Realidad aumentada basada en geolocalización

– Memoria –

Autor: Oscar Iglesias López (oscar.iglesias.lopez@est.fib.upc.edu)
Director: Àlvar Vinacua i Pla (alvar@lsi.upc.edu)

Junio 2014

Querría agradecer a Àlvar Vinacua i Pla por la confianza que han depositado en mi al dirigirme el proyecto y por despertar en mi la pasión por la computación gráfica.

También mi infinito agradecimiento a los profesores: Albert Rubio (LSI), Andrea Zamora (MA2), Guillem Godoy (LSI), Isabel Navazo (LSI), Llorenç Rosselló (MA2) y Quim Trullas (DFEN) porque cada uno de ellos me enseñaron grandes cosas y me ayudaron mucho durante mi paso por la FIB.

Por último y no menos importante, a Ione (mi novia) por haber estado durante toda la carrera animándome en los momento difíciles. A mis padres y mi hermano por ser la mejor familia que se puede tener.

Resumen

Uno de los mayores problemas a los que se enfrenta un turista cuando vista una ciudad es el poder identificar que es lo que esta viendo y conocer más detalles sobre ese punto turístico. Si se sitúa delante de una catedral y no sabe como se llama, deberá recurrir a una guía turística o simplemente preguntar. Pero no siempre es fácil preguntar a los demás transeúntes si no se conoce el idioma local o no siempre se dispone de una guía turística.

Este proyecto quiere proponer una nueva experiencia para los turistas mediante una app para iPhone. Esta app, gracias a la realidad aumentada y a los sensores (GPS, magnetómetro e inclinómetro) de los que disponen un iPhone, podemos identificar que es lo que se encuentra dentro del rango de visión del usuario y mostrarle su nombre. Además, si el usuario lo desea, podrá ampliar la información del punto que desee.

Índice

1. Gestión de proyecto

1.1. Alcance del proyecto

Este proyecto quiere proponer una nueva herramienta turística haciendo uso de la realidad aumentada en los smartphones. Haciendo uso de los sistemas de localización (GPS y brújula) y de la cámara que disponen los smartphones, podemos hacer una app que muestre la información disponible en la red sobre los puntos de interés que tenemos ante nosotros. Adicionalmente, también podemos buscar todos los puntos de interés que tenemos al rededor nuestro y obtener información detallada (i.e teléfonos de contacto, web, fotos, comentarios, etc.) sobre ellos.

Forma parte de este proyecto, cuidar la usabilidad de esta app y maximizar la experiencia del usuario. Para ello deberemos dar respuesta en tiempo real y plantear una interface intuitiva y simple. Queda fuera del alcance de este proyecto, diseñar assets a medida y modelar las vistas desde el punto de vista de un diseñador.

1.2. Estado del arte

Actualmente no hay ninguna app de turismo que cumpla con los objetivos que buscamos. Lo más parecido que hay en la app store (tienda de APPs para iPhone, iPad y iPod) es una app de TMB (TMB Virtual que actualmente a sido retirada de la app store). Esta app lo que hace es muy parecido a lo que buscamos solo que los únicos puntos de interés que nos muestra son de transportes metropolitanos de Barcelona. No es exactamente lo que queremos hacer pero si que podemos coger la idea y analizar los puntos fuertes y las debilidades de esta app. En concreto esta app tiene muy mala aceptación por los usuarios porque resulta un poco confuso su uso y es poco intuitiva.

Para tratar el punto de la realidad aumentada si que existen varias herramientas (unas de pago y otras opensource) que nos facilitara mucho esta tarea. En concreto, la más usada es ARtookit. ARtoolkit es una librería bajo la licencia opensource disponible en varias plataformas (Windows, mac, linux, iPhone, Android, etc.).

Sobre la búsqueda y obtención de información de los puntos turísticos hay varias opciones. La más razonable en usar una API de google que facilita información básica sobre puntos de interés dadas unas coordenadas. Este servicio se llama *Google Places*.

Para poder implementar la app hay que recurrir al manual del SDK oficial de iOS. En el encontraremos como trabajar con el GPS y la brújula. Es un manual muy extenso y en constante evolución.

1.3. Metodologia

Para la realización de este proyecto he usado parcialmente la metodología de trabajo SCRUM. Dado que este proyecto lo realizaba yo solo con la ayuda de mi director no tenia ningún sentido aplicar SCRUM en todo su esplendor. Es por eso que la única parte que hemos aplicado de SCRUM han sido: los sprints, las demos y las retrospectivas. He hecho sprints de una semana. Cada semana de sprint iba de jueves a jueves ya que los jueves eran los días que por motivos de agenda de mi director y mía, podíamos quedar. Los jueves, al final del sprint, realizaba la demo con mi director de proyecto y también hacíamos la retrospectiva viendo todo el trabajo hecho y el que faltaba hacer.

Otra metodología de trabajo que no contemple al principio pero que he incluido a las pocas semanas de empezar el desarrollo de la app es *GIT Flow*. La metodología de trabajo *GIT Flow* nos establece una rama master que contendrá siempre una versión completamente funcional y (...en un principio...) libre de bugs, otra rama de develop de la que iremos haciendo más ramas, una por funcionalidad. El hecho de tener una rama más siempre limpia permite que en cualquier momento, cualquier persona con acceso al repositorio, pueda clonar esta rama y tener la app con una versión lo más avanzada posible y lo más 'limpia' posible.

1.4. Planificación temporal

Figura 1: Planificación temporal

1.5. Estudio de costes

Para realizar el presupuesto, hay que contemplar las siguientes principales partidas.

1.5.1. Coste del capital humano

Esta es la partida más importante en el presupuesto ya que el desarrollo de este proyecto se basa en conocimiento especifico de personal cualificado. Por lo tanto, la persona (o personas) que desarrolle este proyecto a de ser personal cualificado. Dada la naturaleza del proyecto, hay muchas tareas que no son paralelizables y es mucho mejor que sea desarrollado por un solo ingeniero. Además, si contratáramos a dos ingenieros seria más costoso ya que tendríamos más gastos de formación en el proyecto y lo que es peor, la organización se complicaría.

1.5.2. Costes de maquinaria

Para poder realizar este proyecto necesitamos un ordenador Mac y un iPhone. Comprar el iPhone es imprescindible ya que la app que desarrollaremos solo funciona sobre este dispositivo. Es cierto que podemos usar un simulador de iPhone por software pero como esta app usa geolocalización, la cámara y demás sensores reales, el simulador no sirve.

Comprar el ordenador mac también es inevitable. Para poder probar la app en el iPhone (en el proceso de desarrollo y testing) hay que hacerlo usando el entorno de desarrollo Xcode de apple que solo funciona en el sistema operativo Mac OS X. A su vez, Mac OS X solo puede funcionar sobre los ordenadores de Apple con arquitectura x86[1]. Como equipo compraremos un MacBook Pro ya que este equipo cumple con las necesidades del proyecto y además al ser portátil podemos llevarlo a la calle para poder hacer pruebas al aire libre.

1.5.3. Costes de software y licencias

Como IDE de desarrollo usaremos el oficial de Apple, el Xcode que también es gratuito. Si necesitamos añadir algún *asset* (recurso gráfico) a la app, buscaremos alguno que sea de libre uso para no tener que pagar royalties.

[1]La anterior arquitectura PowerPC que usaba Apple tambien a quedado obsoleta para el desarrollo en iOS

Si que necesitaremos comprar la licencia de desarrollo para iOS para poder hacer pruebas en el iPhone cuando estemos desarrollando la app. Además, para poder publicar la app una vez finalizada en la app store hay que mantener la licencia vigente. Dicha licencia se adquiere directamente en el portal de desarrollo de apple y tiene una vigencia anual con renovaciones anuales.

Dado que mi director de proyecto también tiene un iPhone, existía la necesidad de poder distribuirle una versión cada vez que finalizaba una feature. TestFlight[2] es una plataforma gratuita de distribución de apps. Con esta plataforma, cada vez que acababa una feature, generaba un binario de la app (`.ipa`) y la subia a la plataforma testflight indicando que personas podían instalarsela. Acto seguido, los usuarios con permiso para instalarse esta app, recibían un email indicándoles que tenían disponible una nueva versión lista para instalar. Una vez se instalaban la app, si esta producía un crash o cualquier excepción en alguno de los dispositivos de los usuarios, esta anomalía quedaba registrada junto con el estado de la pila en testflight para que posteriormente pudiera analizar por que se a producido el problema.

Todo proyecto, sea grande o pequeño a de tener algún tipo de mecanismo que nos permita poner a buen recaudo los avances en el. Para poder tener un control de versiones he decidido usar el sistema GIT. Son mucho los motivos por los cuales prefiero GIT frente a *subversion*, *mercurial* o *sourcesafe* pero tal vez el motivo que más destaca sobre ellos es la posibilidad de poder hacer commits en local aún cuando no tenemos conexión o simplemente no tenemos un repositorio externo. Otra de las grandes ventajas de GIT es la facilidad de crear ramas y fusionarlas después (muy importante para git flow).

1.5.4. Estimación de los costes

Concepto	Precio (€)	Cantidad	Subtotal (€)
Sueldo ingeniero informático	50	560*	28.000
MacBook Pro de 13”	1300	1	1300
iPhone 5S de 16GB	699	1	699
Licencia de desarrollo iOS	79	1	79
		Total	**30078 €**

*El proyecto esta pensado para ser desarrollado en unas 14 semanas a razón de 40 horas a la semana.

[2] `www.testflightapp.com`

1.5.5. Viabilidad económica

Barcelona esta entre las 10 ciudades más turísticas del mundo y es por eso que este sector mueve mucho dinero en Barcelona. Una app en varios idiomas que ayude, informe y mejore la experiencia del turista será muy fácil venderla.

En un principio se ha planteado poner la app a la venta en la app store a 2.69€ de los cuales, nosotros nos llevamos 1.64€. Así pues, con que la compren 18340 personas, ya cubrimos gastos y a partir de aquí todo serán beneficios. Puede parecer desorbitado plantear que 18340 personas compren la app pero para nada lo es y tenemos muchos ejemplos de ello en la app store. También cabe destacar, que se puede modificar el precio de la app en cualquier momento y así poder ajustar el precio de esta en función de la demanda de mercado o hacer campañas de oferta en temporada baja de turismo.

1.6. Sostenibilidad y compromiso social

Dada la naturaleza del proyecto, este no tendrá un gran impacto medioambiental. Lo que más se consumirá es energía eléctrica. Podemos contratar el abastecimiento eléctrico a una compañía que nos certifique que la totalidad (o parte) del suministro, proviene de fuentes de energía renovable.

También podemos tener en cuenta que Apple certifica que todos sus equipos se han construido aplicando los materiales menos dañinos y así contribuir a la conservación del planeta. A largo plazo, cuando el ordenador o el iPhone queden obsoletos, hay que recordar de llevarlo a un punto especializado en el tratamiento de residuos ya que ambos dispositivos contiene baterías de polímeros de litio (Li-Ion) que son altamente contaminantes y explosivas si no se tratan adecuadamente.

2. Implementación

2.1. Visión general

Esta aplicación se compone de dos modos de vista: el modo mapa (en ocasiones lo referenciamos como mapMode o simplemente Map) y el modo de realidad aumentada (ARMode o simplemente AR). Podemos alternar entre un modo u otro mediante la `TabBar` que siempre estará situada al pie de estos dos modos. Por defecto, cuando se inicia la app, se presenta el mapMode porque es el modo que da más información de donde nos encontramos sobre el mapa.

El mapMode lo veremos con más profundidad en la sección **??** pero básicamente lo que mostrará es el punto en el que se encuentra el usuario, y los POIs que tiene a su alrededor en un perímetro delimitado por una circunferencia de 1Km de radio.

El ARMode (detallado en la sección 2.7) es donde se produce la "magia"de este proyecto. Aquí podremos ver gracias a la realidad aumentada, los *places* (sinónimo de POI) que tenemos en nuestra dirección de visión.

Por último, en la sección 2.8, veremos como se muestra el detalle de un POI que ha seleccionado el usuario.

Pero antes de ver como se implementa el mapMode, ARMode y el detalle del POI, es imprescindible hacer una breve descripción del lenguaje de programación que vamos a usar para este proyecto, **Objective-C** y los patrones de diseño de software más usados en este proyecto (y prácticamente en el 100 % de apps para iOS). Quiero remarcar que la buena implementación de estos patrones de diseño son casi la mitad del éxito de una app en cuanto a fiabilidad se refiere. También daremos un breve repaso a la api sobre la que se apoya este proyecto para obtener datos (*Google Places*) y la librería de realidad aumentada (*ARToolKit*) que nos facilitará mucho el trabajo para implementar el ARMode.

2.2. Objective-C

Para construir una app para iOS se puede hacer de dos maneras: de forma nativa o usando frameworks de desarrollo multi-plataforma en HTML5 (i.e phoneGap, titanium, etc.). Esta segunda manera tiene la ventaja de que un mismo código puede ser exportado (con muy pocas modificaciones para cada plataforma) para que corra sobre varios sistemas diferentes (Windows phone, Android y iOS). Este sistema tiene la ventaja de la alta velocidad de desarrollo pero también esta muy limitada en cuanto a poder aprovechar todo el potencial que puede ofrecer cada plataforma. El resultado suelen ser app con un aspecto muy sencillo, carentes de animaciones

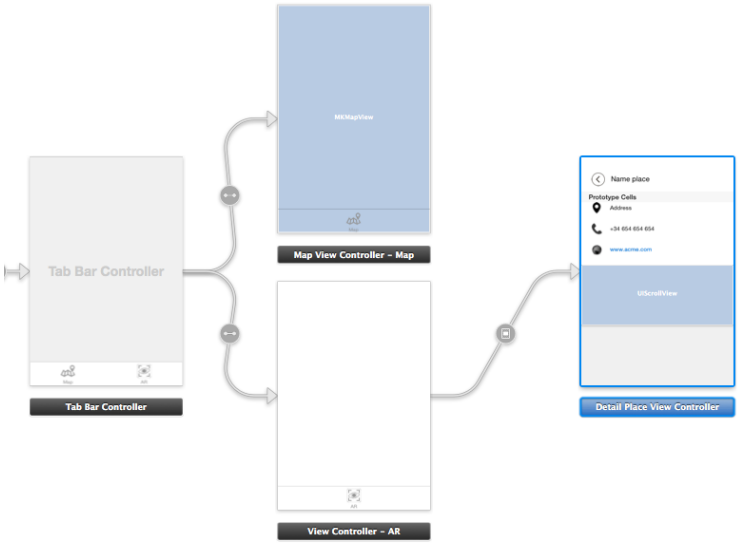

Figura 2: Storyboard de la app

espectaculares y con un rendimiento muy bajo. Concretamente, este proyecto habría sido imposible haberlo desarrollado con alguno de estos frameworks de HTML5 por la necesidad de trabajar con una librería de realidad aumentada (ARToolKit).

Claramente, el mejor resultado de una app pasa por desarrollo nativo en su plataforma. Para desarrollar una app nativa en iOS hemos de usar el SDK oficial de iOS que Apple facilita a los desarrolladores. Para trabajar con este SDK es imprescindible hacerlo usando el lenguaje Objective-C (y C en casos muy concretos).

Objective-C [5] [9] es un lenguaje de programación creado en el año 1980 y que fue adoptado como lenguaje de programación del sistema operativo NeXTSTEP [6]. NeXTSTEP fue un sistema operativo creado por la empresa NeXT [8] fundada por Steve Jobs [4] y que apple compro para mejorar su sistema operativo Mac OS [7].

Objective-C (en ocasiones lo abrevio como Obj-C o ObjC) esta definido como un lenguaje de programación orientado a objetos creado como un super conjunto de C para que implementase un modelo de objetos parecido a Smalltalk [3]. Es un super conjunto de C ya que tiene todo el "potencial" de C pero la notación particular de este lenguaje deja claro que no invocamos a métodos de objetos sino que **pasamos** mensajes a los objetos. Obj-C es un lenguaje de tipado fuerte y estático. Todos los objetos propios de Obj-C se referencian con punteros pero para nosotros es algo transparente ya que no vamos a tener que hacer aritmética de punteros. Bastará con indicar delante del puntero, que tipo de dato es el apuntado y ObjC se encargará de calcular la aritmética de punteros por nosotros.

Al igual que pasa en C, en ObjC para definir una clase tenemos 2 archivos diferentes, la especificación o interface y la implementación. La especificación o interface se define en un archivo que lleva por nombre el propio nombre de la clase con la extensión `.h` y la implementación igual pero con extensión `.m` si solo contiene código Obj-C o `.mm` si además contiene código en C.

La especificación (en ObjC se llama interface) de una clase es pública y cualquier otro objeto podrá consultarlo para instanciar el objeto que especifica o para pasarle mensajes a dicho objeto. ObjC permite la herencia simple. Todos los objetos heredan de alguna super clase y en la cima de esta herencia esta la clase NSObject. Sobre los métodos de una clase hay que notar, que hay dos clases de métodos: métodos de instancia (los que se especifican con el símbolo '-' al principio de su nombre) y métodos de clase (los que se especifican con el símbolo '+' al principio de su nombre). La diferencia entre un método de instancia y un método de clase es que estos segundos no tiene acceso a las variables de instancia porque la clase no se llega a instanciar. Este tipo de métodos de clase suelen ser usados para instanciar la clase de una forma diferente a la estándar. Podemos ver un ejemplo de interface en el código 1

La definición de un método en ObjC se compone por: indicar el tipo de método, El tipo de dato que retorna, el nombre del método y los parámetros. Indicaremos el tipo de método con el símbolo `+` si es un método de clase y con un `-` si es un método de instancia. El tipo de dato que retorna el método viene especificado entre los simbolos `‘(’` y `‘)’`. Si el método no retorna nada, lo denotaremos con la palabra clave `void`. Una de las particularidades de ObjC es que el nombre de los métodos va mezclado con los parámetros que recibe. Esto, a primera vista puede parecer extraño pero realmente es una buena manera de dar sentido a cada parámetro que recibe el método. Podemos tomar como ejemplo, el último método definido en el ejemplo de código 1 donde el nombre del método seria `metodoDeInstancia1ConParam1:yConParam2:` y los parámetros serian `(tipoParam1)param1` y `(tipoParam2)param2`

Código 1: Ejemplo de interface para "myClass"

```
@interface myClass : superclassname
{
    // variables de instancia
}
+ metodoDeClase1;
+ (returnType)metodoDeClase2;
+ (returnType)metodoDeClaseConParametro:(tipoParam)parametro;

- (returnType)metodoDeInstancia1ConParam1:(tipoParam1)param1 yConParam2:(
    tipoParam2)param2;
@end
```

2.3. Los patrones de diseño en iOS

Uno de los aspectos más importantes en el desarrollo de software son los patrones de diseño. Los patrones de diseño nos permitirán tener un código más reusable y entendible para otro programador. Cada tecnología tiene una serie de patrones más importantes sobre los demás. En concreto, en el desarrollo de apps para iOS los patrones más importantes son el patrón *modelo-vista-controlador (MVC)*, el patrón *delegate*, el patrón *singleton* y el patrón *observer* (que es una variante del patrón *observer*). Voy a repasar brevemente en que consiste cada uno de estos patrones y la forma en la que se implementan en iOS. No es mi objetivo dar una visión completamente exhaustiva de cada patrón pero si dar una pincelada sobre cada uno de ellos. Si el lector desea conocer más sobre patrones de diseño y como se implementan, le recomiendo este libro [2]

El desarrollo de una app en iOS solo es posible implementando el patrón MVC. Este patrón es seguramente muy conocido por todos y no será necesaria explicación alguna sobre su importancia y forma de implementación ya que se explica y es necesario su uso en la práctica de la asignatura GRAU-PROP.

2.3.1. El patrón delegate

El patrón delegate es una forma de diseño en la que entra en juego la delegación de responsabilidades. Es decir, Podemos tener una clase `A` que tiene como función calcular las coordenadas geográficas en las que se encuentra el usuario. Esta clase A tiene asignado un delegado al cual la clase A le notificara las coordenadas del usuario pasándole un mensaje (en lenguaje C diríamos que invoca un método del delegado) con dichas coordenadas. Este patrón tal vez es el más importante en el desarrollo de apps para iOS y también es posiblemente el que más cuesta de entender por el desarrollador primerizo.

Para implementar el patrón delegado en iOS necesitamos definir un protocolo. Un protocolo no es más que una serie de 'compromisos' que promete cumplir aquel objeto delegado que se ajuste a este protocolo que estamos definiendo. Para acabar de explicarlo mejor me ayudaré con un ejemplo completo de uso del patrón delegado que he usado en la clase `PlacesLoader` de este proyecto.

Código 2: Ejemplo de protocolo

```
@protocol PlacesLoaderDelegate <NSObject>
@optional
- (void)dataResultJSON:(NSDictionary *)json;
- (void)mediaResult:(UIImage *)image for:(UIImageView *)imageView;
@end
```

Como podemos ver en el ejemplo de código 2 un protocolo comienza con la palabra clave `@protocol` y termina con `@end`[3]. Lo siguiente es definir el nombre del protocolo que por convención en ObjC suele ser el nombre de la clase + la palabra *Delegate* con el formato *lowerCamelCase*[4]. Después del nombre se indica si este protocolo a su vez se ajusta a algún protocolo. Por norma, todos los protocolos se ajustan al protocolo 'base' NSObject y se especifica de la forma `<NSObject>`.

Un protocolo puede definir métodos que han de ser implementados por el delegado de forma obligatoria (`@required`) o de forma opcional (`@optional`). La diferencia entre definir métodos obligatorios u opcionales es que si un método esta definido como obligatorio y el delegado que se ajuste a este protocolo no lo tiene implementado, saltara un error en tiempo de compilación. Por defecto, se toman todos los métodos de un protocolo como obligatorios si el programador no dice lo contrario. Por último se define la firma de los métodos delegados que el objeto delegado ha de implementar.

Una vez definido el protocolo, hay que indicar qué clase se ajustara a dicho protocolo. Esta declaración se formaliza escribiendo `<nombreDelProtocolo>` al lado del nombre de la clase en la declaración de la interface de dicha clase. Podemos ver el ejemplo de código 3 donde la clase `MapViewController` se compromete a ajustarse a 3 protoclos: `CLLocationManagerDelegate`, `MKMapViewDelegate` y `PlacesLoaderDelegate` (protocolo que acabamos de definir en el código 2).

Una vez definido el protocolo y la declaración de ajuste al protocolo por parte del delgado, tenemos que implementar los métodos delegados en el delegado (que en el ejemplo que estamos viendo, es de la clase `MapViewController`).

Código 3: Declaración de que la clase MapViewController se ajusta a los protocolos `CLLocationManagerDelegate`, `MKMapViewDelegate`, `PlacesLoaderDelegate`

```
@interface MapViewController () <CLLocationManagerDelegate, MKMapViewDelegate,
    PlacesLoaderDelegate>
```

Para implementar el/los método/s delegado/s, bastará con ir a la sección `@implementation` de la clase que ha prometido ajustarse al protocolo e implementarlo ahí. Podemos ver un ejemplo de implementación en la clase `MapViewController` del método delegado `dataResultJSON` del protocolo `PlaceLoaderDelegate` en el código 4.

[3]Todas las palabras clave que forman parte de ObjC en exclusiva comienzan con el simbolo `@`

[4]Estilo de escritura CamelCase `http://es.wikipedia.org/wiki/CamelCase`

Código 4: Ejemplo de implementación de método delegado del protocolo PlacesLoaderDelegate

```
#pragma mark - PlacesLoaderDelegate
- (void)dataResultJSON:(NSDictionary *)json
{
    if([[json objectForKey:@"status"] isEqualToString:@"OK"]) {
        id places = [json objectForKey:@"results"];

        Places *placesDataModel = [Places sharedInstance];

        if([places isKindOfClass:[NSArray class]]) {
            [placesDataModel setPlaces:places
                        withLocation:[self.mapView userLocation]];
        }

        [self showPlacesAnnotationsForLocations:placesDataModel.locations];
    }
    self.placeLoaderInProgress = NO;
}
```

Siguiendo nuestro ejemplo, la clase `MapViewController` queremos que sea delegada de la clase `PlacesLoader`. Para indicar justamente esto, desde la clase `MapViewController` tendrá que notificárselo a la instancia de `PlaceLoader` de la forma `PlacesLoader *placeLoader = [[PlacesLoader alloc] initWithDelegate:self];` Esta linea hace varias cosas pero lo que nos atañe en cuanto al *delegate pattern* es la parte en la que cita `...WithDelegate:self` con este fragmento de código le estamos indicando a la instancia de la clase `PlacesLoader` que nosotros (`self` hace referencia al objeto actual) somos el delegado de esta instancia. Como no podía ser de otra manera, la clase `PlacesLoader` tiene un método llamado `initWithDelegate:` que recibe como parámetro la referencia a su delegado y lo asigna a una `@property`[5] de la clase `PlacesLoader` llamada `delegate`. El lector ha de notar la particularidad de la notación de una propiedad delegada. `id` quiere decir que el valor de la propiedad `delegate` puede ser cualquier tipo de clase pero sea la clase que sea, se ajustara al protocolo `PlacesLoaderDelegate` tal y como se denota con `<PlacesLoaderDelegate>`.

Código 5: Property del delegado e inicialización de la clase PlacesLoader con setteo de delagado

```
// Definimos una propiedad publica en la interface de la clase PlacesLoader
@property (nonatomic, weak) id<PlacesLoaderDelegate> delegate;
```

[5]las `@property` las podemos ver como variables de instancia que además ya tiene implementados sus *accessors* (*getters* y *setters*)

```
// Implementamos un inicializador 'custom' de la clase PlacesLoader que setea el
    delegado
- (id)initWithDelegate:(id)delegate
{
    self = [super init];
    if (self) {
        self.delegate = delegate;
    }
    return self;
}
```

Ya solo nos falta un paso para completar la implementación del patrón delegado en este ejemplo. Este paso restante es que la instancia de `PlacesLoader` le envíe un mensaje a su delegado (en nuestro ejemplo es una instancia de la clase `MapViewController`). Para realizar esto, como podemos ver en el código 6, bastara con hacer `[self.delegate dataResultJSON:json];`. Si solo escribimos esta sentencia, dado el contexto de nuestro protocolo, el método delegado `dataResultJSON:` es opcional y eso quiere decir que el delegado puede (o no) tenerlo implementado. Antiguamente, si le pasábamos un mensaje a un objeto delegado y dicho mensaje no tenia ningún selector que contestara, se producía un error en tiempo de ejecución. Actualmente (última versión del compilador LLVM), no se produce dicho error pero sigue siendo una buena practica de programador, asegurarse que el delegado puede contestar a dicho mensaje. Esta comprobación la realizaremos con el método `respondsToSelector:` que contestara con el valor `YES` (equivalente a `true` en C) si el parámetro implícito de este selector (en este ejemplo es el delegado) tiene implementado el selector (así se llaman los métodos en ObjC) `dataResultJSON:`. En caso contrario, `respondsToSelector:` retornara `NO` (equivalente a `false` en C).

Código 6: Comprobamos que el delegado tenga implementado el método delegado `dataResultJSON:` y si lo tiene, se le envia un mensaje a dicho método delegado del objeto delegado

```
NSDictionary *json = [NSDictionary alloc] initWithDictionary:@{...}];
...
if ([self.delegate respondsToSelector:@selector(dataResultJSON:)])
    [self.delegate dataResultJSON:json];
```

2.3.2. El patrón singleton

El patrón singleton es un patrón muy usado en el desarrollo en iOS y del que apple hace mucho uso. A mi juicio es un patrón tan útil como peligroso ya que si

se implementa de forma incorrecta puede provocar problemas de concurrencia y/o integridad de datos.

La idea de este patrón es que limite el número de instancias de una clase a como máximo 1 instancia. Esto nos permite que cualquier clase recupere la instancia de esa clase (si existe) y así evitar que se tenga que ir pasando la instancia de esta clase entre las diferentes clases. Vamos a ver un ejemplo de singleton en la clase `Places` que es un modelo de datos.

Código 7: Declaración en la clase `Places` de un método de clase, público, llamado `sharedInstance`

```
@interface Places : NSObject
...
+ (Places *)sharedInstance;
...
@end
```

Como podemos ver en el ejemplo de código 7, definimos un método de clase (no de instancia porque precisamente lo que queremos recuperar es la instancia!) llamado `sharedInstance` que nos retornara un puntero a una instancia de la clase `Places`.

Código 8: Implementación del singleton en el objeto `Places` usando GCD

```
@implementation Places
+ (Places *)sharedInstance
{
    static Places *instance = nil;
    static dispatch_once_t onceToken;

    dispatch_once(&onceToken, ^{
        instance = [[Places alloc] init];
        instance.locations = [[NSMutableArray alloc] init];
    });

    return instance;
}
...
@end
```

Una vez definido el método en la interface de la clase, hay que implementar dicho método. Hay varias maneras de implementar el singleton pero la más segura es usando *Gran Central Dispatch*[6] ya que si se da una *race condition* entre dos o

[6] *Gran Centra Dispatch (GCD) es el motor de concurrencia en iOS*

más theads podemos tener un problema. Podemos ver un ejemplo en el código 8. Comenzamos definiendo una variable de la clase `Places` que es de tipo `static`. Después también definimos una variable de la clase `dispatch_once_t` (perteneciente a GCD) también de tipo `static`. Después definimos el bloque `dispatch_once` que solo ejecutara su cuerpo una sola vez para la clase `Places`. Finalmente, el método `sharedInstance` siempre retornará el mismo puntero a la instancia `Places`.

Llegados a este punto, solo nos falta ver como se recupera la instancia de la clase `Places` desde cualquier otra clase. Podemos ver en el código 9 como tenemos que hacer un import[7] de la especificación de la clase `Places` y cuando queramos recuperar dicha instancia, lo haremos pasándole un mensaje al selector `sharedInstance` del objeto `Places`.

Código 9: Recuperar instancia de la clase singleton `Places`

```
#import "Places.h"
...
@implementation ARViewController

#pragma mark - Life cycle view
- (void)viewDidLoad
{
    [super viewDidLoad];
    ...
    Places *placesDataModel = [Places sharedInstance];
    ...
}
...
@end
```

2.3.3. El patrón *observer*

El patrón observer en iOS hay que verlo como si fuese un tablón de anuncios donde podemos suscribirnos a unos anuncios en concreto y cada vez que ese o esos anuncios cambien, podemos recibir una notificación. Este patrón nos permitirá que un objeto pueda notificar una acción o cambio en sus propiedades a los demás objetos que estén pendientes de el. Este patrón los usa mucho iOS para notificar eventos del sistema como por ejemplo cuando aparece el teclado, el dispositivo cambia su orientación, etc. Este tablón de anuncios que comentábamos, es una clase singleton que tiene toda app por defecto y se llama `NSNotificationCenter`.

[7]Notad que un import no es lo mismo que un include (que tambien existe en ObjC). Un import hace lo mismo que un include pero además se asegura de que solo se incluye una vez

Los pasos a seguir para implementar este patrón son: definir la notificación (poner el anuncio en `NSNotificationCenter`) y luego hacer las publicaciones al tablón de anuncios cuando queramos realizar una comunicación. Como podemos ver en el ejemplo de código **??**, recuperamos la instancia de la clase singleton `NSNotificationCenter`, Después, con el fragmento `addObserver:self` indicamos que el observador es el objeto en el que nos encontramos. Con el fragmento `selector: @selector(metodoDelegado:)` indicamos que método de la instancia que hayamos indicado en `addObserver:` ha de ser notificada cuando alguien publique un anuncio para el. Con el fragmento `name:@"metodoDelegado"` indicamos que escuchamos las notificaciones que lleven por nombre **metodoDelegado**. Por último, podemos acompañar esta notificación de un contexto indicándolo como parámetro en `object:` (normalmente esto siempre lo dejaremos a `nil`).

Código 10: Ejemplo de definición de notificación para ser resposivos a notificaciones con el nombre `metodoDelegado`

```
[[NSNotificationCenter defaultCenter] addObserver:self
                                         selector:@selector(metodoDelegado:)
                                             name:@"metodoDelegado"
                                           object:nil];
```

Ahora nos falta lanzar las publicaciones al tablón de anuncios (`NSNotificationCenter`). Para hacer una publicación no es necesario que haya ningún observador, simplemente si no hay ningún observador, el sistema descarta la noticia y ya esta. Como podemos ver en el ejemplo de código `observerPublish`, cuando queremos hacer una publicación en el tablón de anuncios, bastara con recupera la instancia del sigleton `NSNotificationCenter` y indicar el nombre de la publicación (`postNotificationName: @"metodoDelegado"`) que además podra ir acompañada de un contexto que es este caso es un objeto del tipo `NSString` que tiene como valor la cadena **"mensajito"**.

Código 11: Ejemplo de publicación de anuncio con el nombre `metodoDelegado` y con contexto

```
    [[NSNotificationCenter defaultCenter] postNotificationName:@"metodoDelegado"
                                                        object:@"mensajito"];
```

Llegados a este punto, hemos definido los observadores, hemos publicado un anuncio y ahora hemos de implementar el *callback* que queremos que se lance cuando se recibe una notificación. Como podemos ver en el ejemplo de código 12, este método recibe como parámetro, un objeto `NSNotification` que entre otras cosas, contendrá el contexto `@"mensajito"` que hemos publicado. Este método debe estar implementado en las clases que estén observado la notificación **@"metodoDelegado"**.

Código 12: *Callback* de una notificación que recibe un contexto y lo muestra por la consola del sistema

```
- (void)metodoDelegado:(NSNotification *)notification
{
  NSString *cadena = (NSString *)[notification object];
  NSLog(@"%@", cadena);
}
```

2.4. Sobre ARToolKit

La realidad aumentada es una técnica de la computación gráfica que permite "fundir" el mundo real con el virtual. El estado del mundo es capturado por una cámara y sobre esta información capturada, se le añade información "virtual" que es mostrada a través de una pantalla al usuario final.

La realidad aumentada a pesar de que no es una técnica nueva, si que se ha dado a conocer hace pocos años por el gran público gracias al video juego *Invizimals* del estudio *Novarama*. Otras empresas como IKEA (véase la figura 3) a usado esta técnica para que el cliente pueda ver como quedarían sus muebles en sus casas.

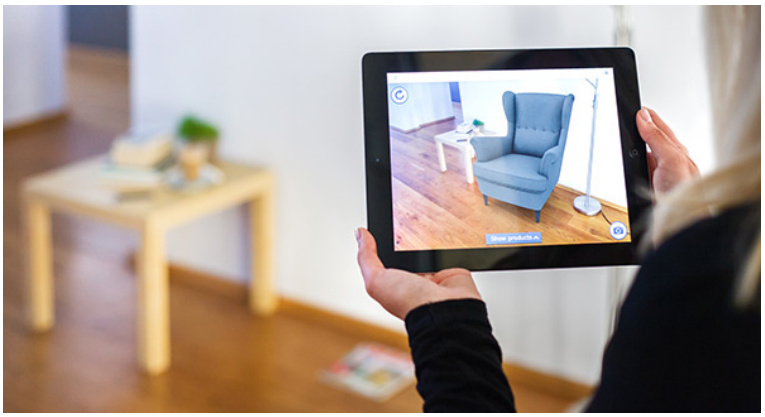

Figura 3: Ejemplo de uso de la realidad aumentada para poder mostar como quedarian los muebles en el contexto de una habitación concreta.

Como hemos dicho antes, la librería usada para resolver el ARmode es ARToolkit. ARToolkit es una librería muy popular en el mundo de la realidad aumentada porque es de código abierto, está escrita en C y actualmente dispone de una infinidad de ports a varios sistemas como Windows, Mac, Linux, iOS, Android, etc. Inicialmente, la función de esta librería es la de reconocer marcadores ("tags") y una vez reconocidos, mostrar sobre su plano, un modelo 3D (como se puede ver en

la figura 3) que reaccionara a los movimientos del tag. Esta funcionalidad nosotros no la podemos aprovechar ya que no vamos a disponer de estos tags. Pero aun así, esta librería nos descarga de las tareas de capturar el video y colocar los tags en la región que toca de la pantalla. ARToolkit usa el inclinómetro y magnetómetro del iPhone para saber hacia dónde esta enfocando el usuario y coloca el marcador en el lugar correcto de la pantalla.

2.5. API Google Places

Google pone a disposición de los usuarios un servicio que aúna todos los puntos (establecimientos, monumentos, recintos feriales, centros comerciales, etc.). Este servicios es conocido como *API Google Places*. Es un servicio completamente gratuito pero has de cumplir una serie de requisitos de registro para poder tener un mayor número de peticiones[8] por día a la API. Para los datos que vamos a necesitar obtener la versión gratuita cumple con creces nuestras necesidades. Para poder usarla hay que registrarse en el servicio de esta API para obtener los tokens que nos permitiran firmar cada petición de recurso de dicha API.

Para interactuar con la api de *google places* hacemos dos llamadas. La primera llamada es para obtener todos los *places* cercanos indicando la localización actual del usuario y el radio que delimitara el perímetro de búsqueda. Adicionalmente también indicamos que tipo[9] de *places* entran dentro de la búsqueda (museum, stadium, university, etc...). Una vez lanzada la petición, la api responde con una estructura JSON (o XML si se prefiere aunque es un formato menos eficiente) donde se relacionan los resultados de la búsqueda. De este response básicamente lo que nos interesa es conocer por cada *place* su nombre, coordenadas geográficas y un identificador para que posteriormente nos podamos referir a este *place* y obtener más datos. Por desgracia, esta api solo facilita las coordenadas geográficas en un plano 2D, es decir, solo se puede obtener longitud y latitud, no la altura. Esto nos imposibilita poder marcar con precisión elementos que están a una altura diferente de la nuestra. Como estos datos no hay manera de obtenerlos (y no los podemos calcular), asumo que todos los places se encuentran a la misma altura que el observador (usuario de la app).

Cuando queremos obtener más datos de un *place* determinado, basta con consumir el recurso `https://maps.googleapis.com/maps/api/place/details/output?parameters` indicando como parámetros la referencia del *place* (entre otras cosas).

[8]Para saber más sobre los limites de uso, visite: `https://developers.google.com/places/policies?hl=es#usage_limits`

[9]Para conocer todos los tipos de locales ver aqui: `https://developers.google.com/places/documentation/supported_types?hl=es`

Una vez lanzada la petición de este recurso, nuevamente, obtenemos un response en formato JSON con todos los datos que tiene google sobre el *place* indicado. Entre estos datos tenemos la dirección desglosada en las diferentes áreas administrativas, fotos, teléfonos de contacto, etc.

2.6. La vista "Map"

El controlador de la vista Map presenta la primera vista que ve el usuario. Una de las primeras cosas que hace este controlador es iniciar el proceso de buscar la localización GPS del usuario.

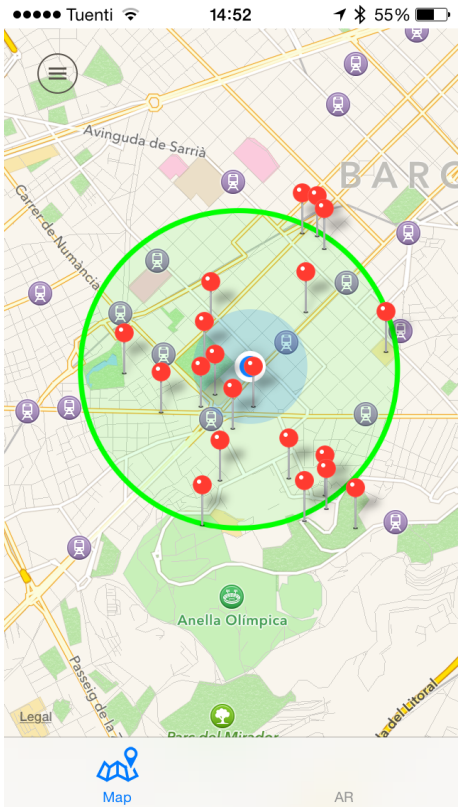

Figura 4: Vista del modo mapa

Tener acceso a la localización GPS es algo que no se puede hacer directamente y es por eso que hay que ceñirse al protocolo `CLLocationManagerDelegate`. Este protocolo establece una serie de métodos que serán invocados por el objeto `CLLocationManager` cada vez que se actualiza la ubicación. En concreto el método del protocolo `CLLocationManagerDelegate` que tenemos que implementar es el `locationManager:didUpdateLocations:`. Dentro de este método definimos que

sólo nos interesan las localizaciones que tienen una precisión inferior a 100 metros. La precisión del GPS varía porque en el momento 0 de activar la actualización de localización, ésta procede de una técnica conocida como A-GPS. A-GPS lo que hace es obtener (o intentarlo) la posición geográfica triangulando la potencia de los repetidores GSM y WIFIs que haya alrededor. Mientras actúa el A-GPS, el GPS empieza. El modulo GPS tiene mayor precisión pero a costa de mayor consumo energético y algo más lento de respuesta. Generalmente cada muestra de localización que se obtiene va mejorando la precisión si estamos quietos o nos movemos poco. Hay que añadir que el GPS tiene un error definido a propósito por sus creadores (el gobierno americano) y ese error varia en función de la región geográfica en la que se encuentra el usuario y de la hora a la que se recoja la muestra. Obviamente, este error no se puede averiguar y hay que convivir con él.

Una vez obtenemos una muestra de la localización con la precisión que queremos, entonces ya podemos iniciar una petición a la API de Google Places para obtener POIs en un perímetro de 1000m con centro en nuestra localización. Para simplificar las funcionalidad de la app, he fijado el perímetro a 1000 metros ya que obtener puntos más lejanos es un poco absurdo si queremos que tenga sentido la realidad aumentada. Una vez recibimos la relación de POIs de la API, procedemos a crear el modelo de datos y a mostrar esos POIs en el mapa.

Sobre el mapa añadiremos tantos markers como POIs hayamos obtenido de la API. Estos markers, como no podía ser de otra manera, se posicionan sobre el mapa en las coordenadas del POI y se les añade un título con el nombre del POI. Adicionalmente, se dibuja sobre el mapa, un circulo que muestra el perímetro sobre el que se hace la búsqueda de *places*.

2.7. La vista "AR"

El modo AR presenta la imagen capturada en tiempo real por la cámara trasera del iPhone y superpone las etiquetas de los places que se encuentran dentro del frustum de visión. Cada etiqueta de places esta compuesta por el nombre del place y la distancia a la que se encuentra desde el observador.

El controlador de este modo (AR Mode) es ARViewController y lo primero que hace es instanciar el objeto *AugmentedRealityController* indicando sobre qué vista se ha de aplicar, cuál es el controlador que lo instancía y qué objeto es su delegado. Lo siguiente es configurar cómo queremos que trabaje la realidad aumentada. La configuración más importante es definir en qué factor se pueden escalar las etiquetas de los places (en función de la distancia a la que se encuentran).

Una vez ya tenemos el objeto *AugmentedRealityController* alocatado en memoria

Figura 5: Vista del modo AR identificando el Parque Joan Miró

y configurado, pasamos a recuperar la instancia del modelo de datos. Para simplificar el paso de datos entre los modos, he optado por crear el modelo de datos de *places* como singleton. Una vez recuperado el modelo de datos iniciamos el proceso de generar las etiquetas para que la AR los pueda superponer cuando tengan que aparecer.

Para marcar cada place en la AR hay que hacer dos cosas: definir en qué punto del espacio se encuentra desde nuestra posición y definir el formato en el que se mostrará. Para definir en qué punto del espacio se encuentra desde nuestra posición usaremos el objeto *ARGeoCoordinate*. Este objeto tiene definido un método de clase (note la diferencia entre método de clase y método de instancia) llamado **+ (ARGeoCoordinate *)coordinateWithLocation:(CLLocation *)location locationTitle:(NSString*)titleOfLocation;** el cual recibirá como parámetros la localización (como objeto CLLocation), el titulo del *place* y una vez alocatado, nos retornara un puntero a su instancia. El objeto CLLocation forma parte del framework *CoreLocation* que a su vez esta incluido en el SDK de iOS. Este objeto tiene todo lo necesario para poder modelar un punto en el espacio. Sus atributos más im-

portantes son las coordenadas sobre el plano (latitud y longitud) y la altura. Como ya hemos comentado anteriormente, la API de Google places no indica a qué altura sobre el nivel del mar se encuentran los places, así pues, en este caso tenemos que obviar la altura y nos centraremos solo en las coordenadas geográficas. Una vez creado el objeto se realiza una calibración usando la posición en la que se encuentra el usuario. Hay que notar que el objeto *ARGeoCoordinate* no posiciona los places de forma absoluta sobre el plano sino de forma relativa a la posición del usuario. Para hacer esta calibración le pasamos el mensaje **calibrateUsingOrigin:(CLLocation *)origin** a la instancia del objeto *ARGeoCoordinate*. Una vez ya tenemos el objeto *ARGeoCoordinate* posicionado respecto al observador, tenemos que indicar qué vista mostrará. Para ello, instanciaremos el objeto **MarkerView**.

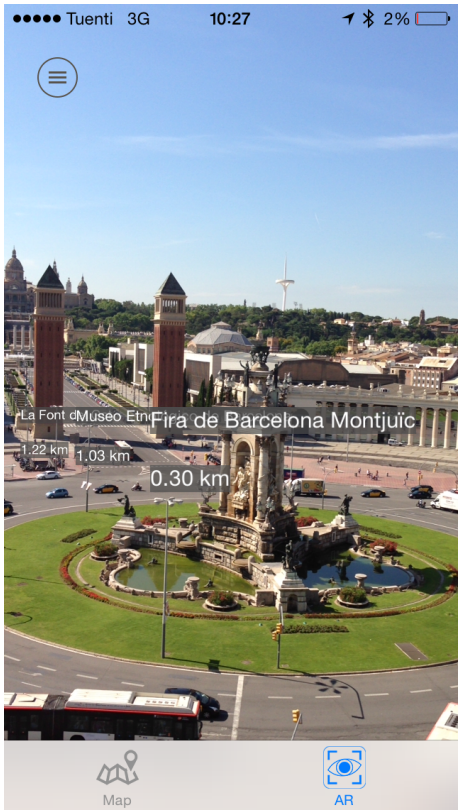

Figura 6: Vista del modo AR identificando varios *places* donde el tamaño de sus etiquetas va en función de la distancia que nos separa de ellos.

El objeto **MarkerView** hereda las propiedades de la clase **UIView** para que pueda ser una subvista de la vista AR. Para inicializar este objeto le pasamos como parámetros, la referencia a la instancia *ARGeoCoordinate* (que contiene la posición relativa del place desde el observador y el titulo del place) y también le indicamos que nosotros (ARViewController) somos su delegado. El motivo por el que definimos el

delgado es para que cuando el usuario haga 'tap' sobre el marcador (que es de la clase **MarkerView**), se notifique al objeto ARViewController que se ha hecho tap sobre un place determinado y así poder mostrar los detalles de este place. Adicionalmente, **MarkerView**) calcula la distancia (en kilómetros con precisión de dos decimales) para que el usuario pueda tener más información sobre la distancia que le separa del POI.

Cuando el usuario hace tap sobre una etiqueta del modo AR, saltan dos métodos delegados. Uno es el método **- (void)locationClicked:(ARGeoCoordinate *)coordinate** que se ajusta al protocolo **ARLocationDelegate** y otro es el método **- (void)didTouchMarkerView:(MarkerView *)markerView** que se ajusta al protocolo **ARMarkerDelegate**. El primero no tiene una aplicación práctica y solo lo usaremos para debbugar. El segundo método es el que instanciará la vista modal de detalle del POI sobre el que el usuario ha hecho tap.

2.8. La vista detalle

Cuando pulsamos sobre un POI del *AR Mode*, se presenta una vista modal que contiene detalles del POI que ha sido pulsado. Los datos mostrados de cada POI son: el nombre del POI, la dirección en la que se encuentra, el teléfono de contacto (si hay), la *url* de la página web (si tiene) y la colección de imágenes que hay sobre este POI en la api de *Google Places* (si hay).

Como hemos dicho, la vista detalle es del tipo modal que instancia la clase `ARViewController`. Podemos ver en el ejemplo de código 13 como lo primero que se hace es instanciar este controlar. Este controlador es del tipo `DetailPlaceViewController`. Opcionalmente, una vez instanciado el objeto, le indicamos que animación queremos que se realice en el momento de la presentación. Antes de presentar este nuevo controlador, asignamos a su propiedad pública `place` la referencia al modelo de datos que contiene los datos del place que queremos mostrar en el detalle. Una vez definidos estos conceptos, ya estamos en disposición de presentar este nuevo controlador.

Código 13: Presentar la vista detalle

```
DetailPlaceViewController *detailPlaceVC = [self.storyboard
    instantiateViewControllerWithIdentifier:@"Detail"];
detailPlaceVC.modalPresentationStyle = UIModalTransitionStyleCrossDissolve;
detailPlaceVC.place = [_locations objectAtIndex:index];
[self presentViewController:detailPlaceVC animated:YES completion:nil];
```

Una vez, el controlador `DetailPlaceViewController` toma el control, lo primero

que hace es lanzar una petición a la api de *Google Places* para obtener más detalles sobre el POI. Como podemos ver en el código 14, recuperar el detalle de un POI es muy parecido a recuperar una lista de POIs. Al objeto `placeLoader` le indicamos que somos su delegado para que una vez tenga el *JSON* de datos del detalle, nos lo notifique.

Código 14: Recuperar información de detalle del POI

```
PlacesLoader *placeLoader = [[PlacesLoader alloc] initWithDelegate:self];
NSMutableString *uri = [NSMutableString stringWithString:apiURL];
[uri appendFormat:@"details/json?reference=%@&sensor=true&key=%@",
                  self.place.reference,
                  apiKey];
NSURL *url = [NSURL URLWithString:[uri
stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding]];
[placeLoader loadDataForURL:url];
```

Cuando la instancia de `PlacesLoader` recupera los datos, se envía un mensaje a su delegado (en este caso `DetailPlacesViewController`). Este método delegado (ver código 15) lo que hace es recoger el *JSON object* que viene bajo el identificador *result*. Después añade al modelo de datos de este *places* los datos que queremos mostrar: teléfono, dirección de la página web y colección de fotos. Hay que destacar que si el *JSON object* no contiene alguna de estos datos, no se producirá una excepción, solo que en vez de asignar un puntero a estos datos, se asignara un `nil`. Por último, cuando hemos recuperado los datos, lo que tenemos que hacer es dar orden al componente `UITableView` (que es el que mostrará los datos) para que se recargue con estos datos nuevos.

Código 15: Método delegado de PlacesLoader en DetailPlaceViewController

```
#pragma mark - PlaceLoaderDelegate
- (void)dataResultJSON:(NSDictionary *)json
{
    json = [json objectForKey:@"result"];

    [self.place setPhoneNumber:[json objectForKey:kPhoneKey]];
    [self.place setWebsite:[json objectForKey:kWebsiteKey]];
    [self.place setPhotos:[json objectForKey:@"photos"]];

    [self.tableDetails reloadData];
}
```

Como he comentado, estos datos se presentan dentro de un componente `UITableView` [1]. No entraré en detalle de como funciona este componente nativo de iOS pero si indicaré que he definido celdas con aspectos diferentes en función de si lo que tenemos

que mostrar es un numero de teléfono, una dirección postal, una dirección web o una colección de fotos. Para el caso de mostrar las fotos, he decidido mostrarlas cobre otro componente nativo de iOS, el `UIScrollView`. Añadir las fotos sobre este componente, nos permitira poder hacer scroll y navegar entre ellas con el famoso gesto *swipe*.

3. Conclusiones y trabajo futuro

Una gran mejora que podríamos añadir a esta app es un recomendador de lugares basados en las preferencias del usuario. Usando técnicas de inteligencia artificial, podríamos dotar a la app de capacidad para aprender sobre los gustos y preferencias turísticas del usuario y recomendarle que puntos puede visitar.

Una de las mayores limitaciones que tiene esta app es que no tiene en cuenta la altura a la que se encuentran los POI. Esta limitación viene dada por la api de *Google Places*. Tal vez, esta limitación se podría salvar cruzando las coordenadas del POI con una base de datos que dadas unas coordenadas, retorne la altura sobre el nivel del mar a la que se encuentre. No sería una solución exacta pero al menos, si nos encontramos en el valle de una montaña y enfocamos a un POI que se encuentra en la cima de una montaña, saldrá bien identificado.

Otro de los puntos débiles de esta app es el tiempo que tarda en recuperar las imágenes de la api de *Google Places*. Eso se podría resolver implementando un sistema de caches para que una vez se a recuperado una imagen, ésta quede almacenada en la capa de persistencia. Una imagen que está almacenada en local tiene un tiempo de recuperación (no es inmediato) pero es mucho más rápido que consumir un recurso de la api.

También podríamos dotar la app de nuevas *features*, Por ejemplo, permitir que el usuario guarde un *place* como favorito para que en cualquier otro momento pueda volver a ver su detalle. En este punto, el límite está en la imaginación.

Bibliografía

[1] Apple. Uitableview class reference. https://developer.apple.com/library/ios/documentation/uikit/reference/UITableView_Class/Reference/Reference.html, 2014. [Internet; consultado 18-mayo-2014].

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[3] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[4] W. Isaacson. *Steve Jobs: La biografía*. Penguin Random House Grupo Editorial España, 2011.

[5] S. Kochan. *Programming in Objective-C 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[6] Wikipedia. Nextstep — wikipedia, la enciclopedia libre. http://es.wikipedia.org/w/index.php?title=NEXTSTEP&oldid=69950211, 2013. [Internet; consultado 24-mayo-2014].

[7] Wikipedia. Mac os — wikipedia, la enciclopedia libre. http://es.wikipedia.org/w/index.php?title=Mac_OS&oldid=74977346, 2014. [Internet; consultado 15-mayo-2014].

[8] Wikipedia. Next — wikipedia, la enciclopedia libre. http://es.wikipedia.org/w/index.php?title=NeXT&oldid=74139742, 2014. [Internet; consultado 17-mayo-2014].

[9] Wikipedia. Objective-c — wikipedia, la enciclopedia libre. http://es.wikipedia.org/w/index.php?title=Objective-C&oldid=73802004, 2014. [Internet; consultado 8-mayo-2014].