

iOS Dev Accelerator

Week 8

- Custom Views

Building better Mousetraps

- Storyboard and XIBs will only get you so far
- Eventually, you'll be building a more complex UI, with elements that aren't available by default
- This could be a `UITableViewCell` or a custom `UIControl`

Life without Interface Builder

- Before iPhone OS 2.1, there was no Interface Builder
- Developers hand-coded all the views
- This worked fine, and still works today.
- Storyboard and XIB files really just contain serialized instances of `UIView`, `UILabel`, `UIButton`, etc...

No storyboard, no XIB

- Skip using a storyboard and XIB files and build all or part of your UI in code.
- Without an initial view controller, the AppDelegate is expected to create a `UIWindow`, set its `rootViewController`, and present it on-screen.
- Handle all of this inside `applicationDidFinishLaunchingWithOptions`

Up and running without Storyboard

- Define a `UIWindow` window property in your AppDelegate
- Initialize one or more view controllers when the app finishes launching
- Add the view controller's view as a subview on your window
- Set the `rootViewController` property on the window and mark it as visible

Up and running without Storyboard

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

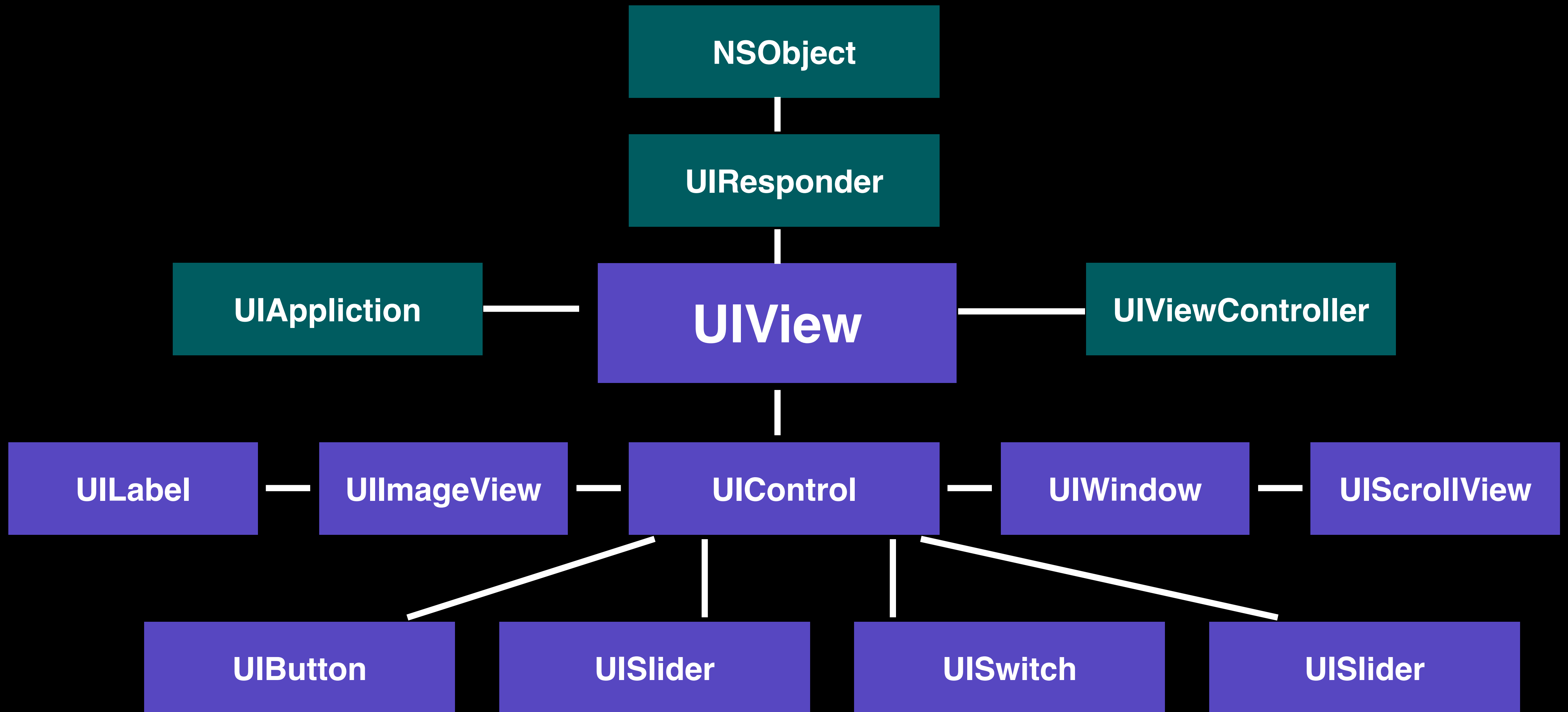
    func application(application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        self.window = UIWindow(frame: UIScreen.mainScreen().bounds)

        var viewController = UIViewController()
        var navController = UINavigationController(rootViewController: viewController)
        self.window?.rootViewController = navController

        self.window?.makeKeyAndVisible()

        return true
    }
}
```

UIView Hierarchy



Subclassing UIView

- Subclass a `UIView` and layout the UI in code or inside a XIB file
- Apple documents extensively what methods should be overridden and when
- Two important ones are `initWithFrame:` and `layoutSubviews`
- If your view is loaded from a NIB, you need to implement `initWithCoder:`

Subclassing UIView

- Methods to override during initialization
 - `initWithFrame`— implement this method and perform custom initialization
 - `initWithCoder`— implement this method if your view is loaded from a NIB and you required custom initialization

Subclassing UIView

- Methods to override for drawing and layout
 - `drawRect`— implement this method if your view will do custom drawing
 - `layoutSubviews`— use this method to position subviews within your view. The frame property is correct when this method is called.
 - `sizeThatFits`— use this if your view should have a custom size, and you need to calculate a size.

Subclassing UIView

- Methods to override for touch handling
 - `touchesBegan:withEvent`
 - `touchesMoved:withEvent`
 - `touchesEnded:withEvent`
- `gestureRecognizerShouldBegin`— implement this if your view supports touch handling and you want to prevent gesture recognizers from triggering default actions

Autoresizing Masks

- Before AutoLayout, there were Autoresizing Masks
- Define how a view should shrink, expand, and resize itself
 - Flexible height and width
 - Flexible margins
- Not dependent on sibling views like AutoLayout is
- Sometimes referred to as Springs and Struts

Autoresizing Masks

- Can still use Autoresizing Masks, but AutoLayout is preferred.
- By default, AutoLayout will attempt to translate any Auto Resizing Masks into constraints
- If you setup constraints manually in code, you need to turn off this default translation
 - `(void)setTranslatesAutoresizingMaskIntoConstraints:(BOOL)flag`

Autoresizing Masks

- Define a UILabel with flexible left and right margins
- The labels margins will adjust from the original frame, based on content

```
UILabel *label = [[UILabel alloc] initWithFrame:CGRectZero];
UIViewAutoresizing mask = (UIViewAutoresizingFlexibleLeftMargin |
                           UIViewAutoresizingFlexibleRightMargin);
[label setAutoresizingMask:mask];

[self.view addSubview:label];
```

Views and Layers

- All views on iOS are **layer backed**
- This means there is one or more underlying `CALayers` behind each `UIView`
- Access a View's layer through the `layer` property
- Perform most operations on the View
- For finer control over rendering and animation, perform operations on the View's layer.

Views and Layers

- Layers or `CALayers`
 - Represent position, shape and anchor point
 - Do not receive touch events
 - Are light-weight and use implicit animations. Changes are animated.
- By contrast, Views or `UIViews`
 - Can receive touch events
 - Are layer backed (always, on iOS)
 - Do not use implicit animations, changes not automatically animated.

Demo

Core Graphics

- Two dimensional drawing engine for iOS and OS X
- Allows for path based drawing, painting with transparency, shading, drawing shadows, color management, anti-aliased rendering
- Will leverage the Graphics Hardware whenever possible.
- Sometimes referred to as Quartz 2D, descendant of QuickDraw
- All drawing requires a Graphics Context

What's a Graphics Context?

- A graphics context represents a drawing destination
- Contains all drawing parameters and any device specific info the drawing system needs.
- Graphics context defines drawing attributes such as the color the use, any clipping paths, line width and style, compositing info, etc.
- An instance of `CGContextRef`

Graphics Context

- Core Graphics is stateful; a graphics property will remain set until you change it.
- Core Graphics maintains a stack of graphics states
- Save the current state using `CGContextSaveState`
- Restore to previous context state using `CGContextRestoreState`
- Need to balance calls to Save state with calls to Restore State

Implementing drawRect

- All `UIView`s will have a `drawRect` method. Implement this method and perform custom drawing operations.
- Only implement if needed; empty implementations are expensive
- `drawRect` is called often and its easy enough to create expensive drawing commands.
- Be wary of how many pixels you are pushing onto the screen.
- When `drawRect` is called, UIKit has already created a graphics context.

Implementing drawRect

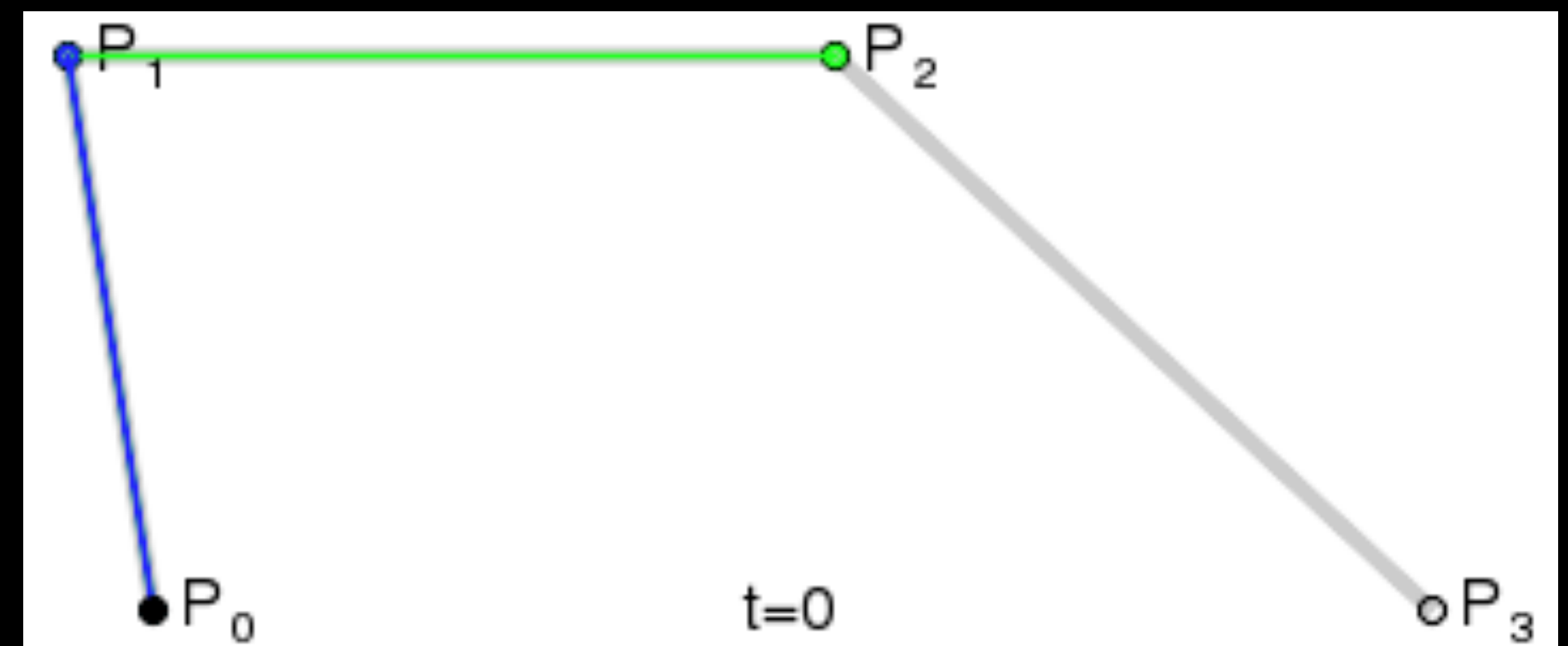
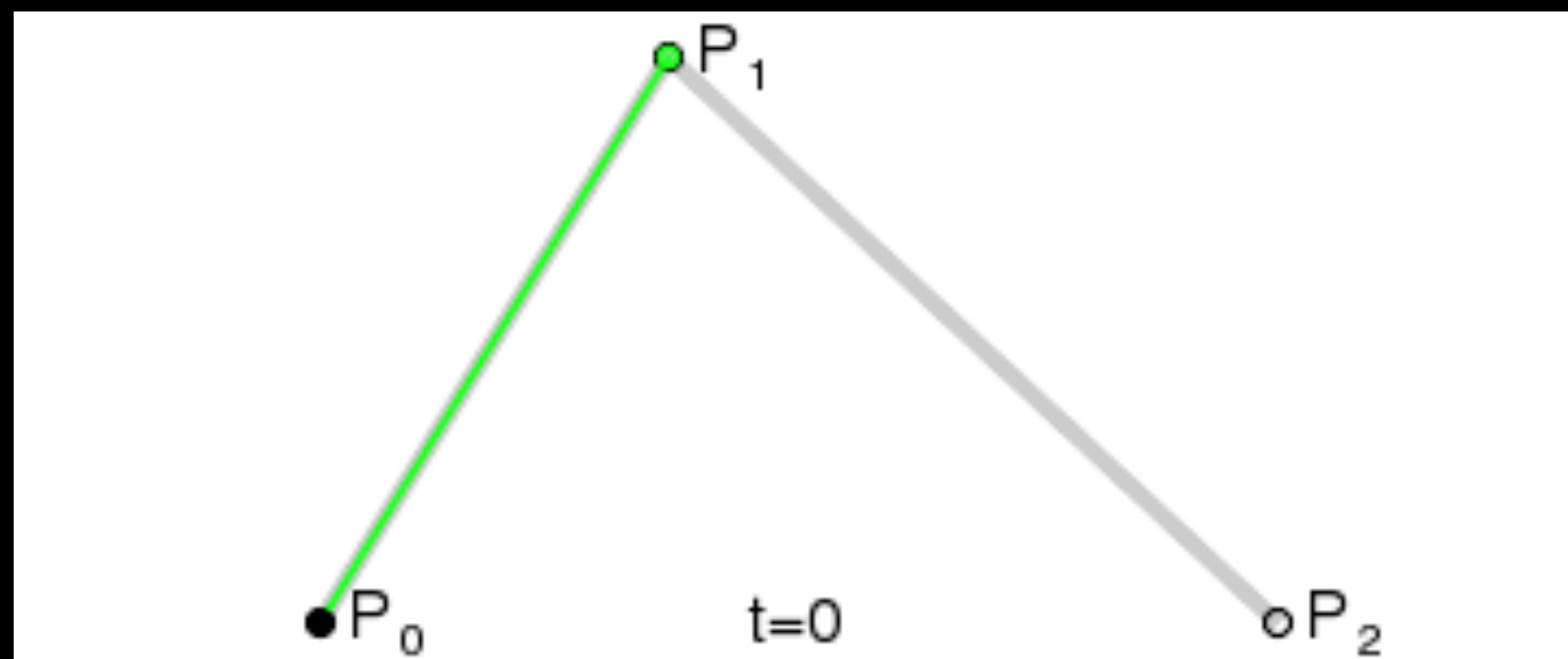
```
class BoxView: UIView {  
    /*  
    // Only override drawRect: if you perform custom drawing.  
    // An empty implementation adversely affects performance during animation.  
    override func drawRect(rect: CGRect) {  
        // Drawing code  
    }  
    */  
}
```

Bézier Curves

- Parametric curve frequently used in computer graphics to model curved lines or surfaces.
- Points on parameter curve are determined by a value to some function, called the parameter— concepts not crucial to iOS drawing, but helpful.
- At a minimum, a Bézier curve will have two endpoints and one control point.
- Control points determine the shape of the curve. They are parameters in the function.

Bézier Curves

- Wikipedia has some great animations that illustrate how various curves are drawn, using control points.
- Below are two simple curves, with one and two control points each.



http://en.wikipedia.org/wiki/Bézier_curve

UIBezierPath

- Describe a path consisting of straight or curved line segments
- `UIBezierPath` has many helpers for paths; pass in a `CGRect` bounding your path
 - `bezierPathWithRect`
 - `bezierPathWithRoundedRect:cornerRadius`
 - `bezierPathWithOvalInRect`
- `UIBezierPaths` can be built using primitive `moveTo`, `lineTo` commands
- Paths can be open or closed.

UIBezierPath, example

- Create a rectangular path, set the line width to 5 points, set the stroke color to blue, and the finally stroke the path.

```
class BoxView: UIView {  
    override func drawRect(rect: CGRect) {  
        let path = UIBezierPath(rect: rect)  
        path.lineWidth = 5.0  
  
        UIColor.blueColor().setStroke()  
  
        path.stroke()  
    }  
}
```



UIBezierPath, example

- Create a rounded rectangular path with a corner radius of 15 points, set the line width to 5 points, set the stroke color to blue and the fill color to purple. Finally stroke and fill the path.

```
class AJSBoxView: UIView {  
    override func drawRect(rect: CGRect) {  
        let path = UIBezierPath(roundedRect: rect,  
                                cornerRadius: 15.0)  
  
        path.lineWidth = 5.0  
  
        UIColor.blueColor().setStroke()  
        UIColor.purpleColor().setFill()  
  
        path.fill()  
        path.stroke()  
    }  
}
```



Demo

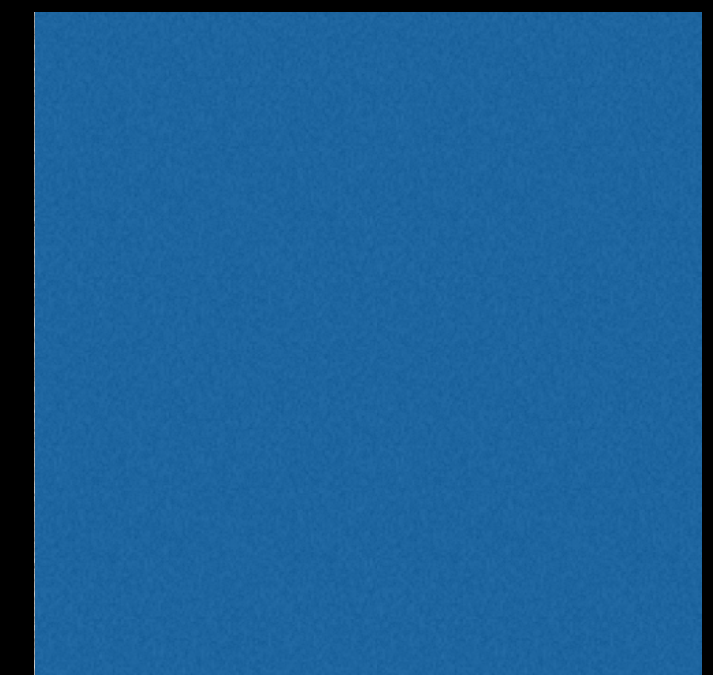
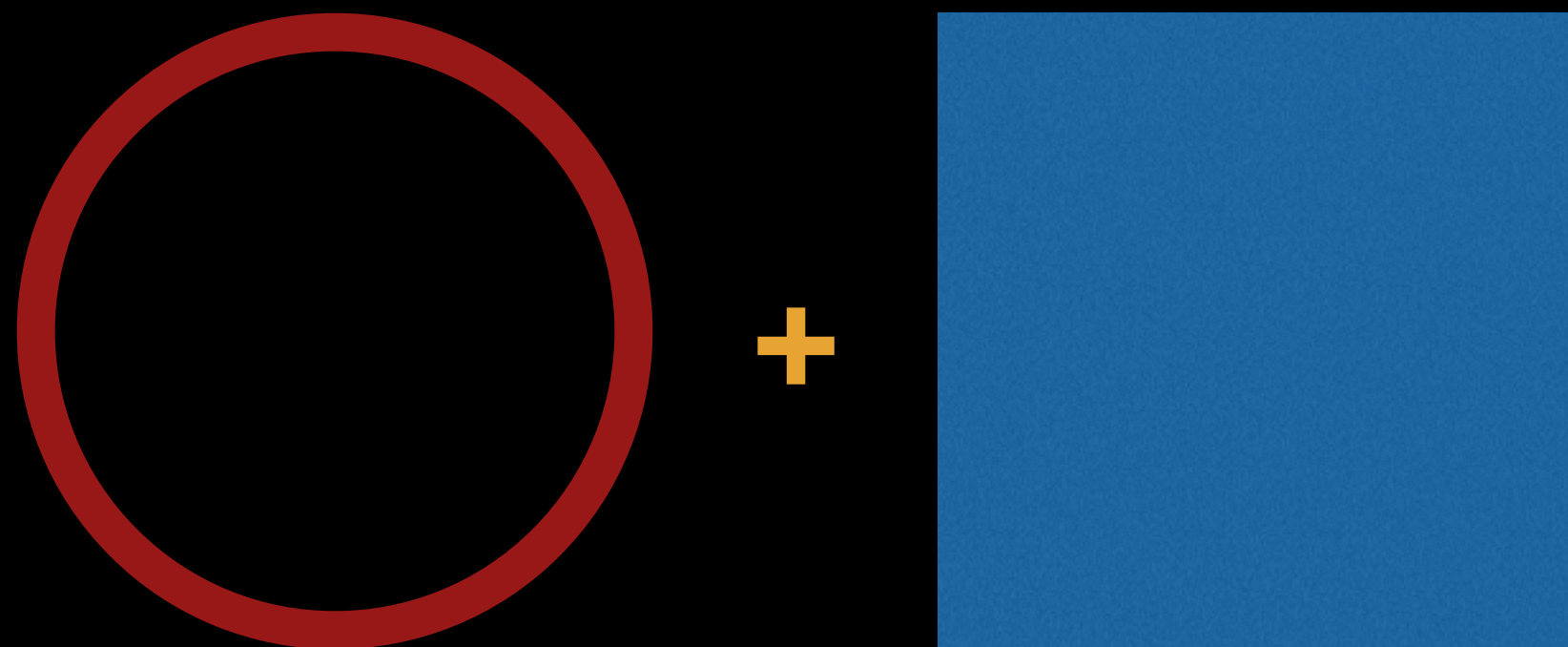
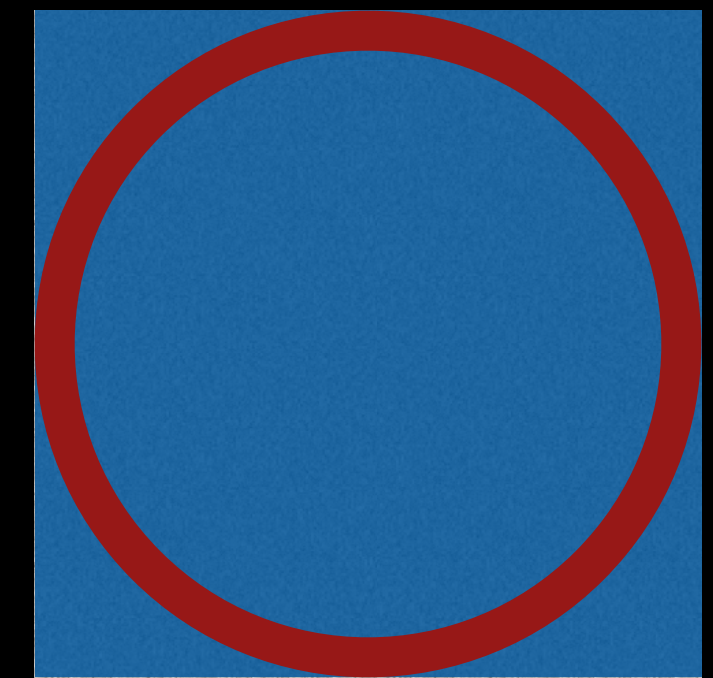
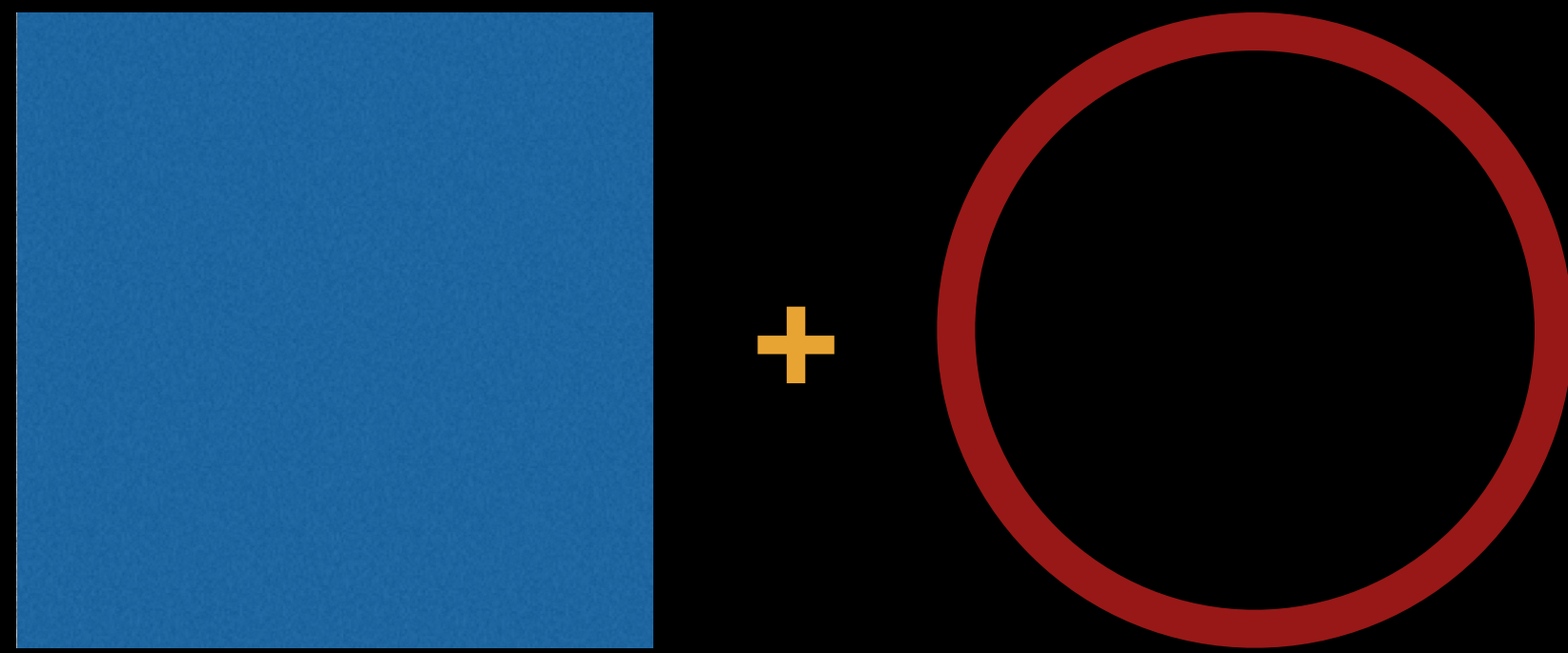
Painter's Model

- Technique for determining which polygons or surfaces are visible, and which are not.
- Each successive drawing operation adds a new layer of “paint” onto the “canvas”
- The **last color wins**, just like if you were painting a room.
- In practice, you should fill a path before stroking it. Otherwise, the stroke is obstructed by the fill.

Painter's Model

Drawing Order

Result



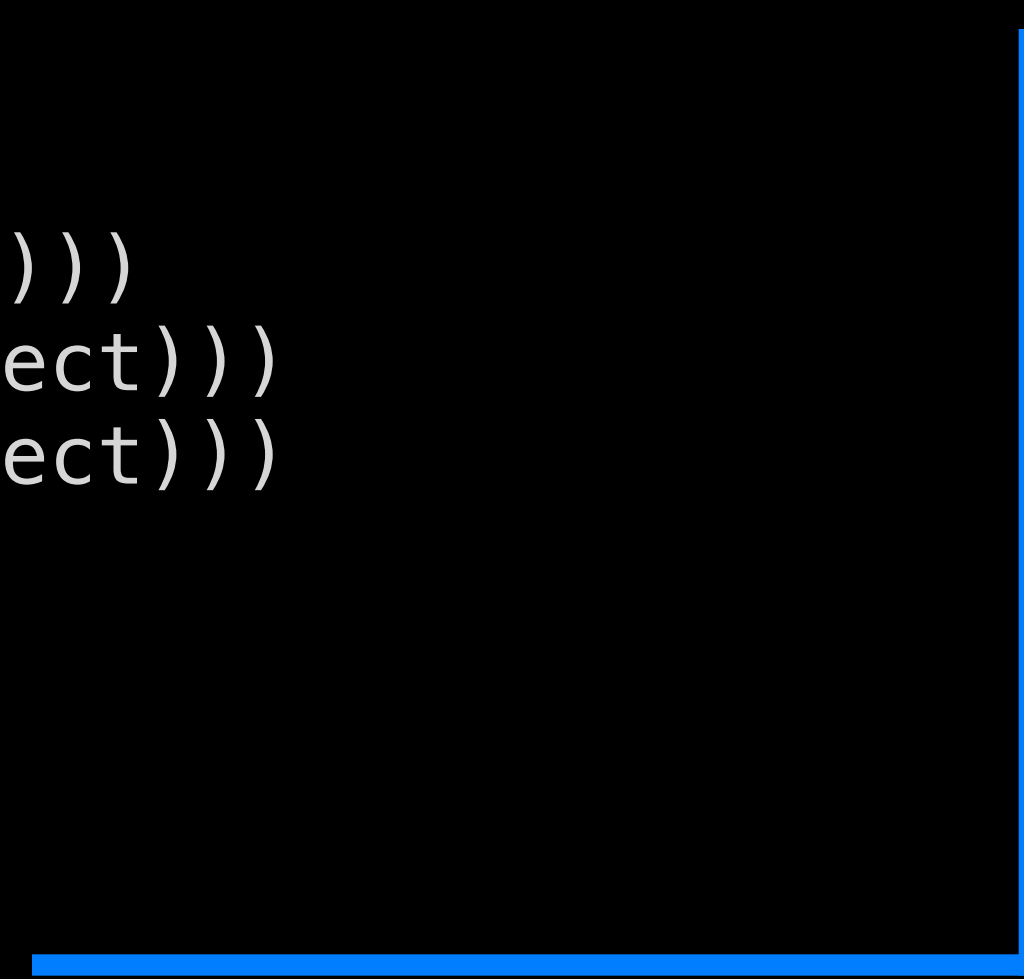
Graphics Pen

- The Core Graphics pen will position itself “in the middle” of a point
- For example when you move the pen to (0, 0) the actual tip of the pen will be centered over (0.5, 0.5)
- The resulting line will be half on one pixel and half on another
- Core Graphics cannot fill “half a pixel” so the resulting line will be drawn using anti-aliasing, i.e not a crisp sharp line, but blurry
- To draw sharp lines with aliasing, offset the pen by 0.5 points

UIBezierPath, example

- Create a path from individual line segments, but first moving the pen, and then adding line segments. Finally stroke the path.

```
class AJSRightAngleView: UIView {  
    override func drawRect(rect: CGRect) {  
        let path = UIBezierPath()  
        path.moveToPoint(CGPointMake(CGRectGetMinX(rect), CGRectGetMaxY(rect)))  
        path.addLineToPoint(CGPointMake(CGRectGetMaxX(rect), CGRectGetMaxY(rect)))  
        path.addLineToPoint(CGPointMake(CGRectGetMaxX(rect), CGRectGetMinY(rect)))  
  
        UIColor.blueColor().setStroke()  
  
        path.stroke()  
    }  
}
```

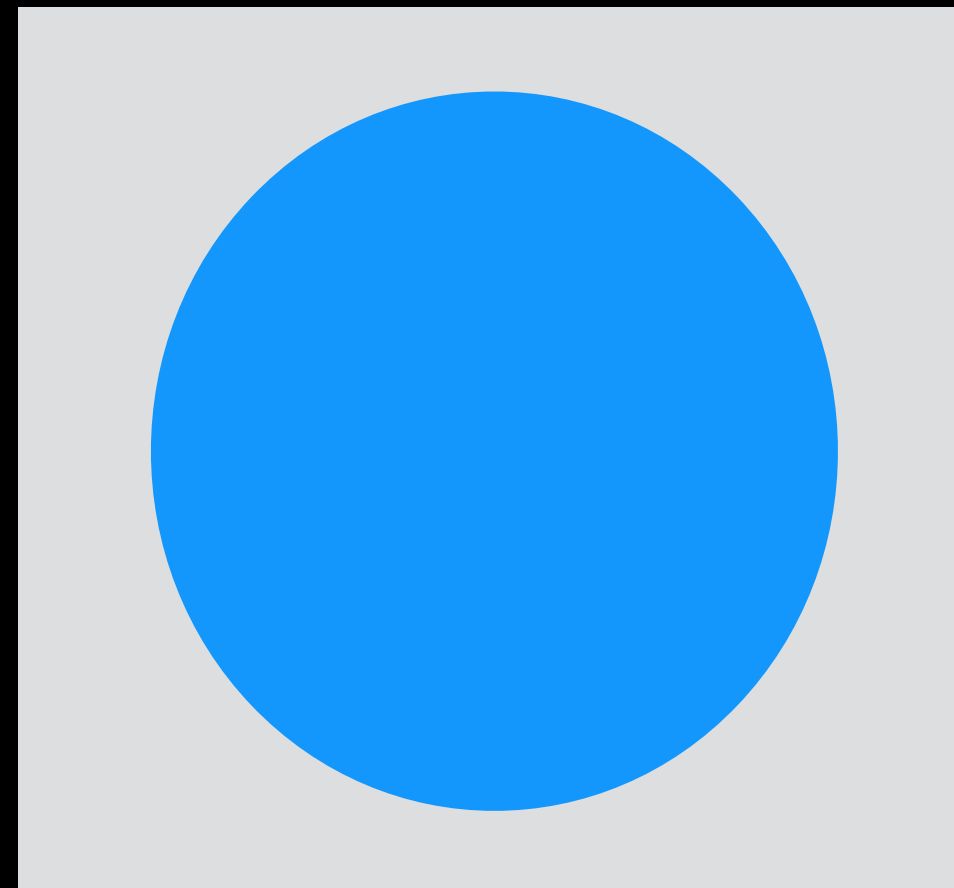
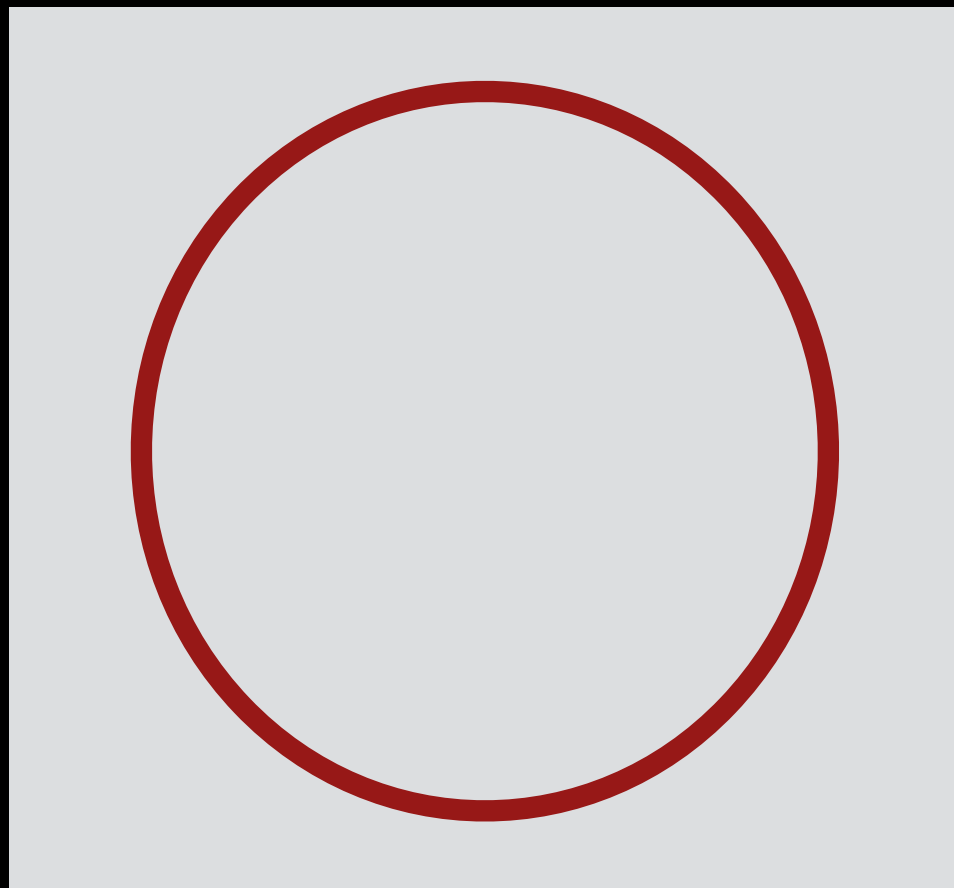


Clipping Paths

- Clipping paths serve as a mask, allowing you to block out part or all of the page you don't wish to paint
- When clipping path is define, Core Graphics will not draw outside of the path.
- When the context is initially defined, the clipping area is the full size of the context.
- To define another clipping region, make a new path and set it as the clipping path.

Clipping Paths

```
override func drawRect(rect: CGRect) {  
    UIColor.blueColor().setFill()  
  
    let clipPath = UIBezierPath(ovalInRect: CGRectInset(rect, 20, 20))  
    clipPath.addClip()  
  
    let path = UIBezierPath(rect: rect)  
    path.fill()  
}
```



Red circle is the clipping path (not drawn)

Blue circle is the resulting fill, masked by
then clipping path

Resizable Images

- Resizable images specify how they can be resized; which areas to stretch and which to not.
- Should use as small an image as possible, and resize/stretch to fit your needs.
- This works for nicely for single color images
- Should use an odd value for then dimensions you are stretching
- Not intended for picture images; only asset images.
 - `(UIImage *)resizableImageWithCapInsets:(UIEdgeInsets)capInsets`

Resizable Images

61



162

```
let image = UIImage(named: "red-button.png")
image!.resizableImageWithCapInsets(UIEdgeInsetsMake(0, 30, 0, 30))

let button = UIButton(frame: CGRectMake(0, 0, 400, 162))
button.setBackgroundImage(image, forState: UIControlState.Normal)
```



Original Asset is 61 x 162 points

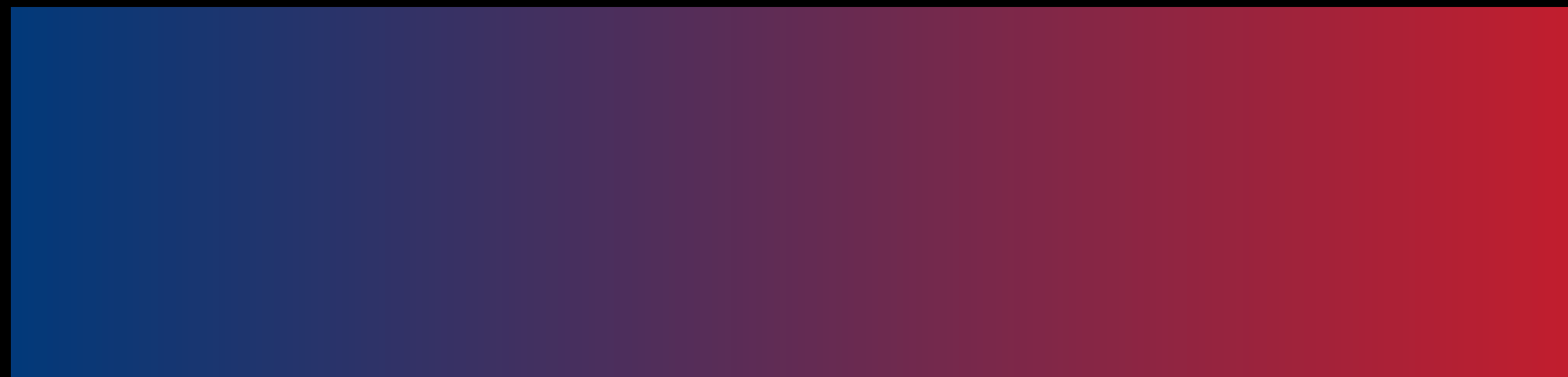
Resized Asset is 400 x 162 points

Images from Color Patterns

- Use a small pattern of colors and construct a new color
- This color can be tiled across a plane
- Helpful technique for keeping resource and memory low, while still styling the UI
- Way to create a brushed metal texture and sand texture effect

Gradients

- A gradient is a color that varies from one color to another
- Core Graphics supports two types of gradients
 - Axial or linear gradients vary the color along an axis defined by two points
 - Radial gradient varies the color along from an origin point and extending out in a circle.



Gradients

```
override func drawRect(rect: CGRect) {
    var context = UIGraphicsGetCurrentContext()

    CGContextSaveGState(context);

    var colorspace = CGColorSpaceCreateDeviceRGB()
    var startComponents = CGColorGetComponents(UIColor.redColor().CGColor)
    var endComponents = CGColorGetComponents(UIColor.blueColor().CGColor)

    var components = [startComponents[0], startComponents[1], startComponents[2], startComponents[3],
                      endComponents[0], endComponents[1], endComponents[2], endComponents[3]]

    var locations:[CGFloat] = [0.0, 1.0]
    var gradient = CGGradientCreateWithColorComponents(colorspace, &components, &locations, 2)
    var startPoint = CGPointMake(0.0, rect.height)
    var endPoint = CGPointMake(rect.width, rect.height)

    CGContextDrawLinearGradient(context, gradient, startPoint, endPoint, 0)

    CGContextRestoreGState(context);
}
```

Demo

CGRects and Insets

- Several helper methods exist to create rectangles and determine positions within a rectangle
 - `CGRectMake(0, 0, 100, 100)`
 - `CGRectGetMaxX()`, `CGRectGetMaxY()`, etc...
- Use `CGRectInset()` to create a rectangle that is inset from another rectangle by some x and y values
 - `CGRectInset(rect, 5.0, 5.0)`
- <http://nshipster.com/cggeometry/>

UI and CG API similarities

- `UIBezierPath` is a wrapper around Core Graphics drawing commands. In the C API, these are methods prefixed with `CG`.
- It is recommended that you use UI drawing primitives inside UIKit
 - There are optimizations to make `UIBezierPath` faster
- You should be using `UIBezierPath` and friends as opposed to `CGMutablePathRef`— but you can still do it all with CG functions.
- <http://stackoverflow.com/a/6353597/4241178>

Additional Resources

- Practical Drawing for iOS Developers, Session 129, WWDC 2011
- Drawing and Printing Guide for iOS
- Quartz 2D Programming Guide