Dibujando 2D en iOS (Nota técnica)

Drowing 2D in iOS (Technical note)

Franklin Hernández-Castro[1]
Jorge Monge-Fallas[2]

Fecha de recepción: 22 de octubre del 2013
Fecha de aprobación: 19 de enero del 2013

Hernández-Castro, F; Monge-Fallas, J. Dibujando 2D en iOS (Nota técnica). *Tecnología en Marcha*. VI Encuentro de Investigación y Extensión. Pág 96-101.

1 eScience Group. Instituto Tecnológico de Costa Rica. Costa Rica. Correo electrónico: franhernandez@itcr.ac.cr.

2 eScience Group. Instituto Tecnológico de Costa Rica. Costa Rica. Correo electrónico: jomonge@itcr.ac.cr.

Palabras claves

iOS; Objetive-C; Core Graphics; Quartz 2D; UIView; DrawRect.

Key words

iOS; Objetive-C; Core Graphics; Quartz 2D; UIView; DrawRect.

Resumen

El objetivo de este artículo es explicar los conceptos básicos necesarios para dibujar en dos dimensiones en iOS. En otras palabras, para hacer dibujos en dos dimensiones para los dispositivos iPhone y iPads en el lenguaje de programación Objective-C.

Este ambiente ofrece varias alternativas con tal objetivo, y en este artículo se tratan las generalidades necesarias para entender cuáles de ellas se usan en cada caso. Además, ofrece un par de ejemplos de cómo se implementan algunas de estas posibilidades.

Abstract

This paper explains basic concepts that are needed to draw in two dimensions in iOS, in other words, to drawing in two dimensions for iPhone and iPads in the programming language Objective-C.

This environment offers several alternatives for this purpose and this article covers the generalities necessary to understand which of them are used in each case. It also offers a couple of examples of how to implement some of these possibilities.

Introducción

El ambiente de programación definido para iPhone o iPad es un ambiente complejo que funciona estrictamente en forma orientada a objetos. Esta característica hace que no sea del todo obvio cómo se hacen ciertas operaciones que en otros ambientes se consideran claras, por mostrar un flujo de control más obvio.

Para poder dibujar en este ambiente, es necesario conocer un par de conceptos previos y haber completado ya una estructura mínima.

Las aplicaciones en iOS se basan en un modelo de programación llamado *Model-View-Controller* (MVC [1]) que define que cada escena (técnicamente llamada *View*) alberga un área donde se colocan los elementos de la interface (textos, botones, etc.), en estas "vistas" o escenarios también se puede dibujar.

De acuerdo con el paradigma de programación orientado a objetos que siguen este ambiente, todos los elementos son objetos de la misma clase *UIView*, hijos de la vista principal, y cada uno con un objetivo específico como albergar un botón o mostrar un texto.

De este modo. podemos dibujar en el fondo de la vista o escena, en la raíz de la jerarquía de la aplicación o crear una vista aparte a modo de pantalla o pizarra, donde hacer nuestros dibujos.

Antes de entrar en detalle de cómo se hacen estos dibujos, es necesario comprender algunos conceptos involucrados en este proceso.

Frameworks, usos y posibilidades

Para dibujar en iOS existen varias posibilidades, en cualquiera de ellas se usan librerías o bibliotecas definidas para ese objetivo. Se puede dividir estas bibliotecas en internas o externas; es decir, en bibliotecas propias de Objetive-C, o bibliotecas generadas por terceros y que se pueden usar al interior de este ambiente.

Con las bibliotecas nativas (propias del ambiente) solo es posible dibujar en dos dimensiones, para dibujar en tres dimensiones generalmente se usa OpenGL-ES que es una versión del OpenGL estándar, adaptada para este ambiente. De hecho, todos los juegos que corren en tres dimensiones en iPhone y iPad usan está única biblioteca.

1 Para más información sobre estructuras básicas ver artículo sobre Flujos de Control, consulte a (Hernández-Castro & Monge, 2011).

En este artículo, hablaremos solamente de las bibliotecas nativas (solo para 2D) que se usan para dibujar. Se puede decir que en este ambiente hay tres bibliotecas principales que se usan con dicho fin[1]:

- UIKit
- Core Animation
- Core Graphics

UIKit

UIKit (abreviatura de *User Interface Kit*) es la biblioteca que permite crear y manipular las vistas de las que hemos hablado, es decir, los elementos que contienen las interfaces como botones, imágenes, *sliders*, etc.

En realidad UIKit es, a su vez, una interface de Quartz 2D para el ambiente de portátiles (iPhone y iPad). Quartz 2D por su parte, es la biblioteca que corre en OSX (el sistema operativo de los ordenadores de apple®) con este mismo fin de manejar los elementos de las interfaces. Este hecho garantiza algo de compatibilidad entre el código de los portátiles (iPhone y iPad) y los de escritorio (laptops y torres).

Con UIKit se crean, borran, mueven y manipulan las mismas vistas o contenedores (*views*) a través de la clase llamada *UIView* (UI vienen de UIKit). La aplicación misma corre en una *UIView* principal o raíz, en la que se agregan "a manera de hijos" otras *views* que contienen el resto de los elementos. Al interior de algunas de éstas vistas se puede dibujar o simplemente cargar una imagen o dibujo vectorial (generalmente en .svg [2]).

Core Animation

Esta es una biblioteca orientada a la animación de las diferentes *views* que, como se ha dicho, son los contenedores en los cuales se "dibujan" los distintos objetos.

En estos casos, Core Animation se encarga de hacer animaciones entre estos diferentes "contenedores". Por ejemplo, si se desea que un botón cambié de lugar al presionarlo, el mejor modo de hacerlo es usando una animación ya definida en esta biblioteca.

Claro que ésta misma tecnología se puede usar para mover un dibujo; en este caso el dibujo se puede cargar en una *view* (como imagen o como vector) y hacer que esta se mueva a través de la pantalla con funciones definidas en Core Animation, lo que sería bastante eficiente pues las funciones de esta biblioteca son de bajo nivel. Este enfoque es muy usado en juegos, donde algún dibujo (como un animal o "alien") se mueve a través de la pantalla con algún fin.

Core Graphics

Nuestra tercera biblioteca es Core Graphics, que es la biblioteca encargada finalmente de hacer eso: ¡dibujar!

Es decir, después de tener una estructura de vistas, botones y demás, cuando finalmente se desea dibujar un "triángulo verde" en la pantalla (por ejemplo), la encargada de hacer eso es la biblioteca Core Graphics. Para lo cual se vale de funciones específicas que veremos en las siguientes secciones.

UIView y DrawRect

Como se dijo anteriormente, la *UIView* es la clase que alberga todos los elementos necesarios para las aplicaciones y, por supuesto, los dibujos realizados en tiempo real.

Cuando se crea un controlador esta clase ya viene asociada al mismo y por tanto no aparece en primera instancia en el conjunto de archivos de la aplicación. En realidad, la vista o *UIView* es una propiedad de cada controlador, y como tal existe desde que el controlador fue creado (generalmente por la plantilla[3]).

Sin embargo, como nosotros deseamos dibujar en ella, necesitamos tenerla explícitamente, de este modo podemos escribir nuestras instrucciones de dibujo. Para lograr este objetivo, lo que generalmente se hace es crear aparte un objeto de la clase *UIView* y, luego, asociarlo al controlador. Dicho de otro modo, creamos un objeto *UIView* y lo declaramos como la propiedad del controlador en el que estamos trabajando.

1 Nahavandipoor, N. (2011)Graphics and Animation on iOS. Sebastopol, CA:O'Reilly Media, Inc.

2 .sgv abreviatura de Scalar Vector Graphics, dibujos en vectores que son interpretados y dibujados automáticamente por objetive-c.

3 La mayoría de las veces los proyectos en iOS empiezan con una estructura que se crea a partir de plantillas (templates) que se optimizan para cada uso como juegos o tablas, etc.

Una vez con el objeto creado y asociado, podemos ver que en su interior hay un método de nombre *drawRect ( )*, (Apple Inc., 2011)es en este método donde se escriben las instrucciones para dibujar. Es decir, *drawRect ( )* es la función o método que será llamado para hacer los dibujos necesarios.

A menudo sucede que este método ni siquiera está implementado porque en muchas escenas (o controladores) no se dibuja, y estas escenas están ahí para albergar botones, tablas o para mostrar alguna imagen o url. De hecho, de no necesitarlo, no se debe implementar el método *drawRect ( )* pues, como es una función de dibujo, ocupa muchos recursos, y si no está implementado el sistema automáticamente ni siquiera lo llamará

En caso de ser implementado el método *drawRect ( )* va a ser llamado por el sistema cuando haya momentos de desuso del procesador o cuando suceda alguna interacción de parte del usuario, tocar un botón o cambiar de vista, por ejemplo.

También, es posible llamar este método con el fin de refrescar el dibujo, esto se hace a través de las instrucciones *setNeedsDisplay ( )* o *setNeedsDisplayInRect ( )*, según se desee refrescar total o parcialmente la vista. Esto es útil cuando se tiene una animación. Por ejemplo, en casos de animaciones como en los juegos, se crea un *timer* (Hernández-Castro & Monge, 2011) que llama a este método *drawRect ( )* cada 1/24 de segundo para mantener la fluidez del movimiento.

Contexto gráfico

Debido a que se puede dibujar en cualquier objeto del tipo *UIView*, un objeto puede dibujar en otro. Por ejemplo, podría suceder que un objeto controlador que calcula una gráfica específica dibuje sus resultados en otro objeto (o en varios) que usa solo de pantalla de salida de datos (*output*) a manera de pizarra. Por esta razón, es necesario definir un "contexto de dibujo" para ejecutar la mayoría de las instrucciones de este tipo.

Es decir, un contexto de dibujo, es un lugar donde dibujar o la definición del mismo. Además de dónde dibujar, un contexto gráfico también guarda parámetros como el color de las líneas y del relleno de las formas, el grosor de las líneas, las transparencias al dibujar, etc.

Los contextos de dibujo son creados automáticamente por Quartz 2D y se pueden leer con la función: *UIGraphicsGetCurrentContext ( )*. De este modo pueden ser especificados como parámetros al dibujar algo, así la función sabe dónde se desea dibujar y con qué características.

Debido a que el contexto puede definir y guardar estas características de dibujo, también se usa para definir una serie específica de condiciones con las que se desea dibujar algo y, luego, regresar a otras. Por ejemplo, si se desea llamar a una función que dibuja un cuadrado rojo con perímetro gris, se podría antes de definir estas características guardar el contexto de dibujo en el que está, después cambiar las características, dibujar el cuadrado con perímetro gris y, luego, volver al contexto guardado de modo de que no se vean afectadas otras partes del dibujo con las condiciones del cuadrado. Esto es posible con instrucciones como *CGContextSaveGState ( )* y *CGContextRestoreGState ( )*.

Colores en iOS

Dentro de este mismo concepto de contexto gráfico es posible, por supuesto, definir los colores con los que se desea dibujar. Para esto existen los espacio cromáticos RGBA, CMYK y gray scale, es decir, rojo-verde-azul-alfa, cian-magenta-amarillo-negro y escala de grises. Todos los espacios se define en ámbitos [0-1] y no en el habitual [0-255] que se usa en casi todos los ambientes gráficos. Si no se define el espacio cromático el mismo por omisión será RGBA.

Una vez definido el espacio de color se pasa a la creación del objeto mismo que identificará el color. Como trabajamos estrictamente orientado a objetos el color es también un objeto, en este caso de la clase *UIColor*, parte de la biblioteca *UIKit*.

La sintaxis involucrada se vería así:

```
UIColor *miColor = [UIColor colorWithRed:0.3f green:0.5f blue:0.6f alpha:1.0f ];
```

Por supuesto, una vez definido el color se debe declarar como color de fondo o de perímetro de las formas que se dibujarán inmediatamente después, esto se hace a través de funciones de la biblioteca Core Graphics:

```
CGContextSetFillColorWithColor(unContexto,miColor);
```

```
CGContextSetStrokeColorWithColor(unContexto
,miColor);
```

Path y polígonos

Finalmente, llegamos a las funciones donde se dibujan las cosas; por ejemplo, los polígonos. En este caso hay funciones pre-establecidas en Core Graphics para dibujar polígonos específicos. Empezamos con un rectángulo, en realidad hay un objeto ya definido en Core Graphics (CG) para este fin, su nombre es CGRect[4] y se produce por una función específica para este efecto *CGRectMake*:

```
CGRect rectangle = CGRectMake(60,170,200,80);
```

Sin embargo, esto no sería suficiente pues se debe primero definir el color, y cómo será usado (como hemos visto en la sección anterior) para, luego, definir el rectángulo (como arriba) y después dibujarlo, que en este ambiente en realidad se trata de agregarlo al contexto gráfico que le corresponda:

```
CGContextAddRect(unContexto, rectangle);
```

Por supuesto, también es posible dibujar a través de puntos. En este caso los puntos se concatenan en un recorrido (*path*) y este recorrido, luego, se añade al contexto.

Para dibujar de este modo hay dos instrucciones principales, una para moverse a algún punto deseado sin dibujar y otra para dibujar entre el punto en que se está y el que se define:

```
CGContextMoveToPoint(unContexto, 100, 100); y
CGContextAddLineToPoint(unContexto, 150, 150);
```

Con estas dos funciones, es posible dibujar cualquier forma con líneas rectas.

Para dibujar formas curvas, se hace desde las primitivas definidas con este fin como círculo y arco, hasta el uso de puntos de control tipo Bézier.

La clase *UIBeziertPath* ofrece la posibilidad de dibujar curvas Bézier cuadráticas y cúbicas, con sus respectivos puntos de control, la figura muestra cómo se definen los puntos:

La sintaxis involucrada se vería como algo así en el caso de la cuadrática:

```
CGContextMoveToPoint (unContexto, 10, 500);
CGContextAddQuadCurveToPoint (unContexto,
150, 10, 300, 500);
```

donde el orden de los parámetros sería el punto de inicio (10,500), el punto de control (150,10) y el punto final (300,500).

El caso de la cúbica se vería así:

```
CGContextMoveToPoint (unContexto, 10, 10);
CGContextAddCurveToPoint (unContexto, 0, 50,
300, 250, 300, 400);
```

donde el punto inicial es (10,10), los puntos de control primero y segundo serían (0,50) y (300,250), respectivamente, y el punto final (300, 400).

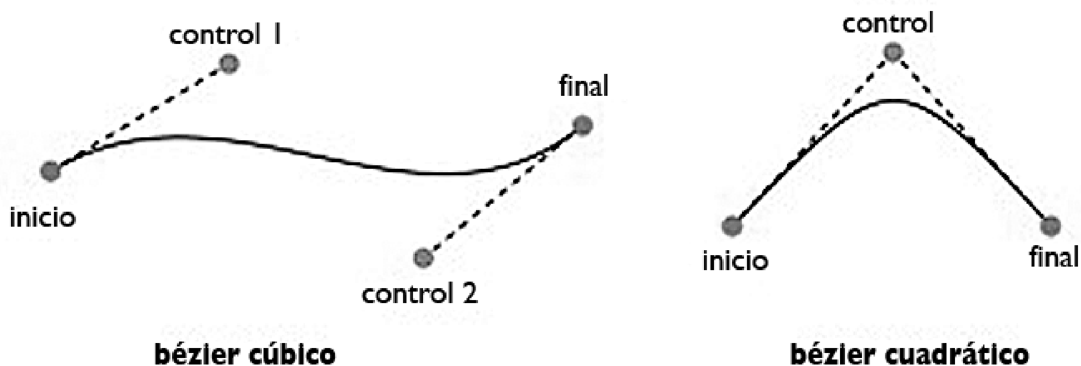

Figura 1. Curvas que se pueden dibujar con la clase UIBeziertPath. (Apple Inc., 2011).

4 El prefijo CG en estas clases viene por supuesto del framework Core Grapchis.

Conclusión

Como se ve, existen algunos conceptos necesarios para dibujar en un ambiente orientado a objetos tan elaborado como Objective-C que no son tan obvios como en otros contextos.

En primera instancia se debe tener clara la estructura de la aplicación. Cada elemento es albergado por un objeto, muchos de ellos son vistas (clase *UIView*). Las vistas principales que contienen los demás objetos (botones, textos, etc.) tiene a su vez controladores de la clase (*UIViweController*) que controlan cuándo, dónde y cómo aparecen esas vistas o escenas en la aplicación.

Una vez que está claro en cuál vista se desea dibujar, se debe declarar esta como propiedad de uno de esos controladores, o crear una subvista a manera de "pizarra" en la que ese controlador dibujará. En estos casos, la subvista será incluida como otro "hijo" de la vista principal o raíz que posee el controlador.

Con esto definido, se sabe en cuál de las vistas o *views* se debe implementar la función *drawRect ( )*, que es la función en la que se realizan finalmente los dibujos. Esta función es llamada por el sistema periódicamente o puede ser "refrescada en código" cuando se desee.

Una vez en el interior de la función, se deben preparar los elementos necesarios para dibujar. El contexto de dibujo se define primero, y con él sus características , así mismo se crean los objetos que albergan los colores (objetos de la clase *UIColor*) y se declaran como los que se desea usar para el relleno o el perímetro.

Ahora finalmente se crean los puntos, polígonos o recorridos (*paths*) que se desean dibujar; luego se añaden al contexto y, finalmente, se despliegan en él.

Dibujar, como cualquier tarea, en estos ambientes es algo complejo. La ventaja es que, cuando se tienen todos los elementos listos y modulares, su uso y reuso, así como su mantenimiento, cambio, actualización y re-utilización en otros ambientes o contextos es muy fácil. El código resultante es, entonces, muy modular y fácil de reutilizar.

Bibliografía

Apple Inc. (2011). *Drawing and Printing Guide for iOS: Graphics & Animation: 2D Drawing*. Cupertino, CA.

Hernández-Castro, F. Monge, J. (2011). *Flujo de control en iOS. Tecnología en Marcha*. Volumen 25, No. 5. Editorial Tecnológica de Costa Rica.