MOBILE PROGRAMMING SERIES

iOS UICOLLECTIONVIEW

THE COMPLETE GUIDE

SECOND EDITION

ASH FURROW

iOS UICollectionView: The Complete Guide

Second Edition

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    [data af_decompressedImageFromJPEGDataWithCallback:
        ^(UIImage *decompressedImage) {
        [cell setImage:decompressedImage];
    }];
});
```

Because JPEG decompression takes only a few milliseconds, I'm updating the cell directly in Listing 2.20. If you're decompressing JPEGs that are megabytes large, this isn't going to work for you, but displaying images that large in a collection view is a bad idea, generally.

If you rerun Instruments, you see that the most expensive operation, overall, is allocating space from `UICollectionViewCell`'s `initWithFrame:`. This is really good. Anecdotally, the app runs way smoother.

High five! But take a look at two other parts of the codebase that could be improved.

The images are 145 by 145 pixels, but your cells are 145 by 100 logical pixels. `UIImageView` is scaling the images down to fit. You could change the content mode to center them, instead, so that they aren't scaled.

Ideally, your image size and cell size should be the same so that the OS doesn't have to resize anything, which improves performance. However, if you're using a third-party API, you won't have control over the image size.

The only other thing I could recommend to improve performance of this example is to turn on the `masksToBounds` property of the cell's layer; you used this property in conjunction with the `cornerRadius`. This causes a strain on the CPU because it requires offscreen rendering passes and can cause a lot of problems. It's something to check if you can't figure out why your collection view is slow.

If the collection view background is opaque, you can use a PNG in a `UIImageView` subview of the `contentView` to mask out the corners. This is a good approach, too, but don't use resizable images if you can avoid doing so. If all your cells are the same size, use a nonresizable `UIImage` to mask the corners because it renders faster.

That's all for the first performance example. Take a look at the next example, called "Performance Problems Example II," in the same code. Build and run the app to get a feel for how it works.

The use case for this app is the following: You've been hired by an up-and-coming startup that just got some angel funding and who are building a social network for cats (see Figure 2.15). You are to prototype the "Facebook Wall" equivalent of their future mobile app so that they can grab millions in venture capital funding.

Figure 2.15 A social network for cats

The app displays comments in cells that have different background colors. The model is set up in `setupModel`. Just ignore this method; it is not relevant to the case study. Also, notice how Xcode lets you use emoji in Objective-C source code. How cool is that?

Profile the app and use the same Core Animation template as the last example. The peak frame rate is 48fps, which isn't terrible, but not ideal. When you open the Extended Detail pane, what you see should cause some alarm (see Figure 2.16).

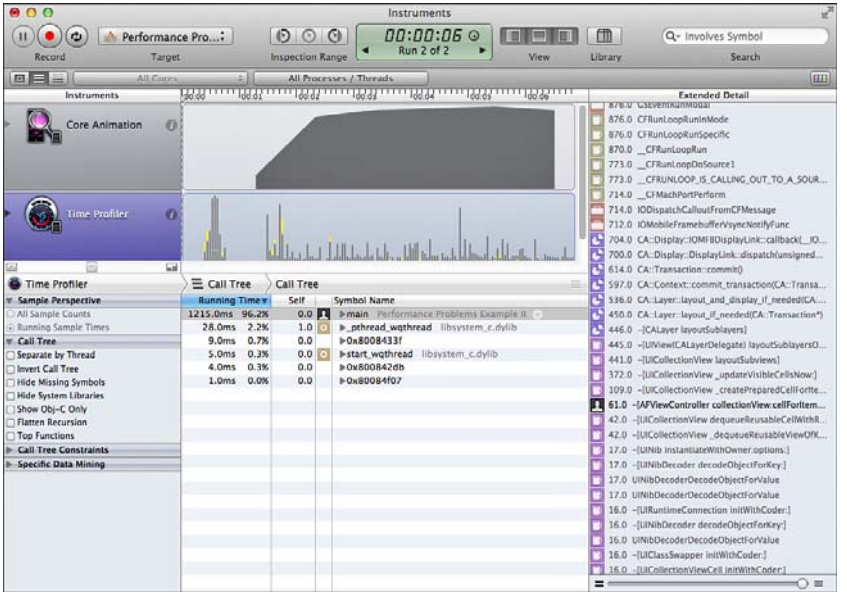

Figure 2.16 First profiling run in Instruments

The most expensive operation performance is unloading the .xib file. What's with that? Open `AFCollectionViewCell.xib` and bask in the sheer existential horror of the view hierarchy.

Obviously, Figure 2.17 represents a pedagogical example. You would never have a view hierarchy in a nib *quite* this bad. Even though you have a quite complex view hierarchy, all you're really doing is displaying some text on a colored background. You can draw this, and the equivalent of all those useless views, a lot faster in `drawRect:`. You can apply the same logic in your own cell subclasses; if you have a complex view hierarchy that takes too long to draw, implement `drawRect:` and ditch the view hierarchy. `drawRect:` can also be a slow performer, however, and using it in a simple example like this is only to illustrate how it's done. You should use it only when drawing components of your view manually is faster than rendering a necessarily complex view hierarchy.

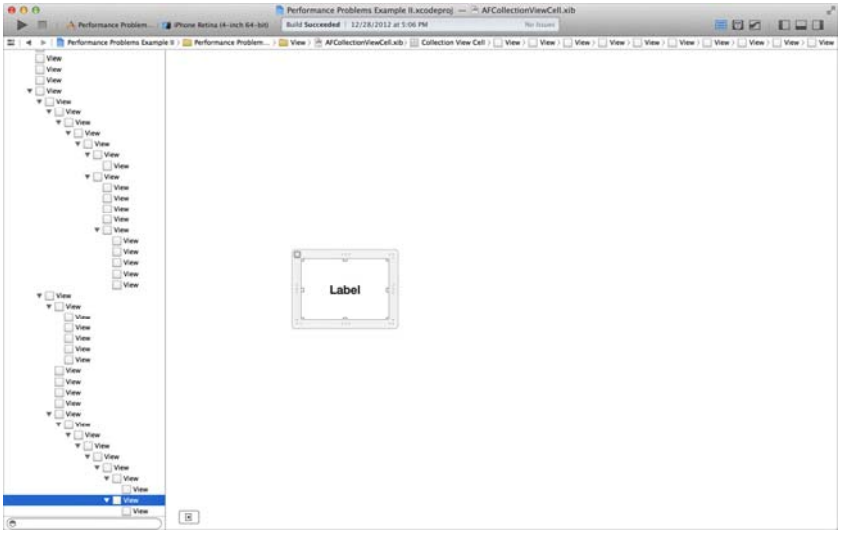

Figure 2.17 Crazy view hierarchy

Delete the .xib and the two properties from the cell's header file. Instead of registering a `UINib` in the view controller's `viewDidLoad`, register a `Class`. Instead of using a separate background view for the color, just draw it in `drawRect:`. Create a new string property for the cell's text. You're going to override the getter and setter for `backgroundColor` to do some clever drawing (see Listing 2.21).

Listing 2.21 New Cell Subclass

```
static inline void addRoundedRectToPath(CGContextRef context, CGRect rect, float
ovalWidth, float ovalHeight)
{
    float fw, fh;
    if (ovalWidth == 0 || ovalHeight == 0) {
        CGContextAddRect(context, rect);
        return;
    }
    CGContextSaveGState(context);
    CGContextTranslateCTM (context, CGRectGetMinX(rect), CGRectGetMinY(rect));
    CGContextScaleCTM (context, ovalWidth, ovalHeight);
    fw = CGRectGetWidth (rect) / ovalWidth;
    fh = CGRectGetHeight (rect) / ovalHeight;
    CGContextMoveToPoint(context, fw, fh/2);
    CGContextAddArcToPoint(context, fw, fh, fw/2, fh, 1);
    CGContextAddArcToPoint(context, 0, fh, 0, fh/2, 1);
    CGContextAddArcToPoint(context, 0, 0, fw/2, 0, 1);
    CGContextAddArcToPoint(context, fw, 0, fw, fh/2, 1);
```

```
    CGContextClosePath(context);
    CGContextRestoreGState(context);
}

@implementation AFCollectionViewCell
{
    UIColor *realBackgroundColor;
}

-(id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    self.opaque = NO;
    self.backgroundColor = [UIColor clearColor];

    return self;
}

-(void)prepareForReuse
{
    ...
    //Not relevant to this part of the case study
}

-(void)drawRect:(CGRect)rect
{
    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSaveGState(context);

    [realBackgroundColor set];

    addRoundedRectToPath(context, self.bounds, 10, 10);
    CGContextClip(context);

    CGContextFillRect(context, self.bounds);

    CGContextRestoreGState(context);

    [[UIColor whiteColor] set];

    [self.text
        drawInRect:CGRectInset(self.bounds, 10, 10)
        withFont:[UIFont boldSystemFontOfSize:20]
        lineBreakMode:NSLineBreakByWordWrapping
```

```
        alignment:NSTextAlignmentCenter];
}

#pragma mark - Overridden Properties

-(void)setBackgroundColor:(UIColor *)backgroundColor
{
    [super setBackgroundColor:[UIColor clearColor]];

    realBackgroundColor = backgroundColor;

    [self setNeedsDisplay];
}

-(UIColor *)backgroundColor
{
    return realBackgroundColor;
}

-(void)setText:(NSString *)text
{
    _text = [text copy];

    [self setNeedsDisplay];
}

@end
```

The `addRoundedRectToPath` C method is handy and can be easily modified to only round certain corners. These methods should usually be placed in a separate source file so that they can be reused. Take a look at Listing 2.22.

Listing 2.22 Decompressing Images in a Background Thread

```
-(void)configureCell:(AFCollectionViewCell *)cell withModel:(AFModel *)model
{
    cell.backgroundColor = [model.color colorWithAlphaComponent:0.6f];
    cell.text = model.comment;
}
```

Listing 2.22 is an efficient implementation. The drawing code is straightforward and the code using the cell works well within the MVC architecture. Reprofile the application.