Chapter 8

Advanced Graphical User Interfaces with Qt

CHAPTER OUTLINE

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

- Describe what Qt is and how it is integrated into modern versions of Maya.
- Dock built-in and custom windows in the Maya GUI.
- Locate, download, and install Qt libraries and tools.
- Define widgets, signals, and slots.
- Identify Qt widgets that correspond to native Maya GUI commands.
- Create a basic user interface with Qt Designer.

Maya Python for Games and Film. DOI: 10.1016/B978-0-12-378578-7.00008-9

- Add Maya functionality to controls in a Qt user interface.
- Load a Qt interface in Maya 2011 and later.
- Use signals and slots to implement advanced GUI controls.
- Describe how PyQt works and what it enables you to do.
- Locate helpful resources for working with Qt.

In Chapter 7, we examined Maya's built-in tools for creating basic windows using the cmds module. While the cmds module provides a very natural and native mechanism for creating GUIs in Maya, you can probably imagine that creating more involved windows, controls, and layouts using commands could be an arduous programming task. Fortunately, Maya 2011 and later have introduced an alternative user interface approach that not only provides more options, but also offers a set of tools that can substantially expedite the creation of a custom GUI. Specifically, Maya 2011 and later have fully integrated the Qt graphics libraries.

In this chapter, we briefly examine what Qt is and how it has been integrated into Maya. We then discuss how you can install Qt libraries and tools. Next, we discuss the basics of using Qt Designer to rapidly create GUIs and use them in Maya. Finally, we provide a brief overview of PyQt.

Qt AND MAYA

To facilitate GUI flexibility and customization, Maya 2011 and later incorporate a graphics library called Qt (pronounced "cute") as the GUI toolkit. This third-party technology is not developed or owned by Autodesk, but is nevertheless tightly integrated into Maya.

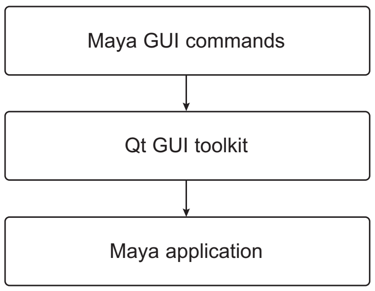
The main benefit of the Qt toolkit is that it is a cross-platform C++ application framework. Recall that Maya ships on three platforms: Windows, OS X, and Linux. As you can imagine, it requires a lot of resources to maintain three separate GUI toolkits across these platforms. In addition to its platform diversity, Qt excels at creating compact, high-performance tools.

The philosophy around Qt is to focus on innovative coding, not infrastructure coding. The idea is that you should spend most of your time making an elegant window, rather than trying to place one button in the middle of the interface. Qt's tools have been designed to facilitate this workflow, which makes it an interesting alternative to using native Maya commands.

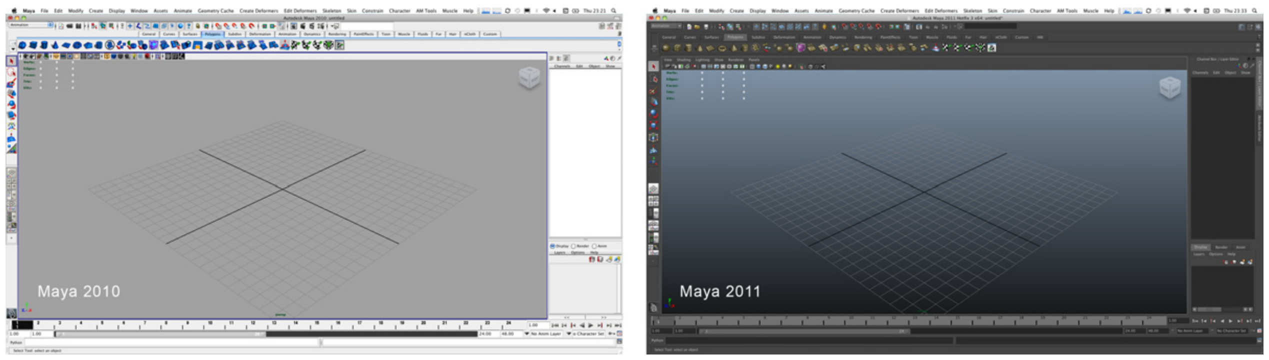
Don't worry! Maya's user interface commands are not being phased out. All of your existing GUIs created with Maya commands will still work. GUI creation with Qt is purely an addition to the Maya environment, and no functionality has been removed or lost. To help understand how Qt fits into modern versions of Maya, refer to Figure 8.1.

Imagine the Qt GUI toolkit as an abstraction layer sitting on top of the Maya application. This illustration closely resembles that shown in Chapter 7. The key point, however, is that all target platforms can now make use of the same GUI toolkit. Prior to Maya 2011, the GUI layer was platform-specific (e.g., Carbon on OS X, Motif on Linux). Now, because all platforms use the same GUI library, there is much greater consistency across operating systems. On top of the Qt layer, the native Maya GUI commands remain unchanged. Behind the scenes, Autodesk has connected the existing GUI commands to the new Qt layer, which allows you to seamlessly transition into Maya 2011 and later.

For Maya 2011 and later, the standard Maya user interface is still built using Maya GUI commands. However, now that the Maya GUI commands are driven by Qt in the background, you will see that there is much more functionality available when customizing the Maya interface (Figure 8.2).



■ **FIGURE 8.1** The Qt GUI toolkit exists as another abstraction layer outside the Maya application, yet can work on all platforms that Maya targets.



■ **FIGURE 8.2** The Maya 2011 GUI is a big step up from the Maya 2010 GUI.

In addition to featuring a striking dark color scheme, Maya's Qt-based GUI permits much better window organization and customization by enabling features like docking, which allows you to insert windows adjacent to other items in the main GUI window. Another helpful feature is the inclusion of drag-and-drop functionality when adding and moving windows around in your layout.

Docking Windows

The easiest way to discover the flexibility of Qt is to take a look at an example that demonstrates some of the new functionality. If you are using Maya 2010 or earlier, you will unfortunately just have to use your imagination. Let's first examine the `dockControl` command, new in Maya 2011, and use it to dock a Graph Editor into the current interface layout.

1. Open the Script Editor and execute the following short script in a Python tab.

```
import maya.cmds as cmds;
layout = cmds.paneLayout(configuration='single');
dock = cmds.dockControl(
    label='Docked Graph Editor',
    height=200,
    allowedArea='all',
    area='bottom',
    floating=False,
    content=layout
);
ge = cmds.scriptedPanel(
    type='graphEditor',
    parent='layout'
);
```

The previous script created a pane layout, `layout`, for the Graph Editor to sit in and then created a dock control, `dock`, to house the pane layout at the bottom of the Maya interface (using its `content` flag). It finally created an instance of the Graph Editor and made it a child of `layout`.

Similarly, you can use the `dockControl` command to dock your own custom windows created using the cmds module. For instance, you can use it to dock the Pose Manager window we discussed in Chapter 7.

2. Ensure that you have downloaded the posemgr.py module from the companion web site and have saved it somewhere in your Python search path.
3. Open the Script Editor and enter the following script in a Python tab.

```
import maya.cmds as cmds;
import posemgr
```

```
pwin = posemgr.AR_PoseManagerWindow();
pwin.create();
cmds.toggleWindowVisibility(pwin.window);
layout = cmds.paneLayout(configuration='single');
dock = cmds.dockControl(
    label=pwin.title,
    width=pwin.size[0],
    allowedArea='all',
    area='right',
    floating=False,
    content=layout
);
cmds.control(pwin.window, e=True, p=layout);
```

As you can see, the process is very similar to that when docking built-in windows. In this case, right after we create the window, we toggle its visibility off, since the **create()** method shows the window by default. Without this toggle, the window would display both floating in the Maya UI and docked on the right side, as the `dockControl` command automatically displays its contents. Note also that closing a dock control does not delete the dock, but simply hides it. You can access closed dock controls by clicking the **RMB** over any dock's title.

INSTALLING Qt TOOLS

The native Maya GUI commands that we introduced in Chapter 7 are sufficient for most users. However, you may find you require more access to the underlying Qt toolkit. Maya does not ship with all of the Qt libraries, but rather only those it requires to run. To work with the additional Qt header files and libraries, you must download the correct source files, and then install and build them. Keep in mind the additional Qt toolkit is optional, and is aimed at developers who wish to create more demanding interfaces through such tools as PyQt and PySide.

The Qt SDK

In this section we will talk briefly about how to obtain and install the Qt Software Development Kit (SDK). The Qt SDK contains additional libraries and development tools, one of which is Qt Designer. Qt Designer is a visual GUI editing application, which allows you to drag and drop controls into a canvas as opposed to manually coding them. *Although Qt Designer is part of the SDK, you do not need to install the SDK to obtain it.* If you are only interested in using Qt Designer, you can skip to the next section of this chapter, as we discuss an alternative method to install it

Table 8.1 Links for the Qt SDK for Maya 2011 and 2012

Maya Version	Operating System	Link
2011	Windows	*ftp://ftp.qt.nokia.com/qt/source/qt-win-opensource-4.5.3-mingw.exe*
2011	OS X	*ftp://ftp.qt.nokia.com/qt/source/qt-mac-cocoa-opensource-4.5.3.dmg*
2011	Linux	*ftp://ftp.qt.nokia.com/qt/source/qt-all-opensource-src-4.5.3.tar.gz*
2012	Windows	*http://get.qt.nokia.com/qtsdk/qt-sdk-win-opensource-2010.05.exe*
2012	OS X	*http://get.qt.nokia.com/qtsdk/qt-sdk-mac-opensource-2010.05.dmg*
2012	Linux	*http://get.qt.nokia.com/qtsdk/qt-sdk-linux-x86_64-opensource-2010.05.bin*

without all of the other tools and libraries, many of which you may never use. If you are still interested, however, read on!

Each version of Maya is built with a specific version of Qt. Before downloading any files, you must first examine the Autodesk Maya API Guide and Reference documentation, and determine which version of Qt you need to download. You can find this information in Maya's help on the page **API Guide → Setting up your build environment → Configuring Qt for use in Maya plug-ins**. For example, Maya 2011 uses version 4.5.3, while Maya 2012 uses 4.7.1. The Qt version is likely to change with every major release of Maya, so it is your responsibility to always check when setting up your Qt environment what version the new Maya release uses.

Once you have verified the Qt version required for your version of Maya, navigate to the Official Qt SDK download site at *http://get.qt.nokia.com/qtsdk/* where you can locate and download the required source package for your version.[1] Table 8.1 lists example links for each platform for Maya 2011 and 2012.

Once you have successfully downloaded the required Qt source package, you may now install it by double clicking the .exe file on Windows or the .dmg file on OS X. For Windows and OS X operating systems, we recommend that you follow the installation wizard using default installation settings. On Linux there is no installation wizard—you are required to

[1]In addition to offering several version options, there are two available license options: the LGPL (Lesser General Public License) free software license and the GPL (General Public License) commercial software license. Determine which license type best suits your needs by consulting the licensing terms and conditions.

configure the source packages yourself. For more detailed installation instructions and steps, consult the docs/html folder that is part of the Qt SDK download. Look for the appropriate Installing Qt page based on your operating system.

After a successful installation, you now have access to a range of Qt tools:

- *Qt Designer* is an application for designing and building GUIs from Qt components.
- *Qt Linguist* is an application to translate applications into local languages.
- *Qt Creator* is a cross-platform integrated development environment (IDE).
- *Qt Assistant* is a help browser featuring comprehensive documentation to work with the Qt tools.
- Other tools and examples such as plugins, frameworks, libraries, and Qt Command Prompt.

Qt DESIGNER

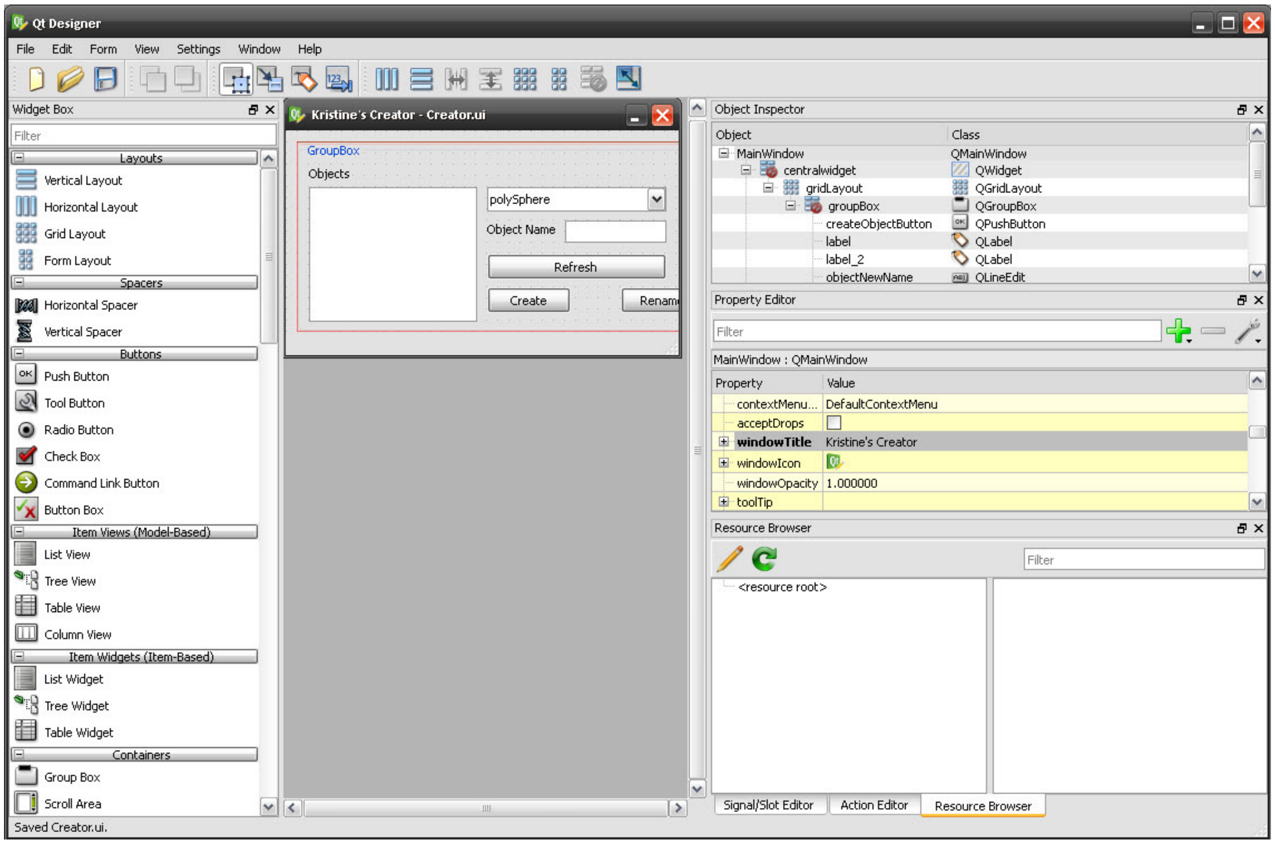
Thus far we have designed our GUIs by using Maya commands. An alternative approach available to modern versions of Maya is to use Qt Designer (Figure 8.3). Qt Designer is an application for designing and building GUIs from Qt components. You can compose and customize your GUIs in a visual drag-and-drop environment. Qt Designer is referred to as a WYSIWYG (what-you-see-is-what-you-get) tool.

In Maya 2011 and later, your WYSIWYG GUIs can be imported directly into Maya, allowing for easy interface building. Moreover, you can embed Maya commands into your GUI by creating dynamic properties, which provide you with an additional level of interface control and complexity. In addition to enabling rapid prototyping to ensure your designs are usable, Qt Designer also facilitates easy layout changes without altering the underlying code of your GUI.

With Qt Designer you can create three types of forms: main windows, custom widgets, and dialogs.

- *Main windows* are generally used to create a window with menu bars, dockable widgets, toolbars, and status bars.
- *Custom widgets* are similar to a main window, except that they are not embedded in a parent widget and thus do not provide the various toolbars as part of the window.
- *Dialogs* are used to provide users with an options window.

Main windows and custom widgets are the most commonly used forms within Maya. Dialogs tend not to be used in Maya because there is no



Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
gridLayout	QGridLayout
groupBox	QGroupBox
createObjectButton	QPushButton
label	QLabel
label_2	QLabel
objectNewName	QLineEdit

MainWindow : QMainWindow

Property	Value
contextMenu...	DefaultContextMenu
acceptDrops	☐
windowTitle	Kristine's Creator
windowIcon	
windowOpacity	1.000000
toolTip	

■ **FIGURE 8.3** The Qt Designer application.

equivalent Maya command for the button box that is provided with the default dialog templates. However, you could easily emulate the dialog functionality with a custom widget using two buttons.

Widgets

All the components that are used within forms are called widgets. A *widget* is any type of control in the interface, such as a button, label, or even the window itself. To be able to query a Qt widget inside of Maya, Maya must have an equivalent representation of the widget available in the Maya commands. Unfortunately, not all widgets in Qt Designer are available in Maya. To access the list of available widget types and their associated Maya commands, use the `listTypes` flag on the `loadUI` command.

```
import maya.cmds as cmds;
for t in cmds.loadUI(listTypes=True):
    print(t);
```

If you print the items in the list, you will see something like the following results.

```
CharacterizationTool:characterizationToolUICmd
QCheckBox:checkBox
QComboBox:optionMenu
QDialog:window
QLabel:text
QLineEdit:textField
QListWidget:textScrollList
QMainWindow:window
QMenu:menu
QProgressBar:progressBar
QPushButton:button
QRadioButton:radioButton
QSlider:intSlider
QTextEdit:scrollField
QWidget:control
TopLevelQWidget:window
```

If you require a widget that is not available, you must manually map the Qt widget to something that is accessible to Maya. Qt provides a built-in mapping mechanism referred to as signals and slots.

Signals and Slots

Signals and slots are central to Qt and help it stand out from other GUI toolkits. *Signals* and *slots* are used to allow objects to communicate with each other and to easily assign behaviors to widgets. In short, GUI events emit certain signals, which are linked to different functions (slots). In essence, the mechanism is a more secure alternative to simply passing function pointers as we do when using Maya commands. There are great resources on the Web to ensure you understand this mechanism, so it is certainly worth referring to the Qt documentation. We will later examine an example demonstrating the use of signals and slots to enable controls that are not automatically supported.

Qt Designer Hands On

To discuss some of the theory behind GUI design and the workflows that are involved in creating a window in Qt Designer that will interact with Maya, it is worth walking through a brief, hands-on example. Note that if you elected not to install the entire Qt SDK, you will need to download and install the Qt Creator application from the Official Qt web site (*http://qt.nokia.com/products*). Qt Creator is an IDE that embeds the Qt Designer application, among other tools.

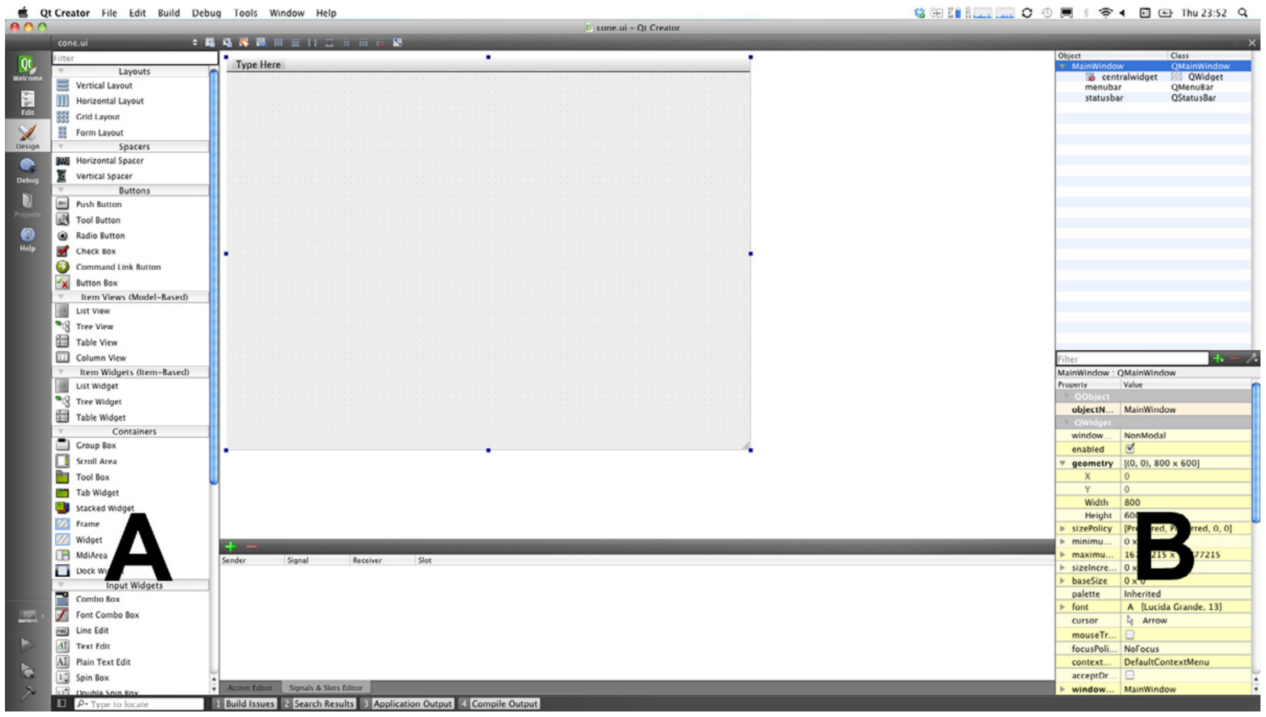
In this section, we'll work through a basic GUI to create a cone that points in a direction specified using a control in the window.

1. Open Qt Designer (or Qt Creator if you are using it). You can find it at one of the following locations based on your operating system:
   - **Windows:** Start → Programs → Qt SDK → Tools → …
   - **OS X:** /Developers/Applications/Qt/…
   - **Linux:** Start → Programming → … or Start → Development → …
2. If you are using Qt Creator, select **File → New File or Project** from the main menu. In the menu that appears, select **Qt** from the Files and Classes group on the left, and **Qt Designer Form** in the group on the right. Press the **Choose** button.
3. Whichever application you are using, you should now see a dialog where you can select what type of widget you would like to create. (If using Qt Designer, you should have seen a default pop-up menu when you launched the application. If you did not, you can simply navigate to **File → New**.) Select the **Main Window** widget from this dialog and **Default Size** from the Screen Size dropdown menu and proceed.
4. If you are using Qt Creator, you must now specify a file name and location. Name the file cone.ui and set its location to your home directory. Skip past the next page in the project wizard, which allows you to specify if you want to use version control.

Whichever application you are using, you should now be in the main application interface, which is very similar in both cases. Although we will not go into all of its details, we will cover what you need to start creating a custom Maya GUI. The main interface is roughly divided into thirds (Figure 8.4). The drag-and-drop widgets/controls are on the left (A), the work area is in the middle, and the editors section is on the right. The Property Editor (B) on the right is one of the most important panes.

As you start designing a custom GUI in the following steps, it may be helpful to think of the main window as a canvas, like those found in two-dimensional image editing packages. Let's start by adding some properties to the main window.

The Property Editor is dynamic and changes based on whatever object is selected in the work area. To ensure you always have the object you want selected, it helps to use the Object Inspector window in the upper right corner. If you are working in Qt Designer and cannot see the Object Inspector window, it may be occluded by another window. Toggle its visibility from the **View** menu so you can see it.



Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
menubar	QMenuBar
statusbar	QStatusBar

MainWindow : QMainWindow

Property	Value
QObject	
objectN...	MainWindow
QWidget	
window...	NonModal
enabled	✓
geometry	[(0, 0), 800 x 600]
X	0
Y	0
Width	800
Height	600
sizePolicy	[Preferred, Preferred, 0, 0]
minimu...	0 x
maximu...	16 215 x 77215
sizeIncre...	0 x
baseSize	0 x 0
palette	Inherited
font	A [Lucida Grande, 13]
cursor	Arrow
mouseTr...	☐
focusPoli...	NoFocus
context...	DefaultContextMenu
acceptDr...	☐
window...	MainWindow

■ **FIGURE 8.4** The Qt Creator interface.

5. Select the `MainWindow` object from the Object Inspector window. You should see information for the `MainWindow` object in the Property Editor.

The Property Editor displays all of the properties associated with the currently selected object. Though it contains many properties, we will only discuss some basics. The first property in the list is called **objectName**. This property contains the name that will be given to your window in Maya, if you need to access it via commands, for instance.

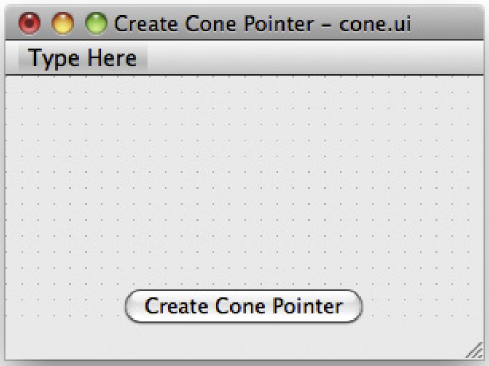
6. Set the value for **objectName** to ar_conePtrWindow.
7. Scroll down in the Property Editor list to find the property **windowTitle**. Set its value to Create Cone Pointer.
8. Adjust the size of the main window to your liking by clicking on the bottom right corner of the window and dragging it in and out. Note that the **geometry** property updates in the Property Editor in real time. You can also expand this property and manually enter a height and width to your liking—a value like 300 × 200 is appropriate for this example.

9. Locate the Widget Box on the left side of the interface. This box allows you to create all of the basic types of widgets built in to Qt. Add a button to the window by dragging the **Push Button** item from the Buttons group into your main window in the canvas (Figure 8.5).
10. Change the label appearing on the button by double clicking on it. Set the text to Create Cone Pointer. Note that the **text** property in the **QAbstractButton** section of the Property Editor shares this value. Adjust the size of the button as needed.

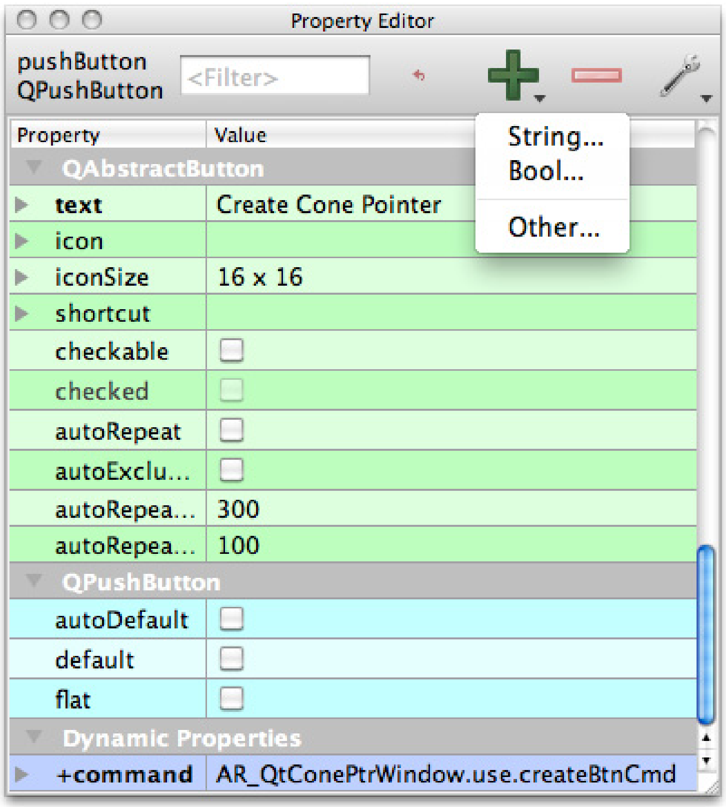
Note how the sections in the Property Editor are organized. Each section represents different properties associated with the selected widget at different abstraction levels. Because all widgets inherit from a **QWidget** class, they all have the same common properties in the **QWidget** section. You can use this organization to help you browse the Qt documentation if you need to know about any particular properties.

11. To make your button interact with Maya, you need to add a dynamic string property to the button. Press the large green **+** button in the Property Editor and select the Option **String**… (Figure 8.6).
12. You should now see a Create Dynamic Property dialog box. To link your button to a Python command in Maya, you must set the Property Name on this new property to be "+command" and then click OK.

As you probably noticed, adding a dynamic property to your button created an entry for it in the bottom of the Property Editor. The **+command** property corresponds to the `command` flag available when creating GUI controls using Maya commands. You can prefix any dynamic property with a **+** symbol to indicate that it maps to a flag of the same name on the



■ **FIGURE 8.5** A **Push Button** widget.



Property	Value
QAbstractButton	
text	Create Cone Pointer
icon	
iconSize	16 x 16
shortcut	
checkable	☐
checked	☐
autoRepeat	☐
autoExclu...	☐
autoRepea...	300
autoRepea...	100
QPushButton	
autoDefault	☐
default	☐
flat	☐
Dynamic Properties	
+command	AR_QtConePtrWindow.use.createBtnCmd

■ **FIGURE 8.6** Adding a dynamic string property in the Property Editor.

corresponding Maya command. Consequently, the string value you assign to this property inherits the same problems we discussed in Chapter 7. Namely, the value you pass to this command is either a string to be executed in __main__ (in which case you must enclose the value you enter in quotation marks) or it may be a function pointer.

Because of the issues associated with executing a string of statements in __main__, you may desire to pass the flag a function pointer as we did in Chapter 7, where we passed names of methods on an instance object by accessing attributes using the `self` name. *While you can still pass the name of a function, it must be a valid name in __main__: You cannot call methods on* `self`. The reason for this requirement is that when Maya builds this user interface, even when it is loaded from inside a class, its controls are not constructed in the context of an instance object, where the `self` name exists, but in __main__.

Consequently, we want to make as few assumptions about __main__ as possible, so we do not want to point to a method on a particular instance name for the GUI like `win`. Although there are numerous ways of handling this issue depending on your production environment and tool deployment,[2] we will make the assumption that the class name for the GUI (which we will code in a few steps) will exist in __main__. If we assume that the class name exists, we can safely access any class attribute associated with it. The mechanics of this process will be clearer in a few steps when we design our class, **AR_QtConePtrWindow**, for this GUI in Python.

13. In the Property Editor, assign the value AR_QtConePtrWindow.use.createBtnCmd to the **+command** property.
14. Drag a **Label** widget into your window and change its display text to "Y-Rotation:".
15. Drag in a **Line Edit** widget next to the label and assign the value inputRotation to its **objectName** attribute. This widget corresponds to Maya's text field control.
16. Double click on the inputRotation widget and enter a default value of 0.0.
17. Save your .ui file as cone.ui in your home folder (Documents folder on Windows). All Qt forms are stored in files with the .ui extension. This file contains XML code that makes up all the properties and widgets inside of your UI.

LOADING A Qt GUI IN MAYA

At this point we can investigate how to load the .ui file that you created in Qt Designer into Maya 2011 and later. To load your custom Qt Designer interface files, you need to use the `loadUI` command. We will issue this command from within a custom class, much as we called the `window` and `showWindow` commands in Chapter 7. We assume that you still have the cone.ui file from the previous example in your home directory.

1. Execute the following code in a Python tab in the Script Editor to create a new class, **AR_QtConePtrWindow**, and then instantiate it and call its **create()** method to display your window.

```
import maya.cmds as cmds;
import os;
```

[2]For example, when you deploy your class in a package of tools, you could include an **import** statement that will execute in __main__ when userSetup is run, using the `eval` function in maya.mel: `maya.mel.eval('python("from moduleName import ClassName");')`.

```
class AR_QtConePtrWindow(object):
    use = None;
    @classmethod
    def showUI(cls, uiFile):
        win = cls(uiFile);
        win.create();
        return win;
    def __init__(self, filePath):
        AR_QtConePtrWindow.use = self;
        self.window = 'ar_conePtrWindow';
        self.rotField = 'inputRotation';
        self.uiFile = filePath;
    def create(self, verbose=False):
        if cmds.window(self.window, exists=True):
            cmds.deleteUI(self.window);
        self.window = cmds.loadUI(
            uiFile=self.uiFile,
            verbose=verbose
        );
        cmds.showWindow(self.window);
    def createBtnCmd(self, *args):
        rotation = None;
        try:
            ctrlPath = '|'.join([
                self.window,
                'centralwidget',
                self.rotField]
            );
            rotation = float(
                cmds.textField(
                    ctrlPath,
                    q=True, text=True
                )
            );
        except: raise;
        cone = cmds.polyCone();
        cmds.setAttr('%s.rx'%cone[0], 90);
        cmds.rotate(0, rotation, 0,
            cone[0],
            relative=True
        );
win = AR_QtConePtrWindow(
    os.path.join(
        os.getenv('HOME'),
        'cone.ui'
    )
);
win.create(verbose=True);
```

Your window should now appear in Maya. One of the first things you may notice is that your Qt Designer interface is integrated seamlessly into Maya. Specifically, the interface automatically takes on the look of Maya's GUI skin. This integration prevents you from having to set all the colors in your custom GUI to match those of the Maya GUI. Your GUI should create a cone when you press its button. Changing the value for the rotation input field changes the cone's default heading when it is created.

Although it bears some similarities to those we created in Chapter 7, this GUI class is more complex and merits some discussion. The first item defined is a class attribute **use**, which you may recognize from the value we assigned the **+command** attribute in Qt Designer.

```
use = None;
```

This attribute, which is initialized to None in the class definition, will temporarily store the instance being created when its **__init__()** method is called.

```
def __init__(self, filePath):
    AR_QtConePtrWindow.use = self;
    self.window = 'ar_conePtrWindow';
    self.rotField = 'inputRotation';
    self.uiFile = filePath;
```

The rationale behind this pattern is that when the **create()** method is called (which we will investigate next), this attribute will be pointing to the most recently created instance. Consequently, when the button command's callback method is bound by referencing this class attribute, it will be pointing to the function associated with the instance on which **create()** is called (assuming you call **create()** after each instantiation, if you were to create duplicates of this window).

You could easily achieve a similar result using a variety of techniques, so your decision really boils down to how your tool suite deploys modules and classes. It is stylistically fairly common to import individual classes using the **from-import** syntax, so it would not be unusual to do so in __main__ via a userSetup script, for example. Feel free to disagree though and do what works for you with your own tools.

After initializing this "cache" variable, we set up some attributes to match the names we assigned to the window and to the input field in Qt Designer. Note that we also necessitate that the constructor be passed a `filePath` argument, which will specify the location of the .ui file on disk.

The `loadUI` command

The **create()** method looks very similar to those in Chapter 7, yet it makes use of a special command, `loadUI`, to load the .ui file.

```
def create(self, verbose=False):
    self.window = cmds.loadUI(
        uiFile=self.uiFile,
        verbose=verbose
    );
    cmds.showWindow(self.window);
```

The `loadUI` command will translate the .ui file into controls recognized by Maya, as though the interface were created with Maya's native GUI commands. Note that we also pass the command the `verbose` flag, which is an optional argument for the **create()** method. This flag will output detailed information on which widgets in the GUI will be functional. Recall that not all Qt widgets have corresponding Maya commands, so although they will be visible, they may not be accessible by other Maya commands. It is possible to access values of unsupported controls indirectly using signals and slots, which we examine shortly. When you called the **create()** method on the instance at the end of the last step, you passed the `verbose` flag, so you may have seen output like the following example in the History Panel.

```
# Creating a QWidget named "centralwidget". #
# Creating a QPushButton named "pushButton". #
# Creating a QmayaLabel named "label". #
# Creating a QmayaField named "inputRotation". #
# Creating a QDial named "dial". #
# Creating a QMenuBar named "menubar". #
# Creating a QStatusBar named "statusbar". #
# There is no Maya command for objects of type
QMainWindowLayout. #
# Executing: import maya.cmds as cmds;cmds.button
('MayaWindow|ar_conePtrWindow|centralwidget|pushButton',
e=True,command=AR_QtConePtrWindow.use.createBtnCmd) #
# There is no Maya command for objects of type Taction. #
# There is no Maya command for objects of type QmayaDragManager. #
# There is no Maya command for objects of type QHBoxLayout. #
# Result: MayaWindow|ar_conePtrWindow|centralwidget|
pushButton #
```

Note also that it is not a requirement to test for the existence of the window and delete it with the `deleteUI` command. Unlike the `window` command, the `loadUI` command will automatically increment the created window's name if there is a conflict, much like naming **transform** nodes.

Another interesting option for loading a Qt interface into Maya is to set the `uiString` flag instead of the `uiFile` flag when calling the `loadUI` command. The `uiString` flag allows you to embed the XML code contained in a .ui file directly into a Maya script as a string. This feature allows you to eliminate the requirement of deploying additional .ui files in your pipeline. Nevertheless, you should always keep your .ui files around somewhere in case you want to make changes later! If you opened your .ui file in a text editor and copied its contents, you could paste them into a `loadUI` call like the following alternative example.

```
self.window = cmds.loadUI(
    uiString= \
r''' # Paste your XML code here
''',
    verbose=verbose
);
```

Since the XML code will be too long to store in one variable, you need to use the correct syntax for wrapping text in Python. To wrap text, use the triple-double-quotation-mark syntax. Note also that we have used the **r** prefix to indicate that this variable stores a raw string. Refer to Chapter 2 if you need a refresher on this syntax.

Before proceeding, it is worth reiterating that our cache, **use**, only needs to be initialized before the `loadUI` command is called, as its invocation will trigger the button's association with the method immediately. Thereafter, we need not be concerned with the fact that **use** will be reassigned if another instance of the GUI were loaded. Recall our discussion on Python's data model in Chapter 2. The pointer to the button's callback method for the first instance of the GUI class will still be pointing to the same item in memory (the method on the first instance), rather than to the method on the item to which the cache happens to be pointing when the control's command is invoked. Consequently, multiple instances of this window will not interfere with one another, since each will point to its own method when it is created.

Accessing Values on Controls

Finally, let's look at the method that is actually called when you press the button.

```
def createBtnCmd(self, *args):
    rotation = None;
    try:
        ctrlPath = '|'.join([
            self.window,
            'centralwidget',
            self.rotField]
        );
        rotation = float(
            cmds.textField(
                ctrlPath,
                q=True, text=True
            )
        );
    except: raise;
    cone = cmds.polyCone();
    cmds.setAttr('%s.rx'%cone[0], 90);
    cmds.rotate(0, rotation, 0,
        cone[0],
        relative=True
    );
```

The first step is to try to get the rotation value entered in the text field as a float. We accomplish this task by constructing the full path to the text field and querying its value. If you examine the Object Inspector in Qt Designer (upper right corner), you can see the hierarchy of widget names as they will come into Maya, which lets us know that there is a widget called centralWidget between the main window and the text field. The rest of the method simply calls appropriate commands to create and rotate the cone.

2. If you would like to dock your new custom interface, execute the following code.

```
dock = cmds.dockControl(
    allowedArea='all',
    height=cmds.window(win.window, q=True, h=True),
    area='top',
    floating=False,
    label='Docked Cone Pointer Window',
    content=win.window
);
```

While your new GUI fits nicely into Maya, it's also not quite as good as it could be. The text input field is a somewhat clunky way for manipulating this GUI, so another control, such as a dial, may be a more interesting way to manipulate the default rotation value. Unfortunately, the dial is not a built-in Maya type, and so it cannot be directly accessed via commands to read its value. Fortunately, as we noted earlier, Qt provides a mechanism for solving this problem: signals and slots.

Mapping Widgets with Signals and Slots

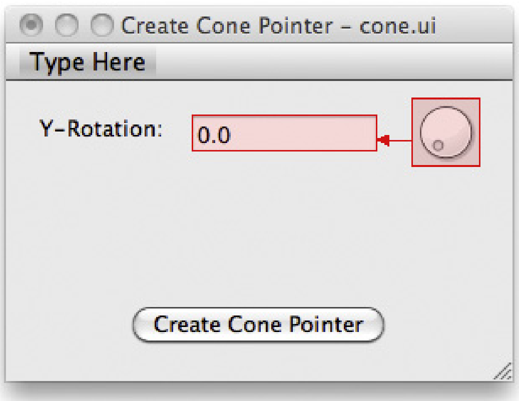
While it is worth reading the Qt documentation to fully understand the concept of signals and slots, we cover a basic example here. Recall that we earlier described the process of mapping signals and slots as effectively a more sophisticated callback system. Consequently, we can use this mechanism to pipe the results of controls that are not supported in Maya into ones that are, which allows us to indirectly query their values with Maya commands in our GUI class's button callback.

3. Return to Qt Designer and drag a **Dial** control into your window next to the **Line Edit** widget. You can find the **Dial** in the Input Widgets section.
4. Set the **objectName** property for the dial to "dialRotation" and set its maximum value to 360 in the **QAbstractSlider** section.
5. Enter Signal/Slot mode by selecting **Edit → Edit Signals/Slots** from the main menu. In this mode, you can no longer move your widgets around on the canvas.
6. Click and drag your dial to the line input and you should see a red arrow appear (as shown in Figure 8.7), as well as a mapping dialog.

In the mapping dialog that appears, the items on the left are different signals that the driving control emits, and the items on the right are the slots to which the signal can connect.

Although we want to map the **valueChanged(int)** signal on the left to a slot that will set the text value of the line edit, there is unfortunately not a slot on the **Line Edit** widget that accepts an integer. We would only be able to perform actions like selecting or clearing the text when the dial rotates, which isn't especially useful. In contrast to using ordinary function pointers with Maya GUI commands, Qt's signal/slot mechanism is type safe, meaning that it will only allow us to link up signals and slots that are sending and receiving the correct types of arguments. Consequently, to pipe an integer into the line edit, we unfortunately need to route the dial through another control.

7. Return to widget mode (**Edit → Edit Widgets**) and add a **Spin Box** widget to your canvas. Set its **maximum** property in the **QSpinBox** section to 360 as well.

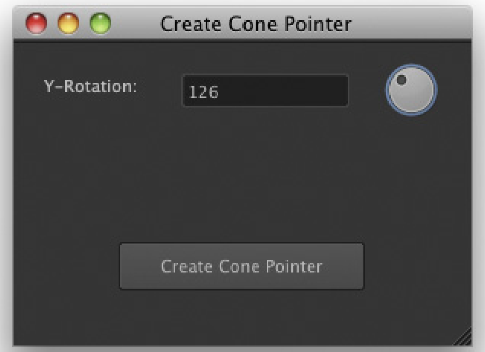


■ **FIGURE 8.7** Connecting a dial control to a line input in Qt Designer.

8. Return to Signal/Slot mode (**Edit → Edit Signals/Slots**) and drag the dial onto the spin box. Select the **valueChanged(int)** signal on the dial, and the **setValue(int)** slot on the spin box, and press OK. Now the dial will update the spin box.
9. Drag the spin box onto the **Line Edit** widget. Select **valueChanged (QString)** on the spin box as the signal, and **setText(QString)** on the line edit as the slot, and press OK. Now, when the value of the spin box is changed, it will be passed to the **Line Edit** widget as a string.
10. Because the spin box is simply there to perform a conversion for you, you probably do not want it cluttering up your GUI. Scale it to a small size and move it over your dial so it fits entirely within the dial. Right-click and select **Send To Back** from the context menu to hide the spin box behind the dial.
11. Return to Maya and execute the following code to create a new instance of your window.

```
win = AR_QtConePtrWindow(
    os.path.join(
        os.getenv('HOME'),
        'cone.ui'
    )
);
win.create(verbose=True);
```

You can now manipulate the dial control to alter the *y*-rotation value in your GUI, which you may find more intuitive. Using signals and slots, you have also ensured that when a user manipulates the dial control, the value passed to the text field will contain a numeric value (Figure 8.8).



■ **FIGURE 8.8** Using signals and slots allows you to integrate nonstandard GUI controls, such as dials, into Qt-based interfaces.

You could probably envision more useful applications of controls such as these to interactively manipulate character or vehicle rigs. Feel free to experiment and get creative with your GUIs as you explore the other widgets available in Qt!

PyQt

Up to this point we have focused on Maya 2011 and later, as Qt is seamlessly integrated into its GUI. However, since the `loadUI` command is not available in earlier versions of Maya, developers need another mechanism if they wish to support Qt-based GUIs in earlier versions of Maya. Fortunately, there is a command-line utility called pyuic4 that allows you to convert Qt Designer files into Python code that you can access in any Python interpreter. To access this tool, however, you must build a set of bindings, such as PyQt or PySide.

As the name suggests, PyQt is a combination of the Python Language and the Qt GUI toolkit. More specifically, PyQt is a set of Python bindings for the Qt GUI toolkit. Because it is built using Qt, PyQt is cross-platform, so your PyQt scripts will successfully run on Windows, OS X, and Linux. PyQt combines all the advantages of Qt and Python, and allows you to leverage all of the knowledge you have gained using Python in Maya.

Installing PyQt

The first step for building PyQt is to download all of the Qt header files as we described when installing the Qt SDK. Thereafter, you can download source code for the PyQt bindings online at *http://www.riverbankcomputing.co.uk/software*.

Once you have these required files, installing and building PyQt is a two-step process. First, you must download, install, and build the sip module. The sip module allows you to create bindings to items in the Qt C++ library. It creates what are called *wrappers*, which allow you to map Python functions and classes to corresponding items in the compiled C++ libraries. The Maya API is automatically available via a similar utility called SWIG, which we will discuss in Chapter 9. Second, you must download, install, and build the PyQt module. The PyQt module allows you to access all the classes and functions for creating Qt windows. When you have acquired the sip and PyQt modules, place them in your appropriate site-packages directory based on your operating system:

- **Windows:** C:\Program Files\Autodesk\Maya<version>\Python\lib\site-packages

- **OS X:** /Applications/Autodesk/maya<version>/Maya.app/Contents/Frameworks/Python.framework/Versions/Current/lib/python<version>/site-packages
- **Linux:** /usr/autodesk/maya/lib/python<version>/site-packages

For Maya, you will be using the Python bindings PyQt4, which are compatible with version 4.x of the Qt GUI toolkit. Unfortunately, you must build PyQt yourself. Because there are different platform installation and build instructions, you should refer to the companion web site to get the most up-to-date information. Note that PyQt is not licensed under LGPL, so these built-in modules cannot be freely distributed. PySide offers parallel functionality and different licensing terms, so you may investigate it if you require the added flexibility.

The PyQt bindings that you build are provided as a set of modules, which contain almost 700 classes and over 6,000 functions and methods that map nicely to the Qt GUI toolkit. Anything that can be done in the Qt GUI toolkit can be accomplished in PyQt.

Using PyQt in Maya 2011+

In addition to enabling the use of Qt in your Python tools, Maya 2011 and later contain support for Qt in the Maya API. Although we discuss the API in more detail starting in Chapter 9, it is worth covering a brief example here.

There are two main modules in PyQt that you need to import when implementing it in your own scripts: QtCore and QtGui. The QtCore module allows you to access the core, non-GUI-related Qt classes, such as Qt's signal and slot mechanism and the event loop. The second module, QtGui, contains most of the GUI classes. Additionally, you must import two other modules: sip and OpenMayaUI. You will be familiar with sip from the build process of PyQt: This module is required to map C++ objects to Python objects. You must also import the OpenMayaUI module, which is an API module for working with the Maya GUI. We will be covering the Maya API generally in all of the following chapters of this book. Your **import** statements should contain the following lines when creating PyQt scripts.

```
from PyQt4 import QtCore;
from PyQt4 import QtGui;
import sip;
import maya.OpenMayaUI as omui;
```

Now we can take a look at some PyQt code to create a window inside of Maya. Autodesk recommends that the safest way to work with Qt from within Maya is to create your own PyQt windows rather than manipulating the existing Maya interface, which can leave Maya unstable. Following this guideline, we offer code here for a window that will act as a template for creating any GUIs inside of Maya. You can reuse this code over and over again to create custom PyQt windows inside of Maya. Based on our window class example in Chapter 7, it should be clear how you could use this template.

```
def getMayaMainWindow():
    accessMainWindow = omui.MQtUtil.mainWindow();
    return sip.wrapinstance(
        long(accessMainWindow),
        QtCore.QObject
    );
class PyQtMayaWindow(QtGui.QMainWindow):
    def __init__(
        self,
        parent=getMayaMainWindow(),
        uniqueHandle='PyQtWindow'
    ):
        QtGui.QMainWindow.__init__(self, parent);
        self.setWindowTitle('PyQt Window');
        self.setObjectName(uniqueHandle);
        self.resize(400, 200);
        self.setWindow();
    def setWindow(self):
        # add PyQt window controls here in inherited classes
        pass;
```

In Maya 2011 and later, there is an API class called **MQtUtil**, which is a utility class to work with Qt inside of Maya. This class has some helpful functions to access the Qt layer underneath the Maya GUI commands layer. We use this class inside of our helper function, **getMayaMainWindow()**, to get a handle to the Maya main window using the **mainWindow()** method. Because you will learn more about working with the Maya API in later chapters, you can merely think of this call as a required step for the time being. We then use the sip module's **wrapinstance()** function to return the reference to the main window as a **QObject**, which is Qt's core object type.

We define our custom window in a **PyQtMayaWindow** class, which inherits from Qt's **QMainWindow** class. In our **PyQtWindow** class's **__init__()** method, we use our helper function to set the main Maya window as the default parent when calling **__init__()** on the base class. We want to make PyQt windows children of the main Maya window because we do not want them to be separate from Maya (and thus be treated by the operating system as though they were separate application windows). We then set the window's title and size and call the **setWindow()** method, which subclasses can override to display any desired controls, or to change the default window title, object name, and size.

When inheriting from this class and overriding the **setWindow()** method, it is advisable to plan the GUI on paper to save time positioning controls, because the window will be defined programmatically. Refer to the PyQt Class Reference Documentation for the specific classes and functions needed to create your widgets and controls, or use the pyuic4 tool in conjunction with Qt Designer to rapidly create your GUI and get right into adding functionality to it.

To create the window specified in this class, simply create an instance of it.

```
pyQtWindow = PyQtMayaWindow();
pyQtWindow.show();
```

Using PyQt in Earlier Versions of Maya

If you are not working in Maya 2011 or later, but are still very interested in the potential of what PyQt scripts can add to your existing tools, you are still able to run your PyQt scripts in an earlier version even though it lacks the Qt-related classes in the API. Although we cannot go into all of the specifics, Autodesk fortunately provides PDF documents with instructions in the following locations to take advantage of PyQt technology in earlier versions of Maya:

- **Windows:** Program Files\Autodesk\Maya<version>\devkit\other\PyQt Scripts
- **OS X:** /Applications/Autodesk/maya<version>/devkit/devkit/other/PyQt Scripts/
- **Linux:** /usr/autodesk/maya<version>/devkit/other/PyQtScripts/

CONCLUDING REMARKS

Equipped with a better understanding of Qt and Maya, as well as some Qt tools and PyQt, you now have the skill set to take GUI design to the next level. Qt does a nice job of catering to a variety of users. Depending on your preferences, you may like the additional level of control that code provides, making PyQt a great fit for you. On the other hand, if you excel when working with visual tools, then Qt Designer will be your tool of choice.

Information on Qt and GUI design could occupy many volumes worth of knowledge, which is why we have covered only the crucial concepts required to work with Qt in Maya. If this chapter has piqued your interest in Qt, we recommend picking up some additional resources to continue on your learning path. The companion web site offers links to books and web sites where you can find information on working with Qt, designing GUIs in general, and working with Qt Designer and Maya.