
Design
SDK
DEVCON
SDK

RWDevCon 2018 Vault

By the raywenderlich.com Tutorial Team

Notice of Rights

Notice of Liability

Trademarks

Table of Contents: Overview

Table of Contents: Extended

Prerequisites

Some tutorials and workshops at RWDevCon 2018 have prerequisites.

If you plan on attending any of these tutorials or workshops, **be sure to complete the steps below before attending the tutorial**.

If you don't, you might not be able to follow along with those tutorials.

> **Note**: All talks require **Xcode 9.2** installed, unless specified otherwise (such as in the ARKit tutorial and workshop).

W3: Practical Instruments Workshop

Come with your oldest iOS 10 or later device! The worse the performance of the phone, the more interesting the experience for this workshop.

W3: Practical Instruments Workshop

Have you been working with iOS for a few years now but always been a little bit too nervous to jump into Instruments and try to track down some problems in your app? Maybe I'm way off and you're a little newer to the game and you're just really interested in trying to improve your app's performance.

Either way, by the end of this workshop you'll have a good feel for how to use Instruments to dive deep into what's happening while your app is working and see exactly where the bottlenecks are.

Practical Instruments Workshop: Demo 1

By Luke Parham

In this demo, you'll see how to add a `CADisplayLink` to your application in order to track frame drop events in real-time. This may or may not seem all that useful at first, but later we'll take this information and combine it with our Instruments skills to track down and fix performance bottlenecks in our app.

Also, this will totally be covered in the live demo, but if you get lost or behind, or just wanna re-read later, this is the literature to come back to!

> **Note**: Begin work with the starter project in **1-Demo1\starter**.

1) Add a Display Link

In case you missed it, a `CADisplayLink` is a special type of timer that gets fired after a vsync event has occurred. If everything is working properly, this should be every 16.67ms (or ~8.34ms on a 120hz device).

The nifty thing is, if you did too much work on any particular turn of the run loop, the timestamp for you display link will be behind for as long as you were dropping frames since the vsync wasn't allowed to occur.

Go to **AppDelegate.swift** and find `application(_:didFinishLaunchingWithOptions:)` and add the following before the `return` statement.

```
let link = CADisplayLink(target: self,
                         selector: #selector(AppDelegate.update(link:)))
link.add(to: RunLoop.main, forMode: .commonModes)
```

This creates a new display link and tells it to call `update(link:)` when the timer fires. Then you add the link to the main run loop.

1a) Using the Display Link

Next, it's time to actually use the display link you just set up. First thing, go to the top of the file and add the following line under the definition of the `window` property.

```
var lastTime: CFTimeInterval = 0.0
```

You'll use this property to keep track of the last time the method was called so you can calculate the time that has passed since then. Now, find the `update(link:)` method and add the follow code inside:

```
//1
if lastTime == 0.0 {
  lastTime = link.timestamp
}

//2
let currentTime = link.timestamp
let elapsedTime = floor((currentTime - lastTime) * 10_000)/10
```

1. Here, you're initializing the `lastTime` property to the first timestamp that was seen if necessary.
2. Then, you're grabbing the `currentTime` and calculating the time that has elapsed since last time. The multiplication by 10,000 and then division by 10 gives you a clean answer in milliseconds which is the most useful unit here.

Now that you have all the values you need, it's time to print a message if you know that you've been dropping frames.

Still inside the `update(link:)` method, add the following if-statement.

```
if elapsedTime > 16.7 {
  print("Frame was dropped with elapsed time of \(elapsedTime) at \
(currentTime)")
}
```

Here, you're just comparing your `elapsedTime` variable to 16.7 to see if you've dropped a frame. While the actual value should be something like 16.666 repeating (you scared?), the time is often reported a tiny amount above that even if it's on time, so this gives you enough wiggle room that you wont' be seeing many false positives.

Finally, and very importantly, add the final line to the method.

```
lastTime = link.timestamp
```

This will update `lastTime` so you aren't just looking at the same timestamp forever.

2) That's it!

Congrats, you now have a simple and convenient system in place for being alerted to the fact that you have frame drops in your app. This may get a little annoying if you leave it on all the time, even when not looking at performance, but it is useful since it will alert you to the slightest dip in framerate when scrolling your scroll views!

66 Practical Instruments Workshop: Challenge 1

By Luke Parham

In this demo, it's time to dive into **Time Profiler** to get a taste of how **Instruments** works in general.

> **Note**: Do your exploration with the starter project in **2-Challenge1\starter**.

1) Debug Symbols

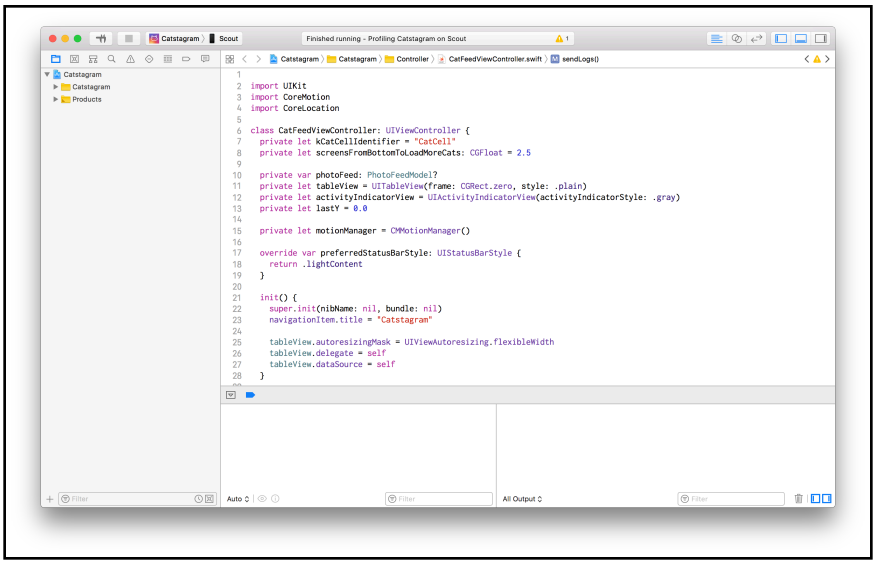
First things first, go into the **Build Settings** for the app and search for **dsym**. Make sure the **Debug Information Format** setting is set to **DWARF with dSYM File**. Without dSYM files, the traces you create will only show you memory addresses instead of actually telling you what methods are being called!

2) Firing Up Instruments

Once you have the starter project open, you have a couple ways of getting to Instruments. I'll list them for you and you can take your pick.

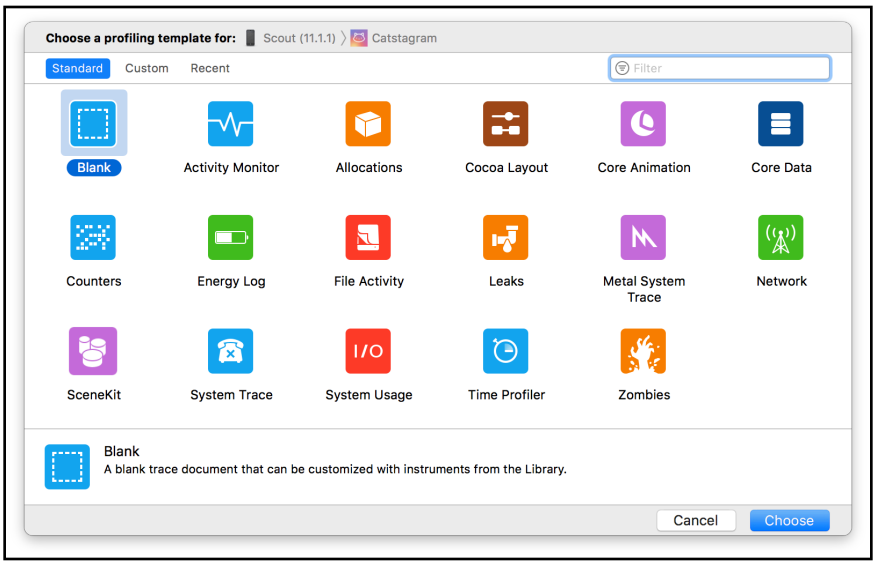
The most efficient way to do it is to press **cmd+i**. This is similar ot pressing **cmd+r** but instead of just running the application, **Instruments** will be launched.

Alternatively, you can click and hold on the **play button** in Xcode. This will show you a list of four options, one of which is **profile**. Choosing this option will change the play button icon into a wrench and subsequent clicks of the button will automatically launch **Instruments**.



You can't tell me that wrench icon doesn't look good.

Once you've chosen how you want to launch Instruments, (choose now!) you'll be presented with this view.



From here you can launch an existing template or even start with a blank one.

This brings us to an important point. The application "Instruments" is, quite literally, comprised of a number of existing tools, each one of which is called an "instrument".

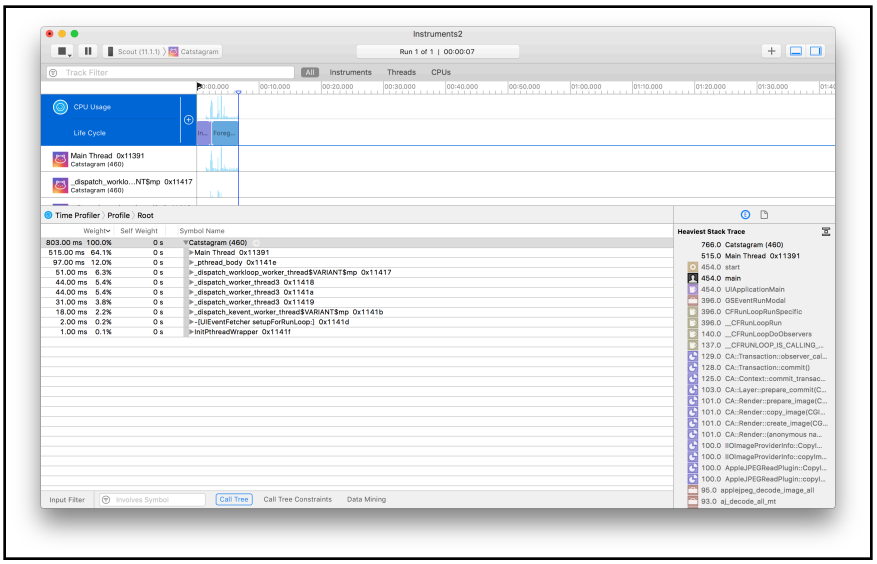
The templates listed here are pre-made collections of individual instruments that work well together to solve a particular problem. That being said, sometimes a template is actually just one instrument.

3) Taking Your First Trace

During this time, feel free to jump into any of the templates you want. The main three we'll look at today are **Time Profiler**, **Core Animation** and **System Trace**. In my opinion, **Time Profiler** is far and away the most useful, though they all have their uses depending on the situation.

Once you've explored a bit, come back to the chooser and choose **Time Profiler**.

Now, click the red **record** button to start you first trace.



To collect data, just play around with the app for a bit. It's a pretty simple app, but you can scroll through the feed and tilt to see images moving around.

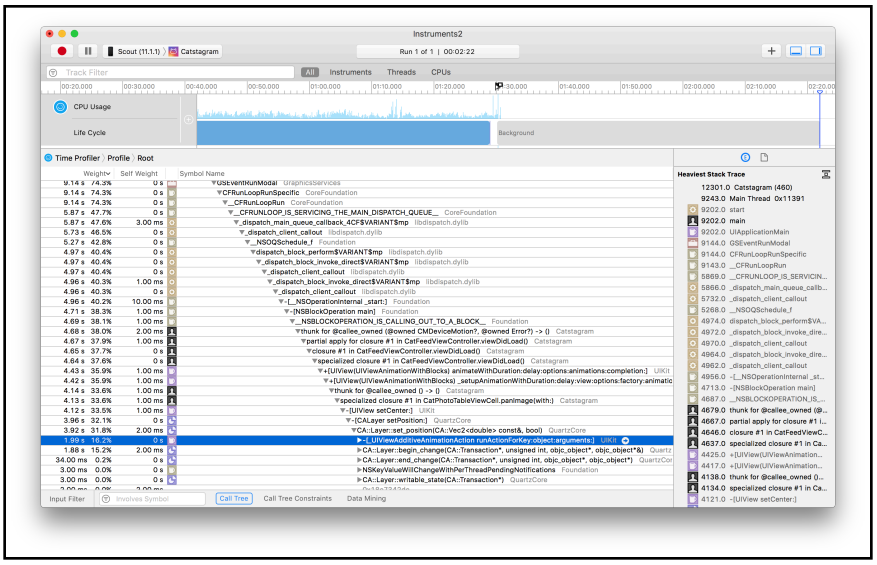
4) Exploring Trace Results

Once you've explored a bit, hit the square **stop** button to finish the trace. Now that you've collected some data, it's time to do some exploring!

Drilling Into Call Tree Branches

The main thing you'll be doing when looking at **Time Profiler** traces is drilling into the method names you see listed in the **Detail Pane**.

The most useful tip for looking through this section is to remember to hold **option** while clicking on the disclosure indicators. This will magically open the subtree up so that you're looking at the most important path.

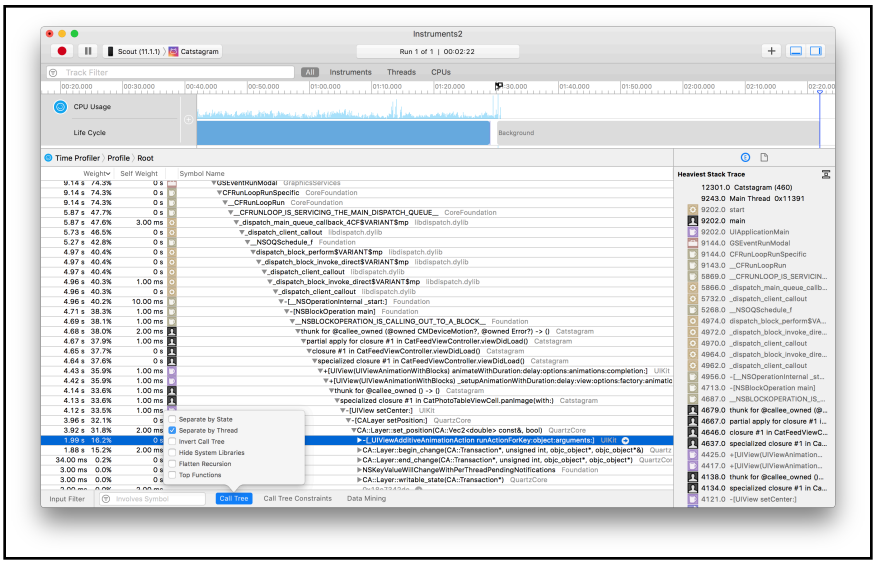


Don't ask me how they do it, but it's beautiful!

Call Tree Options

Now you might be saying to yourself, "Man, this is really a lot of information to dig through, can't I filter some of this out?". Luckily for you, there totally is!

Go to the bottom of the trace window and find the **Call Tree** button.



This button gives you a few options re-arranging or paring down the data you're looking at.

Of particular importance are **Hide System Libraries** and **Separate by Thread**.

The first allows you to filter out all of Apple's code so you only have to look at things you've written. This can be extremely useful, but be careful! Just because

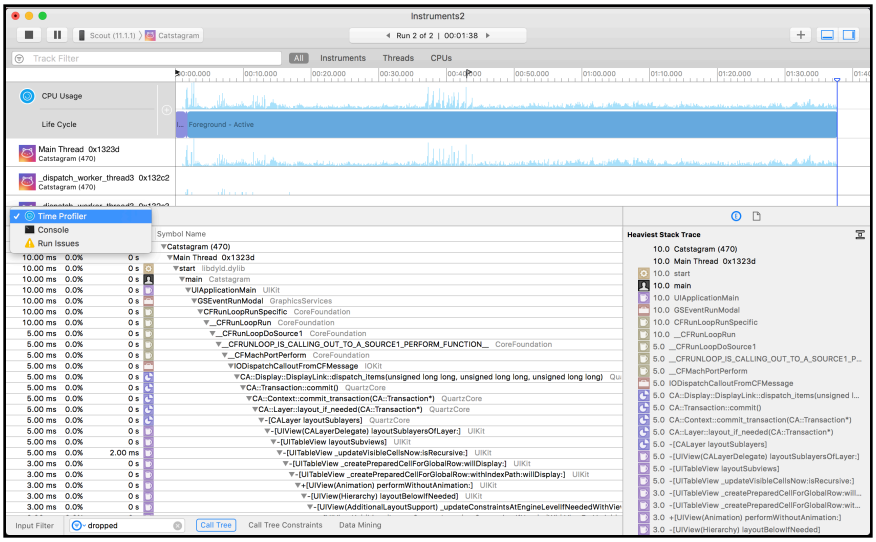
you didn't write a certain method doesn't mean it doesn't exist and ignoring system libraries all the time means you'll definitely miss important work that your code has triggered.

Separate by Thread is useful because it allows you to drill into what work you're doing on the **Main Thread** instead of seeing everything that's going on all together.

This is more often totally fine to leave on, though there are times when a lot of work in the background will affect your app's rendering even if it looks like the **Main Thread** isn't doing much.

5) Looking at Dropped Frames

Finally, go to the top of the **Detail Pane** and click on **Time Profiler**. This will open up a menu where you can choose what you see in the pane's output.

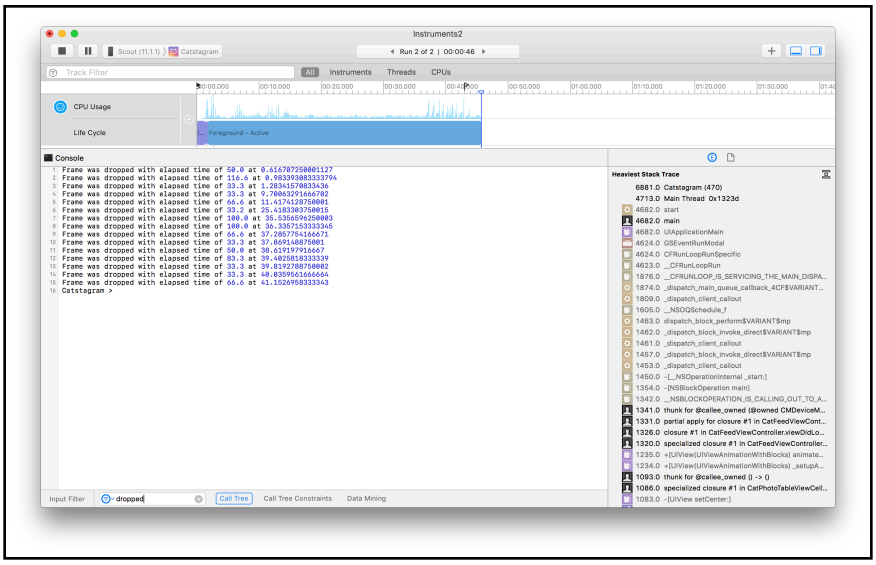


Choose **Console** to see all logs that have been printed to the console during the profiling session.

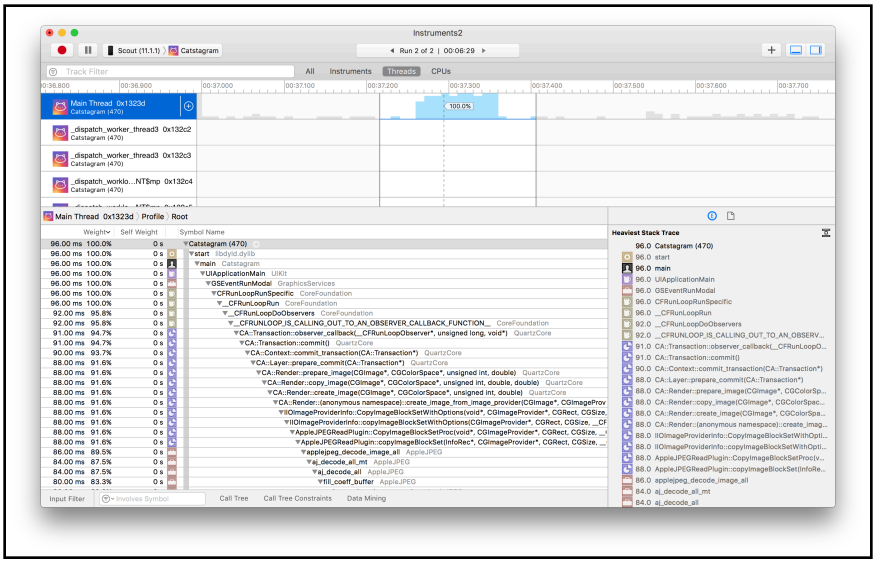
If your logs are like the logs in my app, there's probably a decent amount going on.

At the bottom of the screen there's an **Input Filter** where you can enter text to limit which logs are seen here.

Enter the word **dropped** to only show the frame drop events you started logging in the last demo.



Doing this allows you to take the timestamps you see here and zoom into problem areas in the timeline pane.



6) That's it!

That's it for now! The main thing to remember while you're exploring **Time Profiler** is that it's ok if you're feeling a little overwhelmed. It just takes time to get comfortable looking through these traces, but with a little practice you'll be using them to track down bottlenecks in no time.

67 Practical Instruments Workshop: Challenge 2

By Luke Parham

In this challenge, it's your job to use Time Profiler to actually hunt down some bottlenecks and try to think of some solutions.

> **Note**: Do your exploration with the starter project in **3-Challenge2\starter**.

1) Finding Bottlenecks

In this application, there are at least 3 main sources of work that's being done. For each one you can find, write it down on one of the lines and then below the line write some possible ways to make it faster.

68 Practical Instruments Workshop: Demo 2

By Luke Parham

In this demo, we'll walk through the trace we've taken, and solve some of the more straight-forward bottlenecks we found.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Make sure to use the project found in the **4-Demo2\starter** folder.

1) Getting to Your Home

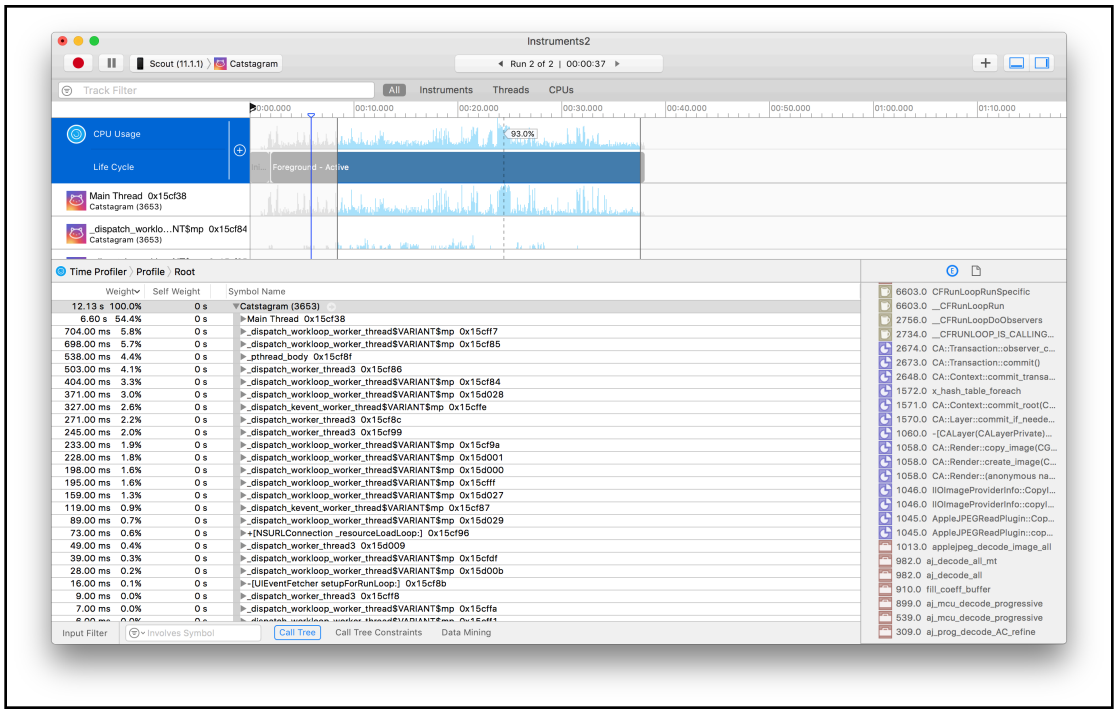
First, use **cmd+i** to start another profiling session. Choose **Time Profiler** from the template options and make sure the app has launched properly.

To fully exercise the app, scroll up and down a bit, let some cats load in and start tilting back and forth while scrolling.

After doing this for a bit, hit the stop button to complete your trace.

Your goal, in case it isn't obvious at this point is to reduce the number of "frame drop" events that occur on average during a run.

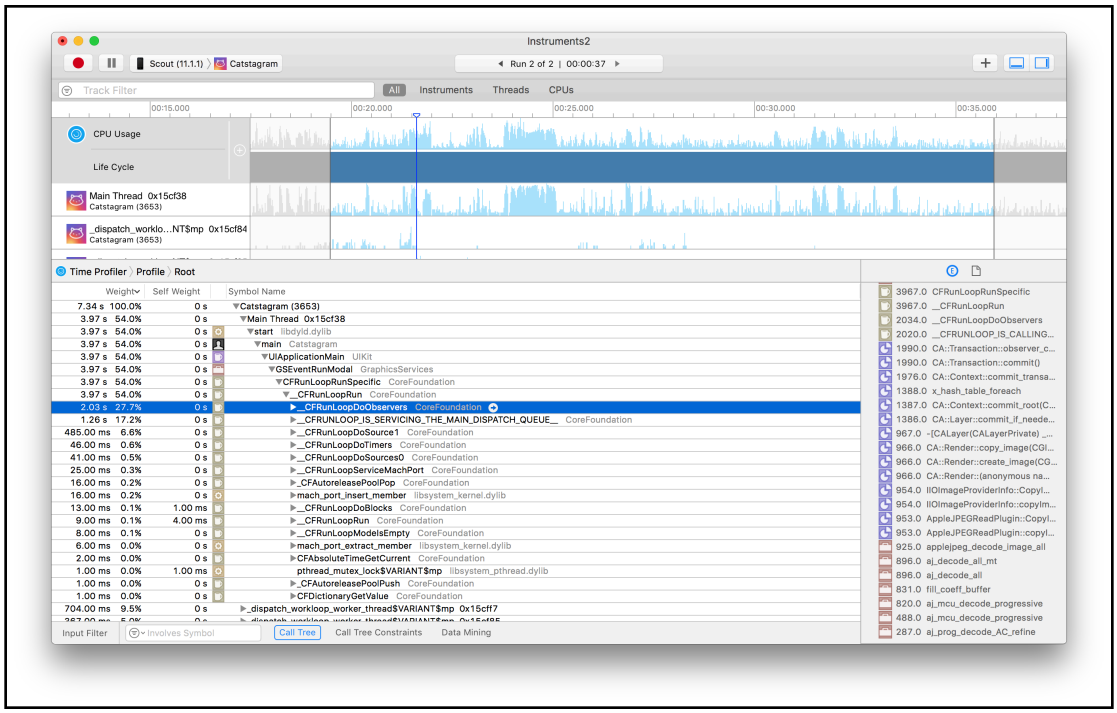
Since actual frame drops happen primarily during scrolling, go to the timeline pane and click and drag to focus on the second half of the trace only. This will remove any methods that have to do with launching the application which will let you focus on things that affect scroll performance.



Next, you'll see what's happening on the **Main Thread** since extra work there is the main factor when it comes to reducing frame drops.

Go to the little triangle (called a "disclosure indicator") next to **Main Thread** in the results panel, hold the **option** key and then click on it.

This little bit of macOS magic allows you to drill down into the most significant sections of the subtree automatically.



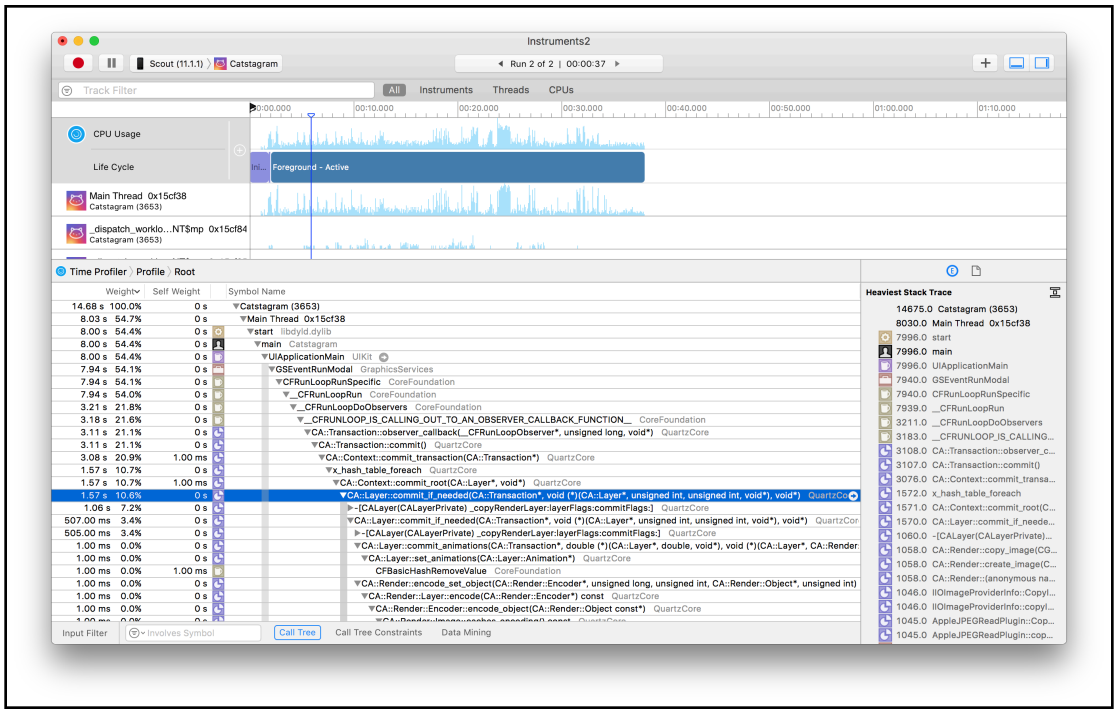
This trick is especially helpful when digging through the calltree since there's often quite a few method names standing between you and what you're looking for.

The exact results of your trace will be different than mine depending on how exactly you interacted with the app and what you focused on in the timeline pane, but odds are you should find yourself at the place you should end up feeling most comfortable, the **_CFRunLoopRun** callbacks.

As I mentioned in the demo, a lot of times you can think of each of these as a path that may lead to a potential bottleneck.

2) Narrowing Things Down

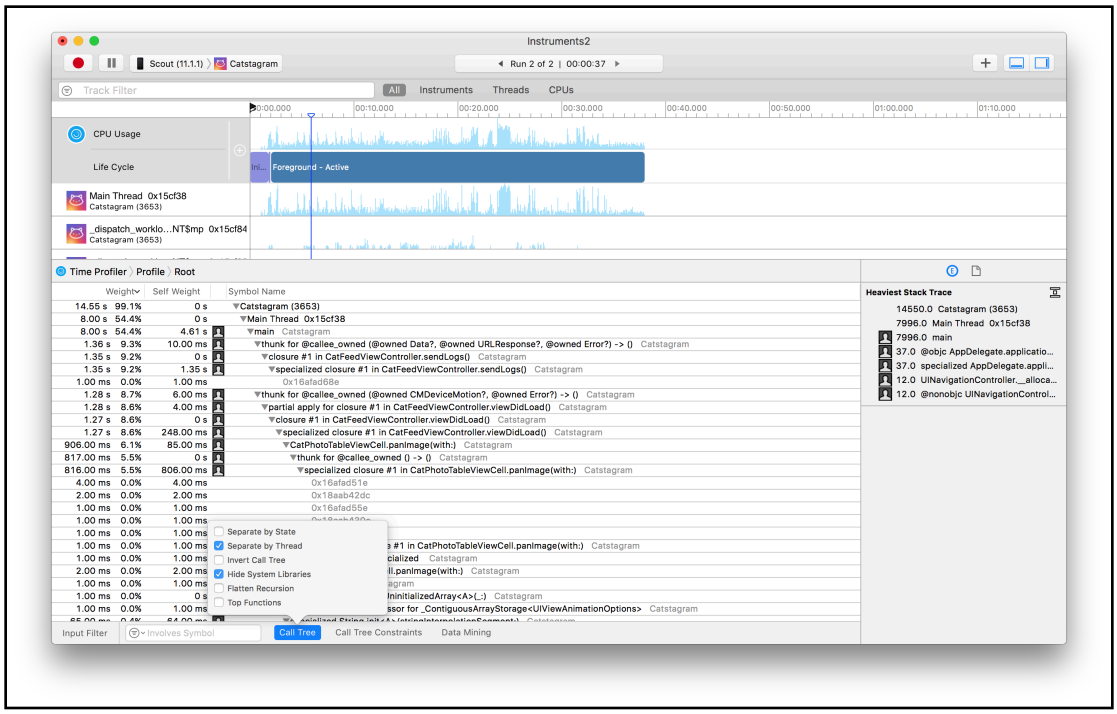
If you click into the first run loop callback, you may run into a trace that looks a little something like this.



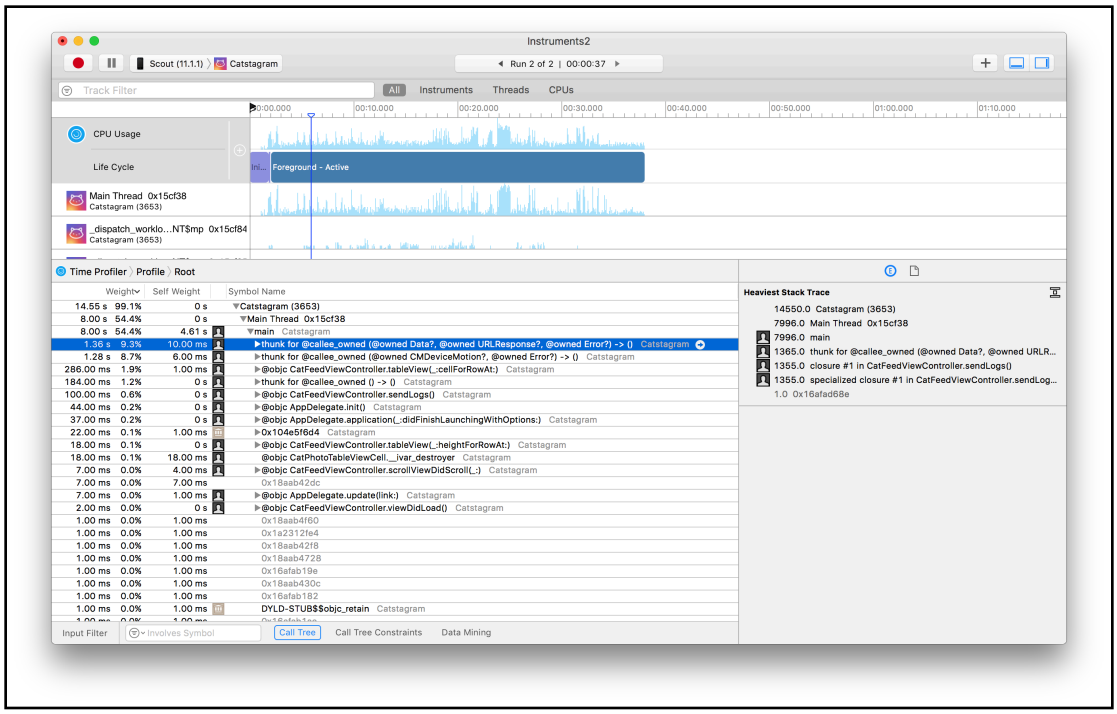
This path is full of internal `CoreAnimation` methods that you haven't called directly, and let's be honest, they're a bit much to look at.

Luckily for you, there's an easy way to rid yourself of all this noise.

Go down to the **Call Tree** menu, click it, and select **Hide System Libraries**.



Now, go to **Main Thread** in the call tree, **option+click** to close the tree, and then **option+click** again to re-open the tree under these new circumstances.



This shows a noticeably readable trace. You have **main** and then right below that you have a callback having to do with networking accounting for ~9% of the work, an callback for motion at ~8% and the `tableView(_:cellForRowAt:)` method accounting for ~2%.

Since this is all code written directly by your (or a teammate), it should be relatively easy to make some improvements.

3) Sending Logs

The key to fixing the first bottleneck is easier than it might seem. This `sendLogs()` method is something that doesn't need to be called as often as it currently is. In general, if you have recurring network events such as sending off logs, it's better to batch these events together and do them all at once.

Not only do events like this take precious CPU cycles, having to power up and then power down the phone's cellular systems has a very large impact on battery life.

To fix this bottleneck, go to the `init()` method in **CatFeedViewController.swift** and **remove** the timer that's been calling this method.

```
// Remove this code
let timer = Timer(timeInterval: 0.1, target: self, selector:
#selector(CatFeedViewController.sendLogs), userInfo: nil, repeats: true)
```

```
RunLoop.main.add(timer, forMode: .commonModes)
```

Next, go to **AppDelegate.swift** and find the empty `applicationWillResignActive(_:)` method.

Add the following line to the method in order to send the logs only once, when the app is leaving the active state.

```
rootVC.sendLogs()
```

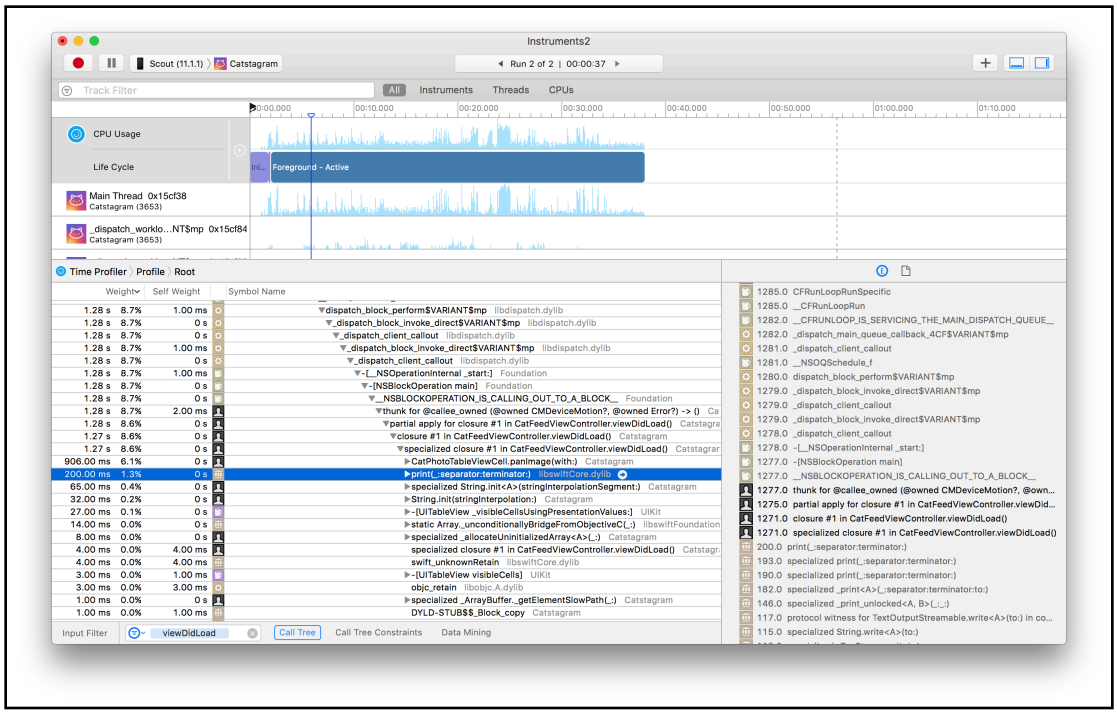
3) The Motion Callback

The next issue seems to be the motion callback in the `viewDidLoad()` method of `CatFeedViewController`.

With system libraries hidden it's not immediately obvious what is happening in this method.

Toggle this setting to reveal all libraries, and then type **viewDidLoad** into the **Input Filter** in the bottom left hand corner.

Once again, do a smart close and smart open on the **Main Thread** and take a look at the results.



As you can see, there seems to be a rogue `print(_:separator:terminator)` in the list of methods called by the closure. On top of that there's a few lines of string manipulation code that's probably related.

Go over to `viewDidLoad()` in `CatFeedViewController` and, as expected, there's an extra print call that was left in this method by accident.

Remove the following line from the motion callback block.

```
print("y \(yRotationRate) and x \(xRotationRate) and z\(zRotationRate)")
```

This might not be a mind blowing performance hack, but you just removed what added up to 2% of the work in your trace.

At the end of the day, a lot of times you're going to find you're accidentally doing a little more work than you have to, especially if you're working with a legacy codebase.

Next, just for good measure, you can reduce the frequency with which this callback is called. Setting it to 0.2 means it will only be called 5 times per second which will reduce the number of times this animation method is called without affecting how it looks to the user.

Add the following line to `viewDidLoad()` just before the call to `startDeviceMotionUpdates(to:withHandler:)`.

```
motionManager.deviceMotionUpdateInterval = 0.2
```

Now that you've made some solid improvements to your code, go ahead and profile again to validate your hypothesis that these changes have improved things.

5) Hmm...

After you've made all these changes, you might expect that your average framedrop numbers will have improved significantly. Alas, your numbers haven't really improved all that much, if at all.

Now to be fair, the changes you made were still good changes! Your codebase has still been improved by your hard work, but it's not quite where you want it to be yet.

Maybe there's something more going on here. Seems like you'll have to keep digging...

69 Practical Instruments Workshop: Demo 3

By Luke Parham

In this demo, we'll remove what is by far the biggest bottleneck on your app's main thread.

The steps here will be explained in the demo, but here are the raw steps in case you miss a step or get stuck.

> **Note**: Make sure to use the project found in the **5-Demo3\starter** folder.

1) Creating the AsyncImageView

Our goal here is to move all of the indirect jpeg decoding that's currently happening on the main thread onto a background thread.

We're going to do that by writing our own version of `UIImageView` called `AsyncImageView`.

First, make sure you have the **View** folder hightlighted and create a new file called **AsyncImageView.swift**.

Next, replace the contents of that file with this class definition.

```
import UIKit

class AsyncImageView: UIView {
  private var _image: UIImage?

  var image: UIImage? {
    get {
      return _image
    }

    set {
```

```
        _image = newValue
      }
    }

    //Decode Images

  }
```

First, you've defined a private `_image` property which will hold the `UIImage` that gets passed in.

Then, you have a public computed property that has a getter and setter.

Believe it or not, you're halfway there.

2) Decoding JPEGs

Next, add the following method definition to the class:

```
  func decodedImage(_ image: UIImage) -> UIImage? {
    guard let newImage = image.cgImage else { return nil }

    return nil
  }
```

In this method, you'll take an image that's passed in and manually decode it yourself. There isn't a very straightforward way to do so, but it can be accomplished by drawing your image into a `CGBitmapContext`.

There's a few pieces involved to get to that point and the first is to create a color space object. To do so, add the following line to the method, before the `return` statement:

```
  let colorSpace = CGColorSpaceCreateDeviceRGB()
```

Next, below this line, create a new bitmap context.

```
  let context = CGContext(data: nil,
                          width: newImage.width,
                          height: newImage.height,
                          bitsPerComponent: 8,
                          bytesPerRow: newImage.width * 4,
                          space: colorSpace,
                          bitmapInfo:
  CGImageAlphaInfo.noneSkipFirst.rawValue)
```

This chunk of code is maybe a little cargo-culty but it basically defines a reasonable context into which your images can be rendered.

Then do the actual drawing by adding the following line:

```
context?.draw(newImage, in: CGRect(x: 0, y: 0,
              width: newImage.width, height: newImage.height))
```

Finally, add the following lines to finish off the method.

```
if let drawnImage = context?.makeImage() {
  return UIImage(cgImage: drawnImage)
}
```

Here you return a fully decoded image if it was successful, otherwise you return nil.

3) The Async Part of Things

Now that you have a method to decode images, it's time to jump to a background thread and use it.

First, add the following two lines to the setter you defined earlier.

```
layer.contents = nil
guard let image = newValue else { return }
```

The first line is to make sure you don't have old images hanging around while the new one is decoding and the second line is to make sure an image was actually passed in.

Next, add the following code block.

```
//1
DispatchQueue.global(qos: .userInitiated).async {
  //2
  let decodedImage = self.decodedImage(image)

  //3
  DispatchQueue.main.async {
      self.layer.contents = decodedImage?.cgImage
  }
}
```

1. First, you jump to the `.userInitiated` quality of service global concurrent queue. This is the second highest priority queue so the work you submit to it will be done relatively quickly.
2. Then you do the actual work by creating a decoded version of the image that was passed in.
3. Finally, you bounce back to the main thread and set the `layer`'s contents to be the `cgImage` backing the decoded image.

The last line is pretty interesting. The `contents` property of a layer defines what's rendered to the screen which means, at the end of the day, any text or images you

see on your phone are most likely just `CGImage`'s that have been rendered and set to be the contents of some layer backing a `UIView`.

4) Using the AsyncImageView

The final step is to actually use the view you just created.

Since you were so forward thinking, all you have to do is go to **CatPhotoTableViewCell.swift** line 36 and replace

```
var userAvatarImageView = UIImageView()
var photoImageView = UIImageView()
```

with

```
var userAvatarImageView = AsyncImageView()
var photoImageView = AsyncImageView()
```

And that's it! Build and run the app to see a significantly reduced number for frame drop events while scrolling the cat table.

70 Practical Instruments Workshop: Challenge 3

By Luke Parham

In this demo, you'll overcome a harsh reality. The truth is, nothing comes for free and a lot of solutions end up having trade-offs whether you realize it or not.

In the case of your async image views, not only do the images take a little longer to pop in, but when you scroll back up they aren't ready and waiting like they were before! Instead they're decoded all over again each time they scroll onto the screen.

> **Note**: Make sure to use the project found in the **6-Challenge3\starter** folder.

1) Adding Caching

An easy way to fix this problem is to add a layer of caching to your app. Doing all that work to decode images, only to have to do it again moments later isn't a great engineering decision after all.

This cache will specifically be a global cache that all `AsyncImageViews` can use to store and re-use images.

Navigate to **AsyncImageView.swift** and add the following extension to the bottom of the file.

```
extension AsyncImageView {
  // 1)
  struct StaticCacheWrapper {
    static var cache = NSCache<AnyObject, AnyObject>()
  }
  // 2)
  class var globalCache: NSCache<AnyObject, AnyObject> {
    get { return StaticCacheWrapper.cache }
```

```
      set { StaticCacheWrapper.cache = newValue }
    }
  }
```

1. First, you create a struct to wrap an `NSCache` instance.
2. Then, you make a `globalCache` class variable that will allow all `AsyncImageViews` to access the `NSCache`.

Next, you just have to use the cache in the method you wrote to decode and return images.

First, go to `decodedImage(_:)` and add the following lines after the `guard` statement.

```
let cachedImage = AsyncImageView.globalCache.object(forKey: image)
if let cachedImage = cachedImage as? UIImage {
  return cachedImage
}
```

Here, all you're doing is using the passing in image as a key to grab the decoded image. If this is successful, then that means this image has already been decoded and can be returned without doing any work!

Next, once you've finished decoding an image, you should probably add it to the cache if you ever want to see any benefits from the lines you just added.

To do so, go to the final `if-let` in the method and right after

```
let decodedImage = UIImage(cgImage: drawnImage)
```

Add the following:

```
AsyncImageView.globalCache.setObject(decodedImage, forKey: image)
```

And that's it! Now you won't be decoding these pesky images any more than you need to.

2) Not Crashing from Memory Pressure

This is a bit of a bonus since, in this particular case, you're probably not going to see crashing here unless you're on a relatively old device.

The big idea is that, when memory warnings are issued, you're pretty much required to drop everything you're doing and dump memory.

A bit of a catch is that memory warnings are issued on the main thread. This means that even if you're clearing memory on main, this `AsyncImageView` will happily continue allocating big bitmaps during a memory pressure event.

This does not make the system happy and can lead to your app's untimely death.

Luckily there's an easy fix. Just add the following in the `AsyncImageView`'s image setter method at the very beginning of the async dispatch call.

```
DispatchQueue.main.sync { }
```

This will perform an empty block on the main thread, and sit and wait until it's complete.

Since the memory warning and any memory clearing operations will be on the the main queue ahead of this block, your app will be effectively blocked from making any big allocations at a crucial time.

Practical Instruments Workshop: Demo 4

By Luke Parham

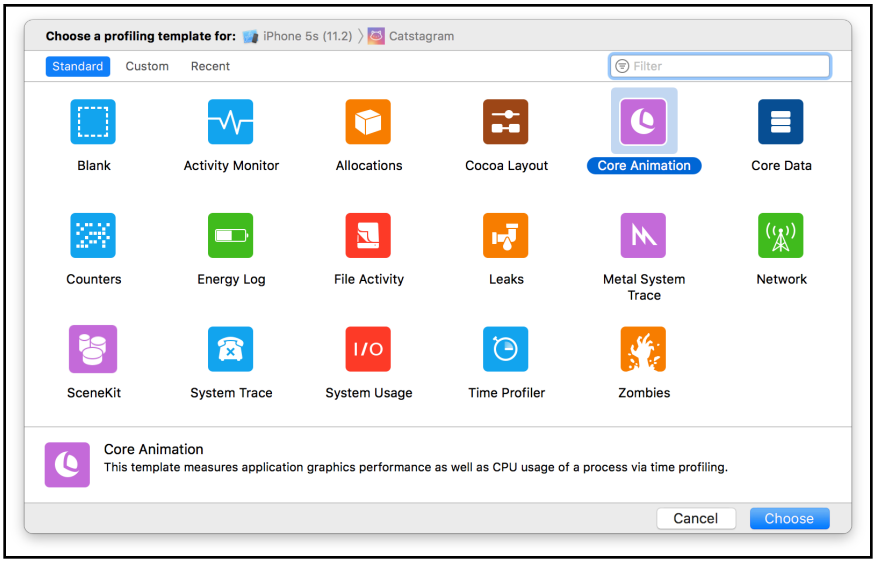
In this demo, you'll see how you can easily stop your app from doing unnecessary blending.

> **Note**: Make sure to use the project found in the **7-Demo4\starter** folder.

1) Getting Started

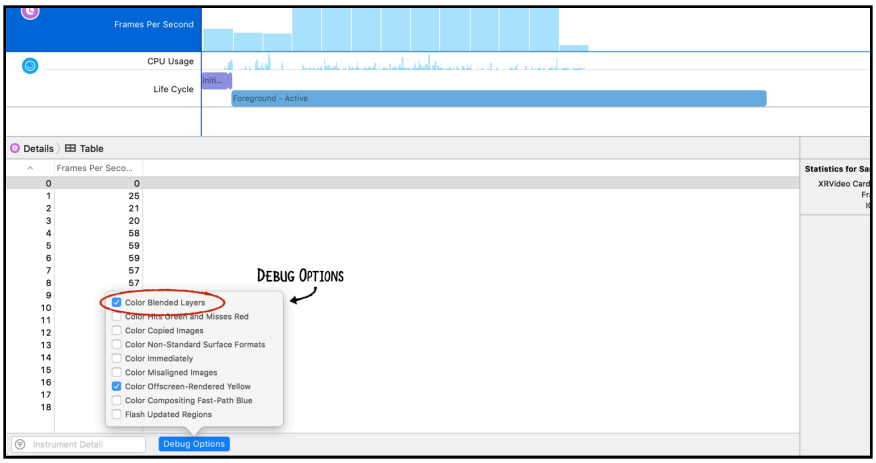
Go ahead and open up the starter app, and hit **cmd+i** to launch Instruments.

This time, select the **Core Animation** template.

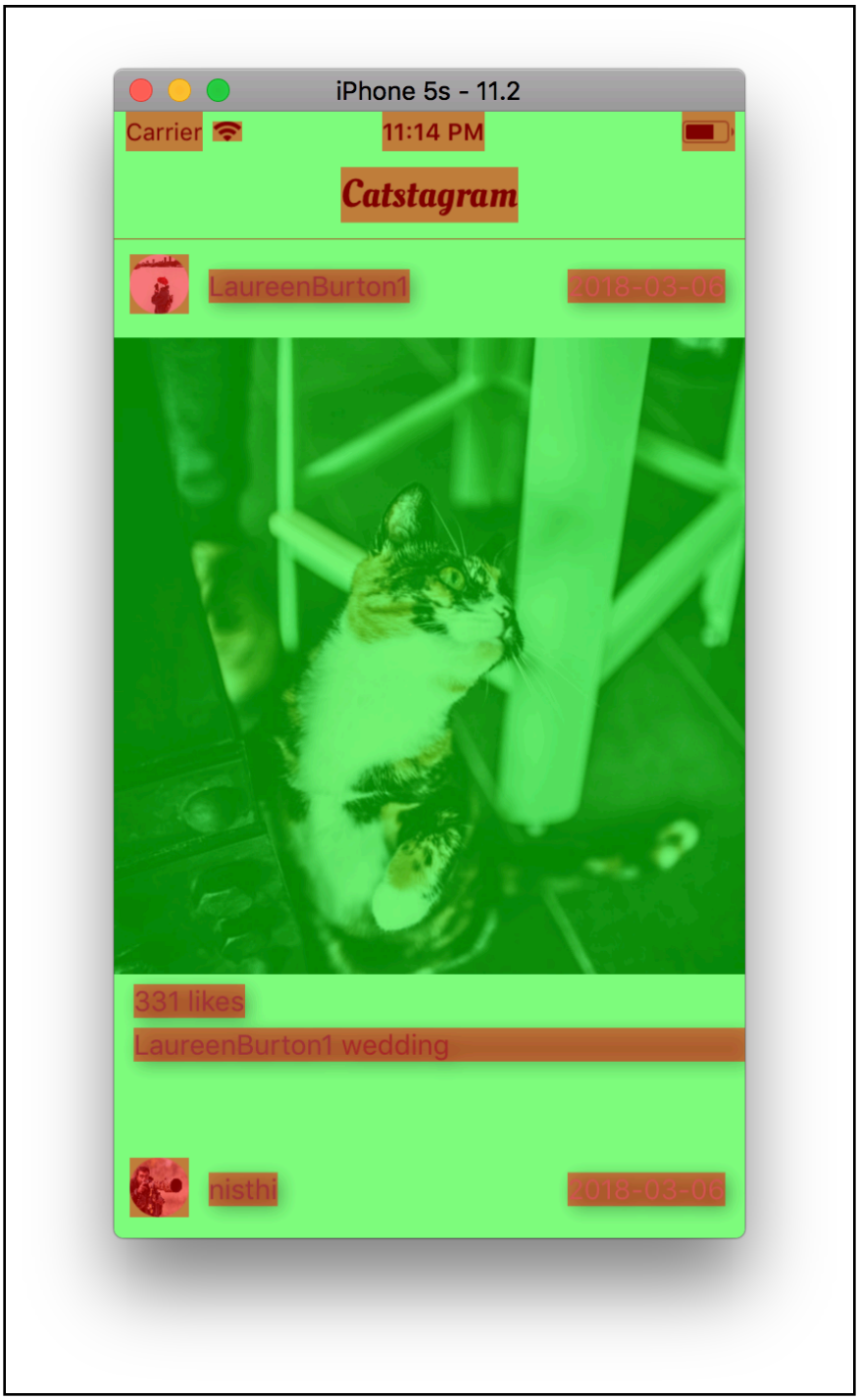


Once the profile starts, you can go to the bottom and open the **Debug Options** overlay.

When you start using the Core Animation instrument, the first thing you'll want to do is to turn on "Color Blended Layers" in the debug options.



Once you've done so, any colors onscreen that have been produced as a result of blending will be tinted **red**.



In this case, the blending is happening between two identical colors, the white table view cells and the white background of the text.

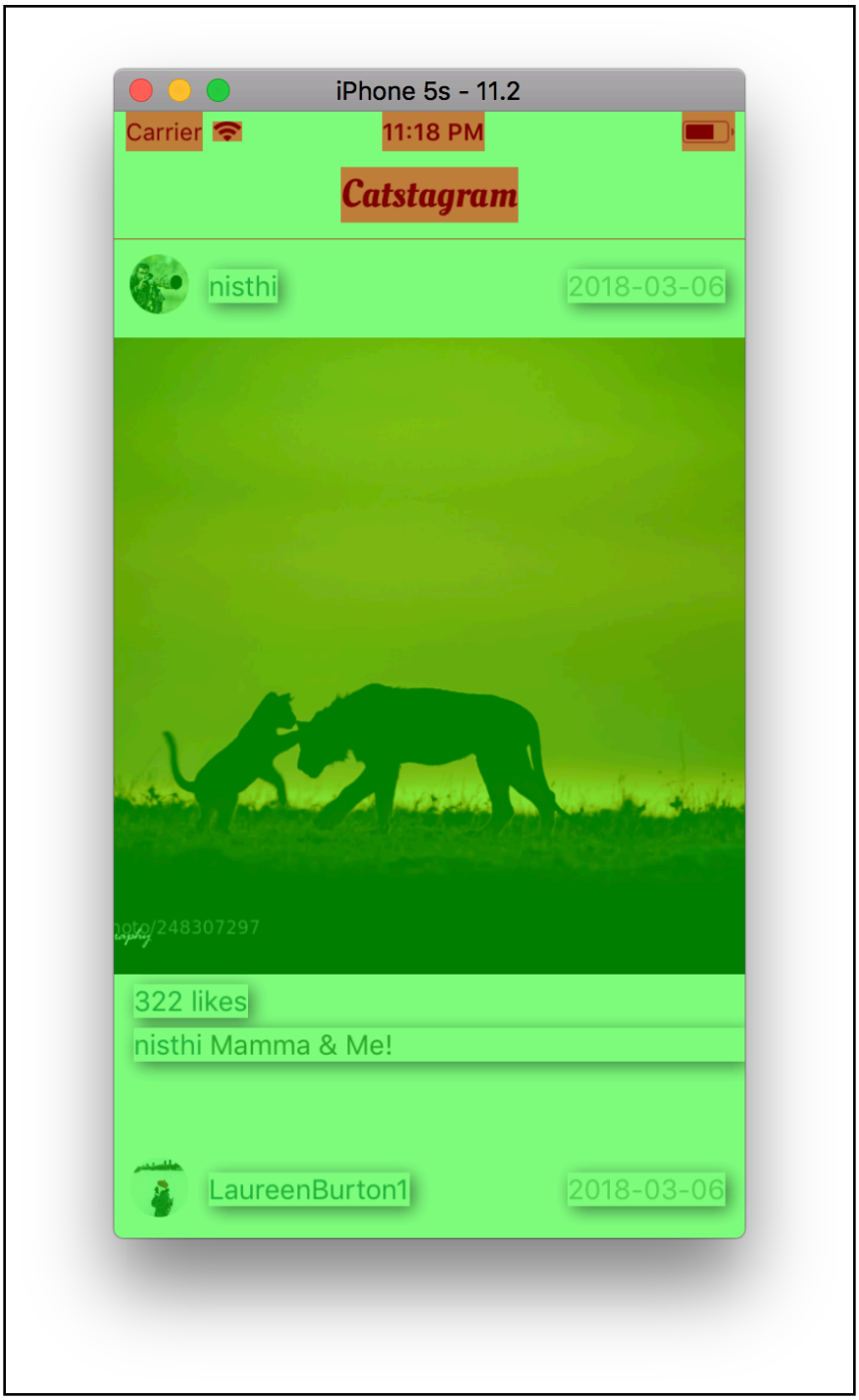
When that is the case, make sure both views have opacity and alpha set to 1.0.

In this case this is already true, but the labels have a background color of `.clear` by default.

To fix it, find the ***Demo 4*** comment and add the following lines.

```
userNameLabel.backgroundColor = .white
photoTimeIntervalSincePostLabel.backgroundColor = .white
photoLikesLabel.backgroundColor = .white
photoDescriptionLabel.backgroundColor = .white
userAvatarImageView.backgroundColor = .white
```

This code gives all of the offending views white backgrounds and will stop the alpha blending.



"But...but now our shadows are all jacked up!", I hear you thinking.

Yeah, you're right, but the alpha blending is fixed and that's all we care about for now. Don't worry, we'll circle back and fix the shadows in the next section.

72 Practical Instruments Workshop: Demo 5

By Luke Parham

When showing views with rounded corners, it's pretty easy to introduce offscreen rendering. Since this can be detrimental to performance, especially on older devices, it's time to learn how to do things a better way.

1) Getting Started

The main goal is to never use the `.cornerRadius` property of a `CALayer`. Instead, you'll pre-draw circular `UIImages` and use those instead.

Making Circular Images

First, go to **UIImage+Extensions.swift** and add the following method definition.

```
func circularImage(withSize size: CGSize) -> UIImage {

}
```

The first thing you'll want to do in this method is grab the screen scale and the bounds of the image.

Then, you'll use those to create a new image context into which you'll draw your circular image.

```
let scale = UIScreen.main.scale
let circleRect = CGRect(x: 0, y: 0, width: size.width * scale, height:
size.height * scale)

UIGraphicsBeginImageContextWithOptions(circleRect.size, false, scale)
```

Next, you create a bezier path that will represent the circle inside of which the image will be visible when you draw.

You define it using the `circleRect` you created earlier as well as a **cornerRadius**

that is equal to half the image's width.

```
let circlePath = UIBezierPath(roundedRect: circleRect, cornerRadius:
circleRect.width/2.0)
```

Then, you call `addClip()` on the path. This method is convenient, but kind of weird if you don't know what it's doing.

Calling it causes the current graphics context on the top of the graphics stack to be clipped using the `UIBezierPath` that you called the method on. This is what keeps the drawing from happening outside of the circle defined by the path.

Then you can call draw using the bounds to render a circular version of the image in memory.

```
circlePath.addClip()
draw(in: circleRect)
```

Finally, you can get a reference to the rendered image by calling `UIGraphicsGetImageFromCurrentImageContext()`.

```
let roundedImage = UIGraphicsGetImageFromCurrentImageContext()
return roundedImage!
```

Using Circular Images

Now, go back to **CatPhotoTableViewCell.swift** and find the **Demo 5b** comment.

Here, you can replace

```
self.userAvatarImageView.image = image

self.userAvatarImageView.layer.cornerRadius =
self.userAvatarImageView.bounds.width/2.0
self.userAvatarImageView.clipsToBounds = true

self.userAvatarImageView.layer.shouldRasterize = true
```

with

```
DispatchQueue.global(qos: .userInitiated).async {
  let roundImage = image.circularImage(withSize: image.size)
  DispatchQueue.main.async {
    self.userAvatarImageView.image = roundImage
  }
}
```

Here, you jump to the background so you don't block the main thread while drawing, then bounce back to main once your image has been successfully drawn with round corners.