iOS Mobile Development
Today

Views
How to draw custom stuff on screen.

Gestures
How to react to user's touch gestures.

Demo
SuperCard
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
  - A view can clip its subviews to its bounds or not (switch for this in Xcode, or method in UIView).

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
The hierarchy is most often constructed in Xcode graphically. Even custom views are often added to the view hierarchy using Xcode (more on this later).

But it can be done in code as well:
- `(void)addSubview:(UIView *)aView;` // sent to `aView`’s (soon to be) superview
- `(void)removeFromSuperview;` // sent to the view that is being removed

The top of this hierarchy for your MVC is the `@property view!` `UIViewController`’s `@property (strong, nonatomic) UIView *view`

It is critical to understand what this very simple `@property` is!
This is the view whose bounds will be changed when autorotation happens, for example.
This is the view you would programmatically add subviews to.
All your MVC’s View’s `UIView`’s eventually have this view as their parent (it’s at the top).
It is automatically hooked up for you when you drag out a View Controller in Xcode.
Initializing a UIView

Yes, you might want to override UIView's designated initializer
More common than overriding UIViewController's designated initializer (but still rare).

But you will also want to set up stuff in **awakeFromNib**
This is because initWithFrame: is **NOT** called for a UIView coming out of a storyboard!
But awakeFromNib is. Same as we talked about with UIViewController.
It's called “awakeFromNib” for historical reasons.

Typical code ...

- (void)setup { ... }
- (void)awakeFromNib { [self setup]; }
- (id)initWithFrame:(CGRect)aRect
  {
    self = [super initWithFrame:aRect];
    [self setup];
    return self;
  }
View Coordinates

**CGFloat**
Just a floating point number (depends on 64-bit or not), 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), 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, bound.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. So you drag out a generic `UIView` from the palette and use the Identity 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). Can also use `init` (frame will be `CGRectZero`).

Example

```objective-c
CGRect labelRect = CGRectMake(20, 20, 50, 30);
UILabel *label = [[UILabel alloc] initWithFrame:labelRect];
label.text = @"Hello!";
[self.view addSubview:label];  // Note self.view!
```
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 in a bit. First we’ll focus 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 directly (a C API, not object-oriented).
Or we can use the object-oriented `UIBezierPath` class (we’ll do it this way).

Core Graphics 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

`UIBezierPath`
Do all of the above, but capture it with an object.
Then ask the object to stroke or fill what you’ve created.
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?

UIBezierPath draws into the current context, so you don't need to get it if using that.

But if you're calling Core Graphics C functions directly, you'll need it (it's an argument to them).

Call the following C function inside your `drawRect:` method to get the current graphics context ...

```c
CGContextRef context = UIGraphicsGetCurrentContext();
```
Define a Path

Begin the path

```objective-c
UIBezierPath *path = [[UIBezierPath alloc] init];
```

Move around, add lines or arcs to the path

```objective-c
[path moveToPoint:CGPointMake(75, 10)];
```
Define a Path

 بداية الـPath

UIBezierPath *path = [[UIBezierPath alloc] init];

إضافة خطوط أو أبواب إلى الـPath

[path moveToPoint:CGPointMake(75, 10)];
[path addLineToPoint:CGPointMake(160, 150)];
Define a Path

Begin the path

UIBezierPath *path = [[UIBezierPath alloc] init];

Move around, add lines or arcs to the path

[path moveToPoint:CGPointMake(75, 10)];
[path addLineToPoint:CGPointMake(160, 150)];
[path addLineToPoint:CGPointMake(10, 150)];
Define a Path

🔗 Begin the path
   UIBezierPath *path = [[UIBezierPath alloc] init];

🔗 Move around, add lines or arcs to the path
   [path moveToPoint:CGPointMake(75, 10)];
   [path addLineToPoint:CGPointMake(160, 150)];
   [path addLineToPoint:CGPointMake(10, 150)];

🔗 Close the path (connects the last point back to the first)
   [path closePath];  // not strictly required but triangle won't have all 3 sides otherwise
Define a Path

Define a Path

Begin the path

UIBezierPath *path = [[UIBezierPath alloc] init];

Move around, add lines or arcs to the path

[path moveToPoint:CGPointMake(75, 10)];
[path addLineToPoint:CGPointMake(160, 150)];
[path addLineToPoint:CGPointMake(10, 150)];

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

[path closePath]; // not strictly required but triangle won’t have all 3 sides otherwise

Now that the path has been created, we can stroke/fill it

Actually, nothing has been drawn yet, we’ve just created the UIBezierPath.
Define a Path

🔗 Begin the path
UIBezierPath *path = [[UIBezierPath alloc] init];

🔗 Move around, add lines or arcs to the path
[path moveToPoint:CGPointMake(75, 10)];
[path addLineToPoint:CGPointMake(160, 150)];
[path addLineToPoint:CGPointMake(10, 150)];

🔗 Close the path (connects the last point back to the first)
[path closePath]; // not strictly required but triangle won't have all 3 sides otherwise

🔗 Now that the path has been created, we can stroke/fill it
Actually, nothing has been drawn yet, we've just created the UIBezierPath.
[[UIColor greenColor] setFill];
[[UIColor redColor] setStroke];
Define a Path

- **Begin the path**
  ```
  UIBezierPath *path = [[UIBezierPath alloc] init];
  ```

- **Move around, add lines or arcs to the path**
  ```
  [path moveToPoint:CGPointMake(75, 10)];
  [path addLineToPoint:CGPointMake(160, 150)];
  [path addLineToPoint:CGPointMake(10, 150)];
  ```

- **Close the path (connects the last point back to the first)**
  ```
  [path closePath];  // not strictly required but triangle won't have all 3 sides otherwise
  ```

- **Now that the path has been created, we can stroke/fill it**
  ```
  [[UIColor greenColor] setFill];
  [[UIColor redColor] setStroke];
  [path fill]; [path stroke];
  ```
Graphics State

- Can also set graphics state
e.g. path.lineWidth = 2.0;  // line width in points (not pixels)

- And draw rounded rects, ovals, etc.
  UIBezierPath *roundedRect = [UIBezierPath bezierPathWithRoundedRect:(CGRect)bounds
cornerRadius:(CGFloat)radius];

  Note: the “casts” in the arguments are just to let you know the types (i.e. they’re not required).
  UIBezierPath *oval = [UIBezierPath bezierPathWithOvalInRect:(CGRect)bounds];
  [roundedRect stroke];
  [oval fill];

- You can use a UIBezierPath to “clip” your drawing
  [roundedRect addClip];  // this would clip all drawing to be inside the roundedRect
Graphics State

Drawing with transparency in UIView

You know that UIColors can have alpha.
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 transparent.
If you don’t, results are unpredictable (this is a performance optimization property, by the way).

UIView’s @property CGFloat alpha can make the entire view partially transparent.
(you might use this to your advantage in your homework to show a “disabled” custom view)
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 (performance-wise)

Also, you can hide a view completely by setting `hidden` property

```objc
@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.
Also this can be used to swap two (or more) views in and out depending on 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 context functions.

```objc
- (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
}

- (void)drawGreenCircle:(CGContextRef)ctxt {
    CGContextSaveGState(ctxt);
    [[UIColor greenColor] setFill];
    // draw my circle
    CGContextRestoreGState(ctxt);
}
```
Drawing Text

We can use a **UILabel** as a subview to draw text in our view

But there are certainly occasions where we want to draw text in our `drawRect`.

To draw in `drawRect`:, use **NSAttributedString**

```objective-c
NSAttributedString *text = ...;
[text drawAtPoint:(CGPoint)p];  // NSAttributedString instance method added by UIKit
```

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

```objective-c
CGSize textSize = [text size];  // another UIKit NSAttributedString instance method
```

You might be disturbed that there are **drawing** methods in Foundation (a non-UI framework!). These **NSAttributedString** methods are **defined in UIKit** via a mechanism called **categories**. (so are the names of the attributes that define UI stuff (e.g. `NSFontAttributeName`). 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

**UIImageView is like UILabel for images**

But again, occasionally you want to draw an image in your `drawRect`.

**Create a UIImage object from a file in your Resources folder**

UIImage *image = [UIImage imageNamed:@"foo.jpg"]; // you did this in Matchismo

**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();
Now blast the `UIImage`'s bits into the current graphics context

```objective-c
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`

```objective-c
NSData *jpgData = UIImageJPEGRepresentation((UIImage *)myImage, (CGFloat)quality);
NSData *pngData = UIImagePNGRepresentation((UIImage *)myImage);
```
By default, when your UIView's bounds change, there is 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! It can be set in Xcode.

```swift
@property (nonatomic) UIViewContentMode contentMode;
```

These content modes **move** the bits of your drawing to that location ...

- UIViewContentMode{Left, Right, Top, Right, BottomLeft, BottomRight, TopLeft, TopRight}

These content modes **stretch** the bits of your drawing ...

- UIViewContentModeScale{ToFill, AspectFill, AspectFit} // bit stretching/shrinking

This content mode calls `drawRect:` to redraw everything when the bounds changes ...

- UIViewContentModeRedraw // it is quite often that this is what you want

Default is **UIViewContentModeScaleToFill** (stretch the bits to fill the bounds)
We've seen how to draw in our UIView, 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 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.

**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.
Adding a gesture recognizer to a UIView from a Controller

- (void)setPannableView:(UIView *)pannableView // maybe this is a setter in a Controller
{
    pannableView = pannableView;
    UIPanGestureRecognizer *pangr =
        [[UIPanGestureRecognizer alloc] initWithTarget:pannableView action:@selector(pan:)];
    [pannableView addGestureRecognizer:pangr];
}

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.).
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 modified the Model.
Adding a gesture recognizer to a `UIView` from a Controller

```objective-c
- (void)setPannableView:(UIView *)pannableView // maybe this is a setter in a Controller
{
    _pannableView = pannableView;
    UIPanGestureRecognizer *pangr =
        [[UIPanGestureRecognizer alloc] initWithTarget:pannableView action:@selector(pan:)];
    [pannableView addGestureRecognizer:pangr];
}
```

This is the action method that will be sent to the target (the `pannableView`) 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'd prefer.

We'll look at the implementation of this method in a moment.
Adding a gesture recognizer to a **UIView** from a Controller

```objective-c
-(void)setPannableView:(UIView *)pannableView // maybe this is a setter in a Controller
{
    _pannableView = pannableView;
    UIPanGestureRecognizer *pangr =
    [[UIPanGestureRecognizer alloc] initWithTarget:pannableView action:@selector(pan:)]);
    [pannableView addGestureRecognizer:pangr];
}
```

If we don’t do this, then even though the **pannableView** implements `pan:`, 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.”
Adding a gesture recognizer to a **UIView** from a Controller

```objective-c
-(void)setPannableView:(UIView *)pannableView // maybe this is a setter in a Controller
{
    _pannableView = pannableView;
    UIPanGestureRecognizer *pangr =
        [[UIPanGestureRecognizer alloc] initWithTarget:pannableView action:@selector(pan:)]);
    [pannableView addGestureRecognizer:pangr];
}
```

Only **UIView** instances can recognize a gesture (because **UIViews** 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).
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:
- (CGPoint)translationInView:(UIView *)aView;
- (CGPoint)velocityInView:(UIView *)aView;
- (void)setTranslation:(CGPoint)translation inView:(UIView *)aView;

Also, the base class, UIGestureRecognizer provides this @property:
@property (readonly) UIGestureRecognizerState state;
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’ll move on to the Changed and eventually the Ended state
Continuous can also go to Cancelled state (if the recognizer realizes it’s not this gesture after all)
So, given these methods, what would **pan** look like?

```objective-c
- (void)pan:(UIPanGestureRecognizer *)recognizer {
    if ((recognizer.state == UIGestureRecognizerStateChanged) ||
        (recognizer.state == UIGestureRecognizerStateEnded)) {
        // We're going to update our view every time the touch moves (and when the touch ends).
        // This is "smooth panning."
    }
}
```
So, given these methods, what would **pan:** look like?

```objc
- (void)pan:(UIPanGestureRecognizer *)recognizer {
    if ((recognizer.state == UIGestureRecognizerStateChanged) ||
        (recognizer.state == UIGestureRecognizerStateEnded)) {
        CGPoint translation = [recognizer translationInView:self];
        // This is the cumulative distance this gesture has moved.
    }
}
```
So, given these methods, what would `pan:` look like?

```swift
- (void)pan:(UIPanGestureRecognizer *)recognizer
{
    if ((recognizer.state == UIGestureRecognizerStateChanged) ||
        (recognizer.state == UIGestureRecognizerStateEnded)) {
        CGPoint translation = [recognizer translationInView: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 an @property called origin
        self.origin = CGPointMake(self.origin.x+translation.x, self.origin.y+translation.y);
    }
}
```
So, given these methods, what would **pan:** look like?

```objc
- (void)pan:(UIPanGestureRecognizer *)recognizer
{
    if ((recognizer.state == UIGestureRecognizerStateChanged) ||
        (recognizer.state == UIGestureRecognizerStateEnded)) {
        CGPoint translation = [recognizer translationInView: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 an @property called origin
        self.origin = CGPointMake(self.origin.x+translation.x, self.origin.y+translation.y);
    }
    [recognizer setTranslation:CGPointZero inView: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.
So, given these methods, what would `pan:` look like?

```objc
- (void)pan:(UIPanGestureRecognizer *)recognizer
{
    if ((recognizer.state == UIGestureRecognizerStateChanged) ||
        (recognizer.state == UIGestureRecognizerStateEnded)) {
        CGPoint translation = [recognizer translationInView: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 an @property called `origin`
        self.origin = CGPointMake(self.origin.x+translation.x, self.origin.y+translation.y);
        [recognizer setTranslation:CGPointZero inView:self];
    }
}
```
Other Concrete Gestures

UIPinchGestureRecognizer
@property CGFloat scale; // note that this is not readonly (can reset each movement)
@property (readonly) CGFloat velocity; // note that this is readonly; scale factor per second

UIRotationGestureRecognizer
@property CGFloat rotation; // not readonly; in radians
@property (readonly) CGFloat velocity; // readonly; radians per second

UISwipeGestureRecognizer
This one you “set up” (w/the following) to find certain swipe types, then look for Recognized state
@property UISwipeGestureRecognizerDirection direction; // what direction swipes you want
@property NSUInteger numberOfTouchesRequired; // two finger swipes? or just one finger? more?

UITapGestureRecognizer
Set up (w/the following) then look for Recognized state
@property NSUInteger numberOfTapsRequired; // single tap or double tap or triple tap, etc.
@property NSUInteger numberOfTouchesRequired; // e.g., require two finger tap?
Demo

SuperCard
Let's make a lot better-looking playing card!

What to watch for ...
Custom UIView with its own.drawRect:
setNeedsDisplay
UIBezierPath
Clipping
Pushing and popping graphics context
Drawing text with NSAttributedString and images with UIImage
Document Outline and Size Inspector in Xcode
Gestures recognizers hooked up in Xcode vs. programmatically
Controller vs. View as gesture handler
Coming Up

Next
Animation
Autolayout