CAPÍTULO 1

Implementación

1.1. Introducción

En este capítulo se muestra la integración de los conocimientos adquiridos para poder llevar a cabo la realidad aumentada en una aplicación real. Si bien era de gran interés del proyecto la exploración de distintos métodos y algoritmos parecía importante poder poner en práctica todo lo desarrollado en un producto final que pudiera parecerse a un prototipo de aplicación comercial. En particular se desarrolló una aplicación pensando en los cuadros de la planta baja del Museo Nacional de Artes Visuales (MNAV). Entre otros autores, tiene cuadros de Pedro Figari, Juan Manuel Blanes y de Joaquín Torres García que se eligieron para hacer el prototipo.
La aplicación consta de distintas funcionalidades que se describen en este capítulo, tales como:

(1) Detección QR

(2) Navegación por listas de cuadros

(3) Comunicación con un Servidor con la base de datos.

(4) Detección SIFT para identificar el cuadro.

(5) Diferentes realidades aumentadas según la obra.

En las próximas secciones se describe más en detalle cada uno de estos puntos y su integración a la aplicación final. También se describe el flujo de la aplicación y algunas clases implementadas.

1.2. Diagrama global de la aplicación

En la descripción de las clases que conforman los bloques principales de la aplicación se hace referencia a conceptos de desarrollo sobre Objective-C, así como también a *frameworks* y herramientas utilizadas que fueron explicadas en el capítulo **??**. Para la comprensión del detalle de la implementación es importante conocer estos conceptos de desarrollo.
Para que sea más sencilla la comprensión de los bloques que componen la aplicación, en la figura 1.1 se muestra un diagrama esquemático de la aplicación que sirve para visualizar cómo es el flujo de la aplicación *a nivel de usuario*.
Al comenzar el recorrido, el usuario tiene la opción de elegir cómo recorrer el museo: de manera autónoma (*Por galería*) o de manera automática (*Por navegación*). En la opción autónoma el

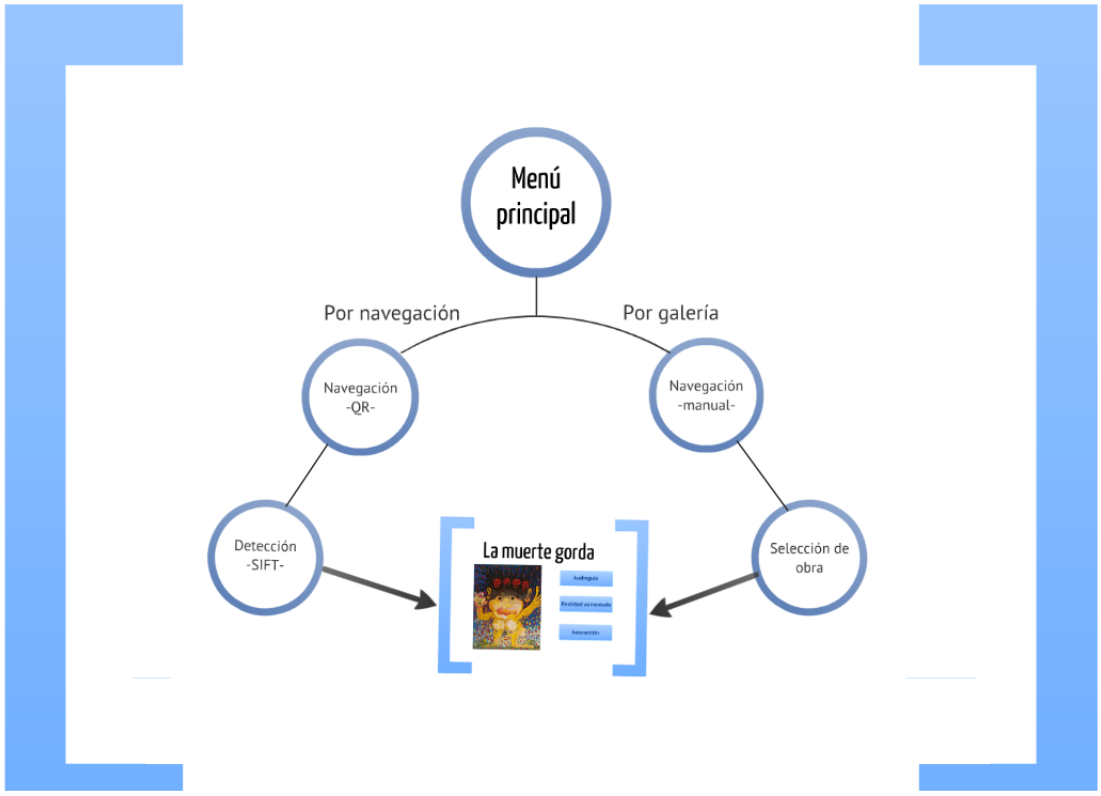

Figura 1.1: Diagrama global de la aplicación

usuario es el encargado de elegir dentro de una lista de autores el que más le interese, y dentro de la lista de cuadros del autor seleccionado, la obra que desea contemplar en detalle. De esta manera el usuario llega eligiendo opciones al cuadro de interés y está listo para comenzar la interacción con la obra, a través de audioguías o realidad aumentada. De la otra manera de recorrer el museo, con la opción automática, el usuario tiene la opción de leer códigos QR desplegados en las distintas salas o secciones del museo, que sirven para identificar en qué parte del museo se encuentra el usuario. De esta manera una vez que el usuario lee el QR, la aplicación lo reconoce y despliega una foto del autor y un mensaje que invita al usuario a continuar con el recorrido. Internamente la aplicación guarda la información en la que está el usuario y la utiliza en la siguiente etapa: reconocimiento de obra. El reconocimiento de la obra se da una vez que el usuario está frente a la misma y toma una foto de ella que es procesada y en pocos segundos la aplicación responde con la imagen original de la obra y con información extra de la misma como una breve descripción, el nombre del autor y ofrece la posibilidad de interactuar con la misma.
De esta manera es que se da el flujo de la aplicación a nivel de usuario, para llegar a un determinado cuadro de interés y generando interacción. Pero este flujo es necesario representarlo en una serie de clases e instancias y con cierta invocación de métodos que cumplan las reglas de Objective-C con las herramientas existentes de desarrollo que provee Xcode. Para tener una idea de cómo se mapea el flujo de la aplicación en el lenguaje de desarrollo, en la figura 1.2, se presenta el Storyboard de la aplicación, que muestra la relación de las distintas clases. A su vez, a dicha imagen se le agregó un número identificador en cada ViewController para poder hacer referencia en la medida que sea necesario detallar determinados aspectos de las clases involucradas.
En las próximas subsecciones se pasan a explicar algunas de las clases implementadas en la aplicación y que tienen cierta relevancia. Se muestran su rol dentro de la aplicación y sus principales

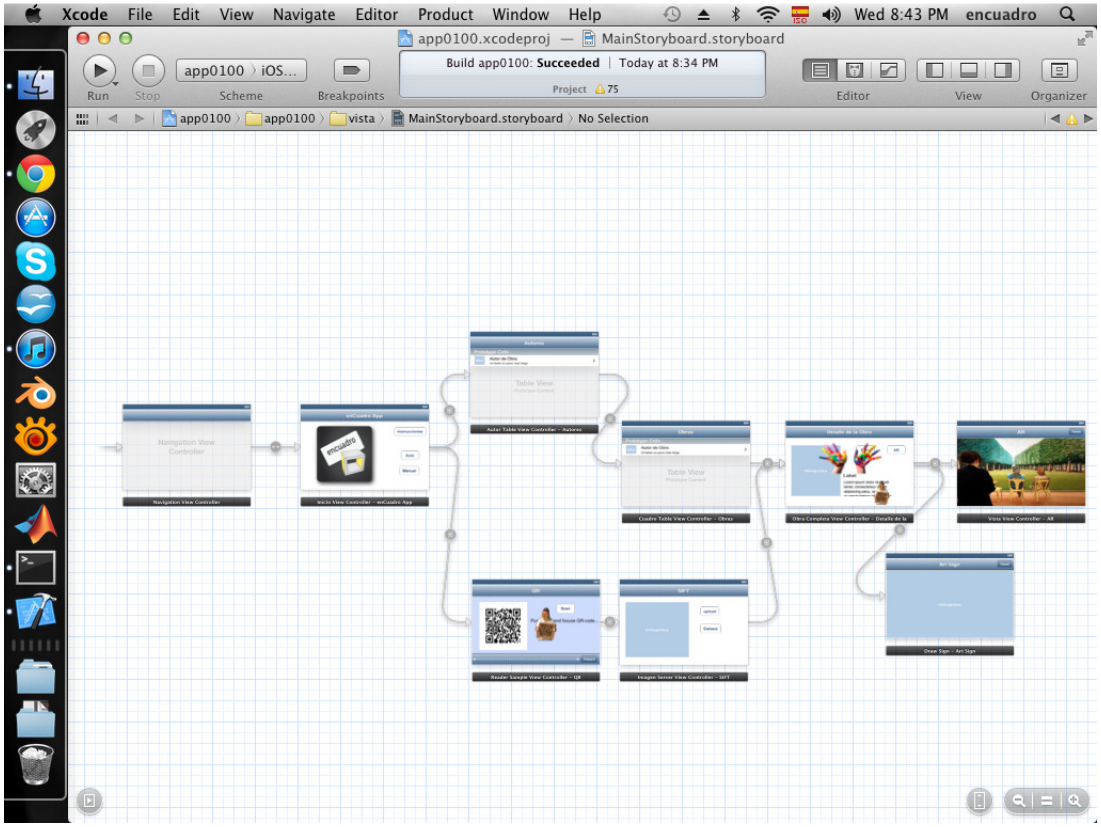

Figura 1.2: Diagrama global de la aplicación

características.

1.2.1. NavigationViewController

La aplicación está embebida dentro de un *UINavigationViewController*. Esto implica que cada uno de los *ViewControllers* que tiene la aplicación es gestionado por esta clase. Es quien se encarga de la presentación y del pasaje de un *ViewController* a otro, creando y destruyendo instancias de cada uno. Está en esta clase la responsabilidad de manejar las jerarquías de los distintos *ViewControllers* así como de mantener cierta integridad visual utilizando las Toolbars ya sea arriba como encabezado o abajo al pie.
Para el caso particular de esta aplicación se optó por reimplementar esta clase ya que se buscaba tener cierto control sobre las rotaciones. En particular se reimplementaron los métodos *supportedInterfaceOrientations* y *preferredInterfaceOrientationForPresentation* de la siguiente manera

```
-(NSUInteger)supportedInterfaceOrientations
{
    NSLog(@"supportedInterfaceOrientations NAVIGATION");
    return UIInterfaceOrientationMaskLandscapeRight;
}

- (UIInterfaceOrientation)preferredInterfaceOrientationForPresentation
{
    NSLog(@"preferredInterfaceOrientationForPresentation NAVIGATION");
    return UIInterfaceOrientationLandscapeRight;
}
```

Esto lo que hace es fijar la orientación de la interfaz de usuario a modo *LandscapeRight*. También es posible lograr esto editando el archivo *Info.plist* que toda aplicación de xcode tiene y agregando el item *SupportedInterfaceOrientations* y completado las opciones que se desean. Las autorotaciones es algo que tiene bastante relevancia en las aplicaciones. En particular se optó por esta forma, es decir, bloquear las autorotaciones y dejar solamente la vista en un sentido, para facilitar la reproyección de la realidad aumentada. De no haberlo hecho de esta manera, con cada rotación de la interfaz se tendrían que intercambiar los ejes de coordenadas en función del sentido de la rotación. Esto es posible de hacer ya que con cada rotación se ejecuta una serie de métodos en forma automática entre los cuales se encuentra el siguiente

```
- (void) willRotateToInterfaceOrientation:(UIInterfaceOrientation)toInterfaceOrientatio
```

Dentro de dicho método sería posible hacer el ajuste de coordenadas ya que el mismo se ejecuta previo a cualquier rotación. El tema de los métodos que son ejecutados al haber un evento del tipo rotación es algo que ha sufrido cambios recientes con la actualización de software a iOS 6. De todas formas, como ya se dijo, se optó por hacer esto más sencillo, simplemente evitando que la interfaz de usuario rote junto con el dispositivo, dejándola fija.

1.2.2. InicioViewController

Este *ViewController* es la pantalla de inicio de la aplicación. En la misma hay un botón que al ser presionado comienza un audio con instrucciones y una presentación sobre cómo es el recorrido y las funcionalidades con las que cuenta la aplicación. También hay dos botones más que dan la opción de elegir la forma de recorrer el museo. El botón de recorrido automático es un *segue* al *ReaderSampleViewController* y el de recorrido manual un *segue* al *AutorTableViewController*.

1.2.3. TableViewControllers

Para el recorrido manual, el usuario es el encargado de seleccionar el autor, luego las obras disponibles del autor seleccionado y luego se muestra un detalle de la obra seleccionada por el usuario presentando una instancia del *ViewController* llamado *ObraCompletaViewController*. Este recorrido que parece bastante intuitivo aparece en muchas aplicaciones de iOS en las que existe una lista de datos. Como ejemplo se ve en aplicaciones que gestionan contenido musical que está ordenado en base a autores, dentro de los mismos, sus discos y dentro de los discos sus canciones por ejemplo. Dado que es algo bastante frecuente, el hecho de navegar en listas de datos, xcode ya tiene implementada una clase llamada *UITableViewController*. De esta manera lo que se hizo fue crear varias clases que heredan de *UITableViewController* y manejar los contenidos de manera jerárquica. A continuación siguen dos clases que se resolvieron de esta manera.

1.2.3.1. AutorTableViewController

Esta clase hereda de *UITableViewController*. En particular tiene algunos métodos que son importantes de destacar si se quiere implementar algo parecido con una lista de datos. El siguiente método:

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
```

por defecto retorna un *0*. El mismo indica la cantidad de secciones con las que cuenta una tabla. Para que tenga sentido y al instanciarse la clase se vea algo de contenido tiene que retornar algo distinto de *0*. Otro método importante es el siguiente:

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)sectio
```

El mismo es el encargado de devolver un número con la cantidad de filas con las que cuenta la sección de la tabla. En esta implementación se devuelve la cantidad de autores.
Un tercer método que tiene mayor importancia es el siguiente:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexP
```

El mismo es el encargado de devolver una *UITableViewCell* que es la que se despliega. Es en este método que se configura el formato de la celda. Para el caso de la aplicación se resolvió generar una clase que hereda de *UITableViewCell* que se llama *CuadroTableViewCell* y que tiene ciertas características como una imagen, autor y obra que son mostradas en la celda. En este método se asocian las características mencionadas de la celda en función del número de fila. Esta clase implementa un método *prepareForSegue* que setea la variable *opcionAutor* en función del autor seleccionado. Esto permite luego en la clase *CuadroTableViewController* desplegar distintas listas de cuadros en función del autor seleccionado.

1.2.3.2. CuadroTableViewController

Esta clase es muy similar a la clase *AutorTableViewController* recién descripta difiriendo simplemente en su contenido. Los conceptos utilizados y métodos implementados son básicamente los mismos pero su contenido es un listado de obras en lugar de autores. Una especificación extra es que al ejecutrase el método *viewDidLoad* se completa una lista de cuadros diferente en función del autor seleccionado. Así como en la clase *AutorTableViewController* en esta también se implementa el método *prepareForSegue* para poder completar los datos de la instancia de la clase con la que se está conectando, completando los datos de la obra seleccionada (autor, obra, imagen, descripción, audio, ARid). El ARid es un identificador de realidad aumentada que asocia una realidad aumentada a cada cuadro. Esto se verá más adelante en la sección 1.6.

1.2.3.3. CuadroTableViewCell

Esta es una clase sencilla que hereda de la clase *UITableViewCell* y simplemente tiene tres *IBOutlets* como propiedades para vincularlas con una imagen, un nombre de autor y un nombre de la obra para cada celda de la tabla que se despliega.

1.2.4. ReaderSampleViewController

Este *ViewController* es el encargado de hacer la lectura de los códigos QR y de invocar los métodos necesarios para realizar la búsqueda de la zona del museo en la que se encuentra el usuario. Esto es, existe un código QR asociado a cada autor (Blanes, Figari y Torres García) y en base al código QR leído se despliega un texto e imagen asociadas al mismo. El funcionamiento de la decodificación se explica un poco más en detalle en la sección 1.3.

1.2.5. ImagenServerViewController

Este *ViewController* es el encargado de la comunicación con el servidor. En el método *viewDidLoad* se instancia un *UIImagePickerController* encargado de implementar una captura de imagen. Una vez que se saca la foto, la misma se muestra en una *UIImageView* y existen dos botones: uno de ellos simplemente dispara una nueva instancia del *UIImagePickerController* dando la opción de

Figura 1.3: Autor y Cuadro TableViewControllers

Figura 1.4: Ejemplo de captura para reconocimiento SIFT

sacar otra foto y el otro botón inicia la comunicación con el servidor.

El botón encargado de la comunicación con el servidor, botón de *upload*, es un *segue* hacia el *ObraCompletaViewController*. Dentro del método *prepareForSegue*, encargado de preparar todo previo a la invocación de *ObraCompletaViewController* se invoca el método *uploadImage*. Este método genera un mensaje HTTP del tipo POST y se lo envía a la IP del servidor. En el cuerpo del mensaje se agrega la foto sacada previamente y se le agrega un string llamado *room*. Este string es completado previamente en el *ReaderSampleViewController* en base al QR detectado, dando la información sobre en qué sala/región del museo se encuentra (sala Figari, sala Blanes o sala Torres García). Este string lo que permite es tener un identificador para poder hacer la búsqueda de la imagen sacada (luego de procesar la imagen con el algoritmo SIFT), en una base de datos más peque?a, que contenga solamente los cuadros de esa región del museo y no todos los cuadros del museo. En caso que el usuario no haya detectado ningún QR y haya seleccionado directamente la opción de sacar una foto para comenzar la comunicación con el servidor, entonces este string estará vacío y la búsqueda de la imagen más parecida se hace en toda la base de datos. En un primer caso, claramente los tiem-

pos son mejores (del orden de 3s en una LAN) que en el segundo (del orden de 6s en una LAN). Luego de establecida la conexión y enviada la consulta POST, el servidor responde con un string *returnString*. Este string contiene el resultado del procesamiento de la imagen enviada. Esto se logra mediante un archivo *upload.php* en el servidor que toma imágenes, ejecuta comandos en terminal (en este caso el ejecutable del algoritmo SIFT) y el resultado de esa ejecución se retorna como string en el php y el mismo se envía a la aplicación. Ese string resultado es recibido por la aplicación con una nomenclatura elegida que sigue la lógica Autor-Número, por ejemplo *Figari3* que se corresponde con la obra número 3 de la base de datos del autor Figari. Ese pasa a ser el identificador de la obra para ir a buscarla a la base de datos del servidor. El servidor cuenta con varias carpetas según la información:

(1) autor

(2) obra

(3) texto

(4) imagen

(5) audio

Para una obra dada, cuyo identificador sea *Figari3* por ejemplo, los archivos correspondientes, con la información son:

(1) http://IPservidor/autores/Figari3.txt

(2) http://IPservidor/obras/Figari3.txt

(3) http://IPservidor/textos/Figari3.txt

(4) http://IPservidor/imagenes/Figari3.jpg

(5) http://IPservidor/audios/Figari3.mp3

De esta manera una vez que se obtiene el identificador se hace una consulta al servidor que pide los distintos archivos de texto, audio e imágenes. Esta información solicitada es devuelta por el servidor y alojada en variables para ser mostradas (imagenes, texto) y reproducidas (audio) en el siguiente *ViewController*, en el *ObraCompletaViewController*.

1.2.6. ObraCompletaViewController

Este *ViewController* simplemente es la presentación de la obra, mostrando el cuadro, título, autor, descripción y distintas opciones para interactuar con el mismo. Tiene dos botones y una animación que funciona como botón. El primero de los botones dispara un audio donde se explican detalles de la obra que el usuario está contemplando. El otro botón conecta con el *VistaViewController*, encargado de mostrar la realidad aumentada, explicado en la sección 1.2.7. La animación que aparece funciona como *segue* hacia otro *ViewController*, llamado *DrawSign* que se explica más adelante en la sección 1.2.8.

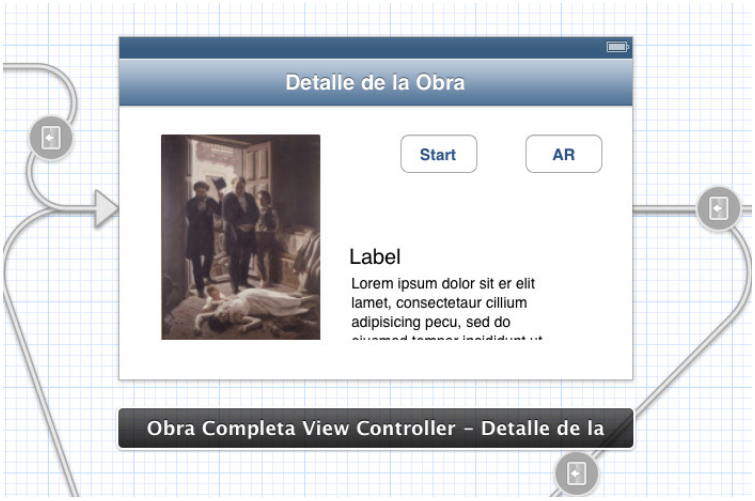

Figura 1.5: Pantalla con la obra completa

1.2.7. VistaViewController

Este *ViewController* es el encargado de mostrar la realidad aumentada. Esta clase, al ser instanciada ejecuta el siguiente método:

```
- (void)viewWillAppear:(BOOL)animated
{
    NSLog(@"VIEW WILL APPEAR VISTA");
    [super viewWillAppear:animated];

    [self hacerRender];

}
```

Este método se ejecuta justo antes de mostrarse la vista del controlador, y como se ve invoca al método homónimo de la clase superior y luego instancia el método *hacerRender*, método que es el encargado de mostrar efectivamente la realidad aumentada. Antes de pasar a explicar los detalles de *hacerRender* se explican algunos detalles generales de las aplicaciones.
Como cualquier programa, en cualquier aplicación de xcode lo que se ejecuta al comenzar es el main. En particular para *xcode* en el main de la aplicación se crea una instancia de la clase que es *appDelegate* de la aplicación. A su vez, al instanciar al *appDelegate* se ejecuta el método *applicationDidFinishLaunching*. En este método típicamente el código por defecto está vacío pero para el caso del *appDelegate* que es utilizado en los ejemplos de *isgl3d* se crea una instancia de una clase que hereda de *UIViewController*, llamada *Isgl3dViewController*. Es sobre esta última clase que se despliegan los *renders*. Aclarados estos puntos se pasa ahora a explicar lo que se hace en el método *hacerRender*.A continuación se muestran algunas de las partes más importantes del método:

```
app0100AppDelegate *appDelegate = (app0100AppDelegate *)[[UIApplication sharedApplicati
    self.viewController=(Isgl3dViewController*)appDelegate.viewController;
```

Con lo anterior lo que se hace es generar una instancia de la clase *app0100AppDelegate* que es puntero al *appDelegate* de la aplicación. Luego, en la segunda línea se le asigna a la propiedad de la propia clase llamada *viewController* (que es de tipo *Isgl3dViewController*) la propiedad de igual nombre pero del *appDelegate* de la aplicación (que fue instanciada en el método *applicationDidFinishLaunching*). Luego se agregan las vistas *viewController.view* y *viewController.videoView*

con valor de transparencia *alpha* nulo y se inicia una animación generando un efecto de *fade out* de la imagen y *fade in* del render. Este tipo de animaciones son sencillas de ejecutra y permiten agregar efectos interesantes a cualquier *UIView*.

1.2.8. DrawSign

Esta clase hereda de *UIViewController* y está pensada para que el usuario pueda dibujar al tocar la pantalla. Un ejemplo de cómo queda el dibujo se puede ver en la siguiente figura.
Se implementó haciendo una reimplementación de los siguientes tres métodos:

Figura 1.6: Ejemplo de dibujo libre

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;
```

Cada vez que una instancia de una clase que hereda de *UIViewController* tiene un evento de tipo *touch* sobre su vista entonces se invocan los métodos mencionados. La secuencia de invocaciones se da al comenzar el toque en la pantalla (*touchesBegan*), al desplazar el dedo sin levantarlo de la pantalla (*touchesMoved*) y al finalizar el *gesture* levantando el dedo de la pantalla (*touchesEnded*). Entonces sabiendo esto, esta clase que tiene como finalidad dibujar, lo que hace es obtener las coordenadas del punto de *touch* invocando el siguiente método:

```
[touch locationInView:self.view]
```

donde *touch* es del tipo *UITouch* y se corresponde con el evento. Una vez que se tienen las coordenadas del punto de contacto en la pantalla se guarda esta posición y al obtener una nueva posición en la pantalla (luego de desplazar el dedo en *touchesMoved*), se dibuja una línea entre el punto actual y el anterior con el método siguiente:

```
[image.image drawInRect:CGRectMake(0, 0, self.view.frame.size.width,
                                   self.view.frame.size.height)];
```

De esta manera lo que se hace realmente es dibujar una línea a la vez entre el punto actual y el anterior. El método *drawInRect* es un método frecuentemente utilizado cuando se quiere dibujar en forma 2D sobre *UIViews* y fue utilizado extensivamente en este proyecto. Finalmente en el método *touchesEnded* lo que se hace es dibujar una línea en el punto actual y sí mismo, generando un punto final al levantar el dedo de la pantalla. Otra característica interesante a mencionar de la capacidad de responder a eventos *touch* es la capacidad de reconocer *gestures* que pueden ser nativos o incluso creados por el propio desarrollador. Para el caso del presente proyecto no se profundizó en la generación de *gestures* propios sino que se limitó a hacer uso de los existentes. Ejemplos de *gestures*

existentes son: *touch*,*double touch*,*multi touch* entre otros. De esta manera si se quiere reconocer un *double touch* por ejemplo, se puede invocar el siguiente método:

```
[touch tapCount];
```

que devuelve la cantidad de veces que se tocó la pantalla en un intervalo corto de tiempo. Esto fue utilizado en esta clase para borrar lo dibujado y poder comenzar a dibujar nuevamente.
A esta clase también se le agregó una *IBAction* que genera un *tweet* con el dibujo generado por el usuario. El mismo es logrado generando una instancia de la clase *TWTweetComposeViewController* y agregándole un texto e imgaen con los siguientes métodos:

```
[controller setInitialText:text];
[controller addImage:img];
```

donde *text* y *img* son el texto del *tweet* y la imagen adjunta. Finalmente se presenta la vista del *TWTweetComposeViewController* y una vez finalizado se vuelve a la instancia *DrawSign*.

1.2.9. TouchVista

Esta clase hereda de la clase *UIView* y se creó para poder manejar eventos *touch* en *ViewControllers* que tienen varias *subviews* y que interesa que se dispare un evento al tocar solamente una de las sub-vistas. Entonces lo que se hace en esos casos es agregar a la sub-vista en cuestión una instancia de *TouchVista* en forma transparente por encima y del mismo tama?o que la sub-vista. De esta manera al tocar la sub-vista se toca en realidad la instancia de *TouchVista* y se invoca el método *touchesBegan*. Este método simplemente tiene el siguiente contenido:

```
[super touchesBegan:touches withEvent:event];
```

También tiene el seteo de una bandera pero eso no tiene tanta relevancia. Lo que se hace en el código anterior es invocar al método *touchesBegan* de la clase superior. Para el caso en que se tiene un *ViewController*, con una sub-vista del tipo *TouchVista* transparente, entonces esta línea invoca directamente el método *touchesBegan* del *ViewController*. Dos de los *ViewControllers* que utilizan esto son *VistaViewController* y *ObraCompletaViewController*.

1.2.10. Isgl3dViewController y app0100AppDelegate

Las clases *Isgl3dViewController* y *app0100AppDelegate* son clases que ya venían implementadas con *Isgl3D*. En particular se tomó el proyecto *Hello World* de dicho *framework* y se trabajó sobre el mismo.
El proyecto *Hello World* es un proyecto básico que lo que hace es generar un *render* sobre un fondo gris. Como lo que se buscó fue hacer realidad aumentada, entonces se trabajó para que el render estuviera por delante de la captura de la cámara. Para lograr esto en el *app0100AppDelegate* se hicieron algunas modificaciones sobre las vistas. A continuación se muestra parte del código que logra esto:

```
UIImageView* vistaImg = [[UIImageView alloc] init];

    /* Se ajusta la pantalla*/
UIScreen *screen = [UIScreen mainScreen];
CGRect fullScreenRect = screen.bounds;
```

```
[vistaImg setCenter:CGPointMake(fullScreenRect.size.width/2, fullScreenRect.size.height
[vistaImg setBounds:fullScreenRect];

[self.window addSubview:vistaImg];
[self.window sendSubviewToBack:vistaImg];
_viewController.videoView = vistaImg;
```

Con esto se ajusta el atributo *videoView* de la propiedad *viewController* que pertenece a la clase *app0100AppDelegate* y es instancia de la clase *Isgl3dViewController*. Este atributo es el que luego se ajusta en forma periódica con cada cuadro en la clase *Isgl3dViewController*.
En xCode, si se quiere simplemente hacer una filmación, sacar fotos y acceder a la galería de las fotos, se utiliza generalmente instancias de la clase *UIImagePickerController*. Esta última clase se instancia en la aplicación en otras clases, como ser *ImagenServerViewController*. Sin embargo si se desea tener una forma de hacer esto pero llegando a más bajo nivel para poder hacer procesamiento, teniendo acceso a los píxeles de la imagen, hacer modificaciones sobre los mismos y en definitiva manipular la salida, entonces la forma más indicada es usando el conocido *Framework* llamado *AVFoundation*. En la clase *Isgl3dViewController* en el método *viewDidLoad* está toda la configuración necesaria para la utilización de *AVFoundation* que permite realizar procesamiento en tiempo real. A continuación se muestra el código con sus comentarios sobre esta configuración.

```
/*Creamos y seteamos la captureSession*/
    self.session = [[AVCaptureSession alloc] init];
    self.session.sessionPreset = AVCaptureSessionPresetMedium;

    /*Creamos al videoDevice*/
    self.videoDevice = [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];

    /*Creamos al videoInput*/
    self.videoInput  = [AVCaptureDeviceInput deviceInputWithDevice:self.videoDevice err

    /*Creamos y seteamos al frameOutpt*/
    self.frameOutput = [[AVCaptureVideoDataOutput alloc] init];

    self.frameOutput.videoSettings = [NSDictionary dictionaryWithObject:[NSNumber numbe

    /*Ahora conectamos todos los objetos*/
    /*Primero le agregamos a la sesion el videoInput y el videoOutput*/

    [self.session addInput: self.videoInput];
    [self.session addOutput: self.frameOutput];
```

Como se ve, es necesario crear una sesión de captura, un dispositivo de captura, una salida de los datos y agregarlas a la sesión. También se puede setear el tipo de captura de la cámara (tiene que ser soportado por el Hardware, sino se genera un error en este punto).
Otra cosa importante que se hace en la clase *Isgl3dViewController* es la configuración del *multithreading*. A continuación se muestra el código que logra esto.

```
dispatch_queue_t processQueue = dispatch_queue_create("procesador", NULL);
[self.frameOutput setSampleBufferDelegate:self queue:processQueue];
dispatch_release(processQueue);
```

Con esto lo que se hace es hacer una instancia de una *Queue*, que representa una cola de procesamiento. De esta manera se puede hacer que ciertas tareas se alojen en esa instancia de cola, que lógicamente es otra distinta que la cola de procesamiento principal (*mainQueue*). Esto mismo es lo que se hace en la segunda línea del código anterior, diciendo que el *Delegate* de los datos de salida (*frameOutput*) es la propia clase y que ese *Delegate* se ejecute en la *Queue* que se instanció en la línea anterior. De esta manera todo lo que sea invocado por el *Delegate* en forma periódica será enviado a una cola distinta de la principal, pudiendo tener entonces, una cola de procesamiento separada de la cola de interfaz de usuario. Esto es algo ampliamente utilizado y es una recomendación de la documentación de iOS, pues se basa en los conceptos de tener la mayor atención posible a la interfaz de usuario, impidiendo en lo posible dejar al usuario esperando por algún eventual procesamiento que se esté llevando a cabo.
Finalmente se da comienzo a la sesión

```
[self.session startRunning];
```

Con esto comienza a invocarse una serie de métodos en forma periódica. El hecho de suceda esto es porque esta clase implementa el protocolo *AVCaptureVideoDataOutputSampleBufferDelegate*. Este protocolo es el que permite decir que el *Delegate* de *frameOutput* es él mismo. Al implementar este protocolo comienza a invocarse en forma periódica (frame a frame) el siguiente método

```
-(void) captureOutput:(AVCaptureOutput *)captureOutput didOutputSampleBuffer:(CMSampleB
```

donde *sampleBuffer* es una referencia al *buffer* que contiene los píxeles de la cámara en ese momento. Así entonces, se accede a los píxeles y se invoca dentro del *captureOutput* periódicamente al método *procesamiento*, encargado de procesar la imagen recibida por la cámara.

1.3. QR

1.3.1. QR. Una realidad

El uso de los identificadores QR (Quick Response), es cada vez más generalizado. Últimamente debido al incremento significativo del uso de *smart devices* el hecho de poder contar con cámara y poder de procesamiento hace que sea frecuente encontrar aplicaciones con el poder de reconocimiento de QRs. Comenzaron a utilizarse en la industria automovolística japonesa como una solución para el trazado en la línea de producción pero su campo de aplicación se ha diversificado y hoy en día se pueden encontrar también como identificatorios de entradas deportivas, tickets de avión, localización geográfica, vínculos a páginas web o en algunos casos también como tarjetas personales.

1.3.2. Qué son realmente los QRs?

Se puede decir que los QRs tienen muchos puntos en común con los códigos de barras pero con la ventaja de poder almacenar mucho más información debido a su bidimensionalidad. Existen distintos tipos de QRs, con distintas capacidades de almacenamiento que dependen de la versión, el

tipo de datos almacenados y del tipo de corrección de errores. En su versión 40 con detección de errores de nivel L, se pueden almacenar alrededor de 4300 caracteres alfanuméricos o 7000 dígitos (frente a los 20-30 dígitos del código de barras) lo cual lo hace muy flexible para cualquier tipo de aplicación de identificación.
En la figura **??** se pueden ver las distintas partes que componen un QR como ser el bloque de control compuesto por las tres esquinas que dan información de la posición, alineamiento y sincronismo, así como también información de versión, formato, corrección de errores y datos. Fuera de toda esa información que podríamos denominar encabezado haciendo analogía con los paquetes de las redes de datos se encuentra la información a almacenar propiamente dicha que conforma el cuerpo del QR.

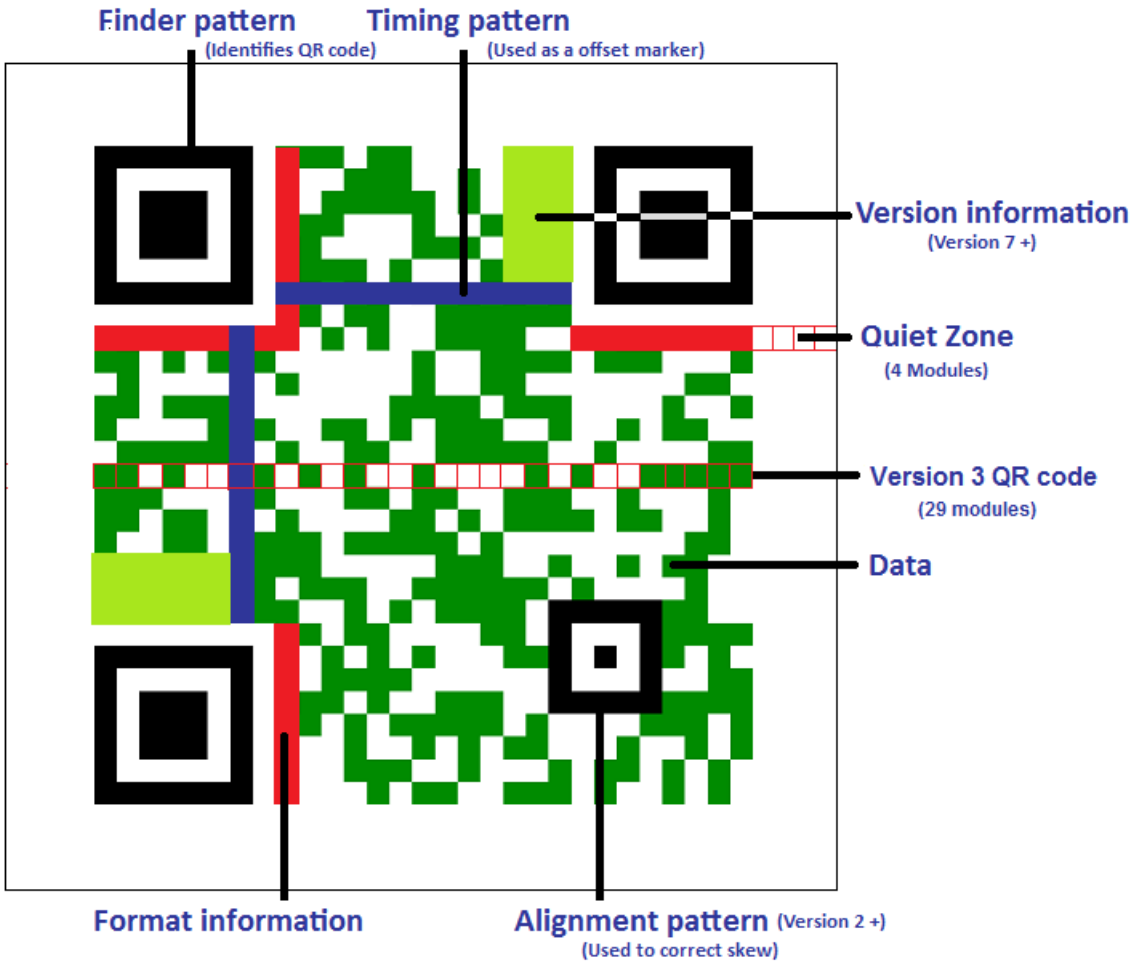

Figura 1.7: Las distintas componentes de un QR. Fuente [**?**].

1.3.3. Codificación y decodificación de QRs

Es fácil darse cuenta que la codificación resulta mucho más sencilla que la decodificación. Para la codificación es necesario comprender el protocolo, las distintas variantes y el tipo de información que se pretende almacenar. Sin embargo para la decodificación, además de tener que cumplir con lo anterior, es necesario contar con buenos sensores y ciertas condiciones de luminosidad y distancia que favorezcan a la cámara y se traduzcan en buenos resultados luego de la detección de errores. Si bien la plataforma es importante para lograr buenos resultados, dada una plataforma, existen variadas aplicaciones tanto para iOS como para Android que cuentan con performances bastante diferentes en función del algoritmo de procesamiento utilizado.
Debido a que el centro del proyecto de fin de carrera no fue la codificación y decodificación de QRs y que además ya existen distintas librerías que resuelven este problema se optó por investigar las distintas variantes e incorporar la más adecuada para la aplicación.
Dentro de todas las librerías que resuelven la decodificación se encuentran ZXing y ZBar como las más destacadas por su popularidad, simplicidad y buena documentación para la fácil implementación. ZXing, denominada así por "Zebra Crossing", es una librería open-source desarrollada en java y que tiene implementaciones que están adaptadas para otros lenguajes como C++, Objetive C o JRuby entre otros.
Por su parte ZBar también tiene soporte sobre varios lenguajes y cuenta con un SDK interesante para desarrollar fácilmente aplicaciones que integren el lector de QR. Se trabajó sobre el código de ejemplo que contiene la implementación de las clases principales para obtener un lector de QRs. Básicamente consta de una clase *ReaderSampleViewController* que hereda de *UIViewController* y que implementa un protocolo llamado *ZBarReaderDelegate* Al presionarse el botón de detección se crea una instancia de la clase *ReaderSampleViewController* y se presenta la vista de cámara. Luego el protocolo se encarga de la captura y procesamiento del QR teniendo como resultado la información que tiene el QR en la variable denominada *ZBarReaderControllerResults*. Esta variable luego se mapea en una hash table con el contenido en formato *NSDictionary*. De esta manera se accede fácilmente al contenido en formato legible y es fácil de hacer una lógica de comparación y búsqueda en una base de datos.

1.3.4. El QR en la aplicación

Para el caso particular de la aplicación se optó por tener un identificador QR para tres artistas elegidos del Museo Nacional de Artes Visuales (MNAV). Los mismos fueron Pedro Figari, Joaquín Torres García, Juan Manuel Blanes. De esta manera para el caso del recorrido del museo a través de la utilización con QRs es posible determinar la posición del usuario debido a imágenes QR debidamente ubicadas en cada zona. Esto sirve como localización y también sirve para lograr que el paso siguiente, que es la identificación de la obra que el usuario tiene enfrente, sea mediante una búsqueda en una base de datos discriminada por autor. Es decir, si el usuario no escanea el QR la búsqueda de la obra a identificar se hará en una base de datos global del museo, pero para el caso que el usuario sí decida escanear el QR entonces se cuenta con la posibilidad de realizar la búsqueda en una base de datos más reducida.

1.3.5. Arte con QRs

La opción de usar los QRs de una manera distinta ha comenzado a ser notoria en los últimos tiempos. Hay quienes desafían a la información *cruda de 1s y 0s* incorporando imágenes y modif-

icando colores y contornos en los QRs tradicionales para lograr un valor estético además del funcional. A continuación se muestran algunos ejemplos de tales casos en los que claramente se ve cómo puede lograrse el mismo resultado de información con el valor agregado de originalidad.

Figura 1.8: Ejemplo de un QR artista. Fuente [**?**].

1.4. Servidor

Si bien el desarrollo de la aplicación es un prototipo de una aplicación comercial y para tal caso no se manejan muchas imágenes y otros datos y registros, para lograr escalabilidad se hace imprescindible contar con un servior. Se pensó con el fin de almacenar toda aquella información relevante en cuanto a registro de obras (imagen, título y autor), descripciones de obras, audioguías, videos, modelos y animaciones para las realidades aumentadas asociadas y cualquier tipo de información que el museo quiera agregar y que por un tema de practicidad no se quiera almacenar dentro de la aplicación. En definitiva, almacenar toda esa información dentro de la aplicación quizá sea rentable para pocas obras, pero lo puede hacer inmanejable para un buen número de obras. Se pensó entonces en la instalación de un servidor que esté ubicado dentro del museo con el cual se tenga una conexión a través de una LAN de (54Mbps). Se aclara este punto pues, en caso de querer hacer un servidor remoto que tenga que ser accedido a través de internet, entonces baja notoriamente su performance, aunque funciona perfectamente.

1.4.1. Creando el servidor

Para la creación del servidor se buscó primeramente la alternativa de hacerlo sobre una máquina con sistema operativo con núcleo Linux, distribución Ubuntu. Luego también se buscó la posibilidad de tener el servidor corriendo sobre una plataforma Unix iOS. Para el segundo caso resultó incluso más sencilla que la primera dado que ya viene pensado por el sistema operativo el hecho de que funcione como servidor. A continuación se explica los pasos que se siguieron para implementar servidores en uno y otro sistema operativo.

1.4.1.1. Servidor iOS

redactar pasos...

1.4.1.2. Servidor LAMP

Se denota servidor LAMP por las siglas de Linux (Sistema Operativo), Apache (Servidor Web), MySQL (Gestor de base de datos), PHP/Perl/Python (lenguaje de programación).
Se instaló entonces el servidor Web Apache, que tiene dentro de sus principales ventajas el hecho de ser multiplataforma, gratis y de código abierto. Para eso desde terminal se debe hacer lo siguiente:

```
sudo apt-get install apache2
```

Con este comando se descarga el paquete *apache2* y se instala. Una vez finalizada la instalación de este paquete ya se cuenta con un servidor y se puede verificar ingresando desde la máquina donde se instaló el servidor abriendo el navegador e ingresando a *http://localhost* o equivalentemente a *http://127.0.0.1* y de esta manera aparece la página por defecto cuyo contenido está dado por el archivo */var/www/index.html*.
Luego de tener instalado el Apache se procede a instalar el php de la siguiente manera

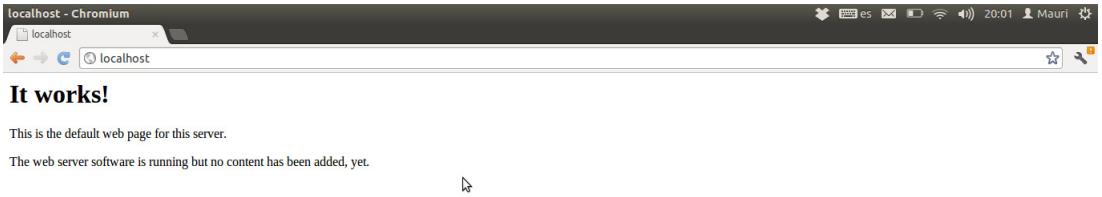

Figura 1.9: Impresión de pantalla al ingresar a http://localhost.

```
sudo apt-get install php5
```

Para el caso particular de los intereses del servidor creado no fue necesario instalar MySQL. Entonces con esto, luego de reiniciar el servidor apache ya se tiene un servidor con intérpetre php instalado. A partir de este momento todo se reduce a comprender bien el lenguaje php y poder realizar peque?os módulos de programación que puedan tomar entradas, procesarlas y arrojar una salida.

1.4.2. Lenguaje php y principales scripts

Como fue dicho en la sección anterior para los intereses el servidor creado, lo fundamental es el hecho de poder recibir archivos o identificadores de los mismos, poder realizar un procesamiento

en el servidor y devolver a la máquina cliente un archivo, mensaje o similar. Para esto se pasa a explicar algunos conceptos básicos de php.
El análogo a un Hello World en php es así:

```
<?php
echo"Hola Mundo";
?>
```

Esto se guarda en un archivo que con extensión .php en la ruta por defecto donde se alojan los php /var/www/holamundo.php. De esta manera al ingresar a la dirección *http://localhost/holamundo.php* se ve el print del texto.
como leer un input y hacer algo (ejemplo suma.php)
como hacer un acction.php ....
MOU SIGUE ESTO...

1.5. SIFT

1.6. Incorporación de la realidad aumentada a la aplicación

asdfasdfasdf

[**?**].