Image Kit Programming Guide

Graphics & Animation: 2D Drawing

2008-06-09

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures and Tables

Introduction to Image Kit Programming Guide

The Image Kit is an Objective-C framework, introduced in Mac OS X v10.5, for browsing, viewing, editing, and processing images in an efficient manner. It also supports browsing Core Image filters, previewing the effects, and providing controls for individual filters. This document describes Image Kit user interface elements and shows, through code examples, how to use Image Kit classes.

Anyone developing a digital media application that supports images will want to read this document to find out how to use the Image Kit to provide the best user experience possible. You don't need to be a seasoned Cocoa programmer to use the Image Kit or to read this guide, although prior experience with Objective-C is helpful and will make it easier to understand the code examples.

The examples in this document use Xcode version 3.0 and Interface Builder version 3.0. If you've used earlier versions of Interface Builder, you'll be delighted by the high level of integration with Xcode that version 3.0 offers. Interface Builder version 3.0 is a major update that offers streamlined operations, a library instead of palettes, and many other features that you'll want to familiarize yourself with. The instructions in this document for creating a user interface with Interface Builder version 3.0 should be sufficient for you to create applications using Image Kit. To get a deeper understanding, you'll want to consult *Interface Builder User Guide*.

Organization of This Document

This document is organized into an overview chapter followed by chapters that provide instructions on how to support various image capabilities.

- "Basics of Using the Image Kit" (page 9) describes the tasks that each of the classes support and gives an overview of using the framework with Xcode and Interface Builder.
- "Viewing, Editing, and Saving Images in an Image View" (page 15) describes the user interface provided for single images and shows how to set up an image view, support editing, and provide an accessory view for saving images.
- "Browsing Images" (page 37) shows the user interface provided for viewing large sets of images and gives instructions for supporting an image browser in an application.
- "Showing Slides" (page 53) provides detailed information for supporting slideshows.
- "Taking Snapshots and Setting Pictures" (page 61) describes a lightweight panel that applications can use to support choosing pictures and taking snapshots that allow the user to provide a "buddy" picture or icon-sized image.
- "Browsing Filters and Setting Input Parameters" (page 69) shows the user interface for browsing Core Image filters, describes how to implement the filter browser in an application, and describes how to obtain a view for the input parameters of a filter.

See Also

The following documents are helpful guides and references for many of the tasks described in the book:

- *Image Kit Reference Collection* contains the class and protocol reference documentation.
- *Core Image Programming Guide* shows how to create Core Image filters (`CIFilter` objects) and apply them to images. (Although *Image Kit Programming Guide* shows how to provide a user interface for browsing and previewing filters, it doesn't show how to apply a filter to an image.)
- *Quartz 2D Programming Guide*, *CGImageSource Reference*, and *CGImage Reference* provide task and reference documentation for creating an image source and extracting an image from the source.
- *Cocoa Fundamentals Guide* is the entry-point conceptual document for anyone wanting to become a Cocoa programmer.

Basics of Using the Image Kit

As consumers accumulate more and more digital media, image applications are faced with handling large amounts of data in an efficient manner. Consumers not only want to open, view, and organize images, but they often need to crop images, adjust brightness, apply effects, view metadata, or perform a number of other image editing operations. The Image Kit framework is a bundle of image handling services that supports these tasks and more. It is designed to operate efficiently while providing a user interface with the look and feel of Mac OS X. By using the Image Kit to perform the image handling tasks that most digital media applications need, you'll be able to focus your code writing efforts on the parts of your application that distinguish it from other applications.

This chapter introduces the tasks supported by Image Kit classes and discusses how to set up Xcode and Interface Builder so that you can successfully build an application using the Image Kit. By reading this chapter, you'll get an idea of what each of the the Image Kit classes can do. To use Image Kit classes effectively, you'll need to read the chapters that describe how to implement the tasks required by your application.

Tasks Supported by the Image Kit

The Image Kit is a high-level Objective-C framework. It is built on a number of other Mac OS X graphics technologies, including Quartz 2D, Core Image, Core Animation, and OpenGL. When you use the Image Kit, you can read any image data that Quartz 2D and the Image I/O frameworks support.

There are eight major categories of tasks that the Image Kit supports, the most basic of which is to view an image. Table 1-1 (page 10) summarizes the tasks you can support with the Image Kit and lists the classes and protocols that you use for each task. Like the `NSImageView` class, the `IKImageView` class displays a single image in a frame and optionally can allow a user to drag an image to the view. Unlike the `NSImageView` class, the `IKImageView` class supports any image file format that Quartz supports.

The `IKImageView` class provides methods for zooming and for setting tool modes for moving, selecting, cropping, rotating, and annotating. With the appropriate tool mode selection, the view automatically displays a selection rectangle, a cropping rectangle, or an annotation oval. (Your application has to implement code that performs the actual data manipulation for these three tool modes.)

The `IKImageEditPanel` class allows users to view image metadata and to adjust digital images by:

- Changing the exposure
- Setting the white and black points
- Adjusting the gamma, saturation, contrast, and brightness
- Sharpening
- Applying color effects: black and white, sepia, antique, fade color, boost color, blur, and invert

The `IKImageView` class has built-in support for the Image Edit panel (`IKImageEditPanel` class). After making the appropriate setting in your application, a user simply clicks an image and the Image Edit panel opens. No other action is needed on your part. The image editing panel can be used independently of the `IKImageView` class by performing the appropriate setup work and implementing the `IKImageEditPanelDataSource` protocol.

Note: The user term for Image Edit panel is Image Edit window. See "Terminology for Users" (page 14) for more information about the terminology that your application should adopt for strings in the user interface, including help tags and help.

The Image Kit provides the `IKImageBrowserView` class for displaying and arranging images in a grid in a way similar to the grid you see in iPhoto. The image browser can display large numbers of images, icons, movies, Quartz Composer compositions, and PDF documents. Users can drag images to the browser, select images, and move them. The Image Kit achieves smooth animation when images are moved in the browser.

The options for saving an image vary depending on the file format of the image. For that reason, the Image Kit provides the `IKSaveOptions` class. Save options appear as an accessory view (pane) in an `NSSavePanel` object.

A slideshow is a popular way for consumers to view digital images. The `IKSlideshow` class and the `IKSlideshowDataSource` protocol provide an easy way for your application to support slideshows of images, PDF documents, and other image data. You can start, pause, and stop slideshows, export slideshows, access a specific item in the show, and perform a number of other tasks.

Mail and iChat are two of the many applications that allow users to supply an icon or photo that represents their identity. The `IKPictureTaker` class provides a lightweight panel for choosing and cropping an image or for taking a snapshot with a digital camera. The panel keeps track of recent pictures, allowing the user to choose from among them as an alternative to navigating to an image or taking a snapshot.

The `IKFilterBrowserView` and `IKFilterBrowserPanel` classes provide a user interface for browsing Core Image image processing filters and previewing their effects.

The `IKFilterUIView` class provides a user interface for Core Image filters, making it easy for you to support image processing with the more than one hundred filters supplied by the system. You can choose which parameters are available to set and the size of the controls. If you want to supply a custom user interface for a filter that you write, you can use the `IKFilterCustomUIProvider` class.

Table 1-1 Tasks and the classes that support them

Task	Classes and protocols
View and edit images	`IKImageView`
Adjust images, apply color effects, view metadata	Use the `IKImageEditPanel` and `IKImageEditPanelDataSource` classes only when you are not using the `IKImageView` class, which has built-in support for the Image Edit panel.
Display and arrange large numbers of images	`IKImageBrowserView`, `IKImageBrowser-DataSource`, `IKImageBrowserDelegate`, and `IKImageBrowserItem`

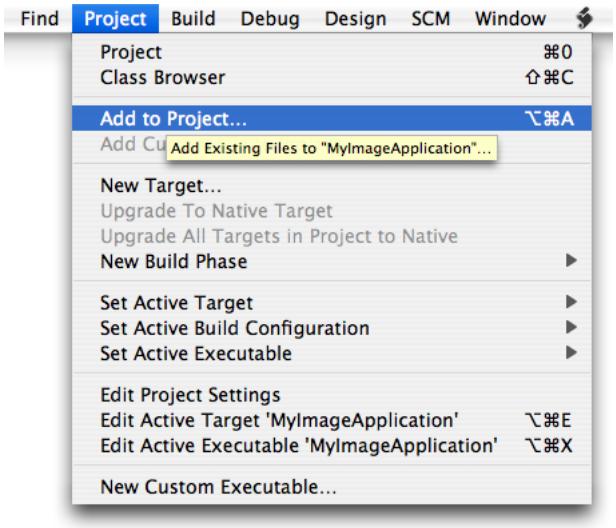
Task	Classes and protocols
Run slideshows	`IKSlideshow` and `IKSlideshowDataSource Protocol`
Choose an icon-sized picture from a directory or take a snapshot with an iSight or other digital camera	`IKPictureTaker`
Save images in a variety of file formats with options appropriate for the format	`IKSaveOptions`
Browse Core Image filters and preview their effects	`IKFilterBrowserView` and `IKFilterBrowserPanel`
View and adjust the input parameters of a Core Image filter	`IKFilterUIView` and `IKFilterCustomUIProvider`

Using the Image Kit in Xcode

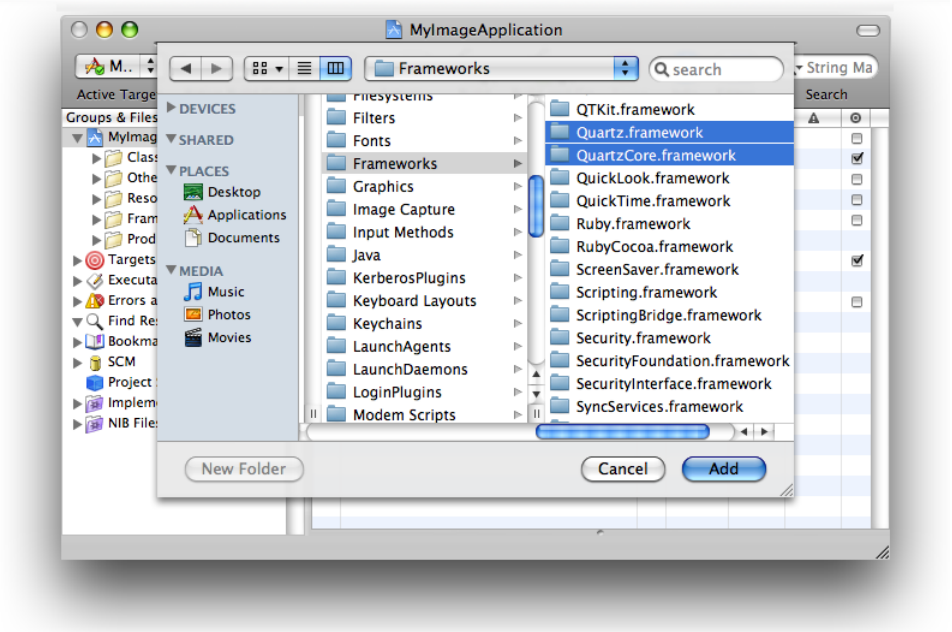
To use the Image Kit framework in Xcode, you need to import the Quartz and Quartz Core frameworks. The Quartz framework contains Image Kit. The Quartz Core framework contains the Core Image classes needed by the `IKFilterBrowserPanel` and `IKFilterBrowserView` classes.

To import these frameworks in Xcode:

1. Open Xcode and create a Cocoa application.
2. Choose Project > Add to Project.



3. Navigate to `System/Library/Frameworks`, choose the `Quartz.framework` and `QuartzCore.framework`, and click Add.



4. In the sheet that appears, click Add.

5. Save the project.

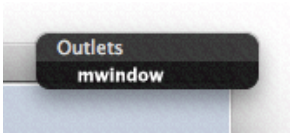
6. After importing the frameworks, make sure that you add `#import <Quartz/Quartz.h>` into the appropriate files.

Using the Image Kit with Interface Builder 3.0

The examples in this document use Interface Builder 3.0. If you've never used Interface Builder before, you may want to skip this section and instead read the first chapter of *Interface Builder User Guide*. If you have used previous versions of Interface Builder, you'll notice some substantial changes. This section points out a few of the changes that you'll encounter when following the instructions in this document to create applications that use Image Kit classes.

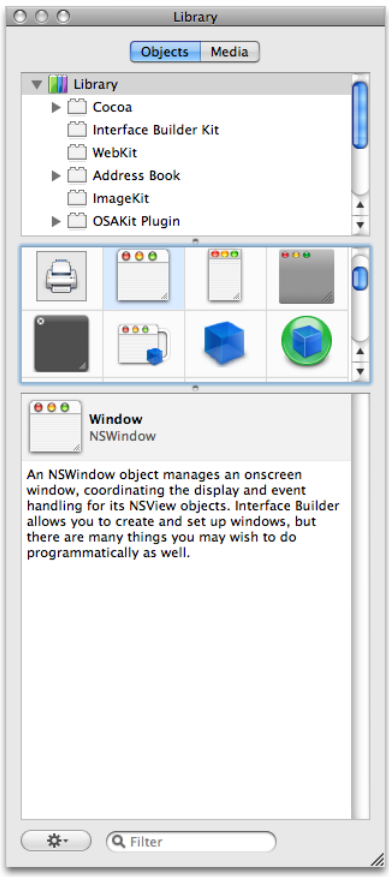
Because of the new integration between Xcode 3.0 and Interface Builder 3.0, the actions and outlets that you declare in the interface file for a class are synchronously updated in Interface Builder. This will become apparent to you when you begin to make connections. When you drag from a controller in the nib document window to a view or other user interface element, a connections panel appears. The **connections panel** is a tool that lets you examine and create connections between objects. Figure 1-1 shows a connections panel for a window that has the instance variable `mwindow` as an outlet. This example is simple; some objects have several outlets as well as received and sent actions associated with it. For a complete description of the connections panel, see *Interface Builder User Guide*.

Figure 1-1 An outlet for a window, shown in a connections panel



The Interface Builder library (shown in Figure 1-2) replaces the palette. The library contains the objects and resources you can use in your nib file. You can browse objects in the library by plug-in—Cocoa, IB, and so on—or use the search field to find an object. You can also create custom groups and smart groups to organize objects the way you want.

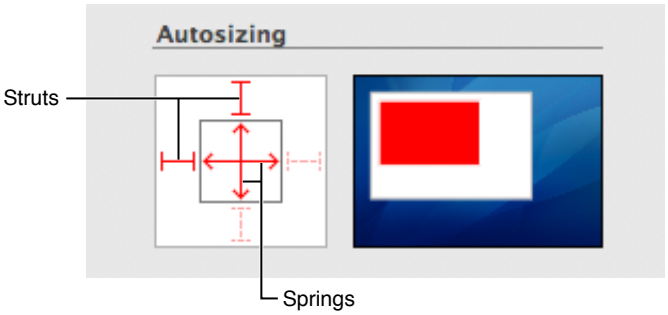
Figure 1-2 The Interface Builder library



The media pane of the library shows images, sounds, and other resource that are available in your Xcode project. To use objects or media from the library, simply select what you want and drop them where you want them in your nib document.

One task you'll perform repeatedly as you create Image Kit applications is to set springs and struts that control the autosizing behavior in the user interface. The Interface Builder size inspector now includes an animation that shows how the springs and struts affects autosizing behavior.

Figure 1-3 Autosizing in the Interface Builder size inspector



Perhaps more noticeable than the new features in Interface Builder are the absence of features and the tasks associated with them. Seasoned Interface Builder users will notice fewer tasks for wiring the user interface. Gone is the need to drag header files to the nib document window or to parse files.

Terminology for Users

Standard terminology helps users to come up to speed quickly using your application. Familiar terminology doesn't get in the way of learning all the impressive features that you put into your application. The Image Kit introduces a number of new classes and protocols, named using terms familiar to developers. When you provide labels in the user interface for your application, write help tags, or provide documentation, your users will be best served if you adopt standard user terminology rather than developer terminology. Table 1-2 provides the user terms for the most common developer terms used in the Image Kit. You can find out more information about providing users with a consistent visual and behavioral experience by reading *Apple Human Interface Guidelines*.

Table 1-2 Developer and user terms for the Image Kit

Developer term	User term
accessory view	pane
image browser view	image browser
Image Edit panel	Image Edit window
Open panel	Open dialog
picture taker panel	picture taker
Save As panel	Save As dialog

Viewing, Editing, and Saving Images in an Image View

The `IKImageView` class displays a single image in a frame and optionally can allow a user to drag an image to it. It is similar to the `NSImageView` class, except that `IKImageView` supports any image file format that Quartz 2D and the Image I/O framework support, including Quartz images (`CGImageRef`), images that have metadata, and images whose location is specified as an `NSURL` object. In addition, the `IKImageView` class supports zoom, rotation, selection, cropping, and other image editing operations.

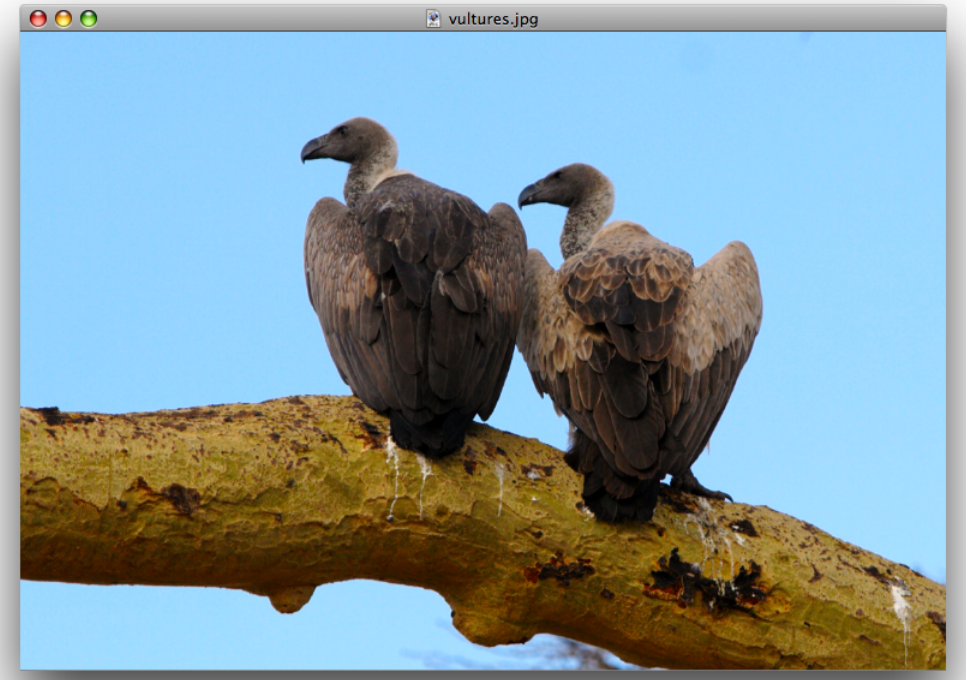
This chapter shows the user interface for an image view, the image edit panel (`IKImageEditPanel` class), and the save options accessory view (pane) (`IKSaveOptions` class). It provides instructions for creating an image editing application incrementally. You'll see how to:

- View images in an image view
- Set up an image view to use the Image Edit panel
- Set up a pane for saving images in various formats
- Add zoom controls
- Add image editing tools
- Add a Save panel that uses image saving options

The Image View User Interface

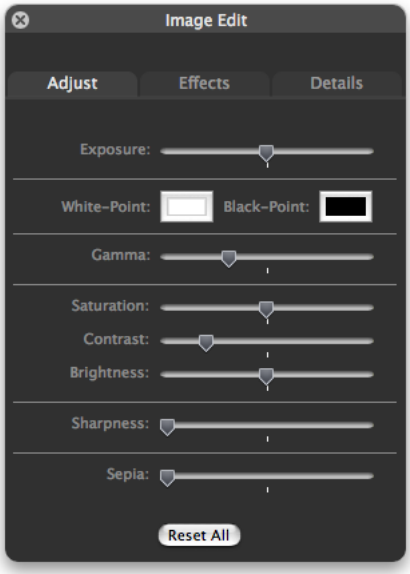
The image view (`IKImageView`) in the user interface looks similar to any view that contains an image. The image shown in Figure 2-1 could just as easily be in an `NSView` object, an `NSImageView` object, or a Carbon window.

Figure 2-1 An image view



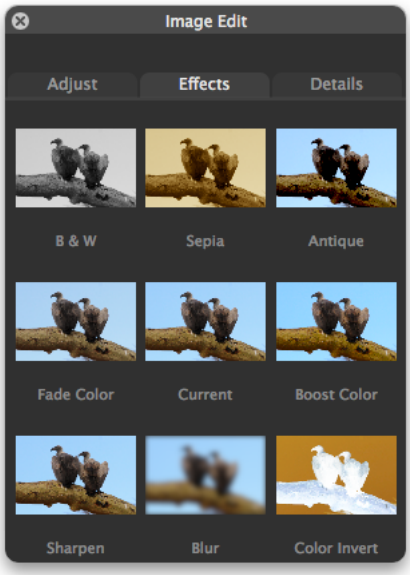
An `IKImageView` object has an important characteristic that no other image container has. When the user double-clicks an image in an image view, an Image Edit panel appears, as shown in Figure 2-2. The panel allows the user to make adjustments to the image, apply a number of effects, and view image metadata and properties. The Adjust pane provides the most commonly used image adjustments.

Figure 2-2 The Adjust pane of the Image Edit panel



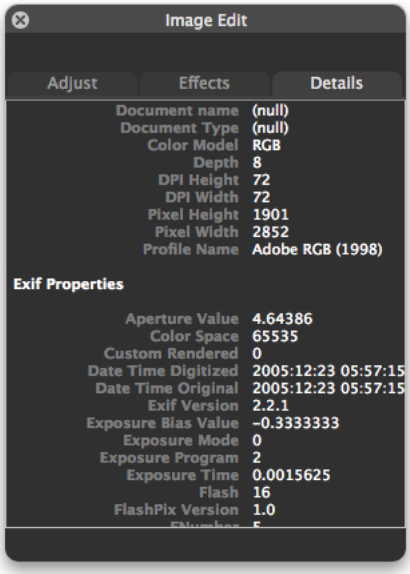
The Effects pane (shown in Figure 2-3) allows the user, with a single click, to preview and apply sharpening, blurring, or color filters.

Figure 2-3 The Effects pane of the Image Edit panel



The Details pane displays all the metadata for an image, as well as a comprehensive list of image properties, as shown in Figure 2-4. If the image information is more than can fit in the pane, the Image Kit supplies scroll bars automatically.

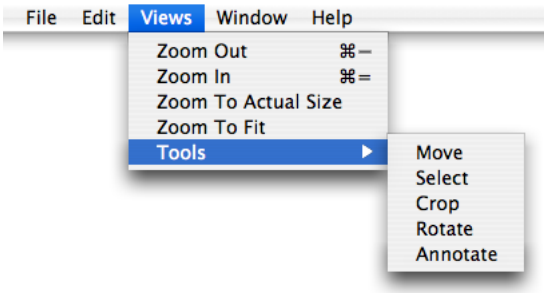
Figure 2-4 The Details pane of the Image Edit panel



There are very few tasks you need to perform for the Image Edit panel to appear. You need to set the appropriate image view state, set up a shared instance of the Image Edit panel (`IKImageEditPanel`), and set the data source. The Image Kit framework takes care of the rest—showing the panel, switching panes, responding to changes made by the user, fetching and displaying image information, and closing the Image Edit panel.

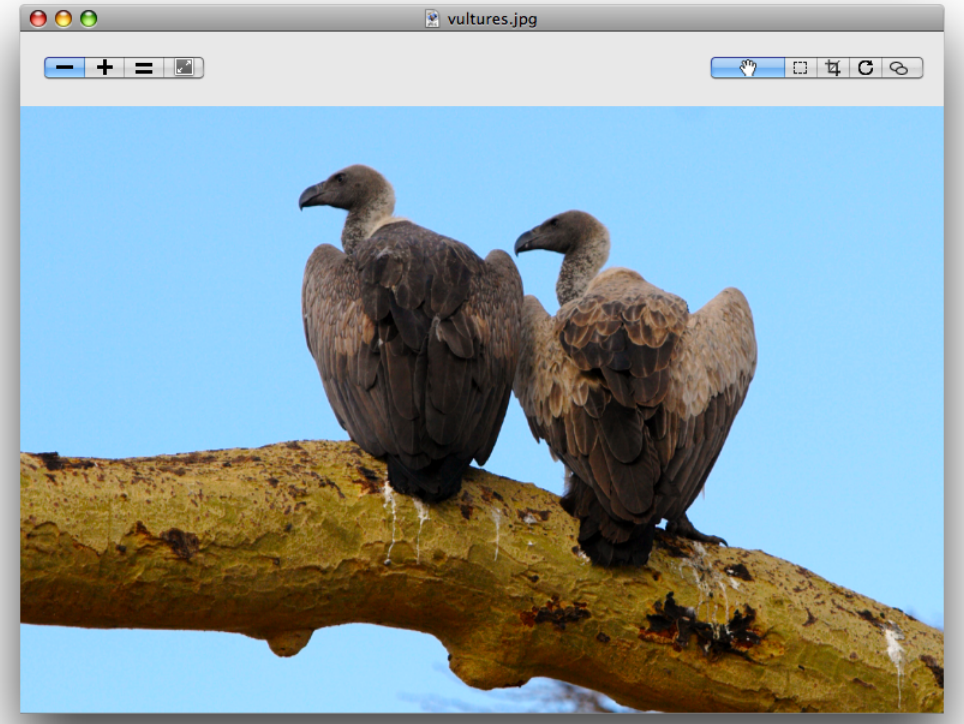
The image view supports the use of tools for moving zooming, cropping, rotating, and applying annotations to an image. You need to provide a user interface for zoom operation and tool selection. At a minimum, you need to set up a menu similar to what's shown in Figure 2-5.

Figure 2-5 A menu that supports zooming and tools



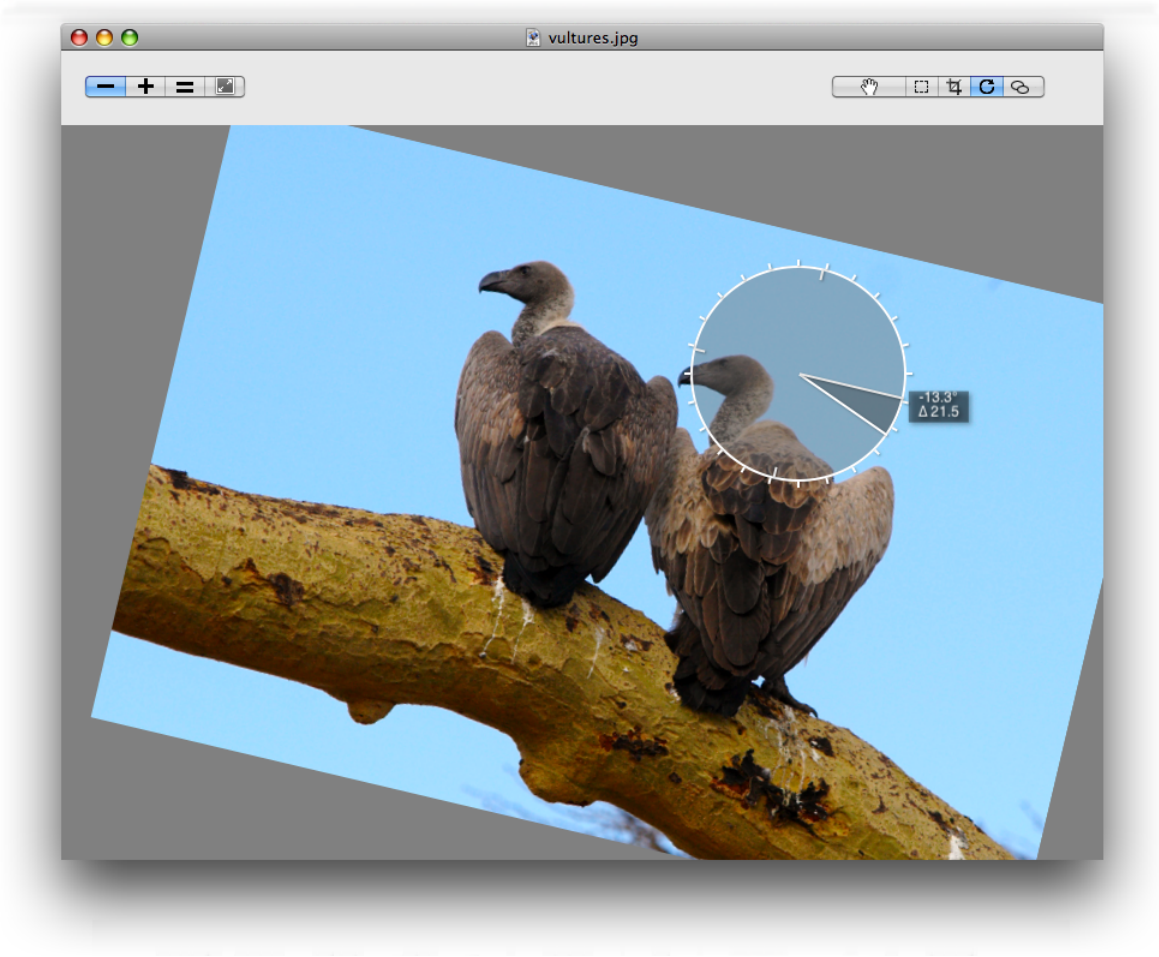
Ideally, you would also provide controls in the image view window, as shown in Figure 2-6. The figure shows two segmented controls (created in Interface Builder using `NSSegmentedControl` controls), each of which uses custom icons that you provide.

Figure 2-6 Controls that support zooming and tool selection



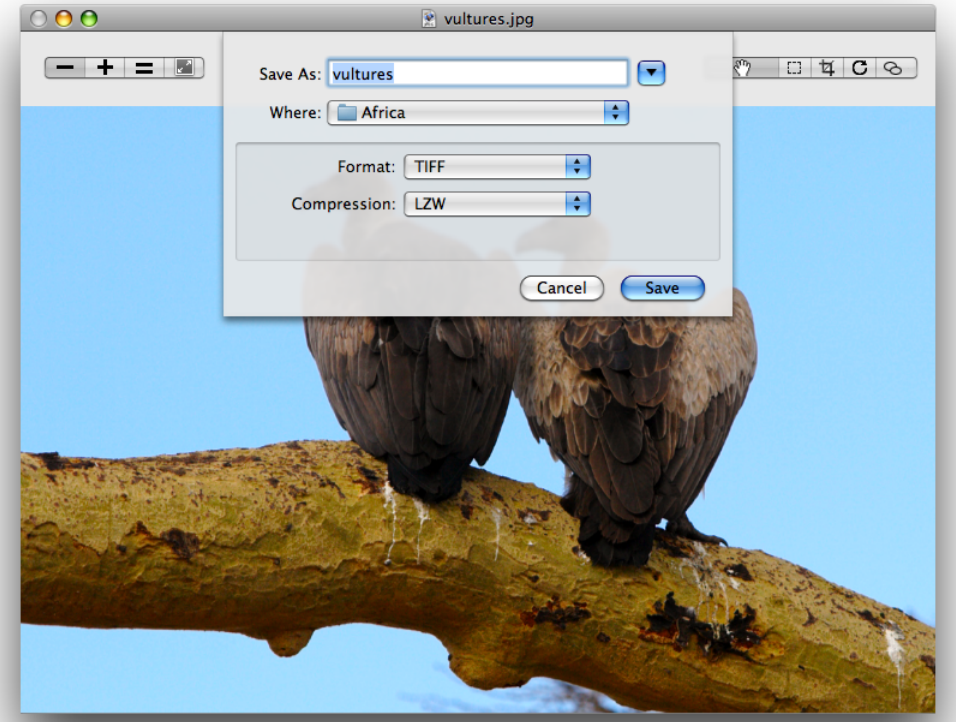
The image view keeps track of the zoom factor, the rotation angle, and whether the image can be edited. You need to respond to the user's selection by setting the tool mode. The Image Kit framework takes care of animating move, zoom, and rotation operations; showing a rotation circle (see Figure 2-7); displaying crop and selection rectangles; and displaying an annotation area. You need to implement copying data to the pasteboard, cropping the image, and handling the text for the annotation area.

Figure 2-7 The rotation user interface



No image viewing and editing application would be complete without the ability to save an edited image. The `IKSaveOptions` class provides an accessory view for an `NSSavePanel` object that allows the user to choose an image file format and to set options appropriate for that format. The user can choose from among a number of formats. The options appropriate for that format appear below the format. For example, for a TIFF format, the user can choose compression options (as shown in Figure 2-8).

Figure 2-8 A Save As dialog with an accessory view (pane) for file format options



Viewing an Image in an Image View

This section shows how to open and display an image in an image view. You'll set the image view options so that the Image Edit panel opens when the user double-clicks the image. First you'll set up the Xcode project, the project files, and the controller interface. Then you'll create the user interface in Interface Builder. Finally, you'll add the necessary routines to the implementation file.

Setting Up the Project, Project Files, and the Controller Interface

Follow these steps to set up the project:

1. Open Xcode, choose File > New Project.
2. Choose Cocoa Application and click Next.
3. Name the project `My Image Viewer`, and click Finish.
4. Choose Project > Add to Project and add the Quartz and Quartz Core frameworks.

   For details, see "Using the Image Kit in Xcode" (page 11).

5. Choose Project > Add to Project, navigate to an image to use as a default image, and click Add.
6. In the sheet that appears, click Add.

   This image appears in the view whenever the application launches.
7. Choose File > New File.
8. Choose Objective-C Class and click Next.
9. Name the file `Controller.m` and keep the option to create the header file. Then click Finish.
10. In the `Controller.h` file, import the Quartz framework by adding this statement just below the statement to import Cocoa:

    ```
    #import <Quartz/Quartz.h>
    ```
11. Add a directive for the `IKImageView` class:

    ```
    @class IKImageView;
    ```
12. Add instance variables to the Controller interface.

    You need an image view and a window to contain the view. You'll set these up later in Interface Builder.

    ```
    IBOutlet IKImageView *  mImageView;
    IBOutlet NSWindow *     mWindow;
    ```

    You need to keep track of image properties and the uniform type identifier of the image in the view.

    ```
    NSDictionary * mImageProperties;
    NSString *     mImageUTType;
    ```
13. Save and close the `Controller.h` file.
14. In the `Controller.m` file, import the Application Kit classes by adding this statement.

    ```
    #import <AppKit/AppKit.h>
    ```
15. Close the `Controller.m` file.

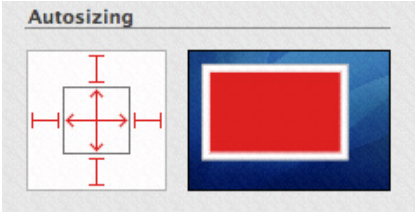
Creating the User Interface

Set up the user interface in Interface Builder by following these steps:

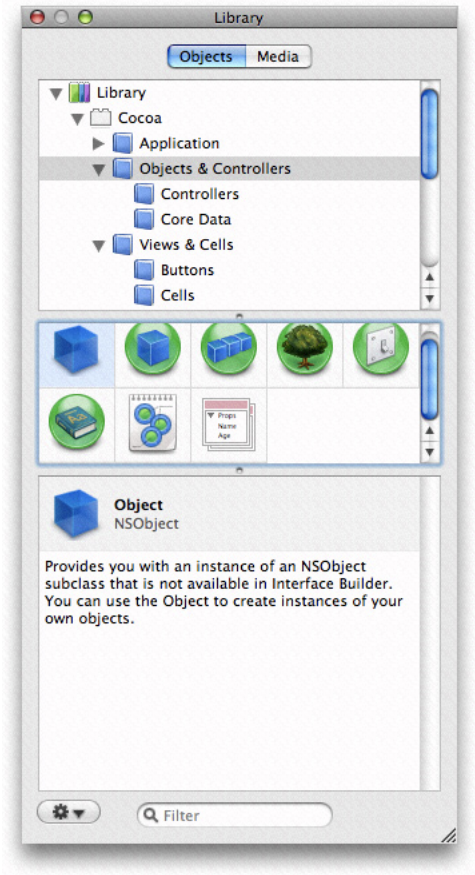
1. Double-click the `MainMenu.nib` file (located in the Resources group) to open Interface Builder.
2. Choose File > Synchronize With Xcode.
3. Double-click the Window icon in the nib document window.
4. In the Size inspector, set the width of the window to 800 and the height to 600.
5. Drag a Image View from the Library to the window and resize the view to fit the window.

Tip: Type Image View in the search field to locate it easily.

6. In the Size inspector, the Autosizing springs should look as follows. If they don't, set them so they do.

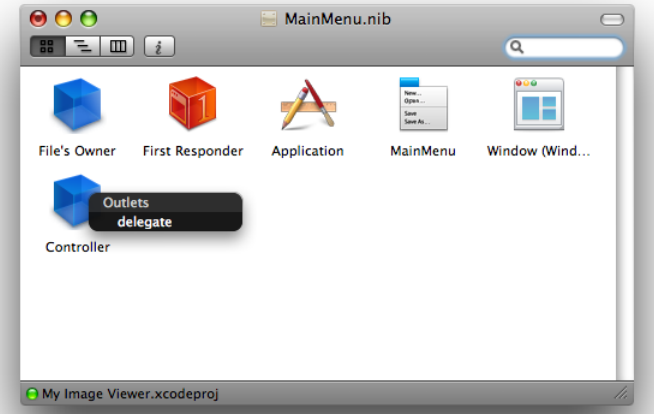


7. Drag an Object (`NSObject`) from the Library to the nib document window.



8. Type Controller in the Name field of the Identity inspector and press Return.
9. Choose Controller from the Class pop-up menu.
10. Control-drag from the controller icon to the title bar of the window. Then click the `mWindow` outlet that appears in the connections panel.
11. Control-drag from the controller icon to the `IKImageView` view. Then click the `mImageView` outlet that appears in the connections panel.

12. Control-drag from the window icon to the controller icon. Then click the `delegate` outlet that appears in the connections panel.



13. Save the nib file.

Adding Routines to the Implementation File

Now you'll go back to Xcode to add code to implement the image viewer.

1. Open the `Controller.m` file.

2. Add an `openImageURL:` method to take care of opening images.

   The easiest way of opening an image is to use the `setImageWithURL:` method. This method is best for RAW images.

```
- (void)openImageURL: (NSURL*)url
{
     [mImageView setImageWithURL: url];
     [mWindow setTitleWithRepresentedFilename: [url path]];
}
```

   An alternate implementation is to use the Quartz opaque data types `CGImageRef` and `CGImageSourceRef` and their associated functions to create an image source, extract an image, get the image properties, and set the image to the image view. If you are using a TIFF file that contains multiple images, you need this implementation to display any image other than the first one. That's because `setImageWithURL:` displays only the first image of a multiple-image file.

```
- (void)openImageURL: (NSURL*)url
{
    CGImageRef          image = NULL;
    CGImageSourceRef    isr = CGImageSourceCreateWithURL( (CFURLRef)url, NULL);

    if (isr)
```

```
    {
        image = CGImageSourceCreateImageAtIndex(isr, 0, NULL);
        if (image)
        {
         mImageProperties = (NSDictionary*)CGImageSourceCopyPropertiesAtIndex(
                isr, 0, (CFDictionaryRef)mImageProperties);
        }
        CFRelease(isr);
    }

    if (image)
    {
      [mImageView setImage: image
          imageProperties: mImageProperties];

     [mWindow setTitleWithRepresentedFilename: [url path]];
      CGImageRelease(image);
    }
}
```

3. Add an `awakeFromNib` method.

   This method first creates a URL for the default image file by getting the path to the resource in the bundle and then converting the path to a URL. After opening the URL, the method sets up the image view so that the Image Edit panel can open, and the image zooms to fit the view.

   Make sure that you substitute the appropriate string for "earring", which should be the name of the default image file without its filename extension. Also, use the appropriate extension.

```
- (void)awakeFromNib
{
    NSString *   path = [[NSBundle mainBundle] pathForResource: @"earring"
                            ofType: @"jpg"];
    NSURL *      url = [NSURL fileURLWithPath: path];

    [self openImageURL: url];

    // customize the IKImageView...
    [mImageView setDoubleClickOpensImageEditPanel: YES];
    [mImageView setCurrentToolMode: IKToolModeMove];
    [mImageView zoomImageToFit: self];
}
```

4. Open the `Controller.h` files and add the following method signature.

```
- (void)openImageURL: (NSURL*)url;
```

5. Click Build and Go.

6. When the application launches, double-click the image to make sure that the Image Edit panel opens.

   If the window opens, but an image does not appear, make sure that you've included the correct filename in your code.

7. Resize the image to make sure that it zooms to fit the size of the window.

   If the image does not zoom to fit, check the connections between the window and the controller in Interface Builder.

The next section shows how to use the `IKSaveOptions` class to add an accessory view to the Save panel.

Saving Images

The `IKSaveOptions` class handles image and PDF saving options. After creating an `NSSavePanel` object, you allocate and initialize a save options accessory view and then add the view to the Save dialog. Keep in mind that `IKSaveOptions` handles the options, but it does not actually save the image or PDF. After you set up the save options accessory panel, you also need to implement a save method.

Follow these steps to add the image options accessory view to the My Image Viewer application:

1. Open the `Controller.h` file and add a save options instance variable.

```
IKSaveOptions * mSaveOptions;
```

2. Add the following method signature:

```
- (IBAction)saveImage: (id)sender;
```

3. In the `Controller.m` file, add a method that presents a Save panel with the save options accessory view (pane).

   The method first creates an instance of the `NSSavePanel` class. Then it allocates a save options object and initializes it with the image properties and UT type of the image that will be saved. Next the code adds the save options view to the Save panel. It shows the Save panel as a sheet, providing a selector that is invoked when the Save panel terminates. (You'll write that method next.)

```
- (IBAction)saveImage: (id)sender
{
    NSSavePanel * savePanel = [NSSavePanel savePanel];
    mSaveOptions = [[IKSaveOptions alloc]
                        initWithImageProperties: mImageProperties
                                      imageUTType: mImageUTType];
    [mSaveOptions addSaveOptionsAccessoryViewToSavePanel: savePanel];

    NSString * fileName = [[mWindow representedFilename] lastPathComponent];
    [savePanel beginSheetForDirectory: NULL
                                 file: fileName
                       modalForWindow: mWindow
                        modalDelegate: self
                       didEndSelector: @selector(savePanelDidEnd:returnCode:contextInfo:)
                          contextInfo: NULL];


}
```

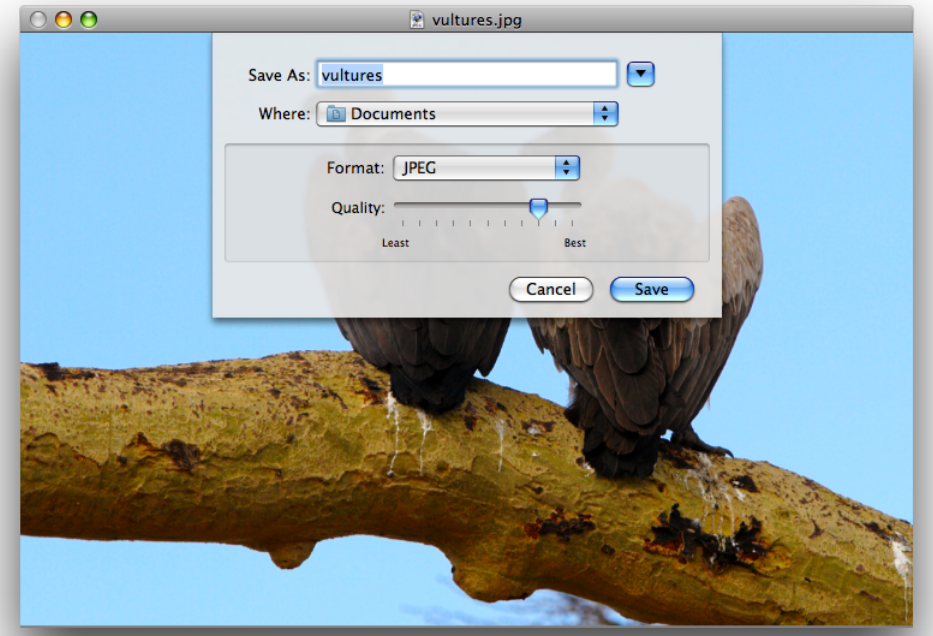
4. Add a `savePanelDidEnd` method that performs the actual saving.

   If the user clicks the Save button, the method obtains the filename and type and retrieves the image from the view. If the image exists, the method uses the `CGImageDestinationRef` opaque type (from the Image I/O framework) to create an image destination based on a URL representation of the file path. It saves the image with its properties, finalizes the destination, and releases the `CGImageDestination` object because it is no longer needed.

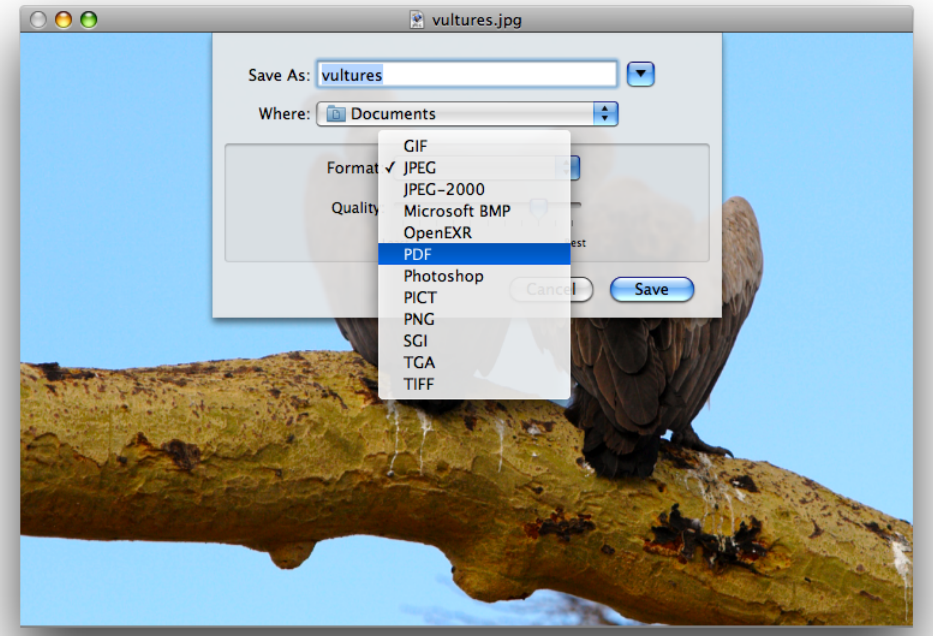
```
- (void)savePanelDidEnd: (NSSavePanel *)sheet
             returnCode: (int)returnCode
            contextInfo: (void *)contextInfo
{
    if (returnCode == NSOKButton)
    {
        NSString * path = [sheet filename];
        NSString * newUTType = [mSaveOptions imageUTType];
        CGImageRef image;

        image = [mImageView image];
        if (image)
        {
            NSURL * url = [NSURL fileURLWithPath: path];
            CGImageDestinationRef dest =
CGImageDestinationCreateWithURL((CFURLRef)url,
                        (CFStringRef)newUTType, 1, NULL);
            if (dest)
            {
                CGImageDestinationAddImage(dest, image,
                            (CFDictionaryRef)[mSaveOptions imageProperties]);
                CGImageDestinationFinalize(dest);
                CFRelease(dest);
            }
        } else
        {
            NSLog(@"*** saveImageToPath - no image");
        }
    }
}
```

5. Save the `Controller.m` file.
6. Open the `MainMenu.nib` file.
7. Double-click the MainMenu icon in the nib document window.
8. Open the File menu so you can see the menu items New, Open, and so on.
9. Control-drag from the Save menu item to the controller.
10. In the connections panel for the controller, choose `saveImage:`.
11. Save the nib file.
12. In Xcode, click Build and Go.

13. After the application launches, choose File > Save. You should see a Save panel appear with an accessory view that looks similar to the following.



Choose other items from the Format menu to see the options that are offered for each image format.



Supporting Zooming

To support zooming, you need to add menu commands or controls (or both) for zooming, and a method to respond to the commands.

To support zooming, follow these steps:

1. Add the following method signature to the `Controller.h` file, then save the file:

```
- (IBAction)doZoom: (id)sender;
```

2. Open the `Controller.m` file and add constants for the zoom factor.

```
#define ZOOM_IN_FACTOR  1.414214 // doubles the area
#define ZOOM_OUT_FACTOR 0.7071068 // halves the area
```

3. Add a method to handle commands to zoom out, zoom in, zoom to fit, and zoom to the actual size of the image.

```
- (IBAction)doZoom: (id)sender
{

    NSInteger zoom;
    CGFloat   zoomFactor;
```

```
    if ([sender isKindOfClass: [NSSegmentedControl class]])
        zoom = [sender selectedSegment];
    else
        zoom = [sender tag];

    switch (zoom)
    {
        case 0:
            zoomFactor = [mImageView zoomFactor];
            [mImageView setZoomFactor: zoomFactor * ZOOM_OUT_FACTOR];
            break;
        case 1:
            zoomFactor = [mImageView zoomFactor];
            [mImageView setZoomFactor: zoomFactor * ZOOM_IN_FACTOR];
            break;
        case 2:
            [mImageView zoomImageToActualSize: self];
            break;
        case 3:
            [mImageView zoomImageToFit: self];
            break;
    }
}
```

4. Open the `MainMenu.nib` file.
5. Double-click the MainMenu icon in the nib document window.
6. Create a new menu by dragging a Submenu Menu item from the Library and dropping it between the Edit and Format menu items.
7. Name the new menu Views.
8. Add menu items from the library to the Views menu so that you have enough for these zoom commands. Use the inspector to add the tags and key equivalents shown in the table.

Command	Tag	Key equivalent
Zoom Out	0	Command –
Zoom In	1	Command =
Zoom to Actual Size	2	
Zoom to Fit	3	

9. Control-drag from each item to the Controller icon. Then, in the connections panel for the controller, click `doZoom:`.
10. Save the `MainMenu.nib` file.
11. In Xcode, click Build and Go. Then make sure that the zoom commands operate as expected.
12. Quit the application and go back to the `MainMenu.nib` file in Interface Builder.

This time you'll revise the window user interface by adding zoom controls.

13. Change the image view size so that the top of the view is 50 pixels from the top of the window.

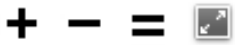
    You'll need this space to put zoom controls.

14. Drag a segmented control from the Library to the top-left side of the window.

15. In the Attributes inspector for the control set the number of segments to 4.

16. Double-click each segment to change its label and to use the inspector to add a tag. Use the commands and tags shown previously for the Views menu.

    If possible, you should add artwork, similar to the following, to the Xcode project so that you can use icons instead of text labels.



17. In the Size inspector, set the Control Size to small.

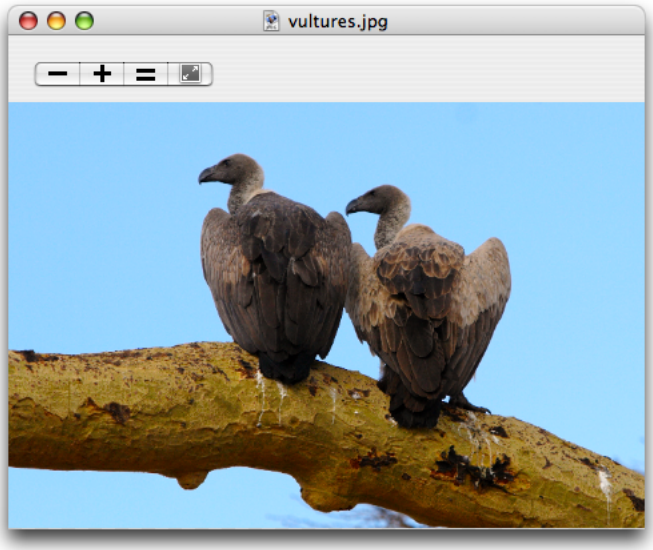
18. In the Attributes inspector, choose Select Any in the Mode pop-up menu.

19. Control-drag from the segmented zoom control to the Controller icon. Then, in the connections panel for the controller, click `doZoom:`.

20. Click the Info button in the nib document window. When the MainMenu.nib Info window opens, set the Deployment Target pop-up menu to Mac OS X v10.5.

21. Save the nib file.

22. In Xcode, click Build and Go. Then make sure that the zoom commands operate as expected.



Adding Image Editing Tools

The `IKImageView` class has built-in support for move, select, crop, rotate, and annotate tools. To set up the image editing application to include the tools for these actions, follow these steps:

1. In Xcode, add image files (Project > Add to Project), similar to the following, for each of the tool modes—move, select, crop, rotate, and annotate.

   You'll need to use an application such as Adobe Illustrator to create icons. See *Apple Human Interface Guidelines* for information on using icons in the interface.

2. Add the following method signature to the `Controller.h` file:

```
- (IBAction)switchToolMode: (id)sender;
```

3. Add a method to the `Controller.m` file to handle setting the tool mode.

```
- (IBAction)switchToolMode: (id)sender
{

    NSInteger newTool;
    if ([sender isKindOfClass: [NSSegmentedControl class]])
        newTool = [sender selectedSegment];
    else
       newTool = [sender tag];

    switch (newTool)
    {
        case 0:
            [mImageView setCurrentToolMode: IKToolModeMove];
            break;
        case 1:
            [mImageView setCurrentToolMode: IKToolModeSelect];
            break;
        case 2:
            [mImageView setCurrentToolMode: IKToolModeCrop];
            break;
        case 3:
            [mImageView setCurrentToolMode: IKToolModeRotate];
            break;
        case 4:
            [mImageView setCurrentToolMode: IKToolModeAnnotate];
            break;
    }
}
```

4. Save the `Controller.m` file.

5. Open the `MainMenu.nib` file.

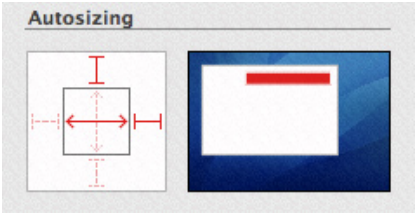
6. Add a submenu to the Views menu and name it Tools.

7. Add the following items to the Tools submenu, using the inspector to add the tag values.

Item	Tag
Move	0
Select	1
Crop	2
Rotate	3
Annotate	4

8. Control-drag from the Move menu item to the Controller icon. Choose `switchToolMode:` in the connections panel for the controller.
9. Repeat the previous step for each of the other items in the Tools menu.
10. Add a segmented control to the top-right side of the window so the control is aligned with the zoom controls.
11. In the Attributes inspector, set the number of segments to 5.
12. In the Size inspector, set the Size to small and the Width to 60.
13. Switch to the Attributes inspector.
14. For each segment, drag the appropriate icon from the Media pane of the Library, and set the segment to autosize.

    From left to right, the icons should represent Move, Select, Crop, Annotate, and Rotate.
15. Choose Select One from the Mode pop-up menu.
16. Make sure that Selected is checked for Segment 1.
17. In the Size inspector set the Autosizing struts and springs so they look as follows:



18. Control-drag from the segmented control to the controller and connect to the `switchToolMode:` action.
19. Save the nib file.
20. In Xcode, click Build and Go. Then test the Tools menu and the toolbar.

The move and rotate tools operate without any additional code on your part. The selection and crop tools copy the selected areas to the pasteboard. You need to provide code that implements pasting from the pasteboard. The annotate tool simply shows a colored circle. You need to write code that supports text entry and editing, and saves the annotation.

Supporting Opening Image Files

As it is now, your application opens a default image file. It would be greatly improved if it allowed the user to choose an image other than the default. Next you'll write an open image method that is invoked when the user chooses File > Open. You need to provide a selector that is invoked when the Open panel closes. If the user chooses an image, your `openImageURL:` method (which you already added to the application) is invoked.

Follow these steps to support letting the user open image files:

1. Add the following method signature to the `Controller.h` file:

```
- (IBAction)openImage: (id)sender;
```

2. Write an open image method.

   The method creates an instance of the `NSOpenPanel` class and defines the allowable filename extensions. It then calls a method that shows the Open panel.

```
- (IBAction)openImage: (id)sender
{
    NSOpenPanel * openPanel = [NSOpenPanel openPanel];
    NSString *    extensions = @"tiff/tif/TIFF/TIF/jpg/jpeg/JPG/JPEG/pdf/PDF";
    NSArray *     types = [extensions pathComponents];

    [openPanel beginSheetForDirectory: NULL
                                 file: NULL
                                types: types
                       modalForWindow: mWindow
                        modalDelegate: self
                       didEndSelector:
@selector(openPanelDidEnd:returnCode:contextInfo:)
                          contextInfo: NULL];

}
```

3. Write a selector method that's invoked when the Open panel terminates.

   If the user chooses an image, this method calls the `openImageURL:` method, passing to it the first item in the selected path.

```
- (void)openPanelDidEnd: (NSOpenPanel *)panel
             returnCode: (int)returnCode
            contextInfo: (void  *)contextInfo
{
    if (returnCode == NSOKButton)
    {
        [self openImageURL: [[panel URLs] objectAtIndex: 0]];
```

```
    }
}
```

4. Save the `Controller.m` file.
5. Double-click the MainMenu icon in the nib document window.
6. Click the File menu so you can see the items in it—New, Open, and so on.
7. Control-drag from the Open menu item to the controller and connect to the `openImage:` action in the connections panel.
8. Save the nib file.
9. In Xcode, click Build and Go.

   Choose File > Open and make sure that you can open an image.

Browsing Images

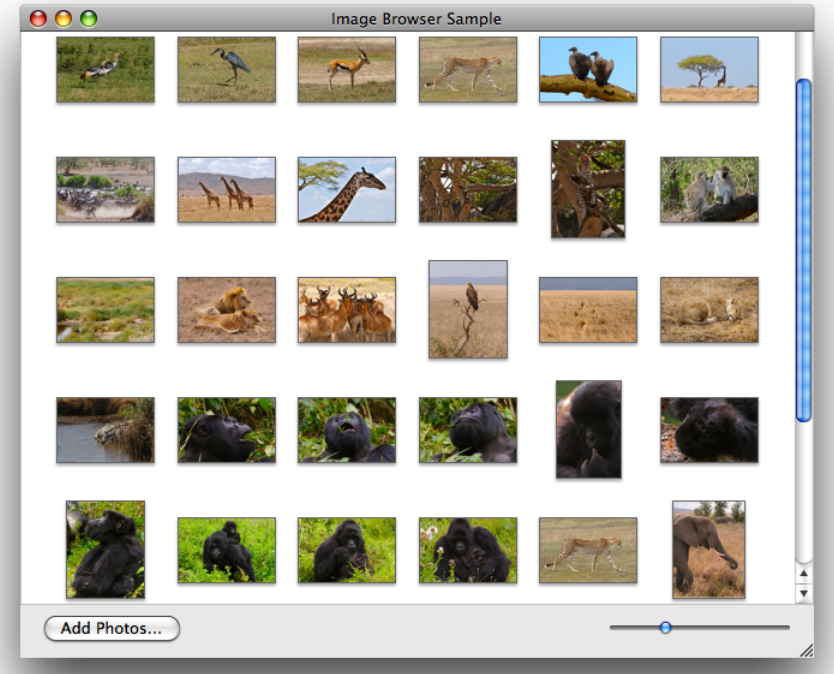
Applications such as iPhoto provide a rich user experience for viewing digital image collections. By using the `IKImageBrowserView` class and its related protocols—`IKImageBrowserDataSource`, `IKImageBrowserDelegate`, and `IKImageBrowserItem`—any application can support browsing large numbers of images efficiently.

This chapter describes the user interface provided by the `IKImageBrowserView` class, discusses the related protocols, lists the image representation types that the browser can display, and includes step-by-step instructions for creating an application that uses the image browser.

The Image Browser User Interface

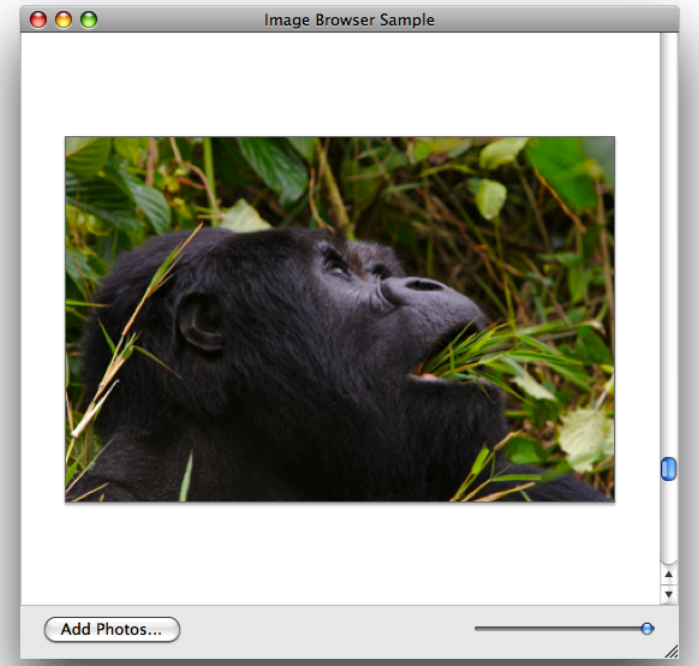
The `IKImageBrowserView` class provides a view that displays an array of images. The size of each image depends on the zoom value that you set for the image browser view. Typically, an application provides a control, such as the slider shown in Figure 3-1, that allows the user to control the zoom value.

Figure 3-1 The image browser view



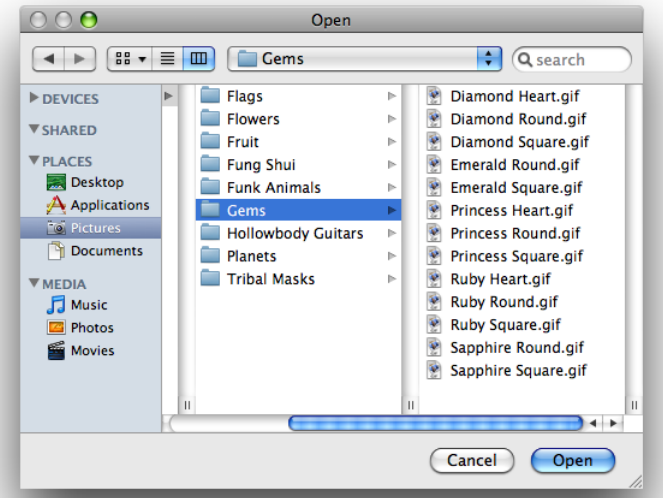
When fully zoomed in, the image browser displays a single image, as shown in Figure 3-2.

Figure 3-2 The image browser zoomed



You can set up the image browser view to support dragging images into the window to add them as well as dragging images within the window to rearrange them. In addition to drag and drop, your application should support adding photos through the `NSOpenPanel` class, as shown in Figure 3-3.

Figure 3-3 Adding images



The `IKImageBrowserView` class also supports scrolling (see Figure 3-2 (page 38)), which is necessary for large image sets or when the user has the zoom value set such that the image set doesn't fit in the view. Your application needs to take one of the following approaches:

- Embed the image browser in an `NSScrollView` object, which is the typical approach.
- Connect the image browser to an `NSScroller` object. This approach is a bit more efficient but doesn't allow auto-hiding of the scrollers.

The Requirements of an Image Browser Application

This section gives an overview of the Image Kit classes and protocols needed to write a full-featured image browser application and describes how such an application works. After reading this section you'll be ready to read the rest of this chapter, which provides step-by-step instructions for these programming tasks:

- "Displaying Images in an Image Browser" (page 42)
- "Supporting Zoom" (page 48)
- "Supporting Removing and Reordering Items" (page 49)
- "Supporting Drag and Drop" (page 51)
- "Setting Browser and Cell Appearance" (page 52)

The Image Kit Classes and Protocols for an Image Browser Application

An image browser application has a number of requirements. In addition to creating an instance of the `IKImageBrowserView` class, you'll need to provide a data source that provides the items to show in the image browser. You'll also need to implement the required methods of the `IKImageBrowserDataSource` and `IKImageBrowserItem` protocols.

The `IKImageBrowserDataSource` protocol defines methods for accessing the data source. You use this protocol to add and remove items, to move items, and to obtain information about individual items or groups. Your application must implement two methods:

- `numberOfItemsInImageBrowser:` returns the number of items in the image browser.
- `imageBrowser:itemAtIndex:` returns the object located at a specified index. The returned object must implement the required methods of the `IKImageBrowserItem` protocol.

Methods that support editing or that return group information are optional. See *IKImageBrowserDataSource Protocol Reference* for details.

The `IKImageBrowserItem` protocol defines required and optional methods that the browser view uses to access a particular item from the data source. You must implement three methods:

- `imageRepresentationType` returns the image representation type. An image browser view can handle a variety of image representations. You must specify which representation type the image browser view uses by returning the appropriate constant. Table 3-1 lists the constants.
- `imageUID` returns a string that uniquely identifies the data source item, such as the string that represents the path to the item.
- `imageRepresentation` returns the representation (such as the path to an image) for an item in the image browser view. The representation must match the image representation type.

You can optionally implement methods that return the image version, title, subtitle, and whether or not the item can be selected. See *IKImageBrowserItem Protocol Reference* for details.

Table 3-1 Image representations and types

Image representation constants	Image representation data types
`IKImageBrowserPathRepresentationType`	A path (`NSString` object)
`IKImageBrowserNSURLRepresentationType`	`NSURL` object
`IKImageBrowserNSImageRepresentationType`	`NSImage` object
`IKImageBrowserCGImageRepresentationType`	`CGImage` object
`IKImageBrowserCGImageSourceRepresentationType`	`CGImageSource` object
`IKImageBrowserNSDataRepresentationType`	`NSData` object
`IKImageBrowserNSBitmapImageRepresentationType`	`NSBitmapImageRep` object
`IKImageBrowserQTMovieRepresentationType`	`QTMovie` object

Image representation constants	Image representation data types
`IKImageBrowserQTMoviePathRepresentationType`	A path (`NSString` object) to a `QTMovie` object
`IKImageBrowserQCCompositionRepresentationType`	`QCComposition` object
`IKImageBrowserQCCompositionPath-RepresentationType`	A path (`NSString` object) to a `QCComposition` object
`IKImageBrowserQuickLookPathRepresentationType`	A path (`NSString` object) to load an object using the Quick Look framework
`IKImageBrowserIconRefRepresentationType`	`IconRef` object
`IKImageBrowserIconRefPathRepresentationType`	A path (`NSString` object) to an `IconRef` object

The `IKImageBrowserDelegate` informal protocol defines methods that respond to user events, such as a selection. See *IKImageBrowserDelegate Protocol Reference* for a description of the delegate methods.

How the Sample Image Browser Application Works

When the sample application that you'll build in the rest of this chapter launches, it allocates two mutable arrays, one that represents images that are displayed in the image browser view (image array) and another that represents images that need to be imported into the image browser view (import images array). Initially, both arrays are empty. The application then performs any setup work for the image browser view, such as setting the style of the cells and whether the movement of items is animated. After the setup is complete, the window with the image browser view opens without any images. The image browser view has a vertical scroll bar. The window has a button for importing images.

Clicking the Import Images button invokes an action to add images to the image browser view. The application uses an instance of the `NSOpenPanel` class to solicit a selection from the user. If the user makes a selection, the Open panel returns the selection as a path. This application allows users to choose either a file or a folder. A folder can contain files, other folders, or both files and folders. However, files are the only items that are added to the image browser view, which means that the application must traverse all paths until each resolves to a single file.

The application creates a data source object—an image object—for each file. The object has one instance variable, the path to the file (represented as an `NSString` object). Each image object is appended to the import images array. After all the objects are added to that array, the application appends that array to the images array. The application then removes all objects from the import images array so that the array can be ready to import more images should the user choose to do so. Finally the application calls the `reloadData` method of the `IKImageBrowserView` class to populate the image browser view with the images represented by the import images array.

The image browser application has five major parts to its implementation:

- The nib file. You need to add an Image Browser view (`IKImageBrowserView` class) to a window. You'll add a scroll view for the vertical scroller, and a button for importing images.

- The window controller. This is the main class. You create it in Xcode and then instantiate it in Interface Builder. The window controller is the data source of the image browser view. The controller manages the mutable array for the image objects that represent the items in the image browser view and the mutable array for imported image objects. In Interface Builder, you'll need to make the appropriate connections between the image browser view and the window controller.
- The `IKImageBrowserDataSource` protocol. You need to implement the required methods for this protocol: `numberOfItemsInImageBrowser:` and `imageBrowser:itemAtIndex:`.
- The `IKImageBrowserItem` protocol. You need to declare an interface for an image object that represents a single item in the image browser view. In the implementation for the image object, you need to provide the required methods for this protocol: `imageUID`, `imageRepresentationType`, and `imageRepresentation`.
- Image importing. You need to set up an Open panel and write code that traverses all folder paths until all paths are resolved to individual items.

The sections that follow provide step-by-step instructions for writing the image browser application.

Displaying Images in an Image Browser

This section shows how to create an application that displays images in an image browser. First, you'll set up the Xcode project, the project files, and the interface for the image browser controller. Then you'll add routines to the implementation. Finally, you'll set up the user interface in Interface Builder.

Setting Up the Project, Project Files, and the Controller Interface

Follow these steps to set up the project:

1. Open Xcode and create a Cocoa application named `Browse Images`.
2. Add the Quartz framework.

   For more details, see"Using the Image Kit in Xcode" (page 11).
3. Choose File > New File.
4. Choose "Objective-C Class NSWindowController subclass" and click Next.
5. Name the file `ImageBrowserController.m` and keep the option to create the header file. Then click Finish.
6. In the `ImageBrowserController.h` file, import the Quartz framework by adding this statement:

   ```
   #import <Quartz/Quartz.h>
   ```
7. Add `IBOutlet` outlet for the image browser and two mutable arrays .

   One array will hold paths to images that are currently displayed in the browser. The other array will hold paths to images that are about to be imported into the browser.

   ```
   @interface ImageBrowserController : NSWindowController {
   ```

```
    IBOutlet id mImageBrowser;
    NSMutableArray * mImages;
    NSMutableArray * mImportedImages;
}
@end
```

8. Add the following method signature:

```
- (IBAction) addImageButtonClicked:(id) sender;
```

9. Save the `ImageBrowserController.h` file.

Adding Routines to the Implementation File

Follow these steps to implement the main tasks of the image browser application:

1. Open the `ImageBrowserController.m` file.

2. In the implementation file, add an `awakeFromNib` method.

   This method allocates and initializes the images and imported images arrays. For visual interest, the method also sets the image browser to animate as it updates.

```
- (void) awakeFromNib
{
    mImages = [[NSMutableArray alloc] init];
    mImportedImages = [[NSMutableArray alloc] init];

    [mImageBrowser setAnimates:YES];
}
```

3. Add a `dealloc` method.

   The method needs to release the two image arrays.

```
- (void) dealloc
{
    [mImages release];
    [mImportedImages release];
    [super dealloc];
}
```

4. Add a method for updating the data source for the image browser controller.

   This method needs to add the recently imported items to the image objects array, then empty the imported images array. Finally it reloads the image browser, which causes the image browser to update the display with the recently added images.

```
- (void) updateDatasource
{
    [mImages addObjectsFromArray:mImportedImages];
    [mImportedImages removeAllObjects];
    [mImageBrowser reloadData];
}
```

5. Implement the two required methods of the image browser data source protocol.

   The number of items in the image browser is simply the number of items in the images array. An item's index is simply its index in the images array.

```
- (int) numberOfItemsInImageBrowser:(IKImageBrowserView *) view
{
    return [mImages count];
}

- (id) imageBrowser:(IKImageBrowserView *) view itemAtIndex:(int) index
{
    return [mImages objectAtIndex:index];
}
```

6. In the `IKImageBrowserController.m` file, immediately after the import statement, create an interface for a data source object.

   The data source object defines one instance variable, a string that is a path to an item to display.

```
@interface MyImageObject : NSObject{
    NSString * mPath;
}
@end
```

7. In the same file, create the implementation for the data source object `MyImageObject`. In the next three steps, you'll be entering methods between the the following two statements, as you would normally for a Cocoa class.

```
@implementation MyImageObject
 // Methods here
@end
```

8. In the `MyImageObject` implementation, add a `dealloc` method that releases the path object.

```
- (void) dealloc
{
    [mPath release];
    [super dealloc];
}
```

9. Write a method that sets the path object.

```
- (void) setPath:(NSString *) path
{
    if(mPath != path){
        [mPath release];
        mPath = [path retain];
    }
}
```

10. Implement the three required methods of the `IKImageBrowserItem` protocol.

    The data source is a file path representation and its unique identifier is the path itself.

```
- (NSString *)  imageRepresentationType
{
    return IKImageBrowserPathRepresentationType;
```

```
}

- (id)  imageRepresentation
{
    return mPath;
}

- (NSString *) imageUID
{
    return mPath;
}
```

11. Now that you are done with the MyImageObject implementation, you need to add a method to the ImageBrowserController implementation. This method is invoked when the user clicks an Import Images button.

    This method calls a routine that displays the Open panel and returns the path chosen by the user. If there is a path, the method sets up an independent thread, calling the `addImagesWithPaths:` method, which you'll write next. You'll write an open files routine later. Check to make sure that you add the following method in the ImageBrowserController implementation,

```
- (IBAction) addImageButtonClicked:(id) sender
{
    NSArray *path = openFiles();

    if(!path){
        NSLog(@"No path selected, return...");
        return;
    }
  [NSThread detachNewThreadSelector:@selector(addImagesWithPaths:) toTarget:self
 withObject:path];
}
```

12. Write a method that adds an array of paths to the data source.

    The method parses all paths in the `paths` array and adds them to a temporary array. (You'll write the `addImagesWithPath:` method in the next step. Note this is *path*, singular.) It then updates the data source in the main thread.

    Add this method after the `addImageButtonClicked:` method.

```
- (void) addImagesWithPaths:(NSArray *) paths
{
    int i, n;

    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    [paths  retain];

    n = [paths count];
    for(i=0; i<n; i++){
        NSString *path = [paths objectAtIndex:i];
        [self addImagesWithPath:path recursive:NO];
    }

    [self performSelectorOnMainThread:@selector(updateDatasource)
                           withObject:nil
                        waitUntilDone:YES];
```

```
    [paths release];
    [pool release];
}
```

13. Write the method that adds images at a path.

    This method checks to see if the path identifies a directory or a file. If the path is a directory, the code parses the directory content and calls the `addImagesAtPath:` method. If the path is to a file, the code calls a method that adds a single image to the imported images array. You'll write the `addAnImageWithPath:` method next.

    Note this method has an option to enable or disable recursion. Enabling recursion allows you to traverse nested folders to retrieve the individual items in each folder.

    Add this method before the `addImagesWithPaths:` method.

```
- (void) addImagesWithPath:(NSString *) path recursive:(BOOL) recursive
{
    int i, n;
    BOOL dir;

    [[NSFileManager defaultManager] fileExistsAtPath:path isDirectory:&dir];
    if(dir){
        NSArray *content = [[NSFileManager defaultManager]
                            directoryContentsAtPath:path];
        n = [content count];
       for(i=0; i<n; i++){
            if(recursive)
               [self addImagesWithPath:
                     [path stringByAppendingPathComponent:
                            [content objectAtIndex:i]]
                            recursive:NO];
            else
              [self addAnImageWithPath:
                     [path stringByAppendingPathComponent:
                           [content objectAtIndex:i]]];
        }
    }
    else
        [self addAnImageWithPath:path];
}
```

14. Write a method that adds a single image to the imported images array.

    The method creates an image object and adds the path to the object. The code then adds the image object to the imported images array.

    Add this method before the `addImagesWithPath:recursive:` method.

```
- (void) addAnImageWithPath:(NSString *) path
{
    MyImageObject *p;

    p = [[MyImageObject alloc] init];
    [p setPath:path];
    [mImportedImages addObject:p];
    [p release];
```

```
}
```

15. Write an open files routine that displays an `NSOpenPanel` object and retrieves the path chosen by the user.

    Place this routine at the top of the `ImageBrowserController.m` file, immediately after the import statement.

```
static NSArray *openFiles()
{
    NSOpenPanel *panel;

    panel = [NSOpenPanel openPanel];
    [panel setFloatingPanel:YES];
    [panel setCanChooseDirectories:YES];
    [panel setCanChooseFiles:YES];
    int i = [panel runModalForTypes:nil];
    if(i == NSOKButton){
        return [panel filenames];
    }

    return nil;
}
```

16. Build the Project.

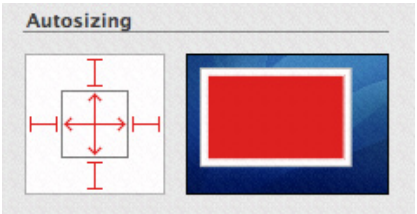
    This ensures that Interface Builder detects the action that you added.

17. Save the `IKImageBrowserController.m` file.

Creating the User Interface

Set up the user interface in Interface Builder by following these steps:

1. Double-click the `MainMenu.nib` file to open Interface Builder.
2. Choose File > Synchronize With Xcode.
3. If it's not already open, double-click the Window icon in the nib document window.
4. Drag a Image Browser View from the Library to the window and resize the view so that you leave space at the bottom of the window for a button.
5. In the Size inspector for the view, make sure that the Autosizing springs and struts look as follows:



6. Choose Layout > Embed Objects In > Scroll View.
7. In the Scroll View Attributes inspector, leave Show Vertical Scroller selected but deselect Show Horizontal Scroller.

   This will allow users to scroll easily through large numbers of images.
8. In the Size pane, set the Autosizing springs and struts so they look the same as those shown in Step 5.
9. Drag an Object (`NSObject`) from the Library to the nib document window.
10. In the Identity inspector, choose ImageBrowserController from the Class pop-up menu.
11. Control-drag from the ImageBrowserController icon to the title bar of the window. Then connect to `window` in the connections panel.
12. Control-drag from the ImageBrowserController icon to the `IKImageBrowserView` view. Then connect to `mImageBrowser` in the connections panel.
13. Control-drag from the `IKImageBrowserView` view to the ImageBrowserController icon. Then connect to _dataSource in the connections panel.
14. Control-drag from the window icon to the controller icon. Then connect to `delegate` in the connections panel.
15. Drag a Push Button from the library to the lower right portion of the window and label it Import Images.
16. In the Size inspector, set Autosizing to have outer struts on the left and bottom.
17. Control-drag from the Import Images button to the the ImageBrowserController icon and connect to the `addImageButtonClicked:` action in the connections panel.
18. Save the nib file.
19. In Xcode, click Build and Go.

    Click the Import Images button and make sure that the image browser works.

Supporting Zoom

Next you'll add the ability for the user to zoom images. You'll define zoom factors, add controls to the interface, and then add a zoom method that's invoked by the controls.

To add the ability for users to zoom images in an image browser:

1. Add a zoom method.

   This method responds to the zoom controls in the user interface. You'll add the controls later. The method needs to set the zoom value and then signal the need to update the browser display.

```
- (IBAction) zoomSliderDidChange:(id)sender
{
    [mImageBrowser setZoomValue:[sender floatValue]];
```

```
    [mImageBrowser setNeedsDisplay:YES];
}
```

2. Add the method signature to the `ImageBrowserController.h` file.

```
- (IBAction) zoomSliderDidChange:(id)sender;
```

3. Save the `ImageBrowserController.h` and `ImageBrowserController.m` files.
4. Double-click the `MainMenu.nib` file to open Interface Builder.
5. Drag a Horizontal Slider from the Library to the browser window and position it in the lower left of the window.
6. In the Size inspector for the slider and set the Autosizing struts appropriately.
7. Set the size to Mini.
8. In the Attributes inspector set the State to Continuous to cause the action method to send its state continuously while the mouse is down.
9. Set the slider minimum and maximum values.

   Enter `0` for the minimum value and `1.0` for the maximum value.
10. Control-drag from the slider to the ImageBrowserController icon and in the connections panel choose `zoomSliderDidChange:`.
11. Save the `MainMenu.nib` file.
12. In Xcode, click Build and Go.

    Try the zoom controls and make sure they work.

Supporting Removing and Reordering Items

The image browser will be far more useful if users can remove items and reorder them. You need to set the option to allow reordering. Then you need to implement the methods defined by the `IKImageBrowserDataSource` protocol that support editing items.

Follow these steps to support removing and reordering:

1. Open the `ImageBrowserController.m` file.
2. Implement the `IKImageBrowserDataSource` protocol method for removing items.

```
- (void) imageBrowser:(IKImageBrowserView *) view removeItemsAtIndexes:
(NSIndexSet *) indexes
{
    [mImages removeObjectsAtIndexes:indexes];
}
```

3. Implement the `IKImageBrowserDataSource` protocol method to move items from one location to another.

   This method first removes items from the data source and stores them temporarily in an array. Then it inserts the removed items into the images array at the new location.

```
- (BOOL) imageBrowser:(IKImageBrowserView *) view  moveItemsAtIndexes: (NSIndexSet
 *)indexes toIndex:(unsigned int)destinationIndex
{
      int index;
      NSMutableArray *temporaryArray;

      temporaryArray = [[[NSMutableArray alloc] init] autorelease];
      for(index=[indexes lastIndex]; index != NSNotFound;
                        index = [indexes indexLessThanIndex:index])
      {
          if (index < destinationIndex)
              destinationIndex --;

          id obj = [mImages objectAtIndex:index];
          [temporaryArray addObject:obj];
          [mImages removeObjectAtIndex:index];
      }

      // Insert at the new destination
      int n = [temporaryArray count];
      for(index=0; index < n; index++){
          [mImages insertObject:[temporaryArray objectAtIndex:index]
                        atIndex:destinationIndex];
      }

      return YES;
}
```

4. In the `awakeFromNib` method, set the image browser to allow reordering.

   After modification, the method should look as follows:

```
- (void) awakeFromNib
{
    mImages = [[NSMutableArray alloc] init];
    mImportedImages = [[NSMutableArray alloc] init];

    [mImageBrowser setAllowsReordering:YES];
    [mImageBrowser setAnimates:YES];

}
```

5. In Xcode, click Build and Go.

   Add images to the browser. Then try deleting a few items. Select several items and move them to a new location.

Note: If you want to support dragging items, you should also implement `imageBrowser:writeItemsAtIndexes:toPasteboard:`.

Supporting Drag and Drop

It is convenient for users to be able to drag items directly to the browser. Drag and drop requires that you set a dragging destination delegate and implement three methods of the `NSDraggingDestination` protocol. (See *Drag and Drop Programming Topics for Cocoa*.)

To support the ability for users to drag and drop images, follow these steps:

1. In the `awakeFromNib` method, set the dragging destination delegate for the image browser.

   After modification, the method should look as follows:

```
- (void) awakeFromNib
{
    mImages = [[NSMutableArray alloc] init];
    mImportedImages = [[NSMutableArray alloc] init];
    [mImageBrowser setAllowsReordering:YES];
    [mImageBrowser setAnimates:YES];
    [mImageBrowser setDraggingDestinationDelegate:self];
}
```

2. Implement the `performDragOperation:` method of the `NSDraggingDestination` protocol.

   This method looks for paths in the pasteboard. If there are paths, the method retrieves them, adds them to the data source, then reloads the image browser.

```
- (BOOL) performDragOperation:(id <NSDraggingInfo>)sender
{
    NSData *data = nil;
    NSString *errorDescription;
    NSPasteboard *pasteboard = [sender draggingPasteboard];

    if ([[pasteboard types] containsObject:NSFilenamesPboardType])
        data = [pasteboard dataForType:NSFilenamesPboardType];
    if(data){
        NSArray *filenames = [NSPropertyListSerialization
            propertyListFromData:data
                mutabilityOption:kCFPropertyListImmutable
                          format:nil
                errorDescription:&errorDescription];
        int i;
        int n = [filenames count];
        for(i=0; i<n; i++){
            [self addImagesWithPath:[filenames objectAtIndex:i] recursive:NO];
        }
        [self updateDatasource];
    }

    return YES;
}
```

3. Implement the `draggingEntered:` method of the `NSDraggingDestination` protocol.

```
- (NSDragOperation)draggingEntered:(id <NSDraggingInfo>)sender
{
    return NSDragOperationCopy;
}
```

4. Implement the `draggingUpdated:` method of the `NSDraggingDestination` protocol.

```
- (NSDragOperation)draggingUpdated:(id <NSDraggingInfo>)sender
{
    return NSDragOperationCopy;
}
```

5. In Xcode, click Build and Go.

   Drag some images to the image browser. Then add some more images. Note that images dragged to the browser are added at the end.

Setting Browser and Cell Appearance

The `IKImageBrowserView` class provides several methods that control the image browser and browser item appearance. You can set the display style for cells so that the cells are shadowed, outlined, or appear with a title or subtitle. You can specify whether to constrain the size of items to their original size. You can also set the background color of the image browser and a number of other options, all of which are detailed in *IKImageBrowserView Class Reference*.

You set cell appearance by including the appropriate statements in the `awakeFromNib:` method, such as:

```
[mImageBrowser setCellsStyleMask:IKCellsStyleOutlined | IKCellsStyleShadowed];
[mImageBrowser setConstrainsToOriginalSize:YES];
```

If you support groups, you can set bezel and disclosure styles on a per-group basis—the group style is not a property of the image browser.

Showing Slides

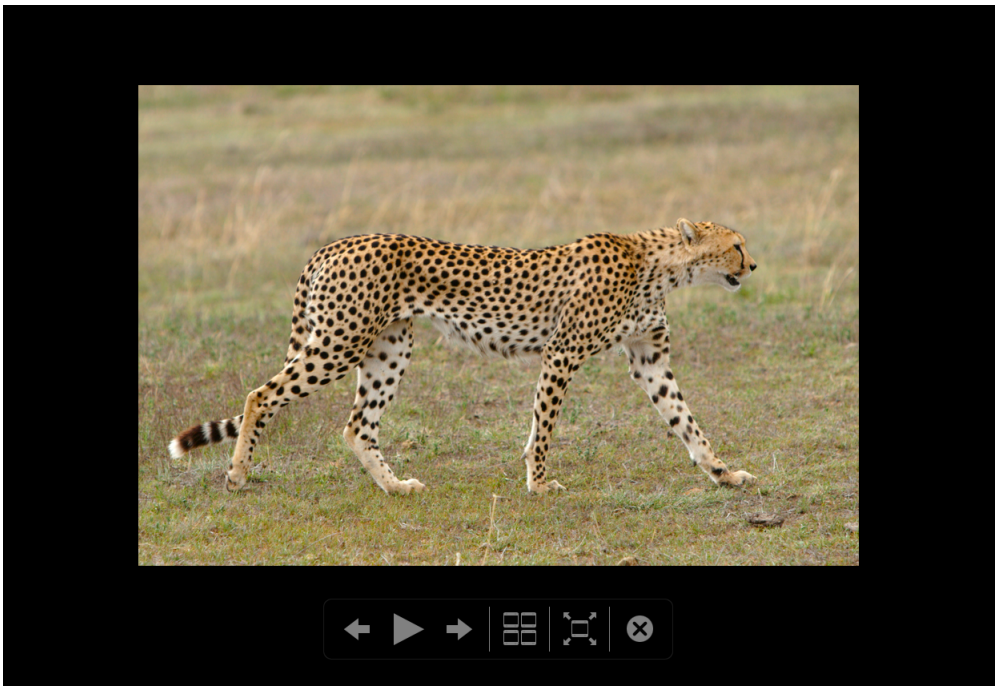
Several applications provided with Mac OS X have a slideshow feature built in, including Mail and Preview. Starting with Mac OS X v10.5, any application can provide slideshow support by using the `IKSlideshow` class. In addition to digital images, a slideshow can display pages from PDF documents, QuickTime movies, Quartz Composer compositions, and custom document formats supported through the Quick Look framework.

This chapter describes the user interface provided by the `IKSlideshow` class and provides step-by-step instructions for creating a slideshow application.

The Slideshow User Interface

When a slideshow runs, by default it opens to the first image provided by the slideshow data source. The slideshow uses the entire screen, as shown in Figure 4-1. The controls at the bottom of the screen allow the user to move forward and backward, pause and play, view an index layout of all items in the show, fit the images to the screen, and close the slideshow.

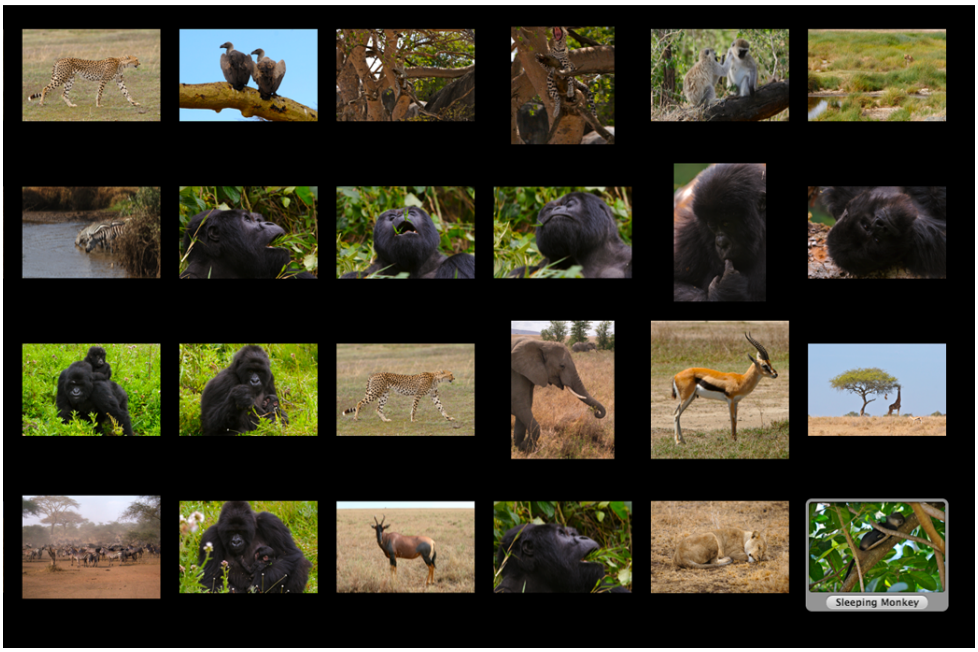
Figure 4-1 A slideshow with controls



The user can override the automatic advance by clicking the pause control or by manually moving through the show using the onscreen controls or the arrow keys on the keyboard.

The index sheet (see Figure 4-2) lets the user see many slides at the same time and to move through them using the arrow keys or by clicking an image. If the user double-clicks an image or presses Return, the slideshow resumes, starting with the selected image.

Figure 4-2 The index sheet



Writing a Slideshow Application

This section describes how to write an application that supports slideshows. First you'll see what needs to be done to support slideshows in any application. Then you'll take a look at a specific implementation of a simple slideshow application and the steps required to write the application.

A slideshow application requires a shared instance of the `IKSlideshow` class and a data source. You obtain the shared instance using the method `sharedSlideshow`. You run the slideshow using the method `runSlideshowWithDataSource:options:`.

You must implement two methods of the `IKSlideshowDataSource` protocol:

- `numberOfSlideshowItems` returns the number of items in the slideshow.
- `slideshowItemAtIndex:` returns the slideshow item at the given index. The item can be specified as an image object (`CGImage`, `NSImage`), a string that represents a path to a file, or as a URL (`NSURL`).

The `IKSlideshow` class defines a number of other methods that are useful depending on the nature of your application. For example, you can export slideshow items to an application, reload data, run or stop a slideshow, set a time interval that determines when the slideshow starts to play automatically, and get the index of the item that's currently playing.

The `IKSlideshowDataSource` protocol defines several optional methods that you might want to implement, such as methods that allow your application to take action at certain points in the slideshow: `slideshowDidStop`, `slideshowWillStart`, and `slideshowDidChangeCurrentIndex:`. See *IKSlideshowDataSource Protocol Reference* for a complete descriptions of the methods in this protocol.

How the Simple Slideshow Application Works

The slideshow application that you'll create in this section is quite simple. It implements the required methods of the the `IKSlideshowDataSource` protocol and allows the user to stop the slideshow and reload another set of images to view. It provides pathnames for each item in the slideshow. Although the paths could lead to any image file format supported by Quartz 2D and the Image I/O framework, including paths to PDF documents, for this application you'll restrict the datasource mode to images. In this mode, Image Kit displays only the first page of any PDF documents that are mixed in with the images in a folder.

The most complicated part of this application is the code that fetches the pathnames for each item in the slideshow. The path-fetching code must be able to accept a folder path and resolve any folder path to the individual items in the path. The user should be able to provide a path to a folder that contains both individual items and other folders, but the resulting slideshow should show only individual files. In other words, the slideshow should not display folder icons, but rather, the items in the folder.

When launched, the simple slideshow application presents the user with an Open panel (dialog) for choosing a folder. The user closes the slideshow by clicking the Close icon provided by the slideshow controls shown in Figure 4-1 (page 53). The user can start another slideshow by choosing File > Choose Images, which presents the Open panel again. The slideshow application terminates when the user chooses Quit from the menu or presses Command-Q.

The user interface for this application is very simple—a Choose Images menu item, which you'll create using Interface Builder. All other user interface elements are provided by the `IKSlideshow` class or other parts of the system.

To control the application, you'll use the `NSWindowController` class. Your controller will have:

- A mutable array for holding the paths to each of the items in the slideshow
- An instance of the `IKSlideshow` class

Writing the Simple Slideshow Application

Now that you have an overview of the slideshow application, it's time to write the code. First you'll set up the Xcode project, the project files, and the controller interface. Then you'll add the necessary routines to the implementation file. Finally, you'll create the user interface in Interface Builder.

Setting Up the Project, Project Files, and the Controller Interface

Follow these steps to set up the project:

1. Create a Cocoa application and name it `Simple Slideshow`.
2. Add the Quartz framework to the project.

   For details, see "Using the Image Kit in Xcode" (page 11).

3. Choose File > New File.
4. Select "Objective-C NSWindowController subclass" and click Next.
5. Name the file `SlideshowController.m` and keep the option to create the header file. Then click Finish.
6. In the `SlideshowController.h` file, import the Quartz framework by adding this statement:

```
#import <Quartz/Quartz.h>
```

7. Add an instance variable for the slideshow and a mutable array for image paths.

   The interface should look as follows:

```
@interface SlideshowController : NSWindowController {
    IKSlideshow        *mSlideshow;
    NSMutableArray *mImagePaths;
}
@end
```

8. Save the `SlideshowController.h` file.

Adding Routines to the Implementation File

Implement the slideshow routines by following these steps:

1. Open the `SlideshowController.m` file.
2. In the implementation file, add an `awakeFromNib` method.

   The method first obtains a shared instance of the slideshow and retains it. Next it loads images. You'll write the `loadImages` method and its supporting methods later on. The `loadImages` method, if successful, will add image paths to the `mImagePaths` array. If paths are added, then the `awakeFromNib` method runs the slideshow, using the paths as the data source.

   The following `awakeFromNib` method sets the mode to images only. If there are PDF documents in any folders that you add, Image Kit renders only the first page.

   Note that the method does not set any options for the slideshow. However, you can set any of the options specified by the slideshow option key constants defined in *IKSlideshow Class Reference*.

```
- (void)awakeFromNib
{
     mSlideshow = [[IKSlideshow sharedSlideshow] retain];
     [self loadImages];
     if ([mImagePaths count] > 0)
        [mSlideshow runSlideshowWithDataSource:(id<IKSlideshowDataSource>)self
                     inMode: IKSlideshowModeImages
                     options: NULL];
}
```

3. Next you need to implement the two required methods of the `IKSlideshowDataSource Protocol` protocol.

The `numberOfSlideshowItems` method simply returns a count the number of paths in the `mImagePaths` array. The `slideshowItemAtIndex:` method returns the index of the `mImagePaths` array for the item in question.

```
- (NSUInteger)numberOfSlideshowItems
{

    return [mImagePaths count];
}


- (id)slideshowItemAtIndex: (NSUInteger)index
{
   int i;
   i = index % [mImagePaths count];
    return [mImagePaths objectAtIndex: i];
}
```

4. Implement a `loadImages` method.

   This and the next few steps are where most of the work is done in the slideshow application. The `loadImages` method allocates and initializes the `mImagePaths` array, if necessary. Otherwise, the method removes all objects in preparation for adding new slideshow items.

   Next the method calls a function to open files. You'll write this routine in the next step. The `openFiles` routine invokes the Open panel (`NSOpenPanel` class) and returns an array of paths or `NULL` if the user cancels the Open panel. If there is an array of paths, the `loadImages` method iterates through the paths and calls the method `addImagesFromPath:`, which you'll write in a moment.

```
- (void)loadImages
{
    if (NULL == mImagePaths)
    {
        mImagePaths = [[NSMutableArray alloc] init];
    } else {
      [mImagePaths removeAllObjects];
    }
    NSArray * array = openFiles();
    if (array != NULL)
    {
        NSEnumerator *  enumerator;
        NSString *      path;

        enumerator = [array objectEnumerator];
        while (path = [enumerator nextObject])
        {
                [self addImagesFromPath: path];
        }
    }
}
```

5. Next you need to include an `openFiles` routine, which you'll add in the `SlideshowController.m` file, but prior to the implementation statement.

   The `openFiles` routine creates an instance of the `NSOpenPanel` class, sets up options to allow the user to choose directories as well as files, runs the panel, and returns either an array of the file names chosen by the user, or `nil` if the user clicks the Cancel button.

```
static NSArray *openFiles()
{
    NSOpenPanel *panel;

    panel = [NSOpenPanel openPanel];
    [panel setFloatingPanel:YES];
    [panel setCanChooseDirectories:YES];
    [panel setCanChooseFiles:NO];
    int i = [panel runModalForTypes:nil];
    if(i == NSOKButton){
        return [panel filenames];
    }
        return nil;
}
```

6. Write the `addImagesFromPath:` method called by the `loadImages` method.

   The first task for this method is to obtain an array of all the string objects contained in the given path. Then the method iterates through the string objects. If the string does not have a filename extension, then it represents a path. In this case the method calls itself to recursively resolve the path into individual items. If the string has an extension, then it represents an individual item that can be added to the image path array.

```
- (void)addImagesFromPath: (NSString *)path
{
   NSArray * array  = [[NSFileManager defaultManager]
                  directoryContentsAtPath: path];


    NSEnumerator *  enumerator;
    NSString *      imagePath;

    enumerator = [array objectEnumerator];
    while (imagePath = [enumerator nextObject])
    {
        if ([[[imagePath pathExtension] lowercaseString] isEqualToString: @""])
        {
             [self addImagesFromPath: [NSString stringWithFormat: @"%@/%@/",
                                    path, imagePath]];

      } else
        {
            [mImagePaths addObject: [NSString stringWithFormat: @"%@/%@",
                                   path, imagePath]];
        }
    }
}
```

7. You need to write a choose images method that is invoked when the user chooses that command.

   Recall that the user can stop a slideshow and start another by choosing File > Choose Images. You'll add this menu item and connect to this action later on.

   This method first calls the `loadImages` method to fetch and add new slideshow items. If there are items to add (that is, the user did not click Cancel in the Open panel), the method reloads the slideshow data and then runs the slideshow.

```
- (IBAction) chooseImages:(id) sender
```

```
{

    [self loadImages];
    if ([mImagePaths count] > 0) {
        [mSlideshow reloadData];
       [mSlideshow  runSlideshowWithDataSource: (id<IKSlideshowDataSource>)self
            inMode: IKSlideshowModeImages
            options: NULL];
    }
}
```

8. You need to provide a `dealloc` method that releases the image paths array and the slideshow object.

```
- (void)dealloc
{
  [mImagePaths release];
  [mSlideshow release];
  [super dealloc];
}
```

9. Save the `SlideshowController.m` file.

10. Open the `SlideshowController.h` file and add these method declarations:

```
- (IBAction)chooseImages:(id)sender;

- (NSUInteger)numberOfSlideshowItems;
- (id)slideshowItemAtIndex: (NSUInteger)index;
- (void)loadImages;
```

11. Save the `SlideshowController.h` file.

Creating the User Interface

Set up the user interface in Interface Builder by following these steps:

1. Double-click the `MainMenu.nib` file to open Interface Builder.
2. Choose File > Synchronize With Xcode.
3. Delete the window icon in the nib document window.
4. Double-click the MainMenu icon in the nib document window.
5. Drag a Menu Item (`NSMenuItem`) from the Library to the File menu and name the item Choose Images.
6. Drag an Object (`NSObject`) from the Library to the nib document window.
7. In the Identity inspector, type SlideshowController in the Name field. THen select `SlideshowController` from the Class pop-up menu.
8. Control-drag from the Choose Images menu item to the SlideshowController and in the connections panel choose `chooseImages:`.
9. Save the nib file.

10. In Xcode, click Build and Go. Then test your application to make sure it works.

    You might get warnings if you placed the methods into your file in the wrong order.

    Check to make sure that you can run a slideshow. Exit the slideshow, and then start another by choosing File > Choose Images.

Taking Snapshots and Setting Pictures

More and more applications provide the user with the opportunity to take a snapshot or provide an image to use as a personal identifier, such as the buddy picture in iChat. The `IKPictureTaker` class provides such support. Its simple user interface lets users choose, crop, or rotate an existing image, or take a snapshot using an iSight camera or other digital camera.

This chapter describes the user interface provided by the `IKPictureTaker` class and gives step-by-step instructions for creating an application that uses the picture taker.

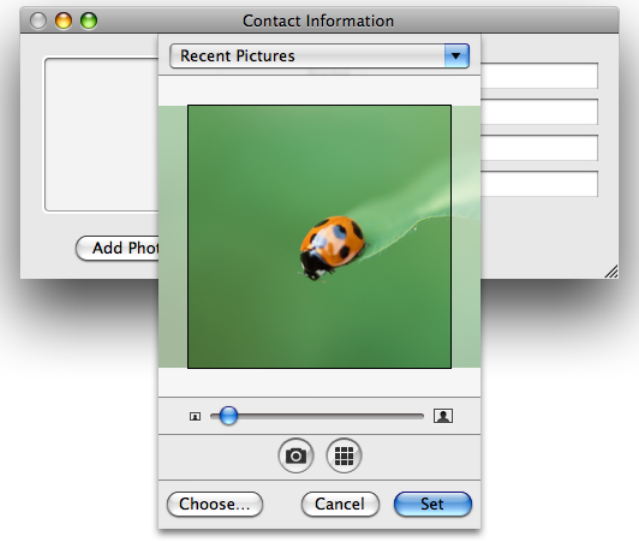
The Picture Taker User Interface

The picture taker panel can appear as a modal window or a sheet. Figure 5-1 shows the picture taker sheet as it appears the first time the user opens it. The application can set the default image. In this example, it is the ladybug image that comes with Mac OS X.

The user has a number of options when the presented with the picture taker:

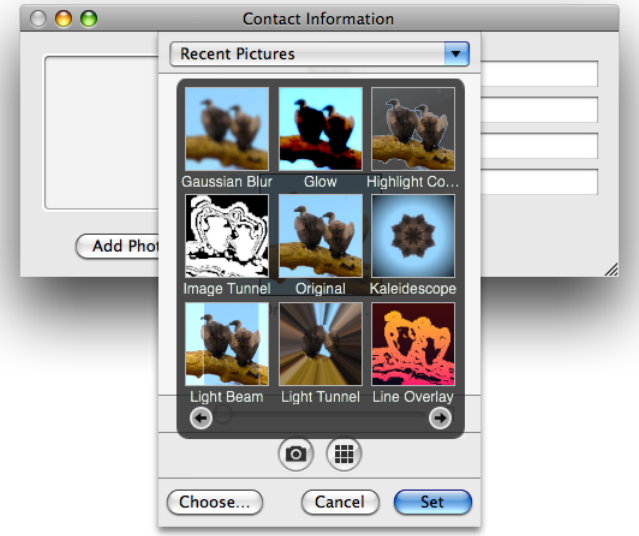
- Accept the default image
- Drag an image directly to the picture taker
- Click Choose and select an image through the Open panel as shown in Figure 5-3 (page 62).
- Click the camera button to take a snapshot

Figure 5-1 The picture taker as a sheet



The user can modify the image by using the slider to size and crop it. If the picture taker has an effects button (located next to the camera button), the user can click it to apply an effect, as shown in Figure 5-2. The effects button is optional. You'll see how to turn on the option in "Using the Picture Taker in the Contact Information Application" (page 65).

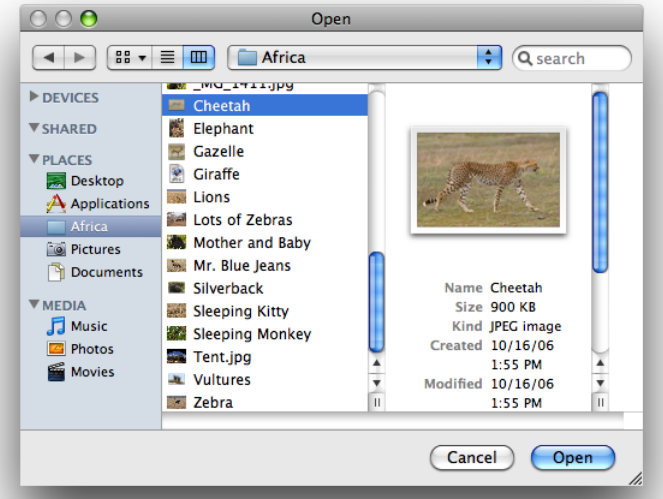
Figure 5-2 Adding an effect



The user clicks the Set button to accept an image and close the picture taker, or the Cancel button to close the sheet without choosing an image.

Figure 5-3 shows the panel that appears when the user clicks the Choose button.

Figure 5-3 Choosing an image with the Open panel



If the user previously chose images for this application, the Recent Pictures menu is populated with those images, as shown in Figure 5-4. The user can choose any of the recent items.

Figure 5-4 The Recent Pictures menu

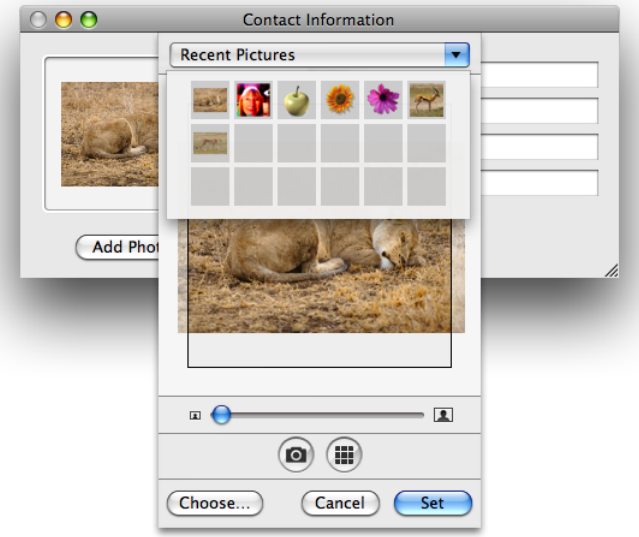
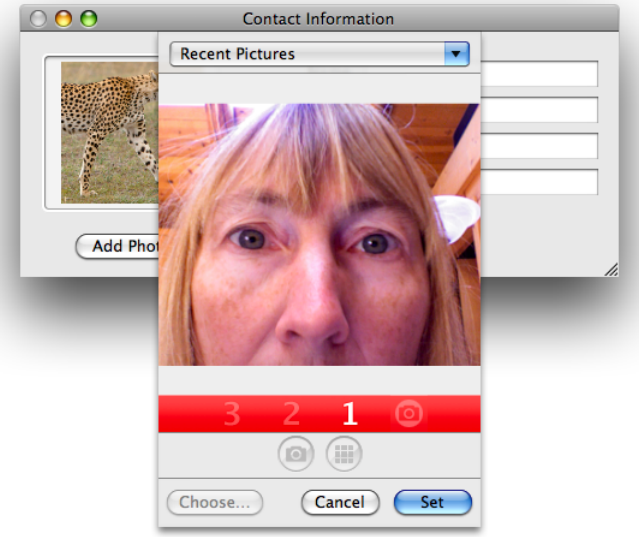


Figure 5-5 shows the user interface that appears after the user clicks the camera button on a computer that has an iSight or other digital camera connected to it. Whatever is positioned in front of the camera appears in the window. The numbers below the window, along with sound, provide a countdown prior to the camera capturing the image.

Figure 5-5 Taking a snapshot



Using the Picture Taker in an Application

This section shows how to write an application that uses a picture taker. First you'll get an overview of how the picture taker works in general, then you'll take a look at how to implement the picture taker in an application that lets the user set up personal contact information.

To display the picture taker in any application, you first need to create a shared instance by calling the `pictureTaker` method of the `IKPictureTaker` class.

```
IKPictureTaker *pictureTaker = [IKPictureTaker pictureTaker];
```

You can customize the picture taker appearance and behavior by using the `setValue:forKey:` method and providing a key constant (defined by the picture taker) along with the appropriate value. For example, you can control the size of the crop area using the key `IKPictureTakerCropAreaSizeKey` and providing a size (as an `NSValue` object). See *IKPictureTaker Class Reference* for a complete list of the options you can set.

You can launch the picture taker as a standalone window using this method:

```
[pictureTaker beginPictureTakerWithDelegate:self
didEndSelector:@selector(pictureTakerDidEnd:returnCode:contextInfo:)
contextInfo:nil];
```

or as a sheet, using this method:

```
[myPictureTaker beginPictureTakerSheetForWindow:aWindow withDelegate:self
didEndSelector:@selector(pictureTakerDidEnd:returnCode:contextInfo:)
contextInfo:nil];
```

You need to provide a selector method that has a method signature that matches the following:

```
- (void) pictureTakerDidEnd:(IKPictureTaker *) picker
```

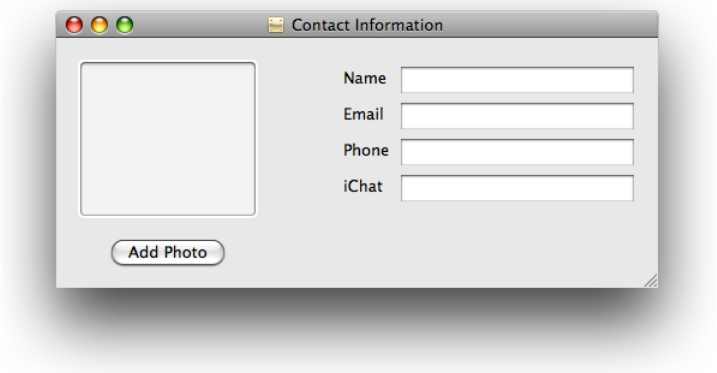
Typically you would retrieve the output image in this callback by using the `outputImage` method of the `IKPictureTaker` class. The output image represents the edited image; the `inputImage` method retrieves the unedited image.

```
- (void) pictureTakerDidEnd:(IKPictureTaker *) picker
    returnCode:(NSInteger) code
    contextInfo:(void*) contextInfo
{
    NSImage *image = [picker outputImage];
}
```

How the Contact Information Application Works

The rest of this chapter shows how to create a Contact Information application that uses the picture taker to add a photo to a contact entry. When it opens, the Contact Information application displays a window that lets the user enter a name, email address, phone number, and iChat address, as shown in Figure 5-6. The user can add an image by clicking the Add Photo button, which will open the picture taker.

Figure 5-6 An application that uses the picture taker



Using the Picture Taker in the Contact Information Application

This section provides step-by-step instructions for implementing the picture taker in the Contact Information application. First you'll set up the Xcode project, the project files, and the controller interface. Then you'll add the necessary routines to the implementation file. Finally, you'll connect the user interface to the picture taker action.

Setting Up the Project, Project Files, and the Controller Interface

Follow these steps to set up the project:

1. In Xcode, create a Cocoa application named Contact Information.
2. Add the Quartz framework to the project.

   For details, see "Using the Image Kit in Xcode" (page 11).
3. Double-click the `MainMenu.nib` file.
4. In the nib document window, double-click the Window icon and create a layout that looks similar to Figure 5-6 (page 65).

   It really doesn't matter what you create. The important elements for you to add from the Library are a push button object (labeled Add Photo) and an Image Well (`NSImageView`) object.
5. Save the nib file and return to Xcode.
6. In Xcode, choose File > New File.
7. Choose "Objective-C Class NSWindowController subclass" and click Next.
8. Name the file `MyController.m` and keep the option to create the header file. Then click Finish.
9. In the `MyController.h` file, import the Quartz framework by adding this statement:

   ```
   #import <Quartz/Quartz.h>
   ```

10. Add an instance variable for the image that appears in the contact information window.

    At this point, the interface should look as follows:

```
@interface MyController: NSWindowController
{
    IBOutlet NSImageView *  mImageView;
}
```

11. Save `MyController.h`.

Adding Routines to the Implementation File

Implement the picture taker routines by following these steps:

1. Open `MyController.m`

2. In the implementation file, first add an `awakeFromNib` method.

    This method needs to retrieve a shared instance of the picture taker and then set the default image to show when the picture taker first opens. You can set any image that you'd like, but the example below uses the ladybug image provided in the Desktop Pictures folder.

    Your `awakeFromNib` method should look similar to the following:

```
- (void) awakeFromNib
{
    IKPictureTaker *picker = [IKPictureTaker pictureTaker];
    [picker setInputImage:[[[NSImage alloc]
        initByReferencingFile:@"/Library/Desktop Pictures/Nature/Ladybug.jpg"]
         autorelease]];

}
```

3. Next, you need to add a method that launches the picture taker.

    The method first retrieves the shared instance of the picture taker. Then it sets an option for showing the effects button. You can set any options you like and that are defined in *IKPictureTaker Class Reference*. Finally, the method launches the picture taker as a sheet. You must provide the window that contains the image view, the delegate (which in this case is `self`), and a selector for a callback method. You'll write the callback next.

```
- (void) launchPictureTaker:(id)sender
{
    IKPictureTaker *picker = [IKPictureTaker pictureTaker];

    [picker setValue:[NSNumber numberWithBool:YES]
              forKey:IKPictureTakerShowEffectsKey];

    [picker beginPictureTakerSheetForWindow:[mImageView window]
               withDelegate:self
               didEndSelector:@selector(pictureTakerDidEnd:code:contextInfo:)
               contextInfo:nil];
}
```

If you prefer to launch the picture taker as a separate window, you can substitute the following for the last call in the `launchPictureTaker:` method.

```
[picker beginPictureTakerWithDelegate:self
          didEndSelector:@selector(pictureTakerDidEnd:code:contextInfo:)
          contextInfo:nil];
```

4. Write a `didEndSelector` method to provide as a callback.

   This method is invoked when the user dismisses the picture taker either by choosing an image or by clicking the Cancel button. If the user chooses an image, the method retrieves the output image and then sets it to the view so that the image that's displayed changes to the newly selected image. Otherwise, the method does nothing.

```
- (void) pictureTakerDidEnd:(IKPictureTaker*) pictureTaker code:(int) returnCode
 contextInfo:(void*) ctxInf
{
    if(returnCode == NSOKButton){
        NSImage *outputImage = [pictureTaker outputImage];
        [mImageView setImage:outputImage];
    }
    else{
        // The user canceled, so there is nothing to do.
    }
}
```

5. Save the `MyController.m` file.

6. Open the `MyController.h` file and add the launch method prototype just before the `@end` statement. When inserted, the interface for `MyController` should look like this:

```
@interface MyController : NSWindowController {
   IBOutlet NSImageView * mImageView;
}

- (IBAction) launchPictureTaker:(id)sender;
@end
```

7. Close the `MyController.h` file.

Connecting the User Interface to the Picture Taker Action

To make the connections necessary for the picture taker to launch when the user clicks the Add Photo button, follow these steps:

1. In Interface Builder, drag an Object (`NSObject`) from the Library to the nib document window.
2. In the Identity inspector, type MyController in the Class field.
3. If the Contact Information window that you created is not open, open it.
4. Control-drag from the Add Photo button to the instance of MyController.
5. In the connections panel, choose `launchPictureTaker:`.

6. Control-drag from MyController to the `NSImageView` instance in the window. Then, in the connections panel, choose `mImageView`.
7. Save the nib file.
8. In Xcode, click Build and Go. Then test your application to make sure the picture taker works.

   Check to make sure the effects button is displayed along with any other options you set up.

Browsing Filters and Setting Input Parameters

When Core Image was introduced in Mac OS X v10.4, it provided developers with access to one hundred built-in image processing filters and the ability to create custom filters. Although image filters are easy to use, the Core Image programming interface does not provide a user interface for choosing filters and for setting filter input parameters. You either used Core Image to set up and apply filters programmatically or created and managed your own user interface. With the introduction of the Image Kit framework in Mac OS X v10.5, all that has changed.

The `IKFilterBrowserPanel` and `IKFilterBrowserView` classes provide a filter browser that lets users:

- Browse filters by category
- Apply a filter and preview its effects
- Choose a filter
- Create filter collections
- See a description of the filter

As a developer, you can choose which filter categories the browser displays and whether or not to show a preview. After a user chooses a filter, you can use the `IKFilterUIView` class to obtain a view that contains controls for the input parameters of the filter.

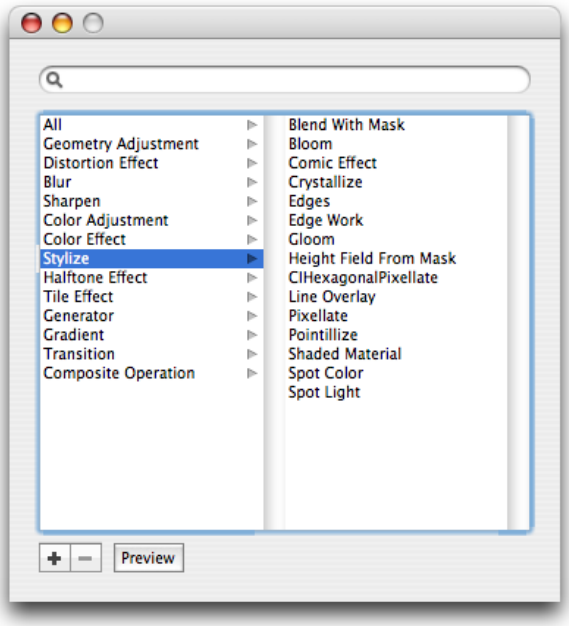
This chapter:

- Describes the filter browser and filter user interface view
- Discusses the user interface options you can set
- Shows how to set up and use the filter browser in an application
- Provides instructions for obtaining a filter user interface view and adding it into an existing view

The Filter Browser

The `IKFilterBrowserPanel` and `IKFilterBrowserView` classes provides a panel that displays a list, by category, of Core Image filters available on the system, as shown in Figure 6-1. The search field at the top allows users to quickly find filters.

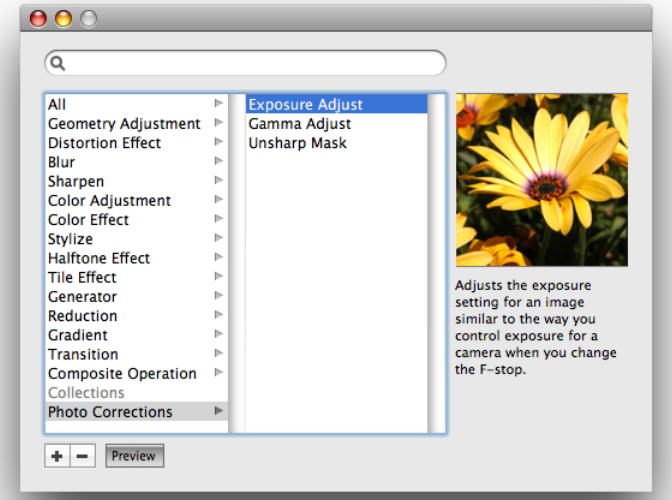
Figure 6-1 The filter browser



The user can click the Preview button to toggle a preview the preview off and on. The preview area lets the user see the effect of applying a filter using the default settings. Your application can set the preview image by registering for the notification `IKFilterBrowserWillPreviewFilterNotification`. A short description of the filter appears below the preview.

By clicking the plus (+) button, the user can add a custom group—referred to as a collection—as shown in Figure 6-2. The user can drag filters to the collection as a convenience for accessing sets of commonly used filters. Collections always appear below the built-in categories. To remove a collection, the user clicks the minus (–) button.

Figure 6-2 A Photo Corrections collection



You can control the visual appearance of the controls, including their size, in the filter browser by specifying the appropriate setting.

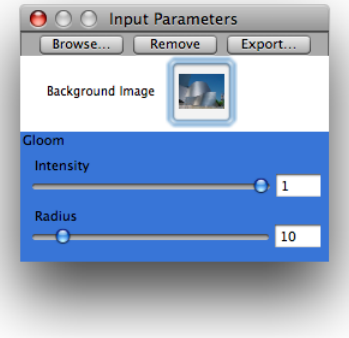
The Filter User Interface View

The `IKFilterUIView` class provides a view that contains controls for the input parameters of a filter. You can specify the size of the controls as miniature (`IKUISizeMini`), small (`IKUISizeSmall`), or regular (`IKUISizeRegular`).

A filter can optionally define basic, intermediate, advanced, and development sets of input parameters. These sets define which input parameters can be exposed in the user interface. For example, a filter that has many input parameters could define a basic set of input parameters for the typical consumer to control, and then programmatically set the other input parameters to default values. The same filter, however, can define an advanced set of input parameters that allow professional customers to control all the input parameters. Your application can request a specific set of input parameters. If the filter defines that set, then the filter user interface view reflects your request.

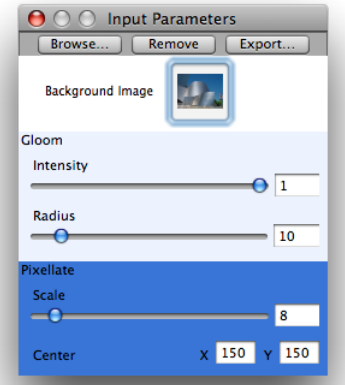
Figure 6-3 shows a window that displays a filter user interface view for the gloom filter provided by Core Image (`CIGloom`). The controls for this filter are labeled Intensity and Radius. In this example, the image that is processed by the filter is labeled Background image and appears in the image well shown in the figure.

Figure 6-3 Controls for the Gloom filter appear below the background image



The output of one Core Image filter can be directed to the input of another Core Image filter. Typically you'll want to write an application that allows users to apply more than one Core Image filter to an image. Figure 6-4 shows an application that inserts user interface views for two Core Image filters into a window—the gloom filter and the pixellate filter (`CIPixellate`). The controls for the pixellate filter are labeled Scale and Center.

Figure 6-4 Two filter views in a window



So far you've seen the user interface views for the built-in filters provided by Core Image. Developers who write image units can provide a custom view for each of the filters defined in the image unit by implementing the `IKFilterCustomUIProvider` protocol. For information on image units, see *Image Unit Tutorial*.

Opening the Filter Browser in an Application

Before you display the filter browser in an application, you need to load all the filters on the system by calling the `loadAllPlugIns` method of the `CIPlugIn` class. Then you create a shared instance of the `IKFilterBrowserPanel` class by calling the `filterBrowserPanelWithStyleMask:` method. You can

choose between the Aqua style (default) and the brushed metal look (`NSTexturedBackgroundWindowMask`). You show the filter browser by calling one of several methods, depending on whether you want the browser to appear as a sheet, a separate window, or in a custom view. In this section, you'll see how to create a filter browser that opens in a separate window.

This section shows hot to create an application that opens a filter browser in a separate window. First you'll set up the Xcode project, the project files, and the controller interface. Next you'll add the necessary routines to the implementation file. Then, you'll create the user interface in Interface Builder. Finally, you'll make a few refinements by adding an options dictionary and using a different background window.

Setting Up the Project, Project Files, and the Controller Interface

Follow these steps to set up the project:

1. In Xcode, create a Cocoa application and name it `My Filter Browser`.
2. Add the Quartz and Quartz Core frameworks to the project.

   For details, see "Using the Image Kit in Xcode" (page 11).
3. Choose File > New File.
4. Choose Objective-C Class and click Next.
5. Name the file `FilterBrowserController.m` and keep the option to create the header file. Then click Finish.
6. In `FilterBrowserController.h` import the Quartz Core and Quartz frameworks by adding these statements:

   ```
   #import <Quartz/Quartz.h>
   #import <QuartzCore/QuartzCore.h>
   ```
7. Create an instance variable for the filter browser. Your interface should look as follows:

   ```
   @interface FilterBrowserController : NSObject {
        IKFilterBrowserPanel *filterBrowserPanel;
   }
   @end
   ```
8. Save the `FilterBrowserController.h` file.

Adding Routines to the Implementation File

To implement the filter browser routines, follow these steps:

1. Open the `FilterBrowserController.m` file.
2. In the implementation file, add an `awakeFromNib` method.

   This method needs to load all Core Image plug-ins, as shown:

```
- (void)awakeFromNib
{
    [CIPlugIn loadAllPlugIns];
}
```

3. Next you'll write a `showFilterBrowserPanel:` method that is invoked when the user chooses the Open Filter Browser menu item. You'll create the menu item later.

   The method first defines a constant to set the mask so that the filter browser uses the brushed metal look. This is optional. The default is to use the Aqua look. Note that you must provide a selector for the method that is invoked when the user closes the browser. You'll write the selector method next.

```
- (IBAction)showFilterBrowserPanel:(id)sender
{
    int myStyleMask = NSTexturedBackgroundWindowMask;
    if(!filterBrowserPanel)
        filterBrowserPanel = [[IKFilterBrowserPanel
               filterBrowserPanelWithStyleMask:myStyleMask] retain];

    [filterBrowserPanel beginWithOptions:NULL modelessDelegate:self
didEndSelector:@selector(browserPanelDidEndSelector:returnCode:contextInfo:)
contextInfo:nil];

}
```

4. Add the `browserPanelDidEndSelector:returnCode:contextInfo:` selector method.

   This method simply takes the filter browser out of the screen list so that is it not visible.

```
- (void)browserPanelDidEndSelector:(NSWindow *)sheet returnCode:(int)returnCode
 contextInfo:(void *)contextInfo
{
    [sheet orderOut:self];
}
```

5. Add a `dealloc` method.

```
- (void)dealloc
{
    [super dealloc];
}
```

6. Save the `FilterBrowserController.m` file.

7. Open the `FilterBrowserController.h` file and add this method declaration, making sure that you insert it before the `@end` statement.

```
- (IBAction)showFilterBrowserPanel:(id)sender;
```

8. Save the `FilterBrowserController.h` file.

Creating the User Interface

Set up the user interface in Interface Builder by following these steps:

1. Double-click the `MainMenu.nib` file to open Interface Builder.
2. Delete the window icon in the nib document window.
3. Choose File > Synchronize With Xcode.
4. Drag an `NSObject` from the library to the nib document window.
5. In the Identity inspector for the `NSObject`, choose FilterBrowserController from the Class pop-up menu.
6. Double-click the MainMenu icon in the nib document window.
7. Drag a submenu item from the Library to the MainMenu and place it between the Format and View menu items. Name the menu Filter.
8. Name the item in the Filter menu Show Filter Browser.
9. Control-drag from the Show Filter Browser menu item to the FilterBrowserController and in the connections panel choose `showFilterBrowserPanel:`.
10. Save the nib file.
11. In Xcode, click Build and Go.

    When the application launches, choose Show Filter Browser from the Filter menu.

    Try the Search feature. Click the Preview button to toggle the preview off and back on. Then click the plus (+) button and add a collection. Drag a few filters into your collection.
12. Quit the application.

    Now that the application runs, you'll make a few refinements by adding an options dictionary and using a different window background.

Refining the User Interface

There are a number of options you can set to modify the filter browser user interface. Follow these steps to change the options:

1. In Xcode, open the `FilterBrowserController.m` file and add a method that creates a dictionary of options. Make sure that you place this method before the `showFilterBrowserPanel:` method .

   The Image Kit provides constants that let you specify the size of controls to use in the browser. The default is to use regular size controls. You'll change the size to miniature by using the key constant `IKUISizeFlavor` and the value `IKUISizeMini`.

```
-(NSDictionary *)createFilterBrowserUIOptions
{
 NSMutableDictionary *options = [[[NSMutableDictionary alloc] init] autorelease];
 [options setObject:IKUISizeMini forKey:IKUISizeFlavor];
 return options;
}
```

When you write a more complex application, you could let the user set the control size as a preference. Then your options dictionary method could set the options based on user preferences.

2. Modify the `showFilterBrowserPanel:` method so that it calls the `createFilterBrowserUIOptions` method.

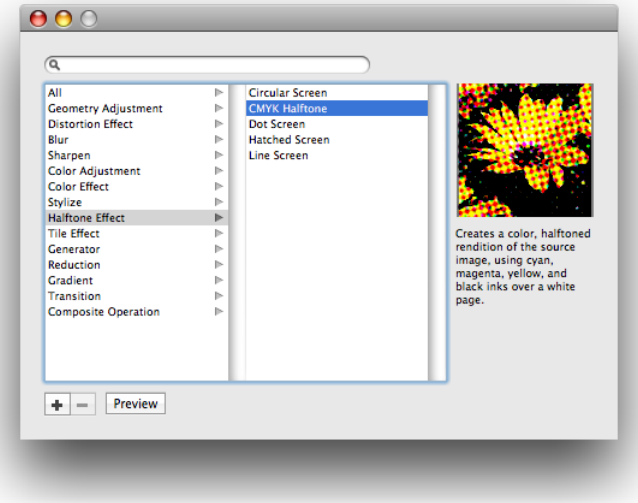
   In addition, set the style mask to `0` to get the default look.

   After modification, the method to show the filter browser should look as follows:

```
- (IBAction)showFilterBrowserPanel:(id)sender
{
  int myStyleMask = 0;
  NSDictionary * options = [self createFilterBrowserUIOptions];

  if(!filterBrowserPanel)
    filterBrowserPanel = [[IKFilterBrowserPanel
        filterBrowserPanelWithStyleMask:myStyleMask] retain];

  [filterBrowserPanel beginWithOptions:options
     modelessDelegate:self
    didEndSelector:@selector(browserPanelDidEndSelector:returnCode:contextInfo:)
     contextInfo:nil];

}
```

3. In Xcode, click Build and Go.

   After these modifications, the filter browser should look similar to the following:



Note that if you created a collection previously, the collection appears in the filter browser. Collections are persistent across launches of an application, on a per user basis.

Getting a Filter User Interface View

This section shows how to get a view for the input parameters of a filter and insert that filter user interface view into an existing view. You'll build upon the code in the last section by first adding code to respond to a user event—a double click—in the filter browser. When the user double-clicks the name of a filter in the filter browser, the application requests a filter user interface view for the selected filter and displays that view, as shown in Figure 6-3 (page 72) and Figure 6-4 (page 72).

You need to register for a user event notification—either `IKFilterBrowserFilterSelectedNotification` or `IKFilterBrowserFilterDoubleClickNotification` of the `IKFilterBrowserPanel` class. After the user chooses a filter, you call the `viewForUIConfiguration:excludedKeys:` method of the `CIFilter` class (an Image Kit addition) to obtain a view for the filter. You can provide a configuration dictionary to specify the size of controls to use for the filter input parameters, just as you did for the filter browser itself. You exclude keys for input parameters that you don't want to show (such as the input image which, in most cases, will already be displayed onscreen).

The application you'll build here doesn't perform any image processing. It simply shows how to set up the user interface. However, it does show how to stack several filter user interface views together. "Applying Filters to an Image" (page 81) discusses using the Image Kit user interface together with Core Image.

Creating a Filter View Controller

Follow these steps to set up the filter view controller:

1. In Xcode, open the project that you created in the last section.
2. Choose File > New File.
3. Choose Objective-C Class and click Next.
4. Name the file `FilterViewController.m` and keep the option to create the header file. Then click Finish.

   This is the controller for the filter user interface view. It obtains the filter, gets a view that contains the controls for the input parameters of the filter, and adds the view to the user interface.
5. In the `FilterViewController.h` file, add statements to import the Quartz and Quartz Core frameworks. You'll also need to add a directive for the FilterBrowserController class to allow the two controllers to communicate.

   ```
   #import <Quartz/Quartz.h>
   #import <QuartzCore/QuartzCore.h>

   @class FilterBrowserController;
   ```
6. Add two `IBOutlet` objects to the interface—one for a view that contains the filter UI view and another for the filter browser controller. You'll set these up in Interface Builder later.

   The interface should look as follows:

   ```
   @interface FilterViewController : NSObject {
       IBOutlet id    filterUIContainerBox;
       IBOutlet FilterBrowserController *browserController;
   ```

```
}
@end
```

7. Add a method signature for an `addFilter:` method.

   This is the method that adds the filter user interface view to the panel. You'll write the method later. When the user double-clicks a filter name, the method is invoked and passed a notification object that contains the name of the selected filter.

   ```
   - (void)addFilter:(NSNotification*)notification;
   ```

8. Save the `FilterViewController.h` file.

Implementing the Filter View Controller

To implement the routines necessary for the filter view controller, follow these steps:

1. In Xcode, open the `FilterViewController.m` file.

2. Add an `awakeFromNib` method. This method sets the background color of the filter view and makes sure that the panel is in front.

   ```
   - (void)awakeFromNib
   {
       [[filterUIContainerBox window] setBackgroundColor:[NSColor whiteColor]];
       [[filterUIContainerBox window] orderFront:self];
   }
   ```

3. Add a `dealloc` method.

   ```
   - (void)dealloc
   {
   [super dealloc];
   }
   ```

4. Write a method that creates and returns a dictionary of user interface options for the view. You'll set the size of the filter input parameter controls. (You can set additional options if you'd like.)

   ```
   -(NSDictionary *)createFilterBrowserUIOptions
   {
     NSMutableDictionary *options = [[[NSMutableDictionary alloc] init] autorelease];
    [options setObject:IKUISizeMini forKey:IKUISizeFlavor];

    return options;
   }
   ```

5. Write an `addFilter:` method.

   This method gets the view for the selected filter and inserts it into the panel you created previously in Xcode. The embedded comments explain what the code does.

   ```
   - (void)addFilter:(NSNotification*)notification
   {
       // Get the filter name from the notification.
       NSString    *filterName = [notification object];
   ```

```
    // Create a CIFilter object.
    CIFilter    *newFilter = [CIFilter filterWithName:filterName];
    // Add a filter user interface view only if a filter object exists.
    if(newFilter)
    {
        // Get  the frame rectangle that contains the filter UI container.
        NSRect  windowFrame = [[filterUIContainerBox window] frame];
        // Set the default values for the filter.
        [newFilter setDefaults];
        // Get an options dictionary, which specifies the size of the controls.
        NSDictionary    *options = [self createFilterBrowserUIOptions];
        // Create  a view for the filter, exclude the input image key.
        IKFilterUIView    *filterContentView =
             [newFilter viewForUIConfiguration:options
                         excludedKeys:[NSArray arrayWithObject:@"inputImage"]];
        // Retrieve  the bounding rectangle for the view.
        NSRect  contentBounds = [filterContentView bounds];
       // Make the view width match the width of the filter UI container.
        contentBounds.size.width = [filterUIContainerBox bounds].size.width;
       // Set up automatic resizing to accommodate additional views.
       [filterContentView setAutoresizingMask:NSViewMinYMargin];
        // Increase the window frame size to accommodate the filter UI view.
        windowFrame.size.height += [filterContentView bounds].size.height;
        windowFrame.origin.y -= [filterContentView bounds].size.height;
        // Set the frame of the filter UI container  to be the window frame.
        // Display the changes and animate for a nice effect.
        [[filterUIContainerBox  window] setFrame:windowFrame
                                         display:YES animate:YES];
        // Add the filter UI view to the filter container.
        [filterUIContainerBox  addSubview:filterContentView];

}
```

6. Save the `FilterViewController.m` file.

Modifying the Filter Browser Controller

Next you'll need to make a few modifications to the filter browser controller to allow the view and browser controllers to communicate. You'll also need to register for a notification whenever the user double-clicks a filter name. Follow these steps:

1. Open the `FilterBrowserController.h` file and add this `import` statement:

```
#import "FilterViewController.h"
```

2. Add an `IBOutlet` for the `FilterViewController`. The modified interface should now look like this:

```
@interface FilterBrowserController : NSObject {
    IKFilterBrowserPanel    *filterBrowserPanel;
    IBOutlet FilterViewController *viewController;

}

- (IBAction)showFilterBrowserPanel:(id)sender;
```

```
@end
```

3. Open the `FilterBrowserController.m` file and modify the `awakeFromNib` method to register for the double-click notification setting the observer as `FilterViewController` and the method as `addFilter:`.

   The `awakeFromNib` method should now looks as follows:

```
- (void)awakeFromNib
{
    [CIPlugIn loadAllPlugIns];
    [[NSNotificationCenter defaultCenter] addObserver:viewController
                          selector:@selector(addFilter:)
                          name:IKFilterBrowserFilterDoubleClickNotification
                          object:nil];

}
```

4. Save and close the `FilterBrowserController.h` and `FilterBrowserController.m` files.

Creating the User Interface

Set up the user interface in Interface Builder by following these steps:

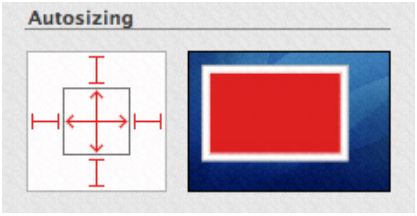
1. Double-click the `MainMenu.nib` file to open Interface Builder.
2. Drag a panel from the Library to the nib document window.
3. Change the name of the Panel icon to FilterInputs.
4. In the Attributes inspector, change the title of the panel to Filter Input Parameters.
5. Choose File > Synchronize With Xcode.
6. Drag an Object (`NSObject`) from the Library to the nib document window.
7. In the Identity inspector, select FilterViewController from the Class pop-up menu.
8. In the nib document window, control-drag from the FilterBrowserController icon to the FilterViewController icon. Then in the connections panel, choose the `viewController` outlet.
9. Control-drag from the FilterViewController icon to the FilterBrowserController icon. Then connect to the `browserController` outlet.
10. Control-drag from the FilterInputs icon to the FilterViewController icon. Then in the connections panel choose `delegate`.
11. Drag a Custom View to the panel. Make sure to hold it over the panel until the panel opens, then drop it in the opened panel.
12. Control-drag from the FilterViewController icon to the custom view. Then in the connections panel choose `filterUIContainerBox`.

13. Click the custom view in the panel.

14. Set the size of the view to 320 by 2 and make sure that the view (which will be a thin line) is positioned at the top of the window.

    This is the view to which you'll add the filter user interface view. Unless you want to start by showing something in this view, its height can be negligible.

15. Set the springs and struts for the Custom View like this:



16. Set the springs and struts for the *content view* of the Custom View so that only the struts—which are the outer Autosizing elements —are set.

    Note: It's easiest to see the content view by setting the view style to column view and then clicking the Filter Inputs icon in the nib document window.

17. Set the size of the panel to 320 by 8.

    Unless you want to show something in the panel other than filter input parameters, the panel height can be as small as possible.

18. Save the `MainMenu.nib` file.

19. In Xcode, click Build and Go.

    Choose Show Filter Browser from the Filter menu. Double-click one or more filters to test your code. Each time you double-click a filter name, the Filter Input Parameters panel should show the input parameters for that filter.

Applying Filters to an Image

You've seen in the previous sections how to write code that presents a simple user interface for the filter browser and filter input parameters. A full-featured image processing application requires a more sophisticated user interface. For example, it would:

- Show the filter input parameter panel only after an image is open and a filter is selected.
- Add code to delineate visually the input parameters of each filter, by adding a bounding box, a filter name, varying the background color for each filter UI view, or by some other means.
- Allow the user to delete a filter from the input parameters panel and possible rearrange the order.

A fully functioning image processing application also requires code that opens and saves images and applies selected filters to an image. The IUUIDemoApplication example provided with the developer tools for Mac OS X v10.5 provides most of the functionality needed in a full-featured image processing application. Now that you know how the user interface portion of the filter browser and filters works, you may want to take a look at IUUIDemoApplication to see how to combine the user interface code with code that applies Core Image filters.

You can find the application in:

```
/Developer/Examples/Quartz/Core Image/
```

IUUIDemoApplication has a controller for the filter browser and another controller for the filter user interface view. It also creates an object that tracks filters and another that tracks filter user interface views. It uses the Core Image programming interface to process images.

If you are not familiar with Core Image, see:

- *Core Image Programming Guide*
- *Core Image Reference Collection*

Glossary

Core Image An image processing programming interface, introduced in Mac OS X v10.4, that is part of the Quartz Core framework.

filter Code that uses Core Image to process digital images.

filter browser The user term for the Image Kit filter browser panel class (`IKFilterBrowserPanel`) which allows users to browse Core Image filters.

image browser view The Image Kit view class (`IKImageBrowserView`) that is optimized for browsing images. The user term is image browser.

Image Edit panel The Image Kit edit panel class (`IKImageEditPanel`) that is optimized for editing images. The user term is Image Edit window.

Image I/O A framework (`ImageIO.framework`) that provides opaque data types for reading data from an image source and writing data to an image destination.

Image Kit A programming interface that supports browsing, viewing, and editing images and browsing and controlling Core Image filters.

image unit A Core Image filter that is packaged for distribution as an `NSBundle` object

picture taker panel The Image Kit picture taker class (`IKPictureTaker`) that allows users to choose images by browsing the file system or by taking a snapshot with an iSight or other digital camera. The user term is picture taker.

Quick Look framework A framework, introduced in Mac OS X v10.5, that provides a plug-in architecture for custom document types. A Quick Look document type can be displayed as a preview in the Finder and as an item in such applications as iPhoto or any application that supports slideshows created with the `IKSlideshow` class.

Document Revision History

This table describes the changes to *Image Kit Programming Guide*.

Date	Notes
2008-06-09	Added information about support for dragging image views.
2007-12-04	Revised a figure and changed an instance variable name.
2007-07-20	New document that explains how to support browsing, viewing, and editing images and browsing and controlling Core Image filters.

Document Revision History