='Position: ',
        numberOfFields=3
    );
    self.rotation = cmds.floatFieldGrp(
        label='Rotation (XYZ): ',
        numberOfFields=3
    );
    self.scale = cmds.floatFieldGrp(
        label='Scale: ',
        numberOfFields=3,
        value=[1.0,1.0,1.0,1.0]
    );
    cmds.setParent(self.optionsForm);
```

The first command creates a frame layout and stores it in the `xformGrp` attribute. Frame layouts are a type of layout that can collect a series of controls in a group with a label and an optional button to collapse and expand the group. We then issue a call to `formLayout` to attach `xformGrp` to the `objType` selector and to the sides of `optionsForm`. As earlier, issuing a new layout command sets the newly created layout as the default parent for subsequent calls to control commands.

Although Maya 2011 and later would allow us to simply proceed to add controls to `xformGrp`, earlier versions of Maya throw an exception because we would be trying to add too many children to the frame layout. Consequently, to support earlier versions of Maya, we have further nested the controls in a column layout, `xformCol`. The next three controls we add all have the column layout as their parent.

The next three calls are to `floatFieldGrp`, which creates a group of floating-point number input fields. We have created a group for each of the basic transformations: translation, rotation, and scale. Note that when we

initialize the value for the `scale` field we must pass it four values, since the command does not know by default if it will consist of one, two, three, or four fields.

Finally, we issue a call to the `setParent` command to make `optionsForm` the current parent again. Without making this call, subsequent controls would be grouped under the `xformCol` column layout, as opposed to being children of the `optionsForm` layout. You may find this pattern more convenient than manually specifying the `parent` argument for each new control or layout you create.

5. Now that you have added your new controls, add the following lines to the **applyBtnCmd()** method to apply the transformations to the newly created object's **transform** node.

```
pos = cmds.floatFieldGrp(
    self.position, q=True,
    value=True
);
rot = cmds.floatFieldGrp(
    self.rotation, q=True,
    value=True
);
scale = cmds.floatFieldGrp(
    self.scale, q=True,
    value=True
);
cmds.xform(newObject[0], t=pos, ro=rot, s=scale);
```

If you call the **showUI()** class method again, you can now enter default translation, rotation, and scale values for the primitive you create.

Color Pickers

We will wrap up this example by adding a color picker to assign default vertex colors to the object.

6. Add the following lines to the end of the **displayOptions()** method to add a color picker control.

```
self.color = cmds.colorSliderGrp(
    label='Vertex Colors: '
);
cmds.formLayout(
    self.optionsForm, e=True,
    attachControl=(
        [self.color,'top',0,self.xformGrp]
    ),
```

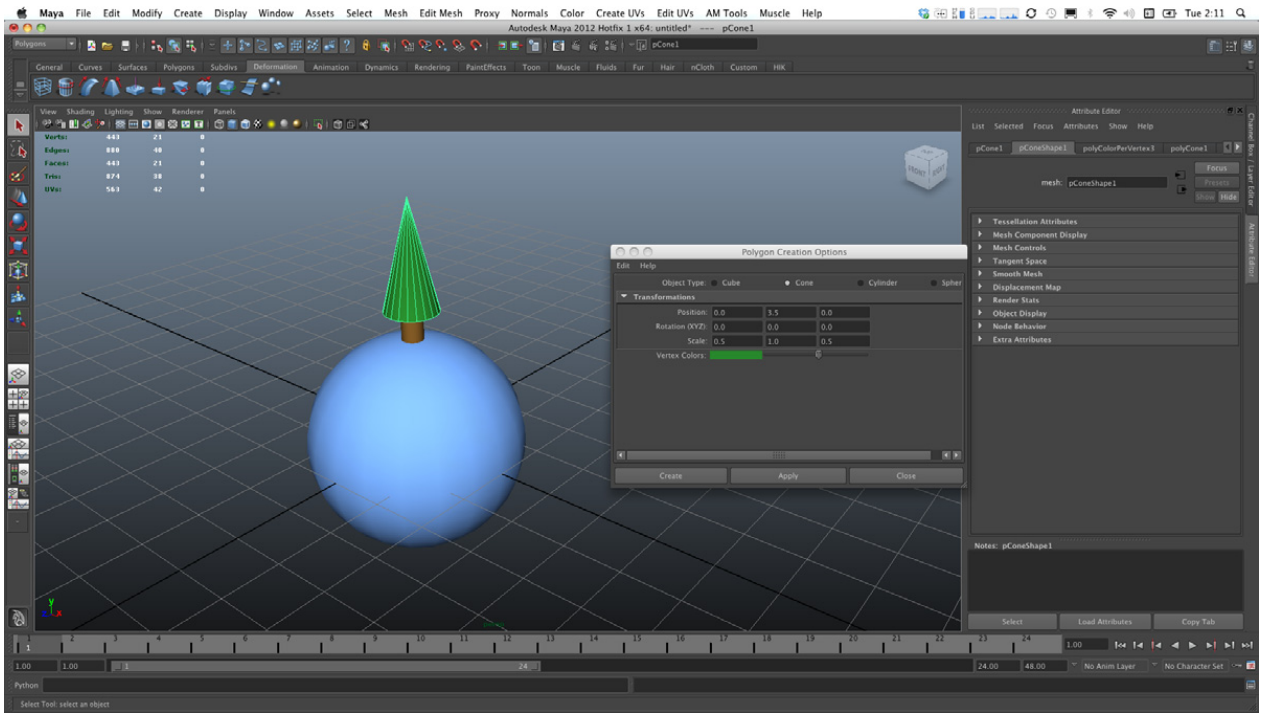
```
        attachForm=(
            [self.color,'left',0],
        )
    );
```

The `colorSliderGrp` command creates a color picker with a luminance slider next to it.

7. Add the following lines to the end of the **applyBtnCmd()** method, after the call to the `xform` command, to apply the selected color to the new model's vertices.

```
col = cmds.colorSliderGrp(
    self.color, q=True,
    rgbValue=True
);
cmds.polyColorPerVertex(
    newObject[0],
    colorRGB=col,
    colorDisplayOption=True
);
```

8. Call the **showUI()** class method again and you should see an example like that shown in Figure 7.7. If you are having difficulties, consult



■ **FIGURE 7.7** The AR_PolyOptionsWindow class in action.

the complete example in the polywin.py module available on the companion web site.

While we could continue augmenting this tool with a variety of interesting controls, there are far too many in the cmds module to cover here! This example has hopefully given you enough exposure to the basics that you should be comfortable exploring a variety of other common GUI controls as you extend the **AR_OptionsWindow** class for your own tools.

CREATING MORE ADVANCED TOOLS

Before concluding this chapter, we want to touch upon a few more topics to help you develop more complex tools on your own. While these topics are not strictly related to Maya's GUI commands, you will no doubt find them helpful in many custom tools you develop. In this section, we examine the posemgr.py module from the companion web site to explore tool structure, data serialization, and working with files.

Pose Manager Window

The posemgr.py module contains a simple tool that enables you to copy and paste a character's pose, as well as save poses to a file. Although the window in this module is very basic to simplify the points we want to make, we have designed the module in a way that you could easily extend it if you wanted to use it in production.

1. Download the posemgr1.ma file from the companion web site and open it. The file contains a basic character with some animation applied. Play the animation to see how it loops.
2. Download the posemgr.py module from the companion web site and save it somewhere in your Python search path.
3. Execute the following lines in Maya to launch the Pose Manager GUI.

```
from posemgr import AR_PoseManagerWindow;
AR_PoseManagerWindow.showUI();
```

   You should see a very barebones GUI window like that shown in Figure 7.8. The window contains a control group for copying and pasting poses, as well as a control group for saving and loading poses.
4. Move the time slider to the first frame, select the character's pelvis joint (CenterRoot), and press the Copy Pose button in the Pose Manager window. The Pose Manager tool works by copying the values for all of the keyable attributes on **transform** nodes in the hierarchy under

■ **FIGURE 7.8** The Pose Manager window contains a few simple controls.

the nodes you have selected. In this case, it is copying the transformation values for all of the joints in the character's hierarchy.

5. Scroll to the final frame on the time slider and then press the Paste Pose button. The character should now be in the same pose it was in when at the start of the animation.
6. With the CenterRoot joint still selected, select the character's whole hierarchy using the **Edit → Select Hierarchy** menu option and set a key on all of its joints (**Animate → Set Key** in the Animation menu set or default hotkey **S**).

If you play the animation now, the start and end frames should match. While this application of the pose manager is handy, it is not especially revolutionary.

7. Scrub to the first frame of the animation, select the CenterRoot joint, and then press the Save Pose button in the Pose Manager window. Save the pose as idle.pse in a location you will remember.
8. Open the file posemgr2.ma and view its animation. You should see the same character with a different motion applied. Unfortunately, its initial pose doesn't quite match the other animation well enough to produce a good blend for a game.
9. Scrub the time slider to the first frame in the animation.
10. Press the Load Pose button in the Pose Manager to pull up a file dialog. Browse to the idle.pse file that you just saved and open it.[5] The character's pose should now match the one you saved.

[5]If you cannot find the file you just saved, and are running Maya 2008 or earlier on OS X, try browsing one folder up your directory tree. See note 7 for more information.

11. Select the character's hierarchy again and set a key as in step 6. The character's motion now begins in the same pose as the other animation, and would be able to produce a much better blend if they were dropped into a game together.

While this tool does not appear to be overly glamorous, it does make use of some important techniques worth discussing. Open the posemgr.py file and examine its source code. The file contains some imports, an attribute, a class, and some assorted functions. While you are free to examine the module's contents in greater detail on your own, we highlight some key points here.

Separating Form and Function

The first point worth discussing is how the module is organized overall. Our previous examples separated form and functionality by implementing different methods in a single class: Each button had a command method associated with it, which could be easily overridden in a child class. This pattern was sufficient for the **AR_PolyOptionsWindow** class, as its actions were not likely to be of interest to other tools.

In comparison, the Pose Manager window incorporates some functionality that may be of interest to other tools or other classes. Consequently, we have defined some attributes in the module outside of the **AR_PoseManagerWindow** class. These include the **kPoseFileExtension** variable (a three-letter file extension for our custom pose files), as well as some functions for doing the core work of the tool, including **exportPose()**, **saveHierarchy()**, and **importPose()**. By making these items available without an **AR_PoseManagerWindow** instance, we would enable other developers to hook into them in other GUIs or even when running in headless mode via mayapy, for example.

Serializing Data with the cPickle Module

The **exportPose()** function takes two arguments: a path to a file and a collection with the names of root nodes.

```
def exportPose(filePath, rootNodes):
    try: f = open(filePath, 'w');
    except:
        cmds.confirmDialog(t='Error', b=['OK'],
            m='Unable to write file: %s'%filePath
        );
        raise;
    data = saveHiearchy(rootNodes, []);
    cPickle.dump(data, f);
    f.close();
```

The first step in this function is to create a file object, `f`, using Python's built-in **open()** function. This function allows you to specify a path to a file and a flag to indicate if you would like read or write permissions for the file. When working in write mode, if no file exists at the specified path, Python will automatically create one. The return value is a built-in type, file, which contains a number of useful methods. We recommend consulting Section 5.9 of Python Standard Library online for more information on working with file objects.

If there is a problem, we create a notification dialog to inform users of the error before raising an exception. An artist who uses a tool that invokes this method won't have to hunt around the console, but a developer using Maya in headless mode would still get an exception and a stack trace.

We then call the **saveHierarchy()** function (which we will examine in a moment), and use the cPickle module to serialize the data it returns. The cPickle module is a built-in module that allows you to serialize and deserialize Python objects effortlessly. Python also includes an analogous built-in module, pickle, which is almost identical. The difference between the two modules is that cPickle is written in C, and so can work up to a thousand times faster than the standard pickle module. The only key downside is that pickle implements two classes (**Pickler** and **Unpickler**) that can be overridden, while cPickle implements them as functions. Because we do not need to override their functionality, we can use the cPickle module here and harness its greater speed.

The most common usage of these modules is to call the **dump()** function (as we do here) to write serialized data to a file, and the **load()** function to read it back in and deserialize it. Because we cannot explore them in greater depth here, you should consult Sections 11.1 and 11.2 of Python Standard Library for more information on the pickle and cPickle modules, as you will find them invaluable in a number of situations.

After writing the file's contents with the **dump()** function, we close the file. Remember that you must always close files when you are done with them![6]

The **saveHierarchy()** function creates the data that we dump into the pose file. As you can see, it has two parameters: `rootNodes` and `data`.

[6]Python 2.5 added special syntax in the form of the **with** keyword, which allows you to open a file and operate on it in a block, after which it is automatically closed. Consult Section 7.5 of Python Language Reference online for more information on its features and usage.

```
def saveHiearchy(rootNodes, data):
    for node in rootNodes:
        nodeType = cmds.nodeType(node);
        if not (nodeType=='transform' or
            nodeType=='joint'): continue;
        keyableAttrs = cmds.listAttr(node, keyable=True);
        if keyableAttrs is not None:
            for attr in keyableAttrs:
                data.append(['%s.%s'%(node,attr),
                    cmds.getAttr('%s.%s'%(node,attr))]
                );
        children = cmds.listRelatives(node, children=True);
        if children is not None: saveHiearchy(children,
data);
    return data;
```

This function is recursive, meaning it calls itself in its body. Recall that when we invoke this function from **exportPose()**, we pass it an empty list for the `data` parameter.

```
data = saveHiearchy(rootNodes, []);
```

The **saveHierachy()** function iterates through all of the **transform** nodes passed in the `rootNodes` argument, skipping over objects that are not **transform** or **joint** nodes. (You could support any object types you like, but we keep it simple here.) For each valid node, it gathers all of the object's keyable attributes and then appends an [object.attribute, value] pair to the `data` list. Once all of these pairs have been appended, it sees if the current node has any children.

If there are any children, the function calls itself, passing in the `children` list as well as the current state of the `data` list. Because lists are mutable, this recursive invocation does not need to capture the result in a return value, as `data` will be mutated in-place. Once the iteration has finished, the function returns. When the function has returned out to the original invocation, where it was called from the **exportPose()** context, and it has completed the last node, the `data` list is returned and then assigned to a variable in **exportPose()**.

When we read a pose back in, we use the **importPose()** function, which takes a path to a pose file as its argument.

```
def importPose(filePath):
    try: f = open(filePath, 'r');
    except:
        cmds.confirmDialog(t='Error', b=['OK'],
            m='Unable to open file: %s'%filePath
        );
        raise;
```

```
        pose = cPickle.load(f);
        f.close();
        errAttrs = [];
        for attrValue in pose:
            try: cmds.setAttr(attrValue[0], attrValue[1]);
            except:
                try: errAttrs.append(attrValue[0]);
                except: errAttrs.append(attrValue);
        if len(errAttrs) > 0:
            importErrorWindow(errAttrs);
            sys.stderr.write(
                'Not all attributes could be loaded.'
            );
```

This function is very straightforward. It begins by trying to open the supplied file in read mode, again displaying a dialog box if there is some kind of problem. If the file opens successfully, we then call the **load()** function in the cPickle module to deserialize the file's contents into the `pose` list and close the file.

The function then iterates through the items in the `pose` list. It tries to set the specified attribute to the specified value in the deserialized [object.attribute, value] pairs. If the scene does not contain the node, or cannot set the specified attribute for some other reason, then it appends the attribute name to a list of error attributes (`errAttrs`) to be able to inform the user of which attributes were by necessity ignored. Because the **importErrorWindow()** function does not introduce anything novel, we leave it to interested readers to inspect its contents in the source code if so desired.

Working with File Dialogs

Although we separated out the Pose Manager's important functions from the window class to enable other tools to hook into them, we needed to implement a mechanism in our sample GUI to provide file paths when invoking these functions. In the **AR_PoseManagerWindow** class you can find four methods—**copyBtnCmd()**, **pasteBtnCmd()**, **saveBtnCmd()**, and **loadBtnCmd()**—which correspond to the Copy Pose, Paste Pose, Save Pose, and Load Pose buttons, respectively.

The **copyBtnCmd()** and **pasteBtnCmd()** methods are fairly straightforward, and so we do not discuss their contents in detail here. Though they implement some minor validation and user feedback, which you can view in the source code, the important point is that they call the **exportPose()** and **importPose()** functions, respectively, passing a `tempFile` attribute as the `filePath` parameter. This attribute is created in the **__init__()** method.

```
self.tempFile = os.path.join(
    os.path.expanduser('~'),
    'temp_pose.%s'%kPoseFileExtension
);
```

This attribute simply initializes a string to provide a path to a file, temp_pose.pse, in the user's home folder, since this location is guaranteed to be writable. Using this technique allows us to easily invoke the same functions as the other button command methods.

The **saveBtnCmd()** and **loadBtnCmd()** methods obtain a file path using a file dialog, and then pass this value to the **exportPose()** and **importPose()** functions, respectively. Although these methods do not merit a detailed discussion, they implement a similar pattern worth noting. First examine the **saveBtnCmd()** method.

```
def saveBtnCmd(self, *args):
    rootNodes = self.getSelection();
    if rootNodes is None: return;
    filePath = '';
    try: filePath = cmds.fileDialog2(
        ff=self.fileFilter, fileMode=0
    );
    except: filePath = cmds.fileDialog(
        dm='*.%s'%kPoseFileExtension, mode=1
    );
    if filePath is None or len(filePath) < 1: return;
    if isinstance(filePath, list): filePath = filePath[0];
    exportPose(
        filePath,
        cmds.ls(sl=True, type='transform')
    );
```

The method begins with some simple selection validation, and then proceeds to a **try-except** clause. Though not strictly necessary, we implement this pattern to support all versions of Maya with Python support. Maya 2011 added the `fileDialog2` command, so we prefer it if it is available. If it does not exist, however, we fall back on the `fileDialog` command, which is available in earlier versions of Maya, to retrieve `filePath`.

The next lines perform some validation on the results retrieved from the file dialog. We can return early if `filePath` is None or has length 0 (e.g., if the user canceled or dismissed the file dialog). The next test, to see if `filePath` is a list, is again just to support all versions of Maya. While `fileDialog` simply returns a string, the `fileDialog2` command returns a list of paths, as it supports the ability to allow users to select

multiple files. Thereafter, the method can simply call the **exportPose()** function using the supplied path. The **loadBtnCmd()** method implements a similar process.

```
def loadBtnCmd(self, *args):
    filePath = '';
    try: filePath = cmds.fileDialog2(
        ff=self.fileFilter, fileMode=1
    );
    except: filePath = cmds.fileDialog(
        dm='*.%s'%kPoseFileExtension, mode=0
    );
    if filePath is None or len(filePath) < 1: return;
    if isinstance(filePath, list): filePath = filePath[0]
    importPose(filePath);
```

Apart from not requiring selection validation (or selection at all), the only major difference in this method concerns the argument values that are passed to the `fileDialog2` and `fileDialog` commands.

The `mode` flag on the `fileDialog` command specifies whether the dialog is in read or write mode. While we specified a value of 0 here for read, the **saveBtnCmd()** method passed a value of 1 for write.[7]

On the other hand, the `fileDialog2` command implements a more flexible `fileMode` flag. Although the arguments we passed are the opposite from those we passed to the `fileDialog` command's `mode` flag (we pass 1 when loading, and 0 when saving), the `fileMode` flag has a very different meaning from the `mode` flag on the `fileDialog` command. The basic idea is that file dialogs don't really need to know whether you plan to read or write: you handle all of that functionality separately. *Rather, it needs to know what you want to allow the user to be able to see and select*. This flag allows you to specify what the file dialog will display, and what the command can return. Although its different argument values are covered in the Python Command Reference, we summarize the different argument values for this flag in Table 7.1.

[7]In Maya 2008 and earlier versions, on some versions of OS X, using `fileDialog` in write mode (`mode=1`) may return a string value with no path separator between the directory and file names. For example, instead of returning "/users/adammechtley/Desktop/untitled.pse", the command would return "/users/adammechtley/Desktopuntitled.pse". If you are running this example on such a version of Maya and are having difficulty finding poses you save, then try looking one folder up from where you expect it to be.

Table 7.1 Possible Argument Values and Their Meanings for the `fileMode` Flag Used with the `fileDialog2` Command

Argument Value	Meaning
0	Any file, whether it exists or not (e.g., when saving a new, nonexistent file)
1	A single existing file (e.g., when loading a file)
2	A directory name (both directories and files appear in the dialog)
3	A directory name (only directories appear in the dialog)
4	The names of one or more existing files (e.g., when loading several files)

CONCLUDING REMARKS

In addition to having mastered some basic GUI commands, you have another hands-on example of some of the benefits of object-oriented programming in Maya using Python. By creating basic classes for common tool GUIs, you can introduce new tools into your studio's suite much more easily than if you were restricted to MEL or if you were manually creating new windows for every new tool. Moreover, you have learned best practices for executing commands with your GUI controls, organizing your GUI windows, and architecting your GUI code in a larger code base. Finally, you have been introduced to the basics of data serialization and working with files in Python. You now have all the foundations necessary to create a range of useful tools, though Python still has much more to offer!