

MOBILE
PROGRAMMING
SERIES



iOS CORE ANIMATION ADVANCED TECHNIQUES

NICK LOCKWOOD

iOS Core Animation: Advanced
Techniques 中文译本

目錄

Introduction	0
图层树	1
图层与视图	1.1
图层的能力	1.2
使用图层	1.3
总结	1.4
寄宿图	2
contents属性	2.1
Custom Drawing	2.2
总结	2.3
图层几何学	3
布局	3.1
锚点	3.2
坐标系	3.3
Hit Testing	3.4
自动布局	3.5
总结	3.6
视觉效果	4
圆角	4.1
图层边框	4.2
阴影	4.3
图层蒙板	4.4
拉伸过滤	4.5
组透明	4.6
总结	4.7
变换	5
仿射变换	5.1

3D变换	5.2
固体对象	5.3
总结	5.4
专用图层	6
CAShapeLayer	6.1
CATextLayer	6.2
CATransformLayer	6.3
CAGradientLayer	6.4
CAReplicatorLayer	6.5
CAScrollLayer	6.6
CATiledLayer	6.7
CAEmitterLayer	6.8
CAEAGLLayer	6.9
AVPlayerLayer	6.10
总结	6.11
隐式动画	7
事务	7.1
完成块	7.2
图层行为	7.3
呈现与模型	7.4
总结	7.5
显式动画	8
属性动画	8.1
动画组	8.2
过渡	8.3
在动画过程中取消动画	8.4
总结	8.5
图层时间	9
`CAMediaTiming`协议	9.1
层级关系时间	9.2

手动动画	9.3
总结	9.4
缓冲	10
动画速度	10.1
自定义缓冲函数	10.2
总结	10.3
基于定时器的动画	11
定时帧	11.1
物理模拟	11.2
总结	11.3
性能调优	12
CPU VS GPU	12.1
测量，而不是猜测	12.2
Instruments	12.3
总结	12.4
高效绘图	13
软件绘图	13.1
矢量图形	13.2
脏矩形	13.3
异步绘制	13.4
总结	13.5
图像IO	14
加载和潜伏	14.1
缓存	14.2
文件格式	14.3
总结	14.4
图层性能	15
隐式绘制	15.1
离屏渲染	15.2
混合和过度绘制	15.3

减少图层数量	15.4
总结	15.5

iOS Core Animation: Advanced Techniques 中文译本

[gitbook上的地址](#)

本书翻译自：[iOS Core Animation: Advanced Techniques](#)

知识是人类进步的阶梯

翻译，喵~

译者为：

[AttackOnDobby](#)

[evenluo](#)

(排名不分先后，感谢他俩的付出！)

我（[ZsIsMe](#)）将译者的稿件搬运至[gitbook](#)上，方便大家查看。

如果在阅读过程中发现有什么问题，请到[这里](#)（本电子书在[github](#)上的地址）开issue，我会尽快改过来。

2015.2.9 电子书在制作过程中，章节可能会不断添加和修改或者合并。

图层的树状结构

巨妖有图层，洋葱也有图层，你有吗？我们都有图层 -- 史莱克

Core Animation其实是一个令人误解的命名。你可能认为它只是用来做动画的，但实际上它是从一个叫做*Layer Kit*这么一个不怎么和动画有关的名字演变而来，所以做动画这只是Core Animation特性的冰山一角。

Core Animation是一个复合引擎，它的职责就是尽可能快地组合屏幕上不同的可视内容，这个内容是被分解成独立的图层，存储在一个叫做图层树的体系之中。于是这个树形成了UIKit以及在iOS应用程序当中你所能看见的一切的基础。

在我们讨论动画之前，我们将从图层树开始，涉及一下Core Animation的静态组合以及布局特性。

图层与视图

如果你曾经在iOS或者Mac OS平台上写过应用程序，你可能会对视图的概念比较熟悉。一个视图就是在屏幕上显示的一个矩形块（比如图片，文字或者视频），它能够拦截类似于鼠标点击或者触摸手势等用户输入。视图在层级关系中可以互相嵌套，一个视图可以管理它的所有子视图的位置。图1.1显示了一种典型的视图层级关系

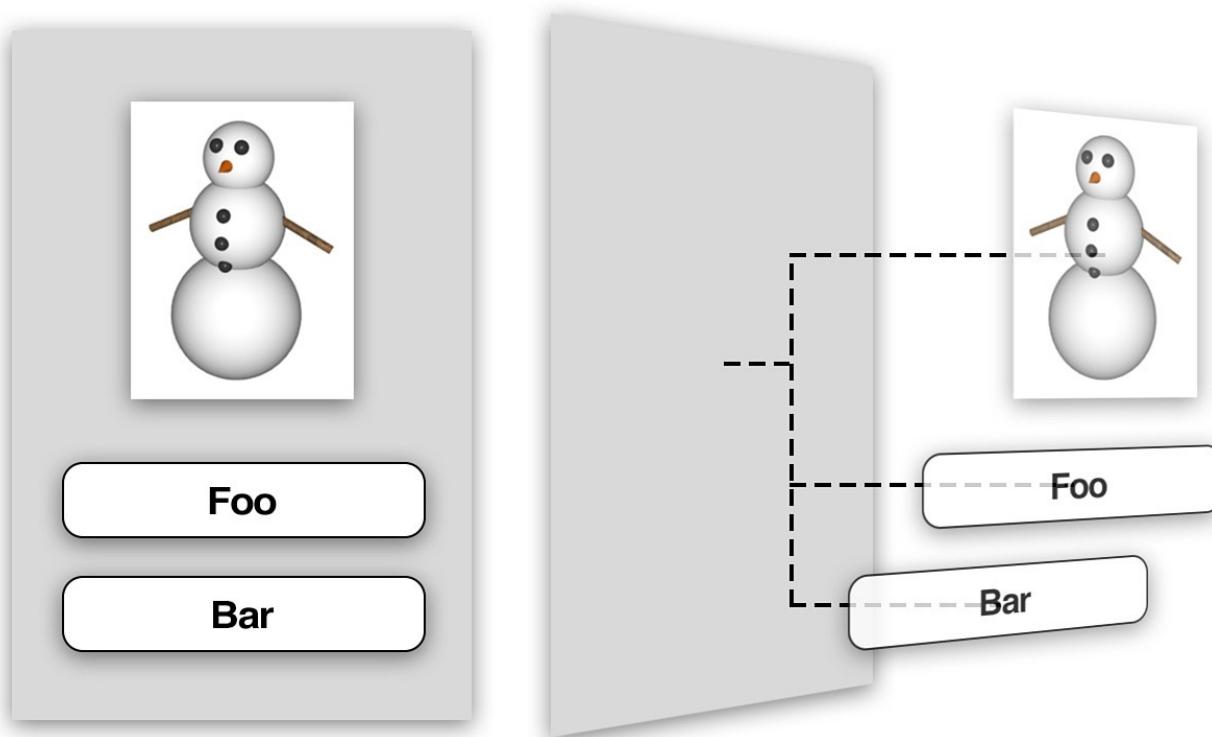


图1.1 一种典型的iOS屏幕（左边）和形成视图的层级关系（右边）

在iOS当中，所有的视图都从一个叫做 `UIView` 的基类派生而来，`UIView` 可以处理触摸事件，可以支持基于 *Core Graphics* 绘图，可以做仿射变换（例如旋转或者缩放），或者简单的类似于滑动或者渐变的动画。

CALayer

`CALayer` 类在概念上和 `UIView` 类似，同样也是一些被层级关系树管理的矩形块，同样也可以包含一些内容（像图片，文本或者背景色），管理子图层的位置。它们有一些方法和属性用来做动画和变换。和 `UIView` 最大的不同是 `CALayer` 不处理用户的交互。

`CALayer` 并不清楚具体的响应链（iOS通过视图层级关系用来传送触摸事件的机制），于是它并不能够响应事件，即使它提供了一些方法来判断是否一个触点在图层的范围之内（具体见第三章，“图层的几何学”）

平行的层级关系

每一个 `UIView` 都有一个 `CALayer` 实例的图层属性，也就是所谓的 *backing layer*，视图的职责就是创建并管理这个图层，以确保当子视图在层级关系中添加或者被移除的时候，他们关联的图层也同样对应在层级关系树当中有相同的操作（见图1.2）。

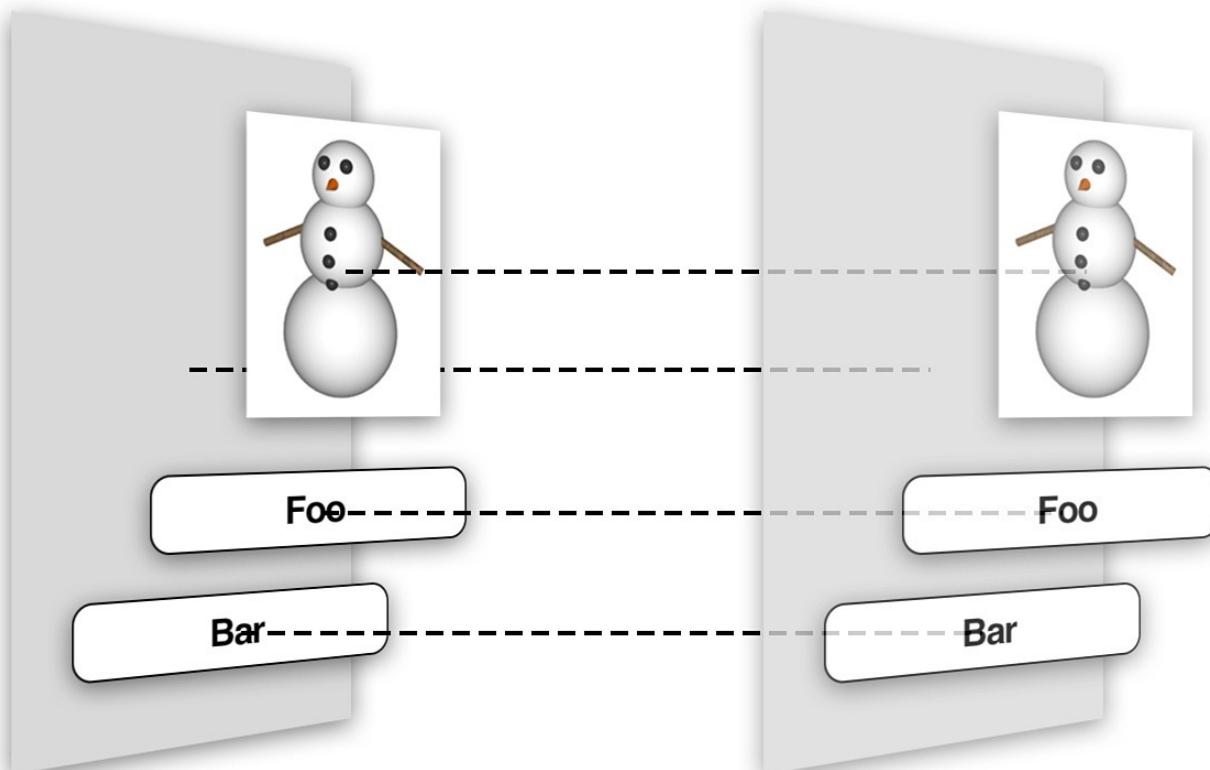


图1.2 图层的树状结构（左边）以及对应的视图层级（右边）

实际上这些背后关联的图层才是真正用来在屏幕上显示和做动画，`UIView` 仅仅是对它的一个封装，提供了一些iOS类似于处理触摸的具体功能，以及Core Animation底层方法的高级接口。

但是为什么iOS要基于 `UIView` 和 `CALayer` 提供两个平行的层级关系呢？为什么不用一个简单的层级来处理所有事情呢？原因在于要做职责分离，这样也能避免很多重复代码。在iOS和Mac OS两个平台上，事件和用户交互有很多地方的不同，基于多点触控的用户界面和基于鼠标键盘有着本质的区别，这就是为什么iOS有 `UIKit` 和 `UIView`，但是Mac OS有 `AppKit` 和 `NSView` 的原因。他们功能上很相似，但是在实现上有着显著的区别。

绘图，布局和动画，相比之下就是类似Mac笔记本和桌面系列一样应用于iPhone和iPad触屏的概念。把这种功能的逻辑分开并应用到独立的Core Animation框架，苹果就能够在iOS和Mac OS之间共享代码，使得对苹果自己的OS开发团队和第三方开发者去开发两个平台的应用更加便捷。

实际上，这里并不是两个层级关系，而是四个，每一个都扮演不同的角色，除了视图层级和图层树之外，还存在呈现树和渲染树，将在第七章“隐式动画”和第十二章“性能调优”分别讨论。

图层的能力

如果说 CALayer 是 UIView 内部实现细节，那我们为什么要全面地了解它呢？苹果当然为我们提供了优美简洁的 UIView 接口，那么我们是否就没必要直接去处理Core Animation的细节了呢？

某种意义上说的确是这样，对一些简单的需求来说，我们确实没必要处理 CALayer ，因为苹果已经通过 UIView 的高级API间接地使得动画变得很简单。

但是这种简单会不可避免地带来一些灵活上的缺陷。如果你略微想在底层做一些改变，或者使用一些苹果没有在 UIView 上实现的接口功能，这时除了介入Core Animation底层之外别无选择。

我们已经证实了图层不能像视图那样处理触摸事件，那么他能做哪些视图不能做的呢？这里有一些 UIView 没有暴露出来的CALayer的功能：

- 阴影，圆角，带颜色的边框
- 3D变换
- 非矩形范围
- 透明遮罩
- 多级非线性动画

我们将会在后续章节中探索这些功能，首先我们要关注一下在应用程序当中 CALayer 是怎样被利用起来的。

使用图层

首先我们来创建一个简单的项目，来操纵一些 `layer` 的属性。打开Xcode，使用*Single View Application*模板创建一个工程。

在屏幕中央创建一个小视图（大约 200×200 的尺寸），当然你可以手工编码，或者使用Interface Builder（随你方便）。确保你的视图控制器要添加一个视图的属性以便可以直接访问它。我们把它称作 `layerView`。

运行项目，应该能在浅灰色屏幕背景中看见一个白色方块（图1.3），如果没看见，可能需要调整一下背景window或者view的颜色

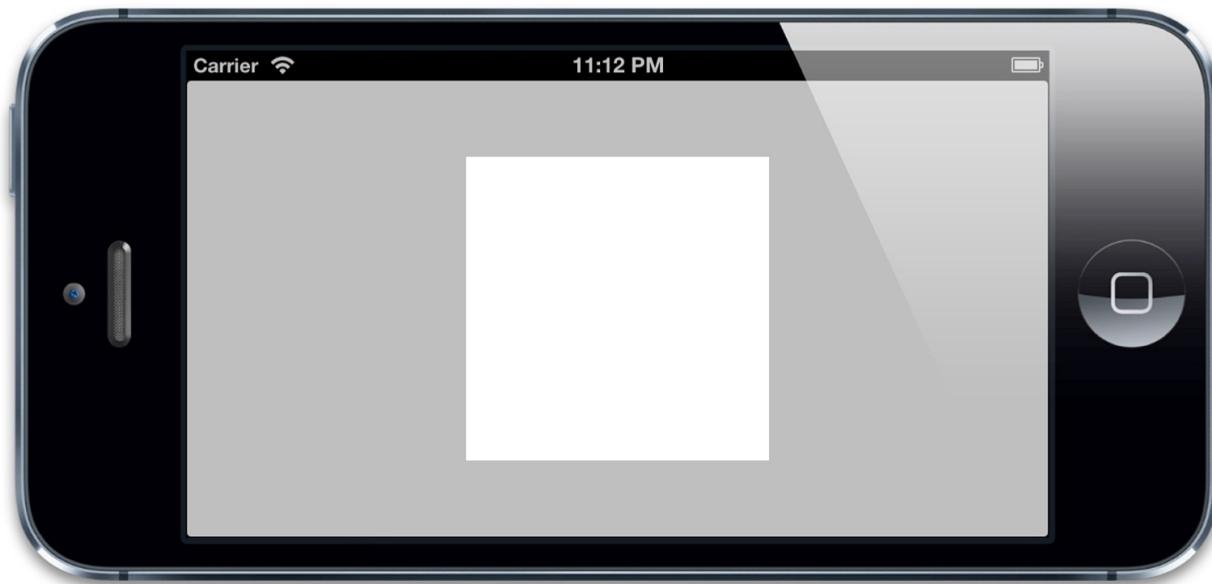


图1.3 灰色背景上的一个白色 `UIView`

这并没有什么令人激动的地方，我们来添加一个色块，在白色方块中间添加一个小的蓝色块。

我们当然可以简单地在已经存在的 `UIView` 上添加一个子视图（随意用代码或者IB），但这不能真正学到任何关于图层的东西。

于是我们来创建一个 `CALayer`，并且把它作为我们视图相关图层的子图层。尽管 `UIView` 类的接口中暴露了图层属性，但是标准的Xcode项目模板并没有包含Core Animation相关头文件。所以如果我们不给项目添加合适的库，是不能够使用任何图层相关的方法或者访问它的属性。所以首先需要添加QuartzCore框架到Build Phases标签（图1.4），然后在vc的.m文件中引入库。

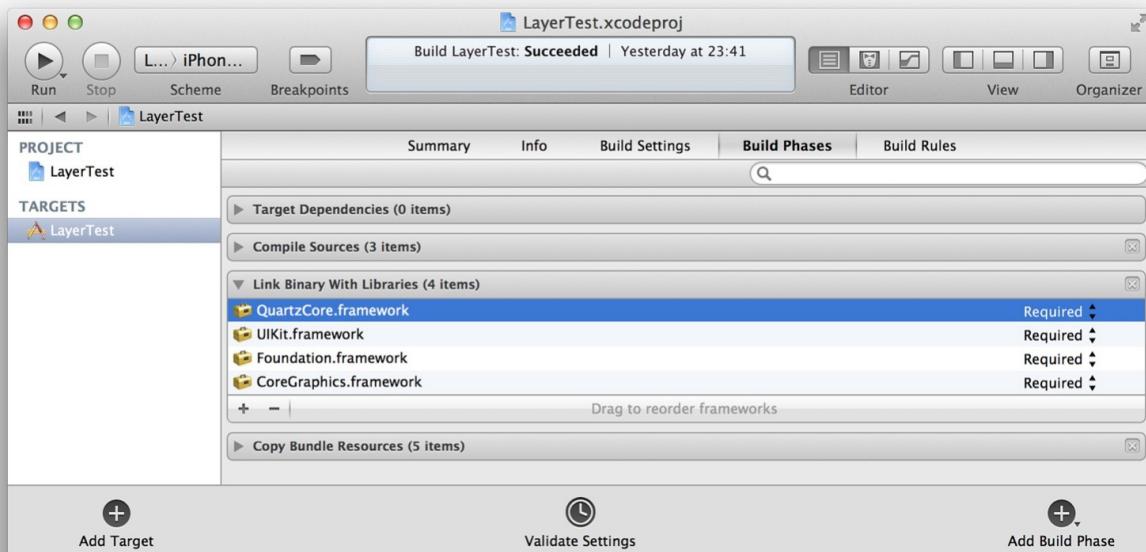


图1.4 把QuartzCore库添加到项目

之后就可以在代码中直接引用 `CALayer` 的属性和方法。在清单1.1中，我们用创建了一个 `CALayer`，设置了它的 `backgroundColor` 属性，然后添加到 `layerView` 背后相关图层的子图层（这段代码的前提是通过IB创建了 `layerView` 并做好了连接），图1.5显示了结果。

清单1.1 给视图添加一个蓝色子图层

```
#import "ViewController.h"  
#import  
@interface ViewController ()  
  
@property (nonatomic, weak) IBOutlet UIView *layerView;  
  
@end  
  
@implementation ViewController  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad];  
    //create sublayer  
    CALayer *blueLayer = [CALayer layer];  
    blueLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);  
    blueLayer.backgroundColor = [UIColor blueColor].CGColor;  
    //add it to our view  
    [self.layerView.layer addSublayer:blueLayer];  
}  
@end
```

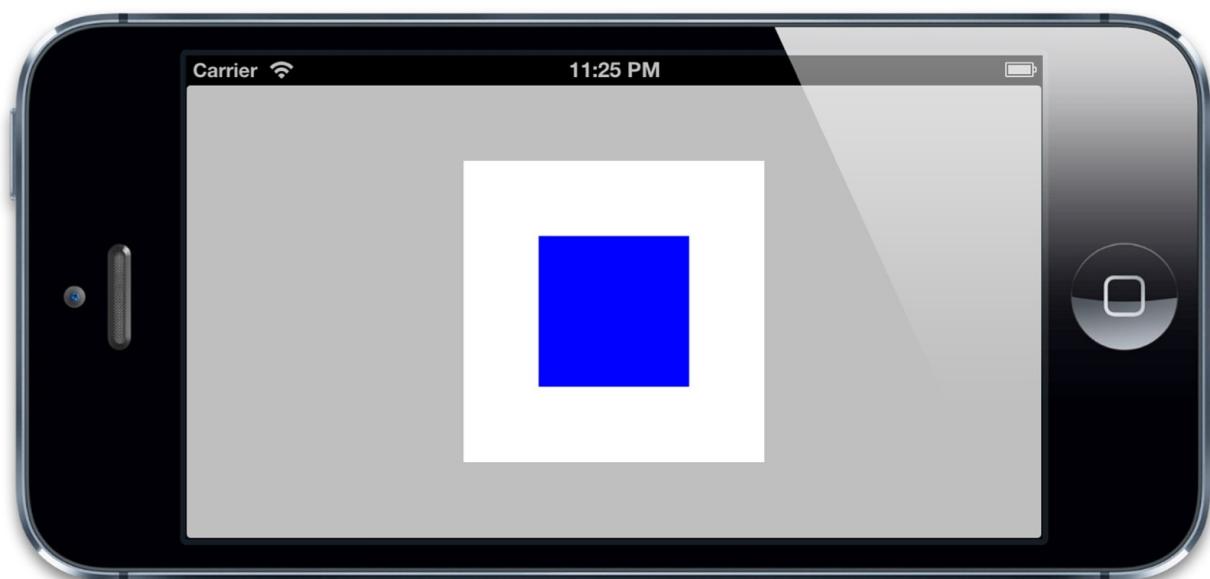


图1.5 白色 UIView 内部嵌套的蓝色 CALayer

一个视图只有一个相关联的图层（自动创建），同时它也可以支持添加无数多个子图层，从清单1.1可以看出，你可以显示创建一个单独的图层，并且把它直接添加到视图关联图层的子图层。尽管可以这样添加图层，但往往我们只是简单地处理视图，他们关联的图层并不需要额外地手动添加子图层。

在Mac OS平台，10.8版本之前，一个显著的性能缺陷就是由于用了视图层级而不是单独在一个视图内使用 CALayer 树状层级。但是在iOS平台，使用轻量级的 UIView 类并没有显著的性能影响（当然在Mac OS 10.8之后， NSView 的性能同样也得到很大程度的提高）。

使用图层关联的视图而不是 CALayer 的好处在于，你能在使用所有 CALayer 底层特性的同时，也可以使用 UIView 的高级API（比如自动排版，布局和事件处理）。

然而，当满足以下条件的时候，你可能更需要使用 CALayer 而不是 UIView：

- 开发同时可以在Mac OS上运行的跨平台应用
- 使用多种 CALayer 的子类（见第六章，“特殊的图层”），并且不想创建额外的 UIView 去包封装它们所有
- 做一些对性能特别挑剔的工作，比如对 UIView 一些可忽略不计的操作都会引起显著的不同（尽管如此，你可能会直接想使用OpenGL绘图）

但是这些例子都很少见，总的来说，处理视图会比单独处理图层更加方便。

总结

这一章阐述了图层的树状结构，说明了如何在iOS中由 `UIView` 的层级关系形成的一种平行的 `CALayer` 层级关系，在后面的实验中，我们创建了自己的 `CALayer`，并把它添加到图层树中。

在第二章，“图层关联的图片”，我们将要研究一下 `CALayer` 关联的图片，以及 Core Animation 提供的操作显示的一些特性。

寄宿图

寄宿图

图片胜过千言万语，界面抵得上千图片 ——Ben Shneiderman

我们在第一章『图层树』中介绍了**CALayer**类并创建了一个简单的有蓝色背景的图层。背景颜色还好啦，但是如果它仅仅是展现了一个单调的颜色未免也太无聊了。事实上**CALayer**类能够包含一张你喜欢的图片，这一章节我们将来探索**CALayer**的寄宿图（即图层中包含的图）。

contents属性

CALayer有一个属性叫做 `contents`，这个属性的类型被定义为 `id`，意味着它可以是任何类型的对象。在这种情况下，你可以给 `contents` 属性赋任何值，你的app仍然能够编译通过。但是，在实践中，如果你给 `contents` 赋的不是 `CGImage`，那么你得到的图层将是空白的。

`contents` 这个奇怪的表现是由 Mac OS 的历史原因造成的。它之所以被定义为 `id` 类型，是因为在 Mac OS 系统上，这个属性对 `CGImage` 和 `NSImage` 类型的值都起作用。如果你试图在 iOS 平台上将 `UIImage` 的值赋给它，只能得到一个空白的图层。一些初识 Core Animation 的 iOS 开发者可能会对这个感到困惑。

头疼的不仅仅是我们在刚才提到的这个问题。事实上，你真正要赋值的类型应该是 `CGImageRef`，它是一个指向 `CGImage` 结构的指针。`UIImage` 有一个 `CGImage` 属性，它返回一个 " `CGImageRef`"，如果你想把这个值直接赋值给 CALayer 的 `contents`，那你将会得到一个编译错误。因为 `CGImageRef` 并不是一个真正的 Cocoa 对象，而是一个 Core Foundation 类型。

尽管 Core Foundation 类型跟 Cocoa 对象在运行时貌似很像（被称作 toll-free bridging），他们并不是类型兼容的，不过你可以通过 `bridged` 关键字转换。如果要给图层的寄宿图赋值，你可以按照以下这个方法：

```
layer.contents = (__bridge id)image.CGImage;
```

如果你没有使用 ARC（自动引用计数），你就不需要 `__bridge` 这部分。但是，你干嘛不用 ARC？！

让我们来继续修改我们在第一章新建的工程，以便能够展示一张图片而不仅仅是一个背景色。我们已经用代码的方式建立一个图层，那我们就不需要额外的图层了。那么我们就直接把 `layerView` 的宿主图层的 `contents` 属性设置成图片。

清单 2.1 更新后的代码。

```

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad]; //load an image
    UIImage *image = [UIImage imageNamed:@"Snowman.png"];

    //add it directly to our view's layer
    self.layerView.layer.contents = (__bridge id)image.CGImage;
}
@end

```

图表2.1 在UIView的宿主图层中显示一张图片



我们用这些简单的代码做了一件很有趣的事情：我们利用CALayer在一个普通的UIView中显示了一张图片。这不是一个UIImageView，它不是我们通常用来展示图片的方法。通过直接操作图层，我们使用了一些新的函数，使得UIView更加有趣了。

contentGravity

你可能已经注意到了我们的雪人看起来有点。。。胖 ==！ 我们加载的图片并不刚好是一个方的，为了适应这个视图，它有一点点被拉伸了。在使用UIImageView的时候遇到过同样的问题，解决方法就是把 `contentMode` 属性设置成更合适的值，像这样：

```
view.contentMode = UIViewContentModeScaleAspectFit;
```

这个方法基本和我们遇到的情况的解决方法已经接近了（你可以试一下：），不过 `UIView` 大多数视觉相关的属性比如 `contentMode`，对这些属性的操作其实是对对应图层的操作。

`CALayer` 与 `contentMode` 对应的属性叫做 `contentsGravity`，但是它是一个 `NSString` 类型，而不是像对应的 `UIKit` 部分，那里面的值是枚举。`contentsGravity` 可选的常量值有以下一些：

- `kCAGravityCenter`
- `kCAGravityTop`
- `kCAGravityBottom`
- `kCAGravityLeft`
- `kCAGravityRight`
- `kCAGravityTopLeft`
- `kCAGravityTopRight`
- `kCAGravityBottomLeft`
- `kCAGravityBottomRight`
- `kCAGravityResize`
- `kCAGravityResizeAspect`
- `kCAGravityResizeAspectFill`

和 `contentMode` 一样，`contentsGravity` 的目的是为了决定内容在图层的边界中怎么对齐，我们将使用 `kCAGravityResizeAspect`，它的效果等同于 `UIViewContentModeScaleAspectFit`，同时它还能在图层中等比例拉伸以适应图层的边界。

```
self.layerView.layer.contentsGravity = kCAGravityResizeAspect;
```

图2.2 可以看到结果



图2.2 正确地设置 `contentsGravity` 的值

contentsScale

`contentsScale` 属性定义了寄宿图的像素尺寸和视图大小的比例，默认情况下它是一个值为 1.0 的浮点数。

`contentsScale` 的目的并不是那么明显。它并不是总会对屏幕上的寄宿图有影响。如果你尝试对我们的例子设置不同的值，你就会发现根本没有任何影响。因为 `contents` 由于设置了 `contentsGravity` 属性，所以它已经被拉伸以适应图层的边界。

如果你只是单纯地想放大图层的 `contents` 图片，你可以通过使用图层的 `transform` 和 `affineTransform` 属性来达到这个目的（见第五章『Transforms』，里面对此有解释），这(指放大)也不是 `contentsScale` 的目的所在。

`contentsScale` 属性其实属于支持高分辨率（又称Hi-DPI或Retina）屏幕机制的一部分。它用来判断在绘制图层的时候应该为寄宿图创建的空间大小，和需要显示的图片的拉伸度（假设并没有设置 `contentsGravity` 属性）。`UIView`有一个类似功能但是非常少用到的 `contentScaleFactor` 属性。

如果 `contentsScale` 设置为 1.0，将会以每个点 1 个像素绘制图片，如果设置为 2.0，则会以每个点 2 个像素绘制图片，这就是我们熟知的 Retina 屏幕。（如果你对像素和点的概念不是很清楚的话，这个章节的后面部分将会对此做出解释）。

这并不会对我们在使用kCAGravityResizeAspect时产生任何影响，因为它就是拉伸图片以适应图层而已，根本不会考虑到分辨率问题。但是如果我们把 `contentsGravity` 设置为kCAGravityCenter（这个值并不会拉伸图片），那将会有很明显的的变化（如图2.3）



图2.3 用错误的 `contentsScale` 属性显示Retina图片

如你所见，我们的雪人不仅有点大还有点像素的颗粒感。那是因为和`UIImage`不同，`CGImage`没有拉伸的概念。当我们使用`UIImage`类去读取我们的雪人图片的时候，他读取了高质量的Retina版本的图片。但是当我们用`CGImage`来设置我们的图层的内容时，拉伸这个因素在转换的时候就丢失了。不过我们可以通过手动设置 `contentsScale` 来修复这个问题（如2.2清单），图2.4是结果

```

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad]; //load an image
    UIImage *image = [UIImage imageNamed:@"Snowman.png"]; //add it d:
    self.layerView.layer.contents = (__bridge id)image.CGImage; //cer
    self.layerView.layer.contentsGravity = kCAGravityCenter;

    //set the contentsScale to match image
    self.layerView.layer.contentsScale = image.scale;
}

@end

```



图2.4 同样的Retina图片设置了正确的 `contentsScale` 之后

当用代码的方式来处理寄宿图的时候，一定要记住要手动的设置图层的 `contentsScale` 属性，否则，你的图片在Retina设备上就显示得不正确啦。代码如下：

```
layer.contentsScale = [UIScreen mainScreen].scale;
```

maskToBounds

现在我们的雪人总算是显示了正确的大小，不过你也许已经发现了另外一些事情：他超出了视图的边界。默认情况下，`UIView`仍然会绘制超过边界的内容或是子视图，在`CALayer`下也是这样的。

`UIView`有一个叫做 `clipsToBounds` 的属性可以用来决定是否显示超出边界的內容，`CALayer`对应的属性叫做 `masksToBounds`，把它设置为 YES，雪人就在边界里啦～（如图2.5）

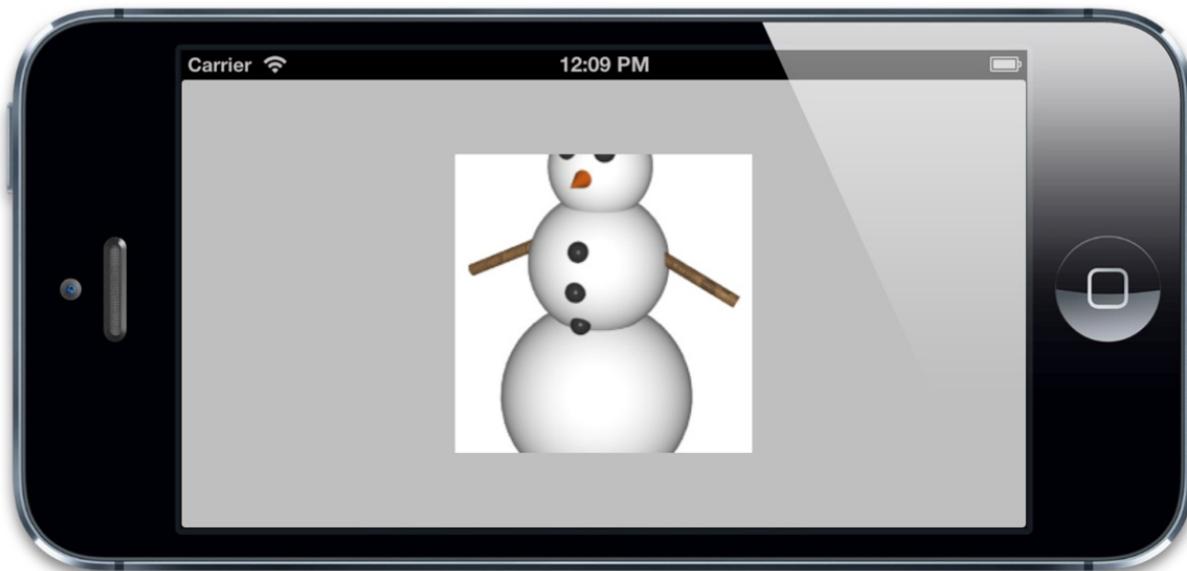


图2.5 使用 `masksToBounds` 来修建图层内容

contentsRect

`CALayer`的 `contentsRect` 属性允许我们在图层边框里显示寄宿图的一个子域。这涉及到图片是如何显示和拉伸的，所以要比 `contentsGravity` 灵活多了

和 `bounds`，`frame` 不同，`contentsRect` 不是按点来计算的，它使用了单位坐标，单位坐标指定在0到1之间，是一个相对值（像素和点就是绝对值）。所以他们是相对与寄宿图的尺寸的。iOS使用了以下的坐标系统：

- 点 —— 在iOS和Mac OS中最常见的坐标体系。点就像是虚拟的像素，也被称作逻辑像素。在标准设备上，一个点就是一个像素，但是在Retina设备上，一个点等于2*2个像素。iOS用点作为屏幕的坐标测算体系就是为了在Retina设备和普通设备上能有一致的视觉效果。

- 像素 —— 物理像素坐标并不会用来屏幕布局，但是仍然与图片有相对关系。
`UIImage`是一个屏幕分辨率解决方案，所以指定点来度量大小。但是一些底层的图片表示如`CGImage`就会使用像素，所以你要清楚在Retina设备和普通设备上，他们表现出来了不同的大小。
- 单位 —— 对于与图片大小或是图层边界相关的显示，单位坐标是一个方便的度量方式，当大小改变的时候，也不需要再次调整。单位坐标在OpenGL这种纹理坐标系统中用得很多，Core Animation中也用到了单位坐标。

默认的 `contentsRect` 是 $\{0, 0, 1, 1\}$ ，这意味着整个寄宿图默认都是可见的，如果我们指定一个小一点的矩形，图片就会被裁剪（如图2.6）

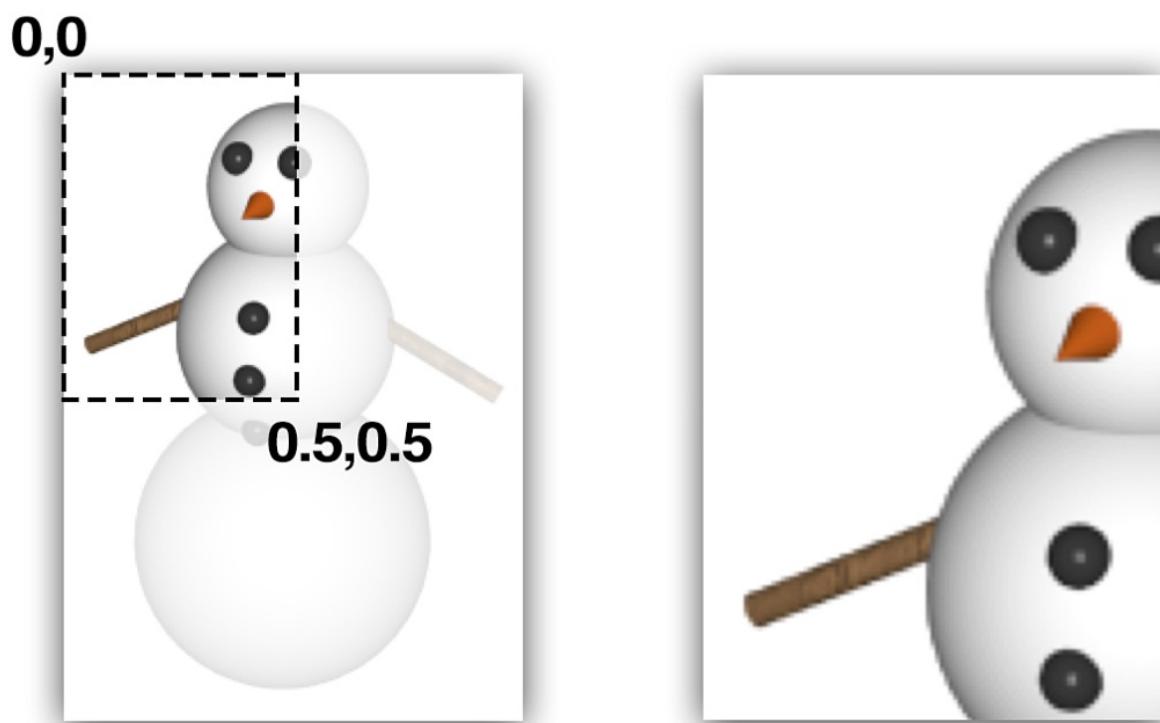


图2.6 一个自定义的 `contentsRect` (左) 和之前显示的内容 (右)

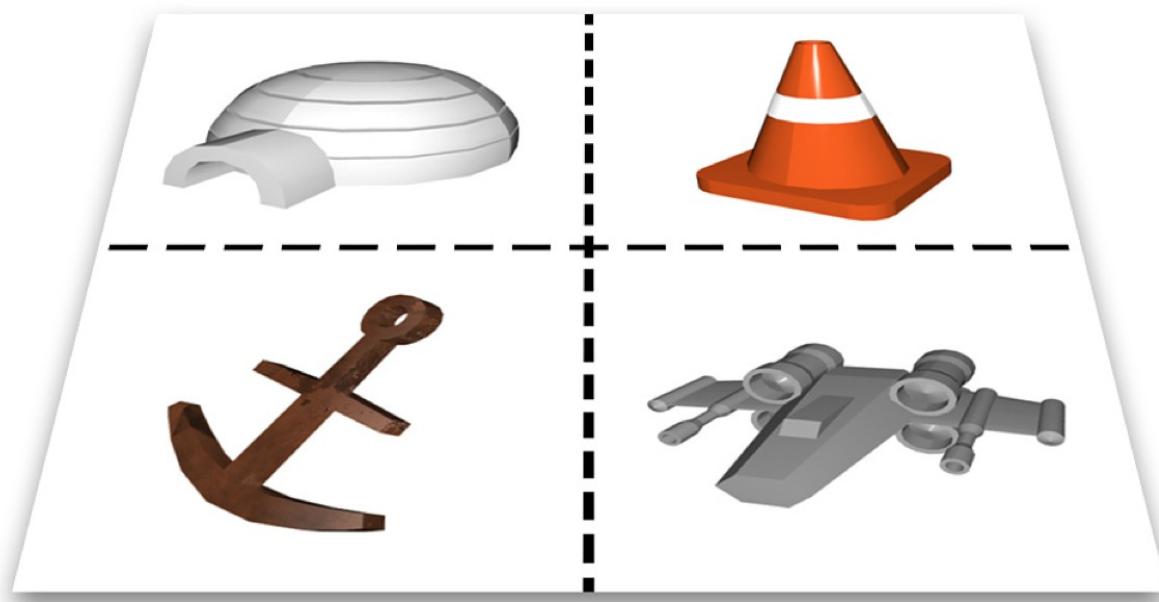
事实上给 `contentsRect` 设置一个负数的原点或是大于 $\{1, 1\}$ 的尺寸也是可以的。这种情况下，最外面的像素会被拉伸以填充剩下的区域。

`contentsRect` 在app中最有趣的地方在于一个叫做*image sprites*（图片拼合）的用法。如果你有游戏编程的经验，那么你一定对图片拼合的概念很熟悉，图片能够在屏幕上独立地变更位置。抛开游戏编程不谈，这个技术常用来指代载入拼合的图片，跟移动图片一点关系也没有。

典型地，图片拼合后可以打包整合到一张大图上一次性载入。相比多次载入不同的图片，这样做能够带来很多方面的好处：内存使用，载入时间，渲染性能等等

2D游戏引擎入Cocos2D使用了拼合技术，它使用OpenGL来显示图片。不过我们可以使用拼合在一个普通的UIKit应用中，对！就是使用 `contentsRect`

首先，我们需要一个拼合后的图表——一个包含小一些的拼合图的大图片。如图2.7所示：

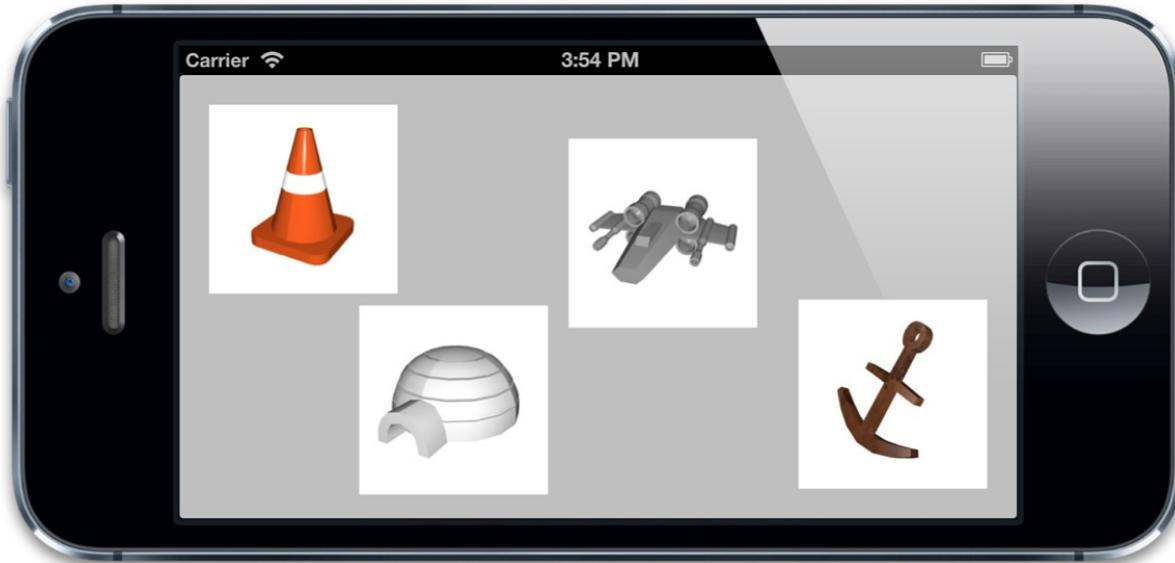


接下来，我们要在app中载入并显示这些拼合图。规则很简单：像平常一样载入我们的大图，然后把它赋值给四个独立的图层的 `contents`，然后设置每个图层的 `contentsRect` 来去掉我们不想显示的部分。

我们的工程中需要一些额外的视图。（为了避免太多代码。我们将使用Interface Builder来拜访他们的位置，如果你愿意还是可以用代码的方式来实现的）。清单2.3有需要的代码，图2.8展示了结果

```
@interface ViewController ()  
@property (nonatomic, weak) IBOutlet UIView *coneView;  
@property (nonatomic, weak) IBOutlet UIView *shipView;  
@property (nonatomic, weak) IBOutlet UIView *iglooView;  
@property (nonatomic, weak) IBOutlet UIView *anchorView;  
@end  
  
@implementation ViewController  
  
- (void)addSpriteImage:(UIImage *)image withContentRect:(CGRect)rect  
{  
    layer.contents = (__bridge id)image.CGImage;  
  
    //scale contents to fit  
    layer.contentsGravity = kCAGravityResizeAspect;  
  
    //set contentsRect  
    layer.contentsRect = rect;  
}  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad]; //load sprite sheet  
    UIImage *image = [UIImage imageNamed:@"Sprites.png"];  
    //set igloo sprite  
    [self addSpriteImage:image withContentRect:CGRectMake(0, 0, 0.5, 0.5)];  
    //set cone sprite  
    [self addSpriteImage:image withContentRect:CGRectMake(0.5, 0, 0.5, 0.5)];  
    //set anchor sprite  
    [self addSpriteImage:image withContentRect:CGRectMake(0, 0.5, 0.5, 0.5)];  
    //set spaceship sprite  
    [self addSpriteImage:image withContentRect:CGRectMake(0.5, 0.5, 0.5, 0.5)];  
}  
@end
```





拼合不仅给app提供了一个整洁的载入方式，还有效地提高了载入性能（单张大图比多张小图载入地更快），但是如果有手动安排的话，他们还是有一些不方便的，如果你需要在一个已经创建好的品和图上做一些尺寸上的修改或者其他变动，无疑是比较麻烦的。

Mac上有一些商业软件可以为你自动拼合图片，这些工具自动生成一个包含拼合后的坐标的XML或者plist文件，拼合图片的使用大大简化。这个文件可以和图片一同载入，并给每个拼合的图层设置 `contentsRect`，这样开发者就不用手动写代码来摆放位置了。

这些文件通常在OpenGL游戏中使用，不过呢，你要是有兴趣在一些常见的app中使用拼合技术，那么一个叫做LayerSprites的开源库

（<https://github.com/nicklockwood/LayerSprites>），它能够读取Cocos2D格式中的拼合图并在普通的Core Animation层中显示出来。

contentsCenter

本章我们介绍的最后一个和内容有关的属性是 `contentsCenter`，看名字你可能会以为它可能跟图片的位置有关，不过这名字着实误导了你。`contentsCenter` 其实是一个CGRect，它定义了一个固定的边框和一个在图层上可拉伸的区域。改变 `contentsCenter` 的值并不会影响到寄宿图的显示，除非这个图层的大小改变了，你才看得到效果。

默认情况下，`contentsCenter` 是 $\{0, 0, 1, 1\}$ ，这意味着如果大小（由 `contentGravity` 决定）改变了，那么寄宿图将会均匀地拉伸开。但是如果我们将原点的值并减小尺寸。我们会在图片的周围创造一个边框。图2.9展示了 `contentsCenter` 设置为 $\{0.25, 0.25, 0.5, 0.5\}$ 的效果。

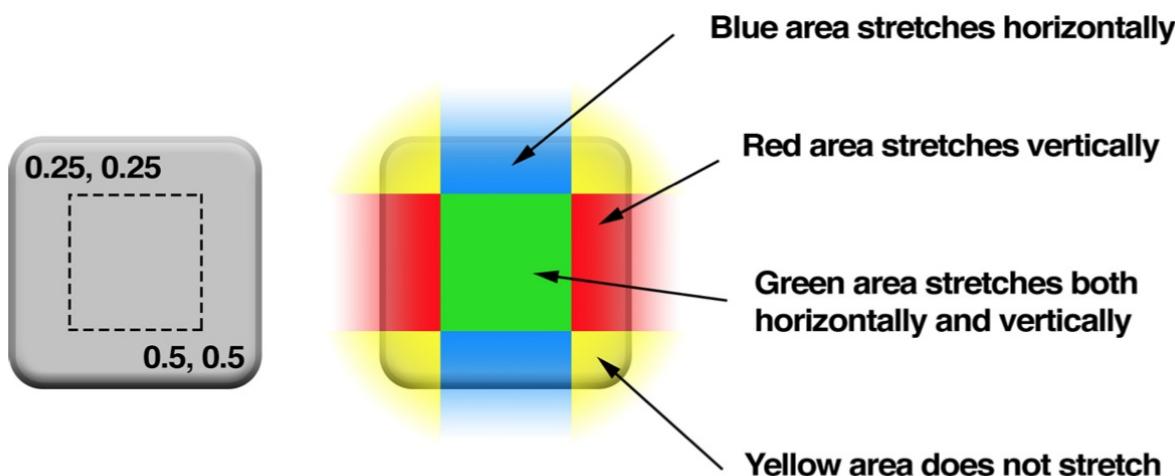


图2.9 `contentsCenter` 的例子

这意味着我们可以随意重设尺寸，边框仍然会是连续的。他工作起来的效果和 `UIImage` 里的 `-resizableImageWithCapInsets:` 方法效果非常类似，只是它可以运用到任何寄宿图，甚至包括在Core Graphics运行时绘制的图形（本章稍后会讲到）。

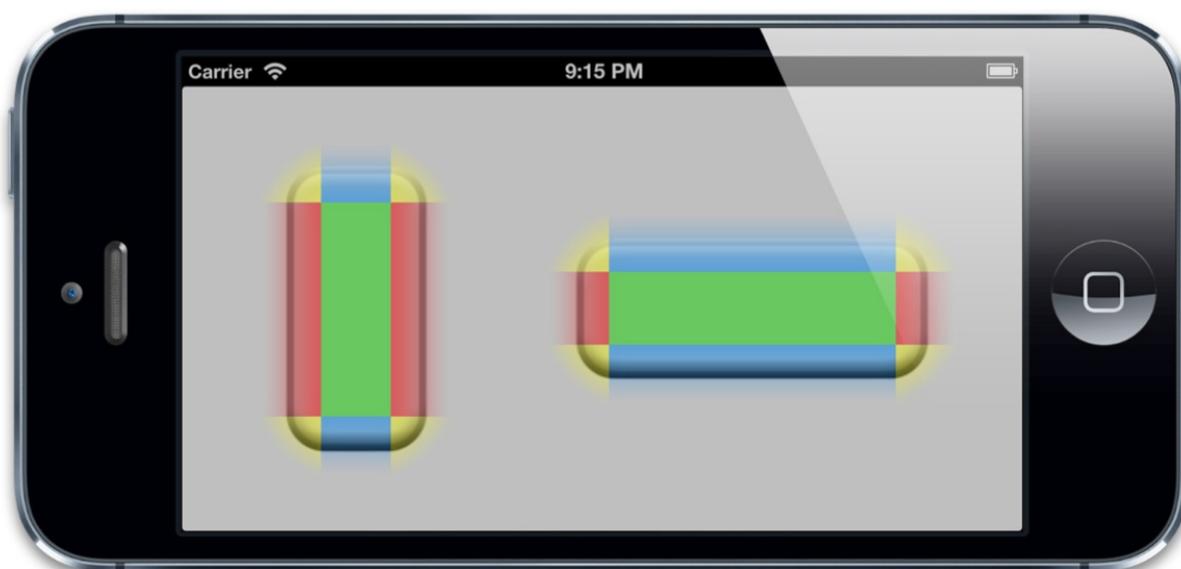


图2.10 同一图片使用不同的 `contentsCenter`

清单2.4 演示了如何编写这些可拉伸视图。不过，contentsCenter的另一个很酷的特性就是，它可以在Interface Builder里面配置，根本不用写代码。如图2.11

清单2.4 用 contentsCenter 设置可拉伸视图

```
@interface ViewController ()  
  
@property (nonatomic, weak) IBOutlet UIView *button1;  
@property (nonatomic, weak) IBOutlet UIView *button2;  
  
@end  
  
@implementation ViewController  
  
- (void)addStretchableImage:(UIImage *)image withContentCenter:(CGRect)rect  
{  
    //set image  
    layer.contents = (__bridge id)image.CGImage;  
  
    //set contentsCenter  
    layer.contentsCenter = rect;  
}  
  
- (void)viewDidLoad  
{  
    [super viewDidLoad]; //load button image  
    UIImage *image = [UIImage imageNamed:@"Button.png"];  
  
    //set button 1  
    [self addStretchableImage:image withContentCenter:CGRectMake(0.25, 0.25, 0.5, 0.5)];  
  
    //set button 2  
    [self addStretchableImage:image withContentCenter:CGRectMake(0.25, 0.75, 0.5, 0.5)];  
}  
  
@end
```

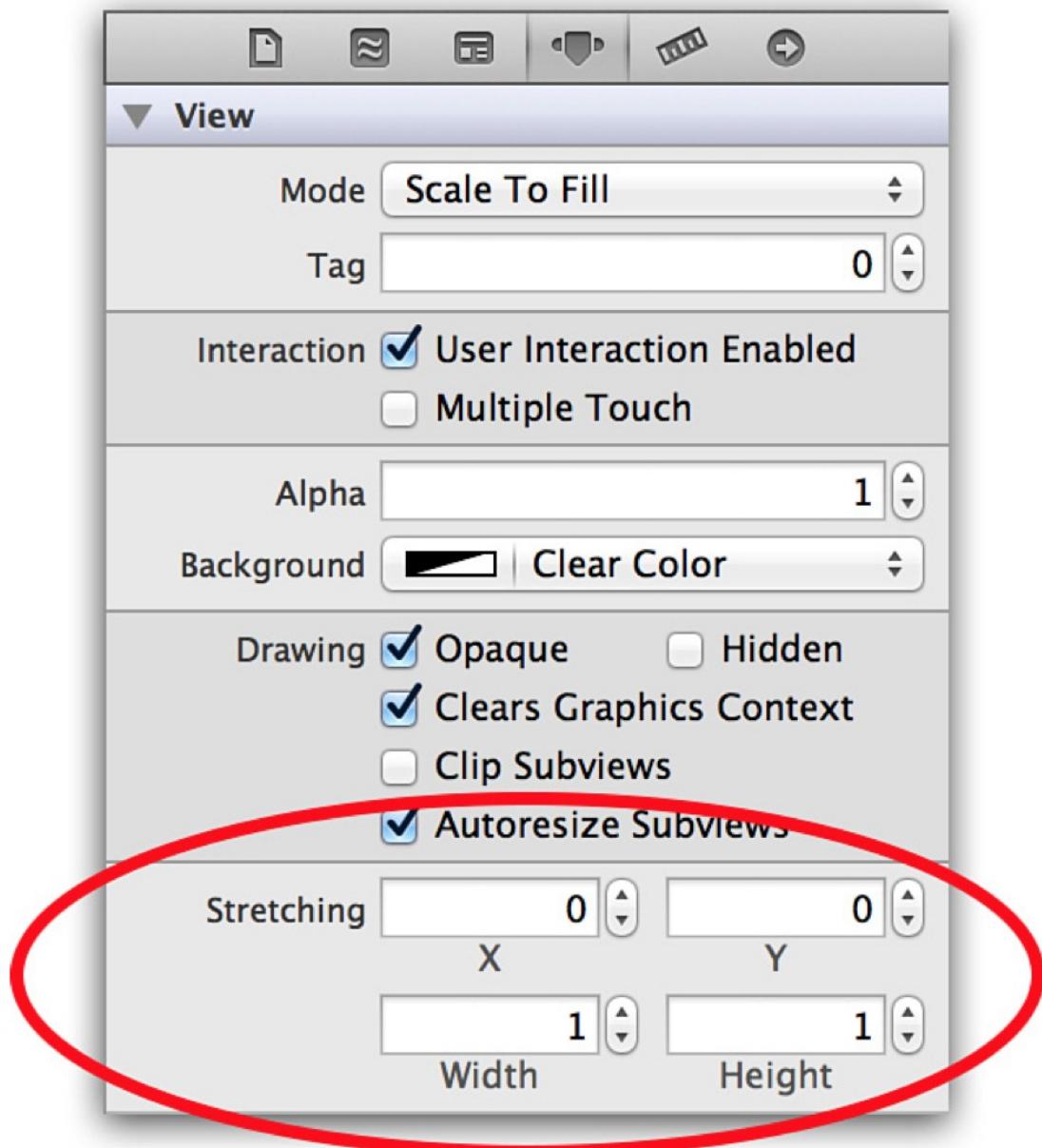


图2.11 用Interface Builder 探测窗口控制 contentsCenter 属性

Custom Drawing

给 `contents` 赋 `CGImage` 的值不是唯一的设置寄宿图的方法。我们也可以直接用 Core Graphics 直接绘制寄宿图。能够通过继承 `UIView` 并实现 `-drawRect:` 方法来自定义绘制。

`-drawRect:` 方法没有默认的实现，因为对 `UIView` 来说，寄宿图并不是必须的，它不在意那到底是单调的颜色还是有一个图片的实例。如果 `UIView` 检测到 `-drawRect:` 方法被调用了，它就会为视图分配一个寄宿图，这个寄宿图的像素尺寸等于视图大小乘以 `contentsScale` 的值。

如果你不需要寄宿图，那就不要创建这个方法了，这会造成 CPU 资源和内存的浪费，这也是为什么苹果建议：如果没有自定义绘制的任务就不要在子类中写一个空的 `-drawRect:` 方法。

当视图在屏幕上出现的时候 `-drawRect:` 方法就会被自动调用。
`-drawRect:` 方法里面的代码利用 Core Graphics 去绘制一个寄宿图，然后内容就会被缓存起来直到它需要被更新（通常是因为开发者调用了 `-setNeedsDisplay` 方法，尽管影响到表现效果的属性值被更改时，一些视图类型会被自动重绘，如 `bounds` 属性）。虽然 `-drawRect:` 方法是一个 `UIView` 方法，事实上都是底层的 `CALayer` 安排了重绘工作和保存了因此产生的图片。

`CALayer` 有一个可选的 `delegate` 属性，实现了 `CALayerDelegate` 协议，当 `CALayer` 需要一个内容特定的信息时，就会从协议中请求。`CALayerDelegate` 是一个非正式协议，其实也就是说没有 `CALayerDelegate @protocol` 可以让你在类里面引用啦。你只需要调用你想调用的方法，`CALayer` 会帮你做剩下的。（`delegate` 属性被声明为 `id` 类型，所有的代理方法都是可选的）。

当需要被重绘时，`CALayer` 会请求它的代理给他一个寄宿图来显示。它通过调用下面这个方法做到的：

```
(void)displayLayer:(CALayer *)layer;
```

趁着这个机会，如果代理想直接设置 `contents` 属性的话，它就可以这么做，不然没有别的方法可以调用了。如果代理不实现 `-displayLayer:` 方法，`CALayer` 就会转而尝试调用下面这个方法：

```
- (void)drawLayer:(CALayer *)layer inContext:(CGContextRef)ctx;
```

在调用这个方法之前，`CALayer`创建了一个合适尺寸的空寄宿图（尺寸由 `bounds` 和 `contentsScale` 决定）和一个Core Graphics的绘制上下文环境，为绘制寄宿图做准备，他作为`ctx`参数传入。

让我们来继续第一章的项目让它实现`CALayerDelegate`并做一些绘图工作吧（见清单2.5）。图2.12是他的结果

清单2.5 实现`CALayerDelegate`

```
@implementation ViewController
- (void)viewDidLoad
{
    [super viewDidLoad];

    //create sublayer
    CALayer *blueLayer = [CALayer layer];
    blueLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
    blueLayer.backgroundColor = [UIColor blueColor].CGColor;

    //set controller as layer delegate
    blueLayer.delegate = self;

    //ensure that layer backing image uses correct scale
    blueLayer.contentsScale = [UIScreen mainScreen].scale; //add layer
    [self.layerView.layer addSublayer:blueLayer];

    //force layer to redraw
    [blueLayer display];
}

- (void)drawLayer:(CALayer *)layer inContext:(CGContextRef)ctx
{
    //draw a thick red circle
    CGContextSetLineWidth(ctx, 10.0f);
    CGContextSetStrokeColorWithColor(ctx, [UIColor redColor].CGColor);
    CGContextStrokeEllipseInRect(ctx, layer.bounds);
}
@end
```

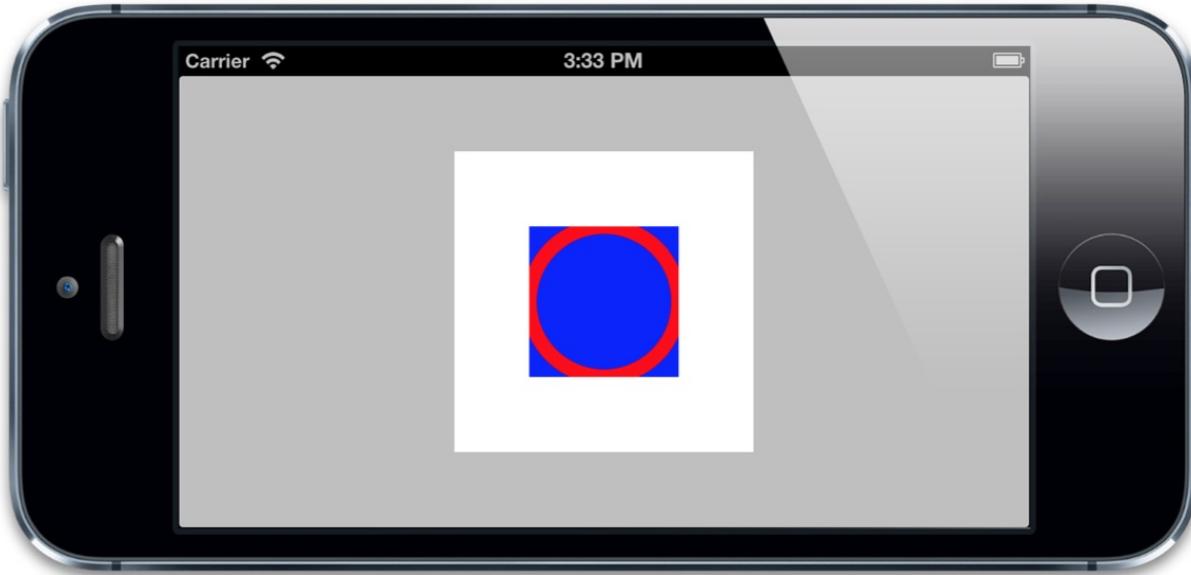


图2.12 实现CALayerDelegate来绘制图层

注意一下一些有趣的事情：

- 我们在blueLayer上显式地调用了 `-display`。不同于UIView，当图层显示在屏幕上时，CALayer不会自动重绘它的内容。它把重绘的决定权交给了开发者。
- 尽管我们没有用 `masksToBounds` 属性，绘制的那个圆仍然沿边界被裁剪了。这是因为当你使用CALayerDelegate绘制寄宿图的时候，并没有对超出边界外的内容提供绘制支持。

现在你理解了CALayerDelegate，并知道怎么使用它。但是除非你创建了一个单独的图层，你几乎没有机会用到CALayerDelegate协议。因为当UIView创建了它的宿主图层时，它就会自动地把图层的delegate设置为它自己，并提供了一个 `-displayLayer:` 的实现，那所有的问题就都没了。

当使用寄宿了视图的图层的时候，你也不必实现 `-displayLayer:` 和 `-drawLayer:inContext:` 方法来绘制你的寄宿图。通常做法是实现UIView的 `-drawRect:` 方法，UIView就会帮你做完剩下的工作，包括在需要重绘的时候调用 `-display` 方法。

总结

本章介绍了寄宿图和一些相关的属性。你学到了如何显示和放置图片，使用拼合技术来显示，以及用**CALayerDelegate**和**Core Graphics**来绘制图层内容。

在第三章，"图层几何学"中，我们将会探讨一下图层的几何，观察他们是如何放置和改变相互的尺寸的。

图层几何学

不熟悉几何学的人就不要来这里了 --柏拉图学院入口的签名

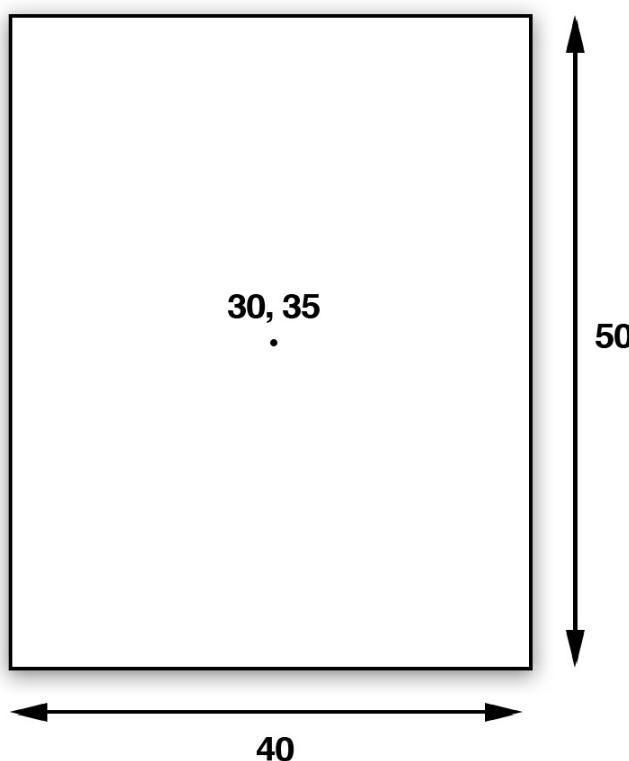
在第二章里面，我们介绍了图层背后的图片，和一些控制图层坐标和旋转的属性。在这一章中，我们将要看一看图层内部是如何根据父图层和兄弟图层来控制位置和尺寸的。另外我们也会涉及如何管理图层的几何结构，以及它是如何被自动调整和自动布局影响的。

布局

`UIView` 有三个比较重要的布局属性：`frame`，`bounds` 和 `center`，`CALayer` 对应地叫做 `frame`，`bounds` 和 `position`。为了能清楚区分，图层用了“`position`”，视图用了“`center`”，但是他们都代表同样的值。

`frame` 代表了图层的外部坐标（也就是在父图层上占据的空间），`bounds` 是内部坐标 ($\{0, 0\}$ 通常是图层的左上角)，`center` 和 `position` 都代表了相对于父图层 `anchorPoint` 所在的位置。`anchorPoint` 的属性将会在后续介绍到，现在把它想成图层的中心点就好了。图3.1显示了这些属性是如何相互依赖的。

10, 10



View:

frame = {10, 10, 40, 50}

bounds = {0, 0, 40, 50}

center = {30, 35}

Layer:

frame = {10, 10, 40, 50}

bounds = {0, 0, 40, 50}

position = {30, 35}

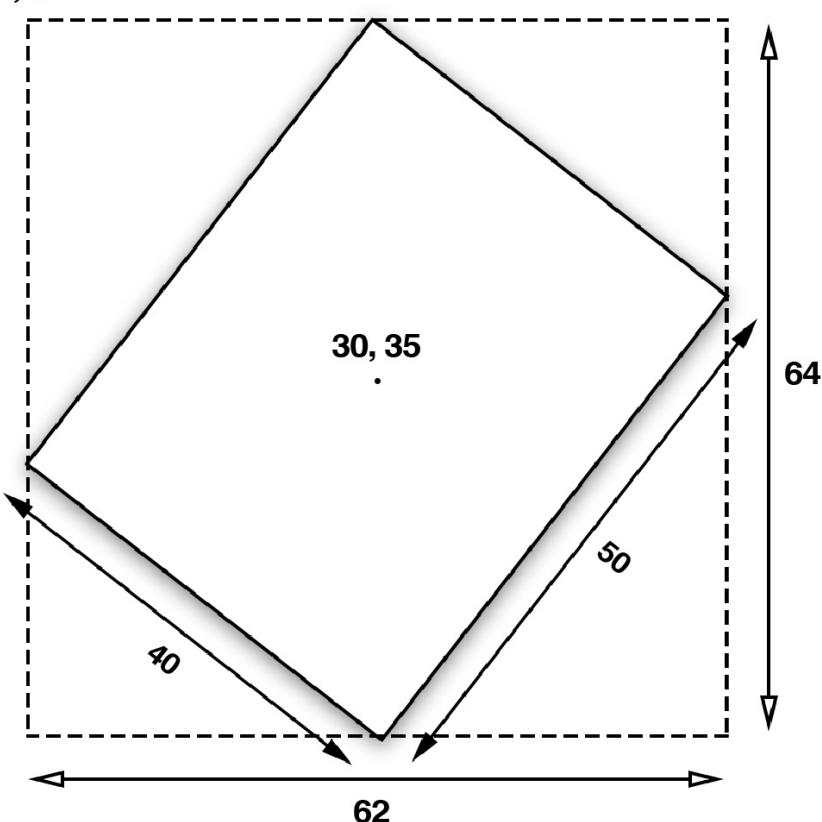
图3.1 `UIView` 和 `CALayer` 的坐标系

视图的 `frame`，`bounds` 和 `center` 属性仅仅是存取方法，当操纵视图的 `frame`，实际上是在改变位于视图下方 `CALayer` 的 `frame`，不能够独立于图层之外改变视图的 `frame`。

对于视图或者图层来说，`frame` 并不是一个非常清晰的属性，它其实是一个虚拟属性，是根据 `bounds`，`position` 和 `transform` 计算而来，所以当其中任何一个值发生改变，`frame`都会变化。相反，改变`frame`的值同样会影响到他们当中的值

记住当对图层做变换的时候，比如旋转或者缩放，`frame` 实际上代表了覆盖在图层旋转之后的整个轴对齐的矩形区域，也就是说 `frame` 的宽高可能和 `bounds` 的宽高不再一致了（图3.2）

-1, 3



View:

`frame = {-1, 3, 62, 64}`

`bounds = {0, 0, 40, 50}`

`center = {30, 35}`

Layer:

`frame = {-1, 3, 62, 64}`

`bounds = {0, 0, 40, 50}`

`position = {30, 35}`

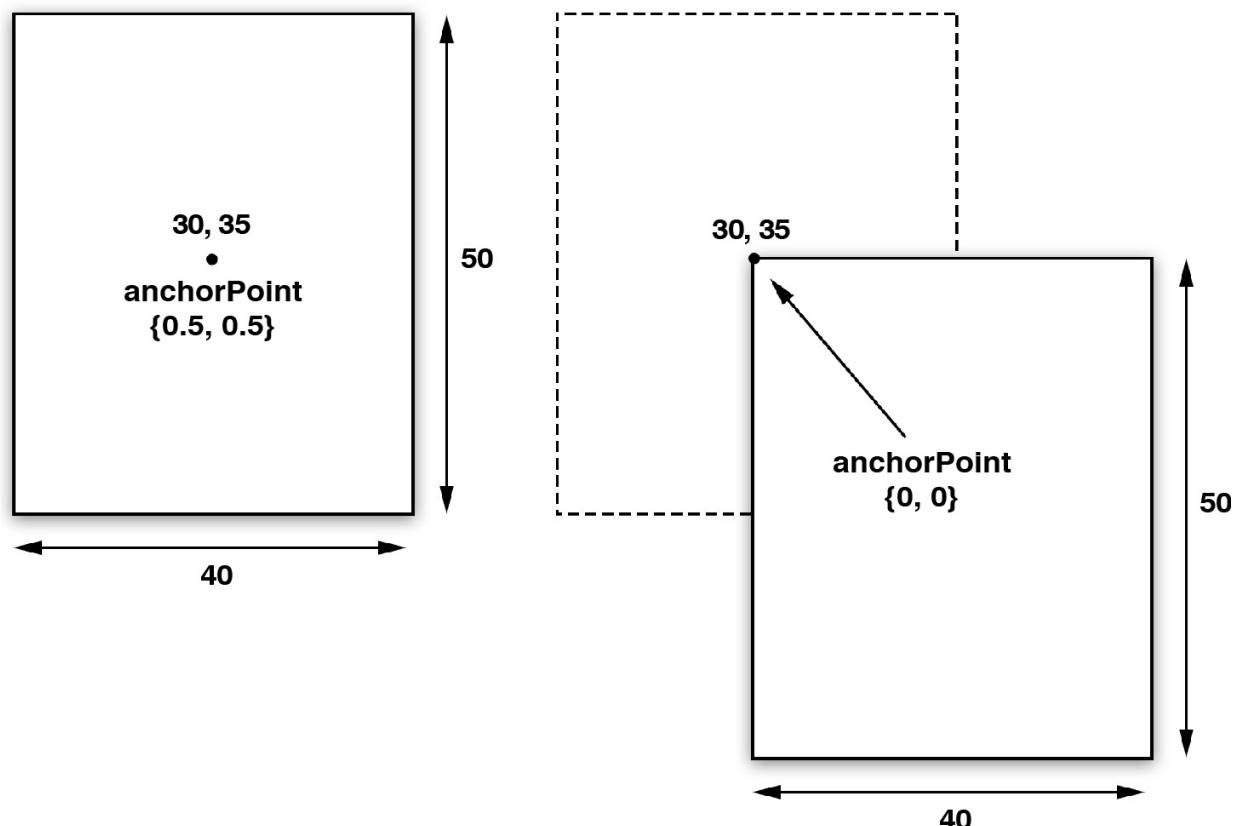
图3.2 旋转一个视图或者图层之后的 `frame` 属性

锚点

之前提到过，视图的 `center` 属性和图层的 `position` 属性都指定了 `anchorPoint` 相对于父图层的位置。图层的 `anchorPoint` 通过 `position` 来控制它的 `frame` 的位置，你可以认为 `anchorPoint` 是用来移动图层的把柄。

默认来说，`anchorPoint` 位于图层的中点，所以图层的将会以这个点为中心放置。`anchorPoint` 属性并没有被 `UIView` 接口暴露出来，这也是视图的 `position` 属性被叫做“center”的原因。但是图层的 `anchorPoint` 可以被移动，比如你可以把它置于图层 `frame` 的左上角，于是图层的内容将会向右下角的 `position` 方向移动（图3.3），而不是居中了。

10, 10

图3.3 改变 `anchorPoint` 的效果

和第二章提到的 `contentsRect` 和 `contentsCenter` 属性类似，`anchorPoint` 用单位坐标来描述，也就是图层的相对坐标，图层左上角是{0, 0}，右下角是{1, 1}，因此默认坐标是{0.5, 0.5}。`anchorPoint` 可以通过指定x和y值小于0或者大于1，使它放置在图层范围之外。

注意在图3.3中，当改变了 `anchorPoint`，`position` 属性保持固定的值并没有发生改变，但是 `frame` 却移动了。

那在什么场合需要改变 `anchorPoint` 呢？既然我们可以随意改变图层位置，那改变 `anchorPoint` 不会造成困惑么？为了举例说明，我们来举一个实用的例子，创建一个模拟闹钟的项目。

钟面和钟表由四张图片组成（图3.4），为了简单说明，我们还是用传统的方式来装载和加载图片，使用四个 `UIImageView` 实例（当然你也可以用正常的视图，设置他们图层的 `contents` 图片）。

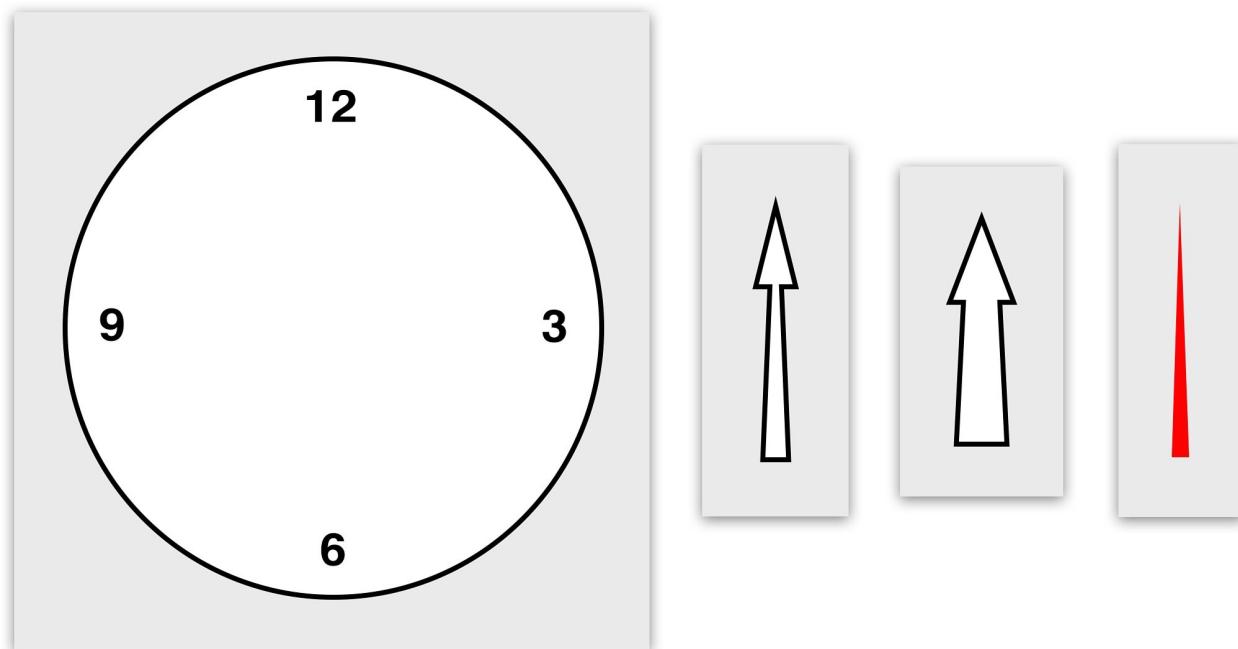


图3.4 组成钟面和钟表的四张图片

闹钟的组件通过IB来排列（图3.5），这些图片视图嵌套在一个容器视图之内，并且自动调整和自动布局都被禁用了。这是因为自动调整会影响到视图的 `frame`，而根据图3.2的演示，当视图旋转的时候，`frame` 是会发生改变的，这将会导致一些布局上的失灵。

我们用 `NSTimer` 来更新闹钟，使用视图的 `transform` 属性来旋转钟表（如果你对这个属性不太熟悉，不要着急，我们将会在第5章“变换”当中详细说明），具体代码见清单3.1

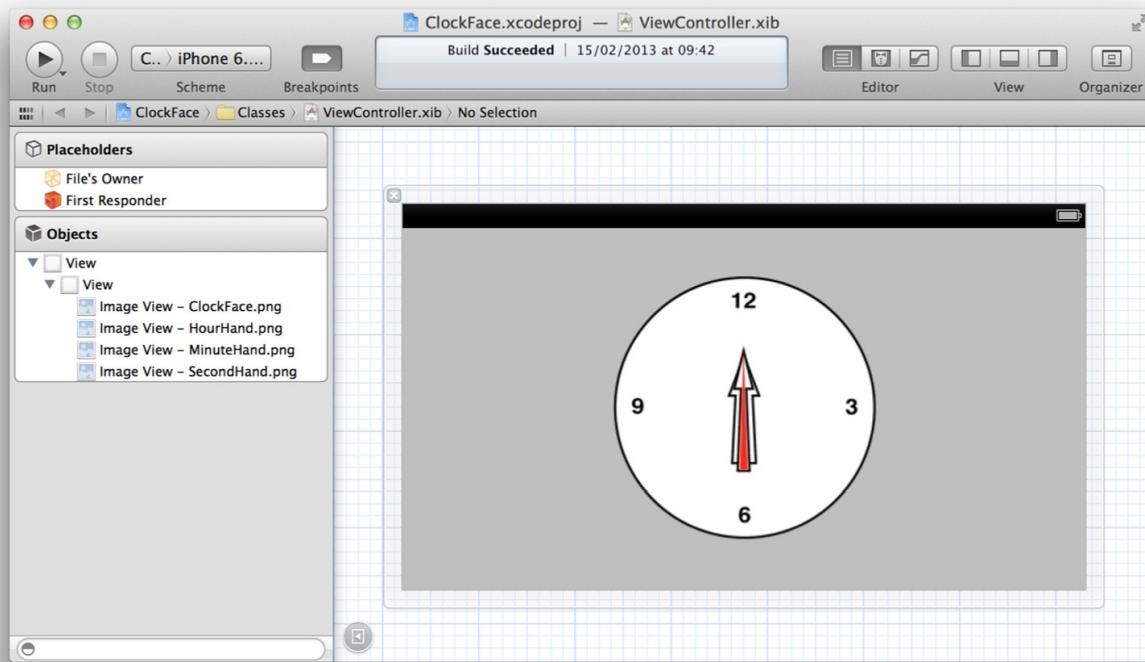


图3.5 在Interface Builder中布局闹钟视图

清单3.1 Clock

```

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIImageView *hourHand;
@property (nonatomic, weak) IBOutlet UIImageView *minuteHand;
@property (nonatomic, weak) IBOutlet UIImageView *secondHand;
@property (nonatomic, weak) NSTimer *timer;

@end

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];
    //start timer
    self.timer = [NSTimer scheduledTimerWithTimeInterval:1.0 target:self selector:@selector(tick) userInfo:nil repeats:YES];
}

//set initial hand positions
[self tick];
}

```

```
- (void)tick
{
    //convert time to hours, minutes and seconds
    NSCalendar *calendar = [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];
    NSUInteger units = NSHourCalendarUnit | NSMinuteCalendarUnit | NSSecondCalendarUnit;
    NSDateComponents *components = [calendar components:units fromDate:[NSDate date]];
    CGFloat hoursAngle = (components.hour / 12.0) * M_PI * 2.0;
    //calculate hour hand angle //calculate minute hand angle
    CGFloat minsAngle = (components.minute / 60.0) * M_PI * 2.0;
    //calculate second hand angle
    CGFloat secsAngle = (components.second / 60.0) * M_PI * 2.0;
    //rotate hands
    self.hourHand.transform = CGAffineTransformMakeRotation(hoursAngle);
    self.minuteHand.transform = CGAffineTransformMakeRotation(minsAngle);
    self.secondHand.transform = CGAffineTransformMakeRotation(secsAngle);
}

@end
```

运行项目，看起来有点奇怪（图3.6），因为钟表的图片在围绕着中心旋转，这并不是我们期待的一个支点。

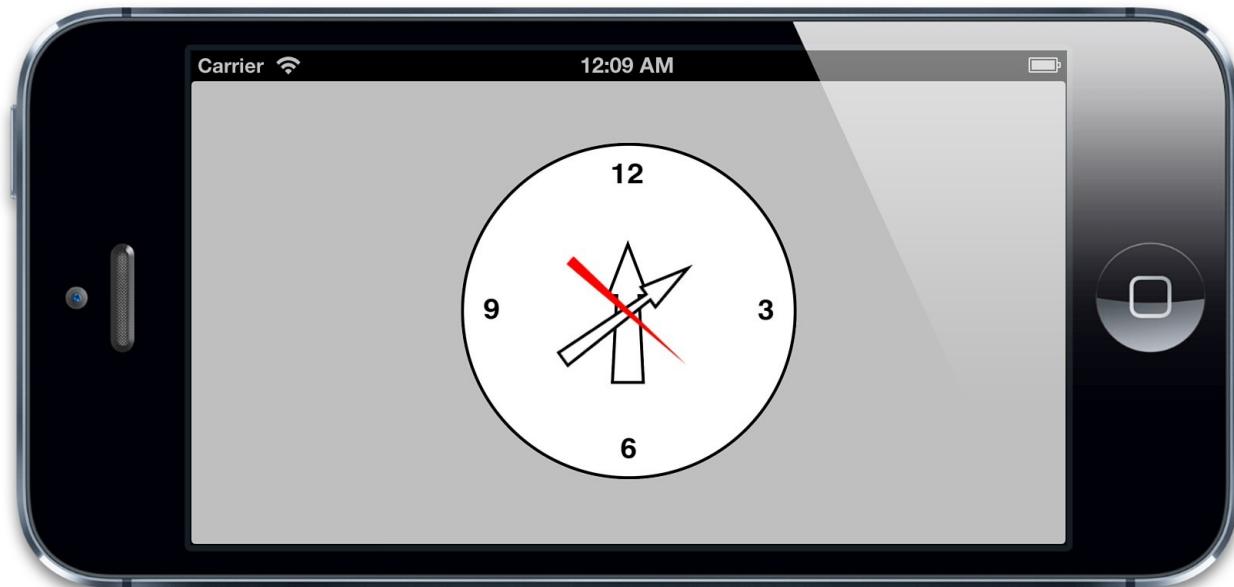


图3.6 钟面，和不对齐的钟指针

你也许会认为可以在Interface Builder当中调整指针图片的位置来解决，但其实并不能达到目的，因为如果不放在钟面中间的话，同样不能正确的旋转。

也许在图片末尾添加一个透明空间也是个解决方案，但这样会让图片变大，也会消耗更多的内存，这样并不优雅。

更好的方案是使用 `anchorPoint` 属性，我们来在 `-viewDidLoad` 方法中添加几行代码来给每个钟指针的 `anchorPoint` 做一些平移（清单3.2），图3.7显示了正确的结果。

清单3.2

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // adjust anchor points

    self.secondHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);
    self.minuteHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);
    self.hourHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);

    // start timer
}
```

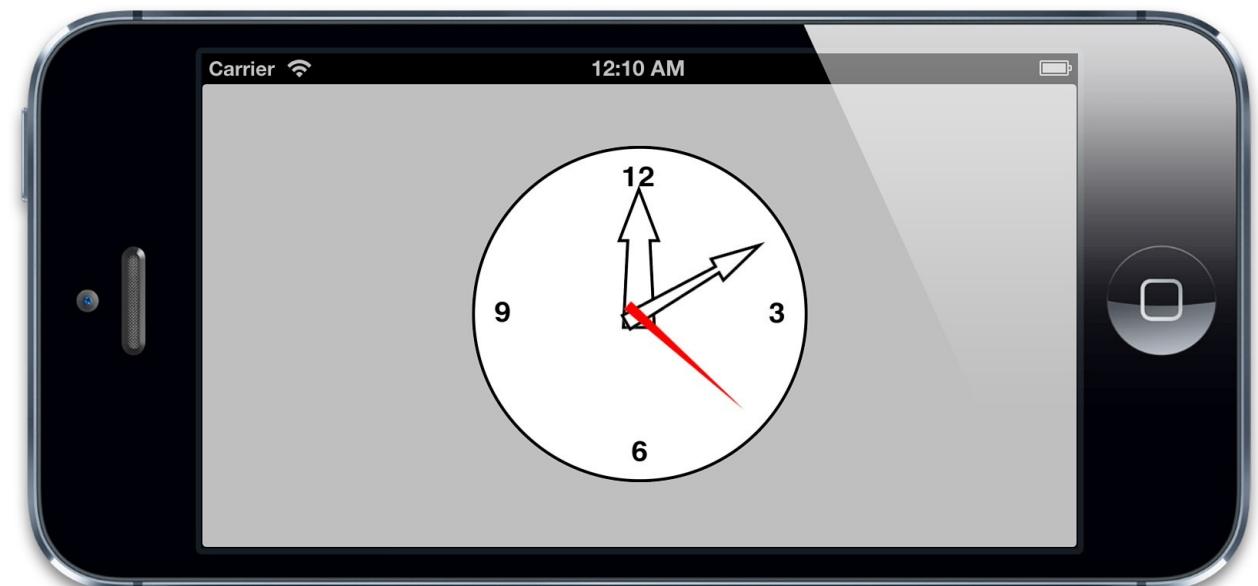


图3.7 钟面，和正确对齐的钟指针

坐标系

和视图一样，图层在图层树当中也是相对于父图层按层级关系放置，一个图层的 `position` 依赖于它父图层的 `bounds`，如果父图层发生了移动，它的所有子图层也会跟着移动。

这样对于放置图层会更加方便，因为你可以通过移动根图层来将它的子图层作为一个整体来移动，但是有时候你需要知道一个图层的绝对位置，或者是相对于另一个图层的位置，而不是它当前父图层的位置。

`CALayer` 给不同坐标系之间的图层转换提供了一些工具类方法：

- ```
- (CGPoint)convertPoint:(CGPoint)point fromLayer:(CALayer *)layer;
- (CGPoint)convertPoint:(CGPoint)point toLayer:(CALayer *)layer;
- (CGRect)convertRect:(CGRect)rect fromLayer:(CALayer *)layer;
- (CGRect)convertRect:(CGRect)rect toLayer:(CALayer *)layer;
```

这些方法可以把定义在一个图层坐标系下的点或者矩形转换成另一个图层坐标系下的点或者矩形。

## 翻转的几何结构

常规说来，在iOS上，一个图层的 `position` 位于父图层的左上角，但是在Mac OS上，通常是位于左下角。Core Animation可以通过 `geometryFlipped` 属性来适配这两种情况，它决定了一个图层的坐标是否相对于父图层垂直翻转，是一个 `BOOL` 类型。在iOS上通过设置它为 `YES` 意味着它的子图层将会被垂直翻转，也就是将会沿着底部排版而不是通常的顶部（它的所有子图层也同理，除非把它们的 `geometryFlipped` 属性也设为 `YES`）。

## Z坐标轴

和 `UIView` 严格的二维坐标系不同，`CALayer` 存在于一个三维空间当中。除了我们已经讨论过的 `position` 和 `anchorPoint` 属性之外，`CALayer` 还有另外两个属性，`zPosition` 和 `anchorPointZ`，二者都是在Z轴上描述图层位置的浮点

类型。

注意这里并没有更深的属性来描述由宽和高做成的 `bounds` 了，图层是一个完全扁平的对象，你可以把它们想象成类似于一页二维的坚硬的纸片，用胶水粘成一个空洞，就像三维结构的折纸一样。

`zPosition` 属性在大多数情况下其实并不常用。在第五章，我们将会涉及 `CATransform3D`，你会知道如何在三维空间移动和旋转图层，除了做变换之外，`zPosition` 最实用的功能就是改变图层的显示顺序了。

通常，图层是根据它们子图层的 `sublayers` 出现的顺序来类绘制的，这就是所谓的画家的算法--就像一个画家在墙上作画--后被绘制上的图层将会遮盖住之前的图层，但是通过增加图层的 `zPosition`，就可以把图层向相机方向前置，于是它就在所有其他图层的前面了（或者至少是小于它的 `zPosition` 值的图层的前面）。

这里所谓的“相机”实际上是相对于用户是视角，这里和iPhone背后的内置相机没有任何关系。

图3.8显示了在Interface Builder内的一对视图，正如你所见，首先出现在视图层级绿色的视图被绘制在红色视图的后面。

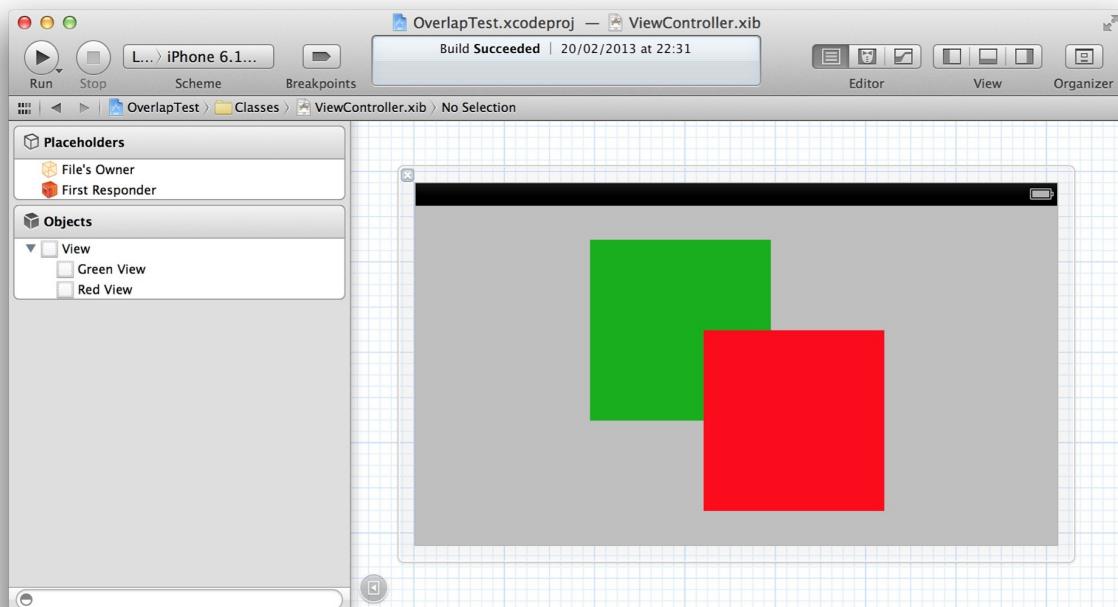


图3.8 在视图层级中绿色视图被绘制在红色视图的后面

我们希望在真实的应用中也能显示出绘图的顺序，同样地，如果我们提高绿色视图的 `zPosition`（清单3.3），我们会发现顺序就反了（图3.9）。其实并不需要增加太多，视图都非常地薄，所以给 `zPosition` 提高一个像素就可以让绿色视图前置，当然0.1或者0.0001也能够做到，但是最好不要这样，因为浮点类型四舍五入的计算可能会造成一些不便的麻烦。

### 清单3.3

```
@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *greenView;
@property (nonatomic, weak) IBOutlet UIView *redView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //move the green view zPosition nearer to the camera
 self.greenView.layer.zPosition = 1.0f;
}
@end
```

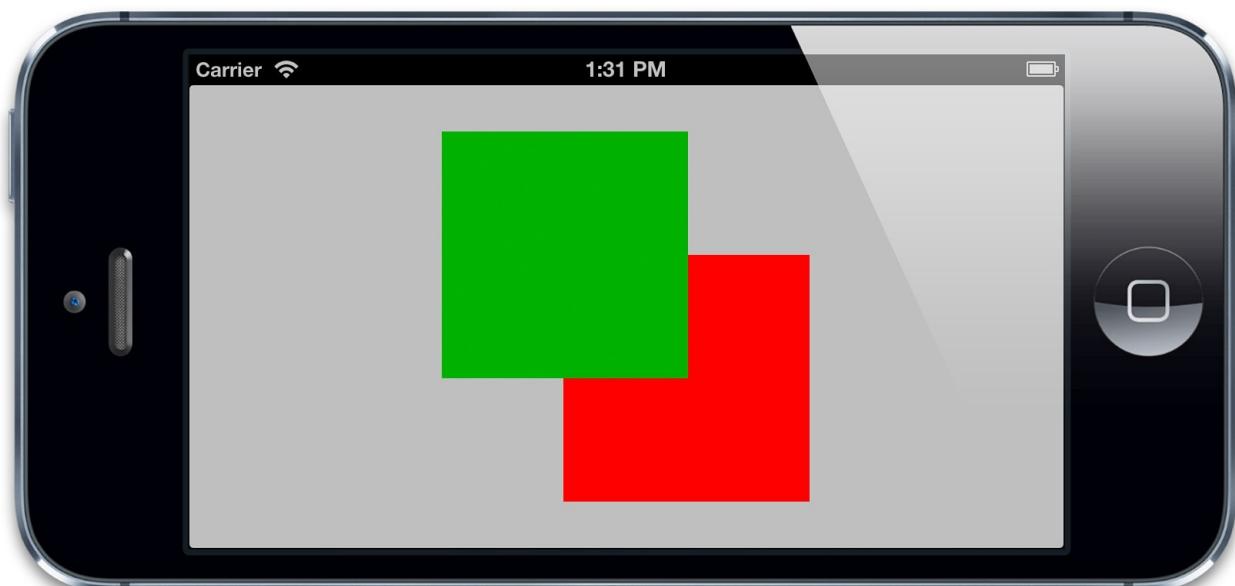


图3.9 绿色视图被绘制在红色视图的前面

# Hit Testing

第一章“图层树”证实了最好使用图层相关视图，而不是创建独立的图层关系。其中一个原因就是要处理额外复杂的触摸事件。

`CALayer` 并不关心任何响应链事件，所以不能直接处理触摸事件或者手势。但是它有一系列的方法帮你处理事件：`-containsPoint:` 和 `-hitTest:`。

`-containsPoint:` 接受一个在本图层坐标系下的 `CGPoint`，如果这个点在图层 `frame` 范围内就返回 `YES`。如清单3.4所示第一章的项目的另一个合适的版本，也就是使用 `-containsPoint:` 方法来判断到底是白色还是蓝色的图层被触摸了（图3.10）。这需要把触摸坐标转换成每个图层坐标系下的坐标，结果很不方便。

清单3.4 使用`containsPoint`判断被点击的图层

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;
@property (nonatomic, weak) CALayer *blueLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create sublayer
 self.blueLayer = [CALayer layer];
 self.blueLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
 self.blueLayer.backgroundColor = [UIColor blueColor].CGColor;
 //add it to our view
 [self.layerView.layer addSublayer:self.blueLayer];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
```

```
//get touch position relative to main view
CGPoint point = [[touches anyObject] locationInView:self.view];
//convert point to the white layer's coordinates
point = [self.layerView.layer convertPoint:point fromLayer:self.view];
//get layer using containsPoint:
if ([self.layerView.layer containsPoint:point]) {
 //convert point to blueLayer's coordinates
 point = [self.blueLayer convertPoint:point fromLayer:self.view];
 if ([self.blueLayer containsPoint:point]) {
 [[[UIAlertView alloc] initWithTitle:@"Inside Blue Layer"
 message:nil
 delegate:nil
 cancelButtonTitle:@"OK"
 otherButtonTitles:nil] show];
 } else {
 [[[UIAlertView alloc] initWithTitle:@"Inside White Layer"
 message:nil
 delegate:nil
 cancelButtonTitle:@"OK"
 otherButtonTitles:nil] show];
 }
}
}

@end
```

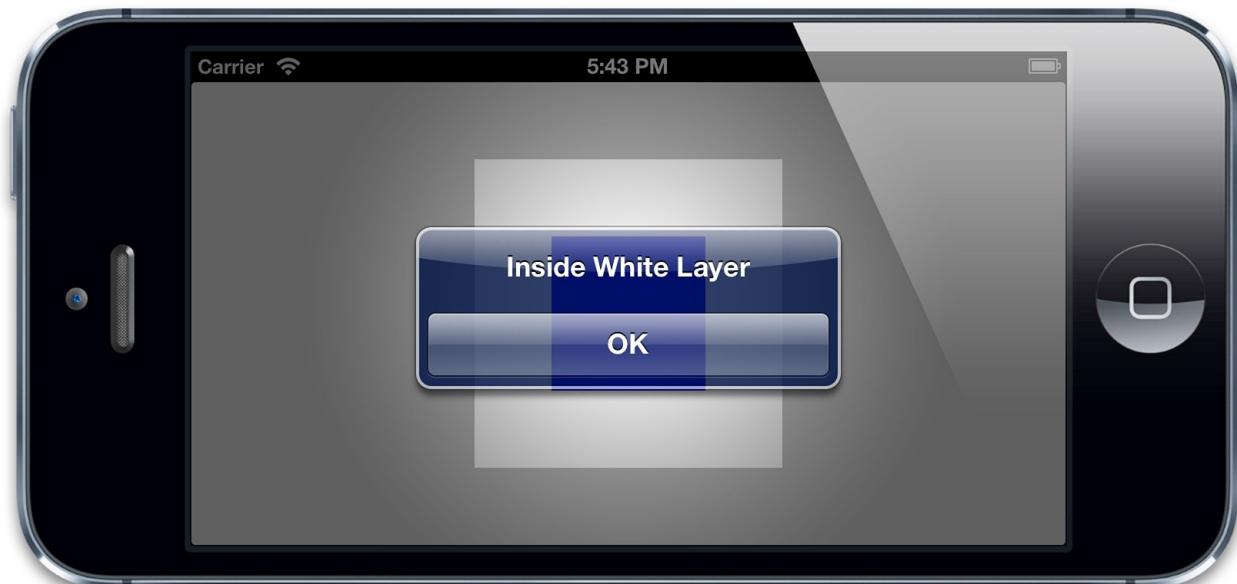


图3.10 点击图层被正确标识

`-hitTest:` 方法同样接受一个 `CGPoint` 类型参数，而不是 `BOOL` 类型，它返回图层本身，或者包含这个坐标点的叶子节点图层。这意味着不再需要像使用 `-containsPoint:` 那样，人工地在每个子图层变换或者测试点击的坐标。如果这个点在最外面图层的范围之外，则返回`nil`。具体使用 `-hitTest:` 方法被点击图层的代码如清单3.5所示。

清单3.5 使用`hitTest`判断被点击的图层

```

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
 //get touch position
 CGPoint point = [[touches anyObject] locationInView:self.view];
 //get touched layer
 CALayer *layer = [self.layerView.layer hitTest:point];
 //get layer using hitTest
 if (layer == self.blueLayer) {
 [[[UIAlertView alloc] initWithTitle:@"Inside Blue Layer"
 message:nil
 delegate:nil
 cancelButtonTitle:@"OK"
 otherButtonTitles:nil] show];
 } else if (layer == self.layerView.layer) {
 [[[UIAlertView alloc] initWithTitle:@"Inside White Layer"
 message:nil
 delegate:nil
 cancelButtonTitle:@"OK"
 otherButtonTitles:nil] show];
 }
}

```

注意当调用图层的 `-hitTest:` 方法时，测算的顺序严格依赖于图层树当中的图层顺序（和`UIView`处理事件类似）。之前提到的 `zPosition` 属性可以明显改变屏幕上图层的顺序，但不能改变事件传递的顺序。

这意味着如果改变了图层的`z`轴顺序，你会发现将不能够检测到最前方的视图点击事件，这是因为被另一个图层遮盖住了，虽然它的 `zPosition` 值较小，但是在图层树中的顺序靠前。我们将在第五章详细讨论这个问题。



# 自动布局

你可能用过 `UIViewAutoresizingMask` 类型的一些常量，应用于当父视图改变尺寸的时候，相应 `UIView` 的 `frame` 也跟着更新的场景（通常用于横竖屏切换）。

在iOS6中，苹果介绍了自动排版机制，它和自动调整不同，并且更加复杂。

在Mac OS平台，`CALayer` 有一个叫做 `layoutManager` 的属性可以通过 `CALayoutManager` 协议和 `CAConstraintLayoutManager` 类来实现自动排版的机制。但由于某些原因，这在iOS上并不适用。

当使用视图的时候，可以充分利用 `UIView` 类接口暴露出来的 `UIViewAutoresizingMask` 和 `NSLayoutConstraint API`，但如果想随意控制 `CALayer` 的布局，就需要手工操作。最简单的方法就是使用 `CALayerDelegate` 如下函数：

```
- (void)layoutSublayersOfLayer:(CALayer *)layer;
```

当图层的 `bounds` 发生改变，或者图层的 `-setNeedsLayout` 方法被调用的时候，这个函数将会被执行。这使得你可以手动地重新摆放或者重新调整子图层的大小，但是不能像 `UIView` 的 `autoresizingMask` 和 `constraints` 属性做到自适应屏幕旋转。

这也是为什么最好使用视图而不是单独的图层来构建应用程序的另一个重要原因之一。

## 总结

本章涉及了 `CALayer` 的集合结构，包括它的 `frame`，`position` 和 `bounds`，介绍了三维空间内图层的概念，以及如何在独立的图层内响应事件，最后简单说明了在iOS平台，Core Animation对自动调整和自动布局支持的缺乏。

在第四章“视觉效果”当中，我们接着介绍一些图层外表的特性。

## 视觉效果

嗯，圆和椭圆还不错，但如果是带圆角的矩形呢？

我们现在能做到那样了么？

史蒂芬·乔布斯

我们在第三章『图层几何学』中讨论了图层的frame，第二章『寄宿图』则讨论了图层不仅仅可以是图片或是颜色的容器；还有一系列内建的特性使得创造美丽优雅的令人深刻的界面元素成为可能。在这一章，我们将会探索一些能够通过使用CALayer属性实现的视觉效果。

# 圆角

圆角矩形是iOS的一个标志性审美特性。这在iOS的每一个地方都得到了体现，不论是主屏幕图标，还是警告弹框，甚至是文本框。按照这流行程度，你可能会认为一定有不借助Photoshop就能轻易创建圆角举行的的方法。恭喜你，猜对了。

CALayer有一个叫做 `cornerRadius` 的属性控制着图层角的曲率。它是一个浮点数，默认为0（为0的时候就是直角），但是你可以把它设置成任意值。默认情况下，这个曲率值只影响背景颜色而不影响背景图片或是子图层。不过，如果把 `masksToBounds` 设置成YES的话，图层里面的所有东西都会被截取。

我们可以通过一个简单的项目来演示这个效果。在Interface Builder中，我们放置一些视图，他们有一些子视图。而且这些子视图有一些超出了边界（如图4.1）。你可能无法看到他们超出了边界，因为在编辑界面的时候，超出的部分总是被Interface Builder裁切掉了。不过，你相信我就好了：）

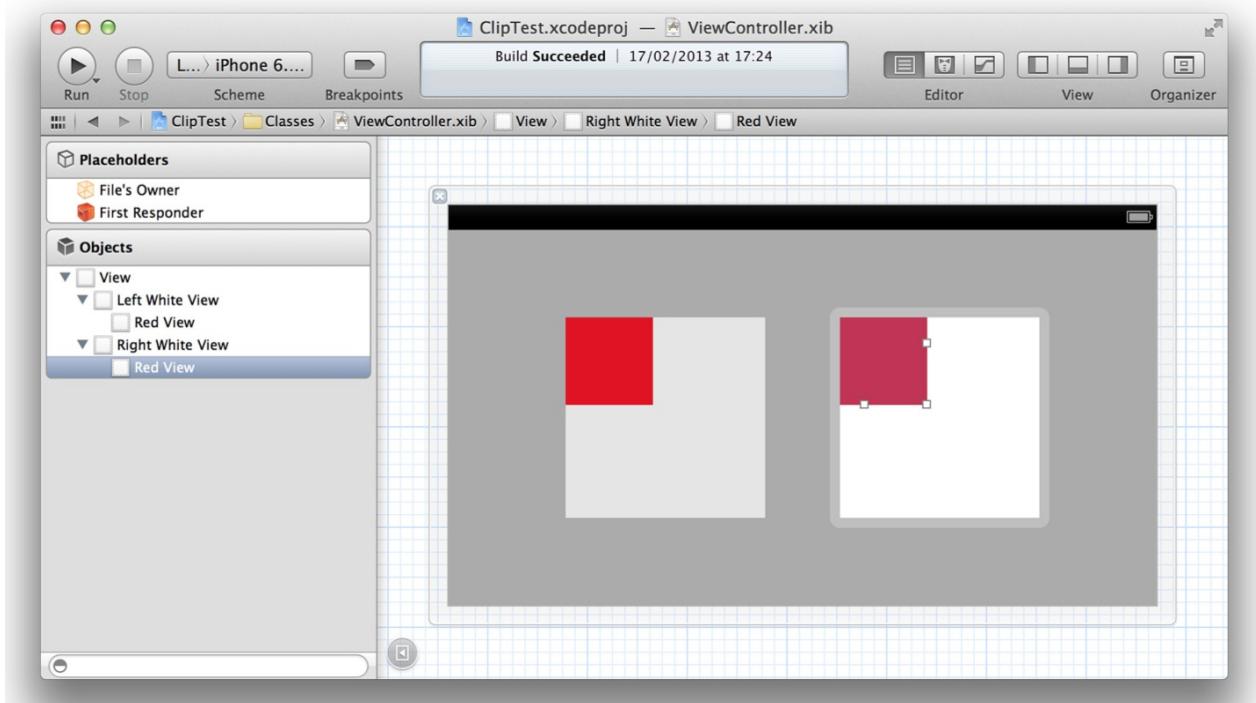


图4.1 两个白色的大视图，他们都包含了小一些的红色视图。

然后在代码中，我们设置角的半径为20个点，并裁剪掉第一个视图的超出部分（见清单4.1）。技术上来说，这些属性都可以在Interface Builder的探测板中分别通过『用户定义运行时属性』和勾选『裁剪子视图』(Clip Subviews)选择框来直接

设置属性的值。不过，在这个示例中，代码能够表示得更清楚。图4.2是运行代码的结果

#### 清单4.1 设置 `cornerRadius` 和 `masksToBounds`

```
@interface ViewController ()

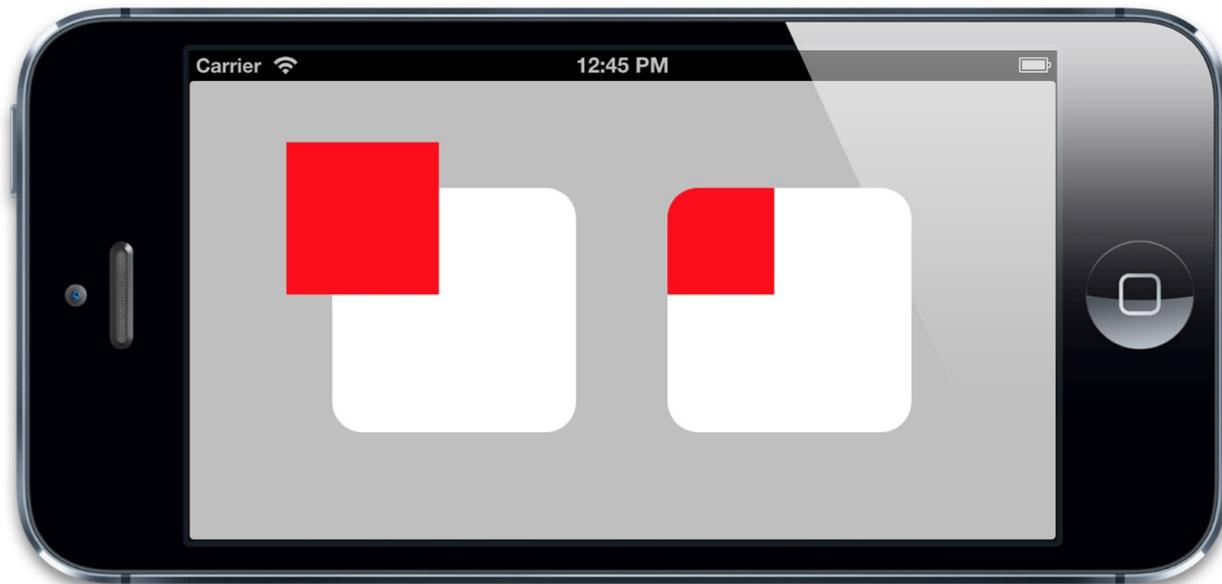
@property (nonatomic, weak) IBOutlet UIView *layerView1;
@property (nonatomic, weak) IBOutlet UIView *layerView2;

@end

@implementation ViewController
- (void)viewDidLoad
{
 [super viewDidLoad];

 //set the corner radius on our layers
 self.layerView1.layer.cornerRadius = 20.0f;
 self.layerView2.layer.cornerRadius = 20.0f;

 //enable clipping on the second layer
 self.layerView2.layer.masksToBounds = YES;
}
@end
```



右图中，红色的子视图沿角半径被裁剪了  
如你所见，右边的子视图沿边界被裁剪了。

单独控制每个层的圆角曲率也不是不可能的。如果想创建有些圆角有些直角的图层或视图时，你可能需要一些不同的方法。比如使用一个图层蒙板（本章稍后会讲到）或者是CAShapeLayer（见第六章『专用图层』）。

## 图层边框

&nbsp; CALayer 另外两个非常有用属性就是 `borderWidth` 和 `borderColor` 。二者共同定义了图层边的绘制样式。这条线（也被称作 `stroke`）沿着图层的 `bounds` 绘制，同时也包含图层的角。

&nbsp; `borderWidth` 是以点为单位的定义边框粗细的浮点数，默认为 0. `borderColor` 定义了边框的颜色，默认为黑色。

&nbsp; `borderColor` 是 `CGColorRef` 类型，而不是 `UIColor`，所以它不是 Cocoa 的内置对象。不过呢，你肯定也清楚图层引用了 `borderColor`，虽然属性声明并不能证明这一点。`CGColorRef` 在引用/释放时候的行为表现得与 `NSObject` 极其相似。但是 Objective-C 语法并不支持这一做法，所以 `CGColorRef` 属性即便是强引用也只能通过 `assign` 关键字来声明。

&nbsp; 边框是绘制在图层边界里面的，而且在所有子内容之前，也在子图层之前。如果我们在之前的示例中（清单 4.2）加入图层的边框，你就能看到到底是怎么一回事了（如图 4.3）。

### 清单 4.2 加上边框

```
@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //set the corner radius on our layers
 self.layerView1.layer.cornerRadius = 20.0f;
 self.layerView2.layer.cornerRadius = 20.0f;

 //add a border to our layers
 self.layerView1.layer.borderWidth = 5.0f;
 self.layerView2.layer.borderWidth = 5.0f;

 //enable clipping on the second layer
 self.layerView2.layer.masksToBounds = YES;
}

@end
```

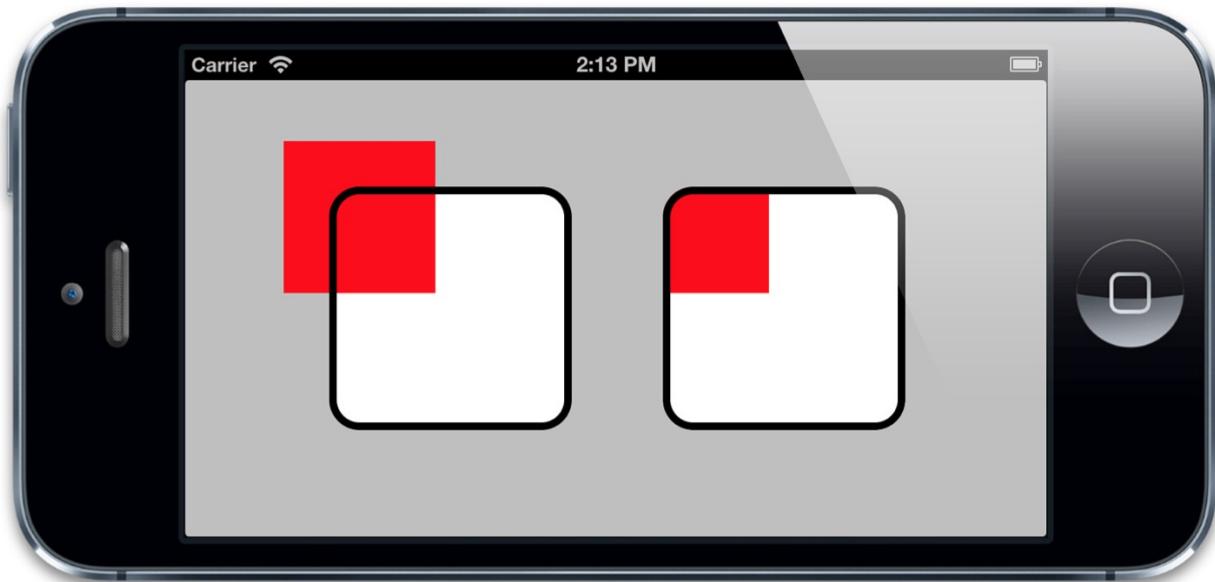


图4.3 给图层增加一个边框

&nbsp;仔细观察会发现边框并不会把寄宿图或子图层的形状计算进来，如果图层的子图层超过了边界，或者是寄宿图在透明区域有一个透明蒙板，边框仍然会沿着图层的边界绘制出来（如图4.4）。

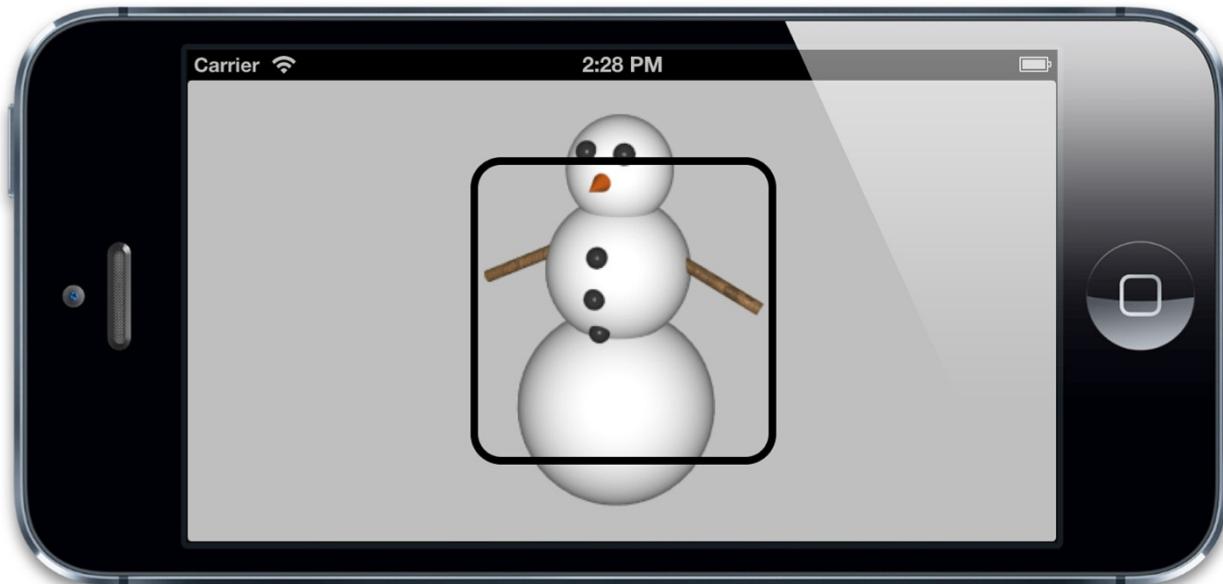


图4.4 边框是跟随图层的边界变化的，而不是图层里面的内容

## 阴影

iOS的另一个常见特性呢，就是阴影。阴影往往可以达到图层深度暗示的效果。也能够用来强调正在显示的图层和优先级（比如说一个在其他视图之前的弹出框），不过有时候他们只是单纯的装饰目的。

给 `shadowOpacity` 属性一个大于默认值（也就是0）的值，阴影就可以显示在任意图层之下。`shadowOpacity` 是一个必须在0.0（不可见）和1.0（完全不透明）之间的浮点数。如果设置为1.0，将会显示一个有轻微模糊的黑色阴影稍微在图层之上。若要改动阴影的表现，你可以使用CALayer的另外三个属性：`shadowColor`，`shadowOffset` 和 `shadowRadius`。

显而易见，`shadowColor` 属性控制着阴影的颜色，和 `borderColor` 和 `backgroundColor` 一样，它的类型也是 `CGColorRef`。阴影默认是黑色，大多数时候你需要的阴影也是黑色的（其他颜色的阴影看起来是不是有一点点奇怪。。。）。

`shadowOffset` 属性控制着阴影的方向和距离。它是一个 `CGSize` 的值，宽度控制这阴影横向的位移，高度控制着纵向的位移。`shadowOffset` 的默认值是 {0, -3}，意即阴影相对于Y轴有3个点的向上位移。

为什么要默认向上的阴影呢？尽管Core Animation是从图层套装演变而来（可以认为是为iOS创建的私有动画框架），但是呢，它却是在Mac OS上面世的，前面有提到，二者的Y轴是颠倒的。这就导致了默认的3个点位移的阴影是向上的。在Mac上，`shadowOffset` 的默认值是阴影向下的，这样你就能理解为什么iOS上的阴影方向是向上的了（如图4.5）。

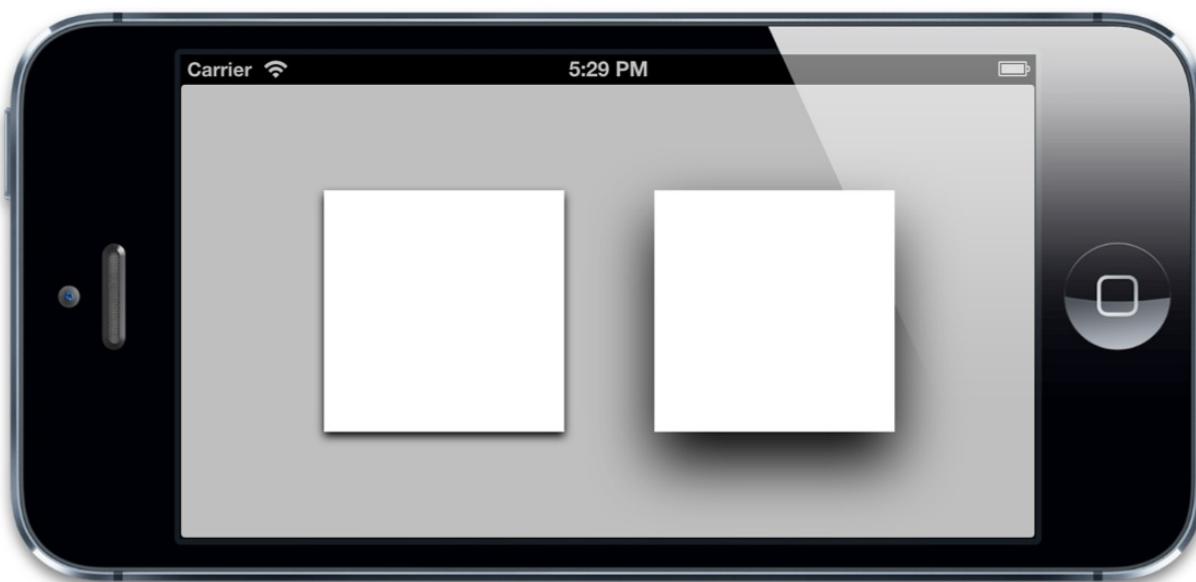


图4.5 在iOS（左）和Mac OS（右）上 `shadowOffset` 的表现。

苹果更倾向于用户界面的阴影应该是垂直向下的，所以在iOS把阴影宽度设为0，然后高度设为一个正值不失为一个做法。

`shadowRadius` 属性控制着阴影的模糊度，当它的值是0的时候，阴影就和视图一样有一个非常确定的边界线。当值越来越大的时候，边界线看上去就会越来越模糊和自然。苹果自家的应用设计更偏向于自然的阴影，所以一个非零值再合适不过了。

通常来讲，如果你想让视图或控件非常醒目独立于背景之外（比如弹出框遮罩层），你就应该给 `shadowRadius` 设置一个稍大的值。阴影越模糊，图层的深度看上去就会更明显（如图4.6）。



## 阴影裁剪

和图层边框不同，图层的阴影继承自内容的外形，而不是根据边界和角半径来确定。为了计算出阴影的形状，Core Animation会将寄宿图（包括子视图，如果有的话）考虑在内，然后通过这些来完美搭配图层形状从而创建一个阴影（见图4.7）。



图4.7 阴影是根据寄宿图的轮廓来确定的

当阴影和裁剪扯上关系的时候就有一个头疼的限制：阴影通常就是在 Layer 的边界之外，如果你开启了 `masksToBounds` 属性，所有从图层中突出的内容都会被才剪掉。如果我们在我们之前的边框示例项目中增加图层的阴影属性时，你就会发现问题所在（见图4.8）。

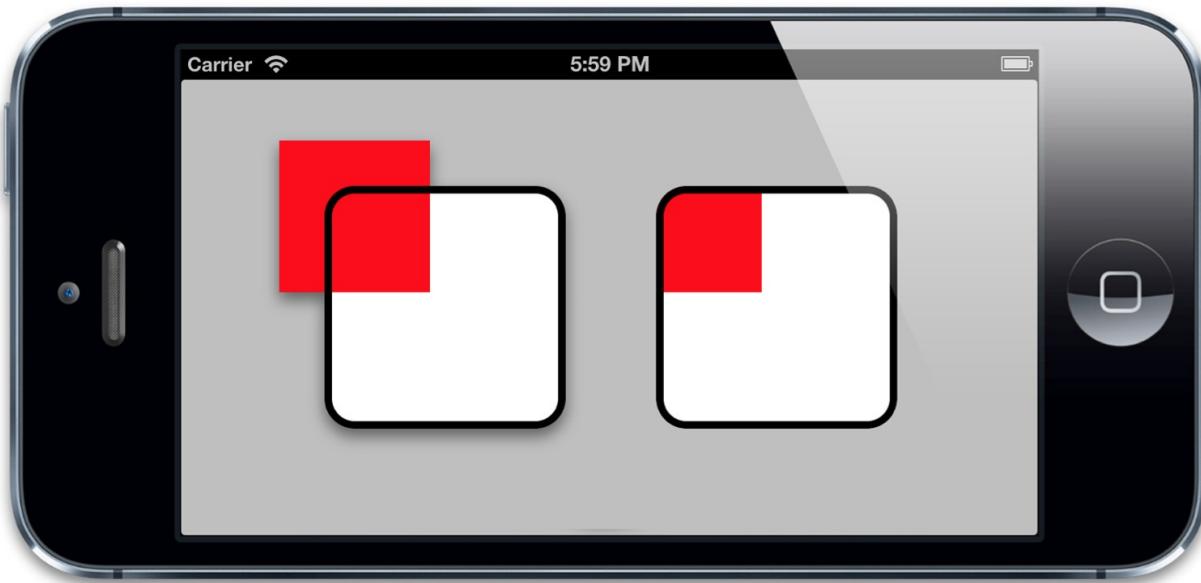


图4.8 `masksToBounds` 属性裁剪掉了阴影和内容

从技术角度来说，这个结果是可以是可以理解的，但确实又不是我们想要的效果。如果你想沿着内容裁切，你需要用到两个图层：一个只画阴影的空的外图层，和一个用 `masksToBounds` 裁剪内容的内图层。

如果我们把之前项目的右边用单独的视图把裁剪的视图包起来，我们就可以解决这个问题（如图4.9）。

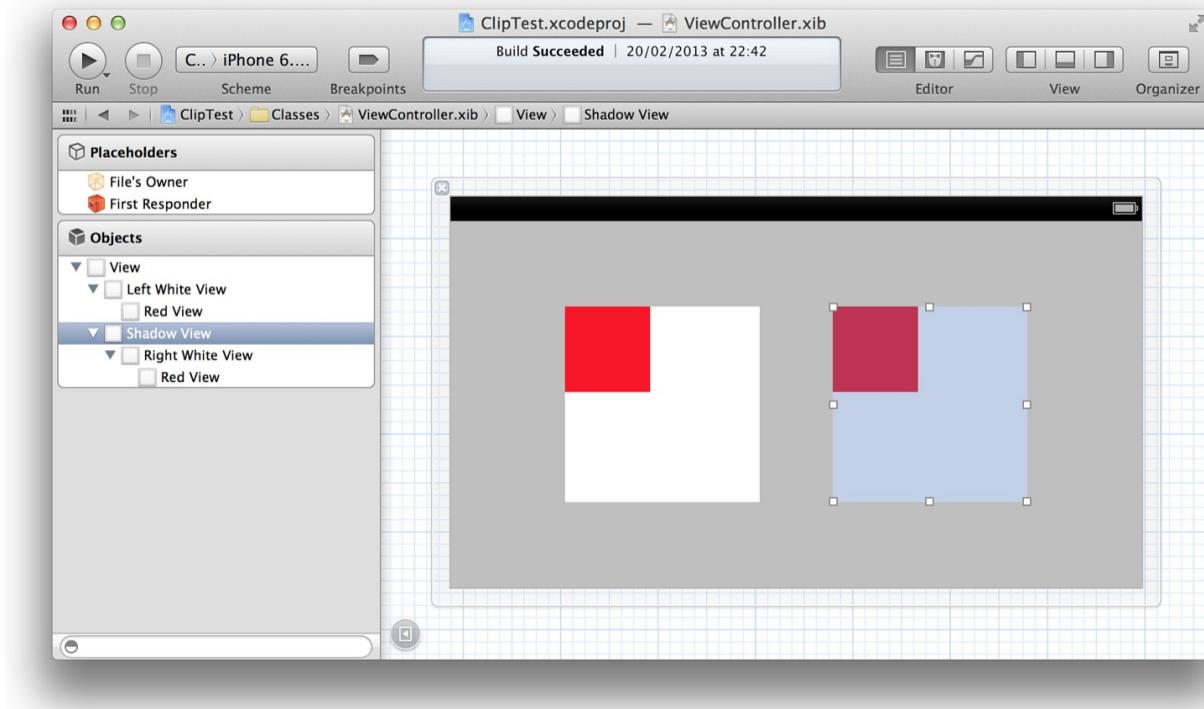


图4.9 右边，用额外的阴影转换视图包裹被裁剪的视图

我们只把阴影用在最外层的视图上，内层视图进行裁剪。清单4.3是代码实现，图4.10是运行结果。

清单4.3 用一个额外的视图来解决阴影裁切的问题

```
@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *layerView1;
@property (nonatomic, weak) IBOutlet UIView *layerView2;
@property (nonatomic, weak) IBOutlet UIView *shadowView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //set the corner radius on our layers
 self.layerView1.layer.cornerRadius = 20.0f;
 self.layerView2.layer.cornerRadius = 20.0f;

 //add a border to our layers
 self.layerView1.layer.borderWidth = 5.0f;
 self.layerView2.layer.borderWidth = 5.0f;

 //add a shadow to layerView1
 self.layerView1.layer.shadowOpacity = 0.5f;
 self.layerView1.layer.shadowOffset = CGSizeMake(0.0f, 5.0f);
 self.layerView1.layer.shadowRadius = 5.0f;

 //add same shadow to shadowView (not layerView2)
 self.shadowView.layer.shadowOpacity = 0.5f;
 self.shadowView.layer.shadowOffset = CGSizeMake(0.0f, 5.0f);
 self.shadowView.layer.shadowRadius = 5.0f;

 //enable clipping on the second layer
 self.layerView2.layer.masksToBounds = YES;
}

@end
```

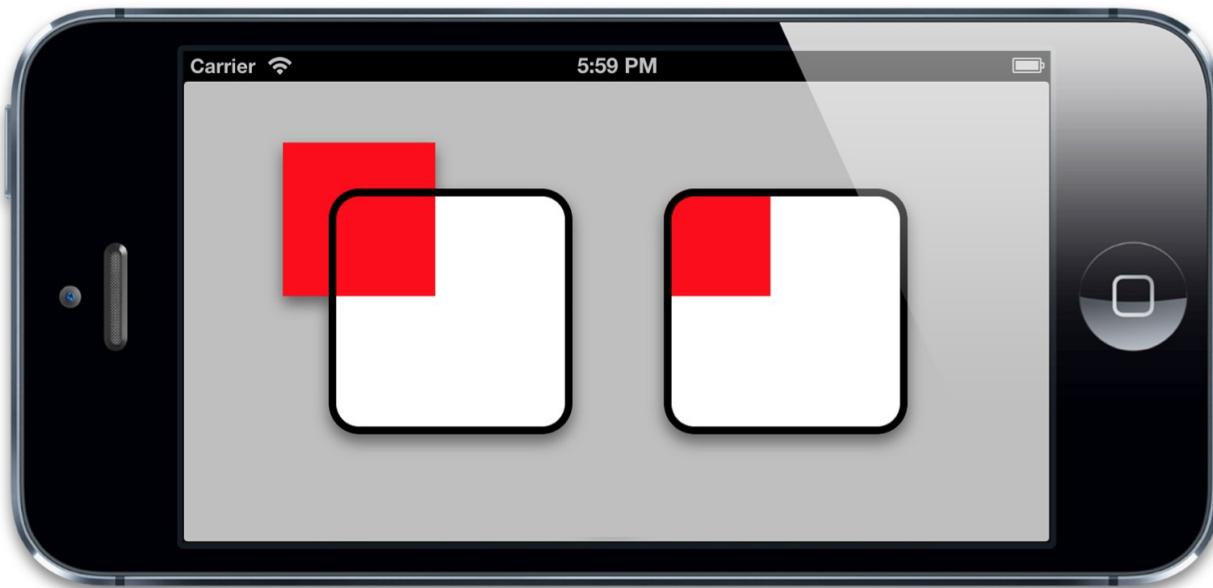


图4.10 右边视图，不受裁切阴影的阴影视图。

## shadowPath属性

我们已经知道图层阴影并不总是方的，而是从图层内容的形状继承而来。这看上去不错，但是实时计算阴影也是一个非常消耗资源的，尤其是图层有多个子图层，每个图层还有一个有透明效果的寄宿图的时候。

如果你事先知道你的阴影形状会是什么样子的，你可以通过指定一个 `shadowPath` 来提高性能。`shadowPath` 是一个 `CGPathRef` 类型（一个指向 `CGPath` 的指针）。`CGPath` 是一个Core Graphics对象，用来指定任意的一个矢量图形。我们可以通过这个属性单独于图层形状之外指定阴影的形状。

图4.11 展示了同一寄宿图的不同阴影设定。如你所见，我们使用的图形很简单，但是它的阴影可以是你想要的任何形状。清单4.4是代码实现。

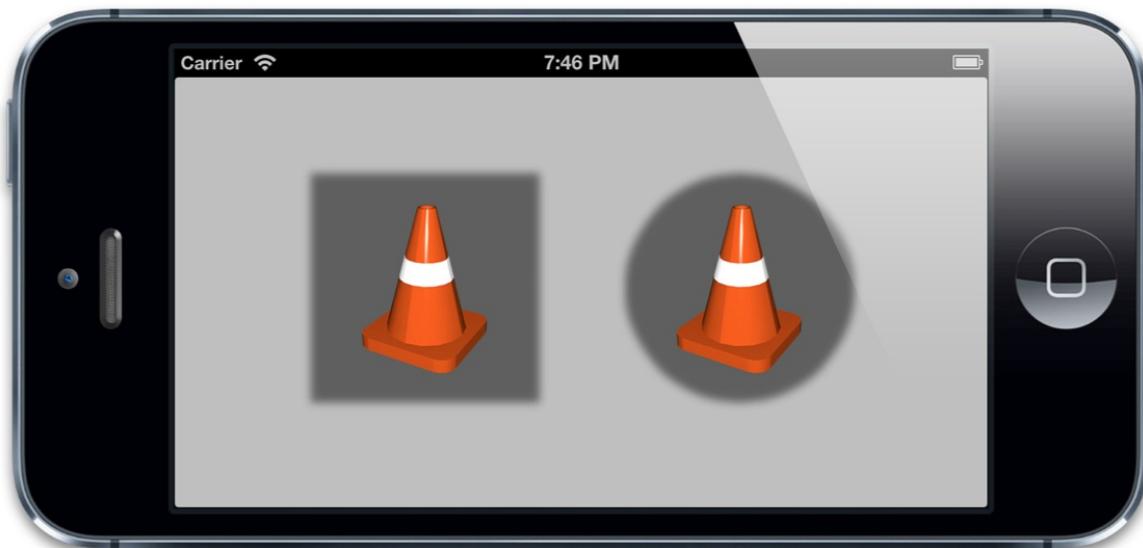


图4.11 用 `shadowPath` 指定任意阴影形状

清单4.4 创建简单的阴影形状

```
@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *layerView1;
@property (nonatomic, weak) IBOutlet UIView *layerView2;
@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //enable layer shadows
 self.layerView1.layer.shadowOpacity = 0.5f;
 self.layerView2.layer.shadowOpacity = 0.5f;

 //create a square shadow
 CGMutablePathRef squarePath = CGPathCreateMutable();
 CGPathAddRect(squarePath, NULL, self.layerView1.bounds);
 self.layerView1.layer.shadowPath = squarePath; CGPathRelease(squa

 //create a circular shadow
 CGMutablePathRef circlePath = CGPathCreateMutable();
 CGPathAddEllipseInRect(circlePath, NULL, self.layerView2.bounds);
 self.layerView2.layer.shadowPath = circlePath; CGPathRelease(circ
}
@end
```

如果是一个矩形或者是圆，用 `CGPath` 会相当简单明了。但是如果是更加复杂一点的图形，`UIBezierPath` 类会更合适，它是一个由UIKit提供的在`CGPath`基础上的Objective-C包装类。

图4.6 大一些的阴影位移和角半径会增加图层的深度即视感

## 图层蒙板

通过 `masksToBounds` 属性，我们可以沿边界裁剪图形；通过 `cornerRadius` 属性，我们还可以设定一个圆角。但是有时候你希望展现的内容不是在一个矩形或圆角矩形。比如，你想展示一个有星形框架的图片，又或者想让一些古卷文字慢慢渐变成背景色，而不是一个突兀的边界。

使用一个32位有alpha通道的png图片通常是创建一个无矩形视图最方便的方法，你可以给它指定一个透明蒙板来实现。但是这个方法不能让你以编码的方式动态地生成蒙板，也不能让子图层或子视图裁剪成同样的形状。

`CALayer`有一个属性叫做 `mask` 可以解决这个问题。这个属性本身就是个 `CALayer`类型，有和其他图层一样的绘制和布局属性。它类似于一个子图层，相对于父图层（即拥有该属性的图层）布局，但是它却不是一个普通的子图层。不同于那些绘制在父图层中的子图层，`mask` 图层定义了父图层的部分可见区域。

`mask` 图层的 `color` 属性是无关紧要的，真正重要的是图层的轮廓。`mask` 属性就像是一个饼干切割机，`mask` 图层实心的部分会被保留下来，其他的则会被抛弃。（如图4.12）

如果 `mask` 图层比父图层要小，只有在 `mask` 图层里面的内容才是它关心的，除此以外的一切都会被隐藏起来。

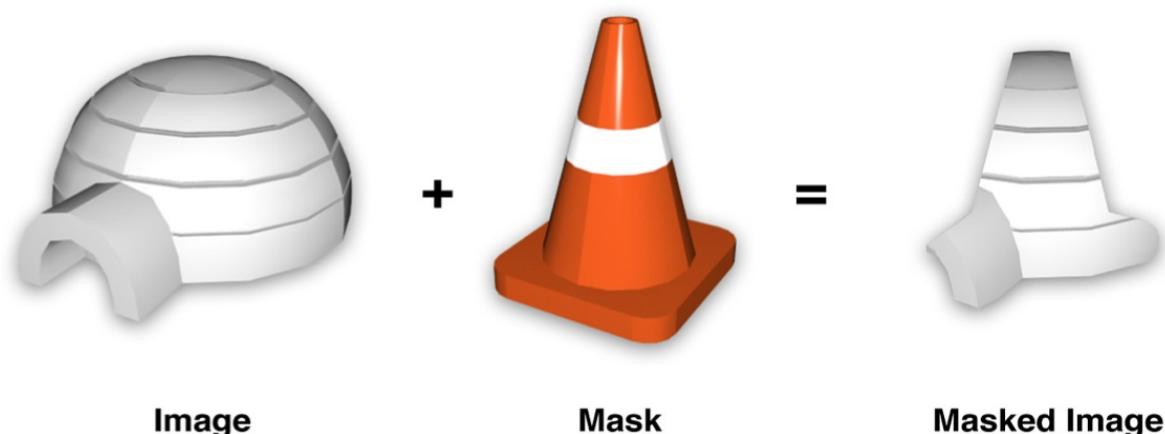


图4.12 把图片和蒙板图层作用在一起的效果

我们将代码演示一下这个过程，创建一个简单的项目，通过图层的 `mask` 属性来作用于图片之上。为了简便一些，我们用Interface Builder来创建一个包含 `UIImageView` 的图片图层。这样我们就只要代码实现蒙板图层了。清单4.5是最终的代码，图4.13是运行后的结果。

#### 清单4.5 应用蒙板图层

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIImageView *imageView;
@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //create mask layer
 CALayer *maskLayer = [CALayer layer];
 maskLayer.frame = self.layerView.bounds;
 UIImage *maskImage = [UIImage imageNamed:@"Cone.png"];
 maskLayer.contents = (__bridge id)maskImage.CGImage;

 //apply mask to image layer
 self.imageView.layer.mask = maskLayer;
}

@end
```



图4.13 使用了 `mask` 之后的UIImageView

CALayer蒙板图层真正厉害的地方在于蒙板图不局限于静态图。任何有图层构成的都可以作为 `mask` 属性，这意味着你的蒙板可以通过代码甚至是动画实时生成。

# 拉伸过滤

最后我们再来谈谈 `minificationFilter` 和 `magnificationFilter` 属性。总得来讲，当我们视图显示一个图片的时候，都应该正确地显示这个图片（意即：以正确的比例和正确的1：1像素显示在屏幕上）。原因如下：

- 能够显示最好的画质，像素既没有被压缩也没有被拉伸。
- 能更好的使用内存，因为这就是所有你要存储的东西。
- 最好的性能表现，CPU不需要为此额外的计算。

不过有时候，显示一个非真实大小的图片确实是需要的效果。比如说一个头像或是图片的缩略图，再比如说一个可以被拖拽和伸缩的大图。这些情况下，为同一图片的不同大小存储不同的图片显得又不切实际。

当图片需要显示不同的大小的时候，有一种叫做拉伸过滤的算法就起到作用了。它作用于原图的像素上并根据需要生成新的像素显示在屏幕上。

事实上，重绘图片大小也没有一个统一的通用算法。这取决于需要拉伸的内容，放大或是缩小的需求等这些因素。`CALayer` 为此提供了三种拉伸过滤方法，他们是：

- `kCAFilterLinear`
- `kCAFilterNearest`
- `kCAFilterTrilinear`

`minification`（缩小图片）和`magnification`（放大图片）默认的过滤器都是 `kCAFilterLinear`，这个过滤器采用双线性滤波算法，它在大多数情况下都表现良好。双线性滤波算法通过对多个像素取样最终生成新的值，得到一个平滑的表现不错的拉伸。但是当放大倍数比较大的时候图片就模糊不清了。

`kCAFilterTrilinear` 和 `kCAFilterLinear` 非常相似，大部分情况下二者都看不出来有什么差别。但是，较双线性滤波算法而言，三线性滤波算法存储了多个大小情况下的图片（也叫多重贴图），并三维取样，同时结合大图和小图的存储进而得到最后的结果。

这个方法的好处在于算法能够从一系列已经接近于最终大小的图片中得到想要的结果，也就是说不要对很多像素同步取样。这不仅提高了性能，也避免了小概率因舍入错误引起的取样失灵的问题。

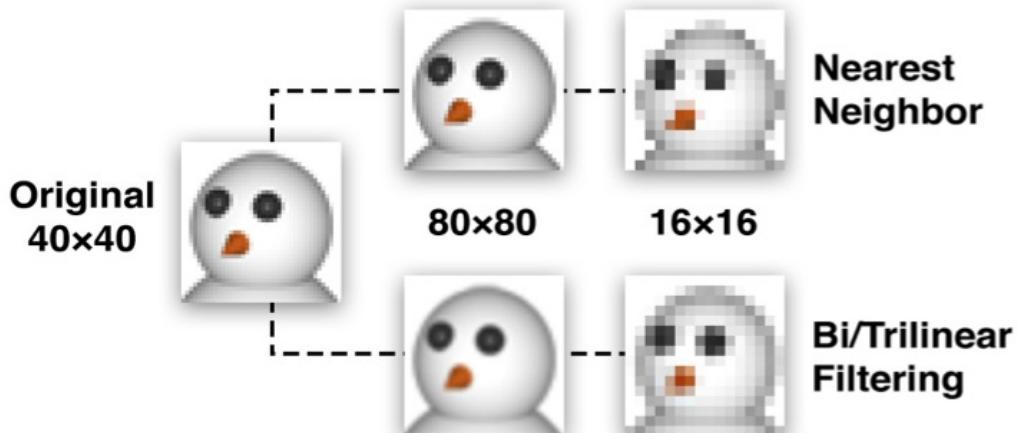


图4.14 对于大图来说，双线性滤波和三线性滤波表现得更出色

`KCAFilterNearest` 是一种比较武断的方法。从名字不难看出，这个算法（也叫最近过滤）就是取样最近的单像素点而不管其他的颜色。这样做非常快，也不会使图片模糊。但是，最明显的效果就是，会使得压缩图片更糟，图片放大之后也显得块状或是马赛克严重。

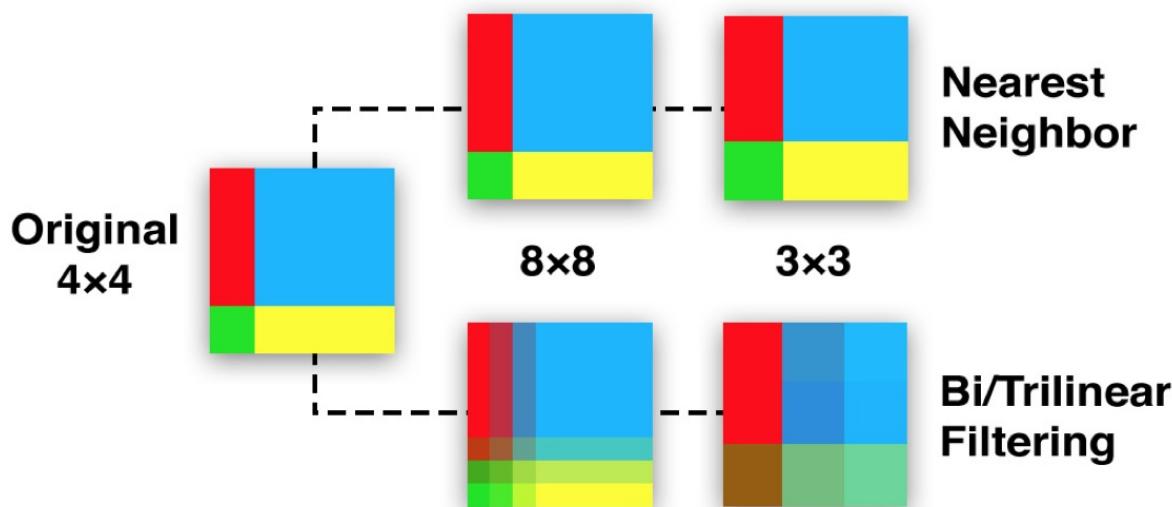


图4.15 对于没有斜线的小图来说，最近过滤算法要好很多

总的来说，对于比较小的图或者是差异特别明显，极少斜线的大图，最近过滤算法会保留这种差异明显的特质以呈现更好的结果。但是对于大多数的图尤其是有很多斜线或是曲线轮廓的图片来说，最近过滤算法会导致更差的结果。换句话说，线性过滤保留了形状，最近过滤则保留了像素的差异。

让我们来实验一下。我们对第三章的时钟项目改动一下，用LCD风格的数字方式显示。我们用简单的像素字体（一种用像素构成字符的字体，而非矢量图形）创造数字显示方式，用图片存储起来，而且用第二章介绍过的拼合技术来显示（如图4.16）。



图4.16 一个简单的运用拼合技术显示的LCD数字风格的像素字体

我们在Interface Builder中放置了六个视图，小时、分钟、秒钟各两个，图4.17显示了这六个视图是如何在Interface Builder中放置的。如果每个都用一个淡出的outlets对象就会显得太多了，所以我们就用了一个IBOutletCollection对象把他们和控制器联系起来，这样我们就可以以数组的方式访问视图了。清单4.6是代码实现。

清单4.6 显示一个LCD风格的时钟

```
@interface ViewController : UIViewController

@property (nonatomic, strong) IBOutletCollection(UICollectionView) NSArray *digitViews;
@property (nonatomic, weak) NSTimer *timer;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad]; //get spritesheet image
 UIImage *digits = [UIImage imageNamed:@"Digits.png"];

 //set up digit views
 for (UIView *view in self.digitViews) {
 //set contents
 view.layer.contents = (__bridge id)digits.CGImage;
 view.layer.contentsRect = CGRectMake(0, 0, 0.1, 1.0);
 view.layer.contentsGravity = kCAGravityResizeAspect;
 }
}
```

```

}

//start timer
self.timer = [NSTimer scheduledTimerWithTimeInterval:1.0 target:self
 selector:@selector(tick)
 userInfo:nil
 repeats:YES];

//set initial clock time
[self tick];
}

- (void)setDigit:(NSInteger)digit forView:(UIView *)view
{
 //adjust contentsRect to select correct digit
 view.layer.contentsRect = CGRectMake(digit * 0.1, 0, 0.1, 1.0);
}

- (void)tick
{
 //convert time to hours, minutes and seconds
 NSCalendar *calendar = [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];
 NSUInteger units = NSHourCalendarUnit | NSMinuteCalendarUnit | NSSecondCalendarUnit;
 NSDateComponents *components = [calendar components:units fromDate:[NSDate date]];

 //set hours
 [self setDigit:components.hour / 10 forView:self.digitViews[0]];
 [self setDigit:components.hour % 10 forView:self.digitViews[1]];

 //set minutes
 [self setDigit:components.minute / 10 forView:self.digitViews[2]];
 [self setDigit:components.minute % 10 forView:self.digitViews[3]];

 //set seconds
 [self setDigit:components.second / 10 forView:self.digitViews[4]];
 [self setDigit:components.second % 10 forView:self.digitViews[5]];
}
@end

```

如图4.18，这样做的确起了效果，但是图片看起来模糊了。看起来默认的 `kCAFilterLinear` 选项让我们失望了。

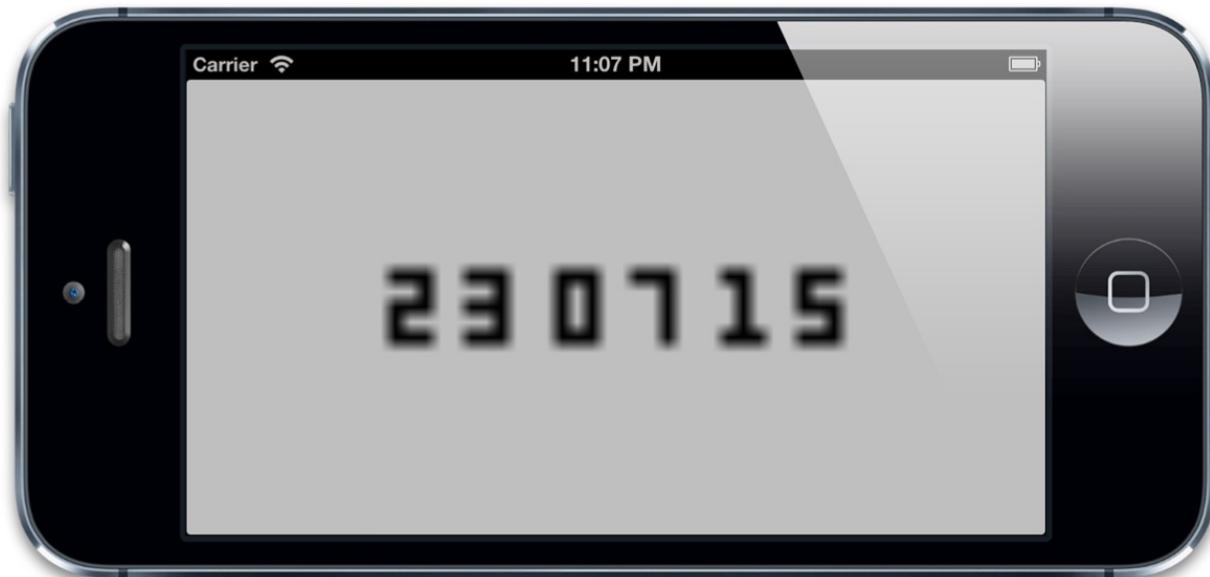


图4.18 一个模糊的时钟，由默认的 `kCAFILTERLinear` 引起

为了能像图4.19中那样，我们需要在for循环中加入如下代码：

```
view.layer.magnificationFilter = kCAFILTERNearest;
```



图4.19 设置了最近过滤之后的清晰显示

## 组透明

`UIView`有一个叫做 `alpha` 的属性来确定视图的透明度。`CALayer`有一个等同的属性叫做 `opacity`，这两个属性都是影响子层级的。也就是说，如果你给一个图层设置了 `opacity` 属性，那它的子图层都会受此影响。

iOS常见的做法是把一个控件的`alpha`值设置为 0.5 (50%) 以使其看上去呈现为不可用状态。对于独立的视图来说还不错，但是当一个控件有子视图的时候就有点奇怪了，图4.20展示了一个内嵌了`UILabel`的自定义`UIButton`；左边是一个不透明的按钮，右边是50%透明度的相同按钮。我们可以注意到，里面的标签的轮廓跟按钮的背景很不搭调。



图4.20 右边的渐隐按钮中，里面的标签清晰可见

这是由透明度的混合叠加造成的，当你显示一个50%透明度的图层时，图层的每个像素都会一半显示自己的颜色，另一半显示图层下面的颜色。这是正常的透明度的表现。但是如果图层包含一个同样显示50%透明的子图层时，你所看到的视图，50%来自子视图，25%来了图层本身的颜色，另外的25%则来自背景色。

在我们的示例中，按钮和表情都是白色背景。虽然他们都是50%的可见度，但是合起来的可见度是75%，所以标签所在的区域看上去就没有周围的部分那么透明。所以上去子视图就高亮了，使得这个显示效果都糟透了。

理想状况下，当你设置了一个图层的透明度，你希望它包含的整个图层树像一个整体一样的透明效果。你可以通过设置Info.plist文件中的 `UIListGroupOpacity` 为 YES 来达到这个效果，但是这个设置会影响到这个应用，整个app可能会受到不良影响。如果 `UIListGroupOpacity` 并未设置，iOS 6和以前的版本会默认为 NO（也许以后的版本会有一些改变）。

另一个方法就是，你可以设置CALayer的一个叫做 `shouldRasterize` 属性（见清单4.7）来实现组透明的效果，如果它被设置为YES，在应用透明度之前，图层及其子图层都会被整合成一个整体的图片，这样就没有透明度混合的问题了（如图 4.21）。

为了启用 `shouldRasterize` 属性，我们设置了图层的 `rasterizationScale` 属性。默认情况下，所有图层拉伸都是1.0，所以如果你使用了 `shouldRasterize` 属性，你就要确保你设置了 `rasterizationScale` 属性去匹配屏幕，以防止出现Retina屏幕像素化的问题。

当 `shouldRasterize` 和 `UIListGroupOpacity` 一起的时候，性能问题就出现了（我们在第12章『速度』和第15章『图层性能』将做出介绍），但是性能碰撞都本地化了（译者注：这句话需要再翻译）。

#### 清单4.7 使用 `shouldRasterize` 属性解决组透明问题

```

@interface ViewController ()
@property (nonatomic, weak) IBOutlet UIView *containerView;
@end

@implementation ViewController

- (UIButton *)customButton
{
 //create button
 CGRect frame = CGRectMake(0, 0, 150, 50);
 UIButton *button = [[UIButton alloc] initWithFrame:frame];
 button.backgroundColor = [UIColor whiteColor];
 button.layer.cornerRadius = 10;

 //add label
 frame = CGRectMake(20, 10, 110, 30);
 UILabel *label = [[UILabel alloc] initWithFrame:frame];
 label.text = @"Hello World";
}

```

```
label.textAlignment = NSTextAlignmentCenter;
[button addSubview:label];
return button;
}

- (void)viewDidLoad
{
 [super viewDidLoad];

 //create opaque button
 UIButton *button1 = [self customButton];
 button1.center = CGPointMake(50, 150);
 [self.containerView addSubview:button1];

 //create translucent button
 UIButton *button2 = [self customButton];

 button2.center = CGPointMake(250, 150);
 button2.alpha = 0.5;
 [self.containerView addSubview:button2];

 //enable rasterization for the translucent button
 button2.layer.shouldRasterize = YES;
 button2.layer.rasterizationScale = [UIScreen mainScreen].scale;
}
@end
```



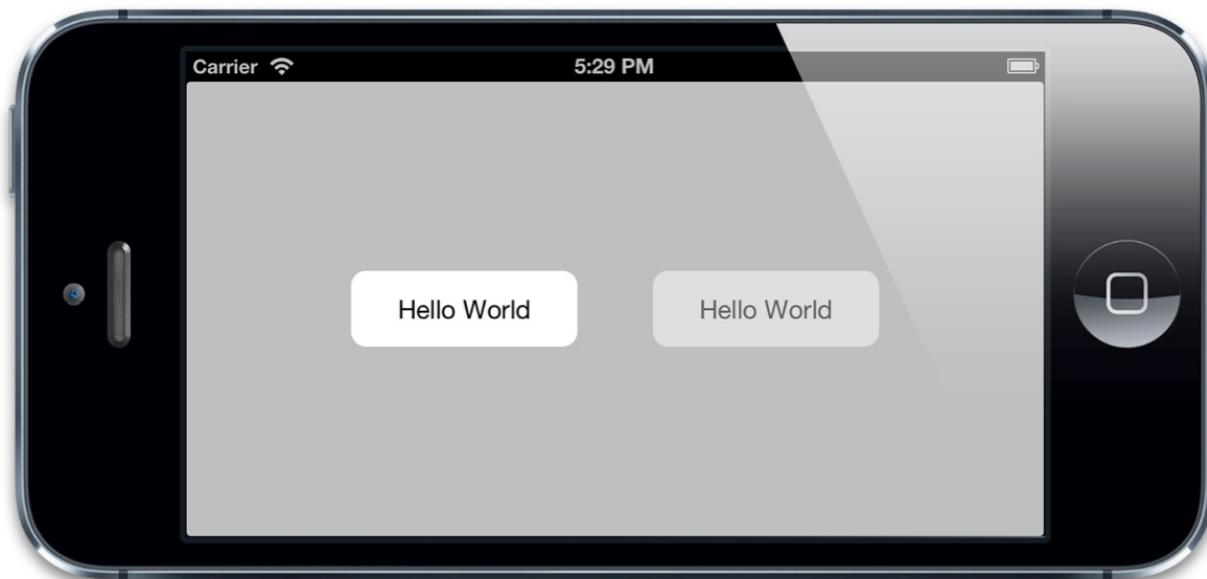


图4.21 修正后的图

## 总结

这一章介绍了一些可以通过代码应用到图层上的视觉效果，比如圆角，阴影和蒙板。我们也了解了拉伸过滤器和组透明。

在第五章，『变换』中，我们将会研究图层变化和3D转换

# 变换

很不幸，没人能告诉你母体是什么，你只能自己体会 -- 骇客帝国

在第四章“可视效果”中，我们研究了一些增强图层和它的内容显示效果的一些技术，在这一章中，我们将要研究可以用来对图层旋转，摆放或者扭曲的 `CGAffineTransform`，以及可以将扁平物体转换成三维空间对象的 `CATransform3D`（而不是仅仅对圆角矩形添加下沉阴影）。

# 仿射变换

在第三章“图层几何学”中，我们使用了 `UIView` 的 `transform` 属性旋转了钟的指针，但并没有解释背后运作的原理，实际上 `UIView` 的 `transform` 属性是一个 `CGAffineTransform` 类型，用于在二维空间做旋转，缩放和平移。`CGAffineTransform` 是一个可以和二维空间向量（例如 `CGPoint`）做乘法的 $3\times 2$ 的矩阵（见图5.1）。

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

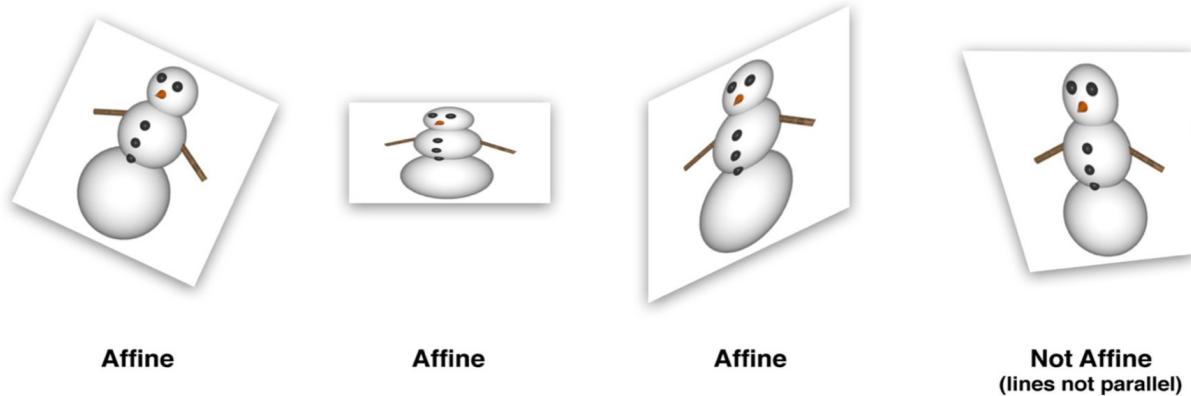
|                |                          |                            |
|----------------|--------------------------|----------------------------|
| <b>CGPoint</b> | <b>CGAffineTransform</b> | <b>Transformed CGPoint</b> |
|----------------|--------------------------|----------------------------|

图5.1 用矩阵表示的 `CGAffineTransform` 和 `CGPoint`

用 `CGPoint` 的每一列和 `CGAffineTransform` 矩阵的每一行对应元素相乘再求和，就形成了一个新的 `CGPoint` 类型的结果。要解释一下图中显示的灰色元素，为了能让矩阵做乘法，左边矩阵的列数一定要和右边矩阵的行数个数相同，所以要给矩阵填充一些标志值，使得既可以让矩阵做乘法，又不改变运算结果，并且没必要存储这些添加的值，因为它们的值不会发生变化，但是要用来做运算。

因此，通常会用 $3\times 3$ （而不是 $2\times 3$ ）的矩阵来做二维变换，你可能会见到3行2列格式的矩阵，这是所谓的以列为主的格式，图5.1所示的是以行为主的格式，只要能保持一致，用哪种格式都无所谓。

当对图层应用变换矩阵，图层矩形内的每一个点都被相应地做变换，从而形成一个新的四边形的形状。`CGAffineTransform` 中的“仿射”的意思是无论变换矩阵用什么值，图层中平行的两条线在变换之后任然保持平行，`CGAffineTransform` 可以做出任意符合上述标注的变换，图5.2显示了一些仿射的和非仿射的变换：



## 创建一个 **CGAffineTransform**

对矩阵数学做一个全面的阐述就超出本书的讨论范围了，不过如果你对矩阵完全不熟悉的话，矩阵变换可能会使你感到畏惧。幸运的是，Core Graphics提供了一系列函数，对完全没有数学基础的开发者也能够简单地做一些变换。如下几个函数都创建了一个 **CGAffineTransform** 实例：

```
CGAffineTransformMakeRotation(CGFloat angle)
CGAffineTransformMakeScale(CGFloat sx, CGFloat sy)
CGAffineTransformMakeTranslation(CGFloat tx, CGFloat ty)
```

旋转和缩放变换都可以很好解释--分别旋转或者缩放一个向量的值。平移变换是指每个点都移动了向量指定的x或者y值--所以如果向量代表了一个点，那它就平移了这个点的距离。

我们用一个很简单的项目来做个demo，把一个原始视图旋转45度角度（图5.3）

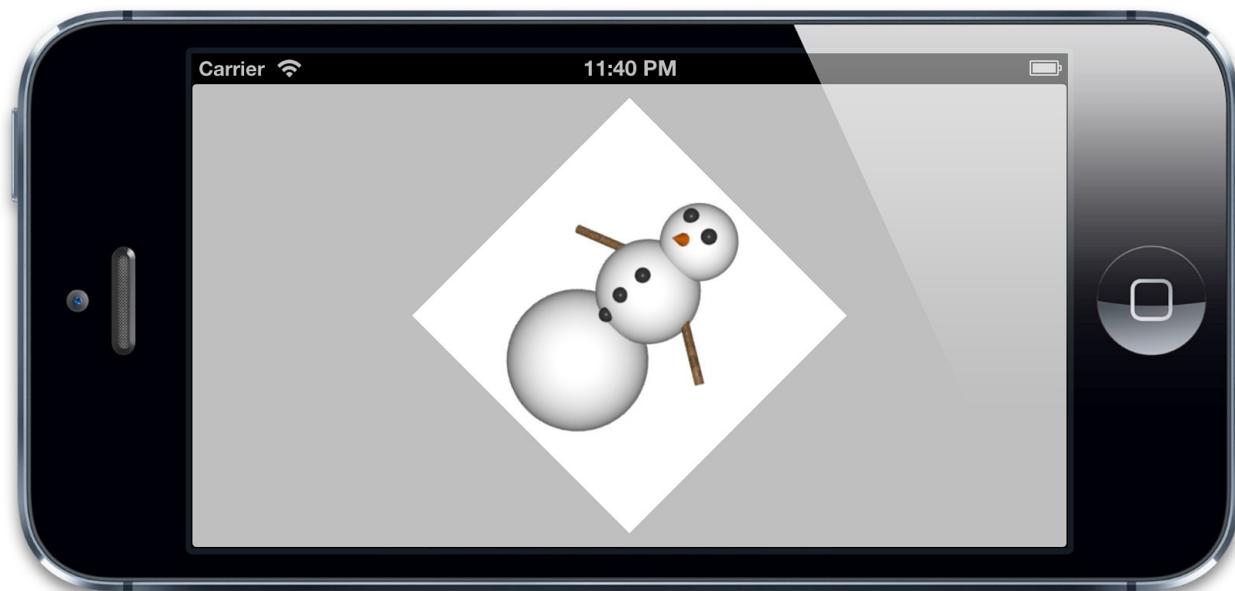


图5.3 使用仿射变换旋转45度角之后的视图

`UIView` 可以通过设置 `transform` 属性做变换，但实际上它只是封装了内部图层的变换。

`CALayer` 同样也有一个 `transform` 属性，但它的类型是 `CATransform3D`，而不是 `CGAffineTransform`，本章后续将会详细解释。`CALayer` 对应于 `UIView` 的 `transform` 属性叫做 `affineTransform`，清单5.1的例子就是使用 `affineTransform` 对图层做了45度顺时针旋转。

清单5.1 使用 `affineTransform` 对图层旋转45度

```

@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *layerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //rotate the layer 45 degrees
 CGAffineTransform transform = CGAffineTransformMakeRotation(M_PI_4);
 self.layerView.layer.affineTransform = transform;
}

@end

```

注意我们使用的旋转常量是 `M_PI_4`，而不是你想象的45，因为iOS的变换函数使用弧度而不是角度作为单位。弧度用数学常量pi的倍数表示，一个pi代表180度，所以四分之一的pi就是45度。

C的数学函数库（iOS会自动引入）提供了pi的一些简便的换算，`M_PI_4` 于是就是pi的四分之一，如果对换算不太清楚的话，可以用如下的宏做换算：

```
#define RADIANS_TO_DEGREES(x) ((x)/M_PI*180.0)
```

## 混合变换

Core Graphics提供了一系列的函数可以在一个变换的基础上做更深层次的变换，如果做一个既要缩放又要旋转的变换，这就会非常有用了。例如下面几个函数：

```

CGAffineTransformRotate(CGAffineTransform t, CGFloat angle)
CGAffineTransformScale(CGAffineTransform t, CGFloat sx, CGFloat sy)
CGAffineTransformTranslate(CGAffineTransform t, CGFloat tx, CGFloat ty)

```

当操纵一个变换的时候，初始生成一个什么都不做的变换很重要--也就是创建一个 `CGAffineTransform` 类型的空值，矩阵论中称作单位矩阵，Core Graphics同样也提供了一个方便的常量：

```
CGAffineTransformIdentity
```

最后，如果需要混合两个已经存在的变换矩阵，就可以使用如下方法，在两个变换的基础上创建一个新的变换：

```
CGAffineTransformConcat(CGAffineTransform t1, CGAffineTransform t2)
```

我们来用这些函数组合一个更加复杂的变换，先缩小50%，再旋转30度，最后向右移动200个像素（清单5.2）。图5.4显示了图层变换最后的结果。

### 清单5.2 使用若干方法创建一个复合变换

```
- (void)viewDidLoad
{
 [super viewDidLoad];
 //create a new transform
 CGAffineTransform transform = CGAffineTransformIdentity;
 //scale by 50%
 transform = CGAffineTransformScale(transform, 0.5, 0.5);
 //rotate by 30 degrees
 transform = CGAffineTransformRotate(transform, M_PI / 180.0 * 30);
 //translate by 200 points
 transform = CGAffineTransformTranslate(transform, 200, 0);
 //apply transform to layer
 self.layerView.layer.affineTransform = transform;
}
```

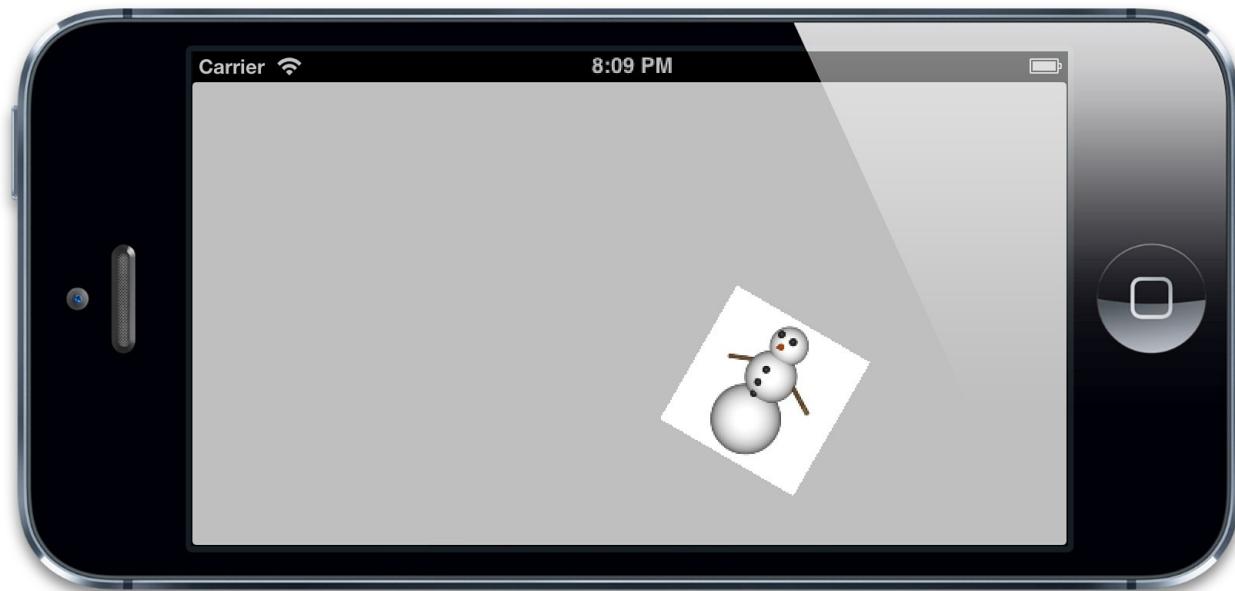


图5.4 顺序应用多个仿射变换之后的结果

图5.4中有些需要注意的地方：图片向右边发生了平移，但并没有指定距离那么远（200像素），另外它还有点向下发生了平移。原因在于当你按顺序做了变换，上一个变换的结果将会影响之后的变换，所以200像素的向右平移同样也被旋转了30度，缩小了50%，所以它实际上是斜向移动了100像素。

这意味着变换的顺序会影响最终的结果，也就是说旋转之后的平移和平移之后的旋转结果可能不同。

```
#define DEGREES_TO_RADIANS(x) ((x)/180.0*M_PI)
```

## 3D变换

CG的前缀告诉我们，`CGAffineTransform` 类型属于Core Graphics框架，Core Graphics实际上是一个严格意义上的2D绘图API，并且 `CGAffineTransform` 仅仅对2D变换有效。

在第三章中，我们提到了 `zPosition` 属性，可以用来让图层靠近或者远离相机（用户视角），`transform` 属性（`CATransform3D` 类型）可以真正做到这点，即让图层在3D空间内移动或者旋转。

和 `CGAffineTransform` 类似，`CATransform3D` 也是一个矩阵，但是和 $2 \times 3$ 的矩阵不同，`CATransform3D` 是一个可以在3维空间内做变换的 $4 \times 4$ 的矩阵（图5.6）。

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \times \begin{bmatrix} m_{11} & m_{21} & m_{31} & m_{41} \\ m_{12} & m_{22} & m_{32} & m_{42} \\ m_{13} & m_{23} & m_{33} & m_{43} \\ m_{14} & m_{24} & m_{34} & m_{44} \end{bmatrix} = \begin{bmatrix} x' & y' & z' & 1 \end{bmatrix}$$

**CGPoint + zPosition**

**CATransform3D**

**Transformed Point**

图5.6 对一个3D像素点做 `CATransform3D` 矩阵变换

和 `CGAffineTransform` 矩阵类似，Core Animation提供了一系列的方法用来创建和组合 `CATransform3D` 类型的矩阵，和Core Graphics的函数类似，但是3D的平移和旋转多处了一个 `z` 参数，并且旋转函数除了 `angle` 之外多出了 `x`, `y`, `z` 三个参数，分别决定了每个坐标轴方向上的旋转：

```
CATransform3DMakeRotation(CGFloat angle, CGFloat x, CGFloat y, CGFloat z)
CATransform3DMakeScale(CGFloat sx, CGFloat sy, CGFloat sz)
CATransform3DMakeTranslation(Gfloat tx, CGFloat ty, CGFloat tz)
```

你应该对X轴和Y轴比较熟悉了，分别以右和下为正方向（回忆第三章，这是iOS上的标准结构，在Mac OS，Y轴朝上为正方向），Z轴和这两个轴分别垂直，指向视角外为正方向（图5.7）。

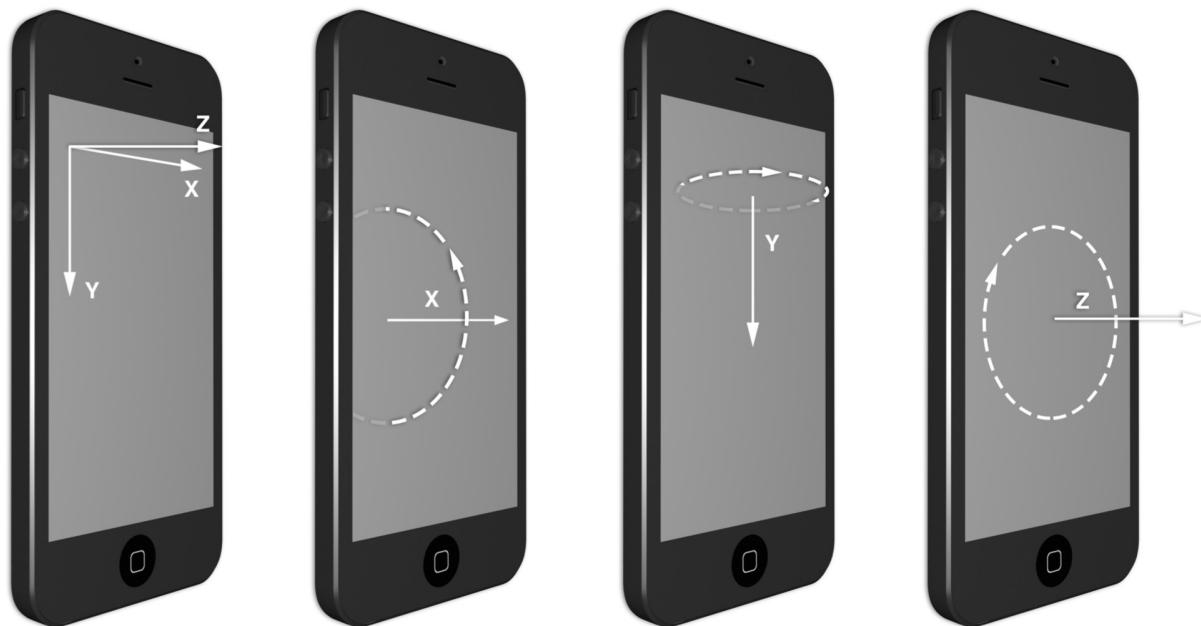


图5.7 X，Y，Z轴，以及围绕它们旋转的方向

由图所见，绕Z轴的旋转等同于之前二维空间的仿射旋转，但是绕X轴和Y轴的旋转就突破了屏幕的二维空间，并且在用户视角看来发生了倾斜。

举个例子：清单5.4的代码使用了 `CATransform3DMakeRotation` 对视图内的图层绕Y轴做了45度角的旋转，我们可以把视图向右倾斜，这样会看得更清晰。

结果见图5.8，但并不像我们期待的那样。

#### 清单5.4 绕Y轴旋转图层

```
@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //rotate the layer 45 degrees along the Y axis
 CATransform3D transform = CATransform3DMakeRotation(M_PI_4, 0,
 self.layerView.layer.transform = transform;
}

@end
```

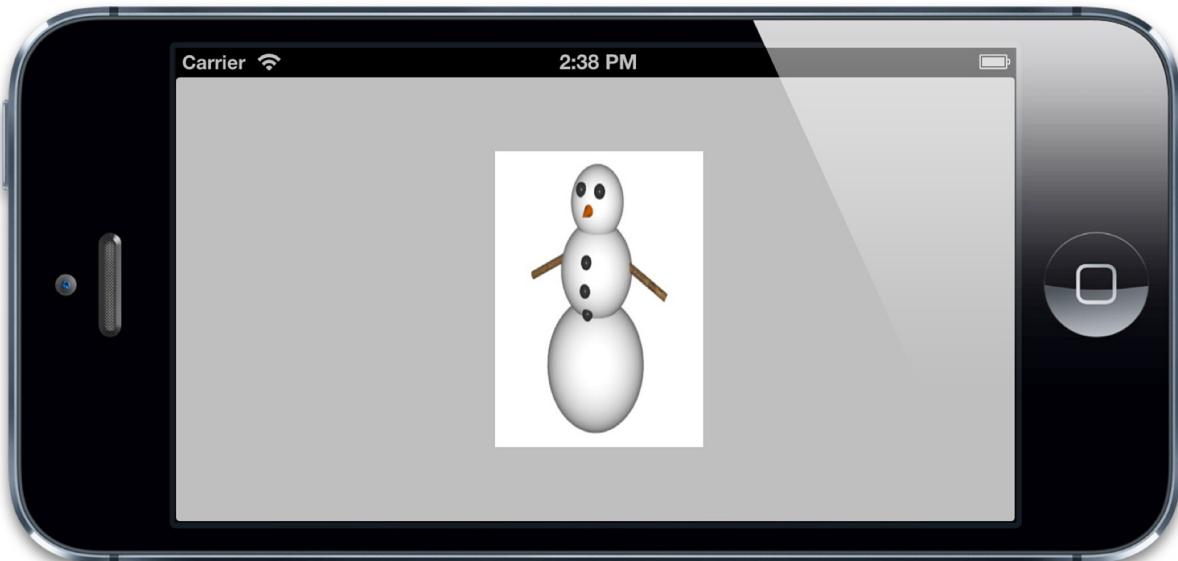


图5.8 绕y轴旋转45度的视图

看起来图层并没有被旋转，而是仅仅在水平方向上的一个压缩，是哪里出了问题呢？

其实完全没错，视图看起来更窄实际上是因为我们在用一个斜向的视角看它，而不是透视。

## 透视投影

在真实世界中，当物体远离我们的的时候，由于视角的原因看起来会变小，理论上说远离我们的视图的边要比靠近视角的边跟短，但实际上并没有发生，而我们当前的视角是等距离的，也就是在3D变换中任然保持平行，和之前提到的仿射变换类似。

在等距投影中，远处的物体和近处的物体保持同样的缩放比例，这种投影也有它自己的用处（例如建筑绘图，颠倒，和伪3D视频），但当前我们并不需要。

为了做一些修正，我们需要引入投影变换（又称作`z`变换）来对除了旋转之外的变换矩阵做一些修改，Core Animation并没有给我们提供设置透视变换的函数，因此我们需要手动修改矩阵值，幸运的是，很简单：

`CATransform3D` 的透视效果通过一个矩阵中一个很简单的元素来控制：`m34`。`m34`（图5.9）用于按比例缩放X和Y的值来计算到底要离视角多远。

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \times \begin{bmatrix} m_{11} & m_{21} & m_{31} & m_{41} \\ m_{12} & m_{22} & m_{32} & m_{42} \\ m_{13} & m_{23} & m_{33} & m_{43} \\ m_{14} & m_{24} & m_{34} & m_{44} \end{bmatrix} = \begin{bmatrix} x' & y' & z' & 1 \end{bmatrix}$$

**CGPoint + zPosition**

**CATransform3D**

**Transformed Point**

图5.9 `CATransform3D` 的 `m34` 元素，用来做透视

`m34` 的默认值是0，我们可以通过设置 `m34` 为 `-1.0 / d` 来应用透视效果，`d` 代表了想象中视角相机和屏幕之间的距离，以像素为单位，那应该如何计算这个距离呢？实际上并不需要，大概估算一个就好了。

因为视角相机实际上并不存在，所以可以根据屏幕上的显示效果自由决定它的防止的位置。通常500-1000就已经很好了，但对于特定的图层有时候更小后者更大的值会看起来更舒服，减少距离的值会增强透视效果，所以一个非常微小的值会让它看起来更加失真，然而一个非常大的值会让它基本失去透视效果，对视图应用透视的代码见清单5.5，结果见图5.10。

清单5.5 对变换应用透视效果

```
@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create a new transform
 CATransform3D transform = CATransform3DIdentity;
 //apply perspective
 transform.m34 = - 1.0 / 500.0;
 //rotate by 45 degrees along the Y axis
 transform = CATransform3DRotate(transform, M_PI_4, 0, 1, 0);
 //apply to layer
 self.layerView.layer.transform = transform;
}

@end
```

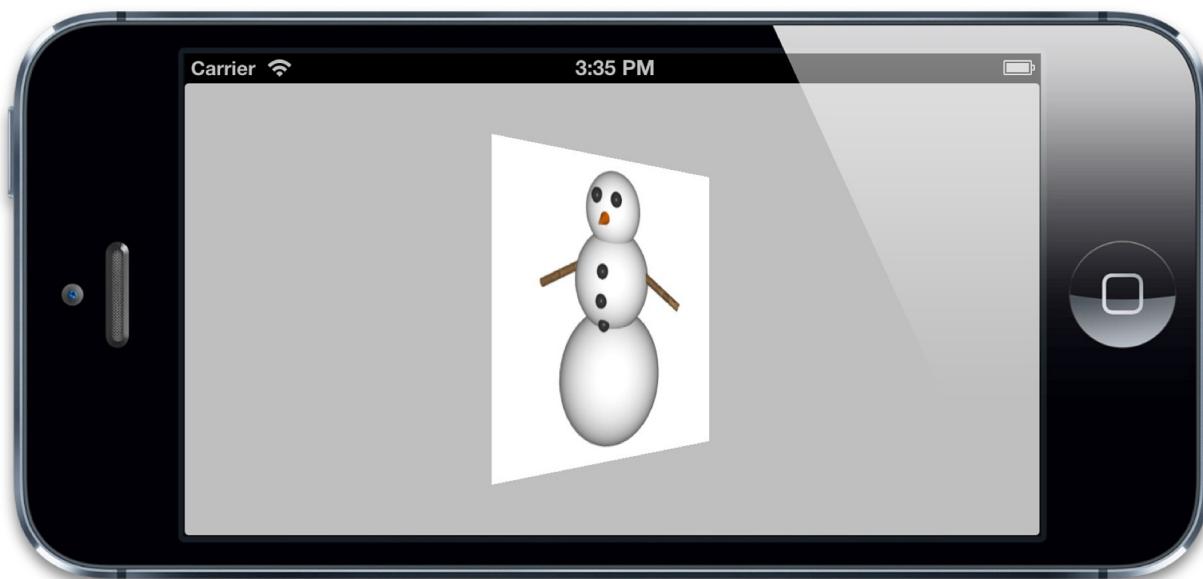


图5.10 应用透视效果之后再次对图层做旋转

## 灭点

当在透视角度绘图的时候，远离相机视角的物体将会变小变远，当远离到一个极限距离，它们可能就缩成了一个点，于是所有的物体最后都汇聚消失在同一个点。

在现实中，这个点通常是视图的中心（图5.11），于是为了在应用中创建拟真效果的透视，这个点应该聚在屏幕中点，或者至少是包含所有3D对象的视图中点。

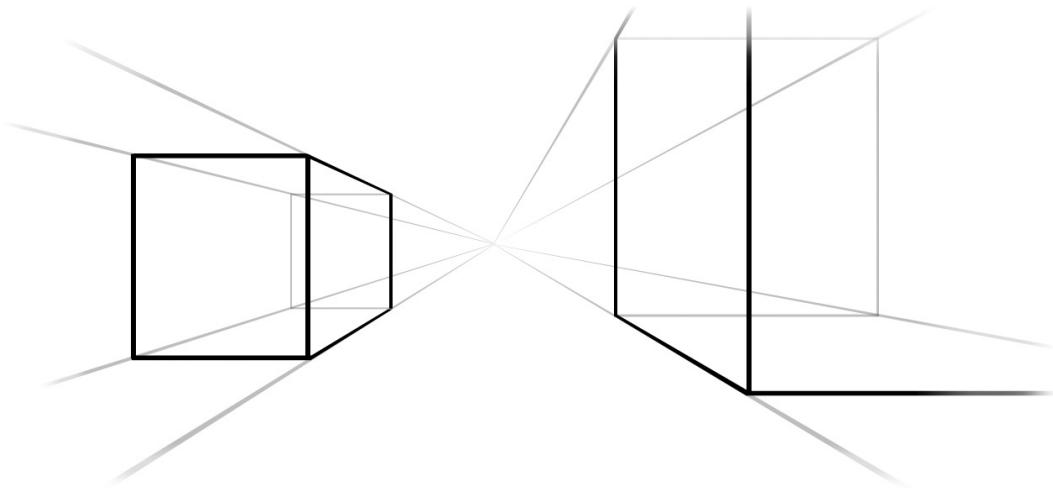


图5.11 灭点

Core Animation定义了这个点位于变换图层的 `anchorPoint`（通常位于图层中心，但也有例外，见第三章）。这就是说，当图层发生变换时，这个点永远位于图层变换之前 `anchorPoint` 的位置。

当改变一个图层的 `position`，你也改变了它的灭点，做3D变换的时候要时刻记住这一点，当你视图通过调整 `m34` 来让它更加有3D效果，应该首先把它放置于屏幕中央，然后通过平移来把它移动到指定位置（而不是直接改变它的 `position`），这样所有的3D图层都共享一个灭点。

## sublayerTransform 属性

如果有多个视图或者图层，每个都做3D变换，那就需要分别设置相同的`m34`值，并且确保在变换之前都在屏幕中央共享同一个 `position`，如果用一个函数封装这些操作的确会更加方便，但仍然有限制（例如，你不能在Interface Builder中摆放视图），这里有一个更好的方法。

`CALayer` 有一个属性叫做 `sublayerTransform`。它也是 `CATransform3D` 类型，但和对一个图层的变换不同，它影响到所有的子图层。这意味着你可以一次性对包含这些图层的容器做变换，于是所有的子图层都自动继承了这个变换方法。

相较而言，通过在一个地方设置透视变换会很方便，同时它会带来另一个显著的优势：灭点被设置在容器图层的中点，从而不需要再对子图层分别设置了。这意味着你可以随意使用 `position` 和 `frame` 来放置子图层，而不需要把它们放置在屏幕

中点，然后为了保证统一的灭点用变换来做平移。

我们来用一个demo举例说明。这里用Interface Builder并排放置两个视图（图5.12），然后通过设置它们容器视图的透视变换，我们可以保证它们有相同的透视和灭点，代码见清单5.6，结果见图5.13。

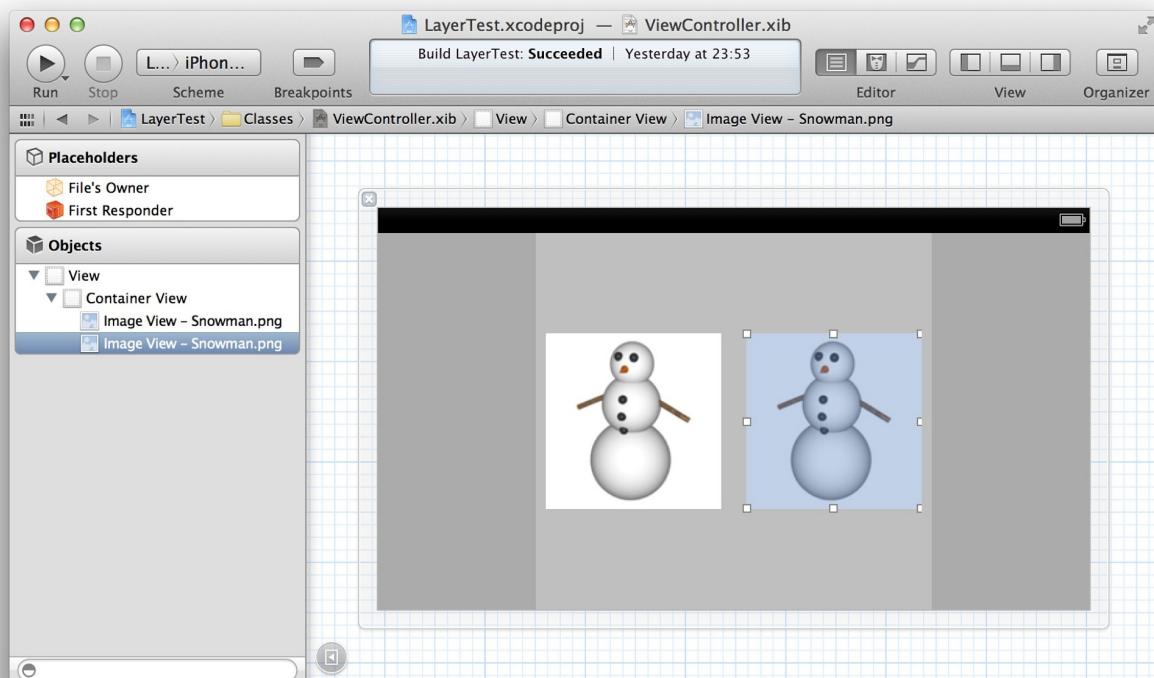


图5.12 在一个视图容器内并排放置两个视图

清单5.6 应用 `sublayerTransform`

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, weak) IBOutlet UIView *layerView1;
@property (nonatomic, weak) IBOutlet UIView *layerView2;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //apply perspective transform to container
 CATransform3D perspective = CATransform3DIdentity;
 perspective.m34 = - 1.0 / 500.0;
 self.containerView.layer.sublayerTransform = perspective;
 //rotate layerView1 by 45 degrees along the Y axis
 CATransform3D transform1 = CATransform3DMakeRotation(M_PI_4, 0,
 self.layerView1.layer.transform = transform1;
 //rotate layerView2 by 45 degrees along the Y axis
 CATransform3D transform2 = CATransform3DMakeRotation(-M_PI_4, 0,
 self.layerView2.layer.transform = transform2;
}


```

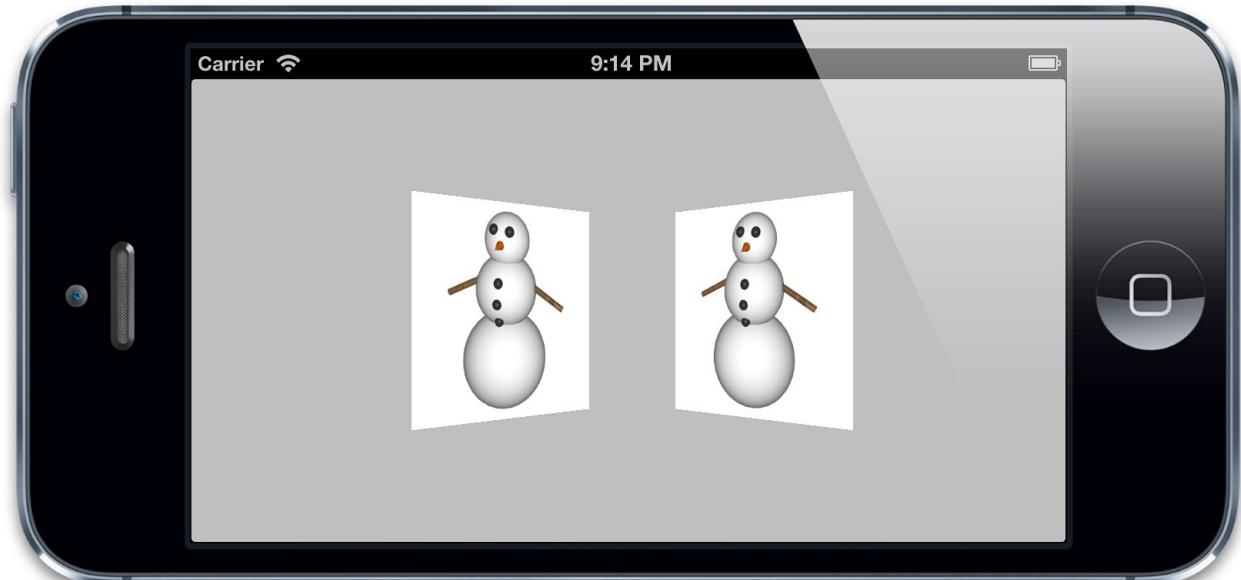


图5.13 通过相同的透视效果分别对视图做变换

## 背面

我们既然可以在3D场景下旋转图层，那么也可以从背面去观察它。如果我们在清单5.4中把角度修改为 `M_PI` (180度) 而不是当前的 `M_PI_4` (45度)，那么将会把图层完全旋转一个半圈，于是完全背对了相机视角。

那么从背部看图层是什么样的呢，见图5.14

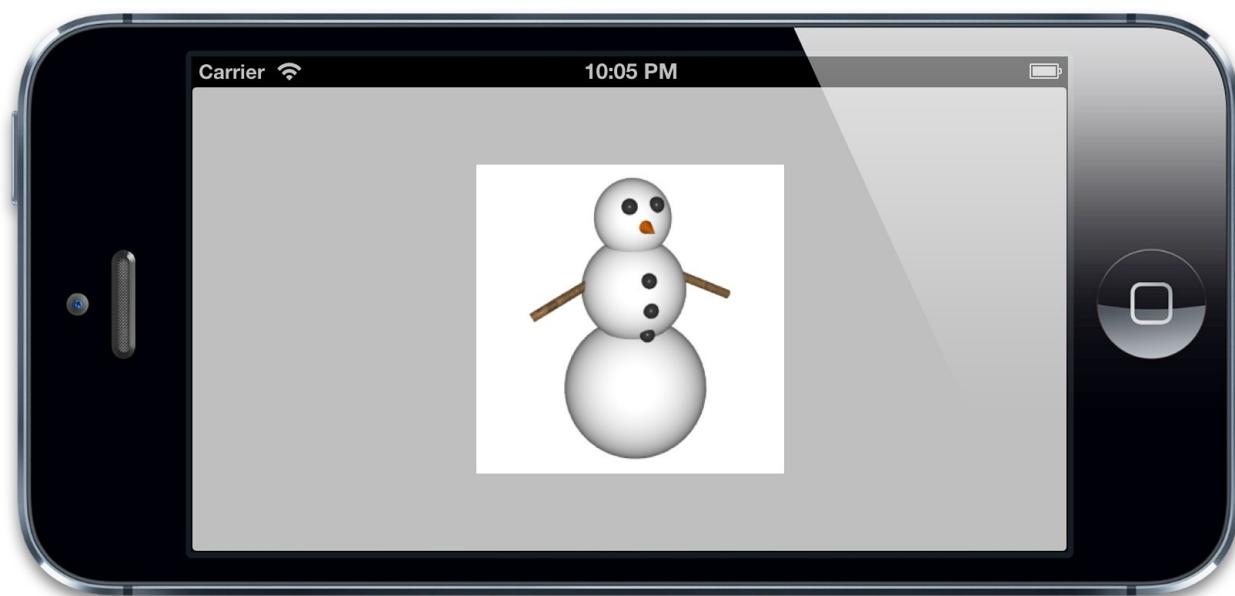


图5.14 视图的背面，一个镜像对称的图片

如你所见，图层是双面绘制的，反面显示的是正面的一个镜像图片。

但这并不是一个很好的特性，因为如果图层包含文本或者其他控件，那用户看到这些内容的镜像图片当然会感到困惑。另外也有可能造成资源的浪费：想象用这些图层形成一个不透明的固态立方体，既然永远都看不见这些图层的背面，那为什么浪费GPU来绘制它们呢？

`CALayer` 有一个叫做 `doubleSided` 的属性来控制图层的背面是否要被绘制。这是一个 `BOOL` 类型，默認為 `YES`，如果设置为 `NO`，那么当图层正面从相机视角消失的时候，它将不会被绘制。

## 扁平化图层

如果对包含已经做过变换的图层的图层做反方向的变换将会发什么什么呢？是不是有点困惑？见图5.15

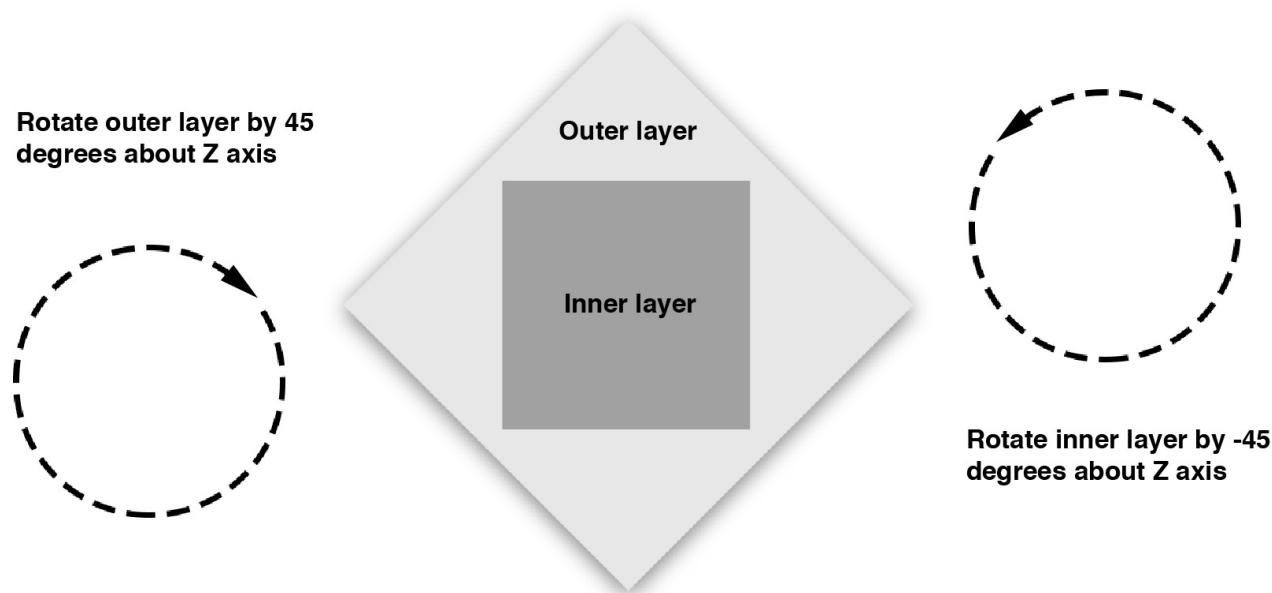


图5.15 反方向变换的嵌套图层

注意做了 $-45$ 度旋转的内部图层是怎样抵消旋转 $45$ 度的图层，从而恢复正常状态的。

如果内部图层相对外部图层做了相反的变换（这里是绕Z轴的旋转），那么按照逻辑这两个变换将被相互抵消。

验证一下，相应代码见清单5.7，结果见5.16

清单5.7 绕Z轴做相反的旋转变换

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *outerView;
@property (nonatomic, weak) IBOutlet UIView *innerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //rotate the outer layer 45 degrees
 CATransform3D outer = CATransform3DMakeRotation(M_PI_4, 0, 0, 1);
 self.outerView.layer.transform = outer;
 //rotate the inner layer -45 degrees
 CATransform3D inner = CATransform3DMakeRotation(-M_PI_4, 0, 0, 1);
 self.innerView.layer.transform = inner;
}

@end
```

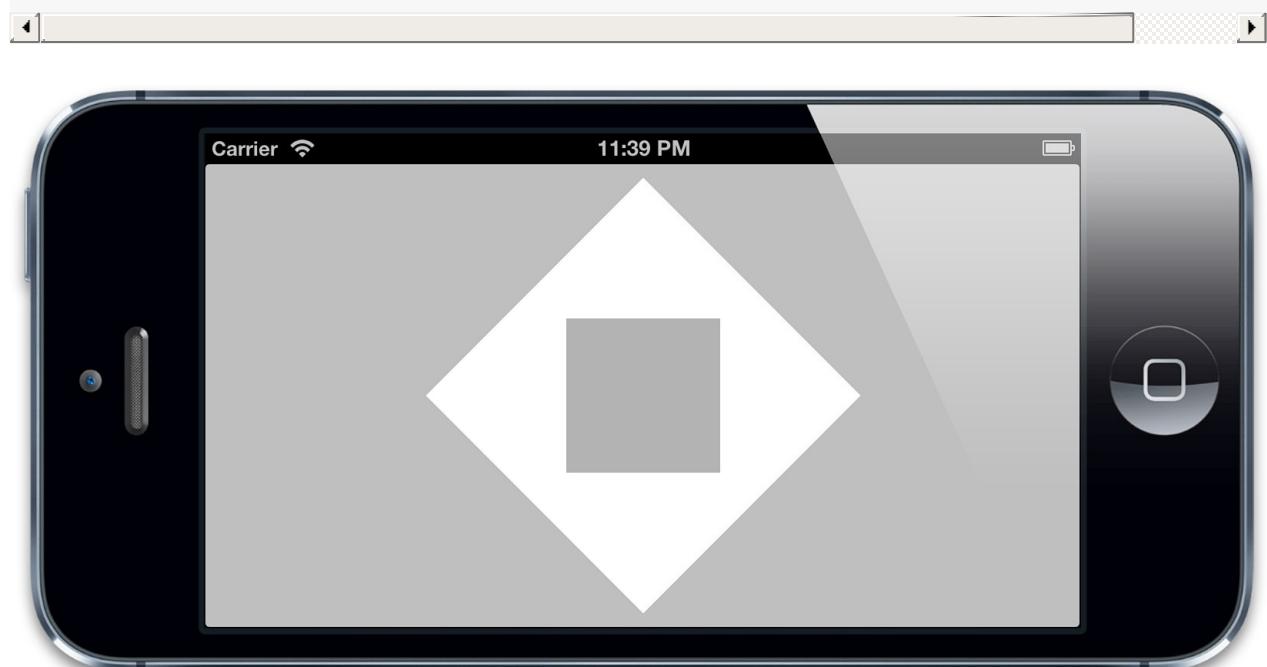


图5.16 旋转后的视图

运行结果和我们预期的一致。现在在3D情况下再试一次。修改代码，让内外两个视图绕Y轴旋转而不是Z轴，再加上透视效果，以便我们观察。注意不能用 `sublayerTransform` 属性，因为内部的图层并不直接是容器图层的子图层，所以这里分别对图层设置透视变换（清单5.8）。

清单5.8 绕Y轴相反的旋转变换

```
- (void)viewDidLoad
{
 [super viewDidLoad];
 //rotate the outer layer 45 degrees
 CATransform3D outer = CATransform3DIdentity;
 outer.m34 = -1.0 / 500.0;
 outer = CATransform3DRotate(outer, M_PI_4, 0, 1, 0);
 self.outerView.layer.transform = outer;
 //rotate the inner layer -45 degrees
 CATransform3D inner = CATransform3DIdentity;
 inner.m34 = -1.0 / 500.0;
 inner = CATransform3DRotate(inner, -M_PI_4, 0, 1, 0);
 self.innerView.layer.transform = inner;
}
```

预期的效果应该如图5.17所示。

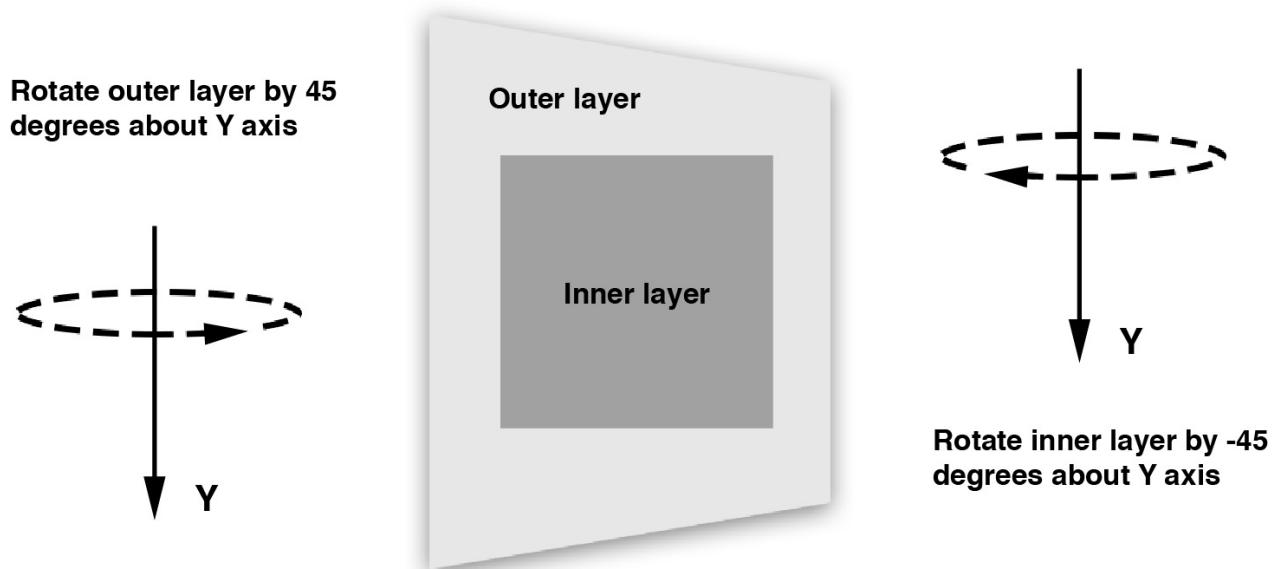


图5.17 绕Y轴做相反旋转的预期结果。

但其实这并不是我们所看到的，相反，我们看到的结果如图5.18所示。发生了什么呢？内部的图层仍然向左侧旋转，并且发生了扭曲，但按道理说它应该保持正面朝上，并且显示正常的方块。

这是由于尽管Core Animation图层存在于3D空间之内，但它们并不都存在同一个3D空间。每个图层的3D场景其实是扁平化的，当你从正面观察一个图层，看到的实际上由子图层创建的想象出来的3D场景，但当你倾斜这个图层，你会发现实际上这个3D场景仅仅是被绘制在图层的表面。

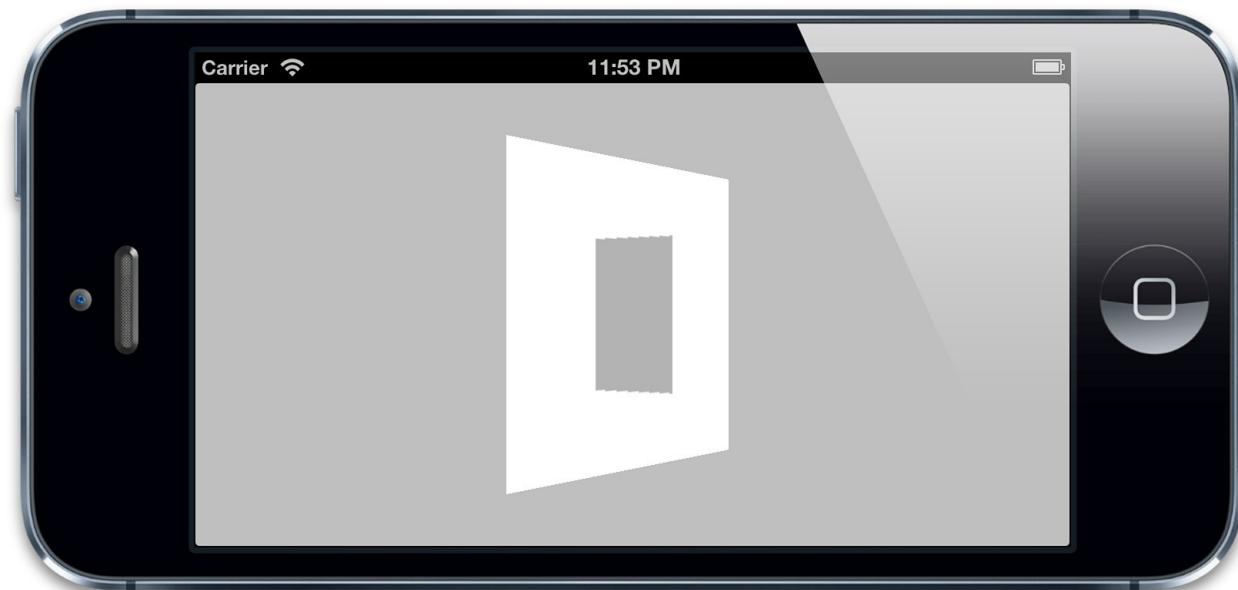


图5.18 绕Y轴做相反旋转的真实结果

类似的，当你在玩一个3D游戏，实际上仅仅是把屏幕做了一次倾斜，或许在游戏中可以看见有一面墙在你面前，但是倾斜屏幕并不能够看见墙里面的东西。所有场景里面绘制的东西并不会随着你观察它的角度改变而发生变化；图层也是同样的道理。

这使得用Core Animation创建非常复杂的3D场景变得十分困难。你不能够使用图层树去创建一个3D结构的层级关系--在相同场景下的任何3D表面必须和同样的图层保持一致，这是因为每个的父视图都把它的子视图扁平化了。

至少当你用正常的 `CALayer` 的时候是这样，`CALayer` 有一个叫做 `CATransformLayer` 的子类来解决这个问题。具体在第六章“特殊的图层”中将会具体讨论。

## 固体对象

现在你懂得了在3D空间的一些图层布局的基础，我们来试着创建一个固态的3D对象（实际上是一个技术上所谓的空洞对象，但它以固态呈现）。我们用六个独立的视图来构建一个立方体的各个面。

在这个例子中，我们用Interface Builder来构建立方体的面（图5.19），我们当然可以用代码来写，但是用Interface Builder的好处是可以方便的在每一个面上添加子视图。记住这些面仅仅是包含视图和控件的普通的用户界面元素，它们完全是我们界面交互的部分，并且当把它折成一个立方体之后也不会改变这个性质。

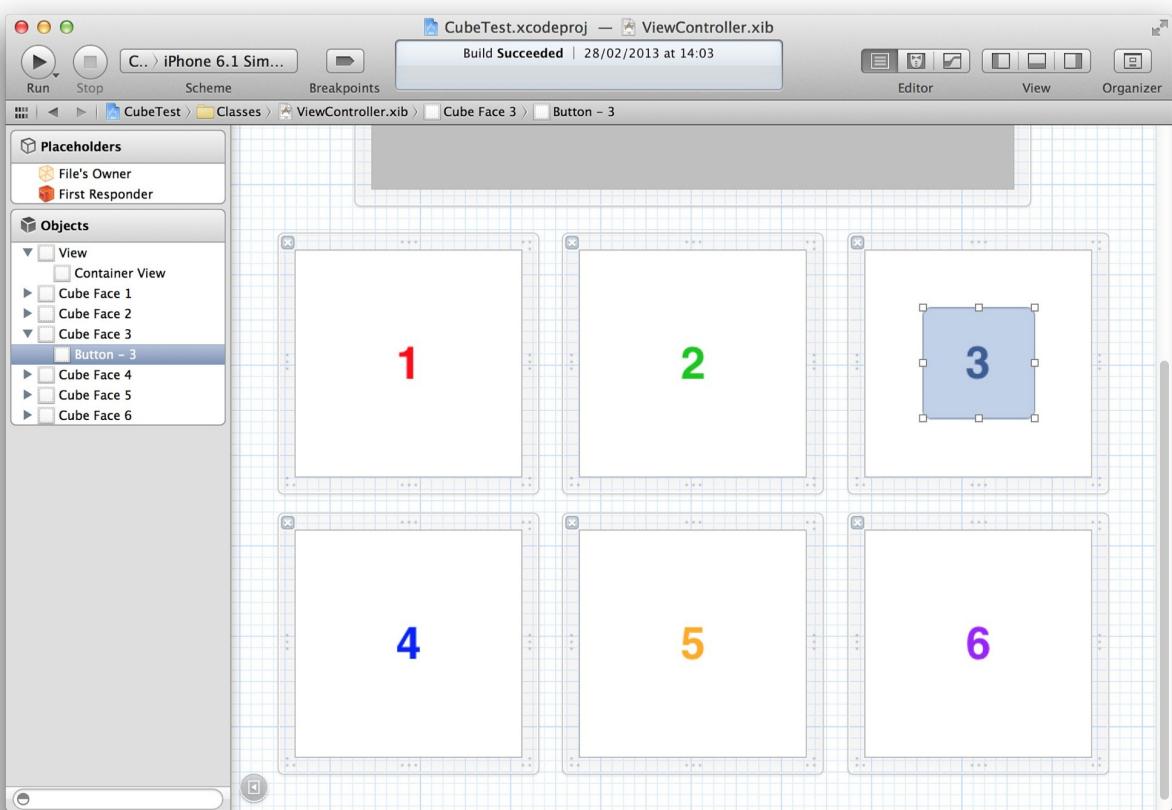


图5.19 用Interface Builder对立方体的六个面进行布局

这些面视图并没有放置在主视图当中，而是松散地排列在根nib文件里面。我们并不关心在这个容器中如何摆放它们的位置，因为后续将会用图层的 `transform` 对它们进行重新布局，并且用Interface Builder在容器视图之外摆放他们可以让我们容易看清楚它们的内容，如果把它们一个叠着一个都塞进主视图，将会变得很难看。

我们把一个有颜色的 `UILabel` 放置在视图内部，是为了清楚的辨别它们之间的关系，并且 `UIButton` 被放置在第三个面视图里面，后面会做简单的解释。

具体把视图组织成立方体的代码见清单5.9，结果见图5.20

### 清单5.9 创建一个立方体

```

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, strong) IBOutletCollection(UIView) NSArray *faces;

@end

@implementation ViewController

- (void)addFace:(NSInteger)index withTransform:(CATransform3D)transform
{
 //get the face view and add it to the container
 UIView *face = self.faces[index];
 [self.containerView addSubview:face];
 //center the face view within the container
 CGSize containerSize = self.containerView.bounds.size;
 face.center = CGPointMake(containerSize.width / 2.0, containerSize.height / 2.0);
 // apply the transform
 face.layer.transform = transform;
}

- (void)viewDidLoad
{
 [super viewDidLoad];
 //set up the container sublayer transform
 CATransform3D perspective = CATransform3DIdentity;
 perspective.m34 = -1.0 / 500.0;
 self.containerView.layer.sublayerTransform = perspective;
 //add cube face 1
 CATransform3D transform = CATransform3DMakeTranslation(0, 0, 100);
 [self addFace:0 withTransform:transform];
 //add cube face 2
 transform = CATransform3DMakeTranslation(100, 0, 0);
}

```

```
 transform = CATransform3DRotate(transform, M_PI_2, 0, 1, 0);
 [self addFace:1 withTransform:transform];
 //add cube face 3
 transform = CATransform3DMakeTranslation(0, -100, 0);
 transform = CATransform3DRotate(transform, M_PI_2, 1, 0, 0);
 [self addFace:2 withTransform:transform];
 //add cube face 4
 transform = CATransform3DMakeTranslation(0, 100, 0);
 transform = CATransform3DRotate(transform, -M_PI_2, 1, 0, 0);
 [self addFace:3 withTransform:transform];
 //add cube face 5
 transform = CATransform3DMakeTranslation(-100, 0, 0);
 transform = CATransform3DRotate(transform, -M_PI_2, 0, 1, 0);
 [self addFace:4 withTransform:transform];
 //add cube face 6
 transform = CATransform3DMakeTranslation(0, 0, -100);
 transform = CATransform3DRotate(transform, M_PI, 0, 1, 0);
 [self addFace:5 withTransform:transform];
}

@end
```

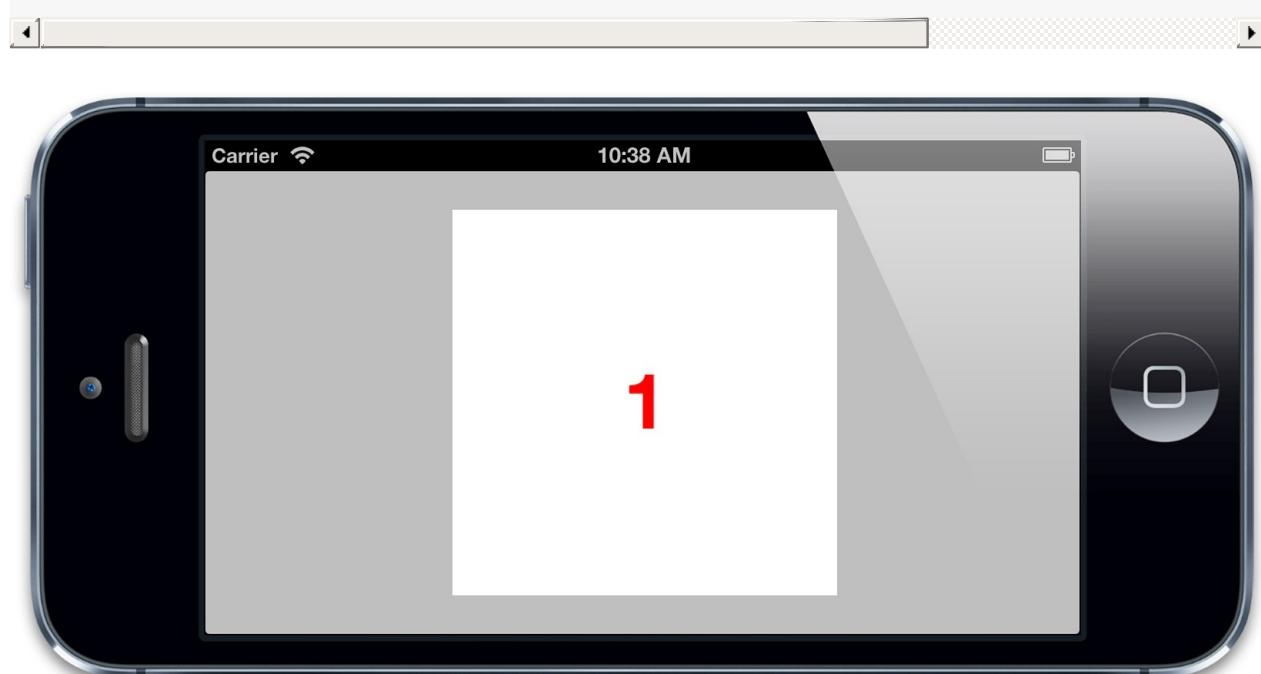


图5.20 正面朝上的立方体

从这个角度看立方体并不是很明显；看起来只是一个方块，为了更好地欣赏它，我们将更换一个不同的视角。

旋转这个立方体将会显得很笨重，因为我们要单独对每个面做旋转。另一个简单的方案是通过调整容器视图的 `sublayerTransform` 去旋转照相机。

添加如下几行去旋转 `containerView` 图层的 `perspective` 变换矩阵：

```
perspective = CATransform3DRotate(perspective, -M_PI_4, 1, 0, 0);
perspective = CATransform3DRotate(perspective, -M_PI_4, 0, 1, 0);
```

这就对相机（或者相对相机的整个场景，你也可以这么认为）绕Y轴旋转45度，并且绕X轴旋转45度。现在从另一个角度去观察立方体，就能看出它的真实面貌（图5.21）。

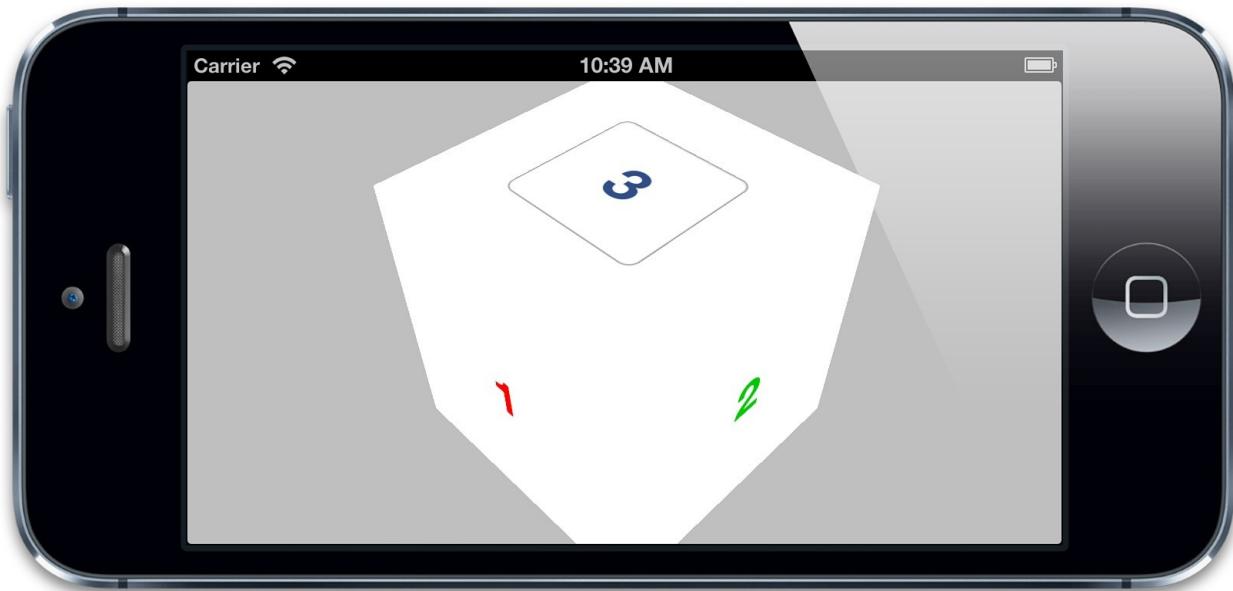


图5.21 从一个边角观察的立方体

## 光亮和阴影

现在它看起来更像是一个立方体沒错了，但是对每个面之间的连接还是很难分辨。Core Animation可以用3D显示图层，但是它对光线并没有概念。如果想让立方体看起来更加真实，需要自己做一个阴影效果。你可以通过改变每个面的背景颜色或者直接用带光亮效果的图片来调整。

如果需要动态地创建光线效果，你可以根据每个视图的方向应用不同的alpha值做出半透明的阴影图层，但为了计算阴影图层的不透明度，你需要得到每个面的正太向量（垂直于表面的向量），然后根据一个想象的光源计算出两个向量叉乘结果。叉乘代表了光源和图层之间的角度，从而决定了它有多大程度上的光亮。

清单5.10实现了这样一个结果，我们用GLKit框架来做向量的计算（你需要引入GLKit库来运行代码），每个面的 CATransform3D 都被转换成 GLKMatrix4，然后通过 GLKMatrix4GetMatrix3 函数得出一个 $3\times 3$ 的旋转矩阵。这个旋转矩阵指定了图层的方向，然后可以用它来得到正太向量的值。

结果如图5.22所示，试着调整 LIGHT\_DIRECTION 和 AMBIENT\_LIGHT 的值来切换光线效果

### 清单5.10 对立方体的表面应用动态的光线效果

```
#import "ViewController.h"
#import
#import

#define LIGHT_DIRECTION 0, 1, -0.5
#define AMBIENT_LIGHT 0.5

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, strong) IBOutletCollection(UIView) NSArray *faces;

@end

@implementation ViewController

- (void)applyLightingToFace:(CALayer *)face
{
 //add lighting layer
 CALayer *layer = [CALayer layer];
 layer.frame = face.bounds;
 [face addSublayer:layer];
 //convert the face transform to matrix
 //((GLKMatrix4 has the same structure as CATransform3D)
 //译者注：GLKMatrix4和CATransform3D内存结构一致，但坐标类型有长度区别
 CATransform3D transform = face.transform;
 GLKMatrix4 matrix4 = *(GLKMatrix4 *)&transform;
 GLKMatrix3 matrix3 = GLKMatrix4GetMatrix3(matrix4);
 //get face normal
 GLKVector3 normal = GLKVector3Make(0, 0, 1);
}
```

```

 normal = GLKMatrix3MultiplyVector3(matrix3, normal);
 normal = GLKVector3Normalize(normal);
 //get dot product with light direction
 GLKVector3 light = GLKVector3Normalize(GLKVector3Make(LIGHT_DIR_X,
 LIGHT_DIR_Y, LIGHT_DIR_Z));
 float dotProduct = GLKVector3DotProduct(light, normal);
 //set lighting layer opacity
 CGFloat shadow = 1 + dotProduct - AMBIENT_LIGHT;
 UIColor *color = [UIColor colorWithWhite:0 alpha:shadow];
 layer.backgroundColor = color.CGColor;
 }

- (void)addFace:(NSInteger)index withTransform:(CATransform3D)transform
{
 //get the face view and add it to the container
 UIView *face = self.faces[index];
 [self.containerView addSubview:face];
 //center the face view within the container
 CGSize containerSize = self.containerView.bounds.size;
 face.center = CGPointMake(containerSize.width / 2.0, containerSize.height / 2.0);
 // apply the transform
 face.layer.transform = transform;
 //apply lighting
 [self applyLightingToFace:face.layer];
}

- (void)viewDidLoad
{
 [super viewDidLoad];
 //set up the container sublayer transform
 CATransform3D perspective = CATransform3DIdentity;
 perspective.m34 = -1.0 / 500.0;
 perspective = CATransform3DRotate(perspective, -M_PI_4, 1, 0, 0);
 perspective = CATransform3DRotate(perspective, -M_PI_4, 0, 1, 0);
 self.containerView.layer.sublayerTransform = perspective;
 //add cube face 1
 CATransform3D transform = CATransform3DMakeTranslation(0, 0, 100);
 [self addFace:0 withTransform:transform];
 //add cube face 2
 transform = CATransform3DMakeTranslation(100, 0, 0);
 transform = CATransform3DRotate(transform, M_PI_2, 0, 1, 0);
}

```

```
[self addFace:1 withTransform:transform];
//add cube face 3
transform = CATransform3DMakeTranslation(0, -100, 0);
transform = CATransform3DRotate(transform, M_PI_2, 1, 0, 0);
[self addFace:2 withTransform:transform];
//add cube face 4
transform = CATransform3DMakeTranslation(0, 100, 0);
transform = CATransform3DRotate(transform, -M_PI_2, 1, 0, 0);
[self addFace:3 withTransform:transform];
//add cube face 5
transform = CATransform3DMakeTranslation(-100, 0, 0);
transform = CATransform3DRotate(transform, -M_PI_2, 0, 1, 0);
[self addFace:4 withTransform:transform];
//add cube face 6
transform = CATransform3DMakeTranslation(0, 0, -100);
transform = CATransform3DRotate(transform, M_PI, 0, 1, 0);
[self addFace:5 withTransform:transform];
}

@end
```

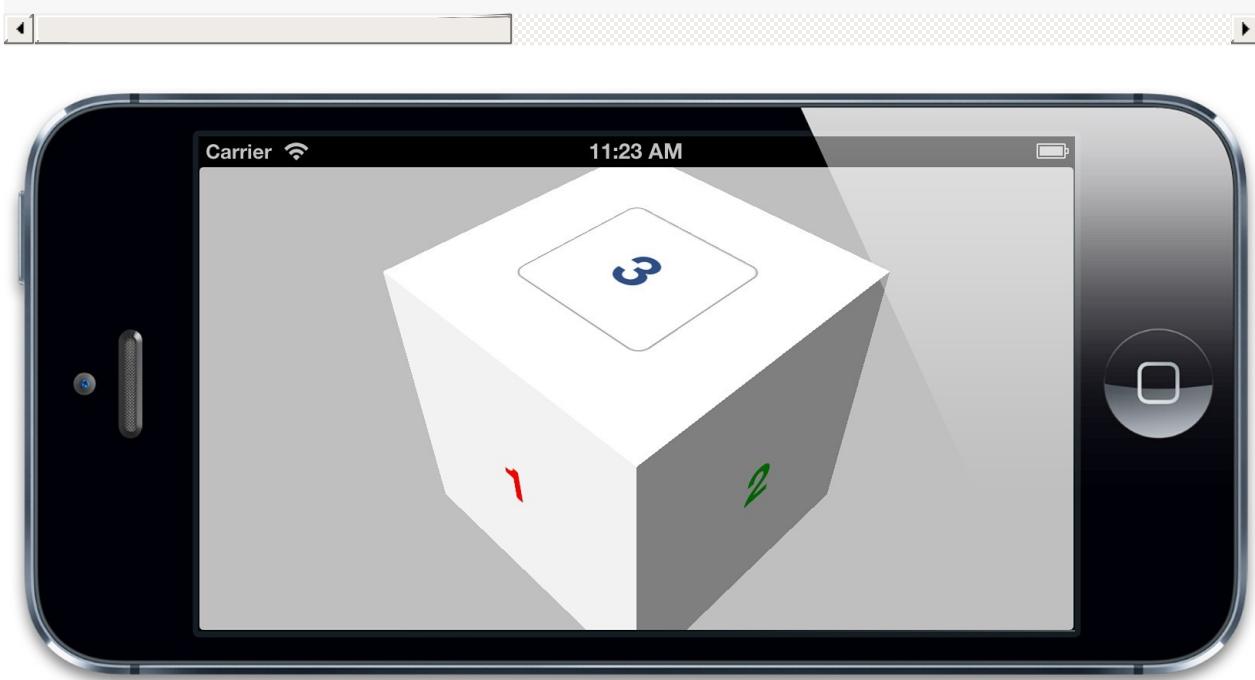


图5.22 动态计算光线效果之后的立方体

## 点击事件

你应该能注意到现在可以在第三个表面的顶部看见按钮了，点击它，什么都没发生，为什么呢？

这并不是因为iOS在3D场景下正确地处理响应事件，实际上是可以做到的。问题在于视图顺序。在第三章中我们简要提到过，点击事件的处理由视图在父视图中的顺序决定的，并不是3D空间中的Z轴顺序。当给立方体添加视图的时候，我们实际上是按照一个顺序添加，所以按照视图/图层顺序来说，4，5，6在3的前面。

即使我们看不见4，5，6的表面（因为被1，2，3遮住了），iOS在事件响应上仍然保持之前的顺序。当试图点击表面3上的按钮，表面4，5，6截断了点击事件（取决于点击的位置），这就和普通的2D布局在按钮上覆盖物体一样。

你也许认为把 `doubleSided` 设置成 `NO` 可以解决这个问题，因为它不再渲染视图后面的内容，但实际上并不起作用。因为背对相机而隐藏的视图仍然会响应点击事件（这和通过设置 `hidden` 属性或者设置 `alpha` 为0而隐藏的视图不同，那两种方式将不会响应事件）。所以即使禁止了双面渲染仍然不能解决这个问题（虽然由于性能问题，还是需要把它设置成 `NO`）。

这里有几种正确的方案：把除了表面3的其他视图 `userInteractionEnabled` 属性都设置成 `NO` 来禁止事件传递。或者简单通过代码把视图3覆盖在视图6上。无论怎样都可以点击按钮了（图5.23）。



图5.23 背景视图不再阻碍按钮，我们可以点击它了

## 总结

这一章涉及了一些2D和3D的变换。你学习了一些矩阵计算的基础，以及如何用Core Animation创建3D场景。你看到了图层背后到底是如何呈现的，并且知道了不能把扁平的图片做成真实的立体效果，最后我们用demo说明了触摸事件的处理，视图中图层添加的层级顺序会比屏幕上显示的顺序更有意义。

第六章我们会研究一些Core Animation提供不同功能的具体的 CALayer 子类。

## 专用图层

复杂的组织都是专门化的

Catharine R. Stimpson

到目前为止，我们已经探讨过 `CALayer` 类了，同时我们也了解到一些非常有用的绘图和动画功能。但是Core Animation图层不仅仅能作用于图片和颜色而已。本章就会学习其他的一些图层类，进一步扩展使用Core Animation绘图的能力。

# CAShapeLayer

在第四章『视觉效果』我们学习到了不使用图片的情况下用 `CGPath` 去构造任意形状的阴影。如果我们能用同样的方式创建相同形状的图层就好了。

`CAShapeLayer` 是一个通过矢量图形而不是 `bitmap` 来绘制的图层子类。你指定诸如颜色和线宽等属性，用 `CGPath` 来定义想要绘制的图形，最后 `CAShapeLayer` 就自动渲染出来了。当然，你也可以用 `Core Graphics` 直接向原始的 `CALayer` 的内容中绘制一个路径，相比直下，使用 `CAShapeLayer` 有以下一些优点：

- 渲染快速。`CAShapeLayer` 使用了硬件加速，绘制同一图形会比用 `Core Graphics` 快很多。
- 高效使用内存。一个 `CAShapeLayer` 不需要像普通 `CALayer` 一样创建一个寄宿图形，所以无论有多大，都不会占用太多的内存。
- 不会被图层边界剪裁掉。一个 `CAShapeLayer` 可以在边界之外绘制。你的图层路径不会像在使用 `Core Graphics` 的普通 `CALayer` 一样被剪裁掉（如我们在第二章所见）。
- 不会出现像素化。当你给 `CAShapeLayer` 做 3D 变换时，它不像一个有寄宿图的普通图层一样变得像素化。

## 创建一个 `CGPath`

`CAShapeLayer` 可以用来绘制所有能够通过 `CGPath` 来表示的形状。这个形状不一定要闭合，图层路径也不一定要不可破，事实上你可以在一个图层上绘制好几个不同的形状。你可以控制一些属性比如 `lineWidth`（线宽，用点表示单位），`lineCap`（线条结尾的样子），和 `lineJoin`（线条之间的结合点的样子）；但是在图层面你只有一次机会设置这些属性。如果你想用不同颜色或风格来绘制多个形状，就不得不为每个形状准备一个图层了。

清单 6.1 的代码用一个 `CAShapeLayer` 渲染一个简单的火柴人。`CAShapeLayer` 属性是 `CGPathRef` 类型，但是我们用 `UIBezierPath` 帮助类创建了图层路径，这样我们就不用考虑人工释放 `CGPath` 了。图 6.1 是代码运行的结果。虽然还不是很完美，但是总算知道了大意对吧！

清单 6.1 用 `CAShapeLayer` 绘制一个火柴人

```
#import "DrawingView.h"
#import

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create path
 UIBezierPath *path = [[UIBezierPath alloc] init];
 [path moveToPoint:CGPointMake(175, 100)];

 [path addArcWithCenter:CGPointMake(150, 100) radius:25 startAngle:M_PI_4 endAngle:M_PI];
 [path moveToPoint:CGPointMake(150, 125)];
 [path addLineToPoint:CGPointMake(150, 175)];
 [path addLineToPoint:CGPointMake(125, 225)];
 [path moveToPoint:CGPointMake(150, 175)];
 [path addLineToPoint:CGPointMake(175, 225)];
 [path moveToPoint:CGPointMake(100, 150)];
 [path addLineToPoint:CGPointMake(200, 150)];

 //create shape layer
 CAShapeLayer *shapeLayer = [CAShapeLayer layer];
 shapeLayer.strokeColor = [UIColor redColor].CGColor;
 shapeLayer.fillColor = [UIColor clearColor].CGColor;
 shapeLayer.lineWidth = 5;
 shapeLayer.lineJoin = kCALineJoinRound;
 shapeLayer.lineCap = kCALineCapRound;
 shapeLayer.path = path.CGPath;
 //add it to our view
 [self.containerView.layer addSublayer:shapeLayer];
}

@end
```

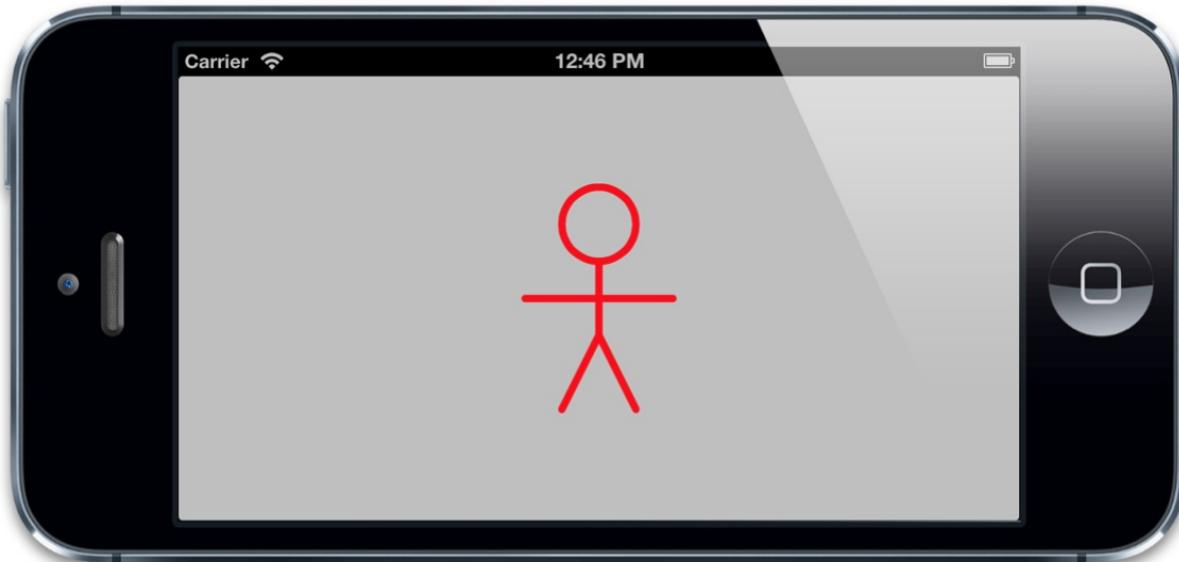


图6.1 用 `CAShapeLayer` 绘制一个简单的火柴人

## 圆角

第二章里面提到了 `CAShapeLayer` 为创建圆角视图提供了一个方法，就是 `CALayer` 的 `cornerRadius` 属性（译者注：其实是在第四章提到的）。虽然使用 `CAShapeLayer` 类需要更多的工作，但是它有一个优势就是可以单独指定每个角。

我们创建圆角矩形其实就是人工绘制单独的直线和弧度，但是事实上 `UIBezierPath` 有自动绘制圆角矩形的构造方法，下面这段代码绘制了一个有三个圆角一个直角的矩形：

```
//define path parameters
CGRect rect = CGRectMake(50, 50, 100, 100);
CGSize radii = CGSizeMake(20, 20);
UIRectCorner corners = UIRectCornerTopRight | UIRectCornerBottomRight;
//create path
UIBezierPath *path = [UIBezierPath bezierPathWithRoundedRect:rect
 cornerRadius:radii];
path.
```

我们可以通过这个图层路径绘制一个既有直角又有圆角的视图。如果我们想依照此图形来剪裁视图内容，我们可以把 `CAShapeLayer` 作为视图的宿主图层，而不是添加一个子视图（图层蒙板的详细解释见第四章『视觉效果』）。

## CATextLayer

用户界面是无法从一个单独的图片里面构建的。一个设计良好的图标能够很好地表现一个按钮或控件的意图，不过你迟早都要需要一个不错的老式风格的文本标签。

如果你想在一个图层里面显示文字，完全可以借助图层代理直接将字符串使用Core Graphics写入图层的内容（这就是UILabel的精髓）。如果越过寄宿于图层的视图，直接在图层上操作，那其实相当繁琐。你要为每一个显示文字的图层创建一个能像图层代理一样工作的类，还要逻辑上判断哪个图层需要显示哪个字符串，更别提还要记录不同的字体，颜色等一系列乱七八糟的东西。

万幸的是这些都是不必要的，Core Animation提供了一个 CALayer 的子类 CATextLayer，它以图层的形式包含了 UILabel 几乎所有的绘制特性，并且额外提供了一些新的特性。

同样，CATextLayer 也要比 UILabel 渲染得快得多。很少有人知道在iOS 6及之前的版本，UILabel 其实是通过WebKit来实现绘制的，这样就造成了当有很多文字的时候就会有极大的性能压力。而 CATextLayer 使用了Core text，并且渲染得非常快。

让我们来尝试用 CATextLayer 来显示一些文字。清单6.2的代码实现了这一功能，结果如图6.2所示。

清单6.2 用 CATextLayer 来实现一个 UILabel

```
@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *labelView;

@end

@implementation ViewController
- (void)viewDidLoad
{
 [super viewDidLoad];

 //create a text layer
 CATextLayer *textLayer = [CATextLayer layer];
 textLayer.frame = self.labelView.bounds;
 [self.labelView.layer addSublayer:textLayer];

 //set text attributes
 textLayer.foregroundColor = [UIColor blackColor].CGColor;
 textLayer.alignmentMode = kCAAlignmentJustified;
 textLayer.wrapped = YES;

 //choose a font
 UIFont *font = [UIFont systemFontOfSize:15];

 //set layer font
 CFStringRef fontName = (__bridge CFStringRef)font.fontName;
 CGFontRef fontRef = CGFontCreateWithName(fontName);
 textLayer.font = fontRef;
 textLayer.fontSize = font.pointSize;
 CGFontRelease(fontRef);

 //choose some text
 NSString *text = @"Lorem ipsum dolor sit amet, consectetur adipis

 //set layer text
 textLayer.string = text;
}
@end
```

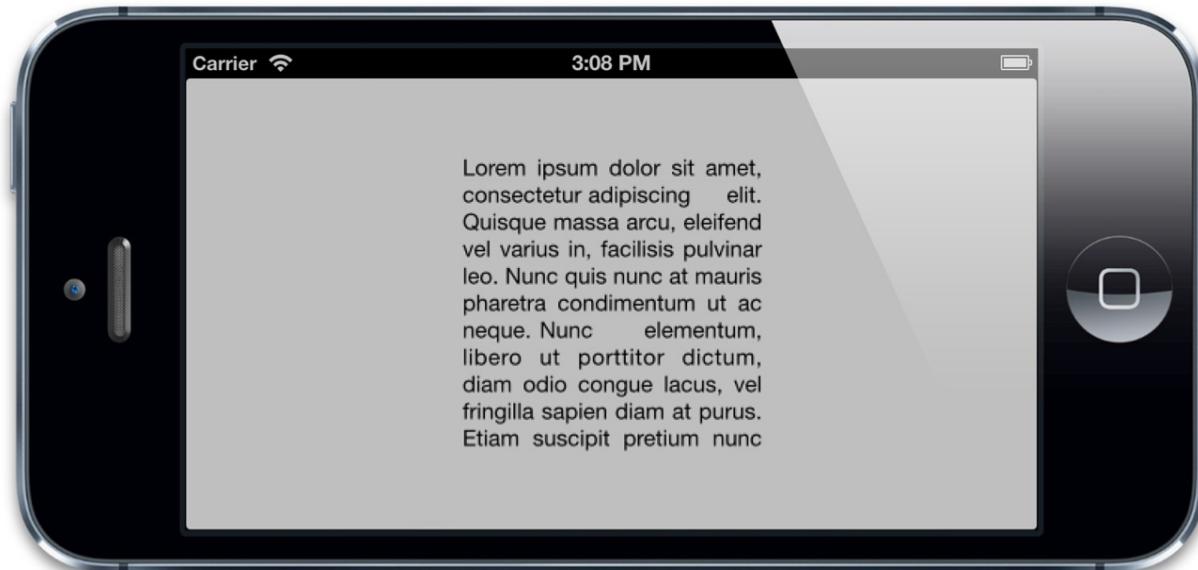
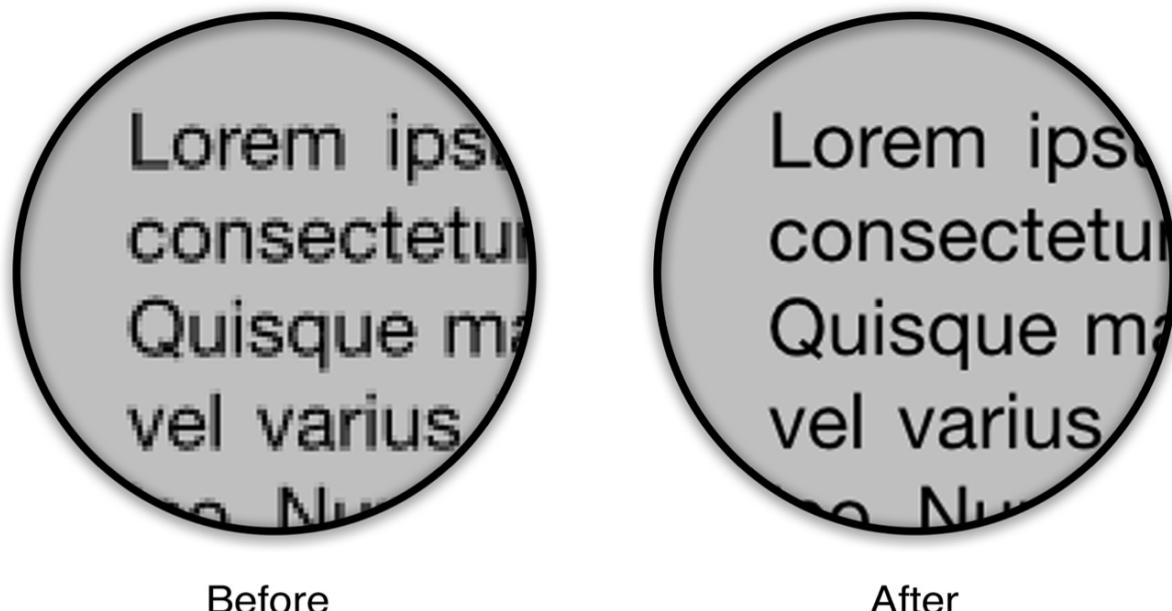


图6.2 用 `CATextLayer` 来显示一个纯文本标签

如果你仔细看这个文本，你会发现一个奇怪的地方：这些文本有一些像素化了。这是因为并没有以Retina的方式渲染，第二章提到了这个 `contentsScale` 属性，用来决定图层内容应该以怎样的分辨率来渲染。`contentsScale` 并不关心屏幕的拉伸因素而总是默认为1.0。如果我们想以Retina的质量来显示文字，我们就得手动地设置 `CATextLayer` 的 `contentsScale` 属性，如下：

```
textLayer.contentsScale = [UIScreen mainScreen].scale;
```

这样就解决了这个问题（如图6.3）



### 图6.3 设置 contentsScale 来匹配屏幕

CATextLayer 的 font 属性不是一个 UIFont 类型，而是一个 CFTypRef 类型。这样可以根据你的具体需要来决定字体属性应该是用 CGFontRef 类型还是 CTFontRef 类型（Core Text字体）。同时字体大小也是用 fontSize 属性单独设置的，因为 CTFontRef 和 CGFontRef 并不像UIFont一样包含点大小。这个例子会告诉你如何将 UIFont 转换成 CGFontRef。

另外，CATextLayer 的 string 属性并不是你想象的 NSString 类型，而是 id 类型。这样你既可以用 NSString 也可以用 NSAttributedString 来指定文本了（注意，NSAttributedString 并不是 NSString 的子类）。属性化字符串是iOS用来渲染字体风格的机制，它以特定的方式来决定指定范围内的字符串的原始信息，比如字体，颜色，字重，斜体等。

## 富文本

iOS 6中，Apple给 UILabel 和其他UIKit文本视图添加了直接的属性化字符串的支持，应该说这是一个很方便的特性。不过事实上从iOS3.2开始 CATextLayer 就已经支持属性化字符串了。这样的话，如果你想要支持更低版本的iOS系统，CATextLayer 无疑是你向界面中增加富文本的好办法，而且也不用去跟复杂的Core Text打交道，也省了用 UIWebView 的麻烦。

让我们编辑一下示例使用到 NSAttributedString（见清单6.3）。iOS 6及以上我们可以用新的 NSTextAttributeName 实例来设置我们的字符串属性，但是练习的目的是为了演示在iOS 5及以下，所以我们用了Core Text，也就是说你需要把Core Text framework添加到你的项目中。否则，编译器是无法识别属性常量的。

### 图6.4是代码运行结果（注意那个红色的下划线文本）

清单6.3 用NSAttributedString实现一个富文本标签。

```
#import "DrawingView.h"
#import
#import

@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *labelView;
```

```
@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //create a text layer
 CATextLayer *textLayer = [CATextLayer layer];
 textLayer.frame = self.labelView.bounds;
 textLayer.contentsScale = [UIScreen mainScreen].scale;
 [self.labelView.layer addSublayer:textLayer];

 //set text attributes
 textLayer.alignmentMode = kCAAlignmentJustified;
 textLayer.wrapped = YES;

 //choose a font
 UIFont *font = [UIFont systemFontOfSize:15];

 //choose some text
 NSString *text = @"Lorem ipsum dolor sit amet, consectetur adipis";

 //create attributed string
 NSMutableAttributedString *string = nil;
 string = [[NSMutableAttributedString alloc] initWithString:text];

 //convert UIFont to a CTFont
 CFStringRef fontName = (__bridge CFStringRef)font.fontName;
 CGFloat fontSize = font.pointSize;
 CTFontRef fontRef = CTFontCreateWithName(fontName, fontSize, NULL);

 //set text attributes
 NSDictionary *attribs = @{
 (__bridge id)kCTForegroundColorAttributeName:(__bridge id)[UIColor redColor]
 (__bridge id)kCTFontAttributeName: (__bridge id)fontRef
 };

 [string setAttributes:attribs range:NSMakeRange(0, [text length])]
```

```

attribs = @{
 (__bridge id)kCTForegroundColorAttributeName: (__bridge id)[UIColor redColor],
 (__bridge id)kCTUnderlineStyleAttributeName: @(kCTUnderlineStyleSingle),
 (__bridge id)kCTFontAttributeName: (__bridge id)fontRef
};

[string setAttributes:attribs range:NSMakeRange(6, 5)];

//release the CTFont we created earlier
CFRelease(fontRef);

//set layer text
textLayer.string = string;
}
@end

```

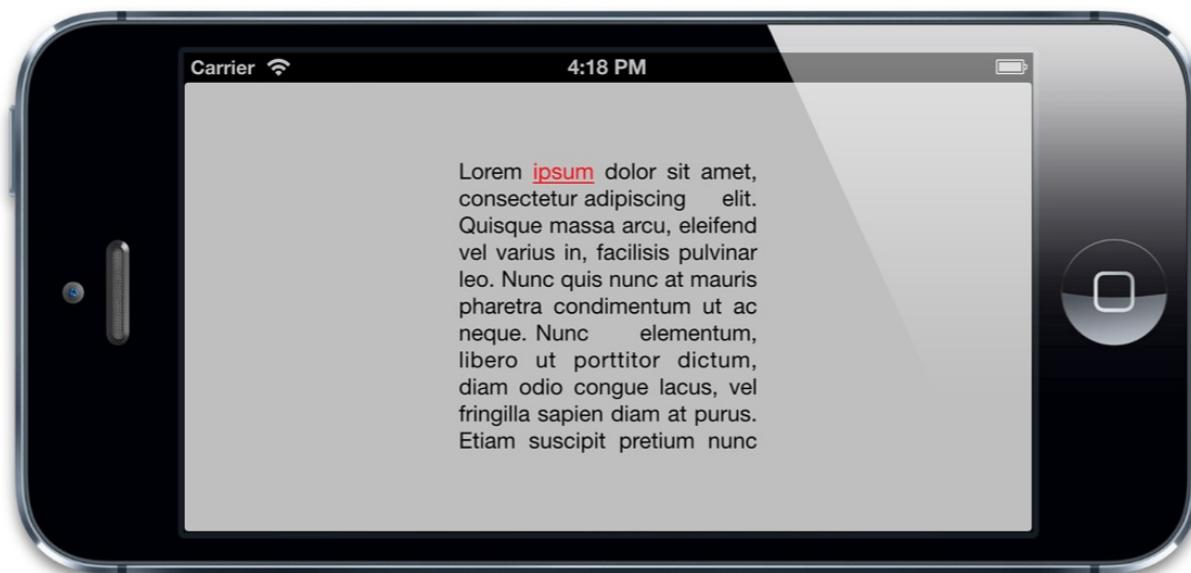


图6.4 用CATextLayer实现一个富文本标签。

## 行距和字距

有必要提一下的是，由于绘制的实现机制不同（Core Text和WebKit），用 `CATextLayer` 渲染和用 `UILabel` 渲染出的文本行距和字距也不是不尽相同的。

二者的差异程度（由使用的字体和字符决定）总的来说挺小，但是如果你想正确的显示普通便签和 `CATextLayer` 就一定要记住这一点。

## UILabel 的替代品

我们已经证实了 `CATextLayer` 比 `UILabel` 有着更好的性能表现，同时还有额外的布局选项并且在iOS 5上支持富文本。但是与一般的标签比较而言会更加繁琐一些。如果我们真的在需求一个 `UILabel` 的可用替代品，最好是能够在Interface Builder上创建我们的标签，而且尽可能地像一般的视图一样正常工作。

我们应该继承 `UILabel`，然后添加一个子图层 `CATextLayer` 并重写显示文本的方法。但是仍然会有由 `UILabel` 的 `-drawRect:` 方法创建的空寄宿图。而且由于 `CALayer` 不支持自动缩放和自动布局，子视图并不是主动跟踪视图边界的大 小，所以每次视图大小被更改，我们不得不手动更新子图层的边界。

我们真正想要的是一个用 `CATextLayer` 作为宿主图层的 `UILabel` 子类，这样就可以随着视图自动调整大小而且也没有冗余的寄宿图啦。

就像我们在第一章『图层树』讨论的一样，每一个 `UIView` 都是寄宿在一个 `CALayer` 的示例上。这个图层是由视图自动创建和管理的，那我们可以用别的图层类型替代它么？一旦被创建，我们就无法代替这个图层了。但是如果继承了 `UIView`，那我们就可以重写 `+layerClass` 方法使得在创建的时候能返回一个不同的图层子类。`UIView` 会在初始化的时候调用 `+layerClass` 方法，然后用它的返回类型来创建宿主图层。

清单6.4 演示了一个 `UILabel` 子类 `LayerLabel` 用 `CATextLayer` 绘制它的问题，而不是调用一般的 `UILabel` 使用的较慢的 `-drawRect:` 方法。`LayerLabel` 示例既可以用代码实现，也可以在Interface Builder实现，只要把普通的标签拖入视图之中，然后设置它的类是`LayerLabel`就可以了。

清单6.4 使用 `CATextLayer` 的 `UILabel` 子类：`LayerLabel`

```
#import "LayerLabel.h"
#import

@implementation LayerLabel
+ (Class)layerClass
{
 //this makes our label create a CATextLayer //instead of a regular
 return [CATextLayer class];
}
```

```
- (CATextLayer *)textLayer
{
 return (CATextLayer *)self.layer;
}

- (void)setUp
{
 //set defaults from UILabel settings
 self.text = self.text;
 self.textColor = self.textColor;
 self.font = self.font;

 //we should really derive these from the UILabel settings too
 //but that's complicated, so for now we'll just hard-code them
 [self textLayer].alignmentMode = kCAAlignmentJustified;

 [self textLayer].wrapped = YES;
 [self.layer display];
}

- (id)initWithFrame:(CGRect)frame
{
 //called when creating label programmatically
 if (self = [super initWithFrame:frame]) {
 [self setUp];
 }
 return self;
}

- (void)awakeFromNib
{
 //called when creating label using Interface Builder
 [self setUp];
}

- (void)setText:(NSString *)text
{
 super.text = text;
 //set layer text
 [self textLayer].string = text;
```

```
}

- (void)setTextColor:(UIColor *)textColor
{
 super.textColor = textColor;
 //set layer text color
 [self textLayer].foregroundColor = textColor.CGColor;
}

- (void)setFont:(UIFont *)font
{
 super.font = font;
 //set layer font
 CFStringRef fontName = (__bridge CFStringRef)font.fontName;
 CGFontRef fontRef = CGFontCreateWithName(fontName);
 [self textLayer].font = fontRef;
 [self textLayer].fontSize = font.pointSize;

 CGFontRelease(fontRef);
}
@end
```

如果你运行代码，你会发现文本并没有像素化，而我们也没有设置 `contentsScale` 属性。把 `CATextLayer` 作为宿主图层的另一好处就是视图自动设置了 `contentsScale` 属性。

在这个简单的例子中，我们只是实现了 `UILabel` 的一部分风格和布局属性，不过稍微再改进一下我们就可以创建一个支持 `UILabel` 所有功能甚至更多功能的 `LayerLabel` 类（你可以在一些线上的开源项目中找到）。

如果你打算支持iOS 6及以上，基于 `CATextLayer` 的标签可能就有有些局限性。但是总得来说，如果想在app里面充分利用 `CALayer` 子类，用 `+layerClass` 来创建基于不同图层的视图是一个简单可复用的方法。

# CATransformLayer

当我们在构造复杂的3D事物的时候，如果能够组织独立元素就太方便了。比如说，你想创造一个孩子的手臂：你就需要确定哪一部分是孩子的手腕，哪一部分是孩子的前臂，哪一部分是孩子的肘，哪一部分是孩子的上臂，哪一部分是孩子的肩膀等等。

当然是允许独立地移动每个区域的啦。以肘为指点会移动前臂和手，而不是肩膀。Core Animation图层很容易就可以让你在2D环境下做出这样的层级体系下的变换，但是3D情况下就不太可能，因为所有的图层都把他的孩子都平面化到一个场景中（第五章『变换』有提到）。

CATransformLayer 解决了这个问题，CATransformLayer 不同于普通的 CALayer，因为它不能显示它自己的内容。只有当存在了一个能作用域子图层的变换它才真正存在。CATransformLayer 并不平面化它的子图层，所以它能够用于构造一个层级的3D结构，比如我的手臂示例。

用代码创建一个手臂需要相当多的代码，所以我就演示得更简单一些吧：在第五章的立方体示例，我们将通过旋转 camera 来解决图层平面化问题而不是像立方体示例代码中用的 sublayerTransform 。这是一个非常不错的技巧，但是只能作用域单个对象上，如果你的场景包含两个立方体，那我们就不能用这个技巧单独旋转他们了。

那么，就让我们来试一试 CATransformLayer 吧，第一个问题就来了：在第五章，我们是用多个视图来构造了我们的立方体，而不是单独的图层。我们不能在不打乱已有的视图层次的前提下在一个本身不是有寄宿图的图层中放置一个寄宿图图层。我们可以创建一个新的 UIView 子类寄宿

在 CATransformLayer （用 +layerClass 方法）之上。但是，为了简化案例，我们仅仅重建了一个单独的图层，而不是使用视图。这意味着我们不能像第五章一样在立方体表面显示按钮和标签，不过我们现在也用不到这个特性。

清单6.5就是代码。我们以我们在第五章使用过的相同基本逻辑放置立方体。但是并不像以前那样直接将立方面添加到容器视图的宿主图层，我们将他们放置到一个 CATransformLayer 中创建一个独立的立方体对象，然后将两个这样的立方体放进容器中。我们随机地给立方面染色以将他们区分开来，这样就不用靠标签或是光亮来区分他们。图6.5是运行结果。

## 清单6.5 用 CATransformLayer 装配一个3D图层体系

```
@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (CALayer *)faceWithTransform:(CATransform3D)transform
{
 //create cube face layer
 CALayer *face = [CALayer layer];
 face.frame = CGRectMake(-50, -50, 100, 100);

 //apply a random color
 CGFloat red = (rand() / (double)INT_MAX);
 CGFloat green = (rand() / (double)INT_MAX);
 CGFloat blue = (rand() / (double)INT_MAX);
 face.backgroundColor = [UIColor colorWithRed:red green:green blue:

 //apply the transform and return
 face.transform = transform;
 return face;
}

- (CALayer *)cubeWithTransform:(CATransform3D)transform
{
 //create cube layer
 CATransformLayer *cube = [CATransformLayer layer];

 //add cube face 1
 CATransform3D ct = CATransform3DMakeTranslation(0, 0, 50);
 [cube addSublayer:[self faceWithTransform:ct]];

 //add cube face 2
 ct = CATransform3DMakeTranslation(50, 0, 0);
 ct = CATransform3DRotate(ct, M_PI_2, 0, 1, 0);
 [cube addSublayer:[self faceWithTransform:ct]];
}
```

```
//add cube face 3
ct = CATransform3DMakeTranslation(0, -50, 0);
ct = CATransform3DRotate(ct, M_PI_2, 1, 0, 0);
[cube addSublayer:[self faceWithTransform:ct]];

//add cube face 4
ct = CATransform3DMakeTranslation(0, 50, 0);
ct = CATransform3DRotate(ct, -M_PI_2, 1, 0, 0);
[cube addSublayer:[self faceWithTransform:ct]];

//add cube face 5
ct = CATransform3DMakeTranslation(-50, 0, 0);
ct = CATransform3DRotate(ct, -M_PI_2, 0, 1, 0);
[cube addSublayer:[self faceWithTransform:ct]];

//add cube face 6
ct = CATransform3DMakeTranslation(0, 0, -50);
ct = CATransform3DRotate(ct, M_PI, 0, 1, 0);
[cube addSublayer:[self faceWithTransform:ct]];

//center the cube layer within the container
CGSize containerSize = self.containerView.bounds.size;
cube.position = CGPointMake(containerSize.width / 2.0, containerSize.height / 2.0);

//apply the transform and return
cube.transform = transform;
return cube;
}

- (void)viewDidLoad
{
 [super viewDidLoad];

 //set up the perspective transform
 CATransform3D pt = CATransform3DIdentity;
 pt.m34 = -1.0 / 500.0;
 self.containerView.layer.sublayerTransform = pt;

 //set up the transform for cube 1 and add it
}
```

```
CATransform3D c1t = CATransform3DIdentity;
c1t = CATransform3DTranslate(c1t, -100, 0, 0);
CALayer *cube1 = [self cubeWithTransform:c1t];
[self.containerView.layer addSublayer:cube1];

//set up the transform for cube 2 and add it
CATransform3D c2t = CATransform3DIdentity;
c2t = CATransform3DTranslate(c2t, 100, 0, 0);
c2t = CATransform3DRotate(c2t, -M_PI_4, 1, 0, 0);
c2t = CATransform3DRotate(c2t, -M_PI_4, 0, 1, 0);
CALayer *cube2 = [self cubeWithTransform:c2t];
[self.containerView.layer addSublayer:cube2];
}

@end
```

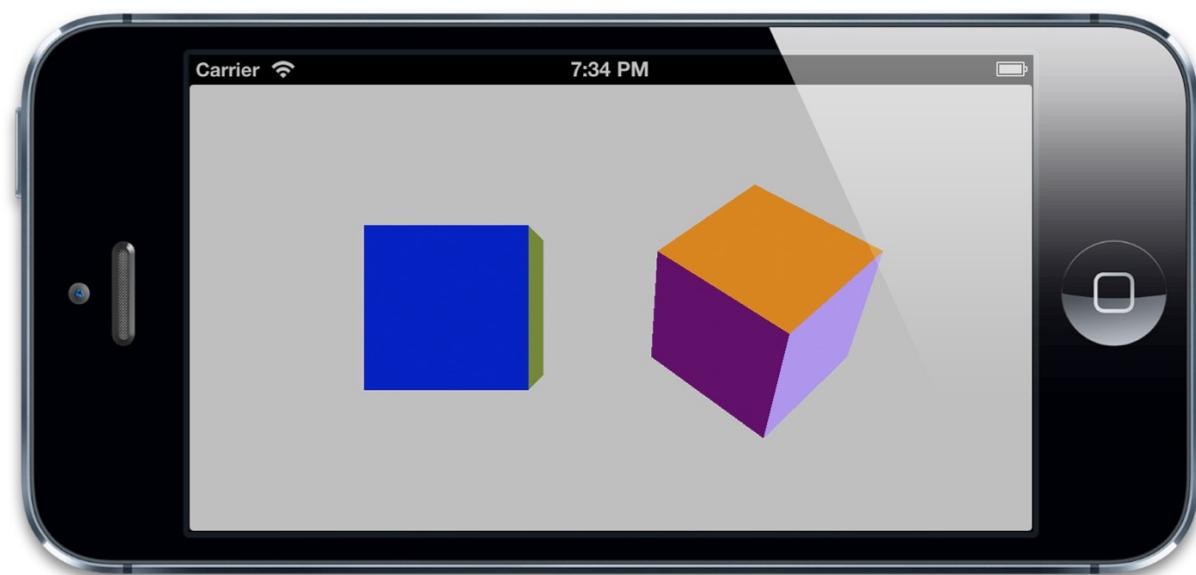


图6.5 同一视角下的俩不同变换的立方体

## CAGradientLayer

`CAGradientLayer` 是用来生成两种或更多颜色平滑渐变的。用Core Graphics复制一个 `CAGradientLayer` 并将内容绘制到一个普通图层的寄宿图也是有可能的，但是 `CAGradientLayer` 的真正好处在于绘制使用了硬件加速。

### 基础渐变

我们将从一个简单的红变蓝的对角线渐变开始（见清单6.6）。这些渐变色彩放在一个数组中，并赋给 `colors` 属性。这个数组成员接受 `CGColorRef` 类型的值（并不是从 `NSObject` 派生而来），所以我们要用通过bridge转换以确保编译正常。

`CAGradientLayer` 也有 `startPoint` 和 `endPoint` 属性，他们决定了渐变的方向。这两个参数是以单位坐标系进行的定义，所以左上角坐标是 $\{0, 0\}$ ，右下角坐标是 $\{1, 1\}$ 。代码运行结果如图6.6

清单6.6 简单的两种颜色的对角线渐变

```
@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create gradient layer and add it to our container view
 CAGradientLayer *gradientLayer = [CAGradientLayer layer];
 gradientLayer.frame = self.containerView.bounds;
 [self.containerView.layer addSublayer:gradientLayer];

 //set gradient colors
 gradientLayer.colors = @[(__bridge id)[UIColor redColor].CGColor,

 //set gradient start and end points
 gradientLayer.startPoint = CGPointMake(0, 0);
 gradientLayer.endPoint = CGPointMake(1, 1);
}
@end
```

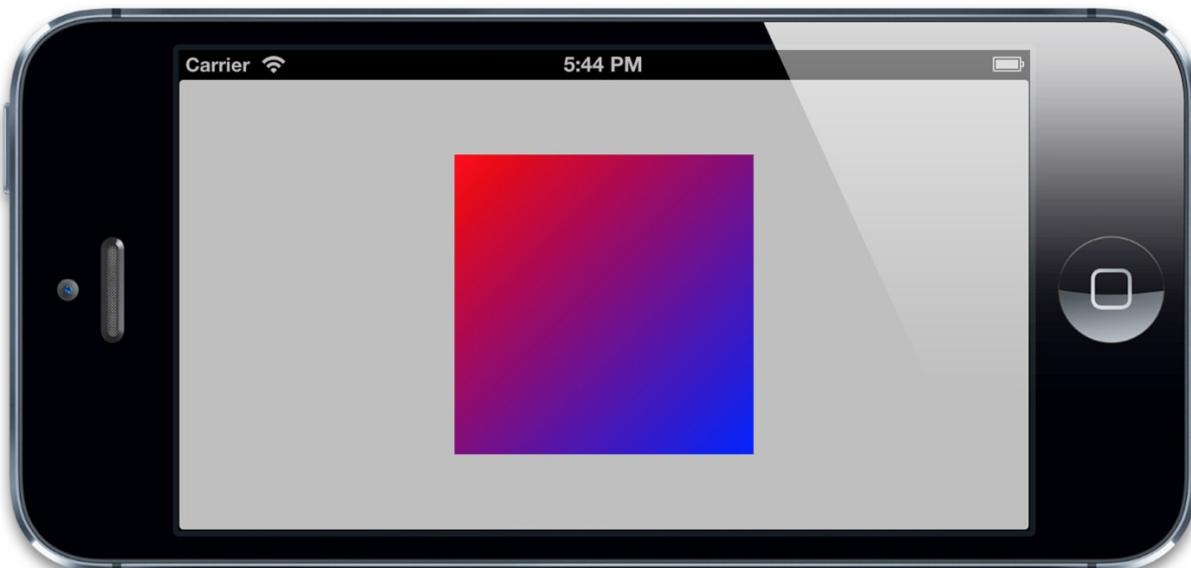


图6.6 用 `CAGradientLayer` 实现简单的两种颜色的对角线渐变

## 多重渐变

如果你愿意，`colors` 属性可以包含很多颜色，所以创建一个彩虹一样的多重渐变也是很简单的。默认情况下，这些颜色在空间上均匀地被渲染，但是我们可以用 `locations` 属性来调整空间。`locations` 属性是一个浮点数值的数组（以 `NSNumber` 包装）。这些浮点数定义了 `colors` 属性中每个不同颜色的位置，同样的，也是以单位坐标系进行标定。0.0代表着渐变的开始，1.0代表着结束。

`locations` 数组并不是强制要求的，但是如果你给它赋值了就一定要确保 `locations` 的数组大小和 `colors` 数组大小一定要相同，否则你将会得到一个空白的渐变。

清单6.7展示了一个基于清单6.6的对角线渐变的代码改造。现在变成了从红到黄最后到绿色的渐变。`locations` 数组指定了0.0，0.25和0.5三个数值，这样这三个渐变就有点像挤在了左上角。（如图6.7）。

清单6.7 在渐变上使用 `locations`

```
- (void)viewDidLoad {
 [super viewDidLoad];

 //create gradient layer and add it to our container view
 CAGradientLayer *gradientLayer = [CAGradientLayer layer];
 gradientLayer.frame = self.containerView.bounds;
 [self.containerView.layer addSublayer:gradientLayer];

 //set gradient colors
 gradientLayer.colors = @[(__bridge id)[UIColor redColor].CGColor,
 (__bridge id)[UIColor yellowColor].CGColor,
 (__bridge id)[UIColor greenColor].CGColor];

 //set locations
 gradientLayer.locations = @[@0.0, @0.25, @0.5];

 //set gradient start and end points
 gradientLayer.startPoint = CGPointMake(0, 0);
 gradientLayer.endPoint = CGPointMake(1, 1);
}
```

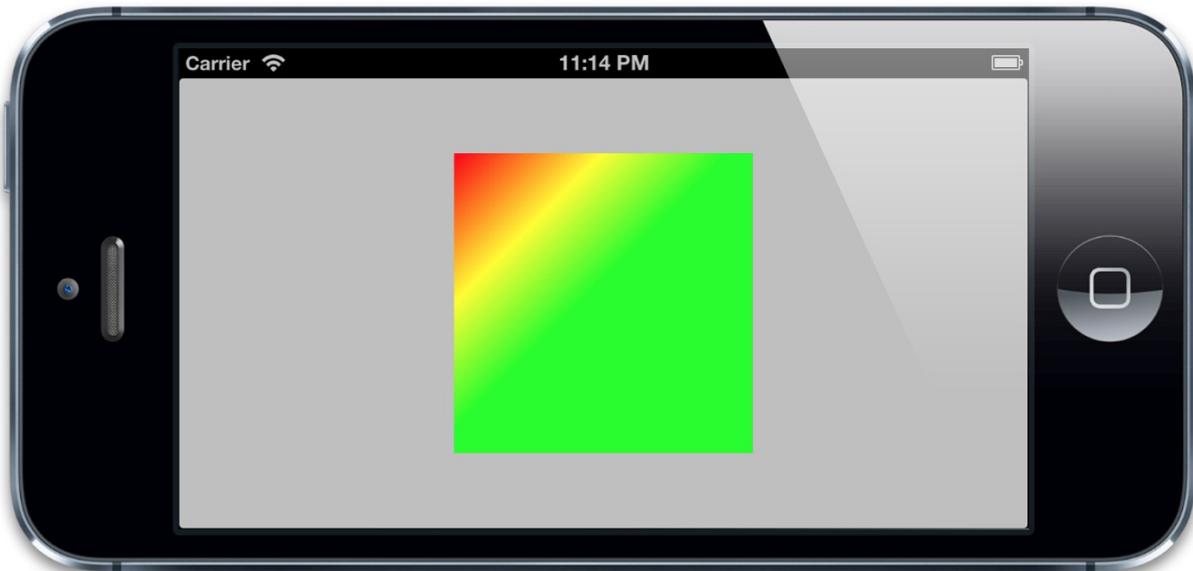


图6.7 用 `locations` 构造偏移至左上角的三色渐变

## CAReplicatorLayer

`CAReplicatorLayer` 的目的是为了高效生成许多相似的图层。它会绘制一个或多个图层的子图层，并在每个复制体上应用不同的变换。看上去演示能够更加解释这些，我们来写个例子吧。

### 重复图层（Repeating Layers）

清单6.8中，我们在屏幕的中间创建了一个小白色方块图层，然后用 `CAReplicatorLayer` 生成十个图层组成一个圆圈。`instanceCount` 属性指定了图层需要重复多少次。`instanceTransform` 指定了一个 `CATransform3D` 3D 变换（这种情况下，下一图层的位移和旋转将会移动到圆圈的下一个点）。

变换是逐步增加的，每个实例都是相对于前一实例布局。这就是为什么这些复制体最终不会出现在同意位置上，图6.8是代码运行结果。

清单6.8 用 `CAReplicatorLayer` 重复图层

```
@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController
- (void)viewDidLoad
{
 [super viewDidLoad];
 //create a replicator layer and add it to our view
 CAReplicatorLayer *replicator = [CAReplicatorLayer layer];
 replicator.frame = self.containerView.bounds;
 [self.containerView.layer addSublayer:replicator];

 //configure the replicator
 replicator.instanceCount = 10;

 //apply a transform for each instance
 CATransform3D transform = CATransform3DIdentity;
 transform = CATransform3DTranslate(transform, 0, 200, 0);
 transform = CATransform3DRotate(transform, M_PI / 5.0, 0, 0, 1);
 transform = CATransform3DTranslate(transform, 0, -200, 0);
 replicator.instanceTransform = transform;

 //apply a color shift for each instance
 replicator.instanceBlueOffset = -0.1;
 replicator.instanceGreenOffset = -0.1;

 //create a sublayer and place it inside the replicator
 CALayer *layer = [CALayer layer];
 layer.frame = CGRectMake(100.0f, 100.0f, 100.0f, 100.0f);
 layer.backgroundColor = [UIColor whiteColor].CGColor;
 [replicator addSublayer:layer];
}
@end
```

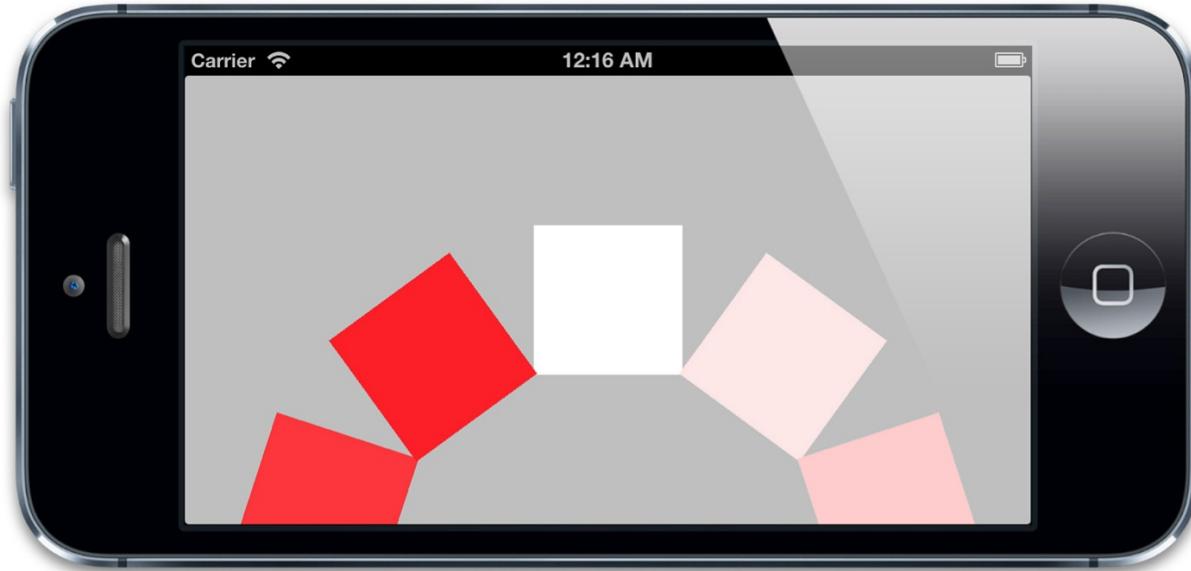


图6.8 用 CAReplicatorLayer 创建一圈图层

注意到当图层在重复的时候，他们的颜色也在变化：这是用 `instanceBlueOffset` 和 `instanceGreenOffset` 属性实现的。通过逐步减少蓝色和绿色通道，我们逐渐将图层颜色转换成了红色。这个复制效果看起来很酷，但是 `CAReplicatorLayer` 真正应用到实际程序上的场景比如：一个游戏中导弹的轨迹云，或者粒子爆炸（尽管iOS 5已经引入了 `CAEmitterLayer`，它更适合创建任意的粒子效果）。除此之外，还有一个实际应用是：反射。

## 反射

使用 `CAReplicatorLayer` 并应用一个负比例变换于一个复制图层，你就可以创建指定视图（或整个视图层次）内容的镜像图片，这样就创建了一个实时的『反射』效果。让我们来尝试实现这个创意：指定一个继承于 `UIView` 的 `ReflectionView`，它会自动产生内容的反射效果。实现这个效果的代码很简单（见清单6.9），实际上用 `ReflectionView` 实现这个效果会更简单，我们只需要把 `ReflectionView` 的实例放置于Interface Builder（见图6.9），它就会实时生成子视图的反射，而不需要别的代码（见图6.10）。

清单6.9 用 CAReplicatorLayer 自动绘制反射

```
#import "ReflectionView.h"
#import

@implementation ReflectionView
```

```
+ (Class)layerClass
{
 return [CARreplicatorLayer class];
}

- (void)setUp
{
 //configure replicator
 CARreplicatorLayer *layer = (CARreplicatorLayer *)self.layer;
 layer.instanceCount = 2;

 //move reflection instance below original and flip vertically
 CATransform3D transform = CATransform3DIdentity;
 CGFloat verticalOffset = self.bounds.size.height + 2;
 transform = CATransform3DTranslate(transform, 0, verticalOffset, 0);
 transform = CATransform3DScale(transform, 1, -1, 0);
 layer.instanceTransform = transform;

 //reduce alpha of reflection layer
 layer.instanceAlphaOffset = -0.6;
}

- (id)initWithFrame:(CGRect)frame
{
 //this is called when view is created in code
 if ((self = [super initWithFrame:frame])) {
 [self setUp];
 }
 return self;
}

- (void)awakeFromNib
{
 //this is called when view is created from a nib
 [self setUp];
}
@end
```

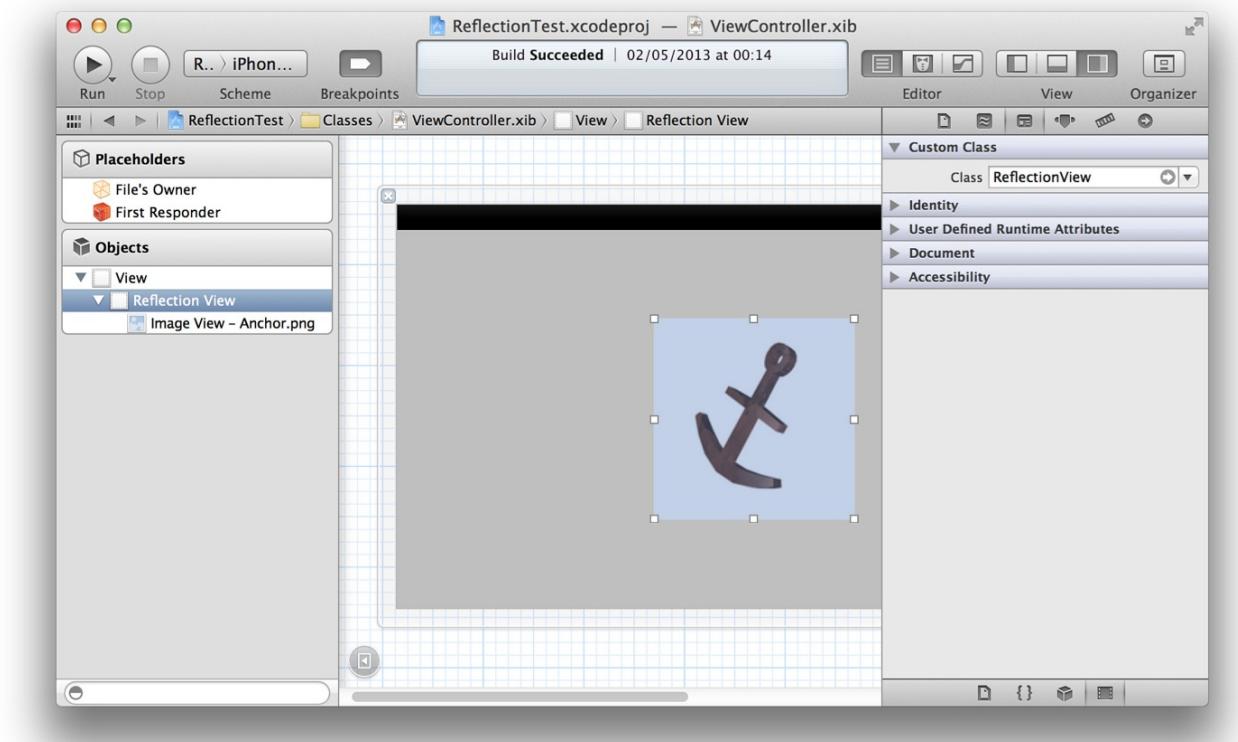


图6.9 在Interface Builder中使用 ReflectionView

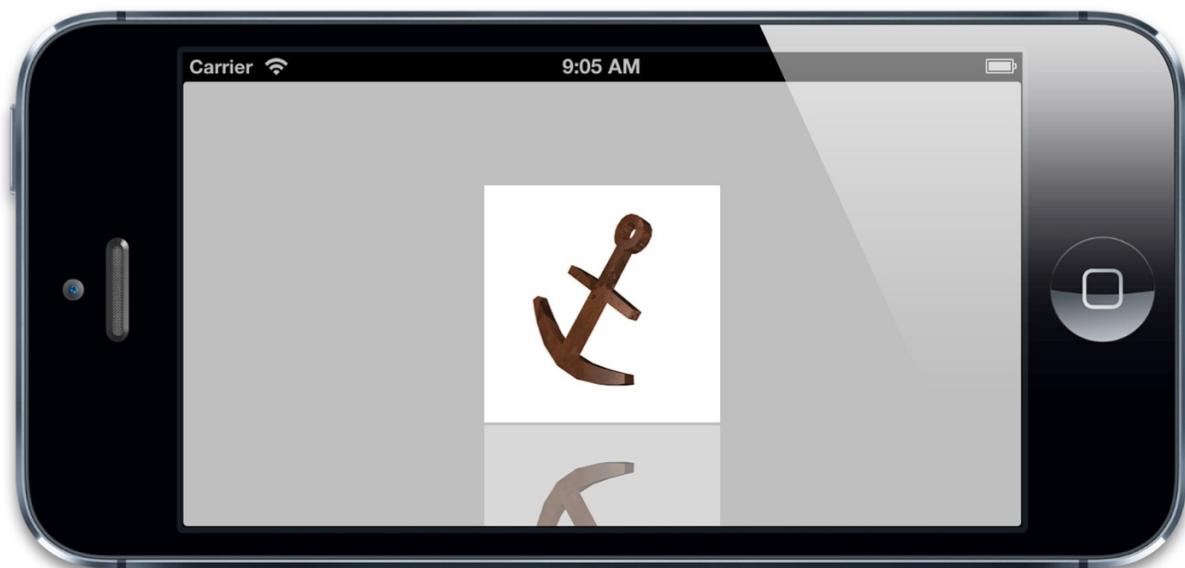


图6.10 ReflectionView 自动实时产生反射效果。

开源代码 `ReflectionView` 完成了一个自适应的渐变淡出效果  
(用 `CAGradientLayer` 和图层蒙板实现) , 代码见  
<https://github.com/nicklockwood/ReflectionView>

## CAScrollLayer

对于一个未转换的图层，它的 `bounds` 和它的 `frame` 是一样的，`frame` 属性是由 `bounds` 属性自动计算而出的，所以更改任意一个值都会更新其他值。

但是如果你只想显示一个大图层里面的一小部分呢。比如说，你可能有一个很大的图片，你希望用户能够随意滑动，或者是一个数据或文本的长列表。在一个典型的 iOS 应用中，你可能会用到 `UITableView` 或是 `UIScrollView`，但是对于独立的图层来说，什么会等价于刚刚提到的 `UITableView` 和 `UIScrollView` 呢？

在第二章中，我们探索了图层的 `contentsRect` 属性的用法，它的确是能够解决在图层中小地方显示大图片的解决方法。但是如果你的图层包含子图层那它就不是一个非常好的解决方案，因为，这样做的话每次你想『滑动』可视区域的时候，你就需要手工重新计算并更新所有的子图层位置。

这个时候就需要 `CAScrollLayer` 了。`CAScrollLayer` 有一个 `- scrollToPoint:` 方法，它自动适应 `bounds` 的原点以便图层内容出现在滑动的地方。注意，这就是它做的所有事情。前面提到过，Core Animation 并不处理用户输入，所以 `CAScrollLayer` 并不负责将触摸事件转换为滑动事件，既不渲染滚动条，也不实现任何 iOS 指定行为例如滑动反弹（当视图滑动超多了它的边界的将会反弹回正确的地方）。

让我们来用 `CAScrollLayer` 来常见一个基本的 `UIScrollView` 替代品。我们将会用 `CAScrollLayer` 作为视图的宿主图层，并创建一个自定义的 `UIView`，然后用 `UIPanGestureRecognizer` 实现触摸事件响应。这段代码见清单 6.10。图 6.11 是运行效果：ScrollView 显示了一个大于它的 `frame` 的 `UIImageView`。

清单 6.10 用 `CAScrollLayer` 实现滑动视图

```
#import "ScrollView.h"
#import <QuartzCore/QuartzCore.h>
+ (Class)layerClass
{
 return [CAScrollLayer class];
}

- (void)setUp
{
```

```
//enable clipping
self.layer.masksToBounds = YES;

//attach pan gesture recognizer
UIPanGestureRecognizer *recognizer = nil;
recognizer = [[UIPanGestureRecognizer alloc] initWithTarget:self action:@selector(handlePan:)];
[self addGestureRecognizer:recognizer];
}

- (id)initWithFrame:(CGRect)frame
{
 //this is called when view is created in code
 if ((self = [super initWithFrame:frame])) {
 [self setUp];
 }
 return self;
}

- (void)awakeFromNib {
 //this is called when view is created from a nib
 [self setUp];
}

- (void)pan:(UIPanGestureRecognizer *)recognizer
{
 //get the offset by subtracting the pan gesture
 //translation from the current bounds origin
 CGPoint offset = self.bounds.origin;
 offset.x -= [recognizer translationInView:self].x;
 offset.y -= [recognizer translationInView:self].y;

 //scroll the layer
 [(CAScrollLayer *)self.layer scrollToPoint:offset];

 //reset the pan gesture translation
 [recognizer setTranslation:CGPointZero inView:self];
}
@end
```



### 图6.11 用 `UIScrollView` 创建一个凑合的滑动视图

不同于 `UIScrollView`，我们定制的滑动视图类并没有实现任何形式的边界检查（bounds checking）。图层内容极有可能滑出视图的边界并无限滑下去。`CAScrollLayer` 并没有等同于 `UIScrollView` 中 `contentSize` 的属性，所以当 `CAScrollLayer` 滑动的时候完全没有一个全局的可滑动区域的概念，也无法自适应它的边界原点至你指定的值。它之所以不能自适应边界大小是因为它不需要，内容完全可以超过边界。

那你一定会奇怪用 `CAScrollLayer` 的意义到底何在，因为你可以简单地用一个普通的 `CALayer` 然后手动适应边界原点啊。真相其实并不复杂，`UIScrollView` 并没有用 `CAScrollLayer`，事实上，就是简单的通过直接操作图层边界来实现滑动。

`CAScrollLayer` 有一个潜在的有用特性。如果你查看 `CAScrollLayer` 的头文件，你就会注意到有一个扩展分类实现了一些方法和属性：

```
- (void)scrollPoint:(CGPoint)p;
- (void)scrollRectToVisible:(CGRect)r;
@property(nonatomic) CGRect visibleRect;
```

看到这些方法和属性名，你也许会以为这些方法给每个 `CALayer` 实例增加了滑动功能。但是事实上他们只是放置在 `CAScrollLayer` 中的图层的实用方法。`scrollPoint:` 方法从图层树中查找并找到第一个可用的 `CAScrollLayer`，然后滑动它使得指定点成为可视的。`scrollRectToVisible:` 方法实现了同样的事情只不过是作用在一个矩形上的。`visibleRect` 属性决定图层（如果存在的话）的哪部分是当前的可视区域。如果你自己实现这些方法就会相对容易明白一点，但是 `CAScrollLayer` 帮你省了这些麻烦，所以当涉及到实现图层滑动的时候就可以用上了。

## CATiledLayer

有些时候你可能需要绘制一个很大的图片，常见的例子就是一个高像素的照片或者是地球表面的详细地图。iOS应用通畅运行在内存受限的设备上，所以读取整个图片到内存中是不明智的。载入大图可能会相当地慢，那些对你看上去比较方便的做法（在主线程调用 `UIImage` 的 `-imageNamed:` 方法或者 `-imageWithContentsOfFile:` 方法）将会阻塞你的用户界面，至少会引起动画卡顿现象。

能高效绘制在iOS上的图片也有一个大小限制。所有显示在屏幕上的图片最终都会被转化为OpenGL纹理，同时OpenGL有一个最大的纹理尺寸（通常是`2048*2048`，或`4096*4096`，这个取决于设备型号）。如果你想在单个纹理中显示一个比这大的图，即便图片已经存在于内存中了，你仍然会遇到很大的性能问题，因为Core Animation强制用CPU处理图片而不是更快的GPU（见第12章『速度的曲调』，和第13章『高效绘图』，它更加详细地解释了软件绘制和硬件绘制）。

`CATiledLayer` 为载入大图造成的性能问题提供了一个解决方案：将大图分解成小片然后将他们单独按需载入。让我们用实验来证明一下。

### 小片裁剪

这个示例中，我们将会从一个`2048*2048`分辨率的雪人图片入手。为了能够从 `CATiledLayer` 中获益，我们需要把这个图片裁切成许多小一些的图片。你可以通过代码来完成这件事情，但是如果你在运行时读入整个图片并裁切，那 `CATiledLayer` 这些所有的性能优点就损失殆尽了。理想情况下来说，最好能够逐个步骤来实现。

清单6.11 演示了一个简单的Mac OS命令行程序，它用 `CATiledLayer` 将一个图片裁剪成小图并存储到不同的文件中。

#### 清单6.11 裁剪图片成小图的终端程序

```
#import

int main(int argc, const char * argv[])
{
 @autoreleasepool{
```

```

//handle incorrect arguments
if (argc < 2) {
 NSLog(@"TileCutter arguments: inputFile");
 return 0;
}

//input file
NSString *inputFile = [NSString stringWithCString:argv[1] e

//tile size
CGFloat tileSize = 256; //output path
NSString *outputPath = [inputFile stringByDeletingPathExte

//load image
NSImage *image = [[NSImage alloc] initWithContentsOfFile:in
NSSize size = [image size];
NSArray *representations = [image representations];
if ([representations count]){
 NSBitmapImageRep *representation = representations[0];
 size.width = [representation pixelsWide];
 size.height = [representation pixelsHigh];
}
NSRect rect = NSMakeRect(0.0, 0.0, size.width, size.height);
CGImageRef imageRef = [image CGImageForProposedRect:&rect c

//calculate rows and columns
NSInteger rows = ceil(size.height / tileSize);
NSInteger cols = ceil(size.width / tileSize);

//generate tiles
for (int y = 0; y < rows; ++y) {
 for (int x = 0; x < cols; ++x) {
 //extract tile image
 CGRect tileRect = CGRectMake(x*tileSize, y*tileSize, t:
 CGImageRef tileImage = CGImageCreateWithImageInRect(im

 //convert to jpeg data
 NSBitmapImageRep *imageRep = [[NSBitmapImageRep alloc]
 NSData *data = [imageRep representationUsingType: NSJPE
 CGImageRelease(tileImage);
}

```

```

 //save file
 NSString *path = [outputPath stringByAppendingFormat:@
 [data writeToFile:path atomically:NO];
 }
}
return 0;
}

```

这个程序将2048\*2048分辨率的雪人图案裁剪成了64个不同的256\*256的小图。

(256\*256是 `CATiledLayer` 的默认小图大小，默认大小可以通过 `tileSize` 属性更改)。程序接受一个图片路径作为命令行的第一个参数。我们可以在编译的 `scheme` 将路径参数硬编码然后就可以在 `Xcode` 中运行了，但是以后作用在另一个图片上就不方便了。所以，我们编译了这个程序并把它保存到敏感的地方，然后从终端调用，如下面所示：

```
> path/to/TileCutterApp path/to/Snowman.jpg
```

The app is very basic, but could easily be extended to support additional arguments such as tile size, or to export images in formats other than JPEG. The result of running it is a sequence of 64 new images, named as follows:

这个程序相当基础，但是能够轻易地扩展支持额外的参数比如小图大小，或者导出格式等等。运行结果是64个新图的序列，如下面命名：

```

Snowman_00_00.jpg
Snowman_00_01.jpg
Snowman_00_02.jpg
...
Snowman_07_07.jpg

```

既然我们有了裁切后的小图，我们就要让iOS程序用到他们。`CATiledLayer` 很好地和 `UIScrollView` 集成在一起。除了设置图层和滑动视图边界以适配整个图片大小，我们真正要做的就是实现 `-drawLayer:inContext:` 方法，当需要载入新的小图时，`CATiledLayer` 就会调用到这个方法。

清单6.12演示了代码。图6.12是代码运行结果。

### 清单6.12 一个简单的滚动 CATiledLayer 实现

```
#import "ViewController.h"
#import

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIScrollView *scrollView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //add the tiled layer
 CATiledLayer *tileLayer = [CATiledLayer layer];
 tileLayer.frame = CGRectMake(0, 0, 2048, 2048);
 tileLayer.delegate = self; [self.scrollView.layer addSublayer:titleLayer];

 //configure the scroll view
 self.scrollView.contentSize = tileLayer.frame.size;

 //draw layer
 [tileLayer setNeedsDisplay];
}

- (void)drawLayer:(CATiledLayer *)layer inContext:(CGContextRef)ctx
{
 //determine tile coordinate
 CGRect bounds = CGContextGetClipBoundingBox(ctx);
 NSInteger x = floor(bounds.origin.x / layer.tileSize.width);
 NSInteger y = floor(bounds.origin.y / layer.tileSize.height);

 //load tile image
 NSString *imageName = [NSString stringWithFormat:@"Snowman_%02d"];
 NSString *imagePath = [[NSBundle mainBundle] pathForResource:imageName
 ofType:@"png"];
 CGImageRef image = CGImageCreateWithPNGData([NSData dataWithContentsOfFile:imagePath]);
 CGContextDrawImage(ctx, bounds, image);
}
```

```

UIImage *tileImage = [UIImage imageNamed:@"imagePath"];

//draw tile
UIGraphicsPushContext(ctx);
[tileImage drawInRect:bounds];
UIGraphicsPopContext();
}

@end

```



图6.12 用 `UIScrollView` 滚动 `CATiledLayer`

当你滑动这个图片，你会发现当 `CATiledLayer` 载入小图的时候，他们会淡入到界面中。这是 `CATiledLayer` 的默认行为。（你可能已经在iOS 6之前的苹果地图程序中见过这个效果）你可以用 `fadeDuration` 属性改变淡入时长或直接禁用掉。`CATiledLayer`（不同于大部分的 `UIKit` 和 `Core Animation`方法）支持多线程绘制，`-drawLayer:inContext:` 方法可以在多个线程中同时地并发调用，所以请小心谨慎地确保你在这个方法中实现的绘制代码是线程安全的。

## Retina小图

你也许已经注意到了这些小图并不是以Retina的分辨率显示的。为了以屏幕的原生分辨率来渲染 `CATiledLayer`，我们需要设置图层的 `contentsScale` 来匹配 `UIScreen` 的 `scale` 属性：

```
tileLayer.contentsScale = [UIScreen mainScreen].scale;
```

有趣的是，`tileSize` 是以像素为单位，而不是点，所以增大了`contentsScale` 就自动有了默认的小图尺寸（现在它是`128*128`的点而不是`256*256`）。所以，我们不需要手工更新小图的尺寸或是在 Retina 分辨率下指定一个不同的小图。我们需要做的是适应小图渲染代码以对应安排`scale` 的变化，然而：

```
//determine tile coordinate
CGRect bounds = CGContextGetClipBoundingBox(ctx);
CGFloat scale = [UIScreen mainScreen].scale;
NSInteger x = floor(bounds.origin.x / layer.tileSize.width * scale);
NSInteger y = floor(bounds.origin.y / layer.tileSize.height * scale)
```

通过这个方法纠正`scale` 也意味着我们的雪人图将以一半的大小渲染在 Retina 设备上（总尺寸是`1024*1024`，而不是`2048*2048`）。这个通常都不会影响到用`CATiledLayer` 正常显示的图片类型（比如照片和地图，他们在设计上就是要支持放大缩小，能够在不同的缩放条件下显示），但是也需要在心里明白。

# CAEmitterLayer

在iOS 5中，苹果引入了一个新的 CALayer 子类叫  
做 CAEmitterLayer 。 CAEmitterLayer 是一个高性能的粒子引擎，被用来创建  
实时例子动画如：烟雾，火，雨等等这些效果。

CAEmitterLayer 看上去像是许多 CAEmitterCell 的容器，这  
些 CAEmitterCell 定义了一个例子效果。你将会为不同的例子效果定义一个或多  
个 CAEmitterCell 作为模版，同时 CAEmitterLayer 负责基于这些模版实例化  
一个粒子流。一个 CAEmitterCell 类似于一个 CALayer : 它有一  
个 contents 属性可以定义为一个 CGImage ，另外还有一些可设置属性控制着表  
现和行为。我们不会对这些属性逐一进行详细的描述，你们可以  
在 CAEmitterCell 类的头文件中找到。

我们来举个例子。我们将利用在一圆中发射不同速度和透明度的粒子创建一个火爆  
炸的效果。清单6.13包含了生成爆炸的代码。图6.13是运行结果

清单6.13 用 CAEmitterLayer 创建爆炸效果

```
#import "ViewController.h"
#import

@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //create particle emitter layer
 CAEmitterLayer *emitter = [CAEmitterLayer layer];
 emitter.frame = self.containerView.bounds;
```

```

[self.containerView.layer addSublayer:emitter];

//configure emitter
emitter.renderMode = kCAEmitterLayerAdditive;
emitter.emitterPosition = CGPointMake(emitter.frame.size.width

//create a particle template
CAEmitterCell *cell = [[CAEmitterCell alloc] init];
cell.contents = (__bridge id)[UIImage imageNamed:@"Spark.png"];
cell.birthRate = 150;
cell.lifetime = 5.0;
cell.color = [UIColor colorWithRed:1 green:0.5 blue:0.1 alpha:1];
cell.alphaSpeed = -0.4;
cell.velocity = 50;
cell.velocityRange = 50;
cell.emissionRange = M_PI * 2.0;

//add particle template to emitter
emitter.emitterCells = @[cell];
}

@end

```

图6.13 火焰爆炸效果

`CAEmitterCell` 的属性基本上可以分为三种：

- 这种粒子的某一属性的初始值。比如，`color` 属性指定了一个可以混合图片内容颜色的混合色。在示例中，我们将它设置为桔色。
- 例子某一属性的变化范围。比如 `emissionRange` 属性的值是 $2\pi$ ，这意味着例子可以从360度任意位置反射出来。如果指定一个小一些的值，就可以创造出一个圆锥形
- 指定值在时间线上的变化。比如，在示例中，我们将 `alphaSpeed` 设置为-0.4，就是说例子的透明度每过一秒就是减少0.4，这样就有发射出去之后逐渐小时的效果。

`CAEmitterLayer` 的属性它自己控制着整个例子系统的位置和形状。一些属性比如 `birthRate`，`lifetime` 和 `celocity`，这些属性在 `CAEmitterCell` 中也有。这些属性会以相乘的方式作用在一起，这样你就可以用一个值来加速或者扩大整个例子系统。其他值得提到的属性有以下这些：

- `preservesDepth` , 是否将3D例子系统平面化到一个图层（默认值）或者可以在3D空间中混合其他的图层
- `renderMode` , 控制着在视觉上粒子图片是如何混合的。你可能已经注意到了示例中我们把它设置为 `kCAEmitterLayerAdditive` , 它实现了这样一个效果：合并例子重叠部分的亮度使得看上去更亮。如果我们把它设置为默认的 `kCAEmitterLayerUnordered` , 效果就没那么好看了（见图6.14）.

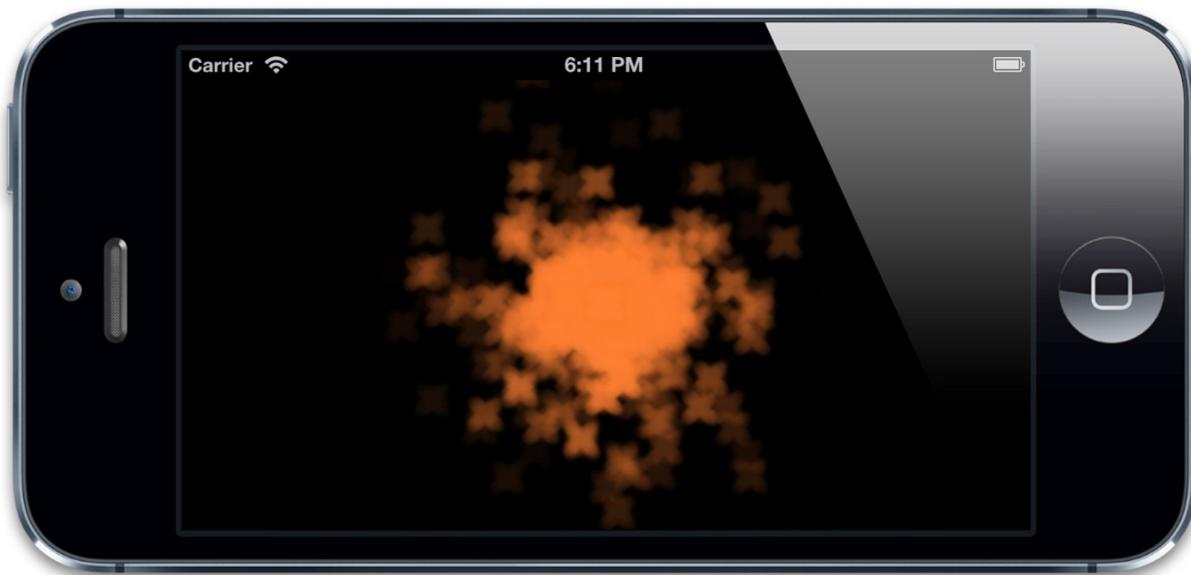


图6.14 禁止混色之后的火焰粒子

# CAEAGLLayer

当iOS要处理高性能图形绘制，必要时就是OpenGL。应该说它应该是最后的杀手锏，至少对于非游戏的应用来说是的。因为相比Core Animation和UIKit框架，它不可思议地复杂。

OpenGL提供了Core Animation的基础，它是底层的C接口，直接和iPhone，iPad的硬件通信，极少地抽象出来的方法。OpenGL没有对象或是图层的继承概念。它只是简单地处理三角形。OpenGL中所有东西都是3D空间中有颜色和纹理的三角形。用起来非常复杂和强大，但是用OpenGL绘制iOS用户界面就需要很多很多的工作了。

为了能够以高性能使用Core Animation，你需要判断你需要绘制哪种内容（矢量图形，例子，文本，等等），然后选择合适的图层去呈现这些内容，Core Animation中只有一些类型的内容是被高度优化的；所以如果你想绘制的东西并不能找到标准的图层类，想要得到高性能就比较费事情了。

因为OpenGL根本不会对你的内容进行假设，它能够绘制得相当快。利用OpenGL，你可以绘制任何你知道必要的集合信息和形状逻辑的内容。所以很多游戏都喜欢用OpenGL（这些情况下，Core Animation的限制就明显了：它优化过的内容类型并不一定能满足需求），但是这样依赖，方便的高度抽象接口就没了。

在iOS 5中，苹果引入了一个新的框架叫做GLKit，它去掉了一些设置OpenGL的复杂性，提供了一个叫做 GLKView 的 UIView 的子类，帮你处理大部分的设置和绘制工作。前提是各种各样的OpenGL绘图缓冲的底层可配置项仍然需要你用 CAEAGLLayer 完成，它是 CALayer 的一个子类，用来显示任意的OpenGL图形。

大部分情况下你都不需要手动设置 CAEAGLLayer （假设用GLKView），过去的日子就不要再提了。特别的，我们将设置一个OpenGL ES 2.0的上下文，它是现代的iOS设备的标准做法。

尽管不需要GLKit也可以做到这一切，但是GLKit囊括了很多额外的工作，比如设置顶点和片段着色器，这些都以类C语言叫做GLSL自包含在程序中，同时在运行时载入到图形硬件中。编写GLSL代码和设置 EAGLLayer 没有什么关系，所以我们将用 GLKBaseEffect 类将着色逻辑抽象出来。其他的事情，我们还是会有以往的方式。

在开始之前，你需要将GLKit和OpenGL ES框架加入到你的项目中，然后就可以实现清单6.14中的代码，里面是设置一个 GAEAGLLayer 的最少工作，它使用了 OpenGL ES 2.0 的绘图上下文，并渲染了一个有色三角（见图6.15）。

#### 清单6.14 用 CAEAGLLayer 绘制一个三角形

```
#import "ViewController.h"
#import
#import

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *glView;
@property (nonatomic, strong) EAGLContext *glContext;
@property (nonatomic, strong) CAEAGLLayer *glLayer;
@property (nonatomic, assign) GLuint framebuffer;
@property (nonatomic, assign) GLuint colorRenderbuffer;
@property (nonatomic, assign) GLint framebufferWidth;
@property (nonatomic, assign) GLint framebufferHeight;
@property (nonatomic, strong) GLKBaseEffect *effect;

@end

@implementation ViewController

- (void)setUpBuffers
{
 //set up frame buffer
 glGenFramebuffers(1, &_framebuffer);
 glBindFramebuffer(GL_FRAMEBUFFER, _framebuffer);

 //set up color render buffer
 glGenRenderbuffers(1, &_colorRenderbuffer);
 glBindRenderbuffer(GL_RENDERBUFFER, _colorRenderbuffer);
 glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
 [self.glContext renderbufferStorage:GL_RENDERBUFFER fromDrawable:_framebuffer];
 glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_WIDTH, &_framebufferWidth);
 glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_HEIGHT, &_framebufferHeight);

 //check success
}
```

```
if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
 NSLog(@"Failed to make complete framebuffer object: %i", glError);
}

- (void)tearDownBuffers
{
 if (_framebuffer) {
 //delete framebuffer
 glDeleteFramebuffers(1, &_framebuffer);
 _framebuffer = 0;
 }

 if (_colorRenderbuffer) {
 //delete color render buffer
 glDeleteRenderbuffers(1, &_colorRenderbuffer);
 _colorRenderbuffer = 0;
 }
}

- (void)drawFrame {
 //bind framebuffer & set viewport
 glBindFramebuffer(GL_FRAMEBUFFER, _framebuffer);
 glViewport(0, 0, _framebufferWidth, _framebufferHeight);

 //bind shader program
 [self.effect prepareToDraw];

 //clear the screen
 glClear(GL_COLOR_BUFFER_BIT); glClearColor(0.0, 0.0, 0.0, 1.0);

 //set up vertices
 GLfloat vertices[] = {
 -0.5f, -0.5f, -1.0f, 0.0f, 0.5f, -1.0f, 0.5f, -0.5f, -1.0f,
 };

 //set up colors
 GLfloat colors[] = {
 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, 0.0f,
 };
}
```

```
//draw triangle
glEnableVertexAttribArray(GLKVertexAttribPosition);
glEnableVertexAttribArray(GLKVertexAttribColor);
glVertexAttribPointer(GLKVertexAttribPosition, 3, GL_FLOAT, GL_FALSE,
glVertexAttribPointer(GLKVertexAttribColor, 4, GL_FLOAT, GL_FALSE,
glDrawArrays(GL_TRIANGLES, 0, 3);

//present render buffer
glBindRenderbuffer(GL_RENDERBUFFER, _colorRenderbuffer);
[self.glContext presentRenderbuffer:GL_RENDERBUFFER];
}

- (void)viewDidLoad
{
 [super viewDidLoad];
 //set up context
 self.glContext = [[EAGLContext alloc] initWithAPI: kEAGLRenderingAPIOpenGL2];
 [EAGLContext setCurrentContext:self.glContext];

 //set up layer
 self.glLayer = [CAEAGLLayer layer];
 self.glLayer.frame = self.glView.bounds;
 [self.glView.layer addSublayer:self.glLayer];
 self.glLayer.drawableProperties = @{{kEAGLDrawablePropertyRetainCount:@(1),
 kEAGLDrawablePropertyColorFormat:@(kEAGLColorFormatBGRA),
 kEAGLDrawablePropertyPixelFormat:@(kEAGLPixelFormatBGRA8888)}};

 //set up base effect
 self.effect = [[GLKBaseEffect alloc] init];

 //set up buffers
 [self setUpBuffers];

 //draw frame
 [self drawFrame];
}

- (void)viewDidUnload
{
 [self tearDownBuffers];
 [super viewDidUnload];
```

```
}

- (void)dealloc
{
 [self tearDownBuffers];
 [EAGLContext setCurrentContext:nil];
}
@end
```



图6.15 用OpenGL渲染的 CAEAGLLayer 图层

在一个真正的OpenGL应用中，我们可能会用 `NSTimer` 或 `CADisplayLink` 周期性地每秒钟调用 `-drawRrame` 方法60次，同时会将几何图形生成和绘制分开以便不会每次都重新生成三角形的顶点（这样也可以让我们绘制其他的一些东西而不是一个三角形而已），不过上面这个例子已经足够演示了绘图原则了。

# AVPlayerLayer

最后一个图层类型是 `AVPlayerLayer`。尽管它不是Core Animation框架的一部分（AV前缀看上去像），`AVPlayerLayer` 是有别的框架（AVFoundation）提供的，它和Core Animation紧密地结合在一起，提供了一个 `CALayer` 子类来显示自定义的内容类型。

`AVPlayerLayer` 是用来在iOS上播放视频的。他是高级接口例如 `MPMoviePlayer` 的底层实现，提供了显示视频的底层控制。`AVPlayerLayer` 的使用相当简单：你可以用 `+playerLayerWithPlayer:` 方法创建一个已经绑定了视频播放器的图层，或者你可以先创建一个图层，然后用 `player` 属性绑定一个 `AVPlayer` 实例。

在我们开始之前，我们需要添加AVFoundation到我们的项目中。然后，清单6.15创建了一个简单的电影播放器，图6.16是代码运行结果。

清单6.15 用 `AVPlayerLayer` 播放视频

```
#import "ViewController.h"
#import
#import

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView; @end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //get video URL
 NSURL *URL = [[NSBundle mainBundle] URLForResource:@"Ship" withExtension:@"mp4"];

 //create player and player layer
 AVPlayer *player = [AVPlayer playerWithURL:URL];
 AVPlayerLayer *playerLayer = [AVPlayerLayer playerLayerWithPlayer:player];

 //set player layer frame and attach it to our view
 playerLayer.frame = self.containerView.bounds;
 [self.containerView.layer addSublayer:playerLayer];

 //play the video
 [player play];
}
@end
```

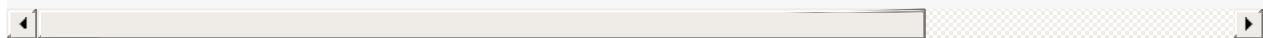




图6.16 用 `AVPlayerLayer` 图层播放视频的截图

我们用代码创建了一个 `AVPlayerLayer`，但是我们仍然把它添加到了一个容器视图中，而不是直接在 `controller` 中的主视图上添加。这样其实是为了可以使用自动布局限制使得图层在最中间；否则，一旦设备被旋转了我们就要手动重新放置位置，因为 `Core Animation` 并不支持自动大小和自动布局（见第三章『图层几何学』）。

当然，因为 `AVPlayerLayer` 是 `CALayer` 的子类，它继承了父类的所有特性。我们并不会受限于要在一个矩形中播放视频；清单6.16演示了在3D，圆角，有色边框，蒙板，阴影等效果（见图6.17）。

清单6.16 给视频增加变换，边框和圆角

```
- (void)viewDidLoad
{
 ...
 //set player layer frame and attach it to our view
 playerLayer.frame = self.containerView.bounds;
 [self.containerView.layer addSublayer:playerLayer];

 //transform layer
 CATransform3D transform = CATransform3DIdentity;
 transform.m34 = -1.0 / 500.0;
 transform = CATransform3DRotate(transform, M_PI_4, 1, 1, 0);
 playerLayer.transform = transform;

 //add rounded corners and border
 playerLayer.masksToBounds = YES;
 playerLayer.cornerRadius = 20.0;
 playerLayer.borderColor = [UIColor redColor].CGColor;
 playerLayer.borderWidth = 5.0;

 //play the video
 [player play];
}
```

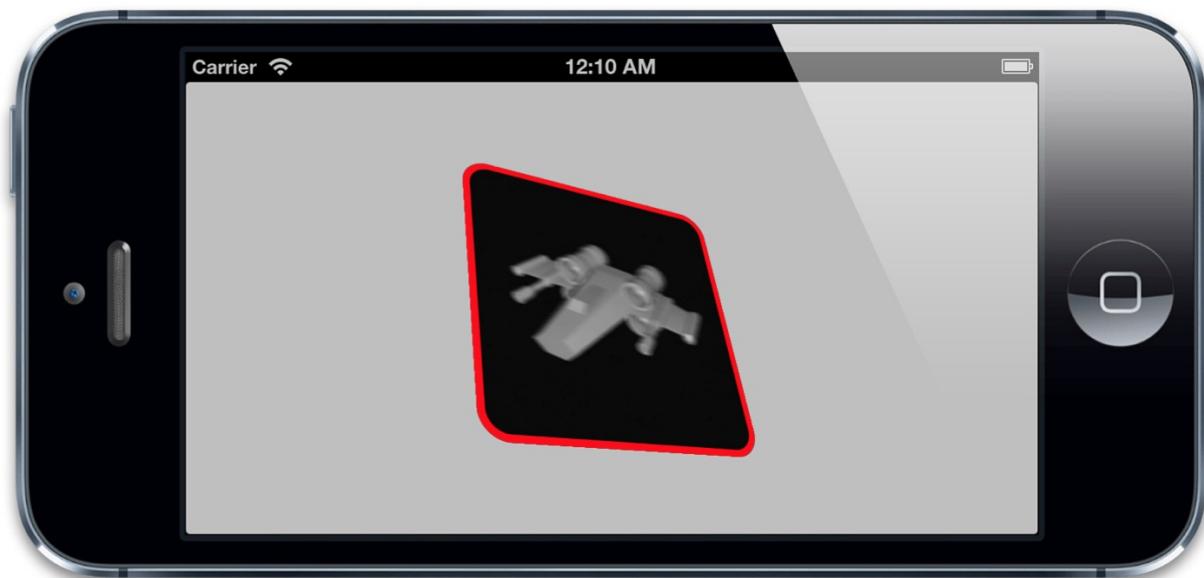


图6.17 3D视角下的边框和圆角 AVPlayerLayer

## 总结

这一章我们简要概述了一些专用图层以及用他们实现的一些效果，我们只是了解到这些图层的皮毛，像 `CATiledLayer` 和 `CAEmitterLayer` 这些类可以单独写一章的。但是，重点是记住 `CALayer` 是用途很大的，而且它并没有为所有可能的场景进行优化。为了获得Core Animation最好的性能，你需要为你的工作选对正确的工具，希望你能够挖掘这些不同的 `CALayer` 子类的功能。这一章我们通过 `CAEmitterLayer` 和 `AVPlayerLayer` 类简单地接触到了一些动画，在第二章，我们将继续深入研究动画，就从隐式动画开始。

## 隐式动画

按照我的意思去做，而不是我说的。 -- 埃德娜，辛普森

我们在第一部分讨论了Core Animation除了动画之外可以做到的任何事情。但是动画是Core Animation库一个非常显著的特性。这一章我们来看看它是怎么做到的。具体来说，我们先来讨论框架自动完成的隐式动画（除非你明确禁用了这个功能）。

# 事务

Core Animation 基于一个假设，说屏幕上的任何东西都可以（或者可能）做动画。动画并不需要你在 Core Animation 中手动打开，相反需要明确地关闭，否则他会一直存在。

当你改变 `CALayer` 的一个可做动画的属性，它并不能立刻在屏幕上体现出来。相反，它是从先前的值平滑过渡到新的值。这一切都是默认的行为，你不需要做额外的操作。

这看起来这太棒了，似乎不太真实，我们来用一个 demo 解释一下：首先和第一章“图层树”一样创建一个蓝色的方块，然后添加一个按钮，随机改变它的颜色。代码见清单 7.1。点击按钮，你会发现图层的颜色平滑过渡到一个新值，而不是跳变（图 7.1）。

## 清单 7.1 随机改变图层颜色

```
@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *layerView;
@property (nonatomic, weak) IBOutlet CALayer *colorLayer; /*热心人发现
*/
@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create sublayer
 self.colorLayer = [CALayer layer];
 self.colorLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
 self.colorLayer.backgroundColor = [UIColor blueColor].CGColor;
 //add it to our view
 [self.layerView.layer addSublayer:self.colorLayer];
}

- (IBAction)changeColor
{
 //randomize the layer background color
 CGFloat red = arc4random() / (CGFloat)INT_MAX;
 CGFloat green = arc4random() / (CGFloat)INT_MAX;
 CGFloat blue = arc4random() / (CGFloat)INT_MAX;
 self.colorLayer.backgroundColor = [UIColor colorWithRed:red green:green blue:blue];
}

@end
```

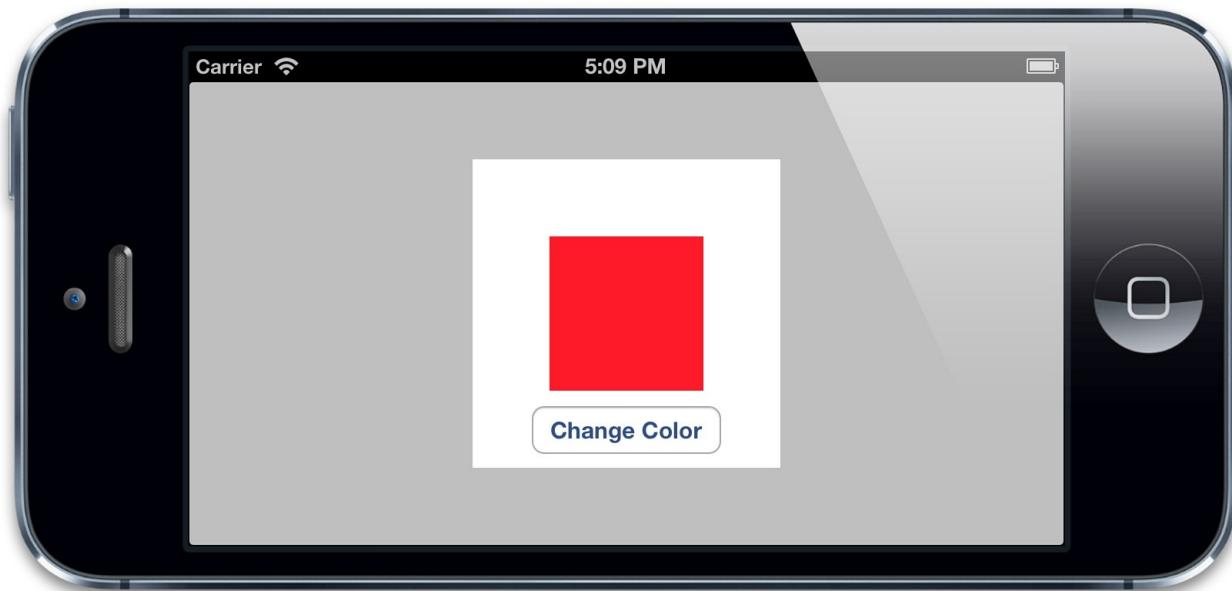


图7.1 添加一个按钮来控制图层颜色

这其实就是所谓的隐式动画。之所以叫隐式是因为我们并没有指定任何动画的类型。我们仅仅改变了一个属性，然后Core Animation来决定如何并且何时去做动画。Core Animaiton同样支持显式动画，下章详细说明。

但当你改变一个属性，Core Animation是如何判断动画类型和持续时间的呢？实际上动画执行的时间取决于当前事务的设置，动画类型取决于图层行为。

事务实际上是Core Animation用来包含一系列属性动画集合的机制，任何用指定事务去改变可以做动画的图层属性都不会立刻发生变化，而是当事务一旦提交的时候开始用一个动画过渡到新值。

事务是通过 `CATransaction` 类来做管理，这个类的设计有些奇怪，不像你从它的命名预期的那样去管理一个简单的事务，而是管理了一叠你不能访问的事务。`CATransaction` 没有属性或者实例方法，并且也不能用 `+alloc` 和 `-init` 方法创建它。但是可以用 `+begin` 和 `+commit` 分别来入栈或者出栈。

任何可以做动画的图层属性都会被添加到栈顶的事务，你可以通过 `+setAnimationDuration:` 方法设置当前事务的动画时间，或者通过 `+animationDuration` 方法来获取值（默认0.25秒）。

Core Animation在每个*run loop*周期中自动开始一次新的事务（*run loop*是iOS负责收集用户输入，处理定时器或者网络事件并且重新绘制屏幕的东西），即使你不显式的用 `[CATransaction begin]` 开始一次事务，任何在一次*run loop*循环中属性的改变都会被集中起来，然后做一次0.25秒的动画。

明白这些之后，我们就可以轻松修改动画的时间了。我们当然可以用当前事务的 `+setAnimationDuration:` 方法来修改动画时间，但在这里我们首先起一个新的事务，于是修改时间就不会有别的副作用。因为修改当前事务的时间可能会导致同一时刻别的动画（如屏幕旋转），所以最好还是在调整动画之前压入一个新的事务。

修改后的代码见清单7.2。运行程序，你会发现色块颜色比之前变得更慢了。

### 清单7.2 使用 `CATransaction` 控制动画时间

```
- (IBAction)changeColor
{
 //begin a new transaction
 [CATransaction begin];
 //set the animation duration to 1 second
 [CATransaction setAnimationDuration:1.0];
 //randomize the layer background color
 CGFloat red = arc4random() / (CGFloat)INT_MAX;
 CGFloat green = arc4random() / (CGFloat)INT_MAX;
 CGFloat blue = arc4random() / (CGFloat)INT_MAX;
 self.colorLayer.backgroundColor = [UIColor colorWithRed:red green:green blue:blue alpha:1];
 //commit the transaction
 [CATransaction commit];
}
```



如果你用过 `UIView` 的动画方法做过一些动画效果，那么应该对这个模式不陌生。`UIView` 有两个方法，`+beginAnimations:context:` 和 `+commitAnimations`，和 `CATransaction` 的 `+begin` 和 `+commit` 方法类似。实际上在 `+beginAnimations:context:` 和 `+commitAnimations` 之间所有视图或者图层属性的改变而做的动画都是由于设置了 `CATransaction` 的原因。

在iOS4中，苹果对`UIView`添加了一种基于block的动画方法：`+animateWithDuration:animations:`。这样写对做一堆的属性动画在语法上会更加简单，但实质上它们都是在做同样的事情。

CATransaction 的 +begin 和 +commit 方法

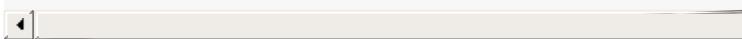
在 +animateWithDuration:animations: 内部自动调用，这样block中所有属性的改变都会被事务所包含。这样也可以避免开发者由于对 +begin 和 +commit 匹配的失误造成的风险。

# 完成块

基于 `UIView` 的 `block` 的动画允许你在动画结束的时候提供一个完成的动作。`CATransaction` 接口提供的 `+setCompletionBlock:` 方法也有同样的功能。我们来调整上个例子，在颜色变化结束之后执行一些操作。我们来添加一个完成之后的 `block`，用来在每次颜色变化结束之后切换到另一个旋转90度的动画。代码见清单7.3，运行结果见图7.2。

清单7.3 在颜色动画完成之后添加一个回调

```
- (IBAction)changeColor
{
 //begin a new transaction
 [CATransaction begin];
 //set the animation duration to 1 second
 [CATransaction setAnimationDuration:1.0];
 //add the spin animation on completion
 [CATransaction setCompletionBlock:^{
 //rotate the layer 90 degrees
 CGAffineTransform transform = self.colorLayer.affineTransform;
 transform = CGAffineTransformRotate(transform, M_PI_2);
 self.colorLayer.affineTransform = transform;
 }];
 //randomize the layer background color
 CGFloat red = arc4random() / (CGFloat)INT_MAX;
 CGFloat green = arc4random() / (CGFloat)INT_MAX;
 CGFloat blue = arc4random() / (CGFloat)INT_MAX;
 self.colorLayer.backgroundColor = [UIColor colorWithRed:red green:green blue:blue alpha:1];
 //commit the transaction
 [CATransaction commit];
}
```



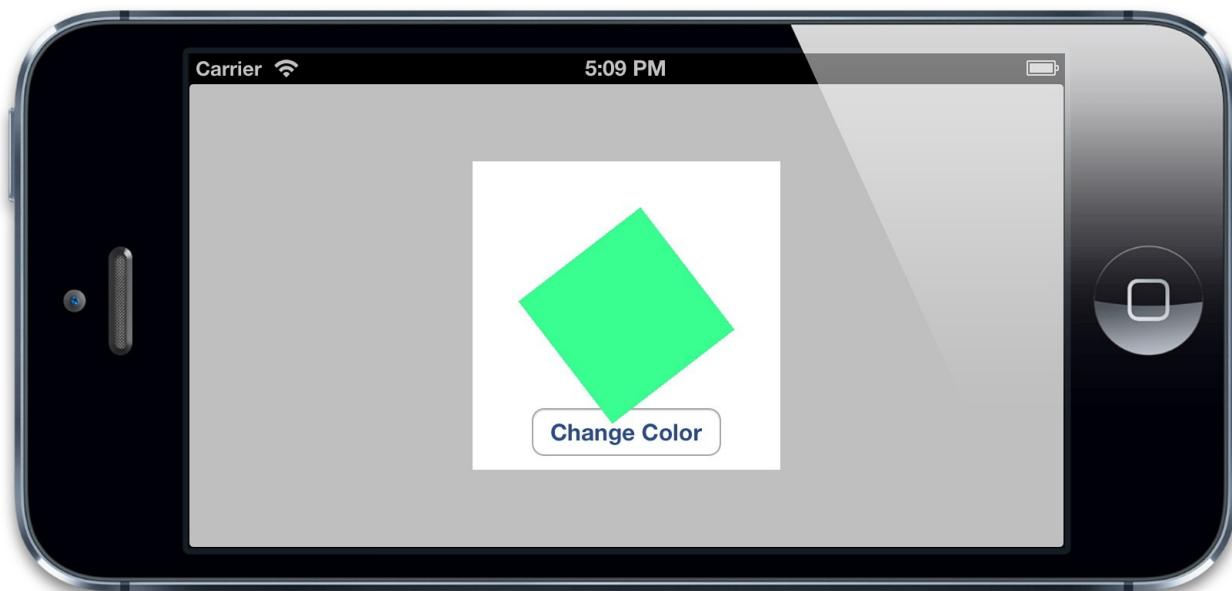


图7.2 颜色渐变之完成之后再做一次旋转

注意旋转变动画要比颜色渐变快得多，这是因为完成块是在颜色渐变的事务提交并出栈之后才被执行，于是，用默认的事务做变换，默认的时间也就变成了0.25秒。

# 图层行为

现在来做个实验，试着直接对 UIView 关联的图层做动画而不是一个单独的图层。清单 7.4 是对清单 7.2 代码的一点修改，移除了 colorLayer，并且直接设置 layerView 关联图层的背景色。

清单 7.4 直接设置图层的属性

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //set the color of our layerView backing layer directly
 self.layerView.layer.backgroundColor = [UIColor blueColor].CGColor
}

- (IBAction)changeColor
{
 //begin a new transaction
 [CATransaction begin];
 //set the animation duration to 1 second
 [CATransaction setAnimationDuration:1.0];
 //randomize the layer background color
 CGFloat red = arc4random() / (CGFloat)INT_MAX;
 CGFloat green = arc4random() / (CGFloat)INT_MAX;
 CGFloat blue = arc4random() / (CGFloat)INT_MAX;
 self.layerView.layer.backgroundColor = [UIColor colorWithRed:red
 //commit the transaction
 [CATransaction commit];
}
```

运行程序，你会发现当按下按钮，图层颜色瞬间切换到新的值，而不是之前平滑过渡的动画。发生了什么呢？`UIView` 关联图层给禁用了。

试想一下，如果 `UIView` 的属性都有动画特性的话，那么无论在什么时候修改它，我们都应该能注意到的。所以，如果说UIKit建立在Core Animation（默认对所有东西都做动画）之上，那么隐式动画是如何被UIKit禁用掉呢？

我们知道Core Animation通常对 `CALayer` 的所有属性（可动画的属性）做动画，但是 `UIView` 把它关联的图层的这个特性关闭了。为了更好说明这一点，我们需要知道隐式动画是如何实现的。

我们把改变属性时 `CALayer` 自动应用的动画称作行为，当 `CALayer` 的属性被修改时候，它会调用 `-actionForKey:` 方法，传递属性的名称。剩下的操作都在 `CALayer` 的头文件中有详细的说明，实质上是如下几步：

- 图层首先检测它是否有委托，并且是否实现 `CALayerDelegate` 协议指定的 `-actionForLayer:forKey:` 方法。如果有，直接调用并返回结果。
- 如果没有委托，或者委托没有实现 `-actionForLayer:forKey:` 方法，图层接着检查包含属性名称对应行为映射的 `actions` 字典。
- 如果 `actions`字典 没有包含对应的属性，那么图层接着在它的 `style` 字典接着搜索属性名。
- 最后，如果在 `style` 里面也找不到对应的行为，那么图层将会直接调用定义了每个属性的标准行为的 `-defaultActionForKey:` 方法。

所以一轮完整的搜索结束之后，`-actionForKey:` 要么返回空（这种情况下将不会有动画发生），要么是 `CAACTION` 协议对应的对象，最后 `CALayer` 拿这个结果去对先前和当前的值做动画。

于是这就解释了UIKit是如何禁用隐式动画的：每个 `UIView` 对它关联的图层都扮演了一个委托，并且提供了 `-actionForLayer:forKey:` 的实现方法。当不在一个动画块的实现中，`UIView` 对所有图层行为返回 `nil`，但是在动画block范围之内，它就返回了一个非空值。我们可以用一个demo做个简单的实验（清单7.5）

#### 清单7.5 测试`UIView`的 `actionForLayer:forKey:` 实现

```

@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *layerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //test layer action when outside of animation block
 NSLog(@"%@", [self.layerView actionForLayer:self.layer]);
 //begin animation block
 [UIView beginAnimations:nil context:nil];
 //test layer action when inside of animation block
 NSLog(@"%@", [self.layerView actionForLayer:self.layer]);
 //end animation block
 [UIView commitAnimations];
}

@end

```

运行程序，控制台显示结果如下：

```

$ LayerTest[21215:c07] Outside: <null>
$ LayerTest[21215:c07] Inside: <CABasicAnimation: 0x757f090>

```

于是我们可以预言，当属性在动画块之外发生改变，`UIView` 直接通过返回 `nil` 来禁用隐式动画。但如果在动画块范围之内，根据动画具体类型返回相应的属性，在这个例子就是 `CABasicAnimation`（第八章“显式动画”将会提到）。

当然返回 `nil` 并不是禁用隐式动画唯一的办法，`CATransaction` 有个方法叫做 `+setDisableActions:`，可以用来对所有属性打开或者关闭隐式动画。如果在清单 7.2 的 `[CATransaction begin]` 之后添加下面的代码，同样也会阻止动画的发生：

```
[CATransaction setDisableActions:YES];
```

总结一下，我们知道了如下几点

- `UIView` 关联的图层禁用了隐式动画，对这种图层做动画的唯一办法就是使用 `UIView` 的动画函数（而不是依赖 `CATransaction`），或者继承 `UIView`，并覆盖 `-actionForLayer:forKey:` 方法，或者直接创建一个显式动画（具体细节见第八章）。
- 对于单独存在的图层，我们可以通过实现图层的 `-actionForLayer:forKey:` 委托方法，或者提供一个 `actions` 字典来控制隐式动画。

我们来对颜色渐变的例子使用一个不同的行为，通过给 `colorLayer` 设置一个自定义的 `actions` 字典。我们也可以使用委托来实现，但是 `actions` 字典可以写更少的代码。那么到底该如何创建一个合适的行为对象呢？

行为通常是一个被Core Animation隐式调用的显式动画对象。这里我们使用的是一個实现了 `CATransition` 的实例，叫做推进过渡。

第八章中将会详细解释过渡，不过对于现在，知道 `CATransition` 响应 `CAACTION` 协议，并且可以当做一个图层行为就足够了。结果很赞，不论在什么时候改变背景颜色，新的色块都是从左侧滑入，而不是默认的渐变效果。

#### 清单7.6 实现自定义行为

```
@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *layerView;
@property (nonatomic, weak) IBOutlet CALayer *colorLayer; /*热心人发现
*/

@end

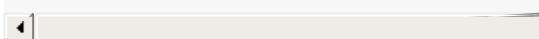
@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //create sublayer
 self.colorLayer = [CALayer layer];
 self.colorLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
 self.colorLayer.backgroundColor = [UIColor blueColor].CGColor;
 //add a custom action
 CATransition *transition = [CATransition animation];
 transition.type = kCATransitionPush;
 transition.subtype = kCATransitionFromLeft;
 self.colorLayer.actions = @{@"backgroundColor": transition};
 //add it to our view
 [self.layerView.layer addSublayer:self.colorLayer];
}

- (IBAction)changeColor
{
 //randomize the layer background color
 CGFloat red = arc4random() / (CGFloat)INT_MAX;
 CGFloat green = arc4random() / (CGFloat)INT_MAX;
 CGFloat blue = arc4random() / (CGFloat)INT_MAX;
 self.colorLayer.backgroundColor = [UIColor colorWithRed:red green:green blue:blue alpha:1.0];
}

@end
```



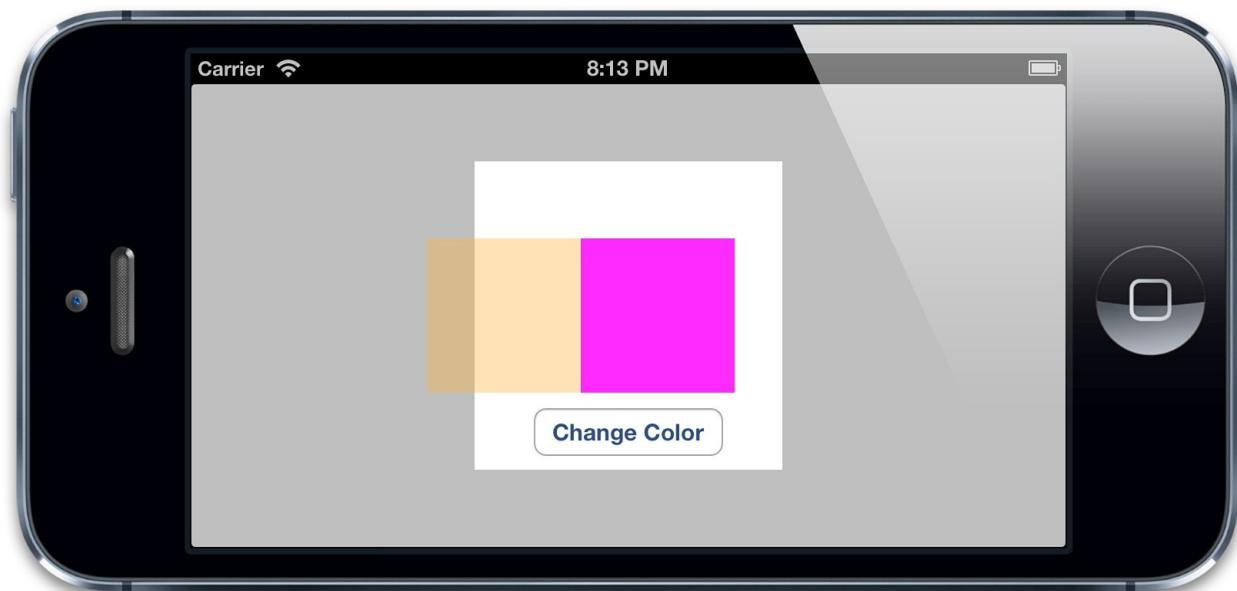


图7.3 使用推进过渡的色值动画

# 呈现与模型

`CALayer` 的属性行为其实很不正常，因为改变一个图层的属性并没有立刻生效，而是通过一段时间渐变更新。这是怎么做到的呢？

当你改变一个图层的属性，属性值的确是立刻更新的（如果你读取它的数据，你会发现它的值在你设置它的那一刻就已经生效了），但是屏幕上并没有马上发生改变。这是因为你设置的属性并没有直接调整图层的外观，相反，他只是定义了图层动画结束之后将要变化的外观。

当设置 `CALayer` 的属性，实际上是在定义当前事务结束之后图层如何显示的模型。`Core Animation`扮演了一个控制器的角色，并且负责根据图层行为和事务设置去不断更新视图的这些属性在屏幕上的状态。

我们讨论的就是一个典型的微型MVC模式。`CALayer` 是一个连接用户界面（就是MVC中的`view`）虚构的类，但是在界面本身这个场景下，`CALayer` 的行为更像是存储了视图如何显示和动画的数据模型。实际上，在苹果自己的文档中，图层树通常都是值的图层树模型。

在iOS中，屏幕每秒钟重绘60次。如果动画时长比60分之一秒要长，`Core Animation`就需要在设置一次新值和新值生效之间，对屏幕上的图层进行重新组织。这意味着 `CALayer` 除了“真实”值（就是你设置的值）之外，必须要知道当前显示在屏幕上的属性值的记录。

每个图层属性的显示值都被存储在一个叫做呈现图层的独立图层当中，他可以通过`-presentationLayer` 方法来访问。这个呈现图层实际上是模型图层的复制，但是它的属性值代表了在任何指定时刻当前外观效果。换句话说，你可以通过呈现图层的值来获取当前屏幕上真正显示出来的值（图7.4）。

我们在第一章中提到除了图层树，另外还有呈现树。呈现树通过图层树中所有图层的呈现图层所形成。注意呈现图层仅仅当图层首次被提交（就是首次第一次在屏幕上显示）的时候创建，所以在那之前调用`-presentationLayer` 将会返回`nil`。

你可能注意到有一个叫做`-modelLayer` 的方法。在呈现图层上调用`-modelLayer` 将会返回它正在呈现所依赖的`CALayer`。通常在一个图层上调用`-modelLayer` 会返回`-self`（实际上我们已经创建的原始图层就是一种数据模型）。

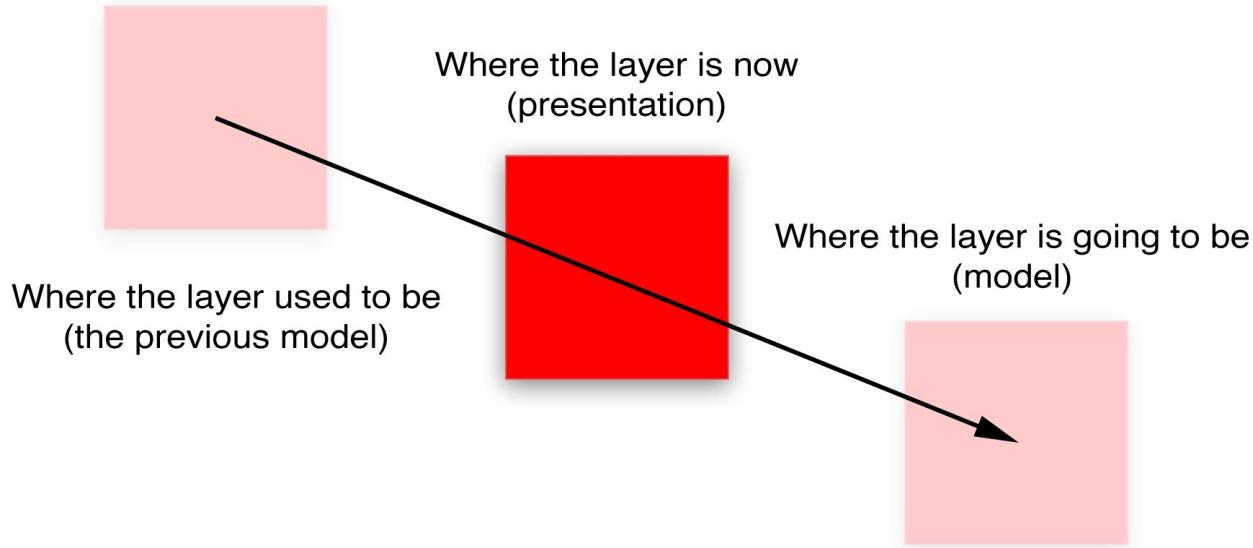


图7.4 一个移动的图层是如何通过数据模型呈现的

大多数情况下，你不需要直接访问呈现图层，你可以通过和模型图层的交互，来让Core Animation更新显示。两种情况下呈现图层会变得很有用，一个是同步动画，一个是处理用户交互。

- 如果你在实现一个基于定时器的动画（见第11章“基于定时器的动画”），而不仅仅是基于事务的动画，这个时候准确地知道在某一时刻图层显示在什么位置就会对正确摆放图层很有用了。
- 如果你想让你做动画的图层响应用户输入，你可以使用`-hitTest:`方法（见第三章“图层几何学”）来判断指定图层是否被触摸，这时候对呈现图层而不是模型图层调用`-hitTest:`会显得更有意义，因为呈现图层代表了用户当前看到的图层位置，而不是当前动画结束之后的位置。

我们可以用一个简单的案例来证明后者（见清单7.7）。在这个例子中，点击屏幕上的任意位置将会让图层平移到那里。点击图层本身可以随机改变它的颜色。我们通过对呈现图层调用`-hitTest:`来判断是否被点击。

如果修改代码让`-hitTest:`直接作用于`colorLayer`而不是呈现图层，你会发现当图层移动的时候它并不能正确显示。这时候你就需要点击图层将要移动到的位置而不是图层本身来响应点击（这就是为什么用呈现图层来响应交互的原因）。

#### 清单7.7 使用`presentationLayer`图层来判断当前图层位置

```
@interface ViewController ()

@property (nonatomic, strong) CALayer *colorLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create a red layer
 self.colorLayer = [CALayer layer];
 self.colorLayer.frame = CGRectMake(0, 0, 100, 100);
 self.colorLayer.position = CGPointMake(self.view.bounds.size.w:
 self.colorLayer.backgroundColor = [UIColor redColor].CGColor;
 [self.view.layer addSublayer:self.colorLayer];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
 //get the touch point
 CGPoint point = [[touches anyObject] locationInView:self.view];
 //check if we've tapped the moving layer
 if ([self.colorLayer.presentationLayer hitTest:point]) {
 //randomize the layer background color
 CGFloat red = arc4random() / (CGFloat)INT_MAX;
 CGFloat green = arc4random() / (CGFloat)INT_MAX;
 CGFloat blue = arc4random() / (CGFloat)INT_MAX;
 self.colorLayer.backgroundColor = [UIColor colorWithRed:red:
 } else {
 //otherwise (slowly) move the layer to new position
 [CATransaction begin];
 [CATransaction setAnimationDuration:4.0];
 self.colorLayer.position = point;
 [CATransaction commit];
 }
}
```

@end

## 总结

这一章讨论了隐式动画，还有Core Animation对指定属性选择合适的动画行为的机制。同时你知道了UIKit是如何充分利用Core Animation的隐式动画机制来强化它的显式系统，以及动画是如何被默认禁用并且当需要的时候启用的。最后，你了解了呈现和模型图层，以及Core Animation是如何通过它们来判断出图层当前位置以及将要到达的位置。

在下一章中，我们将研究Core Animation提供的显式动画类型，既可以对图层属性做动画，也可以覆盖默认的图层行为。

## 显式动画

如果想让事情变得顺利，只有靠自己 -- 夏尔·纪尧姆

上一章介绍了隐式动画的概念。隐式动画是在iOS平台创建动态用户界面的一种直接方式，也是UIKit动画机制的基础，不过它并不能涵盖所有的动画类型。在这一章中，我们将要研究一下显式动画，它能够对一些属性做指定的自定义动画，或者创建非线性动画，比如沿着任意一条曲线移动。

## 属性动画

`CAAnimationDelegate` 在任何头文件中都找不到，但是可以在 `CAAnimation` 头文件或者苹果开发者文档中找到相关函数。在这个例子中，我们用 `- animationDidStop:finished:` 方法在动画结束之后来更新图层的 `backgroundColor`。

当更新属性的时候，我们需要设置一个新的事务，并且禁用图层行为。否则动画会发生两次，一个是因为显式的 `CABasicAnimation`，另一次是因为隐式动画，具体实现见订单8.3。

清单8.3 动画完成之后修改图层的背景色

```
@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create sublayer
 self.colorLayer = [CALayer layer];
 self.colorLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
 self.colorLayer.backgroundColor = [UIColor blueColor].CGColor;
 //add it to our view
 [self.layerView.layer addSublayer:self.colorLayer];
}

- (IBAction)changeColor
{
 //create a new random color
 CGFloat red = arc4random() / (CGFloat)INT_MAX;
 CGFloat green = arc4random() / (CGFloat)INT_MAX;
 CGFloat blue = arc4random() / (CGFloat)INT_MAX;
 UIColor *color = [UIColor colorWithRed:red green:green blue:blue];
 //create a basic animation
 CABasicAnimation *animation = [CABasicAnimation animation];
 animation.keyPath = @"backgroundColor";
 animation.toValue = (__bridge id)color.CGColor;
 animation.delegate = self;
}
```

```

 //apply animation to layer
 [self.colorLayer addAnimation:animation forKey:nil];
}

- (void)animationDidStop:(CABasicAnimation *)anim finished:(BOOL)f
{
 //set the backgroundColor property to match animation toValue
 [CATransaction begin];
 [CATransaction setDisableActions:YES];
 self.colorLayer.backgroundColor = (__bridge CGColorRef)anim.toValue;
 [CATransaction commit];
}

@end

```

对 `CAAnimation` 而言，使用委托模式而不是一个完成块会带来一个问题，就是当你有多个动画的时候，无法在在回调方法中区分。在一个视图控制器中创建动画的时候，通常会用控制器本身作为一个委托（如清单8.3所示），但是所有的动画都会调用同一个回调方法，所以你就需要判断到底是那个图层的调用。

考虑一下第三章的闹钟，“图层几何学”，我们通过简单地每秒更新指针的角度来实现一个钟，但如果指针动态地转向新的位置会更加真实。

我们不能通过隐式动画来实现因为这些指针都是 `UIView` 的实例，所以图层的隐式动画都被禁用了。我们可以简单地通过 `UIView` 的动画方法来实现。但如果想更好地控制动画时间，使用显式动画会更好（更多内容见第十章）。使用 `CABasicAnimation` 来做动画可能会更加复杂，因为我们需要在 `-animationDidStop:finished:` 中检测指针状态（用于设置结束的位置）。

动画本身会作为一个参数传入委托的方法，也许你会认为可以控制器中把动画存储为一个属性，然后在回调用比较，但实际上并不起作用，因为委托传入的动画参数是原始值的一个深拷贝，从而不是同一个值。

当使用 `-addAnimation:forKey:` 把动画添加到图层，这里有一个到目前为止我们都设置为 `nil` 的 `key` 参数。这里的键是 `-animationForKey:` 方法找到对应动画的唯一标识符，而当前动画的所有键都可以用 `animationKeys` 获取。如果我们对每个动画都关联一个唯一的键，就可以对每个图层循环所有键，然后调用 `-animationForKey:` 来比对结果。尽管这不是一个优雅的实现。

幸运的是，还有一种更加简单的方法。像所有的 `NSObject` 子类一样，`CAAnimation` 实现了 KVC（键-值-编码）协议，于是你可以用 `-setValue:forKey:` 和 `-valueForKey:` 方法来存取属性。但是 `CAAnimation` 有一个不同的性能：它更像一个 `NSDictionary`，可以让你随意设置键值对，即使和你使用的动画类所声明的属性并不匹配。

这意味着你可以对动画用任意类型打标签。在这里，我们给 `UIView` 类型的指针添加的动画，所以可以简单地判断动画到底属于哪个视图，然后在委托方法中用这个信息正确地更新钟的指针（清单 8.4）。

#### 清单 8.4 使用 KVC 对动画打标签

```

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIImageView *hourHand;
@property (nonatomic, weak) IBOutlet UIImageView *minuteHand;
@property (nonatomic, weak) IBOutlet UIImageView *secondHand;
@property (nonatomic, weak) NSTimer *timer;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //adjust anchor points
 self.secondHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);
 self.minuteHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);
 self.hourHand.layer.anchorPoint = CGPointMake(0.5f, 0.9f);
 //start timer
 self.timer = [NSTimer scheduledTimerWithTimeInterval:1.0 target:self selector:@selector(tick) userInfo:nil repeats:YES];
 //set initial hand positions
 [self updateHandsAnimated:NO];
}

- (void)tick
{
 [self updateHandsAnimated:YES];
}

```

```

- (void)updateHandsAnimated:(BOOL)animated
{
 //convert time to hours, minutes and seconds
 NSCalendar *calendar = [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];
 NSUInteger units = NSHourCalendarUnit | NSMinuteCalendarUnit | NSSecondCalendarUnit;
 NSDateComponents *components = [calendar components:units fromDate:[NSDate date]];
 CGFloat hourAngle = (components.hour / 12.0) * M_PI * 2.0;
 //calculate hour hand angle //calculate minute hand angle
 CGFloat minuteAngle = (components.minute / 60.0) * M_PI * 2.0;
 //calculate second hand angle
 CGFloat secondAngle = (components.second / 60.0) * M_PI * 2.0;
 //rotate hands
 [self setAngle:hourAngle forHand:self.hourHand animated:animated];
 [self setAngle:minuteAngle forHand:self.minuteHand animated:animated];
 [self setAngle:secondAngle forHand:self.secondHand animated:animated];
}

- (void)setAngle:(CGFloat)angle forHand:(UIView *)handView animated:(BOOL)animated
{
 //generate transform
 CATransform3D transform = CATransform3DMakeRotation(angle, 0, 0, 1);
 if (animated) {
 //create transform animation
 CABasicAnimation *animation = [CABasicAnimation animation];
 [self updateHandsAnimated:NO];
 animation.keyPath = @"transform";
 animation.toValue = [NSValue valueWithCATransform3D:transform];
 animation.duration = 0.5;
 animation.delegate = self;
 [animation setValue:handView forKey:@"handView"];
 [handView.layer addAnimation:animation forKey:nil];
 } else {
 //set transform directly
 handView.layer.transform = transform;
 }
}

- (void)animationDidStop:(CABasicAnimation *)anim finished:(BOOL)finished
{
}

```

```
//set final position for hand view
UIView *handView = [anim valueForKey:@"handView"];
handView.layer.transform = [anim.toValue CATransform3DValue];
}
```

我们成功的识别出每个图层停止动画的时间，然后更新它的变换到一个新值，很好。

不幸的是，即使做了这些，还是有个问题，清单8.4在模拟器上运行的很好，但当真正跑在iOS设备上时，我们发现在 `-animationDidStop:finished:` 委托方法调用之前，指针会迅速返回到原始值，这个清单8.3图层颜色发生的情况一样。

问题在于回调方法在动画完成之前已经被调用了，但不能保证这发生在属性动画返回初始状态之前。这同时也很好地说明了为什么要在真实的设备上测试动画代码，而不仅仅是模拟器。

我们可以用一个 `fillMode` 属性来解决这个问题，下一章会详细说明，这里知道在动画之前设置它比在动画结束之后更新属性更加方便。

## 关键帧动画

`CABasicAnimation` 揭示了大多数隐式动画背后依赖的机制，这的确很有趣，但是显式地给图层添加 `CABasicAnimation` 相较于隐式动画而言，只能说费力不讨好。

`CAKeyframeAnimation` 是另一种UIKit没有暴露出来但功能强大的类。和 `CABasicAnimation` 类似，`CAKeyframeAnimation` 同样是 `CAPropertyAnimation` 的一个子类，它依然作用于单一的一个属性，但是和 `CABasicAnimation` 不一样的是，它不限制于设置一个起始和结束的值，而是可以根据一连串随意的值来做动画。

关键帧起源于传动动画，意思是指主导的动画在显著改变发生时重绘当前帧（也就是关键帧），每帧之间剩下的绘制（可以通过关键帧推算出）将由熟练的艺术家来完成。`CAKeyframeAnimation` 也是同样的道理：你提供了显著的帧，然后Core Animation在每帧之间进行插入。

我们可以用之前使用颜色图层的例子来演示，设置一个颜色的数组，然后通过关键帧动画播放出来（清单8.5）

## 清单8.5 使用 CAKeyframeAnimation 应用一系列颜色的变化

```

- (IBAction)changeColor
{
 //create a keyframe animation
 CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
 animation.keyPath = @"backgroundColor";
 animation.duration = 2.0;
 animation.values = @[
 (__bridge id)[UIColor blueColor].CGColor,
 (__bridge id)[UIColor redColor].CGColor,
 (__bridge id)[UIColor greenColor].CGColor,
 (__bridge id)[UIColor blueColor].CGColor];
 //apply animation to layer
 [self.colorLayer addAnimation:animation forKey:nil];
}

```

注意到序列中开始和结束的颜色都是蓝色，这是因为 CAKeyframeAnimation 并不能自动把当前值作为第一帧（就像 CABasicAnimation 那样把 fromValue 设为 nil）。动画会在开始的时候突然跳转到第一帧的值，然后在动画结束的时候突然恢复到原始的值。所以为了动画的平滑特性，我们需要开始和结束的关键帧来匹配当前属性的值。

当然可以创建一个结束和开始值不同的动画，那样的话就需要在动画启动之前手动更新属性和最后一帧的值保持一致，就和之前讨论的一样。

我们用 duration 属性把动画时间从默认的0.25秒增加到2秒，以便于动画做的不那么快。运行它，你会发现动画通过颜色不断循环，但效果看起来有些奇怪。原因在于动画以一个恒定的步调在运行。当在每个动画之间过渡的时候并没有减速，这就产生了一个略微奇怪的效果，为了让动画看起来更自然，我们需要调整一下缓冲，第十章将会详细说明。

提供一个数组的值就可以按照颜色变化做动画，但一般来说用数组来描述动画运动并不直观。 CAKeyframeAnimation 有另一种方式去指定动画，就是使用 CGPath 。 path 属性可以用一种直观的方式，使用Core Graphics函数定义运动序列来绘制动画。

我们来用一个宇宙飞船沿着一个简单曲线的实例演示一下。为了创建路径，我们需要使用一个三次贝塞尔曲线，它是一种使用开始点，结束点和另外两个控制点来定义形状的曲线，可以通过使用一个基于C的Core Graphics绘图指令来创建，不过用UIKit提供的 UIBezierPath 类会更简单。

我们这次用 CAShapeLayer 来在屏幕上绘制曲线，尽管对动画来说并不是必须的，但这会让我们的动画更加形象。绘制完 CGPath 之后，我们用它来创建一个 CAKeyframeAnimation ，然后用它来应用到我们的宇宙飞船。代码见清单 8.6，结果见图 8.1。

#### 清单 8.6 沿着一个贝塞尔曲线对图层做动画

```
@interface ViewController ()

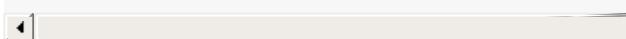
@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create a path
 UIBezierPath *bezierPath = [[UIBezierPath alloc] init];
 [bezierPath moveToPoint:CGPointMake(0, 150)];
 [bezierPath addCurveToPoint:CGPointMake(300, 150) controlPoint:
 //draw the path using a CAShapeLayer
 CAShapeLayer *pathLayer = [CAShapeLayer layer];
 pathLayer.path = bezierPath.CGPath;
 pathLayer.fillColor = [UIColor clearColor].CGColor;
 pathLayer.strokeColor = [UIColor redColor].CGColor;
 pathLayer.lineWidth = 3.0f;
 [self.containerView.layer addSublayer:pathLayer];
 //add the ship
 CALayer *shipLayer = [CALayer layer];
 shipLayer.frame = CGRectMake(0, 0, 64, 64);
 shipLayer.position = CGPointMake(0, 150);
 shipLayer.contents = (__bridge id)[UIImage imageNamed: @"Ship.p
 [self.containerView.layer addSublayer:shipLayer];
 //create the keyframe animation
 CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
 animation.keyPath = @"position";
 animation.duration = 4.0;
 animation.path = bezierPath.CGPath;
 [shipLayer addAnimation:animation forKey:nil];
}

@end
```



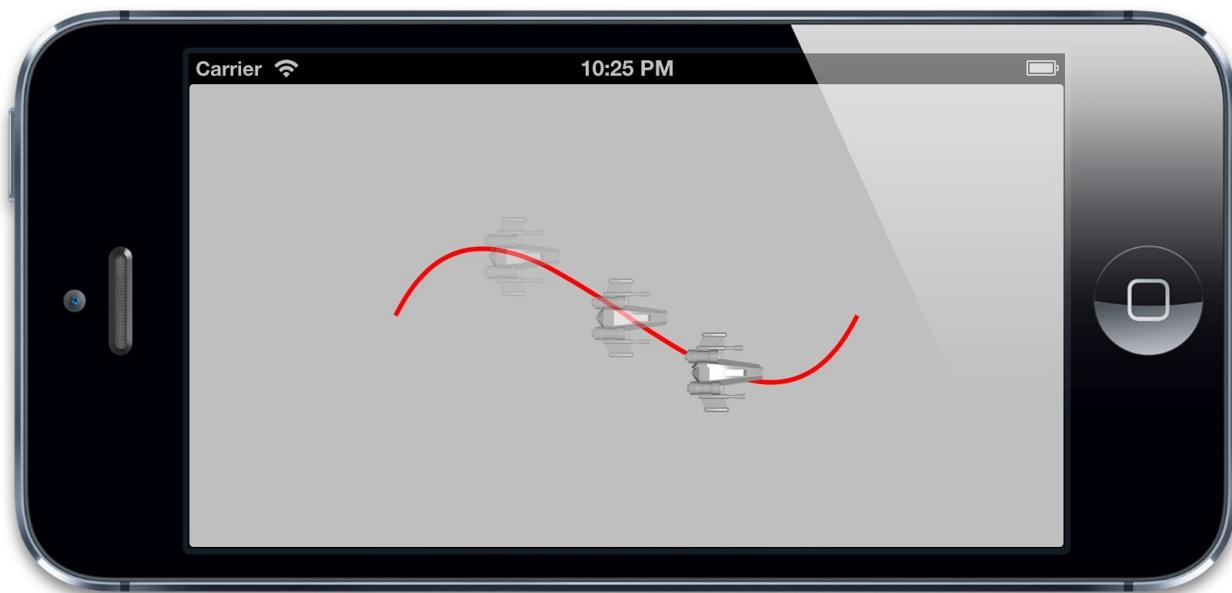


图8.1 沿着一个贝塞尔曲线移动的宇宙飞船图片

运行示例，你会发现飞船的动画有些不太真实，这是因为当它运动的时候永远指向右边，而不是指向曲线切线的方向。你可以调整它的 `affineTransform` 来对运动方向做动画，但很可能和其它的动画冲突。

幸运的是，苹果预见到了这点，并且给 `CAKeyframeAnimation` 添加了一个 `rotationMode` 的属性。设置它为常量 `kCAAnimationRotateAuto`（清单 8.7），图层将会根据曲线的切线自动旋转（图8.2）。

#### 清单8.7 通过 `rotationMode` 自动对齐图层到曲线

```
- (void)viewDidLoad
{
 [super viewDidLoad];
 //create a path
 ...
 //create the keyframe animation
 CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
 animation.keyPath = @"position";
 animation.duration = 4.0;
 animation.path = bezierPath.CGPath;
 animation.rotationMode = kCAAnimationRotateAuto;
 [shipLayer addAnimation:animation forKey:nil];
}
```

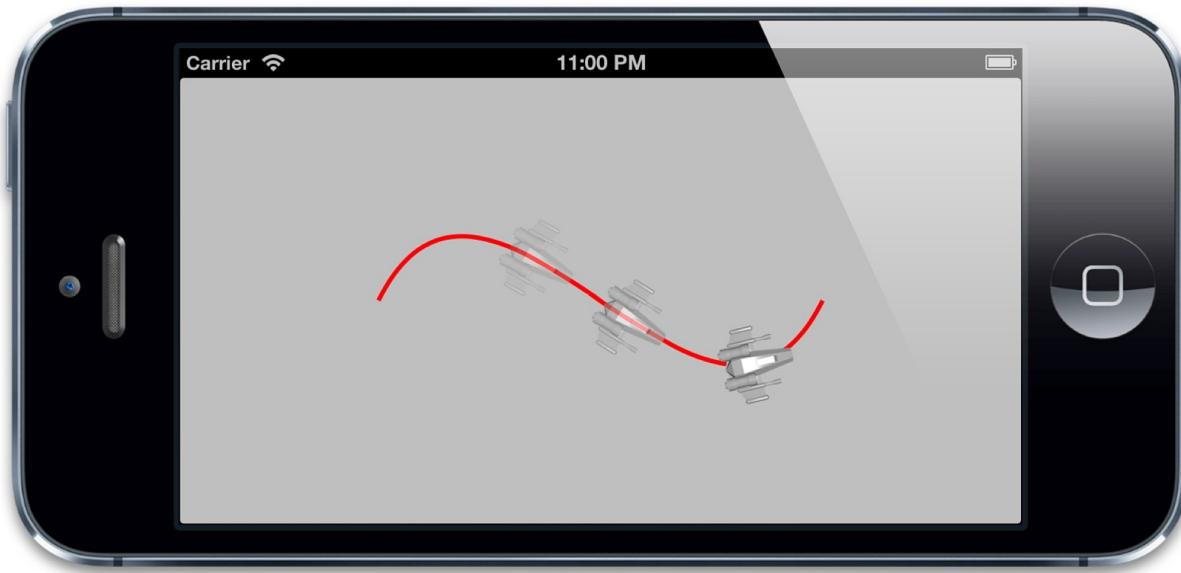


图8.2 匹配曲线切线方向的飞船图层

## 虚拟属性

之前提到过属性动画实际上是针对于关键路径而不是一个键，这就意味着可以对子属性甚至是虚拟属性做动画。但是虚拟属性到底是什么呢？

考虑一个旋转的动画：如果想要对一个物体做旋转的动画，那就需要作用于 `transform` 属性，因为 `CALayer` 没有显式提供角度或者方向之类的属性，代码如清单8.8所示

清单8.8 用 `transform` 属性对图层做动画

```

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //add the ship
 CALayer *shipLayer = [CALayer layer];
 shipLayer.frame = CGRectMake(0, 0, 128, 128);
 shipLayer.position = CGPointMake(150, 150);
 shipLayer.contents = (__bridge id)[UIImage imageNamed: @"Ship.png"];
 [self.containerView.layer addSublayer:shipLayer];
 //animate the ship rotation
 CABasicAnimation *animation = [CABasicAnimation animation];
 animation.keyPath = @"transform";
 animation.duration = 2.0;
 animation.toValue = [NSValue valueWithCATransform3D: CATransform3DMakeRotation(M_PI, 0, 0, 1)];
 [shipLayer addAnimation:animation forKey:nil];
}

@end

```

这么做是可行的，但看起来更像是运气而不是设计的原因，如果我们把旋转的值从 `M_PI`（180度）调整到 `2 * M_PI`（360度），然后运行程序，会发现这时候飞船完全不动了。这是因为这里的矩阵做了一次360度的旋转，和做了0度是一样的，所以最后的值根本没变。

现在继续使用 `M_PI`，但这次用 `byValue` 而不是 `toValue`。也许你会认为这和设置 `toValue` 结果一样，因为 $0 + 90\text{度} == 90\text{度}$ ，但实际上飞船的图片变大了，并没有做任何旋转，这是因为变换矩阵不能像角度值那样叠加。

那么如果需要独立于角度之外单独对平移或者缩放做动画呢？由于都需要我们来修改 `transform` 属性，实时地重新计算每个时间点的每个变换效果，然后根据这些创建一个复杂的关键帧动画，这一切都是为了对图层的一个独立做一个简单的动画。

幸运的是，有一个更好的解决方案：为了旋转图层，我们可以对 `transform.rotation` 关键路径应用动画，而不是 `transform` 本身（清单 8.9）。

#### 清单8.9 对虚拟的 `transform.rotation` 属性做动画

```
@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //add the ship
 CALayer *shipLayer = [CALayer layer];
 shipLayer.frame = CGRectMake(0, 0, 128, 128);
 shipLayer.position = CGPointMake(150, 150);
 shipLayer.contents = (__bridge id)[UIImage imageNamed: @"Ship.png"];
 [self.containerView.layer addSublayer:shipLayer];
 //animate the ship rotation
 CABasicAnimation *animation = [CABasicAnimation animation];
 animation.keyPath = @"transform.rotation";
 animation.duration = 2.0;
 animation.byValue = @(M_PI * 2);
 [shipLayer addAnimation:animation forKey:nil];
}

@end
```

结果运行的特别好，用 `transform.rotation` 而不是 `transform` 做动画的好处如下：

- 我们可以不通过关键帧一步旋转多于180度的动画。
- 可以用相对值而不是绝对值旋转（设置 `byValue` 而不是 `toValue`）。
- 可以不用创建 `CATransform3D`，而是使用一个简单的数值来指定角度。
- 不会和 `transform.position` 或者 `transform.scale` 冲突（同样是使用关键路径来做独立的动画属性）。

`transform.rotation` 属性有一个奇怪的问题是它其实并不存在。这是因为 `CATransform3D` 并不是一个对象，它实际上是一个结构体，也没有符合KVC相关属性，`transform.rotation` 实际上是一个 `CALayer` 用于处理动画变换的虚拟属性。

你不能直接设置 `transform.rotation` 或者 `transform.scale`，他们不能被直接使用。当你对他们做动画时，Core Animation自动地根据通过 `CAValueFunction` 来计算的值来更新 `transform` 属性。

`CAValueFunction` 用于把我们赋给虚拟的 `transform.rotation` 简单浮点值转换成真正的用于摆放图层的 `CATransform3D` 矩阵值。你可以通过设置 `CAPropertyAnimation` 的 `valueFunction` 属性来改变，于是你设置的函数将会覆盖默认的函数。

`CAValueFunction` 看起来似乎是对那些不能简单相加的属性（例如变换矩阵）做动画的非常有用的机制，但由于 `CAValueFunction` 的实现细节是私有的，所以目前不能通过继承它来自定义。你可以通过使用苹果目前已经提供的常量（目前都是和变换矩阵的虚拟属性相关，所以没太多使用场景了，因为这些属性都有了默认的实现方式）。

## 动画组

### 动画组

`CABasicAnimation` 和 `CAKeyframeAnimation` 仅仅作用于单独的属性，而 `CAAnimationGroup` 可以把这些动画组合在一起。`CAAnimationGroup` 是另一个继承于 `CAAnimation` 的子类，它添加了一个 `animations` 数组的属性，用来组合别的动画。我们把清单8.6那种关键帧动画和调整图层背景色的基础动画组合起来（清单8.10），结果如图8.3所示。

清单8.10 组合关键帧动画和基础动画

```
- (void)viewDidLoad
{
 [super viewDidLoad];
 //create a path
 UIBezierPath *bezierPath = [[UIBezierPath alloc] init];
 [bezierPath moveToPoint:CGPointMake(0, 150)];
 [bezierPath addCurveToPoint:CGPointMake(300, 150) controlPoint:
 //draw the path using a CAShapeLayer
 CAShapeLayer *pathLayer = [CAShapeLayer layer];
 pathLayer.path = bezierPath.CGPath;
 pathLayer.fillColor = [UIColor clearColor].CGColor;
 pathLayer.strokeColor = [UIColor redColor].CGColor;
 pathLayer.lineWidth = 3.0f;
 [self.containerView.layer addSublayer:pathLayer];
 //add a colored layer
 CALayer *colorLayer = [CALayer layer];
 colorLayer.frame = CGRectMake(0, 0, 64, 64);
 colorLayer.position = CGPointMake(0, 150);
 colorLayer.backgroundColor = [UIColor greenColor].CGColor;
 [self.containerView.layer addSublayer:colorLayer];
 //create the position animation
 CAKeyframeAnimation *animation1 = [CAKeyframeAnimation animation];
 animation1.keyPath = @"position";
 animation1.path = bezierPath.CGPath;
 animation1.rotationMode = kCAAnimationRotateAuto;
 //create the color animation
 CABasicAnimation *animation2 = [CABasicAnimation animation];
 animation2.keyPath = @"backgroundColor";
 animation2.toValue = (__bridge id)[UIColor redColor].CGColor;
 //create group animation
 CAAimationGroup *groupAnimation = [CAAnimationGroup animation];
 groupAnimation.animations = @[animation1, animation2];
 groupAnimation.duration = 4.0;
 //add the animation to the color layer
 [colorLayer addAnimation:groupAnimation forKey:nil];
}
```

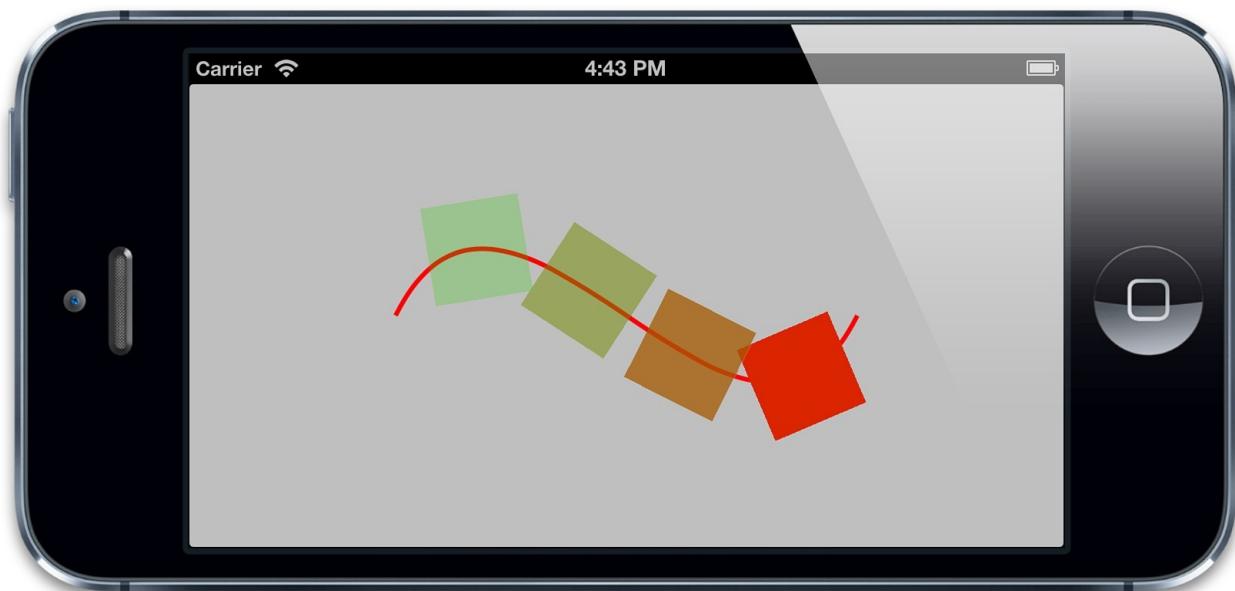


图8.3 关键帧路径和基础动画的组合

## 过渡

有时候对于iOS应用程序来说，希望能通过属性动画来对比较难做动画的布局进行一些改变。比如交换一段文本和图片，或者用一段网格视图来替换，等等。属性动画只对图层的可动画属性起作用，所以如果要改变一个不能动画的属性（比如图片），或者从层级关系中添加或者移除图层，属性动画将不起作用。

于是就有了过渡的概念。过渡并不像属性动画那样平滑地在两个值之间做动画，而是影响到整个图层的变化。过渡动画首先展示之前的图层外观，然后通过一个交换过渡到新的外观。

为了创建一个过渡动画，我们将使用 `CATransition`，同样是另一个 `CAAnimation` 的子类，和别的子类不同，`CATransition` 有一个 `type` 和 `subtype` 来标识变换效果。`type` 属性是一个 `NSString` 类型，可以被设置成如下类型：

```
kCATransitionFade
kCATransitionMoveIn
kCATransitionPush
kCATransitionReveal
```

到目前为止你只能使用上述四种类型，但你可以通过一些别的方法来自定义过渡效果，后续会详细介绍。

默认的过渡类型是 `kCATransitionFade`，当你在改变图层属性之后，就创建了一个平滑的淡入淡出效果。

我们在第七章的例子中就已经用到过 `kCATransitionPush`，它创建了一个新的图层，从边缘的一侧滑动进来，把旧图层从另一侧推出去的效果。

`kCATransitionMoveIn` 和 `kCATransitionReveal` 与 `kCATransitionPush` 类似，都实现了一个定向滑动的动画，但是有一些细微的不同，`kCATransitionMoveIn` 从顶部滑动进入，但不像推送动画那样把老图层推走，然而 `kCATransitionReveal` 把原始的图层滑动出去来显示新的外观，而不是把新的图层滑动进入。

后面三种过渡类型都有一个默认的动画方向，它们都从左侧滑入，但是你可以通过 `subtype` 来控制它们的方向，提供了如下四种类型：

```
kCATransitionFromRight
kCATransitionFromLeft
kCATransitionFromTop
kCATransitionFromBottom
```

一个简单的用 `CATransition` 来对非动画属性做动画的例子如清单8.11所示，这里我们对 `UIImageView` 的 `image` 属性做修改，但是隐式动画或者 `CAPropertyAnimation` 都不能对它做动画，因为Core Animation不知道如何在插图图片。通过对图层应用一个淡入淡出的过渡，我们可以忽略它的内容来做平滑动画（图8.4），我们来尝试修改过渡的 `type` 常量来观察其它效果。

清单8.11 使用 `CATransition` 来对 `UIImageView` 做动画

```

@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIImageView *imageView;
@property (nonatomic, copy) NSArray *images;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //set up images
 self.images = @[[UIImage imageNamed:@"Anchor.png"],
 [UIImage imageNamed:@"Cone.png"],
 [UIImage imageNamed:@"Igloo.png"],
 [UIImage imageNamed:@"Spaceship.png"]];
}

- (IBAction)switchImage
{
 //set up crossfade transition
 CATransition *transition = [CATransition animation];
 transition.type = kCATransitionFade;
 //apply transition to imageview backing layer
 [self.imageView.layer addAnimation:transition forKey:nil];
 //cycle to next image
 UIImage *currentImage = self.imageView.image;
 NSUInteger index = [self.images indexOfObject:currentImage];
 index = (index + 1) % [self.images count];
 self.imageView.image = self.images[index];
}

@end

```

你可以从代码中看出，过渡动画和之前的属性动画或者动画组添加到图层上的方式一致，都是通过 `-addAnimation:forKey:` 方法。但是和属性动画不同的是，对指定的图层一次只能使用一次 `CATransition`，因此，无论你对动画的键设置什么

值，过渡动画都会对它的键设置成“transition”，也就是常量 `kCATransition`。

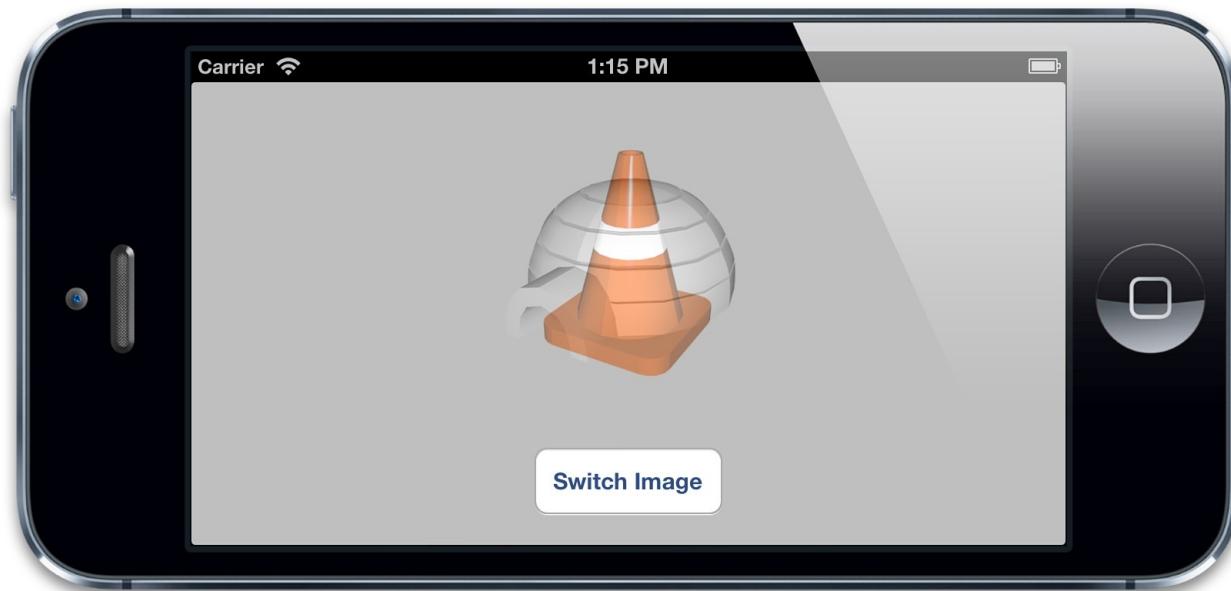


图8.4 使用 `CATransition` 对图像平滑淡入淡出

## 隐式过渡

`CATransition` 可以对图层任何变化平滑过渡的事实使得它成为那些不好做动画的属性图层行为的理想候选。苹果当然意识到了这点，并且当设置了 `CALayer` 的 `content` 属性的时候，`CATransition` 的确是默认的行为。但是对于视图关联的图层，或者是其他隐式动画的行为，这个特性依然是被禁用的，但是对于你自己创建的图层，这意味着对图层 `contents` 图片做的改动都会自动附上淡入淡出的动画。

我们在第七章使用 `CATransition` 作为一个图层行为来改变图层的背景色，当然 `backgroundColor` 属性可以通过正常的 `CAPropertyAnimation` 来实现，但这不是说不可以使用 `CATransition` 来实行。

## 对图层树的动画

`CATransition` 并不作用于指定的图层属性，这就是说你可以在即使不能准确得知改变了什么的情况下对图层做动画，例如，在不知道 `UITableView` 哪一行被添加或者删除的情况下，直接就可以平滑地刷新它，或者在不知道 `UIViewController` 内部的视图层级的情况下对两个不同的实例做过渡动画。

这些例子和我们之前所讨论的情况完全不同，因为它们不仅涉及到图层的属性，而且是整个图层树的改变--我们在这种动画的过程中手动在层级关系中添加或者移除图层。

这里用到了一个小诡计，要确保 `CATransition` 添加到的图层在过渡动画发生时不会在树状结构中被移除，否则 `CATransition` 将会和图层一起被移除。一般来说，你只需要将动画添加到被影响图层的 `superlayer`。

在清单8.2中，我们展示了如何在 `UITabBarController` 切换标签的时候添加淡入淡出的动画。这里我们建立了默认的标签应用程序模板，然后

用 `UITabBarControllerDelegate` 的 -

`tabBarController:didSelectViewController:` 方法来应用过渡动画。我们把动画添加到 `UITabBarController` 的视图图层上，于是在标签被替换的时候动画不会被移除。

清单8.12 对 `UITabBarController` 做动画

```

#import "AppDelegate.h"
#import "FirstViewController.h"
#import "SecondViewController.h"
#import
@implementation AppDelegate
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
 self.window = [[UIWindow alloc] initWithFrame: [[UIScreen mainScreen] bounds]];
 UIViewController *viewController1 = [[FirstViewController alloc] initWithNibName:@"FirstViewController" bundle:nil];
 UIViewController *viewController2 = [[SecondViewController alloc] initWithNibName:@"SecondViewController" bundle:nil];
 self.tabBarController = [[UITabBarController alloc] init];
 self.tabBarController.viewControllers = @[viewController1, viewController2];
 self.tabBarController.delegate = self;
 self.window.rootViewController = self.tabBarController;
 [self.window makeKeyAndVisible];
 return YES;
}
- (void)tabBarController:(UITabBarController *)tabBarController didSelectViewController:(UIViewController *)selectedViewController
{
 //set up crossfade transition
 CATransition *transition = [CATransition animation];
 transition.type = kCATransitionFade;
 //apply transition to tab bar controller's view
 [self.tabBarController.view.layer addAnimation:transition forKey:kCATransition];
}
@end

```

## 自定义动画

我们证实了过渡是一种对那些不太好做平滑动画属性的强大工具，但是 `CATransition` 的提供的动画类型太少了。

更奇怪的是苹果通过 `UIView`

`+transitionFromView:toView:duration:options:completion:` 和 `+transitionWithView:duration:options:animations:` 方法提供了 Core Animation 的过渡特性。但是这里的可用的过渡选项和 `CATransition` 的 `type` 属性提供的常量完全不同。`UIView` 过渡方法中 `options` 参数可以由如下常量指定：

`UIViewControllerAnimatedTransitioning`

`UIViewControllerAnimatedTransitioning`

`UIViewControllerAnimatedTransitioning`

`UIViewControllerAnimatedTransitioning`

`UIViewControllerAnimatedTransitioning`

`UIViewControllerAnimatedTransitioning`

`UIViewControllerAnimatedTransitioning`

除了 `UIViewControllerAnimatedTransitioning` 之外，剩下的值

和 `CATransition` 类型完全没关系。你可以用之前例子修改过的版本来测试一下

（见清单8.13）。

清单8.13 使用UIKit提供的方法来做过渡动画

```

@interface ViewController ()
@property (nonatomic, weak) IBOutlet UIImageView *imageView;
@property (nonatomic, copy) NSArray *images;
@end
@implementation ViewController
- (void)viewDidLoad
{
 [super viewDidLoad]; //set up images
 self.images = @[[UIImage imageNamed:@"Anchor.png"],
 [UIImage imageNamed:@"Cone.png"],
 [UIImage imageNamed:@"Igloo.png"],
 [UIImage imageNamed:@"Spaceship.png"]];
}
- (IBAction)switchImage
{
 [UIView transitionWithView:self.imageView duration:1.0
 options:UIViewAnimationOptionTransitionFlipFromLeft
 animations:^{
 //cycle to next image
 UIImage *currentImage = self.imageView.image;
 NSUInteger index = [self.images indexOfObject:currentImage];
 index = (index + 1) % [self.images count];
 self.imageView.image = self.images[index];
 }
 completion:NULL];
}
@end

```

文档暗示过在iOS5（带来了Core Image框架）之后，可以通过 `CATransition` 的 `filter` 属性，用 `CIFilter` 来创建其它的过渡效果。但是直到iOS6都做不到这点。试图对 `CATransition` 使用Core Image的滤镜完全没效果（但是在Mac OS中是可行的，也许文档是想表达这个意思）。

因此，根据要实现的效果，你只用关心是用 `CATransition` 还是用 `UIView` 的过渡方法就可以了。希望下个版本的iOS系统可以通过 `CATransition` 很好的支持Core Image的过渡滤镜效果（或许甚至会有新的方法）。

但这并不意味着在iOS上就不能实现自定义的过渡效果了。这只是意味着你需要做一些额外的工作。就像之前提到的那样，过渡动画做基础的原则就是对原始的图层外观截图，然后添加一段动画，平滑过渡到图层改变之后那个截图的效果。如果我们知道如何对图层截图，我们就可以使用属性动画来代替 `CATransition` 或者是 `UIKit` 的过渡方法来实现动画。

事实证明，对图层做截图还是很简单的。`CALayer` 有一个 `-renderInContext:` 方法，可以通过把它绘制到Core Graphics的上下文中捕获当前内容的图片，然后在另外的视图中显示出来。如果我们把这个截屏视图置于原始视图之上，就可以遮住真实视图的所有变化，于是重新创建了一个简单的过渡效果。

清单8.14演示了一个基本的实现。我们对当前视图状态截图，然后在我们改变原始视图的背景色的时候对截图快速转动并且淡出，图8.5展示了我们自定义的过渡效果。

为了让事情更简单，我们用 `UIView -animateWithDuration:completion:` 方法来实现。虽然用 `CABasicAnimation` 可以达到同样的效果，但是那样的话我们就需要对图层的变换和不透明属性创建单独的动画，然后当动画结束的是哦户在 `CAAnimationDelegate` 中把 `coverView` 从屏幕中移除。

清单8.14 用 `renderInContext:` 创建自定义过渡效果

```
@implementation ViewController
- (IBAction)performTransition
{
 //preserve the current view snapshot
 UIGraphicsBeginImageContextWithOptions(self.view.bounds.size, YES, 0.0);
 [self.view.layer renderInContext:UIGraphicsGetCurrentContext()];
 UIImage *coverImage = UIGraphicsGetImageFromCurrentImageContext();
 UIGraphicsEndImageContext();

 //insert snapshot view in front of this one
 UIView *coverView = [[UIImageView alloc] initWithImage:coverImage];
 coverView.frame = self.view.bounds;
 [self.view addSubview:coverView];

 //update the view (we'll simply randomize the layer background
 CGFloat red = arc4random() / (CGFloat)INT_MAX;
 CGFloat green = arc4random() / (CGFloat)INT_MAX;
 CGFloat blue = arc4random() / (CGFloat)INT_MAX;
 self.view.backgroundColor = [UIColor colorWithRed:red green:green blue:blue alpha:1.0];

 //perform animation (anything you like)
 [UIView animateWithDuration:1.0 animations:^{
 //scale, rotate and fade the view
 CGAffineTransform transform = CGAffineTransformMakeScale(0.5, 0.5);
 transform = CGAffineTransformRotate(transform, M_PI_2);
 coverView.transform = transform;
 coverView.alpha = 0.0;
 } completion:^(BOOL finished) {
 //remove the cover view now we're finished with it
 [coverView removeFromSuperview];
 }];
}
@end
```

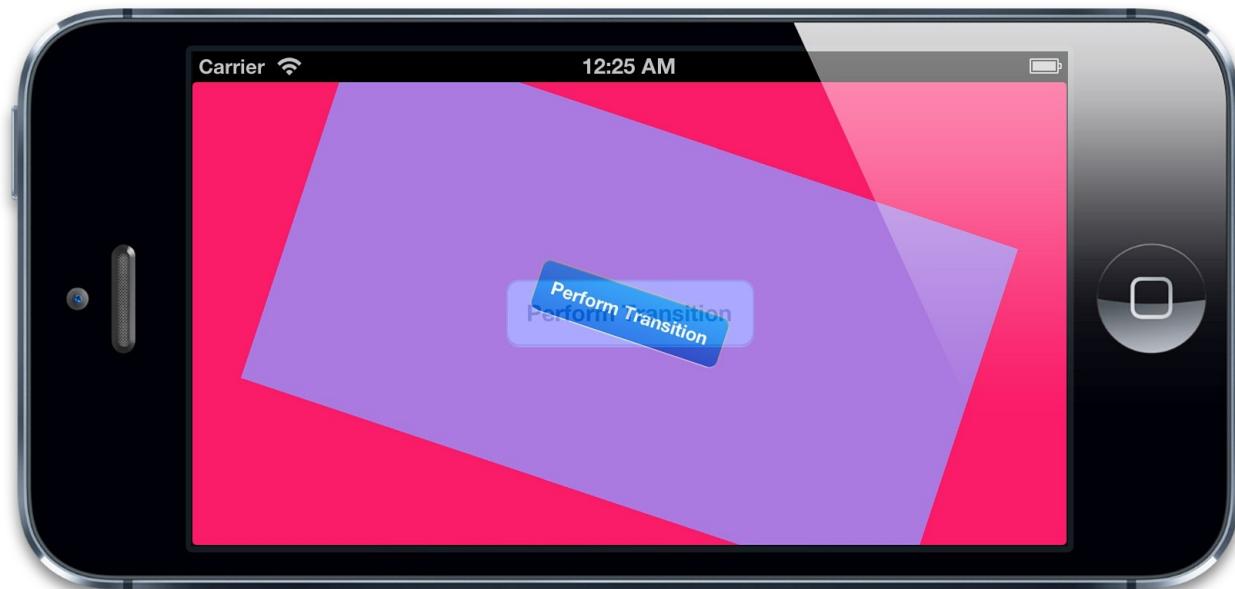


图8.5 使用 `renderInContext:` 创建自定义过渡效果

这里有个警告：`-renderInContext:` 捕获了图层的图片和子图层，但是不能对子图层正确地处理变换效果，而且对视频和OpenGL内容也不起作用。但是用 `CATransition`，或者用私有的截屏方式就没有这个限制了。

# 在动画过程中取消动画

之前提到过，你可以用 `-addAnimation:forKey:` 方法中的 `key` 参数来在添加动画之后检索一个动画，使用如下方法：

```
- (CAAnimation *)animationForKey:(NSString *)key;
```

但并不支持在动画运行过程中修改动画，所以这个方法主要用来检测动画的属性，或者判断它是否被添加到当前图层中。

为了终止一个指定的动画，你可以用如下方法把它从图层移除掉：

```
- (void)removeAnimationForKey:(NSString *)key;
```

或者移除所有动画：

```
- (void)removeAllAnimations;
```

动画一旦被移除，图层的外观就立刻更新到当前的模型图层的值。一般说来，动画在结束之后被自动移除，除非设置 `removedOnCompletion` 为 `NO`，如果你设置动画在结束之后不被自动移除，那么当它不需要的时候你要手动移除它；否则它会一直存在于内存中，直到图层被销毁。

我们来扩展之前旋转飞船的示例，这里添加一个按钮来停止或者启动动画。这一次我们用一个非 `nil` 的值作为动画的键，以便之后可以移除它。-

`animationDidStop:finished:` 方法中的 `flag` 参数表明了动画是自然结束还是被打断，我们可以在控制台打印出来。如果你用停止按钮来终止动画，它会打印 `NO`，如果允许它完成，它会打印 `YES`。

清单8.15是更新后的示例代码，图8.6显示了结果。

清单8.15 开始和停止一个动画

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
```

```
@property (nonatomic, strong) CALayer *shipLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //add the ship
 self.shipLayer = [CALayer layer];
 self.shipLayer.frame = CGRectMake(0, 0, 128, 128);
 self.shipLayer.position = CGPointMake(150, 150);
 self.shipLayer.contents = (__bridge id)[UIImage imageNamed: @""];
 [self.containerView.layer addSublayer:self.shipLayer];
}

- (IBAction)start
{
 //animate the ship rotation
 CABasicAnimation *animation = [CABasicAnimation animation];
 animation.keyPath = @"transform.rotation";
 animation.duration = 2.0;
 animation.byValue = @(_M_PI * 2);
 animation.delegate = self;
 [self.shipLayer addAnimation:animation forKey:@"rotateAnimation"];
}

- (IBAction)stop
{
 [self.shipLayer removeAnimationForKey:@"rotateAnimation"];
}

- (void)animationDidStop:(CAAnimation *)anim finished:(BOOL)flag
{
 //log that the animation stopped
 NSLog(@"The animation stopped (finished: %@)", flag? @"YES": @"");
}

@end
```

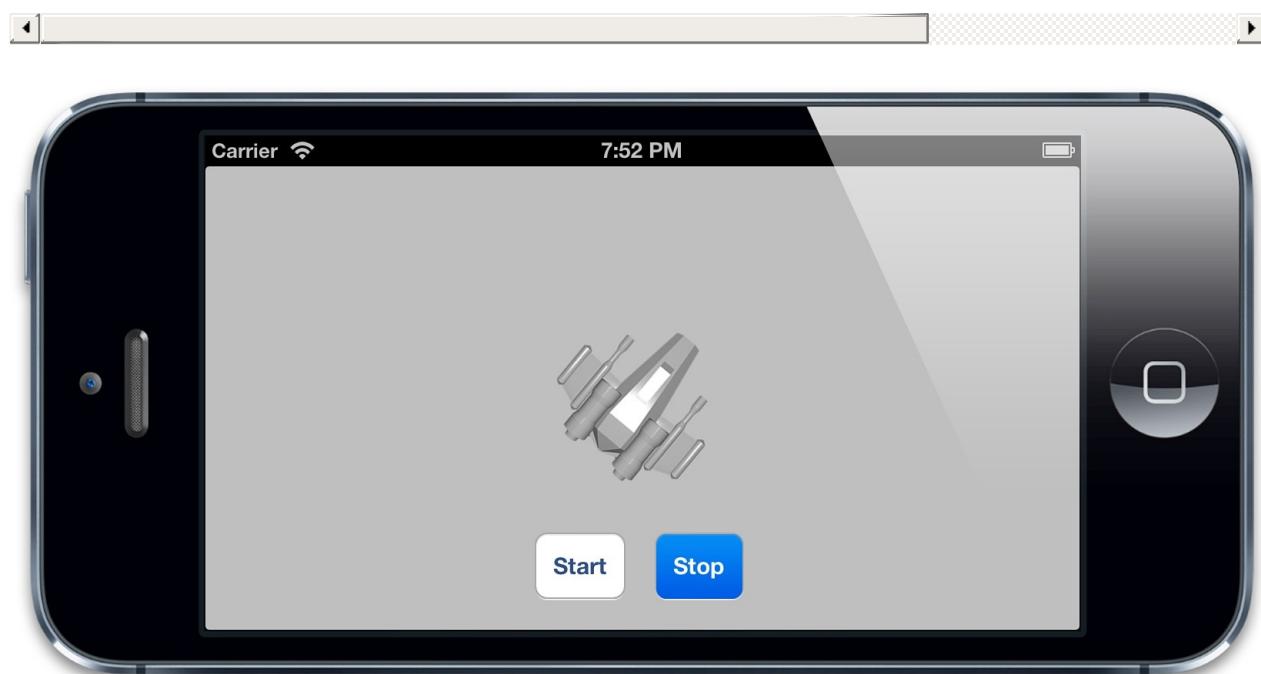


图8.6 通过开始和停止按钮控制的旋转动画

## 总结

这一章中，我们涉及了属性动画（你可以对单独的图层属性动画有更加具体的控制），动画组（把多个属性动画组合成一个独立单元）以及过度（影响整个图层，可以用来对图层的任何内容做任何类型的动画，包括子图层的添加和移除）。

在第九章中，我们继续学习 `CAMediaTiming` 协议，来看一看Core Animation是怎样处理逝去的时间。

## 图层时间

时间和空间最大的区别在于，时间不能被复用 -- 弗斯特梅里克

在上面两章中，我们探讨了可以用 `CAAnimation` 和它的子类实现的多种图层动画。动画的发生是需要持续一段时间的，所以计时对整个概念来说至关重要。在这一章中，我们来看看 `CAMediaTiming`，看看Core Animation是如何跟踪时间的。

## CAMediaTiming 协议

`CAMediaTiming` 协议定义了在一段动画内用来控制逝去时间的属性的集合，`CALayer` 和 `CAAnimation` 都实现了这个协议，所以时间可以被任意基于一个图层或者一段动画的类控制。

### 持续和重复

我们在第八章“显式动画”中简单提到过 `duration`（`CAMediaTiming` 的属性之一），`duration` 是一个 `CFTimeInterval` 的类型（类似于 `NSTimeInterval` 的一种双精度浮点类型），对将要进行的动画的一次迭代指定了时间。

这里的一次迭代是什么意思呢？`CAMediaTiming` 另外还有一个属性叫做 `repeatCount`，代表动画重复的迭代次数。如果 `duration` 是 2，`repeatCount` 设为 3.5（三个半迭代），那么完整的动画时长将是 7 秒。

`duration` 和 `repeatCount` 默认都是 0。但这不意味着动画时长为 0 秒，或者 0 次，这里的 0 仅仅代表了“默认”，也就是 0.25 秒和 1 次，你可以用一个简单的测试来尝试为这两个属性赋多个值，如清单 9.1，图 9.1 展示了程序的结果。

#### 清单 9.1 测试 `duration` 和 `repeatCount`

```

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, weak) IBOutlet UITextField *durationField;
@property (nonatomic, weak) IBOutlet UITextField *repeatField;
@property (nonatomic, weak) IBOutlet UIButton *startButton;
@property (nonatomic, strong) CALayer *shipLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
}

```

```

 //add the ship
 self.shipLayer = [CALayer layer];
 self.shipLayer.frame = CGRectMake(0, 0, 128, 128);
 self.shipLayer.position = CGPointMake(150, 150);
 self.shipLayer.contents = (__bridge id)[UIImage imageNamed: @"ship"];
 [self.containerView.layer addSublayer:self.shipLayer];
}

- (void)setControlsEnabled:(BOOL)enabled
{
 for (UIControl *control in @[self.durationField, self.repeatField])
 control.enabled = enabled;
 control.alpha = enabled? 1.0f: 0.25f;
}
}

- (IBAction)hideKeyboard
{
 [self.durationField resignFirstResponder];
 [self.repeatField resignFirstResponder];
}

- (IBAction)start
{
 CFTimeInterval duration = [self.durationField.text doubleValue];
 float repeatCount = [self.repeatField.text floatValue];
 //animate the ship rotation
 CABasicAnimation *animation = [CABasicAnimation animation];
 animation.keyPath = @"transform.rotation";
 animation.duration = duration;
 animation.repeatCount = repeatCount;
 animation.byValue = @(M_PI * 2);
 animation.delegate = self;
 [self.shipLayer addAnimation:animation forKey:@"rotateAnimation"];
 //disable controls
 [self setControlsEnabled:NO];
}

- (void)animationDidStop:(CAAnimation *)anim finished:(BOOL)flag
{

```

```
//reenable controls
[self setControlsEnabled:YES];
}

@end
```

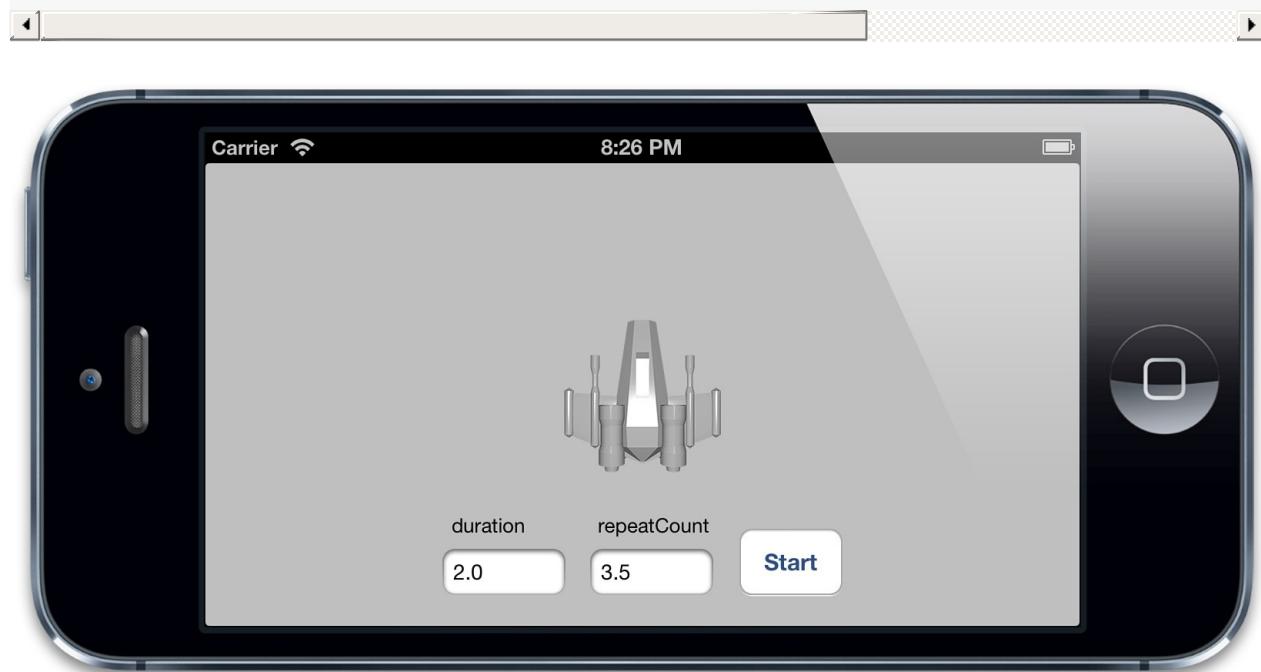
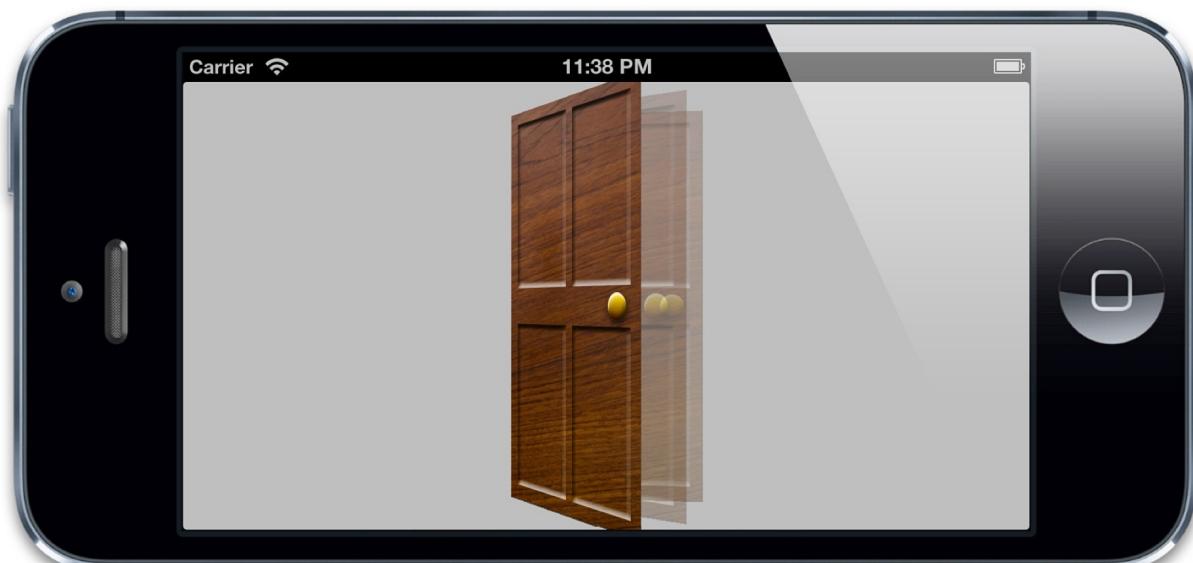


图9.1 演示 duration 和 repeatCount 的测试程序

创建重复动画的另一种方式是使用 `repeatDuration` 属性，它让动画重复一个指定的时间，而不是指定次数。你甚至设置一个叫做 `autoreverses` 的属性（BOOL类型）在每次间隔交替循环过程中自动回放。这对于播放一段连续非循环的动画很有用，例如打开一扇门，然后关上它（图9.2）。



## 图9.2 摆动门的动画

对门进行摆动的代码见清单9.2。我们用了 `autoreverses` 来使门在打开后自动关闭，在这里我们把 `repeatDuration` 设置为 `INFINITY`，于是动画无限循环播放，设置 `repeatCount` 为 `INFINITY` 也有同样的效果。注意 `repeatCount` 和 `repeatDuration` 可能会相互冲突，所以你只要对其中一个指定非零值。对两个属性都设置非0值的行为没有被定义。

### 清单9.2 使用 `autoreverses` 属性实现门的摇摆

```

@interface ViewController ()

@property (nonatomic, weak) UIView *containerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //add the door
 CALayer *doorLayer = [CALayer layer];
 doorLayer.frame = CGRectMake(0, 0, 128, 256);
 doorLayer.position = CGPointMake(150 - 64, 150);
 doorLayer.anchorPoint = CGPointMake(0, 0.5);
 doorLayer.contents = (__bridge id)[UIImage imageNamed: @"Door.png"];
 [self.containerView.layer addSublayer:doorLayer];
 //apply perspective transform
 CATransform3D perspective = CATransform3DIdentity;
 perspective.m34 = -1.0 / 500.0;
 self.containerView.layer.sublayerTransform = perspective;
 //apply swinging animation
 CABasicAnimation *animation = [CABasicAnimation animation];
 animation.keyPath = @"transform.rotation.y";
 animation.toValue = @(-M_PI_2);
 animation.duration = 2.0;
 animation.repeatDuration = INFINITY;
 animation.autoreverses = YES;
 [doorLayer addAnimation:animation forKey:nil];
}

@end

```

## 相对时间

每次讨论到Core Animation，时间都是相对的，每个动画都有它自己描述的时间，可以独立地加速，延时或者偏移。

`beginTime` 指定了动画开始之前的的延迟时间。这里的延迟从动画添加到可见图层的那一刻开始测量，默认是0（就是说动画会立刻执行）。

`speed` 是一个时间的倍数，默认1.0，减少它会减慢图层/动画的时间，增加它会加快速度。如果2.0的速度，那么对于一个 `duration` 为1的动画，实际上在0.5秒的时候就已经完成了。

`timeOffset` 和 `beginTime` 类似，但是和增加 `beginTime` 导致的延迟动画不同，增加 `timeOffset` 只是让动画快进到某一点，例如，对于一个持续1秒的动画来说，设置 `timeOffset` 为0.5意味着动画将从一半的地方开始。

和 `beginTime` 不同的是，`timeOffset` 并不受 `speed` 的影响。所以如果你把 `speed` 设为2.0，把 `timeOffset` 设置为0.5，那么你的动画将从动画最后结束的地方开始，因为1秒的动画实际上被缩短到了0.5秒。然而即使使用了 `timeOffset` 让动画从结束的地方开始，它仍然播放了一个完整的时长，这个动画仅仅是循环了一圈，然后从头开始播放。

可以用清单9.3的测试程序验证一下，设置 `speed` 和 `timeOffset` 滑块到随意的值，然后点击播放来观察效果（见图9.3）

### 清单9.3 测试 `timeOffset` 和 `speed` 属性

```

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, weak) IBOutlet UILabel *speedLabel;
@property (nonatomic, weak) IBOutlet UILabel *timeOffsetLabel;
@property (nonatomic, weak) IBOutlet UISlider *speedSlider;
@property (nonatomic, weak) IBOutlet UISlider *timeOffsetSlider;
@property (nonatomic, strong) UIBezierPath *bezierPath;
@property (nonatomic, strong) CALayer *shipLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create a path
}

```

```

 self.bezierPath = [[UIBezierPath alloc] init];
 [self.bezierPath moveToPoint:CGPointMake(0, 150)];
 [self.bezierPath addCurveToPoint:CGPointMake(300, 150) controlPoint1:CGPointMake(150, 150) controlPoint2:CGPointMake(250, 150)];
 //draw the path using a CAShapeLayer
 CAShapeLayer *pathLayer = [CAShapeLayer layer];
 pathLayer.path = self.bezierPath.CGPath;
 pathLayer.fillColor = [UIColor clearColor].CGColor;
 pathLayer.strokeColor = [UIColor redColor].CGColor;
 pathLayer.lineWidth = 3.0f;
 [self.containerView.layer addSublayer:pathLayer];
 //add the ship
 self.shipLayer = [CALayer layer];
 self.shipLayer.frame = CGRectMake(0, 0, 64, 64);
 self.shipLayer.position = CGPointMake(0, 150);
 self.shipLayer.contents = (__bridge id)[UIImage imageNamed: @"ship.png"];
 [self.containerView.layer addSublayer:self.shipLayer];
 //set initial values
 [self updateSliders];
}

- (IBAction)updateSliders
{
 CFTimeInterval timeOffset = self.timeOffsetSlider.value;
 self.timeOffsetLabel.text = [NSString stringWithFormat:@"%.2f", timeOffset];
 float speed = self.speedSlider.value;
 self.speedLabel.text = [NSString stringWithFormat:@"%.2f", speed];
}

- (IBAction)play
{
 //create the keyframe animation
 CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
 animation.keyPath = @"position";
 animation.timeOffset = self.timeOffsetSlider.value;
 animation.speed = self.speedSlider.value;
 animation.duration = 1.0;
 animation.path = self.bezierPath.CGPath;
 animation.rotationMode = kCAAnimationRotateAuto;
 animation.removedOnCompletion = NO;
 [self.shipLayer addAnimation:animation forKey:@"slide"];
}

```

```
}
```

```
@end
```



图9.3 测试时间偏移和速度的简单的应用程序

## fillMode

对于 `beginTime` 非0的一段动画来说，会出现一个当动画添加到图层上但什么也没发生的状态。类似的，`removeOnCompletion` 被设置为 `NO` 的动画将会在动画结束的时候仍然保持之前的状态。这就产生了一个问题，当动画开始之前和动画结束之后，被设置动画的属性将会是什么值呢？

一种可能是属性和动画没被添加之前保持一致，也就是在模型图层定义的值（见第七章“隐式动画”，模型图层和呈现图层的解释）。

另一种可能是保持动画开始之前那一帧，或者动画结束之后的那一帧。这就是所谓的填充，因为动画开始和结束的值用来填充开始之前和结束之后的时间。

这种行为就交给开发者了，它可以被 `CAMediaTiming` 的 `fillMode` 来控制。`fillMode` 是一个 `NSString` 类型，可以接受如下四种常量：

```
kCAFillModeForwards
kCAFillModeBackwards
kCAFillModeBoth
kCAFillModeRemoved
```

默认是 `kCAFillModeRemoved`，当动画不再播放的时候就显示图层模型指定的值剩下的三种类型向前，向后或者即向前又向后去填充动画状态，使得动画在开始前或者结束后仍然保持开始和结束那一刻的值。

这就对避免在动画结束的时候急速返回提供另一种方案（见第八章）。但是记住了，当用它来解决这个问题的时候，需要把 `removeOnCompletion` 设置为 `NO`，另外需要给动画添加一个非空的键，于是可以在不需要动画的时候把它从图层上移除。

## 层级关系时间

在第三章“图层几何学”中，你已经了解到每个图层是如何相对在图层树中的父图层定义它的坐标系的。动画时间和它类似，每个动画和图层在时间上都有它自己的层级概念，相对于它的父亲来测量。对图层调整时间将会影响到它本身和子图层的动画，但不会影响到父图层。另一个相似点是所有的动画都被按照层级组合（使用 `CAAnimationGroup` 实例）。

对 `CALayer` 或者 `CAGroupAnimation` 调

整 `duration` 和 `repeatCount` / `repeatDuration` 属性并不会影响到子动画。但是 `beginTime`，`timeOffset` 和 `speed` 属性将会影响到子动画。然而在层级关系中，`beginTime` 指定了父图层开始动画（或者组合关系中的父动画）和对象将要开始自己动画之间的偏移。类似的，调

整 `CALayer` 和 `CAGroupAnimation` 的 `speed` 属性将会对动画以及子动画速度应用一个缩放的因子。

## 全局时间和本地时间

CoreAnimation有一个全局时间的概念，也就是所谓的马赫时间（“马赫”实际上是 iOS和Mac OS系统内核的命名）。马赫时间在设备上所有进程都是全局的--但是在不同设备上并不是全局的--不过这已经足够对动画的参考点提供便利了，你可以使用 `CACurrentMediaTime` 函数来访问马赫时间：

```
CFTimeInterval time = CACurrentMediaTime();
```

这个函数返回的值其实无关紧要（它返回了设备自从上次启动后的秒数，并不是你所关心的），它真实的作用在于对动画的时间测量提供了一个相对值。注意当设备休眠的时候马赫时间会暂停，也就是所有的 `CAAnimations`（基于马赫时间）同样也会暂停。

因此马赫时间对长时间测量并不有用。比如用 `CACurrentMediaTime` 去更新一个实时闹钟并不明智。（可以用 `[NSDate date]` 代替，就像第三章例子所示）。

每个 `CALayer` 和 `CAAnimation` 实例都有自己本地时间的概念，是根据父图层/动画层级关系中的 `beginTime`，`timeOffset` 和 `speed` 属性计算。就和转换不同图层之间坐标关系一样，`CALayer` 同样也提供了方法来转换不同图层之间的本地时间。如下：

```
- (CFTimeInterval)convertTime:(CFTimeInterval)t fromLayer:(CALayer *)fromLayer toLayer:(CALayer *)toLayer;
```

当用来同步不同图层之间有不同的 `speed`，`timeOffset` 和 `beginTime` 的动画，这些方法会很有用。

## 暂停，倒回和快进

设置动画的 `speed` 属性为0可以暂停动画，但在动画被添加到图层之后不太可能再修改它了，所以不能对正在进行的动画使用这个属性。给图层添加一个 `CAAnimation` 实际上是给动画对象做了一个不可改变的拷贝，所以对原始动画对象属性的改变对真实的动画并没有作用。相反，直接用 `-animationForKey:` 来检索图层正在进行的动画可以返回正确的动画对象，但是修改它的属性将会抛出异常。

如果移除图层正在进行的动画，图层将会急速返回动画之前的状态。但如果在动画移除之前拷贝图层到模型图层，动画将会看起来暂停在那里。但是不好的地方在于之后就不能再恢复动画了。

一个简单的方法是可以利用 `CAMediaTiming` 来暂停图层本身。如果把图层的 `speed` 设置成0，它会暂停任何添加到图层上的动画。类似的，设置 `speed` 大于1.0将会快进，设置成一个负值将会倒回动画。

通过增加主窗口图层的 `speed`，可以暂停整个应用程序的动画。这对UI自动化提供了好处，我们可以加速所有的视图动画来进行自动化测试（注意对于在主窗口之外的视图并不会被影响，比如 `UIAlertView`）。可以在app delegate设置如下进行验证：

```
self.window.layer.speed = 100;
```

你也可以通过这种方式来减速，但其实也可以在模拟器通过切换慢速动画来实现。

## 手动动画

`timeOffset` 一个很有用的功能在于你可以让你手动控制动画进程，通过设置 `speed` 为 0，可以禁用动画的自动播放，然后来使用 `timeOffset` 来来回显示动画序列。这可以使得运用手势来手动控制动画变得很简单。

举个简单的例子：还是之前关门的动画，修改代码来用手势控制动画。我们给视图添加一个 `UIPanGestureRecognizer`，然后用 `timeOffset` 左右摇晃。

因为在动画添加到图层之后不能再做修改了，我们来通过调整 `layer` 的 `timeOffset` 达到同样的效果（清单9.4）。

清单9.4 通过触摸手势手动控制动画

```

@interface ViewController ()

@property (nonatomic, weak) UIView *containerView;
@property (nonatomic, strong) CALayer *doorLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //add the door
 self.doorLayer = [CALayer layer];
 self.doorLayer.frame = CGRectMake(0, 0, 128, 256);
 self.doorLayer.position = CGPointMake(150 - 64, 150);
 self.doorLayer.anchorPoint = CGPointMake(0, 0.5);
 self.doorLayer.contents = (__bridge id)[UIImage imageNamed:@"Door"];
 [self.containerView.layer addSublayer:self.doorLayer];
 //apply perspective transform
 CATransform3D perspective = CATransform3DIdentity;
 perspective.m34 = -1.0 / 500.0;
 self.containerView.layer.sublayerTransform = perspective;
 //add pan gesture recognizer to handle swipes
 UIPanGestureRecognizer *pan = [[UIPanGestureRecognizer alloc] :

```

```

[pan addTarget:self action:@selector(pan:)];
[self.view addGestureRecognizer:pan];
//pause all layer animations
self.doorLayer.speed = 0.0;
//apply swinging animation (which won't play because layer is paused)
CABasicAnimation *animation = [CABasicAnimation animation];
animation.keyPath = @"transform.rotation.y";
animation.toValue = @(-M_PI_2);
animation.duration = 1.0;
[self.doorLayer addAnimation:animation forKey:nil];
}

- (void)pan:(UIPanGestureRecognizer *)pan
{
 //get horizontal component of pan gesture
 CGFloat x = [pan translationInView:self.view].x;
 //convert from points to animation duration //using a reasonable multiplier
 x /= 200.0f;
 //update timeOffset and clamp result
 CFTimeInterval timeOffset = self.doorLayer.timeOffset;
 timeOffset = MIN(0.999, MAX(0.0, timeOffset - x));
 self.doorLayer.timeOffset = timeOffset;
 //reset pan gesture
 [pan setTranslation:CGPointZero inView:self.view];
}

@end

```

这其实是个小诡计，也许相对于设置个动画然后每次显示一帧而言，用移动手势来直接设置门的 `transform` 会更简单。

在这个例子中的确是这样，但是对于比如说关键帧这样更加复杂的情况，或者有多个图层的动画组，相对于实时计算每个图层的属性而言，这就显得方便的多了。

## 总结

在这一章，我们了解了 `CAMediaTiming` 协议，以及Core Animation用来操作时间控制动画的机制。在下一章，我们将要接触 缓冲 ，另一个用来使动画更加真实的操作时间的技术。

## 缓冲

生活和艺术一样，最美的永远是曲线。 -- 爱德华布尔沃 - 利顿

在第九章“图层时间”中，我们讨论了动画时间和 `CAMediaTiming` 协议。现在我们来看一下另一个和时间相关的机制--所谓的缓冲。`Core Animation` 使用缓冲来使动画移动更平滑更自然，而不是看起来的那种机械和人工，在这一章我们将要研究如何对你的动画控制和自定义缓冲曲线。

## 动画速度

动画实际上就是一段时间内的变化，这就暗示了变化一定是随着某个特定的速率进行。速率由以下公式计算而来：

$$\text{velocity} = \text{change} / \text{time}$$

这里的变化可以指的是一个物体移动的距离，时间指动画持续的时长，用这样一个移动可以更加形象的描述（比如 `position` 和 `bounds` 属性的动画），但实际上它应用于任意可以做动画的属性（比如 `color` 和 `opacity`）。

上面的等式假设了速度在整个动画过程中都是恒定不变的（就如同第八章“显式动画”的情况），对于这种恒定速度的动画我们称之为“线性步调”，而且从技术的角度而言这也是实现动画最简单的方式，但也是完全不真实的一种效果。

考虑一个场景，一辆车行驶在一定距离内，它并不会一开始以60mph的速度行驶，然后到达终点后突然变成0mph。一是因为需要无限大的加速度（即使是最好的车也不会在0秒内从0跑到60），另外不然的话会干死所有乘客。在现实中，它会慢慢地加速到全速，然后当它接近终点的时候，它会慢慢地减速，直到最后停下来。

那么对于一个掉落到地上的物体又会怎样呢？它会首先停在空中，然后一直加速到落到地面，然后突然停止（然后由于积累的动能转换伴随着一声巨响，砰！）。

现实生活中的任何一个物体都会在运动中加速或者减速。那么我们如何在动画中实现这种加速度呢？一种方法是使用物理引擎来对运动物体的摩擦和动量来建模，然而这会使得计算过于复杂。我们称这种类型的方程为缓冲函数，幸运的是，Core Animation内嵌了一系列标准函数提供给我们使用。

## CAMediaTimingFunction

那么该如何使用缓冲方程式呢？首先需要设置 `CAAnimation` 的 `timingFunction` 属性，是 `CAMediaTimingFunction` 类的一个对象。如果想改变隐式动画的计时函数，同样也可以使用 `CATransaction` 的 `+setAnimationTimingFunction:` 方法。

这里有一些方式来创建 `CAMediaTimingFunction`，最简单的方式是调用 `+timingFunctionWithName:` 的构造方法。这里传入如下几个常量之一：

```
kCAMediaTimingFunctionLinear
kCAMediaTimingFunctionEaseIn
kCAMediaTimingFunctionEaseOut
kCAMediaTimingFunctionEaseInEaseOut
kCAMediaTimingFunctionDefault
```

`kCAMediaTimingFunctionLinear` 选项创建了一个线性的计时函数，同样也是 `CAAnimation` 的 `timingFunction` 属性为空时候的默认函数。线性步调对于那些立即加速并且保持匀速到达终点的场景会有意义（例如射出枪膛的子弹），但是默认来说它看起来很奇怪，因为对大多数的动画来说确实很少用到。

`kCAMediaTimingFunctionEaseIn` 常量创建了一个慢慢加速然后突然停止的方法。对于之前提到的自由落体的例子来说很适合，或者比如对准一个目标的导弹的发射。

`kCAMediaTimingFunctionEaseOut` 则恰恰相反，它以一个全速开始，然后慢慢减速停止。它有一个削弱的效果，应用的场景比如一扇门慢慢地关上，而不是砰地一声。

`kCAMediaTimingFunctionEaseInEaseOut` 创建了一个慢慢加速然后再慢慢减速的过程。这是现实世界大多数物体移动的方式，也是大多数动画来说最好的选择。如果只可以用一种缓冲函数的话，那就必须是它了。那么你会疑惑为什么这不是默认的选择，实际上当使用 `UIView` 的动画方法时，他的确是默认的，但当创建 `CAAnimation` 的时候，就需要手动设置它了。

最后还有一个 `kCAMediaTimingFunctionDefault`，它和 `kCAMediaTimingFunctionEaseInEaseOut` 很类似，但是加速和减速的过程都稍微有些慢。它和 `kCAMediaTimingFunctionEaseInEaseOut` 的区别很难察觉，可能是苹果觉得它对于隐式动画来说更适合（然后对UIKit就改变了想法，而是使用 `kCAMediaTimingFunctionEaseInEaseOut` 作为默认效果），虽然它的名字说是默认的，但还是要记住当创建显式的 `CAAnimation` 它并不是默认选项（换句话说，默认的图层行为动画用 `kCAMediaTimingFunctionDefault` 作为它们的计时方法）。

你可以使用一个简单的测试工程来实验一下（清单10.1），在运行之前改变缓冲函数的代码，然后点击任何地方来观察图层是如何通过指定的缓冲移动的。

### 清单10.1 缓冲函数的简单测试

```
@interface ViewController ()

@property (nonatomic, strong) CALayer *colorLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create a red layer
 self.colorLayer = [CALayer layer];
 self.colorLayer.frame = CGRectMake(0, 0, 100, 100);
 self.colorLayer.position = CGPointMake(self.view.bounds.size.w:
 self.colorLayer.backgroundColor = [UIColor redColor].CGColor;
 [self.view.layer addSublayer:self.colorLayer];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
 //configure the transaction
 [CATransaction begin];
 [CATransaction setAnimationDuration:1.0];
 [CATransaction setAnimationTimingFunction:[CAMediaTimingFunction
 //set the position
 self.colorLayer.position = [[touches anyObject] locationInView:
 //commit transaction
 [CATransaction commit];
}

@end
```

## UIView 的动画缓冲

UIKit的动画也同样支持这些缓冲方法的使用，尽管语法和常量有些不同，为了改变 UIView 动画的缓冲选项，给 options 参数添加如下常量之一：

```
UIViewControllerAnimatedOptionCurveEaseInOut
UIViewControllerAnimatedOptionCurveEaseIn
UIViewControllerAnimatedOptionCurveEaseOut
UIViewControllerAnimatedOptionCurveLinear
```

它们和 CAMediaTimingFunction 紧密关联， UIViewAnimationOptionCurveEaseInOut 是默认值（这里没有 kCAMediaTimingFunctionDefault 相对应的值了）。

具体使用方法见清单10.2（注意到这里不再使用 UIView 额外添加的图层，因为 UIKit的动画并不支持这类图层）。

清单10.2 使用UIKit动画的缓冲测试工程

```
@interface ViewController ()

@property (nonatomic, strong) UIView *colorView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create a red layer
 self.colorView = [[UIView alloc] init];
 self.colorView.bounds = CGRectMake(0, 0, 100, 100);
 self.colorView.center = CGPointMake(self.view.bounds.size.width / 2,
 self.view.bounds.size.height / 2);
 self.colorView.backgroundColor = [UIColor redColor];
 [self.view addSubview:self.colorView];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
 //perform the animation
 [UIView animateWithDuration:1.0 delay:0.0
 options:UIViewAnimationOptionCurveEaseOut
 animations:^{
 //set the position
 self.colorView.center = [[touches anyObject] locationInView:
 self.view];
 }
 completion:NULL];
}

@end
```

## 缓冲和关键帧动画

或许你会回想起第八章里面颜色切换的关键帧动画由于线性变换的原因（见清单8.5）看起来有些奇怪，使得颜色变换非常不自然。为了纠正这点，我们来用更加合适的缓冲方法，例如 `kCAMediaTimingFunctionEaseIn`，给图层的颜色变化添加一点脉冲效果，让它更像现实中的一个彩色灯泡。

我们不想给整个动画过程应用这个效果，我们希望对每个动画的过程重复这样的缓冲，于是每次颜色的变换都会有脉冲效果。

`CAKeyframeAnimation` 有一个 `NSArray` 类型的 `timingFunctions` 属性，我们可以用它来对每次动画的步骤指定不同的计时函数。但是指定函数的个数一定要等于 `keyframes` 数组的元素个数减一，因为它是描述每一帧之间动画速度的函数。

在这个例子中，我们自始至终想使用同一个缓冲函数，但我们同样需要一个函数的数组来告诉动画不停地重复每个步骤，而不是在整个动画序列只做一次缓冲，我们简单地使用包含多个相同函数拷贝的数组就可以了（见清单10.3）。

运行更新后的代码，你会发现动画看起来更加自然了。

#### 清单10.3 对 `CAKeyframeAnimation` 使用 `CAMediaTimingFunction`

```

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;
@property (nonatomic, weak) IBOutlet CALayer *colorLayer;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create sublayer
 self.colorLayer = [CALayer layer];
 self.colorLayer.frame = CGRectMake(50.0f, 50.0f, 100.0f, 100.0f);
 self.colorLayer.backgroundColor = [UIColor blueColor].CGColor;
 //add it to our view
 [self.layerView.layer addSublayer:self.colorLayer];
}

```

```
- (IBAction)changeColor
{
 //create a keyframe animation
 CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
 animation.keyPath = @"backgroundColor";
 animation.duration = 2.0;
 animation.values = @[
 (__bridge id)[UIColor blueColor].CGColor,
 (__bridge id)[UIColor redColor].CGColor,
 (__bridge id)[UIColor greenColor].CGColor,
 (__bridge id)[UIColor blueColor].CGColor];
 //add timing function
 CAMediaTimingFunction *fn = [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionEaseIn];
 animation.timingFunctions = @[fn, fn, fn];
 //apply animation to layer
 [self.colorLayer addAnimation:animation forKey:nil];
}

@end
```



## 自定义缓冲函数

在第八章中，我们给时钟项目添加了动画。看起来很赞，但是如果有合适的缓冲函数就更好了。在显示世界中，钟表指针转动的时候，通常起步很慢，然后迅速啪地一声，最后缓冲到终点。但是标准的缓冲函数在这里每一个适合它，那该如何创建一个新的呢？

除了 `+functionWithName:` 之外，`CAMediaTimingFunction` 同样有另一个构造函数，一个有四个浮点参数的 `+functionWithControlPoints::::`（注意这里奇怪的语法，并没有包含具体每个参数的名称，这在objective-C中是合法的，但是却违反了苹果对方法命名的指导方针，而且看起来是一个奇怪的设计）。

使用这个方法，我们可以创建一个自定义的缓冲函数，来匹配我们的时钟动画，为了理解如何使用这个方法，我们要了解一些 `CAMediaTimingFunction` 是如何工作的。

## 三次贝塞尔曲线

`CAMediaTimingFunction` 函数的主要原则在于它把输入的时间转换成起点和终点之间成比例的改变。我们可以用一个简单的图标来解释，横轴代表时间，纵轴代表改变的量，于是线性的缓冲就是一条从起点开始的简单的斜线（图10.1）。

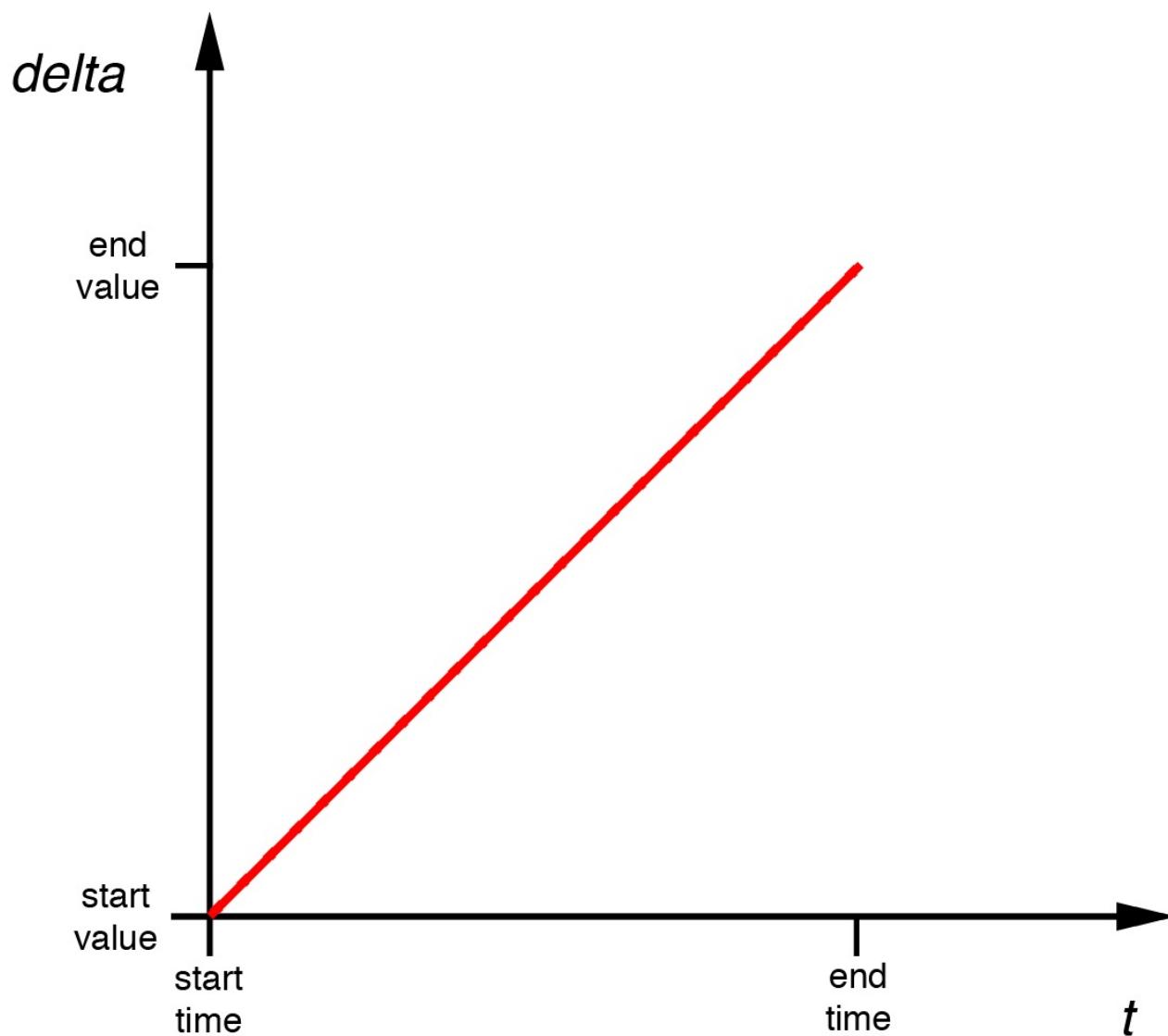


图10.1 线性缓冲函数的图像

这条曲线的斜率代表了速度，斜率的改变代表了加速度，原则上来说，任何加速的曲线都可以用这种图像来表示，但是 `CAMediaTimingFunction` 使用了一个叫做三次贝塞尔曲线的函数，它只可以产出指定缓冲函数的子集（我们之前在第八章中创建 `CAKeyframeAnimation` 路径的时候提到过三次贝塞尔曲线）。

你或许会回想起，一个三次贝塞尔曲线通过四个点来定义，第一个和最后一个点代表了曲线的起点和终点，剩下中间两个点叫做控制点，因为它们控制了曲线的形状，贝塞尔曲线的控制点其实是位于曲线之外的点，也就是说曲线并不一定要穿过它们。你可以把它们想象成吸引经过它们曲线的磁铁。

图10.2展示了三次贝塞尔缓冲函数的例子

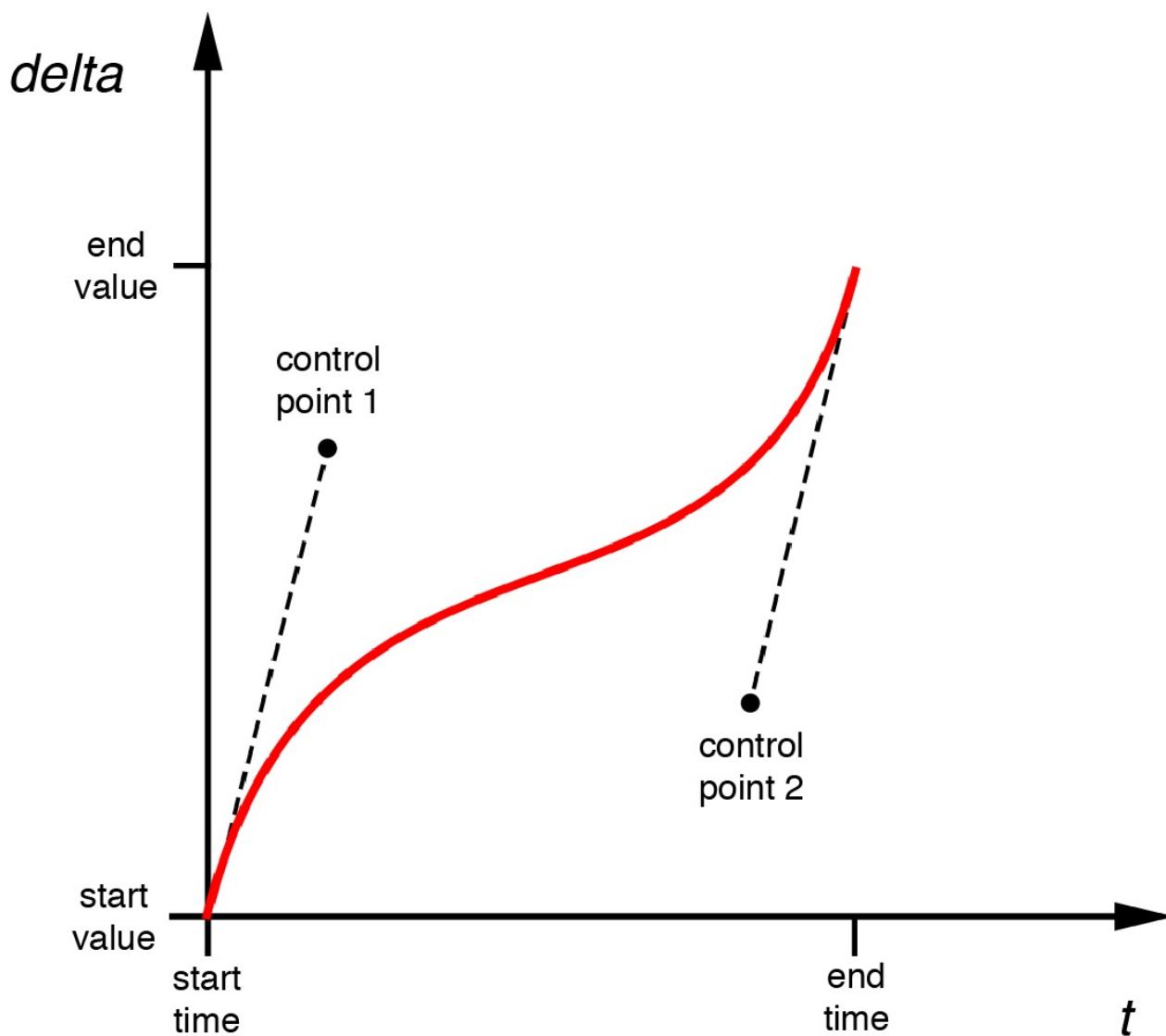


图 10.2 三次贝塞尔缓冲函数

实际上它是一个很奇怪的函数，先加速，然后减速，最后快到达终点的时候又加速，那么标准的缓冲函数又该如何用图像来表示呢？

`CAMediaTimingFunction` 有一个叫做 `-getControlPointAtIndex:values:` 的方法，可以用来检索曲线的点，这个方法的设计的确有点奇怪（或许也就只有苹果能回答为什么不简单返回一个 `CGPoint`），但是使用它我们可以找到标准缓冲函数的点，然后用 `UIBezierPath` 和 `CAShapeLayer` 来把它画出来。

曲线的起始和终点始终是 $\{0, 0\}$ 和 $\{1, 1\}$ ，于是我们只需要检索曲线的第二个和第三个点（控制点）。具体代码见清单 10.4。所有的标准缓冲函数的图像见图 10.3。

清单 10.4 使用 `UIBezierPath` 绘制 `CAMediaTimingFunction`

```
@interface ViewController ()

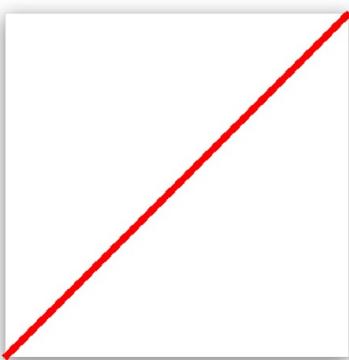
@property (nonatomic, weak) IBOutlet UIView *layerView;

@end

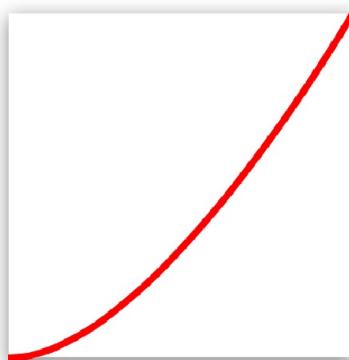
@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //create timing function
 CAMediaTimingFunction *function = [CAMediaTimingFunction functionWithControlPoints:
 //get control points
 CGPointMake controlPoint1, controlPoint2;
 [function getControlPointAtIndex:1 values:(float *)&controlPoint1];
 [function getControlPointAtIndex:2 values:(float *)&controlPoint2];
 //create curve
 UIBezierPath *path = [[UIBezierPath alloc] init];
 [path moveToPoint:CGPointZero];
 [path addCurveToPoint:CGPointMake(1, 1)
 controlPoint1:controlPoint1 controlPoint2:controlPoint2];
 //scale the path up to a reasonable size for display
 [path applyTransform:CGAffineTransformMakeScale(200, 200)];
 //create shape layer
 CAShapeLayer *shapeLayer = [CAShapeLayer layer];
 shapeLayer.strokeColor = [UIColor redColor].CGColor;
 shapeLayer.fillColor = [UIColor clearColor].CGColor;
 shapeLayer.lineWidth = 4.0f;
 shapeLayer.path = path.CGPath;
 [self.layerView.layer addSublayer:shapeLayer];
 //flip geometry so that 0,0 is in the bottom-left
 self.layerView.layer.geometryFlipped = YES;
}

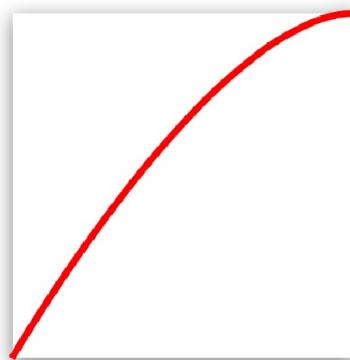
@end
```



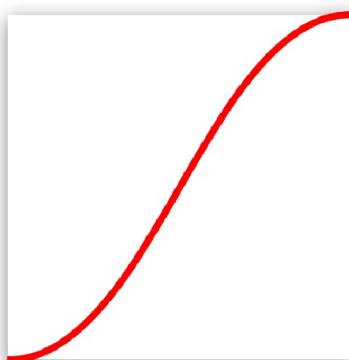
kCAMediaTimingFunctionLinear



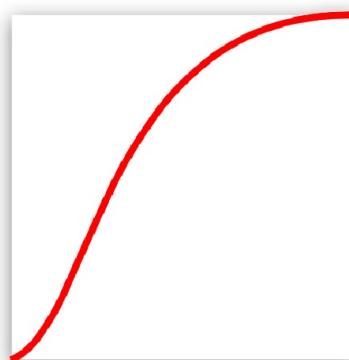
kCAMediaTimingFunctionEaseIn



kCAMediaTimingFunctionEaseOut



kCAMediaTimingFunctionEaseInEaseOut



kCAMediaTimingFunctionDefault

图10.3 标准 `CAMediaTimingFunction` 缓冲曲线

那么对于我们自定义时钟指针的缓冲函数来说，我们需要初始微弱，然后迅速上升，最后缓冲到终点的曲线，通过一些实验之后，最终结果如下：

```
[CAMediaTimingFunction functionWithControlPoints:1 :0 :0.75 :1];
```

如果把它转换成缓冲函数的图像，最后如图10.4所示，如果把它添加到时钟的程序，就形成了之前一直期待的非常赞的效果（见代码清单10.5）。

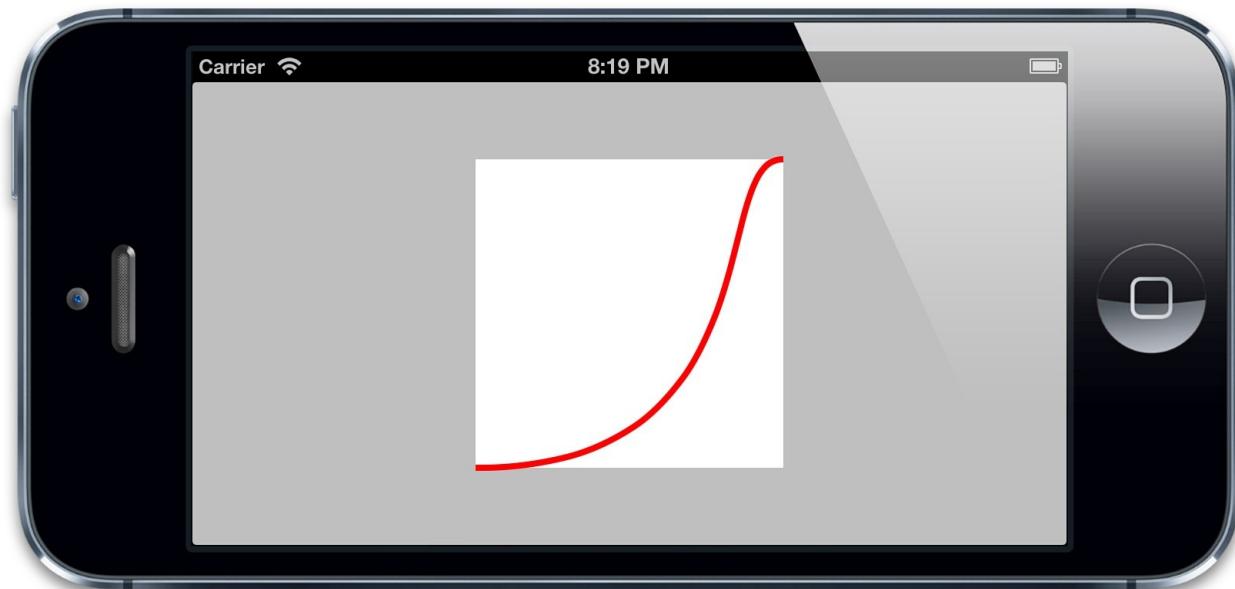


图10.4 自定义适合时钟的缓冲函数

清单10.5 添加了自定义缓冲函数的时钟程序

```
- (void)setAngle:(CGFloat)angle forHand:(UIView *)handView animated:(BOOL)animated {
 //generate transform
 CATransform3D transform = CATransform3DMakeRotation(angle, 0, 0, 1);
 if (animated) {
 //create transform animation
 CABasicAnimation *animation = [CABasicAnimation animation];
 animation.keyPath = @"transform";
 animation.fromValue = [handView.layer.presentationLayer valueForKey:@"transform"];
 animation.toValue = [NSValue valueWithCATransform3D:transform];
 animation.duration = 0.5;
 animation.delegate = self;
 animation.timingFunction = [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionEaseIn];
 //apply animation
 handView.layer.transform = transform;
 [handView.layer addAnimation:animation forKey:nil];
 } else {
 //set transform directly
 handView.layer.transform = transform;
 }
}
```

## 更加复杂的动画曲线

考虑一个橡胶球掉落到坚硬的地面的场景，当开始下落的时候，它会持续加速知道落到地面，然后经过几次反弹，最后停下来。如果用一张图来说明，它会如图10.5所示。

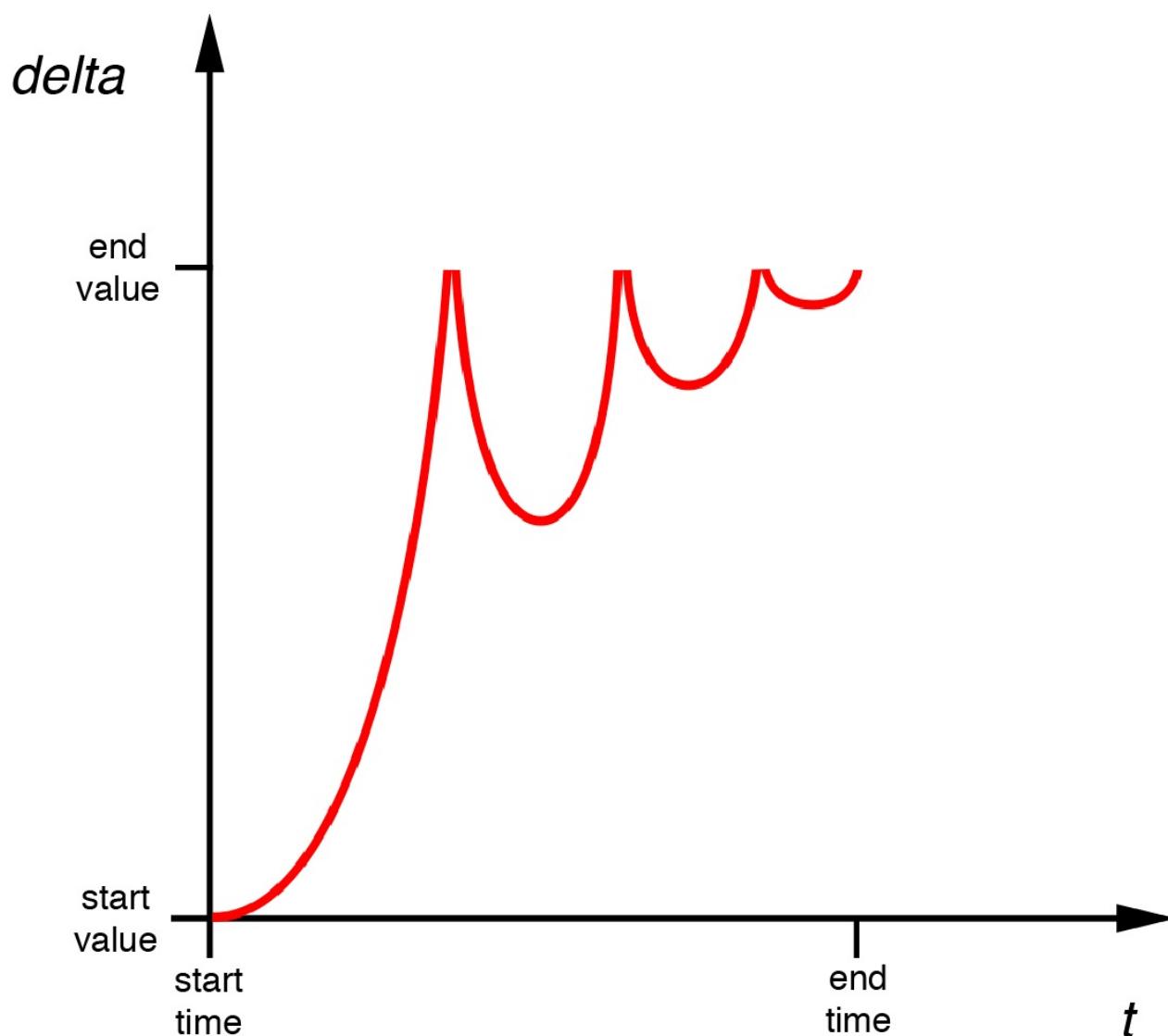


图10.5 一个没法用三次贝塞尔曲线描述的反弹的动画

这种效果没法用一个简单的三次贝塞尔曲线表示，于是不能用 `CAMediaTimingFunction` 来完成。但如果想要实现这样的效果，可以用如下几种方法：

- 用 `CAKeyframeAnimation` 创建一个动画，然后分割成几个步骤，每个小步骤使用自己的计时函数（具体下节介绍）。
- 使用定时器逐帧更新实现动画（见第11章，“基于定时器的动画”）。

## 基于关键帧的缓冲

为了使用关键帧实现反弹动画，我们需要在缓冲曲线中对每一个显著的点创建一个关键帧（在这个情况下，关键点也就是每次反弹的峰值），然后应用缓冲函数把每段曲线连接起来。同时，我们还需要通过 `keyTimes` 来指定每个关键帧的时间偏移，由于每次反弹的时间都会减少，于是关键帧并不会均匀分布。

清单10.6展示了实现反弹球动画的代码（见图10.6）

清单10.6 使用关键帧实现反弹球的动画

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, strong) UIImageView *ballView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //add ball image view
 UIImage *ballImage = [UIImage imageNamed:@"Ball.png"];
 self.ballView = [[UIImageView alloc] initWithImage:ballImage];
 [self.containerView addSubview:self.ballView];
 //animate
 [self animate];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
 //replay animation on tap
 [self animate];
}

- (void)animate
{
 //reset ball to top of screen
}
```

```
self.ballView.center = CGPointMake(150, 32);
//create keyframe animation
CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
animation.keyPath = @"position";
animation.duration = 1.0;
animation.delegate = self;
animation.values = @[
 [NSValue valueWithCGPoint:CGPointMake(150,
]);

animation.timingFunctions = @[
 [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionLinear],
 [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionEaseIn],
 [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionEaseOut],
 [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionEaseInEaseOut],
 [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionSpring],
 [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionDefault],
 [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionCustom],
 [CAMediaTimingFunction functionWithName:kCAMediaTimingFunctionNone];

animation.keyTimes = @[@0.0, @0.3, @0.5, @0.7, @0.8, @0.9, @0.95];
//apply animation
self.ballView.layer.position = CGPointMake(150, 268);
[self.ballView.layer addAnimation:animation forKey:nil];
}

@end
```

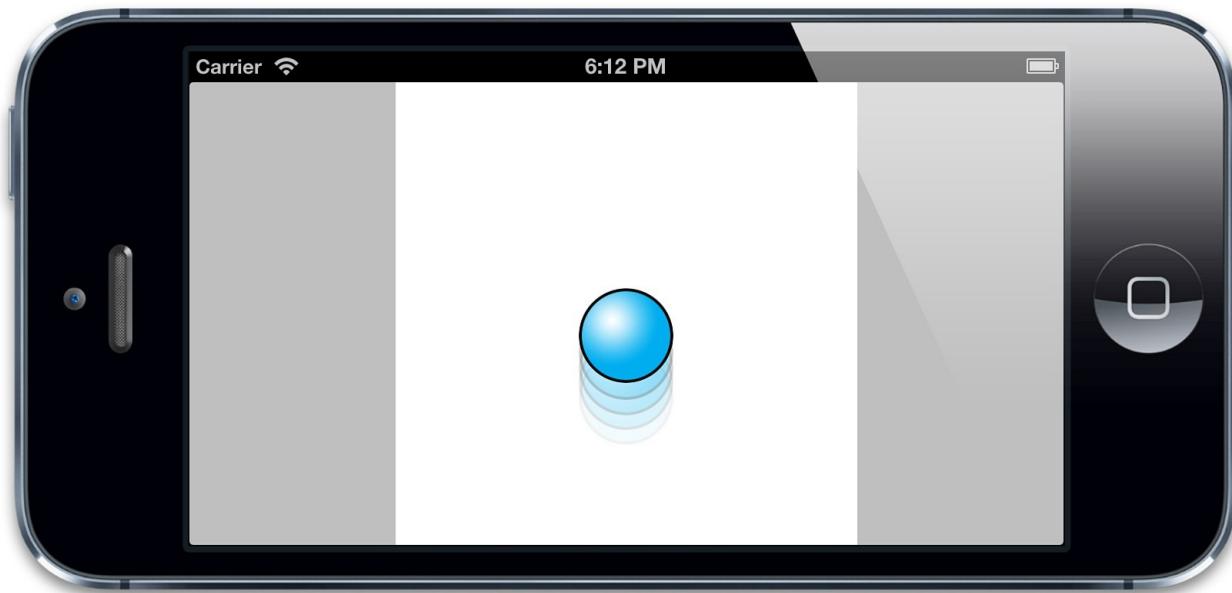


图10.6 使用关键帧实现的反弹球动画

这种方式还算不错，但是实现起来略显笨重（因为要不停地尝试计算各种关键帧和时间偏移）并且和动画强绑定了（因为如果要改变动画的一个属性，那就意味着要重新计算所有的关键帧）。那该如何写一个方法，用缓冲函数来把任何简单的属性动画转换成关键帧动画呢，下面我们来实现它。

## 流程自动化

在清单10.6中，我们把动画分割成相当大的几块，然后用Core Animation的缓冲进入和缓冲退出函数来大约形成我们想要的曲线。但如果我们将动画分割成更小的几部分，那么我们就可以用直线来拼接这些曲线（也就是线性缓冲）。为了实现自动化，我们需要知道如何做如下两件事情：

- 自动把任意属性动画分割成多个关键帧
- 用一个数学函数表示弹性动画，使得可以对帧做便宜

为了解决第一个问题，我们需要复制Core Animation的插值机制。这是一个传入起点和终点，然后在这两个点之间指定时间点产出一个新点的机制。对于简单的浮点起始值，公式如下（假设时间从0到1）：

```
value = (endValue - startValue) * time + startValue;
```

那么如果要插入一个类似于 `CGPoint`，`CGColorRef` 或者 `CATransform3D` 这种更加复杂类型的值，我们可以简单地对每个独立的元素应用这个方法（也就 `CGPoint` 中的x和y值，`CGColorRef` 中的红，蓝，绿，透明值，或者是 `CATransform3D` 中独立矩阵的坐标）。我们同样需要一些逻辑在插值之前对对象拆解值，然后在插值之后在重新封装成对象，也就是说需要实时地检查类型。

一旦我们可以用代码获取属性动画的起始值之间的任意插值，我们就可以把动画分割成许多独立的关键帧，然后产出一个线性的关键帧动画。清单10.7展示了相关代码。

注意到我们用了  $60 \times$  动画时间（秒做单位）作为关键帧的个数，这时因为Core Animation按照每秒60帧去渲染屏幕更新，所以如果我们每秒生成60个关键帧，就可以保证动画足够的平滑（尽管实际上很可能用更少的帧率就可以达到很好的效果）。

我们在示例中仅仅引入了对 `CGPoint` 类型的插值代码。但是，从代码中很清楚能看出如何扩展成支持别的类型。作为不能识别类型的备选方案，我们仅仅在前一半返回了 `fromValue`，在后一半返回了 `toValue`。

#### 清单10.7 使用插入的值创建一个关键帧动画

```
float interpolate(float from, float to, float time)
{
 return (to - from) * time + from;
}

- (id)interpolateFromValue:(id)fromValue toValue:(id)toValue time:(float)time
{
 if ([fromValue isKindOfClass:[NSValue class]]) {
 //get type
 const char *type = [fromValue objCType];
 if (strcmp(type, @encode(CGPoint)) == 0) {
 CGPoint from = [fromValue CGPointValue];
 CGPoint to = [toValue CGPointValue];
 CGPoint result = CGPointMake(interpolate(from.x, to.x,
 return [NSValue valueWithCGPoint:result];
 }
 }
 //provide safe default implementation
 return (time < 0.5)? fromValue: toValue;
}
```

```

}

- (void)animate
{
 //reset ball to top of screen
 self.ballView.center = CGPointMake(150, 32);
 //set up animation parameters
 NSValue *fromValue = [NSValue valueWithCGPoint:CGPointMake(150,
 NSValue *toValue = [NSValue valueWithCGPoint:CGPointMake(150,
 CFTimeInterval duration = 1.0;
 //generate keyframes
 NSInteger numFrames = duration * 60;
 NSMutableArray *frames = [NSMutableArray array];
 for (int i = 0; i < numFrames; i++) {
 float time = 1 / (float)numFrames * i;
 [frames addObject:[self interpolateFromValue:fromValue toValue:toValue atTime:time]];
 }
 //create keyframe animation
 CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
 animation.keyPath = @"position";
 animation.duration = 1.0;
 animation.delegate = self;
 animation.values = frames;
 //apply animation
 [self.ballView.layer addAnimation:animation forKey:nil];
}

```



这可以起到作用，但效果并不是很好，到目前为止我们所完成的只是一个非常复杂的方式来使用线性缓冲复制 CABasicAnimation 的行为。这种方式的好处在于我们可以更加精确地控制缓冲，这也意味着我们可以应用一个完全定制的缓冲函数。那么该如何做呢？

缓冲背后的数学并不很简单，但是幸运的是我们不需要一一实现它。罗伯特·彭纳有一个网页关于缓冲函数 (<http://www.robertpenner.com/easing>)，包含了大多数普遍的缓冲函数的多种编程语言的实现的链接，包括C。这里是一个缓冲进入缓冲退出函数的示例（实际上有很多不同的方式去实现它）。

```
float quadraticEaseInOut(float t)
{
 return (t < 0.5)? (2 * t * t): (-2 * t * t) + (4 * t) - 1;
}
```

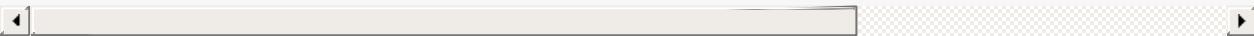
对我们的弹性球来说，我们可以使用 `bounceEaseOut` 函数：

```
float bounceEaseOut(float t)
{
 if (t < 4/11.0) {
 return (121 * t * t)/16.0;
 } else if (t < 8/11.0) {
 return (363/40.0 * t * t) - (99/10.0 * t) + 17/5.0;
 } else if (t < 9/10.0) {
 return (4356/361.0 * t * t) - (35442/1805.0 * t) + 16061/1805.0;
 }
 return (54/5.0 * t * t) - (513/25.0 * t) + 268/25.0;
}
```

如果修改清单10.7的代码来引入 `bounceEaseOut` 方法，我们的任务就是仅仅交换缓冲函数，现在就可以选择任意的缓冲类型创建动画了（见清单10.8）。

#### 清单10.8 用关键帧实现自定义的缓冲函数

```
- (void)animate
{
 //reset ball to top of screen
 self.ballView.center = CGPointMake(150, 32);
 //set up animation parameters
 NSValue *fromValue = [NSValue valueWithCGPoint:CGPointMake(150,
 NSValue *toValue = [NSValue valueWithCGPoint:CGPointMake(150,
 CFTimeInterval duration = 1.0;
 //generate keyframes
 NSInteger numFrames = duration * 60;
 NSMutableArray *frames = [NSMutableArray array];
 for (int i = 0; i < numFrames; i++) {
 float time = 1/(float)numFrames * i;
 //apply easing
 time = bounceEaseOut(time);
 //add keyframe
 [frames addObject:[self interpolateFromValue:fromValue toVa
 }
 //create keyframe animation
 CAKeyframeAnimation *animation = [CAKeyframeAnimation animation];
 animation.keyPath = @"position";
 animation.duration = 1.0;
 animation.delegate = self;
 animation.values = frames;
 //apply animation
 [self.ballView.layer addAnimation:animation forKey:nil];
}
```



## 总结

在这一章中，我们了解了有关缓冲和 `CAMediaTimingFunction` 类，它可以允许我们创建自定义的缓冲函数来完善我们的动画，同样了解了如何用 `CAKeyframeAnimation` 来避开 `CAMediaTimingFunction` 的限制，创建完全自定义的缓冲函数。

在下一章中，我们将要研究基于定时器的动画--另一个给我们对动画更多控制的选择，并且实现对动画的实时操纵。

## 基于定时器的动画

我可以指导你，但是你必须按照我说的做。 -- 骇客帝国

在第10章“缓冲”中，我们研究了 `CAMediaTimingFunction`，它是一个通过控制动画缓冲来模拟物理效果例如加速或者减速来增强现实感的东西，那么如果想更加真实地模拟物理交互或者实时根据用户输入修改动画该怎么办呢？在这一章中，我们将继续探索一种能够允许我们精确地控制一帧一帧展示的基于定时器的动画。

## 定时帧

动画看起来是用来显示一段连续的运动过程，但实际上当在固定位置上展示像素的时候并不能做到这一点。一般来说这种显示都无法做到连续的移动，能做的仅仅是足够快地展示一系列静态图片，只是看起来像是做了运动。

我们之前提到过iOS按照每秒60次刷新屏幕，然后 CAAnimation 计算出需要展示的新的帧，然后在每次屏幕更新的时候同步绘制上去， CAAnimation 最机智的地方在于每次刷新需要展示的时候去计算插值和缓冲。

在第10章中，我们解决了如何自定义缓冲函数，然后根据需要展示的帧的数组来告诉 CAKeyframeAnimation 的实例如何去绘制。所有的Core Animation实际上都是按照一定的序列来显示这些帧，那么我们可以自己做到这些么？

### NSTimer

实际上，我们在第三章“图层几何学”中已经做过类似的东西，就是时钟那个例子，我们用了 NSTimer 来对钟表的指针做定时动画，一秒钟更新一次，但是如果我们将频率调整成一秒钟更新60次的话，原理是完全相同的。

我们来试着用 NSTimer 来修改第十章中弹性球的例子。由于现在我们在定时器启动之后连续计算动画帧，我们需要在类中添加一些额外的属性来存储动画的 fromValue ， toValue ， duration 和当前的 timeOffset （见清单 11.1）。

#### 清单11.1 使用 NSTimer 实现弹性球动画

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, strong) UIImageView *ballView;
@property (nonatomic, strong) NSTimer *timer;
@property (nonatomic, assign) NSTimeInterval duration;
@property (nonatomic, assign) NSTimeInterval timeOffset;
@property (nonatomic, strong) id fromValue;
@property (nonatomic, strong) id toValue;

@end
```

```
@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //add ball image view
 UIImage *ballImage = [UIImage imageNamed:@"Ball.png"];
 self.ballView = [[UIImageView alloc] initWithImage:ballImage];
 [self.containerView addSubview:self.ballView];
 //animate
 [self animate];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
 //replay animation on tap
 [self animate];
}

float interpolate(float from, float to, float time)
{
 return (to - from) * time + from;
}

- (id)interpolateFromValue:(id)fromValue toValue:(id)toValue time:(float)time
{
 if ([fromValue isKindOfClass:[NSValue class]]) {
 //get type
 const char *type = [(NSValue *)fromValue objCType];
 if (strcmp(type, @encode(CGPoint)) == 0) {
 CGPoint from = [fromValue CGPointValue];
 CGPoint to = [toValue CGPointValue];
 CGPoint result = CGPointMake(interpolate(from.x, to.x,
 time), interpolate(from.y, to.y, time));
 return [NSValue valueWithCGPoint:result];
 }
 }
 //provide safe default implementation
 return (time < 0.5)? fromValue: toValue;
}
```

```

float bounceEaseOut(float t)
{
 if (t < 4/11.0) {
 return (121 * t * t)/16.0;
 } else if (t < 8/11.0) {
 return (363/40.0 * t * t) - (99/10.0 * t) + 17/5.0;
 } else if (t < 9/10.0) {
 return (4356/361.0 * t * t) - (35442/1805.0 * t) + 16061/1805.0;
 }
 return (54/5.0 * t * t) - (513/25.0 * t) + 268/25.0;
}

- (void)animate
{
 //reset ball to top of screen
 self.ballView.center = CGPointMake(150, 32);
 //configure the animation
 self.duration = 1.0;
 self.timeOffset = 0.0;
 self.fromValue = [NSValue valueWithCGPoint:CGPointMake(150, 32)];
 self.toValue = [NSValue valueWithCGPoint:CGPointMake(150, 268)];
 //stop the timer if it's already running
 [self.timer invalidate];
 //start the timer
 self.timer = [NSTimer scheduledTimerWithTimeInterval:1/60.0
 target:self
 selector:@selector(step)
 userInfo:nil
 repeats:YES];
}

- (void)step:(NSTimer *)step
{
 //update time offset
 self.timeOffset = MIN(self.timeOffset + 1/60.0, self.duration);
 //get normalized time offset (in range 0 - 1)
 float time = self.timeOffset / self.duration;
 //apply easing
 time = bounceEaseOut(time);
}

```

```

//interpolate position
id position = [self interpolateFromValue:self.fromValue
 toValue:self.toValue
 time:time];
//move ball view to new position
self.ballView.center = [position CGPointValue];
//stop the timer if we've reached the end of the animation
if (self.timeOffset >= self.duration) {
 [self.timer invalidate];
 self.timer = nil;
}
}

@end

```

很赞，而且和基于关键帧例子的代码一样很多，但是如果想一次性在屏幕上对很多东西做动画，很明显就会有很多问题。

`NSTimer` 并不是最佳方案，为了理解这点，我们需要确切地知道 `NSTimer` 是如何工作的。iOS上的每个线程都管理了一个 `NSRunLoop`，字面上看就是通过一个循环来完成一些任务列表。但是对主线程，这些任务包含如下几项：

- 处理触摸事件
- 发送和接受网络数据包
- 执行使用`gcd`的代码
- 处理计时器行为
- 屏幕重绘

当你设置一个 `NSTimer`，他会被插入到当前任务列表中，然后直到指定时间过去之后才会被执行。但是何时启动定时器并没有一个时间上限，而且它只会在列表中上一个任务完成之后开始执行。这通常会导致有几毫秒的延迟，但是如果上一个任务过了很久才完成就会导致延迟很长一段时间。

屏幕重绘的频率是一秒钟六十次，但是和定时器行为一样，如果列表中上一个执行了很长时间，它也会延迟。这些延迟都是一个随机值，于是就不能保证定时器精准地一秒钟执行六十次。有时候发生在屏幕重绘之后，这就会使得更新屏幕会有个延迟，看起来就是动画卡壳了。有时候定时器会在屏幕更新的时候执行两次，于是动画看起来就跳动了。

我们可以通过一些途径来优化：

- 我们可以用 `CADisplayLink` 让更新频率严格控制在每次屏幕刷新之后。
- 基于真实帧的持续时间而不是假设的更新频率来做动画。
- 调整动画计时器的 `run loop` 模式，这样就不会被别的事件干扰。

## CADisplayLink

`CADisplayLink` 是CoreAnimation提供的另一个类似于 `NSTimer` 的类，它总是在屏幕完成一次更新之前启动，它的接口设计的和 `NSTimer` 很类似，所以它实际上就是一个内置实现的替代，但是和 `timeInterval` 以秒为单位不同，`CADisplayLink` 有一个整型的 `frameInterval` 属性，指定了间隔多少帧之后才执行。默认值是1，意味着每次屏幕更新之前都会执行一次。但是如果动画的代码执行起来超过了六十分之一秒，你可以指定 `frameInterval` 为2，就是说动画每隔一帧执行一次（一秒钟30帧）或者3，也就是一秒钟20次，等等。

用 `CADisplayLink` 而不是 `NSTimer`，会保证帧率足够连续，使得动画看起来更加平滑，但即使 `CADisplayLink` 也不能保证每一帧都按计划执行，一些失去控制的离散的任务或者事件（例如资源紧张的后台程序）可能会导致动画偶尔地丢帧。当使用 `NSTimer` 的时候，一旦有机会计时器就会开启，但是 `CADisplayLink` 却不一样：如果它丢失了帧，就会直接忽略它们，然后在下一次更新的时候接着运行。

## 计算帧的持续时间

无论是使用 `NSTimer` 还是 `CADisplayLink`，我们仍然需要处理一帧的时间超出了预期的六十分之一秒。由于我们不能够计算出一帧真实的持续时间，所以需要手动测量。我们可以在每帧开始刷新的时候用 `CACurrentMediaTime()` 记录当前时间，然后和上一帧记录的时间去比较。

通过比较这些时间，我们就可以得到真实的每帧持续的时间，然后代替硬编码的六十分之一秒。我们来更新一下上个例子（见清单11.2）。

清单11.2 通过测量每帧持续的时间来使得动画更加平滑

```
@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *containerView;
```

```

@property (nonatomic, strong) UIImageView *ballView;
@property (nonatomic, strong) CADisplayLink *timer;
@property (nonatomic, assign) CFTimeInterval duration;
@property (nonatomic, assign) CFTimeInterval timeOffset;
@property (nonatomic, assign) CFTimeInterval lastStep;
@property (nonatomic, strong) id fromValue;
@property (nonatomic, strong) id toValue;

@end

@implementation ViewController

...

- (void)animate
{
 //reset ball to top of screen
 self.ballView.center = CGPointMake(150, 32);
 //configure the animation
 self.duration = 1.0;
 self.timeOffset = 0.0;
 self.fromValue = [NSValue valueWithCGPoint:CGPointMake(150, 32)];
 self.toValue = [NSValue valueWithCGPoint:CGPointMake(150, 268)];
 //stop the timer if it's already running
 [self.timer invalidate];
 //start the timer
 self.lastStep =CACurrentMediaTime();
 self.timer = [CADisplayLink displayLinkWithTarget:self
 selector:@selector(step:)];
 [self.timer addToRunLoop:[NSRunLoop mainRunLoop]
 forMode:NSEventRunMode];
}

- (void)step:(CADisplayLink *)timer
{
 //calculate time delta
 CFTimeInterval thisStep =CACurrentMediaTime();
 CFTimeInterval stepDuration = thisStep - self.lastStep;
 self.lastStep = thisStep;
 //update time offset
}

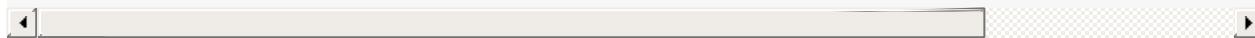
```

```

 self.timeOffset = MIN(self.timeOffset + stepDuration, self.duration);
 //get normalized time offset (in range 0 - 1)
 float time = self.timeOffset / self.duration;
 //apply easing
 time = bounceEaseOut(time);
 //interpolate position
 id position = [self interpolateFromValue:self.fromValue toValue:self.toValue
 time:time];
 //move ball view to new position
 self.ballView.center = [position CGPointValue];
 //stop the timer if we've reached the end of the animation
 if (self.timeOffset >= self.duration) {
 [self.timer invalidate];
 self.timer = nil;
 }
 }

@end

```



## Run Loop 模式

注意到当创建 `CADisplayLink` 的时候，我们需要指定一个 `run loop` 和 `run loop mode`，对于 `run loop` 来说，我们就使用了主线程的 `run loop`，因为任何用户界面的更新都需要在主线程执行，但是模式的选择就并不那么清楚了，每个添加到 `run loop` 的任务都有一个指定了优先级的模式，为了保证用户界面保持平滑，iOS 会提供和用户界面相关任务的优先级，而且当 UI 很活跃的时候的确会暂停一些别的任务。

一个典型的例子就是当是用 `UIScrollView` 滑动的时候，重绘滚动视图的内容会比别的任务优先级更高，所以标准的 `NSTimer` 和网络请求就不会启动，一些常见的 `run loop` 模式如下：

- `NSDefaultRunLoopMode` - 标准优先级
- `NSRunLoopCommonModes` - 高优先级
- `UITrackingRunLoopMode` - 用于 `UIScrollView` 和别的控件的动画

在我们的例子中，我们是用了 `NSDefaultRunLoopMode`，但是不能保证动画平滑的运行，所以就可以用 `NSRunLoopCommonModes` 来替代。但是要小心，因为如果动画在一个高帧率情况下运行，你会发现一些别的类似于定时器的任务或者类似于滑动的其他iOS动画会暂停，直到动画结束。

同样可以同时对 `CADisplayLink` 指定多个run loop模式，于是我们可以同时加入 `NSDefaultRunLoopMode` 和 `UITrackingRunLoopMode` 来保证它不会被滑动打断，也不会被其他UIKit控件动画影响性能，像这样：

```
self.timer = [CADisplayLink displayLinkWithTarget:self selector:@selector(step)]
[self.timer addToRunLoop:[NSRunLoop mainRunLoop] forMode:NSDefaultRunLoopMode]
[self.timer addToRunLoop:[NSRunLoop mainRunLoop] forMode:UITrackingRunLoopMode]
```

和 `CADisplayLink` 类似，`NSTimer` 同样也可以使用不同的run loop模式配置，通过别的函数，而不是 `+scheduledTimerWithTimeInterval:` 构造器

```
self.timer = [NSTimer timerWithTimeInterval:1/60.0
 target:self
 selector:@selector(step:)
 userInfo:nil
 repeats:YES];
[[NSRunLoop mainRunLoop] addTimer:self.timer
 forMode:NSRunLoopCommonModes];
```

## 物理模拟

即使使用了基于定时器的动画来复制第10章中关键帧的行为，但还是会有一些本质上的区别：在关键帧的实现中，我们提前计算了所有帧，但是在新的解决方案中，我们实际上实在按需要在计算。意义在于我们可以根据用户输入实时修改动画的逻辑，或者和别的实时动画系统例如物理引擎进行整合。

## Chipmunk

我们来基于物理学创建一个真实的重力模拟效果来取代当前基于缓冲的弹性动画，但即使模拟2D的物理效果就已近极其复杂了，所以就不要尝试去实现它了，直接用开源的物理引擎库好了。

我们将要使用的物理引擎叫做Chipmunk。另外的2D物理引擎也同样可以（例如Box2D），但是Chipmunk使用纯C写的，而不是C++，好处在于更容易和Objective-C项目整合。Chipmunk有很多版本，包括一个和Objective-C绑定的“indie”版本。C语言的版本是免费的，所以我们就用它好了。在本书写作的时候6.1.4是最新的版本；你可以从<http://chipmunk-physics.net>下载它。

Chipmunk完整的物理引擎相当巨大复杂，但是我们只会使用如下几个类：

- `cpSpace` - 这是所有的物理结构体的容器。它有一个大小和一个可选的重力矢量
- `cpBody` - 它是一个固态无弹力的刚体。它有一个坐标，以及其他物理属性，例如质量，运动和摩擦系数等等。
- `cpShape` - 它是一个抽象的几何形状，用来检测碰撞。可以给结构体添加一个或多个 `cpShape`，而且 `cpShape` 有各种子类来代表不同形状的类型。

在例子中，我们来对一个木箱建模，然后在重力的影响下下落。我们来创建一个 `Crate` 类，包含屏幕上的可视效果（一个 `UIImageView`）和一个物理模型（一个 `cpBody` 和一个 `cpPolyShape`，一个 `cpShape` 的多边形子类来代表矩形木箱）。

用C版本的Chipmunk会带来一些挑战，因为它现在并不支持Objective-C的引用计数模型，所以我们需要准确的创建和释放对象。为了简化，我们把 `cpShape` 和 `cpBody` 的生命周期和 `Crate` 类进行绑定，然后在木箱的 -

`init` 方法中创建，在 `-dealloc` 中释放。木箱物理属性的配置很复杂，所以阅读了 Chipmunk 文档会很有意义。

视图控制器用来管理 `cpSpace`，还有和之前一样的计时器逻辑。在每一步中，我们更新 `cpSpace`（用来进行物理计算和所有结构体的重新摆放）然后迭代对象，然后再更新我们的木箱视图的位置来匹配木箱的模型（在这里，实际上只有一个结构体，但是之后我们将要添加更多）。

Chipmunk 使用了一个和 UIKit 颠倒的坐标系（Y 轴向上为正方向）。为了使得物理模型和视图之间的同步更简单，我们需要通过使用 `geometryFlipped` 属性翻转容器视图的集合坐标（第 3 章中有提到），于是模型和视图都共享一个相同的坐标系。

具体的代码见清单 11.3。注意到我们并没有在任何地方释放 `cpSpace` 对象。在这个例子中，内存空间将会在整个 app 的生命周期中一直存在，所以这没有问题。但是在现实世界的场景中，我们需要像创建木箱结构体和形状一样去管理我们的空间，封装在标准的 Cocoa 对象中，然后来管理 Chipmunk 对象的生命周期。图 11.1 展示了掉落的木箱。

### 清单 11.3 使用物理学来对掉落的木箱建模

```
#import "ViewController.h"
#import
#import "chipmunk.h"

@interface Crate : UIImageView

@property (nonatomic, assign) cpBody *body;
@property (nonatomic, assign) cpShape *shape;

@end

@implementation Crate

#define MASS 100

- (id)initWithFrame:(CGRect)frame
{
 if ((self = [super initWithFrame:frame])) {
 //set image
 self.image = [UIImage imageNamed:@"Crate.png"];
 }
}
```

```

 self.contentMode = UIViewContentModeScaleAspectFill;
 //create the body
 self.body = cpBodyNew(MASS, cpMomentForBox(MASS, frame.size.width * frame.size.height));
 //create the shape
 cpVect corners[] = {
 cpv(0, 0),
 cpv(0, frame.size.height),
 cpv(frame.size.width, frame.size.height),
 cpv(frame.size.width, 0),
 };
 self.shape = cpPolyShapeNew(self.body, 4, corners, cpv(-frame.size.width / 2, -frame.size.height / 2));
 //set shape friction & elasticity
 cpShapeSetFriction(self.shape, 0.5);
 cpShapeSetElasticity(self.shape, 0.8);
 //link the crate to the shape
 //so we can refer to crate from callback later on
 self.shape->data = (__bridge void *)self;
 //set the body position to match view
 cpBodySetPos(self.body, cpv(frame.origin.x + frame.size.width / 2, frame.origin.y + frame.size.height / 2));
 }

 return self;
}

- (void)dealloc
{
 //release shape and body
 cpShapeFree(_shape);
 cpBodyFree(_body);
}

@end

@interface ViewController ()

@property (nonatomic, weak) IBOutlet UIView *containerView;
@property (nonatomic, assign) cpSpace *space;
@property (nonatomic, strong) CADisplayLink *timer;
@property (nonatomic, assign) CFTimeInterval lastStep;

@end

```

```

@implementation ViewController

#define GRAVITY 1000

- (void)viewDidLoad
{
 //invert view coordinate system to match physics
 self.containerView.layer.geometryFlipped = YES;
 //set up physics space
 self.space = cpSpaceNew();
 cpSpaceSetGravity(self.space, cpv(0, -GRAVITY));
 //add a crate
 Crate *crate = [[Crate alloc] initWithFrame:CGRectMake(100, 0,
 [self.containerView addSubview:crate];
 cpSpaceAddBody(self.space, crate.body);
 cpSpaceAddShape(self.space, crate.shape);
 //start the timer
 self.lastStep = CACurrentMediaTime();
 self.timer = [CADisplayLink displayLinkWithTarget:self
 selector:@selector(step)];
 [self.timer addToRunLoop:[NSRunLoop mainRunLoop]
 forMode:NSTimerRunLoopMode];
}

void updateShape(cpShape *shape, void *unused)
{
 //get the crate object associated with the shape
 Crate *crate = (__bridge Crate *)shape->data;
 //update crate view position and angle to match physics shape
 cpBody *body = shape->body;
 crate.center = cpBodyGetPos(body);
 crate.transform = CGAffineTransformMakeRotation(cpBodyGetAngle(body));
}

- (void)step:(CADisplayLink *)timer
{
 //calculate step duration
 CFTimeInterval thisStep = CACurrentMediaTime();
 CFTimeInterval stepDuration = thisStep - self.lastStep;
}

```

```

 self.lastStep = thisStep;
 //update physics
 cpSpaceStep(self.space, stepDuration);
 //update all the shapes
 cpSpaceEachShape(self.space, &updateShape, NULL);
}

@end

```

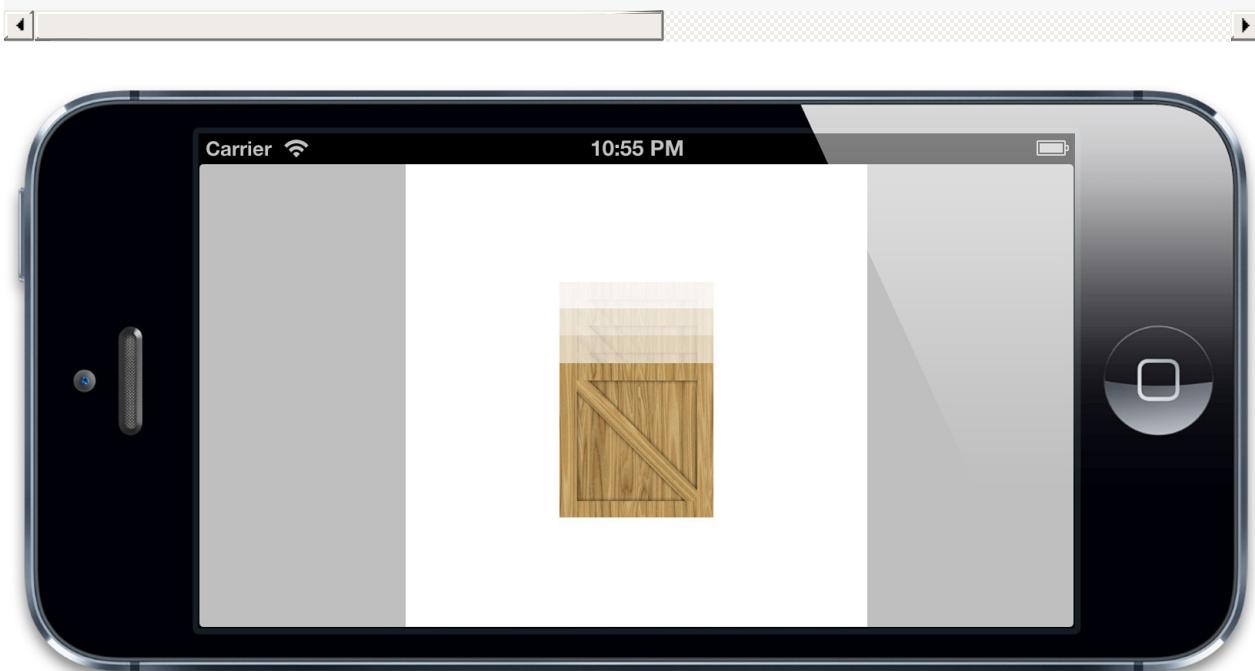


图11.1 一个木箱图片，根据模拟的重力掉落

## 添加用户交互

下一步就是在视图周围添加一道不可见的墙，这样木箱就不会掉落到屏幕之外。或许你会用另一个矩形的 `cpPolyShape` 来实现，就和之前创建木箱那样，但是我们需要检测的是木箱何时离开视图，而不是何时碰撞，所以我们需要一个空心而不是固体矩形。

我们可以通过给 `cpSpace` 添加四个 `cpSegmentShape` 对象（`cpSegmentShape` 代表一条直线，所以四个拼起来就是一个矩形）。然后赋给空间的 `staticBody` 属性（一个不被重力影响的结构体）而不是像木箱那样一个新的 `cpBody` 实例，因为我们不想让这个边框矩形滑出屏幕或者被一个下落的木箱击中而消失。

同样可以再添加一些木箱来做一些交互。最后再添加一个加速器，这样可以通过倾斜手机来调整重力矢量（为了测试需要在一台真实的设备上运行程序，因为模拟器不支持加速器事件，即使旋转屏幕）。清单11.4展示了更新后的代码，运行结果见图11.2。

由于示例只支持横屏模式，所以交换加速计矢量的x和y值。如果在竖屏下运行程序，请把他们换回来，不然重力方向就错乱了。试一下就知道了，木箱会沿着横向移动。

#### 清单11.4 使用围墙和多个木箱的更新后的代码

```

- (void)addCrateWithFrame:(CGRect)frame
{
 Crate *crate = [[Crate alloc] initWithFrame:frame];
 [self.containerView addSubview:crate];
 cpSpaceAddBody(self.space, crate.body);
 cpSpaceAddShape(self.space, crate.shape);
}

- (void)addWallShapeWithStart:(cpVect)start end:(cpVect)end
{
 cpShape *wall = cpSegmentShapeNew(self.space->staticBody, start);
 cpShapeSetCollisionType(wall, 2);
 cpShapeSetFriction(wall, 0.5);
 cpShapeSetElasticity(wall, 0.8);
 cpSpaceAddStaticShape(self.space, wall);
}

- (void)viewDidLoad
{
 //invert view coordinate system to match physics
 self.containerView.layer.geometryFlipped = YES;
 //set up physics space
 self.space = cpSpaceNew();
 cpSpaceSetGravity(self.space, cpv(0, -GRAVITY));
 //add wall around edge of view
 [self addWallShapeWithStart:cpv(0, 0) end:cpv(300, 0)];
 [self addWallShapeWithStart:cpv(300, 0) end:cpv(300, 300)];
 [self addWallShapeWithStart:cpv(300, 300) end:cpv(0, 300)];
 [self addWallShapeWithStart:cpv(0, 300) end:cpv(0, 0)];
}

```

```
//add a crates
[self addCrateWithFrame:CGRectMake(0, 0, 32, 32)];
[self addCrateWithFrame:CGRectMake(32, 0, 32, 32)];
[self addCrateWithFrame:CGRectMake(64, 0, 64, 64)];
[self addCrateWithFrame:CGRectMake(128, 0, 32, 32)];
[self addCrateWithFrame:CGRectMake(0, 32, 64, 64)];
//start the timer
self.lastStep = CACurrentMediaTime();
self.timer = [CADisplayLink displayLinkWithTarget:self
 selector:@selector(step:)];
[self.timer addToRunLoop:[NSRunLoop mainRunLoop]
 forMode:NSDefaultRunLoopMode];
//update gravity using accelerometer
[UIAccelerometer sharedAccelerometer].delegate = self;
[UIAccelerometer sharedAccelerometer].updateInterval = 1/60.0;
}

- (void)accelerometer:(UIAccelerometer *)accelerometer didAccelerate:(UIAcceleration *)acceleration
{
 //update gravity
 cpSpaceSetGravity(self.space, cpv(acceleration.y * GRAVITY, -acceleration.x * GRAVITY));
}
```

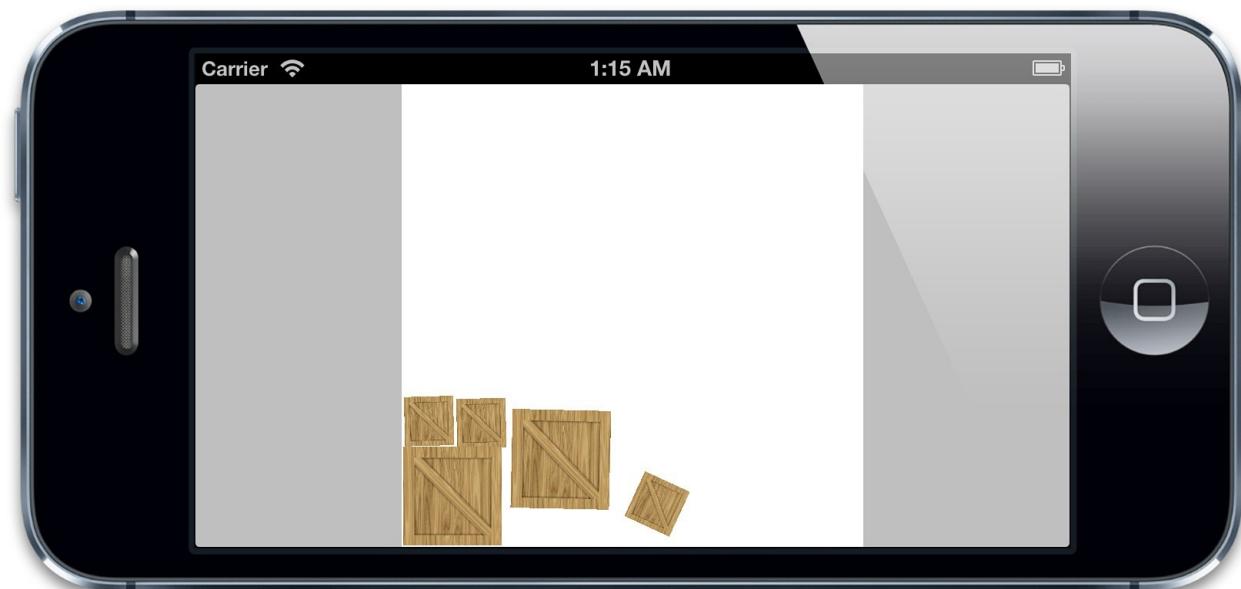
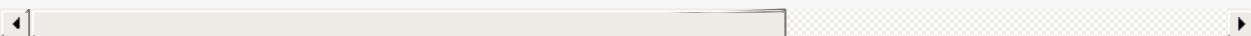


图11.1 真实引力场下的木箱交互

## 模拟时间以及固定的时间步长

对于实现动画的缓冲效果来说，计算每帧持续的时间是一个很好的解决方案，但是对模拟物理效果并不理想。通过一个可变的时间步长来实现有着两个弊端：

- 如果时间步长不是固定的，精确的值，物理效果的模拟也就随之不确定。这意味着即使是传入相同的输入值，也可能在不同场合下有着不同的效果。有时候没多大影响，但是在基于物理引擎的游戏下，玩家就会由于相同的操作行为导致不同的结果而感到困惑。同样也会让测试变得麻烦。
- 由于性能故障造成的丢帧或者像电话呼入的中断都可能会造成不正确的结果。考虑一个像子弹那样快速移动物体，每一帧的更新都需要移动子弹，检测碰撞。如果两帧之间的时间加长了，子弹就会在这一步移动更远的距离，穿过围墙或者是别的障碍，这样就丢失了碰撞。

我们想得到的理想的效果就是通过固定的时间步长来计算物理效果，但是在屏幕发生重绘的时候仍然能够同步更新视图（可能会由于在我们控制范围之外造成不可预知的效果）。

幸运的是，由于我们的模型（在这个例子中就是Chipmunk的 `cpSpace` 中的 `cpBody`）被视图（就是屏幕上代表木箱的 `UIView` 对象）分离，于是就很简单了。我们只需要根据屏幕刷新的时间跟踪时间步长，然后根据每帧去计算一个或者多个模拟出来的效果。

我们可以通过一个简单的循环来实现。通过每次 `CADisplayLink` 的启动来通知屏幕将要刷新，然后记录下当前的 `CACurrentMediaTime()`。我们需要在一个小增量中提前重复物理模拟（这里用120分之一秒）直到赶上显示的时间。然后更新我们的视图，在屏幕刷新的时候匹配当前物理结构体的显示位置。

清单11.5展示了固定时间步长版本的代码

清单11.5 固定时间步长的木箱模拟

```
#define SIMULATION_STEP (1/120.0)

- (void)step:(CADisplayLink *)timer
{
 //calculate frame step duration
 CFTimeInterval frameTime = CACurrentMediaTime();
 //update simulation
 while (self.lastStep < frameTime) {
 cpSpaceStep(self.space, SIMULATION_STEP);
 self.lastStep += SIMULATION_STEP;
 }

 //update all the shapes
 cpSpaceEachShape(self.space, &updateShape, NULL);
}
```

## 避免死亡螺旋

当使用固定的模拟时间步长时候，有一件事情一定要注意，就是用来计算物理效果的现实世界的时间并不会加速模拟时间步长。在我们的例子中，我们随意选择了120分之一秒来模拟物理效果。Chipmunk很快，我们的例子也很简单，所以 `cpSpaceStep()` 会完成的很好，不会延迟帧的更新。

但是如果场景很复杂，比如有上百个物体之间的交互，物理计算就会很复杂，`cpSpaceStep()` 的计算也可能会超出1/120秒。我们没有测量出物理步长的时间，因为我们假设了相对于帧刷新来说并不重要，但是如果模拟步长更久的话，就会延迟帧率。

如果帧刷新的时间延迟的话会变得很糟糕，我们的模拟需要执行更多的次数来同步真实的时间。这些额外的步骤就会继续延迟帧的更新，等等。这就是所谓的死亡螺旋，因为最后的结果就是帧率变得越来越慢，直到最后应用程序卡死了。

我们可以通过添加一些代码在设备上来对物理步骤计算真实世界的时间，然后自动调整固定时间步长，但是实际上它不可行。其实只要保证你给容错留下足够的边长，然后在期望支持的最慢的设备上进行测试就可以了。如果物理计算超过了模拟时间的50%，就需要考虑增加模拟时间步长（或者简化场景）。如果模拟时间步长

增加到超过1/60秒（一个完整的屏幕更新时间），你就需要减少动画帧率到一秒30帧或者增加 `CADisplayLink` 的 `frameInterval` 来保证不会随机丢帧，不然你的动画将会看起来不平滑。

## 物理模拟

## 性能调优

代码应该运行的尽量快，而不是更快 - 理查德

在第一和第二部分，我们了解了Core Animation提供的关于绘制和动画的一些特性。Core Animation功能和性能都非常强大，但如果你对背后的原理不清楚的话也会降低效率。让它达到最优的状态是一门艺术。在这章中，我们将探究一些动画运行慢的原因，以及如何去修复这些问题。

## CPU VS GPU

关于绘图和动画有两种处理的方式：CPU（中央处理器）和GPU（图形处理器）。在现代iOS设备中，都有可以运行不同软件的可编程芯片，但是由于历史原因，我们可以说CPU所做的工作都在软件层面，而GPU在硬件层面。

总的来说，我们可以用软件（使用CPU）做任何事情，但是对于图像处理，通常用硬件会更快，因为GPU使用图像对高度并行浮点运算做了优化。由于某些原因，我们想尽可能把屏幕渲染的工作交给硬件去处理。问题在于GPU并没有无限制处理性能，而且一旦资源用完的话，性能就会开始下降了（即使CPU并没有完全占用）

大多数动画性能优化都是关于智能利用GPU和CPU，使得它们都不会超出负荷。于是我们首先需要知道Core Animation是如何在这两个处理器之间分配工作的。

## 动画的舞台

Core Animation处在iOS的核心地位：应用内和应用间都会用到它。一个简单的动画可能同步显示多个app的内容，例如当在iPad上多个程序之间使用手势切换，会使得多个程序同时显示在屏幕上。在一个特定的应用中用代码实现它是没有意义的，因为在iOS中不可能实现这种效果（App都是被沙箱管理，不能访问别的视图）。

动画和屏幕上组合的图层实际上被一个单独的进程管理，而不是你的应用程序。这个进程就是所谓的渲染服务。在iOS5和之前的版本是*SpringBoard*进程（同时管理着iOS的主屏）。在iOS6之后的版本中叫做*BackBoard*。

当运行一段动画时候，这个过程会被四个分离的阶段被打破：

- 布局 - 这是准备你的视图/图层的层级关系，以及设置图层属性（位置，背景色，边框等等）的阶段。
- 显示 - 这是图层的寄宿图片被绘制的阶段。绘制有可能涉及你的`-drawRect:` 和 `-drawLayer:inContext:` 方法的调用路径。
- 准备 - 这是Core Animation准备发送动画数据到渲染服务的阶段。这同时也是Core Animation将要执行一些别的事务例如解码动画过程中将要显示的图片的时间点。

- 提交 - 这是最后的阶段，Core Animation打包所有图层和动画属性，然后通过IPC（内部处理通信）发送到渲染服务进行显示。

但是这些仅仅阶段仅仅发生在你的应用程序之内，在动画在屏幕上显示之前仍然有更多的工作。一旦打包的图层和动画到达渲染服务进程，他们会被反序列化来形成另一个叫做渲染树的图层树（在第一章“图层树”中提到过）。使用这个树状结构，渲染服务对动画的每一帧做出如下工作：

- 对所有的图层属性计算中间值，设置OpenGL几何形状（纹理化的三角形）来执行渲染
- 在屏幕上渲染可见的三角形

所以一共有六个阶段；最后两个阶段在动画过程中不停地重复。前五个阶段都在软件层面处理（通过CPU），只有最后一个被GPU执行。而且，你真正只能控制前两个阶段：布局和显示。Core Animation框架在内部处理剩下的事务，你也控制不了它。

这并不是个问题，因为在布局和显示阶段，你可以决定哪些由CPU执行，哪些交给GPU去做。那么改如何判断呢？

## GPU相关的操作

GPU为一个具体的任务做了优化：它用来采集图片和形状（三角形），运行变换，应用纹理和混合然后把它们输送到屏幕上。现代iOS设备上可编程的GPU在这些操作的执行上又很大的灵活性，但是Core Animation并没有暴露出直接的接口。除非你想绕开Core Animation并编写你自己的OpenGL着色器，从根本上解决硬件加速的问题，那么剩下的所有都还是需要在CPU的软件层面上完成。

宽泛的说，大多数 CALayer 的属性都是用GPU来绘制。比如如果你设置图层背景或者边框的颜色，那么这些可以通过着色的三角板实时绘制出来。如果对一个 contents 属性设置一张图片，然后裁剪它 - 它就会被纹理的三角形绘制出来，而不需要软件层面做任何绘制。

但是有一些事情会降低（基于GPU）图层绘制，比如：

- 太多的几何结构 - 这发生在需要太多的三角板来做变换，以应对处理器的栅格化的时候。现代iOS设备的图形芯片可以处理几百万个三角板，所以在Core Animation中几何结构并不是GPU的瓶颈所在。但由于图层在显示之前通过IPC

发送到渲染服务器的时候（图层实际上是由很多小物体组成的特别重量级的对象），太多的图层就会引起**CPU**的瓶颈。这就限制了一次展示的图层个数（见本章后续“**CPU相关操作**”）。

- 重绘 - 主要由重叠的半透明图层引起。**GPU**的填充比率（用颜色填充像素的比率）是有限的，所以需要避免重绘（每一帧用相同的像素填充多次）的发生。在现代iOS设备上，**GPU**都会应对重绘；即使是iPhone 3GS都可以处理高达2.5的重绘比率，并任然保持60帧率的渲染（这意味着你可以绘制一个半的整屏的冗余信息，而不影响性能），并且新设备可以处理更多。
- 离屏绘制 - 这发生在当不能直接在屏幕上绘制，并且必须绘制到离屏图片的上下文中的时候。离屏绘制发生在基于**CPU**或者是**GPU**的渲染，或者是为离屏图片分配额外内存，以及切换绘制上下文，这些都会降低**GPU**性能。对于特定图层效果的使用，比如圆角，图层遮罩，阴影或者是图层光栅化都会强制Core Animation提前渲染图层的离屏绘制。但这不意味着你需要避免使用这些效果，只是要明白这会带来性能的负面影响。
- 过大的图片 - 如果视图绘制超出**GPU**支持的2048x2048或者4096x4096尺寸的纹理，就必须用**CPU**在图层每次显示之前对图片预处理，同样也会降低性能。

## CPU相关的操作

大多数工作在Core Animation的**CPU**都发生在动画开始之前。这意味着它不会影响到帧率，所以很好，但是他会延迟动画开始的时间，让你的界面看起来会比较迟钝。

以下**CPU**的操作都会延迟动画的开始时间：

- 布局计算 - 如果你的视图层级过于复杂，当视图呈现或者修改的时候，计算图层帧率就会消耗一部分时间。特别是使用iOS6的自动布局机制尤为明显，它应该是比老版的自动调整逻辑加强了**CPU**的工作。
- 视图懒加载 - iOS只会当视图控制器的视图显示到屏幕上时才会加载它。这对内存使用和程序启动时间很有好处，但是当呈现到屏幕上之前，按下按钮导致的许多工作都会不能被及时响应。比如控制器从数据库中获取数据，或者视图从一个nib文件中加载，或者涉及IO的图片显示（见后续“**IO相关操作**”），都会比**CPU**正常操作慢得多。

- Core Graphics绘制 - 如果对视图实现了 `-drawRect:` 方法，或者 `CALayerDelegate` 的 `-drawLayer:inContext:` 方法，那么在绘制任何东西之前都会产生一个巨大的性能开销。为了支持对图层内容的任意绘制，Core Animation必须创建一个内存中等大小的寄宿图片。然后一旦绘制结束之后，必须把图片数据通过IPC传到渲染服务器。在此基础上，Core Graphics绘制就会变得十分缓慢，所以在一个对性能十分挑剔的场景下这样做十分不好。
- 解压图片 - PNG或者JPEG压缩之后的图片文件会比同质量的位图小得多。但是在图片绘制到屏幕上之前，必须把它扩展成完整的未解压的尺寸（通常等同于图片宽  $\times$  高  $\times$  4个字节）。为了节省内存，iOS通常直到真正绘制的时候才去解码图片（14章“图片IO”会更详细讨论）。根据你加载图片的方式，第一次对图层内容赋值的时候（直接或者间接使用 `UIImageView`）或者把它绘制到 Core Graphics中，都需要对它解压，这样的话，对于一个较大的图片，都会占用一定的时间。

当图层被成功打包，发送到渲染服务器之后，CPU仍然要做如下工作：为了显示屏幕上的图层，Core Animation必须对渲染树种的每个可见图层通过OpenGL循环转换成纹理三角板。由于GPU并不知晓Core Animation图层的任何结构，所以必须要由CPU做这些事情。这里CPU涉及的工作和图层个数成正比，所以如果在你的层级关系中有太多的图层，就会导致CPU没一帧的渲染，即使这些事情不是你的应用程序可控的。

## IO相关操作

还有一项没涉及的就是IO相关工作。上下文中的IO（输入/输出）指的是例如闪存或者网络接口的硬件访问。一些动画可能需要从山村（甚至是远程URL）来加载。一个典型的例子就是两个视图控制器之间的过渡效果，这就需要从一个nib文件或者是它的内容中懒加载，或者一个旋转的图片，可能在内存中尺寸太大，需要动态滚动来加载。

IO比内存访问更慢，所以如果动画涉及到IO，就是一个大问题。总的来说，这就需要使用灵敏但尴尬的技术，也就是多线程，缓存和投机加载（提前加载当前不需要的资源，但是之后可能需要用到）。这些技术将会在第14章中讨论。

## 测量，而不是猜测

于是现在你知道有哪些点可能会影响动画性能，那该如何修复呢？好吧，其实不需要。有很多种诡计来优化动画，但如果盲目使用的话，可能会造成更多性能上的问题，而不是修复。

如何正确的测量而不是猜测这点很重要。根据性能相关的知识写出代码不同于仓促的优化。前者很好，后者实际上就是在浪费时间。

那该如何测量呢？第一步就是确保在真实环境下测试你的程序。

## 真机测试，而不是模拟器

当你开始做一些性能方面的工作时，一定要在真机上测试，而不是模拟器。模拟器虽然是加快开发效率的一把利器，但它不能提供准确的真机性能参数。

模拟器运行在你的Mac上，然而Mac上的CPU往往比iOS设备要快。相反，Mac上的GPU和iOS设备的完全不一样，模拟器不得已要在软件层面（CPU）模拟设备的GPU，这意味着GPU相关的操作在模拟器上运行的更慢，尤其是使用 `CAEAGLLayer` 来写一些OpenGL的代码时候。

这就是说在模拟器上的测试出的性能会高度失真。如果动画在模拟器上运行流畅，可能在真机上十分糟糕。如果在模拟器上运行的很卡，也可能在真机上很平滑。你无法确定。

另一件重要的事情就是性能测试一定要用发布配置，而不是调试模式。因为当用发布环境打包的时候，编译器会引入一系列提高性能的优化，例如去掉调试符号或者移除并重新组织代码。你也可以自己做到这些，例如在发布环境禁用`NSLog`语句。你只关心发布性能，那才是你需要测试的点。

最后，最好在你支持的设备中性能最差的设备上测试：如果基于iOS6开发，这意味着最好在iPhone 3GS或者iPad2上测试。如果可能的话，测试不同的设备和iOS版本，因为苹果在不同的iOS版本和设备中做了一些改变，这也可能影响到一些性能。例如iPad3明显要在动画渲染上比iPad2慢很多，因为渲染4倍多的像素点（为了支持视网膜显示）。

## 保持一致的帧率

为了做到动画的平滑，你需要以60FPS（帧每秒）的速度运行，以同步屏幕刷新速率。通过基于 `NSTimer` 或者 `CADisplayLink` 的动画你可以降低到30FPS，而且效果还不错，但是没办法通过Core Animation做到这点。如果不保持60FPS的速率，就可能随机丢帧，影响到体验。

你可以在使用的过程中明显感到有没有丢帧，但没办法通过肉眼来得到具体的数据，也没法知道你的做法有没有真的提高性能。你需要的是一系列精确的数据。

你可以在程序中用 `CADisplayLink` 来测量帧率（就像11章“基于定时器的动画”中那样），然后在屏幕上显示出来，但应用内的FPS显示并不能够完全真实测量出Core Animation性能，因为它仅仅测出应用内的帧率。我们知道很多动画都在应用之外发生（在渲染服务器进程中处理），但同时应用内FPS计数的确可以对某些性能问题提供参考，一旦找出一个问题的地方，你就需要得到更多精确详细的数据来定位到问题所在。苹果提供了一个强大的*Instruments*工具集来帮我们做到这些。

# Instruments

Instruments是Xcode套件中没有被充分利用的一个工具。很多iOS开发者从没用过Instruments，或者只是用Leaks工具检测循环引用。实际上有很多Instruments工具，包括为动画性能调优的东西。

你可以通过在菜单中选择Profile选项来打开Instruments（在这之前，记住要把目标设置成iOS设备，而不是模拟器）。然后将会显示出图12.1（如果没有看到所有选项，你可能设置成了模拟器选项）。



图12.1 Instruments工具选项窗口

就像之前提到的那样，你应该始终将程序设置成发布选项。幸运的是，配置文件默认就是发布选项，所以你不需要在分析的时候调整编译策略。

我们将讨论如下几个工具：

- 时间分析器 - 用来测量被方法/函数打断的CPU使用情况。
- **Core Animation** - 用来调试各种Core Animation性能问题。
- **OpenGL ES驱动** - 用来调试GPU性能问题。这个工具在编写Open GL代码的时候很有用，但有时也用来处理Core Animation的工作。

Instruments的一个很棒的功能在于它可以创建我们自定义的工具集。除了你初始选择的工具之外，如果在Instruments中打开Library窗口，你可以拖拽别的工具到左侧边栏。我们将创建以上我们提到的三个工具，然后就可以并行使用了（见图12.2）。

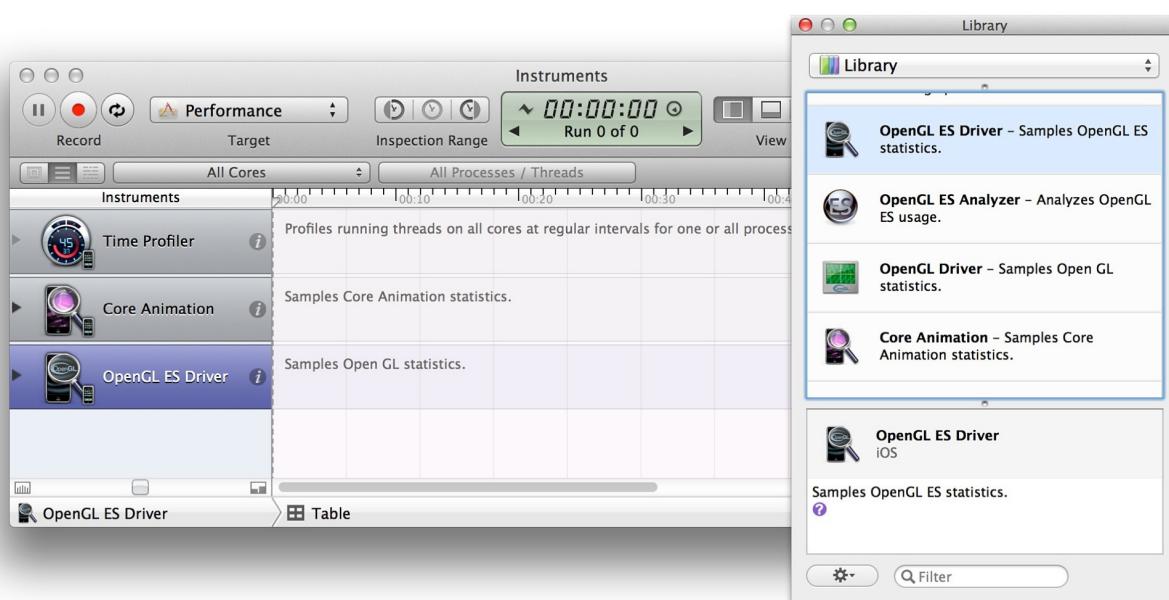


图12.2 添加额外的工具到Instruments侧边栏

## 时间分析器

时间分析器工具用来检测CPU的使用情况。它可以告诉我们程序中的哪个方法正在消耗大量的CPU时间。使用大量的CPU并不一定是个问题 - 你可能期望动画路径对CPU非常依赖，因为动画往往是iOS设备中最苛刻的任务。

但是如果你有性能问题，查看CPU时间对于判断性能是不是和CPU相关，以及定位到函数都很有帮助（见图12.3）。

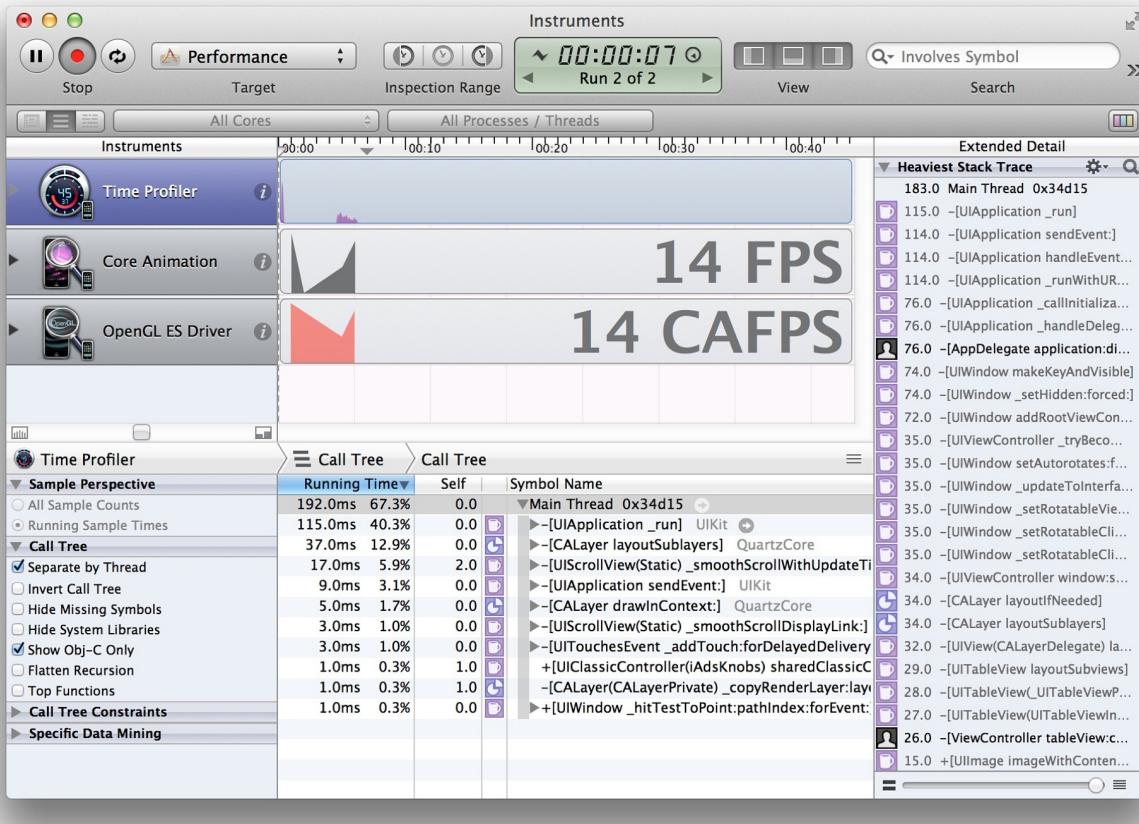


图12.3 时间分析器工具

时间分析器有一些选项来帮助我们定位到我们关心的方法。可以使用左侧的复选框来打开。其中最有用的是如下几点：

- 通过线程分离 - 这可以通过执行的线程进行分组。如果代码被多线程分离的话，那么就可以判断到底是哪个线程造成了问题。
- 隐藏系统库 - 可以隐藏所有苹果的框架代码，来帮助我们寻找哪一段代码造成了性能瓶颈。由于我们不能优化框架方法，所以这对定位到我们能实际修复的代码很有用。
- 只显示Obj-C代码 - 隐藏除了Objective-C之外的所有代码。大多数内部的Core Animation代码都是用C或者C++函数，所以这对我们集中精力到我们代码中显式调用的方法就很有用。

## Core Animation

Core Animation工具用来监测Core Animation性能。它给我们提供了周期性的FPS，并且考虑到了发生在程序之外的动画（见图12.4）。

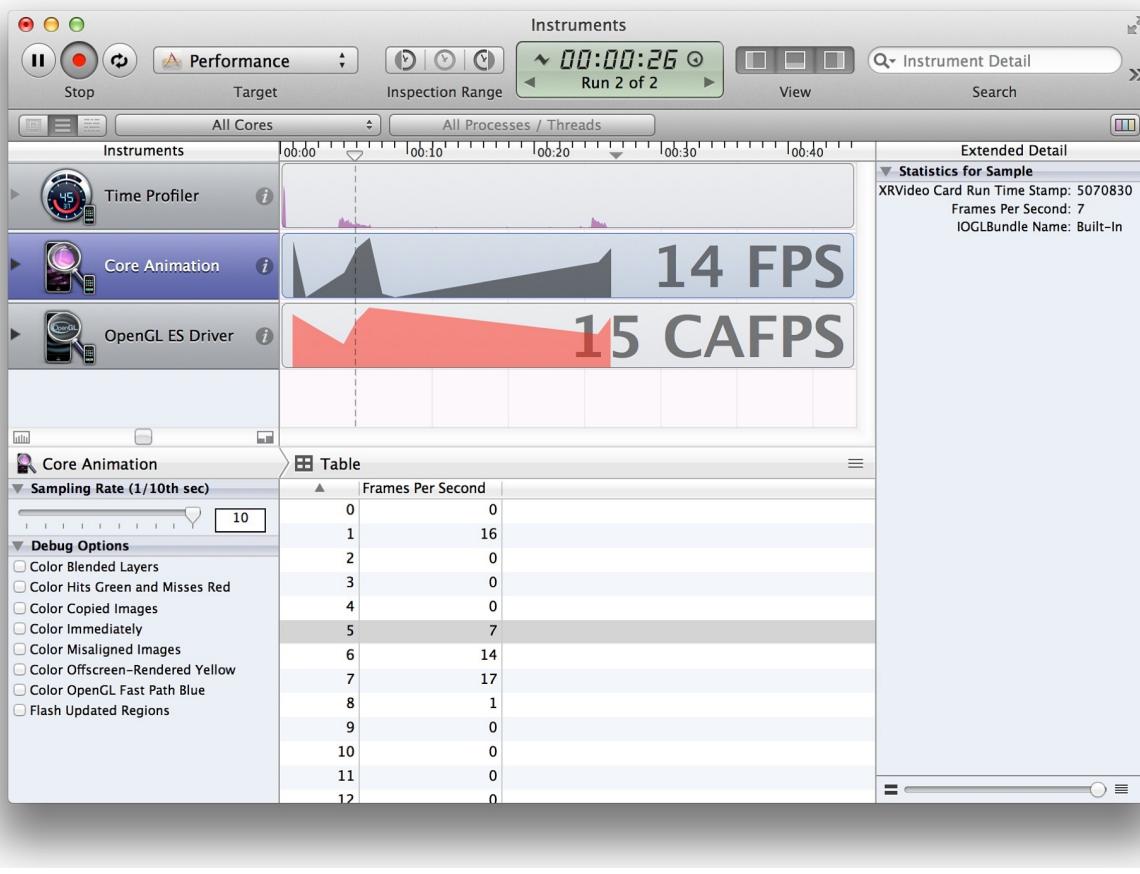


图12.4 使用可视化调试选项的Core Animation工具

Core Animation工具也提供了一系列复选框选项来帮助调试渲染瓶颈：

- **Color Blended Layers** - 这个选项基于渲染程度对屏幕中的混合区域进行绿到红的高亮（也就是多个半透明图层的叠加）。由于重绘的原因，混合对GPU性能会有影响，同时也是滑动或者动画帧率下降的罪魁祸首之一。
- **Color Hits Green and Misses Red** - 当使用 `shouldRasterize` 属性的时候，耗时的图层绘制会被缓存，然后当做一个简单的扁平图片呈现。当缓存再生的时候这个选项就用红色对栅格化图层进行了高亮。如果缓存频繁再生的话，就意味着栅格化可能会有负面的性能影响了（更多关于使用 `shouldRasterize` 的细节见第15章“图层性能”）。
- **Color Copied Images** - 有时候寄宿图片的生成意味着Core Animation被强制生成一些图片，然后发送到渲染服务器，而不是简单的指向原始指针。这个选项把这些图片渲染成蓝色。复制图片对内存和CPU使用来说都是一项非常昂贵的操作，所以应该尽可能的避免。

- **Color Immediately** - 通常Core Animation Instruments以每毫秒10次的频率更新图层调试颜色。对某些效果来说，这显然太慢了。这个选项就可以用来设置每帧都更新（可能会影响到渲染性能，而且会导致帧率测量不准，所以不要一直都设置它）。
- **Color Misaligned Images** - 这里会高亮那些被缩放或者拉伸以及没有正确对齐到像素边界的图片（也就是非整型坐标）。这些中的大多数通常都会导致图片的不正常缩放，如果把一张大图当缩略图显示，或者不正确地模糊图像，那么这个选项将会帮你识别出问题所在。
- **Color Offscreen-Rendered Yellow** - 这里会把那些需要离屏渲染的图层高亮成黄色。这些图层很可能需要用 `shadowPath` 或者 `shouldRasterize` 来优化。
- **Color OpenGL Fast Path Blue** - 这个选项会对任何直接使用OpenGL绘制的图层进行高亮。如果仅仅使用UIKit或者Core Animation的API，那么不会有任何效果。如果使用 `GLKView` 或者 `CAEAGLLayer`，那如果不显示蓝色块的话就意味着你正在强制CPU渲染额外的纹理，而不是绘制到屏幕。
- **Flash Updated Regions** - 这个选项会对重绘的内容高亮成黄色（也就是任何在软件层面使用Core Graphics绘制的图层）。这种绘图的速度很慢。如果频繁发生这种情况的话，这意味着有一个隐藏的bug或者说通过增加缓存或者使用替代方案会有提升性能的空间。

这些高亮图层的选项同样在iOS模拟器的调试菜单也可用（图12.5）。我们之前说过用模拟器测试性能并不好，但如果能通过这些高亮选项识别出性能问题出在什么地方的话，那么使用iOS模拟器来验证问题是否解决也是比真机测试更有效的。

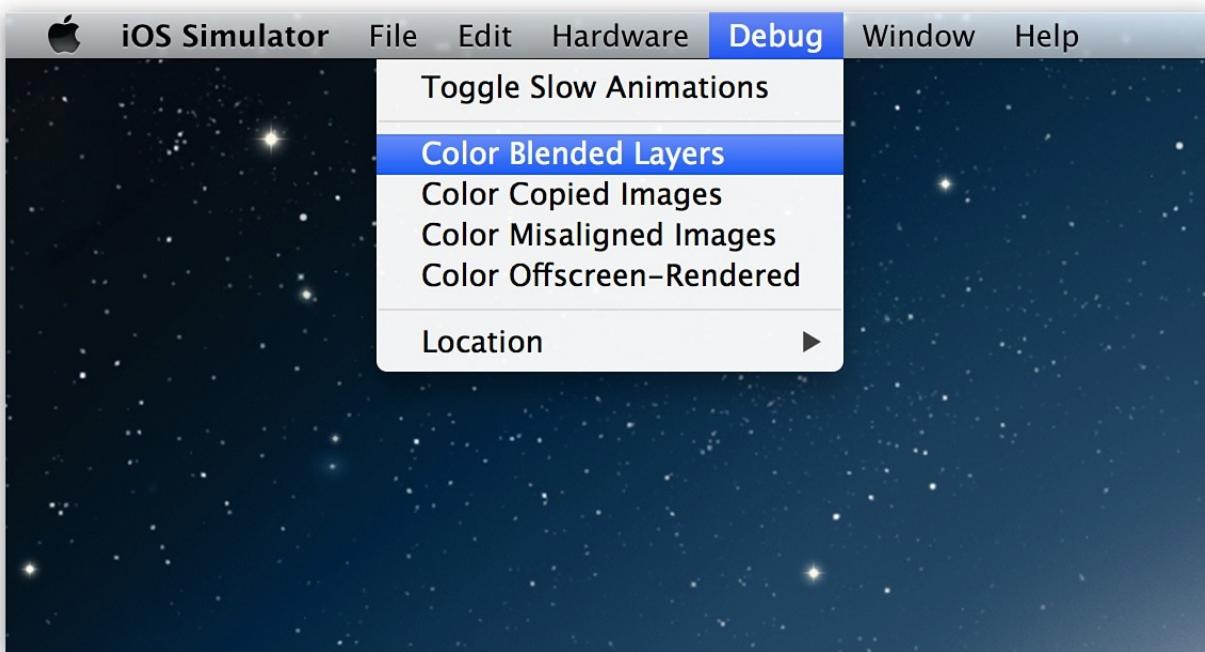


图12.5 iOS模拟器中Core Animation可视化调试选项

## OpenGL ES驱动

OpenGL ES驱动工具可以帮你测量GPU的利用率，同样也是一个很好的来判断和GPU相关动画性能的指示器。它同样也提供了类似Core Animation那样显示FPS的工具（图12.6）。

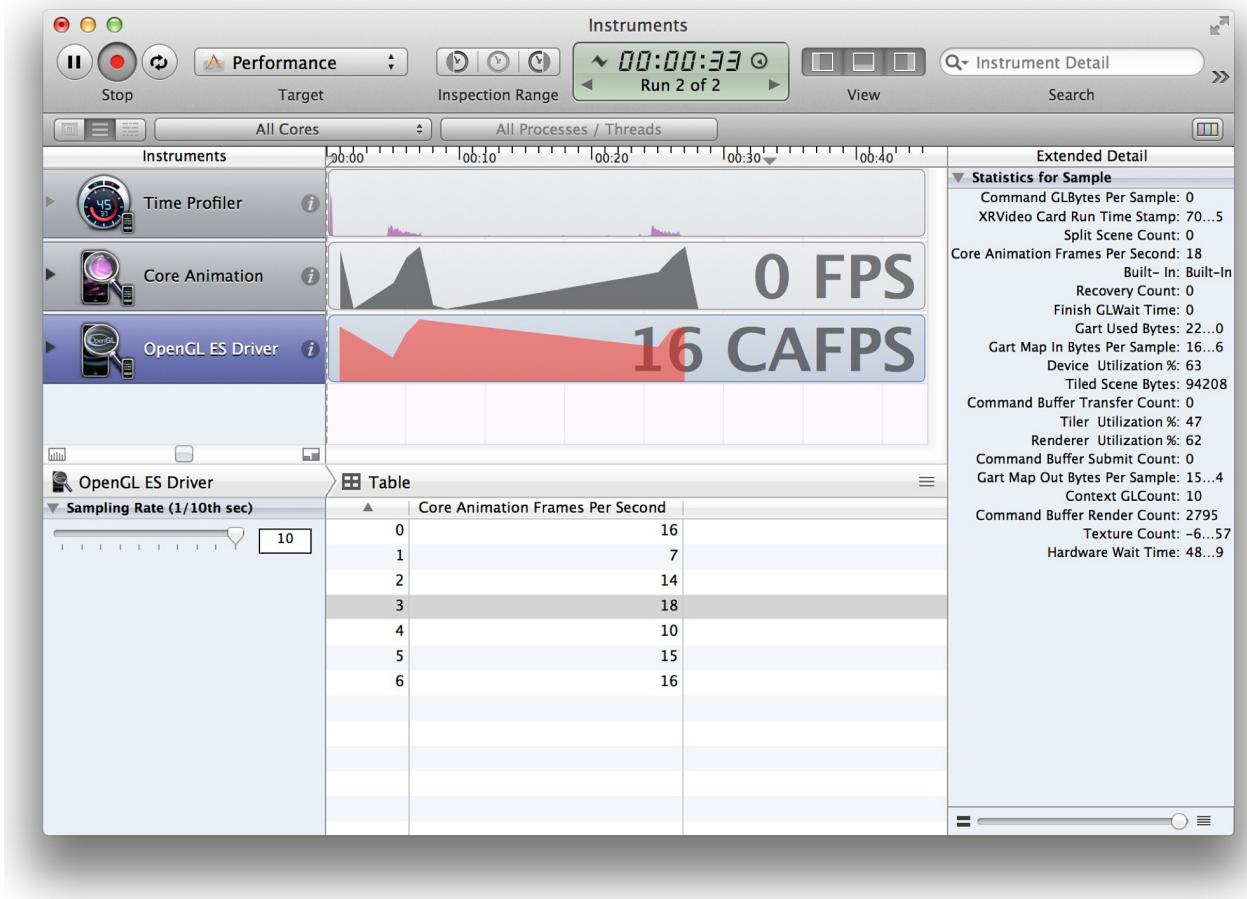


图12.6 OpenGL ES驱动工具

侧栏的邮编是一系列有用的工具。其中和Core Animation性能最相关的是如下几点：

- **Renderer Utilization** - 如果这个值超过了~50%，就意味着你的动画可能对帧率有所限制，很可能因为离屏渲染或者是重绘导致的过度混合。
- **Tiler Utilization** - 如果这个值超过了~50%，就意味着你的动画可能限制于几何结构方面，也就是在屏幕上有很多的图层占用了。

## 一个可用的案例

现在我们已经对Instruments中动画性能工具非常熟悉了，那么可以用它在现实中解决一些实际问题。

我们创建一个简单的显示模拟联系人姓名和头像列表的应用。注意即使把头像图片存在应用本地，为了使应用看起来更真实，我们分别实时加载图片，而不是用`- imageNamed:`预加载。同样添加一些图层阴影来使得列表显示得更真实。清单12.1展示了最初版本的实现。

## 清单12.1 使用假数据的一个简单联系人列表

```
#import "ViewController.h"
#import

@interface ViewController : UIViewController

@property (nonatomic, strong) NSArray *items;
@property (nonatomic, weak) IBOutlet UITableView *tableView;

@end

@implementation ViewController

- (NSString *)randomName
{
 NSArray *first = @[@"Alice", @"Bob", @"Bill", @"Charles", @"Dan"];
 NSArray *last = @[@"Appleseed", @"Bandicoot", @"Caravan", @"Dak"];
 NSUInteger index1 = (rand()/(double)INT_MAX) * [first count];
 NSUInteger index2 = (rand()/(double)INT_MAX) * [last count];
 return [NSString stringWithFormat:@"%@ %@", first[index1], last[index2]];
}

- (NSString *)randomAvatar
{
 NSArray *images = @[@"Snowman", @"Igloo", @"Cone", @"Spaceship"];
 NSUInteger index = (rand()/(double)INT_MAX) * [images count];
 return images[index];
}

- (void)viewDidLoad
{
 [super viewDidLoad];
 //set up data
 NSMutableArray *array = [NSMutableArray array];
 for (int i = 0; i < 1000; i++) {
 //add name
 [array addObject:@{@"name": [self randomName], @"image": [self randomAvatar]}];
 }
 self.items = array;
}
```

```
//register cell class
[self.tableView registerClass:[UITableViewCell class] forCellReuseIdentifier:@"Cell"]

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
 return [self.items count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
 //dequeue cell
 UITableViewCell *cell = [self.tableView dequeueReusableCellWithIdentifier:@"Cell"];
 //load image
 NSDictionary *item = self.items[indexPath.row];
 NSString *filePath = [[NSBundle mainBundle] pathForResource:item[@"name"] ofType:@"png"];
 //set image and text
 cell.imageView.image = [UIImage imageWithContentsOfFile:filePath];
 cell.textLabel.text = item[@"name"];
 //set image shadow
 cell.imageView.layer.shadowOffset = CGSizeMake(0, 5);
 cell.imageView.layer.shadowOpacity = 0.75;
 cell.clipsToBounds = YES;
 //set text shadow
 cell.textLabel.backgroundColor = [UIColor clearColor];
 cell.textLabel.layer.shadowOffset = CGSizeMake(0, 2);
 cell.textLabel.layer.shadowOpacity = 0.5;
 return cell;
}

@end
```

当快速滑动的时候就会非常卡（见图12.7的FPS计数器）。



图12.7 滑动帧率降到15FPS

仅凭直觉，我们猜测性能瓶颈应该在图片加载。我们实时从闪存加载图片，而且没有缓存，所以很可能是这个原因。我们可以用一些很赞的代码修复，然后使用GCD异步加载图片，然后缓存。。。等一下，在开始编码之前，测试一下假设是否成立。首先用我们的三个Instruments工具分析一下程序来定位问题。我们推测问题可能和图片加载相关，所以用Time Profiler工具来试试（图12.8）。

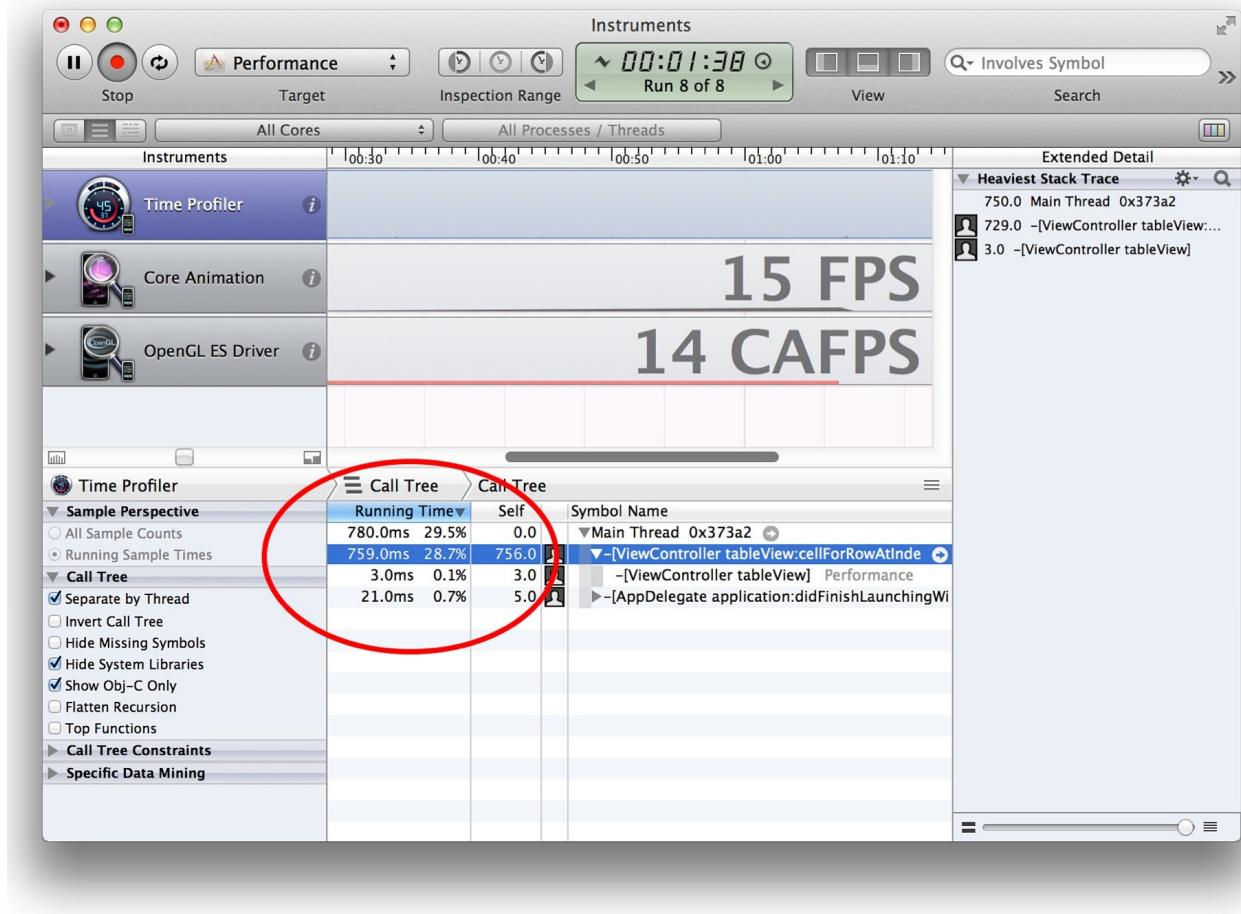


图12.8 用The timing profile分析联系人列表

`-tableView:cellForRowAtIndexPath:` 中的CPU时间总利用率只有~28%（也就是加载头像图片的地方），非常低。于是建议是CPU/IO并不是真正的限制因素。然后看看是不是GPU的问题：在OpenGL ES Driver工具中检测GPU利用率（图12.9）。

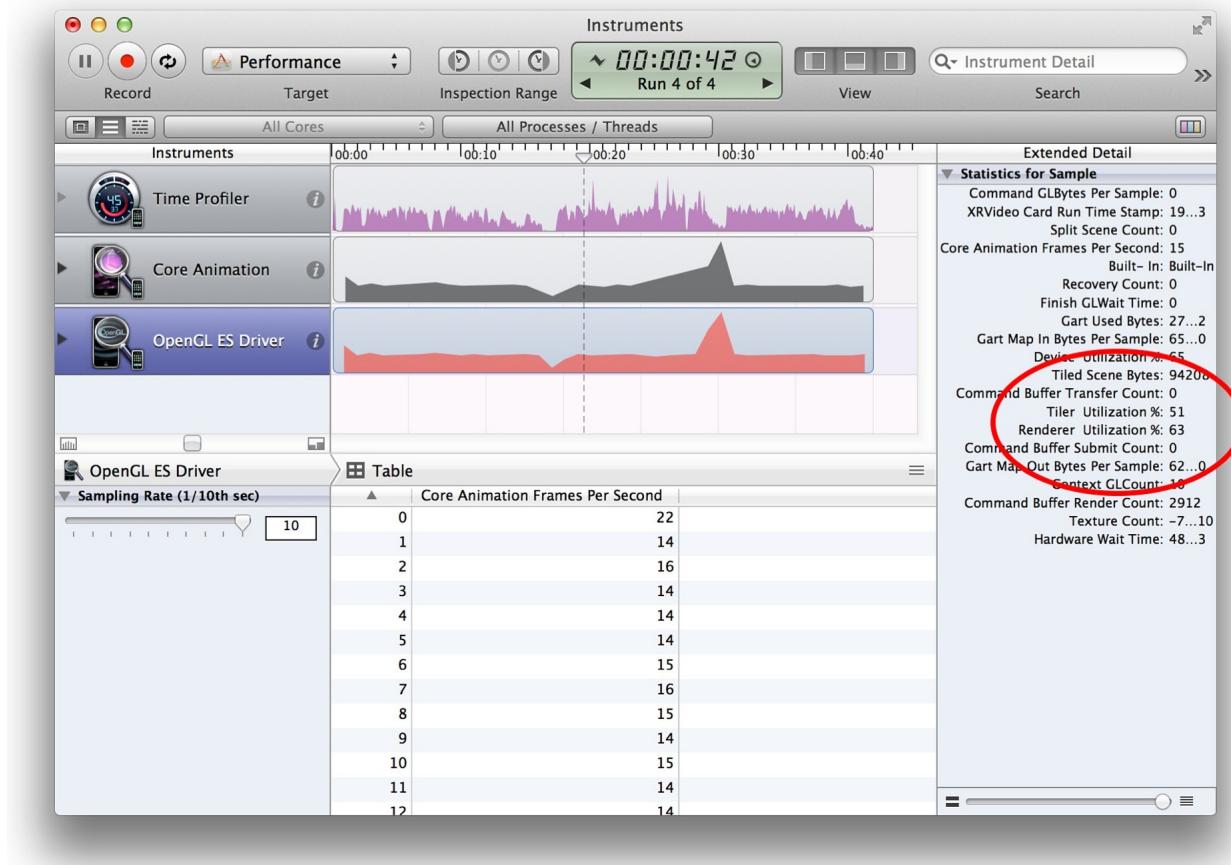


图12.9 OpenGL ES Driver工具显示的GPU利用率

渲染服务利用率的值达到51%和63%。看起来GPU需要做很多工作来渲染联系人列表。

为什么GPU利用率这么高呢？我们来用Core Animation调试工具选项来检查屏幕。首先打开Color Blended Layers（图12.10）。



图12.10 使用Color Blended Layers选项调试程序

屏幕上所有红色的部分都意味着字符标签视图的高级别混合，这很正常，因为我们把背景设置成了透明色来显示阴影效果。这就解释了为什么渲染利用率这么高了。

那么离屏绘制呢？打开Core Animation工具的Color Offscreen - Rendered Yellow选项（图12.11）。

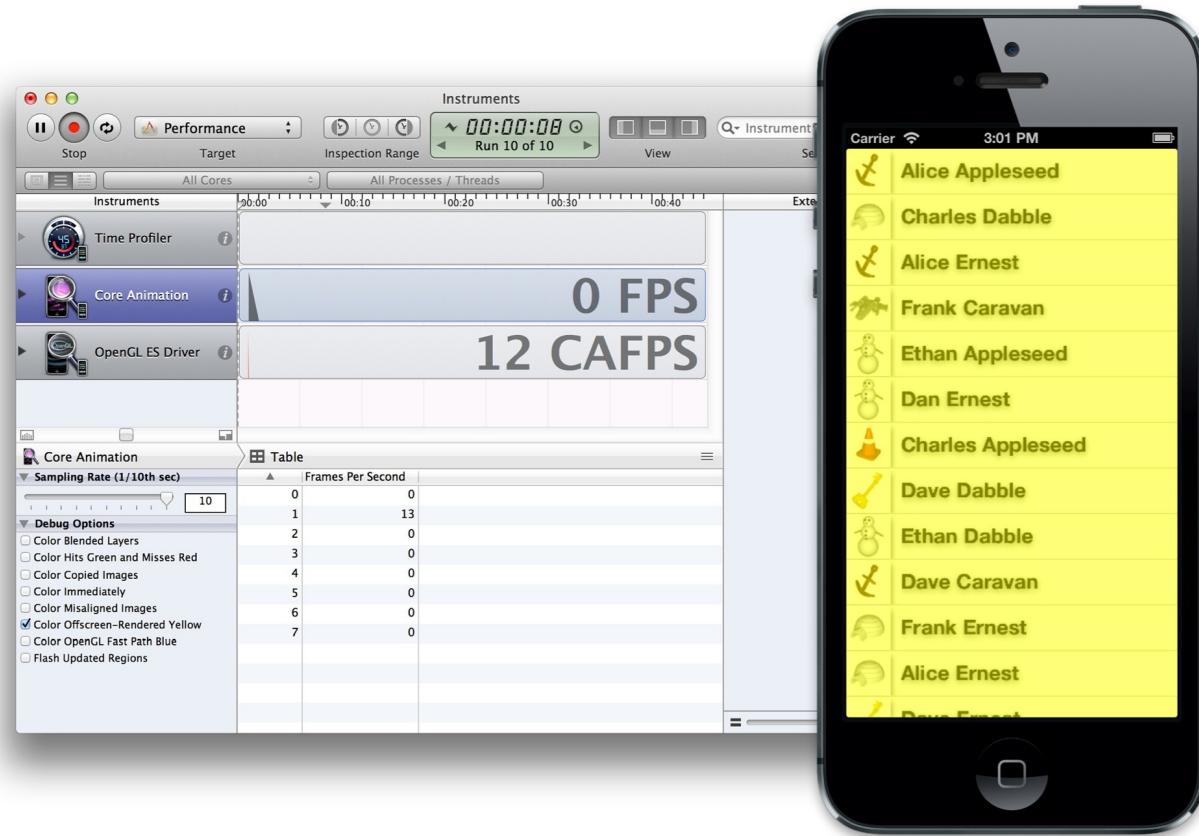


图12.11 Color Offscreen–Rendered Yellow选项

所有的表格单元内容都在离屏绘制。这一定是因为我们给图片和标签视图添加的阴影效果。在代码中禁用阴影，然后看下性能是否有提高（图12.12）。



图12.12 禁用阴影之后运行程序接近60FPS

问题解决了。干掉阴影之后，滑动很流畅。但是我们的联系人列表看起来没有之前好了。那如何保持阴影效果而且不会影响性能呢？

好吧，每一行的字符和头像在每一帧刷新的时候并不需要变，所以看起来 `UITableViewCell` 的图层非常适合做缓存。我们可以使用 `shouldRasterize` 来缓存图层内容。这将会让图层离屏之后渲染一次然后把结果保存起来，直到下次利用的时候去更新（见清单12.2）。

清单12.2 使用 `shouldRasterize` 提高性能

```

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath{
 //dequeue cell
 UITableViewCell *cell = [self.tableView dequeueReusableCellWithIdentifier];
 ...
 //set text shadow
 cell.textLabel.backgroundColor = [UIColor clearColor];
 cell.textLabel.layer.shadowOffset = CGSizeMake(0, 2);
 cell.textLabel.layer.shadowOpacity = 0.5;
 //rasterize
 cell.layer.shouldRasterize = YES;
 cell.layer.rasterizationScale = [UIScreen mainScreen].scale;
 return cell;
}

```

我们仍然离屏绘制图层内容，但是由于显式地禁用了栅格化，Core Animation就对绘图缓存了结果，于是对提高了性能。我们可以验证缓存是否有效，在Core Animation工具中点击Color Hits Green and Misses Red选项（图12.13）。



### 图12.13 Color Hits Green and Misses Red验证了缓存有效

结果和预期一致 - 大部分都是绿色，只有当滑动到屏幕上的时候会闪烁成红色。因此，现在帧率更加平滑了。

所以我们最初的设想是错的。图片的加载并不是真正的瓶颈所在，而且试图把它置于一个复杂的多线程加载和缓存的实现都将是徒劳。所以在动手修复之前验证问题所在是个很好的习惯！

## 总结

在这章中，我们学习了Core Animation是如何渲染，以及我们可能出现的瓶颈所在。你同样学习了如何使用Instruments来检测和修复性能问题。

在下三章中，我们将对每个普通程序的性能陷阱进行详细讨论，然后学习如何修复。

## 高效绘图

不必要的效率考虑往往是性能问题的万恶之源。——William Allan Wulf

在第12章『速度的曲率』我们学习如何用Instruments来诊断Core Animation性能问题。在构建一个iOS app的时候会遇到很多潜在的性能陷阱，但是在本章我们将着眼于有关绘制的性能问题。

## 软件绘图

术语绘图通常在Core Animation的上下文中指代软件绘图（意即：不由GPU协助的绘图）。在iOS中，软件绘图通常是由Core Graphics框架完成来完成。但是，在一些必要的情况下，相比Core Animation和OpenGL，Core Graphics要慢了不少。

软件绘图不仅效率低，还会消耗可观的内存。`CALayer` 只需要一些与自己相关的内存：只有它的寄宿图会消耗一定的内存空间。即使直接赋给 `contents` 属性一张图片，也不需要增加额外的照片存储大小。如果相同的一张图片被多个图层作为 `contents` 属性，那么他们将会共用同一块内存，而不是复制内存块。

但是一旦你实现了 `CALayerDelegate` 协议中的 `-drawLayer:inContext:` 方法或者 `UIView` 中的 `-drawRect:` 方法（其实就是前者的包装方法），图层就创建了一个绘制上下文，这个上下文需要的大小的内存可从这个算式得出：图层宽\*图层高\*4字节，宽高的单位均为像素。对于一个在Retina iPad上的全屏图层来说，这个内存量就是  $2048 \times 1526 \times 4$  字节，相当于12MB内存，图层每次重绘的时候都需要重新抹掉内存然后重新分配。

软件绘图的代价昂贵，除非绝对必要，你应该避免重绘你的视图。提高绘制性能的秘诀就在于尽量避免去绘制。

## 矢量图形

我们用Core Graphics来绘图的一个通常原因就是只是用图片或是图层效果不能轻易地绘制出矢量图形。矢量绘图包含一下这些：

- 任意多边形（不仅仅是一个矩形）
- 斜线或曲线
- 文本
- 渐变

举个例子，清单13.1 展示了一个基本的画线应用。这个应用将用户的触摸手势转换成一个 UIBezierPath 上的点，然后绘制到视图。我们在一个 UIView 子类 DrawingView 中实现了所有的绘制逻辑，这个情况下我们没有用上view controller。但是如果你喜欢你可以在view controller中实现触摸事件处理。图13.1 是代码运行结果。

清单13.1 用Core Graphics实现一个简单的绘图应用

```
#import "DrawingView.h"

@interface DrawingView : UIView

@property (nonatomic, strong) UIBezierPath *path;

@end

@implementation DrawingView

- (void)awakeFromNib
{
 //create a mutable path
 self.path = [[UIBezierPath alloc] init];
 self.path.lineJoinStyle = kCGLineJoinRound;
 self.path.lineCapStyle = kCGLineCapRound;

 self.path.lineWidth = 5;
}
```

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
 //get the starting point
 CGPoint point = [[touches anyObject] locationInView:self];

 //move the path drawing cursor to the starting point
 [self.path moveToPoint:point];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
 //get the current point
 CGPoint point = [[touches anyObject] locationInView:self];

 //add a new line segment to our path
 [self.path addLineToPoint:point];

 //redraw the view
 [self setNeedsDisplay];
}

- (void)drawRect:(CGRect)rect
{
 //draw path
 [[UIColor clearColor] setFill];
 [[UIColor redColor] setStroke];
 [self.path stroke];
}
@end
```



图13.1 用Core Graphics做一个简单的『素描』

这样实现的问题在于，我们画得越多，程序就会越慢。因为每次移动手指的时候都会重绘整个贝塞尔路径（`UIBezierPath`），随着路径越来越复杂，每次重绘的工作就会增加，直接导致了帧数的下降。看来我们需要一个更好的方法了。

Core Animation为这些图形类型的绘制提供了专门的类，并给他们提供硬件支持（第六章『专有图层』有详细提到）。`CAShapeLayer` 可以绘制多边形，直线和曲线。`CATextLayer` 可以绘制文本。`CAGradientLayer` 用来绘制渐变。这些总体上都比Core Graphics更快，同时他们也避免了创造一个寄宿图。

如果稍微将之前的代码变动一下，用 `CAShapeLayer` 替代Core Graphics，性能就会得到提高（见清单13.2）。虽然随着路径复杂性的增加，绘制性能依然会下降，但是只有当非常非常浮躁的绘制时才会感到明显的帧率差异。

### 清单13.2 用 `CAShapeLayer` 重新实现绘图应用

```
#import "DrawingView.h"
#import

@interface DrawingView ()

@property (nonatomic, strong) UIBezierPath *path;

@end

@implementation DrawingView
```

```
+ (Class)layerClass
{
 //this makes our view create a CAShapeLayer
 //instead of a CALayer for its backing layer
 return [CAShapeLayer class];
}

- (void)awakeFromNib
{
 //create a mutable path
 self.path = [[UIBezierPath alloc] init];

 //configure the layer
 CAShapeLayer *shapeLayer = (CAShapeLayer *)self.layer;
 shapeLayer.strokeColor = [UIColor redColor].CGColor;
 shapeLayer.fillColor = [UIColor clearColor].CGColor;
 shapeLayer.lineJoin = kCALineJoinRound;
 shapeLayer.lineCap = kCALineCapRound;
 shapeLayer.lineWidth = 5;
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
 //get the starting point
 CGPoint point = [[touches anyObject] locationInView:self];

 //move the path drawing cursor to the starting point
 [self.path moveToPoint:point];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
 //get the current point
 CGPoint point = [[touches anyObject] locationInView:self];

 //add a new line segment to our path
 [self.path addLineToPoint:point];

 //update the layer with a copy of the path
}
```

```
((CAShapeLayer *)self.layer).path = self.path.CGPath;
}
@end
```

## 脏矩形

有时候用 `CAShapeLayer` 或者其他矢量图形图层替代Core Graphics并不是那么切实可行。比如我们的绘图应用：我们用线条完美地完成了矢量绘制。但是设想一下如果我们能进一步提高应用的性能，让它就像一个黑板一样工作，然后用『粉笔』来绘制线条。模拟粉笔最简单的方法就是用一个『线刷』图片然后将它粘贴到用户手指碰触的地方，但是这个方法用 `CAShapeLayer` 没办法实现。

我们可以给每个『线刷』创建一个独立的图层，但是实现起来有很大的问题。屏幕上允许同时出现图层上线数量大约是几百，那样我们很快就会超出的。这种情况下我们没什么办法，就用Core Graphics吧（除非你想用OpenGL做一些更复杂的事情）。

我们的『黑板』应用的最初实现见清单13.3，我们更改了之前版本的 `DrawingView`，用一个画刷位置的数组代替 `UIBezierPath`。图13.2是运行结果

### 清单13.3 简单的类似黑板的应用

```
#import "DrawingView.h"
#import
#define BRUSH_SIZE 32

@interface DrawingView : UIView

@property (nonatomic, strong) NSMutableArray *strokes;

@end

@implementation DrawingView

- (void)awakeFromNib
{
 //create array
 self.strokes = [NSMutableArray array];
}

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
```

```
{
 //get the starting point
 CGPoint point = [[touches anyObject] locationInView:self];

 //add brush stroke
 [self addBrushStrokeAtPoint:point];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
 //get the touch point
 CGPoint point = [[touches anyObject] locationInView:self];

 //add brush stroke
 [self addBrushStrokeAtPoint:point];
}

- (void)addBrushStrokeAtPoint:(CGPoint)point
{
 //add brush stroke to array
 [self.strokes addObject:[NSValue valueWithCGPoint:point]];

 //needs redraw
 [self setNeedsDisplay];
}

- (void)drawRect:(CGRect)rect
{
 //redraw strokes
 for (NSValue *value in self.strokes) {
 //get point
 CGPoint point = [value CGPointValue];

 //get brush rect
 CGRect brushRect = CGRectMake(point.x - BRUSH_SIZE/2, point.y -
 BRUSH_SIZE/2, BRUSH_SIZE, BRUSH_SIZE);

 //draw brush stroke
 [[UIImage imageNamed:@"Chalk.png"] drawInRect:brushRect];
 }
}
```



图13.2 用程序绘制一个简单的『素描』

这个实现在模拟器上表现还不错，但是在真实设备上就没那么好了。问题在于每次手指移动的时候我们就会重绘之前的线刷，即使场景的大部分并没有改变。我们绘制地越多，就会越慢。随着时间的增加每次重绘需要更多的时间，帧数也会下降（见图13.3），如何提高性能呢？



图13.3 帧率和线条质量会随时间下降。

为了减少不必要的绘制，Mac OS和iOS设备将会把屏幕区分为需要重绘的区域和不需要重绘的区域。那些需要重绘的部分被称作『脏区域』。在实际应用中，鉴于非矩形区域边界裁剪和混合的复杂性，通常会区分出包含指定视图的矩形位置，而这个位置就是『脏矩形』。

当一个视图被改动过了，TA可能需要重绘。但是很多情况下，只是这个视图的一部分被改变了，所以重绘整个寄宿图就太浪费了。但是Core Animation通常并不了解你的自定义绘图代码，它也不能自己计算出脏区域的位置。然而，你的确可以提供这些信息。

当你检测到指定视图或图层的指定部分需要被重绘，你直接调用 `-setNeedsDisplayInRect:` 来标记它，然后将影响到的矩形作为参数传入。这样就会在一次视图刷新时调用视图的 `-drawRect:` （或图层代理的 `-drawLayer:inContext:` 方法）。

传入 `-drawLayer:inContext:` 的 `CGContext` 参数会自动被裁切以适应对应的矩形。为了确定矩形的尺寸大小，你可以用 `CGContextGetClipBoundingBox()` 方法来从上下文获得大小。调用 `-drawRect()` 会更简单，因为 `CGRect` 会作为参数直接传入。

你应该将你的绘制工作限制在这个矩形中。任何在此区域之外的绘制都将被自动无视，但是这样CPU花在计算和抛弃上的时间就浪费了，实在是太不值得了。

相比依赖于Core Graphics为你重绘，裁剪出自己的绘制区域可能会让你避免不必要的操作。那就是说，如果你的裁剪逻辑相当复杂，那还是让Core Graphics来代劳吧，记住：当你能高效完成的时候才这样做。

清单13.4 展示了一个 `-addBrushStrokeAtPoint:` 方法的升级版，它只重绘当前线刷的附近区域。另外也会刷新之前线刷的附近区域，我们也可以用 `CGRectIntersectsRect()` 来避免重绘任何旧的线刷以不至于覆盖已更新过的区域。这样做会显著地提高绘制效率（见图13.4）

清单13.4 用 `-setNeedsDisplayInRect:` 来减少不必要的绘制

```
- (void)addBrushStrokeAtPoint:(CGPoint)point
{
 //add brush stroke to array
 [self.strokes addObject:[NSValue valueWithCGPoint:point]];

 //set dirty rect
 [self setNeedsDisplayInRect:[self brushRectForPoint:point]];
}

- (CGRect)brushRectForPoint:(CGPoint)point
{
 return CGRectMake(point.x - BRUSH_SIZE/2, point.y - BRUSH_SIZE,
};

- (void)drawRect:(CGRect)rect
{
 //redraw strokes
 for (NSValue *value in self.strokes) {
 //get point
 CGPoint point = [value CGPointValue];

 //get brush rect
 CGRect brushRect = [self brushRectForPoint:point];

 //only draw brush stroke if it intersects dirty rect
 if (CGRectIntersectsRect(rect, brushRect)) {
 //draw brush stroke
 [[UIImage imageNamed:@"Chalk.png"] drawInRect:brushRect];
 }
 }
}
```

图13.4 更好的帧率和顺滑线条

## 异步绘制

UIKit的单线程天性意味着寄宿图通畅要在主线程上更新，这意味着绘制会打断用户交互，甚至让整个app看起来处于无响应状态。我们对此无能为力，但是如果能避免用户等待绘制完成就好多了。

针对这个问题，有一些方法可以用到：一些情况下，我们可以推测性地提前在另外一个线程上绘制内容，然后将由此绘出的图片直接设置为图层的内容。这实现起来可能不是很方便，但是在特定情况下是可行的。Core Animation提供了一些选择：`CATiledLayer` 和 `drawsAsynchronously` 属性。

### **CATiledLayer**

我们在第六章简单探索了一下 `CATiledLayer`。除了将图层再次分割成独立更新的小块（类似于脏矩形自动更新的概念），`CATiledLayer` 还有一个有趣的特性：在多个线程中为每个小块同时调用 `-drawLayer:inContext:` 方法。这就避免了阻塞用户交互而且能够利用多核心新片来更快地绘制。只有一个小块的 `CATiledLayer` 是实现异步更新图片视图的简单方法。

### **drawsAsynchronously**

iOS 6中，苹果为 `CALayer` 引入了这个令人好奇的属性，`drawsAsynchronously` 属性对传入 `-drawLayer:inContext:` 的 `CGContext`进行改动，允许`CGContext`延缓绘制命令的执行以至于不阻塞用户交互。

它与 `CATiledLayer` 使用的异步绘制并不相同。它自己的 `-drawLayer:inContext:` 方法只会在主线程调用，但是`CGContext`并不等待每个绘制命令的结束。相反地，它会将命令加入队列，当方法返回时，在后台线程逐个执行真正的绘制。

根据苹果的说法。这个特性在需要频繁重绘的视图上效果最好（比如我们的绘图应用，或者诸如 `UITableViewCell` 之类的），对那些只绘制一次或很少重绘的图层内容来说没什么太大的帮助。

## 总结

本章我们主要围绕用Core Graphics软件绘制讨论了一些性能挑战，然后探索了一些改进方法：比如提高绘制性能或者减少需要绘制的数量。

第14章，『图像IO』，我们将讨论图片的载入性能。

# 图像IO

| 潜伏期值得思考 - 凯文 帕萨特

在第13章“高效绘图”中，我们研究了和Core Graphics绘图相关的性能问题，以及如何修复。和绘图性能相关紧密相关的是图像性能。在这一章中，我们将研究如何优化从闪存驱动器或者网络中加载和显示图片。

## 加载和潜伏

绘图实际消耗的时间通常并不是影响性能的因素。图片消耗很大一部分内存，而且不太可能把需要显示的图片都保留在内存中，所以需要在应用运行的时候周期性地加载和卸载图片。

图片文件加载的速度被CPU和IO（输入/输出）同时影响。iOS设备中的闪存已经比传统硬盘快很多了，但仍然比RAM慢将近200倍左右，这就需要很小心地管理加载，来避免延迟。

只要有可能，试着在程序生命周期不易察觉的时候来加载图片，例如启动，或者在屏幕切换的过程中。按下按钮和按钮响应事件之间最大的延迟大概是200ms，这比动画每一帧切换的16ms小得多。你可以在程序首次启动的时候加载图片，但是如果20秒内无法启动程序的话，iOS检测计时器就会终止你的应用（而且如果启动大于2，3秒的话用户就会抱怨了）。

有些时候，提前加载所有的东西并不明智。比如说包含上千张图片的图片传送带：用户希望能够平滑快速翻动图片，所以就不可能提前预加载所有图片；那样会消耗太多的时间和内存。

有时候图片也需要从远程网络连接中下载，这将会比从磁盘加载要消耗更多的时间，甚至可能由于连接问题而加载失败（在几秒钟尝试之后）。你不能够在主线程中加载网络造成等待，所以需要后台线程。

## 线程加载

在第12章“性能调优”我们的联系人列表例子中，图片都非常小，所以可以在主线程同步加载。但是对于大图来说，这样做就不太合适了，因为加载会消耗很长一段时间，造成滑动的不流畅。滑动动画会在主线程的run loop中更新，所以会有更多运行在渲染服务进程中CPU相关的性能问题。

清单14.1显示了一个通过 `UICollectionView` 实现的基础的图片传送器。图片在主线程中 `-collectionView:cellForItemAtIndexPath:` 方法中同步加载（见图14.1）。

清单14.1 使用 `UICollectionView` 实现的图片传送器

```
#import "ViewController.h"

@interface ViewController()

@property (nonatomic, copy) NSArray *imagePaths;
@property (nonatomic, weak) IBOutlet UICollectionView *collectionView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 //set up data
 self.imagePaths =
 [[NSBundle mainBundle] pathsForResourcesOfType:@"png" inDirectory:@""];
 //register cell class
 [self.collectionView registerClass:[UICollectionViewCell class]
 forCellWithReuseIdentifier:@"cellIdentifier"];
}

- (NSInteger)collectionView:(UICollectionView *)collectionView numberOfItemsInSection:(NSInteger)section
{
 return [self.imagePaths count];
}

- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
 //dequeue cell
 UICollectionViewCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"cellIdentifier" forIndexPath:indexPath];

 //add image view
 const NSInteger imageTag = 99;
 UIImageView *imageView = (UIImageView *)[cell viewWithTag:imageTag];
 if (!imageView) {
 imageView = [[UIImageView alloc] initWithFrame: cell.contentView.bounds];
 imageView.tag = imageTag;
 [cell.contentView addSubview:imageView];
 }
 imageView.image = [UIImage imageNamed:[self.imagePaths objectAtIndex:indexPath.item]];
}
```

```
}

//set image
NSString *imagePath = self.imagePaths[indexPath.row];
imageView.image = [UIImage imageWithContentsOfFile:imagePath];
return cell;
}

@end
```

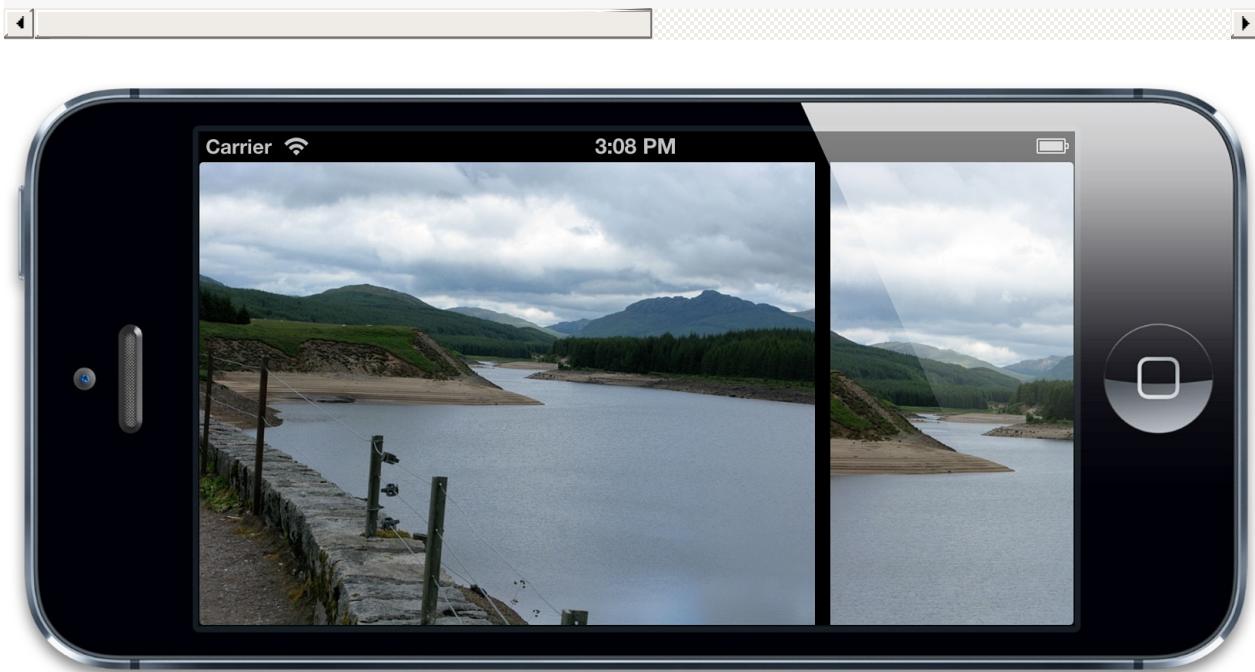


图14.1 运行中的图片传送器

传送器中的图片尺寸为800x600像素的PNG，对iPhone5来说，1/60秒要加载大概700KB左右的图片。当传送器滚动的时候，图片也在实时加载，于是（预期中的）卡顿就发生了。时间分析工具（图14.2）显示了很多时间都消耗在了 `UIImage` 的 `+imageWithContentsOfFile:` 方法中了。很明显，图片加载造成了瓶颈。

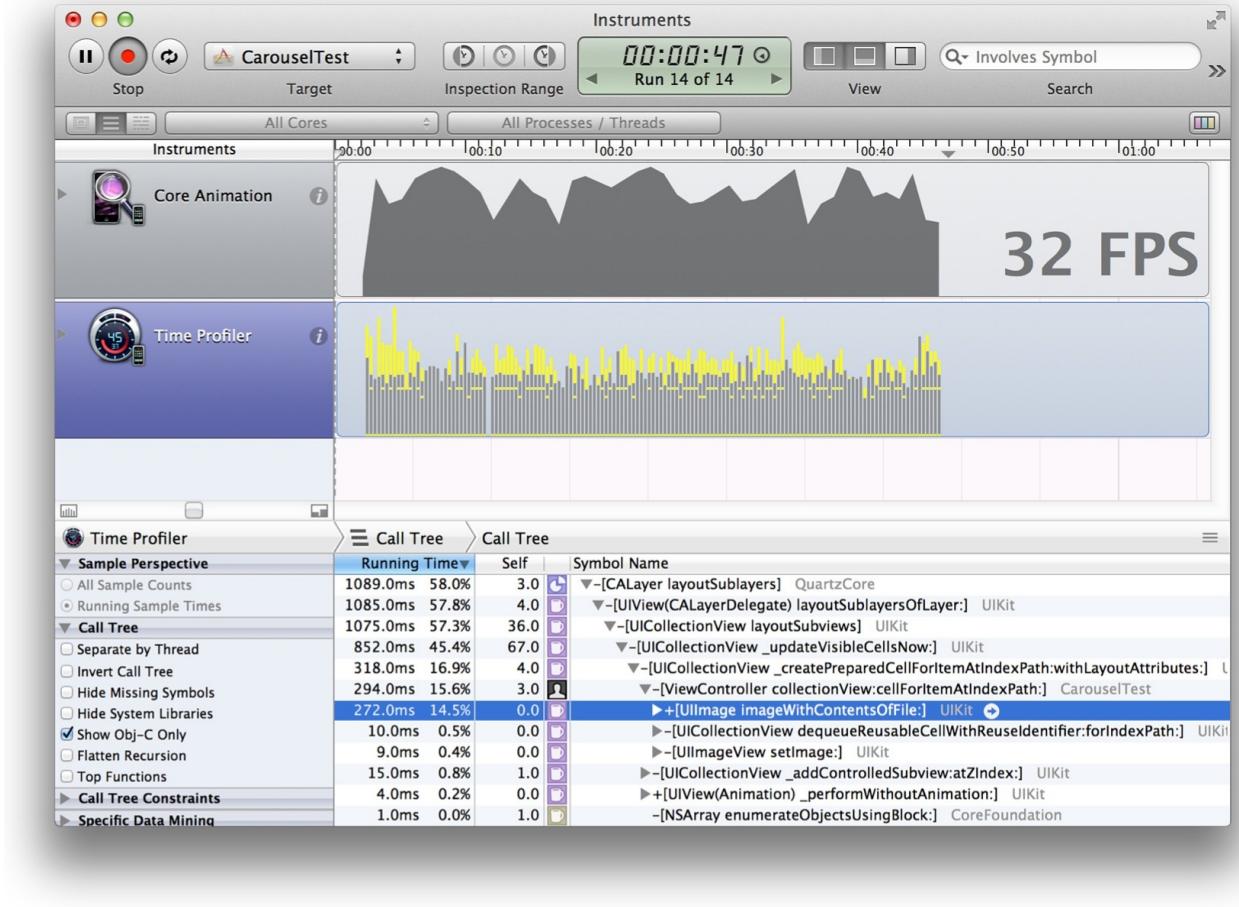


图14.2 时间分析工具展示了CPU瓶颈

这里提升性能唯一的方式就是在另一个线程中加载图片。这并不能够降低实际的加载时间（可能情况会更糟，因为系统可能要消耗CPU时间来处理加载的图片数据），但是主线程能够有时间做一些别的事情，比如响应用户输入，以及滑动动画。

为了在后台线程加载图片，我们可以使用GCD或者 NSOperationQueue 创建自定义线程，或者使用 CATiledLayer 。为了从远程网络加载图片，我们可以使用异步的 NSURLConnection ，但是对本地存储的图片，并不十分有效。

## GCD和 NSOperationQueue

GCD (Grand Central Dispatch) 和 NSOperationQueue 很类似，都给我们提供了队列闭包块来在线程中按一定顺序来执行。 NSOperationQueue 有一个 Objective-C 接口（而不是使用 GCD 的全局 C 函数），同样在操作优先级和依赖关系上提供了很好的粒度控制，但是需要更多地设置代码。

清单14.2显示了在低优先级的后台队列而不是主线程使用GCD加载图片的`- collectionView:cellForItemAtIndexPath:`方法，然后当需要加载图片到视图的时候切换到主线程，因为在后台线程访问视图会有安全隐患。

由于视图在`UICollectionView`会被循环利用，我们加载图片的时候不能确定是否被不同的索引重新复用。为了避免图片加载到错误的视图中，我们在加载前把单元格打上索引的标签，然后在设置图片的时候检测标签是否发生了改变。

#### 清单14.2 使用GCD加载传送图片

```

- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
 cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
 //dequeue cell
 UICollectionViewCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"ImageCell" forIndexPath:indexPath];

 //add image view
 const NSInteger imageTag = 99;
 UIImageView *imageView = (UIImageView *)[cell viewWithTag:imageTag];
 if (!imageView) {
 imageView = [[UIImageView alloc] initWithFrame: cell.contentView.bounds];
 imageView.tag = imageTag;
 [cell.contentView addSubview:imageView];
 }
 //tag cell with index and clear current image
 cell.tag = indexPath.row;
 imageView.image = nil;
 //switch to background thread
 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
 //load image
 NSInteger index = indexPath.row;
 NSString *imagePath = self.imagePaths[index];
 UIImage *image = [UIImage imageWithContentsOfFile:imagePath];
 //set image on main thread, but only if index still matches
 dispatch_async(dispatch_get_main_queue(), ^{
 if (index == cell.tag) {
 imageView.image = image;
 }
 });
 });
 return cell;
}

```

当运行更新后的版本，性能比之前不用线程的版本好多了，但仍然并不完美（图14.3）。

我们可以看到 `+imageWithContentsOfFile:` 方法并不在CPU时间轨迹的最顶部，所以我们的确修复了延迟加载的问题。问题在于我们假设传送器的性能瓶颈在于图片文件的加载，但实际上并不是这样。加载图片数据到内存中只是问题的第一

部分。

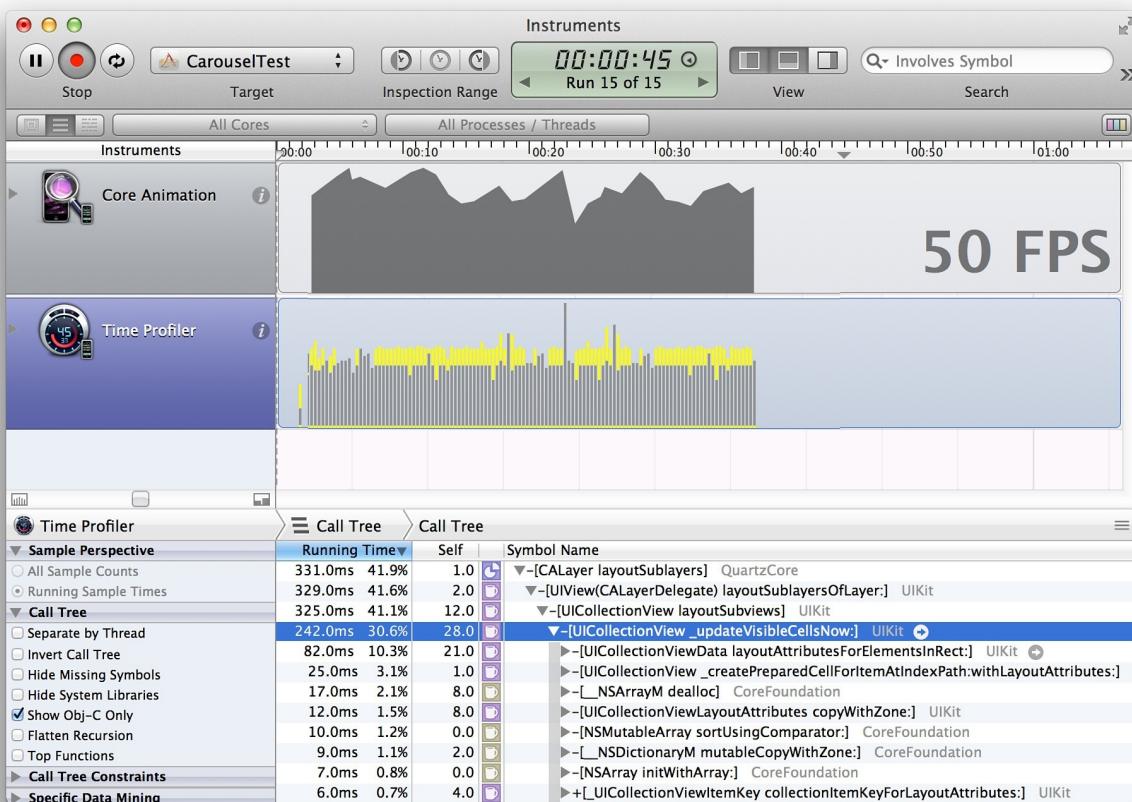


图14.3 使用后台线程加载图片来提升性能

## 延迟解压

一旦图片文件被加载就必须要进行解码，解码过程是一个相当复杂的任务，需要消耗非常长的时间。解码后的图片将同样使用相当大的内存。

用于加载的CPU时间相对于解码来说根据图片格式而不同。对于PNG图片来说，加载会比JPEG更长，因为文件可能更大，但是解码会相对较快，而且Xcode会把PNG图片进行解码优化之后引入工程。JPEG图片更小，加载更快，但是解压的步骤要消耗更长的时间，因为JPEG解压算法比基于zip的PNG算法更加复杂。

当加载图片的时候，iOS通常会延迟解压图片的时间，直到加载到内存之后。这就会在准备绘制图片的时候影响性能，因为需要在绘制之前进行解压（通常是消耗时间的问题所在）。

最简单的方法就是使用 `UIImage` 的 `+imageNamed:` 方法避免延时加载。不像 `+imageWithContentsOfFile:` (和其他别的 `UIImage` 加载方法)，这个方法会在加载图片之后立刻进行解压 (就和本章之前我们谈到的好处一样)。问题在于 `+imageNamed:` 只对从应用资源束中的图片有效，所以对用户生成的图片内容或者是下载的图片就没法使用了。

另一种立刻加载图片的方法就是把它设置成图层内容，或者是 `UIImageView` 的 `image` 属性。不幸的是，这又需要在主线程执行，所以不会对性能有所提升。

第三种方式就是绕过 `UIKit`，像下面这样使用 `ImageIO` 框架：

```
NSInteger index = indexPath.row;
NSURL * imageURL = [NSURL fileURLWithPath:self.imagePaths[index]];
NSDictionary * options = @{@"__bridge id")kCGImageSourceShouldCache: (CGImageSourceRef source = CGImageSourceCreateWithURL((__bridge CFURLRef)imageURL, options);
CGImageRef imageRef = CGImageSourceCreateImageAtIndex(source, 0, (__bridge CFDictionaryRef)options);
UIImage * image = [UIImage imageWithCGImage:imageRef];
CGImageRelease(imageRef);
CFRelease(source);
```

这样就可以使用 `kCGImageSourceShouldCache` 来创建图片，强制图片立刻解压，然后在图片的生命周期保留解压后的版本。

最后一种方式就是使用 `UIKit` 加载图片，但是立刻会知道 `CGContext` 中去。图片必须要在绘制之前解压，所以就强制了解压的及时性。这样的好处在于绘制图片可以再后台线程 (例如加载本身) 执行，而不会阻塞 UI。

有两种方式可以为强制解压提前渲染图片：

- 将图片的一个像素绘制成一个像素大小的 `CGContext`。这样仍然会解压整张图片，但是绘制本身并没有消耗任何时间。这样的好处在于加载的图片并不会在特定的设备上为绘制做优化，所以可以在任何时间点绘制出来。同样 iOS 也可以丢弃解压后的图片来节省内存了。
- 将整张图片绘制到 `CGContext` 中，丢弃原始的图片，并且用一个从上下文内容中新的图片来代替。这样比绘制单一像素那样需要更加复杂的计算，但是因此产生的图片将会为绘制做优化，而且由于原始压缩图片被抛弃了，iOS 就不能够随时丢弃任何解压后的图片来节省内存了。

需要注意的是苹果特别推荐了不要使用这些诡计来绕过标准图片解压逻辑（所以也是他们选择用默认处理方式的原因），但是如果你使用很多大图来构建应用，那如果想提升性能，就只能和系统博弈了。

如果不使用 `+imageNamed:`，那么把整张图片绘制到 `CGContext` 可能是最佳的方式了。尽管你可能认为多余的绘制相较于其他的解压技术而言性能不是很高，但是新创建的图片（在特定的设备上做过优化）可能比原始图片绘制的更快。

同样，如果想显示图片到比原始尺寸小的容器中，那么一次性在后台线程重新绘制到正确的尺寸会比每次显示的时候都做缩放会更有效（尽管在这个例子中我们加载的图片呈现正确的尺寸，所以不需要多余的优化）。

如果修改了 `-collectionView:cellForItemAtIndexPath:` 方法来重绘图片（清单14.3），你会发现滑动更加平滑。

### 清单14.3 强制图片解压显示

```

- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
 cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
 //dequeue cell
 UICollectionViewCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"ImageCell" forIndexPath:indexPath];
 ...
 //switch to background thread
 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0));
 //load image
 NSInteger index = indexPath.row;
 NSString *imagePath = self.imagePaths[index];
 UIImage *image = [UIImage imageWithContentsOfFile:imagePath];
 //redraw image using device context
 UIGraphicsBeginImageContextWithOptions(imageView.bounds.size, NO, 0);
 [image drawInRect:imageView.bounds];
 image = UIGraphicsGetImageFromCurrentImageContext();
 UIGraphicsEndImageContext();
 //set image on main thread, but only if index still matches
 dispatch_async(dispatch_get_main_queue(), ^{
 if (index == cell.tag) {
 imageView.image = image;
 }
 });
}
return cell;
}

```

## CATiledLayer

如第6章“专用图层”中的例子所示，`CATiledLayer` 可以用来异步加载和显示大型图片，而不阻塞用户输入。但是我们同样可以用 `CATiledLayer` 在 `UICollectionView` 中为每个表格创建分离的 `CATiledLayer` 实例加载传动器图片，每个表格仅使用一个图层。

这样使用 `CATiledLayer` 有几个潜在的弊端：

- `CATiledLayer` 的队列和缓存算法没有暴露出来，所以我们只能祈祷它能匹配我们的需求

- `CATiledLayer` 需要我们每次重绘图片到 `CGContext` 中，即使它已经解压缩，而且和我们单元格尺寸一样（因此可以直接用作图层内容，而不需要重绘）。

我们来看看这些弊端有没有造成不同：清单14.4显示了使用 `CATiledLayer` 对图片传送器的重新实现。

清单14.4 使用 `CATiledLayer` 的图片传送器

```
#import "ViewController.h"
#import

@interface ViewController()

@property (nonatomic, copy) NSArray *imagePaths;
@property (nonatomic, weak) IBOutlet UICollectionView *collectionView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 //set up data
 self.imagePaths = [[NSBundle mainBundle] pathsForResourcesOfType:@"png"];
 [self.collectionView registerClass:[UICollectionViewCell class]
 forCellWithReuseIdentifier:@"cellIdentifier"];
}

- (NSInteger)collectionView:(UICollectionView *)collectionView numberOfItemsInSection:(NSInteger)section
{
 return [self.imagePaths count];
}

- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
 //dequeue cell
 UICollectionViewCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"cellIdentifier" forIndexPath:indexPath];
 //add the tiled layer
 CATiledLayer *tileLayer = [cell.contentView.layer.sublayers last];
 if (!tileLayer) {
```

```

 tileLayer = [CATiledLayer layer];
 tileLayer.frame = cell.bounds;
 tileLayer.contentsScale = [UIScreen mainScreen].scale;
 tileLayer.tileSize = CGSizeMake(cell.bounds.size.width * [l
 tileLayer.delegate = self;
 [tileLayer setValue:@(indexPath.row) forKey:@"index"];
 [cell.contentView.layer addSublayer:tileLayer];

 }

 //tag the layer with the correct index and reload
 tileLayer.contents = nil;
 [tileLayer setValue:@(indexPath.row) forKey:@"index"];
 [tileLayer setNeedsDisplay];
 return cell;
}

- (void)drawLayer:(CATiledLayer *)layer inContext:(CGContextRef)ctx
{
 //get image index
 NSInteger index = [[layer valueForKey:@"index"] integerValue];
 //load tile image
 NSString *imagePath = self.imagePaths[index];
 UIImage *tileImage = [UIImage imageWithContentsOfFile:imagePath];
 //calculate image rect
 CGFloat aspectRatio = tileImage.size.height / tileImage.size.w
 CGRect imageRect = CGRectMakeZero;
 imageRect.size.width = layer.bounds.size.width;
 imageRect.size.height = layer.bounds.size.height * aspectRatio;
 imageRect.origin.y = (layer.bounds.size.height - imageRect.size
 //draw tile
 UIGraphicsPushContext(ctx);
 [tileImage drawInRect:imageRect];
 UIGraphicsPopContext();
}

@end

```

需要解释几点：

- `CATiledLayer` 的 `tileSize` 属性单位是像素，而不是点，所以为了保证瓦片和表格尺寸一致，需要乘以屏幕比例因子。
- 在 `-drawLayer:inContext:` 方法中，我们需要知道图层属于哪一个 `indexPath` 以加载正确的图片。这里我们利用了 `CALayer` 的 KVC 来存储和检索任意的值，将图层和索引打标签。

结果 `CATiledLayer` 工作的很好，性能问题解决了，而且和用 GCD 实现的代码量差不多。仅有一个问题在于图片加载到屏幕上后有一个明显的淡入（图 14.4）。

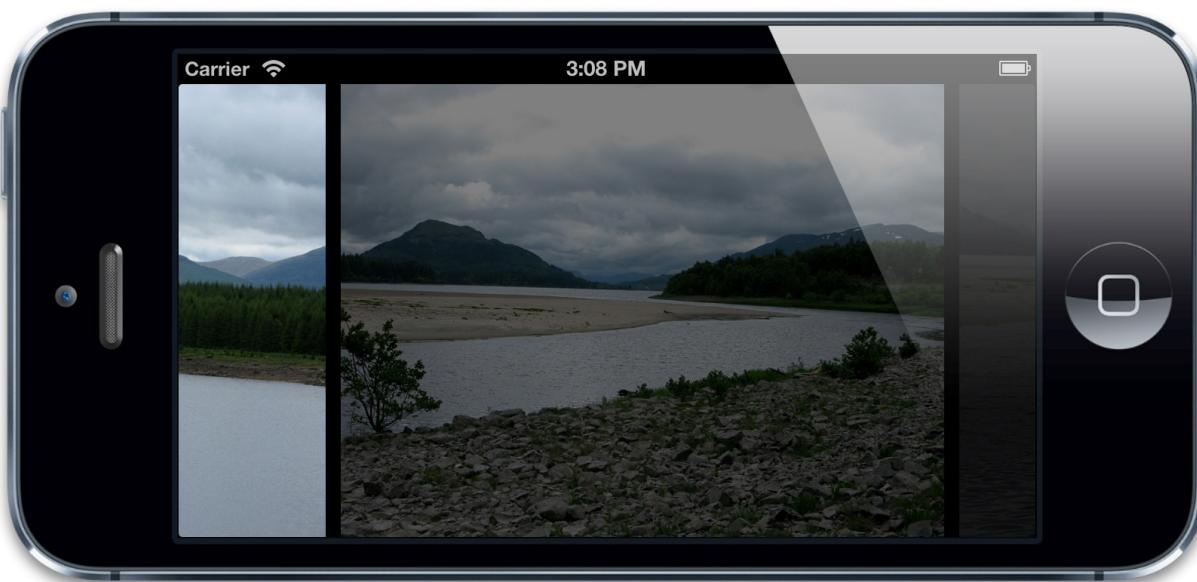


图 14.4 加载图片之后的淡入

我们可以调整 `CATiledLayer` 的 `fadeDuration` 属性来调整淡入的速度，或者直接将整个渐变移除，但是这并没有根本性地去除问题：在图片加载到准备绘制的时候总会有一个延迟，这将会导致滑动时候新图片的跳入。这并不是 `CATiledLayer` 的问题，使用 GCD 的版本也有这个问题。

即使使用上述我们讨论的所有加载图片和缓存的技术，有时候仍然会发现实时加载大图还是有问题。就和 13 章中提到的那样，iPad 上一整个视网膜屏图片分辨率达到了 2048x1536，而且会消耗 12MB 的 RAM（未压缩）。第三代 iPad 的硬件并不能支持 1/60 秒的帧率加载，解压和显示这种图片。即使用后台线程加载来避免动画卡顿，仍然解决不了问题。

我们可以在加载的同时显示一个占位图片，但这并没有根本解决问题，我们可以做到更好。

## 分辨率交换

视网膜分辨率（根据苹果市场定义）代表了人的肉眼在正常视角距离能够分辨的最小像素尺寸。但是这只能应用于静态像素。当观察一个移动图片时，你的眼睛就会对细节不敏感，于是一个低分辨率的图片和视网膜质量的图片没什么区别了。

如果需要快速加载和显示移动大图，简单的办法就是欺骗人眼，在移动传送器的时候显示一个小图（或者低分辨率），然后当停止的时候再换成大图。这意味着我们需要对每张图片存储两份不同分辨率的副本，但是幸运的是，由于需要同时支持 Retina 和非 Retina 设备，本来这就是普遍要做的。

如果从远程源或者用户的相册加载没有可用的低分辨率版本图片，那就可以动态将大图绘制到较小的 `CGContext`，然后存储到某处以备复用。

为了做到图片交换，我们需要利用 `UIScrollView` 的一些实现 `UIScrollViewDelegate` 协议的委托方法（和其他类似于 `UITableView` 和 `UICollectionView` 基于滚动视图的控件一样）：

```
- (void)scrollViewDidEndDragging:(UIScrollView *)scrollView willDecelerate:(BOOL)willDecelerate;
- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView;
```

你可以使用这几个方法来检测传送器是否停止滚动，然后加载高分辨率的图片。只要高分辨率图片和低分辨率图片尺寸颜色保持一致，你会很难察觉到替换的过程（确保在同一台机器使用相同的图像程序或者脚本生成这些图片）。

# 缓存

如果有很多张图片要显示，最好不要提前把所有都加载进来，而是应该当移出屏幕之后立刻销毁。通过选择性的缓存，你就可以避免来回滚动时图片重复性的加载了。

缓存其实很简单：就是存储昂贵计算后的结果（或者是从闪存或者网络加载的文件）在内存中，以便后续使用，这样访问起来很快。问题在于缓存本质上是一个权衡过程 - 为了提升性能而消耗了内存，但是由于内存是一个非常宝贵的资源，所以不能把所有东西都做缓存。

何时将何物做缓存（做多久）并不总是很明显。幸运的是，大多情况下，iOS都为我们做好了图片的缓存。

## +imageNamed: 方法

之前我们提到使用 `[UIImage imageNamed:]` 加载图片有个好处在于可以立刻解压图片而不用等到绘制的时候。但是 `[UIImage imageNamed:]` 方法有另一个非常显著的好处：它在内存中自动缓存了解压后的图片，即使你自己没有保留对它的任何引用。

对于iOS应用那些主要的图片（例如图标，按钮和背景图片），使用 `[UIImage imageNamed:]` 加载图片是最简单最有效的方式。在nib文件中引用的图片同样也是这个机制，所以你很多时候都在隐式的使用它。

但是 `[UIImage imageNamed:]` 并不适用任何情况。它为用户界面做了优化，但是并不是对应用程序需要显示的所有类型的图片都适用。有些时候你还是要实现自己的缓存机制，原因如下：

- `[UIImage imageNamed:]` 方法仅仅适用于在应用程序资源束目录下的图片，但是大多数应用的许多图片都要从网络或者是用户的相机中获取，所以 `[UIImage imageNamed:]` 就没法用了。
- `[UIImage imageNamed:]` 缓存用来存储应用界面的图片（按钮，背景等）。如果对照片这种大图也用这种缓存，那么iOS系统就很可能会移除这些图片来节省内存。那么在切换页面时性能就会下降，因为这些图片都需要重新

加载。对传送器的图片使用一个单独的缓存机制就可以把它和应用图片的生命周期解耦。

- `[UIImage imageNamed:]` 缓存机制并不是公开的，所以你不能很好地控制它。例如，你没法做到检测图片是否在加载之前就做了缓存，不能够设置缓存大小，当图片没用的时候也不能把它从缓存中移除。

## 自定义缓存

构建一个所谓的缓存系统非常困难。菲尔 卡尔顿曾经说过：“在计算机科学中只有两件难事：缓存和命名”。

如果要写自己的图片缓存的话，那该如何实现呢？让我们来看看要涉及哪些方面：

- 选择一个合适的缓存键 - 缓存键用来做图片的唯一标识。如果实时创建图片，通常不太好生成一个字符串来区别的图片。在我们的图片传送带例子中就很简单，我们可以用图片的文件名或者表格索引。
- 提前缓存 - 如果生成和加载数据的代价很大，你可能想当第一次需要用到的时候再去加载和缓存。提前加载的逻辑是应用内在就有的，但是在我们的例子中，这也非常好实现，因为对于一个给定的位置和滚动方向，我们就可以精确地判断出哪一张图片将会出现。
- 缓存失效 - 如果图片文件发生了变化，怎样才能通知到缓存更新呢？这是个非常困难的问题（就像菲尔 卡尔顿提到的），但是幸运的是当从程序资源加载静态图片的时候并不需要考虑这些。对用户提供的图片来说（可能会被修改或者覆盖），一个比较好的方式就是当图片缓存的时候打上一个时间戳以便当文件更新的时候作比较。
- 缓存回收 - 当内存不够的时候，如何判断哪些缓存需要清空呢？这就需要到你写一个合适的算法了。幸运的是，对缓存回收的问题，苹果提供了一个叫做 `NSCache` 通用的解决方案

## NSCache

`NSCache` 和 `NSDictionary` 类似。你可以通过 `-setObject:forKey:` 和 `-objectForKey:` 方法分别来插入，检索。和字典不同的是，`NSCache` 在系统低内存的时候自动丢弃存储的对象。

`NSCache` 用来判断何时丢弃对象的算法并没有在文档中给出，但是你可以使用 `-setCountLimit:` 方法设置缓存大小，以及 `-setObject:forKey:cost:` 来对每个存储的对象指定消耗的值来提供一些暗示。

指定消耗数值可以用来指定相对的重建成本。如果对大图指定一个大的消耗值，那么缓存就知道这些物体的存储更加昂贵，于是当有大的性能问题的时候才会丢弃这些物体。你也可以用 `-setTotalCostLimit:` 方法来指定全体缓存的尺寸。

`NSCache` 是一个普遍的缓存解决方案，我们创建一个比传送器案例更好的自定义的缓存类。（例如，我们可以基于不同的缓存图片索引和当前中间索引来判断哪些图片需要首先被释放）。但是 `NSCache` 对我们当前的缓存需求来说已经足够了；没必要过早做优化。

使用图片缓存和提前加载的实现来扩展之前的传送器案例，然后来看看是否效果更好（见清单14.5）。

#### 清单14.5 添加缓存

```
#import "ViewController.h"

@interface ViewController()

@property (nonatomic, copy) NSArray *imagePaths;
@property (nonatomic, weak) IBOutlet UICollectionView *collectionView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 //set up data
 self.imagePaths = [[NSBundle mainBundle] pathsForResourcesOfType:@"png"];
 //register cell class
 [self.collectionView registerClass:[UICollectionViewCell class]
 forCellWithReuseIdentifier:@"cellIdentifier"];
}

- (NSInteger)collectionView:(UICollectionView *)collectionView numberOfItemsInSection:(NSInteger)section
{
 return [self.imagePaths count];
}
```

```

}

- (UIImage *)loadImageAtIndex:(NSUInteger)index
{
 //set up cache
 static NSCache *cache = nil;
 if (!cache) {
 cache = [[NSCache alloc] init];
 }
 //if already cached, return immediately
 UIImage *image = [cache objectForKey:@(index)];
 if (image) {
 return [image isKindOfClass:[NSNull class]]? nil: image;
 }
 //set placeholder to avoid reloading image multiple times
 [cache setObject:[NSNull null] forKey:@(index)];
 //switch to background thread
 dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT), ^{
 //load image
 NSString *imagePath = self.imagePaths[index];
 UIImage *image = [UIImage imageWithContentsOfFile:imagePath];
 //redraw image using device context
 UIGraphicsBeginImageContextWithOptions(image.size, YES, 0);
 [image drawAtPoint:CGPointZero];
 image = UIGraphicsGetImageFromCurrentImageContext();
 UIGraphicsEndImageContext();
 //set image for correct image view
 dispatch_async(dispatch_get_main_queue(), ^{
 //cache the image
 [cache setObject:image forKey:@(index)];
 //display the image
 NSIndexPath *indexPath = [NSIndexPath indexPathForItem:index, [self.collectionView numberOfItemsInSection]-1];
 UIImageView *imageView = [cell.contentView.subviews lastObject];
 imageView.image = image;
 });
 });
 //not loaded yet
 return nil;
}

- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
 UICollectionViewCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"Cell" forIndexPath:indexPath];
 cell.imageView.image = [self loadImageAtIndex:indexPath.item];
 return cell;
}

```

```
{
 //dequeue cell
 UICollectionViewCell *cell = [collectionView dequeueReusableCellWithReuseIdentifier:@"ImageCell" forIndexPath:indexPath];
 //add image view
 UIImageView *imageView = [cell.contentView.subviews lastObject];
 if (!imageView) {
 imageView = [[UIImageView alloc] initWithFrame:cell.contentView.bounds];
 imageView.contentMode = UIViewContentModeScaleAspectFit;
 [cell.contentView addSubview:imageView];
 }
 //set or load image for this index
 imageView.image = [self loadImageAtIndex:indexPath.item];
 //preload image for previous and next index
 if (indexPath.item < [self.imagePaths count] - 1) {
 [self loadImageAtIndex:indexPath.item + 1]; }
 if (indexPath.item > 0) {
 [self loadImageAtIndex:indexPath.item - 1]; }
 return cell;
}

@end
```



果然效果更好了！当滚动的时候虽然还有一些图片进入的延迟，但是已经非常罕见了。缓存意味着我们做了更少的加载。这里提前加载逻辑非常粗暴，其实可以把滑动速度和方向也考虑进来，但这已经比之前没做缓存的版本好很多了。

## 文件格式

图片加载性能取决于加载大图的时间和解压小图时间的权衡。很多苹果的文档都说PNG是iOS所有图片加载的最好格式。但这是极度误导的过时信息了。

PNG图片使用的无损压缩算法可以比使用JPEG的图片做到更快地解压，但是由于闪存访问的原因，这些加载的时间并没有什么区别。

清单14.6展示了标准的应用程序加载不同尺寸图片所需要时间的一些代码。为了保证实验的准确性，我们会测量每张图片的加载和绘制时间来确保考虑到解压性能的因素。另外每隔一秒重复加载和绘制图片，这样就可以取到平均时间，使得结果更加准确。

### 清单14.6

```
#import "ViewController.h"

static NSString *const ImageFolder = @"Coast Photos";

@interface ViewController : UIViewController

@property (nonatomic, copy) NSArray *items;
@property (nonatomic, weak) IBOutlet UITableView *tableView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //set up image names
 self.items = @[@"2048x1536", @"1024x768", @"512x384", @"256x192"];
}

- (CFTimeInterval)loadImageForOneSec:(NSString *)path
{
 //create drawing context to use for decompression
```

```

UIGraphicsBeginImageContext(CGSizeMake(1, 1));
//start timing
NSInteger imagesLoaded = 0;
CFTimeInterval endTime = 0;
CFTimeInterval startTime = CFAbsoluteTimeGetCurrent();
while (endTime - startTime < 1) {
 //load image
 UIImage *image = [UIImage imageWithContentsOfFile:path];
 //decompress image by drawing it
 [image drawAtPoint:CGPointZero];
 //update totals
 imagesLoaded++;
 endTime = CFAbsoluteTimeGetCurrent();
}
//close context
UIGraphicsEndImageContext();
//calculate time per image
return (endTime - startTime) / imagesLoaded;
}

- (void)loadImageAtIndex:(NSUInteger)index
{
 //load on background thread so as not to
 //prevent the UI from updating between runs dispatch_async(
 dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
 //setup
 NSString *fileName = self.items[index];
 NSString *pngPath = [[NSBundle mainBundle] pathForResource:
 fileName
 ofType:@"png"
 inDirectory:@""];
 NSString *jpgPath = [[NSBundle mainBundle] pathForResource:
 fileName
 ofType:@"jpg"
 inDirectory:@""];
 //load
 NSInteger pngTime = [self loadImageForOneSec:pngPath] * 100;
 NSInteger jpgTime = [self loadImageForOneSec:jpgPath] * 100;
 //updated UI on main thread
 dispatch_async(dispatch_get_main_queue(), ^{
 //find table cell and update
 NSIndexPath *indexPath = [NSIndexPath indexPathForRow:index
 inSection:0];
 NSIndexPath *oldIndexPath = [NSIndexPath indexPathForRow:indexPath.row
 inSection:0];
 [self.tableView beginUpdates];
 [self.tableView moveRowAtIndexPath:indexPath toIndexPath:oldIndexPath];
 [self.tableView endUpdates];
 });
 });
}

```

```
UITableViewController *cell = [self.tableView cellForRowAtIndexPath:indexPath];
cell.detailTextLabel.text = [NSString stringWithFormat:@"%@", [self.items objectAtIndex:indexPath.row]];
}
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section
{
 return [self.items count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
 //dequeue cell
 UITableViewCell *cell = [self.tableView dequeueReusableCellWithIdentifier:@"ImageCell"];
 if (!cell) {
 cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"ImageCell"];
 }
 //set up cell
 NSString *imageName = self.items[indexPath.row];
 cell.textLabel.text = imageName;
 cell.detailTextLabel.text = @"Loading...";
 //load image
 [self loadImageAtIndex:indexPath.row];
 return cell;
}

@end
```



PNG和JPEG压缩算法作用于两种不同的图片类型：JPEG对于噪点大的图片效果很好；但是PNG更适合于扁平颜色，锋利的线条或者一些渐变色的图片。为了让测评的基准更加公平，我们用一些不同的图片来做实验：一张照片和一张彩虹色的渐变。JPEG版本的图片都用默认的Photoshop60%“高质量”设置编码。结果见图片14.5。

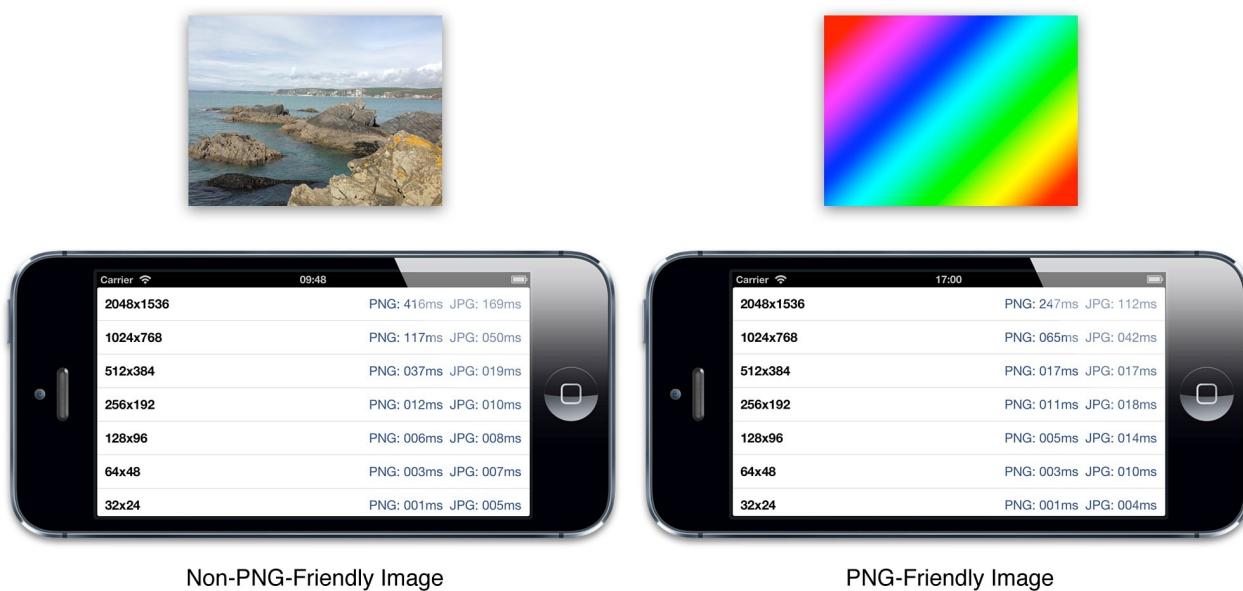


图14.5 不同类型图片的相对加载性能

如结果所示，相对于不友好的PNG图片，相同像素的JPEG图片总是比PNG加载更快，除非一些非常小的图片。但对于友好的PNG图片，一些中大尺寸的图效果还是很好的。

所以对于之前的图片传送器程序来说，JPEG会是个不错的选择。如果用JPEG的话，一些多线程和缓存策略都没必要了。

但JPEG图片并不是所有情况都适用。如果图片需要一些透明效果，或者压缩之后细节损耗很多，那就该考虑用别的格式了。苹果在iOS系统中对PNG和JPEG都做了一些优化，所以普通情况下都应该用这种格式。也就是说在一些特殊的情况下才应该使用别的格式。

## 混合图片

对于包含透明的图片来说，最好是使用压缩透明通道的PNG图片和压缩RGB部分的JPEG图片混合起来加载。这就对任何格式都适用了，而且无论从质量还是文件尺寸还是加载性能来说都和PNG和JPEG的图片相近。相关分别加载颜色和遮罩图片并在运行时合成的代码见14.7。

清单14.7 从PNG遮罩和JPEG创建的混合图片

```

#import "ViewController.h"

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIImageView *imageView;

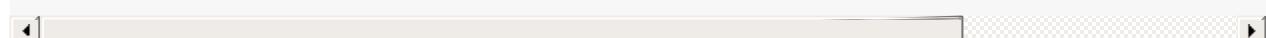
@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //load color image
 UIImage *image = [UIImage imageNamed:@"Snowman.jpg"];
 //load mask image
 UIImage *mask = [UIImage imageNamed:@"SnowmanMask.png"];
 //convert mask to correct format
 CGColorSpaceRef graySpace = CGColorSpaceCreateDeviceGray();
 CGImageRef maskRef = CGImageCreateCopyWithColorSpace(mask.CGImage,
 CGColorSpaceRelease(graySpace));
 //combine images
 CGImageRef resultRef = CGImageCreateWithMask(image.CGImage, maskRef);
 UIImage *result = [UIImage imageWithCGImage:resultRef];
 CGImageRelease(resultRef);
 CGImageRelease(maskRef);
 //display result
 self.imageView.image = result;
}

@end

```



对每张图片都使用两个独立的文件确实有些累赘。**JPNG**的库（<https://github.com/nicklockwood/JPNG>）对这个技术提供了一个开源的可以复用的实现，并且添加了直接使用 `+imageNamed:` 和 `+imageWithContentsOfFile:` 方法的支持。

## JPEG 2000

除了JPEG和PNG之外iOS还支持别的一些格式，例如TIFF和GIF，但是由于他们质量压缩得更厉害，性能比JPEG和PNG糟糕的多，所以大多数情况并不用考虑。

但是iOS之后，苹果低调添加了对JPEG 2000图片格式的支持，所以大多数人并不知道。它甚至并不被Xcode很好的支持 - JPEG 2000图片都没在Interface Builder中显示。

但是JPEG 2000图片在（设备和模拟器）运行时会有效，而且比JPEG质量更好，同样也对透明通道有很好的支持。但是JPEG 2000图片在加载和显示图片方面明显要比PNG和JPEG慢得多，所以对图片大小比运行效率更敏感的时候，使用它是一个不错的选择。

但仍然要对JPEG 2000保持关注，因为在后续iOS版本说不定就对它的性能做提升，但是在现阶段，混合图片对更小尺寸和质量的文件性能会更好。

## PVRTC

当前市场的每个iOS设备都使用了Imagination Technologies PowerVR图像芯片作为GPU。PowerVR芯片支持一种叫做PVRTC（PowerVR Texture Compression）的标准图片压缩。

和iOS上可用的大多数图片格式不同，PVRTC不用提前解压就可以被直接绘制到屏幕上。这意味着在加载图片之后不需要有解压操作，所以内存中的图片比其他图片格式大大减少了（这取决于压缩设置，大概只有1/60那么大）。

但是PVRTC仍然有一些弊端：

- 尽管加载的时候消耗了更少的RAM，PVRTC文件比JPEG要大，有时候甚至比PNG还要大（这取决于具体内容），因为压缩算法是针对于性能，而不是文件尺寸。
- PVRTC必须要是二维正方形，如果源图片不满足这些要求，那必须要在转换成PVRTC的时候强制拉伸或者填充空白空间。
- 质量并不是很好，尤其是透明图片。通常看起来更像严重压缩的JPEG文件。

- PVRTC不能用Core Graphics绘制，也不能在普通的 UIImageView 显示，也不能直接用作图层的内容。你必须要用作OpenGL纹理加载PVRTC图片，然后映射到一对三角板来在 CAEAGLLayer 或者 GLKView 中显示。
- 创建一个OpenGL纹理来绘制PVRTC图片的开销相当昂贵。除非你想把所有图片绘制到一个相同的上下文，不然这完全不能发挥PVRTC的优势。
- PVRTC使用了一个不对称的压缩算法。尽管它几乎立即解压，但是压缩过程相当漫长。在一个现代快速的桌面Mac电脑上，它甚至要消耗一分钟甚至更多来生成一个PVRTC大图。因此在iOS设备上最好不要实时生成。

如果你愿意使用OpenGL，而且即使提前生成图片也能忍受得了，那么PVRTC将会提供相对于别的可用格式来说非常高效的加载性能。比如，可以在主线程1/60秒之内加载并显示一张2048×2048的PVRTC图片（这已经足够大来填充一个视网膜屏幕的iPad了），这就避免了很多使用线程或者缓存等等复杂的技术难度。

Xcode包含了一些命令行工具例如*texturetool*来生成PVRTC图片，但是用起来很不方便（它存在于Xcode应用程序束中），而且很受限制。一个更好的方案就是使用Imagination Technologies *PVRTexTool*，可以从<http://www.imgtec.com/powervr/insider/sdkdownloads>免费获得。

安装了PVRTexTool之后，就可以使用如下命令在终端中把一个合适大小的PNG图片转换成PVRTC文件：

```
/Applications/Imagination/PowerVR/GraphicsSDK/PVRTexTool/CL/OSX_x86
```

清单14.8的代码展示了加载和显示PVRTC图片的步骤（第6章 CAEAGLLayer 例子代码改动而来）。

#### 清单14.8 加载和显示PVRTC图片

```
#import "ViewController.h"
#import
#import

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *glView;
@property (nonatomic, strong) EAGLContext *glContext;
```

```
@property (nonatomic, strong) CAEAGLLayer *glLayer;
@property (nonatomic, assign) GLuint framebuffer;
@property (nonatomic, assign) GLuint colorRenderbuffer;
@property (nonatomic, assign) GLint framebufferWidth;
@property (nonatomic, assign) GLint framebufferHeight;
@property (nonatomic, strong) GLKBaseEffect *effect;
@property (nonatomic, strong) GLKTextureInfo *textureInfo;

@end

@implementation ViewController

- (void)setUpBuffers
{
 //set up frame buffer
 glGenFramebuffers(1, &_framebuffer);
 glBindFramebuffer(GL_FRAMEBUFFER, _framebuffer);
 //set up color render buffer
 glGenRenderbuffers(1, &_colorRenderbuffer);
 glBindRenderbuffer(GL_RENDERBUFFER, _colorRenderbuffer);
 glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
 [self.glContext renderbufferStorage:GL_RENDERBUFFER fromDrawable:_colorRenderbuffer];
 glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_WIDTH, &framebufferWidth);
 glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_HEIGHT, &framebufferHeight);
 //check success
 if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
 NSLog(@"Failed to make complete framebuffer object: %i", glCheckFramebufferStatus(GL_FRAMEBUFFER));
}

- (void)tearDownBuffers
{
 if (_framebuffer) {
 //delete framebuffer
 glDeleteFramebuffers(1, &_framebuffer);
 _framebuffer = 0;
 }
 if (_colorRenderbuffer) {
 //delete color render buffer
 glDeleteRenderbuffers(1, &_colorRenderbuffer);
 }
}
```

```
_colorRenderbuffer = 0;
}

{
 - (void)drawFrame
{
 //bind framebuffer & set viewport
 glBindFramebuffer(GL_FRAMEBUFFER, _framebuffer);
 glViewport(0, 0, _framebufferWidth, _framebufferHeight);
 //bind shader program
 [self.effect prepareToDraw];
 //clear the screen
 glClear(GL_COLOR_BUFFER_BIT);
 glClearColor(0.0, 0.0, 0.0, 0.0);
 //set up vertices
 GLfloat vertices[] = {
 -1.0f, -1.0f, -1.0f, 1.0f, 1.0f, 1.0f, 1.0f, -1.0f
 };
 //set up colors
 GLfloat texCoords[] = {
 0.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f
 };
 //draw triangle
 glEnableVertexAttribArray(GLKVertexAttribPosition);
 glEnableVertexAttribArray(GLKVertexAttribTexCoord0);
 glVertexAttribPointer(GLKVertexAttribPosition, 2, GL_FLOAT, GL_
 glVertexAttribPointer(GLKVertexAttribTexCoord0, 2, GL_FLOAT, GI
 glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
 //present render buffer
 glBindRenderbuffer(GL_RENDERBUFFER, _colorRenderbuffer);
 [self.glContext presentRenderbuffer:GL_RENDERBUFFER];
}

- (void)viewDidLoad
{
 [super viewDidLoad];
 //set up context
 self.glContext = [[EAGLContext alloc] initWithAPI:kEAGLRendering
 [EAGLContext setCurrentContext:self.glContext];
 //set up layer
```

```
 self.glLayer = [CAEAGLLayer layer];
 self.glLayer.frame = self.glView.bounds;
 self.glLayer.opaque = NO;
 [self.glView.layer addSublayer:self.glLayer];
 self.glLayer.drawableProperties = @{@"kEAGLDrawablePropertyRetain"
//load texture
 glActiveTexture(GL_TEXTURE0);
 NSString *imageFile = [[NSBundle mainBundle] pathForResource:@""
self.textureInfo = [GLKTextureLoader textureWithContentsOfFile:@"
//create texture
 GLKEffectPropertyTexture *texture = [[GLKEffectPropertyTexture alloc] init];
 texture.enabled = YES;
 texture.envMode = GLKTextureEnvModeDecal;
 texture.name = self.textureInfo.name;
//set up base effect
 self.effect = [[GLKBaseEffect alloc] init];
 self.effect.texture2d0.name = texture.name;
//set up buffers
 [self setUpBuffers];
//draw frame
 [self drawFrame];
}

- (void)viewDidUnload
{
 [self tearDownBuffers];
 [super viewDidUnload];
}

- (void)dealloc
{
 [self tearDownBuffers];
 [EAGLContext setCurrentContext:nil];
}

@end
```



如你所见，非常不容易，如果你对在常规应用中使用PVRTC图片很感兴趣的话（例如基于OpenGL的游戏），可以参考一下 `GLView` 的库（<https://github.com/nicklockwood/GLView>），它提供了一个简单的 `GLImageView` 类，重新实现了 `UIImageView` 的各种功能，但同时提供了PVRTC图片，而不需要你写任何OpenGL代码。

## 总结

在这章中，我们研究了和图片加载解压相关的性能问题，并延展了一系列解决方案。

在第15章“图层性能”中，我们将讨论和图层渲染和组合相关的性能问题。

## 图层性能

要更快性能，也要做对正确的事情。——Stephen R. Covey

在第14章『图像IO』讨论如何高效地载入和显示图像，通过视图来避免可能引起动画帧率下降的性能问题。在最后一章，我们将着重图层树本身，以发掘最好的性能。

## 隐式绘制

寄宿图可以通过Core Graphics直接绘制，也可以直接载入一个图片文件并赋值给`contents`属性，或事先绘制一个屏幕之外的`CGContext`上下文。在之前的两章中我们讨论了这些场景下的优化。但是除了常见的显式创建寄宿图，你也可以通过以下三种方式创建隐式的：1，使用特性的图层属性。2，特定的视图。3，特定的图层子类。

了解这个情况为什么发生何时发生是很重要的，它能够让你避免引入不必要的软件绘制行为。

## 文本

`CATextLayer` 和 `UILabel` 都是直接将文本绘制在图层的寄宿图中。事实上这两种方式用了完全不同的渲染方式：在iOS 6及之前，`UILabel` 用WebKit的HTML渲染引擎来绘制文本，而 `CATextLayer` 用的是Core Text.后者渲染更迅速，所以在所有需要绘制大量文本的情形下都优先使用它吧。但是这两种方法都用了软件的方式绘制，因此他们实际上要比硬件加速合成方式要慢。

不论如何，尽可能地避免改变那些包含文本的视图的`frame`，因为这样做的话文本就需要重绘。例如，如果你想在图层的角落里显示一段静态的文本，但是这个图层经常改动，你就应该把文本放在一个子图层中。

## 光栅化

在第四章『视觉效果』中我们提到了`CALayer` 的`shouldRasterize` 属性，它可以解决重叠透明图层的混合失灵问题。同样在第12章『速度的曲调』中，它也是作为绘制复杂图层树结构的优化方法。

启用`shouldRasterize` 属性会将图层绘制到一个屏幕之外的图像。然后这个图像将会被缓存起来并绘制到实际图层的`contents` 和子图层。如果有太多的子图层或者有复杂的效果应用，这样做就会比重绘所有事务的所有帧划得来得多。但是光栅化原始图像需要时间，而且还会消耗额外的内存。

当我们使用得当时，光栅化可以提供很大的性能优势（如你在第12章所见），但是一定要避免作用在内容不断变动的图层上，否则它缓存方面的好处就会消失，而且会让性能变的更糟。

为了检测你是否正确地使用了光栅化方式，用Instrument查看一下**Color Hits Green**和**Misses Red**项目，是否已光栅化图像被频繁地刷新（这样就说明图层并不是光栅化的好选择，或则你无意间触发了不必要的改变导致了重绘行为）。

## 离屏渲染

当图层属性的混合体被指定为在未预合成之前不能直接在屏幕上绘制时，屏幕外渲染就被唤起了。屏幕外渲染并不意味着软件绘制，但是它意味着图层必须在被显示之前在一个屏幕外上下文中被渲染（不论CPU还是GPU）。图层的以下属性将会触发屏幕外绘制：

- 圆角（当和 `maskToBounds` 一起使用时）
- 图层蒙板
- 阴影

屏幕外渲染和我们启用光栅化时相似，除了它并没有像光栅化图层那么消耗大，子图层并没有被影响到，而且结果也没有被缓存，所以不会有长期的内存占用。但是，如果太多图层在屏幕外渲染依然会影响到性能。

有时候我们可以把那些需要屏幕外绘制的图层开启光栅化以作为一个优化方式，前提是这些图层并不会被频繁地重绘。

对于那些需要动画而且要在屏幕外渲染的图层来说，你可以用 `CAShapeLayer`，`contentsCenter` 或者 `shadowPath` 来获得同样的表现而且较少地影响到性能。

## CAShapeLayer

`cornerRadius` 和 `maskToBounds` 独立作用的时候都不会有太大的性能问题，但是当他俩结合在一起，就触发了屏幕外渲染。有时候你想显示圆角并沿着图层裁切子图层的时候，你可能会发现你并不需要沿着圆角裁切，这个情况下用 `CAShapeLayer` 就可以避免这个问题了。

你想要的只是圆角且沿着矩形边界裁切，同时还不希望引起性能问题。其实你可以用现成的 `UIBezierPath` 的构造器 `+bezierPathWithRoundedRect:cornerRadius:`（见清单15.1）。这样做并不会比直接用 `cornerRadius` 更快，但是它避免了性能问题。

清单15.1 用 `CAShapeLayer` 画一个圆角矩形

```
#import "ViewController.h"
#import

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIView *layerView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //create shape layer
 CAShapeLayer *blueLayer = [CAShapeLayer layer];
 blueLayer.frame = CGRectMake(50, 50, 100, 100);
 blueLayer.fillColor = [UIColor blueColor].CGColor;
 blueLayer.path = [UIBezierPath bezierPathWithRoundedRect:
 CGRectMake(0, 0, 100, 100) cornerRadius:20].CGPath;

 //add it to our view
 [self.layerView.layer addSublayer:blueLayer];
}

@end
```

## 可伸缩图片

另一个创建圆角矩形的方法就是用一个圆形内容图片并结合第二章『寄宿图』提到的 `contentsCenter` 属性去创建一个可伸缩图片（见清单15.2）。理论上来说，这个应该比用 `CAShapeLayer` 要快，因为一个可拉伸图片只需要18个三角形（一个图片是由一个 $3 \times 3$ 网格渲染而成），然而，许多都需要渲染成一个顺滑的曲线。在实际应用上，二者并没有太大的区别。

### 清单15.2 用可伸缩图片绘制圆角矩形

```
@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //create layer
 CALayer *blueLayer = [CALayer layer];
 blueLayer.frame = CGRectMake(50, 50, 100, 100);
 blueLayer.contentsCenter = CGRectMake(0.5, 0.5, 0.0, 0.0);
 blueLayer.contentsScale = [UIScreen mainScreen].scale;
 blueLayer.contents = (__bridge id)[UIImage imageNamed:@"Circle"];
 //add it to our view
 [self.layerView.layer addSublayer:blueLayer];
}

@end
```

使用可伸缩图片的优势在于它可以绘制成任意边框效果而不需要额外的性能消耗。举个例子，可伸缩图片甚至还可以显示出矩形阴影的效果。

## shadowPath

在第2章我们有提到 `shadowPath` 属性。如果图层是一个简单几何图形如矩形或者圆角矩形（假设不包含任何透明部分或者子图层），创建出一个对应形状的阴影路径就比较容易，而且Core Animation绘制这个阴影也相当简单，避免了屏幕外的图层部分的预排版需求。这对性能来说很有帮助。

如果你的图层是一个更复杂的图形，生成正确的阴影路径可能就比较难了，这样子的话你可以考虑用绘图软件预先生成一个阴影背景图。

## 混合和过度绘制

在第12章有提到，GPU每一帧可以绘制的像素有一个最大限制（就是所谓的fill rate），这个情况下可以轻易地绘制整个屏幕的所有像素。但是如果由于重叠图层的关系需要不停地重绘同一区域的话，掉帧就可能发生了。

GPU会放弃绘制那些完全被其他图层遮挡的像素，但是要计算出一个图层是否被遮挡也是相当复杂并且会消耗处理器资源。同样，合并不同图层的透明重叠像素（即混合）消耗的资源也是相当可观的。所以为了加速处理进程，不到必须时刻不要使用透明图层。任何情况下，你应该这样做：

- 给视图的 `backgroundColor` 属性设置一个固定的，不透明的颜色
- 设置 `opaque` 属性为YES

这样做减少了混合行为（因为编译器知道在图层之后的东西都不会对最终的像素颜色产生影响）并且计算得到了加速，避免了过度绘制行为因为Core Animation可以舍弃所有被完全遮盖住的图层，而不用每个像素都去计算一遍。

如果用到了图像，尽量避免透明除非非常必要。如果图像要显示在一个固定的背景颜色或是固定的背景图之前，你没必要相对前景移动，你只需要预填充背景图片就可以避免运行时混色了。

如果是文本的话，一个白色背景的 `UILabel` （或者其他颜色）会比透明背景要更高效。

最后，明智地使用 `shouldRasterize` 属性，可以将一个固定的图层体系折叠成单张图片，这样就不需要每一帧重新合成了，也就不会有因为子图层之间的混合和过度绘制的性能问题了。

## 减少图层数量

初始化图层，处理图层，打包通过IPC发给渲染引擎，转化成OpenGL几何图形，这些是一个图层的大致资源开销。事实上，一次性能够在屏幕上显示的最大图层数量也是有限的。

确切的限制数量取决于iOS设备，图层类型，图层内容和属性等。但是总得说来可以容纳上百或上千个，下面我们将演示即使图层本身并没有做什么也会遇到的性能问题。

## 裁切

在对图层做任何优化之前，你需要确定你不是在创建一些不可见的图层，图层在以下几种情况下回事不可见的：

- 图层在屏幕边界之外，或是在父图层边界之外。
- 完全在一个不透明图层之后。
- 完全透明

Core Animation非常擅长处理对视觉效果无意义的图层。但是经常性地，你自己的代码会比Core Animation更早地想知道一个图层是否是有用的。理想状况下，在图层对象在创建之前就想知道，以避免创建和配置不必要的额外工作。

举个例子。清单15.3的代码展示了一个简单的滚动3D图层矩阵。这看上去很酷，尤其是图层在移动的时候（见图15.1），但是绘制他们并不是很麻烦，因为这些图层就是一些简单的矩形色块。

### 清单15.3 绘制3D图层矩阵

```
#import "ViewController.h"
#import

#define WIDTH 10
#define HEIGHT 10
#define DEPTH 10
#define SIZE 100
#define SPACING 150
#define CAMERA_DISTANCE 500
```

```
@interface ViewController ()

@property (nonatomic, strong) IBOutlet UIScrollView *scrollView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];

 //set content size
 self.scrollView.contentSize = CGSizeMake((WIDTH - 1)*SPACING, (HEIGHT - 1)*SPACING);

 //set up perspective transform
 CATransform3D transform = CATransform3DIdentity;
 transform.m34 = -1.0 / CAMERA_DISTANCE;
 self.scrollView.layer.sublayerTransform = transform;

 //create layers
 for (int z = DEPTH - 1; z >= 0; z--) {
 for (int y = 0; y < HEIGHT; y++) {
 for (int x = 0; x < WIDTH; x++) {
 //create layer
 CALayer *layer = [CALayer layer];
 layer.frame = CGRectMake(0, 0, SIZE, SIZE);
 layer.position = CGPointMake(x*SPACING, y*SPACING);
 layer.zPosition = -z*SPACING;
 //set background color
 layer.backgroundColor = [UIColor colorWithRed:(x+1)/DEPTH green:(y+1)/HEIGHT blue:(z+1)/DEPTH alpha:1.0];
 //attach to scroll view
 [self.scrollView.layer addSublayer:layer];
 }
 }
 }

 //log
 NSLog(@"displayed: %i", DEPTH*HEIGHT*WIDTH);
}
```

```
}
```

```
@end
```



图15.1 滚动的3D图层矩阵

`WIDTH`，`HEIGHT` 和 `DEPTH` 常量控制着图层的生成。在这个情况下，我们得到的是 $10 \times 10 \times 10$ 个图层，总量为1000个，不过一次性显示在屏幕上的大约就几百个。

如果把 `WIDTH` 和 `HEIGHT` 常量增加到100，我们的程序就会慢得像龟爬了。这样我们有了100000个图层，性能下降一点儿也不奇怪。

但是显示在屏幕上的图层数量并没有增加，那么根本没有额外的东西需要绘制。程序慢下来的原因其实是因为在管理这些图层上花掉了不少功夫。他们大部分对渲染的最终结果没有贡献，但是在丢弃这么图层之前，Core Animation要强制计算每个图层的位置，就这样，我们的帧率就慢了下来。

我们的图层是被安排在一个均匀的栅格中，我们可以计算出哪些图层会被最终显示在屏幕上，根本不需要对每个图层的位置进行计算。这个计算并不简单，因为我们还要考虑到透视的问题。如果我们直接这样做了，Core Animation就不用费神了。

既然这样，让我们来重构我们的代码吧。改造后，随着视图的滚动动态地实例化图层而不是事先都分配好。这样，在创造他们之前，我们就可以计算出是否需要他。接着，我们增加一些代码去计算可视区域这样就可以排除区域之外的图层了。清单15.4是改造后的结果。

## 清单15.4 排除可视区域之外的图层

```
#import "ViewController.h"
#import

#define WIDTH 100
#define HEIGHT 100
#define DEPTH 10
#define SIZE 100
#define SPACING 150
#define CAMERA_DISTANCE 500
#define PERSPECTIVE(z) (float)CAMERA_DISTANCE/(z + CAMERA_DISTANCE)

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIScrollView *scrollView;

@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad];
 //set content size
 self.scrollView.contentSize = CGSizeMake((WIDTH - 1)*SPACING, 0);
 //set up perspective transform
 CATransform3D transform = CATransform3DIdentity;
 transform.m34 = -1.0 / CAMERA_DISTANCE;
 self.scrollView.layer.sublayerTransform = transform;
}

- (void)viewDidLayoutSubviews
{
 [self updateLayers];
}

- (void)scrollViewDidScroll:(UIScrollView *)scrollView
{
 [self updateLayers];
}
```

```
}

- (void)updateLayers
{
 //calculate clipping bounds
 CGRect bounds = self.scrollView.bounds;
 bounds.origin = self.scrollView.contentOffset;
 bounds = CGRectInset(bounds, -SIZE/2, -SIZE/2);
 //create layers
 NSMutableArray *visibleLayers = [NSMutableArray array];
 for (int z = DEPTH - 1; z >= 0; z--)
 {
 //increase bounds size to compensate for perspective
 CGRect adjusted = bounds;
 adjusted.size.width /= PERSPECTIVE(z*SPACING);
 adjusted.size.height /= PERSPECTIVE(z*SPACING);
 adjusted.origin.x -= (adjusted.size.width - bounds.size.width) * z * SPACING;
 adjusted.origin.y -= (adjusted.size.height - bounds.size.height) * z * SPACING;
 for (int y = 0; y < HEIGHT; y++) {
 //check if vertically outside visible rect
 if (y*SPACING < adjusted.origin.y || y*SPACING >= adjusted.size.height)
 {
 continue;
 }
 for (int x = 0; x < WIDTH; x++) {
 //check if horizontally outside visible rect
 if (x*SPACING < adjusted.origin.x || x*SPACING >= adjusted.size.width)
 {
 continue;
 }

 //create layer
 CALayer *layer = [CALayer layer];
 layer.frame = CGRectMake(0, 0, SIZE, SIZE);
 layer.position = CGPointMake(x*SPACING, y*SPACING);
 layer.zPosition = -z*SPACING;
 //set background color
 layer.backgroundColor = [UIColor colorWithRed:(float)rand()/(float)RAND_MAX green:(float)rand()/(float)RAND_MAX blue:(float)rand()/(float)RAND_MAX alpha:1.0];
 //attach to scroll view
 [visibleLayers addObject:layer];
 }
 }
 }
}
```

```

 }
 }

}

//update layers
self.scrollView.layer.sublayers = visibleLayers;
//log
NSLog(@"displayed: %i/%i", [visibleLayers count], DEPTH*HEIGHT);
}
@end

```

这个计算机制并不具有普适性，但是原则上是一样。（当你用一个 `UITableView` 或者 `UICollectionView` 时，系统做了类似的事情）。这样做的结果？我们的程序可以处理成百上千个『虚拟』图层而且完全没有性能问题！因为它不需要一次性实例化几百个图层。

## 对象回收

处理巨大数量的相似视图或图层时还有一个技巧就是回收他们。对象回收在iOS颇为常见； `UITableView` 和 `UICollectionView` 都有用到， `MKMapView` 中的动画pin码也有用到，还有其他很多例子。

对象回收的基础原则就是你需要创建一个相似对象池。当一个对象的指定实例（本例子中指的是图层）结束了使命，你把它添加到对象池中。每次当你需要一个实例时，你就从池中取出一个。当且仅当池中为空时再创建一个新的。

这样做的好处在于避免了不断创建和释放对象（相当消耗资源，因为涉及到内存的分配和销毁）而且也不必给相似实例重复赋值。

好了，让我们再次更新代码吧（见清单15.5）

**清单15.5** 通过回收减少不必要的分配

```

@interface ViewController : UIViewController

@property (nonatomic, weak) IBOutlet UIScrollView *scrollView;
@property (nonatomic, strong) NSMutableSet *recyclePool;

```

```
@end

@implementation ViewController

- (void)viewDidLoad
{
 [super viewDidLoad]; //create recycle pool
 self.recyclePool = [NSMutableSet set];
 //set content size
 self.scrollView.contentSize = CGSizeMake((WIDTH - 1)*SPACING, (HEIGHT - 1)*SPACING);
 //set up perspective transform
 CATransform3D transform = CATransform3DIdentity;
 transform.m34 = -1.0 / CAMERA_DISTANCE;
 self.scrollView.layer.sublayerTransform = transform;
}

- (void)viewDidLayoutSubviews
{
 [self updateLayers];
}

- (void)scrollViewDidScroll:(UIScrollView *)scrollView
{
 [self updateLayers];
}

- (void)updateLayers {
 //calculate clipping bounds
 CGRect bounds = self.scrollView.bounds;
 bounds.origin = self.scrollView.contentOffset;
 bounds = CGRectMakeInset(bounds, -SIZE/2, -SIZE/2);
 //add existing layers to pool
 [self.recyclePool addObjectsFromArray:self.scrollView.layer.sublayers];
 //disable animation
 [CATransaction begin];
 [CATransaction setDisableActions:YES];
 //create layers
 NSInteger recycled = 0;
 NSMutableArray *visibleLayers = [NSMutableArray array];
}
```

```

for (int z = DEPTH - 1; z >= 0; z--) {
 //increase bounds size to compensate for perspective
 CGRect adjusted = bounds;
 adjusted.size.width /= PERSPECTIVE(z*SPACING);
 adjusted.size.height /= PERSPECTIVE(z*SPACING);
 adjusted.origin.x -= (adjusted.size.width - bounds.size.width) * z * SPACING;
 for (int y = 0; y < HEIGHT; y++) {
 //check if vertically outside visible rect
 if (y*SPACING < adjusted.origin.y ||
 y*SPACING >= adjusted.origin.y + adjusted.size.height)
 {
 continue;
 }
 for (int x = 0; x < WIDTH; x++) {
 //check if horizontally outside visible rect
 if (x*SPACING < adjusted.origin.x ||
 x*SPACING >= adjusted.origin.x + adjusted.size.width)
 {
 continue;
 }
 //recycle layer if available
 CALayer *layer = [self.recyclePool anyObject]; if (layer)
 {

 recycled++;
 [self.recyclePool removeObject:layer];
 }
 else
 {
 layer = [CALayer layer];
 layer.frame = CGRectMake(0, 0, SIZE, SIZE);
 }
 //set position
 layer.position = CGPointMake(x*SPACING, y*SPACING);
 //set background color
 layer.backgroundColor =
 [UIColor colorWithRed:(1.0-z*(1.0/DEPTH)) green:0 blue:0 alpha:1].CGColor;
 //attach to scroll view
 [visibleLayers addObject:layer];
 }
 }
 [CATransaction commit]; //update layers
}

```

```

 self.scrollView.layer.sublayers = visibleLayers;
 //log
 NSLog(@"displayed: %i/%i recycled: %i",
 [visibleLayers count], DEPTH*HEIGHT*WIDTH, recycled);
}
@end

```

本例中，我们只有图层对象这一种类型，但是UIKit有时候用一个标识符字符串来区分存储在不同对象池中的不同的可回收对象类型。

你可能注意到当设置图层属性时我们用了一个 `CATransaction` 来抑制动画效果。在此之前并不需要这样做，因为在显示之前我们给所有图层设置一次属性。但是既然图层正在被回收，禁止隐式动画就有必要了，不然当属性值改变时，图层的隐式动画就会被触发。

## Core Graphics绘制

当排除掉对屏幕显示没有任何贡献的图层或者视图之后，长远看来，你可能仍然需要减少图层的数量。例如，如果你正在使用多个 `UILabel` 或者 `UIImageView` 实例去显示固定内容，你可以把他们全部替换成一个单独的视图，然后用 `-drawRect:` 方法绘制出那些复杂的视图层级。

这个提议看上去并不合理因为大家都知道软件绘制行为要比GPU合成要慢而且还需要更多的内存空间，但是在因为图层数量而使得性能受限的情况下，软件绘制很可能提高性能呢，因为它避免了图层分配和操作问题。

你可以自己实验一下这个情况，它包含了性能和栅格化的权衡，但是意味着你可以从图层树上去掉子图层（用 `shouldRasterize`，与完全遮挡图层相反）。

### **-renderInContext:** 方法

用Core Graphics去绘制一个静态布局有时候会比用层级的 `UIView` 实例来得快，但是使用 `UIView` 实例要简单得多而且比用手写代码写出相同效果要可靠得多，更边说Interface Builder来得直接明了。为了性能而舍弃这些便利实在是不应该。

幸好，你不必这样，如果大量的视图或者图层真的关联到了屏幕上将会是一个大问题。没有与图层树相关联的图层不会被送到渲染引擎，也没有性能问题（在他们被创建和配置之后）。

使用 `CALayer` 的 `-renderInContext:` 方法，你可以将图层及其子图层快照进一个Core Graphics上下文然后得到一个图片，它可以直接显示在 `UIImageView` 中，或者作为另一个图层的 `contents`。不同于 `shouldRasterize` —— 要求图层与图层树相关联 ——，这个方法没有持续的性能消耗。

当图层内容改变时，刷新这张图片的机会取决于你（不同于 `shouldRasterize`，它自动地处理缓存和缓存验证），但是一旦图片被生成，相比于让Core Animation处理一个复杂的图层树，你节省了相当可观的性能。

## 总结

本章学习了使用Core Animation图层可能遇到的性能瓶颈，并讨论了如何避免或减小压力。你学习了如何管理包含上千虚拟图层的场景（事实上只创建了几百个）。同时也学习了一些有用的技巧，选择性地选取光栅化或者绘制图层内容在合适的时候重新分配给CPU和GPU。这些就是我们要讲的关于Core Animation的全部了（至少可以等到苹果发明什么新的玩意儿）。