Finger Painter

Overview

- Create a drawing using the touch-based user interface of iOS, and Core Graphics.
- Discover how to respond to **touch events.**
- Analyze and develop a drawing strategy using touch locations.
- Discover Core Graphics contexts and procedural drawing idioms.

Part 1

- Add a custom view to the interface and draw a straight line with **Core Graphics.**

Part 2

- Draw a line as the user touches the screen.
- Discover how touching the device screen generates **event-driven** method calls.

Part 3

- Use the touch events to obtain **coordinates** for drawing.

Part 4

- Draw a **continuous line** as a finger moves across the screen.

Part 5

- Implement the drawing method with Core Graphics.

Part 6

- Enable a button that **clears the image** from the screen.

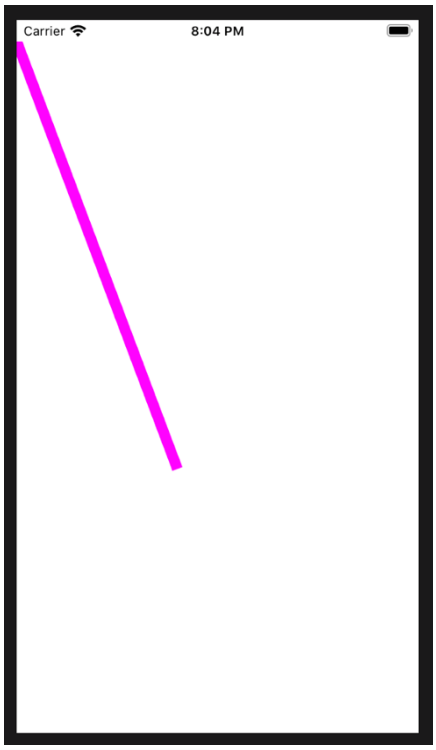
Learning Outcomes

- Relate interface components to the **UIView** class.
- Practice adding a new class to an Xcode project.
- Discover how interface components in Interface Builder can be bound to specific classes.
- Discover common **Core Graphics** functions and the **CGContextRef** data type.
- Replicate using Core Graphics functions to draw a line.
- Implement **UIResponder** methods in a controller to handle touch events.
- Discover how touching the device screen generates event-driven method calls.
- Recognize how touch events can send multiple **UITouch** objects to event handlers.
- Discover how to obtain point coordinates from **UITouch** objects.
- Generalize the purposes of **UIView** and **UIImageView** interface elements.
- Practice establishing **outlet** connections between a view and controller, and declaring properties.
- Strategize a drawing method using points generated by touch input.
- Practice using swift **optional binding** to check for the presence of values.
- Observe how external and local argument names can lend better semantics when calling and implementing functions.
- Recognize the procedural steps to draw lines with Core Graphics.
- Practice establishing an **action** connection from a view to a controller method.
- Replicate implementing button behaviors to add additional features to an app.

Vocabulary

- UIView
- UITouch
- UIImageView
- UIImage
- UIResponder
- subclass
- superclass
- override
- context
- point
- path
- stroke
- Core Graphics
- CGContextRef
- CGPoint
- event
- set (data structure)
- outlet connection
- property
- optional
- optional binding
- if let
- action
- nil

Part 1: Draw a line

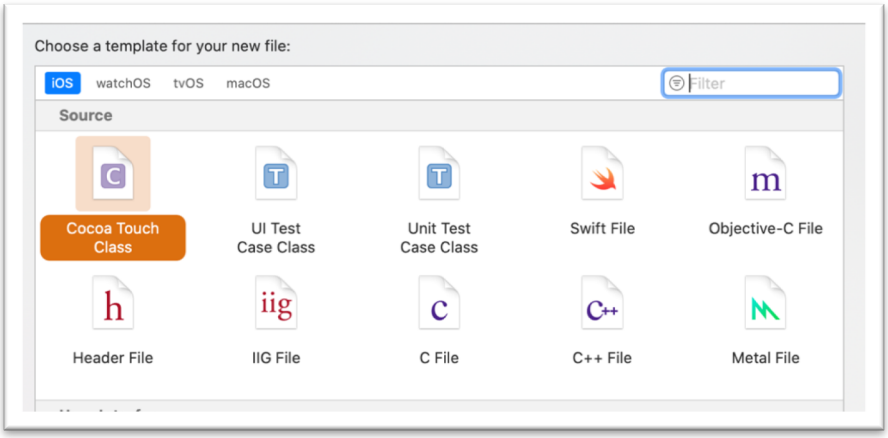


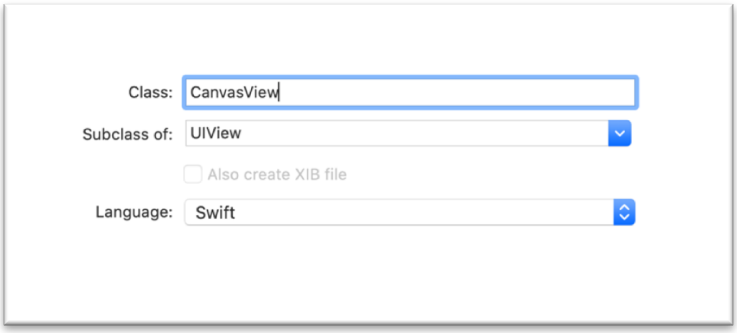
Objective: Add a custom view to the interface and override the drawRect: method to draw a straight line with Core Graphics.

Interface components are subclasses of UIView, and by overriding its drawRect: method, you can draw on the screen. You can create a custom UIView subclass to represent a custom view, in order to control exactly how the view is drawn.

➤ Create a new project. Call it Finger Painter.

➤ Add a new class (⌘ N) called **CanvasView**. New file, iOS source, CocoaTouch class, subclassed from UIView. Name it CanvasView.swift.





➤ Open Main.storyboard. Add a new **view** object. Add **constraints** to connect the top, bottom, left and right edges of the view to the edges of the screen with a distance of 0.

➤ With the new View object selected, open the Identity Inspector ( ), and change the Class to **CanvasView**. The Canvas View in Interface Builder is now bound to the CanvasView class. The Canvas View will be drawn using the code you will write in the CanvasView class.

We need to add code to the CanvasView to draw something on the screen. Explore the UIView class reference, paying particular attention to the **drawRect:** method. When iOS renders a view, it will call the drawRect: method of the view object. So, we will put our drawing in the drawRect: method.

➤ Update the CanvasView implementation with a custom drawRect: method. (Uncomment the drawRect: method that came with the CanvasView.)

```
override func draw(_ rect: CGRect) {
      let context = UIGraphicsGetCurrentContext()
      context?.setStrokeColor(UIColor.magenta.cgColor)
      context?.setLineWidth(10)
      context?.beginPath()
      context?.move(to: CGPoint(x: 0, y: 0))
      context?.addLine(to: CGPoint(x: 150, y: 400))
      context?.strokePath()
}
```

➤ Run the app (⌘ R), and observe the diagonal magenta line appear.

Look at the CG function names in drawRect:. CG stands for **Core Graphics**. The Core Graphics framework is used to implement custom drawing with graphics primitives, such as lines. The context represents a drawing environment, and is used by other Core Graphics functions to create a drawing.

Core Graphics uses a "painter's model" for drawing, not unlike how one might pick up a brush of a particular color and width, choose a starting and ending point, and then stroke the path between the two points.

- The `setStrokeColor` function sets the active color for the drawing context.
- The `setLineWidth` function sets the active width of "the brush stroke" for a context.
- The `beginPath` function indicates that a path of points is about to be defined.
- The `move:To:Point` function indicates the first point in the path.
- The `addLine:To:Point` function draws a line from the previous point to a new point.
- The `strokePath` function then "paints the stroke" for the connected points in the path.

Obtaining a drawing context, picking "a brush color and width," creating a path, then stroking that path, is a common line-drawing idiom for Core Graphics.

Think about:

- What would you change to make the line color blue or red?
- What would you change to draw a diagonal line from the upper left to the lower right?
- Where is the origin on the canvas?
- In what direction to the x and y axes increase?

Modifications and Extensions

- Modify draw:Rect to fill the screen with diagonal lines. Maybe generate them randomly.

```
for i in 0...10 {
      context?.setStrokeColor(UIColor.orange.cgColor)
      context?.beginPath()
      context?.move(to: CGPoint(
            x: randomFloat()*rect.width,
            y: randomFloat()*rect.height))
      context?.addLine(to: CGPoint(
            x: randomFloat()*rect.width,
            y: randomFloat()*rect.height))
      context?.strokePath()
}

func randomFloat() -> CGFloat {
      return CGFloat(arc4random() % 10000) / 10000
}
```

- Investigate the addEllipse:In:Rect function, and fill the screen with multiple colored circles.

```
context?.beginPath()
context?.setStrokeColor(UIColor.green.cgColor)
context?.addEllipse(in: CGRect(x: 50, y: 60, width: 70, height: 80))
context?.strokePath()
```

Part 2: Respond to touch events

Objective: We want to draw a line as the user touches the screen. Using drawRect: is ok for simple custom views that don't change very much, but how can we create a custom view that continuously updates as we touch the screen?

One approach is to draw a line from point to point as the user drags a finger across the screen. App view controllers inherit from **UIViewController**, which inherits from **UIResponder**. So, we can use UIResponder to detect the dragging finger.

Using the Xcode Documentation and API Reference (⇧⌘ 0), explore the UIResponder class reference, drawing attention to the touchesBegan:withEvent: and touchesMoved:withEvent: methods. Overriding touchesBegan:withEvent: and touchesMoved:withEvent: in the view controller can facilitate drawing a continuous line as the user drags a finger on the screen.

➤ In the ViewController class, add an implementation of touchesBegan:withEvent:, which gets called as soon as a user touches the screen. (You can just type "touchesBegan" and Xcode will help you fill in the rest.)

```
override func touchesBegan(_ touches: Set<UITouch>,
                           with event: UIEvent?) {
      // print a message
      print ("touches began")
}
```

➤ Add an implementation of touchesMoved:withEvent:, which is called repeatedly, as the user drags a finger across the screen. (You can just type "touchesMoved" and Xcode will help you fill in the rest.)

```
override func touchesMoved(_ touches: Set<UITouch>,
                           with event: UIEvent?) {
      // print a message
      print ("touches moved")
}
```

Part 3: Obtain point coordinates

Objective: Capture point coordinates within the touch event handlers, and inspect the x and y components of the touch coordinates.

Because iOS can respond to multiple touches (with multiple fingers), a **Set** of **UITouch** objects is passed to touchesBegan:withEvent: and touchesMoved:withEvent:.

➤ Update the implementation of touchesBegan:withEvent:. Add only the code in green.

```
override func touchesBegan(touches: Set<UITouch>,
                           withEvent event: UIEvent) {
      let touch = touches.first!
      let point = touch.location(in: view)
      print (point.x, point.y)
}
```

Explore the UITouch class reference, especially the **locationInView:** method. touchesBegan: withEvent: is passed a Set of objects, touches; we retrieve the first object in the Set. All ViewController objects have an inherited view property. The touchesBegan: withEvent: method obtains a **CGPoint**, representing a coordinate within the view, from the UITouch object.

➤ Update the implementation of touchesMoved:withEvent:. Add only the green code.

```
override func touchesMoved(touches: Set<UITouch>,
                           withEvent event: UIEvent) {
      let touch = touches.first!
      let point = touch.location(in: view)
      print (point.x, point.y)
}
```

touchesMoved:withEvent: will be called multiple times while the finger drags across the screen. The CGPoint obtained from the touch events can be used to draw a line from point to point as the finger moves.

➤ Run the app (⌘ R), click the screen to simulate a touch, and observe the coordinates printed on the console. Click and drag on the screen to simulate a finger moving across the screen, and observe the coordinates printed on the console by touchesMoved: withEvent:.

Part 4: Endpoints

Objective: Add an Image View for the drawing, and update the implementation of the event handlers to simulate the act of drawing from point to point.

Now that we have the points of the touches, how might we draw a continuous line as a finger moves across the screen? One approach is to draw a line from point to point as the user drags a finger across the screen, updating the image displayed on the screen as the user "draws."

drawRect: has a particular purpose, to draw static or data-driven images. But it does not easily support user interface events like touches, and can become inefficient for dynamic drawing. We can use a UIImageView object to accomplish the task of displaying a continuously updated image containing the line to be drawn.

➤ Using Interface Builder, delete the custom CanvasView object from the interface.

➤ Using Interface Builder and the Object Library (+), add a new Image View object. Add constraints to make the image view fill the screen.

➤ Using the Assistant Editor (^⌥⌘↩), Control-drag from the Image View to the ViewController class to create an outlet property. Call the outlet variable “canvas.”

```
@IBOutlet weak var canvas: UIImageView!
```

The view controller will need to keep track of the first point obtained when touchesBegan: withEvent: is called, to serve as the first starting point for the line to be drawn.

➤ Declare a CGPoint property in the ViewController class.

```
var start: CGPoint?
```

The start point will frequently change as the user touches the screen. The type is an optional, since the ViewController initializer will not assign a value to the property. It will get a value later.

➤ Update the implementation of touchesBegan:withEvent: to obtain the coordinate from the UIImageView, and to store the point of the touch in the start property.

```
override func touchesBegan (touches: Set<UITouch>,
                            withEvent event: UIEvent) {
      let touch = touches.first!
      start = touch.locationInView(canvas)
}
```

Because the drawn line will follow the finger as it moves, the touchesMoved:withEvent: method should draw a line from the controller start property to the new point captured within the touchesMoved:withEvent: method. For each subsequent movement, a new start should be stored, so that the next time touchesMoved:withEvent: is called, a line can be drawn from the new start to the new point captured by the subsequent call to touchesMoved:withEvent:.

➤ Update the implementation of touchesMoved:withEvent:. There will be an error, and we will fix it soon.

```
override func touchesMoved(touches: Set<UITouch>,
                           withEvent event: UIEvent) {
      let touch = touches.first!
      let end = touch.locationInView(canvas)
      if let start = self.start {
            drawFromPoint(start, toPoint: end)
      }
      start = end
}
```

Swift **optional binding** is used to check the value of the start property, ensuring that a CGPoint value has been assigned, before passing it to drawFromPoint:toPoint:.

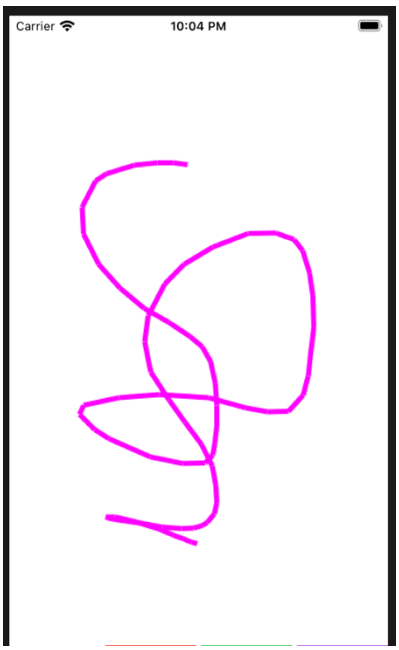
➤ Add an empty implementation of drawFromPoint:toPoint: to fix the error.

```
func drawFromPoint(_ start: CGPoint, toPoint end: CGPoint){
      // print coordinate here
      print (start, end)
}
```

Within the function, we will use the local argument name **end**. Outside the function, we will call the function with the external argument name **toPoint**. This makes the code more readable both outside the function and inside the function.

➤ Run the app (⌘ R), click and drag to simulate a moving touch, and observe the console output.

Part 5: Draw the line segments



Objective: Implement the drawing method with Core Graphics.

Similar to drawing a single line with Core Graphics in Lesson 1, one must follow a similar idiom when implementing drawFromPoint:toPoint:

a) Obtaining a drawing context,
b) Setting the stroke color and width,
c) Creating a path with points, and then
d) Stroking the path.

➤ Update the implementation of drawFromPoint:toPoint:. (For the "draw the new line segment" portion, I copied code from the old CanvasView.swift file we are no longer using.)

```
func drawFromPoint(start: CGPoint, toPoint end: CGPoint) {
      // print coordinates
      print (start, end)

      // set the context to that of an image
      UIGraphicsBeginImageContext(canvas.frame.size)
      let context = UIGraphicsGetCurrentContext()

      // draw the existing image onto the current context
      canvas.image?.draw(in: canvas.bounds)

      // draw the new line segment
      context?.setLineWidth(5)
      context?.setStrokeColor(UIColor.magenta.cgColor)
      context?.beginPath()
      context?.move(to: start)
      context?.addLine(to: end)
      context?.strokePath()

      // obtain a UIImage object from the context
      let newImage = UIGraphicsGetImageFromCurrentImageContext()
      UIGraphicsEndImageContext()

      // set image to the new image
      canvas.image = newImage
}
```

The call to UIGraphicsBeginImageContext establishes that all drawing operations will take place on an image, rather than the view itself. Subsequent drawing operations will occur to this "cached" image managed by Core Graphics.

The first drawing operation is calling the drawInRect: method on the UIImageView's current image, which tells it to "draw itself onto the current context." In other words, the lines already drawn on the UIImageView's image are first drawn on the new context, so that they do not disappear after subsequent calls to drawFromPoint:toPoint:.

The drawing functions setLineWidth, setStrokeColor, beginPath, move:To:Point, addLine: To:Point and strokePath do what they sound like they do. These are common kinds of drawing functions in all programming languages and platforms.

The results of the drawing operations are obtained as a UIImage object from the UIGraphics-GetImageFromCurrentImageContext function. The resulting image is then assigned to the image property of the controller's UIImageView canvas.

For every call to drawFromPoint:toPoint:,

a) A new UIImage is generated,
b) The existing UIImageView image is drawn onto the new UIImage,
c) The new line is drawn on the new UIImage, and then
d) The new UIImage is assigned to the UIImageView for continuous display.

➤ Run the app (⌘ R), click and drag the mouse on the screen to simulate a finger movement, and observe the line drawing.

Part 6: Add a clear button

Objective: How can we clear the image so we can start a new drawing? Enable a button that clears the image from the screen.

➤ **Embed** ( ) the UIImageView in a **Stack View**. Set the stack view to Axis Vertical, Alignment Fill, Distribution Fill, and Spacing 0.

➤ Using the Object Library ( ), add a button in the vertical stack view. Using the Attributes Inspector ( ), change the button text to "Clear". Using the Assistant Editor (^⌥⌘↩), Control-drag from the button to the ViewController class to create a controller action called clearImage:. Add code to delete the image.

```
@IBAction func clearImage(
            sender: UIButton) {
      canvas.image = nil
}
```

➤ Run the app (⌘ R), create a drawing, tap the Clear button, and observe the image disappear.

➤ Embed ( ) the Clear button in a **Horizontal Stack View**. Set Alignment Fill, Distribution Fill Equally, and Spacing 5. You can use this horizontal stack view to add more buttons for changing colors and line width.

Modifications and Extensions

- Add more buttons to change the color of the line. (see above image)
- Add more buttons to change the thickness of the line.
- Modify the drawing implementation such that the line changes color or thickness the longer the entire moving touch event is. You can do this with random numbers and/or an array of colors.