Views

- A view (i.e. `UIView` subclass) represents a rectangular area
  Defines a coordinate space
- Draws and handles events in that rectangle
- Hierarchical
  A view has only one superview `- (UIView *)superview`
  But can have many (or zero) subviews `- (NSArray *)subviews`
  Subview order (in `subviews` array) matters: those later in the array are on top of those earlier
- `UIWindow`
  The `UIView` at the top of the view hierarchy
  Only have one `UIWindow` (generally) in an iOS application
  It's all about views, not windows

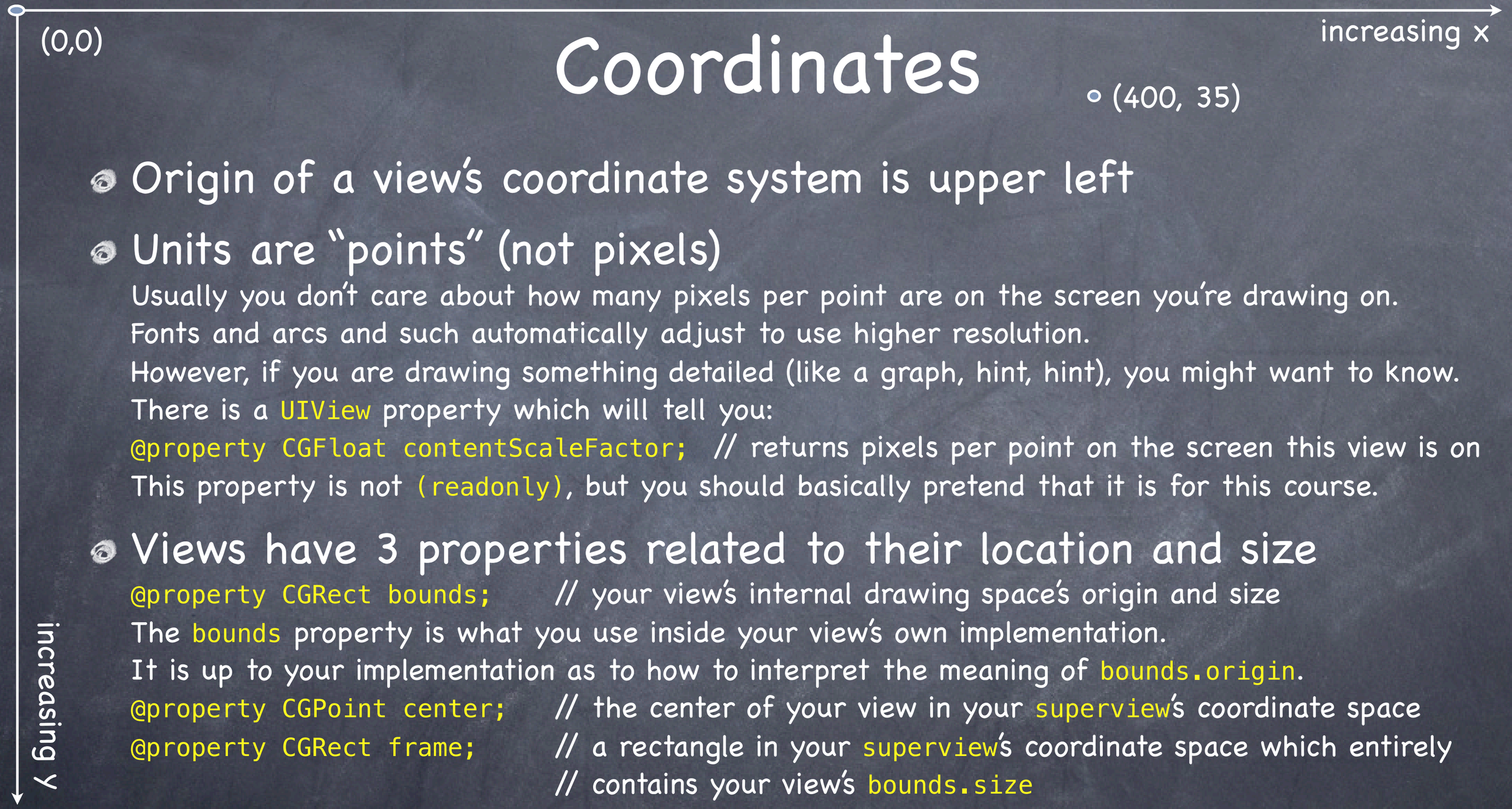
Views

- The hierarchy is most often constructed in Xcode graphically
  Even custom views are added to the view hierarchy using Xcode (more on this later).
- But it can be done in code as well

```
- (void)addSubview:(UIView *)aView;
- (void)removeFromSuperview;
```

View Coordinates

- CGFloat
  Just a floating point number, but we always use it for graphics.
- CGPoint
  C struct with two CGFloats in it: x and y.
```
CGPoint p = CGPointMake(34.5, 22.0);
p.x += 20;  // move right by 20 points
```
- CGSize
  C struct with two CGFloats in it: width and height.
```
CGSize s = CGSizeMake(100.0, 200.0);
s.height += 50;  // make the size 50 points taller
```
- CGRect
  C struct with a CGPoint origin and a CGSize size.
```
CGRect aRect = CGRectMake(45.0, 75.5, 300, 500);
aRect.size.height += 45;  // make the rectangle 45 points taller
aRect.origin.x += 30;     // move the rectangle to the right 30 points
```



Coordinates

- Origin of a view's coordinate system is upper left
- Units are "points" (not pixels)

  Usually you don't care about how many pixels per point are on the screen you're drawing on.
  Fonts and arcs and such automatically adjust to use higher resolution.
  However, if you are drawing something detailed (like a graph, hint, hint), you might want to know.
  There is a `UIView` property which will tell you:

```
@property CGFloat contentScaleFactor;  // returns pixels per point on the screen this view is on
```

  This property is not `(readonly)`, but you should basically pretend that it is for this course.

- Views have 3 properties related to their location and size

```
@property CGRect bounds;     // your view's internal drawing space's origin and size
```

  The `bounds` property is what you use inside your view's own implementation.
  It is up to your implementation as to how to interpret the meaning of `bounds.origin`.

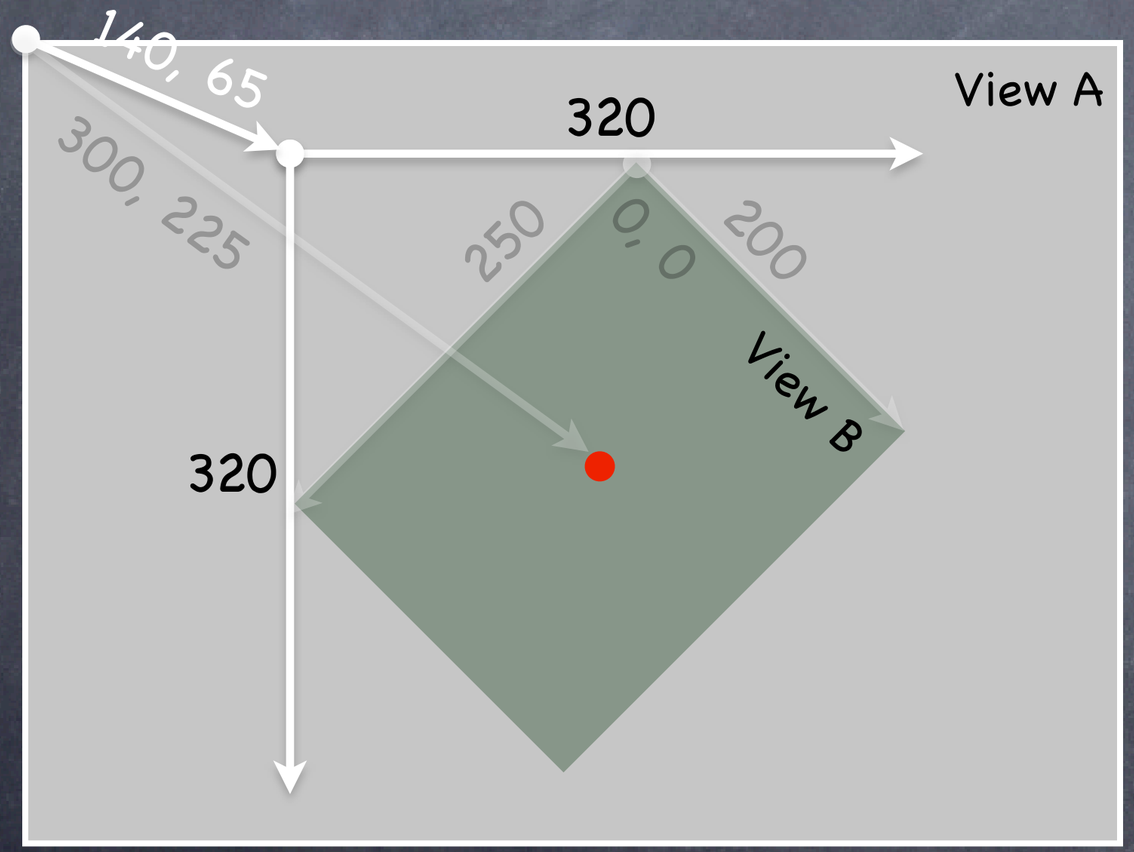
```
@property CGPoint center;    // the center of your view in your superview's coordinate space
@property CGRect frame;      // a rectangle in your superview's coordinate space which entirely
                             // contains your view's bounds.size
```

Coordinates

- Use `frame` and `center` to position the view in the hierarchy

  These are used by `superviews`, never inside your `UIView` subclass's implementation.

  You might think `frame.size` is always equal to `bounds.size`, but you'd be wrong …



Because views can be rotated
(and scaled and translated too).

View B's `bounds` = `((0,0),(200,250))`

View B's `frame` = `((140,65),(320,320))`

View B's `center` = `(300,225)`

View B's middle in its own coordinate space is
`(bound.size.width/2+bounds.origin.x,`
`bounds.size.height/2+bounds.origin.y)`
which is `(100,125)` in this case.

Views are rarely rotated, but don't
misuse `frame` or `center` by assuming that.

Creating Views

- Most often you create views in Xcode

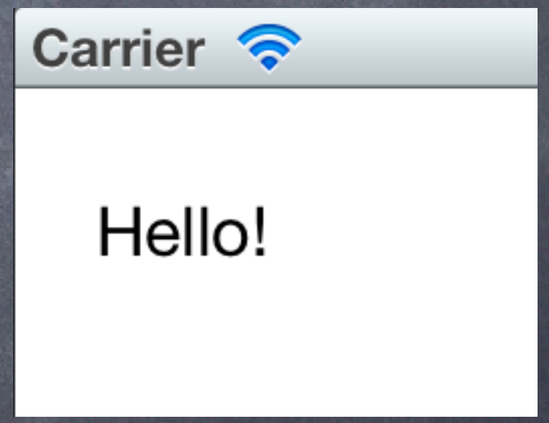
  Of course, Xcode's palette knows nothing about a custom view class you might create.
  In that case, you drag out a generic `UIView` from the palette and use the Inspector
  to change the class of the `UIView` to your custom class (demo of this later).

- How do you create a `UIView` in code (i.e. not in Xcode)?

  Just use `alloc` and `initWithFrame:` (`UIView`'s designated initializer).

- Example

```
CGRect labelRect = CGRectMake(20, 20, 50, 30);
UILabel *label = [[UILabel alloc] initWithFrame:labelRect];
label.text = @"Hello!";
[self.view addSubview:label];  // we'll talk about self.view later
                               // (it is a Controller's top-level view)
```



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'll talk about handling touch events later. For now we're focussing on drawing.
- Drawing is easy … create a `UIView` subclass & override 1 method
  `- (void)drawRect:(CGRect)aRect;`
  You can optimize by not drawing outside of `aRect` if you want (but not required).
- NEVER call `drawRect:`!! EVER! Or else!
  Instead, let iOS know that your view's visual is out of date with one of these `UIView` methods:
  `- (void)setNeedsDisplay;`
  `- (void)setNeedsDisplayInRect:(CGRect)aRect;`
  It will then set everything up and call `drawRect:` for you at an appropriate time
  Obviously, the second version will call your `drawRect:` with only rectangles that need updates

Custom Views

- So how do I implement my `drawRect:`?
  Use the Core Graphics framework
- The API is C (not object-oriented)
- Concepts
  Get a context to draw into (iOS will prepare one each time your `drawRect:` is called)
  Create paths (out of lines, arcs, etc.)
  Set colors, fonts, textures, linewidths, linecaps, etc.
  Stroke or fill the above-created paths

Context

- The context determines where your drawing goes
  - Screen (the only one we're 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 `drawRect:`
  - A new one is set up for you each time `drawRect:` is called
  - So never cache the current graphics context in `drawRect:` to use later!
- How to get this magic context?
  - Call the following C function inside your `drawRect:` method to get the current graphics context …

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

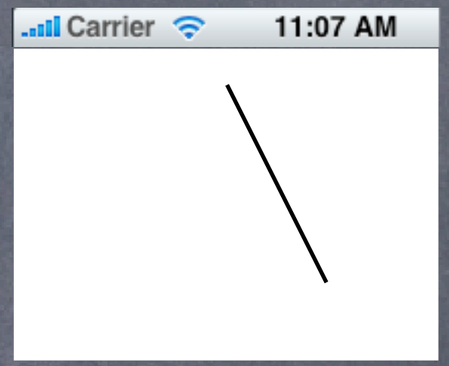
Define a Path

- Begin the path

```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path

```
CGContextMoveToPoint(context, 75, 10);
CGContextAddLineToPoint(context, 160, 150);
```



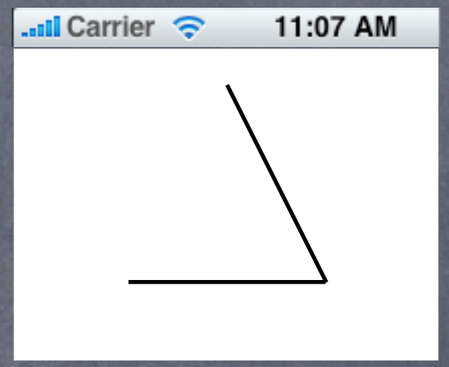
Define a Path

- Begin the path

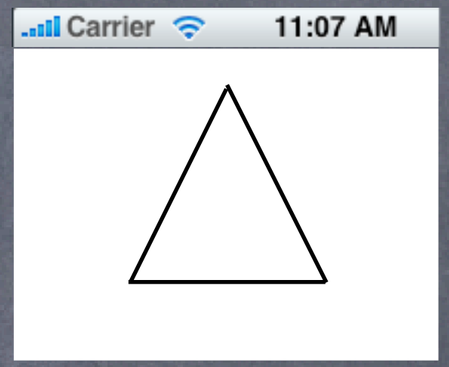
```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path

```
CGContextMoveToPoint(context, 75, 10);
CGContextAddLineToPoint(context, 160, 150);
CGContextAddLineToPoint(context, 10, 150);
```



Define a Path



- Begin the path

```
CGContextBeginPath(context);
```

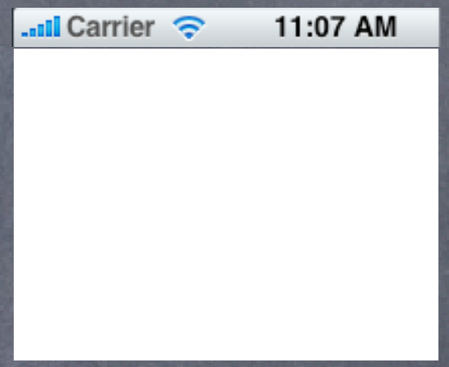
- Move around, add lines or arcs to the path

```
CGContextMoveToPoint(context, 75, 10);
CGContextAddLineToPoint(context, 160, 150);
CGContextAddLineToPoint(context, 10, 150);
```

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

```
CGContextClosePath(context);    // not strictly required
```

Define a Path



- Begin the path

```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path

```
CGContextMoveToPoint(context, 75, 10);
CGContextAddLineToPoint(context, 160, 150);
CGContextAddLineToPoint(context, 10, 150);
```

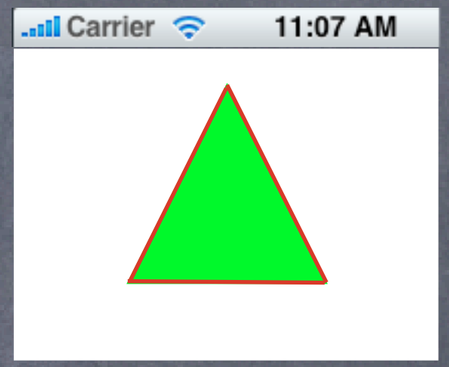
- Close the path (connects the last point back to the first)

```
CGContextClosePath(context);    // 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

```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path

```
CGContextMoveToPoint(context, 75, 10);
CGContextAddLineToPoint(context, 160, 150);
CGContextAddLineToPoint(context, 10, 150);
```

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

```
CGContextClosePath(context);    // 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 greenColor] setFill];    // object-oriented convenience method (more in a moment)
[[UIColor redColor] setStroke];
CGContextDrawPath(context, kCGPathFillStroke);  // kCGPathFillStroke is a constant
```

Define a Path

- It is also possible to save a path and reuse it
  Similar functions to the previous slide, but starting with `CGPath` instead of `CGContext`
  We won't be covering those, but you can certainly feel free to look them up in the documentation

Graphics State

- UIColor class for setting colors

```
UIColor *red = [UIColor redColor];    // class method, returns autoreleased instance
UIColor *custom = [[UIColor alloc] initWithRed:(CGFloat)red     // 0.0 to 1.0
                                          blue:(CGFloat)blue
                                         green:(CGFloat)green
                                         alpha:(CGFloat)alpha];  // 0.0 to 1.0 (opaque)
[red setFill];   // fill color set in current graphics context (stroke color not set)
[custom set];    // sets both stroke and fill color to custom (would override [red setFill])
```

- Drawing with transparency in UIView

Note the alpha above. 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 @property BOOL opaque to NO 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 @property CGFloat alpha can make the entire view partially transparent.

View Transparency

- What happens when views overlap?
  As mentioned before, `subviews` list order determine's 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 (performance-wise)
- Also, you can hide a view completely by setting `hidden` property

```
@property (nonatomic) BOOL hidden;
myView.hidden = YES;        // view will not be on screen and will not handle events
```

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

Graphics State

- Some other graphics state set with C functions, e.g. ...

```
CGContextSetLineWidth(context, 1.0);  // line width in points (not pixels)
CGContextSetFillPattern(context, (CGPatternRef)pattern, (CGFloat[])components);
```

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 push and pop context functions.

```
- (void)drawGreenCircle:(CGContextRef)ctxt {
    UIGraphicsPushContext(ctxt);
    [[UIColor greenColor] setFill];
    // draw my circle
    UIGraphicsPopContext();
}
- (void)drawRect:(CGRect)aRect {
    CGContextRef context = UIGraphicsGetCurrentContext();
    [[UIColor redColor] setFill];
    // do some stuff
    [self drawGreenCircle:context];
    // do more stuff and expect fill color to be red
}
```

Drawing Text

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

```
UIFont *myFont = [UIFont systemFontOfSize:12.0];
UIFont *theFont = [UIFont fontWithName:@“Helvetica” size:36.0];
NSArray *availableFonts = [UIFont familyNames];
```

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

```
NSString *text = …;
[text drawAtPoint:(CGPoint)p withFont:theFont];  // NSString instance method
```

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

```
CGSize textSize = [text sizeWithFont:myFont];    // NSString instance method
```

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 a mechanism called categories.
Categories are an Objective-C way to add methods to an existing class without subclassing.
We'll cover how (and when) to use this a bit later in this course.

Drawing Images

- Use `UIImageView` to draw images, but if you feel you must …
  We'll cover `UIImageView` later in the course.
- Create a `UIImage` object from a file in your `Resources` folder

```
UIImage *image = [UIImage imageNamed:@"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 …)

```
UIImage *image = [[UIImage alloc] initWithContentsOfFile:(NSString *)fullPath];
UIImage *image = [[UIImage alloc] initWithData:(NSData *)imageData];
```

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

```
UIGraphicsBeginImageContext(CGSize);
// draw with CGContext functions
UIImage *myImage = UIGraphicsGetImageFromCurrentContext();
UIGraphicsEndImageContext();
```

Drawing Images

- Now blast the UIImage's bits into the current graphics context

```
UIImage *image = …;
[image drawAtPoint:(CGPoint)p];                 // p is upper left corner of the image
[image drawInRect:(CGRect)r];                   // scales the image to fit in r
[image drawAsPatternInRect:(CGRect)patRect];    // tiles the image into patRect
```

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

```
NSData *jpgData = UIImageJPEGRepresentation((UIImage *)myImage, (CGFloat)quality);
NSData *pngData = UIImagePNGRepresentation((UIImage *)myImage);
```

Autorotation

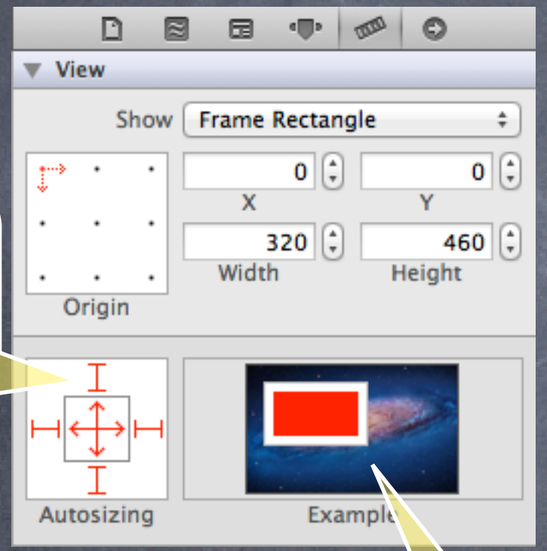
- What goes on in your Controller when the device is rotated?
  You can control whether the user interface rotates along with it
- Implement the following method in your Controller

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation
{
    return UIInterfaceOrientationIsPortrait(orientation); // only support portrait
    return YES; // support all orientations
    return (orientation != UIInterfaceOrientationPortraitUpsideDown); // anything but
}
```

- If you support an orientation, what will happen when rotated?
  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 Xcode.
  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

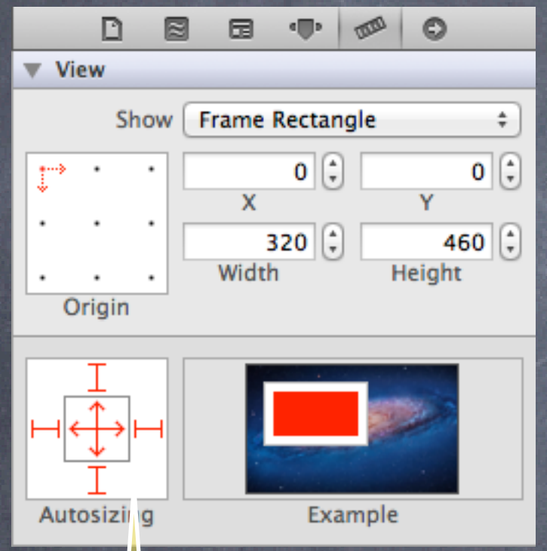


Click to toggle on and off.

Preview

Struts and Springs

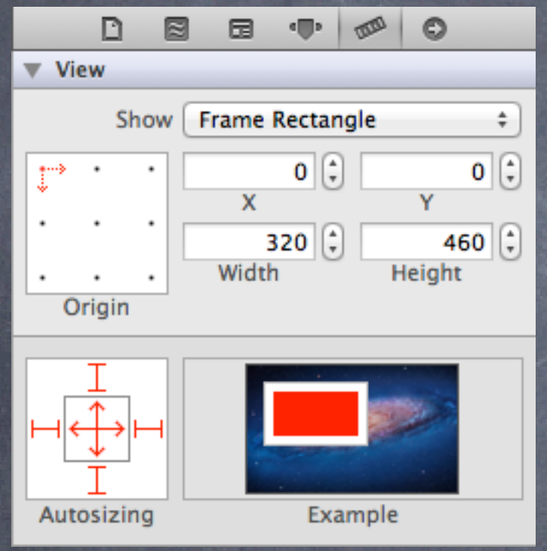
- Set a view's struts and springs in size inspector in Xcode



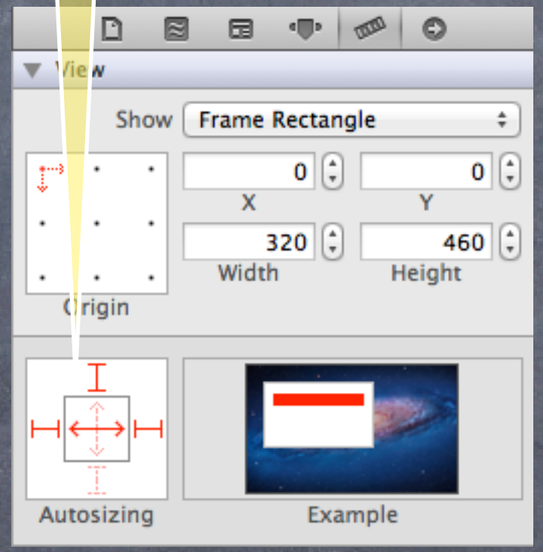
Grows and shrinks as its `superview`'s `bounds` grow and shrink because struts fixed to all sides and both springs allow expansion.

Struts and Springs

- Set a view's struts and springs in size inspector in Xcode

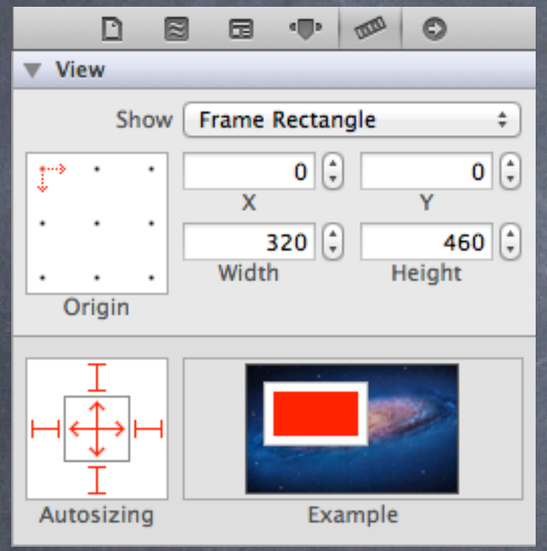


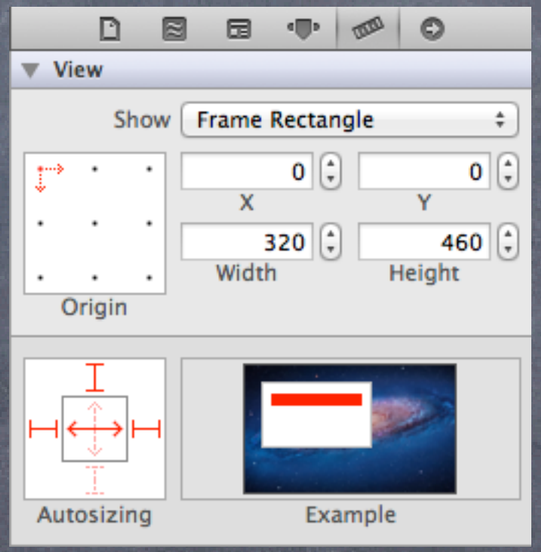
Grows and shrinks only horizontally as its superview's bounds grow and shrink and sticks to the top in its superview.



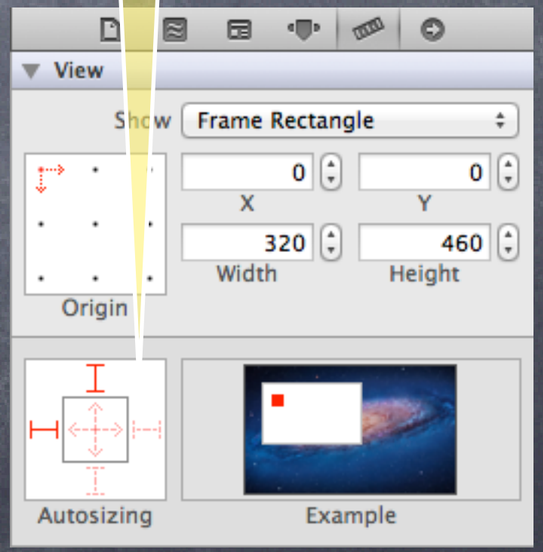
Struts and Springs

- Set a view's struts and springs in size inspector in Xcode





Sticks to upper left corner (fixed size).

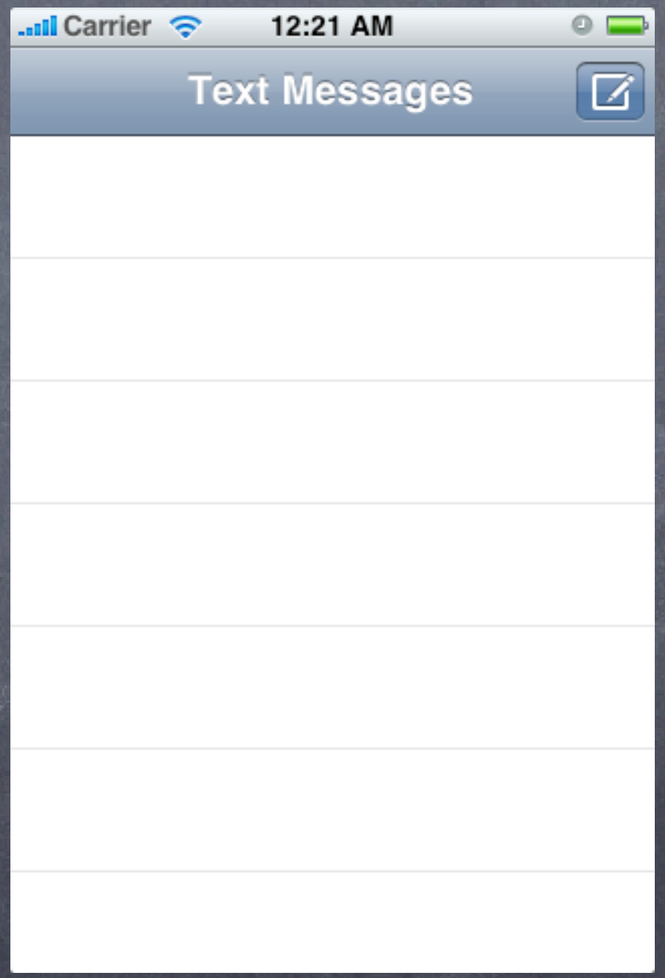


Redraw on bounds change?

- By default, when your UIView's bounds change, no redraw
  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!
  `@property (nonatomic) UIViewContentMode contentMode;`
  `UIViewContentMode{Left,Right,Top,Right,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.
  `UIViewContentModeScale{ToFill,AspectFill,AspectFit}` // bit stretching/shrinking
  `UIViewContentModeRedraw` // call `drawRect:` (this is many times what you want)
  Default is `UIViewContentModeScaleToFill`
- You can control which of your bits get stretched
  `@property (nonatomic) CGRect contentStretch;`
  Rectangle of ((0, 0), (1, 1)) stretches all the bits.
  Something smaller stretches only a portion of the bits. If width/height is 0, duplicates a pixel.

Modal View Controllers

- Making a view controller's view appear temporarily
  And blocking all other "navigation" in the application until the user has dealt with this view.



Modal View Controllers

- This is what we mean by segueing to a View Controller "modally"

  "Modally" means "in a mode": nothing else can go on while that other View Controller is up.

  There are different transition styles and presentation styles (more on this in a moment).

  Just ctrl-drag from a button to the modal View Controller. Inspect segue to set style.

- Can be done from code as well (less common)

  Example. Putting up a modal view that asks the user to find an address.

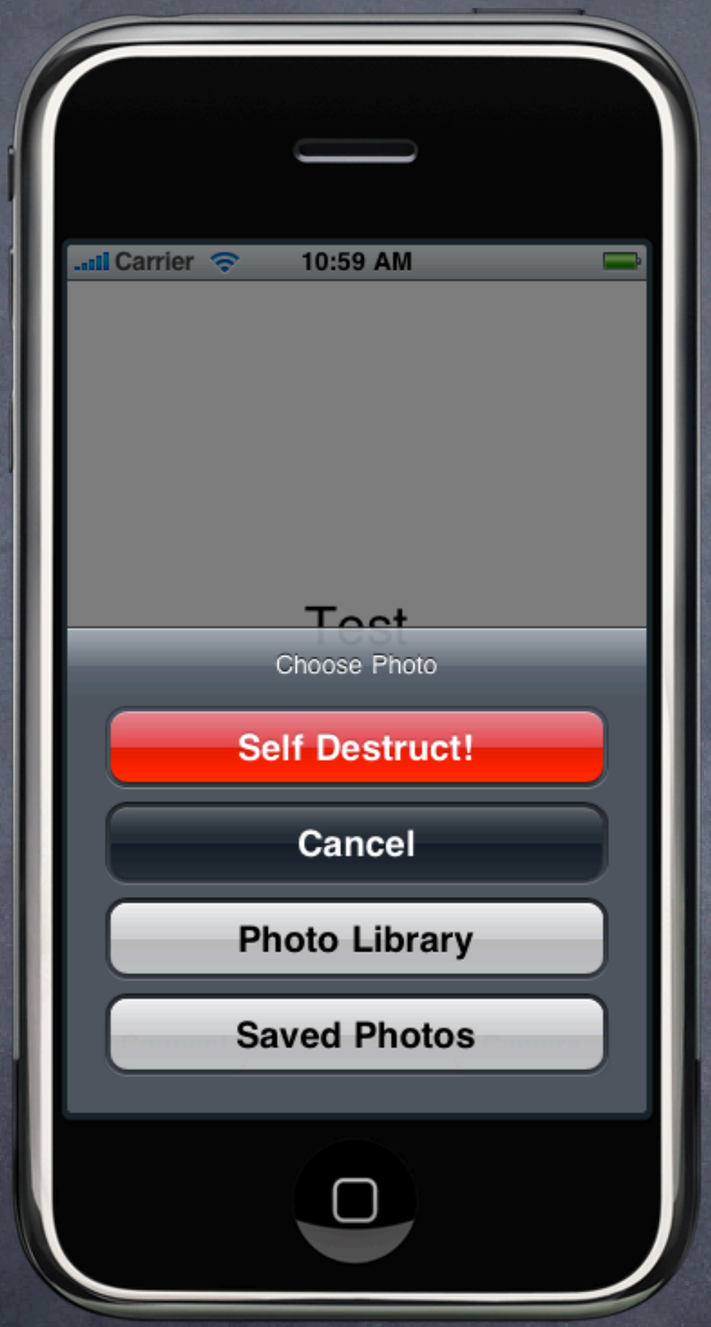
```
- (IBAction)lookupAddress
{
    AddressLookupViewController *alvc =
        [self.storyboard instantiateViewControllerWithIdentifier:@"AddressLookup"];
    [self presentModalViewController:alvc animated:YES completion:^{
        // alvc is now on screen; often we do nothing here
    }];
}
```

  This method will fill the entire screen with `alvc`'s `view` and immediately return after the block.

  The user will then not be able to do anything except interact with `alvc`'s view.

Alerts

- Two kinds of "pop up and ask the user something" mechanisms
  - Action Sheets
  - Alerts
- Action Sheets
  - Slides up from the bottom of the screen on iPhone/iPod Touch, and in a popover on iPad.
  - Can be displayed from a tab bar, toolbar, bar button item or from a rectangular area in a view.
  - Usually asks questions that have more than two answers.
- Alerts
  - Pop up in the middle of the screen.
  - Usually ask questions with only two (or one) answers (e.g. OK/Cancel, Yes/No, etc.).
  - Very disruptive to your user-interface, so use carefully.
  - Often used for "asynchronous" problems ("connection reset" or "network fetch failed").


Carrier
10:59 AM
Test
Choose Photo
Self Destruct!
Cancel
Photo Library
Saved Photos


Carrier
10:36 AM
Flickr Fetch
Network access failed!
Ignore
Try Again
Connect
Camera