



POLITÉCNICA

ETSIT  
UPM

*dit*  
UPM

# Desarrollo de Apps para iOS UIViews

IWEB 2015-2016  
Santiago Pavón

ver: 2015.09.23

# UIView

# ¿Qué es una UIView?

- Clase base de la que derivan las views usadas para crear un GUI.
- El GUI es una jerarquía (árbol) de views.
- Propiedades para recorrer el árbol:

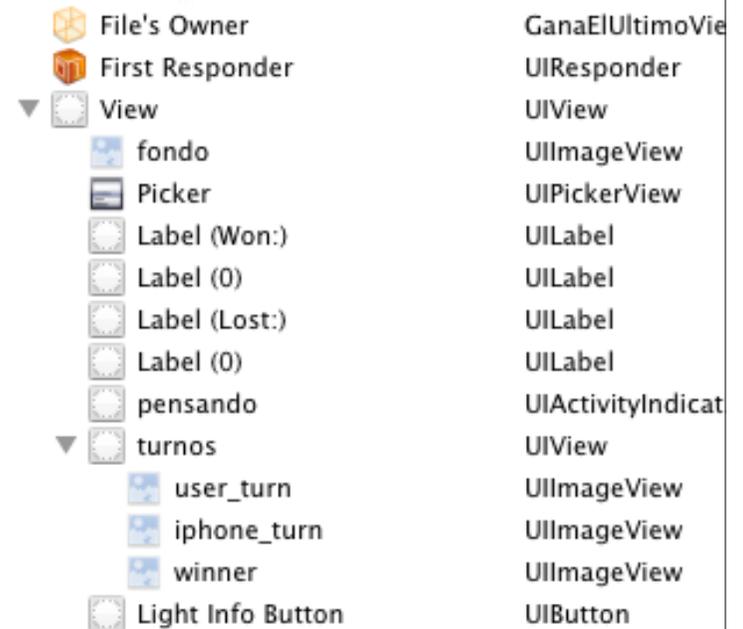
- La view padre:

**superview: UIView**

- Array con las views hijas:

**subviews: [AnyObject]**

- La raíz de la jerarquía de views es un objeto **UIWindow**.



# Construir el GUI

- La jerarquía de views puede construirse con el Interface Builder de Xcode.
- O programáticamente:

Inicializador designado de UIView.

```
init(frame: CGRect)
```

```
func addSubview(view: UIView)
```

```
func removeFromSuperview()
```

Crear jerarquía de views.

```
func insertSubview(view: UIView, atIndex: Int)
```

```
func insertSubview(view: UIView, belowSubview: UIView)
```

```
func insertSubview(view: UIView, aboveSubview: UIView)
```

```
let rect = CGRectMake(50,50,120,40);  
  
let l = UILabel(frame:rect)  
  
l.text = "Hola mundo"  
  
view.addSubview(l)
```

# El contenido de la View

- La view define un área rectangular en el que se puede dibujar.
- Si el contenido sobrepasa el rectángulo, se puede pintar o no (**clipToBounds**).
- Se puede controlar su nivel de transparencia (**alpha**).
- Puede ser opaca o no (**opaque**).
- La view entera se puede ocultar (**hidden**).
- Rehacer el contenido de una view cuando cambia su tamaño (**contentMode**):
  - reusar imagen cacheada con el contenido actual reajustando tamaño y posición,
  - repintar otra vez el contenido.
- Transformaciones afines (**transform**).
- Su color de fondo (**backgroundColor**).
- etc.
  - Ver la documentación

# Eventos

- Podemos gestionar los eventos que ocurren en la view.
- Se puede configurar para:
  - ignorar los eventos del usuario.
  - decidir si soportar multi-touch.
  - establecer reconocedores de gestos.
  - bloquear el envío de eventos a otras views.
  - etc.

# Gestión de Memoria

- Al añadir una view a la jerarquía se retiene (strong), aumentando su contador de retenciones.
  - No es necesario que nosotros retenemos la view con una nueva propiedad.
- Pero al sacar a view de la jerarquía se llama a release, es decir, se disminuye en uno el valor de su contador de retenciones.
  - Cuidado que el contador de retenciones podría llegar a cero en este momento.
    - Podemos evitarlo asignándolo previamente la view a una variable strong.

# Coordenadas

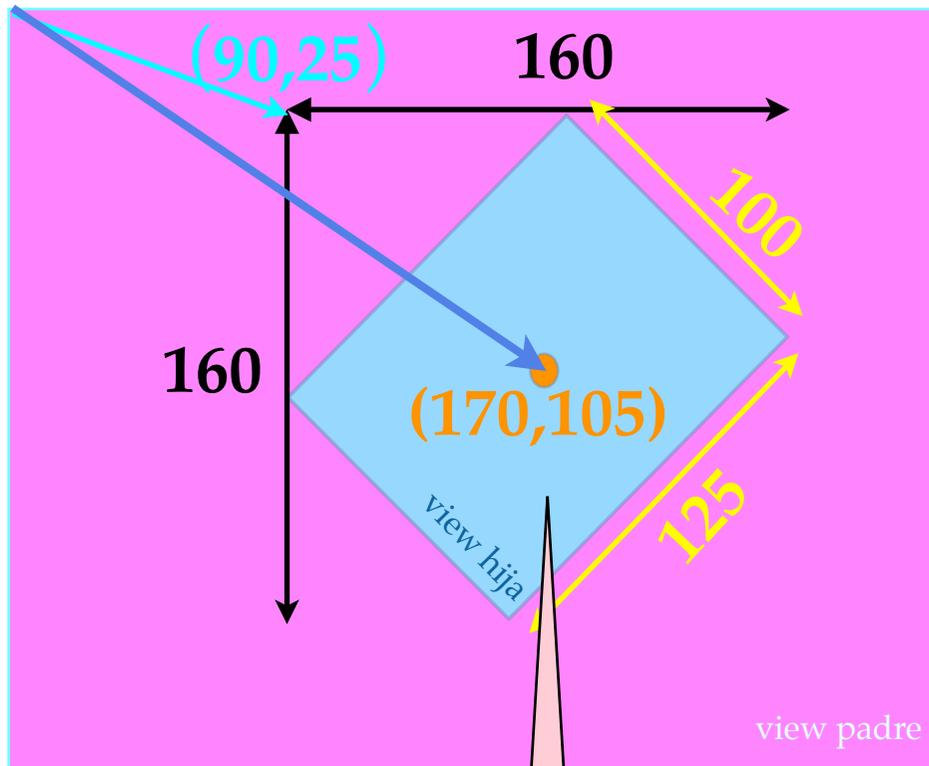
# Algunas Funciones y Tipos CG

- **CGFloat**
  - Es una struct que contiene un número real.
- **CGPoint**
  - Es una struct con las propiedades: **x:CGFloat**, **y:CGFloat**.
  - Representa un punto situado en (x,y).
- **CGSize**
  - Es una struct con las propiedades: **width:CGFloat**, **height:CGFloat**.
  - Representa un tamaño.
- **CGRect**
  - Es una struct con las propiedades: **origin:CGPoint**, **size:CGSize**.
  - Representa un rectángulo con el origen y tamaño indicados.
- Existen funciones muy cómodas para crear estas estructuras:

```
let p: CGPoint = CGPointMake(x,y)
let s: CGSize  = CGSizeMake(w,h)
let r: CGRect  = CGRectMake(x,y,w,h)
```

# Coordenadas

- El origen de coordenadas está situado arriba a la izquierda
- Las unidades son puntos, no píxeles.
  - Normalmente usamos puntos cuando dibujamos.
    - El dibujo se ajusta automáticamente a la mayor resolución.
      - la propiedad **contentScaleFactor** contiene número de pixels por punto de la `UIView`.
- Propiedades:
  - bounds**: `CGRect`
    - El origen y tamaño de la view en su sistema de coordenadas.
  - center**: `CGPoint`
    - El centro de la view en el sistema de coordenadas de la view padre.
  - frame**: `CGRect`
    - Rectángulo que contiene completamente a la view en el sistema de coordenadas de su view padre (la view hija puede estar girada).



View hija:

bounds =  $((0, 0), (100, 125))$

frame =  $(90, 25), (160, 160)$

center =  $(170, 105)$

# UIView: Dibujar

# Alternativas para Dibujar

- Podemos usar:
  - **Quartz 2D**
    - Pertenece al framework Core Graphics.
      - Consultar: *Quartz 2D Programming Guide*.
      - API con funciones en C (*pueden llamarse desde Swift*).
    - El framework UIKit proporciona varios envoltorios para facilitar su uso.
      - UIColor, UIFont, UIBezierPath, ...
  - **OpenGL ES**

# Dibujar con Quartz 2D

- Para hacer dibujos personales en la pantalla deberemos crear una subclase de **UIView**.

- Para dibujar hay que redefinir el método:

```
func drawRect(rect: CGRect)
```

- Incluyendo en él las sentencias de pintado.
  - El rectángulo pasado como argumento indica la zona que hay que repintar.
- Nunca llamaremos directamente a **drawRect**
    - Cuando haya que repintar el contenido de la UIView, llamaremos a uno de estos métodos:

```
func setNeedsDisplay()
```

```
func setNeedsDisplayInRect(rect: CGRect)
```

- Y el sistema llamará a `drawRect` en el siguiente ciclo de pintado.

# Path

- Un path es un dibujo creado con líneas rectas y curvas, arcos, elipses y rectángulos.
  - Puede ser abierto o cerrado, relleno o no, con líneas sólidas o discontinuas, etc.
- Como se usa:
  - Crear un path.
  - Establecer el punto inicial de pintado.
  - Añadimos tramos al path (líneas, arcos, ...).
    - Al final, podemos (opcional) cerrar el path.
  - Definir el color / tipo / patrón de las líneas, el color del relleno, zona de clip, etc...
  - Acabamos ordenando que se pinte el path creado.
    - En ese momento se pintan las líneas que lo forman, el relleno o ambos.
- Un path también puede guardarse para reusarlo varias veces.

# UIKit Envoltorios

- UIKit proporcionan métodos útiles para simplificar el código de pintado que hay que programar:

- Establecer el color de las líneas y de relleno:

```
UIColor.blueColor().set()
```

- Establecer el color de las líneas:

```
UIColor.greenColor().setStroke]()
```

- Establecer el color de relleno:

```
UIColor.redColor().setFill()
```

- Crear un color personalizado:

```
let c = UIColor(red:0.2, green:0.1, blue:0.8, alpha:1.0)
```

- Establecer el font:

```
let font1 = UIFont.systemFontOfSize(20)
```

```
let font2 = UIFont.systemFontSize()
```

```
let font3 = UIFont(name: "HelveticaNeue-Thin", size: 24)
```

- Dibujar un rectángulo, relleno o solo el contorno:

```
let rect = CGRectMake(10,10,100,100)
```

```
UIRectFill(rect)
```

```
UIRectFrame(rect)
```

- Dibujar un path (rectángulo redondeado):

```
let path = UIBezierPath(roundedRect:rect cornerRadius:20)
```

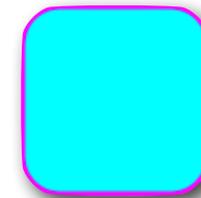
```
path.lineWidth = 3;
```

```
UIColor.magentaColor().setStroke()
```

```
UIColor.cyanColor.setFill()
```

```
path.fill()
```

```
path.stroke()
```



- Dibujar un path (una copa) con UIBezierPath:

```
let path = UIBezierPath()

path.moveToPoint(CGPointMake(10, 10))
path.addArcWithCenter(CGPointMake(90, 10),
    radius: 80,
    startAngle: CGFloat(M_PI),
    endAngle: CGFloat(M_PI_2),
    clockwise: false)

path.lineToPoint(CGPointMake(95, 180))
path.lineToPoint(CGPointMake(40, 200))
path.lineToPoint(CGPointMake(170, 200))
path.lineToPoint(CGPointMake(115, 180))
path.lineToPoint(CGPointMake(120, 90))
path.addArcWithCenter(CGPointMake(120, 10),
    radius: 80,
    startAngle: CGFloat(M_PI_2),
    endAngle: CGFloat(0),
    clockwise: false)

path.closePath()

path.lineWidth = 3
UIColor.magentaColor().setStroke()
UIColor.cyanColor().setFill()
path.fill()
path.stroke()
```



# Dibujar Texto

```
let text = "hola"
```

```
let attrs = [NSFontAttributeName: UIFont.systemFontOfSize(28)]
```

```
let pos = CGPointMake(10,10)
```

```
text.drawAtPoint(pos, withAttributes: attrs)
```

hola

```
let rect = CGRectMake(20,40,30,200)
```

```
text.drawInRect(rect, withAttributes: attrs)
```

h

o

a

- Es mejor usar un objeto UILabel.

# Dibujar Imágenes

```
if let img = UIImage(named: "fondo.png") {
```

Pintar la imagen en la posición dada.

Crear imagen usando fichero del bundle de la aplicación.

```
img.drawAtPoint(pos)
```

Reescalar imagen y pintarla en el rectángulo dado.

```
img.drawInRect(rect)
```

```
img.drawAsPatternInRect(rect)
```

Pintar la imagen repetidas (tiled) veces para llenar el rectángulo dado.

- Es mejor usar un objeto `UIImageView`.

# Cambios de tamaño

- Cuando cambia el tamaño de una view:
  - Si el valor de **contentMode** es **UIViewContentModeRedraw**, se redibuja su contenido otra vez.
  - Para otro valor de **contentMode** se repinta el contenido usando un bitmap cacheado que contiene la imagen actual.
    - Se reescala y se pinta el bitmap si el valor de **contentMode** es **UIViewContentModeScaleToFill**, **UIViewContentModeScaleAspectFit**.
    - O se dibuja el bitmap en la posición indicada si el valor de **contentMode** es **UIViewContentModeCenter**, **UIViewContentModeLeft**, **UIViewContentModeRight**, **UIViewContentModeTop**, **UIViewContentModeBottomLeft**, **UIViewContentModeBottomRight**, **UIViewContentModeTopLeft**, **UIViewContentModeTopRight**.

# Crear UIImages

- Usando un fichero en el main bundle de la aplicación:

```
let img = UIImage(named: "fondo.png")
```

- Usando el path de un fichero:

```
let paths = NSSearchPathForDirectoriesInDomains(  
    .DocumentDirectory, .UserDomainMask, true)  
  
let docs = paths[0] as String  
let filePath = docs.stringByAppendingPathComponent("mundo.jpg")  
let img = UIImage(contentsOfFile: filePath)
```

- Desde un buffer de bytes:

```
let url = NSURL(string: "http://www.dit.upm.es/~ssantiago/gif/fotos/  
spg073.jpg")  
let error = NSErrorPointer()  
let data = NSData.dataWithContentsOfURL(url, options: .allZeros, error: error)  
let img = UIImage(data: data!)
```

- Dibujándola en un contexto gráfico:

```
UIGraphicsBeginImageContext(CGSizeMake(200,200))  
// dibujar usando funciones CGContext o UIBezierPath  
let img = UIGraphicsGetImageFromCurrentImageContext()  
UIGraphicsEndImageContext()
```

# Crear JPEG y PNG

- El contenido de una UIImage puede guardarse en un NSData en formato JPEG o PNG.

```
func UIImageJPEGRepresentation(_ image: UIImage!,  
                               _ compressionQuality: CGFloat)  
    -> NSData!
```

```
func UIImagePNGRepresentation(_ image: UIImage!)  
    -> NSData!
```

# Contexto Gráfico

- El pintado se hace usando un contexto gráfico.
- Independiza del elemento sobre el que se pinta
  - pantalla, imagen, impresora, PDF, ...
- Se obtiene usando el método:

```
func UIGraphicsGetCurrentContext () -> CGContextRef!
```

- Cada vez que se llama a `drawRect` se crea automáticamente un nuevo contexto gráfico.
  - Ese contexto sólo es válido para la actual llamada a `drawRect`.
  - No se puede guardar para usarlo más tarde.

# Dibujar una Copa

```
override func drawRect(rect: CGRect) {  
    let context: CGContextRef = UIGraphicsGetCurrentContext()  
    CGContextBeginPath(context)  
    CGContextMoveToPoint(context, 10, 10)  
    CGContextAddArc(context , CGFloat(90), 10, 80, CGFloat(M_PI), CGFloat(M_PI_2), 1)  
    CGContextAddLineToPoint(context, 95, 180)  
    CGContextAddLineToPoint(context, 40, 200)  
    CGContextAddLineToPoint(context, 170, 200)  
    CGContextAddLineToPoint(context, 115, 180)  
    CGContextAddLineToPoint(context, 120, 90)  
    CGContextAddArc(context , 120, 10, 80, CGFloat(M_PI_2), 0, 1)  
    CGContextClosePath(context)  
    CGContextSetLineWidth(context, 3)  
    CGContextSetStrokeColorWithColor(context, UIColor.magentaColor().CGColor)  
    CGContextSetFillColorWithColor(context, UIColor.cyanColor().CGColor)  
    CGContextDrawPath(context, kCGPathFillStroke)  
}
```

# Dibujar una **L** Morada

```
CGContextRef context = UIGraphicsGetCurrentContext();
```

```
CGContextBeginPath(context);
```

```
CGContextSetLineWidth(context, 3);
```

```
CGContextSetStrokeColorWithColor(context,  
    [UIColor magentaColor].CGColor);
```

```
CGContextMoveToPoint(context, 10, 10);
```

```
CGContextAddLineToPoint(context, 10, 20);
```

```
CGContextAddLineToPoint(context, 20, 20);
```

```
CGContextDrawPath(context, kCGPathStroke);
```

# Push y Pop del Contexto Gráfico

- Si se realizan modificaciones al estado del contexto gráfico, pero no se quiere perder el estado actual:
  - puede guardarse el estado actual del contexto gráfico con:  
**CGContextSaveGState**(contexto)
  - realizar las modificaciones que se deseen en el contexto gráfico
  - y recuperar el estado inicial con:  
**CGContextRestoreGState**(contexto)
- Ejemplo de uso:
  - drawRect invoca un método auxiliar que cambia el estado del contexto para hacer ciertos dibujos.
  - Este método hace un push al principio y un pop al final para que los cambios del estado sean locales a él.

```
override func drawRect(rect: CGRect) {  
    drawAxis()  
    drawObject()  
}
```

Los cambios realizados por este método en el contexto gráfico no afectan a siguiente método

```
private func drawAxis() {  
  
    let context = UIGraphicsGetCurrentContext()  
  
    // Guardar el estado del contexto actual  
    CGContextSaveGState(context)  
  
    // Realizo cambios en el estado del contexto  
    CGContextSetLineWidth(context, 1)  
    CGContextSetStrokeColorWithColor(context,  
                                       UIColor.redColor().CGColor)  
  
    // Dibujar  
  
    // Recupero el estado inicial  
    CGContextRestoreGState(context)  
}  
  
private func drawObject() {  
    // . . .  
}
```

# Inicialización de UIView

# Inicialización

- Existen varios inicializadores de UIView y sus subclases:

## **init(frame: CGRect)**

- Es el inicializador que llamamos (*típicamente*) explícitamente nosotros al crear un objeto UIView programáticamente.
- Este inicializador lo redefiniremos en nuestras clases derivadas de UIView para añadir las sentencias de configuración que necesitemos para las nuevas vistas.

## **init(coder: NSCoder)**

- Es el inicializador que se llama al reconstruir un objeto serializado.
  - Es decir, es el inicializador que se llama al cargar las views desde un fichero storyboard o nib.
- Especificado en el protocolo NSCoder.
- **IMPORTANTE:** Cuando se usa este inicializador, no se invoca el otro.
- Si tenemos que realizar alguna configuración personalizada adicional, podemos hacerla en el método **awakeFromNib**.
  - Este método se invoca en todos los objetos cargados de un fichero NIB o storyboard.

- Patrón típico para no repetir el mismo código de configuración en el método **init(frame:)** y en el método **awakeFromNib:**

```
class MyView: UIView {  
  
    override init(frame: CGRect) {  
        super.init(frame: frame)  
        setup()  
    }  
  
    required init(coder aDecoder: NSCoder) {  
        super.init(coder: aDecoder)  
    }  
  
    override func awakeFromNib() {  
        super.awakeFromNib()  
        setup()  
    }  
  
    private func setup() {  
        // Configuración adicional.  
    }  
  
}
```

Así, el método **setup** se ejecutará siempre, independientemente de como creamos el objeto **UIView**

# Threads

# Main Thread

- La manipulación de la interface de usuario debe hacerse sólo desde el main thread de la aplicación.
- Los métodos de UIView (y de otras clases de UIKit) sólo pueden invocarse desde el main thread.
- Únicamente la creación de los objetos UIView puede hacerse desde otro thread.

# Visualización en Vivo de la UIView en Storyboard

# Visualizar UIView

- Para visualizar en vivo el contenido de una UIView personalizada en el storyboard, marcar la clase con **@IBDesignable**.
- Los atributos que se deseen ajustar desde el inspector de atributos deben marcarse con **@IBInspectable**.
- Si es necesario realizar una configuración especial para que la visualización en el IB sea la adecuada, sobrescribir el método **prepareForInterfaceBuilder**.
- También puede incluirse código condicional para que solo se ejecute cuando el IB está presentado la view. Usando la directiva:

```
#if TARGET_INTERFACE_BUILDER
#else
#endif
```

```
@IBDesignable
class TrajectoryView: UIView {

    @IBInspectable
    var lineWidth : Double = 3.0

    @IBInspectable
    var trajectoryColor : UIColor = UIColor.redColor()

    @IBInspectable
    var targetImage : UIImage!

    @IBInspectable
    var birdImage : UIImage!

    // . . .
    override func prepareForInterfaceBuilder() {
        dataSource = FakeDataSource()
    }

    // . . .
```

