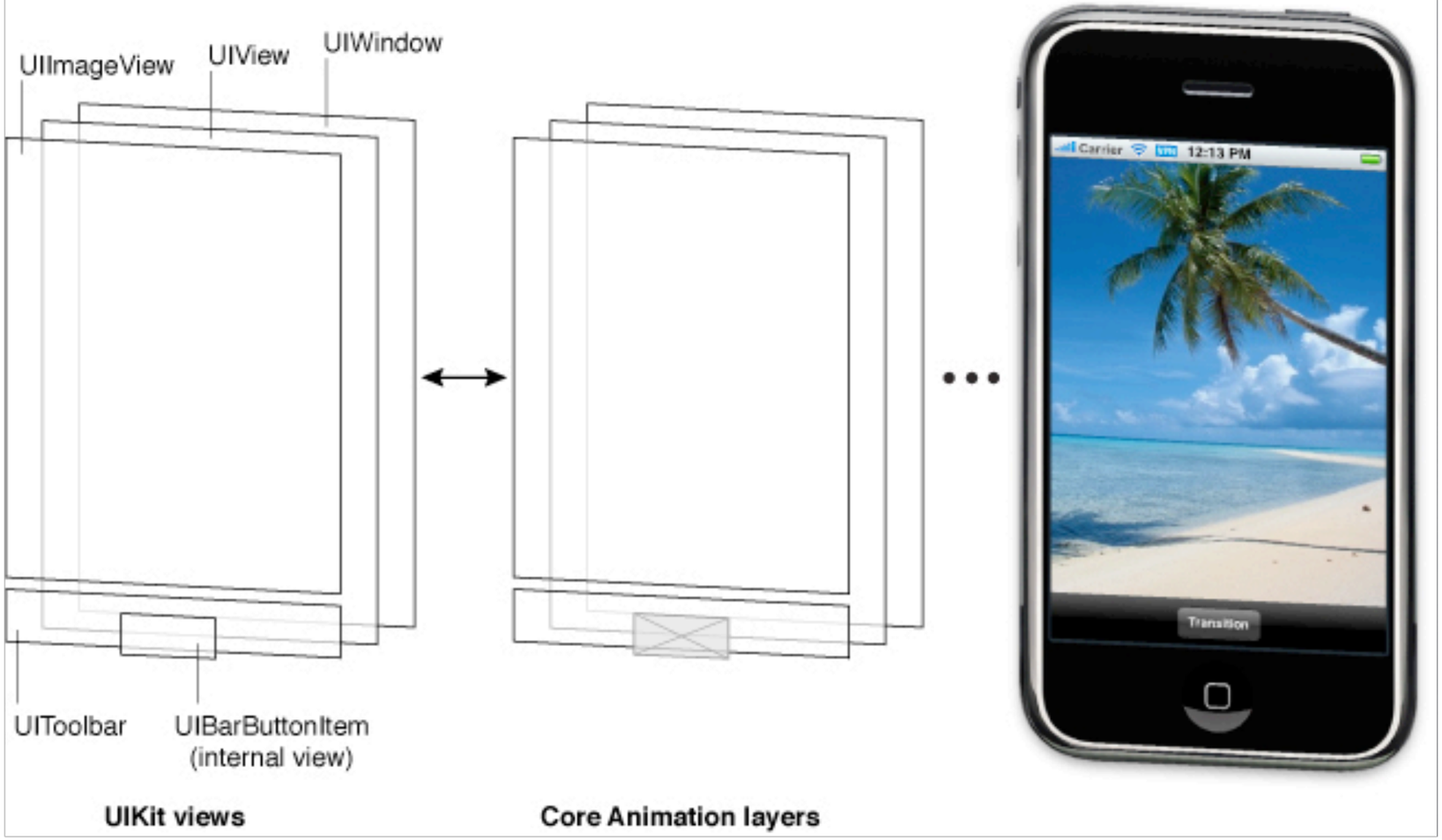
Views

Matthias Tretter, @myell0w

View Architecture

- Views and Windows present your App's User Interface and handle interactions with that interface
- UIKit provides several out-of-the-box views (e.g. Buttons, Labels, Switches, etc.)
- You can define custom views by subclassing UIView

View Architecture



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

- Hierarchy can be created in Xcode Interface Builder graphically (you can even add your custom view subclasses)
- Can be done in code as well:

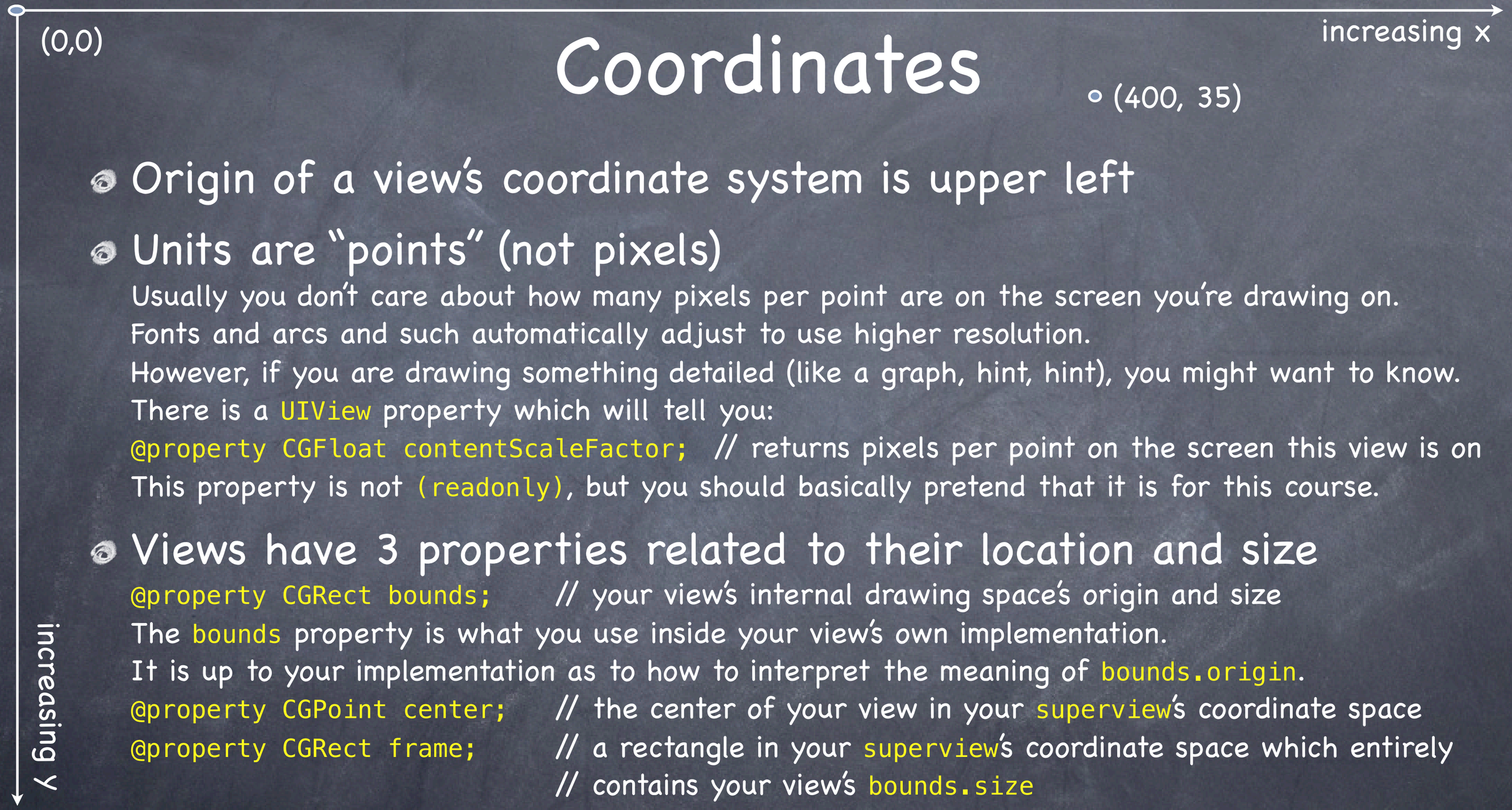
```
- (void)addSubview:(UIView *)aView
- (void)removeFromSuperview
```

- Add to Controller's view:

```
[self.view addSubview:myView]
```

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`.

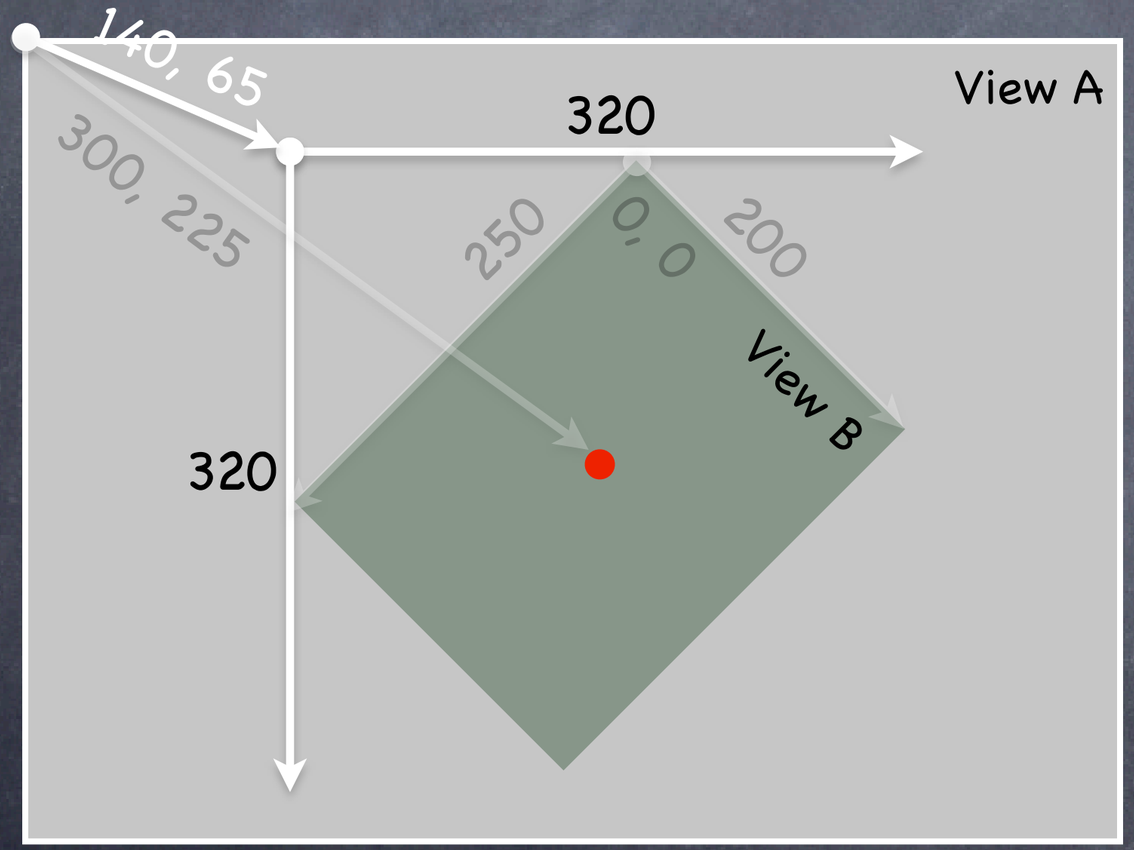
```
@property CGPoint center;    // the center of your view in your superview's coordinate space
@property CGRect frame;      // a rectangle in your superview's coordinate space which entirely
                             // contains your view's bounds.size
```

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 UIView Subclasses

- **When would I want to create my own UIView subclass?**
  custom drawing on screen
  need to handle touch events in a special way
  reusability (MVC)

- Drawing is easy .. create a UIView subclass and override
  `- (void)drawRect:(CGRect)rect`

- **NEVER** call `drawRect:` yourself!
  instead, let iOS know that the view needs an update:
  `- (void)setNeedsDisplay`
  `- (void)setNeedsDisplayInRect:(CGRect)rect`

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();
```

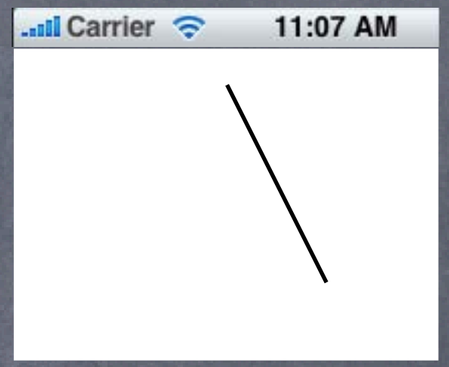
Define a Path

- Begin the path

```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path

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



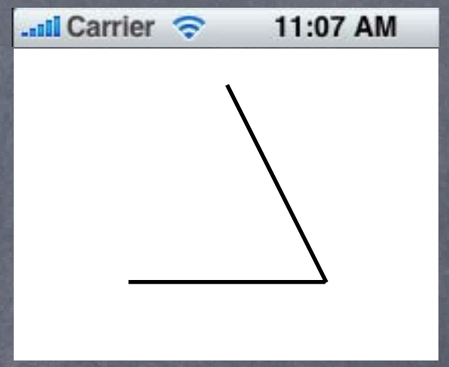
Define a Path

- Begin the path

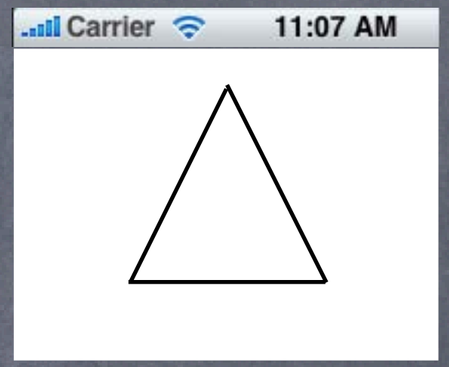
```
CGContextBeginPath(context);
```

- 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);
```

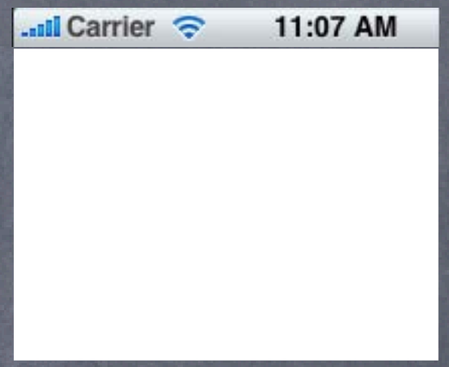
- Move around, add lines or arcs to the path

```
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);
```

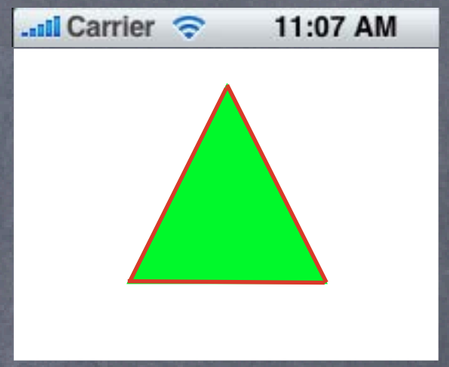
- Close the path (connects the last point back to the first)

```
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
```

Other CGContext* Functions

CGContextClip / CGContextClipToRect / CGContextClipToMask

CGContextFillRect / CGContextStrokeRect

CGContextFillEllipseInRect / CGContextStrokeEllipseInRect

CGContextRotateCTM / CGContextScaleCTM / CGContextTranslateCTM

CGContextDrawLinearGradient / CGContextDrawRadialGradient

CGContextDrawImage / CGContextDrawPDFPage

CGContextSetLineWidth

etc...

UIColor

- UIColor encapsulates a color value (color space, opacity)

```
UIColor *red = [UIColor redColor];  // pre-defined color values
UIColor *c = [UIColor colorWithRed:(CGFloat)red   // 0.0 to 1.0
                             green:(CGFloat)green
                              blue:(CGFloat)blue
                             alpha:(CGFloat)alpha;
```

- Setting fill/stroke color of current graphics context

```
[red setFill];
CGContextSetFillColorWithColor(context, red.CGColor)

[c setStroke];
CGContextSetStrokeColorWithColor(context, c.CGColor)
```

UIView transparency

- UIColor can be transparent (has alpha value)
- You can draw with transparent color in drawRect:
- UIView has a backgroundColor property
- UIView has a alpha property (make entire view transparent)
- Be sure to set @property BOOL opaque to NO for a view which is partially or fully transparent (performance optimization)

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

- 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
}
```

```
CGContextSaveGState(ctxt);
```

```
CGContextRestoreGState(ctxt);
```

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

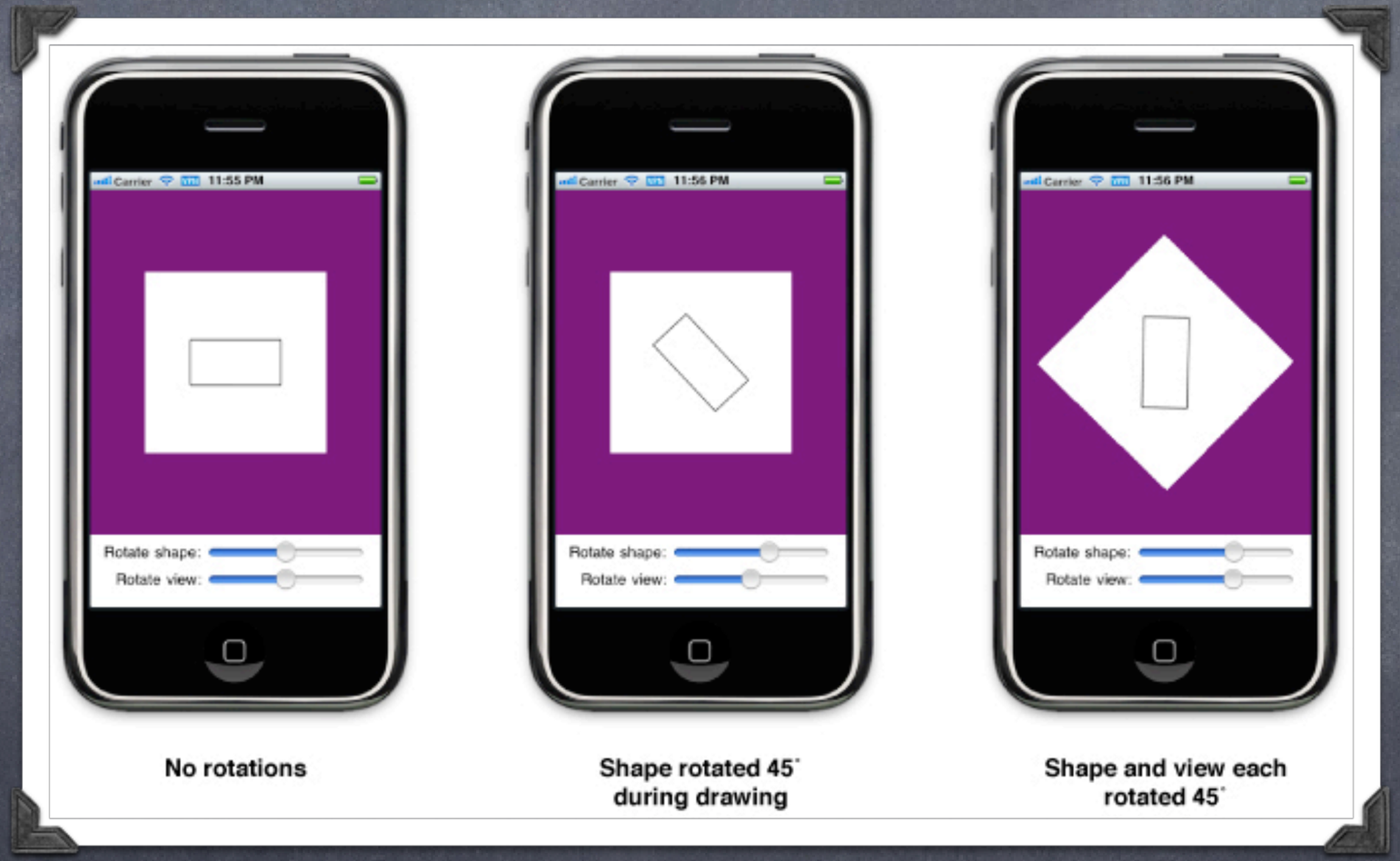
- Now blast the UIImage's bits into the current graphics context

```
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);
```

Demo App: Rotate Views



Further Literature

- View Programming Guide for iOS, Apple
  http://developer.apple.com/library/ios/#documentation/windowsviews/conceptual/viewpg_iphoneos/Introduction/Introduction.html