CHAPTER 2
Drawing

The views illustrated in Chapter 1 were mostly colored rectangles; they had a `backgroundColor` and no more. But that, of course, is not what a real iOS program looks like. Everything the user sees is a UIView, and what the user sees is a lot more than a bunch of colored rectangles. That's because the views that the user sees have *content*. They contain *drawing*.

Many UIView subclasses, such as a UIButton or a UILabel, know how to draw themselves. Sooner or later, you're also going to want to do some drawing of your own. You can prepare your drawing as an image file beforehand. You can draw an image as your app runs, in code. You can display an image in your interface in a UIView subclass that knows how to show an image, such as a UIImageView or a UIButton. A pure UIView is all about drawing, and it leaves that drawing largely up to you; your code determines what the view draws, and hence what it looks like in your interface.

This chapter discusses the mechanics of drawing. Don't be afraid to write drawing code of your own! It isn't difficult, and it's often the best way to make your app look the way you want it to. (For how to draw text, see Chapter 10.)

Images and Image Views

The basic general UIKit image class is UIImage. UIImage can read a file from disk, so if an image does not need to be created dynamically, but has already been created before your app runs, then drawing may be as simple as providing an image file as a resource in your app's bundle. The system knows how to work with many standard image file types, such as TIFF, JPEG, GIF, and PNG; when an image file is to be included in your app bundle, iOS has a special affinity for PNG files, and you should prefer them whenever possible. You can also obtain image data in some other way, such as by downloading it, and transform this into a UIImage. Conversely, your code can construct a UIImage

for display in your interface or for saving to disk (image file output is discussed in Chapter 23).

Image Files

A pre-existing image file in your app's bundle can be obtained through the UIImage initializer `init(named:)`. This method looks in two places for the image:

Asset catalog
: We look in the asset catalog for an image set with the supplied name. The name is case-sensitive.

Top level of app bundle
: We look at the top level of the app's bundle for an image file with the supplied name. The name is case-sensitive and should include the file extension; if it doesn't include a file extension, *.png* is assumed.

When calling `init(named:)`, an asset catalog is searched before the top level of the app's bundle. If there are multiple asset catalogs, they are all searched, but the search order is indeterminate and cannot be specified, so avoid image sets with the same name.

A nice thing about `init(named:)` is that the image data may be cached in memory, and if you ask for the same image by calling `init(named:)` again later, the cached data may be supplied immediately. Alternatively, you can read an image file from anywhere in your app's bundle directly and without caching, using `init(contentsOfFile:)`, which expects a pathname string; you can get a reference to your app's bundle with `NSBundle.mainBundle()`, and NSBundle then provides instance methods for getting the pathname of a file within the bundle, such as `pathForResource:ofType:`.

Methods that specify a resource in the app bundle, such as `init(named:)` and `pathForResource:ofType:`, respond to suffixes in the name of an actual resource file. On a device with a double-resolution screen, when an image is obtained by name from the app bundle, a file with the same name extended by `@2x`, if there is one, will be used automatically, with the resulting UIImage marked as double-resolution by assigning it a `scale` property value of `2.0`. Similarly, if there is a file with the same name extended by `@3x`, it will be used on the triple-resolution screen of the iPhone 6 Plus, with a `scale` property value of `3.0`.

In this way, your app can contain multiple versions of an image file at different resolutions. Thanks to the `scale` property, a high-resolution version of an image is drawn at the same size as the single-resolution image. Thus, on a high-resolution screen, your code continues to work without change, but your images look sharper.

Similarly, a file with the same name extended by `~ipad` will automatically be used if the app is running natively on an iPad. You can use this in a universal app to supply different images automatically depending on whether the app runs on an iPhone or iPod touch,

on the one hand, or on an iPad, on the other. (This is true not just for images but for *any* resource obtained by name from the bundle. See Apple's *Resource Programming Guide*.)

One of the great benefits of an asset catalog, though, is that you can forget all about those name suffix conventions. An asset catalog knows when to use an alternate image within an image set, not from its name, but from its place in the catalog. Put the single-, double-, and triple-resolution alternatives into the slots marked "1x," "2x," and "3x" respectively. For a distinct iPad version of an image, check iPhone and iPad in the Attributes inspector for the image set; separate slots for those device types will appear in the asset catalog.

An asset catalog can also distinguish between versions of an image intended for different size class situations. (See the discussion of size classes and trait collections in Chapter 1.) In the Attributes inspector for your image set, use the Width and Height pop-up menus to specify which size class possibilities you want to distinguish. Thus, for example, if we're on an iPhone with the app rotated to landscape orientation, and if there's both an Any height and a Compact height alternative in the image set, the Compact height version is used. These features are live as the app runs; if the app rotates from landscape to portrait, and there's both an Any height and a Compact height alternative in the image set, the Compact height version is replaced with the Any height version in your interface, automatically.

The way an asset catalog performs all this magic is through trait collections and the UIImageAsset class. When an image is extracted from an asset catalog through `init(named:)` and the name of its image set, its `imageAsset` property is a UIImageAsset. All the images in that image set are available through the UIImageAsset; each image has a trait collection associated with it (its `traitCollection`), and you can ask an image's `imageAsset` for the image from the same image set appropriate to a particular trait collection, by calling `imageWithTraitCollection:`.

A built-in interface object that displays an image is automatically trait collection–aware; it receives the `traitCollectionDidChange:` message and responds accordingly. We can imagine how this works under the hood by building a UIView with an `image` property that does the same thing:

```
class MyView: UIView {
    var image : UIImage!
    override func traitCollectionDidChange(previous: UITraitCollection?) {
        self.setNeedsDisplay() // causes drawRect to be called
    }
    override func drawRect(rect: CGRect) {
        if var im = self.image {
            if let asset = self.image.imageAsset {
                let tc = self.traitCollection
                im = asset.imageWithTraitCollection(tc)
            }
```

```
                im.drawAtPoint(CGPointZero)
            }
        }
    }
```

Moreover, your code can combine images into a UIImageAsset — the code equivalent of an image set in an asset catalog, but *without* an asset catalog. Thus you could create images in real time (as I'll describe later in this chapter), or fetch images out of your app bundle, and configure them so that one is used when an iPhone app is in portrait orientation and the other is used when the app is in landscape orientation, *automatically*. For example:

```
let tcdisp = UITraitCollection(displayScale: UIScreen.mainScreen().scale)
let tcphone = UITraitCollection(userInterfaceIdiom: .Phone)
let tcreg = UITraitCollection(verticalSizeClass: .Regular)
let tc1 = UITraitCollection(traitsFromCollections: [tcdisp, tcphone, tcreg])
let tccom = UITraitCollection(verticalSizeClass: .Compact)
let tc2 = UITraitCollection(traitsFromCollections: [tcdisp, tcphone, tccom])
let moods = UIImageAsset()
let frowney = UIImage(named:"frowney")!
let smiley = UIImage(named:"smiley")!
moods.registerImage(frowney, withTraitCollection: tc1)
moods.registerImage(smiley, withTraitCollection: tc2)
```

After that, if `frowney` is placed into the interface — for example, by handing it over to a UIImageView as its image, as I'll explain in a moment — it automatically alternates with `smiley` when the app changes orientation. The remarkable thing is that this works even though there is no persistent reference to `frowney`, `smiley`, or the UIImageAsset (`moods`). The reason is that `frowney` and `smiley` are cached by the system (because of the call to `init(named:)`), and they each maintain a strong reference to the UIImageAsset with which they are registered.

You can also specify a target trait collection while fetching an image from the asset catalog or from your app bundle, by calling `init(named:inBundle:compatibleWithTraitCollection:)`. The bundle specified will usually be `nil`, meaning the app's main bundle.

Image Views

Many built-in Cocoa interface objects will accept a UIImage as part of how they draw themselves; for example, a UIButton can display an image, and a UINavigationBar or a UITabBar can have a background image. I'll discuss those in Chapter 12. But when you simply want an image to appear in your interface, you'll probably hand it to an image view — a UIImageView — which has the most knowledge and flexibility with regard to displaying images and is intended for this purpose.

The nib editor supplies some shortcuts in this regard: the Attributes inspector of an interface object that can have an image will have a pop-up menu listing known images

in your project, and such images are also listed in the Media library (Command-Option-Control-4). Media library images can often be dragged onto an interface object in the canvas to assign them, and if you drag a Media library image into a plain view, it is transformed into a UIImageView displaying that image.

A UIImageView can actually have *two* images, one assigned to its `image` property and the other assigned to its `highlightedImage` property; the value of the UIImageView's `highlighted` property dictates which of the two is displayed at any given moment. A UIImageView does not automatically highlight itself merely because the user taps it, the way a button does. However, there are certain situations where a UIImageView will respond to the highlighting of its surroundings; for example, within a table view cell, a UIImageView will show its highlighted image when the cell is highlighted (Chapter 8).

A UIImageView is a UIView, so it can have a background color in addition to its image, it can have an alpha (transparency) value, and so forth (see Chapter 1). An image may have areas that are transparent, and a UIImageView will respect this; thus an image of any shape can appear. A UIImageView without a background color is invisible except for its image, so the image simply appears in the interface, without the user being aware that it resides in a rectangular host. A UIImageView without an image and without a background color is invisible, so you could start with an empty UIImageView in the place where you will later need an image and subsequently assign the image in code. You can assign a new image to substitute one image for another, or set the image view's `image` property to `nil` to remove it.

How a UIImageView draws its image depends upon the setting of its `contentMode` property (UIViewContentMode). (The `contentMode` property is inherited from UIView; I'll discuss its more general purpose later in this chapter.) For example, `.ScaleToFill` means the image's width and height are set to the width and height of the view, thus filling the view completely even if this alters the image's aspect ratio; `.Center` means the image is drawn centered in the view without altering its size. The best way to get a feel for the meanings of the various `contentMode` settings is to assign a UIImageView a small image in a nib and then, in the Attributes inspector, change the Mode pop-up menu, and see where and how the image draws itself.

You should also pay attention to a UIImageView's `clipsToBounds` property; if it is `false`, its image, even if it is larger than the image view and even if it is not scaled down by the `contentMode`, may be displayed in its entirety, extending beyond the image view itself.

When creating a UIImageView in code, you can take advantage of a convenience initializer, `init(image:)` (or `init(image:highlightedImage:)`). The default `contentMode` is `.ScaleToFill`, but the image is not initially scaled; rather, *the view itself is sized to match the image*. You will still probably need to position the UIImageView correctly

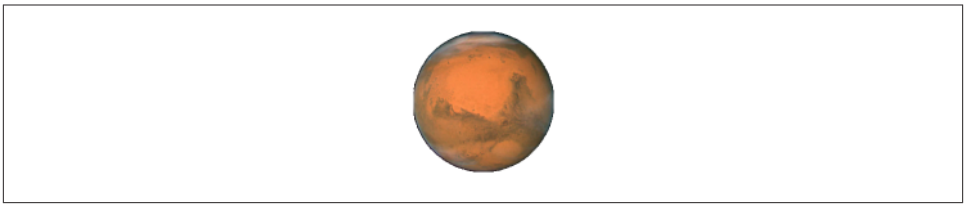

Figure 2-1. Mars appears in my interface

in its superview. In this example, I'll put a picture of the planet Mars in the center of the app's interface (Figure 2-1; for the CGRect `center` property, see Appendix B):

```
let iv = UIImageView(image:UIImage(named:"Mars")) // asset catalog
mainview.addSubview(iv)
iv.center = iv.superview!.bounds.center
iv.frame.makeIntegralInPlace()
```

What happens to the size of an existing UIImageView when you assign an image to it depends on whether the image view is using autolayout. If it isn't, or if its size is constrained absolutely, the image view's size doesn't change. But under autolayout, the size of the new image becomes the image view's new `intrinsicContentSize`, so the image view will adopt the image's size unless other constraints prevent.

An image view automatically acquires its `alignmentRectInsets` from its image's `alignmentRectInsets`. Thus, if you're going to be aligning the image view to some other object using autolayout, you can attach appropriate `alignmentRectInsets` to the image that the image view will display, and the image view will do the right thing. To do so, derive a new image by calling the original image's `imageWithAlignmentRectInsets:`.

In theory, you should be able to set an image's `alignmentRectInsets` in an asset catalog. As of this writing, however, this feature is not working correctly.

Resizable Images

Certain places in the interface require a resizable image; for example, a custom image that serves as the track of a slider or progress view (Chapter 12) must be resizable, so that it can fill a space of any length. And there can frequently be other situations where you want to fill a background by tiling or stretching an existing image.

To make a resizable image, start with a normal image and call its `resizableImageWithCapInsets:resizingMode:` method. The `capInsets:` argument is a UIEdgeInsets, whose components represent distances inward from the edges of the image. In a context larger than the image, a resizable image can behave in one of two ways, depending on the `resizingMode:` value (UIImageResizingMode):

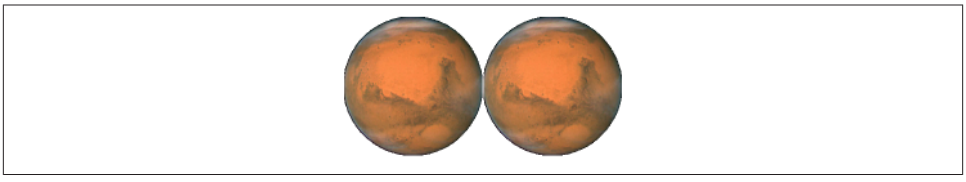

Figure 2-2. Tiling the entire image of Mars

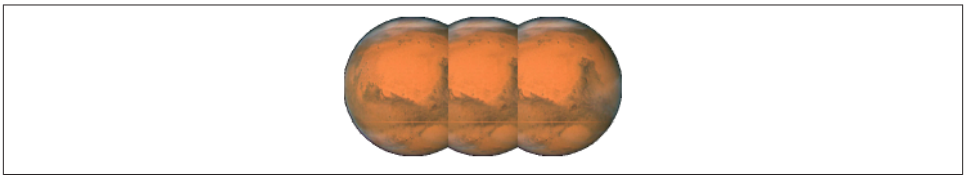

Figure 2-3. Tiling the interior of Mars

`.Tile`
: The interior rectangle of the inset area is tiled (repeated) in the interior; each edge is formed by tiling the corresponding edge rectangle outside the inset area. The four corner rectangles outside the inset area are drawn unchanged.

`.Stretch`
: The interior rectangle of the inset area is stretched *once* to fill the interior; each edge is formed by stretching the corresponding edge rectangle outside the inset area *once*. The four corner rectangles outside the inset area are drawn unchanged.

In these examples, assume that `self.iv` is a UIImageView with absolute height and width (so that it won't adopt the size of its image) and with a `contentMode` of `.ScaleToFill` (so that the image will exhibit resizing behavior). First, I'll illustrate tiling an entire image (Figure 2-2); note that the `capInsets:` is `UIEdgeInsetsZero`:

```
let mars = UIImage(named:"Mars")!
let marsTiled = mars.resizableImageWithCapInsets(
    UIEdgeInsetsZero, resizingMode: .Tile)
self.iv.image = marsTiled
```

Now we'll tile the interior of the image, changing the `capInsets:` argument from the previous code (Figure 2-3):

```
let marsTiled = mars.resizableImageWithCapInsets(
    UIEdgeInsetsMake(
        mars.size.height / 4.0,
        mars.size.width / 4.0,
        mars.size.height / 4.0,
        mars.size.width / 4.0
    ), resizingMode: .Tile)
```

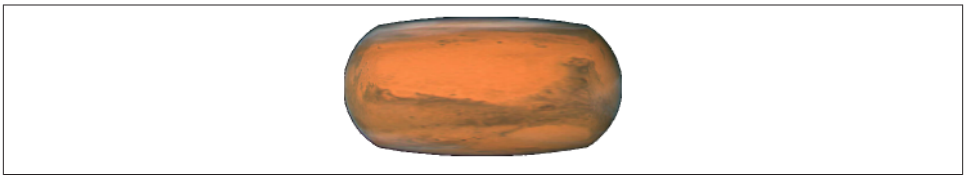

Figure 2-4. Stretching the interior of Mars

Figure 2-5. Stretching a few pixels at the interior of Mars

Next, I'll illustrate stretching. We'll start by changing just the `resizingMode:` from the previous code (Figure 2-4):

```
let marsTiled = mars.resizableImageWithCapInsets(
    UIEdgeInsetsMake(
        mars.size.height / 4.0,
        mars.size.width / 4.0,
        mars.size.height / 4.0,
        mars.size.width / 4.0
    ), resizingMode: .Stretch)
```

A common stretching strategy is to make almost half the original image serve as a cap inset, leaving just a pixel or two in the center to fill the entire interior of the resulting image (Figure 2-5):

```
let marsTiled = mars.resizableImageWithCapInsets(
    UIEdgeInsetsMake(
        mars.size.height / 2.0 - 1,
        mars.size.width / 2.0 - 1,
        mars.size.height / 2.0 - 1,
        mars.size.width / 2.0 - 1
    ), resizingMode: .Stretch)
```

You should also experiment with different scaling `contentMode` settings. In the preceding example, if the image view's `contentMode` is `.ScaleAspectFill`, and if the image view's `clipsToBounds` is `true`, we get a sort of gradient effect, because the top and bottom of the stretched image are outside the image view and aren't drawn (Figure 2-6).

Alternatively, you can configure a resizable image without code, in the project's asset catalog. It is often the case that a particular image will be used in your app chiefly as a resizable image, and always with the same `capInsets:` and `resizingMode:`, so it makes

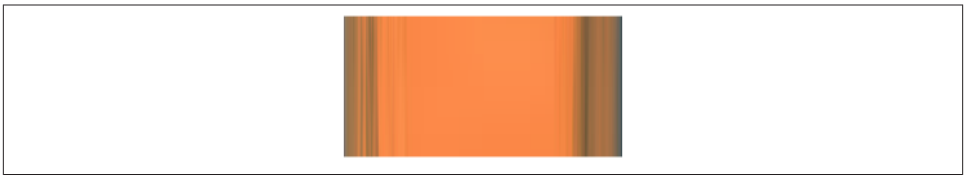

Figure 2-6. Mars, stretched and clipped

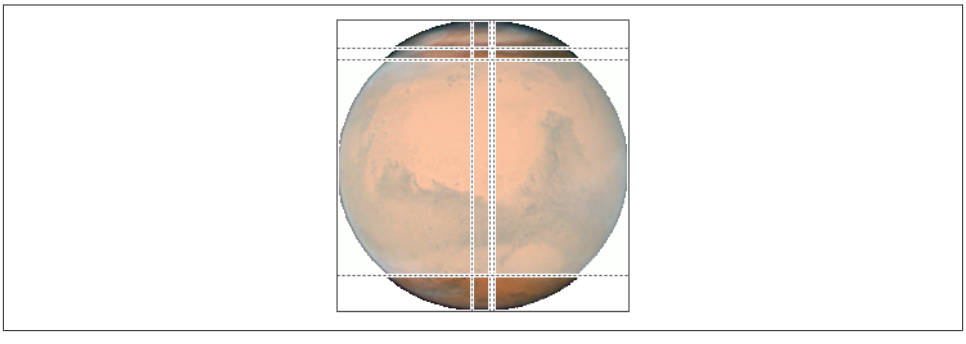

Figure 2-7. Mars, sliced in the asset catalog

sense to configure this image once rather than having to repeat the same code. And even if an image is configured in the asset catalog to be resizable, it can appear in your interface as a normal image as well — for example, if you assign it to an image view that resizes itself to fit its image, or that doesn't scale its image.

To configure an image in an asset catalog as a resizable image, select the image and, in the Slicing section of the Attributes inspector, change the Slices pop-up menu to Horizontal, Vertical, or Horizontal and Vertical. When you do this, additional interface appears. You can specify the `resizingMode` with another pop-up menu. You can work numerically, or click Show Slicing at the lower right of the canvas and work graphically. The graphical editor is zoomable, so zoom in to work comfortably.

This feature is actually even more powerful than `resizableImageWithCapInsets:resizingMode:`. It lets you specify the end caps *separately* from the tiled or stretched region, with the rest of the image being sliced out. In Figure 2-7, for example, the dark areas at the top left, top right, bottom left, and bottom right will be drawn as is. The narrow bands will be stretched, and the small rectangle at the top center will be stretched to fill most of the interior. But the rest of the image, the large central area covered by a sort of gauze curtain, will be omitted entirely. The result is shown in Figure 2-8.

Figure 2-8. Mars, sliced and stretched

Image Rendering Mode

Several places in an iOS app's interface automatically treat an image as a *transparency mask*, also known as a *template*. This means that the image color values are ignored, and only the transparency (alpha) values of each pixel matter. The image shown on the screen is formed by combining the image's transparency values with a single tint color. Such, for example, is the behavior of a tab bar item's image.

The way an image will be treated is a property of the image, its `renderingMode`. This property is read-only; to change it, start with an image and generate a new image with a different rendering mode, by calling `imageWithRenderingMode:`. The rendering mode values (UIImageRenderingMode) are:

- `.Automatic`
- `.AlwaysOriginal`
- `.AlwaysTemplate`

The default is `.Automatic`, which means that the image is drawn normally everywhere except in certain limited contexts, where it is used as a transparency mask.

With the `renderingMode` property, you can *force* an image to be drawn normally, even in a context that would usually treat it as a transparency mask. You can also do the opposite: you can *force* an image to be treated as a transparency mask, even in a context that would otherwise treat it normally.

To accompany this feature, iOS gives every UIView a `tintColor`, which will be used to tint any template images it contains. Moreover, this `tintColor` by default is inherited down the view hierarchy, and indeed throughout the entire app, starting with the window (Chapter 1). Thus, assigning your app's main window a tint color is probably one of the few changes you'll make to the window; otherwise, your app adopts the system's blue tint color. (Alternatively, if you're using a main storyboard, set the Global Tint color in its File inspector.) Individual views can be assigned their own tint color, which is inherited by their subviews. Figure 2-9 shows two buttons displaying the same background image, one in normal rendering mode, the other in template rendering mode,

Figure 2-9. One image in two rendering modes

in an app whose window tint color is red. (I'll say more about template images and `tintColor` in Chapter 12.)

An asset catalog can assign an image a rendering mode. Select the image set in the asset catalog, and use the Render As pop-up menu in the Attributes inspector to set the rendering mode to Default (`.Automatic`), Original Image (`.AlwaysOriginal`), or Template Image (`.AlwaysTemplate`). This is an excellent approach whenever you have an image that you will use primarily in a specific rendering mode, because it saves you from having to remember to set that rendering mode in code every time you fetch the image. Instead, any time you call `init(named:)`, this image arrives with the rendering mode already set.

Reversible Images

New in iOS 9, the entire interface is automatically reversed when your app runs on a system for which your app is localized if the system language is right-to-left. In general, this probably won't affect your images. The runtime assumes that you *don't* want images to be reversed when the interface is reversed, so its default behavior is to leave them alone.

Nevertheless, you *might* want an image reversed when the interface is reversed. For example, suppose you've drawn an arrow pointing in the direction from which new interface will arrive when the user taps a button. If the button pushes a view controller onto a navigation interface, that direction is from the right on a left-to-right system, but from the left on a right-to-left system. This image has directional meaning within the app's own interface; it needs to flip horizontally when the interface is reversed.

To make this possible, call the image's `imageFlippedForRightToLeftLayoutDirection` and use the resulting image in your interface. On a left-to-right system, the normal image will be used; on a right-to-left system, a reversed version of the image will be created and used automatically. You can override this behavior, even if the image is reversible, for a particular UIView displaying the image, such as a UIImageView, by setting that view's `semanticContentAttribute` to prevent mirroring.

Unfortunately, there's no way to designate an image as reversible in an asset catalog. Thus, if your image appears in the interface automatically — because of the way a view, such as an image view, is configured in the nib editor — you'll have to intervene in code.

In this example, my view controller's `viewDidLoad` pulls the image out of an image view (`self.iv`) and replaces its with a reversible version of itself:

```
override func viewDidLoad() {
    super.viewDidLoad()
    self.iv.image =
        self.iv.image?.imageFlippedForRightToLeftLayoutDirection()
}
```

Graphics Contexts

Instead of plopping an existing image file directly into your interface, you may want to create some drawing yourself, in code. To do so, you will need a *graphics context*.

A graphics context is basically a place you can draw. Conversely, you can't draw in code unless you've got a graphics context. There are several ways in which you might obtain a graphics context; in this chapter I will concentrate on two, which have proven in my experience to be far and away the most common:

You create an image context
: The function `UIGraphicsBeginImageContextWithOptions` creates a graphics context suitable for use as an image. You then draw into this context to generate the image. When you've done that, you call `UIGraphicsGetImageFromCurrentImageContext` to turn the context into a UIImage, and then `UIGraphicsEndImageContext` to dismiss the context. Now you have a UIImage that you can display in your interface or draw into some other graphics context or save as a file.

Cocoa hands you a graphics context
: You subclass UIView and implement `drawRect:`. At the time your `drawRect:` implementation is called, Cocoa has already created a graphics context and is asking you to draw into it, right now; whatever you draw is what the UIView will display.

  A slight variant of this situation is that you subclass CALayer and implement `drawInContext:`, or else implement `drawLayer:inContext:` in the layer's delegate; layers are discussed in Chapter 3.

Moreover, at any given moment there either is or is not a *current graphics context*:

- `UIGraphicsBeginImageContextWithOptions` not only creates an image context, it also makes that context the current graphics context.
- When `drawRect:` is called, the UIView's drawing context is already the current graphics context.
- Callbacks with a `context:` parameter have *not* made any context the current graphics context; rather, that parameter is a reference to a graphics context into which

you are invited to draw, but if you need to make it current so that you *can* draw into it, doing so is up to you.

What beginners find most confusing about drawing is that there are two sets of tools for drawing, which take different attitudes toward the context in which they will draw. One set needs a current context; the other just needs a context:

UIKit
: Various Cocoa classes know how to draw themselves; these include UIImage, NSString (for drawing text), UIBezierPath (for drawing shapes), and UIColor. Some of these classes provide convenience methods with limited abilities; others are extremely powerful. In many cases, UIKit will be all you'll need.

  With UIKit, you can draw *only into the current context*. So if you're in a `UIGraphicsBeginImageContextWithOptions` or `drawRect:` situation, you can use the UIKit convenience methods directly; there is a current context and it's the one you want to draw into. If you've been handed a `context:` parameter, on the other hand, then if you want to use the UIKit convenience methods, you'll have to make that context the current context; you do this by calling `UIGraphicsPushContext` (and be sure to restore things with `UIGraphicsPopContext` later).

Core Graphics
: This is the full drawing API. Core Graphics, often referred to as Quartz, or Quartz 2D, is the drawing system that underlies all iOS drawing — UIKit drawing is built on top of it — so it is low-level and consists of C functions. There are a lot of them! This chapter will familiarize you with the fundamentals; for complete information, you'll want to study Apple's *Quartz 2D Programming Guide*.

  With Core Graphics, you must *specify a graphics context* (a CGContext) to draw into, explicitly, in every function call. If you've been handed a `context:` parameter, then that's probably the graphics context you want to draw into. But in a `UIGraphicsBeginImageContextWithOptions` or `drawRect:` situation, you have no reference to a context; to use Core Graphics, you need to get such a reference. Since the context you want to draw into is the current graphics context, you call `UIGraphicsGetCurrentContext` to get the needed reference.

You don't have to use UIKit or Core Graphics *exclusively*. On the contrary, you can intermingle UIKit calls and Core Graphics calls in the same chunk of code to operate on the same graphics context. They merely represent two different ways of telling a graphics context what to do.

So we have two sets of tools and three ways in which a context might be supplied; that makes six ways of drawing. I'll now demonstrate all six of them! Without worrying just

yet about the actual drawing commands, focus your attention on how the context is specified and on whether we're using UIKit or Core Graphics. First, I'll draw a blue circle by implementing a UIView subclass's `drawRect:`, using UIKit to draw into the current context, which Cocoa has already prepared for me:

```
override func drawRect(rect: CGRect) {
    let p = UIBezierPath(ovalInRect: CGRectMake(0,0,100,100))
    UIColor.blueColor().setFill()
    p.fill()
}
```

Now I'll do the same thing with Core Graphics; this will require that I first get a reference to the current context:

```
override func drawRect(rect: CGRect) {
    let con = UIGraphicsGetCurrentContext()!
    CGContextAddEllipseInRect(con, CGRectMake(0,0,100,100))
    CGContextSetFillColorWithColor(con, UIColor.blueColor().CGColor)
    CGContextFillPath(con)
}
```

Next, I'll implement a UIView subclass's `drawLayer:inContext:`. In this case, we're handed a reference to a context, but it isn't the current context. So I have to make it the current context in order to use UIKit:

```
override func drawLayer(layer: CALayer, inContext con: CGContext) {
    UIGraphicsPushContext(con)
    let p = UIBezierPath(ovalInRect: CGRectMake(0,0,100,100))
    UIColor.blueColor().setFill()
    p.fill()
    UIGraphicsPopContext()
}
```

To use Core Graphics in `drawLayer:inContext:`, I simply keep referring to the context I was handed:

```
override func drawLayer(layer: CALayer, inContext con: CGContext) {
    CGContextAddEllipseInRect(con, CGRectMake(0,0,100,100))
    CGContextSetFillColorWithColor(con, UIColor.blueColor().CGColor)
    CGContextFillPath(con)
}
```

Finally, I'll make a UIImage of a blue circle. We can do this at any time (we don't need to wait for some particular method to be called) and in any class (we don't need to be in a UIView subclass). The resulting UIImage (here called `im`) is suitable anywhere you would use a UIImage. For instance, you could hand it over to a visible UIImageView as its `image`, thus causing the image to appear onscreen. Or you could save it as a file. Or, as I'll explain in the next section, you could use it in another drawing.

First, I'll draw my image using UIKit:

```
UIGraphicsBeginImageContextWithOptions(CGSizeMake(100,100), false, 0)
let p = UIBezierPath(ovalInRect: CGRectMake(0,0,100,100))
UIColor.blueColor().setFill()
p.fill()
let im = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
// im is the blue circle image, do something with it here ...
```

Here's the same thing using Core Graphics:

```
UIGraphicsBeginImageContextWithOptions(CGSizeMake(100,100), false, 0)
let con = UIGraphicsGetCurrentContext()!
CGContextAddEllipseInRect(con, CGRectMake(0,0,100,100))
CGContextSetFillColorWithColor(con, UIColor.blueColor().CGColor)
CGContextFillPath(con)
let im = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
// im is the blue circle image, do something with it here ...
```

You may be wondering about the arguments to `UIGraphicsBeginImageContextWithOptions`. The first argument is obviously the size of the image to be created. The second argument declares whether the image should be opaque; if I had passed `true` instead of `false` here, my image would have a black background, which I don't want. The third argument specifies the image scale, corresponding to the UIImage `scale` property I discussed earlier; by passing `0`, I'm telling the system to set the scale for me in accordance with the main screen resolution, so my image will look good on both single-resolution and high-resolution devices.

The dance required to begin an image context, draw into it, extract the image, and end the context, is a bit tedious and error-prone. A utility function that I provide in Appendix B has the advantage that you provide just the drawing instructions, and the utility function does the dance and returns the image.

UIImage Drawing

A UIImage provides methods for drawing itself into the current context. We know how to obtain a UIImage, and we know how to obtain a graphics context and make it the current context, so we can experiment with these methods.

Here, I'll make a UIImage consisting of two pictures of Mars side by side (Figure 2-10):

```
let mars = UIImage(named:"Mars")!
let sz = mars.size
UIGraphicsBeginImageContextWithOptions(
    CGSizeMake(sz.width*2, sz.height), false, 0)
```

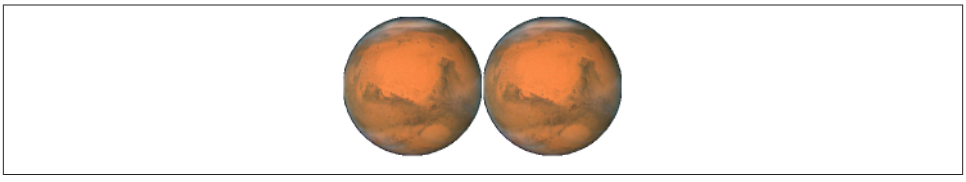

Figure 2-10. Two images of Mars combined side by side

```
mars.drawAtPoint(CGPointMake(0,0))
mars.drawAtPoint(CGPointMake(sz.width,0))
let im = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
```

Observe that image scaling works perfectly in that example. If we have multiple resolution versions of our original Mars image, the correct one for the current device is used, and is assigned the correct `scale` value. Our call to `UIGraphicsBeginImageContextWithOptions` has a third argument of `0`, so the image context that we are drawing into also has the correct `scale`. And the image that results from calling `UIGraphicsGetImageFromCurrentImageContext` has the correct `scale` as well. Thus, this same code produces an image that looks correct on the current device, whatever its screen resolution may be.

Additional UIImage methods let you scale an image into a desired rectangle as you draw, and specify the compositing (blend) mode whereby the image should combine with whatever is already present. To illustrate, I'll create an image showing Mars centered in another image of Mars that's twice as large, using the `.Multiply` blend mode (Figure 2-11):

```
let mars = UIImage(named:"Mars")!
let sz = mars.size
UIGraphicsBeginImageContextWithOptions(
    CGSizeMake(sz.width*2, sz.height*2), false, 0)
mars.drawInRect(CGRectMake(0,0,sz.width*2, sz.height*2))
mars.drawInRect(
    CGRectMake(sz.width/2.0, sz.height/2.0, sz.width, sz.height),
    blendMode: .Multiply, alpha: 1.0)
let im = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
```

There is no UIImage drawing method for specifying the source rectangle — that is, for times when you want to extract a smaller region of the original image. You can work around this by creating a smaller graphics context and positioning the image drawing so that the desired region falls into it. For example, to obtain an image of the right half of Mars, you'd make a graphics context half the width of the `mars` image, and then draw `mars` shifted left, so that only its right half intersects the graphics context. There is no

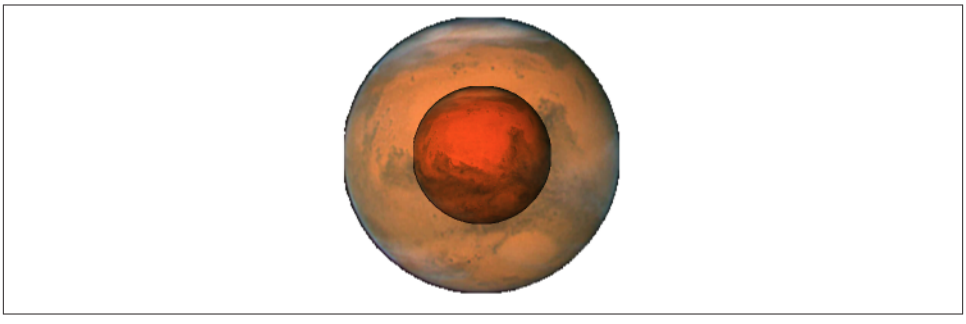

Figure 2-11. Two images of Mars in different sizes, composited

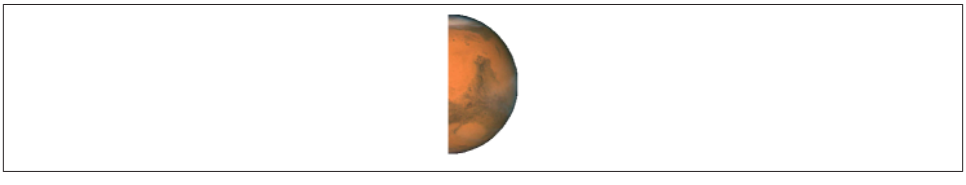

Figure 2-12. Half the original image of Mars

harm in doing this, and it's a perfectly standard strategy; the left half of `mars` simply isn't drawn (Figure 2-12):

```
let mars = UIImage(named:"Mars")!
let sz = mars.size
UIGraphicsBeginImageContextWithOptions(
    CGSizeMake(sz.width/2.0, sz.height), false, 0)
mars.drawAtPoint(CGPointMake(-sz.width/2.0,0))
let im = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
```

CGImage Drawing

The Core Graphics version of UIImage is CGImage. They are easily converted to one another: a UIImage has a `CGImage` property that accesses its Quartz image data, and you can make a UIImage from a CGImage using `init(CGImage:)` or its more configurable sibling, `init(CGImage:scale:orientation:)`.

A CGImage lets you create a new image directly from a rectangular region of the original image, which you can't do with UIImage. (A CGImage has other powers a UIImage doesn't have; for example, you can apply an image mask to a CGImage.) I'll demonstrate by splitting the image of Mars in half and drawing the two halves separately (Figure 2-13):

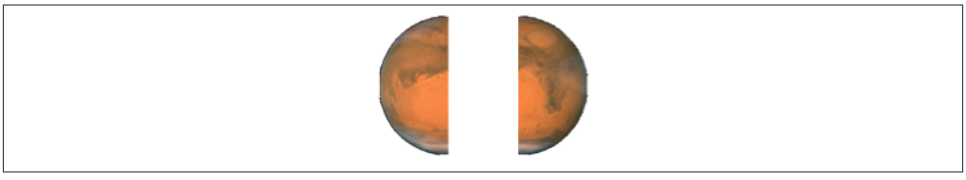

Figure 2-13. Image of Mars split in half (and flipped)

```
let mars = UIImage(named:"Mars")!
// extract each half as CGImage
let marsCG = mars.CGImage
let sz = mars.size
let marsLeft = CGImageCreateWithImageInRect(
    marsCG,
    CGRectMake(0,0,sz.width/2.0,sz.height))
let marsRight = CGImageCreateWithImageInRect(
    marsCG,
    CGRectMake(sz.width/2.0,0,sz.width/2.0,sz.height))
// draw each CGImage
UIGraphicsBeginImageContextWithOptions(
    CGSizeMake(sz.width*1.5, sz.height), false, 0)
let con = UIGraphicsGetCurrentContext()!
CGContextDrawImage(con,
    CGRectMake(0,0,sz.width/2.0,sz.height), marsLeft)
CGContextDrawImage(con,
    CGRectMake(sz.width,0,sz.width/2.0,sz.height), marsRight)
let im = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
```

But there's a problem with that example: the drawing is upside-down! It isn't rotated; it's mirrored top to bottom, or, to use the technical term, *flipped*. This phenomenon can arise when you create a CGImage and then draw it with `CGContextDrawImage`, and is due to a mismatch in the native coordinate systems of the source and target contexts.

There are various ways of compensating for this mismatch between the coordinate systems. One is to draw the CGImage into an intermediate UIImage and extract *another* CGImage from that. Example 2-1 presents a utility function for doing this.

Example 2-1. Utility for flipping an image drawing

```
func flip (im: CGImage) -> CGImage {
    let sz = CGSizeMake(
        CGFloat(CGImageGetWidth(im)),
        CGFloat(CGImageGetHeight(im)))
    UIGraphicsBeginImageContextWithOptions(sz, false, 0)
    CGContextDrawImage(UIGraphicsGetCurrentContext()!,
        CGRectMake(0, 0, sz.width, sz.height), im)
```

```
    let result = UIGraphicsGetImageFromCurrentImageContext().CGImage
    UIGraphicsEndImageContext()
    return result!
}
```

Armed with the utility function from Example 2-1, we can fix our calls to `CGContextDrawImage` in the previous example so that they draw the halves of Mars the right way up:

```
    CGContextDrawImage(con,
        CGRectMake(0,0,sz.width/2.0,sz.height), flip(marsLeft!))
    CGContextDrawImage(con,
        CGRectMake(sz.width,0,sz.width/2.0,sz.height), flip(marsRight!))
```

However, we've *still* got a problem: on a high-resolution device, if there is a high-resolution variant of our image file, the drawing comes out all wrong. The reason is that we are obtaining our initial Mars image using UIImage's `init(named:)`, which returns a UIImage that compensates for the increased size of a high-resolution image by setting its own `scale` property to match. But a CGImage doesn't have a `scale` property, and knows nothing of the fact that the image dimensions are increased! Therefore, on a high-resolution device, the CGImage that we extract from our Mars UIImage as `mars.CGImage` is larger (in each dimension) than `mars.size`, and all our calculations after that are wrong.

The best solution for dealing a CGImage, therefore, is to wrap it in a UIImage and draw the UIImage instead of the CGImage. The UIImage can be formed in such a way as to compensate for scale: call `init(CGImage:scale:orientation:)` as you form the UIImage from the CGImage. Moreover, by drawing a UIImage instead of a CGImage, we avoid the flipping problem! So here's an approach that deals with both flipping and scale, with no need for the `flip` utility:

```
    let mars = UIImage(named:"Mars")!
    let sz = mars.size
    let marsCG = mars.CGImage
    let szCG = CGSizeMake(
        CGFloat(CGImageGetWidth(marsCG)),
        CGFloat(CGImageGetHeight(marsCG)))
    let marsLeft =
        CGImageCreateWithImageInRect(
            marsCG, CGRectMake(0,0,szCG.width/2.0,szCG.height))
    let marsRight =
        CGImageCreateWithImageInRect(
            marsCG, CGRectMake(szCG.width/2.0,0,szCG.width/2.0,szCG.height))
    UIGraphicsBeginImageContextWithOptions(
        CGSizeMake(sz.width*1.5, sz.height), false, 0)
    // instead of calling flip, pass through UIImage
    UIImage(CGImage: marsLeft!, scale: mars.scale,
        orientation: mars.imageOrientation)
        .drawAtPoint(CGPointMake(0,0))
```

Why Flipping Happens

The ultimate source of accidental flipping is that Core Graphics comes from the OS X world, where the coordinate system's origin is located by default at the bottom left and the positive y-direction is upward, whereas on iOS the origin is located by default at the top left and the positive y-direction is downward. In most drawing situations, no problem arises, because the coordinate system of the graphics context is adjusted to compensate. Thus, the default coordinate system for drawing in a Core Graphics context on iOS has the origin at the top left, just as you expect. But creating and drawing a CGImage exposes the "impedance mismatch" between the two worlds.

```
UIImage(CGImage: marsRight!, scale: mars.scale,
    orientation: mars.imageOrientation)
    .drawAtPoint(CGPointMake(sz.width,0))
let im = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
```

Yet another solution to flipping is to apply a transform to the graphics context before drawing the CGImage, effectively flipping the context's internal coordinate system. This is elegant, but can be confusing if there are other transforms in play. I'll talk more about graphics context transforms later in this chapter.

Snapshots

An entire view — anything from a single button to your whole interface, complete with its contained hierarchy of views — can be drawn into the current graphics context by calling the UIView instance method `drawViewHierarchyInRect:afterScreenUpdates:`. (This method is much faster than the CALayer method `renderInContext:`; nevertheless, `renderInContext:` does still come in handy, as I'll show in Chapter 5.) The result is a *snapshot* of the original view: it looks like the original view, but it's basically just a bitmap image of it, a lightweight visual duplicate.

An even faster way to obtain a snapshot of a view is to use the UIView (or UIScreen) instance method `snapshotViewAfterScreenUpdates:`. The result is a UIView, not a UIImage; it's rather like a UIImageView that knows how to draw only one image, namely the snapshot. Such a snapshot view will typically be used as is, but you can enlarge its bounds and the snapshot image will stretch. If you want the stretched snapshot to behave like a resizable image, call `resizableSnapshotViewFromRect:afterScreenUpdates:withCapInsets:` instead. It is perfectly reasonable to make a snapshot view from a snapshot view.

Snapshots are useful because of the dynamic nature of the iOS interface. For example, you might place a snapshot of a view in your interface in front of the real view to hide

what's happening, or use it during an animation to present the illusion of a view moving when in fact it's just a snapshot.

Here's an example from one of my apps. It's a card game, and its `views` portray cards. I want to animate the removal of all those cards from the board, flying away to an offscreen point. But I don't want to animate the views themselves! They need to stay put, to portray future cards. So I make a snapshot view of each of the card views; I then make the card views invisible, put the snapshot views in their place, and animate the snapshot views. This code will mean more to you after you've read Chapter 4, but the strategy is evident:

```
for v in views {
    let snapshot = v.snapshotViewAfterScreenUpdates(false)
    let snap = MySnapBehavior(item:snapshot, snapToPoint:CGPointMake(
        self.anim.referenceView!.bounds.midX,
        -self.anim.referenceView!.bounds.height))
    self.snaps.append(snapshot) // keep a list so we can remove them later
    snapshot.frame = v.frame
    v.hidden = true
    self.anim.referenceView!.addSubview(snapshot)
    self.anim.addBehavior(snap)
}
```

CIFilter and CIImage

The "CI" in CIFilter and CIImage stands for Core Image, a technology for transforming images through mathematical filters. Core Image started life on the desktop (OS X), and when it was migrated into iOS, some of the filters available on the desktop were not available in iOS (presumably because they were too intensive mathematically for a mobile device). Over the years, however, more and more OS X filters were added to the iOS repertory, and now, new in iOS 9, the two have complete parity: *all* OS X filters are available in iOS, and the two platforms have nearly identical APIs.

A filter is a CIFilter. The available filters fall naturally into several broad categories:

Patterns and gradients
: These filters create CIImages that can then be combined with other CIImages, such as a single color, a checkerboard, stripes, or a gradient.

Compositing
: These filters combine one image with another, using compositing blend modes familiar from image processing programs such as Photoshop.

Color
: These filters adjust or otherwise modify the colors of an image. Thus you can alter an image's saturation, hue, brightness, contrast, gamma and white point, exposure, shadows and highlights, and so on.

Geometric
: These filters perform basic geometric transformations on an image, such as scaling, rotation, and cropping.

Transformation
: These filters distort, blur, or stylize an image.

Transition
: These filters provide a frame of a transition between one image and another; by asking for frames in sequence, you can animate the transition (I'll demonstrate in Chapter 4).

Special purpose
: These filters perform highly specialized operations such as face detection and generation of QR codes.

The basic use of a CIFilter is quite simple:

- You specify what filter you want by supplying its string name; to learn what these names are, consult Apple's *Core Image Filter Reference*, or call the CIFilter class method `filterNamesInCategories:` with a `nil` argument.
- Each filter has a small number of keys and values that determine its behavior (as if a filter were a kind of dictionary). You can learn about these keys entirely in code, but typically you'll consult the documentation. For each key that you're interested in, you supply a key–value pair. In supplying values, a number must be wrapped up as an NSNumber (Swift will take care of this for you), and there are a few supporting classes such as CIVector (like CGPoint and CGRect combined) and CIColor, whose use is easy to grasp.

Among a CIFilter's keys are the input image or images on which the filter is to operate; such an image must be a CIImage. You can obtain this CIImage from a CGImage with `init(CGImage:)` or from a UIImage with `init(image:)`.

Do not attempt, as a shortcut, to obtain a CIImage directly from a UIImage through the UIImage's `CIImage` property. This property does *not* transform a UIImage into a CIImage! It merely points to the CIImage that *already* backs the UIImage, if the UIImage *is* backed by a CIImage; but your images are *not* backed by a CIImage, but rather by a CGImage. I'll explain where a CIImage-backed UIImage comes from in just a moment.

Alternatively, you can obtain a CIImage as the output of a filter — which means that *filters can be chained together*.

There are three ways to describe and use a filter:

- Create the filter with CIFilter's `init(name:)`. Now append the keys and values by calling `setValue:forKey:` repeatedly, or by calling `setValuesForKeysWithDictionary:`. Obtain the output CIImage as the filter's `outputImage`.
- Create the filter and supply the keys and values in a single move, by calling CIFilter's `init(name:withInputParameters:)`. Obtain the output CIImage as the filter's `outputImage`.
- If a CIFilter requires an input image and you already have a CIImage to fulfill this role, specify the filter and supply the keys and values, *and receive the output CIImage as a result*, all in a single move, by calling the CIImage instance method `imageByApplyingFilter:withInputParameters:`.

As you build a chain of filters, nothing actually happens. The only calculation-intensive move comes at the very end, when you transform the final CIImage in the chain into a bitmap drawing. This is called *rendering* the image. There are two main ways to do this:

With a CIContext
: Create a CIContext (by calling `init(options:)`) and then call `createCGImage:fromRect:`, handing it the final CIImage as the first argument. This renders the image. The only mildly tricky thing here is that a CIImage doesn't have a frame or bounds; it has an `extent`. You will often use this as the second argument to `createCGImage:fromRect:`. The final output CGImage is ready for any purpose, such as for display in your app, for transformation into a UIImage, or for use in further drawing.

  This approach has the advantage of giving you full control over the moment when rendering takes place. But be warned: creating a CIContext is expensive! Wherever possible, create your CIContext once, beforehand — preferably, once per app — and reuse it each time you render.

With a UIImage
: Create a UIImage directly from the final CIImage by calling `init(CIImage:)` or `init(CIImage:scale:orientation:)`. You can then draw the UIImage into some graphics context. At the moment of drawing, the image is rendered.

Apple claims that you can simply hand a UIImage created by calling `init(CIImage:)` to a UIImageView, as its `image`, and that the UIImageView will render the image. In my experience, this is *not true*. You must draw the image *explicitly* in order to render it.

(Other ways to render a CIImage involve things like GLKView or CAEAGLLayer, which are not discussed in this book. They have the advantage of being very fast and suitable for animated or rapid rendering.)

Figure 2-14. A photo of me, vignetted

To illustrate, I'll start with an ordinary photo of myself (it's true I'm wearing a motorcycle helmet, but it's still ordinary) and create a circular vignette effect (Figure 2-14). We derive from the image of me (`moi`) a CIImage (`moici`). We use a CIFilter (`grad`) to form a radial gradient between the default colors of white and black. Then we use a second CIFilter that treats the radial gradient as a mask for blending between the photo of me and a default clear background: where the radial gradient is white (everything inside the gradient's inner radius) we see just me, and where the radial gradient is black (everything outside the gradient's outer radius) we see just the clear color, with a gradation in between, so that the image fades away in the circular band between the gradient's radii. The code illustrates two different ways of configuring a CIFilter:

```
let moi = UIImage(named:"Moi")!
let moici = CIImage(image:moi)!
let moiextent = moici.extent
let center = CIVector(x: moiextent.width/2.0, y: moiextent.height/2.0)
let smallerDimension = min(moiextent.width, moiextent.height)
let largerDimension = max(moiextent.width, moiextent.height)
// first filter
let grad = CIFilter(name: "CIRadialGradient")!
grad.setValue(center, forKey:"inputCenter")
grad.setValue(smallerDimension/2.0 * 0.85, forKey:"inputRadius0")
grad.setValue(largerDimension/2.0, forKey:"inputRadius1")
let gradimage = grad.outputImage!
// second filter
let blendimage = moici.imageByApplyingFilter(
    "CIBlendWithMask", withInputParameters: [
        "inputMaskImage":gradimage
    ])
```

We now have the final CIImage in the chain (`blendimage`); remember, the processor has not yet performed any rendering. Now, however, we want to generate the final bitmap and display it. Let's say we're going to display it as the `image` of a UIImageView. We can do it in two different ways. We can create a CGImage by passing the CIImage

through a CIContext which we have prepared beforehand as a property, `self.context`, by calling `CIContext(options: nil)`:

```
let moicg = self.context.createCGImage(blendimage, fromRect: moiextent)
self.iv.image = UIImage(CGImage: moicg)
```

Alternatively, we can capture our final CIImage as a UIImage and then draw with it in order to generate the bitmap output of the filter chain:

```
UIGraphicsBeginImageContextWithOptions(moiextent.size, false, 0)
UIImage(CIImage: blendimage).drawInRect(moiextent)
let im = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
self.iv.image = im
```

A filter chain can be encapsulated into a single custom filter by subclassing CIFilter. Your subclass just needs to override the `outputImage` property (and possibly other methods such as `setDefaults`), with additional properties to make it key–value coding compliant for any input keys. Here's our vignette filter as a simple CIFilter subclass, where the input keys are the input image and a percentage that adjusts the gradient's smaller radius:

```
class MyVignetteFilter : CIFilter {
    var inputImage : CIImage?
    var inputPercentage : NSNumber? = 1.0
    override var outputImage : CIImage? {
        return self.makeOutputImage()
    }
    private func makeOutputImage () -> CIImage? {
        guard let inputImage = self.inputImage else {return nil}
        guard let inputPercentage = self.inputPercentage else {return nil}
        let extent = inputImage.extent
        let grad = CIFilter(name: "CIRadialGradient")!
        let center = CIVector(x: extent.width/2.0, y: extent.height/2.0)
        let smallerDimension = min(extent.width, extent.height)
        let largerDimension = max(extent.width, extent.height)
        grad.setValue(center, forKey:"inputCenter")
        grad.setValue(smallerDimension/2.0 * CGFloat(inputPercentage),
            forKey:"inputRadius0")
        grad.setValue(largerDimension/2.0, forKey:"inputRadius1")
        let blend = CIFilter(name: "CIBlendWithMask")!
        blend.setValue(inputImage, forKey: "inputImage")
        blend.setValue(grad.outputImage, forKey: "inputMaskImage")
        return blend.outputImage
    }
}
```

And here's how to use our CIFilter subclass and display its output:

```
let vig = MyVignetteFilter()
let moici = CIImage(image: UIImage(named:"Moi")!)!
vig.setValuesForKeysWithDictionary([
    "inputImage":moici,
    "inputPercentage":0.7
])
let outim = vig.outputImage!
let outimcg = self.context.createCGImage(outim, fromRect: outim.extent)
self.iv.image = UIImage(CGImage: outimcg)
```

You can also create your own CIFilter from scratch — not by combining existing filters, but by coding the actual mathematics of the filter. The details are outside the scope of this book; you'll want to look at the CIKernel class.

Blur and Vibrancy Views

Certain views on iOS, such as navigation bars and the control center, are translucent and display a blurred rendition of what's behind them. To help you imitate this effect, iOS provides the UIVisualEffectView class. You can place other views in front of a UIVisualEffectView, but any subviews should be placed inside its `contentView`. To tint what's seen through a UIVisualEffectView, set the `backgroundColor` of its `contentView`.

To use a UIVisualEffectView, create it with `init(effect:)`; the `effect:` argument will be an instance of a UIVisualEffect subclass:

UIBlurEffect
: To initialize a UIBlurEffect, call `init(style:)`; the styles (UIBlurEffectStyle) are `.Dark`, `.Light`, and `.ExtraLight`. (`.ExtraLight` is suitable particularly for pieces of interface that function like a navigation bar or toolbar.) For example:

```
let fuzzy = UIVisualEffectView(effect:(UIBlurEffect(style:.Light)))
```

UIVibrancyEffect
: To initialize a UIVibrancyEffect, call `init(forBlurEffect:)`. Vibrancy tints a view so as to make it harmonize with the blurred colors underneath it. The intention here is that the vibrancy effect view should sit in front of a blur effect view, typically in its `contentView`, adding vibrancy to a single UIView that's inside its *own* `contentView`; you tell the vibrancy effect what the underlying blur effect is, so that they harmonize. You can fetch a visual effect view's blur effect as its `effect` property, but that's a UIVisualEffect — the superclass — so you'll have to cast to a UIBlurEffect in order to hand it to `init(forBlurEffect:)`.

Here's an example of a blur effect view covering and blurring the interface (`mainview`), and containing a UILabel wrapped in a vibrancy effect view (Figure 2-15):

Figure 2-15. A blurred background and a vibrant label

```
let blur = UIVisualEffectView(effect: UIBlurEffect(style: .ExtraLight))
blur.frame = mainview.bounds
blur.autoresizingMask = [.FlexibleWidth, .FlexibleHeight]
let vib = UIVisualEffectView(effect: UIVibrancyEffect(
    forBlurEffect: blur.effect as! UIBlurEffect))
let lab = UILabel()
lab.text = "Hello, world!"
lab.sizeToFit()
vib.frame = lab.frame
vib.contentView.addSubview(lab)
vib.center = CGPointMake(blur.bounds.midX, blur.bounds.midY)
vib.autoresizingMask = [.FlexibleTopMargin, .FlexibleBottomMargin,
    .FlexibleLeftMargin, .FlexibleRightMargin]
blur.contentView.addSubview(vib)
mainview.addSubview(blur)
```

Apple seems to think that vibrancy makes a view more legible in conjunction with the underlying blur, but I'm not persuaded. The vibrant view's color is made to harmonize with the blurred color behind it, but harmony implies similarity, which can make the vibrant view *less* legible. You'll have to experiment. With the particular interface I'm blurring, the vibrant label in Figure 2-15 looks okay with a `.Dark` or `.ExtraLight` blur effect view, but is very hard to see with a `.Light` blur effect view.

There are a lot of useful additional notes, well worth consulting, in the *UIVisualEffectView.h* header. For example, the header points out that an image displayed in an image view needs to be a template image in order to receive the benefit of a vibrancy effect view.

Observe that both a blur effect view and a blur effect view with an embedded vibrancy effect view are available as built-in objects in the nib editor.

Drawing a UIView

The examples of drawing so far in this chapter have mostly produced UIImage objects, chiefly by calling `UIGraphicsBeginImageContextWithOptions` to obtain a graphics context, suitable for display by a UIImageView or any other interface object that knows how to display an image. But, as I've already explained, a UIView provides a graphics

context; whatever you draw into that graphics context will appear in that view. The technique here is to subclass UIView and implement the subclass's `drawRect:` method.

So, for example, let's say we have a UIView subclass called MyView. You would then instantiate this class and get the instance into the view hierarchy. One way to do this would be to drag a UIView into a view in the nib editor and set its class to MyView in the Identity inspector; another would be to create the MyView instance and put it into the interface in code.

The result is that, from time to time, MyView's `drawRect:` will be called. This is your subclass, so you get to write the code that runs at that moment. Whatever you draw will appear inside the MyView instance. There will usually be no need to call `super`, since UIView's own implementation of `drawRect:` does nothing. At the time that `drawRect:` is called, the current graphics context has already been set to the view's own graphics context. You can use Core Graphics functions or UIKit convenience methods to draw into that context. I gave some basic examples earlier in this chapter ("Graphics Contexts" on page 76).

You should *never* call `drawRect:` yourself! If a view needs updating and you want its `drawRect:` called, send the view the `setNeedsDisplay` message. This will cause `drawRect:` to be called at the next proper moment. Also, don't override `drawRect:` unless you are assured that this is legal. For example, it is not legal to override `drawRect:` in a subclass of UIImageView; you cannot combine your drawing with that of the UIImageView.

The need to draw in real time, on demand, surprises some beginners, who worry that drawing may be a time-consuming operation. This can indeed be a reasonable consideration, and where the same drawing will be used in many places in your interface, it may well make sense to construct a UIImage instead, once, and then reuse that UIImage by drawing it in a view's `drawRect:`. In general, however, you should not optimize prematurely. The code for a drawing operation may appear verbose and yet be extremely fast. Moreover, the iOS drawing system is efficient; it doesn't call `drawRect:` unless it has to (or is told to, through a call to `setNeedsDisplay`), and once a view has drawn itself, the result is cached so that the cached drawing can be reused instead of repeating the drawing operation from scratch. (Apple refers to this cached drawing as the view's *bitmap backing store*.) You can readily satisfy yourself of this fact with some caveman debugging, logging in your `drawRect:` implementation; you may be amazed to discover that your custom UIView's `drawRect:` code is called only once in the entire lifetime of the app! In fact, moving code to `drawRect:` is commonly a way to *increase* efficiency. This is because it is more efficient for the drawing engine to render directly onto the screen than for it to render offscreen and then copy those pixels onto the screen.

Where drawing is extensive and can be compartmentalized into sections, you may be able to gain some additional efficiency by paying attention to the `rect` parameter passed into `drawRect:`. It designates the region of the view's bounds that needs refreshing. Normally, this is the view's entire bounds; but if you call `setNeedsDisplayInRect:`, it will be the CGRect that you passed in as argument. You could respond by drawing only what goes into those bounds; but even if you don't, your drawing will be clipped to those bounds, so, while you may not spend less time drawing, the system will draw more efficiently.

When creating a custom UIView subclass instance in code, you may be surprised (and annoyed) to find that the view has a black background:

```
let mv = MyView(frame:CGRectMake(20,20,150,100))
self.view.addSubview(mv)
```

This can be frustrating if what you expected and wanted was a transparent background, and it's a source of considerable confusion among beginners. The black background arises when two things are true:

- The view's `backgroundColor` is `nil`.
- The view's `opaque` is `true`.

Unfortunately, when creating a UIView in code, both those things *are* true by default! So if you don't want the black background, you must do something about one or the other of them (or both). If a view isn't going to be opaque, its `opaque` should be set to `false` anyway, so that's probably the cleanest solution:

```
let mv = MyView(frame:CGRectMake(20,20,150,100))
self.view.addSubview(mv)
mv.opaque = false
```

Alternatively, this being your own UIView subclass, you could implement its `init(frame:)` (the designated initializer) to have the view set its *own* `opaque` to `false`:

```
override init(frame: CGRect) {
    super.init(frame:frame)
    self.opaque = false
}
```

With a UIView created in the nib, on the other hand, the black background problem doesn't arise. This is because such a UIView's `backgroundColor` is not `nil`. The nib assigns it *some* actual background color, even if that color is `UIColor.clearColor()`.

Of course, if a view fills its rectangle with opaque drawing or has an opaque background color, you can leave `opaque` set to `true` and gain some drawing efficiency (see Chapter 1).

Graphics Context Settings

As you draw in a graphics context, the drawing obeys the context's current settings. Thus, the procedure is always to configure the context's settings first, and then draw. For example, to draw a red line followed by a blue line, you would first set the context's line color to red, and then draw the first line; then you'd set the context's line color to blue, and then draw the second line. To the eye, it appears that the redness and blueness are properties of the individual lines, but in fact, at the time you draw each line, line color is a feature of the entire graphics context. This is true regardless of whether you use UIKit methods or Core Graphics functions.

A graphics context thus has, at every moment, a *state*, which is the sum total of all its settings; the way a piece of drawing looks is the result of what the graphics context's state was at the moment that piece of drawing was performed. To help you manipulate entire states, the graphics context provides a *stack* for holding states. Every time you call `CGContextSaveGState`, the context pushes the entire current state onto the stack; every time you call `CGContextRestoreGState`, the context retrieves the state from the top of the stack (the state that was most recently pushed) and sets itself to that state.

Thus, a common pattern is:

1. Call `CGContextSaveGState`.
2. Manipulate the context's settings, thus changing its state.
3. Draw.
4. Call `CGContextRestoreGState` to restore the state and the settings to what they were before you manipulated them.

You do not have to do this before *every* manipulation of a context's settings, however, because settings don't necessarily conflict with one another or with past settings. You can set the context's line color to red and then later to blue without any difficulty. But in certain situations you do want your manipulation of settings to be undoable, and I'll point out several such situations later in this chapter.

Many of the settings that constitute a graphics context's state, and that determine the behavior and appearance of drawing performed at that moment, are similar to those of any drawing application. Here are some of them, along with some of the commands that determine them. I list Core Graphics functions, followed by some UIKit convenience methods that call them:

Line thickness and dash style
: `CGContextSetLineWidth`, `CGContextSetLineDash` (and UIBezierPath `lineWidth`, `setLineDash:count:phase:`)

Line end-cap style and join style
: `CGContextSetLineCap`, `CGContextSetLineJoin`, `CGContextSetMiterLimit` (and UIBezierPath `lineCapStyle`, `lineJoinStyle`, `miterLimit`)

Line color or pattern
: `CGContextSetRGBStrokeColor`, `CGContextSetGrayStrokeColor`, `CGContextSetStrokeColorWithColor`, `CGContextSetStrokePattern` (and UIColor `setStroke`)

Fill color or pattern
: `CGContextSetRGBFillColor`, `CGContextSetGrayFillColor`, `CGContextSetFillColorWithColor`, `CGContextSetFillPattern` (and UIColor `setFill`)

Shadow
: `CGContextSetShadow`, `CGContextSetShadowWithColor`

Overall transparency and compositing
: `CGContextSetAlpha`, `CGContextSetBlendMode`

Anti-aliasing
: `CGContextSetShouldAntialias`

Additional settings include:

Clipping area
: Drawing outside the clipping area is not physically drawn.

Transform (or "CTM," for "current transform matrix")
: Changes how points that you specify in subsequent drawing commands are mapped onto the physical space of the canvas.

Many of these settings will be illustrated by examples later in this chapter.

Paths and Shapes

By issuing a series of instructions for moving an imaginary pen, you construct a *path*, tracing it out from point to point. You must first tell the pen where to position itself, setting the current point; after that, you issue a series of commands telling it how to trace out each subsequent piece of the path. Each additional piece of the path starts at the current point; its end becomes the new current point.

Note that a path, in and of itself, does *not* constitute drawing! First you provide a path; *then* you draw. Drawing can mean stroking the path or filling the path, or both. Again, this should be a familiar notion from certain drawing applications.

Here are some path-drawing commands you're likely to give:

Position the current point
: `CGContextMoveToPoint`

Trace a line
: `CGContextAddLineToPoint`, `CGContextAddLines`

Trace a rectangle
: `CGContextAddRect`, `CGContextAddRects`

Trace an ellipse or circle
: `CGContextAddEllipseInRect`

Trace an arc
: `CGContextAddArcToPoint`, `CGContextAddArc`

Trace a Bezier curve with one or two control points
: `CGContextAddQuadCurveToPoint`, `CGContextAddCurveToPoint`

Close the current path
: `CGContextClosePath`. This appends a line from the last point of the path to the first point. There's no need to do this if you're about to fill the path, since it's done for you.

Stroke or fill the current path
: `CGContextStrokePath`, `CGContextFillPath`, `CGContextEOFillPath`, `CGContextDrawPath`. Stroking or filling the current path *clears the path*. Use `CGContextDrawPath` if you want both to fill and to stroke the path in a single command, because if you merely stroke it first with `CGContextStrokePath`, the path is cleared and you can no longer fill it. There are also a lot of convenience functions that create a path and stroke or fill it all in a single move:

  - `CGContextStrokeLineSegments`
  - `CGContextStrokeRect`
  - `CGContextStrokeRectWithWidth`
  - `CGContextFillRect`
  - `CGContextFillRects`
  - `CGContextStrokeEllipseInRect`
  - `CGContextFillEllipseInRect`

A path can be compound, meaning that it consists of multiple independent pieces. For example, a single path might consist of two separate closed shapes: a rectangle and a circle. When you call `CGContextMoveToPoint` in the middle of constructing a path (that is, after tracing out a path and without clearing it by filling or stroking it), you pick up the imaginary pen and move it to a new location without tracing a segment, thus preparing to start an independent piece of the same path. If you're worried, as you begin to trace out a path, that there might be an existing path and that your new path might

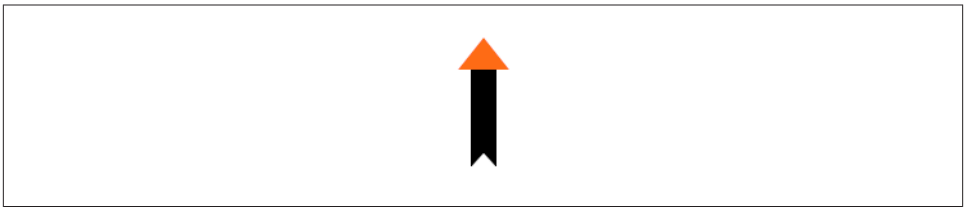

Figure 2-16. A simple path drawing

be seen as a compound part of that existing path, you can call `CGContextBeginPath` to specify that this is a different path; many of Apple's examples do this, but in practice I usually do not find it necessary.

To illustrate the typical use of path-drawing commands, I'll generate the up-pointing arrow shown in Figure 2-16. This might not be the best way to create the arrow, and I'm deliberately avoiding use of the convenience functions, but it's clear and shows a nice basic variety of typical commands:

```
// obtain the current graphics context
let con = UIGraphicsGetCurrentContext()!
// draw a black (by default) vertical line, the shaft of the arrow
CGContextMoveToPoint(con, 100, 100)
CGContextAddLineToPoint(con, 100, 19)
CGContextSetLineWidth(con, 20)
CGContextStrokePath(con)
// draw a red triangle, the point of the arrow
CGContextSetFillColorWithColor(con, UIColor.redColor().CGColor)
CGContextMoveToPoint(con, 80, 25)
CGContextAddLineToPoint(con, 100, 0)
CGContextAddLineToPoint(con, 120, 25)
CGContextFillPath(con)
// snip a triangle out of the shaft by drawing in Clear blend mode
CGContextMoveToPoint(con, 90, 101)
CGContextAddLineToPoint(con, 100, 90)
CGContextAddLineToPoint(con, 110, 101)
CGContextSetBlendMode(con, .Clear)
CGContextFillPath(con)
```

If a path needs to be reused or shared, you can encapsulate it as a CGPath. You can copy the graphics context's current path using `CGContextCopyPath`. Even without a graphics context, you can create a new CGMutablePath (with `CGPathCreateMutable`, for example) and construct the path using various CGPath functions that parallel the CGContext path-construction functions. Also, there are a number of CGPath functions for creating a path based on simple geometry or based on an existing path:

- `CGPathCreateWithRect`
- `CGPathCreateWithEllipseInRect`

- `CGPathCreateWithRoundedRect`
- `CGPathCreateCopyByStrokingPath`
- `CGPathCreateCopyByDashingPath`
- `CGPathCreateCopyByTransformingPath`

The UIKit class UIBezierPath wraps CGPath (in its `CGPath` property); it provides methods parallel to the CGContext and CGPath functions for constructing a path, such as:

- `init(rect:)`
- `init(ovalInRect:)`
- `init(roundedRect:cornerRadius:)`
- `moveToPoint:`
- `addLineToPoint:`
- `addArcWithCenter:radius:startAngle:endAngle:clockwise:`
- `addQuadCurveToPoint:controlPoint:`
- `addCurveToPoint:controlPoint1:controlPoint2:`
- `closePath`

When you call the UIBezierPath instance method `fill` or `stroke` (or `fillWithBlendMode:alpha:` or `strokeWithBlendMode:alpha:`), the current graphics context settings are saved, the wrapped CGPath is made the current graphics context's path and stroked or filled, and the current graphics context settings are restored.

Thus, using UIBezierPath together with UIColor, we could rewrite our arrow-drawing routine entirely with UIKit methods:

```
let p = UIBezierPath()
// shaft
p.moveToPoint(CGPointMake(100,100))
p.addLineToPoint(CGPointMake(100, 19))
p.lineWidth = 20
p.stroke()
// point
UIColor.redColor().set()
p.removeAllPoints()
p.moveToPoint(CGPointMake(80,25))
p.addLineToPoint(CGPointMake(100, 0))
p.addLineToPoint(CGPointMake(120, 25))
p.fill()
// snip
p.removeAllPoints()
```

```
p.moveToPoint(CGPointMake(90,101))
p.addLineToPoint(CGPointMake(100, 90))
p.addLineToPoint(CGPointMake(110, 101))
p.fillWithBlendMode(.Clear, alpha:1.0)
```

There's no savings of code here over calling Core Graphics functions, so your choice of Core Graphics or UIKit is a matter of taste. UIBezierPath is also useful when you want to capture a CGPath and pass it around as an object; an example appears in Chapter 21. See also the discussion in Chapter 3 of CAShapeLayer, which takes a CGPath that you've constructed and draws it for you within its own bounds.

Clipping

Instead of drawing a path by stroking or filling, you might use a path to mask out areas, protecting them from future drawing. This is called *clipping*. By default, a graphics context's clipping region is the entire graphics context: you can draw anywhere within the context.

The clipping area is a feature of the context as a whole, and any new clipping area is applied by intersecting it with the existing clipping area; so if you apply your own clipping region, the way to remove it from the graphics context later is to plan ahead and wrap things with calls to `CGContextSaveGState` and `CGContextRestoreGState`.

To illustrate, I'll rewrite the code that generated our original arrow (Figure 2-16) to use clipping instead of a blend mode to "punch out" the triangular notch in the tail of the arrow. This is a little tricky, because what we want to clip to is not the region inside the triangle but the region outside it. To express this, we'll use a compound path consisting of more than one closed area — the triangle, and the drawing area as a whole (which we can obtain with `CGContextGetClipBoundingBox`).

Both when filling a compound path and when using it to express a clipping region, the system follows one of two rules:

Winding rule
: The fill or clipping area is denoted by an alternation in the direction (clockwise or counterclockwise) of the path demarcating each region.

Even-odd rule (EO)
: The fill or clipping area is denoted by a simple count of the paths demarcating each region.

Our situation is extremely simple, so it's easier to use the even-odd rule. So we set up the clipping area using `CGContextEOClip` and then draw the arrow:

```
// obtain the current graphics context
let con = UIGraphicsGetCurrentContext()!
// punch triangular hole in context clipping region
CGContextMoveToPoint(con, 90, 100)
```

How Big Is My Context?

At first blush, it appears that there's no way to learn a graphics context's size. Typically, this doesn't matter, because either you created the graphics context or it's the graphics context of some object whose size you know, such as a UIView. But in fact, because the default clipping region of a graphics context is the entire context, you can use `CGContextGetClipBoundingBox` to learn the context's "bounds" (before changing the clipping region, of course).

```
CGContextAddLineToPoint(con, 100, 90)
CGContextAddLineToPoint(con, 110, 100)
CGContextClosePath(con)
CGContextAddRect(con, CGContextGetClipBoundingBox(con))
CGContextEOClip(con)
// draw the vertical line
CGContextMoveToPoint(con, 100, 100)
CGContextAddLineToPoint(con, 100, 19)
CGContextSetLineWidth(con, 20)
CGContextStrokePath(con)
// draw the red triangle, the point of the arrow
CGContextSetFillColorWithColor(con, UIColor.redColor().CGColor)
CGContextMoveToPoint(con, 80, 25)
CGContextAddLineToPoint(con, 100, 0)
CGContextAddLineToPoint(con, 120, 25)
CGContextFillPath(con)
```

The UIBezierPath clipping commands are `usesEvenOddFillRule` and `addClip`.

Gradients

Gradients can range from the simple to the complex. A simple gradient (which is all I'll describe here) is determined by a color at one endpoint along with a color at the other endpoint, plus (optionally) colors at intermediate points; the gradient is then painted either linearly between two points or radially between two circles.

You can't use a gradient as a path's fill color, but you can restrict a gradient to a path's shape by clipping, which amounts to the same thing.

To illustrate, I'll redraw our arrow, using a linear gradient as the "shaft" of the arrow (Figure 2-17):

```
// obtain the current graphics context
let con = UIGraphicsGetCurrentContext()!
CGContextSaveGState(con)
// punch triangular hole in context clipping region
CGContextMoveToPoint(con, 90, 100)
CGContextAddLineToPoint(con, 100, 90)
```

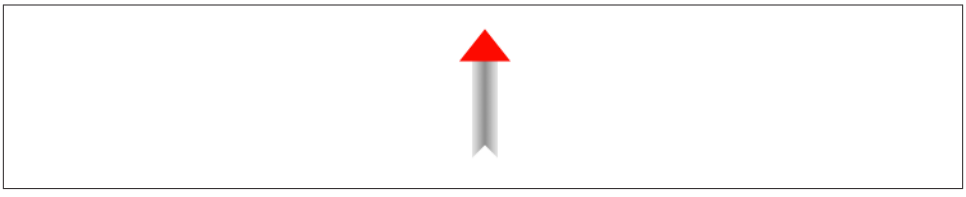

Figure 2-17. Drawing with a gradient

```
CGContextAddLineToPoint(con, 110, 100)
CGContextClosePath(con)
CGContextAddRect(con, CGContextGetClipBoundingBox(con))
CGContextEOClip(con)
// draw the vertical line, add its shape to the clipping region
CGContextMoveToPoint(con, 100, 100)
CGContextAddLineToPoint(con, 100, 19)
CGContextSetLineWidth(con, 20)
CGContextReplacePathWithStrokedPath(con)
CGContextClip(con)
// draw the gradient
let locs : [CGFloat] = [ 0.0, 0.5, 1.0 ]
let colors : [CGFloat] = [
    0.8, 0.4, // starting color, transparent light gray
    0.1, 0.5, // intermediate color, darker less transparent gray
    0.8, 0.4, // ending color, transparent light gray
]
let sp = CGColorSpaceCreateDeviceGray()
let grad =
    CGGradientCreateWithColorComponents (sp, colors, locs, 3)
CGContextDrawLinearGradient (
    con, grad, CGPointMake(89,0), CGPointMake(111,0), [])
// done clipping
CGContextRestoreGState(con)
// draw the red triangle, the point of the arrow
CGContextSetFillColorWithColor(con, UIColor.redColor().CGColor)
CGContextMoveToPoint(con, 80, 25)
CGContextAddLineToPoint(con, 100, 0)
CGContextAddLineToPoint(con, 120, 25)
CGContextFillPath(con)
```

The call to `CGContextReplacePathWithStrokedPath` pretends to stroke the current path, using the current line width and other line-related context state settings, but then creates a new path representing the outside of that stroked path. Thus, instead of a thick line we have a rectangular region that we can use as the clip region.

We then create the gradient and paint it. The procedure is verbose but simple; everything is boilerplate. We describe the gradient as an array of locations on the continuum between one endpoint (`0.0`) and the other endpoint (`1.0`), along with the color components of the colors corresponding to each location; in this case, I want the gradient

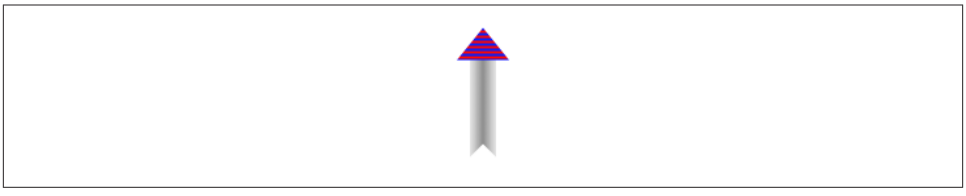

Figure 2-18. A patterned fill

to be lighter at the edges and darker in the middle, so I use three locations, with the dark one at `0.5`. We must also supply a color space; this will tell the gradient how to interpret our color components. Finally, we create the gradient and paint it into place.

(There are also gradient CIFilters, as I demonstrated earlier in this chapter; for yet another way to create a simple gradient, see the discussion of CAGradientLayer in the next chapter.)

Colors and Patterns

A color is a CGColor. CGColor is not difficult to work with, and can be converted to and from a UIColor through UIColor's `init(CGColor:)` and `CGColor` methods.

A pattern is also a kind of color. You can create a pattern color and stroke or fill with it. The simplest way is to draw a minimal tile of the pattern into a UIImage and create the color by calling UIColor's `init(patternImage:)`. To illustrate, I'll create a pattern of horizontal stripes and use it to paint the point of the arrow instead of a solid red color (Figure 2-18):

```
// CGContextSetFillColorWithColor(con, UIColor.redColor().CGColor)
// not any more, we're going to paint with a pattern instead of red!
// create the pattern image tile
UIGraphicsBeginImageContextWithOptions(CGSizeMake(4,4), false, 0)
let imcon = UIGraphicsGetCurrentContext()!
CGContextSetFillColorWithColor(imcon, UIColor.redColor().CGColor)
CGContextFillRect(imcon, CGRectMake(0,0,4,4))
CGContextSetFillColorWithColor(imcon, UIColor.blueColor().CGColor)
CGContextFillRect(imcon, CGRectMake(0,0,4,2))
let stripes = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
// paint the point of the arrow with it
let stripesPattern = UIColor(patternImage:stripes)
stripesPattern.setFill()
let p = UIBezierPath()
p.moveToPoint(CGPointMake(80,25))
p.addLineToPoint(CGPointMake(100,0))
p.addLineToPoint(CGPointMake(120,25))
p.fill()
```

The Core Graphics equivalent, CGPattern, is considerably more powerful, but also much more elaborate:

```
let sp2 = CGColorSpaceCreatePattern(nil)
CGContextSetFillColorSpace(con, sp2)
let drawStripes : CGPatternDrawPatternCallback = {
    _, con in
    CGContextSetFillColorWithColor(con!, UIColor.redColor().CGColor)
    CGContextFillRect(con!, CGRectMake(0,0,4,4))
    CGContextSetFillColorWithColor(con!, UIColor.blueColor().CGColor)
    CGContextFillRect(con!, CGRectMake(0,0,4,2))
}
var callbacks = CGPatternCallbacks(
    version: 0, drawPattern: drawStripes, releaseInfo: nil)
let patt = CGPatternCreate(nil, CGRectMake(0,0,4,4),
    CGAffineTransformIdentity, 4, 4,
    .ConstantSpacingMinimalDistortion,
    true, &callbacks)
var alph : CGFloat = 1.0
CGContextSetFillPattern(con, patt, &alph)
CGContextMoveToPoint(con, 80, 25)
CGContextAddLineToPoint(con, 100, 0)
CGContextAddLineToPoint(con, 120, 25)
CGContextFillPath(con)
```

To understand that code, it helps to read it backwards. Everything revolves around the call to `CGPatternCreate`. A pattern is a drawing in a rectangular "cell"; we have to state both the size of the cell (the second argument) and the spacing between origin points of cells (the fourth and fifth arguments). In this case, the cell is 4×4, and every cell exactly touches its neighbors both horizontally and vertically. We have to supply a transform to be applied to the cell (the third argument); in this case, we're not doing anything with this transform, so we supply the identity transform. We supply a tiling rule (the sixth argument). We have to state whether this is a color pattern or a stencil pattern; it's a color pattern, so the seventh argument is `true`. And we have to supply a pointer to a callback function that actually draws the pattern into its cell (the eighth argument).

Except that that's *not* what we have to supply as the eighth argument. What we actually have to supply here is a pointer to a CGPatternCallbacks struct. This struct consists of the number `0` and pointers to *two* functions, one called to draw the pattern into its cell, the other called when the pattern is released. We're not specifying the second function, however; it is for memory management, and we don't need it in this simple example.

As you can see, the actual pattern-drawing code (`drawStripes`) is very simple. The only tricky issue is that the call to `CGPatternCreate` must be in agreement with the pattern-drawing function as to the size of a cell, or the pattern won't come out the way you expect. We know in this case that the cell is 4×4. So we fill it with red, and then fill its lower half with blue. When these cells are tiled touching each other horizontally and vertically, we get the stripes that you see in Figure 2-18.

Having generated the CGPattern with `CGPatternCreate`, we call `CGContextSetFillPattern`; instead of setting a fill color, we're setting a fill pattern, to be used the next time we fill a path (in this case, the triangular arrowhead). The third parameter to `CGContextSetFillPattern` is a pointer to a CGFloat, so we have to set up the CGFloat itself beforehand. The second parameter is the CGPattern.

The only thing left to explain is the first two lines of that code. It turns out that before you can call `CGContextSetFillPattern` with a colored pattern, you have to set the context's fill color space to a pattern color space. If you neglect to do this, you'll get an error when you call `CGContextSetFillPattern`. This means that the code as presented has left the graphics context in an undesirable state, with its fill color space set to a pattern color space. This would cause trouble if we were later to try to set the fill color to a normal color. The solution, as usual, is to wrap the code in calls to `CGContextSaveGState` and `CGContextRestoreGState`.

You may have observed in Figure 2-18 that the stripes do not fit neatly inside the triangle of the arrowhead: the bottommost stripe is something like half a blue stripe. This is because a pattern is positioned not with respect to the shape you are filling (or stroking), but with respect to the graphics context as a whole. We could shift the pattern position by calling `CGContextSetPatternPhase` before drawing.

Graphics Context Transforms

Just as a UIView can have a transform, so can a graphics context. However, applying a transform to a graphics context has no effect on the drawing that's already in it; it affects only the drawing that takes place after it is applied, altering the way the coordinates you provide are mapped onto the graphics context's area. A graphics context's transform is called its CTM, for "current transform matrix."

It is quite usual to take full advantage of a graphics context's CTM to save yourself from performing even simple calculations. You can multiply the current transform by any CGAffineTransform using `CGContextConcatCTM`; there are also convenience functions for applying a translate, scale, or rotate transform to the current transform.

The base transform for a graphics context is already set for you when you obtain the context; this is how the system is able to map context drawing coordinates onto screen coordinates. Whatever transforms you apply are applied to the current transform, so the base transform remains in effect and drawing continues to work. You can return to the base transform after applying your own transforms by wrapping your code in calls to `CGContextSaveGState` and `CGContextRestoreGState`.

For example, we have hitherto been drawing our upward-pointing arrow with code that knows how to place that arrow at only one location: the top left of its rectangle is hard-coded at `(80,0)`. This is silly. It makes the code hard to understand, as well as inflexible

and difficult to reuse. Surely the sensible thing would be to draw the arrow at `(0,0)`, by subtracting 80 from all the x-values in our existing code. Now it is easy to draw the arrow at *any* position, simply by applying a translate transform beforehand, mapping `(0,0)` to the desired top-left corner of the arrow. So, to draw it at `(80,0)`, we would say:

```
CGContextTranslateCTM(con, 80, 0)
// now draw the arrow at (0,0)
```

A rotate transform is particularly useful, allowing you to draw in a rotated orientation without any nasty trigonometry. However, it's a bit tricky because the point around which the rotation takes place is the origin. This is rarely what you want, so you have to apply a translate transform first, to map the origin to the point around which you really want to rotate. But then, after rotating, in order to figure out where to draw you will probably have to reverse your translate transform.

To illustrate, here's code to draw our arrow repeatedly at several angles, pivoting around the end of its tail (Figure 2-19). Since the arrow will be drawn multiple times, I'll start by encapsulating the drawing of the arrow as a UIImage. This is not merely to reduce repetition and make drawing more efficient; it's also because we want the entire arrow to pivot, including the pattern stripes, and this is the simplest way to achieve that:

```
func arrowImage () -> UIImage {
    UIGraphicsBeginImageContextWithOptions(CGSizeMake(40,100), false, 0.0)
    // obtain the current graphics context
    let con = UIGraphicsGetCurrentContext()!
    // draw the arrow into the image context
    // draw it at (0,0)! adjust all x-values by subtracting 80
    // ... actual code omitted ...
    let im = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()
    return im
}
```

We produce the arrow image once and store it somewhere — I'll use a property accessed as `self.arrow`. In our `drawRect:` implementation, we draw the arrow image multiple times:

```
override func drawRect(rect: CGRect) {
    let con = UIGraphicsGetCurrentContext()!
    self.arrow.drawAtPoint(CGPointMake(0,0))
    for _ in 0..<3 {
        CGContextTranslateCTM(con, 20, 100)
        CGContextRotateCTM(con, 30 * CGFloat(M_PI)/180.0)
        CGContextTranslateCTM(con, -20, -100)
        self.arrow.drawAtPoint(CGPointMake(0,0))
    }
}
```

A transform is also one more solution for the "flip" problem we encountered earlier with `CGContextDrawImage`. Instead of reversing the drawing, we can reverse the context

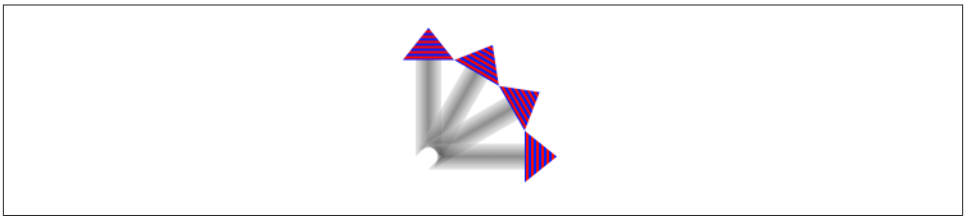

Figure 2-19. Drawing rotated with a CTM

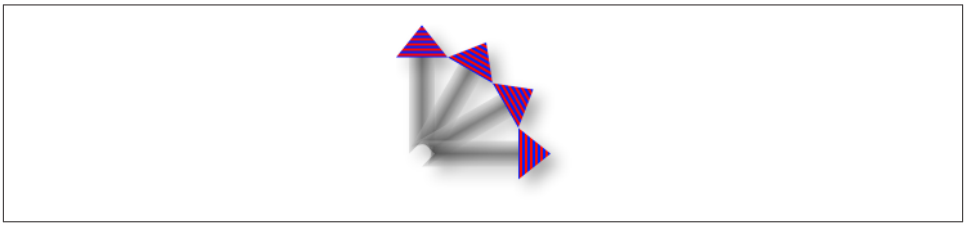

Figure 2-20. Drawing with a shadow

into which we draw it. Essentially, we apply a "flip" transform to the context's coordinate system. You move the context's top downward, and then reverse the direction of the y-coordinate by applying a scale transform whose y-multiplier is `-1`:

```
CGContextTranslateCTM(con, 0, theHeight)
CGContextScaleCTM(con, 1.0, -1.0)
```

How far down you move the context's top (`theHeight`) depends on how you intend to draw the image.

Shadows

To add a shadow to a drawing, give the context a shadow value before drawing. The shadow position is expressed as a CGSize, where the positive direction for both values indicates down and to the right. The blur value is an open-ended positive number; Apple doesn't explain how the scale works, but experimentation shows that 12 is nice and blurry, 99 is so blurry as to be shapeless, and higher values become problematic.

Figure 2-20 shows the result of the same code that generated Figure 2-19, except that before we start drawing the arrow repeatedly, we give the context a shadow:

```
let con = UIGraphicsGetCurrentContext()!
CGContextSetShadow(con, CGSizeMake(7, 7), 12)
self.arrow.drawAtPoint(CGPointMake(0,0)) // ... and so on
```

It may not be evident from Figure 2-20, but we are adding a shadow each time we draw. Thus the arrows are able to cast shadows on one another. Suppose, however, that we

want all the arrows to cast a single shadow collectively. The way to achieve this is with a *transparency layer*; this is basically a subcontext that accumulates all drawing and then adds the shadow. Our code for drawing the shadowed arrows now looks like this:

```
let con = UIGraphicsGetCurrentContext()!
CGContextSetShadow(con, CGSizeMake(7, 7), 12)
CGContextBeginTransparencyLayer(con, nil)
self.arrow.drawAtPoint(CGPointMake(0,0))
for _ in 0..<3 {
    CGContextTranslateCTM(con, 20, 100)
    CGContextRotateCTM(con, 30 * CGFloat(M_PI)/180.0)
    CGContextTranslateCTM(con, -20, -100)
    self.arrow.drawAtPoint(CGPointMake(0,0))
}
CGContextEndTransparencyLayer(con)
```

Erasing

The function `CGContextClearRect` erases all existing drawing in a rectangle; combined with clipping, it can erase an area of any shape. The result can "punch a hole" through all existing drawing.

The behavior of `CGContextClearRect` depends on whether the context is transparent or opaque. This is particularly obvious and intuitive when drawing into an image context. If the image context is transparent — the second argument to `UIGraphicsBeginImageContextWithOptions` is `false` — `CGContextClearRect` erases to transparent; otherwise it erases to black.

When drawing directly into a view (as with `drawRect:` or `drawLayer:inContext:`), if the view's background color is `nil` or a color with even a tiny bit of transparency, the result of `CGContextClearRect` will appear to be transparent, punching a hole right through the view including its background color; if the background color is completely opaque, the result of `CGContextClearRect` will be black. This is because the view's background color determines whether the view's graphics context is transparent or opaque; thus, this is essentially the same behavior that I described in the preceding paragraph.

Figure 2-21 illustrates; the blue square on the left has been partly cut away to black, while the blue square on the right has been partly cut away to transparency. Yet these are instances of the same UIView subclass, drawn with exactly the same code! The UIView subclass's `drawRect:` looks like this:

```
let con = UIGraphicsGetCurrentContext()!
CGContextSetFillColorWithColor(con, UIColor.blueColor().CGColor)
CGContextFillRect(con, rect)
CGContextClearRect(con, CGRectMake(0,0,30,30))
```

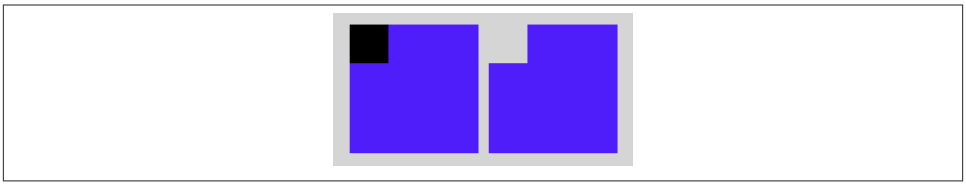

Figure 2-21. The very strange behavior of CGContextClearRect

The difference between the views in Figure 2-21 is that the `backgroundColor` of the first view is solid red with an alpha of `1`, while the `backgroundColor` of the second view is solid red with an alpha of `0.99`. This difference is utterly imperceptible to the eye (not to mention that the red color never appears, as it is covered with a blue fill), but it completely changes the effect of `CGContextClearRect`.

Points and Pixels

A point is a dimensionless location described by an x-coordinate and a y-coordinate. When you draw in a graphics context, you specify the points at which to draw, and this works regardless of the device's resolution, because Core Graphics maps your drawing nicely onto the physical output using the base CTM and anti-aliasing. Therefore, throughout this chapter I've concerned myself with graphics context points, disregarding their relationship to screen pixels.

However, pixels do exist. A pixel is a physical, integral, dimensioned unit of display in the real world. Whole-numbered points effectively lie between pixels, and this can matter if you're fussy, especially on a single-resolution device. For example, if a vertical path with whole-number coordinates is stroked with a line width of 1, half the line falls on each side of the path, and the drawn line on the screen of a single-resolution device will seem to be 2 pixels wide (because the device can't illuminate half a pixel).

You will sometimes encounter advice suggesting that if this effect is objectionable, you should try shifting the line's position by `0.5`, to center it in its pixels. This advice may appear to work, but it makes some simpleminded assumptions. A more sophisticated approach is to obtain the UIView's `contentScaleFactor` property. You can divide by this value to convert from pixels to points. Consider also that the most accurate way to draw a vertical or horizontal line is not to stroke a path but to fill a rectangle. So this UIView subclass code will draw a perfect 1-pixel-wide vertical line on any device (`con` is the current graphics context):

```
CGContextFillRect(con, CGRectMake(100,0,1.0/self.contentScaleFactor,100))
```

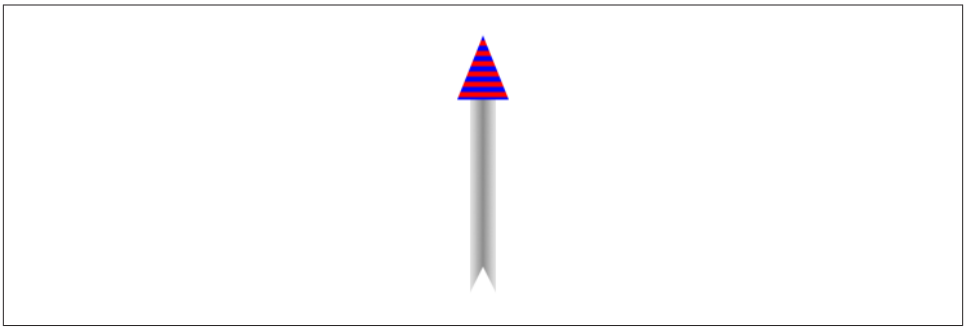

Figure 2-22. Automatic stretching of content

Content Mode

A view that draws something within itself, as opposed to merely having a background color and subviews (as in the previous chapter), has *content*. This means that its `contentMode` property becomes important whenever the view is resized. As I mentioned earlier, the drawing system will avoid asking a view to redraw itself from scratch if possible; instead, it will use the cached result of the previous drawing operation (the bitmap backing store). So, if the view is resized, the system may simply stretch or shrink or reposition the cached drawing, if your `contentMode` setting instructs it to do so.

It's a little tricky to illustrate this point when the view's content is coming from `drawRect:`, because I have to arrange for the view to obtain its content (from `drawRect:`) and then cause it to be resized without also causing it to be redrawn (that is, without `drawRect:` being called *again*). Here's how I'll do that. As the app starts up, I'll create an instance of a UIView subclass, MyView, that knows how to draw our arrow. Then I'll use delayed performance to resize the instance after the window has shown and the interface has been initially displayed (for the `delay` utility function, see Appendix B):

```
delay(0.1) {
    mv.bounds.size.height *= 2 // mv is the MyView instance
}
```

We double the height of the view without causing `drawRect:` to be called. The result is that the view's drawing appears at double its correct height. For example, if our view's `drawRect:` code is the same as the code that generated Figure 2-17, we get Figure 2-22.

Sooner or later, however, `drawRect:` will be called, and the drawing will be refreshed in accordance with our code. Our code doesn't say to draw the arrow at a height that is relative to the height of the view's bounds; it draws the arrow at a fixed height. Thus, the arrow will snap back to its original size.

A view's `contentMode` property should therefore usually be in agreement with how the view draws itself. Our `drawRect:` code dictates the size and position of the arrow relative to the view's bounds origin, its top left; so we could set its `contentMode` to `.TopLeft`. Alternatively, we could set it to `.Redraw`; this will cause automatic scaling of the cached content to be turned off — instead, when the view is resized, its `setNeedsDisplay` method will be called, ultimately triggering `drawRect:` to redraw the content.