Stanford CS193p

Developing Applications for iOS
Spring 2016

Today

- Memory Management for Reference Types
  - Controlling when things leave the heap
- Closure Capture
  - Closures capture things into the heap too
- Extensions
  - A simple, powerful, but easily overused code management syntax
- Protocols
  - Last (but certainly not least important) typing mechanism in Swift
- Delegation
  - An important use of protocols used throughout the iOS frameworks API
- Scroll View
  - Scrolling around in and zooming in on big things on a small screen

Memory Management

- Automatic Reference Counting

  Reference types (`classes`) are stored in the heap.

  How does the system know when to reclaim the memory for these from the heap?

  It "counts references" to each of them and when there are zero references, they get tossed.

  This is done automatically.

  It is known as "Automatic Reference Counting" and it is NOT garbage collection.

- Influencing ARC

  You can influence ARC by how you declare a reference-type var with these keywords …

  `strong`

  `weak`

  `unowned`

Memory Management

- strong

  strong is "normal" reference counting
  As long as anyone, anywhere has a strong pointer to an instance, it will stay in the heap

- weak

  weak means "if no one else is interested in this, then neither am I, set me to nil in that case"
  Because it has to be nil-able, weak only applies to Optional pointers to reference types
  A weak pointer will NEVER keep an object in the heap
  Great example: outlets (strongly held by the view hierarchy, so outlets can be weak)

- unowned

  unowned means "don't reference count this; crash if I'm wrong"
  This is very rarely used
  Usually only to break memory cycles between objects (more on that in a moment)

Closures

- Capturing

  Closures are stored in the heap as well (i.e. they are reference types).
  They can be put in `Arrays`, `Dictionarys`, etc. They are a first-class type in Swift.

  What is more, they "capture" variables they use from the surrounding code into the heap too.
  Those captured variables need to stay in the heap as long as the closure stays in the heap.
  This can create a memory cycle …

Closures

- Example

Imagine we added public API to allow a UnaryOperation to be added to the CalculatorBrain

```
func addUnaryOperation(symbol: String, operation: (Double) -> Double)
```

This method would do nothing more than add a UnaryOperation to our Dictionary of enum

Now let's imagine a View Controller was to add the operation "red square root".
This operation will do square root, but it will also turn the display red.

```
addUnaryOperation("🔴√", operation: { (x: Double) -> Double in
    display.textColor = UIColor.redColor()
    return sqrt(x)
})
```

Closures

- Example

  Imagine we added public API to allow a UnaryOperation to be added to the CalculatorBrain

  ```
  func addUnaryOperation(symbol: String, operation: (Double) -> Double)
  ```

  This method would do nothing more than add a UnaryOperation to our Dictionary of enum

  Now let's imagine a View Controller was to add the operation "red square root".
  This operation will do square root, but it will also turn the display red.

  ```
  addUnaryOperation("🔴√", operation: { (x: Double) -> Double in
      display.textColor = UIColor.redColor()
      return sqrt(x)
  })
  ```

Closures

- Example

Imagine we added public API to allow a UnaryOperation to be added to the CalculatorBrain

```
func addUnaryOperation(symbol: String, operation: (Double) -> Double)
```

This method would do nothing more than add a UnaryOperation to our Dictionary of enum

Now let's imagine a View Controller was to add the operation "red square root".
This operation will do square root, but it will also turn the display red.

```
addUnaryOperation("🔴√") { (x: Double) -> Double in
    display.textColor = UIColor.redColor()
    return sqrt(x)
}
```

Closures

- Example

  Imagine we added public API to allow a UnaryOperation to be added to the CalculatorBrain

  `func addUnaryOperation(symbol: String, operation: (Double) -> Double)`

  This method would do nothing more than add a UnaryOperation to our Dictionary of enum

  Now let's imagine a View Controller was to add the operation "red square root".

  This operation will do square root, but it will also turn the display red.

```
addUnaryOperation("🔴√") { (x: Double) -> Double in
    display.textColor = UIColor.redColor()
    return sqrt(x)
}
```

Closures

- Example

  Imagine we added public API to allow a `UnaryOperation` to be added to the `CalculatorBrain`

  `func addUnaryOperation(symbol: String, operation: (Double) -> Double)`

  This method would do nothing more than add a `UnaryOperation` to our `Dictionary` of `enum`

  Now let's imagine a View Controller was to add the operation "red square root".

  This operation will do square root, but it will also turn the display red.

```
addUnaryOperation("🔴√") {
    display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

  But this will not compile.

Closures

- Example

Imagine we added public API to allow a UnaryOperation to be added to the CalculatorBrain

```
func addUnaryOperation(symbol: String, operation: (Double) -> Double)
```

This method would do nothing more than add a UnaryOperation to our Dictionary of enum

Now let's imagine a View Controller was to add the operation "red square root".
This operation will do square root, but it will also turn the display red.

```
addUnaryOperation("🔴√") {
    self.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Swift forces you to put self. here to remind you that self will get captured!
The Model and the Controller now point to each other through the closure.
And thus neither can ever leave the heap. This is called a memory cycle.

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation(“🔴√”) { [ <special variable declarations> ] in
    self.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation("🔴√") { [ me = self ] in
    me.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior ...

```
addUnaryOperation("🔴√") { [ unowned me = self ] in
    me.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation(“🔴√”) { [ unowned self = self ] in
    self.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation("🔴√") { [ unowned self ] in
    self.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation(“🔴√”) { [ weak self ] in
    self.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation("🔴√") { [ weak self ] in
    self?.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Closures

- So how do we break this cycle?

  Swift lets you control this capture behavior …

```
addUnaryOperation("🔴√") { [ weak weakSelf = self ] in
    weakSelf?.display.textColor = UIColor.redColor()
    return sqrt($0)
}
```

Demo

- Red Square Root

  Let's do what we just talked about and see it in action in our Calculator

Extensions

- Extending existing data structures

You can add methods/properties to a class/struct/enum (even if you don't have the source).
For example, this adds a method contentViewController to UIViewController …

```
extension UIViewController {
    var contentViewController: UIViewController {
        if let navcon = self as? UINavigationController {
            return navcon.visibleViewController
        } else {
            return self
        }
    }
}
```

… it can be used to clean up prepareForSegue code …

```
var destination: UIViewController? = segue.destinationViewController
if let navcon = destination as? UINavigationController {
    destination = navcon.visibleViewController
}
if let myvc = destination as? MyVC { … }
```

Extensions

- Extending existing data structures

You can add methods/properties to a `class/struct/enum` (even if you don't have the source).

For example, this adds a method `contentViewController` to `UIViewController` …

```
extension UIViewController {
    var contentViewController: UIViewController {
        if let navcon = self as? UINavigationController {
            return navcon.visibleViewController
        } else {
            return self
        }
    }
}
```

… it can be used to clean up `prepareForSegue` code …

```
if let myvc = segue.destinationViewController.contentViewController as? MyVC { … }
```

Extensions

- Extending existing data structures

  You can add methods/properties to a `class/struct/enum` (even if you don't have the source).

  For example, this adds a method `contentViewController` to `UIViewController` …

```
extension UIViewController {
    var contentViewController: UIViewController {
        if let navcon = self as? UINavigationController {
            return navcon.visibleViewController
        } else {
            return self
        }
    }
}
```

  Notice that when it refers to `self`, it means the thing it is extending (`UIViewController`).

Extensions

- Extending existing data structures
  - You can add methods/properties to a `class`/`struct`/`enum` (even if you don't have the source).
- There are some restrictions
  - You can't re-implement methods or properties that are already there (only add new ones).
  - The properties you add can have no storage associated with them (computed only).
- This feature is easily abused
  - It should be used to add clarity to readability not obfuscation!
  - Don't use it as a substitute for good object-oriented design technique.
  - Best used (at least for beginners) for very small, well-contained helper functions.
  - Can actually be used well to organize code but requires architectural commitment.
  - When in doubt (for now), don't do it.

Protocols

- Protocols are a way to express an API more concisely

  Instead of forcing the caller of an API to pass a specific `class`, `struct`, or `enum`,
  an API can let callers pass any `class/struct/enum` that the caller wants
  but can require that they implement certain methods and/or properties that the API wants.
  To specify which methods and properties the API wants, the API is expressed using a Protocol.
  A Protocol is simply a collection of method and property declarations.

- A Protocol is a TYPE

  It can be used almost anywhere any other type is used: `var`s, function parameters, etc.

- The implementation of a Protocol's methods and properties

  The implementation is provided by an implementing type (any `class`, `struct` or `enum`).
  Because of this, a protocol can have no storage associated with it
  (any storage required to implement the protocol is provided by an implementing type).
  It is possible to add implementation to a protocol via an `extension` to that protocol
  (but remember that `extension`s also cannot use any storage)

Protocols

- There are four aspects to a protocol
  1. the `protocol` declaration (what properties and methods are in the `protocol`)
  2. the declaration where a `class`, `struct` or `enum` claims that it implements a `protocol`
  3. the code that implements the `protocol` in said `class`, `struct` or `enum`
  4. optionally, an `extension` to the protocol which provides some default implementation

- Optional methods in a protocol

  Normally any protocol implementor must implement all the methods/properties in the protocol.
  However, it is possible to mark some methods in a protocol `optional`
  (don't get confused with the type Optional, this is a different thing).
  Any protocol that has `optional` methods must be marked `@objc`.
  And any optional-protocol implementing `class` must inherit from `NSObject`.
  Used often in iOS for delegation (more later on this).
  Except for delegation, a protocol with `optional` methods is rarely (if ever) used.
  As you can tell from the `@objc` designation, it's mostly for backwards compatibility.

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements `SomeProtocol` must also implement `InheritedProtocol1` and 2

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

  Anyone that implements `SomeProtocol` must also implement `InheritedProtocol1` and 2

  You must specify whether a property is `get` only or both `get` and `set`

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements SomeProtocol must also implement InheritedProtocol1 and 2

You must specify whether a property is get only or both get and set

Any functions that are expected to mutate the receiver should be marked mutating

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : class, InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements SomeProtocol must also implement InheritedProtocol1 and 2
You must specify whether a property is get only or both get and set
Any functions that are expected to mutate the receiver should be marked mutating
(unless you are going to restrict your protocol to class implementers only with class keyword)

Protocols

- Declaration of the protocol itself

```
protocol SomeProtocol : InheritedProtocol1, InheritedProtocol2 {
    var someProperty: Int { get set }
    func aMethod(arg1: Double, anotherArgument: String) -> SomeType
    mutating func changeIt()
    init(arg: Type)
}
```

Anyone that implements SomeProtocol must also implement InheritedProtocol1 and 2
You must specify whether a property is get only or both get and set
Any functions that are expected to mutate the receiver should be marked mutating
(unless you are going to restrict your protocol to class implementers only with class keyword)
You can even specify that implementers must implement a given initializer

Protocols

- How an implementer says "I implement that protocol"

```
class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class

Protocols

- How an implementer says "I implement that protocol"

```
enum SomeEnum : SomeProtocol, AnotherProtocol {
    // implementation of SomeEnum here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class
Obviously, enums and structs would not have the superclass part

Protocols

- How an implementer says "I implement that protocol"

```
struct SomeStruct : SomeProtocol, AnotherProtocol {
    // implementation of SomeStruct here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class
Obviously, enums and structs would not have the superclass part

Protocols

- How an implementer says "I implement that protocol"

```
struct SomeStruct : SomeProtocol, AnotherProtocol {
    // implementation of SomeStruct here
    // which must include all the properties and methods in SomeProtocol & AnotherProtocol
}
```

Claims of conformance to protocols are listed after the superclass for a class
Obviously, enums and structs would not have the superclass part
Any number of protocols can be implemented by a given class, struct or enum

Protocols

- How an implementer says "I implement that protocol"

```
class SomeClass : SuperclassOfSomeClass, SomeProtocol, AnotherProtocol {
    // implementation of SomeClass here, including …
    required init(…)
}
```

Claims of conformance to protocols are listed after the superclass for a class
Obviously, enums and structs would not have the superclass part
Any number of protocols can be implemented by a given class, struct or enum
In a class, inits must be marked required (or otherwise a subclass might not conform)

Protocols

- How an implementer says "I implement that protocol"

```
extension Something : SomeProtocol {
    // implementation of SomeProtocol here
    // no stored properties though
}
```

Claims of conformance to protocols are listed after the superclass for a class
Obviously, enums and structs would not have the superclass part
Any number of protocols can be implemented by a given class, struct or enum
In a class, inits must be marked required (or otherwise a subclass might not conform)
You are allowed to add protocol conformance via an extension

Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

Protocols

- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

Protocols

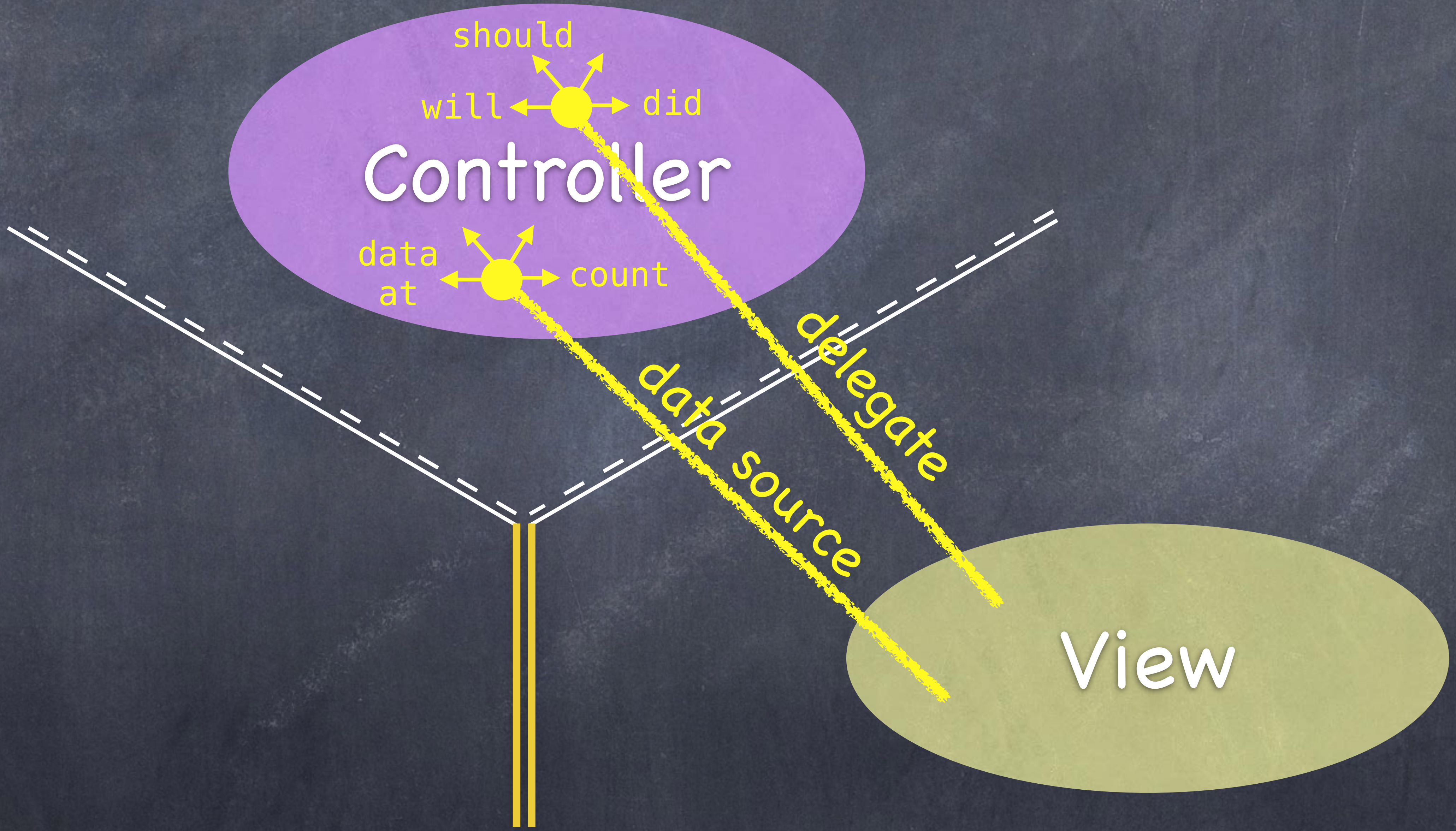
- Using protocols like the type that they are!

```
protocol Moveable {
    mutating func moveTo(p: CGPoint)
}
class Car : Moveable {
    func moveTo(p: CGPoint) { … }
    func changeOil()
}
struct Shape : Moveable {
    mutating func moveTo(p: CGPoint) { … }
    func draw()
}

let prius: Car = Car()
let square: Shape = Shape()
```

```
var thingToMove: Moveable = prius
thingToMove.moveTo(…)
thingToMove.changeOil()
thingToMove = square
let thingsToMove: [Moveable] = [prius, square]

func slide(slider: Moveable) {
    let positionToSlideTo = …
    slider.moveTo(positionToSlideTo)
}
slide(prius)
slide(square)
func slipAndSlide(x: protocol<Slippery,Moveable>)
slipAndSlide(prius)
```

Delegation

- A very important use of `protocols`
  - It's a way to implement "blind communication" between a View and its Controller



Delegation

- A very important use of `protocols`

  It's a way to implement "blind communication" between a View and its Controller

- How it plays out ...

  1. A View declares a delegation `protocol` (i.e. what the View wants the Controller to do for it)
  2. The View's API has a `weak` `delegate` property whose type is that delegation `protocol`
  3. The View uses the `delegate` property to get/do things it can't own or control on its own
  4. The Controller declares that it implements the `protocol`
  5. The Controller sets `self` as the `delegate` of the View by setting the property in #2 above
  6. The Controller implements the `protocol` (probably it has lots of `optional` methods in it)

- Now the View is hooked up to the Controller

  But the View still has no idea what the Controller is, so the View remains generic/reusable

- This mechanism is found throughout iOS

  However, it was designed pre-closures in Swift. Closures are often a better option.

Delegation

- Example

  UIScrollView (which we'll talk about in a moment) has a delegate property …

  ```
  weak var delegate: UIScrollViewDelegate?
  ```

  The UIScrollViewDelegate protocol looks like this …

  ```
  @objc protocol UIScrollViewDelegate {
      optional func scrollViewDidScroll(scrollView: UIScrollView)
      optional func viewForZoomingInScrollView(scrollView: UIScrollView) -> UIView
     … and many more …
  }
  ```

  A Controller with a UIScrollView in its View would be declared like this …

  ```
  class MyViewController : UIViewController, UIScrollViewDelegate { … }
  ```

  … and in its viewDidLoad() or in the scroll view outlet setter, it would do …

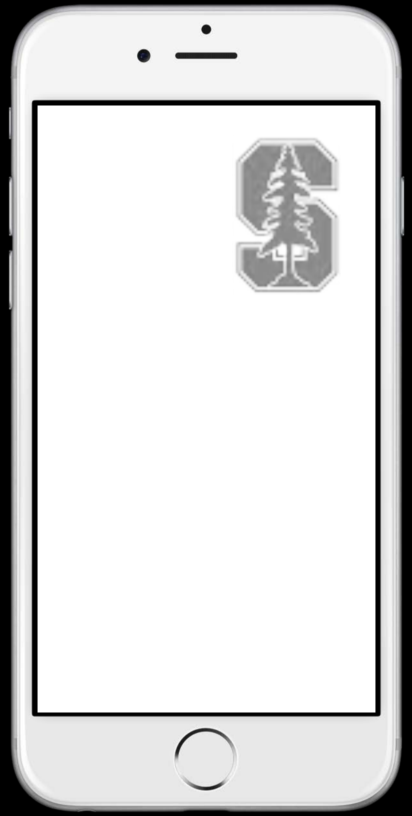
  ```
  scrollView.delegate = self
  ```

  … and it then would implement any of the protocol's methods it is interested in.

UIScrollView

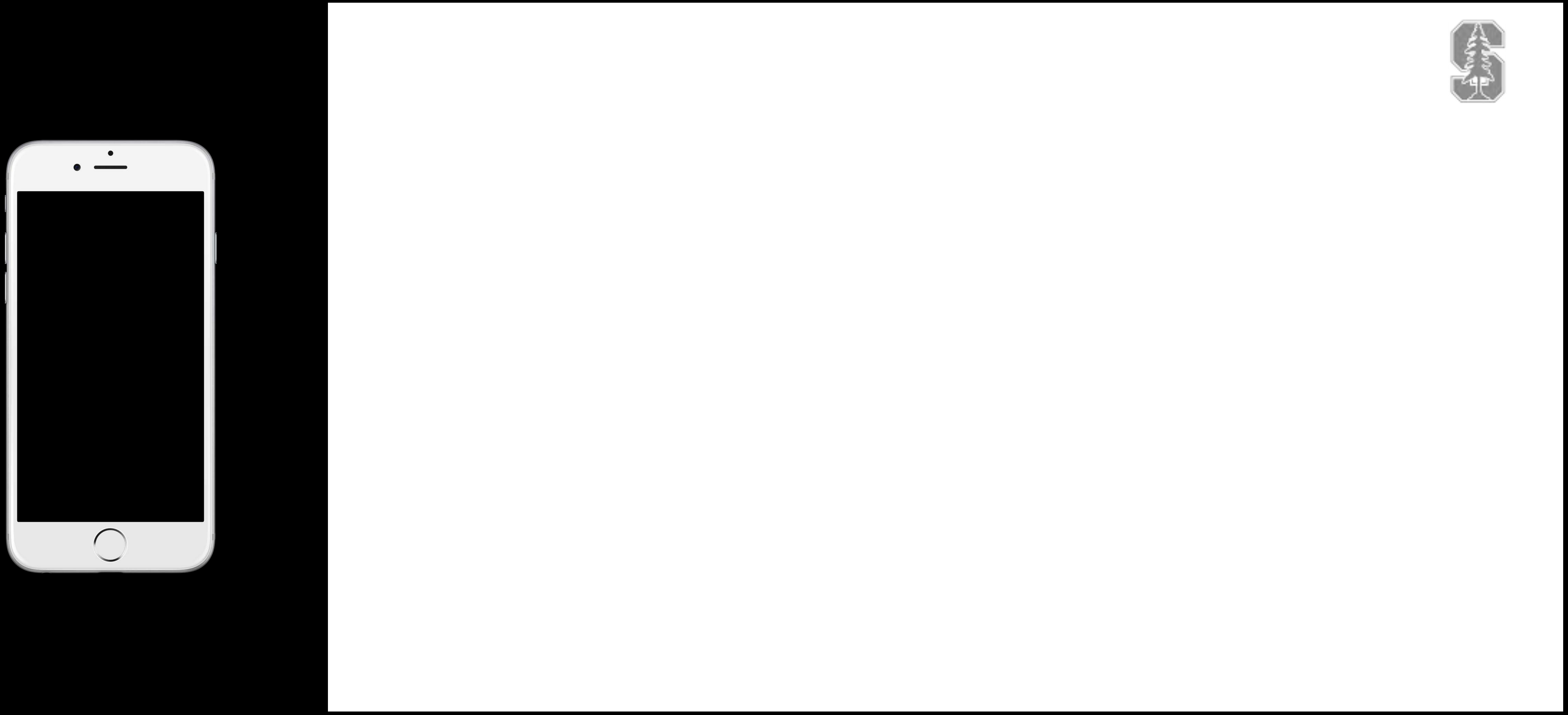
Adding subviews to a normal UIView …

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



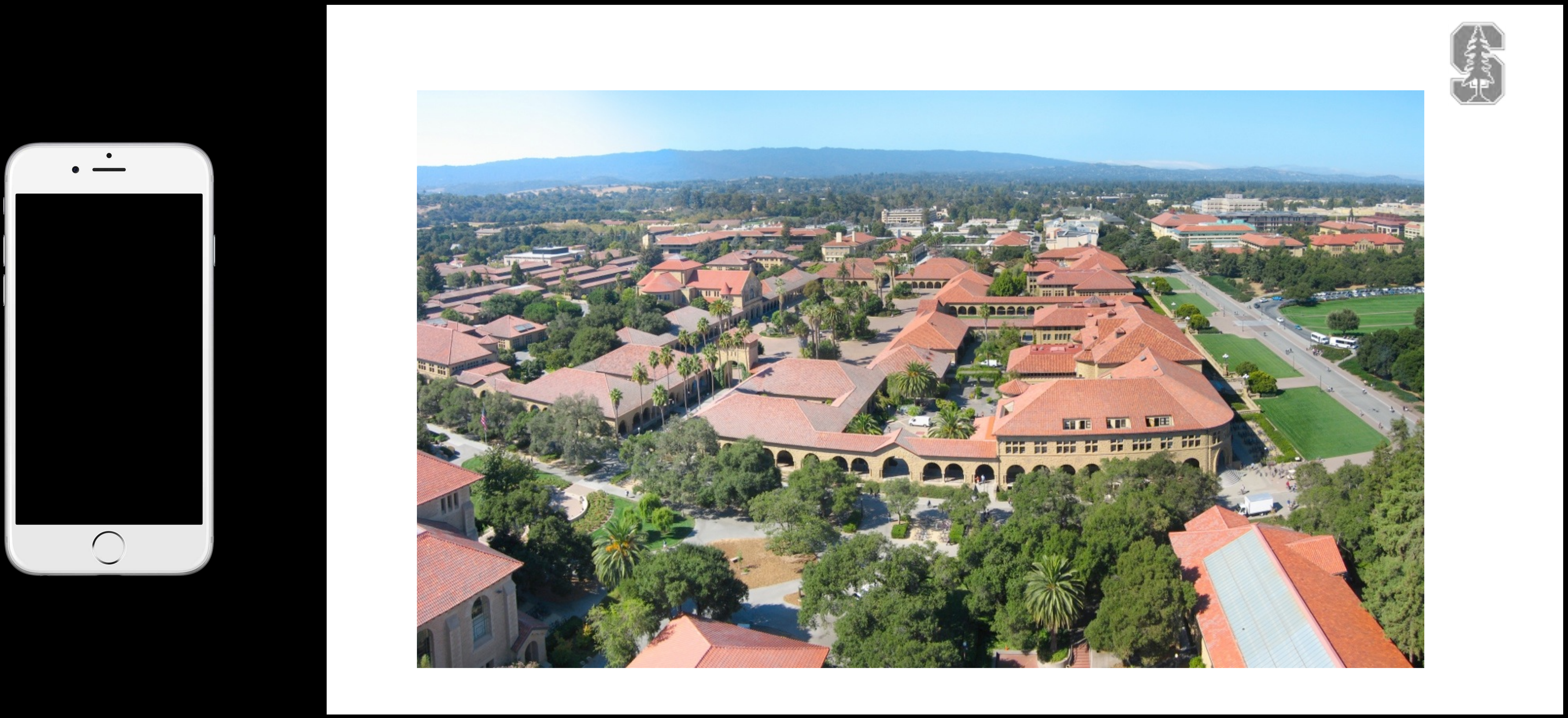
Adding subviews to a UIScrollView ...

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



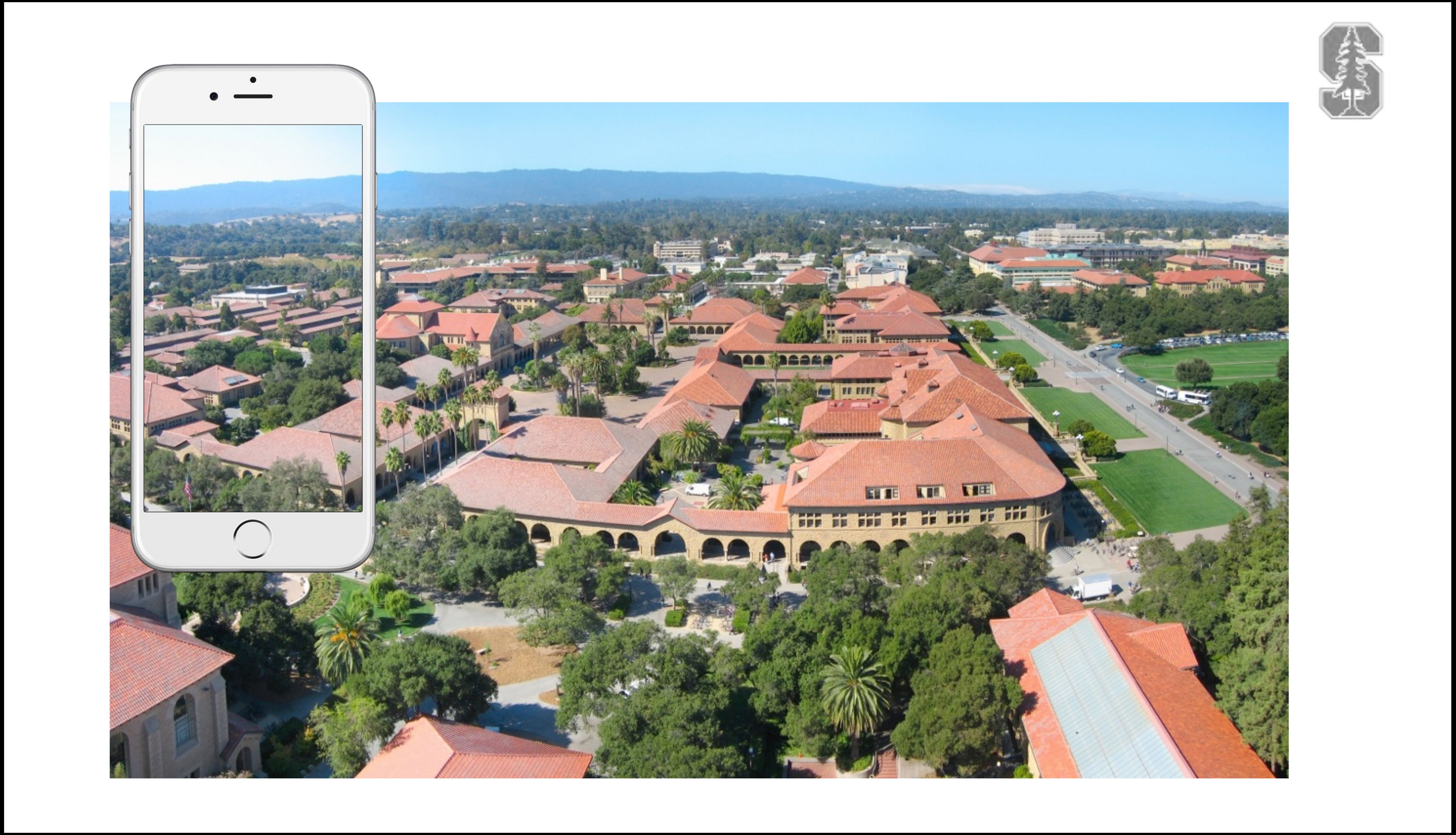
Adding subviews to a UIScrollView …

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

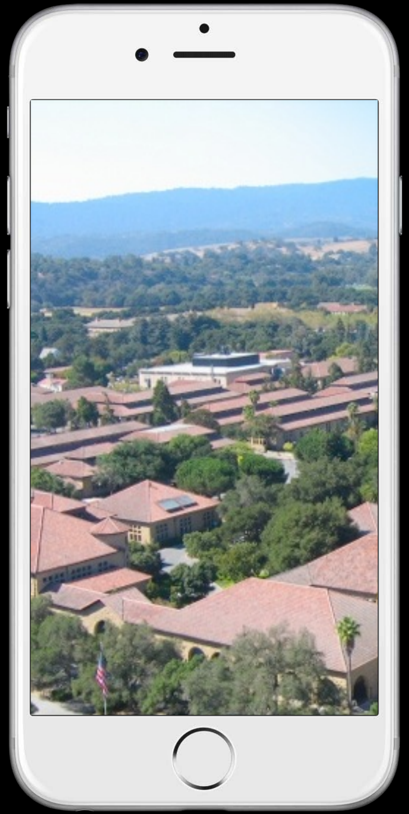


Adding subviews to a UIScrollView …

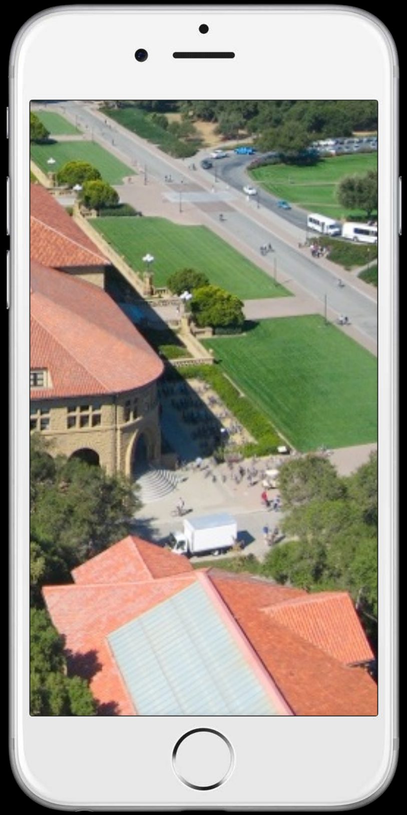
```
scrollView.contentSize = CGSize(width: 3000, height: 2000)
aerial.frame = CGRect(x: 150, y: 200, width: 2500, height: 1600)
scrollView.addSubview(aerial)
```



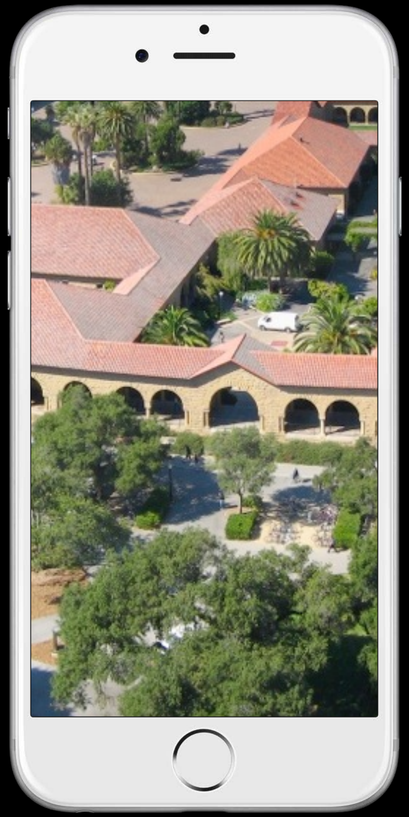
Scrolling in a `UIScrollView` …



Scrolling in a `UIScrollView` …



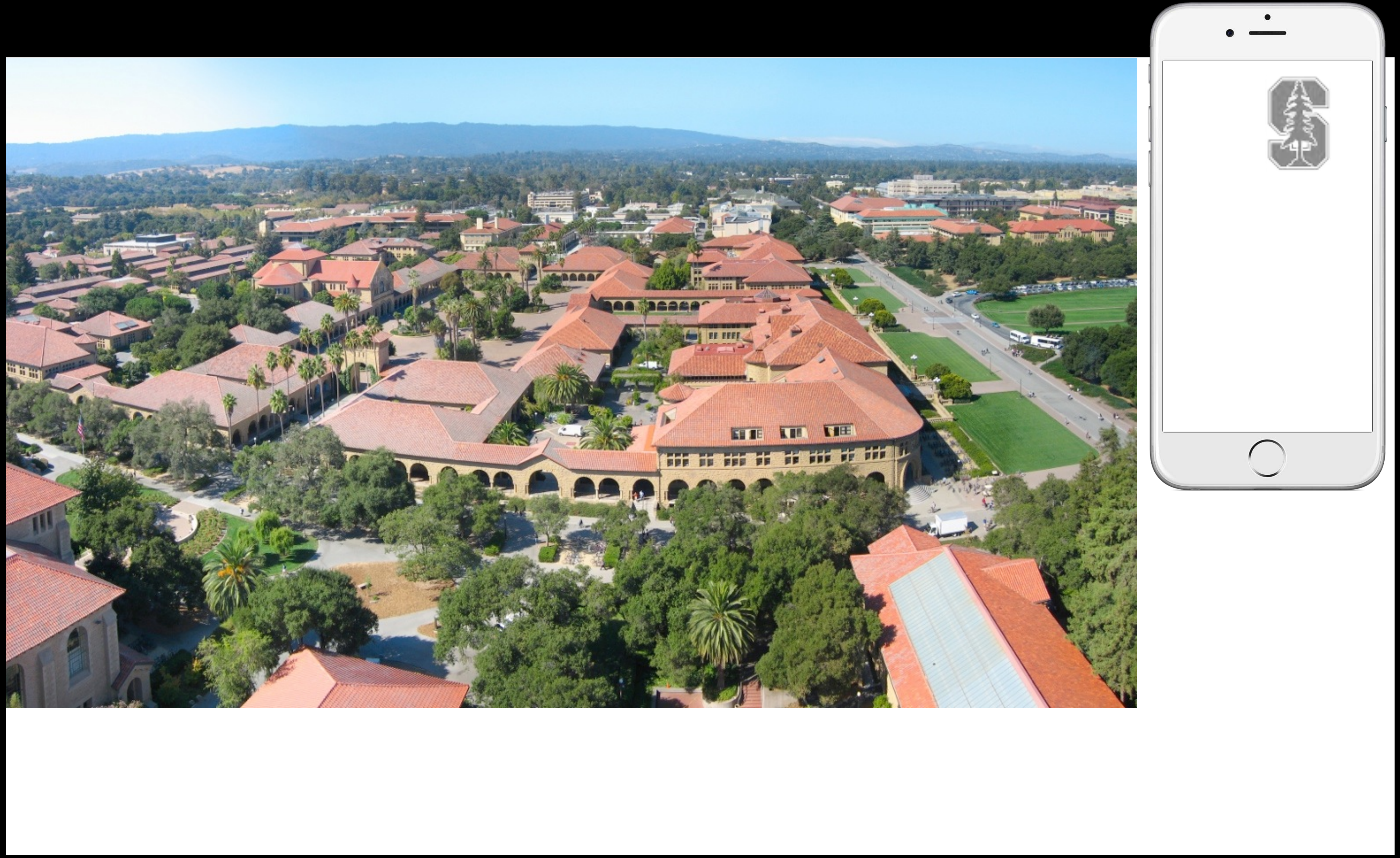
Scrolling in a `UIScrollView` …



Scrolling in a UIScrollView …

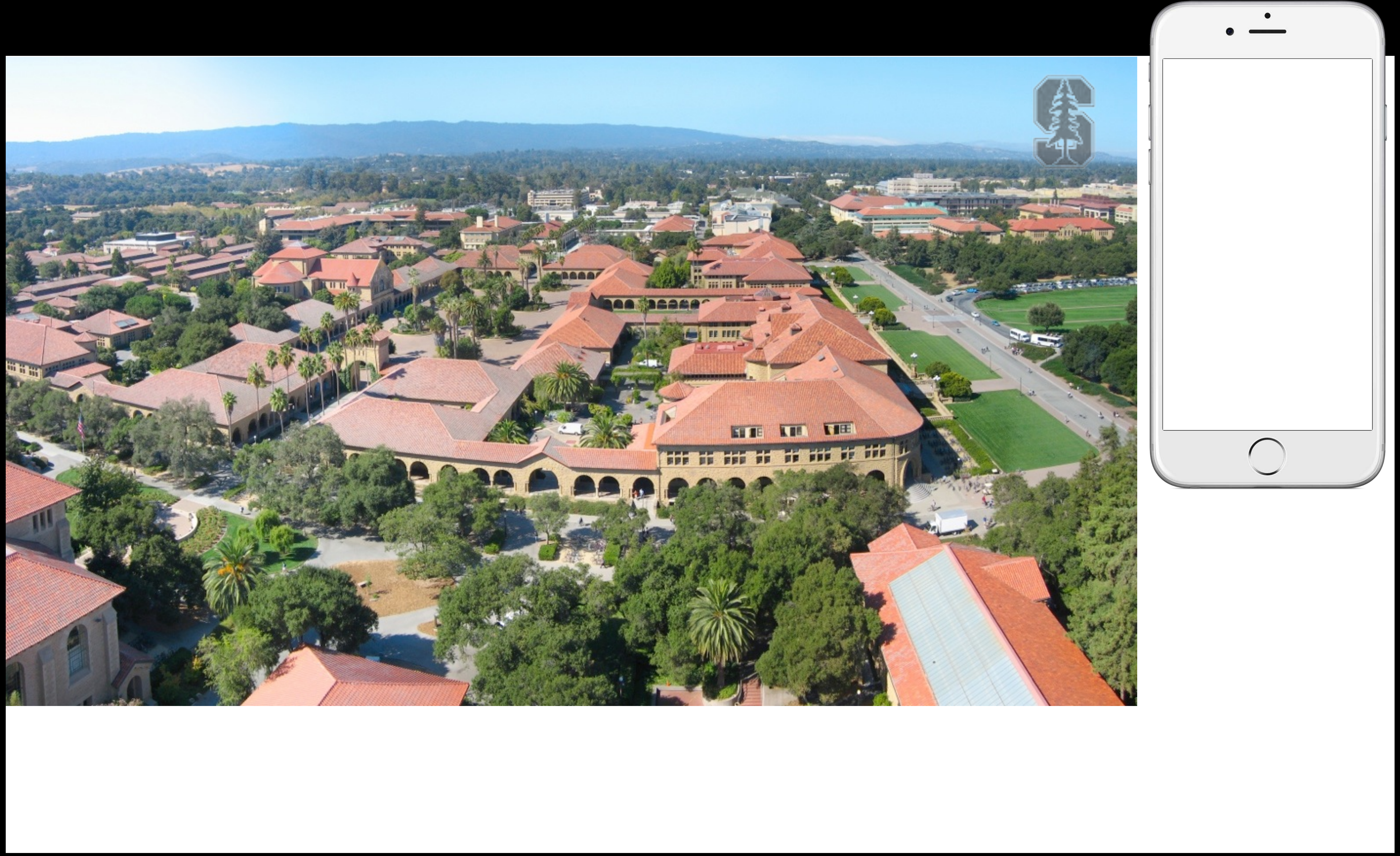
Positioning subviews in a UIScrollView …

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



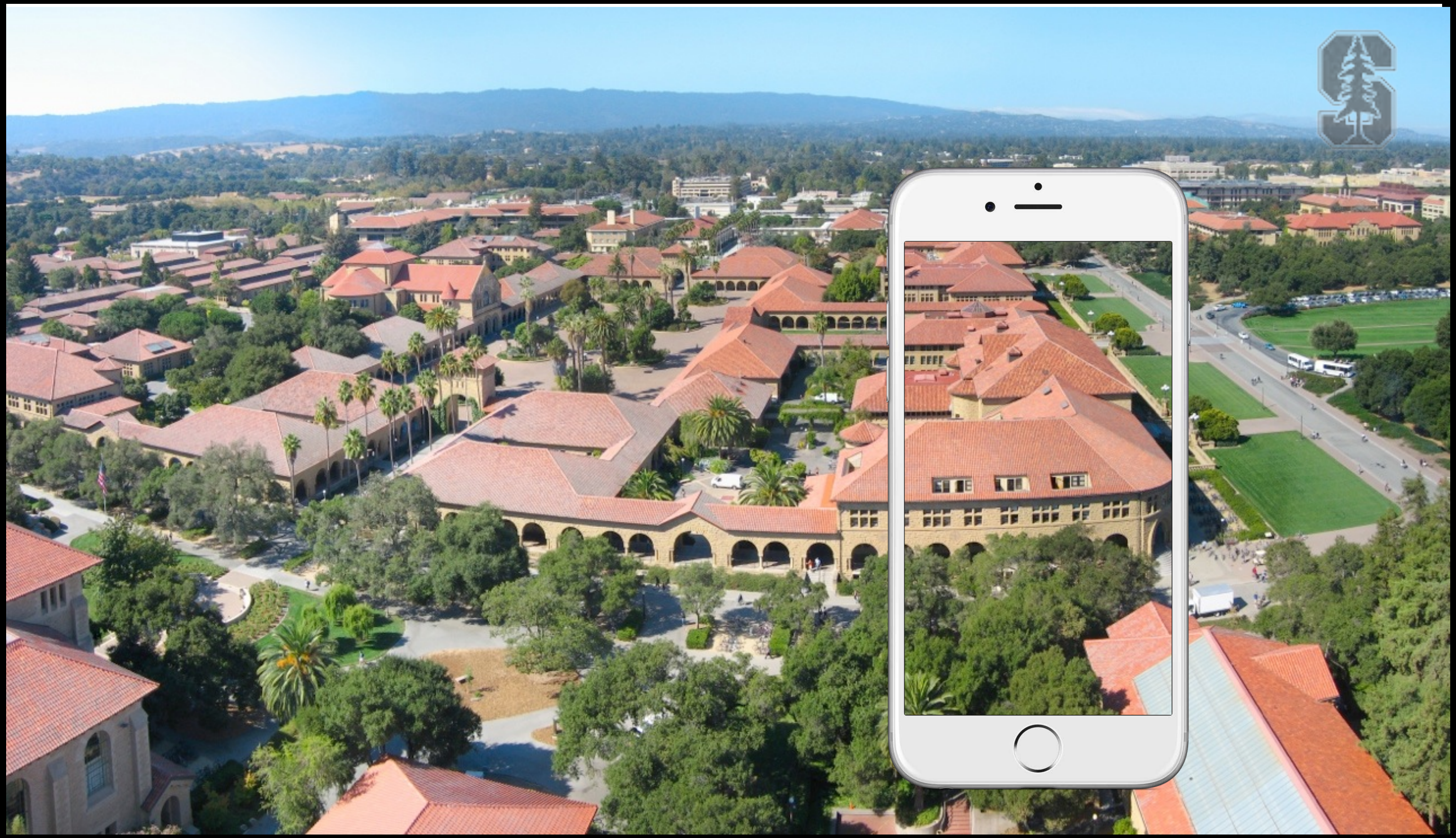
Positioning subviews in a UIScrollView …

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

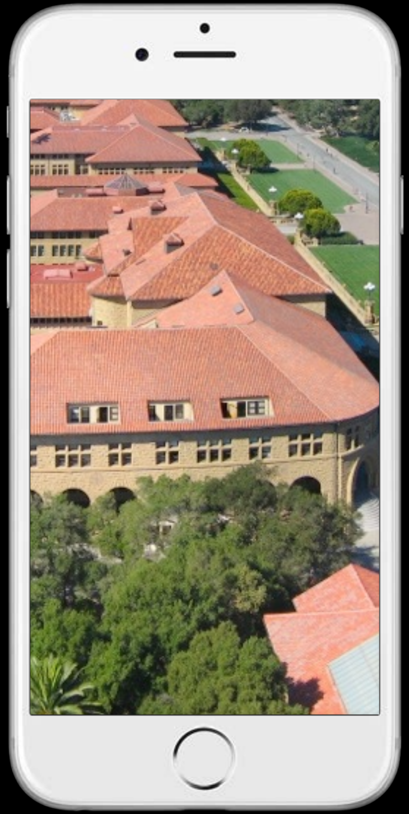


Positioning subviews in a UIScrollView …

```
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)
```

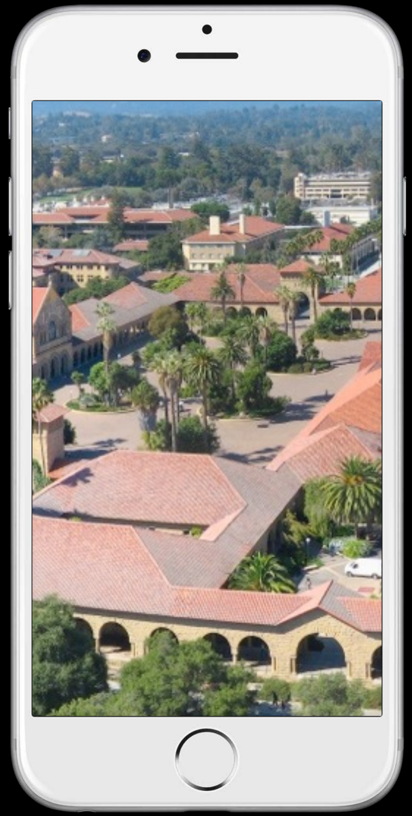


That's it!



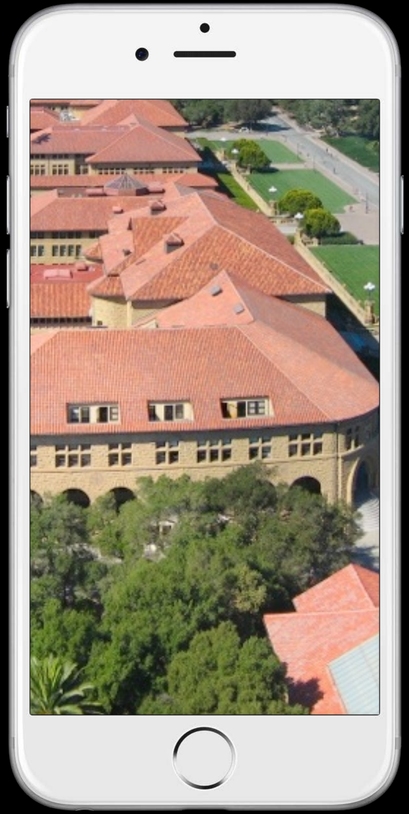
That's it!

That's it!



That's it!

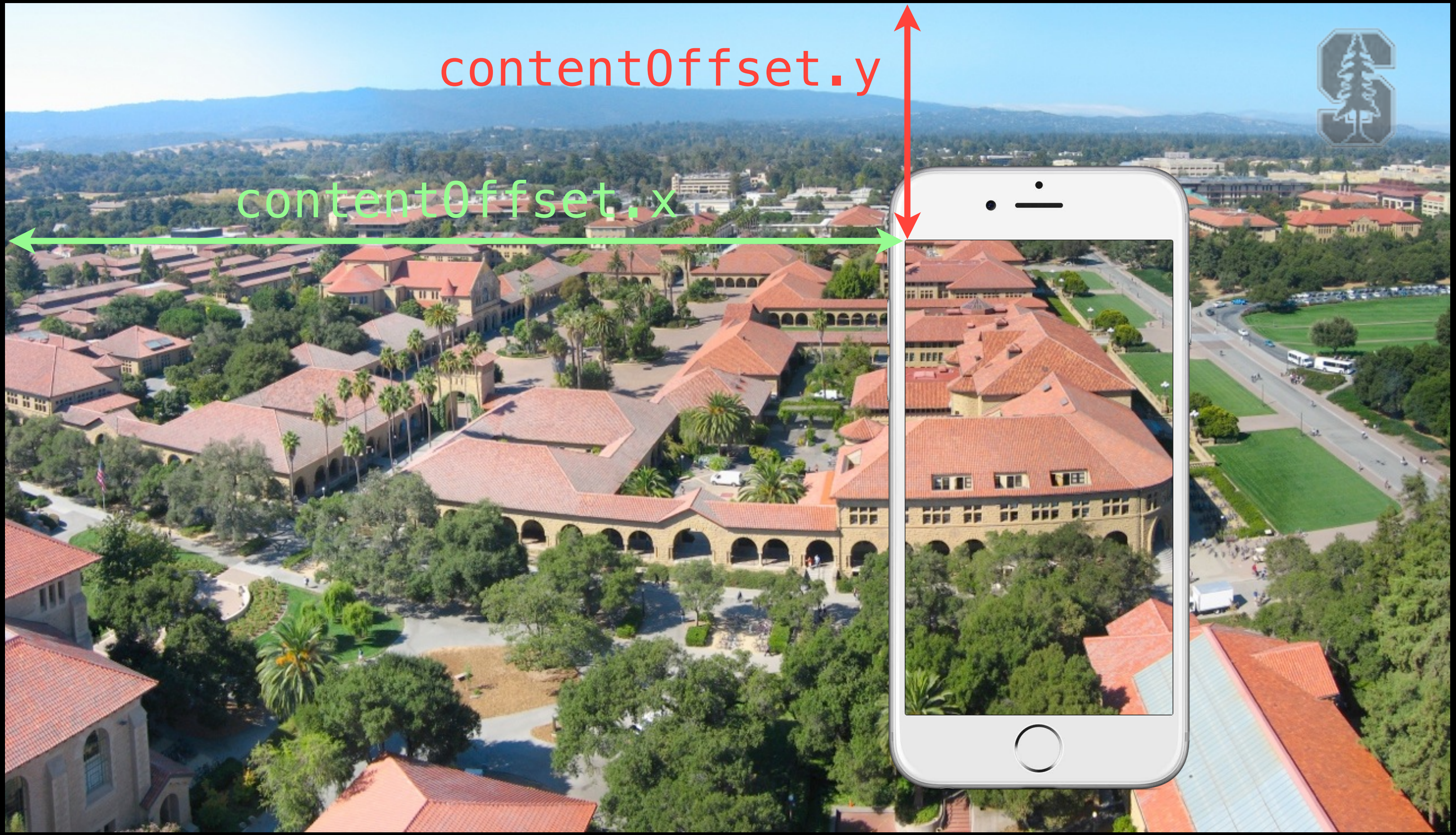
That's it!



Where in the content is the scroll view currently positioned?

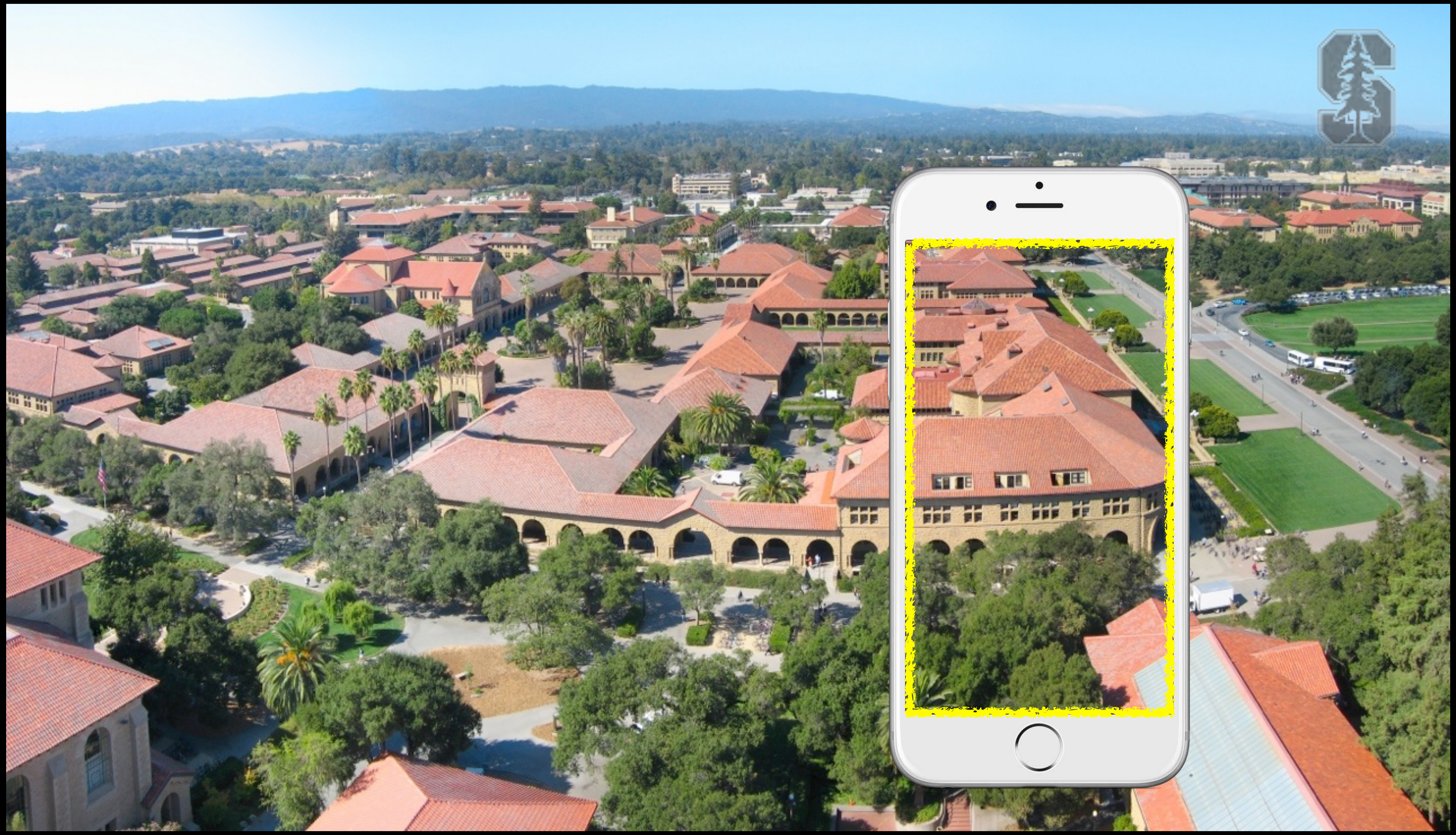
```
let upperLeftOfVisible: CGPoint = scrollView.contentOffset
```

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
```

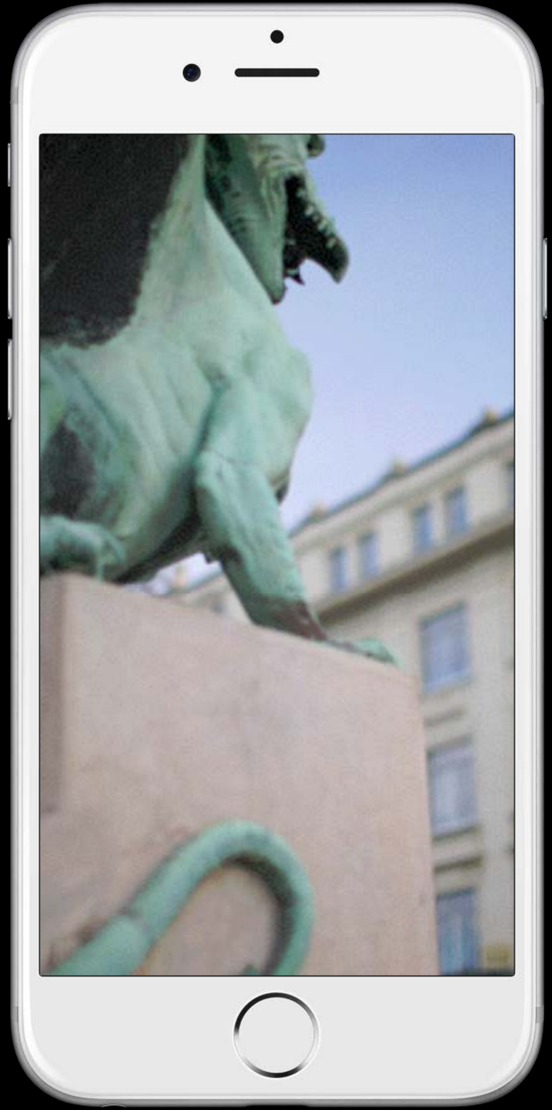
  If your scroll view only has one subview, you return it here. More than one? Up to you.

- Zooming programatically

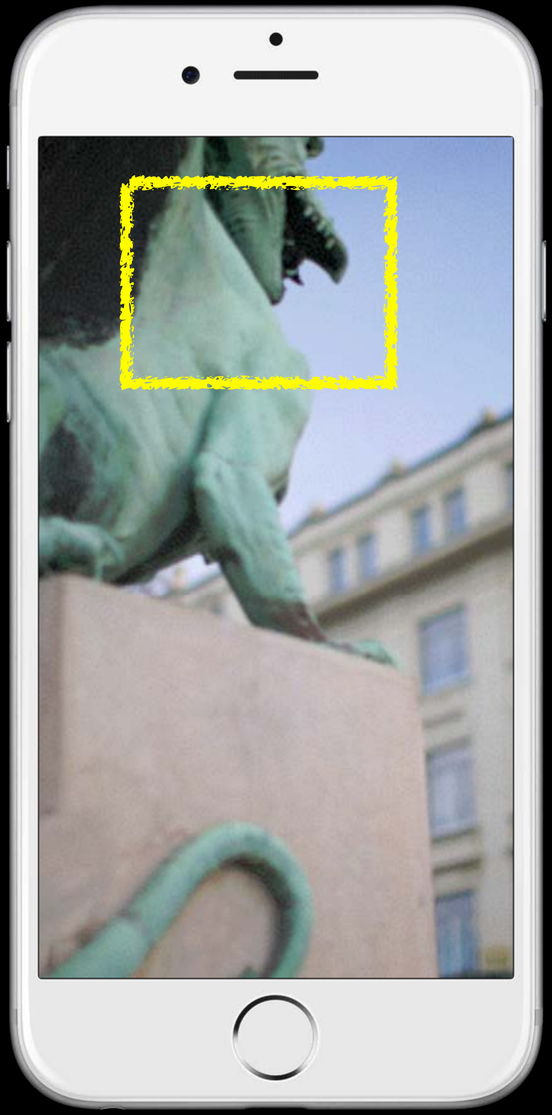
```
var zoomScale: CGFloat
func setZoomScale(CGFloat, animated: Bool)
func zoomToRect(CGRect, animated: Bool)
```

```
scrollView.zoomScale = 1.2
```

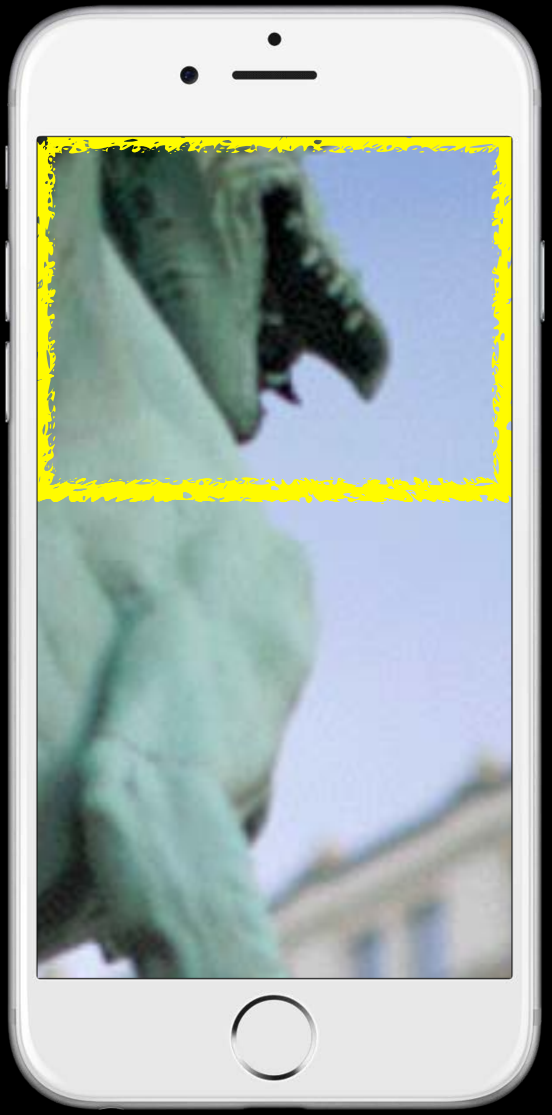
```
scrollView.zoomScale = 1.0
```



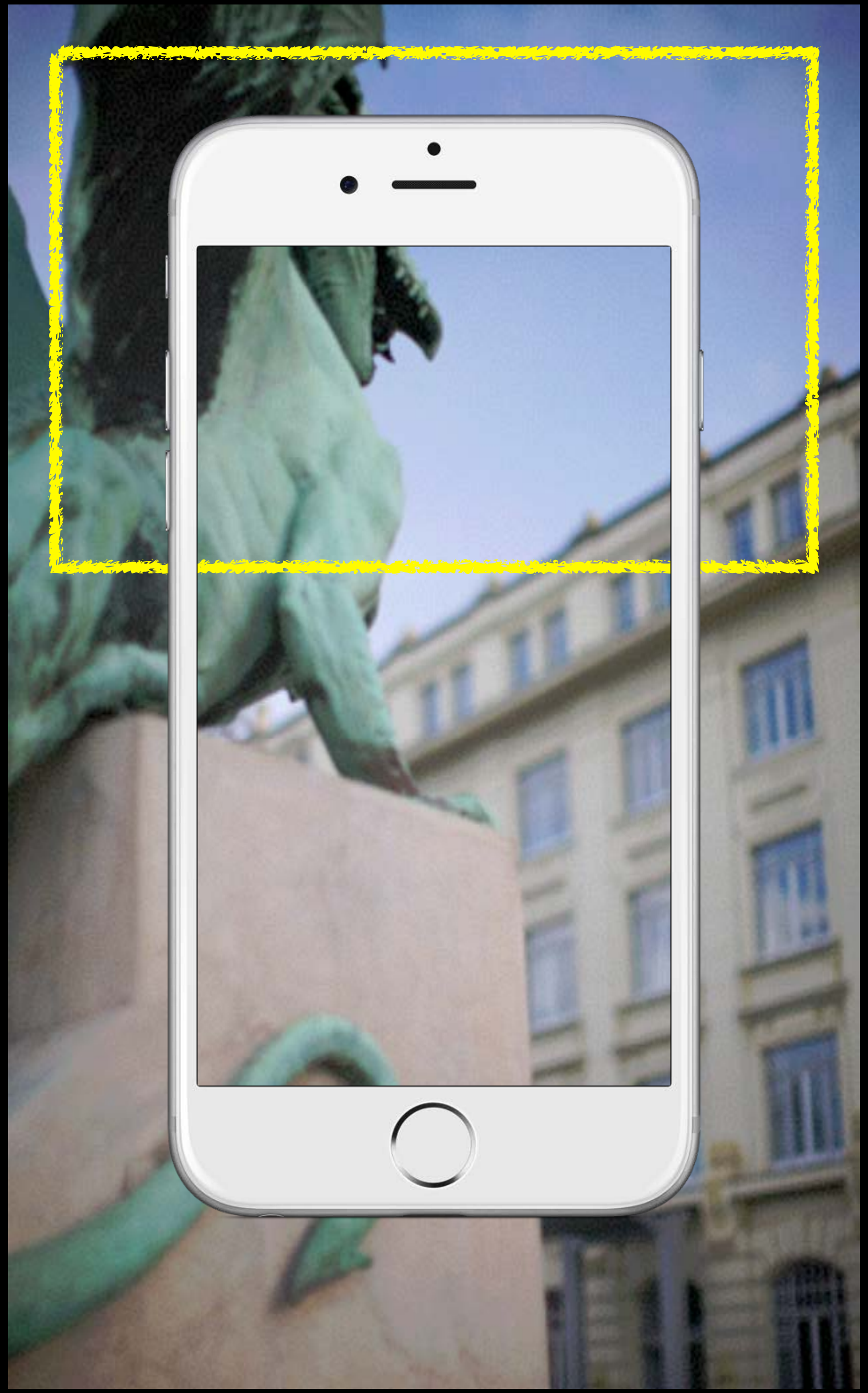
```
scrollView.zoomScale = 1.2
```

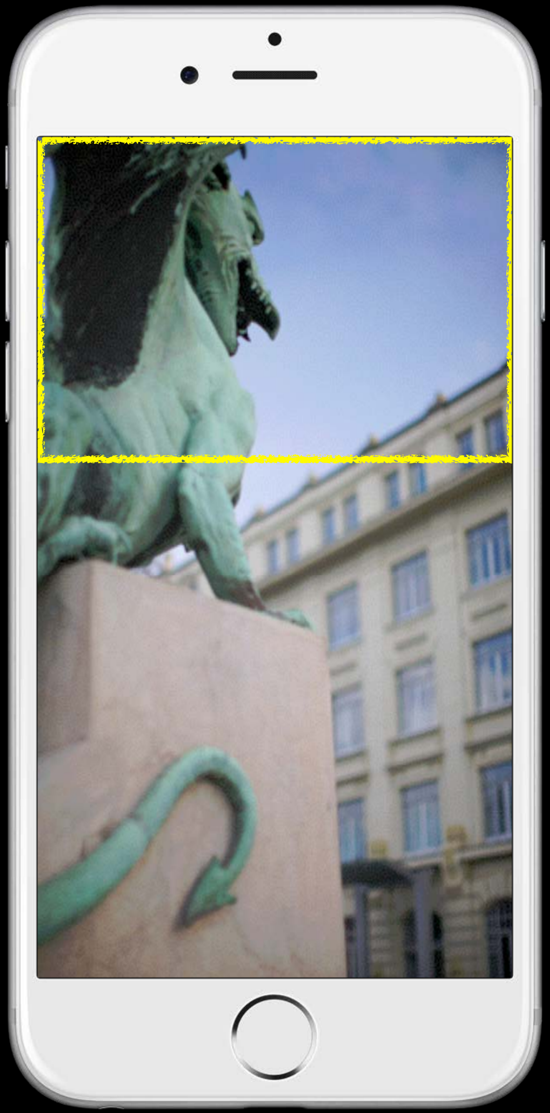


```
zoomToRect(CGRect, animated: Bool)
```



```
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.