MOBILE PROGRAMMING SERIES

iOS CORE ANIMATION

ADVANCED TECHNIQUES

NICK LOCKWOOD

iOS Core Animation

iOS Core Animation

Advanced Techniques

Nick Lockwood

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

ISBN-13: 978-0-13-344075-1
ISBN-10: 0-13-344075-3

Editor-in-Chief
Mark Taub

Acquisitions Editor
Trina MacDonald

Angie Doyle

Development Editor
Michael Thurston

Managing Editor
Kristy Hart

Project Editor
Sara Schumacher

Copy Editor
Keith Cline

Proofreader
Sarah Kearns

Technical Reviewer
Mugunth Kumar

Publishing Coordinator
Olivia Basegio

For Zoe

Contents at a Glance

Table of Contents

About the Author

Nick Lockwood is head of iOS development at the digital agency AKQA in London, and a prolific developer of applications and open source libraries. He has been working with the iOS platform for the past four years, after making the switch from HTML5 web-app development. Nick first picked up a programming book in 1993 at a middle school rummage sale and hasn't looked back since. He lives in Sidcup with his wife and daughter.

Acknowledgments

I would like to thank my wife, Angela, and daughter, Zoe, for putting up with seeing even less of me than normal while I wrote this. Thanks to David Deutsch and P.J. Cook for their valuable feedback on the early manuscript. Thanks to Mugunth Kumar for generously agreeing to be my technical editor, and to all the staff at Pearson, particularly Michael Thurston, Sara Schumacher, Angie Doyle, Trina MacDonald, and Olivia Basegio, who between them somehow managed to extract a book from me in the space of four months. Thanks to my parents, for buying me my first programming book at the tender age of 12, and so starting me on the path to my eventual career. Finally, thanks to Kate for all the cups of tea!

Editor's Note: We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can e-mail or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or e-mail address. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail: trina.macdonald@pearson.com
Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education, Inc.
75 Arlington St., Ste. 300
Boston, MA 02116

Reader Services

Visit our website and register this book at **informit.com/register** for convenient access to any updates, downloads, or errata that might be available for this book.

Preface

When Apple engineers created the iPhone, they faced a challenge: they needed to create a modern, fast, and fluid user interface, the likes of which had never been seen outside of a video game, and they needed to do it on mobile hardware that was a decade behind current desktops and laptops in terms of graphics performance.

They also had an opportunity: to rebuild AppKit (the Mac OS user interface framework) from the ground up using modern graphics technology without needing to maintain support for the legacy applications inherited from nearly 25 years of Macintosh and NeXTSTEP history.

Their solution was a private framework called *Layer Kit*, developed by the iPhone software team to provide a high-performance, hardware-accelerated animation and compositing library to replace the slower, Quartz-based software drawing used by AppKit. This framework actually debuted first on Mac OS 10.5 under the name *Core Animation*, shortly before the iPhone was announced.

Core Animation is not just a set of functions for performing animations; it lies at the very heart of iOS, powering everything you see on screen. Even if you never invoke it explicitly, you are using it *implicitly* every time you display a view or transition from one screen to the next. As an iOS developer, you can build great applications without ever consciously touching Core Animation, but if you truly embrace its features, you can achieve much richer user experiences.

The purpose of this book is to demystify Core Animation, to bring it out from behind the curtains and help you to harness its full power to make spectacular applications. By the end, you will have learned exactly where and when to use Core Animation, how to work with and around its limitations, and what to do to avoid performance pitfalls so that your apps can be as responsive as Apple's own.

Audience and Material

This is neither a beginner's guide to Cocoa nor an introduction to the iOS platform for Mac developers. It is *most definitely not* an introduction to programming in general.

This book is written for an audience that is reasonably familiar with Xcode, Cocoa Touch, and UIKit but has no prior knowledge of the Core Animation framework. You need not be a very seasoned developer (perhaps you have just finished your "Hello World" project and are looking for something a bit more advanced), but you will need to be fluent in Objective-C to be able to follow the code examples.

Although no prior knowledge of Core Animation is assumed, this is not merely a cursory introduction to the framework. The aim of this book is to leave no stone unturned when it comes to the Core Animation APIs. Even if you have already used Core Animation for many years in an iOS-specific context, it would be surprising if you did not find things in here that you don't already know about or fully understand.

This book is specifically geared toward the iOS platform. Where appropriate, differences between Core Animation on iOS and Mac OS are mentioned, but Mac-specific Core Animation features such as `CALayoutManager` or Core Image integration or are not discussed in detail. If you are already well versed in Core Animation on Mac OS, much of the material will already be familiar to you, but this should serve as a useful conversion guide.

Core Animation has been a key part of iOS since the beginning, and the majority of its features are available on older iOS versions. Any methods or classes that are new to iOS 6 are highlighted as such, but the purpose of this book is to document the *current* feature set of Core Animation; so with very few exceptions, the task of providing backward compatibility for earlier iOS versions is left as an exercise for the reader.

Book Structure

This book is structured in a linear fashion, with each chapter building upon concepts introduced in the previous ones. That said, wherever we refer back to a concept covered in an earlier chapter, the chapter is referenced explicitly so that you can read the book in a nonlinear fashion if you prefer.

The subject matter is split into three parts, dealing with static content, animation, and performance optimization, respectively. These parts are self-contained, so (for example) if you already know about animation and layout, you can dive straight into the part on performance.

Each chapter contains figures and example code to illustrate the topics discussed. The sample code projects are available for download from www.informit.com/title/9780133440751 if you prefer not to retype them by hand.

Before We Begin

The examples in this book were written and tested using Xcode 4.6 on Mac OS 10.8 (*Mountain Lion*). The latest version of Xcode can be downloaded free of charge from the Mac App Store, and most of the examples can be run in the iOS simulator. In addition, you need to sign up for a free Apple developer account to access most of the tools and documentation for the classes referenced in the book.

The code examples have all been tested with iOS 6.1, but most will either run unmodified on iOS 5+, or can be trivially modified to do so by removing noncritical iOS 6 features such as Autolayout. All examples make use of modern Objective-C practices such as ARC (Automatic Reference Counting), automatic property synthesis, and object literals and so require Xcode 4.5 and iOS 4 as a minimum.

The examples in the final, performance-focused section of the book must be installed on a physical iPhone 5 running iOS 6.1 to demonstrate the exact behavior described in the text. To run the examples on an iPhone, you need to pay for an iOS developer license, which you can purchase directly from Apple. These examples will still work on the simulator, or a different device or iOS version, but will likely exhibit different performance characteristics.

This page intentionally left blank

1 The Layer Tree

Ogres have layers. Onions have layers. You get it? We both have layers.

Shrek

Core Animation is misleadingly named. You might assume that its primary purpose is *animation*, but actually animation is only one facet of the framework, which originally bore the less animation-centric name of *Layer Kit.*

Core Animation is a *compositing engine*; its job is to compose different pieces of visual content on the screen, and to do so as fast as possible. The content in question is divided into individual *layers* stored in a hierarchy known as the *layer tree.* This tree forms the underpinning for all of UIKit, and for everything that you see on the screen in an iOS application.

Before we discuss animation at all, we're going to cover Core Animation's *static* compositing and layout features, starting with the layer tree.

Layers and Views

If you've ever created an app for iOS or Mac OS, you'll be familiar with the concept of a *view.* A view is a rectangular object that displays content (such as images, text, or video), and intercepts user input such as mouse clicks or touch gestures. Views can be nested inside one another to form a hierarchy, where each view manages the position of its children (*subviews*). Figure 1.1 shows a diagram of a typical view hierarchy.

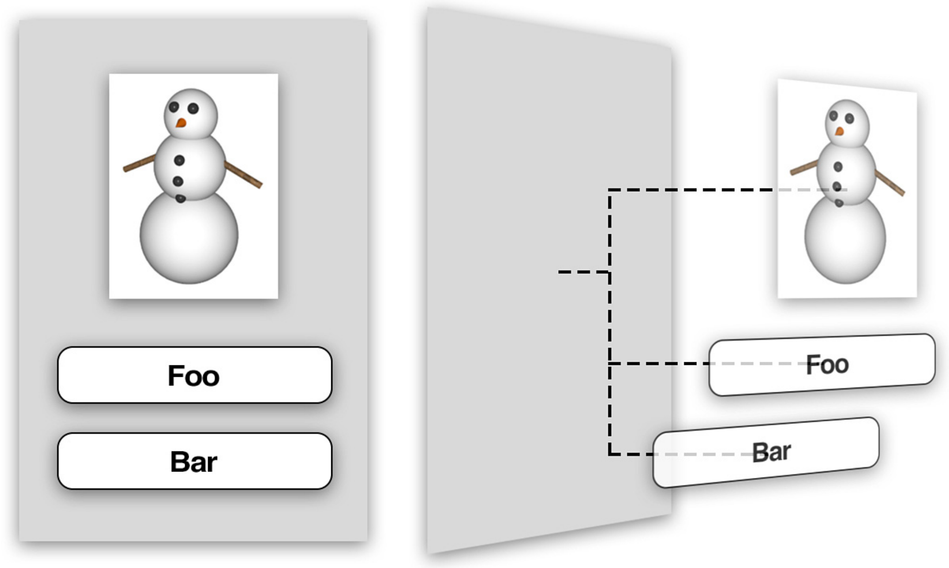

Figure 1.1 A typical iOS screen (left) and the view hierarchy that forms it (right)

In iOS, views all inherit from a common base class, `UIView`. `UIView` handles touch events and supports *Core Graphics*-based drawing, affine transforms (such as rotation or scaling), and simple animations such as sliding and fading.

What you may not realize is that `UIView` does not deal with most of these tasks itself. Rendering, layout, and animation are all managed by a Core Animation class called `CALayer`.

CALayer

The `CALayer` class is conceptually very similar to `UIView`. Layers, like views, are rectangular objects that can be arranged into a hierarchical tree. Like views, they can contain content (such as an image, text, or a background color) and manage the position of their children (*sublayers*). They have methods and properties for performing animations and transforms. The only major feature of `UIView` that isn't handled by `CALayer` is user interaction.

`CALayer` is not aware of the *responder chain* (the mechanism that iOS uses to propagate touch events through the view hierarchy) and so cannot respond to events, although it does provide methods to help determine whether a particular touch point is within the bounds of a layer (more on this in Chapter 3, "Layer Geometry").

Parallel Hierarchies

Every `UIView` has a `layer` property that is an instance of `CALayer`. This is known as the *backing layer*. The view is responsible for creating and managing this layer and for ensuring that when subviews are added or removed from the view hierarchy that their corresponding backing layers are connected up in parallel within the layer tree (see Figure 1.2).

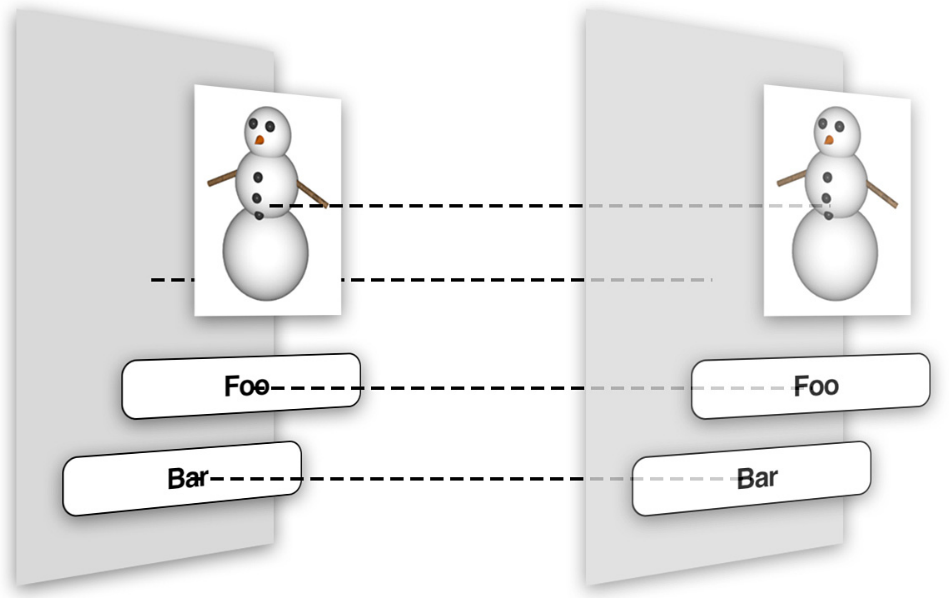

Figure 1.2 The structure of the layer tree (left) mirrors that of the view hierarchy (right)

It is actually these backing layers that are responsible for the display and animation of everything you see onscreen. `UIView` is simply a wrapper, providing iOS-specific functionality such as touch handling and high-level interfaces for some of Core Animation's low-level functionality.

Why does iOS have these two parallel hierarchies based on `UIView` and `CALayer`? Why not a single hierarchy that handles everything? The reason is to separate responsibilities, and so avoid duplicating code. Events and user interaction work quite differently on iOS than they do on Mac OS; a user interface based on multiple concurrent finger touches (*multitouch*) is a fundamentally different paradigm to a mouse and keyboard, which is why iOS has UIKit and `UIView` and Mac OS has AppKit and `NSView`. They are functionally similar, but differ significantly in the implementation.

Drawing, layout, and animation, in contrast, are concepts that apply just as much to touchscreen devices like the iPhone and iPad as they do to their laptop and desktop cousins. By separating out the logic for this functionality into the standalone Core Animation framework, Apple is able to share that code between iOS and Mac OS, making things simpler both for Apple's own OS development teams and for third-party developers who make apps that target both platforms.

In fact, there are not two, but *four* such hierarchies, each performing a different role. In addition to the view hierarchy and layer tree, there are the *presentation tree* and *render tree*, which we discuss in Chapter 7, "Implicit Animations," and Chapter 12, "Tuning for Speed," respectively.

Layer Capabilities

So if `CALayer` is just an implementation detail of the inner workings of `UIView`, why do we need to know about it at all? Surely Apple provides the nice, simple `UIView` interface precisely so that we don't need to deal directly with gnarly details of Core Animation itself?

This is true to some extent. For simple purposes, we don't really need to deal directly with `CALayer`, because Apple has made it easy to leverage powerful features like animation *indirectly* via the `UIView` interface using simple high-level APIs.

But with that simplicity comes a loss of *flexibility.* If you want to do something slightly out of the ordinary, or make use of a feature that Apple has not chosen to expose in the `UIView` class interface, you have no choice but to venture down into Core Animation to explore the lower-level options.

We've established that layers cannot handle touch events like `UIView` can, so what can they do that views *can't*? Here are some features of `CALayer` that are not exposed by `UIView`:

- Drop shadows, rounded corners, and colored borders
- 3D transforms and positioning
- Nonrectangular bounds
- Alpha masking of content
- Multistep, nonlinear animations

We explore these features in the following chapters, but for now let's look at how `CALayer` can be utilized within an app.

Working with Layers

Let’s start by creating a simple project that will allow us to manipulate the properties of a layer. In Xcode, create a new iOS project using the *Single View Application* template.

Create a small view (around 200×200 points) in the middle of the screen. You can do this either programmatically or using Interface Builder (whichever you are more comfortable with). Just make sure that you include a property in your view controller so that you can access the small view directly. We’ll call it `layerView`.

If you run the project, you should see a white square in the middle of a light gray background (see Figure 1.3). If you don’t see that, you may need to tweak the background colors of the window/view.

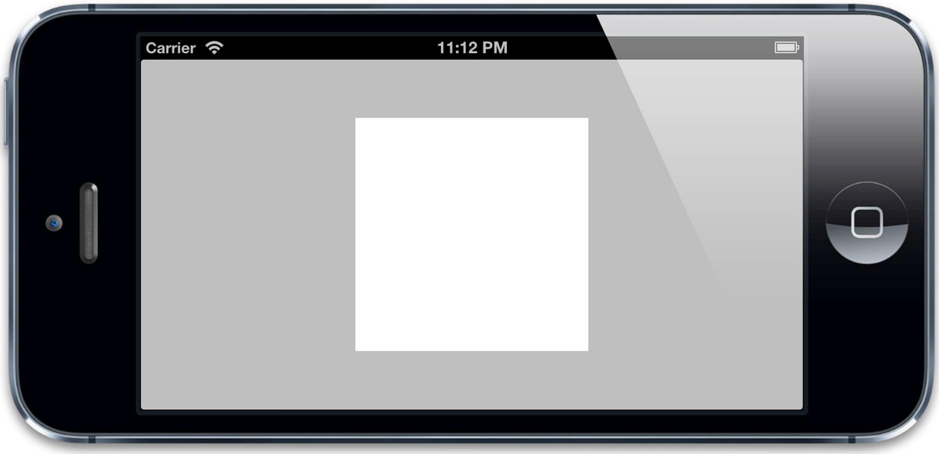

Figure 1.3 A white `UIView` on a gray background

That’s not very exciting, so let’s add a splash of color. We’ll place a small blue square inside the white one.

We could achieve this effect by simply using another `UIView` and adding it as a subview to the one we’ve already created (either in code or with Interface Builder), but that wouldn’t really teach us anything about layers.

Instead, let’s create a `CALayer` and add it as a sublayer to the backing layer of our view. Although the `layer` property is exposed in the `UIView` class interface, the standard Xcode project templates for iOS apps do not include the Core Animation headers, so we cannot call any methods or access any properties of the layer until we add the appropriate framework to the project. To do that, first add the QuartzCore framework in the application target’s Build Phases tab (see Figure 1.4), and then import `<QuartzCore/QuartzCore.h>` in the view controller’s `.m` file.

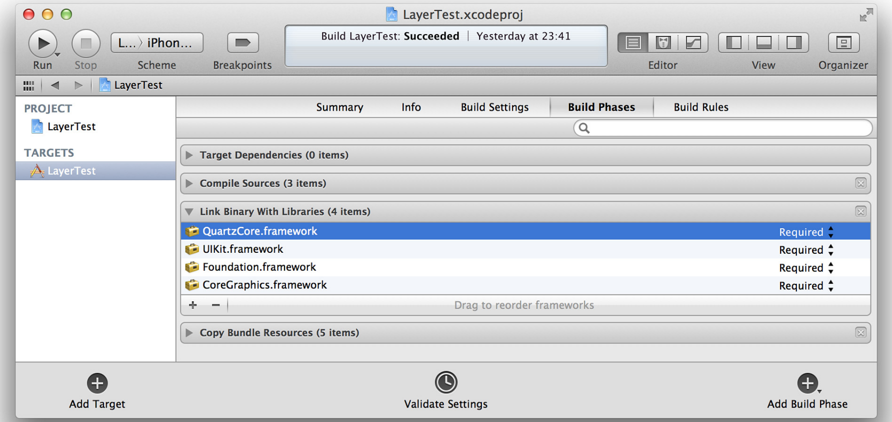
LayerTest.xcodeproj
Run
Stop
L... › iPhon...
Scheme
Breakpoints
Build LayerTest: Succeeded | Yesterday at 23:41
Editor
View
Organizer
LayerTest
PROJECT
LayerTest
TARGETS
LayerTest
Summary
Info
Build Settings
Build Phases
Build Rules
Target Dependencies (0 items)
Compile Sources (3 items)
Link Binary With Libraries (4 items)
QuartzCore.framework Required
UIKit.framework Required
Foundation.framework Required
CoreGraphics.framework Required
Drag to reorder frameworks
Copy Bundle Resources (5 items)
Add Target
Validate Settings
Add Build Phase

Figure 1.4 Adding the QuartzCore framework to the project

After doing that, we can directly reference `CALayer` and its properties and methods in our code. In Listing 1.1, we create a new `CALayer` programmatically, set its `backgroundColor` property, and then add it as a sublayer to the `layerView` backing layer. (The code assumes that we created the view using Interface Builder and that we have already linked up the `layerView` outlet.) Figure 1.5 shows the result.

The `CALayer` `backgroundColor` property is of type `CGColorRef`, not `UIColor` like the `UIView` class’s `backgroundColor`, so we need to use the `CGColor` property of our `UIColor` object when setting the color. You can create a `CGColor` directly using Core Graphics methods if you prefer, but using `UIColor` saves you from having to manually release the color when you no longer need it.

Listing 1.1 Adding a Blue Sublayer to the View

```
#import "ViewController.h"
#import <QuartzCore/QuartzCore.h>

@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
```

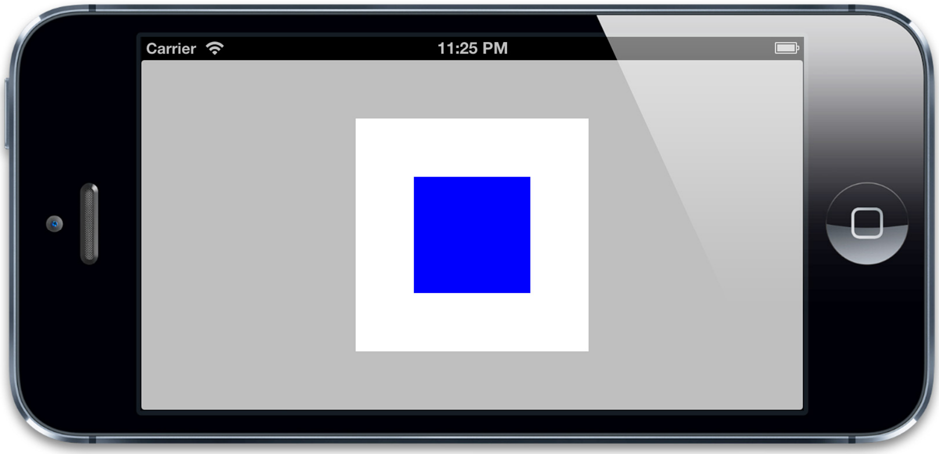

Figure 1.5 A small blue `CALayer` nested inside a white `UIView`

A view has only one *backing* layer (created automatically) but can *host* an unlimited number of additional layers. As Listing 1.1 shows, you can explicitly create standalone layers and add them directly as sublayers of the backing layer of a view. Although it is possible to add layers in this way, more often than not you will simply work with views and their backing layers and won't need to manually create additional hosted layers.

On Mac OS, prior to version 10.8, a significant performance penalty was involved in using hierarchies of layer-backed views instead of standalone `CALayer` trees hosted inside a single view. But the lightweight `UIView` class in iOS barely has any negative impact on performance when working with layers. (In Mac OS 10.8, the performance of `NSView` is greatly improved, as well.)

The benefit of using a layer-backed view instead of a hosted `CALayer` is that while you still get access to all the low-level `CALayer` features, you don't lose out on the high-level APIs (such as autoresizing, autolayout, and event handling) provided by the `UIView` class.

You might still want to use a hosted `CALayer` instead of a layer-backed `UIView` in a real-world application for a few reasons, however:

- You might be writing cross-platform code that will also need to work on a Mac.
- You might be working with multiple `CALayer` subclasses (see Chapter 6, "Specialized Layers") and have no desire to create new `UIView` subclasses to host them all.
- You might be doing such performance-critical work that even the negligible overhead of maintaining the extra `UIView` object makes a measurable difference (although in that case, you'll probably want to use something like OpenGL for your drawing anyway).

But these cases are rare, and in general, layer-backed views are a lot easier to work with than hosted layers.

Summary

This chapter explored the layer tree, a hierarchy of `CALayer` objects that exists in parallel beneath the `UIView` hierarchy that forms the iOS interface. We also created our own `CALayer` and added it to the layer tree as an experiment.

In Chapter 2, "The Backing Image," we look at the `CALayer` backing image and at the properties that Core Animation provides for manipulating how it is displayed.