CHAPTER 3
Layers

The tale told in Chapters 1 and 2 of how a UIView works and how it draws itself is only half the story. A UIView has a partner called its *layer*, a CALayer. A UIView does not actually draw itself onto the screen; it draws itself into its layer, and it is the layer that is portrayed on the screen. As I've already mentioned, a view is not redrawn frequently; instead, its drawing is cached, and the cached version of the drawing (the bitmap backing store) is used where possible. The cached version is, in fact, the layer. What I spoke of in Chapter 2 as the view's graphics context is actually the layer's graphics context.

This might seem to be a mere implementation detail, but layers are important and interesting. To understand layers is to understand views more deeply; layers extend the power of views. In particular:

Layers have properties that affect drawing.
: Layers have drawing-related properties beyond those of a UIView. Because a layer is the recipient and presenter of a view's drawing, you can modify how a view is drawn on the screen by accessing the layer's properties. In other words, by reaching down to the level of its layer, you can make a view do things you can't do through UIView methods alone.

Layers can be combined within a single view.
: A UIView's partner layer can contain additional layers. Since the purpose of layers is to draw, portraying visible material on the screen, this allows a UIView's drawing to be composited of multiple distinct pieces. This can make drawing easier, with the constituents of a drawing being treated as objects.

Layers are the basis of animation.
: Animation allows you to add clarity, emphasis, and just plain coolness to your interface. Layers are made to be animated; the "CA" in "CALayer" stands for "Core Animation." Animation is the subject of Chapter 4.

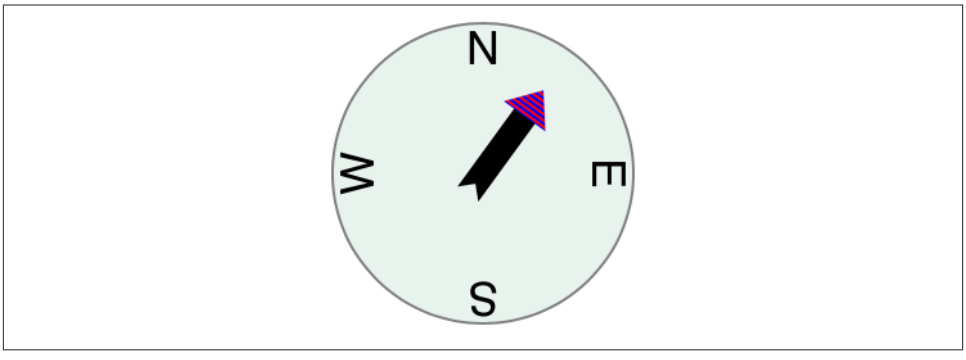

Figure 3-1. A compass, composed of layers

For example, suppose we want to add a compass indicator to our app's interface. Figure 3-1 portrays a simple version of such a compass. It takes advantage of the arrow that we figured out how to draw in Chapter 2; the arrow is drawn into a layer of its own. The other parts of the compass are layers too: the circle is a layer, and each of the cardinal point letters is a layer. The drawing is thus easy to composite in code (and later in this chapter, that's exactly what we'll do); even more intriguing, the pieces can be repositioned and animated separately, so it's easy to rotate the arrow without moving the circle (and in Chapter 4, that's exactly what we'll do).

The documentation discusses layers chiefly in connection with animation (in particular, in the *Core Animation Programming Guide*). This categorization gives the impression that layers are of interest only if you intend to animate. That's misleading. Layers are the basis of animation, but they are also the basis of view drawing, and are useful and important even if you don't use them for animation.

View and Layer

A UIView instance has an accompanying CALayer instance, accessible as the view's `layer` property. This layer has a special status: it is partnered with this view to embody all of the view's drawing. The layer has no corresponding `view` property, but the view is the layer's `delegate`. The documentation sometimes speaks of this layer as the view's *underlying layer*.

By default, when a UIView is instantiated, its layer is an instance of CALayer. If you subclass UIView and you want your subclass's underlying layer to be an instance of a CALayer subclass (built-in or your own), implement the UIView subclass's `layerClass` class method to return that CALayer subclass.

That, for example, is how the compass in Figure 3-1 is created. We have a UIView subclass, CompassView, and a CALayer subclass, CompassLayer. Here is CompassView's implementation:

```
class CompassView : UIView {
    override class func layerClass() -> AnyClass {
        return CompassLayer.self
    }
}
```

Thus, when CompassView is instantiated, its underlying layer is a CompassLayer. In this example, there is no drawing in CompassView; its job — in this case, its *only* job — is to give CompassLayer a place in the visible interface, because a layer cannot appear without a view.

Because every view has an underlying layer, there is a tight integration between the two. The layer portrays all the view's drawing; if the view draws, it does so by contributing to the layer's drawing. The view is the layer's delegate. And the view's properties are often merely a convenience for accessing the layer's properties. For example, when you set the view's `backgroundColor`, you are really setting the layer's `backgroundColor`, and if you set the layer's `backgroundColor` directly, the view's `backgroundColor` is set to match. Similarly, the view's `frame` is really the layer's `frame` and *vice versa*.

A CALayer's `delegate` property is settable, and can be an instance of any NSObject-based class (CALayerDelegate is an informal protocol, a category injected into NSObject). But a UIView and its underlying layer have a special relationship. A UIView *must* be the delegate of its underlying layer; moreover, it must *not* be the delegate of any *other* layer. *Don't do anything to mess this up.* If you do, drawing will stop working correctly.

The view draws into its layer, and the layer caches that drawing; the layer can then be manipulated, changing the view's appearance, without necessarily asking the view to redraw itself. This is a source of great efficiency in the drawing system. It also explains such phenomena as the content stretching that we encountered in the last section of Chapter 2: when the view's bounds size changes, the drawing system, by default, simply stretches or repositions the cached layer image, until such time as the view is told to draw freshly (`drawRect:`), replacing the layer's content.

Layers and Sublayers

A layer can have sublayers, and a layer has at most one superlayer. Thus there is a tree of layers. This is similar and parallel to the tree of views (Chapter 1). In fact, so tight is the integration between a view and its underlying layer that these hierarchies are effectively the same hierarchy. Given a view and its underlying layer, that layer's superlayer

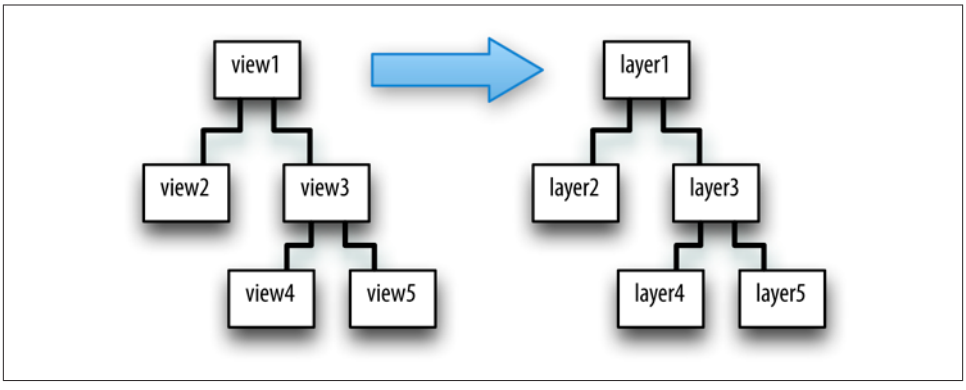

Figure 3-2. A hierarchy of views and the hierarchy of layers underlying it

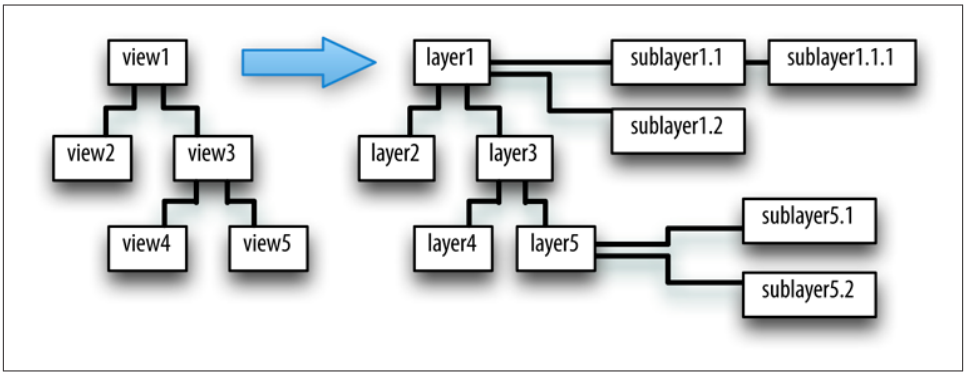

Figure 3-3. Layers that have sublayers of their own

is the view's superview's underlying layer, and that layer has as sublayers all the underlying layers of all the view's subviews. Indeed, because the layers are how the views actually get drawn, one might say that the view hierarchy really *is* a layer hierarchy (Figure 3-2).

At the same time, the layer hierarchy can go beyond the view hierarchy. A view has exactly one underlying layer, but a layer can have sublayers that are not the underlying layers of any view. So the hierarchy of layers that underlie views exactly matches the hierarchy of views, but the total layer tree may be a superset of that hierarchy. In Figure 3-3, we see the same view-and-layer hierarchy as in Figure 3-2, but two of the layers have additional sublayers that are theirs alone (that is, sublayers that are not any view's underlying layers).

From a visual standpoint, there may be nothing to distinguish a hierarchy of views from a hierarchy of layers. For example, in Chapter 1 we drew three overlapping rectangles

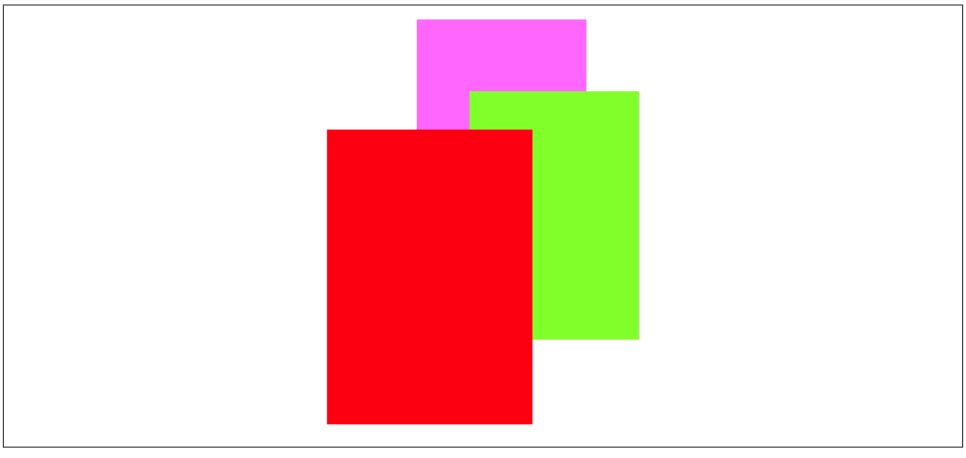

Figure 3-4. Overlapping layers

using a hierarchy of views (Figure 1-1). This code gives exactly the same visible display by manipulating layers (Figure 3-4):

```
let lay1 = CALayer()
lay1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1).CGColor
lay1.frame = CGRectMake(113, 111, 132, 194)
mainview.layer.addSublayer(lay1)
let lay2 = CALayer()
lay2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1).CGColor
lay2.frame = CGRectMake(41, 56, 132, 194)
lay1.addSublayer(lay2)
let lay3 = CALayer()
lay3.backgroundColor = UIColor(red: 1, green: 0, blue: 0, alpha: 1).CGColor
lay3.frame = CGRectMake(43, 197, 160, 230)
mainview.layer.addSublayer(lay3)
```

A view's subview's underlying layer is a sublayer of that view's underlaying layer, just like any other sublayers of that view's underlying layer. Therefore, it can be positioned anywhere among them in the drawing order. The fact that a view can be interspersed among the sublayers of its superview's underlying layer is surprising to beginners. For example, let's construct Figure 3-4 again, but between adding `lay2` and `lay3` to the interface, we'll add a subview:

```
// ...
lay1.addSublayer(lay2)
let iv = UIImageView(image:UIImage(named:"smiley"))
mainview.addSubview(iv)
iv.frame.origin = CGPointMake(180,180)
let lay3 = CALayer() // the red rectangle
// ...
```

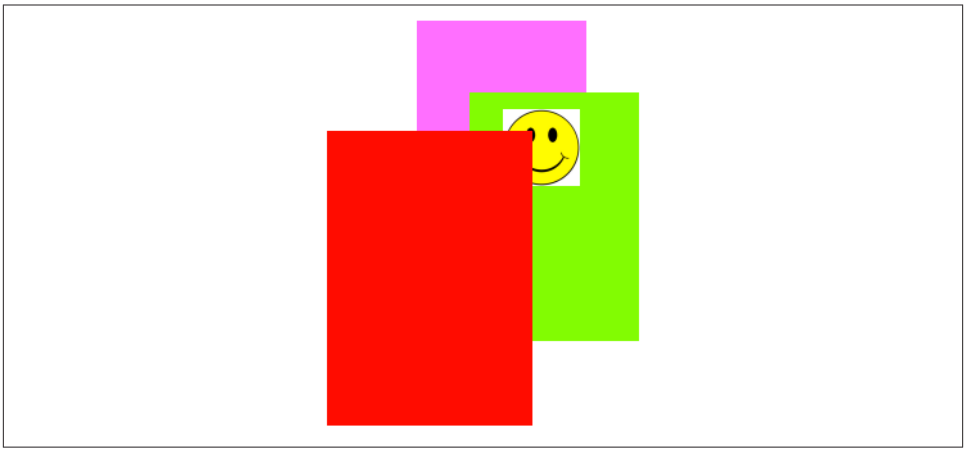

Figure 3-5. Overlapping layers and a view

The result is Figure 3-5. The smiley face was added to the interface before the red (left) rectangle, so it appears behind that rectangle. By reversing the order in which the red rectangle (`lay3`) and the smiley face (`iv`) are added to the interface, the smiley face can be made to appear in front of that rectangle. The smiley face is a *view*, whereas the rectangle is just a *layer*; so they are not siblings as views, since the rectangle is not a view. But the smiley face is both a view and its layer; as layers, the smiley face and the rectangle *are* siblings, since they have the same superlayer, so either one can be made to appear in front of the other.

Whether a layer displays regions of its sublayers that lie outside that layer's own bounds depends upon the value of its `masksToBounds` property. This is parallel to a view's `clipsToBounds` property, and indeed, for a layer that is its view's underlying layer, they are the same thing. In Figures 3-4 and 3-5, the layers all have `clipsToBounds` set to `false` (the default); that's why the right layer is visible beyond the bounds of the middle layer, which is its superlayer.

Like a UIView, a CALayer has a `hidden` property that can be set to take it and its sublayers out of the visible interface without actually removing it from its superlayer.

Manipulating the Layer Hierarchy

Layers come with a full set of methods for reading and manipulating the layer hierarchy, parallel to the methods for reading and manipulating the view hierarchy. A layer has a `superlayer` property and a `sublayers` property, along with these methods:

- `addSublayer:`
- `insertSublayer:atIndex:`

- `insertSublayer:below:`, `insertSublayer:above:`
- `replaceSublayer:with:`
- `removeFromSuperlayer`

Unlike a view's `subviews` property, a layer's `sublayers` property is writable; thus, you can give a layer multiple sublayers in a single move, by assigning to its `sublayers` property. To remove all of a layer's sublayers, set its `sublayers` property to `nil`.

Although a layer's sublayers have an order, reflected in the `sublayers` order and regulated with the methods I've just mentioned, this is not necessarily the same as their back-to-front drawing order. By default, it is, but a layer also has a `zPosition` property, a CGFloat, and this also determines drawing order. The rule is that all sublayers with the same `zPosition` are drawn in the order they are listed among their `sublayers` siblings, but lower `zPosition` siblings are drawn before higher `zPosition` siblings. (The default `zPosition` is `0.0`.)

Sometimes, the `zPosition` property is a more convenient way of dictating drawing order than sibling order is. For example, if layers represent playing cards laid out in a solitaire game, it will likely be a lot easier and more flexible to determine how the cards overlap by setting their `zPosition` than by rearranging their sibling order. Moreover, a subview's layer is itself just a layer, so you can rearrange the drawing order of subviews by setting the `zPosition` of their underlying layers. In our code constructing Figure 3-5, if we assign the image view's underlying layer a `zPosition` of `1`, it is drawn in front of the red (left) rectangle:

```
mainview.addSubview(iv)
iv.layer.zPosition = 1
```

Methods are also provided for converting between the coordinate systems of layers within the same layer hierarchy:

- `convertPoint:fromLayer:`, `convertPoint:toLayer:`
- `convertRect:fromLayer:`, `convertRect:toLayer:`

Positioning a Sublayer

Layer coordinate systems and positioning are similar to those of views. A layer's own internal coordinate system is expressed by its `bounds`, just like a view; its size is its bounds size, and its bounds origin is the internal coordinate at its top left.

However, a sublayer's position within its superlayer is not described by its `center`, like a view; a layer does not have a `center`. Instead, a sublayer's position within its superlayer is defined by a combination of two properties:

`position`
: A point expressed in the superlayer's coordinate system.

`anchorPoint`
: Where the `position` point is located, with respect to the layer's own bounds. It is a CGPoint describing a fraction (or multiple) of the layer's own bounds width and bounds height. Thus, for example, `(0.0,0.0)` is the top left of the layer's bounds, and `(1.0,1.0)` is the bottom right of the layer's bounds.

Here's an analogy; I didn't make it up, but it's pretty apt. Think of the sublayer as pinned to its superlayer; then you have to say both where the pin passes through the sublayer (the `anchorPoint`) and where it passes through the superlayer (the `position`).

If the `anchorPoint` is `(0.5,0.5)` (the default), the `position` property works like a view's `center` property. A view's `center` is thus a special case of a layer's `position`. This is quite typical of the relationship between view properties and layer properties; the view properties are often a simpler — but less powerful — version of the layer properties.

A layer's `position` and `anchorPoint` are orthogonal (independent); changing one does not change the other. Therefore, changing either of them without changing the other changes where the layer is drawn within its superlayer.

For example, in Figure 3-1, the most important point in the circle is its center; all the other objects need to be positioned with respect to it. Therefore they all have the same `position`: the center of the circle. But they differ in their `anchorPoint`. For example, the arrow's `anchorPoint` is `(0.5,0.8)`, the middle of the shaft, near the tail. On the other hand, the `anchorPoint` of a cardinal point letter is `(0.5,3.0)`, well outside the letter's bounds, so as to place the letter near the edge of the circle.

A layer's `frame` is a purely derived property. When you get the `frame`, it is calculated from the bounds size along with the `position` and `anchorPoint`. When you set the `frame`, you set the bounds size and `position`. In general, you should regard the `frame` as a convenient façade and no more. Nevertheless, it is convenient! For example, to position a sublayer so that it exactly overlaps its superlayer, you can just set the sublayer's `frame` to the superlayer's `bounds`.

A layer created in code (as opposed to a view's underlying layer) has a `frame` and `bounds` of `(0.0,0.0,0.0,0.0)` and will not be visible on the screen even when you add it to a superlayer that is on the screen. Be sure to give your layer a nonzero width and height if you want to be able to see it. Creating a layer and adding it to a superlayer and then wondering why it isn't appearing in the interface is a common beginner error.

CAScrollLayer

If you're going to be moving a layer's bounds origin as a way of repositioning its sublayers *en masse*, you might like to make the layer a CAScrollLayer, a CALayer subclass that provides convenience methods for this sort of thing. (Despite the name, a CAScrollLayer provides no scrolling interface; the user can't scroll it by dragging, for example.) By default, a CAScrollLayer's `masksToBounds` property is `true`; thus, the CAScrollLayer acts like a window through which you see can only what is within its bounds. (You can set its `masksToBounds` to `false`, but this would be an odd thing to do, as it somewhat defeats the purpose.)

To move the CAScrollLayer's bounds, you can talk either to it or to a sublayer (at any depth):

Talking to the CAScrollLayer

`scrollToPoint:`
: Changes the CAScrollLayer's bounds origin to that point.

`scrollToRect:`
: Changes the CAScrollLayer's bounds origin minimally so that the given portion of the bounds rect is visible.

Talking to a sublayer

`scrollPoint:`
: Changes the CAScrollLayer's bounds origin so that the given point *of the sublayer* is at the top left of the CAScrollLayer.

`scrollRectToVisible:`
: Changes the CAScrollLayer's bounds origin so that the given rect *of the sublayer's bounds* is within the CAScrollLayer's bounds area. You can also ask the sublayer for its `visibleRect`, the part of this sublayer now within the CAScrollLayer's bounds.

Layout of Sublayers

The view hierarchy is actually a layer hierarchy (Figure 3-2). The positioning of a view within its superview is actually the positioning of its layer within its superlayer (the superview's layer). A view can be repositioned and resized automatically in accordance with its `autoresizingMask` or through autolayout based on its constraints. Thus, there is automatic layout for layers *if they are the underlying layers of views*. Otherwise, there is *no* automatic layout for layers in iOS. The only option for layout of sublayers that are not the underlying layers of views is manual layout that you perform in code.

When a layer needs layout, either because its bounds have changed or because you called `setNeedsLayout`, you can respond in either of two ways:

- The layer's `layoutSublayers` method is called; to respond, override `layoutSublayers` in your CALayer subclass.
- Alternatively, implement `layoutSublayersOfLayer:` in the layer's delegate. (Remember, if the layer is a view's underlying layer, the view is its delegate.)

To do effective manual layout of sublayers, you'll probably need a way to identify or refer to the sublayers. There is no layer equivalent of `viewWithTag:`, so such identification and reference is entirely up to you. Key–value coding can be helpful here; layers implement key–value coding in a special way, discussed at the end of this chapter.

For a view's underlying layer, `layoutSublayers` or `layoutSublayersOfLayer:` is called after the view's `layoutSubviews`. Under autolayout, you must call `super` or else autolayout will break. Moreover, these methods may be called more than once during the course of autolayout; if you're looking for an automatically generated signal that it's time to do manual layout of sublayers, a view layout event might be a better choice (see "View Events Related to Layout" on page 62).

Drawing in a Layer

The simplest way to make something appear in a layer is through its `contents` property. This is parallel to the `image` in a UIImageView (Chapter 2). It is expected to be a CGImage (or `nil`, signifying no image). So, for example, here's how we might modify the code that generated Figure 3-5 in such a way as to generate the smiley face as a layer rather than a view:

```
let lay4 = CALayer()
let im = UIImage(named:"smiley")!
lay4.frame = CGRect(origin:CGPointMake(180,180), size:im.size)
lay4.contents = im.CGImage
mainview.layer.addSublayer(lay4)
```

Setting a layer's `contents` to a UIImage, rather than a CGImage, will *fail silently* — the image doesn't appear, but there is no error either. This is absolutely maddening, and I wish I had a nickel for every time I've done it and then wasted hours figuring out why my layer isn't appearing.

There are also four methods that can be implemented to provide or draw a layer's content on demand, similar to a UIView's `drawRect:`. A layer is very conservative about calling these methods (and you must not call any of them directly). When a layer *does* call these methods, I will say that the layer *redisplays itself*. Here is how a layer can be caused to redisplay itself:

- If the layer's `needsDisplayOnBoundsChange` property is `false` (the default), then the only way to cause the layer to redisplay itself is by calling `setNeedsDisplay` (or `setNeedsDisplayInRect:`). Even this might not cause the layer to redisplay itself right away; if that's crucial, then you will also call `displayIfNeeded`.
- If the layer's `needsDisplayOnBoundsChange` property is `true`, then the layer will also redisplay itself when the layer's bounds change (rather like a view's `.Redraw` content mode).

Here are the four methods that can be called when a layer redisplays itself; pick one to implement (don't try to combine them, you'll just confuse things):

`display` *in a subclass*
: Your CALayer subclass can override `display`. There's no graphics context at this point, so `display` is pretty much limited to setting the `contents` image.

`displayLayer:` *in the delegate*
: You can set the CALayer's `delegate` property and implement `displayLayer:` in the delegate. As with `display`, there's no graphics context, so you'll just be setting the `contents` image.

`drawInContext:` *in a subclass*
: Your CALayer subclass can override `drawInContext:`. The parameter is a graphics context into which you can draw directly; it is *not* automatically made the current context.

`drawLayer:inContext:` *in the delegate*
: You can set the CALayer's `delegate` property and implement `drawLayer:inContext:`. The second parameter is a graphics context into which you can draw directly; it is *not* automatically made the current context.

Assigning a layer a `contents` image and drawing directly into the layer are, in effect, mutually exclusive. So:

- If a layer's `contents` is assigned an image, this image is shown immediately and replaces whatever drawing may have been displayed in the layer.
- If a layer redisplays itself and `drawInContext:` or `drawLayer:inContext:` draws into the layer, the drawing replaces whatever image may have been displayed in the layer.
- If a layer redisplays itself and none of the four methods provides any content, the layer will be empty.

If a layer is a view's underlying layer, you usually won't use any of the four methods to draw into the layer: you'll use the view's `drawRect:`. However, you *can* use these methods

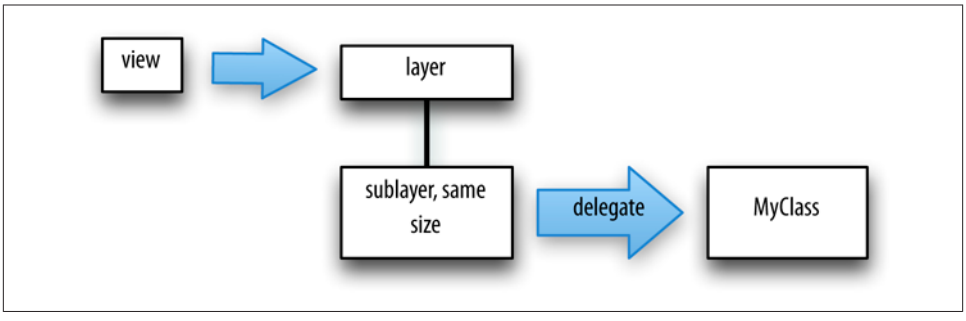

Figure 3-6. A view and a layer delegate that draws into it

if you really want to. In that case, you will probably want to implement `drawRect:` anyway, leaving that implementation empty. The reason is that this causes the layer to redisplay itself at appropriate moments. When a view is sent `setNeedsDisplay` — including when the view first appears — the view's underlying layer is also sent `setNeedsDisplay`, *unless the view has no `drawRect:` implementation* (because in that case, it is assumed that the view never needs redrawing). So, if you're drawing a view entirely by drawing to its underlying layer directly, and if you want the underlying layer to be redisplayed automatically when the view is told to redraw itself, you should implement `drawRect:` to do nothing. (This technique has no effect on sublayers of the underlying layer.)

Thus, these are legitimate (but unusual) techniques for drawing into a view:

- The view subclass implements an empty `drawRect:`, along with either `displayLayer:` or `drawLayer:inContext:`.
- The view subclass implements an empty `drawRect:` plus `layerClass`, to give the view a custom layer subclass — and the custom layer subclass implements either `display` or `drawInContext:`.

Remember, you must not set the `delegate` property of a view's underlying layer! The view is its delegate and must remain its delegate. A useful architecture for drawing into a layer through a delegate of your choosing is to treat a view as a *layer-hosting* view: the view and its underlying layer do nothing except to serve as a host to a sublayer of the view's underlying layer, which is where the drawing occurs (Figure 3-6).

A layer has a scale, its `contentsScale`, which maps point distances in the layer's graphics context to pixel distances on the device. A layer that's managed by Cocoa, if it has contents, will adjust its `contentsScale` automatically as needed; for example, if a view implements `drawRect:`, then on a device with a double-resolution screen its underlying layer is assigned a `contentsScale` of 2. A layer that you are creating and managing

yourself, however, has no such automatic behavior; it's up to you, if you plan to draw into the layer, to set its `contentsScale` appropriately. Content drawn into a layer with a `contentsScale` of 1 may appear pixellated or fuzzy on a high-resolution screen. And when you're starting with a UIImage and assigning its CGImage as a layer's `contents`, if there's a mismatch between the UIImage's `scale` and the layer's `contentsScale`, then the image may be displayed at the wrong size.

Three layer properties strongly affect what the layer displays, in ways that can be baffling to beginners:

`backgroundColor`
: Equivalent to a view's `backgroundColor` (and if this layer is a view's underlying layer, it *is* the view's `backgroundColor`). Changing the `backgroundColor` takes effect immediately. Think of the `backgroundColor` as separate from the layer's own drawing, and as painted *behind* the layer's own drawing.

`opacity`
: Affects the overall apparent transparency of the layer. It is equivalent to a view's `alpha` (and if this layer is a view's underlying layer, it *is* the view's `alpha`). It affects the apparent transparency of the layer's sublayers as well. It affects the apparent transparency of the background color and the apparent transparency of the layer's content separately (just as with a view's `alpha`). Changing the `opacity` property takes effect immediately.

`opaque`
: Determines whether the layer's graphics context is opaque. An opaque graphics context is black; you can draw on top of that blackness, but the blackness is still there. A non-opaque graphics context is clear; where no drawing is, it is completely transparent. Changing the `opaque` property has no effect until the layer redisplays itself. A view's underlying layer's `opaque` property is independent of the view's `opaque` property; they are unrelated and do entirely different things.

If a layer is the underlying layer of a view that implements `drawRect:`, then setting the view's `backgroundColor` changes the layer's `opaque` — setting it to `true` if the new background color is opaque (alpha component of `1`), to `false` otherwise. This is the reason behind the strange behavior of `CGContextClearRect` described in Chapter 2.

Also, when drawing directly into a *layer*, the behavior of `GCContextClearRect` differs from what was described in Chapter 2 for drawing into a *view*: instead of punching a hole through the background color, it effectively paints with the layer's background color. (This can have curious side effects.)

I regard all this as deeply weird.

Content Resizing and Positioning

A layer's content is stored (cached) as a bitmap which is then treated like an image and drawn in relation to the layer's bounds in accordance with various layer properties:

- If the content came from setting the layer's `contents` property to an image, the cached content is that image; its size is the point size of the CGImage we started with.
- If the content came from drawing directly into the layer's graphics context (`drawInContext:`, `drawLayer:inContext:`), the cached content is the layer's entire graphics context; its size is the point size of the layer itself at the time the drawing was performed.

The layer properties in question cause the cached content to be resized, repositioned, cropped, and so on, as it is displayed. The properties are:

`contentsGravity`
: This property, a string, is parallel to a UIView's `contentMode` property, and describes how the content should be positioned or stretched in relation to the bounds. For example, `kCAGravityCenter` means the content is centered in the bounds without resizing; `kCAGravityResize` (the default) means the content is sized to fit the bounds, even if this means distorting its aspect; and so forth.

For historical reasons, the terms `Bottom` and `Top` in the names of the `contentsGravity` settings have the opposite of their expected meanings.

`contentsRect`
: A CGRect expressing the proportion of the content that is to be displayed. The default is `(0.0,0.0,1.0,1.0)`, meaning the entire content is displayed. The specified part of the content is sized and positioned in relation to the bounds in accordance with the `contentsGravity`. Thus, for example, by setting the `contentsRect`, you can scale up part of the content to fill the bounds, or slide part of a larger image into view without redrawing or changing the `contents` image.

  You can also use the `contentsRect` to scale down the content, by specifying a larger `contentsRect` such as `(-0.5,-0.5,1.5,1.5)`; but any content pixels that touch the edge of the `contentsRect` will then be extended outward to the edge of the layer (to prevent this, make sure that the outermost pixels of the content are all empty).

`contentsCenter`
: A CGRect, structured like `contentsRect`, expressing the central region of nine rectangular regions of the `contentsRect` that are variously allowed to stretch if the

`contentsGravity` calls for stretching. The central region (the actual value of the `contentsCenter`) stretches in both directions. Of the other eight regions (inferred from the value you provide), the four corner regions don't stretch, and the four side regions stretch in one direction. (This should remind you of how a resizable image stretches! See Chapter 2.)

If a layer's content comes from drawing directly into its graphics context, then the layer's `contentsGravity`, of itself, has no effect, because the size of the graphics context, by definition, fits the size of the layer exactly; there is nothing to stretch or reposition. But the `contentsGravity` *will* have an effect on such a layer if its `contentsRect` is not `(0.0,0.0,1.0,1.0)`, because now we're specifying a rectangle of some *other* size; the `contentsGravity` describes how to fit that rectangle into the layer.

Again, if a layer's content comes from drawing directly into its graphics context, then when the layer is resized, if the layer is asked to display itself again, the drawing is performed again, and once more the layer's content fits the size of the layer exactly. But if the layer's bounds are resized when `needsDisplayOnBoundsChange` is `false`, then the layer does *not* redisplay itself, so its cached content no longer fits the layer, and the `contentsGravity` matters.

By a judicious combination of settings, you can get the layer to perform some clever drawing for you that might be difficult to perform directly. For example, Figure 3-7 shows the result of the following settings:

```
arrow.needsDisplayOnBoundsChange = false
arrow.contentsCenter = CGRectMake(0.0, 0.4, 1.0, 0.6)
arrow.contentsGravity = kCAGravityResizeAspect
arrow.bounds.insetInPlace(dx: -20, dy: -20)
```

Because `needsDisplayOnBoundsChange` is `false`, the content is not redisplayed when the arrow's bounds are increased; instead, the cached content is used. The `contentsGravity` setting tells us to resize proportionally; therefore, the arrow is both longer and wider than in Figure 3-1, but not in such a way as to distort its proportions. However, notice that although the triangular arrowhead is wider, it is not longer; the increase in length is due entirely to the stretching of the arrow's shaft. That's because the `contentsCenter` region is within the shaft.

A layer's `masksToBounds` property has the same effect on its content that it has on its sublayers. If it is `false`, the whole content is displayed, even if that content (after taking account of the `contentsGravity` and `contentsRect`) is larger then the layer. If it is `true`, only the part of the content within the layer's bounds will be displayed.

The value of a layer's bounds origin does not affect where its content is drawn. It affects only where its sublayers are drawn.

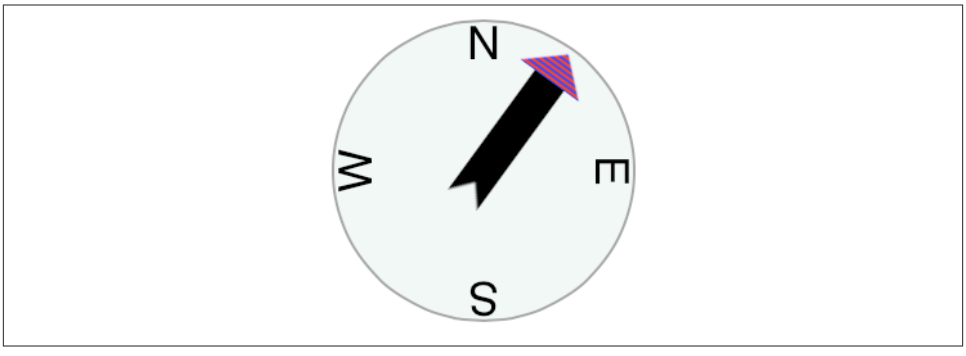

Figure 3-7. One way of resizing the compass arrow

Layers that Draw Themselves

A few built-in CALayer subclasses provide some basic but extremely helpful self-drawing ability:

CATextLayer
: A CATextLayer has a `string` property, which can be an NSString or NSAttributedString, along with other text formatting properties, somewhat like a simplified UILabel; it draws its `string`. The default text color, the `foregroundColor` property, is white, which is unlikely to be what you want. The text is different from the `contents` and is mutually exclusive with it: either the contents image or the text will be drawn, but not both, so in general you should not give a CATextLayer any contents image. In Figures 3-1 and 3-7, the cardinal point letters are CATextLayer instances.

CAShapeLayer
: A CAShapeLayer has a `path` property, which is a CGPath. It fills or strokes this path, or both, depending on its `fillColor` and `strokeColor` values, and displays the result; the default is a `fillColor` of black and no `strokeColor`. It has properties for line thickness, dash style, end-cap style, and join style, similar to a graphics context; it also has the remarkable ability to draw only part of its path (`strokeStart` and `strokeEnd`), making it very easy, for example, to draw an arc of an ellipse. A CAShapeLayer may also have `contents`; the shape is displayed on top of the contents image, but there is no property permitting you to specify a compositing mode. In Figures 3-1 and 3-7, the background circle is a CAShapeLayer instance, stroked with gray and filled with a lighter, slightly transparent gray.

CAGradientLayer
: A CAGradientLayer covers its background with a simple linear gradient; thus, it's an easy way to draw a gradient in your interface (and if you need something more

Figure 3-8. A gradient drawn behind the compass

elaborate you can always draw with Core Graphics instead). The gradient is defined much as in the Core Graphics gradient example in Chapter 2, an array of locations and an array of corresponding colors, along with a start and end point. To clip the gradient's shape, you can add a mask to the CAGradientLayer (masks are discussed later in this chapter). A CAGradientLayer's `contents` are not displayed.

Figure 3-8 shows our compass drawn with an extra CAGradientLayer behind it.

Transforms

The way a layer is drawn on the screen can be modified though a transform. This is not surprising, because a view can have a transform (see Chapter 1), and a view is drawn on the screen by its layer. But a layer's transform is more powerful than a view's transform; you can use it to accomplish things that you can't accomplish with a view's transform alone.

In the simplest case, when a transform is two-dimensional, you can access a layer's transform through the `affineTransform` method (and the corresponding setter, `setAffineTransform:`). The value is a CGAffineTransform, familiar from Chapters 1 and 2. The transform is applied around the `anchorPoint`. (Thus, the `anchorPoint` has a second purpose that I didn't tell you about when discussing it earlier.)

You now know everything needed to understand the code that generated Figure 3-8, so here it is. In this code, `self` is the CompassLayer; it does no drawing of its own, but merely assembles and configures its sublayers. The four cardinal point letters are each drawn by a CATextLayer; they are drawn at the same coordinates, but they have different rotation transforms, and are anchored so that their rotation is centered at the center of the circle. To generate the arrow, we make ourselves the arrow layer's delegate and call `setNeedsDisplay`; this causes `drawLayer:inContext:` to be called in CompassLayer (that code is just the same code we developed for drawing the arrow in Chapter 2, and

is not repeated here). The arrow layer is positioned by an `anchorPoint` pinning its tail to the center of the circle, and rotated around that pin by a transform:

```
// the gradient
let g = CAGradientLayer()
g.contentsScale = UIScreen.mainScreen().scale
g.frame = self.bounds
g.colors = [
    UIColor.blackColor().CGColor,
    UIColor.redColor().CGColor
]
g.locations = [0.0,1.0]
self.addSublayer(g)
// the circle
let circle = CAShapeLayer()
circle.contentsScale = UIScreen.mainScreen().scale
circle.lineWidth = 2.0
circle.fillColor = UIColor(red:0.9, green:0.95, blue:0.93, alpha:0.9).CGColor
circle.strokeColor = UIColor.grayColor().CGColor
let p = CGPathCreateMutable()
CGPathAddEllipseInRect(p, nil, CGRectInset(self.bounds, 3, 3))
circle.path = p
self.addSublayer(circle)
circle.bounds = self.bounds
circle.position = self.bounds.center
// the four cardinal points
let pts = "NESW"
for (ix,c) in pts.characters.enumerate() {
    let t = CATextLayer()
    t.contentsScale = UIScreen.mainScreen().scale
    t.string = String(c)
    t.bounds = CGRectMake(0,0,40,40)
    t.position = circle.bounds.center
    let vert = circle.bounds.midY / t.bounds.height
    t.anchorPoint = CGPointMake(0.5, vert)
    t.alignmentMode = kCAAlignmentCenter
    t.foregroundColor = UIColor.blackColor().CGColor
    t.setAffineTransform(
        CGAffineTransformMakeRotation(CGFloat(ix)*CGFloat(M_PI)/2.0))
    circle.addSublayer(t)
}
// the arrow
let arrow = CALayer()
arrow.contentsScale = UIScreen.mainScreen().scale
arrow.bounds = CGRectMake(0, 0, 40, 100)
arrow.position = self.bounds.center
arrow.anchorPoint = CGPointMake(0.5, 0.8)
arrow.delegate = self // we will draw the arrow in the delegate method
arrow.setAffineTransform(CGAffineTransformMakeRotation(CGFloat(M_PI)/5.0))
self.addSublayer(arrow)
arrow.setNeedsDisplay() // draw, please
```

A full-fledged layer transform, the value of the `transform` property, takes place in three-dimensional space; its description includes a z-axis, perpendicular to both the x-axis and y-axis. (By default, the positive z-axis points out of the screen, toward the viewer's face.) Layers do not magically give you realistic three-dimensional rendering — for that you would use OpenGL, which is beyond the scope of this discussion. Layers are two-dimensional objects, and they are designed for speed and simplicity. Nevertheless, they do operate in three dimensions, quite sufficiently to give a cartoonish but effective sense of reality, especially when performing an animation. We've all seen the screen image flip like turning over a piece of paper to reveal what's on the back; that's a rotation in three dimensions.

A three-dimensional transform takes place around a three-dimensional extension of the `anchorPoint`, whose z-component is supplied by the `anchorPointZ` property. Thus, in the reduced default case where `anchorPointZ` is `0.0`, the `anchorPoint` is sufficient, as we've already seen in using CGAffineTransform.

The transform itself is described mathematically by a struct called a CATransform3D. The *Core Animation Function Reference* lists the functions for working with these transforms. They are a lot like the CGAffineTransform functions, except they've got a third dimension. For example, the function for making a 2D scale transform, `CGAffineTransformMakeScale`, takes two parameters; the function for making a 3D scale transform, `CATransform3DMakeScale`, takes three parameters.

The rotation 3D transform is a little more complicated. In addition to the angle, you also have to supply three coordinates describing the vector around which the rotation is to take place. Perhaps you've forgotten from your high-school math what a vector is, or perhaps trying to visualize three dimensions boggles your mind, so think of it this way.

Pretend for purposes of discussion that the anchor point is the origin, `(0.0,0.0,0.0)`. Now imagine an arrow emanating from the anchor point; its other end, the pointy end, is described by the three coordinates you provide. Now imagine a plane that intersects the anchor point, perpendicular to the arrow. That is the plane in which the rotation will take place; a positive angle is a clockwise rotation, as seen from the side of the plane with the arrow (Figure 3-9). In effect, the three coordinates you supply describe (relative to the anchor point) where your eye would have to be to see this rotation as an old-fashioned two-dimensional rotation.

A vector specifies a direction, not a point. Thus it makes no difference on what scale you give the coordinates: `(1.0,1.0,1.0)` means the same thing as `(10.0,10.0,10.0)`. If the three values are `(0.0,0.0,1.0)`, with all other things being equal, the case is collapsed to a simple CGAffineTransform, because the rotational plane is the screen. If the three values are `(0.0,0.0,-1.0)`, it's a backward CGAffineTransform, so that a

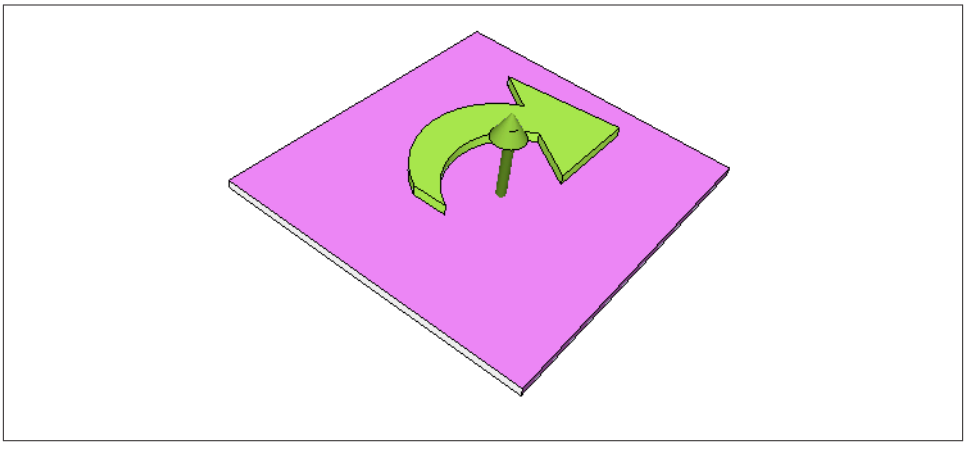

Figure 3-9. An anchor point plus a vector defines a rotation plane

positive angle looks counterclockwise (because we are looking at the "back side" of the rotational plane).

A layer can itself be rotated in such a way that its "back" is showing. For example, the following rotation flips a layer around its y-axis:

```
someLayer.transform = CATransform3DMakeRotation(CGFloat(M_PI), 0, 1, 0)
```

By default, the layer is considered double-sided, so when it is flipped to show its "back," what's drawn is an appropriately reversed version of the content of the layer (along with its sublayers, which by default are still drawn in front of the layer, but reversed and positioned in accordance with the layer's transformed coordinate system). But if the layer's `doubleSided` property is `false`, then when it is flipped to show its "back," the layer disappears (along with its sublayers); its "back" is transparent and empty.

Depth

There are two ways to place layers at different nominal depths with respect to their siblings. One is through the z-component of their `position`, which is the `zPosition` property. (Thus the `zPosition`, too, has a second purpose that I didn't tell you about earlier.) The other is to apply a transform that translates the layer's position in the z-direction. These two values, the z-component of a layer's position and the z-component of its translation transform, are related; in some sense, the `zPosition` is a shorthand for a translation transform in the z-direction. (If you provide both a `zPosition` and a z-direction translation, you can rapidly confuse yourself.)

In the real world, changing an object's `zPosition` would make it appear larger or smaller, as it is positioned closer or further away; but this, by default, is not the case in the world

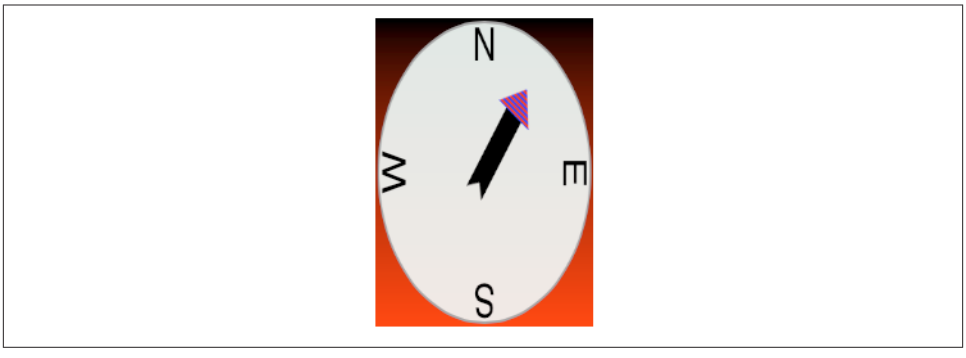

Figure 3-10. A disappointing page-turn rotation

of layer drawing. There is no attempt to portray perspective; the layer planes are drawn at their actual size and flattened onto one another, with no illusion of distance. (This is called *orthographic projection*, and is the way blueprints are often drawn to display an object from one side.)

However, there's a widely used trick for introducing a quality of perspective into the way layers are drawn: make them sublayers of a layer whose `sublayerTransform` property maps all points onto a "distant" plane. (This is probably just about the only thing the `sublayerTransform` property is ever used for.) Combined with orthographic projection, the effect is to apply one-point perspective to the drawing, so that things do get perceptibly smaller in the negative z-direction.

For example, let's try applying a sort of "page-turn" rotation to our compass: we'll anchor it at its right side and then rotate it around the y-axis. Here, the sublayer we're rotating (accessed through a property, `rotationLayer`) is the gradient layer, and the circle and arrow are its sublayers so that they rotate with it:

```
self.rotationLayer.anchorPoint = CGPointMake(1,0.5)
self.rotationLayer.position = CGPointMake(
    self.bounds.maxX, self.bounds.midY)
self.rotationLayer.transform = CATransform3DMakeRotation(
    CGFloat(M_PI)/4.0, 0, 1, 0)
```

The results are disappointing (Figure 3-10); the compass looks more squashed than rotated. Now, however, we'll also apply the distance-mapping transform. The superlayer here is `self`:

```
var transform = CATransform3DIdentity
transform.m34 = -1.0/1000.0
self.sublayerTransform = transform
```

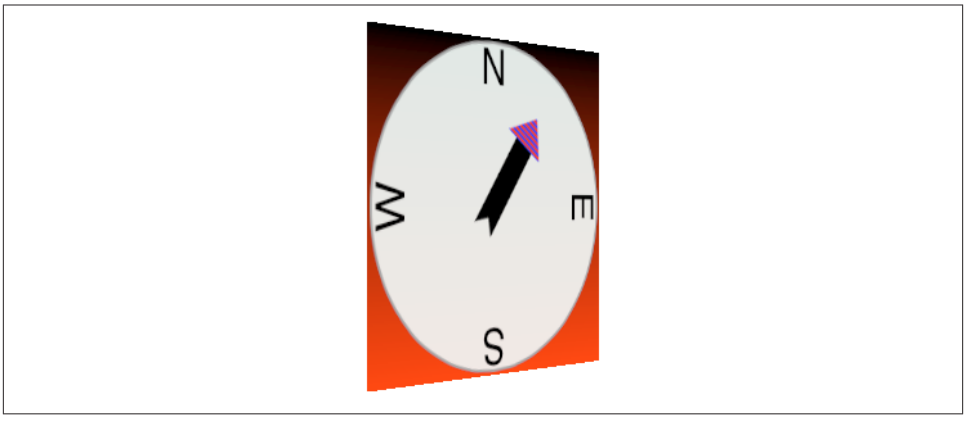

Figure 3-11. A dramatic page-turn rotation

The results (shown in Figure 3-11) are better, and you can experiment with values to replace `1000.0`; for example, `500.0` gives an even more exaggerated effect. Also, the `zPosition` of the `rotationLayer` will now affect how large it is.

Another way to draw layers with depth is to use CATransformLayer. This CALayer subclass doesn't do any drawing of its own; it is intended solely as a host for other layers. It has the remarkable feature that you can apply a transform to it and it will maintain the depth relationships among its own sublayers. For example:

```
// lay1 is a layer, f is a CGRect
let lay2 = CALayer()
lay2.frame = f
lay2.backgroundColor = UIColor.blueColor().CGColor
lay1.addSublayer(lay2)
let lay3 = CALayer()
lay3.frame = f.offsetBy(dx: 20, dy: 30)
lay3.backgroundColor = UIColor.greenColor().CGColor
lay3.zPosition = 10
lay1.addSublayer(lay3)
lay1.transform = CATransform3DMakeRotation(CGFloat(M_PI), 0, 1, 0)
```

In that code, the superlayer `lay1` has two sublayers, `lay2` and `lay3`. The sublayers are added in that order, so `lay3` is drawn in front of `lay2`. Then `lay1` is flipped like a page being turned by setting its `transform`. If `lay1` is a normal CALayer, the sublayer drawing order doesn't change; `lay3` is *still* drawn in front of `lay2`, even after the transform is applied. But if `lay1` is a CATransformLayer, `lay3` is drawn *behind* `lay2` after the transform; they are both sublayers of `lay1`, so their depth relationship is maintained.

Figure 3-12 shows our page-turn rotation yet again, still with the `sublayerTransform` applied to `self`, but this time the only sublayer of `self` is a CATransformLayer:

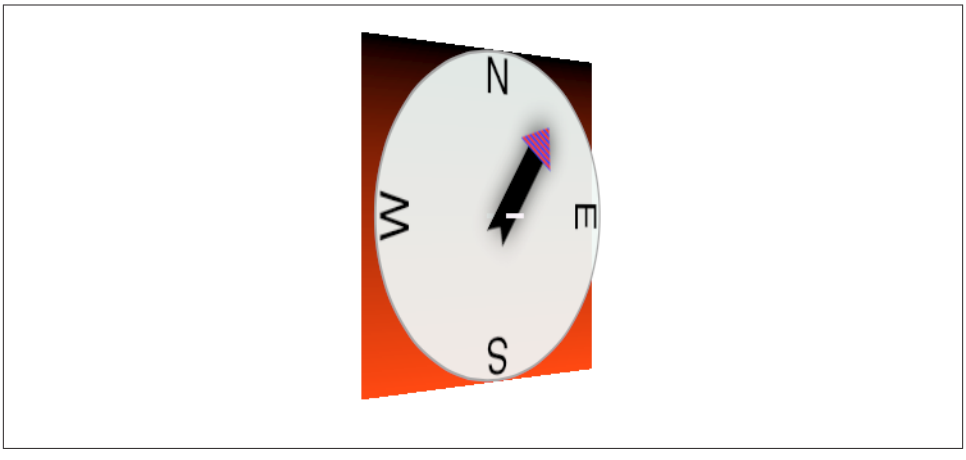

Figure 3-12. Page-turn rotation applied to a CATransformLayer

```
var transform = CATransform3DIdentity
transform.m34 = -1.0/1000.0
self.sublayerTransform = transform
let master = CATransformLayer()
master.frame = self.bounds
self.addSublayer(master)
self.rotationLayer = master
```

The CATransformLayer, to which the page-turn transform is applied, holds the gradient layer, the circle layer, and the arrow layer. Those three layers are at different depths (using different `zPosition` settings), and I've tried to emphasize the arrow's separation from the circle by adding a shadow (discussed in the next section):

```
circle.zPosition = 10
arrow.shadowOpacity = 1.0
arrow.shadowRadius = 10
arrow.zPosition = 20
```

You can see from its apparent offset that the circle layer floats in front of the gradient layer, but I wish you could see this page-turn as an animation, which makes the circle jump right out from the gradient as the rotation proceeds.

Even more remarkable, I've added a little white peg sticking through the arrow and running into the circle! It is a CAShapeLayer, rotated to be perpendicular to the CATransformLayer (I'll explain the rotation code later in this chapter):

```
let peg = CAShapeLayer()
peg.contentsScale = UIScreen.mainScreen().scale
peg.bounds = CGRectMake(0,0,3.5,50)
let p2 = CGPathCreateMutable()
CGPathAddRect(p2, nil, peg.bounds)
peg.path = p2
```

```
peg.fillColor = UIColor(red:1.0, green:0.95, blue:1.0, alpha:0.95).CGColor
peg.anchorPoint = CGPointMake(0.5,0.5)
peg.position = master.bounds.center
master.addSublayer(peg)
peg.setValue(M_PI/2, forKeyPath:"transform.rotation.x")
peg.setValue(M_PI/2, forKeyPath:"transform.rotation.z")
peg.zPosition = 15
```

In that code, the peg runs straight out of the circle toward the viewer, so it is initially seen end-on, and because a layer has no thickness, it is invisible. But as the CATransformLayer pivots in our page-turn rotation, the peg maintains its orientation relative to the circle, and comes into view. In effect, the drawing portrays a 3D model constructed entirely out of layers.

There is, I think, a slight additional gain in realism if the same `sublayerTransform` is applied also to the CATransformLayer, but I have not done so here.

Shadows, Borders, and Masks

A CALayer has many additional properties that affect details of how it is drawn. Since these drawing details can be applied to a UIView's underlying layer, they are effectively view features as well.

A CALayer can have a shadow, defined by its `shadowColor`, `shadowOpacity`, `shadowRadius`, and `shadowOffset` properties. To make the layer draw a shadow, set the `shadowOpacity` to a nonzero value. The shadow is normally based on the shape of the layer's nontransparent region, but deriving this shape can be calculation-intensive (so much so that in early versions of iOS, layer shadows weren't implemented). You can vastly improve performance by defining the shape yourself and assigning this shape as a CGPath to the `shadowPath` property.

If a layer's `masksToBounds` is `true`, no part of its shadow lying outside its bounds is drawn. (This includes the underlying layer of a view whose `clipsToBounds` is `true`.) Wondering why the shadow isn't appearing for a layer that masks to its bounds is a common beginner quandary.

A CALayer can have a border (`borderWidth`, `borderColor`); the `borderWidth` is drawn inward from the bounds, potentially covering some of the content unless you compensate.

A CALayer can be bounded by a rounded rectangle, by giving it a `cornerRadius` greater than zero. If the layer has a border, the border has rounded corners too. If the layer has a `backgroundColor`, that background is clipped to the shape of the rounded rectangle.

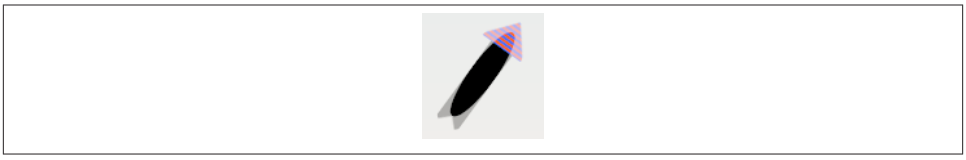

Figure 3-13. A layer with a mask

If the layer's `masksToBounds` is `true`, the layer's content and its sublayers are clipped by the rounded corners.

A CALayer can have a `mask`. This is itself a layer, whose content must be provided somehow. The transparency of the mask's content in a particular spot becomes (all other things being equal) the transparency of the layer at that spot. The mask's colors (hues) are irrelevant; only transparency matters. To position the mask, pretend it's a sublayer.

For example, Figure 3-13 shows our arrow layer, with the gray circle layer behind it, and a mask applied to the arrow layer. The mask is silly, but it illustrates very well how masks work: it's an ellipse, with an opaque fill and a thick, semitransparent stroke. Here's the code that generates and applies the mask:

```
let mask = CAShapeLayer()
mask.frame = arrow.bounds
let path = CGPathCreateMutable()
CGPathAddEllipseInRect(path, nil, CGRectInset(mask.bounds, 10, 10))
mask.strokeColor = UIColor(white:0.0, alpha:0.5).CGColor
mask.lineWidth = 20
mask.path = path
arrow.mask = mask
```

Using a mask, we can do manually and in a more general way what the `cornerRadius` and `masksToBounds` properties do. For example, here's a utility method that generates a CALayer suitable for use as a rounded rectangle `mask`:

```
func maskOfSize(sz:CGSize, roundingCorners rad:CGFloat) -> CALayer {
    let r = CGRect(origin:CGPointZero, size:sz)
    UIGraphicsBeginImageContextWithOptions(r.size, false, 0)
    let con = UIGraphicsGetCurrentContext()!
    CGContextSetFillColorWithColor(
        con, UIColor(white:0, alpha:0).CGColor)
    CGContextFillRect(con, r)
    CGContextSetFillColorWithColor(
        con, UIColor(white:0, alpha:1).CGColor)
    let p = UIBezierPath(roundedRect:r, cornerRadius:rad)
    p.fill()
    let im = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()
    let mask = CALayer()
```

```
        mask.frame = r
        mask.contents = im.CGImage
        return mask
    }
```

The CALayer returned from that method can be placed as a mask anywhere in a layer by adjusting its frame origin and assigning it as the layer's `mask`. The result is that all of that layer's content drawing and its sublayers (including, if this layer is a view's underlying layer, the view's subviews) are clipped to the rounded rectangle shape; everything outside that shape is not drawn. That's just one example of the sort of thing you can do with a mask. A mask can have values between opaque and transparent, and it can be any shape. And the transparent region doesn't have to be on the outside of the mask; you can use a mask that's opaque on the outside and transparent on the inside to punch a hole in a layer (or a view).

Alternatively, you can apply a mask as a view directly to another view through its `maskView` property, rather than having to drop down to the level of layers. This may be a notational convenience, but it is not functionally distinct from applying the mask view's layer to the view's layer; under the hood, in fact, it *is* applying the mask view's layer to the view's layer. Thus, for example, it does nothing to solve the problem that the mask is not automatically resized along with the view.

Layer Efficiency

By now, you're probably envisioning all sorts of compositing fun, with layers masking sublayers and laid semitransparently over other layers. There's nothing wrong with that, but when an iOS device is asked to shift its drawing from place to place, the movement may stutter because the device lacks the necessary computing power to composite repeatedly and rapidly. This sort of issue is likely to emerge particularly when your code performs an animation (Chapter 4) or when the user is able to animate drawing through touch, as when scrolling a table view (Chapter 8). You may be able to detect these problems by eye, and you can quantify them on a device by using the Core Animation template in Instruments, which shows the frame rate achieved during animation. Also, both the Core Animation template and the Simulator's Debug menu let you summon colored overlays that provide clues as to possible sources of inefficient drawing which can lead to such problems.

In general, opaque drawing is most efficient. (Nonopaque drawing is what Instruments marks in red as "blended layers.") If a layer will always be shown over a background consisting of a single color, you can give the layer its own background of that same color; when additional layer content is supplied, the visual effect will be the same as if that additional layer content were composited over a transparent background. For example, instead of an image masked to a rounded rectangle (with a layer's `cornerRadius` or `mask` property), you could use Core Graphics to clip the drawing of that image to a rounded

rectangle shape within the graphics context of a layer whose background color is the same as that of the destination in front of which the drawing will be shown.

Another way to gain some efficiency is by "freezing" the entirety of the layer's drawing as a bitmap. In effect, you're drawing everything in the layer to a secondary cache and using the cache to draw to the screen. Copying from a cache is less efficient than drawing directly to the screen, but this inefficiency may be compensated for, if there's a deep or complex layer tree, by not having to composite that tree every time we render. To do this, set the layer's `shouldRasterize` to `true` and its `rasterizationScale` to some sensible value (probably `UIScreen.mainScreen().scale`). You can always turn rasterization off again by setting `shouldRasterize` to `false`, so it's easy to rasterize just before some massive or sluggish rearrangement of the screen and then unrasterize afterward.

In addition, there's a layer property `drawsAsynchronously`. The default is `false`. If set to `true`, the layer's graphics context accumulates drawing commands and obeys them later on a background thread. Thus, your drawing commands run very quickly, because they are not in fact being obeyed at the time you issue them. I haven't had occasion to use this, but presumably there could be situations where it keeps your app responsive when drawing would otherwise be time-consuming.

Layers and Key–Value Coding

All of a layer's properties are accessible through key–value coding by way of keys with the same name as the property. Thus, to apply a mask to a layer, instead of saying this:

```
layer.mask = mask
```

we could have said:

```
layer.setValue(mask, forKey: "mask")
```

In addition, CATransform3D and CGAffineTransform values can be expressed through key–value coding and key paths. For example, instead of writing this:

```
self.rotationLayer.transform = CATransform3DMakeRotation(
    CGFloat(M_PI)/4.0, 0, 1, 0)
```

we can write this:

```
self.rotationLayer.setValue(M_PI/4, forKeyPath:"transform.rotation.y")
```

This notation is possible because CATransform3D is key–value coding compliant for a repertoire of keys and key paths. These are not properties, however; a CATransform3D doesn't have a `rotation` property. It doesn't have *any* properties, because it isn't even an object. You cannot say:

```
self.rotationLayer.transform.rotation.y = //... No, sorry
```

The transform key paths you'll use most often are:

- `"rotation.x"`, `"rotation.y"`, `"rotation.z"`
- `"rotation"` (same as `"rotation.z"`)
- `"scale.x"`, `"scale.y"`, `"scale.z"`
- `"translation.x"`, `"translation.y"`, `"translation.z"`
- `"translation"` (two-dimensional, a CGSize)

The Quartz Core framework also injects KVC compliance into CGPoint, CGSize, and CGRect, allowing you to use keys and key paths matching their struct component names. For a complete list of KVC compliant classes related to CALayer, along with the keys and key paths they implement, plus rules for how to wrap nonobject values as objects, see "Core Animation Extensions to Key-Value Coding" in Apple's *Core Animation Programming Guide*.

Moreover, you can treat a CALayer as a kind of dictionary, and get and set the value for *any* key. This means you can attach arbitrary information to an individual layer instance and retrieve it later. For example, earlier I mentioned that to apply manual layout to a layer's sublayers, you will need a way of identifying those sublayers. This feature could provide a way of doing that. For example:

```
myLayer1.setValue("Manny", forKey:"name")
myLayer2.setValue("Moe", forKey:"name")
```

A layer doesn't have a `name` property; the `"name"` key is something I'm attaching to these layers arbitrarily. Now I can identify these layers later by getting the value of their respective `"name"` keys.

Also, CALayer has a `defaultValueForKey:` class method; to implement it, you'll need to subclass and override. In the case of keys whose value you want to provide a default for, return that value; otherwise, return the value that comes from calling `super`. Thus, even if a value for a particular key has never been explicitly provided, it can have a non-`nil` value.

The truth is that this feature, though delightful (and I often wish that all classes behaved like this), is not put there for your convenience and enjoyment. It's there to serve as the basis for animation, which is the subject of the next chapter.