CAPÍTULO 1

Implementación

1.1. Introducción

En este capítulo se muestra la integración de los conocimientos adquiridos a lo largo del proyecto para poder llevar a cabo la realidad aumentada en una aplicación real. Si bien el objetivo principal del proyecto era 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 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 describen 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 misma que sirve para visualizar cómo es su flujo *a nivel de usuario*.
Al comenzar el recorrido, el usuario tiene la opción de elegir cómo recorrer el museo: de manera *autónoma* o de manera *automática*. En la opción autónoma el usuario es el encargado de elegir

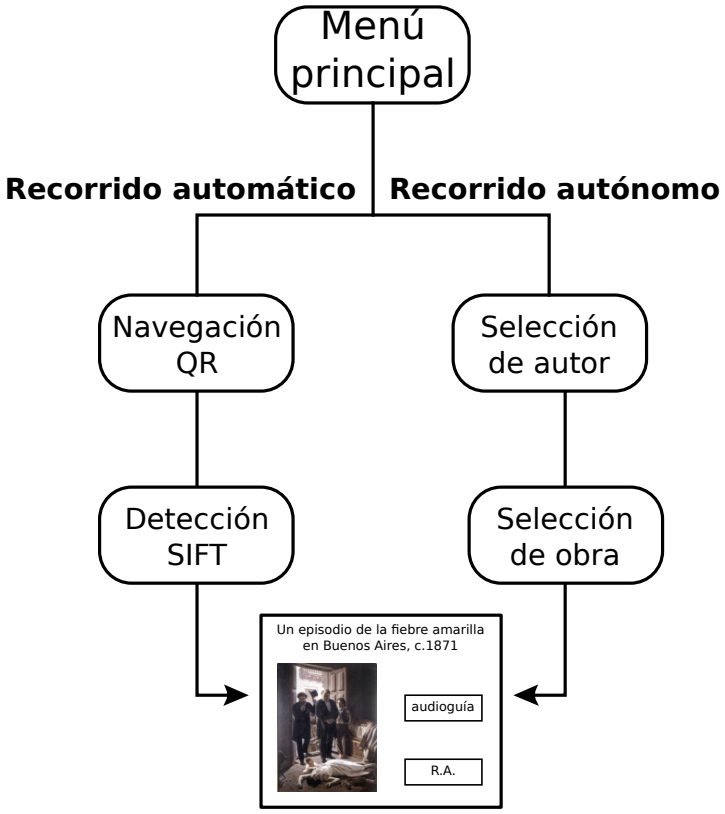

Figura 1.1: Diagrama global de la aplicación

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 el usuario puede comenzar la interacción con la obra, a través de audioguías o realidad aumentada. Ver Figura 1.1

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 así entonces interactuar con él. 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 misma, que muestra la relación entre las distintas clases. Se recuerda al lector que el *Storyboard* es una herramienta de programación gráfica, que permite generar instancias de clases y vínculos entre las mismas en forma visual a la vez de ser una representación gráfica de la interfaz de usuario. A su vez, a la Figura 1.2 se le agregó un número identificador en cada *ViewController* para poder referencialos en la medida que sea necesario detallar determinados aspectos de las clases involucradas.

En las próximas subsecciones se explican algunas de las clases implementadas en la aplicación y

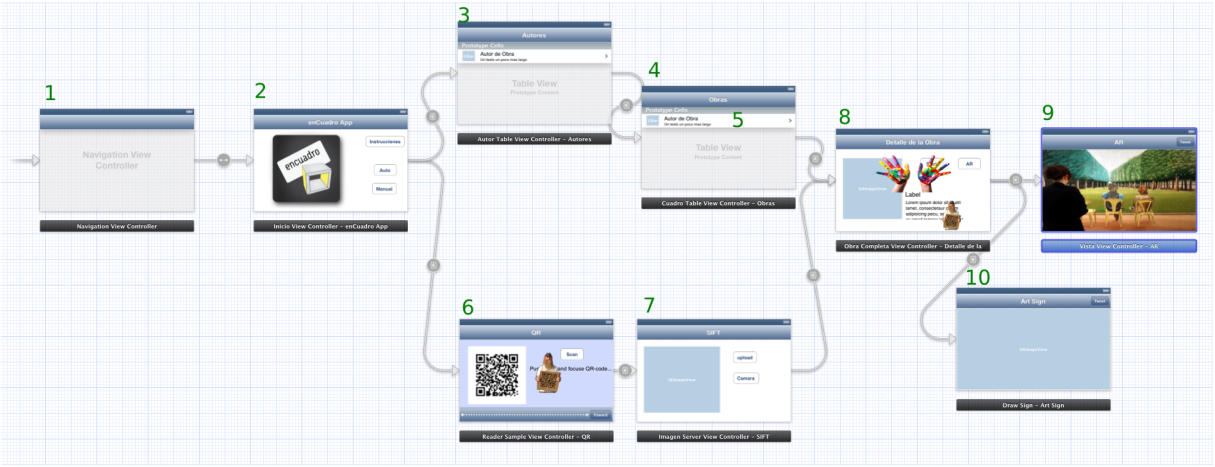

Figura 1.2: *Storyboard* de la aplicación

que además tienen cierta relevancia. Se muestran su rol dentro de la aplicación y sus principales características.

1.2.1. NavigationViewController

Esta clase se ve en la Figura 1.2, identificada con el número 1. 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. Las *Toolbars* son botones que se pueden agregar en los extremos de los *ViewControllers* para realizar una funcionalidad específica.
El hecho de contar con una jerarquía permite entre otras cosas, la posibilidad de hacer un cambio (en la interfaz de usuario por ejemplo), en todos los *ViewControllers*, simplemente afectando a la clase *NavigationViewController* y sin necesidad de cambiar cada uno de ellos por separado. Esto resulta particularmente práctico en aplicaciones con bastantes *ViewControllers* y lo único que tiene que hacer el desarrollador es aclarar que ciertos atributos sean manejados por la clase encargada de la navegación dentro de la aplicación.
Por otra parte, es deseable tener un criterio común para todos los *ViewControllers* en la orientación de la aplicación con respecto a la orientación del dispositivo. Es decir, es posible lograr por ejemplo, que frente a rotaciones del dispositivo, la interfaz de usuario acompañe la rotación y gire también, o también es posible permitir que rotaciones del dispositivo en determinado sentido se vean reflejados en una rotación de la interfaz de usuario y otras no. Para esto se definen cuatro posibles posiciones para el dispositivo con ayuda del acelerómetro: *Potrait*, *Upside Down*, *Landscape Left* y *Landscape Right*. Las mismas se pueden ver en la Figura 1.3.

Para el caso particular de esta aplicación se optó por reimplementar la clase *UINavigationViewController* bajo el nombre *NavigationViewController* ya que se buscaba tener cierto control sobre las rotaciones de la interfaz de usuario, por lo que se decidió afectar los métodos que estuvieran a cargo de las rotaciones de interfaz de usuario. En particular se reimplementaron los métodos *supportedInterfaceOrientations* y *preferredInterfaceOrientationForPresentation* de la siguiente manera

```
-(NSUInteger)supportedInterfaceOrientations
```

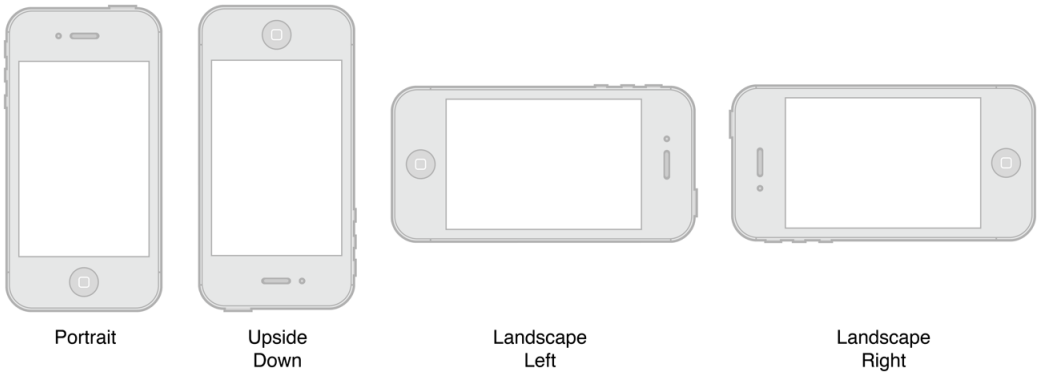

Figura 1.3: Orientaciones posibles del dispositivo.

```
{
    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 hubiera sido posible lograrlo editando el archivo *Info.plist* que toda aplicación de Xcode tiene, agregando el item *SupportedInterfaceOrientations* y completado las opciones que se desean. Las rotaciones de interfaz de usuario son algo con bastante relevancia en las aplicaciones. En particular se optó por bloquear las rotaciones de interfaz, dejándola fija, 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)
toInterfaceOrientation duration:(NSTimeInterval)duration;
```

Dentro de dicho método sería posible hacer el ajuste de coordenadas correspondiente. La serie de 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.

1.2.2. InicioViewController

Este *ViewController* es la pantalla de inicio de la aplicación, identificado con el número 2 en la Figura 1.2. 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 al usuario la opción de elegir la forma de recorrer el museo: autónoma o automática. El botón de recorrido automático instancia al *ReaderSampleViewController* y el de recorrido autónomo instancia al *AutorTableViewController*.

1.2.3. UITableViewControllers

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 existen listas de datos. Un ejemplo son las 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. Como navegar en listas de datos es algo bastante frecuente, Xcode ya tiene implementada una clase llamada *UITableViewController*. En la Figura 1.4 se puede ver un ejemplo con varios tipos de tablas que organizan la información. Como se puede ver la tabla es una forma sencilla de organizar la información en la que existe una sola columna y muchas filas, llamadas celdas. También pueden existir secciones, con un encabezado y pie de sección. Volviendo a la aplicación lo que se hizo entonces fue crear varias

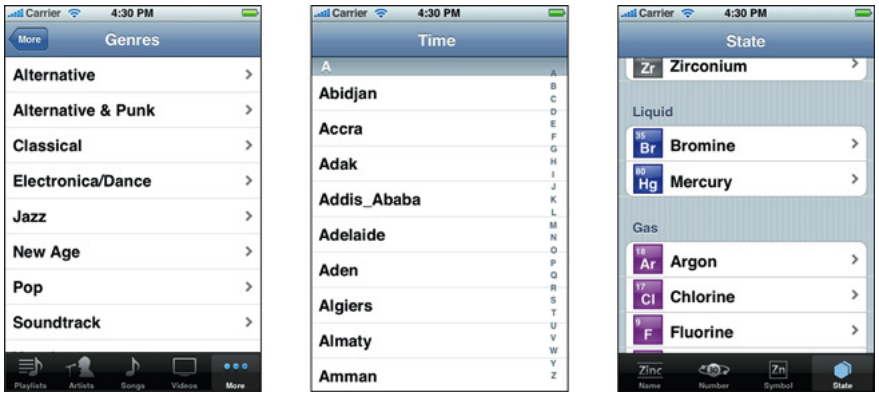

Figura 1.4: Ejemplos de TableViewControllers con distintos tipos de tablas.

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 (identificada con el número 3 en la Figura 1.2) hereda de *UITableViewController* y cumple la función de almacenar la lista de autores disponibles dentro del museo que a los efectos del prototipo como se dijo son: Figari, Blanes y Torres García. En lo que sigue se explican algunos detalles importantes que se tuvieron que comprender para poder organizar la información en tablas de datos (lo cual también aplica para la clase *CuadroTableViewController* que se describe en la siguiente subsección).
Uno de los métodos implementados por esta clase es el siguiente:

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

que 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:

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

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, de mayor importancia, es el siguiente:

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
(NSIndexPath *)indexPath
```

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 le asigna un valor a 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 (identificada con el número 4 en la Figura 1.2) es muy similar a la clase *AutorTableViewController* recién descripta pero que difiere 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 instanciarse la clase 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, con 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* (identificada con el número 5 en la Figura 1.2) y simplemente tiene tres atributos asociados a nivel de interfaz de usuario: una imagen, un nombre de autor y un nombre de obra para cada celda de la tabla que se despliega.

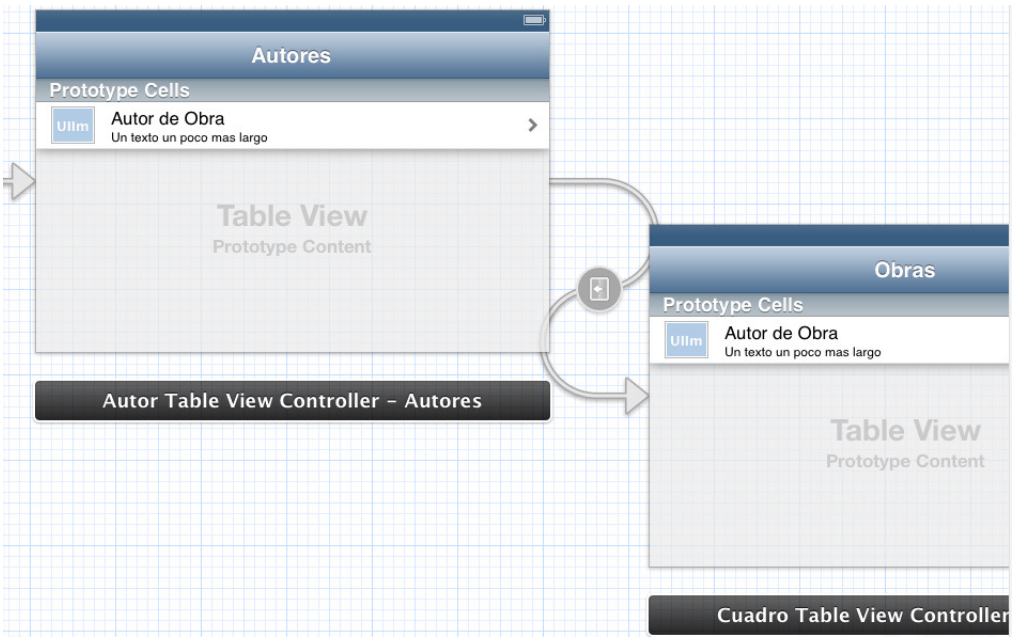

Figura 1.5: Autor y Cuadro TableViewControllers

1.2.4. ReaderSampleViewController

Este *ViewController* (identificada con el número 6 en la Figura 1.2) 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 y una imagen asociados 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* (identificada con el número 7 en la Figura 1.2) es el encargado de la comunicación con el servidor. Al instanciarse la clase, también se instancia la clase *UIImagePickerController*, encargada de implementar una captura de imagen. Una vez que se toma una fotografía a la obra, 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 volver a tomar la fotografía y el otro botón inicia la comunicación con el servidor. Ver Figura 1.6.

Figura 1.6: Ejemplo de captura para reconocimiento SIFT

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 adjunta la foto tomada previamente y se le agrega una variable llamada *room*. Esta variable es completada previamente en el *ReaderSampleViewController* en base al QR detectado, dando información respecto de en qué sala/región del museo se encuentra el usuario (sala Figari, sala Blanes o sala Torres García). Esta variable lo que permite es tener un identificador para poder realizar la búsqueda de la imagen tomada en una base de datos más pequeña, que contenga solamente los cuadros de la región del museo en cuestión. En caso que el usuario se haya salteado la detección QR y haya seleccionado directamente la opción de tomar una fotografía a la obra para comenzar la comunicación con el servidor, entonces la variable *room* estará vacía y la búsqueda de la obra se realiza en toda la base de datos del museo. El gran valor agregado de la detección QR es la velocidad con la que el servidor devuelve información respecto de a qué obra se fotografió. Para la búsqueda con detección QR, los tiempos son claramente mejores (del orden de 3s en una LAN),

mientras que cuando el usuario se ahorra este paso, los tiempos aumentan al doble (del orden de 6s en una LAN).

Luego de establecida la conexión y enviada la consulta POST, el servidor responde con otra variable llamada *returnString*. Esta variable contiene un identificador de obra que indica qué obra fue fotografiada. Esto se logra mediante un archivo *upload.php* en el servidor que recibe la imagen y le ejecuta un algoritmo de detección de características llamado SIFT, que le retorna al PHP el identificador en cuestión. Detalles sobre el algoritmo SIFT se pueden ver más adelante en la sección 1.5. El archivo *upload.php* entrega esta información a la aplicación. La variable *returnString* es recibida por la aplicación con cierta nomenclatura en particular, que sigue la lógica Autor-Número, por ejemplo "Figari3" se corresponde con la obra número 3 de la base de datos del autor Figari. Con este identificador de obra, la aplicación le pide al servidor cierta información de interés acerca de la misma, como por ejemplo el nombre completo de la obra, el nombre de su autor, una breve descripción. El servidor cuenta con varias carpetas a las que la aplicación accede remotamente:

(1) **autor:** contiene el nombre del autor de cada obra.

(2) **obra:** contiene el nombre completo de cada obra.

(3) **texto:** contiene una breve descripción de cada obra.

(4) **imagen:** contiene una imagen de cada obra.

(5) **audio:** contiene una audioguía asociada a cada obra.

Esta información solicitada es alojada en variables que son mostradas (imagenes, texto) y reproducidas (audio) en el siguiente *ViewController*, el *ObraCompletaViewController*. Ver Figura 1.7.

1.2.6. ObraCompletaViewController

Este *ViewController* (identificada con el número 8 en la Figura 1.2) simplemente es la presentación de la obra, muestra una imagen del cuadro, título, autor, descripción y distintas opciones para interactuar con el mismo. Tiene dos botones y una animación que funciona como botón. Ver Figura 1.7. El primero de los botones dispara una audioguía relacionada con 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.

1.2.7. VistaViewController

Este *ViewController* (identificada con el número 9 en la Figura 1.2) 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];

}
```

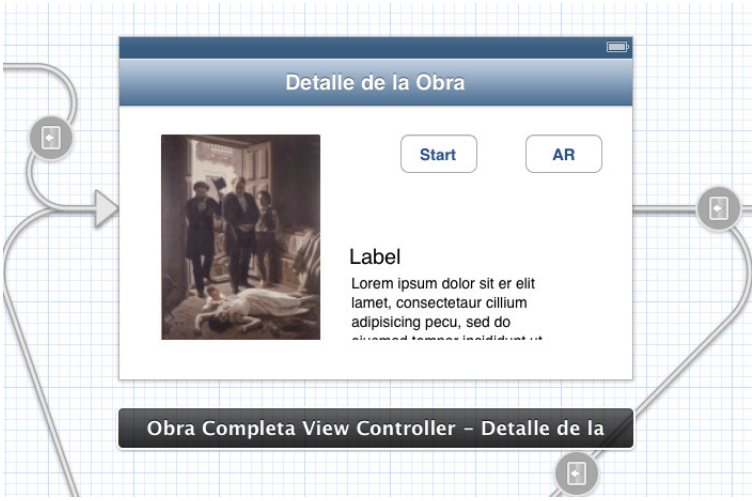

Figura 1.7: Pantalla con la obra completa

Este método se ejecuta justo antes de que el controlador despliegue el contenido de la pantalla, y como se ve invoca al método homónimo de la clase superior y luego al método *hacerRender*, encargado de mostrar efectivamente la realidad aumentada. Antes de explicar los detalles de *hacerRender* se comentan algunos detalles generales de las aplicaciones iOS.

Como en cualquier programa, en las aplicaciones de Xcode, lo que se ejecuta al comenzar es el *main*. En este tipo de aplicaciones en particular, el main crea una instancia de la clase *appDelegate* (delegado de la aplicación). A su vez, al instanciarse al *appDelegate* se ejecuta el método *applicationDidFinishLaunching*. En este método, típicamente el código por defecto está vacío, pero cuando se trabaja con ISGL3D, este método crea un objeto que hereda de *UIViewController*, llamado *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
sharedApplication] delegate];
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 *views 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 ejecutar con el *framework* Core Animation y permiten agregar efectos interesantes a cualquier *UIView*.

1.2.8. DrawSign

Esta clase (identificada con el número 10 en la Figura 1.2) 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 Figura 1.8. Se implementó haciendo mediante una reimplementación de los siguientes tres métodos:

Figura 1.8: 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* detecta un toque sobre la pantalla (evento *touch*), 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*). Se obtienen entonces las coordenadas del punto de toque sobre la pantalla invocando el siguiente método:

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

donde *touch* es del tipo *UITouch* y tiene propiedades que dependen del 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)];
```

El método *drawInRect* sirve para 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 respecto de la capacidad de responder a eventos *touch* es el reconocimiento de *gestures* que pueden ser nativos o incluso creados por el propio desarrollador. Un *gesture* es una forma característica de tocar la pantalla. 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 imagen con los siguientes métodos:

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

donde *text* e *img* son el texto del *tweet* y la imagen adjunta. Finalmente se presenta la *view* 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 tan sólo una de ellas. Entonces lo que se hace en esos casos es agregar a la *subview* en cuestión una instancia de *TouchVista* en forma transparente por encima y del mismo tamaño. De esta manera al tocar la *subview* se toca en realidad la instancia de *TouchVista* y se invoca el método *touchesBegan*. Este método simplemente configura una bandera y configura lo siguienteo:

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

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 *subview* 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. Realidad Aumentada en ISGL3D

Para realizar realidad aumentada se necesita poder hacer un *render* por encima de las imágenes capturadas por la cámara del dispositivo en tiempo real. Sin embargo, cuando se crea un proyecto de ISGL3D, este permite realizar *renders* pero sobre un fondo estático y gris (o cualquier otro color configurable). Resulta entonces necesario configurar el proyecto de manera de reemplzar al fondo antes mencionado por imágenes capturadas por la cámara. Para lograr esto, hubo que trabajar sobre las clases *Isgl3dViewController* y *app0100AppDelegate*. A continuación se muestran algunas modificaciones sobre estas dos clases

```
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*.

En Xcode, si se quiere simplemente hacer una filmación, sacar fotos o 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*. Si lo que se desea es acceder a los píxeles de las imágenes capturadas, para luego por procesarlos en tiempo real, entonces la forma más indicada es usando el conocido *framework AVFoundation*. En la clase *Isgl3dViewController*, en el método *viewDidLoad* está toda la configuración necesaria para la utilización de *AVFoundation*. 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, luego un dispositivo de captura y una salida de los datos y agregarlos a la sesión. También se puede configurar 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];
```

Como la clase *Isgl3dViewController* implementa el protocolo *AVCaptureVideoDataOutputSampleBufferDelegate*, una vez que comienza la sesión, se invoca cuadro a cuadro el siguiente método:

```
-(void) captureOutput:(AVCaptureOutput *)captureOutput didOutputSampleBuffer
:(CMSampleBufferRef)sampleBuffer fromConnection:(AVCaptureConnection *)connection;
```

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. Identificadores 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 una cámara, cierto poder de procesamiento y por lo general hasta una conexión a móvil internet, hace que sea cada vez más frecuente encontrar aplicaciones con el poder de reconocer QRs. Comenzaron utilizándose 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 y en algunos casos también como tarjetas personales.

1.3.2. ¿Qué son realmente los QR?

Se puede decir que los Qs 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 QR, 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 1.9 se pueden ver las distintas partes que componen un QR, como por ejemplo el bloque de control, compuesto por las tres esquinas idénticas que dan información de la posición, la información de alineamiento y el patrón de sincronismo; así como también la indicación de versión, formato y la corrección de errores. Fuera de toda esa información, que podría verse como el encabezado, haciendo analogía con los paquetes de las redes de datos, se encuentran los datos. Que podr´ian verse como el cuerpo del QR.

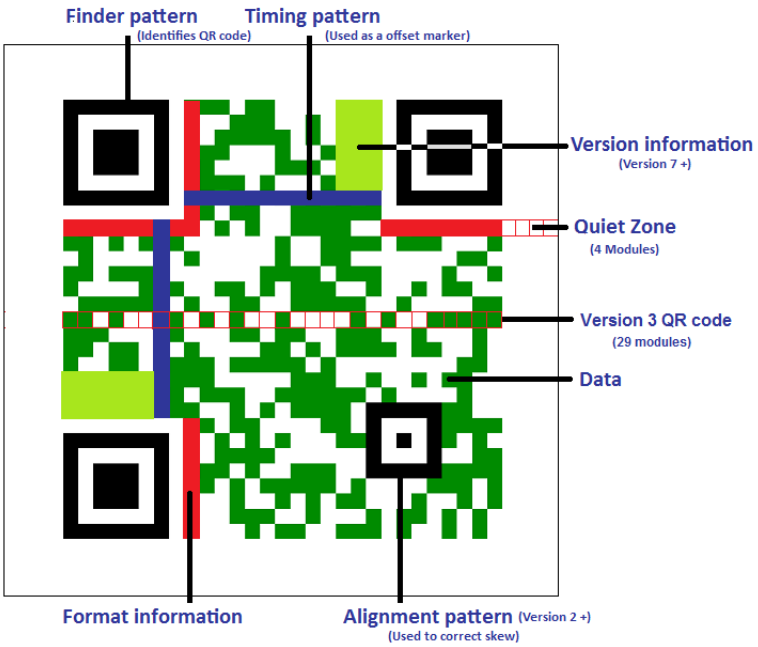

Figura 1.9: Las distintas componentes de un QR. Fuente (poner fuente).

1.3.3. Codificación y decodificación de códigos QR

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 presente proyecto no fue la codificación y decodificación de QRs, y que además ya existen distintas librerías que resuelven muy bien este problema, se optó por investigar varias de ellas e incorporar la más adecuada a la aplicación.

Entre todas las librerías que resuelven la decodificación, las llamadas ZXing y ZBar son quizá las más destacadas, por su popularidad, simplicidad y buena documentación para la fácil implementación. ZXing, denominada así por “Zebra Crossin”, es una librería gratis y en código abierto desarrollada en java y que tiene implementaciones que están adaptadas para otros lenguajes como C++, Objetive-C y JRuby, entre otros.

Por su parte ZBar también tiene soporte sobre varios lenguajes y cuenta con un kit de desarrollo interesante para lograr 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. Este 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 previa de la cámara. Luego el protocolo se encarga de la captura y procesamiento del QR almacenando como resultado la información embebida en este en la variable denominada *ZBarReaderControllerResults*. Esta va-

riable 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 cada uno de los tres artistas elegidos del Museo Nacional de Artes Visuales (MNAV). Los mismos fueron Pedro Figari, Joaquín Torres García y Juan Manuel Blanes. De esta manera, para el caso del recorrido del museo de forma automática, es posible determinar la posición del usuario utilizando 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 en 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 reducida y por lo tanto más veloz.

1.3.5. Buen gusto para los QR

La opción de usar los QR 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 modificando colores y contornos en los QR tradicionales para lograr un valor estético además del funcional. Véase en la figura 1.10 un ejemplo de cómo puede lograrse el mismo resultado pero con el valor agregado de originalidad.

Figura 1.10: Ejemplo de un QR creativo. Fuente (poner 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 impres-

cindible 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

```
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.

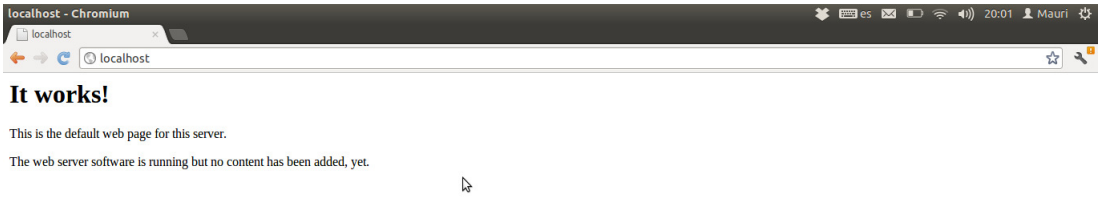

Figura 1.11: Impresión de pantalla al ingresar a http://localhost.

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

[**?**].