Graphics and Animation in iOS

Evolve Advanced Track, Chapter 6

Xamarin Inc

Overview

iOS includes the *Core Graphics* and *Core Animation* frameworks to provide low-level drawing and animation support respectively. These frameworks are what enable the rich graphical capabilities within UIKit. All of the ultra-smooth animations in iOS such as scrolling of tables and swiping between different views perform as well as they do because they rely on Core Animation internally.

Both these frameworks can be used together to create beautiful, animated 2D graphics. In fact Core Animation can even transform 2D graphics in 3D space, creating amazing, cinematic experiences. However, to create true 3D graphics, you would need to use something like OpenGL ES or for games, turn to an API such as MonoGame, although 3D is beyond the scope of this chapter.

This chapter will examine several of the features of each framework, both to gain a better understanding of how UIKit works internally, as well as to show how to enrich the graphical capabilities of applications.

Core Graphics

Core Graphics is a low-level 2D graphics framework that allows drawing device independent graphics. All 2D drawing in UIKit uses Core Graphics internally.

Core Graphics supports drawing in a number of scenarios including:

- Drawing to the screen via a `UIView`.
- Drawing images in memory or on screen.
- Creating and drawing to a PDF.
- Reading and drawing an existing PDF.

Geometric Space

Regardless of the scenario, all drawing done with Core Graphics is done in geometric space, meaning working in abstract points rather than pixels. You describe what you want drawn in terms of geometry and drawing state such as colors, line styles, etc. and Core Graphics handles translating everything into pixels. Such state is added to a graphics context, which you can think of like a painter's canvas.

There are a few benefits to this approach:

- Drawing code becomes dynamic, and can subsequently modify graphics at runtime.
- Reducing the need for static images in the application bundle can reduce application size.
- Graphics become more resilient to resolution changes across devices.

Drawing in a UIView Subclass

Every `UIView` has a `Draw` method that is called by the system when it needs to be drawn. To add drawing code to a view, subclass `UIView` and override `Draw`:

```
public class TriangleView : UIView
{
        public override void Draw (RectangleF rect)
        {
                base.Draw (rect);
        }
}
```

Draw should never be called directly. It is called by the system during run loop processing. The first time through the run loop after a view is added to the view hierarchy, its `Draw` method is called. Subsequent calls to `Draw` occur when the view is marked as needing to be drawn by calling either `SetNeedsDisplay` or `SetNeedsDisplayInRect` on the view.

Pattern for Graphics Code

The code in the `Draw` implementation should describe what it wants drawn. The drawing code follows a pattern in which it sets some drawing state and calls a method to request it be drawn. This pattern can be generalized as follows:

1. Get a graphics context.
2. Set up drawing attributes.
3. Create some geometry from drawing primitives.
4. Call a Draw or Stroke method.

Basic Drawing Example

For example, consider the following code snippet:

```
//get graphics context
using(CGContext g = UIGraphics.GetCurrentContext ()){

        //set up drawing attributes
        g.SetLineWidth(4);
        UIColor.Purple.SetFill ();
        UIColor.Black.SetStroke ();

        //create geometry
        path = new CGPath ();

        path.AddLines(new PointF[]{
                new PointF(100,200),
                new PointF(160,100),
                new PointF(220,200)});

        path.CloseSubpath();

        //add geometry to graphics context and draw it
        g.AddPath(path);
```

```
            g.DrawPath(CGPathDrawingMode.FillStroke);
        }
```

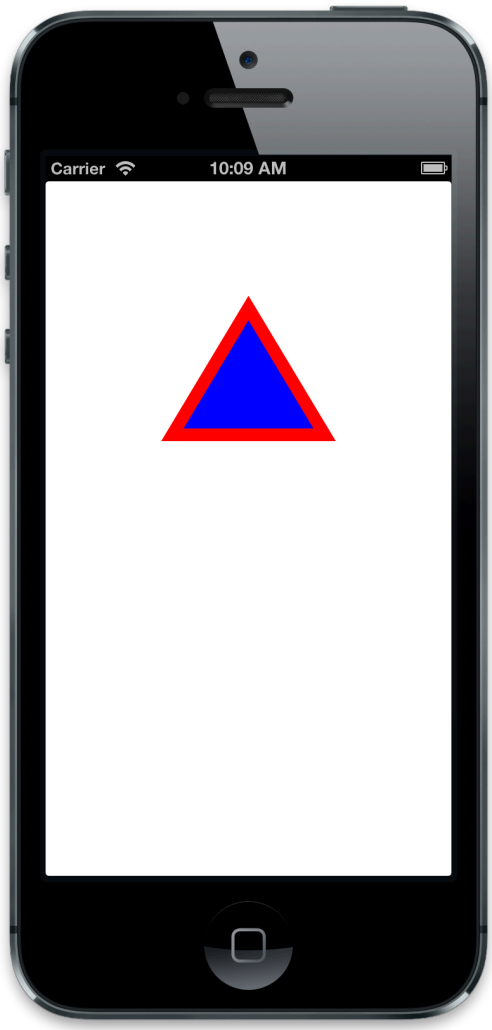
Let's break this code down. It first gets the current graphics context to use for drawing. You can think of a graphics context as the canvas that drawing happens on, containing all the state about the drawing, such as stroke and fill colors, as well as the geometry to draw.

After getting a graphics context the code sets up some attributes to use when drawing, in this case the line width, stroke and fill colors. Any subsequent drawing will then use these attributes because they are maintained in the graphics context's state.

To create geometry the code uses a `CGPath`, which allows a graphics path to be described from lines and curves. In this case, the path adds lines connecting an array of points to make up a triangle. Core Graphics uses a coordinate system for view drawing, where the origin is in the upper left, with positive x-direct to the right and the positive-y direction down.

Once the path is created, it's added to the graphics context so that calling `AddPath` and `DrawPath` respectively can draw it.

The resulting view is shown below:



Creating Gradient Fills

Richer forms of drawing are also available. For example, Core Graphics allows creating gradient fills and applying clipping paths. To draw a gradient fill inside the path from the previous example, first the path needs to be set as the clipping path:

```
// add the path back to the graphics context so that it is the current
path
g.AddPath (path);
// set the current path to be the clipping path
g.Clip ();
```

Setting the current path as the clipping path constrains all subsequent drawing within the geometry of the path, such as the following code, which draws a linear gradient:

```
// the color space determines how Core Graphics interprets color
information
using (CGColorSpace rgb = CGColorSpace.CreateDeviceRGB()) {
        CGGradient gradient = new CGGradient (rgb, new CGColor[] {
```

```
                UIColor.Blue.CGColor,
                UIColor.Yellow.CGColor
            });

            // draw a linear gradient
            g.DrawLinearGradient (gradient,
                new PointF (path.BoundingBox.Left, path.BoundingBox.Top),
                new PointF (path.BoundingBox.Right,
                    path.BoundingBox.Bottom),
                CGGradientDrawingOptions.DrawsBeforeStartLocation);
        }
```

These changes produce a gradient fill as shown below:

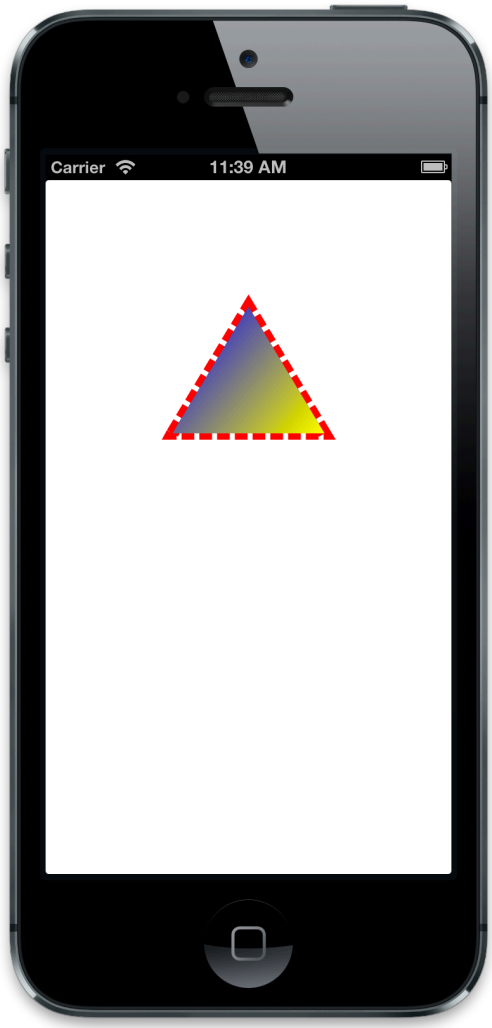
Modifying Line Patterns

The drawing attributes of lines can also be modified with Core Graphics. This includes changing the line width and stroke color, as well as the line pattern itself, as seen in the following code:

```
//use a dashed line
```

```
g.SetLineDash (0, new float[]{10, 4});
```

Adding this code before any drawing operations results in dashed strokes 10 units long, with 4 units of spacing between dashes, as shown below:



Drawing Images and Text

In addition to drawing paths in a view's graphics context, Core Graphics also supports drawing images and text. To draw an image, simply create a `CGImage` and pass it to a `DrawImage` call:

```
public override void Draw (RectangleF rect)
{
        base.Draw (rect);

        using(CGContext g = UIGraphics.GetCurrentContext ()){
                g.DrawImage (rect,
                        UIImage.FromFile ("MyImage.png").CGImage);
        }
```

```
}
```

However, this produces an image drawn upside down, as shown below:

The reason for this is Core Graphics origin for image drawing is in the lower left, while the view has its origin in the upper left. Therefore, to display the image correctly, the origin needs to be modified, which can be accomplished by modifying the *Current Transformation Matrix (CTM)*. The CTM defines where points live, also known as *user space*. Inverting the CTM in the y direction and shifting it by the bounds' height in the negative y direction can flip the image.

The graphics context has helper methods to transform the CTM. In this case, `ScaleCTM` "flips" the drawing and `TranslateCTM` shifts it to the upper left, as shown below:

```
public override void Draw (RectangleF rect)
{
        base.Draw (rect);

        using(CGContext g = UIGraphics.GetCurrentContext ()){
```

```
                    // scale and translate the CTM so the image appears upright
                    g.ScaleCTM (1, -1);
                    g.TranslateCTM (0, -Bounds.Height);
                    g.DrawImage (rect,
                            UIImage.FromFile ("MyImage.png").CGImage);
                }
        }
```

The resulting image is then displayed upright:

> **Note:** Changes to the graphics context apply to all subsequent drawing operations. Therefore, when the CTM is transformed, it will affect any additional drawing. For example, if you drew the triangle after the CTM transformation, it would appear upside down.

Adding Text to the Image

As with paths and images, drawing text with Core Graphics involves the same basic pattern of setting some graphics state and calling a method to draw. In the

case of text, the method to display text is `ShowText`. When added to the image drawing example, the following code draws some text using Core Graphics:

```
public override void Draw (RectangleF rect)
{
        base.Draw (rect);

        // image drawing code omitted for brevity ...

        // translate the CTM by the font size so it displays on screen
        float fontSize = 50f;
        g.TranslateCTM (0, fontSize);

        // set general-purpose graphics state
        g.SetLineWidth (2.0f);
        g.SetStrokeColor (UIColor.Green.CGColor);
        g.SetFillColor (UIColor.Purple.CGColor);
        g.SetShadowWithColor (new SizeF (5, 5), 0, UIColor.Blue.CGColor);

        // set text specific graphics state
        g.SetTextDrawingMode (CGTextDrawingMode.FillStroke);
        g.SelectFont ("Helvetica", fontSize, CGTextEncoding.MacRoman);

         // show the text
        g.ShowText ("Hello Evolve!");
}
```

As you can see, setting the graphics state for text drawing is similar to drawing geometry. For text drawing however, the text drawing mode and the font are applied as well. In this case, a shadow is also applied, although applying shadows works that same for path drawing.

The resulting text is displayed with the image as shown below:

Memory-Backed Images

In addition to drawing to a view's graphics context, Core Graphics supports drawing memory backed images, also known as drawing off-screen. Doing so requires:

- Creating a graphics context that is backed by an in memory bitmap
- Setting drawing state and issuing drawing commands
- Getting the image from the context
- Removing the context

Unlike the `Draw` method, where the context is supplied by the view, in this case you create the context in one of two ways:

1. By calling `UIGraphics.BeginImageContext` (or `BeginImageContextWithOptions`)
2. By creating a new `CGBitmapContextInstance`

`CGBitmapContextInstance` is useful when you are working directly with the image bits, such as for cases where you are using a custom image manipulation algorithm. In all other cases, you should use `BeginImageContext` or `BeginImageContextWithOptions`.

Once you have an image context, adding drawing code is just like it is in a `UIView` subclass. For example, the code example used earlier to draw a triangle can be used to draw to an image in memory instead of in a `UIView`, as shown below:

```
UIImage DrawTriangle ()
{
        UIImage triangleImage;

        //push a memory backed bitmap context on the context stack
        UIGraphics.BeginImageContext (new SizeF (200.0f, 200.0f));

        //get graphics context
        using(CGContext g = UIGraphics.GetCurrentContext ()){

                //set up drawing attributes
                g.SetLineWidth(4);
                UIColor.Purple.SetFill ();
                UIColor.Black.SetStroke ();

                //create geometry
                path = new CGPath ();

                path.AddLines(new PointF[]{
                        new PointF(100,200),
                        new PointF(160,100),
                        new PointF(220,200)});

                path.CloseSubpath();

                //add geometry to graphics context and draw it
                g.AddPath(path);
                g.DrawPath(CGPathDrawingMode.FillStroke);

                //get a UIImage from the context
                triangleImage =
                        UIGraphics.GetImageFromCurrentImageContext ();
        }
        return triangleImage;
}
```

A common use of drawing to a memory-backed bitmap is to capture an image from any `UIView`. For example, the following code renders a view's layer to a bitmap context an creates a `UIImage` from it:

```
UIGraphics.BeginImageContext (cellView.Frame.Size);
//render the view's layer in the current context
anyView.Layer.RenderInContext (UIGraphics.GetCurrentContext ());
//get a UIImage from the context
UIImage anyViewImage = UIGraphics.GetImageFromCurrentImageContext ();
```

```
UIGraphics.EndImageContext ();
```

Drawing PDFs

In addition to images, Core Graphics supports PDF drawing. Like images, you can render a PDF in memory as well as read a PDF for rendering in a `UIView`.

Memory-Backed PDF

For an in-memory PDF, you need to create a PDF context by calling `BeginPDFContext`. Drawing to PDF is granular to pages. Each page is started by calling `BeginPDFPage` and completed by calling `EndPDFContent`, with the graphics code in between. Also, as with image drawing, memory backed PDF drawing uses an origin in the lower left, which can be accounted by modifying the CTM just like with images.

The following code shows how to draw text to a PDF:

```
//data buffer to hold the PDF
NSMutableData data = new NSMutableData ();
//create a PDF with empty rectangle, which will configure it for 8.5x11
inches
UIGraphics.BeginPDFContext (data, RectangleF.Empty, null);
//start a PDF page
UIGraphics.BeginPDFPage ();
//graphics code
using(CGContext g = UIGraphics.GetCurrentContext ()){
        g.ScaleCTM (1, -1);
        g.TranslateCTM (0, -25);
        g.SelectFont ("Helvetica", 25, CGTextEncoding.MacRoman);
        g.ShowText ("Hello Evolve");
}
//complete a PDF page
UIGraphics.EndPDFContent ();
```

The resulting text is drawn to the PDF, which is then contained in an `NSData` that can be saved, uploaded, emailed, etc.

PDF in a UIView

Core Graphics also supports reading a PDF from a file and rendering it in a view using the `CGPDFDocument` class. The `CGPDFDocument` class represents a PDF in code, and can be used to read and draw pages.

For example, the following code in a `UIView` subclass reads a PDF from a file into a `CGPDFDocument`:

```
public class PDFView : UIView
{
        CGPDFDocument pdfDoc;

        public PDFView ()
        {
                //create a CGPDFDocument from file.pdf included in the main
bundle
                pdfDoc = CGPDFDocument.FromFile ("file.pdf");
        }
```

```
        public override void Draw (Rectangle rect)
        {
                ...
        }
}
```

The `Draw` method can then use the `CGPDFDocument` to read a page into `CGPDFPage` and render it by calling `DrawPDFPage`, as shown below:

```
public override void Draw (RectangleF rect)
{
        base.Draw (rect);

        //flip the CTM so the PDF will be drawn upright
        using(CGContext g = UIGraphics.GetCurrentContext ()){
                g.TranslateCTM (0, Bounds.Height);
                g.ScaleCTM (1, -1);

                // render the first page of the PDF
                using (CGPDFPage pdfPage = pdfDoc.GetPage (1)) {

                        //get the affine transform that defines where
                        //the PDF is drawn
                        CGAffineTransform t = pdfPage.GetDrawingTransform
                                (CGPDFBox.Crop, rect, 0, true);
                        //concatenate the pdf transform with the CTM for
                        //display in the view
                        g.ConcatCTM (t);

                        //draw the pdf page
                        g.DrawPDFPage (pdfPage);
                }
        }
}
```

Core Animation

iOS uses the Core Animation framework to create animation effects such as transitioning between views, sliding menus and scrolling effects to name a few. There are two ways to work with animation. The first way is via UIKit, which includes view-based animations as well as animated transitions between controllers. The second way is working with Core Animation layers directly for finer-grained control.

UIKit Animation

UIKit provides several features that make it easy to add animation to an application. Although it uses Core Animation internally, it abstracts it away so you work only with views and controllers.

This section discusses UIKit animation features including:

- Transitions between controllers
- Transitions between views
- View property animation

View Controller Transitions

`UIViewController` provides built-in support for transitioning between view controllers through the `PresentViewController` method. When using `PresentViewController`, the transition to the second controller can optionally be animated.

For example, consider an application with two controllers, where touching a button in the first controller calls `PresentViewController` to display a second controller. To control what transition animation is used to show the second controller, simply set its `ModalTransitionStyle` property as shown below:

```
SecondViewController vc2 = new SecondViewController {
        ModalTransitionStyle = UIModalTransitionStyle.PartialCurl
};
```

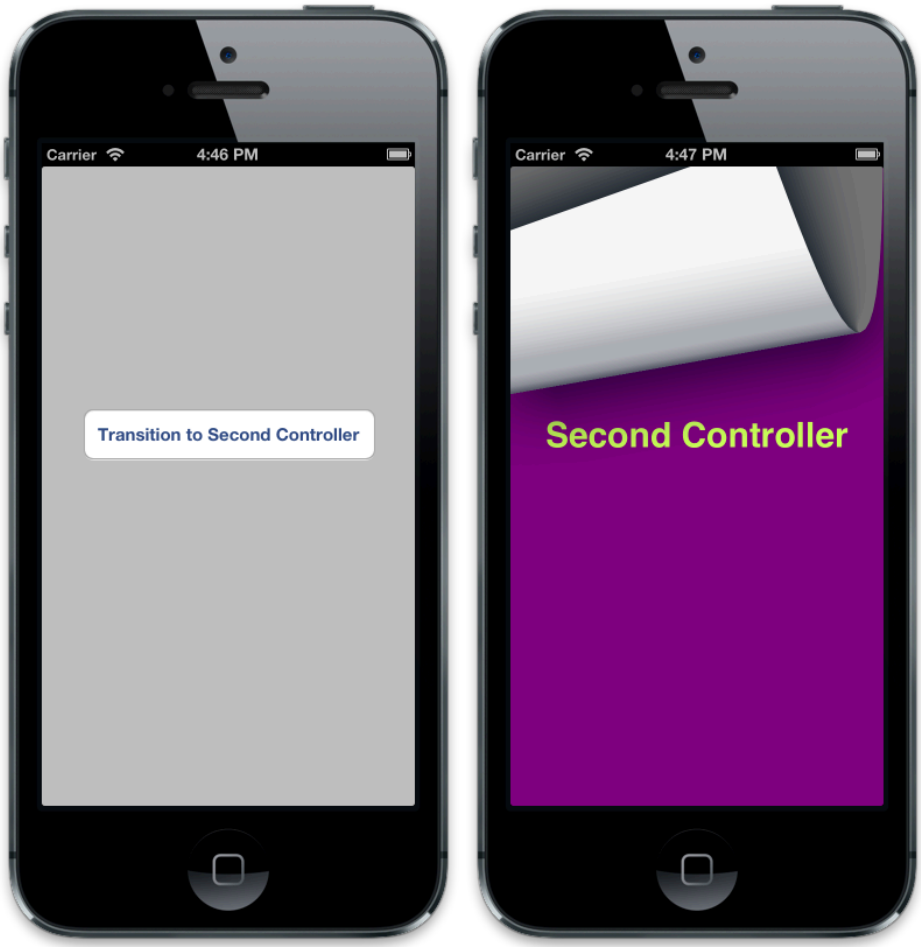
In this case a `PartialCurl` animation is used, although several others are available, including:

- `CrossDissolve`
- `CrossHorizontal`
- `FlipHorizontal`

To animate the transition, pass `true` as the second argument to `PresentViewController`:

```
PresentViewController (vc2, true, null);
```

The following screenshot shows what the transition looks like for the `PartialCurl` case:



View Transitions

In addition to transitions between controllers, UIKit also supports animating transitions between views to swap one view for another.

For example, say you had a controller with `UIImageView`, where tapping on the image should display a second `UIImageView`. To animate the image view's superview to transition to the second image view is as simple as calling `UIView.Transition`, passing it the `toView` and `fromView` as shown below:

```
UIView.Transition (
        fromView: view1,
        toView: view2,
        duration: 2,
        options: UIViewAnimationOptions.TransitionFlipFromTop |
                UIViewAnimationOptions.CurveEaseInOut,
        completion: () => { Console.WriteLine ("transition complete"); });
```

Note: The sample code shown here is available in the AnimationSamples project that accompanies this chapter.

`UIView.Transition` also takes a `duration` parameter that controls how long the animation runs, as well as `options` to specify things such as the animation to use

and the easing function. Additionally, you can specify a completion handler that will be called when the animation completes.

The screenshots below show the animated transition between the image views when `TransitionFlipFromTop` is used:

View Property Animations

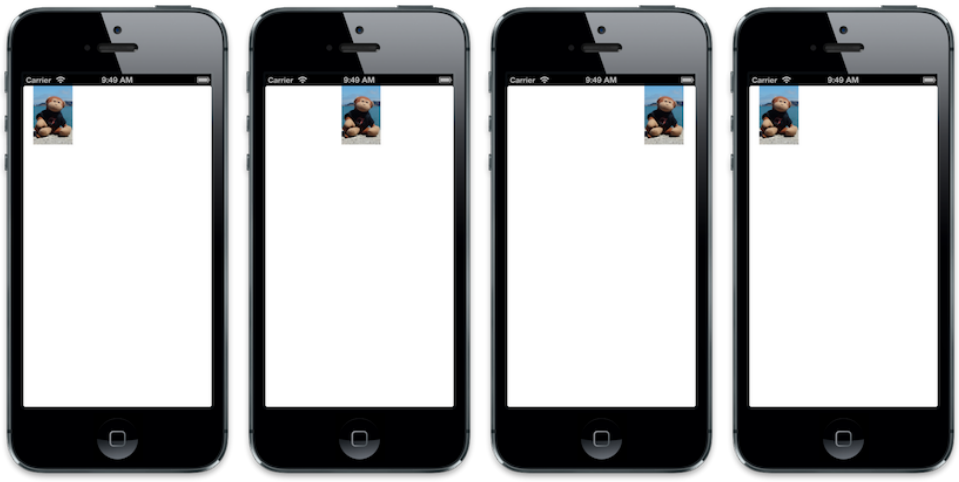
UIKit supports animating a variety of properties on the UIView class for free, including:

- Frame
- Bounds
- Center
- Alpha
- Transform
- Color

These animations happen implicitly by specifying property changes in an `NSAction` delegate passed to the static `UIView.Animate` method. For example, the following code animates the center point of a `UIImageView`:

```
pt = imgView.Center;

UIView.Animate (
        duration: 2,
        delay: 0,
        options: UIViewAnimationOptions.CurveEaseInOut |
                UIViewAnimationOptions.Autoreverse,
        animation: () => {
                imgView.Center = new PointF (View.Bounds.GetMaxX ()
                        - imgView.Frame.Width / 2, pt.Y);},
        completion: () => {
                imgView.Center = pt; }
);
```

This results in an image animating back and forth across the top of the screen, as shown below:



As with the `Transition` method, `Animate` allows the duration to be set, along with the easing function. This example also used the `UIViewAnimationOptions.Autoreverse` option, which causes the animation to animate from the value back to the initial one. However, the code also sets the `Center` back to its initial value in a completion handler. While an animation is interpolating property values over time, the actual model value of the property is always the final value that has been set. In this example, the value is a point near the right side of the superview. Without setting the `Center` to the initial point, which is where the animation completes due to the `Autoreverse` being set, the image would snap back to the right side after the animation completes, as shown below:

Core Animation

`UIView` animations allow a lot of capability and should be used if possible due to the ease of implementation. As mentioned earlier, UIView animations use the Core Animation framework. However, some things cannot be done with `UIView` animations, such as animating additional properties that cannot be animated with a view, or interpolating along a non-linear path. In such cases where you need finer control, Core Animation can be used directly as well.

Layers

When working with Core Animation, animation happens via *layers*, which are of type `CALayer`. A layer is conceptually similar to a view in that there is a layer hierarchy, much like there is a view hierarchy. Actually, layers back views, with the view adding support for user interaction. You can access the layer of any view via the view's `Layer` property. In fact, the context used in `Draw` method of `UIView` is actually created from the layer. Internally, the layer backing a `UIView` has its delegate set to the view itself, which is what calls `Draw`. So when drawing to a `UIView`, you are actually drawing to its layer.

Layer animations can be either implicit or explicit. Implicit animations are declarative. You simply declare what layer properties should change and the

animation just works. Explicit animations on the other hand are created via an animation class that is added to a layer. Explicit animations allow addition control over how an animation is created. The following sections delve into implicit and explicit animations in greater depth.

Implicit Animations

One way to animate the properties of a layer is via an implicit animation. `UIView` animations create implicit animations. However, you can create implicit animations directly against a layer as well.

For example, the following code sets a layer's `Contents` from an image, sets a border width and color, and adds the layer as a sublayer of the view's layer:

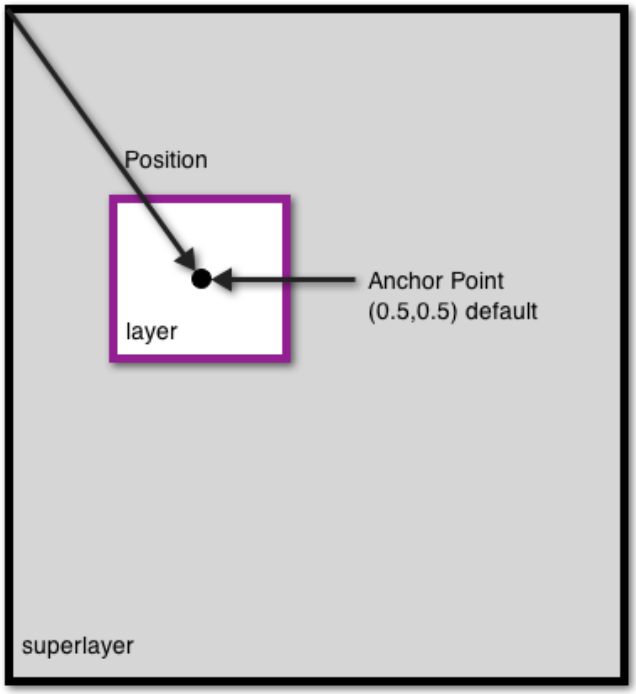
```
public override void ViewDidLoad ()
{
        base.ViewDidLoad ();

        layer = new CALayer ();
        layer.Bounds = new RectangleF (0, 0, 50, 50);
        layer.Position = new PointF (50, 50);
        layer.Contents = UIImage.FromFile ("monkey2.png").CGImage;
        layer.ContentsGravity = CALayer.GravityResize;
        layer.BorderWidth = 1.5f;
        layer.BorderColor = UIColor.Green.CGColor;

        View.Layer.AddSublayer (layer);
}
```

To add an implicit animation for the layer, simply wrap property changes in a `CATransaction`. This allows animating properties that would not be animatable with a view animation, such as the `BorderWidth` and `BorderColor` as shown below:

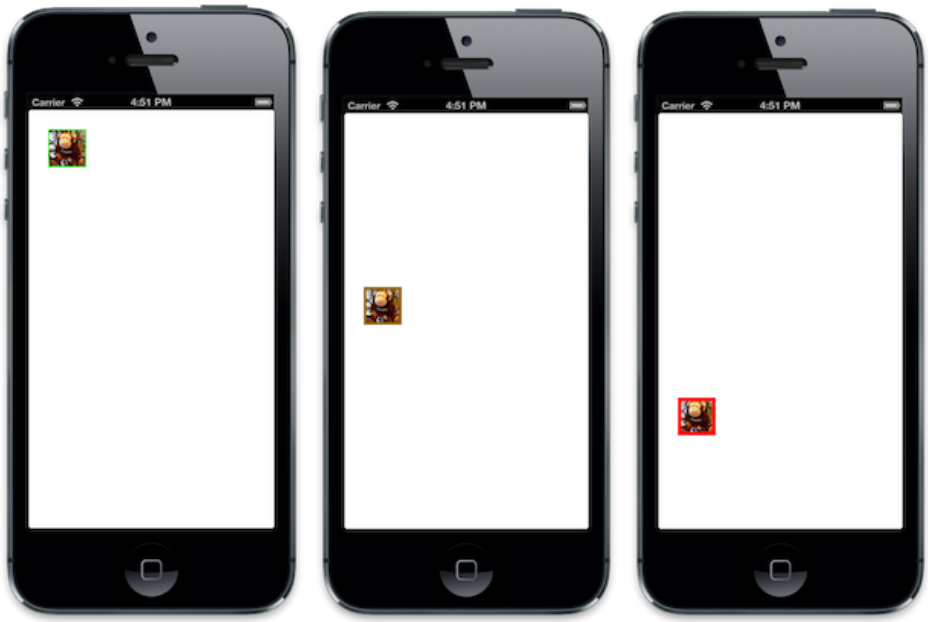
```
public override void ViewDidAppear (bool animated)
{
        base.ViewDidAppear (animated);

        CATransaction.Begin ();
        CATransaction.AnimationDuration = 2;
        layer.Position = new PointF (50, 400);
        layer.BorderWidth = 5.0f;
        layer.BorderColor = UIColor.Red.CGColor;
        CATransaction.Commit ();
}
```

This code also animates the layer's `Position`, which is the location of the layer's anchor point measured from the upper left of the superlayer's coordinates. The anchor point of a layer is a normalized point within the layer's coordinate system.

The following figure shows the position and anchor point:



When the example is run, the `Position`, `BorderWidth` and `BorderColor` animate as shown in the following screenshots:



Explicit Animations

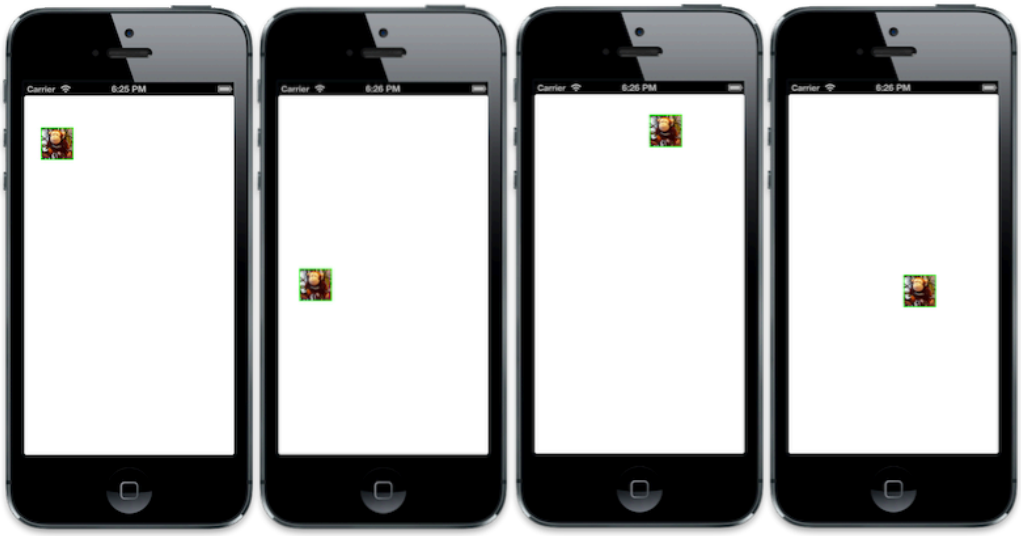
In addition to implicit animations, Core Animation includes a variety of classes that inherit from `CAAnimation` that let you encapsulate animations that are then explicitly added to a layer. These allow finer-grained control over animations, such as modifying the start value of an animation, grouping animations and specifying keyframes to allow non-linear paths.

The following code shows an example of an explicit animation using a `CAKeyframeAnimation` for the layer shown earlier (in the Implicit Animation section):

```
public override void ViewDidAppear (bool animated)
{
        base.ViewDidAppear (animated);

        // get the initial value to start the animation from
        PointF fromPt = layer.Position;

        // set the position to coincide with the final animation value
        // to prevent it from snapping back to the starting position
        // after the animation completes
        layer.Position = new PointF (200, 300);

        // create a path for the animation to follow
        CGPath path = new CGPath ();
        path.AddLines (new PointF[] { fromPt, new PointF (50, 300), new
PointF (200, 50), new PointF (200, 300) });

        // create a keyframe animation for the position using the path
        CAKeyFrameAnimation animPosition =
(CAKeyFrameAnimation)CAKeyFrameAnimation.FromKeyPath ("position");
        animPosition.Path = path;
        animPosition.Duration = 2;

        // add the animation to the layer
        // the "position" key is used to overwrite the implicit animation
        // created when the layer position is set above
        layer.AddAnimation (animPosition, "position");
}
```

This code changes the `Position` of the layer by creating a path that is then used to define a keyframe animation. Notice that the layer's `Position` is set to the final value of the `Position` from the animation. Without this, the layer would abruptly return to its `Position` before the animation because the animation only changes the presentation value and not the actual model value. By setting the model value to the final value from the animation, the layer stay in place at the end of the animation.

The following screenshots show the layer containing the image animating through the specified path:



Walkthrough - Drawing and Animating a Path

For this walkthrough we are going to draw a path using Core Graphics in response to touch input. Then, we are code to add a `CALayer` containing an image that we will animate along the path.

The following screenshot shows the completed application:

Drawing a Path

Open the GraphicsDemo starter project and open the `DemoView` class.

In `DemoView` add a `CGPath` variable to the class and instantiate it in the constructor. Also declare to `PointF` variable that we will use to capture the touch point that we construct the path from:

```
public class DemoView : UIView
{
        CGPath path;
        PointF initialPoint;
        PointF latestPoint;

        public DemoView ()
        {
                BackgroundColor = UIColor.White;

                path = new CGPath ();
        }
}
```

Next, override `TouchesBegan` and `TouchesMoved`, and add the following implementations to capture the initial touch point and each subsequent touch point respectively:

```
public override void TouchesBegan (MonoTouch.Foundation.NSSet touches,
UIEvent evt)
{
        base.TouchesBegan (touches, evt);

        UITouch touch = touches.AnyObject as UITouch;

        if (touch != null) {
                initialPoint = touch.LocationInView (this);
        }
}

public override void TouchesMoved (MonoTouch.Foundation.NSSet touches,
UIEvent evt)
{
        base.TouchesMoved (touches, evt);

        UITouch touch = touches.AnyObject as UITouch;

        if (touch != null) {
                latestPoint = touch.LocationInView (this);
                SetNeedsDisplay ();
        }
}
```

We call `SetNeedsDisplay` each time touches move in order for `Draw` to be called on the next run loop pass.

We'll add lines to the path in the `Draw` method and use a red, dashed line to draw with. Implement `Draw` with the code shown below:

```
public override void Draw (System.Drawing.RectangleF rect)
{
        base.Draw (rect);

        if (!initialPoint.IsEmpty) {
                //get graphics context
                CGContext g = UIGraphics.GetCurrentContext ();

                //set up drawing attributes
                g.SetLineWidth (2);
                UIColor.Red.SetStroke ();

                //add lines to the touch points
                if (path.IsEmpty) {
                        path.AddLines (new PointF[]{initialPoint,
latestPoint});
                } else {
                        path.AddLineToPoint (latestPoint);
                }

                //use a dashed line
                g.SetLineDash (0, new float[]{5, 2});

                //add geometry to graphics context and draw it
```

```
                g.AddPath (path);
                g.DrawPath (CGPathDrawingMode.Stroke);
            }
        }
```

If we run the application now, we can touch to draw on the screen, as shown in the following screenshot:

Animating Along a Path

Next, let's add the code to animate a layer along the drawn path.

Add a `CALayer` variable to the class and create it in the constructor:

```
CALayer layer;

public DemoView ()
{
        BackgroundColor = UIColor.White;

        path = new CGPath ();

        //create layer
        layer = new CALayer ();
        layer.Bounds = new RectangleF (0, 0, 50, 50);
        layer.Position = new PointF (50, 50);
```

```
        layer.Contents = UIImage.FromFile ("monkey.png").CGImage;
        layer.ContentsGravity = CALayer.GravityResizeAspect;
        layer.BorderWidth = 1.5f;
        layer.CornerRadius = 5;
        layer.BorderColor = UIColor.Blue.CGColor;
        layer.BackgroundColor = UIColor.Purple.CGColor;
}
```

We'll add the layer as a sublayer of the view's layer when the user lifts up their finger from the screen. Then, we will create a keyframe animation using the path, animating the layer's `Position`.

Add the following code to the `TouchesEnded` method to accomplish this:

```
public override void TouchesEnded (MonoTouch.Foundation.NSSet touches,
UIEvent evt)
{
        base.TouchesEnded (touches, evt);

        //add layer with image and animate along path

        if (layer.SuperLayer == null)
                Layer.AddSublayer (layer);

        // create a keyframe animation for the position using the path
        layer.Position = latestPoint;
        CAKeyFrameAnimation animPosition =
(CAKeyFrameAnimation)CAKeyFrameAnimation.FromKeyPath ("position");
        animPosition.Path = path;
        animPosition.Duration = 3;
        layer.AddAnimation (animPosition, "position");
}
```

Run the application now and after drawing, a layer with an image is added that travels along the drawn path:

Summary

In this chapter we looked at the graphics and animation capabilities provided via the *Core Graphics* and *Core Animation* frameworks. We saw how to use Core Graphics to draw geometry, images and PDFs within the context of a `UIView`, as well as to memory-backed graphics contexts. We then examined Core Animation, showing both how it powers animations in UIKit, and how it can be used directly for lower-level animation control. Finally we stepped through a walkthrough that tied graphics and animation concepts together.