Stanford CS193p

Developing Applications for iPhone 4, iPod Touch, & iPad

Fall 2010

Today

- One last Objective-C topic: Protocols
  Using protocols to define/implement/use a data source and/or delegate
- Views
  `UIView` and `UIWindow` classes
  View Hierarchy
  Transparency
  Memory Management
  Coordinate Space
- Custom Views
  Creating a subclass of `UIView`
  Drawing with Core Graphics
- Demo
  Custom View / Delegation / Core Graphics

Protocols

- Similar to `@interface`, but no implementation

```
@protocol Foo
- (void)doSomething;    // implementors must implement this
@optional
- (int)getSomething;    // implementors do not need to implement this
@required
- (NSArray *)getManySomethings:(int)howMany;     // must implement
@end
```

- The above is added to a header file

  Either its own header file (e.g. `Foo.h`)

  Or the header file of the class which wants other classes to implement it

  For example, the `UIScrollViewDelegate` protocol is defined in `UIScrollView.h`

Protocols

- Classes then implement it

  They must proclaim that they implement it in their `@interface`

```
@interface MyClass : NSObject <Foo>
    ...
@end
```

- You must implement all non-`@optional` methods
- Can now declare `id` variables with added protocol requirement

```
id <Foo> obj = [[MyClass alloc] init];  // compiler will love this!
id <Foo> obj = [NSArray array];  // compiler will not like this one bit!
```

- Also can declare arguments to methods to require protocol

```
- (void)giveMeFooObject:(id <Foo>)anObjectImplementingFoo;
```

  If you call this and pass an object which does not implement `Foo` ... compiler warning!

Protocols

- Just like static typing, this is all just compiler-helping-you stuff
  It makes no difference at runtime
- Think of it as documentation for your method interfaces
- Number one use of protocols in iOS: `delegate`s and `dataSource`s
  The `delegate` or `dataSource` is always defined as an `assign @property`
  `@property (assign) id <UISomeObjectDelegate> delegate;`
  Always assumed that the object serving as delegate will outlive the object doing the delegating
  Usually true because one is a View object (e.g. `UIScrollView`) & the other is a Controller
  Controllers usually create and clean up their View objects (because they are their "minions")
  Thus the Controller will always outlive its View objects

Protocols

```
@protocol UIScrollViewDelegate
@optional
- (void)scrollViewWillBeginDragging:(UIScrollView *)scrollView
- (void)scrollViewDidEndDragging:(UIScrollView *)scrollView
                  willDecelerate:(BOOL)decelerate
...
@end

@interface UIScrollView : UIView
@property (assign) id <UIScrollViewDelegate> delegate;
@end

@interface MyViewController : UIViewController <UIScrollViewDelegate>
...
@end

MyViewController *myVC = [[MyViewController alloc] init];
UIScrollView *scrollView = ...;
scrollView.delegate = myViewController;  // compiler won't complain
```

Views

- A view (i.e. `UIView` subclass) represents a rectangular area
  Defines a coordinate space
- Draws and handles events in that rectangle
- Hierarchical
  Only one superview `- (UIView *)superview`
  Can have many (or zero) subviews `- (NSArray *)subviews`
  Subview order (in that array) matters: those later in the array are on top of those earlier

Views

- The hierarchy is most often constructed in Interface Builder
  Even custom views are added to the view hierarchy using Interface Builder
- But it can be done in code as well

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

- Managing the order of subviews (not very common)

```
- (void)insertSubview:(UIView *)aView atIndex:(int)index;
- (void)insertSubview:(UIView *)aView belowSubview:(UIView *)otherView;
- (void)insertSubview:(UIView *)aView aboveSubview:(UIView *)otherView;
```

View Transparency

- What happens when views overlap?
  - As mentioned earlier, `subviews` list order determine's who's in front
  - Lower ones can "show through" transparent views sitting on top of them though
- When you are drawing, you can draw with transparency
  - By default, drawing is fully opaque
  - We'll cover drawing in a few slides
- Also, you can hide a view completely by setting `hidden` property

```
@property 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

View Memory Management

- A superview retains its subviews
  Once you put a view into the view hierarchy, you can release your ownership if you want
- Be careful when you remove a view from the hierarchy
  If you want to keep using a view, retain ownership before you send removeFromSuperview
  Removing a view from the hierarchy immediately causes a release on it (not autorelease)
  So if there are no other owners, it will be immediately deallocated (and its subviews released)
- IBOutlets are retained
  You would think this would not be necessary since they are in a Controller's view's hierarchy.
  But the hierarchy may change (an outlet's superview might be removed, for example).
  So they are retained for safety's sake.
  But that means we must release them at some point.
  We have been failing to do this so far in our demos and homework.
  So when and how do we do this?

IBOutlet Memory Management

- When do we need to `release` our outlets?

  Obviously we need to do it in `dealloc`.

  But there's another time we need to do it … when a Controller's `view` is "unloaded".
  This "unloading" `release`s the Controller's `view` in low memory situations.
  This only happens if the `view` is offscreen at the time the memory is needed.
  In reality, this is unlikely (because there are bigger memory fish to fry like sounds and images).
  But it's the right thing to do, so we'll do it!
  The Controller can always recreate the view by reloading it from the `.xib` (for example).

- `UIViewController` calls a method on itself after view load/unload

  `- (void)viewDidLoad` is called just after the Controller's `view` has been created (& outlets are set)
  This method is an awesome place to set initial state in an outlet (if you couldn't do it in IB).

  `- (void)viewDidUnload` is called just after the view has been "unloaded"
  This is the place we can release our outlets in the case of unloading.

IBOutlet Memory Management

- We don't just call release on our outlets though
  We do it using a property-friendly syntax.
- We create a property for each IBOutlet
  Interface Builder will call the setter when it hooks up the outlet!
  `@property (retain) IBOutlet UILabel *display;`
  Doing this "documents" the retained nature of an IBOutlet in our source code.
  This property can be public (in the .h file) or private (in the .m using () magic).
- Here's how we release our outlets in viewDidUnload & dealloc

```
- (void)releaseOutlets {
    self.myOutlet = nil;
    self.myOtherOutlet = nil;
}

- (void)viewDidUnload {
    [self releaseOutlets];
}
```

```
// private method to share code between viewDidUnload and dealloc
// releases the outlet in @synthesized setter
// releases the outlet in @synthesized setter

- (void)dealloc {
    [self releaseOutlets];
    [super dealloc];
}
```

IBOutlet Memory Management

- Just to be 100% clear why `self.outlet = nil` releases outlet

```
@property (retain) IBOutlet UILabel *display;

@synthesize display;
```

Reminder that this is the code generated for a setter by `@synthesize` for a property with `retain`

```
- (void)setDisplay:(UILabel *)anObject
{
    [display release];
    display = [anObject retain];    // if anObject is nil, this message send just returns nil
}
```

Now imagine we do `self.display = nil` in `viewDidUnload` or `dealloc`.
The code above will be executed with `anObject` set to `nil`.
On the first line, the old `display` will get `released` (yay!).
Then, on the second line, `display` will be set to `nil`.
Having `display` be `nil` is nice because the rest of our code will know that our `view` is unloaded.

So always create an `@property` and do `self.outlet = nil` in `viewDidUnload/dealloc`
The unloading seems like (useless?) extra code most of the time, but we do it anyway.

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

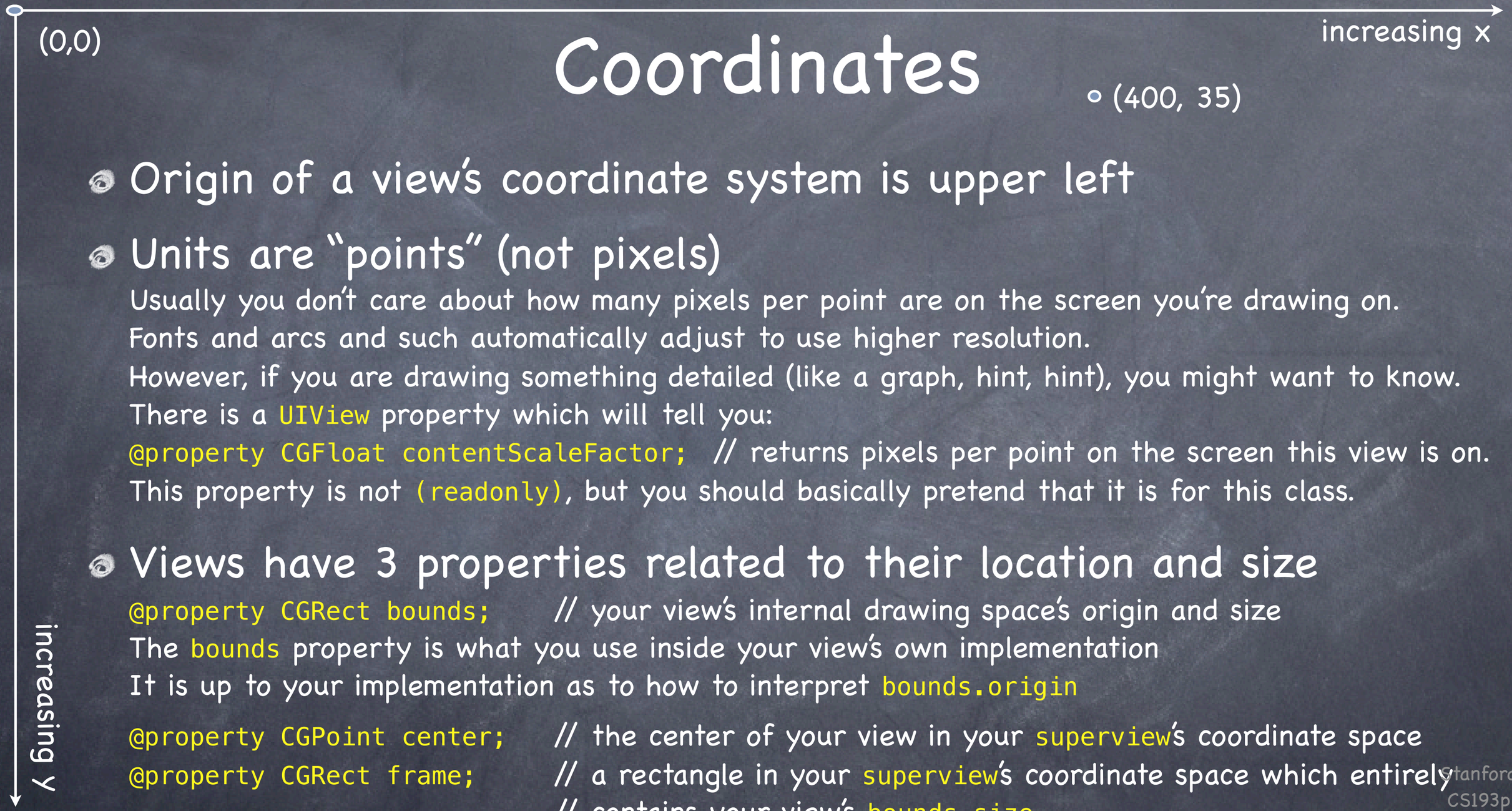
  C struct with two CGFloats in it: width and height.

```
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 class.

- Views have 3 properties related to their location and size

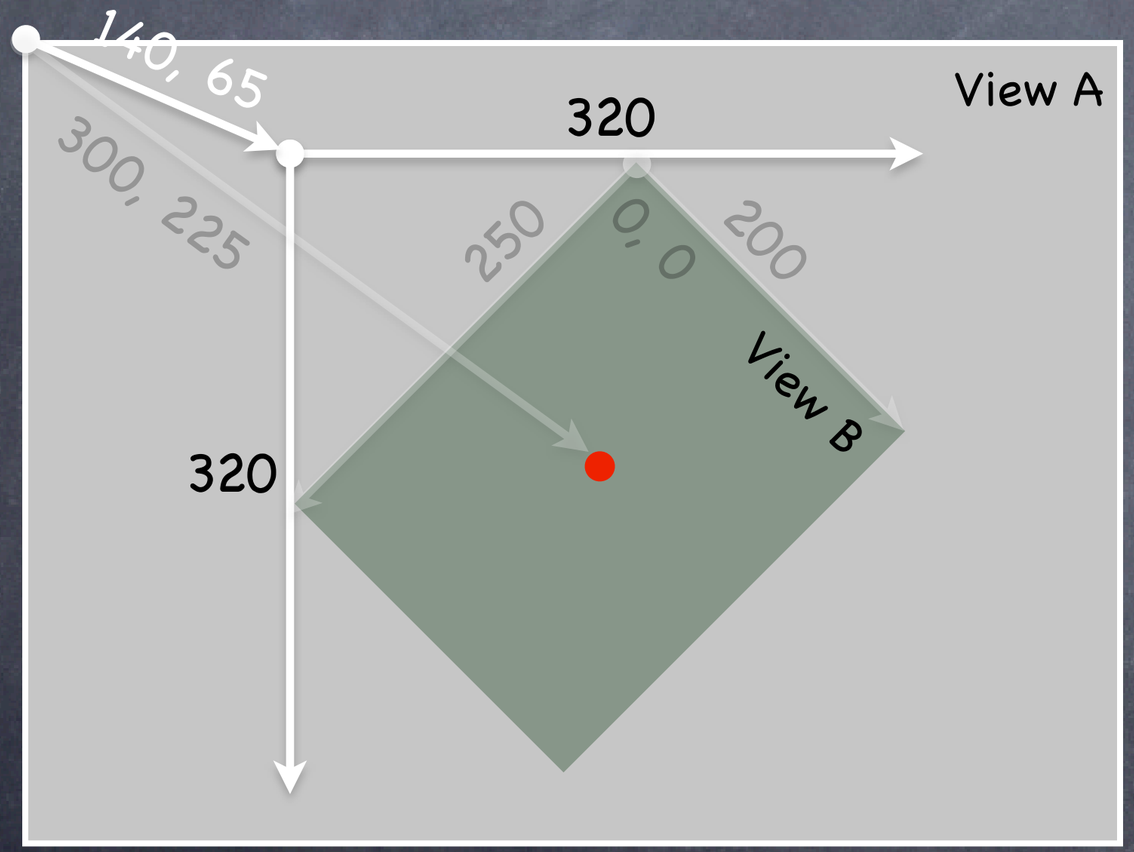
```
@property CGRect bounds;     // your view's internal drawing space's origin and size
```

  The `bounds` property is what you use inside your view's own implementation
  It is up to your implementation as to how to interpret `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
  They 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)`

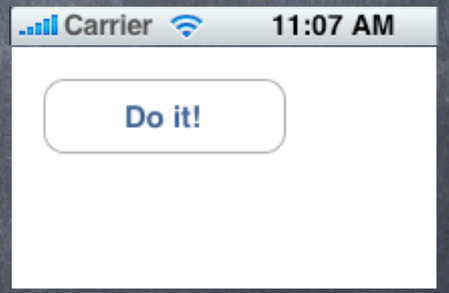
View B's middle in its own coordinate space is
`(bound.size.width/2+bounds.origin.x,`
`bounds.size.height/2+bounds.origin.y)`
which is `(100,125)` in this case.

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

Creating Views

- Most often you create views in Interface Builder
  Of course, Interface Builder knows nothing about a custom view class you might create.
  In that case, you drag out a generic `UIView` from the Library window and use the Inspector to change the class of the `UIView` to your custom class.
- How do you create a UIView in code (i.e. not in IB)?
  Just use `alloc` and `initWithFrame:` (`UIView`'s designated initializer).
- Example

```
CGRect buttonRect = CGRectMake(20, 20, 120, 37);
UIButton *button = [[UIButton alloc] initWithFrame:buttonRect];
button.titleLabel.text = @"Do it!";
[window addSubview:button];  // we'll talk about window later
[button release];    // okay because button is in view hierarchy now
```



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 next week. This week is 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 drawing 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

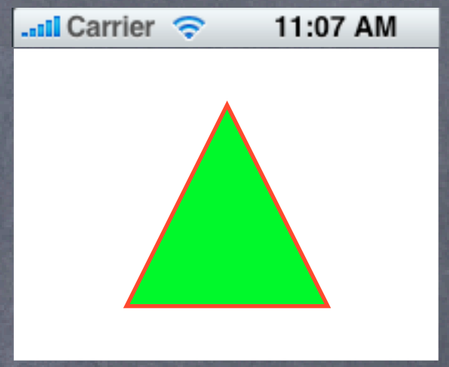
Context

- 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();
```

  - You do not have to free it or anything later, just use it for all drawing.

Define a Path



- Begin the path

```
CGContextBeginPath(context);
```

- Move around, add lines or arcs to the path

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

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

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

- Set any graphics state (more later), then stroke/fill the path

```
[[UIColor greenColor] setFill];    // object-oriented convenience method
[[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.

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.
You won't need to do that in this class, but this seemed like a good time to mention it!

Drawing Images

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

```
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);
```

Next Time

- Continuation of Happiness Demo
  Hook up Controller to Model and View
- Application Lifecycle
  From creation through event-handling and delegate method calling
- View Controller Lifecycle
  Same thing, but for `UIViewControllers`
- Navigation Controllers
  Building multi-screen applications
- Another Demo
  Navigation Controller

Demo

- Happiness
  Shows a level of happiness graphically using a smiley/frowny face
- Model
  `int happiness;` // very simple Model!
- View
  Custom view called `FaceView`
- Controller
  `HappinessViewController`
- Watch for …
  `drawRect:`
  How `FaceView` delegates its data ownership to the Controller with a protocol