Stanford CS193p

Developing Applications for iOS
Winter 2015

Today

- Addendum to Autolayout Demo

  How do we make views appear in some Size Classes, but not others?

- Scroll View

  Displaying big things on a small screen

- Multithreading

  Keeping the main (UI) thread clear and unblocked

Autolayout Addendum

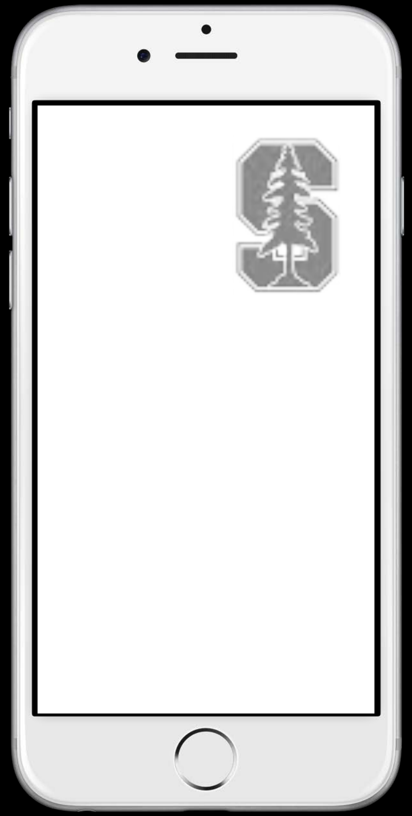
- Two minor things

  Controlling whether a view appears or not in a given size class

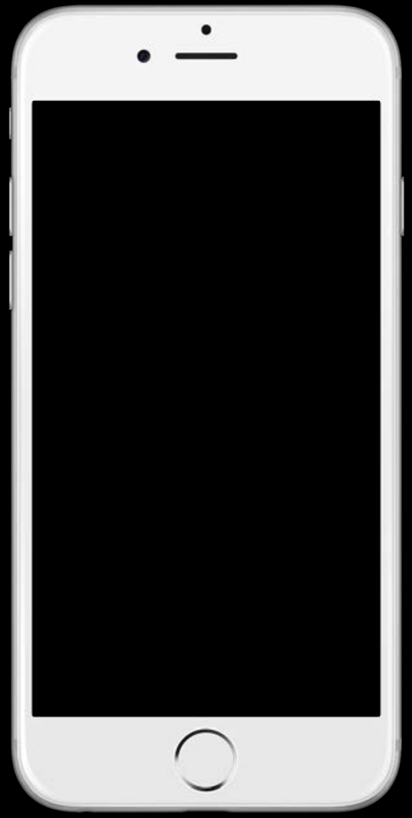
  How to "inspect" what constraints are in a given size class

Adding subviews to a normal UIView …

```
logo.frame = CGRect(x: 300, y: 50, width: 120, height: 180)
scrollView.addSubview(logo)
```



Adding subviews to a UIScrollView ...

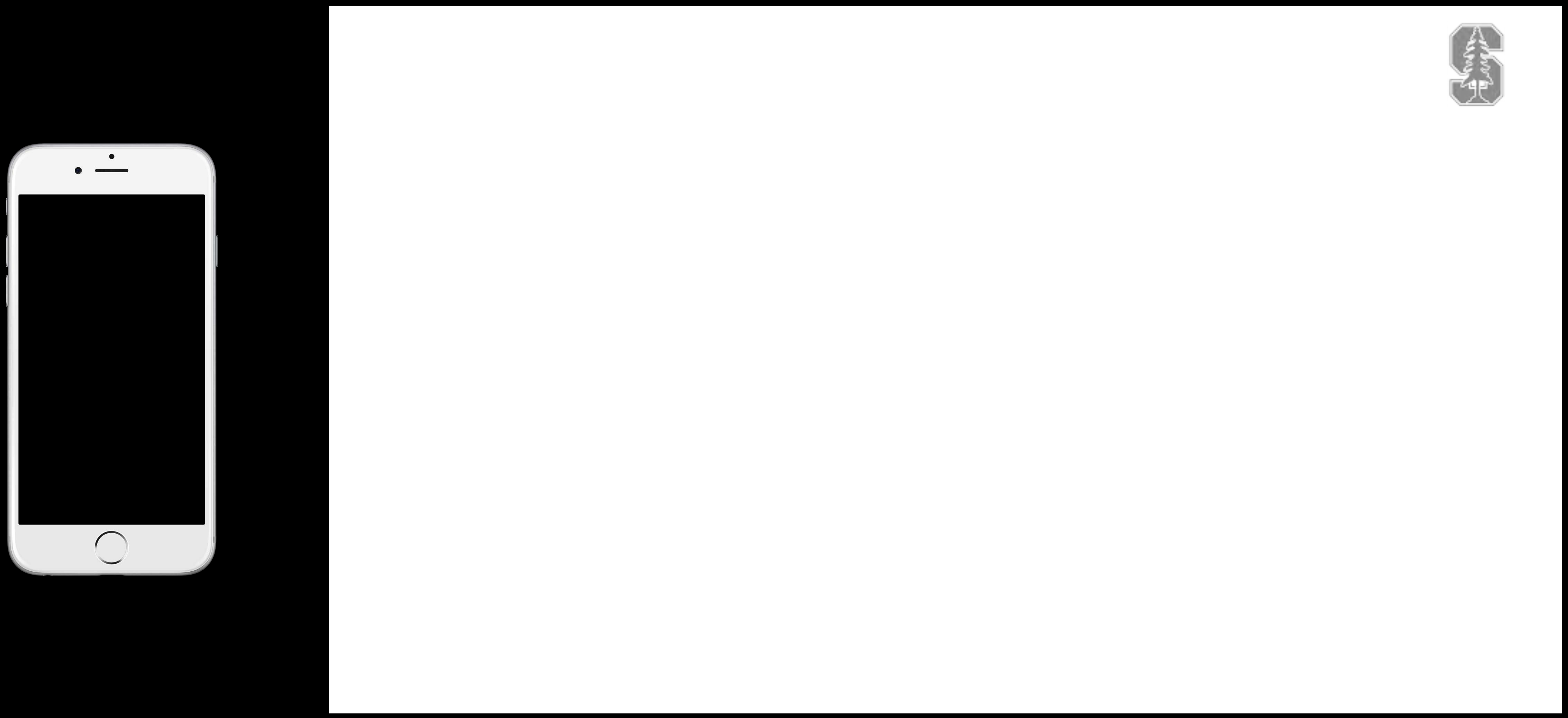


Adding subviews to a UIScrollView …

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
```

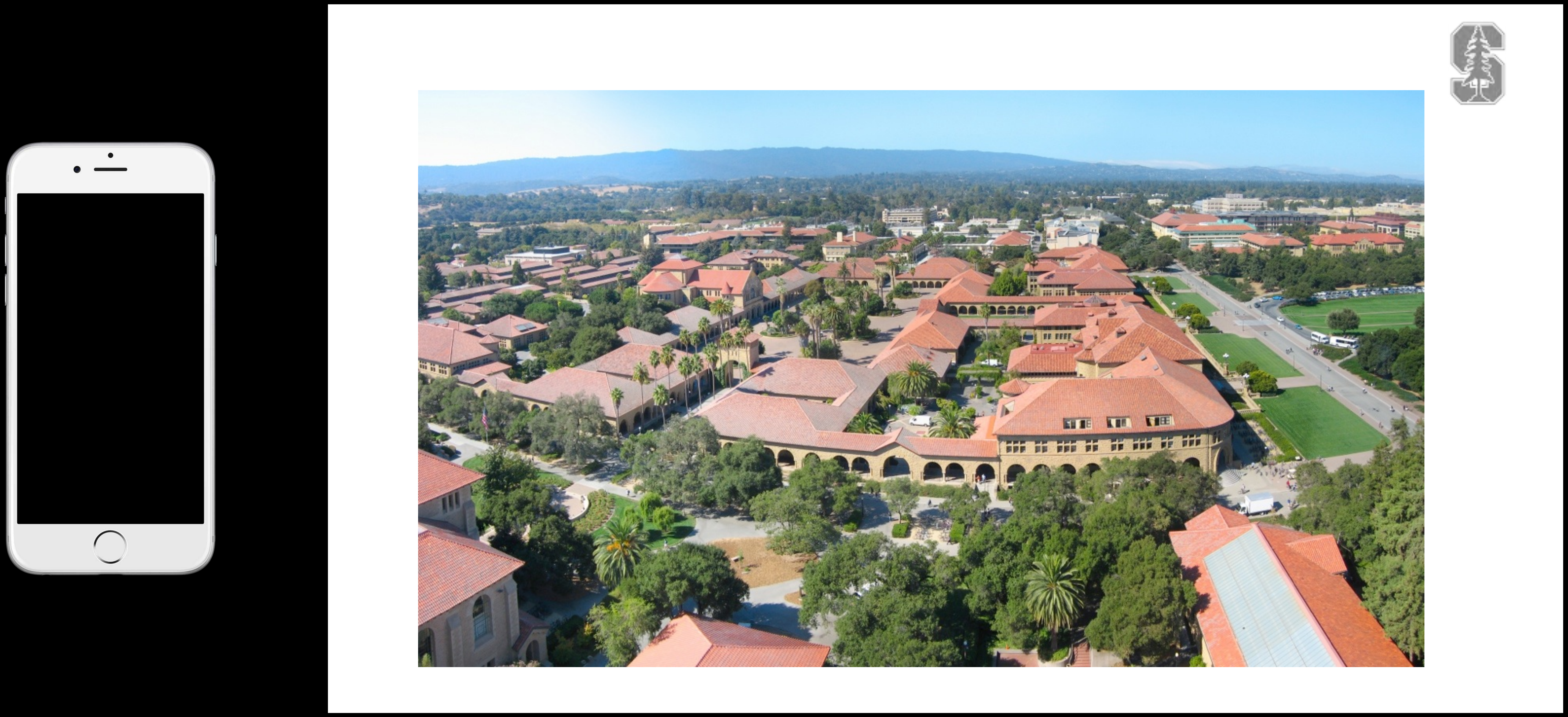
Adding subviews to a UIScrollView …

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
logo.frame = CGRect(x: 2700, y: 50, width: 120, height: 180)
scrollView.addSubview(logo)
```



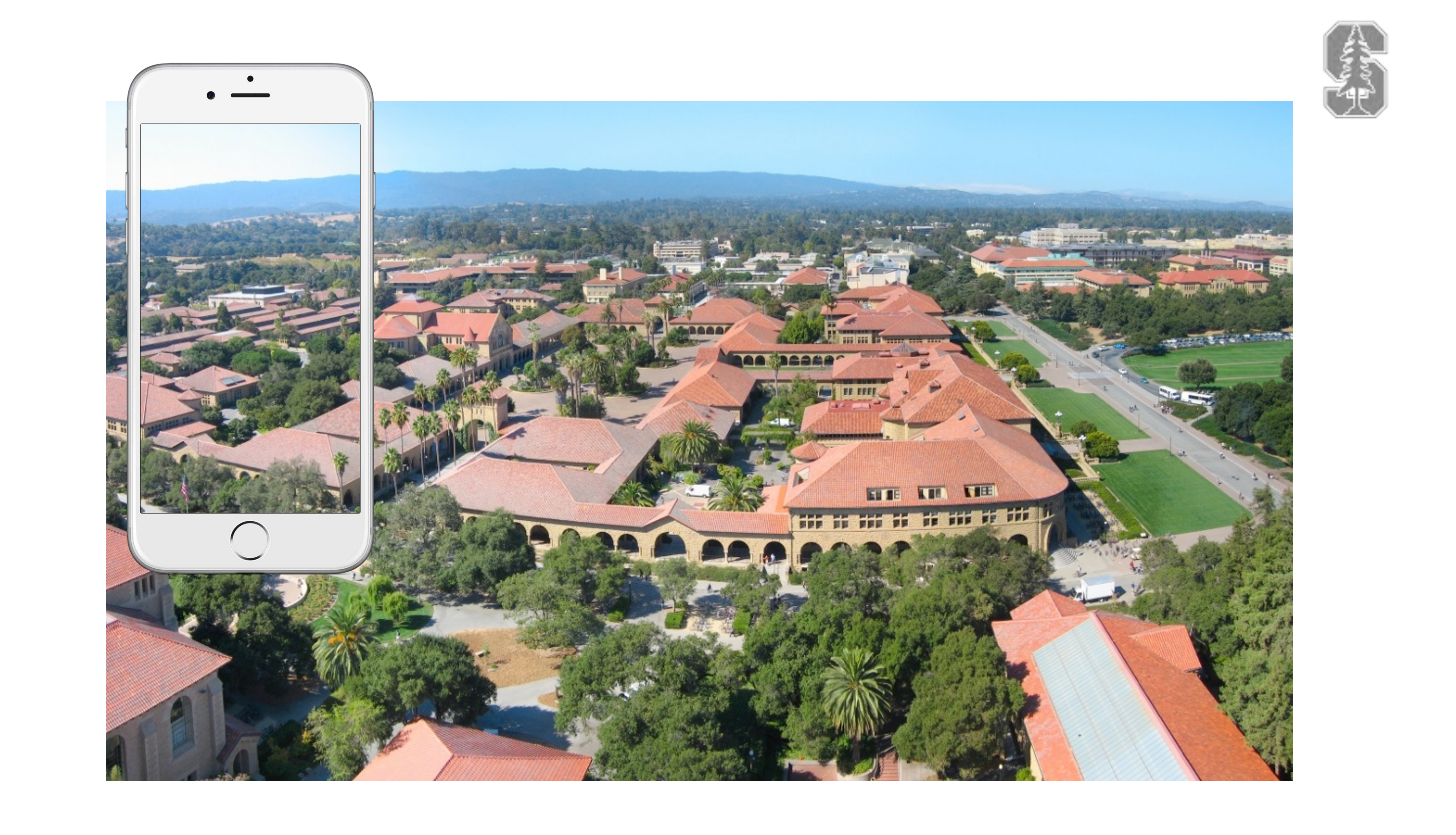
Adding subviews to a UIScrollView …

```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
aerial.frame = CGRect(x: 150, y: 200, width: 2500, height: 1600)
scrollView.addSubview(aerial)
```

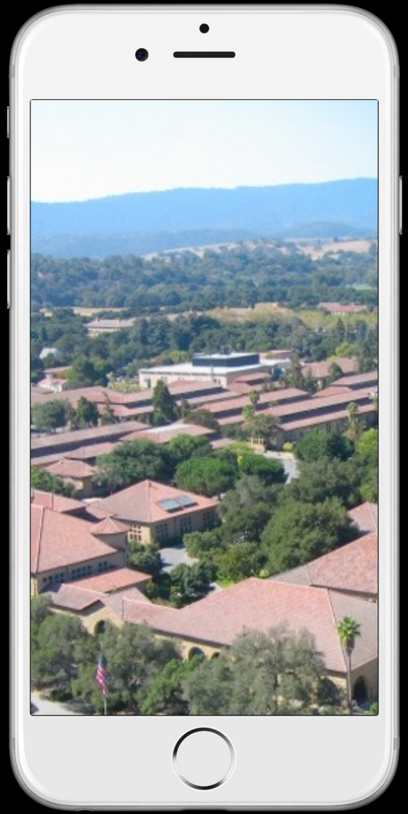


Adding subviews to a UIScrollView …

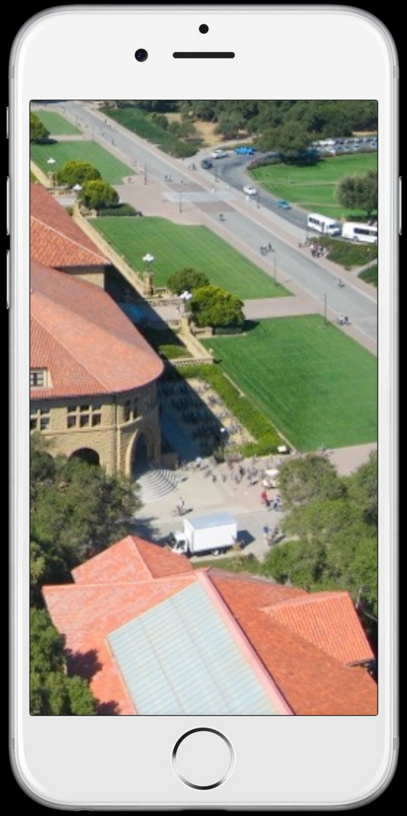
```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
aerial.frame = CGRect(x: 150, y: 200, width: 2500, height: 1600)
scrollView.addSubview(aerial)
```



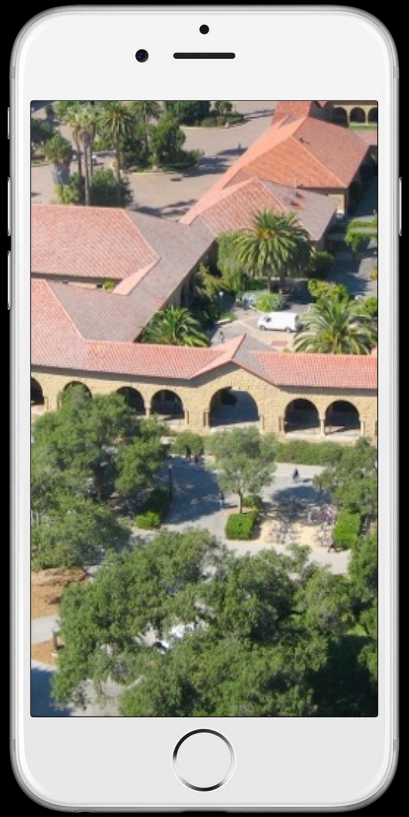
Scrolling in a UIScrollView …



Scrolling in a UIScrollView ...

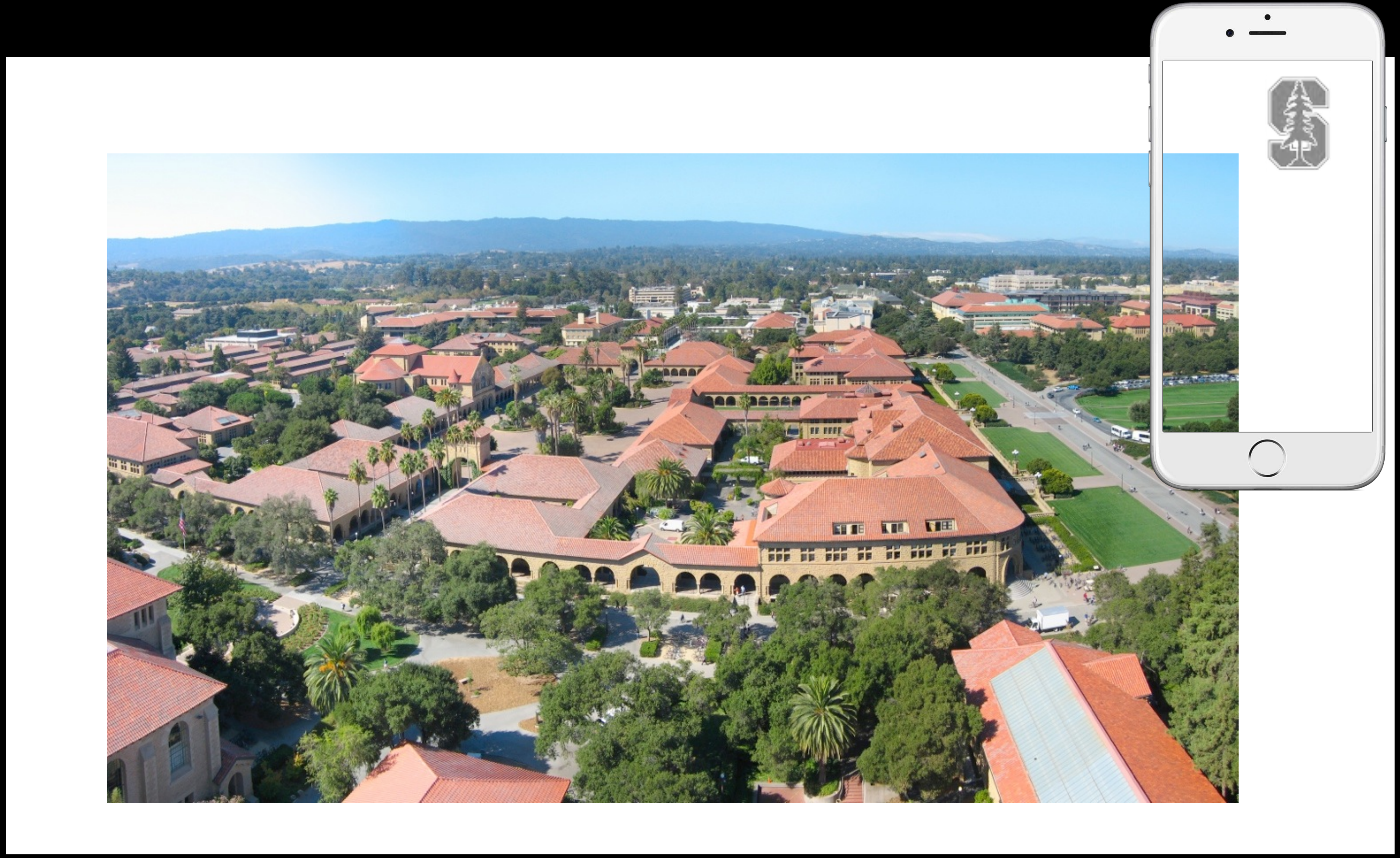


Scrolling in a UIScrollView …



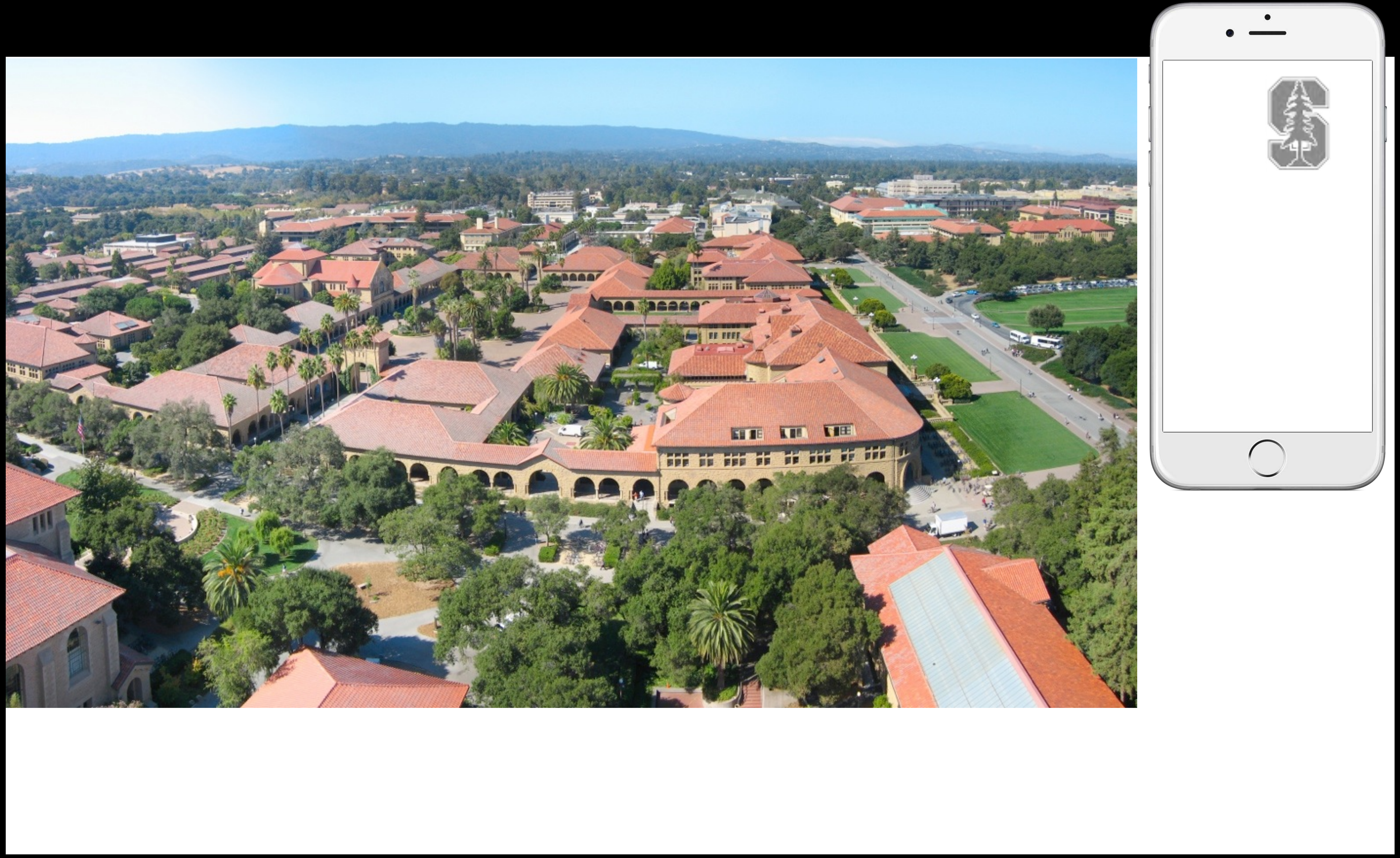
Scrolling in a UIScrollView …

Positioning subviews in a UIScrollView …



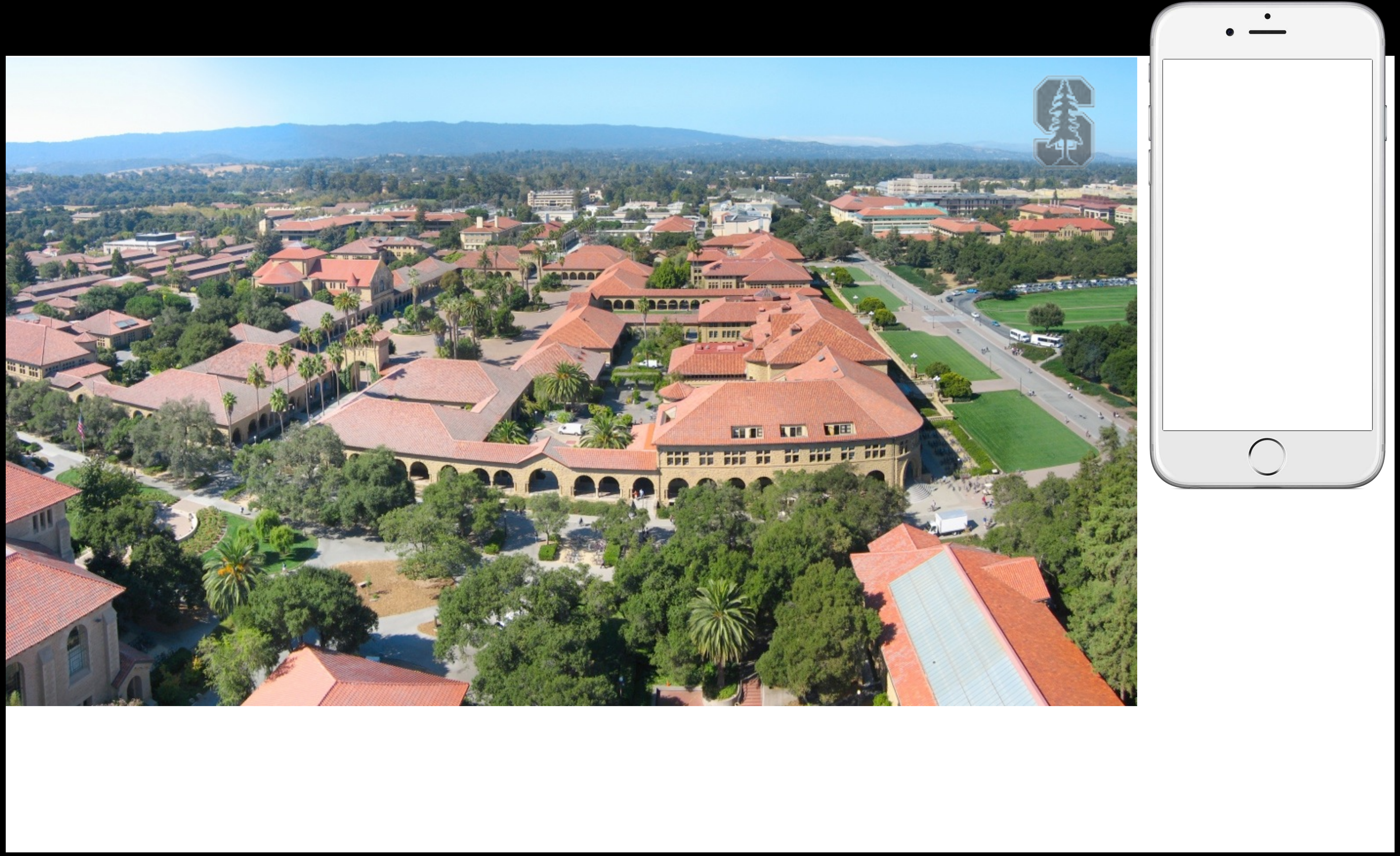
Positioning subviews in a UIScrollView …

```
aerial.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)
```



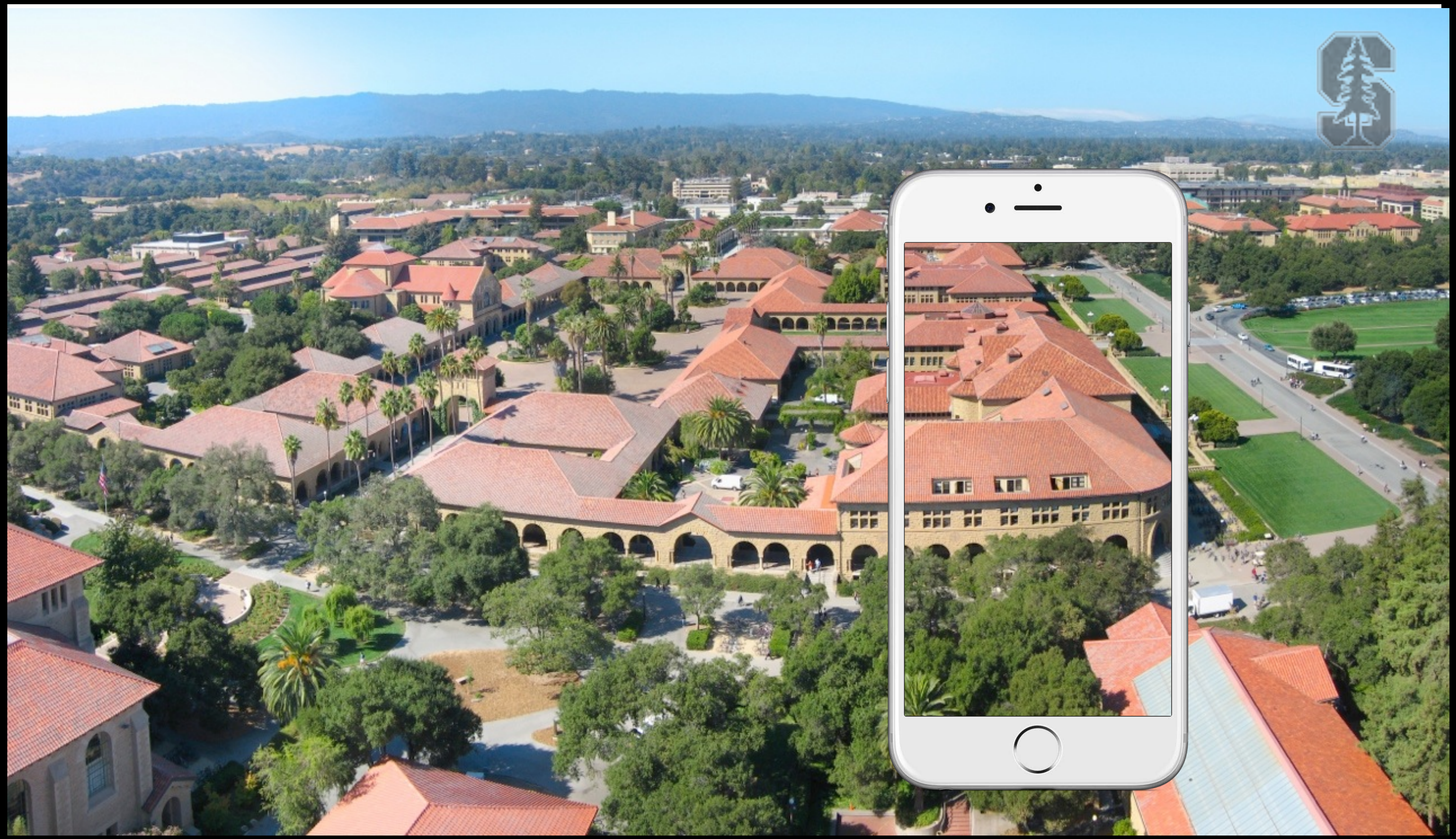
Positioning subviews in a UIScrollView …

```
aerial.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)
logo.frame = CGRect(x: 2300, y: 50, width: 120, height: 180)
```

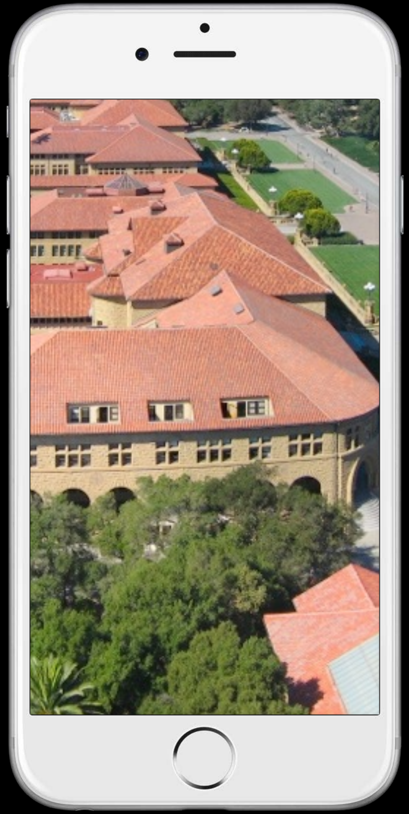


Positioning subviews in a UIScrollView …

```
aerial.frame = CGRect(x: 0, y: 0, width: 2500, height: 1600)
logo.frame = CGRect(x: 2300, y: 50, width: 120, height: 180)
scrollView.contentSize = CGSize(width: 2500, height: 1600)
```

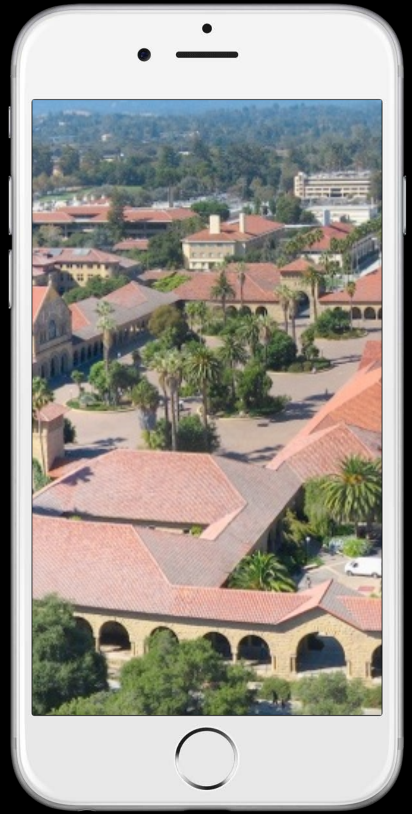


That's it!



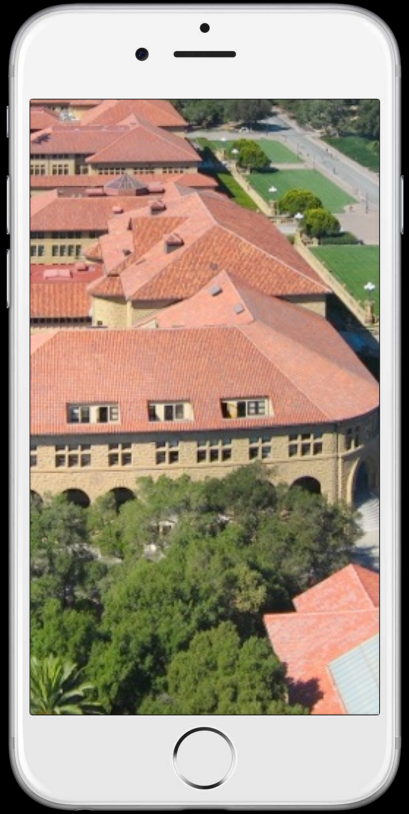
That's it!

That's it!



That's it!

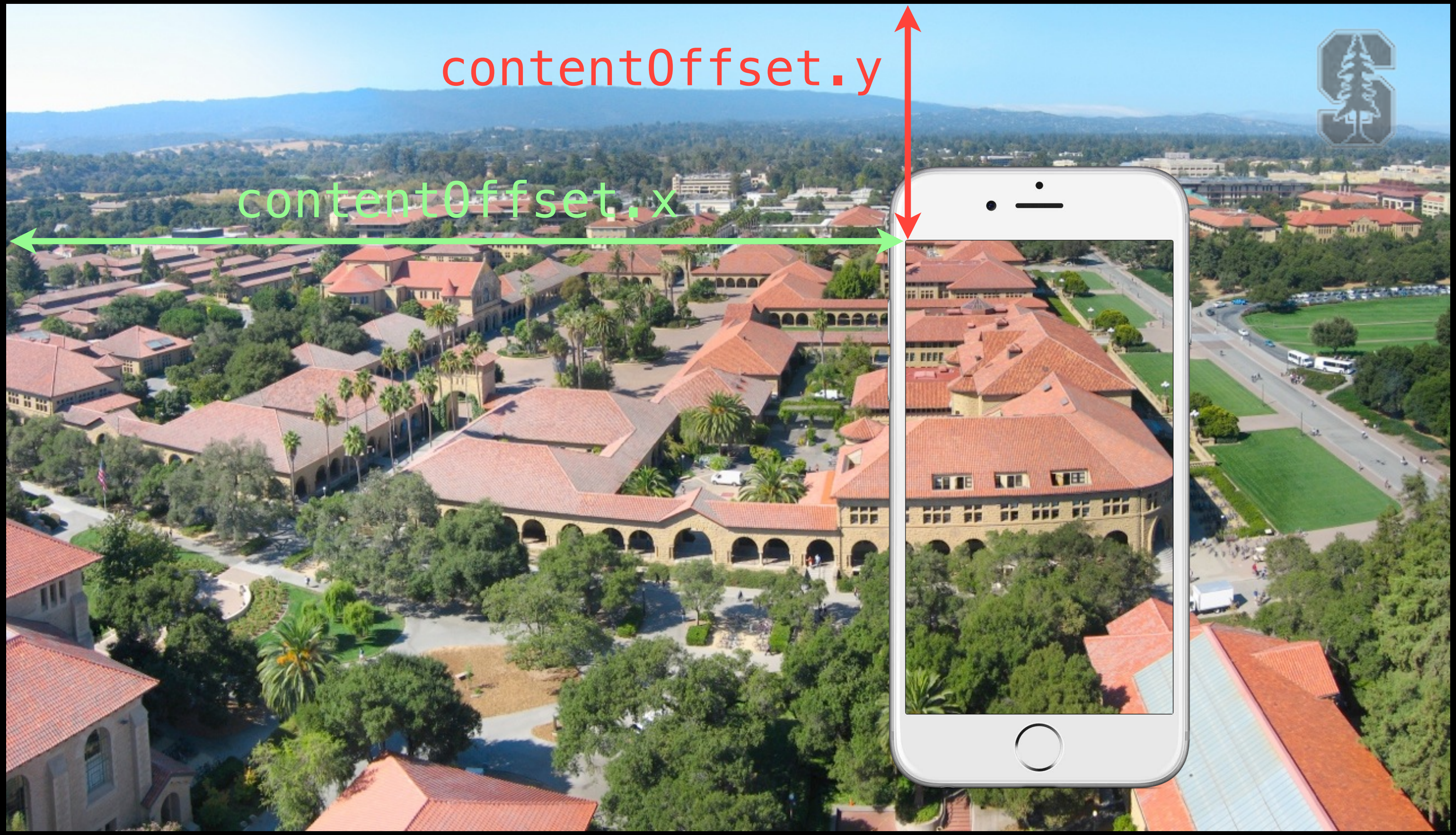
That's it!



Where in the content is the scroll view currently positioned?

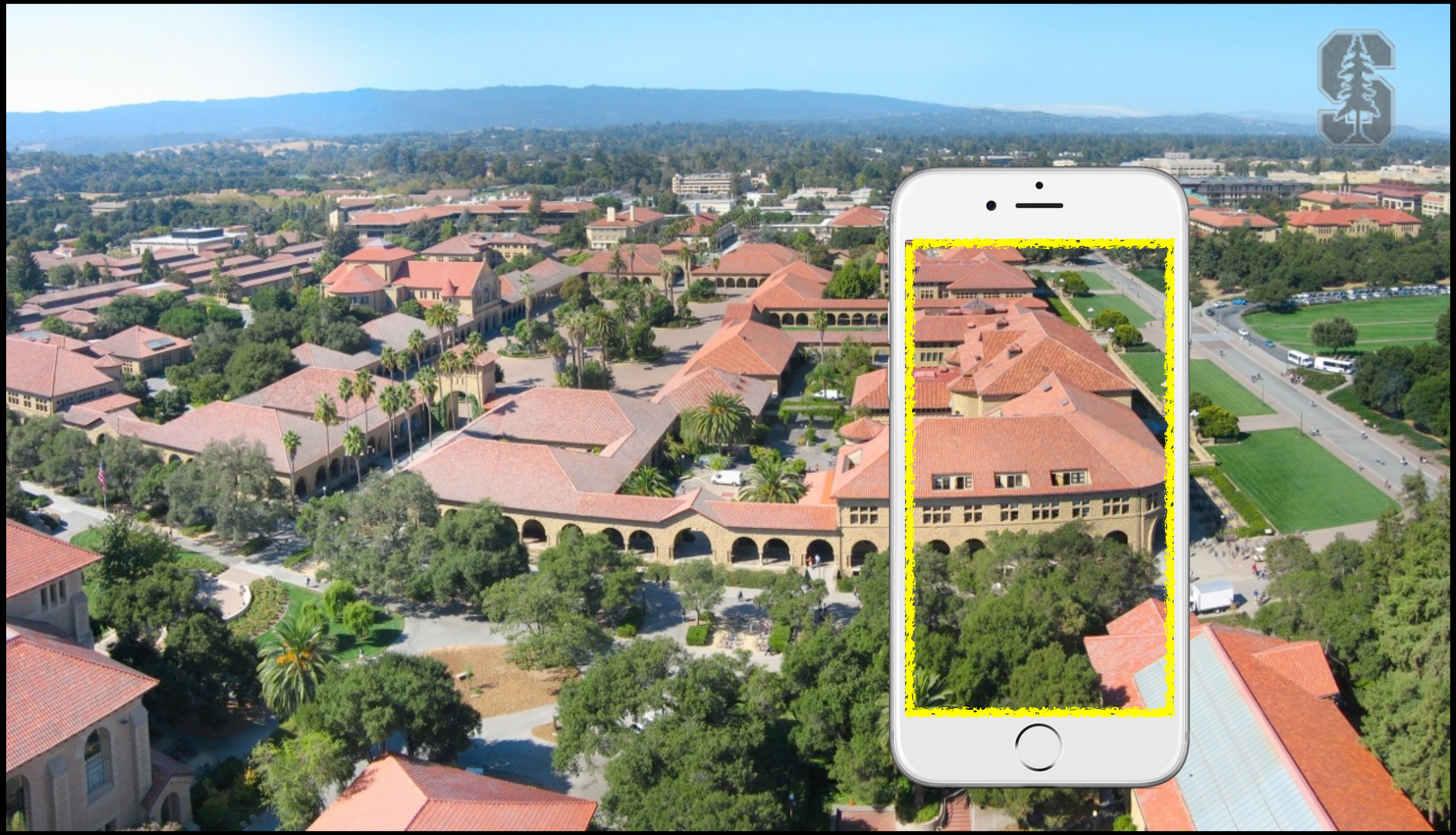
```
let upperLeftOfVisible: CGPoint = scrollView.contentOffset
```

In the content area's coordinate system.



What area in a subview is currently visible?

```
let visibleRect: CGRect = aerial.convertRect(scrollView.bounds, fromView: scrollView)
```



Why the `convertRect`? Because the `scrollView`’s `bounds` are in the `scrollView`’s coordinate system.
And there might be zooming going on inside the `scrollView` too …

UIScrollView

- How do you create one?

  Just like any other UIView. Drag out in a storyboard or use UIScrollView(frame:).

  Or select a UIView in your storyboard and choose "Embed In -> Scroll View" from Editor menu.

- To add your "too big" UIView in code using addSubview …

  ```
  let image = UIImage(named: "bigimage.jpg")
  let iv = UIImageView(image: image)  // iv.frame.size will = image.size
  scrollView.addSubview(iv)
  ```

  Add more subviews if you want.

  All of the subviews' frames will be in the UIScrollView's content area's coordinate system (that is, (0,0) in the upper left & width and height of contentSize.width & .height).

- Now don't forget to set the contentSize

  Common bug is to do the above 3 lines of code (or embed in Xcode) and forget to say:

  `scrollView.contentSize = imageView.bounds.size` (for example)

UIScrollView

- Scrolling programmatically

  ```
  func scrollRectToVisible(CGRect, animated: Bool)
  ```

- Other things you can control in a scroll view

  Whether scrolling is enabled.

  Locking scroll direction to user's first "move".

  The style of the scroll indicators (call `flashScrollIndicators` when your scroll view appears).

  Whether the actual content is "inset" from the content area (`contentInset` property).

UIScrollView

- Zoming

  All UIView's have a property (transform) which is an affine transform (translate, scale, rotate).

  Scroll view simply modifies this transform when you zoom.

  Zooming is also going to affect the scroll view's contentSize and contentOffset.

- Will not work without minimum/maximum zoom scale being set

  ```
  scrollView.minimumZoomScale = 0.5   // 0.5 means half its normal size
  scrollView.maximumZoomScale = 2.0   // 2.0 means twice its normal size
  ```

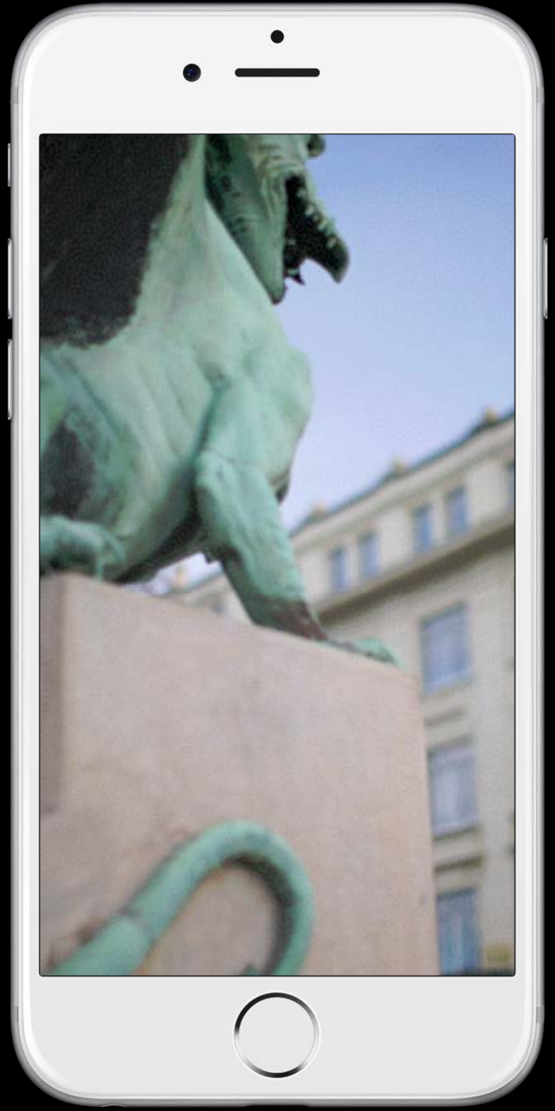
- Will not work without delegate method to specify view to zoom

  ```
  func viewForZoomingInScrollView(sender: UIScrollView) -> UIView
  ```

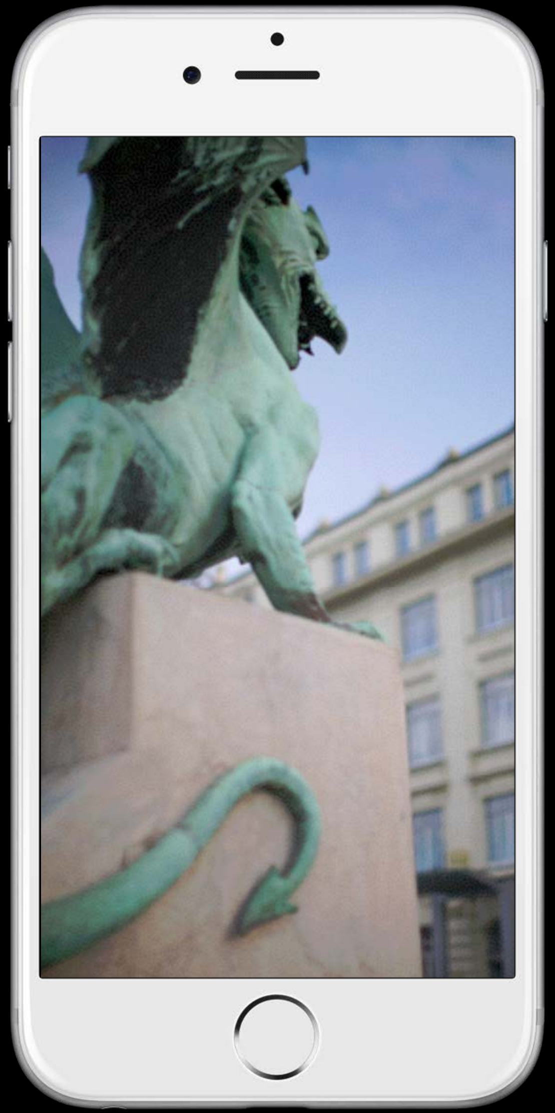
  If your scroll view only has one subview, you return it here. More than one? Up to you.

- Zooming programatically

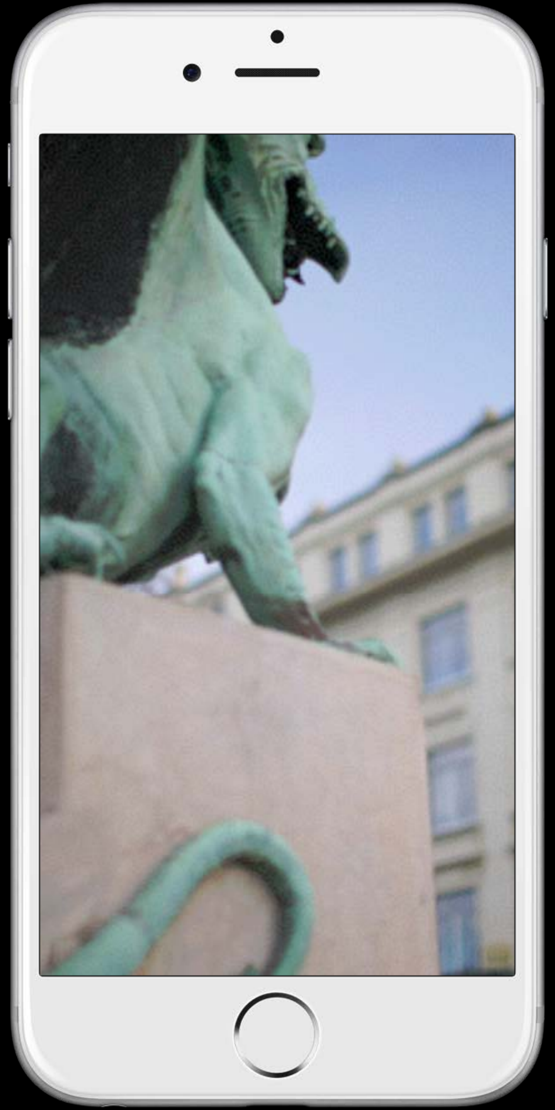
  ```
  var zoomScale: CGFloat
  func setZoomScale(CGFloat, animated: Bool)
  func zoomToRect(CGRect, animated: Bool)
  ```



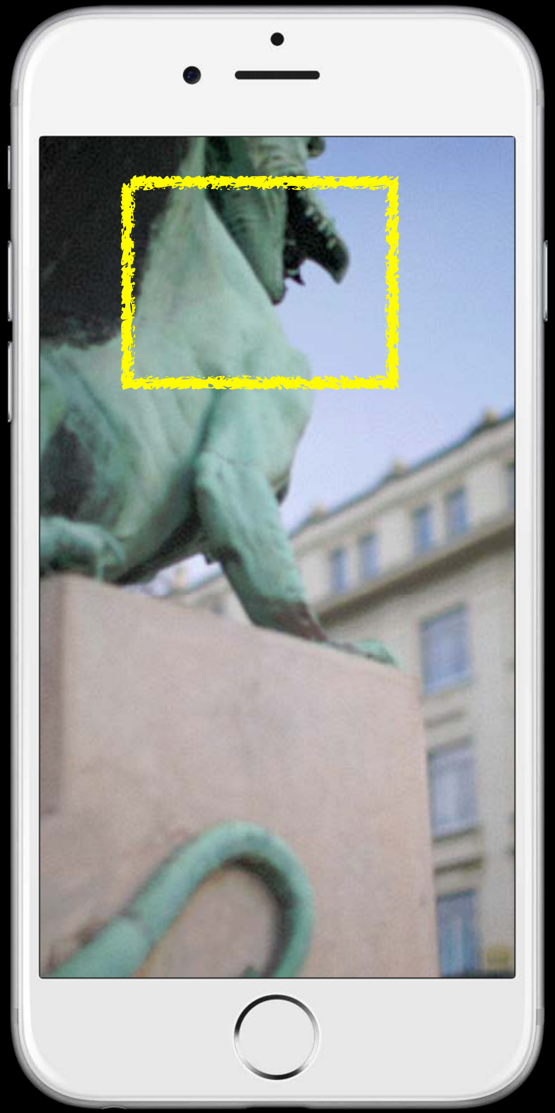
```
scrollView.zoomScale = 1.2
```



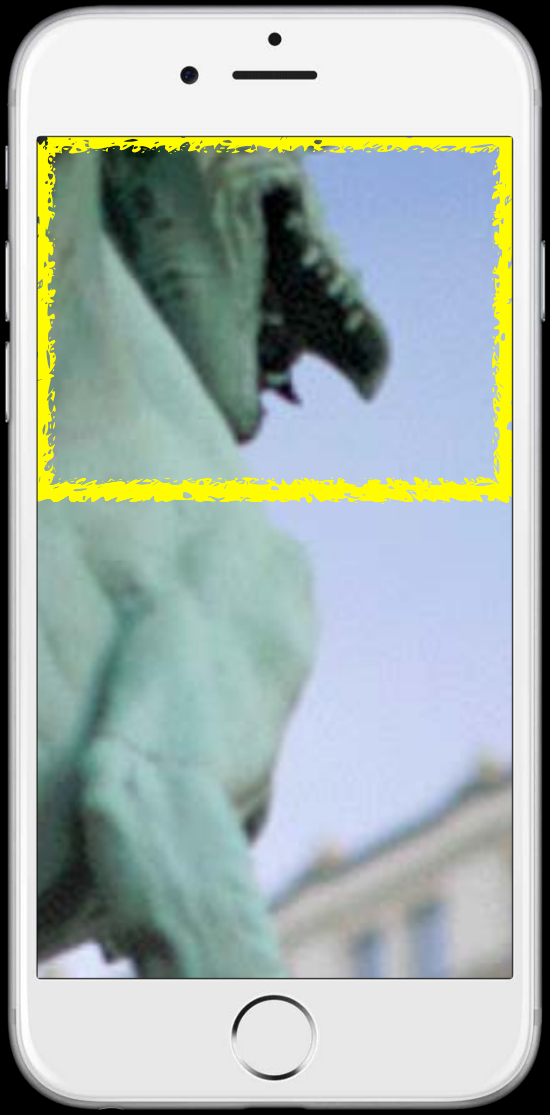
`scrollView.zoomScale = 1.0`



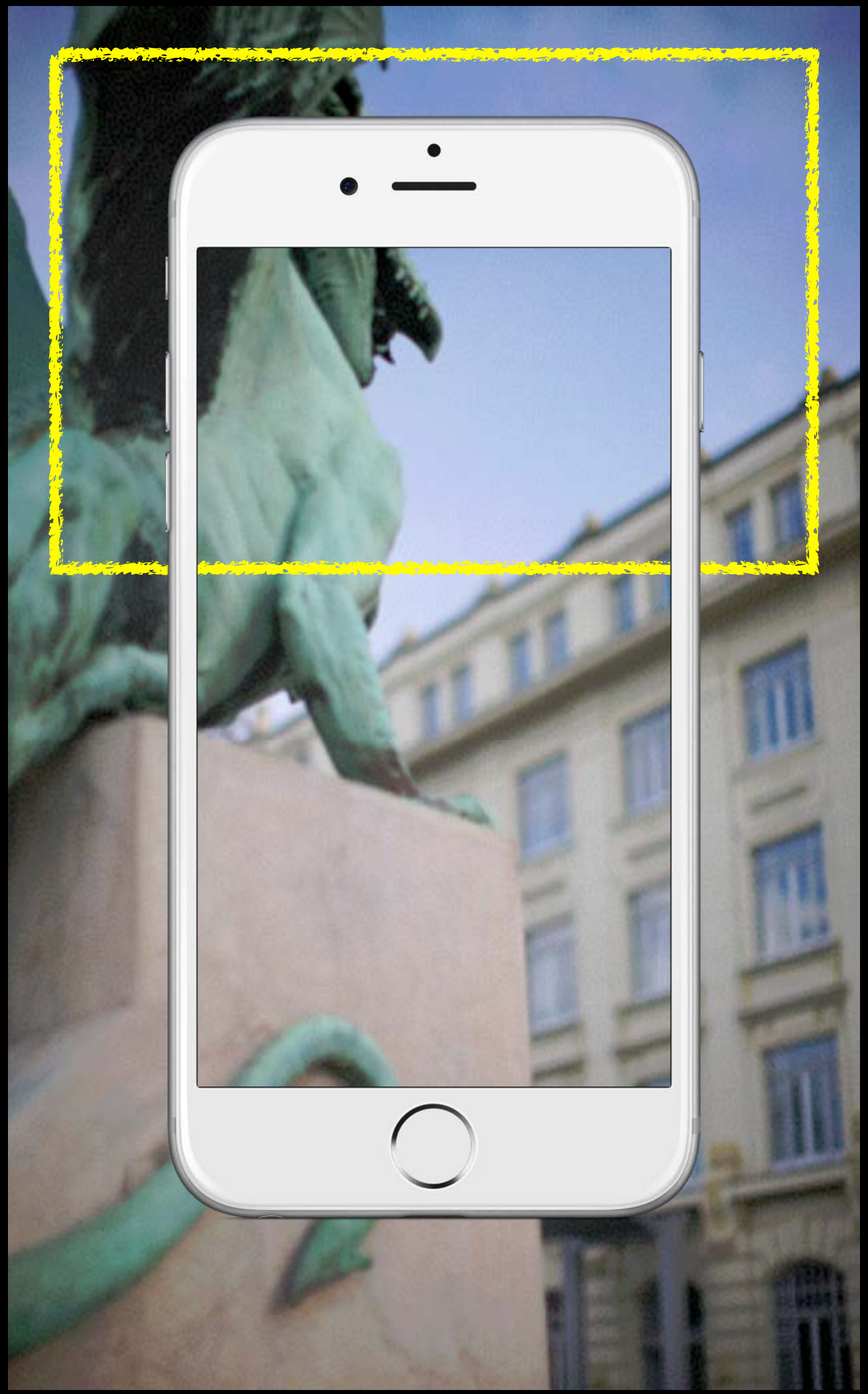
```
scrollView.zoomScale = 1.2
```



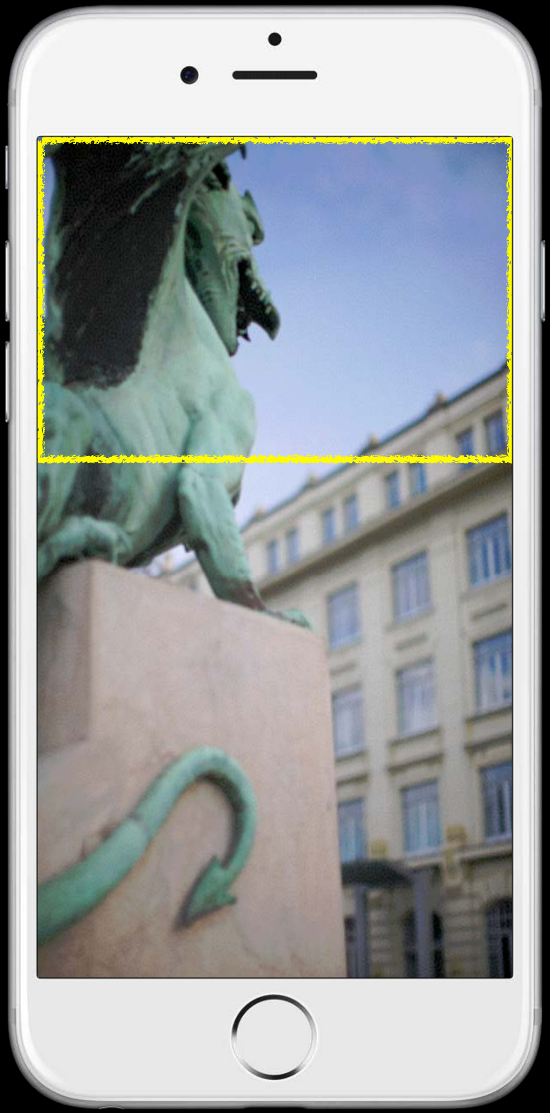
```
zoomToRect(CGRect, animated: Bool)
```



```
zoomToRect(CGRect, animated: Bool)
```



`zoomToRect(CGRect, animated: Bool)`



`zoomToRect(CGRect, animated: Bool)`

UIScrollView

- Lots and lots of delegate methods!

  The scroll view will keep you up to date with what's going on.

- Example: delegate method will notify you when zooming ends

```
func scrollViewDidEndZooming(UIScrollView,
                             withView: UIView,  // from delegate method above
                             atScale: CGFloat)
```

  If you redraw your view at the new scale, be sure to reset the transform back to identity.

Demo

- Imaginarium

  Panning and zooming in on a big image.

Closures

- Capturing

  Closures “capture” variables in the surrounding context

  That means that it keeps those variables around as long as the closure stays around

  You can even make assignments to the variables or modify what they point to

  This can lead to some very elegant code …

Closures

- Interesting use of a closure …

  It might be that sometimes using a closure is a better tool than delegation.

  For example, consider the following code …

```
class Grapher {
    var yForX: ((x: Double) -> Double?)? // completely and utterly generic
}

let grapher = Grapher()
let graphingBrain = CalculatorBrain()
graphingBrain.program = theProgramToGraph
grapher.yForX = { (x: Double) -> Double? in
    graphingBrain.variableValues[“M”] = x
    return graphingBrain.evaluate() // gets captured and reused each time yForX is called
}
```

  For your assignment, we wanted you to learn delegation, but this is cool too.

Closures

- Capture Danger

  We have to be a little bit careful about capturing because of memory management

  Specifically, we don't want to create a memory cycle

  Closures capture pointers (i.e. it keeps what they point to in memory)

  If a captured pointer points (directly or indirectly) back at the closure, that's a problem

  Because now there will always be a pointer to the closure and to the captured thing

  Neither will ever be able to leave the heap

Closures

- A “danger case” for closures …

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { println(“\(value)”) }
    func setupMyAction() {
        var x: Int = 0
        action = {
            x = x + 1
            self.show(x)
        }
    }
    func doMyAction10times() { for i in 1…10 { action() } }
}
```

So this will actually work. It will print 1 2 3 4 5 6 7 8 9 10!

Closures

- A “danger case” for closures …

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { println(“\(value)”) }
    func setupMyAction() {
        var x: Int = 0
        action = {
            x = x + 1
            self.show(x)
        }
    }
    func doMyAction10times() { for i in 1…10 { action() } }
}
```

This is cool because it captured that x for as long as this closure is around.

Closures

- A “danger case” for closures …

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { println(“\(value)”) }
    func setupMyAction() {
        var x: Int = 0
        action = {
            x = x + 1
            self.show(x)
        }
    }
    func doMyAction10times() { for i in 1…10 { action() } }
}
```

And this is cool too. It makes sure `self` stays around so we can call `show`.

Closures

- A “danger case” for closures …

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { println(“\(value)”) }
    func setupMyAction() {
        var x: Int = 0
        action = {
            x = x + 1
            self.show(x)
        }
    }
    func doMyAction10times() { for i in 1…10 { action() } }
}
```

But what’s not so cool is that `self` points to this closure (via its `action` property).
Neither can now ever leave the heap (they point to each other).

Closures

- A “danger case” for closures …

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { println(“\(value)”) }
    func setupMyAction() {
        var x: Int = 0
        action = {
            x = x + 1
            self.show(x)
        }
    }
    func doMyAction10times() { for i in 1…10 { action() } }
}
```

How can we fix this?

We need to tell the closure not to keep that self in memory.

Closures

- A "danger case" for closures …

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { println("\(value)") }
    func setupMyAction() {
        var x: Int = 0
        action = { [unowned self] in
            x = x + 1
            self.show(x)
        }
    }
    func doMyAction10times() { for i in 1…10 { action() } }
}
```

Here's how we do that.

Now that reference to self inside the closure will not keep self in memory.

That self will still live as long as someone ELSE has a pointer to it though.

Closures

- A “danger case” for closures …

```
class Foo {
    var action: () -> Void = { }
    func show(value: Int) { println(“\(value)”) }
    func setupMyAction() {
        var x: Int = 0
        action = { [unowned self] in
            x = x + 1
            self.show(x)
        }
    }
    func doMyAction10times() { for i in 1…10 { action() } }
}
```

If you are struggling with this, please re-read your reading assignment.
Specifically the Closures and Automatic Reference Counting sections.

Multithreading

- Queues

  Multithreading is mostly about "queues" in iOS.

  Functions (usually closures) are lined up in a queue.

  Then those functions are pulled off the queue and executed on an associated thread.

- Main Queue

  There is a very special queue called the "main queue."

  All UI activity MUST occur on this queue and this queue only.

  And, conversely, non-UI activity that is at all time consuming must NOT occur on that queue.

  We want our UI to be responsive!

  Functions are pulled off and worked on in the main queue only when it is "quiet".

- Other Queues

  Mostly iOS will create these for us as needed.

  We'll give a quick overview of how to create your own (but usually not necessary).

Multithreading

- Executing a function on another queue

```
let queue: dispatch_queue_t = <get the queue you want, more on this in a moment>
dispatch_async(queue) { /* do what you want to do here */ }
```

- The main queue (a serial queue)

```
let mainQ: dispatch_queue_t = dispatch_get_main_queue()
let mainQ: NSOperationQueue = NSOperationQueue.mainQueue() // for object-oriented APIs
```

All UI stuff must be done on this queue!

And all time-consuming (or, worse, potentially blocking) stuff must be done off this queue!

Common code to write ...

```
dispatch_async(notTheMainQueue) {
    // do something that might block or takes a while
    dispatch_async(dispatch_get_main_queue()) {
        // call UI functions with the results of the above
    }
}
```

Multithreading

- Other (concurrent) queues (i.e. not the main queue)

  Most non-main-queue work will happen on a concurrent queue with a certain quality of service

  ```
  QOS_CLASS_USER_INTERACTIVE   // quick and high priority
  QOS_CLASS_USER_INITIATED     // high priority, might take time
  QOS_CLASS_UTILITY            // long running
  QOS_CLASS_BACKGROUND         // user not concerned with this (prefetching, etc.)
  let qos = Int(<one of the above>.value) // ugh, historical reasons
  let queue = dispatch_get_global_queue(qos, 0)
  ```

  You will probably use these queues to do any work that you don’t want to block the main queue

- You can create your own serial queue if you need serialization

  ```
  let serialQ = dispatch_queue_create(“name”, DISPATCH_QUEUE_SERIAL)
  ```

  Maybe you are downloading a bunch of things things from a certain website
  but you don’t want to deluge that website, so you queue the requests up serially

  Or maybe the things you are doing depend on each other in a serial fashion

Multithreading

- Doing something in the future

```
let delayInSeconds = 25.0
let delay = Int64(delayInSeconds*Double(NSEC_PER_MSEC)) // ugh, historical reasons
let dispatchTime = dispatch_time(DISPATCH_TIME_NOW, delay) // adds delay to NOW
dispatch_after(dispatchTime, dispatch_get_main_queue()) {
    // do something on the main queue 25 seconds from now
}
```

- We are only seeing the tip of the iceberg

  There is a lot more to GCD

  You can do locking, protect critical sections, readers and writers, synchronous dispatch, etc.

  Check out the documentation if you are interested

Multithreading

- Multithreaded iOS API

  Quite a few places in iOS will do what they do off the main queue

  They might even afford you the opportunity to do something off the main queue

  You may pass in a function (a closure, usually) that sometimes executes off the main thread

  Don't forget that if you want to do UI stuff there, you must dispatch back to the main queue!

Multithreading

- Example of a multithreaded iOS API

This API lets you fetch something from an http URL to a local file

Obviously it can’t do that on the main thread!

```
let session = NSURLSession(NSURLSessionConfiguration.defaultSessionConfiguration())
if let url = NSURL(string: “http://url”) {
    let request = NSURLRequest(URL: url)
    let task = session.downloadTaskWithRequest(request) { (localURL, response, error) in
        /* I want to do UI things here with the result of the download, can I? */
    }
    task.resume()
}
```

The answer to the above comment is “no”.

That’s because the block will be run off the main queue.

How do we deal with this?

One way is to use a variant of this API that lets you specify the queue to run on.

Another way is …

Multithreading

How to do UI stuff safely

You can simply dispatch back to the main queue …

```
let session = NSURLSession(NSURLSessionConfiguration.defaultSessionConfiguration())
if let url = NSURL(string: “http://url”) {
    let request = NSURLRequest(URL: url)
    let task = session.downloadTaskWithRequest(request) { (localURL, response, error) in
        dispatch_async(dispatch_get_main_queue()) {
            /* I want to do something in the UI here, can I? */
        }
    }
    task.resume()
}
```

Yes! Because the UI code you want to do has been dispatched back to the main queue.
But understand that that code might run MINUTES after the request is fired off.
The user might have long ago given up on whatever was being fetched.

Demo

- Multithreaded Imaginarium

  Let's not block the main queue's thread by doing our URL request in a different thread

  If we have time, we can also give the user some feedback that "we're working on it"