Developing Applications for iOS

Lecture 5:
Views, Drawing and Gestures

Prof. Dr. Radu Ionescu
raducu.ionescu@gmail.com
Faculty of Mathematics and Computer Science
University of Bucharest

Content

Views, Drawing and Gestures:

- Drawing Paths
- Drawing Text
- Drawing Images
- Error Handling
- Gesture Recognizers

Custom Views

When would I want to create my own `UIView` subclass?

- I want to do some custom drawing on screen.
- I need to handle touch events in a special way (i.e. different than a button or slider does).
- We will talk about handling touch events later. For now we are focussing on drawing.

Drawing is easy

- Create a `UIView` subclass and override one method:

```
func draw(_ rect: CGRect)
```

- You can optimize by not drawing outside of the `rect` if you want (but not required).

Custom Views

NEVER call `draw()`!

- Instead, let iOS know that your view's visual is out of date with one of these `UIView` methods:

```
func setNeedsDisplay()
```

```
func setNeedsDisplay(_ rect: CGRect)
```

- It will then set everything up and call `draw()` for you at an appropriate time.
- Obviously, the second version will call your `draw()` with only the rectangle that need updates.

Custom Views

- So how do I implement my own `draw()` method?
- We use the Core Graphics framework.

Concepts

- Get a context to draw into (iOS will prepare one each time your `draw()` method is called).
- Create paths (out of lines, arcs, etc).
- Set colors, fonts, textures, line widths, line caps, etc.
- Stroke or fill the above-created paths.

Context

The context determines where your drawing goes

- Screen (the only one we are going to talk about today)
- Offscreen Bitmap
- PDF
- Printer

For normal drawing, `UIKit` sets up the current context for you

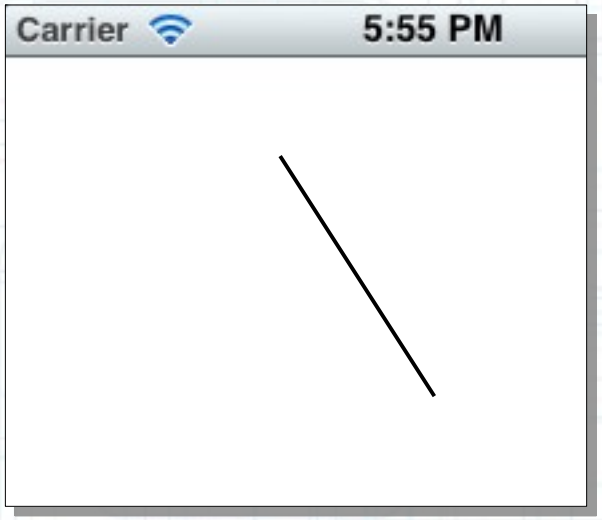
- But it is only valid during that particular call to `draw()`.
- A new one is set up for you each time `draw()` is called. So never cache the current graphics context in `draw()` to use later!

How to get this magic context?

- Call the following function inside your `draw()` method to get the current graphics context:

```
let context = UIGraphicsGetCurrentContext()
```

Define a Path



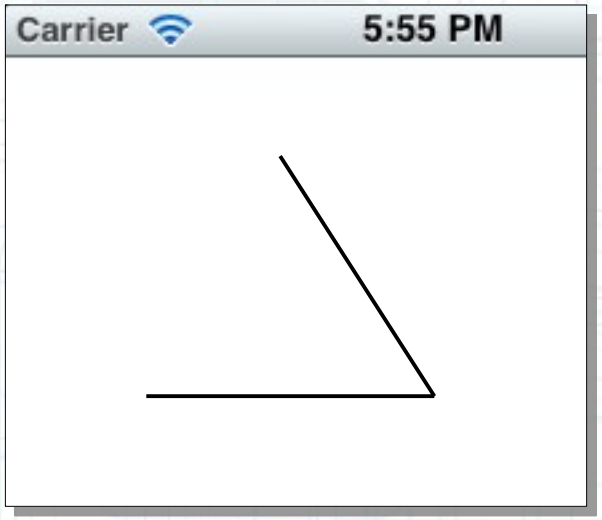
- Begin the path:

```
context.beginPath();
```

- Move around, add lines or arcs to the path:

```
context.move(to: CGPoint(x: 85, y: 30))
context.addLine(to: CGPoint(x: 160, y: 170))
```

Define a Path



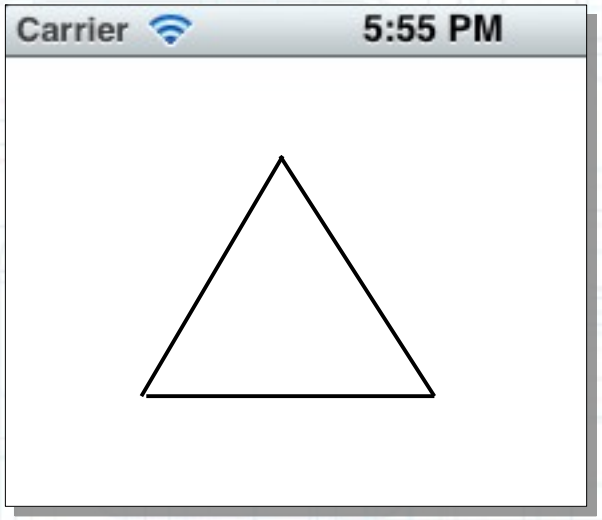
- Begin the path:

```
context.beginPath();
```

- Move around, add lines or arcs to the path:

```
context.move(to: CGPoint(x: 85, y: 30))
context.addLine(to: CGPoint(x: 160, y: 170))
context.addLine(to: CGPoint(x: 10, y: 170))
```

Define a Path



- Begin the path:

```
context.beginPath();
```

- Move around, add lines or arcs to the path:

```
context.move(to: CGPoint(x: 85, y: 30))
context.addLine(to: CGPoint(x: 160, y: 170))
context.addLine(to: CGPoint(x: 10, y: 170))
```

- Close the path (connects the last point back to the first):

```
context.closePath() // not strictly required
```

Define a Path

- Begin the path:

```
context.beginPath();
```

- Move around, add lines or arcs to the path:

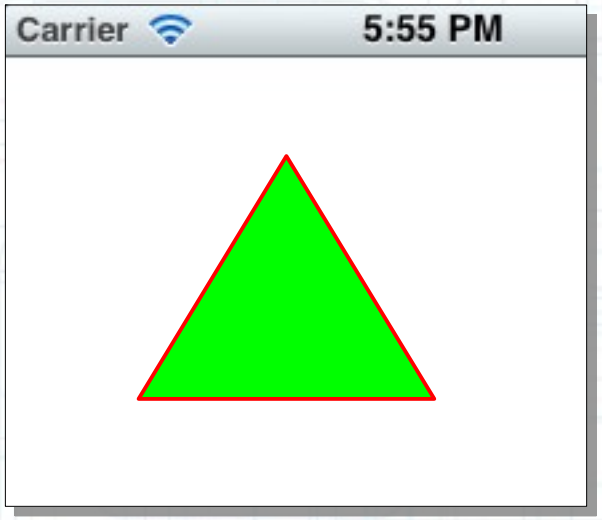
```
context.move(to: CGPoint(x: 85, y: 30))
context.addLine(to: CGPoint(x: 160, y: 170))
context.addLine(to: CGPoint(x: 10, y: 170))
```

- Close the path (connects the last point back to the first):

```
context.closePath() // not strictly required
```

- Actually the above draws nothing (yet)!
- You have to set the graphics state and then fill/stroke the above path to see anything.

Define a Path



- Begin the path:

```
context.beginPath()
```

- Move around, add lines or arcs to the path:

```
context.move(to: CGPoint(x: 85, y: 30))
context.addLine(to: CGPoint(x: 160, y: 170))
context.addLine(to: CGPoint(x: 10, y: 170))
```

- Close the path (connects the last point back to the first):

```
context.closePath() // not strictly required
```

- Actually the above draws nothing (yet)!
- You have to set the graphics state and then fill/stroke the above path to see anything.

```
UIColor.green.setFill()
UIColor.red.setStroke()
/* object-oriented convenience methods
   (more in a moment) */
context.drawPath(using: .fillStroke)
// .fillStroke is an CGPathDrawingMode enum case
```

Define a Path

- You can draw arcs, rectangles, ellipses and so on.
- It is also possible to save a path and reuse it.
- There are similar functions to the previous slide that let you do this.
- We won't be covering those, but you can certainly feel free to look up `CGPath` and `CGContext` in the documentation.

Graphics State

`UIColor` class for setting colors

- It has class methods for creating most common colors:

```
let red = UIColor.red
let invisible = UIColor.clear
```

- Custom colors can be created using initializers:

```
let custom = UIColor(red: 0.5,
                     green: 0.8,
                     blue: 1.0,
                     alpha: 0.8)

let woodTexture = UIImage(contentsOfFile: "wood.png")
let pattern = UIColor(patternImage: woodTexture!)
```

- To set the fill/stroke color in current graphics context:

```
pattern.setFill()
red.setStroke()
custom.set() /* sets both stroke and fill color to
                custom (would override the previous) */
```

View Transparency

Drawing with transparency in `UIView`

- Note the `alpha` component of `UIColor`s. This is how you can draw with transparency in your `drawRect:`.
- `UIView` also has a `backgroundColor` property which can be set to transparent values.
- Be sure to set `Bool opaque` property to `false` in a view which is partially or fully transparent.
- If you don't, results are unpredictable (this is a performance optimization property, by the way).
- The `UIView`'s `CGFloat alpha` property can make the entire view partially transparent.

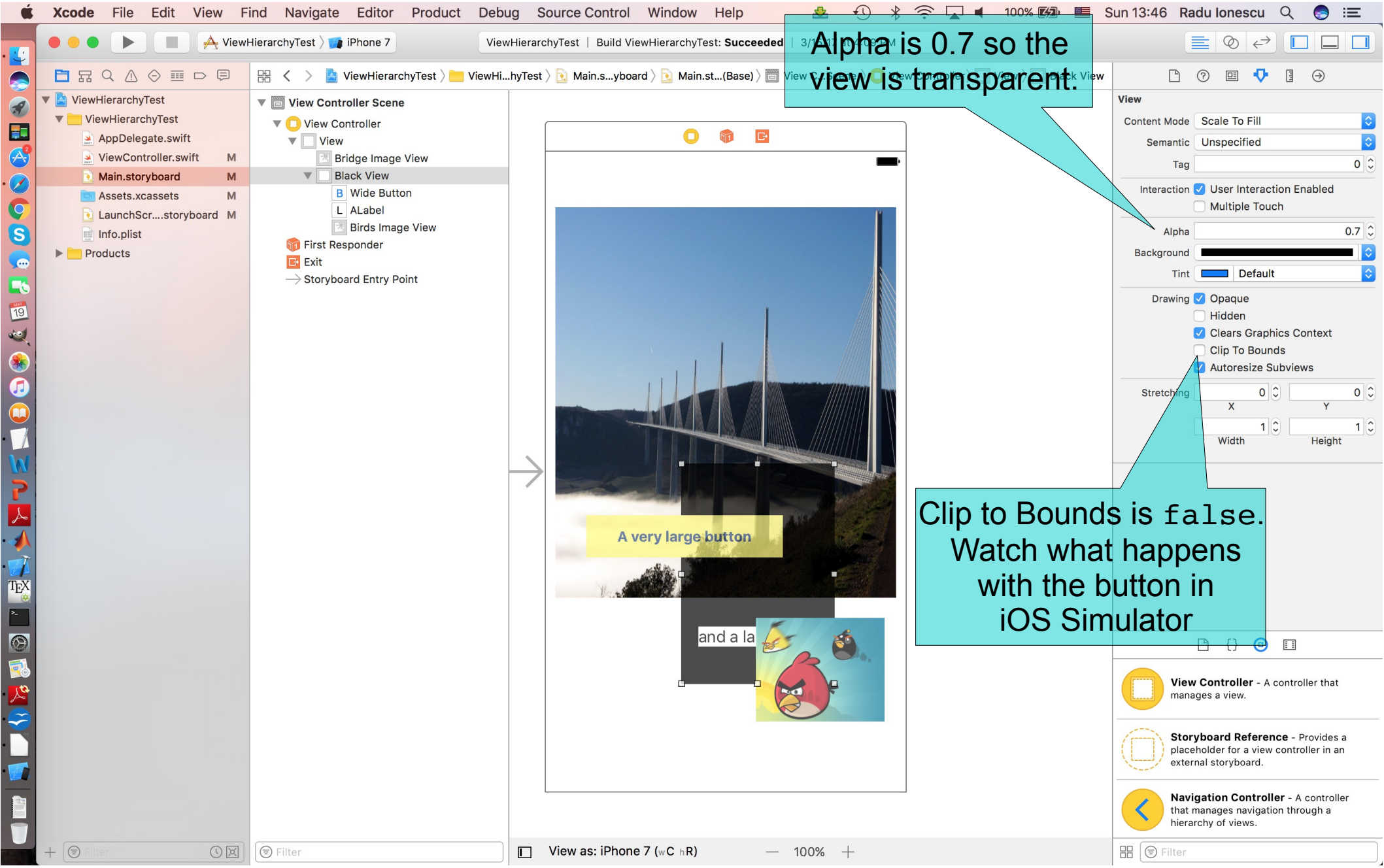
View Transparency

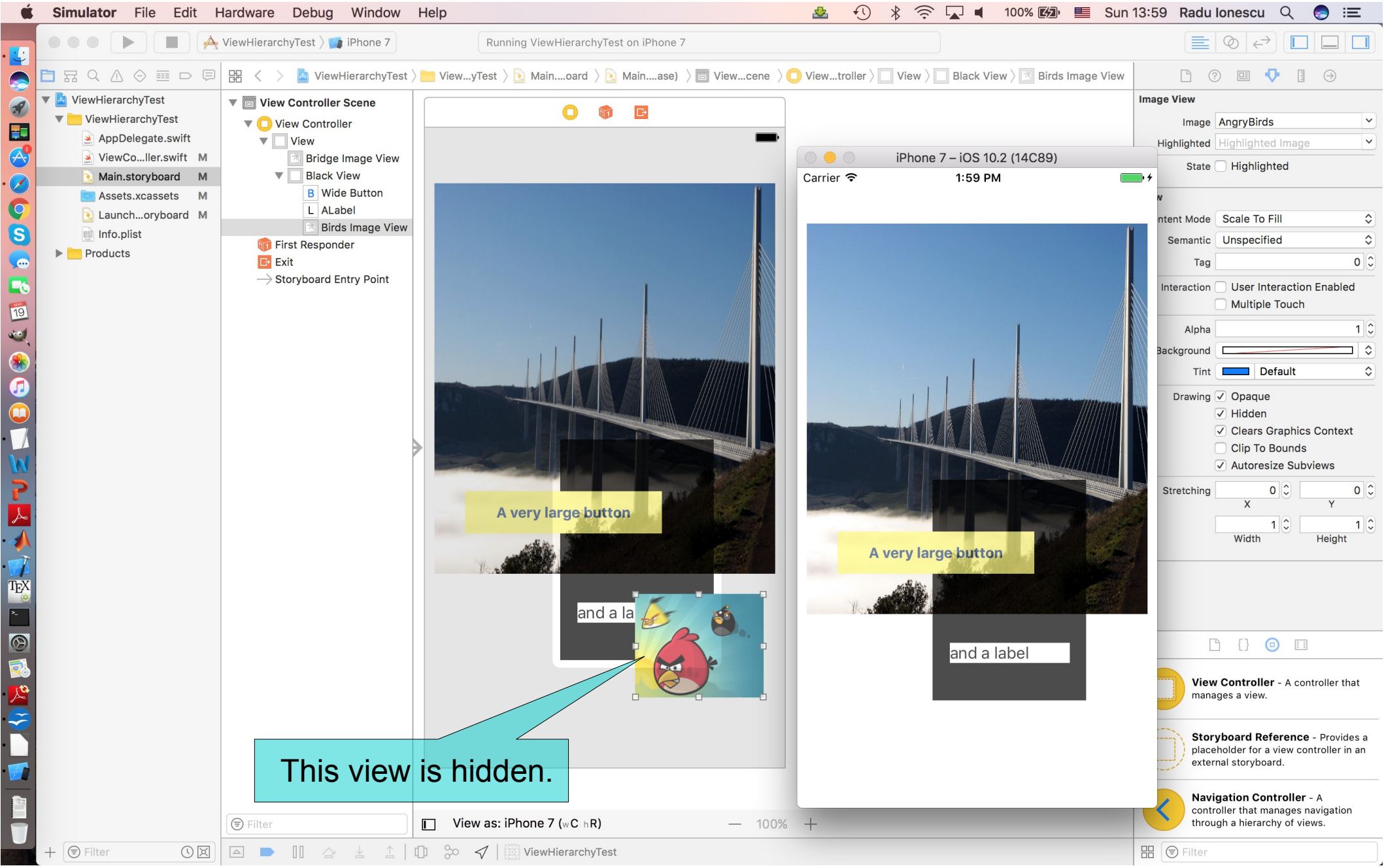
What happens when views overlap?

- As mentioned before, `subviews` list order determines who's in front.
- Lower ones (earlier in `subviews` array) can "show through" transparent views on top of them.
- Default drawing is opaque. Transparency is not cheap when you think of performance.
- Also, you can hide a view completely by setting the `Bool hidden` property.
- The view will not be on screen and will not handle events if you set:

```
myView.hidden = true;
```

- This is not as uncommon as you might think. On a small screen, keeping it de-cluttered by hiding currently unusable views make sense.


Alpha is 0.7 so the view is transparent.
Clip to Bounds is false. Watch what happens with the button in iOS Simulator
A very large button
and a la
View Controller - A controller that manages a view.
Storyboard Reference - Provides a placeholder for a view controller in an external storyboard.
Navigation Controller - A controller that manages navigation through a hierarchy of views.


Simulator File Edit Hardware Debug Window Help
Sun 13:59 Radu Ionescu
ViewHierarchyTest
AppDelegate.swift
ViewCo...ller.swift M
Main.storyboard M
Assets.xcassets M
Launch...oryboard M
Info.plist
Products
View Controller Scene
View Controller
View
Bridge Image View
Black View
Wide Button
ALabel
Birds Image View
First Responder
Exit
Storyboard Entry Point
A very large button
and a la
iPhone 7 – iOS 10.2 (14C89)
Carrier
1:59 PM
A very large button
and a label
This view is hidden.
Image View
Image AngryBirds
Highlighted Highlighted Image
State Highlighted
Scale To Fill
Unspecified
Tag
User Interaction Enabled
Multiple Touch
Alpha
Background
Tint Default
Drawing Opaque
Hidden
Clears Graphics Context
Clip To Bounds
Autoresize Subviews
Stretching
X
Y
Width
Height
View Controller - A controller that manages a view.
Storyboard Reference - Provides a placeholder for a view controller in an external storyboard.
Navigation Controller - A controller that manages navigation through a hierarchy of views.
View as: iPhone 7 (wC hR)
100%

Graphics State

Some other graphics state settings

- Set line width (in points, not pixels):

```
context.setLineWidth(5.0)
```

- Set the fill pattern in specified graphics context:

```
context.setFillPattern(pattern: CGPattern,
        colorComponents: UnsafePointer<CGFloat>)
```

Graphics State

Special considerations for defining drawing “subroutines”

- What if you wanted to have a utility method that draws something
- You don’t want that utility method to mess up the graphics state of the calling method.
- Use save and restore graphics state functions:

```
func drawBlueCircle(context: CGContext)
{
    context.saveGState()
    UIColor.blue.setFill()
    let r = CGRect(x: 20, y: 150, width: 100, height: 100)
    context.addEllipse(in: r)
    context.drawPath(using: .fill)
    context.restoreGState()
}
override func draw(_ rect: CGRect)
{
    ...
    self.drawBlueCircle(context: context)
    ...
}
```

Drawing Text

- Use `UILabel` to draw text, but if you feel you must ...
- Use `UIFont` object in `UIKit` to get a font:

```
let f = UIFont.systemFont(ofSize: 18.0)
let h = UIFont(name: "Helvetica", size: 36.0)
let availableFonts = UIFont.familyNames
```

- Then use special `NSString` methods to draw the text:

```
let text: NSString = "Text is drawn"
text.draw(at: CGPoint(x: 20, y: 300),
    withAttributes: [NSAttributedString.Key.font : f])
// NSString instance method
```

- How much space will a piece of text will take up when drawn?

```
let textSize = text.size(attributes:
                [NSAttributedString.Key.font : h ?? f])
// NSString instance method
```

Drawing Text

- You might be disturbed that there is a Foundation method for drawing (which is a UIKit thing).
- But actually these `NSString` methods are defined in UIKit via **extensions**.
- Remember: Extensions are the Swift's way to add methods to an existing class without subclassing.

Drawing Images

- Use `UIImageView` to draw images, but if you feel you must ...

  (Note that we will cover `UIImageView` later in the course)

- Create a `UIImage` object from a file in your Resources folder:

```
let image = UIImage(named: "foo.jpg")
```

- Or create one from a named file or from raw data:

  (of course, we haven't talked about the file system yet, but …)

```
let img1 = UIImage(contentsOfFile: path)
let url = URL(string: "http://www.web.com/img.png")
let data = try Data(contentsOf: url!)
let img2 = UIImage(data: data)
```

- Or you can even create one by drawing with `CGContext` functions:

```
UIGraphicsBeginImageContext(CGSize(width: 100,
                                   height: 100))
// draw with CGContext functions
let img = UIGraphicsGetImageFromCurrentImageContext()
UIGraphicsEndImageContext()
```

Drawing Images

- To draw the `UIImage`'s bits into the current graphics context:

```
let img = ...
img?.draw(at: CGPoint(x: 20, y: 350))
// at: is the upper left corner of the image
```

- Scale the image to fit in a `CGRect`:

```
img?.draw(in: CGRect(x: 0, y: 0,
                     width: 80, height: 80))
```

- Tile the image into a `CGRect`:

```
img?.drawAsPattern(in: CGRect(x: 0, y: 0,
                              width: 80, height: 80))
```

- You can also get a PNG or JPG data representation of `UIImage`:

```
let jpgData = UIImageJPEGRepresentation(finalImage!,
                                        CGFloat(0.8))

let pngData = UIImagePNGRepresentation(finalImage!)
```

Error Handling

- Some operations aren't guaranteed to always complete execution or produce a useful output.
- Optionals are used to represent the absence of a value, but when an operation fails, it's often useful to understand what caused the failure, so that your code can respond accordingly.
- In Swift, errors are represented by values of types that conform to the `Error` protocol.
- Swift enumerations are particularly well suited to modeling a group of related error conditions with associated values:

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

Throwing and Handling Errors

- Throwing an error lets you indicate that something unexpected happened and the normal flow of execution can't continue. You use a `throw` statement to throw an error:

```
throw VendingMachineError.insufficientFunds(coinsNeeded: 5)
```

- When a function throws an error, it changes the flow of your program, so it is important that you can quickly identify places in your code that can throw errors. You do this by writing the `try` keyword before a piece of code that can throw an error.

There are four ways to handle errors in Swift:

- You can propagate the error from a function to the code that calls that function
- You can handle the error using a `do-catch` statement
- You can handle the error as an optional value
- You can assert that the error will not occur

Propagating Errors

- To indicate that a function, method, or initializer can throw an error, you write the `throws` keyword in the function's declaration after its parameters. A function marked with throws is called a throwing function:

```
func vend(itemNamed name: String) throws {
    guard let item = inventory[name] else {
        throw VendingMachineError.invalidSelection
    }
    guard item.count > 0 else {
        throw VendingMachineError.outOfStock
    }
    guard item.price <= coinsDeposited else {
    throw VendingMachineError.insufficientFunds
            (coinsNeeded: item.price – coinsDeposited)
    }
    coinsDeposited -= item.price
    var newItem = item
    newItem.count -= 1
    inventory[name] = newItem
    print("Dispensing \(name)")
}
```

Handling Errors using `do-catch`

- You use a `do-catch` statement to handle errors by running a block of code. If an error is thrown by the code in the `do` clause, it is matched against the `catch` clauses to determine which one of them can handle the error:

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice",
                         vendingMachine: vendingMachine)
}
catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
}
catch VendingMachineError.outOfStock {
    print("Out of Stock.")
}
catch VendingMachineError.insufficientFunds(let coins) {
    print("Insufficient funds. Insert  \(coins) coins.")
}
// Prints "Insufficient funds. Insert 2 coins."
```

Converting Errors to Optional Values

- You use `try?` to handle an error by converting it to an optional value. If an error is thrown while evaluating the `try?` expression, the value of the expression is `nil`. For example, in the following code `x` and `y` have the same value and behavior:

```
func someThrowingFunction() throws -> Int {
    // …
}

let x = try? SomeThrowingFunction()

let y: Int ?

do {
    y = try someThrowingFunction()
}
catch {
    y = nil
}
```

Specifying Cleanup Actions

- You use a `defer` statement to execute a set of statements just before code execution leaves the current block of code.

  This statement lets you do any necessary cleanup that should be performed regardless of how execution leaves the block of code.

- The deferred statements may not contain any code that would transfer control out of the statements, such as `break` or `return`, or by throwing an error.

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }

        while let line = try file.readline() {
            // work with the file
        }
        // close(file) is called here
    }
}
```

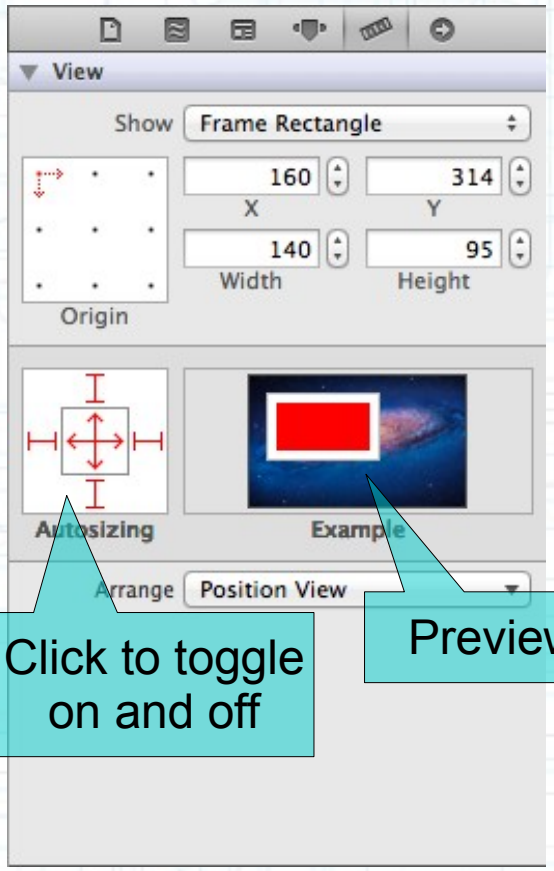
Autorotation

If you support autorotation, what will happen when rotated?

- Rotations are treated as a change in the size of the view controller's view and are therefore reported using the `viewWillTransition(to:with:)` method.
- When the interface orientation changes, UIKit calls this method on the window's root view controller. That view controller then notifies its child view controllers, propagating the message throughout the view controller hierarchy.
- The `frame` of all `subviews` in your Controller's View will be adjusted.
- The adjustment is based on their "struts and springs".
- You set "struts and springs" in Interface Builder.
- When a view's `bounds` changes because its `frame` is altered, does `drawRect:` get called again? No.

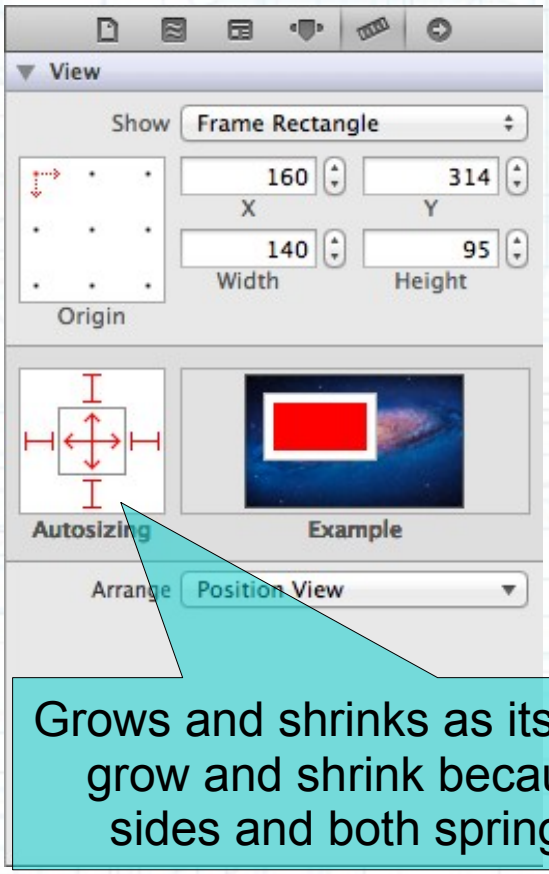
Struts and Springs

Set a view's struts and springs in Size Inspector in Xcode



Struts and Springs

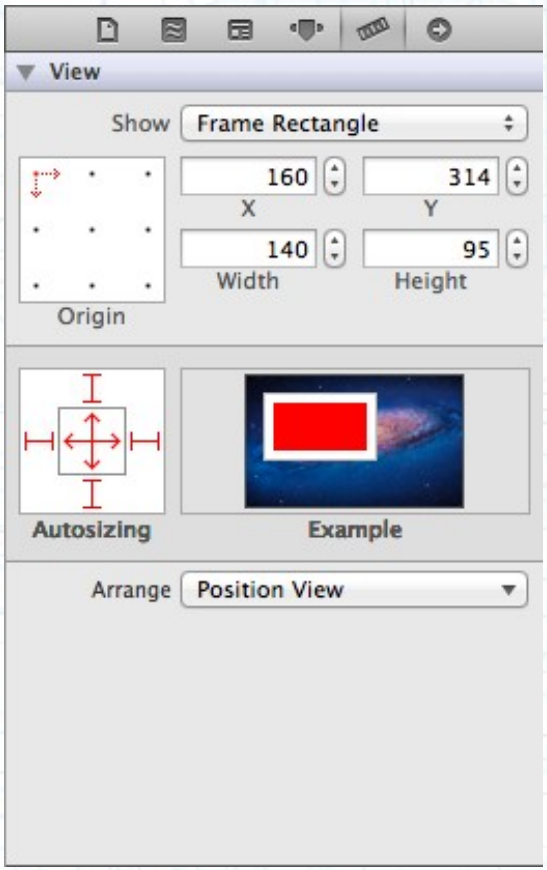
Set a view's struts and springs in Size Inspector in Xcode

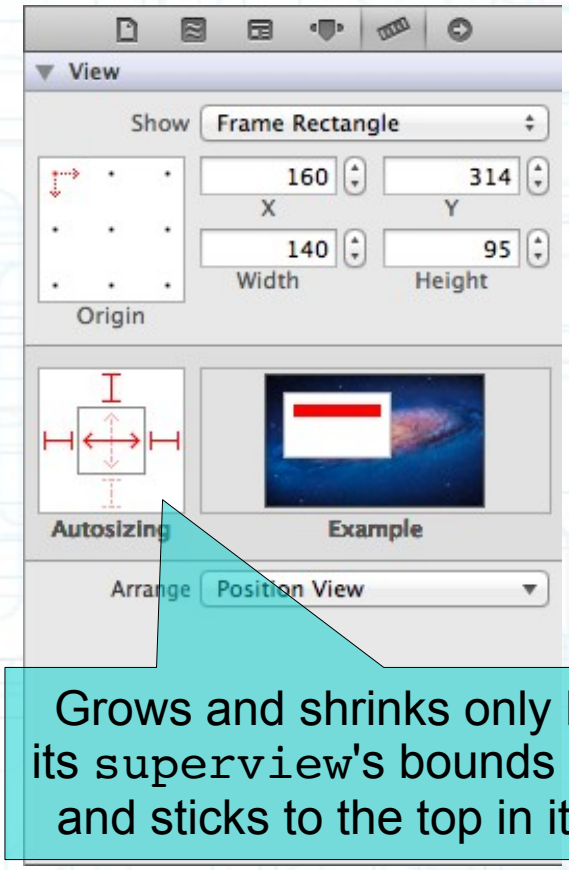


Grows and shrinks as its `superview`'s bounds grow and shrink because struts fixed to all sides and both springs allow expansion.

Struts and Springs

Set a view's struts and springs in Size Inspector in Xcode

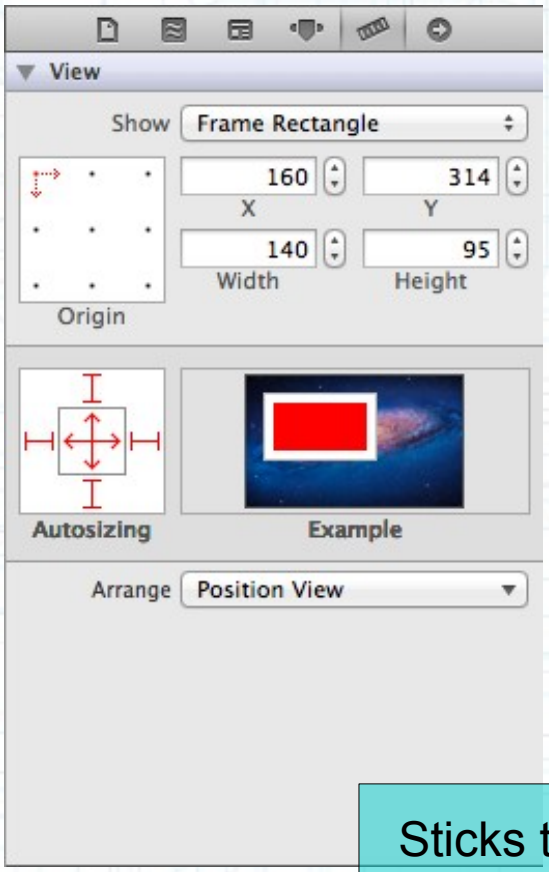


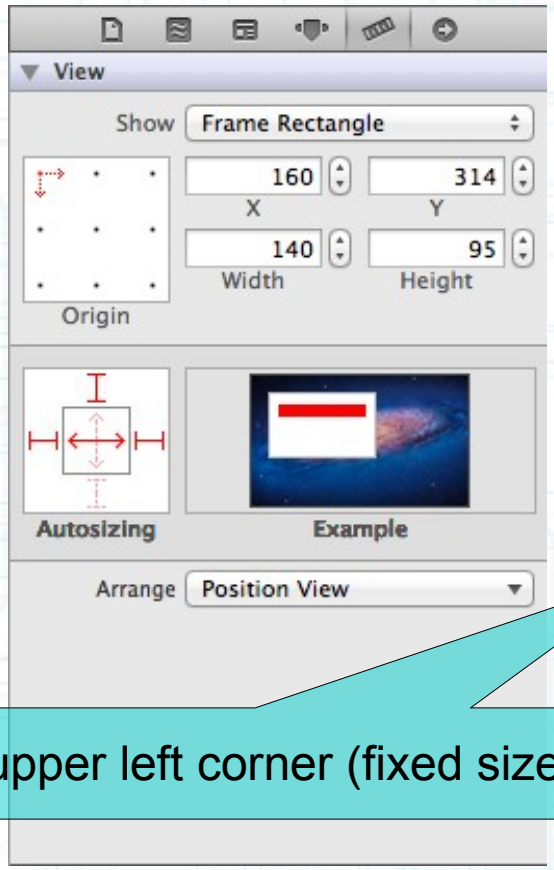


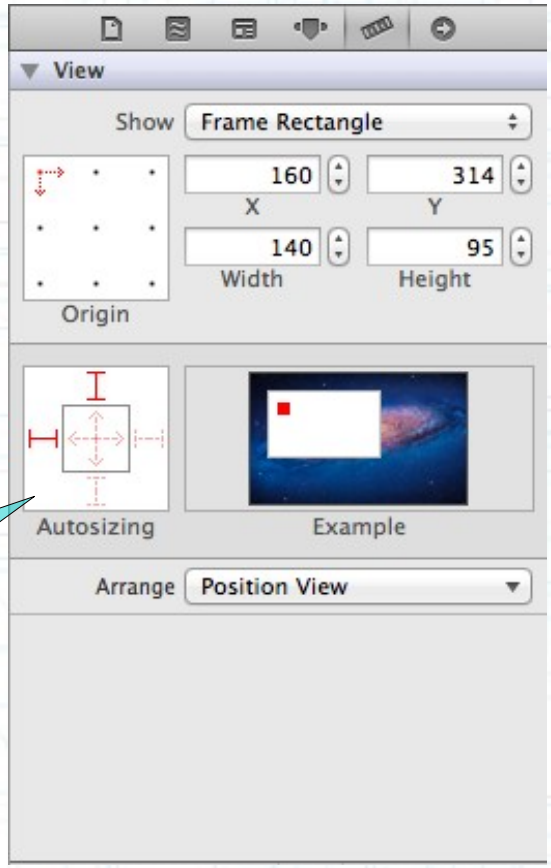
Grows and shrinks only horizontally as its `superview`'s bounds grow and shrink and sticks to the top in its `superview`.

Struts and Springs

Set a view's struts and springs in Size Inspector in Xcode







Sticks to upper left corner (fixed size).

Redraw on bounds change

- By default, there is no redraw when your `UIView`'s bounds change.
- Instead, the "bits" of your view will be stretched or squished or moved.
- Often this is not what you want. Luckily, there is a `UIView` property to control this:

```
var contentMode: UIViewContentMode { get set }
```

```
UIViewContentMode.{left, right, top, bottom,
 bottomLeft, bottomRight, topLeft, topRight, ...}
```

- The above is not springs and struts! This is after springs and struts have been applied!
- These content modes move the bits of your drawing to that location.

Redraw on bounds change

- For bit stretching or shrinking set `contentMode` to:

```
UIViewContentMode.scale{ToFill, AspectFill, AspectFit}
```

- But many times you want just to call `drawRect:`. For this use:

```
UIViewContentMode.redraw
```

- Default is `UIViewContentMode.scaleToFill`.

Initializing a `UIView`

- Yes, you can override `init(frame:)`. Use previously explained syntax:

```
super.init(frame: frame)
```

- But you will also want to set up stuff in `awakeFromNib`.

  This is because `init(frame:)` is **NOT** called for a `UIView` coming out of a storyboard!

  But `awakeFromNib` is. It's called "`awakeFromNib`" for historical reasons.

- Typical code:

```
func setup() { /* set up your custom view here */ }

override func awakeFromNib() { self.setup() }

override init(frame: CGRect) {
    super.init(frame: frame)
    self.setup()
}
```

UIGestureRecognizer

We've seen how to draw in our `UIView`, but how do we get touches?

- We can get notified of the raw touch events (touch down, moved, up).
- Or we can react to certain, predefined "gestures". This latter is the preferable way to go.

Gestures are recognized by the class `UIGestureRecognizer`

- This class is "abstract". We only actually use "concrete subclasses" of it.
- There are two sides to using a gesture recognizer:

1. Adding a gesture recognizer to a `UIView` to ask it to recognize that gesture.

2. Providing the implementation of a method to "handle" that gesture when it happens.

UIGestureRecognizer

Usually #1 is done by a Controller

- Though occasionally a `UIView` will do it to itself if it just doesn't make sense without that gesture.

Usually #2 is provided by the `UIView` itself

- But it would not be unreasonable for the Controller to do it.
- Or for the Controller to decide it wants to handle a gesture differently than the view does.

UIGestureRecognizer

Adding a gesture recognizer to a `UIView` from a Controller

```
func setPannable(view: UIView)
{
    let panGR = UIPanGestureRecognizer(target: view,
        action: #selector(CustomView.pan(recognizer:)))

}
```

This is a concrete subclass of `UIGestureRecognizer` that recognizes “panning” (moving something around with your finger). There are, of course, other concrete subclasses (for swipe, pinch, tap, etc).

UIGestureRecognizer

Adding a gesture recognizer to a `UIView` from a Controller

```
func setPannable(view: UIView)
{
    let panGR = UIPanGestureRecognizer(target: view,
        action: #selector(CustomView.pan(recognizer:)))

}
```

Note that we are specifying the view itself as the target to handle a pan gesture when it is recognized. Thus the view will be both the recognizer and the handler of the gesture. The `UIView` does not have to handle the gesture. It could be, for example, the Controller that handles it. The View would generally handle gestures to modify how the View is drawn. The Controller would have to handle gestures that modify the Model.

UIGestureRecognizer

Adding a gesture recognizer to a `UIView` from a Controller

```
func setPannable(view: UIView)
{
    let panGR = UIPanGestureRecognizer(target: view,
        action: #selector(CustomView.pan(recognizer:)))

}
```

This is the action method that will be sent to the target (the pannable `view`) during the handling of the recognition of this gesture. This version of the action message takes one argument (which is the `UIGestureRecognizer` that sends the action), but there is another version that takes no arguments if you would prefer it. We'll look at the implementation of this method in a moment.

UIGestureRecognizer

Adding a gesture recognizer to a `UIView` from a Controller

```
func setPannable(view: UIView)
{
    let panGR = UIPanGestureRecognizer(target: view,
        action: #selector(CustomView.pan(recognizer:)))
    view.addGestureRecognizer(panGR)
}
```

If we don't do this, then even though the `view` implements `pan(recognizer:)`, it would never get called because we would have never added this gesture recognizer to the view's list of gestures that it recognizes. Think of this as "turning the handling of this gesture on."

UIGestureRecognizer

Adding a gesture recognizer to a `UIView` from a Controller

```
func setPannable(view: UIView)
{
    let panGR = UIPanGestureRecognizer(target: view,
        action: #selector(CustomView.pan(recognizer:)))
    view.addGestureRecognizer(panGR)
}
```

- Only `UIView` instances can recognize a gesture (because `UIView`s handle all touch input).
- But any object can tell a `UIView` to recognize a gesture (by adding a recognizer to the `UIView`).
- And any object can handle the recognition of a gesture (by being the target of the gesture's action).

UIGestureRecognizer

- How do we implement the target of a gesture recognizer?

  Each concrete class provides some methods to help you do that.
- For example, `UIPanGestureRecognizer` provides 3 methods:

```
func velocity(in view: UIView?) -> CGPoint
func translation(in view: UIView?) -> CGPoint
func setTranslation(_ translation: CGPoint,
                    in view: UIView?)
```

UIGestureRecognizer

- The base class `UIGestureRecognizer` provides this property:

```
var state: UIGestureRecognizerState { get set }
```

- Gesture Recognizers sit around in the state `.possible` until they start to be recognized.
- Then they either go to `.recognized` (for discrete gestures like a tap).
- Or they go to `.began` (for continuous gestures like a pan).
- At any time, the state can change to `.failed` (so watch out for that).
- If the gesture is continuous, it will move on to the `.changed` and eventually the `.ended` state.
- Continuous gestures can also go to `.cancelled` state (if the recognizer realizes it's not this gesture after all).

UIGestureRecognizer

So, given these methods, what would `pan:` look like?

```
func pan(recognizer: UIPanGestureRecognizer)
{




}
```

UIGestureRecognizer

So, given these methods, what would `pan:` look like?

```
func pan(recognizer: UIPanGestureRecognizer)
{
    if recognizer.state == .changed ||
       recognizer.state == .ended
    {









    }
}
```

We are going to update our view every time the touch moves (and when the touch ends). This is “smooth panning”.

UIGestureRecognizer

So, given these methods, what would `pan:` look like?

```
func pan(recognizer: UIPanGestureRecognizer)
{
    if recognizer.state == .changed ||
       recognizer.state == .ended
    {
        let translation = recognizer.translation(in: self)

    }
}
```

This is the cumulative distance this gesture has moved.

UIGestureRecognizer

So, given these methods, what would `pan:` look like?

```
func pan(recognizer: UIPanGestureRecognizer)
{
    if recognizer.state == .changed ||
       recognizer.state == .ended
    {
       let translation = recognizer.translation(in: self)

       /* Move something in myself (I'm a UIView)
        * by translation.x and translation.y.
        * For example, if I were a graph and my origin
        * was set by a property called origin. */
       self.origin.x += translation.x
       self.origin.y += translation.y


    }
}
```

UIGestureRecognizer

So, given these methods, what would `pan:` look like?

```
func pan(recognizer: UIPanGestureRecognizer)
{
    if recognizer.state == .changed ||
       recognizer.state == .ended
    {
        let translation = recognizer.translation(in: self)

        /* Move something in myself (I'm a UIView)
         * by translation.x and translation.y.
         * For example, if I were a graph and my origin
         * was set by a property called origin. */
        self.origin.x += translation.x
        self.origin.y += translation.y
        recognizer.setTranslation(CGPoint(x: 0, y: 0),
                                  in: self)
    }
}
```

Here we are resetting the cumulative distance to zero. Now each time this is called, we'll get the "incremental" movement of the gesture (which is what we want). If we wanted the "cumulative" movement of the gesture, we would not include this line of code.

Other Concrete Gestures

UIPinchGestureRecognizer

```
var scale: CGFloat { get set }
//note that this is not readonly (can reset each movement)
var velocity: CGFloat { get }
//note that this is readonly; scale factor per second
```

UIRotationGestureRecognizer

```
var rotation: CGFloat { get set }
//note that this is not readonly; in radians
var velocity: CGFloat { get }
//note that this is readonly; in radians per second
```

Other Concrete Gestures

`UISwipeGestureRecognizer`

- This one you "set up" (with the following) to find certain swipe types, then look for `Recognized` state:

```
var direction: UISwipeGestureRecognizerDirection {get set}
// what direction swipes you want
var numberOfTouchesRequired: Int { get set }
// two finger swipes? or just one finger? or more?
```

`UITapGestureRecognizer`

- Set up (with the following) then look for `Recognized` state:

```
var numberOfTapsRequired: Int { get set }
// single tap or double tap or triple tap, etc.
var numberOfTouchesRequired: Int { get set }
// e.g., require two finger tap?
```

Other Concrete Gestures

`UIScreenEdgePanGestureRecognizer`

- Inherits from `UIPanGestureRecognizer`:

```
var edges: UIRectEdge { get set }
// the acceptable starting edges for the gesture
```

`UILongPressGestureRecognizer`

```
var minimumPressDuration: CFTimeInterval { get set }
// time interval in seconds; default is 0.5 seconds
var numberOfTouchesRequired: Int { get set }
// e.g., require two finger long press?
var numberOfTapsRequired: Int { get set }
var allowableMovement: CGFloat { get set }
// maximum movement of fingers before the gesture fails
```

Next Time

View Controllers and Storyboarding:

- MVCs Working Together
- Segues
- Navigation Controllers
- View Controllers
- Tab Bar Controller