Extracted from:

Core Animation for Mac OS X and the iPhone

Creating Compelling Dynamic User Interfaces

This PDF file contains pages extracted from Core Animation for Mac OS X and the iPhone, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2008 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

If I were to wish for anything, I should not wish for wealth and power, but for the passionate sense of potential—for the eye which, ever young and ardent, sees the possible. Pleasure disappoints; possibility never.

► Søren Kierkegaard

Chapter 6

Filtered Views

In Mac OS X Leopard we have more than 125 different filters to correct the color, sharpen, distort, and stylize. We can preview the full set with the CI Filter Browser widget (in a typical install, you can find the widget at /Developer/Extras/Core Image/CI Filter Browser.wdgt). And we can use and combine all these filters to modify the look of our layer-backed views.

Core Image filters let us use the GPU to manipulate images. Besides the blindingly fast performance that we get from using the GPU to do the image manipulation, Core Image also allows us to use the sophisticated OpenGL shading language to write our own custom filters. An exploration of Core Image could fill a book all its own. In this chapter, we focus on how filters apply to Core Animation.

Recall that a layer-backed view has its drawing content cached in its layer. Once the content of the view is cached, you can then treat it much like you would an image. In fact, that is how the filtering works with views. Once you turn on layer backing and once the view's drawing is cached, you can apply Core Image filters, and they act on the cached drawing just as they would an image. There are countless ways you can use these filters, and as you gain more experience, your imagination will run wild with what is possible.

Another really cool thing you can do with these filters is to use the transition filters for transitioning between subviews. We will learn how to use the CI transition filters in this chapter.

One commonly used Core Image filter is Bloom. A Bloom filter softens the edges of an image and brightens the lighter parts. Overall, it makes the image appear softer and as if it were glowing. We can adjust and animate a couple of properties in the Bloom filter. The first is the radius. The radius specifies how many pixels are used in the effect (the larger the radius, the greater the effect of the filter). The second attribute is the

intensity of the filter. A common use of this filter is to highlight a layer. And if the intensity of the filter is animated, then the content of the layer will appear to pulse. This is really cool and can be used to great effect with UI elements. One in particular from Apple's World Wide Developer Conference (WWDC) demos of Core Animation is a recipe application's menu system. As the user changes the selected recipe, a pulsing white rectangle follows the selection.

The examples that I develop in this chapter are more focused on how the technologies fit together. I purposefully chose to use filters that are less commonly used than something like Bloom. I want you to see how things work without having to think about things like "Oh, that would be prettier if it had a shadow." I use filters that are unlikely to be used in a real application so that you can focus on how to apply the filters. I will leave the beautification of your application with the perfect filter effects to you.

Let's get started with a quick overview of what the filters are and how we use them.

6.1 View Filters

You can apply filters in three different ways to layer-backed views: backgroundFilters, contentFilters, and compositingFilters. As the names imply, the filters act on different parts of the view.

The background filters will apply the filter to the background of the view. The background is any part of the view that is not drawn on. Another way to think about it is that any part of the view's superview that is visible through the view will have the filtered applied. A background filter might be used to emphasize a view by softly blurring the background (via one of the blur filters).

Filters in the contentFilters array are applied to the view but not to the background. Any drawing done in the view will have the content filter applied. Using a content filter is a good way to change the user's perception of the content to either emphasize it or de-emphasize it.

Finally, compositingFilters allow us to change the way our view content is composited with the background content. There is a huge range of compositing operations possible from simply replacing the background (the default) to inverting the colors of the background.

The filters are chained together for us and applied in order from the first to the last filter. Each filter's inputImage is set to the outputImage

CLICK HERE to purchase this book now.

of the previous filter. The first filter's inputImage will be the content of the view as cached in the layer, and the outputImage of the last filter is what will be displayed. This chaining behavior takes care of the input and output images of all the filters, so you don't have to set any of these keys for the filters.

Once a filter is attached to a layer-backed view, the layer becomes a "manager" of sorts for the filter. Don't manipulate the filter directly once it is attached to a view. Instead, use key-value coding (KVC) to modify the filter. For example, if you have a Box Blur filter attached as a content filter of a view, don't change the radius directly. Use the KVC method called setValue:forKeyPath: on the view like this:

```
[myView setValue:[NSNumber numberWithFloat:2.5]
        forKeyPath:@"contentFilters.myFilter.inputRadius"]
```

This way, the view knows when the filter is changed so that it can take the appropriate action (apply the changed filter, cache the result, and so on). If you make changes to the filter without going through the layer, you will have some undefined behavior. From experience, typically the filter stops being applied, but occasionally garbage is copied to the screen.

Another thing that is often confusing with filters that are attached to a view or layer is that they should be given a unique name. Despite that the various filters types (background, content, and composite) are stored in arrays, the view or layer finds the filters as if they were in a dictionary. For example, in the code before the key path, contentFilters.myFilter.inputRadius looks in the array of contentFilters for a filter named myFilter. Once found, it sets the value for the key inputRadius. Now for the part that confuses people: the CIFilter class method filterWithName: expects the name of the filter class (that is, CIBoxFilter), not the name of the instance. So once you create the filter (with filterWithName:), you need to set the new instance's name. The code will look something like this to create an instance of the CIPointillize filter:

```
CIFilter *pointalize = [CIFilter filterWithName:@"CIPointillize"
                                 keysAndValues:kCIInputRadiusKey,
                                              [NSNumber numberWithFloat:1.0f],
                                              kCIInputCenterKey, center, nil];
pointalize.name = @"pointalize";
// later in the code we would have this line of code to change
// the value of the inputRadius for the filter
[myView setValue:[NSNumber numberWithFloat:14.0f]
                forKeyPath:[NSString stringWithFormat:
                    @"contentFilters.pointalize.%@", kCIInputRadiusKey]];
```

Click Here to purchase this book now.

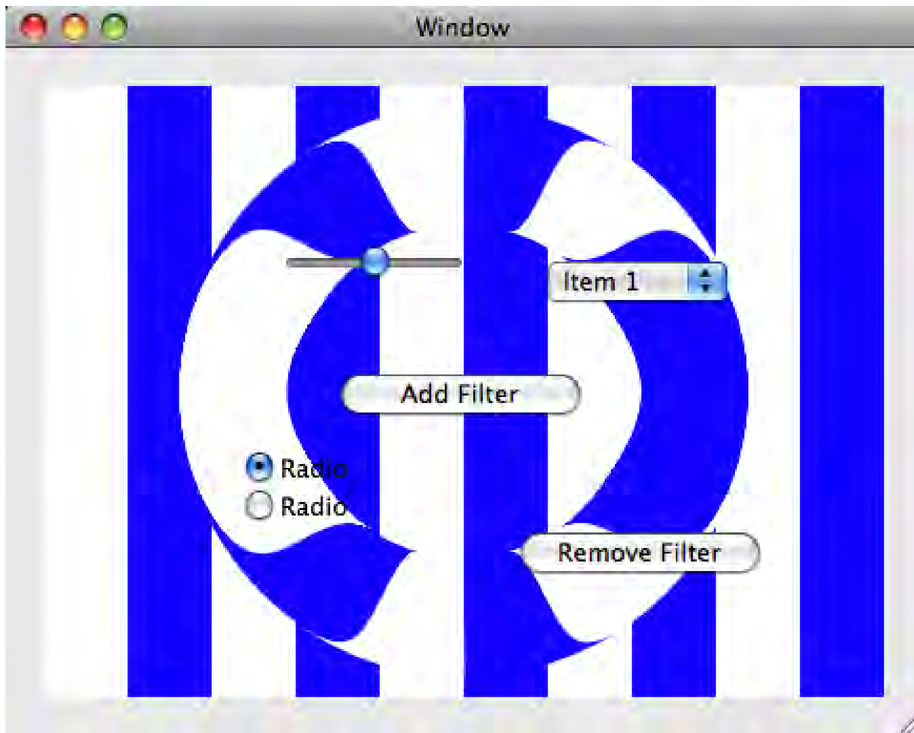

Figure 6.1: Background filter applied

The important thing to remember here is that you need to set the name property on the filter to be able to change its values once it's added to one of the filter arrays on your view.

6.2 Background Filters

As you know, views have drawing done in them only by the view's own drawRect: or by any of the view's subviews that implement drawRect:. The rest of the view's bounds are left as transparent, and the superview will "show through" the view. This transparent area is the "background" to which the filters are applied. Let's look at an example to get a feel for what applying a background filter looks like.

In Figure 6.1, the background view draws blue stripes so that the filter can be better seen. The view in the foreground has a bunch of controls as subviews. The view containing the controls has a CITorusLensDistor-

tion filter applied (and its width is animated), which is essentially like placing a glass distortion in the shape of a torus over the background. Notice that the background filter is applied only to the superview. None of the controls is filtered. In fact, the controls continue to function as usual, without regard to the filter. Let's look at the code to see how this all works:

Download FilteredViews/BackgroundFilteredView/BackgroundFilteredView.m

```
Line 1  - (void) applyFilter {
     -    CIVector *center = [CIVector
     -                        vectorWithX:NSMidX([self bounds])
     -                        Y:NSMidY([self bounds])];
     5    CIFilter *torus = [CIFilter filterWithName:@"CITorusLensDistortion"
     -                                 keysAndValues:kCIInputCenterKey, center,
     -                       kCIInputRadiusKey, [NSNumber numberWithFloat:150.0f],
     -                       kCIInputWidthKey, [NSNumber numberWithFloat:2.0f],
     -                       kCIInputRefractionKey, [NSNumber numberWithFloat:1.7f],
    10                       nil];
     -    torus.name = @"torus";
     -
     -    [controls setBackgroundFilters:[NSArray arrayWithObjects:torus, nil]];
     -    [self addAnimationToTorusFilter];
    15  }
```

The applyFilter method on line 1 is responsible for creating the filter and adding it to the backgroundFilters property of the controls subview as well as attaching the animation. Using the filter, as you can see, is very simple; most of the code in this method is just setting up the torus filter:

Download FilteredViews/BackgroundFilteredView/BackgroundFilteredView.m

```
Line 1  - (void) addAnimationToTorusFilter {
     -    NSString *keyPath = [NSString stringWithFormat:
     -                          @"backgroundFilters.torus.%@",
     -                          kCIInputWidthKey];
     5    CABasicAnimation *animation = [CABasicAnimation
     -                                   animationWithKeyPath:keyPath];
     -    animation.fromValue = [NSNumber numberWithFloat:50.0f];
     -    animation.toValue = [NSNumber numberWithFloat:80.0f];
     -    animation.duration = 1.0;
    10    animation.repeatCount = 1e100f;
     -    animation.timingFunction = [CAMediaTimingFunction functionWithName:
     -                                kCAMediaTimingFunctionEaseInEaseOut];
     -    animation.autoreverses = YES;
     -    [[controls layer] addAnimation:animation forKey:@"torusAnimation"];
    15  }
```

Adding the animation is similarly simple; all you have to do is create the animation, configure it, and then add it to the layer of your view. Here are some things to note about how this animation is applied: with the

CLICK HERE to purchase this book now.

keyPath, notice that it's set to backgroundFilters.torus.kCIInputWidth. backgroundFilters is the key into the filters list of our view. torus is the name you gave to the filter in applyFilter. And the kCIInputWidthKey is a constant that specifies the width property of the torus filter.

The really cool part is that all the properties of the filter can be similarly animated. If you wanted to animate the outer radius of the torus from 120 to 180, you'd simply create a new animation that was tied to backgroundFilters.torus.kCIInputRadiusKey and set its toValue to 120 and its fromValue to 180. Adding this animation to the layer would then cause the outer radius of the torus to animate in addition to the width. Then you'd have two animations going on at once. You can even add several filters and animate each of them independently. Of course, you can go overboard too and overwhelm your users. Moderation is the best course of action. There are so many cool things to do that it's hard not to get carried away!

Download FilteredViews/BackgroundFilteredView/BackgroundFilteredView.m

```
Line 1  - (void)awakeFromNib {
     -    [self setWantsLayer:YES];
     -    [self applyFilter];
     -  }
```

The awakeFromNib method on line 1 is adding the filter when the NIB file is loaded, but more important, it is setting the view to be layer-backed. For the background filters to be applied, the superview must be layer-backed (which will in turn make the view layer-backed as well).

6.3 Content Filters

A content filter lets you filter the content of the view instead of the background. We'll explore this idea by manipulating controls as you would any other view content. If you apply a filter to the content of a view, all the content including controls will have that filter applied to it. I'm not suggesting that you apply filters to controls as a typical use case, but it does serve our purpose here of seeing what applying these filters can do for us.

Let's take a look at an example to help solidify our knowledge of filters on the content of layer-backed views. This example has two views: the background view that is responsible for drawing the stripes and putting the content filter onto its only subview and the subview that does nothing but hold a set of controls. The UI is shown in Figure 6.2, on the next page.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Core Animation for Mac OS X and the iPhone's Home Page
http://pragprog.com/titles/bdcora
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/bdcora.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	`orders@pragprog.com`
Non-English Versions:	`translations@pragprog.com`
Pragmatic Teaching:	`academic@pragprog.com`
Author Proposals:	`proposals@pragprog.com`