
Índice general

Índice general	1
1. Hardware y Software	5
1.1. Introducción	5
1.2. Software de procesamiento de imágenes	5
1.2.1. Lenguaje C	5
1.2.2. Librerías y recursos	5
1.2.2.1. OpenCV	5
1.2.2.2. IPOL	5
1.2.2.3. ITK e ImageMagik	5
1.3. Dispositivos móviles de Apple Inc.	5
1.3.1. iPhone	5
1.3.2. iPad	6
1.3.3. iPod Touch	6
1.4. Software de Apple Inc	6
1.4.1. Sistemas Operativos	6
1.4.2. Objective-C	7
1.4.3. Xcode: Herramientas y Librerías	7
1.4.3.1. Cocoa Touch Layer	8
1.4.3.2. Media Layer	10
1.4.3.3. Core Services	11
1.4.3.4. Core OS	12
1.4.3.5. Simulador	12
1.4.3.6. Instruments	12
1.5. Herramientas	13
1.5.1. GIT	13
1.5.2. GoogleCode	13
1.5.3. Github	13
2. Detección	14
2.1. Tipos de características	14
2.2. Bordes y esquinas	14
2.2.1. Detector de bordes de Canny	14
2.2.2. Detector de bordes y esquinas de Harris	14
2.2.3. SUSAN Y FAST	14
2.3. Líneas y segmentos de línea	14
2.3.1. Detector de líneas de Hough	14
2.3.2. Detector de segmentos de línea: LSD	14

2.3.3. Detector de segmentos de línea: EDLines	14
2.4. Regiones y puntos de interés	14
2.4.1. FAST	14
2.4.2. Blobs	14
2.5. Descriptores	14
3. Marcadores	15
3.1. Sistemas basados en marcadores planos	15
3.1.1. ARToolKit	16
3.1.2. ARTag	16
3.2. Marcador QR	17
3.2.1. Estructura del marcador	17
3.2.2. Diseño	18
3.2.3. Parámetros de diseño	19
3.2.4. Diseños utilizados	20
3.3. Detección	21
3.3.1. Detección de segmentos de línea	21
3.3.2. Filtrado y agrupamiento de segmentos	21
3.3.3. Determinación de correspondencias	23
3.3.4. Detección robusta	27
3.3.5. Resultados	28
4. LSD: “Line Segment Detection”	29
4.1. Introducción	29
4.2. <i>Line-support regions</i>	29
4.3. Aproximación de las regiones por rectángulos	30
4.4. Validación de segmentos	31
4.5. Refinamiento de los candidatos	32
4.6. Optimización del algoritmo para tiempo real	32
4.6.1. Filtro Gaussiano	33
4.6.2. <i>Level-line angles</i>	35
4.6.3. Refinamiento y mejora de los candidatos	35
4.6.4. Algorirmo en precisión simple	35
4.6.5. Resultados	36
4.6.5.1. Filtro Gaussiano	36
4.6.5.2. <i>Line Segment Detection</i>	36
5. Modelo de cámara y estimación de pose monocular	39
5.1. Introducción	39
5.2. Modelo de cámara <i>pin-hole</i> [1]	39
5.2.1. Fundamentos y definiciones	39
5.2.2. Matriz de proyección	41
5.3. Distorsión introducida por las lentes	44
5.4. Métodos para la calibración de cámara	44
6. POSIT: <i>POS</i> with <i>ITERations</i>	48
6.1. Introducción	48
6.2. POSIT clásico	48
6.2.1. Notación y definición formal del problema de estimación de pose	48

6.2.2. SOP: Scaled Ortographic Projection	50
6.2.3. Ecuaciones para calcular la proyección perspectiva	50
6.2.4. Algoritmo	51
6.2.5. POSIT para puntos coplanares	52
6.3. SoftPOSIT	55
6.3.1. Modern POSIT	55
6.3.2. Calculo de pose sin correspondencias	58
6.3.3. Matriz de asignación	59
6.4. Modern POSIT Coplanar	60
6.5. Resultados	60
7. Filtrado de Kalman para estimación de pose	65
7.1. Kalman clásico	65
7.2. Kalman con sensores	65
7.3. Kalman robusto	65
8. Herramientas de <i>rendering</i>	66
8.1. Introducción	66
8.2. ISGL3D	67
9. Casos de Uso	73
9.1. Introducción	73
9.2. Caso de Uso 01	73
9.2.1. Comentarios sobre el caso de uso	73
9.2.2. Detalles constructivos	73
9.3. Caso de Uso 02	73
9.3.1. Comentarios sobre el caso de uso	73
9.3.2. Detalles constructivos	73
9.3.3. <i>CGAffineTransform</i> y <i>CATransform3D</i>	74
9.3.4. Resolución de Homografía	75
9.4. Caso de Uso 03	77
9.4.1. Comentarios sobre el caso de uso	77
9.4.2. Detalles constructivos	77
9.5. Caso de Uso 04	77
9.5.1. Comentarios sobre el caso de uso	77
9.5.2. Detalles constructivos	77
10. Implementación	78
10.1. Introducción	78
10.2. Diagrama global de la aplicación	78
10.2.1. NavigationViewController	79
10.2.2. InicioViewController	80
10.2.3. TableViewControllers	80
10.2.3.1. AutorTableViewController	80
10.2.3.2. CuadroTableViewController	81
10.2.3.3. CuadroTableViewCell	81
10.2.4. ReaderSampleViewController	81
10.2.5. ImagenServerViewController	81
10.2.6. ObraCompletaViewController	83

10.2.7. VistaViewController	84
10.2.8. DrawSign	85
10.2.9. TouchVista	86
10.2.10.Isgl3dViewController y app0100AppDelegate	86
10.3. QR	88
10.3.1. QR. Una realidad	88
10.3.2. Qué son realmente los QRs?	88
10.3.3. Codificación y decodificación de QRs	90
10.3.4. El QR en la aplicación	90
10.3.5. Arte con QRs	90
10.4. Servidor	91
10.4.1. Creando el servidor	91
10.4.1.1. Servidor iOS	92
10.4.1.2. Servidor LAMP	92
10.4.2. Lenguaje php y principales scripts	92
10.5. SIFT	93
10.6. Incorporación de la realidad aumentada a la aplicación	93

Bibliografía	95
---------------------	-----------

CAPÍTULO 1

Hardware y Software

1.1. Introducción

Introducción

1.2. Software de procesamiento de imágenes

Software de procesamiento de imágenes

1.2.1. Lenguaje C

Ventajas del Lenguaje C para procesamiento de imágenes

1.2.2. Librerías y recursos

1.2.2.1. OpenCV

1.2.2.2. IPOL

1.2.2.3. ITK e ImageMagik

1.3. Dispositivos móviles de Apple Inc.

Al trabajar con Apple se cuenta con la ventaja de contar con pocas variantes en cuanto al Hardware utilizado. Básicamente existen tres tipos de dispositivos en los que se pueden desarrollar: iPhone, iPad y iPod Touch. Para cada variante de plataforma existen distintos modelos que hacen que algunas características importantes como capacidad de procesamiento, resolución de cámara o tamaño de la pantalla entre otras puedan verse afectadas. A continuación se relata el surgimiento de cada uno de los dispositivos al mercado y se describen brevemente las principales características.

1.3.1. iPhone

Sin dudas el iPhone fue uno de los saltos más grandes en el mundo tecnológico en los últimos años. Logró llenar el hueco que los PDAs de la década de los 90 no habían sabido completar y comenzó a desplazar al invento que revolucionó el mercado de los contenidos de música, el iPod. Gracias a su pantalla táctil capacitiva de alta sensibilidad logró reunir todas las funcionalidades

agregando solamente un gran botón y algunos extra para controlar volumen o desbloquear el dispositivo.

La primera generación del iPhone fue lanzada por Apple en Junio de 2007 en Estados Unidos, luego de una gran inversión de la operadora ATT que exigía exclusividad de venta dentro de dicho país durante los siguientes cuatro años. La misma soportaba tecnología GSM cuatribanda y se lanzó en dos variantes de 4GB y 8GB de ROM. El segundo modelo lanzó como novedad el soporte de tecnología 3G cuatribanda y GPS asistido. Luego le siguieron el iPhone 3GS, 4, 4S y el 5, siendo este último, la sexta y última generación disponible al momento de la redacción de este trabajo.

HACER TABLA COMPARANDO

Las dimensiones del iPhone 5 son de 58.6 x 123.8 x 7.6 millimetres, resolución de pantalla de 640 x 1136, tiene una velocidad de reloj en la CPU de 1200MHz, RAM de 1GB y la ROM varía según la variante (16GB, 30GB o 64GB).

1.3.2. iPad

Esta línea de dispositivos es la más potente en lo que respecta a capacidad de procesamiento.

1.3.3. iPod Touch

1.4. Software de Apple Inc.

1.4.1. Sistemas Operativos

Para poder desarrollar aplicaciones sobre dispositivos móviles de la firma Apple Inc es necesario contar con computadoras que corran el sistema operativo **Mac OS X**. Esto puede ser llevado a cabo, ya sea adquiriendo plataformas de desarrollo de la mencionada firma o creando máquinas virtuales que corran dicho sistema operativo. Para la segunda opción (la más económica pero con ciertas dificultades de performance), es necesario que la computadora cuente con virtualización de hardware. Se comenzó trabajando de esta manera hasta el momento de adquirir plataformas de desarrollo que contaran con Mac OS X en forma nativa.

Mac OS X refiere a la versión número 10 (en números romanos) de una serie de sistemas operativos que comenzaron a desarrollarse en la década de los 80 (Mac OS 1 data del año 1984). En los últimos 28 años se han ido sucediendo nuevas versiones que han ido mejorando características en la estructura de datos con la incorporación de la jerarquía de archivos en Mac OS 3 por ejemplo, en la búsqueda de archivos, con la simultaneidad de tareas, multiplicidad de usuarios o incluso con el énfasis en la interfaz de usuario por mencionar algunas características importantes en la evolución de esta familia de sistemas operativos. Dentro de Mac OS X existen distintas versiones, siendo la más actual la Versión 10.8: Mountain Lion lanzada durante 2012.

Por su parte todas las plataformas móviles de Apple Inc corren otro dispositivo de código cerrado: **iOS**. Originalmente llamado así por ser el sistema operativo utilizado por la plataforma iPhone, este sistema operativo está también en las plataformas iPad, iPod Touch y Apple TV en todas sus versiones. La versión más reciente de este SO es el iOS 6.1.

Una de las grandes innovaciones de estas plataformas es el hecho de poder desarrollar aplicaciones y correrlas en el propio dispositivo (por supuesto también sucede lo mismo en el mundo Android). Para poder lograr esto, es necesario como se ha dicho, contar con una máquina que corra Mac OS X y contar con el SDK apropiado llamado **Xcode**. Este entorno de desarrollo y su lenguaje se explican en la sección 1.4.2.

1.4.2. Objective-C

El lenguaje que fue elegido por Apple Inc para desarrollar sobre plataformas móviles es Objective-C. Este lenguaje fue desarrollado en la década de 1980 como un superconjunto de C orientado a objetos. Es decir que es una extensión del standard ANSI C que incorpora un modelo orientado a objetos basado en **Smalltalk**. Una de las diferencias sustanciales del modelo orientado a objetos de Objective-C respecto a otros lenguajes como Java o C++, es el hecho de la invocación de los métodos (procedimientos) de las instancias de clases. En objective-C esta invocación se da enviando mensajes, algo que se hereda de Smalltalk. Así entonces para invocar un método se procede con el siguiente código por ejemplo:

```
[receiver message];
```

Donde *receiver* es un objeto que recibe un mensaje (acción) *message* a realizar. Esta acción puede tener parámetros asociados, como por ejemplo el siguiente código real:

```
[myRectangle setWidth:20.0];
```

Esta diferencia conceptual de utilizar mensajes se representa en el hecho de que en tiempo de compilación estos mensajes no son más que una etiqueta y no están asociados al bloque de código como es el caso de Java o C++. Entonces es factible que suceda el hecho de que ese mensaje o método no esté implementado por esa clase y recién en tiempo de ejecución es que saltará el error al sustituirse esa etiqueta por un código inexistente, pues un objeto recibe un mensaje para realizar un método que no está dentro de su repertorio. Para esto es que en la documentación de Apple Inc se recomienda utilizar ciertos trucos para garantizar que el objeto que reciba el mensaje sea capaz de responder correctamente, como por ejemplo consultando primero si es capaz de realizar dicha acción y luego en caso de poder realizar dicha acción.

Otro detalle a destacar es que este lenguaje, al igual que Java también soporta la herencia múltiple. Esto es, dado un conjunto de métodos que son comunes a un conjunto de clases pero que no llegan a tener un lazo tan fuerte como para estar jerárquicamente relacionadas con una superclase común, se puede generar una clase abstracta cuyos métodos sean implementados por más de una clase sin necesidad de generar ese vínculo fuerte que es la herencia. Así como en Java existen las interfaces, que hacen esto posible, en Objective-C existen los protocolos. Existen protocolos formales e informales y con métodos obligatorios de implementar y otros opcionales. Una clase que implemente un protocolo dado tiene que tener dentro de su encabezado declarado el nombre del protocolo. Esto es:

```
@interface ClassName : ItsSuperclass < protocol list >
```

Hay otras particularidades del lenguaje pero que no van más allá de la sintaxis como los métodos de clase y los métodos de instancia, como los métodos *get* y *set* para acceder y setear atributos (propiedades) de los objetos, como la notación de *import* en lugar de *include* para quienes están acostumbrados a C y así varias detalles más. Sin embargo más allá de estas y otras diferencias y particularidades resulta un lenguaje relativamente ágil y dentro de todo sencillo de aprender para quien tiene ya un conocimiento de otros lenguajes orientados a objetos.

1.4.3. Xcode: Herramientas y Librerías

Como se dijo anteriormente el entorno de desarrollo de aplicaciones típico es Xcode, el cual es gratuito y permite compilar código C, C++, Objective-C, Objective-C++, Java y AppleScript. Xcode integra en una sola interfaz todo lo que involucra código, diseño de interfaz de usuario (**Interface Builder**) y *debugging*. También viene con un conjunto herramientas útiles para evaluar la performance de la aplicación en distintos aspectos que se llama **Instruments**. Por otra parte viene con

un conjunto importante de *Frameworks* entre los cuales se encuentran **Cocoa** y **Cocoa Touch** que proveen de herramientas útiles para desarrollar más fácilmente aplicaciones para Mac OS X e iOS respectivamente.

Las aplicaciones que corren sobre los distintos dispositivos como iPhone, iPod Touch, iPad o AppleTV están desarrolladas en Objective-C pero sobre la base de estas librerías o *Frameworks* de iOS que se pueden separar en cuatro grandes capas según el nivel de abstracción: Cocoa Touch, Media, Core Services y Core OS. Así entonces, dentro de cada capa existen distintos *Frameworks* según la

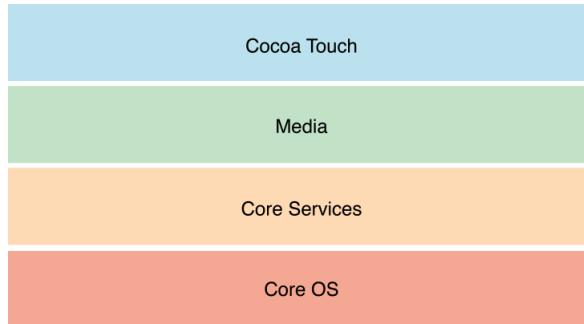


Figura 1.1: Capas de iOS

funcionalidad. A continuación se explica un poco más en detalle el rol de cada capa, los distintos *Frameworks* que tiene cada una y para qué sirven.

1.4.3.1. Cocoa Touch Layer

Cocoa Touch es la capa de más alto nivel de iOS y es la encargada de proveer al desarrollador de ciertos *Frameworks* que permitan lograr distintas tecnologías como la posibilidad de multitarea, el ingreso de órdenes a la aplicación a través de la pantalla táctil, notificaciones y alertas, preservación del estado de la aplicación al salir de la misma, reconocimiento de gestos en la pantalla y otro tipo de funcionalidades de alto nivel. Permiten al desarrollador, sin tener que involucrarse demasiado a bajo nivel, el acceso a determinados servicios que ya están resueltos en forma bastante modular.

Cocoa Touch está basado en la arquitectura **Modelo-Vista-Controlador**, en el que se separa en tres áreas distintas el modelo de la información, la interfaz de usuario y el conjunto de reglas que negocian la presentación de la información en base a la interacción con el usuario. Así pues, el usuario y una aplicación se podrían considerar dos sistemas que interaccionan. Por su parte el usuario tiene como entrada la vista de la aplicación y como salida tiene su respuesta a esta entrada, generando efectos sobre el control de la aplicación. Por otro lado la aplicación tiene como entrada las órdenes dadas por el usuario que tienen efectos sobre el modelo de la información y este sobre la vista, quien resulta ser la salida de la aplicación. Esta interacción se puede ilustrar con la figura 1.2. Como se dijo, dentro de Cocoa Touch, existen distintos *Frameworks* enfocados en permitirle al desarrollador resolver en alto nivel distintos aspectos. Los mismos son los siguientes:

- (1) Address Book UI Framework
- (2) Event Kit UI Framework
- (3) Game Kit Framework
- (4) iAd Framework
- (5) Map Kit Framework

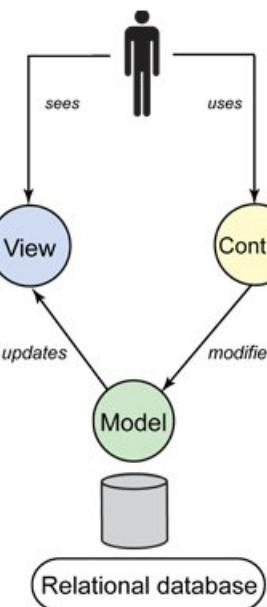


Figura 1.2: Interacción entre las tres partes del MVC

- (6) Message UI Framework
- (7) Twitter Framework
- (8) UIKit Framework

Quizá sea bueno mencionar que varias de estas API no fueron utilizadas en el presente proyecto dada su función específica y que no fueron necesarias. Sin embargo hay una en particular que tiene bastante importancia y que permite la mayoría de las funcionalidades básicas que toda aplicación tiene. Se trata del **UIKit**, encargado de gestionar la aplicación, su interfaz de usuario y gráficos, encargado soportar eventos frente al toque de la pantalla, de manejar sensores como el acelerómetro y giroscopio, y de tener acceso a la cámara y galería de fotos entre lo más importante a destacar. El soporte de la multitarea y de **Storyboards** también está a cargo de este *Framework*.

Hay funcionalidades que han ido cambiando con las distintas versiones de iOS. Una de ellas y quizás una que ha generado bastantes diferencias respecto a versiones anteriores a iOS 5, es esta última, el Storyboard, una herramienta muy útil de programación gráfica, que permite generar instancias de clases y vínculos entre las mismas en forma visual a la vez de ser una contraparte de interfaz de usuario. Con una biblioteca de objetos disponibles, listos para ser usados, mediante el uso de Storyboard se hace accesible con algunas horas de dedicación implementar aplicaciones sencillas. Esta herramienta vino para sustituir los archivos .nib que permitían diseñar la interfaz pero no tantas funcionalidades programáticas como el Storyboard. En particular éste último permite agregar la funcionalidad de *segues*, encargados de vincular un *ViewController* con otro. Este tipo de diferencias vinieron con la idea de evitar la necesidad de implementar ciertos bloques de código en forma repetitiva. Un Storyboard luce como en la figura 1.2.

Si bien se podría extender bastante más la explicación sobre los detalles de Cocoa Touch, a los efectos del presente proyecto, no es de tanta relevancia excederse en este punto.

1.4.3.2. Media Layer

Media Layer es la capa encargada de gestionar correctamente elementos multimedia y es posible distinguir tres grandes grupos que engloban distintos *Frameworks*: Gráficos, Audio y Video.

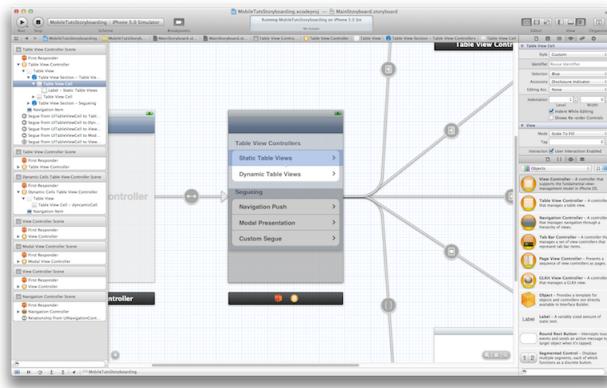


Figura 1.3: Ejemplo de Storyboard.

Dentro de las tecnologías más destacadas está todo lo vinculado a **gráficos** 2D y 3D, dentro de lo que se puede incluir algunos *Frameworks* bastante utilizados en el presente proyecto, tales como: **Core Graphics**, muy utilizado para dibujos 2D, **Quartz Core**, quien contiene las herramientas necesarias para interactuar con otro *Framework* para animación de vistas, de una capa de más bajo nivel como *Core Animation*, que es comentado más adelante en la sección 1.4.3.3. También es parte de lo vinculado a gráficos, el *Framework Core Image*, contenido lo vinculado a procesamiento de imágenes a través filtros que utilizan directamente la unidad de procesamiento de gráficos (GPU) y otros dos *Frameworks* bastante importantes en lo que respecta a *rendering* como **OpenGL ES** y **GLKit** (utilizado por el motor de juegos *Isgl3d* entre otros).

Por otra parte hay otra gran familia de *Frameworks* dentro de Media Layer que apunta a resolver todo lo vinculado al manejo de audio, ya sea de grabación como procesamiento y reproducción de alta calidad. Existen algunos SDK como **iSpeech** o **Dragon Mobile** que resuelven de manera similar al proyecto Siri, el procesamiento de la voz humana reconociendo palabras e interpretando, que utilizan algunos de los *Frameworks* de procesamiento de audio de esta familia.

En cuanto a lo vinculado al manejo de video, como parte de esta capa, se tienen dos *Framework* importantes con distintos niveles de abstracción: **MediaPlayer** y **AVFoundation**. También existen otros *Frameworks* fuera de esta capa que son capaces de manejar video como la clase `UIImagePickerController` (muy utilizada en el proyecto) del mencionado `UIKit`. En la figura 1.4 se esquematiza el nivel de abstracción de los *Frameworks* de las distintas capas que son capaces de manejar multimedia. Con `MediaPlayer` es posible reproducir audio y video muy fácilmente en determinada área

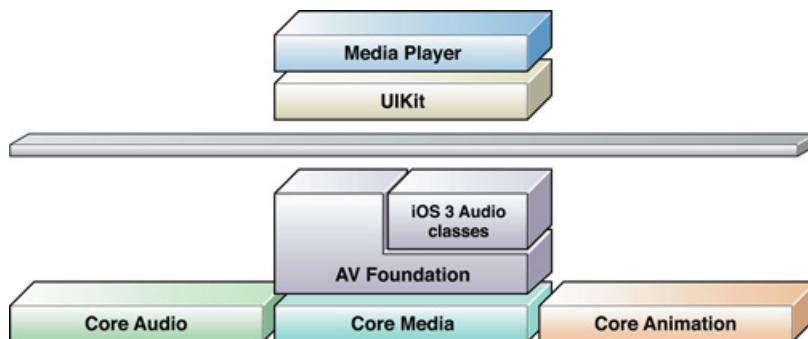


Figura 1.4: Frameworks de las distintas capas para manejo de video

de la pantalla ya sea desde un URL o de un archivo, es posible mostrar o no los elementos de control del video así como también controlar, volumen y tamaño de la pantalla. Por su parte, con `AVFoundation` es posible capturar con la cámara, reproducir, editar y procesar audio y video. Es posible

implementar ciertos protocolos que hace de esto algo relativamente sencillo.

1.4.3.3. Core Services

Core Services es la capa de más bajo nivel de iOS y contiene los elementos fundamentales sobre los que se construyen las capas superiores. Es posible que al comenzar a programar para iOS no se tenga mucha interacción con esta capa pero sin embargo existen algunos conceptos importantes de esta capa que sí vale la pena mencionar dado que en el presente proyecto se tuvieron que entender y discutir. Una de ellas es la *Automatic Reference Counting* o **ARC**. Esta funcionalidad compete a la reserva y liberación de memoria por parte de los objetos. La idea básica es lograr que el uso de memoria sea el mínimo posible, logrando que los objetos existan en la medida que son necesarios y que su memoria sea liberada ni bien sea posible. Típicamente, al crear una instancia de un objeto se

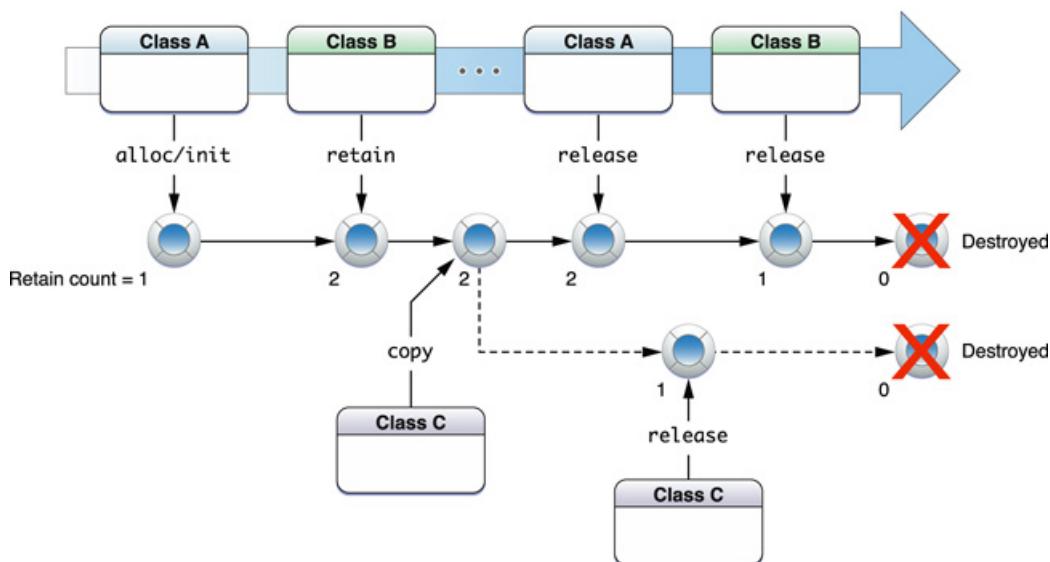


Figura 1.5: Ciclo de vida de objetos, Manual-Retain-Release.

incrementa un contador y al liberar se decrementa y entonces se tiene cierto control sobre la reserva y liberación de memoria en base al contador. Sin embargo, la liberación de memoria reservada por objetos queda bajo la responsabilidad del desarrollador y en casos de un código complejo puede llegar a ser habitual olvidarse de la liberación de memoria. Lo anterior refiere a una gestión manual de la reserva y liberación conocido como *manual retain-release*. Para no tener que enfrentar este tema y poder instanciar clases sin tener presente la posterior liberación de memoria (pues quizás se sepa cuándo no será más necesario un objeto o no), se utiliza ARC. Esta funcionalidad evalúa el ciclo de vida de los objetos y agrega código en tiempo de compilación en caso de considerarlo necesario. Es bueno aclarar que esto refiere a memoria reservada pura y exclusivamente por objetos, es decir mediante *alloc*. En caso de tratarse de memoria reservada para variables de lenguaje C (*malloc*), es necesario proceder de igual manera que en dicho lenguaje, liberando la memoria mediante un *free*. Además del ARC, Core Services permite el manejo de archivos XML y manejo de base de datos SQL así como también la protección de datos cuando el dispositivo está bloqueado entre otros servicios importantes. Tiene varios *Frameworks* como **Core Media** que logran un nivel aún más bajo que AVFoundation para el manejo de multimedia, **Quick Look** para las vistas previas de archivos, **Social** que viene a suplantar el *Framework* para la utilización de Twitter que existe en iOS 5 y extiende la gestión para otras redes sociales, **Core Motion** para el manejo de sensores como el acelerómetro y el giroscopio, **Core Telephony** para el manejo de la información de red del dispositivo

como elemento de la red de telefonía, **CFNetwork** para el manejo de protocolos de red como http, https, ftp y resolución de servidores DNS, entre otros *Frameworks* importantes dentro de la capa.

1.4.3.4. Core OS

Con esta capa de iOS en general es difícil que el desarrollador tenga que involucrarse directamente dado que es la de más bajo nivel. Salvo que se esté frente a una aplicación que requiera aspectos de seguridad o comunicación con HW externo, esta capa solamente existe para ser la base sobre la cual se desarrollan los *Frameworks* de las capas de más alto nivel. Los distintos *Frameworks* que tiene están enfocados en resolver temas de procesamiento basados en el hardware de iOS, en comunicarse con dispositivos externos basados en iOS y de garantizar la seguridad de los datos de una aplicación.

1.4.3.5. Simulador

Uno de los detalles más importantes del entorno de desarrollo es la capacidad de simular lo que se programa antes de probarlo en un dispositivo. Esto es útil por cuestiones de seguridad e incluso permite programar sin la necesidad a priori de contar con una plataforma. Esto existe para *Xcode* y es necesario decir que funciona muy bien, generando una representación bastante fiel de lo que sucede en el dispositivo real. La única crítica que se le podría hacer es el hecho de no contar con cámara y para el caso de aplicaciones de realidad aumentada esto es algo bastante importante. Sin embargo, sin ser eso, el simulador cuenta con conexión a internet, pantalla multitáctil, con información de GPS ingresada por el programador, acceso a la galería de fotos, capacidad de procesamiento y todas las funcionalidades que un dispositivo real tiene.

1.4.3.6. Instruments

Dentro de las herramientas que vienen con el entorno de desarrollo viene *Instruments*, un set de herramientas que permiten analizar la performance de una aplicación para iOS o para Mac OS X desde distintos puntos de vista. Resulta muy útil pues muchas veces sucede que una aplicación compila y se ejecuta correctamente y sin embargo puede el desarrollador puede no estar conforme en cuanto a los tiempos de procesamiento o el uso de memoria consumido.

En el presente proyecto se hizo uso principalmente del **Time Profiler** que permite analizar tiempos y del **Memory Leak** que permite hacer un análisis de la reserva de memoria no liberada.

El **Time Profiler**....

El **Memory Leak** ...

1.5. Herramientas

Herramientas

1.5.1. GIT

1.5.2. GoogleCode

1.5.3. Github

CAPÍTULO 2

Detección

2.1. Tipos de características

2.2. Bordes y esquinas

2.2.1. Detector de bordes de Canny

2.2.2. Detector de bordes y esquinas de Harris

2.2.3. SUSAN Y FAST

2.3. Líneas y segmentos de línea

2.3.1. Detector de líneas de Hough

2.3.2. Detector de segmentos de línea: LSD

2.3.3. Detector de segmentos de línea: EDLines

2.4. Regiones y puntos de interés

2.4.1. FAST

2.4.2. Blobs

2.5. Descriptores

SIFT (puntero a capítulo que tiene SIFT para reconocimiento o mismo acá)
SURF, ETC ETC.

CAPÍTULO 3

Marcadores

La inclusión de *marcadores*, en inglés *markers*, en la escena ayuda al problema de extracción de características y por lo tanto al problema de estimación de pose [9]. Estos por construcción son elementos que presentan una detección estable en la imagen para el tipo de característica que se desea extraer así como medidas fácilmente utilizables para la estimación de la pose.

Los marcadores planos se pueden obtener mediante la construcción en una geometría coplanar de una serie de primitivas identificables como esquinas, segmentos o líneas. Un único marcador plano puede contener por si solo todas las seis restricciones espaciales necesarias para definir un marco de coordenadas asociado a su pose.

Como se explica en la sección ?? el problema de estimación de pose requiere de una serie de correspondencias $\mathbf{M}_i \leftrightarrow \mathbf{m}_i$ entre puntos 3D en la escena en coordenadas del mundo y puntos en la imagen.

En el primer lugar se explican brevemente algunos de los sistemas de Realidad Aumentada mas populares basados en marcadores planos. En segundo lugar se propone el diseño de un marcador específico para la aplicación a este proyecto y se desarrollan las soluciones a los algoritmos de detección de dicho marcador mostrando algunos resultados parciales en el proceso. Por último se muestran algunos resultados de la detección.

3.1. Sistemas basados en marcadores planos

Existen muchos sistemas de visión basados en *marcadores planos* con aplicación en Realidad Aumentada y Navegación. Algunos de ellos son ARToolKit, ARTag y ARStudio utilizados para Realidad Aumentada ?? . A continuación se realiza una breve descripción del funcionamiento de los mismos.

Los sistemas basados en marcadores planos utilizan típicamente marcadores bitonales. Esto permite reducir la sensibilidad a las condiciones de luz de la escena y a las configuraciones de la cámara por lo que no hay necesidad de identificar tonos de grises y la regla de decisión para cada píxel puede ser reducida, en la versión mas simple, a un umbral o *threshold*. El diseño de los marcadores depende en gran medida de la aplicación. En la figura 3.1 se muestran algunos marcadores planos para aplicaciones de Realidad Aumentada en donde cada uno de ellos provee suficientes puntos para permitir el cálculo de pose tridimensional y adicionalmente contienen cierta información en su interior para permitir su identificación.

Es importante que los marcadores puedan ser localizados en un campo de visión amplio para permitir la correcta detección bajo la distorsión asociada a la transformación proyectiva que lleva el marcador en el mundo real al plano de imagen. Por otro lado, si los marcadores contienen informa-

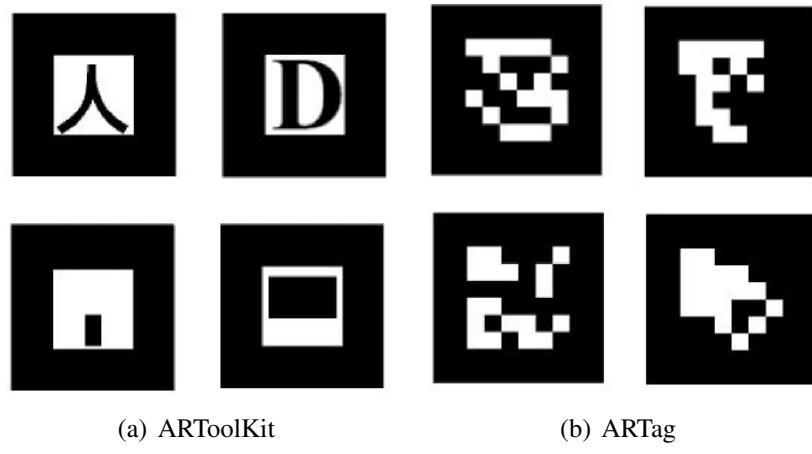


Figura 3.1: Cuatro ejemplos de marcadores para los sistemas de Realidad Aumentada indicados. Figura tomada de ??

ción en su interior, esta no debe ser muy densa para permitir la recuperación de la misma a mayor distancia. Típicamente, esta información es solo una identificación entre marcadores de un mismo sistema por lo que la información es poca y esto no es un problema.

Estos sistemas contienen ciertos puntos característicos con los que se realiza el cálculo de pose. En general su contorno es basado en un cuadrilátero y se utilizan las cuatro esquinas del contorno del marcador para realizar el cálculo.

3.1.1. ARToolKit

ARToolKit es un muy popular sistema de marcadores planos para Realidad Aumentada e Interacción Hombre Computador debido a ser de código abierto. Los marcadores bitonales consisten en un cuadrado con borde negro y un patrón en el interior.

La primer etapa del proceso de reconocimiento consisten en detectar los bordes negros. Esto se realiza buscando grupos conexos de píxeles (*blobs*) que están por debajo de un determinado umbral. Posteriormente se extrae el contorno de cada grupo esos grupos que están rodeados por cuatro líneas rectas son marcados como marcadores potenciales. Las cuatro esquinas de cada marcador potencial son utilizados para calcular la homografía y así remover la distorsión perspectiva. Con el marcador en una vista canónica, se procede a identificar el patrón interno muestreando en una grilla, de por lo general 16×16 o 32×32 , los valores de gris. Con esto se construye un vector característico y se compara por correlación con una librería de vectores de característicos logrando la identificación del mismo.

Este sistema tiene algunas desventajas o “puntos débiles”. En primer lugar la detección es basada en un umbral por lo que las condiciones de iluminación pueden afectar fuertemente la efectividad de la misma. Dado que el código esta disponible este se puede modificar para realizar *threshold* local o adaptivo por ejemplo. Otras desventajas están relacionadas con el proceso de identificación del marcador frente a la librería.

3.1.2. ARTag

ARTag es otro sistema de marcadores planos para Realidad Aumentada e Interacción Humano Computador. Los marcadores son también bitonales y basados en un borde negro. En contraste con el ARToolKit este sistema utiliza un enfoque basado en bordes por lo que es mas robusto a condi-

ciones de iluminación. Los bordes son unidos en segmentos que a su vez se unen en cuadriláteros. Al igual que con ARToolKit con las esquinas se calcula la homografía y se muestrea en el interior del marcador pero con una grilla de 6×6 .

El sistema puede lidiar con condiciones de iluminación cambiantes, oclusiones y segmentos partidos hasta cierto punto. El proceso de identificación de los marcadores entre sí con la información en su interior es a su vez más veloz que el de ARToolKit.

3.2. Marcador QR

El enfoque elegido para la detección de características utilizando marcadores parte del trabajo de fin de curso de *Matías Tailanián* para el curso *Tratamiento de imágenes por computadora*¹. La elección se basa principalmente en los buenos resultados obtenidos para dicho trabajo con un enfoque relativamente simple. El trabajo desarrolla, entre otras cosas, un diseño de marcador y un sistema de detección de marcadores basado en el detector de segmentos LSD[7] por su buena *performance*.

El marcador utilizado está basado en la estructura de detección incluida en los códigos *QR* y se muestra en la figura 3.2. Éste consiste en tres grupos idénticos de tres cuadrados concéntricos superpuestos en “capas”. La primer capa contiene el cuadrado negro de mayor tamaño, en la segunda capa ubica el cuadrado mediano en color blanco y en la última capa un cuadrado negro pequeño. De esta forma se logra un fuerte contraste en los lados de cada uno de los cuadrados facilitando la detección de bordes o líneas. El resultado de una detección de líneas para esta configuración produce para cada cuadrado la detección de sus lados. A diferencia de los códigos *QR* la disposición espacial de los grupos de cuadrados es distinta para evitar ambigüedades en la identificación de los mismos entre sí.

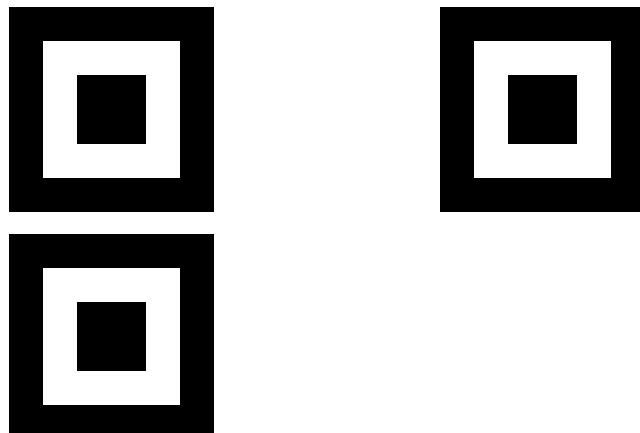


Figura 3.2: Marcador propuesto basado en la estructura de detección de códigos QR.

3.2.1. Estructura del marcador

A continuación se presentan algunas definiciones de las estructuras básicas que componen el marcador propuesto. Estas son de utilidad para el diseño y forman un flujo natural y escalable para el desarrollo del algoritmo de determinación de correspondencias.

¹Autoposicionamiento 3D - <http://sites.google.com/site/autoposicionamiento3d/>

Los elementos más básicos en la estructura son los *segmentos* los cuales consisten en un par de puntos en la imagen, $\mathbf{p} = (p_x, p_y)$ y $\mathbf{q} = (q_x, q_y)$. Estos *segmentos* forman lo que son los lados del *cuadrilátero*, el próximo elemento estructural del marcador.

Un *cuadrilátero* o *quadrilateral* en inglés, al que se le denomina Ql , está determinado por cuatro segmentos conexos y distintos entre sí. El cuadrilátero tiene dos propiedades notables; el *centro* definido como el punto medio entre sus cuatro vértices y el *perímetro* definido como la suma de el largo de sus cuatro lados. Los *vértices* de un cuadrilátero se determinan mediante la intersección, en sentido amplio, de dos segmentos contiguos. Es decir, si s_1 es contiguo a s_2 dadas las recta r_1 que pasa por los puntos $(\mathbf{p}_1, \mathbf{q}_1)$ del segmento s_1 y la recta r_2 que pasa por los puntos $(\mathbf{p}_2, \mathbf{q}_2)$ del segmento s_2 , se determina el vértice correspondiente como la intersección $r_1 \cap r_2$.

A un *conjunto de cuadriláteros* o *quadrilateral set* se le denomina $QlSet$ y se construye a partir de M cuadriláteros, con $M > 1$. Los cuadriláteros comparten un mismo centro pero se diferencian en un factor de escala. A partir de dichos cuadriláteros se construye un lista ordenada $(Ql[0], Ql[1], \dots, Ql[M - 1])$ en donde el orden viene dado por el valor de perímetro de cada Ql . Se define el *centro del grupo de cuadriláteros*, \mathbf{c}_i , como el promedio de los centros de cada Ql de la lista ordenada.

Finalmente el *marcador QR* está constituido por N conjuntos de cuadriláteros dispuestos en una geometría particular. Esta geometría permite la determinación de un sistema de coordenadas; un origen y dos ejes a utilizar. Se tiene una lista ordenada $(QlSet[0], QlSet[1], \dots, QlSet[N - 1])$ en donde el orden se puede determinar mediante la disposición espacial de los mismos o a partir de hipótesis razonables.

Un marcador proveerá un numero de $4 \times M \times N$ vértices y por lo tanto la misma cantidad de puntos para proveer las correspondencias $\mathbf{M}_i \leftrightarrow \mathbf{m}_i$ al algoritmo de estimación de pose.

3.2.2. Diseño

En base a las estructuras previamente definidas es que se describe el diseño del marcador. Como ya se explicó se toma un marcador tipo QR basado en cuadriláteros y mas específicamente en tres conjuntos de tres cuadrados dispuestos en como se muestra en la figura 3.2.

Los tres cuadriláteros correspondientes a un mismo conjunto de cuadriláteros tienen idéntica alineación e idéntico centro. Los diferencia un factor de escala, esto es, $Ql[0]$ tiene lado l mientras que $Ql[1]$ y $Ql[2]$ tienen lado $2l$ y $3l$ respectivamente. Esto se puede ver en la figura 3.3. Adicionalmente se define un sistema de coordenadas con centro en el centro del $QlSet$ y ejes definidos como x horizontal a la derecha e y vertical hacia abajo. Esta convención en las direcciones de los ejes es muy utilizada en el área de Procesamiento de Imágenes para definir las direcciones de los ejes de una imagen. Definido el sistema de coordenadas de puede fijar un orden a los vértices v_{j_1} de cada cuadrilátero $Ql[j]$ como,

$$\begin{aligned} v_{j_0} &= (a/2, a/2) & v_{j_2} &= (-a/2, -a/2) \\ v_{j_1} &= (a/2, -a/2) & v_{j_3} &= (-a/2, a/2) \end{aligned}$$

con $a = (j + 1) \times l$. El orden aquí explicado se puede ver también junto con el sistema de coordenadas en la figura 3.4.

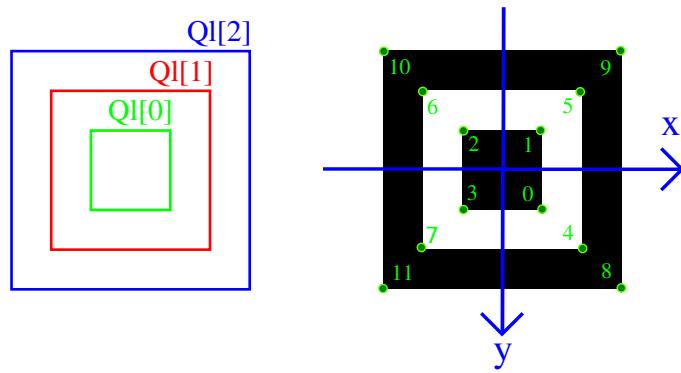


Figura 3.3: Detalle de un *QlSet*. A la izquierda se muestra el resultado de la detección de un *QlSet* y el orden interno de sus cuadriláteros y a la derecha el orden de los vértices respecto al sistema de coordenadas local.

Un detalle del marcador completo se muestra en la figura 3.4 en donde se define el conjunto *i* de cuadriláteros concéntricos como el $QlSet[i]$ y se definen los respectivos centros de cada uno de ellos como \mathbf{c}_i . El sistema de coordenadas del marcador QR tiene centro en el centro del $QlSet[0]$ y ejes de coordenadas idénticos al definido para cada *Ql*. Se tiene además que los ejes de coordenadas pueden ser obtenidos mediante los vectores normalizados,

$$\mathbf{x} = \frac{\mathbf{c}_1 - \mathbf{c}_0}{\|\mathbf{c}_1 - \mathbf{c}_0\|} \quad \mathbf{y} = \frac{\mathbf{c}_2 - \mathbf{c}_0}{\|\mathbf{c}_2 - \mathbf{c}_0\|} \quad (3.1)$$

La disposición de los *QlSet* es tal que la distancia indicada d_{01} definida como la norma del vector entre los centros \mathbf{c}_1 y \mathbf{c}_0 es significativamente mayor que la distancia d_{02} definida como la norma del vector entre los centros \mathbf{c}_2 y \mathbf{c}_1 . Esto es, $d_{01} \gg d_{02}$. Este criterio facilita la identificación de los *QlSet* entre sí basados únicamente en la posición de sus centros y es explicado en la sección de determinación de correspondencias (sec.: 3.3.3).

3.2.3. Parámetros de diseño

Provisto el diseño del marcador descrito, quedan definidos ciertos parámetros **estructurales** que fueron de tomados fijos a lo largo del proyecto pero que podrían ser cambiados para trabajos futuros asociados. Estos parámetros son:

- *M*: cantidad de conjuntos de cuadriláteros.
- *N*: cantidad de cuadriláteros por conjuntos de cuadriláteros.
- Geometría: geometría de los cuadriláteros (*Ql*).
- Disposición: disposición espacial de los conjuntos de cuadriláteros (*QlSet*).

El criterio de elección de *M* y *N* parte del diseño los códigos QR como ya fue explicado. La detección por segmentos de línea resulta una cantidad de $3 \times QlSet's$ conteniendo $3 \times Ql's$ cada uno. Bajo esta elección de parámetros se tienen 36 segmentos y vértices. Se tiene entonces un número de puntos característicos razonable para la estimación de pose.

La elección de *cuadrados* como parámetro de geometría se basa en la necesidad de tener igual resolución en los dos ejes del marcador. De esta forma se asegura una distancia límite en donde, en un caso ideal enfrentado al marcador, la detección de segmentos de línea falla simultáneamente en

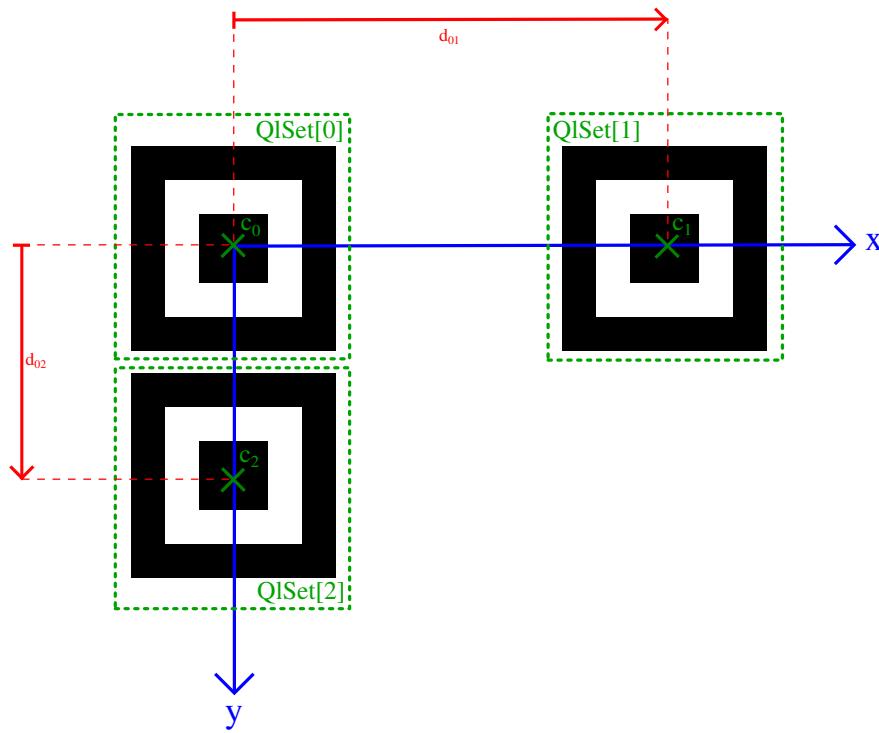


Figura 3.4: Detalle del marcador propuesto formando un sistema de coordenadas.

los segmentos verticales como en los horizontales. De otra forma se tendría una dirección que limita mas que la otra desaprovechando resolución.

La disposición espacial de los conjuntos de cuadriláteros esta en primer lugar limitada a un plano y en segundo lugar es tal que se puede definir ejes de coordenadas ortogonales mediante los centros como se muestra en la figura 3.4.

Por otro lado se tiene otro juego de parámetros **dinámicos** que concluyen con el diseño del marcador. Estos parámetros conservan la estructura intrínseca del marcador permitiendo versatilidad en la aplicación y sin la necesidad de modificación alguna de los algoritmos desarrollados. Estos son:

- d_{ij} : distancia entre los centros $QlSet[j]$ con $QlSet[i]$.
- l : lado del cuadrilátero mas pequeño ($Ql[0]$) de los $QlSet$.

En este caso se debe cumplir siempre la condición impuesta previamente en donde $d_{01} \gg d_{02}$. De otra forma se deberán realizar ciertas hipótesis no genéricas o se deberá aumentar ligeramente la complejidad del algoritmo para la identificación del marcador.

3.2.4. Diseños utilizados

- **Test:** Durante el desarrollo de los algoritmos de detección e identificación de los vértices del marcador QR se trabajó con determinados parámetros de diseño de dimensiones apropiadas para posibilitar el traslado y las pruebas domésticas.

- $l = 30mm$
- $d_{01} = 190mm$

- $d_{02} = 100mm$

- **Da Vinci**
- **Artigas**
- **Mapa**

3.3. Detección

La etapa de detección del marcador se puede separar en tres grandes bloques; la detección de segmentos de línea, el filtrado de segmentos y la determinación de correspondencias (figura ??). En esta sección se muestran algunos resultados para la detección de segmentos de línea por LSD y se desarrolla en profundidad los algoritmos desarrollados durante el proyecto para el filtrado de segmentos y determinación de correspondencias.

3.3.1. Detección de segmentos de línea

La detección de segmentos de línea se realiza mediante el uso del algoritmo LSD el cual se detalla en el capítulo ???. En forma resumida, dicho algoritmo toma como entrada una imagen en escala de grises de tamaño $W \times H$ y devuelve una lista de segmentos en forma de pares de puntos de origen y destino.

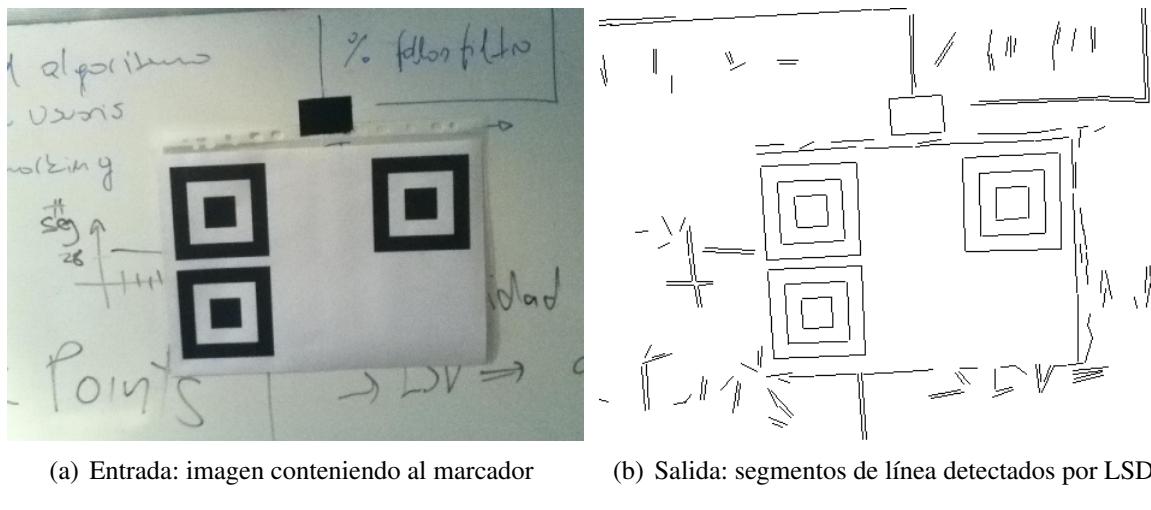


Figura 3.5: Resultados del algoritmo de detección de segmentos de línea LSD.

3.3.2. Filtrado y agrupamiento de segmentos

El filtrado y agrupamiento de segmentos consiste en la búsqueda de conjuntos de cuatro segmentos conexos en la lista de segmentos de línea detectados por LSD. Los conjuntos de segmentos conexos encontrados se devuelven en una lista en el mismo formato a la de LSD pero agrupados de a cuatro. A continuación se realiza una breve descripción del algoritmo de filtrado de segmentos implementado.

Se parte de una lista de m segmentos de línea,

$$\mathbf{L} = (\mathbf{s}_0 \quad \mathbf{s}_1 \quad \dots \quad \mathbf{s}_{m-1})^t \quad (3.2)$$

y se recorre en i en busca de segmentos vecinos. La estrategia utilizada consiste en buscar, para el i -ésimo segmento \mathbf{s}_i , dos segmentos vecinos. En una primera etapa \mathbf{s}_j y en una segunda etapa \mathbf{s}_k , de forma que se forme una “U” como se muestra en la figura 3.6. La tercera etapa de búsqueda consiste en completar ese conjunto con un cuarto segmento \mathbf{s}_l que cierre la “U”.

Dos segmentos \mathbf{s}_i y \mathbf{s}_j son vecinos si se cumple que la distancia euclíadiana entre puntos, d_{ij} , es menor a un cierto umbral para alguna de las combinaciones $\mathbf{p}_i \leftrightarrow \mathbf{p}_j$, $\mathbf{q}_i \leftrightarrow \mathbf{q}_j$, $\mathbf{p}_i \leftrightarrow \mathbf{q}_j$ o $\mathbf{q}_i \leftrightarrow \mathbf{p}_j$. En la primera etapa de la búsqueda se testean todas las posibilidades mientras que en la segunda etapa se testean solo los puntos del segmento que no fueron utilizados. Por ejemplo, si se encontró la correspondencia $\mathbf{p}_i \leftrightarrow \mathbf{p}_j$ se busca el k -ésimo segmento \mathbf{s}_k que cumple que la distancia euclíadiana d_{ik} es menor a cierto umbral para alguna de las combinaciones $\mathbf{q}_i \leftrightarrow \mathbf{p}_k$ y $\mathbf{q}_i \leftrightarrow \mathbf{q}_k$. En la tercera etapa la chequeo se realiza de forma mas aún mas restringida probando para el segmento \sim_l correspondencia simultanea entre sus puntos y solamente un punto cada uno de los segmentos \sim_j y \sim_k .

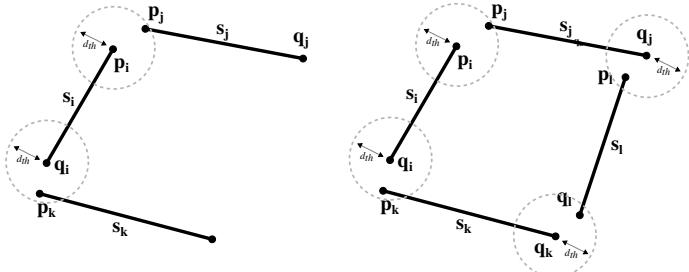


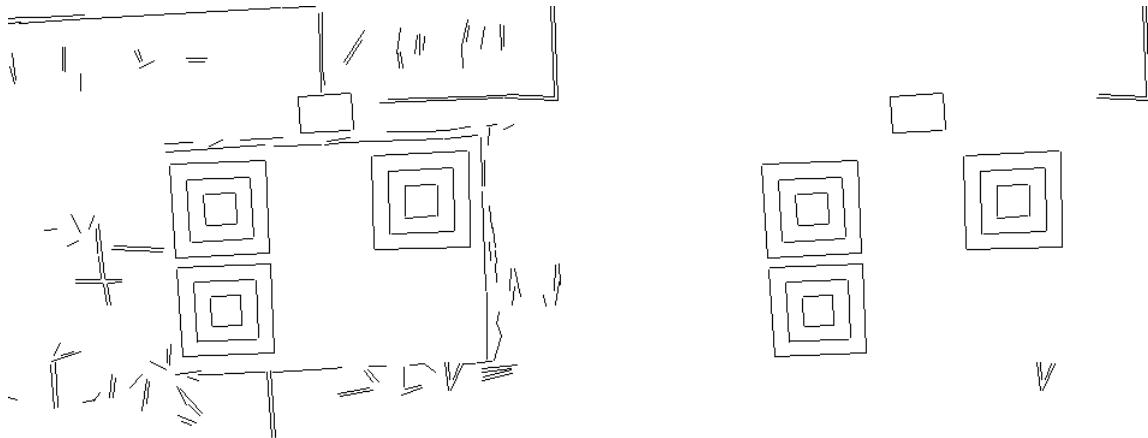
Figura 3.6: Conjunto de cuadriláteros conexos. A la izquierda la primera y segunda etapa del filtrado completadas para el segmento \sim_i en donde se busca una “U”. A la derecha la última etapa en donde se cierra la “U” con el segmento \sim_l .

Una vez encontrado el conjunto de cuatro segmentos conexos se marcan estos segmentos como utilizados, se guardan en una lista de salida y se continúa con el segmento $i + 1$ hasta recorrer los m segmentos de la lista de entrada. De esta forma se obtiene una lista de salida \mathbf{S} de n segmentos en donde n es por construcción múltiplo de cuatro.

En la figura 3.3.2 se muestran los resultados obtenidos para el algoritmo tomando como entrada la lista de segmentos de LSD. Se puede ver que los lados de los cuadrados del marcador son detectados correctamente pero también hay otras detecciones presentes. Por ejemplo el rectángulo negro correspondiente a un trozo de cinta negra que sostén el marcador (ver figura ??(a)). También sobreviven otro tipo de elementos indeseados que se explican a continuación.

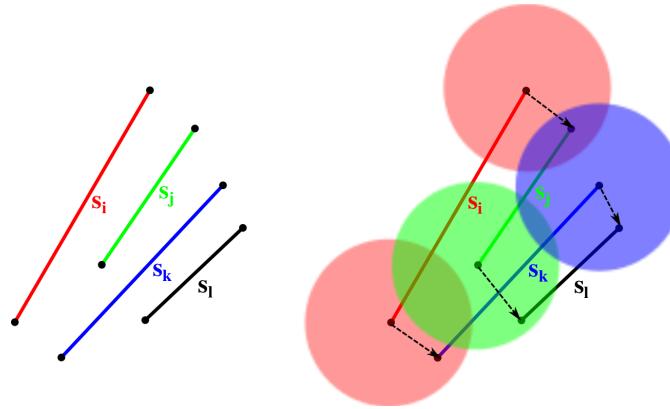
El algoritmo descrito es simple y provee resultados aceptables en general pero es propenso a tanto a detectar *falsos positivos* como al *sobre-filtrado* algunos conjuntos.

La detección de falsos positivos se puede atribuir principalmente a la condición de vecindad utilizada en donde un caso como el que se muestra en la figura 3.8 de un conjunto de segmentos paralelos cercanos y de tamaño similar “sobrevive” al filtrado de segmentos. De forma de evitar estos falsos positivos, se podría considerar implementar una condición de vecindad que tome en cuenta el punto de intersección entre los segmentos y la distancia de este punto a los puntos \mathbf{p} , \mathbf{q} mas cercanos de cada segmento. Como se explicará en la sección ??, debido a que el algoritmo de determinación de correspondencias realiza la intersección entre estos segmentos se puede chequear alguna condición sobre los segmentos o su intersección y en ese momento filtrar estos casos.



(a) Entrada: segmentos de línea detectados por LSD (b) Salida: segmentos de línea filtrados y agrupados

Figura 3.7: Resultados del algoritmo de filtrado y agrupamiento de segmentos de línea.

Figura 3.8: Posible configuración de segmentos paralelos que “sobreviven” al filtrado. A la izquierda el grupo de segmentos, a la derecha se muestra como se desarrolla el filtrado de s_i .

El sobre-filtrado de segmentos tiende a ocurrir cuando no se cumple la condición de distancia entre segmentos vecinos cuando visualmente si lo son. Se debe principalmente a que se utiliza para el filtrado un valor de d_{th} fijo que resulta en buenos resultados para la aplicación pero en ciertas circunstancias produce este problema. Esta medida de distancia se podría tomar relativa al largo del los segmentos a *testear* de forma de generalizar el valor pero se debería analizar un poco mas en detalle la posible implementación para que resulte en buenos resultados y no introduzca otra clase de errores.

3.3.3. Determinación de correspondencias

Se detalla a continuación el algoritmo de determinación de correspondencias a partir de grupos de cuatro segmentos de línea conexos. Para ese algoritmo se hace uso de los elementos estructurales del marcado (sec.: 3.2.1), de forma de desarrollar un algoritmo modular, escalable y simple.

Se toma como entrada la lista de segmentos filtrados y agrupados

$$\mathbf{S} = (\mathbf{s}_0 \ \mathbf{s}_1 \ \dots \ \mathbf{s}_i \ \mathbf{s}_{i+1} \ \mathbf{s}_{i+2} \ \mathbf{s}_{i+3} \ \dots \ \mathbf{s}_{n-1})^t \quad (3.3)$$

en donde cada segmento se compone de un punto inicial \mathbf{p}_i y un punto final \mathbf{q}_i , $\mathbf{s}_i = (\mathbf{p}_i, \mathbf{q}_i)$, con n múltiplo de cuatro. Si i también lo es, entonces el sub-conjunto, $\mathbf{S}_i = (\mathbf{s}_i \ \mathbf{s}_{i+1} \ \mathbf{s}_{i+2} \ \mathbf{s}_{i+3})^t$, corresponde a un conjunto de cuatro segmentos del línea conexos.

Para cada sub-conjunto \mathbf{S}_i se intersectan entre sí los segmentos obteniendo una lista de cuatro vértices, $\mathbf{V}_i = (\mathbf{v}_i \ \mathbf{v}_{i+1} \ \mathbf{v}_{i+2} \ \mathbf{v}_{i+3})^t$. Si \mathbf{r}_i es la recta que pasa por los puntos \mathbf{p}_i y \mathbf{q}_i del segmento s_i , la lista de vértices se obtiene como sigue,

$$\begin{aligned}\mathbf{v}_i &= \mathbf{r}_i \cap \mathbf{r}_{i+1} \\ \mathbf{v}_{i+1} &= \mathbf{r}_i \cap \mathbf{r}_{i+2} \\ \mathbf{v}_{i+2} &= \mathbf{r}_{i+3} \cap \mathbf{r}_{i+2} \\ \mathbf{v}_{i+3} &= \mathbf{r}_{i+3} \cap \mathbf{r}_{i+1}\end{aligned}$$

resultando en dos posibles configuraciones de vértices. Las dos configuraciones se muestran en la figura 3.9 en donde una de ellas tiene sentido horario y la otra antihorario partiendo de v_i .

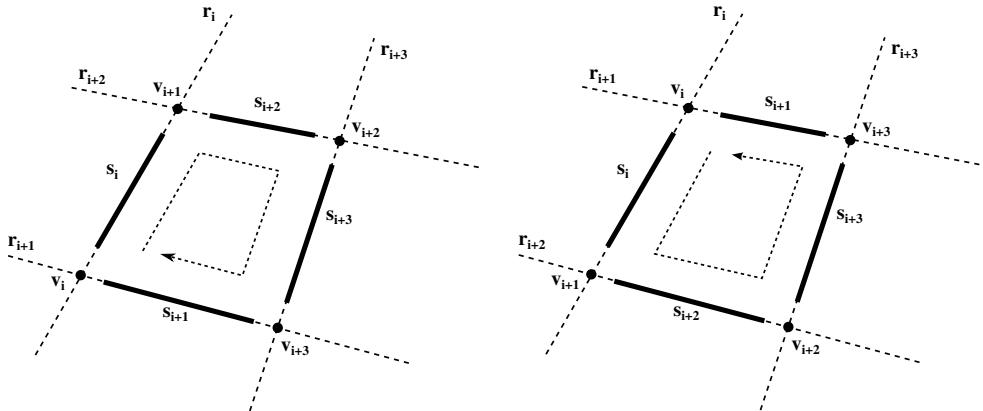


Figura 3.9: Posibles configuraciones de vértices posterior a la intersección de conjuntos de segmentos pertenecientes a un cuadrilátero.

Posterior a la intersección se realiza un chequeo sobre el valor de las coordenadas de los vértices. Si alguno de ellos se encuentra fuera de los límites de la imagen, el conjunto de cuatro segmentos es marcado como inválido. Este chequeo resulta en el filtrado de “falsos cuadriláteros” corrigiendo un defecto del filtrado de segmentos, como por ejemplo un grupo de segmentos paralelos cercanos como ya se explicó.

Para cada uno de los conjuntos de vértices se construye con ellos un elemento cuadrilátero que se almacena en una lista de cuadriláteros

$$QlList = (Ql[0] \ Ql[1] \ \dots \ Ql[i] \ \dots \ Ql[\frac{n}{4}])^t$$

A partir de esa lista de cuadriláteros, se buscan grupos de tres cuadriláteros $QlSet$ que “compartan” un mismo centro. Para esto se recorre ordenadamente la lista en i buscando para cada cuadrilátero dos cuadriláteros j y k que cumplan que la distancia entre sus centros y el del i -ésimo cuadrilátero sea menor a cierto umbral c_{th} ,

$$d_{ij} = ||\mathbf{c}_i - \mathbf{c}_j|| < c_{th}, \quad d_{ik} = ||\mathbf{c}_i - \mathbf{c}_k|| < c_{th}. \quad (3.4)$$

Estos cuadriláteros se marcan en la lista como utilizados con ellos se forma el l -ésimo $QlSet$ ordenándolos según su perímetro, de menor a mayor como

$$QlSet[l] = (Ql[0] \ Ql[1] \ Ql[2])$$

con $l = (0, 1, 2)$. Esta búsqueda se realiza hasta encontrar un total de tres $QlSet$ completos de forma de obtener un marcador completo, esto es, detectando todos los cuadriláteros que lo componen.

Una vez obtenida la lista de tres $QlSet$,

$$QlSetList = (QlSet[0] \quad QlSet[1] \quad QlSet[2])$$

ésta se ordena de forma que su disposición espacial se corresponda con la del marcador QR. Para esto se calculan las distancias entre los centros de cada $QlSet$ y se toma el índice i como el índice que produce el vector de menor distancia, $\mathbf{u}_i = \mathbf{c}_{i+1} - \mathbf{c}_i$. En este punto que es importante que la condición de distancia entre los centros de los $QlSet$ se cumpla, $d_{10} \gg d_{20}$, para una simple identificación. Bajo una transformación proyectiva del marcador, es posible que esta relación se modifique e incluso que deje de valer pero imponiendo la condición “mucho mayor” se asegura que el algoritmo funciona correctamente para condiciones razonables. Esto es, para proyecciones o poses que se encuentran dentro de las hipótesis uso de la aplicación.

Una vez seleccionado el vector \mathbf{u}_i , se tienen obtiene el juego de vectores $(\mathbf{u}_i, \mathbf{u}_{i+1}, \mathbf{u}_{i+2})$ como se muestra en la figura 3.10.

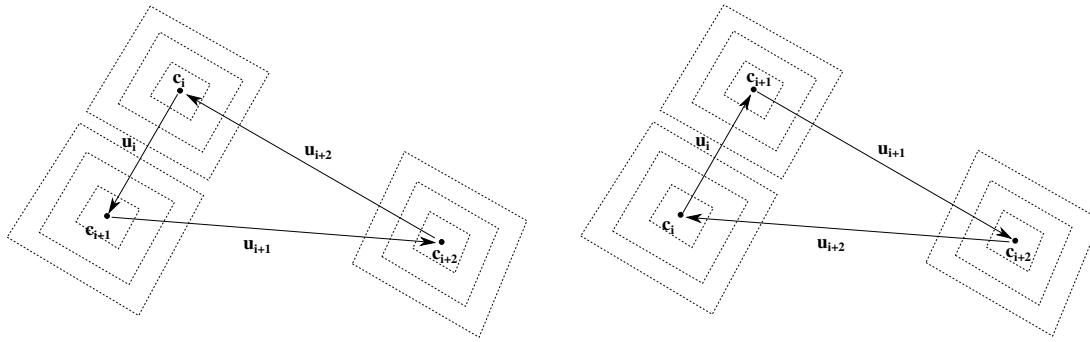


Figura 3.10: Vértices de cada Ql ordenados respecto al signo de sus proyecciones contra el sistema de coordenadas local a cada $QlSet$.

Existen solo dos posibles configuraciones para estos vectores por lo que se utiliza este conocimiento para ordenar los $QlSet$ de la lista realizando el producto vectorial, aumentando la dimensión de los vectores $\hat{\mathbf{u}}_i$ y $\hat{\mathbf{u}}_{i+1}$ con coordenada $z = 0$,

$$\mathbf{b} = \hat{\mathbf{u}}_i \times \hat{\mathbf{u}}_{i+1}.$$

Si el vector \mathbf{b} tiene valor en la coordenada z positivo se ordena como,

$$\begin{aligned} QlSet[0] &\leftarrow QlSet[i] \\ QlSet[1] &\leftarrow QlSet[i + 2] \\ QlSet[2] &\leftarrow QlSet[i + 1] \end{aligned}$$

o de lo contrario se ordena como,

$$\begin{aligned} QlSet[0] &\leftarrow QlSet[i + 1] \\ QlSet[1] &\leftarrow QlSet[i + 2] \\ QlSet[2] &\leftarrow QlSet[i] \end{aligned}$$

Por ultimo se construye un marcador QR que contiene la lista de tres $QlSet$ ordenados según lo indicado permitiendo la definición de un centro de coordenadas como el centro \mathbf{c}_0 del $QlSet[0]$ y ejes de coordenadas definidos en la ecuación 3.1. Los ejes de este sistema de coordenadas permiten, para cada Ql de cada $QlSet$, proyectar los vértices sobre el sistema de coordenadas local al $QlSet$ y

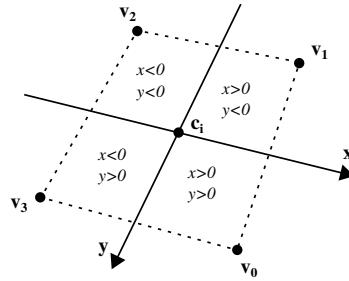


Figura 3.11: Posibles configuraciones de centros resultan en la orientación de los vectores \mathbf{u}_{i+k} .

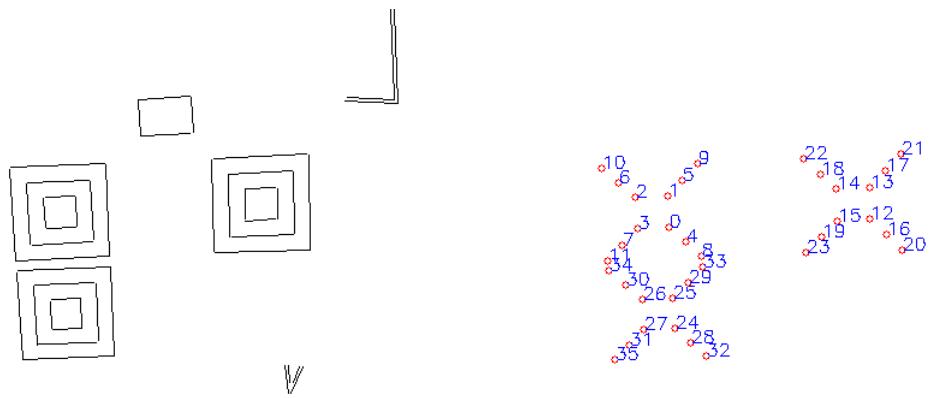
según su signo ordenarlos como se muestra en la figura 3.11. De esta forma, recorriendo ordenadamente los elementos del marcador, se ordenan los vértices de cada Ql del marcador.

Por último, a partir del marcador ordenado, se extrae una lista de vértices que se corresponde con la lista de vértices del marcador en coordenadas del mundo. Este recorrido se realiza en el siguiente orden,

```

for  $i = (0, 1, 2)$  do
    for  $j = (0, 1, 2)$  do
        for  $k = (0, 1, 2, 3)$  do
            So obtiene el punto vértice:  $\mathbf{p} = QlSet[i] \rightarrow Ql[j] \rightarrow v[k]$ ;
            Se agrega a la lista de correspondencias  $\mathbf{m}_l \leftarrow \mathbf{p}$ ;
            Se incrementa  $l$ ;
    
```

Se determinan las correspondencias $\mathbf{M}_i \leftrightarrow \mathbf{m}_i$ necesarias para la estimación de pose las cuales se muestran en la figura 3.3.3. Se puede ver que el algoritmo de determinación de correspondencias funciona correctamente por lo que los “falsos” cuadriláteros que sobreviven al filtrado de segmentos no son un problema.



(a) Entrada: segmentos de línea filtrados y agrupados

(b) Salida: puntos vértices ordenados.

Figura 3.12: Resultados del algoritmo de determinación de correspondencias.

3.3.4. Detección robusta

El algoritmo descripto al momento requiere que dentro de la lista de segmentos filtrados se encuentren todos los segmentos que componen el marcador pero este requerimiento representa un problema importante en cuanto a el desempeño del algoritmo. En caso de que esto no se cumpla no es posible proporcionar las correspondencias necesarias para la estimación de pose y no se tendrá una pose válida para ese cuadro o *frame* para la aplicación. En aplicaciones en tiempo real en donde el procesamiento de la imagen es la mayor limitante, la fluidez visual dada por el *frame rate* se ve notablemente perjudicada resultando en que el sistema sea incomodo e incluso inutilizable. Es por esto que en esta sección se desarrolla la extensión del algoritmo de determinación de correspondencias para una cantidad menor de segmentos detectados y filtrados que resulta en una mejor sustancial en la cantidad de *frames* en los cuales es posible determinar correspondencias y obtener así una pose válida.

Se busca una determinación de correspondencias mas robusta pero manteniendo las esencia del algoritmo desarrollado. Por esto se tienen dos aspectos a tomar en cuenta; la detección de *QlSet*'s se realiza basada en la búsqueda de cuadriláteros concéntricos por lo que se debe contar con un mínimo de dos cuadriláteros por *QlSet* para permitir la diferenciación entre un conjunto de segmentos filtrados debido a que pertenecen al marcador y a otro conjunto que no pertenece pero si cumple con las condiciones, por ejemplo podría ser el marco de una obra o cualquier elemento en la escena que forme un cuadrilátero. Esto fija un límite de no menos de 24 segmentos necesarios para el funcionamiento. El otro aspecto a tomar en cuenta se refiere a la forma en que se ordenan los *Ql*'s dentro de cada *QlSet*. Como ya se explicó el orden se basa en la medida del perímetro de los *Ql*'s ordenando de menor a mayor por lo que será necesario contar con, al menos, un *QlSet* completo de forma de tener una referencia a la hora de identificar los *QlSet*'s incompletos hallados. Por lo tanto la extensión del algoritmo permite una correcta identificación de los vértices del marcador con un número mayor o igual a 28 segmentos.

La implementación de esta extensión del algoritmo se realizó manteniendo la estructura básica descrita anteriormente y se detalla aquí solamente los agregados realizados.

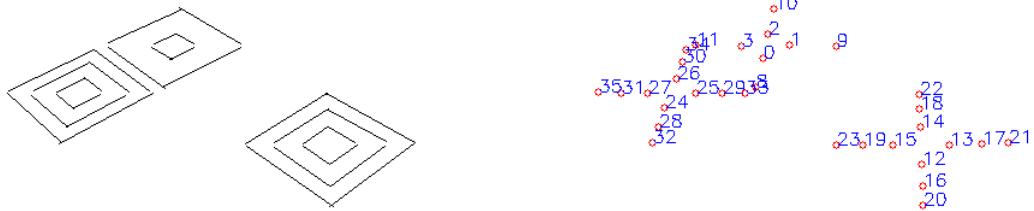
Al realizar la búsqueda de conjuntos de cuadriláteros concéntricos se buscan en primer lugar los *QlSet*'s completos y luego en caso de que estos no lleguen a ser tres, se intenta completar buscando *QlSet*'s incompletos o sea conjuntos de dos cuadriláteros que comparten un mismo centro. Estos se agrupan en una lista de la misma forma en que se describió anteriormente pero dejando el tercer cuadrilátero, *Ql*[2], marcado como inválido.

Una vez completada la lista de tres *QlSet*, con al menos uno de ellos detectado completo, se ordenan en primer lugar los *QlSet* completos y de ellos se extrae una lista de perímetros promedio. Esta lista de perímetros promedio se utiliza para el ordenamiento de los *QlSet* incompletos comparando con los perímetros de los *Ql*[0] y *Ql*[1] de cada *QlSet*. El *Ql*[2] previamente marcado como inválido se posiciona por descarte en la posición que corresponda.

Al momento de proporcionar la lista de vértices ordenados \mathbf{m}_i y correspondientes con los del modelo \mathbf{M}_i , se introducen valores inválidos para los *Ql*'s marcados como inválidos. Por último se realiza un recorte de las dos listas de puntos en base a estos valores inválidos, se recorre la lista de puntos en la imagen \mathbf{m}_i y se extraen de la lista de puntos en la imagen y de los puntos del modelo los puntos inválidos obteniendo un juego de al menos 28 correspondencias $\mathbf{m}'_i \leftrightarrow \mathbf{M}'_i$ para el algoritmo de estimación de pose.

En la figura 3.3.4(a) se muestran imagen en la que falla el filtrado de segmentos para uno de los cuadriláteros mientras que en la figura 3.3.4(b) se puede ver como el algoritmo de determinación de

correspondencias provee 32 correspondencias ordenadas correctamente, diferenciando en el *QlSet* incompleto los vértices.



(a) Entrada: segmentos de línea filtrados y agrupados

(b) Salida: puntos vértices ordenados.

Figura 3.13: Resultados del algoritmo de determinación de correspondencias robusto para una falla en el filtrado de segmentos.

3.3.5. Resultados

Mas imágenes con resultados?

Todo sobre la misma imagen de entrada?

O toda la secuencia para distintas imágenes?

CAPÍTULO 4

LSD: “Line Segment Detection”

4.1. Introducción

LSD es un algoritmo de detección de segmentos publicado por Rafael Grompone von Gioi, Jérémie Jakubowicz, Jean-Michel Morel y Gregory Randall en abril de 2010. Es temporalmente lineal, tiene presición inferior a un píxel y no requiere de un tuneo previo de parámetros, como casi todos los demás algoritmos de idéntica función; puede ser considerado el estado del arte en cuanto a detección de segmentos en imágenes digitales. Como cualquier otro algoritmo de detección de segmentos, LSD basa su estudio en la búsqueda de contornos angostos dentro de la imagen. Estos son regiones en donde el nivel de brillo de la imagen cambia notoriamente entre píxeles vecinos, por lo que el gradiente de la misma resulta de vital importancia. Se genera previo al análisis de la imagen, un campo de orientaciones asociadas a cada uno de los píxeles denominado por los autores *level-line orientation field*. Dicho campo se obtiene de calcular las orientaciones ortogonales a los ángulos asociados al gradiente de la imagen. Luego, LSD puede verse como una composición de tres pasos:

- (1) División de la imagen en las llamadas *line-support regions*, que son grupos conexos de píxeles con idéntica orientación, hasta cierta tolerancia.
- (2) Búsqueda del segmento que mejor aproxime cada *line-support region*: aproximación de las regiones por rectángulos.
- (3) Validación o no de cada segmento detectado en el punto anterior.

Los puntos (1) y (2) están basados en el algoritmo de detección de segmentos de Burns, Hanson y Riseman y el punto (3) es una adaptación del método *a contrario* de Desolneux, Moisan y Morel.

4.2. *Line-support regions*

El primer paso de LSD es el dividir la imagen en regiones conexas de píxeles con igual orientación, a menos de cierta tolerancia τ , llamadas *line-support regions*. El método para realizar tal división es del tipo “región creciente”; cada región comienza por un píxel y cierto ángulo asociado, que en este caso coincide con el de este primer píxel. Luego, se testean sus ocho vecinos y los que cuenten con un ángulo similar al de la región son incluídos en la misma. En cada iteración el

ángulo asociado a la región es calculado como el promedio de las orientaciones de cada píxel dentro de la *line-support region*; la iteración termina cuando ya no se pueden agregar más píxeles a esta.

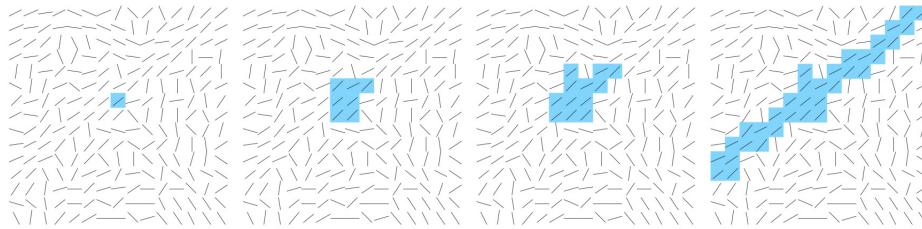


Figura 4.1: Proceso de crecimiento de una región. El ángulo asociado cada píxel de la imagen está representado por los pequeños segmentos y los píxeles coloreados representan la formación de la región. Fuente [7].

Los píxeles agregados a una región son marcados de manera que no vuelvan a ser testeados. Para mejorar el desempeño del algoritmo, las regiones comienzan a evaluarse por los píxeles con gradientes de mayor amplitud ya que estos representan mejor los bordes. Existen algunos casos puntuales en los que el proceso de búsqueda de *line-support regions* puede arrojar errores. Por ejemplo, cuando se tienen dos segmentos que se juntan y que son colineales a no ser por la tolerancia τ descripta anteriormente, se detectarán ambos segmentos como uno solo; ver figura 4.2. Este potencial problema es heredado del algoritmo de Burns, Hanson y Riseman.



Figura 4.2: Potencial problema heredado del algoritmo de Burns, Hanson y Riseman. Izq.: Imagen original. Ctro.: Segmento detectado. Der.: Segmentos que deberían haberse detectado. Fuente [7].

Sin embargo, LSD plantea un método para ahorrarte este tipo de problemas. Durante el proceso de crecimiento de las regiones, también se realiza la aproximación rectangular a dicha región (paso (2) de los tres definidos anteriormente); y si menos cierto porcentaje umbral de los píxeles dentro del rectángulo corresponden a la *line-support region*, lo que se tiene no es un segmento. Se detiene entonces el crecimiento de la región.

4.3. Aproximación de las regiones por rectángulos

Cada *line-support region* debe ser asociada a un segmento. Cada segmento será determinado por su centro, su dirección, su anchura y su longitud. A diferencia de lo que pudiése dictar la intuición, la dirección asociada al segmento no se corresponde con la asociada a la región (el promedio de las direcciones de cada uno de los píxeles). Sin embargo, se elige el centro del segmento como el centro de masa de la región y su dirección como el eje de inercia principal de la misma; la magnitud del gradiente asociado a cada píxel hace las veces de masa. La idea detrás de este método es que los píxeles con un gradiente mayor en módulo, se corresponden mejor con la percepción de un borde. La anchura y la longitud del segmento son elegidos de manera de cubrir el 99 % de la masa de la región.

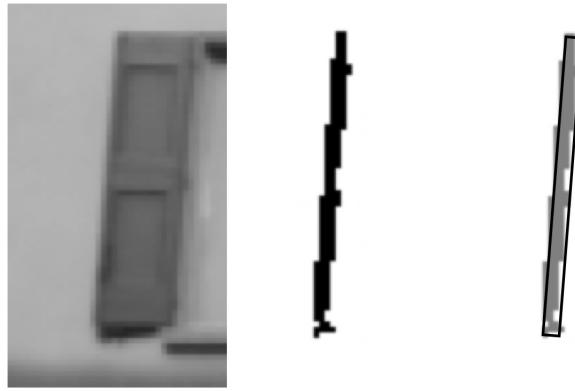


Figura 4.3: Búsqueda del segmento que mejor aproxime cada *line-support region*: aproximación de una región por un rectángulo. Izq.: Imagen original. Ctro.: Una de las regiones computadas. Der.: Aproximación rectangular que cubre el 99 % de la masa de la región. Fuente [7].

4.4. Validación de segmentos

La validación de los segmentos previamente detectados se plantea como un método de test de hipótesis. Se utiliza un modelo *a contrario*: dada una imagen de ruido blanco y Gaussiano, se sabe que cualquier tipo de estructura detectada sobre la misma será casual. En rigor, se sabe que para cualquier imagen de este tipo, su *level-line orientation field* toma, para cada píxel, valores independientes y uniformemente distribuidos entre $[0, 2\pi]$. Dado entonces un segmento en la imagen analizada, se estudia la probabilidad de que dicha detección se dé en la imagen de ruido, y si ésta es lo suficientemente baja, el segmento se considerará válido, de lo contrario se considerará que se está bajo la hipótesis H_0 : un conjunto aleatorio de píxeles que casualmente se alinearon de manera de detectar un segmento.

Para estudiar la probabilidad de ocurrencia de una cierta detección en la imagen de ruido, se deben tomar en cuenta todos los rectángulos potenciales dentro de la misma. Dada una imagen $N \times N$, habrán N^4 orientaciones posibles para los segmentos, N^2 puntos de inicio y N^2 puntos de fin. Si se consideran N posibles valores para la anchura de los rectángulos, se obtienen N^5 posibles segmentos. Por su parte, dado cierto rectángulo r , detectado en la imagen x , se denota $k(r, x)$ a la cantidad de píxeles alineados dentro del mismo. Se define además un valor llamado *Number of False Alarms* (NFA) que está fuertemente relacionado con la probabilidad de detectar al rectángulo en cuestión en la imagen de ruido X :

$$NFA(r, x) = N^5 \cdot P_{H_0}[k(r, X) \geq k(r, x)]$$

véase que el valor se logra al multiplicar la probabilidad de que un segmento de la imagen de ruido, de tamaño igual a r , tenga un número mayor o igual de píxeles alineados que éste, por la cantidad potencial de segmentos N^5 . Cuanto menor sea el número NFA, más significativo será el segmento detectado r ; pues tendrá una probabilidad de aparición menor en una imagen sin estructuras. De esta manera, se descartará H_0 , o lo que es lo mismo, se aceptará el segmento detectado como válido, si y sólo si:

$$NFA(r) \leq \varepsilon$$

donde empíricamente $\varepsilon = 1$ para todos los casos.

Si se toma en cuenta que cada píxel de la imagen ruidosa toma un valor independiente de los demás, se concluye que también lo harán su gradiente y su *level-line orientation field*. De esta manera, dada una orientación aleatoria cualquiera, la probabilidad de que uno de los píxeles de la imagen cuente

con dicha orientación, a menos de la ya mencionada tolerancia τ , será:

$$p = \frac{\tau}{\pi}$$

además, se puede modelar la probabilidad de que cierto rectángulo en la imagen ruidosa, con cualquier orientación, formado por $n(r)$ píxeles, cuente con al menos $k(r)$ de ellos alineados, como una distribución binomial:

$$P_{H_0}[k(r, X) \geq k(r, x)] = B(n(r), k(r), p).$$

Finalmente, el valor *Number of False Alarms* será calculado para cada segmento detectado en la imagen analizada de la siguiente manera:

$$NFA(r, x) = N^5 \cdot B(n(r), k(r), p);$$

si dicho valor es menor o igual a $\varepsilon = 1$, el segmento se tomará como válido; de lo contrario se descartará.

4.5. Refinamiento de los candidatos

Por lo que se vió hasta el momento, la mejor aproximación rectangular a una *line-support region* es la que obtenga un valor NFA menor. Para los segmentos que no son validados, se prueban algunas variaciones a la aproximación original con el objetivo de disminuir su valor NFA y así entonces validarlos. Es claro que este paso no es significativo para segmentos largos y bien definidos, ya que estos serán validados en la primera inspección; sin embargo, ayuda a detectar segmentos más pequeños y algo ruidosos.

Lo que se hace es probar distintos valores para la anchura del segmento y para sus posiciones laterales, ya que estas son los parámetros peor estimados en la aproximación rectangular, pero tienen un efecto muy grande a la hora de validar los segmentos. Es que un error de un píxel en el ancho de un segmento, puede agregar una gran cantidad de píxeles no alineados a este (tantos como el largo del segmento), y esto se ve reflejado en un valor mayor de NFA y puede llevar a una no detección.

Otro método para el refinamiento de los candidatos es la disminución de la tolerancia τ . Si los puntos dentro del rectángulo efectivamente corresponden a un segmento, aunque la tolerancia disminuya, se computará prácticamente misma cantidad de segmentos alineados; y con una probabilidad menor de ocurrencia ($\frac{\tau}{\pi}$), el valor NFA obtenido será menor. Los nuevos valores testeados de tolerancia son: $\frac{\tau}{2}, \frac{\tau}{4}, \frac{\tau}{8}, \frac{\tau}{16}$ y $\frac{\tau}{32}$. El nuevo valor NFA asociado al segmento será el menor de todos los calculados.

4.6. Optimización del algoritmo para tiempo real

Que un algoritmo de procesamiento de imágenes digitales sea temporalmente lineal significa que su tiempo de ejecución crece linealmente con el tamaño de la imagen en cuestión. Se sabe que estos algoritmos son ideales para el procesamiento en tiempo real. Si bien, como se aclaró algunos párrafos atrás, LSD es temporalmente lineal, este no fue pensado para ser ejecutado en tiempo real. Así entonces, para poder aumentar la tasa de cuadros por segundo total de la aplicación, hubo que

realizar algunos cambios mínimos en el código, siempre buscando que estos no sean sustantivos, con el objetivo de alterar lo menos posible el desempeño del algoritmo. Se trabajó sobre ciertos bloques en particular.

4.6.1. Filtro Gaussiano

Antes de procesar la imagen con el algoritmo tal y como se vió en secciones anteriores, la misma es filtrada con un filtro Gaussiano. Se busca en primer lugar, disminuir el tamaño de la imagen de entrada con el objetivo de disminuir el volumen de información procesada. Además, al difuminar la imagen, se conservan únicamente los bordes más pronunciados. Para este proyecto en particular, se escogió la escala del submuestreo fija en 0,5, un poco más adelante en la corriente sección se explicará por qué.

El filtrado de la imagen se hace en dos pasos, primero a lo ancho y luego a lo largo. Se utiliza el núcleo Gaussiano normalizado de la figura 4.4.

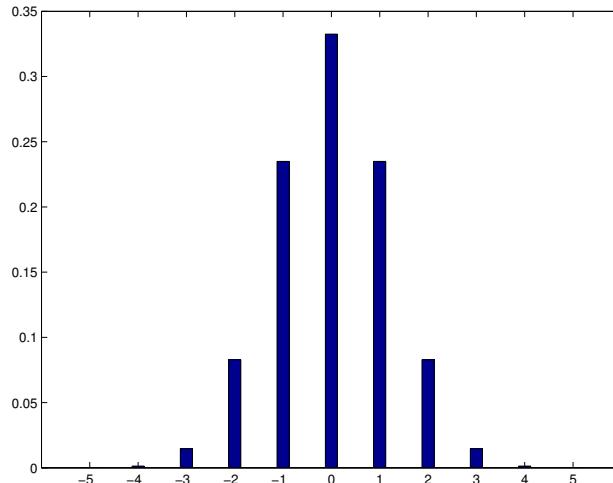


Figura 4.4: Núcleo Gaussiano utilizado por LSD. $\sigma = 1,2$.

De esta manera, se crea una imagen auxiliar vacía y escalada en x pero no en y , y se recorre asignándole a cada píxel en x su valor correspondiente, obtenido del promedio del píxel $\frac{x}{\text{escala}}$ en la imagen original y sus vecinos, todos ponderados por el núcleo Gaussiano centrado en $\frac{x}{\text{escala}}$. Luego se crea otra imagen, pero esta vez escalada tanto en x como en y , y se recorre asignándole a cada píxel en y su valor correspondiente, obtenido del promedio del píxel $\frac{y}{\text{escala}}$ en la imagen auxiliar y sus vecinos, todos ponderados por el núcleo Gaussiano centrado en $\frac{y}{\text{escala}}$. En la figura 4.5 se muestra la relación entre las imágenes.

Véase que cuando en el submuestreo $\frac{1}{\text{escala}}$ no es un entero, el centro del núcleo Gaussiano no siempre debe caer justo sobre un píxel en particular en la imagen original, sino que debe hacerlo entre dos de ellos. Lo que se hace entonces es mover $\pm 0,5$ píxeles al centro del núcleo en cada asignación de los píxeles en las imágenes escaladas; de manera de que la ponderación en el promediado de los píxeles de la imagen original (y luego la auxiliar) sea la debida. Aunque esta operación

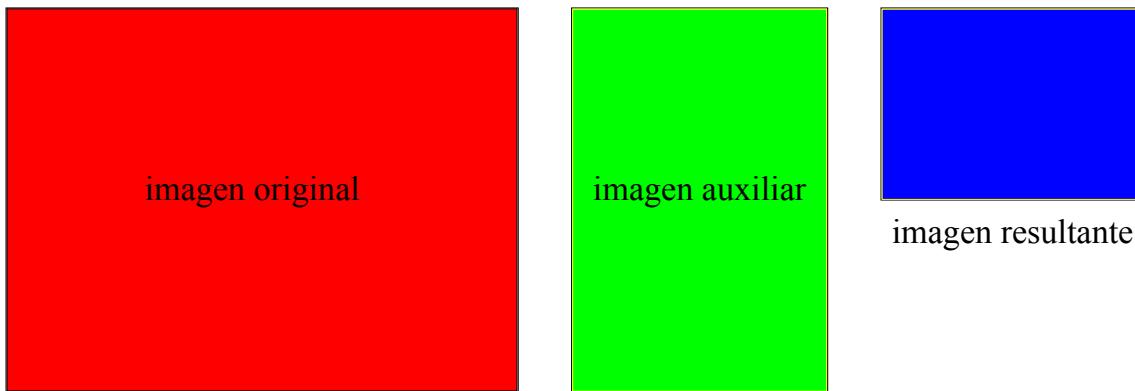


Figura 4.5: Relación entre las imágenes en consideradas en el filtro Gaussiano. Escala: 0,5.

le agrega precisión al algoritmo, también le agrega un gran costo computacional, ya que lo que se hace es crear un nuevo núcleo Gaussiano en cada caso. En particular, para una imagen escalada de 240×180 píxeles (dimensiones efectivamente utilizadas en este proyecto), debido al filtrado en dos pasos, el núcleo Gaussiano se crea y se destruye $86400 + 43200 = 129600$ veces.

Se decidió redondear la escala de submuestreo en 0,5, ya que los valores utilizados empíricamente hasta el momento rondaban este valor, y se concluyó que para dicha escala, el núcleo Gaussiano debía permanecer constante, siempre centrado en su sexta muestra (ver figura 4.4); por lo que se lo quitó de la iteración y actualmente se crea una sola vez al ingresar la imagen al filtro. Es importante destacar que esta optimización es transparente para el algoritmo si y sólo si $\frac{1}{\text{escala}} = n$, donde n es un entero.

Otro cambio que se le realizó al filtrado Gaussiano fue la supresión de las condiciones de borde. Cuando se filtra cualquier imagen con un filtro con memoria, algo importante a tener en cuenta son las condiciones de borde, ya que para el procesamiento de los extremos de la imagen, estos filtros requieren de píxeles que están fuera de sus límites. Algunas de las soluciones a este problema son periodizar la imagen, simetrizarla o hasta asumir el valor 0 para los píxeles que estén fuera de esta. La opción escogida por LSD es la simetrización. Demás está decir que este proceso requiere de cierto costo computacional extra, por lo que se lo decidió suprimir. Actualmente, la imagen escalada no es computada en sus píxeles terminales; estos son 3 al inicio de cada línea o columna y 2 al final de cada una de ellas, irrelevantes en el tamaño total de la imagen. Ver figura 4.6.

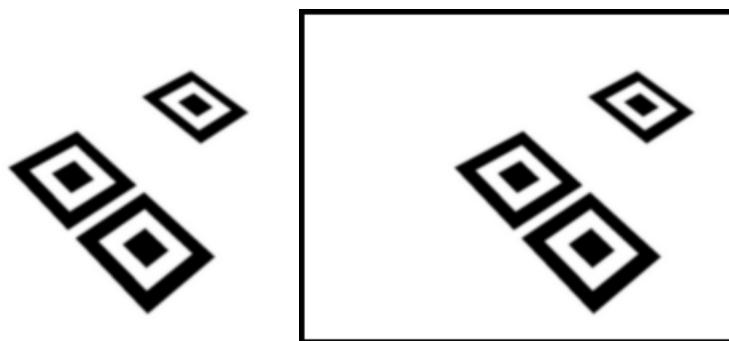


Figura 4.6: Imagen artificial del marcador trasladado y rotado, filtrada con el filtro Gaussiano. Izq.: Filtro Original. Der.: Filtro sin las condiciones de borde.

4.6.2. *Level-line angles*

La función *ll_angles* es quien calcula el gradiente de la imagen previamente filtrada para luego obtener el llamado *level-line orientation field*, en donde más tarde se hallarán los candidatos a segmentos. Lo que se hizo en esta función fué limitar el cálculo del gradiente a los píxeles donde la imagen escalada haya sido efectivamente computada. De esta manera se ahorra procesamiento innecesario, además de no detectarse las líneas negras en el contorno de la imagen (figura 4.6), que de no ser así se detectarían.

4.6.3. Refinamiento y mejora de los candidatos

Se vió en la explicación del algoritmo el problema de que si hubiesen dos o más segmentos que formen entre ellos ángulos menores o iguales al valor umbral τ , estos serían detectados como uno único, heredado del algoritmo de Burns, Hanson y Riseman; y se explicó cómo, mediante un refinamiento de los segmentos, LSD soluciona este problema. Se vió además que luego de la validación o no de los segmentos previamente detectados, se realiza una mejora de los mismos para intentar que los no validados a causa de una mala estimación rectangular, sí puedan serlo.

Como en este proyecto en particular se trabaja con marcadores formados por cuadrados concéntricos, de bordes bien marcados y que forman ángulos rectos entre sí, el refinamiento y la mejora de los candidatos no es algo que afecte la detección de los mismos; y por consiguiente se suprimieron ambos bloques. Como era de esperarse, dichas supresiones no significaron un cambio considerable en el algoritmo desde el punto de vista del desempeño ni del tiempo de ejecución cuando tan sólo se enfoca al marcador. Sin embargo, si las imágenes capturadas cuentan con muchos segmentos (imágenes naturales genéricas), se ve que la detección de los mismos es menos precisa que la del algoritmo original, pero que los tiempos de procesamiento son notablemente inferiores.

4.6.4. Algorirmo en precisión simple

Originalmente, LSD fue implementado en precisión doble o *double* (64 bits por valor). Sin embargo, el *ipad 2* (dispositivo para el cual se optimizó el algoritmo), cuenta con un procesador *ARM Cortex-A9*, cuyo bus de datos es de 32 bits. Se decidió entonces probar cambiar al algoritmo a precisión simple o *float* (32 bits por valor) y los resultados fueron realmente buenos. No sólo el algoritmo bajó su tiempo de ejecución, sino que además no existen cambios notorios en el desempeño del mismo.

4.6.5. Resultados

4.6.5.1. Filtro Gaussiano



Figura 4.7: Imagen sintética del marcador trasladado y rotado.

Se analizaron los tiempos promedio para la ejecución del filtro Gaussiano original y del optimizado, ambos con precisión doble y simple. Las imágenes de prueba fueron las de la figura 4.7; sépase que por cómo es el algoritmo, el contenido de la imagen es independiente del tiempo de procesamiento en cualquiera de los casos. Los valores relevantes del experimento se muestran en las tablas 4.1 y 4.2:

- **Precisión doble (*double*)**

	Filtro original	Filtro optimizado
Tamaño de imagen de entrada	480×360	480×360
Escala	0,5	0,5
Tamaño de imagen de salida	240×180	240×180
Segmentos detectados	36	36
Tiempo medio de procesamiento	36ms	29ms

Tabla 4.1: Comparación entre los tiempos de ejecución del filtro Gaussiano optimizado y el original. Ambos con precisión doble.

- **Precisión simple (*float*)**

	Filtro original	Filtro optimizado
Tamaño de imagen de entrada	480×360	480×360
Escala	0,5	0,5
Tamaño de imagen de salida	240×180	240×180
Segmentos detectados	36	36
Tiempo medio de procesamiento	28ms	20ms

Tabla 4.2: Comparación entre los tiempos de ejecución del filtro Gaussiano optimizado y el original. Ambos con precisión simple.

4.6.5.2. Line Segment Detection

Se analizaron los tiempos conjuntos para la ejecución de LSD más el filtro Gaussiano, los originales y los optimizados, ambos con precisión doble y simple. Se probaron ambos bloques juntos



Figura 4.8: Imagen *zebras.png*.

ya que el algoritmo original está implementado con éstos integrados. Las imágenes de prueba fueron la del marcador sintético (figura 4.7) y *zebras.png* mostrada en la figura 4.8. Los valores relevantes de los experimentos se muestran en las tablas 4.3, 4.4, 4.5 y 4.6.

■ Precisión doble (*double*)

	Algoritmo original	Algoritmo optimizado
Imagen urilizada	marcador sintético	marcador sintético
Tamaño de imagen de entrada	480×360	480×360
Escala	0,5	0,5
Tamaño de imagen de salida	240×180	240×180
Segmentos detectados	36	36
Tiempo medio de procesamiento	55,4ms	48ms

Tabla 4.3: Comparación entre los tiempos de ejecución del filtro Gaussiano más LSD optimizados y los originales, para la imagen 4.7. En todos los casos con comprecisión doble.

	Algoritmo original	Algoritmo optimizado
Imagen urilizada	<i>zebras.png</i>	<i>zebras.png</i>
Tamaño de imagen de entrada	480×360	480×360
Escala	0,5	0,5
Tamaño de imagen de salida	240×180	240×180
Segmentos detectados	251	179
Tiempo medio de procesamiento	179,7ms	94,4ms

Tabla 4.4: Comparación entre los tiempos de ejecución del filtro Gaussiano más LSD optimizados y los originales, para la imagen 4.8. En todos los casos con comprecisión doble.

■ Precisión simple (*float*)

	Algoritmo original	Algoritmo optimizado
Imagen urilizada	marcador sintético	marcador sintético
Tamaño de imagen de entrada	480×360	480×360
Escala	0,5	0,5
Tamaño de imagen de salida	240×180	240×180
Segmentos detectados	36	36
Tiempo medio de procesamiento	47,8ms	38,8ms

Tabla 4.5: Comparación entre los tiempos de ejecución del filtro Gaussiano más LSD optimizados y los originales, para la imagen 4.7. En todos los casos con compreción simple.

	Algoritmo original	Algoritmo optimizado
Imagen urilizada	<i>zebras.png</i>	<i>zebras.png</i>
Tamaño de imagen de entrada	480×360	480×360
Escala	0,5	0,5
Tamaño de imagen de salida	240×180	240×180
Segmentos detectados	252	182
Tiempo medio de procesamiento	189,8ms	90,8ms

Tabla 4.6: Comparación entre los tiempos de ejecución del filtro Gaussiano más LSD optimizados y los originales, para la imagen 4.8. En todos los casos con compreción simple.

CAPÍTULO 5

Modelo de cámara y estimación de pose monocular

5.1. Introducción

Se le llama “estimación de pose” al proceso mediante el cual se calcula en qué punto del mundo y con qué orientación se encuentra determinado objeto respecto de un eje de coordenadas previamente definido al que se lo llama “ejes del mundo”. Las aplicaciones de realidad aumentada requieren de un modelado preciso del entorno respecto de estos ejes, para poder ubicar correctamente los agregados virtuales dentro del modelo y luego dibujarlos de forma coherente en la imagen vista por el usuario. El objeto cuya estimación de pose resulta de mayor importancia es la cámara, ya que por ésta es por donde se mira la escena y es respecto de ésta que los objetos virtuales deben ubicarse de manera consistente. Una forma de estimar la pose de la cámara es mediante el uso de las imágenes capturadas por ella misma. Asimismo, el concepto “monocular” hace referencia al uso de una sola cámara, ya que es posible trabajar con más de una.

Para poder obtener información relevante a partir de las imágenes tomadas por una cámara, resulta necesario contar con un modelo preciso de su arquitectura ya que no todas las cámaras son iguales. El modelo más comúnmente utilizado es el denominado *pin-hole*. Para modelar completamente la arquitectura de la cámara se deben estimar ciertos “parámetros intrínsecos” a ésta, y eso se logra luego de realizados ciertos experimentos. A la estimación de estos parámetros se le denomina “calibración de la cámara”.

En este capítulo se verá en detalle el modelo de cámara *pin-hole*, tomando en cuenta la distorsión introducida por las lentes. Más adelante, se mencionarán distintos métodos para la calibración de una cámara y se verá en detalle un en particular, el método de Zhang. **JUANI ACÁ TENES QUE PONER VOS QUÉ MÁS VA A TENER ESTE CAPÍTULO.**

5.2. Modelo de cámara *pin-hole* [1]

5.2.1. Fundamentos y definiciones

Este modelo consiste en un centro óptico C, en donde convergen todos los rayos de la proyección y un plano imagen en el cual la imagen es proyectada. Se define “distancia focal” (f) como la distancia entre el centro óptico C y el cruce del eje óptico por el plano imagen (punto P). Ver

Imagen 5.1.

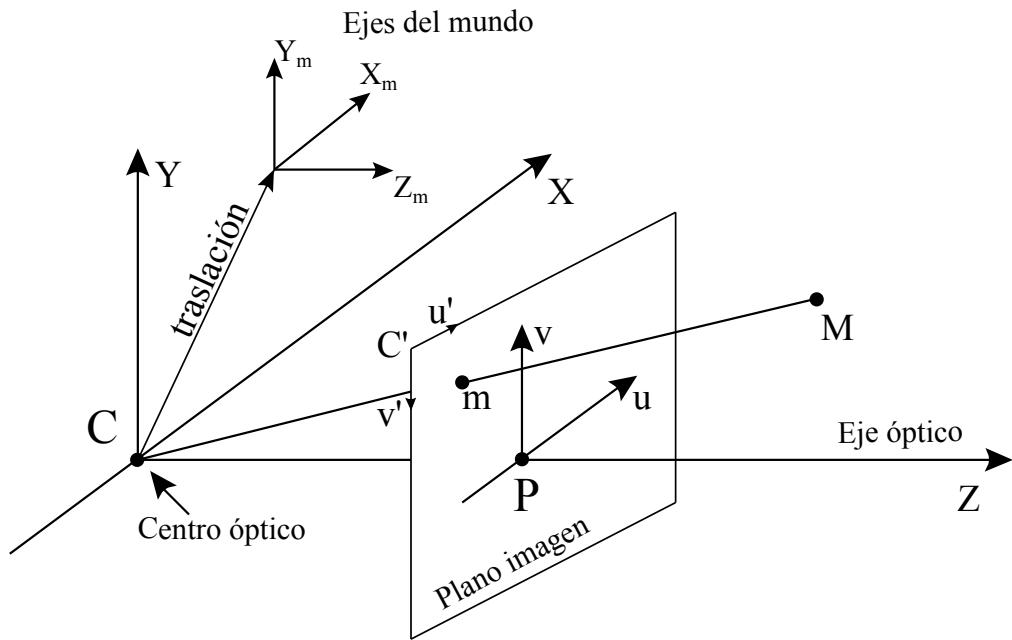


Figura 5.1: Modelo de cámara pin-hole.

Para modelar el proceso de proyección (proceso en el que se asocia al punto **M** del mundo, un punto **m** en la imagen), es necesario referirse a varias transformaciones y varios ejes de coordenadas.

- *Coordenadas del mundo:* son las coordenadas que describen la posición 3D del punto **M**. Se definen respecto de los *ejes del mundo* (X_m, Y_m, Z_m). La elección de los ejes del mundo es arbitraria.
- *Coordenadas de la cámara:* son las coordenadas que describen la posición del punto **M** respecto de los ejes de la cámara (X, Y, Z).
- *Coordenadas de la imagen:* son las coordenadas que describen la posición del punto 2D, **m**, respecto del centro del plano imagen, P. Los ejes de este sistema de coordenadas son (u, v).
- *Coordenadas normalizadas de la imagen:* son las coordenadas que describen la posición del punto 2D, **m**, respecto del eje de coordenadas (u', v') situado en la esquina superior izquierda del plano imagen.

La transformación que lleva al punto **M**, expresado respecto de los ejes del mundo, al punto **m**, expresado respecto del sistema de coordenadas normalizadas de la imagen, se puede ver como la composición de dos transformaciones menores. La primera, es la que realiza la proyección que transforma a un punto definido respecto del sistema de coordenadas de la cámara (X, Y, Z) en otro punto sobre el plano imagen expresado respecto del sistema de coordenadas normalizadas de la imagen (u', v'). Véase que una vez calculada esta transformación, es una constante característica de cada cámara. Al conjunto de valores que definen esta transformación, se le llama “parámetros intrínsecos” de la cámara. La segunda, es la transformación que lleva de expresar a un punto respecto de los ejes del mundo (X_m, Y_m, Z_m), a los ejes de la cámara (X, Y, Z). Esta última transformación varía conforme se mueve la cámara (respecto de los ejes del mundo) y el conjunto de valores que la definen es denominado “parámetros extrínsecos” de la cámara. Del cálculo de estos parámetros es

que se obtiene la estimación de la pose de la cámara.

De lo anterior se concluye rápidamente que si se le llama H a la matriz proyección total, tal que:

$$\mathbf{m} = H \cdot \mathbf{M},$$

entonces:

$$H = I \cdot E$$

donde I corresponde a la matriz proyección asociada a los parámetros intrínsecos y E corresponde a la matriz asociada a los parámetros extrínsecos. Ambos juegos de parámetros acarrean información muy valiosa:

■ **Parámetros extrínsecos:** pose de la cámara.

- Traslación: ubicación del centro óptico de la cámara respecto de los ejes del mundo.
- Rotación: rotación del sistema de coordenadas de la cámara (X, Y, Z), respecto de los ejes del mundo.

■ **Parámetros intrínsecos:** parámetros propios de la cámara. Dependen de su geometría interna y de su óptica.

- Punto principal ($\mathbf{P} = [u'_P, v'_P]$): es el punto intersección entre el eje óptico y el plano imagen. Las coordenadas de este punto vienen dadas en píxeles y son expresadas respecto del sistema normalizado de la imagen.
- Factores de conversión píxel-milímetros (d_u, d_v): indican el número de píxeles por milímetro que utiliza la cámara en las direcciones u y v respectivamente.
- Distancia focal (f): distancia entre el centro óptico (\mathbf{C}) y el punto principal (\mathbf{P}). Su unidad es el milímetro.
- Factor de proporción (s): indica la proporción entre las dimensiones horizontal y vertical de un píxel.

5.2.2. Matriz de proyección

En la sección anterior se vió que es posible hallar una “matriz de proyección” H que dependa tanto de los parámetros intrínsecos de la cámara como de sus parámetros extrínsecos:

$$\mathbf{m} = H \cdot \mathbf{M}$$

donde \mathbf{M} y \mathbf{m} son los puntos ya definidos y vienen expresados en “coordenadas homogéneas”. Por más información acerca de este tipo de coordenadas ver [8].

Para determinar la forma de la matriz de proyección se estudia cómo se relacionan las coordenadas de \mathbf{M} con las coordenadas de \mathbf{m} ; para hallar esta relación se debe analizar cada transformación, entre los sistemas de coordenadas mencionados con anterioridad, por separado.

■ **Proyección 3D - 2D:** de las coordenadas homogéneas del punto \mathbf{M} expresadas en el sistema de coordenadas de la cámara (X_0, Y_0, Z_0, T_0), a las coordenadas homogéneas del punto \mathbf{m} expresadas en el sistema de coordenadas de la imagen (u_0, v_0, s_0):

Se desprende de la imagen 5.1 y algo de geometría proyectiva la siguiente relación entre las coordenadas en cuestión y la distancia focal (f):

$$\frac{f}{Z_0} = \frac{u_0}{X_0} = \frac{v_0}{Y_0}$$

A partir de la relación anterior:

$$\begin{pmatrix} u_0 \\ v_0 \end{pmatrix} = \frac{f}{Z_0} \begin{pmatrix} X_0 \\ Y_0 \end{pmatrix}$$

Expresado en forma matricial, en coordenadas homogéneas:

$$\begin{pmatrix} u_0 \\ v_0 \\ s_0 \end{pmatrix} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X_0 \\ Y_0 \\ Z_0 \\ 1 \end{pmatrix}$$

- **Transformación imagen - imagen:** de las coordenadas homogéneas del punto **m** expresadas respecto del sistema de coordenadas de la imagen (u_0, v_0, s_0) , a las coordenadas homogéneas de él mismo pero expresadas respecto del sistema de coordenadas normalizadas de la imagen (u'_0, v'_0, s'_0) :

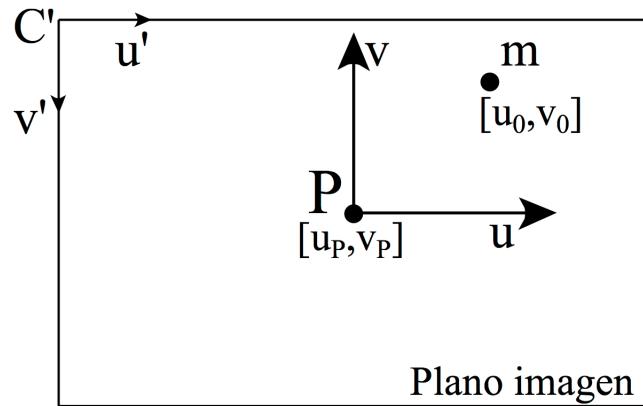


Figura 5.2: Relación entre el sistema de coordenadas de la imagen y el sistema de coordenadas normalizadas de la imagen.

Se les suma, a las coordenadas de **m** respecto del sistema de la imagen, la posición del punto **P** respecto del sistema normalizado de la imagen (u'_P, v'_P) . Las coordenadas de **m** dejan de ser expresadas en milímetros para ser expresadas en píxeles. Aparecen los factores de conversión d_u y d_v :

$$\begin{aligned} u'_0 &= d_u \cdot u_0 + u'_P \\ v'_0 &= d_v \cdot v_0 + v'_P \end{aligned}$$

Se obtiene entonces la siguiente relación matricial, en coordenadas homogéneas:

$$\begin{pmatrix} u'_0 \\ v'_0 \\ s'_0 \end{pmatrix} = \begin{pmatrix} d_u & 0 & u'_P \\ 0 & d_v & v'_P \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} u_0 \\ v_0 \\ 1 \end{pmatrix}$$

- **Matriz de parámetros intrínsecos (I):** de las coordenadas homogéneas del punto \mathbf{M} expresadas en el sistema de coordenadas de la cámara (X_0, Y_0, Z_0, T_0) , a las coordenadas homogéneas del punto \mathbf{m} expresadas respecto del sistema de coordenadas normalizadas de la imagen (u'_0, v'_0, s'_0) :

Se obtiene combinando las dos últimas transformaciones. Nótese que como ya se aclaró, depende únicamente de parámetros propios de la construcción de la cámara:

$$I = \begin{pmatrix} d_u \cdot f & 0 & u'_P & 0 \\ 0 & d_v \cdot f & v'_P & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Nota: De forma genérica se puede agregar a la matriz de parámetros intrínsecos del modelo *pin-hole* un parámetro s llamado en inglés *skew parameter*, o “parámetro de proporción” en Español. Este parámetro toma valores distintos de cero muy rara vez, pues modela los casos en los que los ejes x e y de los píxeles de la cámara no son perpendiculares entre sí. En casos realistas, $s \neq 0$ cuando por ejemplo se toma una fotografía de una fotografía. La matriz de parámetros intrínsecos, tomando en cuenta este parámetro, tendrá la forma:

$$I = \begin{pmatrix} d_u \cdot f & s & u'_P & 0 \\ 0 & d_v \cdot f & v'_P & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- **Matriz de parámetros extrínsecos (E):** de las coordenadas homogéneas del punto \mathbf{M} expresadas respecto del sistema de coordenadas del mundo $(X_{m0}, Y_{m0}, Z_{m0}, T_{m0})$, a las coordenadas homogéneas de él mismo pero expresadas respecto del sistema de coordenadas de la cámara (X_0, Y_0, Z_0, T_0) :

Se obtiene de estimar la pose de la cámara respecto de los ejes del mundo y es la combinación de, primero una rotación R , y luego una traslación T . Se obtiene entonces la siguiente representación matricial:

$$\begin{pmatrix} X_0 \\ Y_0 \\ Z_0 \\ T_0 \end{pmatrix} = \begin{pmatrix} R & T \end{pmatrix} \begin{pmatrix} X_{m0} \\ Y_{m0} \\ Z_{m0} \\ T_{m0} \end{pmatrix}$$

donde la matriz de parámetros extrínsecos desarrollada toma la forma:

$$E = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- **Matriz de proyección (H):** de las coordenadas homogéneas del punto \mathbf{M} expresadas respecto del sistema de coordenadas del mundo $(X_{m0}, Y_{m0}, Z_{m0}, T_{m0})$, a las coordenadas homogéneas del punto \mathbf{m} expresadas respecto del sistema de coordenadas normalizadas de la imagen (u'_0, v'_0, s'_0) :

Es la proyección total y se obtiene combinando las dos transformaciones anteriores:

$$\begin{pmatrix} u'_0 \\ v'_0 \\ s'_0 \end{pmatrix} = \begin{pmatrix} d_u \cdot f & 0 & u'_P & 0 \\ 0 & d_v \cdot f & v'_P & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} X_{m0} \\ Y_{m0} \\ Z_{m0} \\ T_{m0} \end{pmatrix}$$

5.3. Distorsión introducida por las lentes

Hasta el momento se asumió que el modelo lineal presentado para la proyección de cualquier punto del mundo en el plano imagen de la cámara es lo suficientemente preciso en todos los casos. Sin embargo, en casos reales, y cuando las lentes de las cámaras no son del todo buenas, la distorsión introducida por estas se hace notar. Dado el punto \mathbf{M} de coordenadas (X_0, Y_0, Z_0) respecto de los ejes de la cámara, se le llama distorsión a la diferencia entre su proyección ideal en el plano imagen (u_0, v_0) y su proyección real $(\tilde{u}_0, \tilde{v}_0)$. La más común de todas, es la denominada “distorsión radial”, ya que su magnitud depende del radio medido desde el punto principal del plano imagen, hasta las coordenadas del punto en cuestión.

La forma de solucionar el presente problema es realizar una corrección de la distorsión, modelando a la misma de la siguiente manera:

$$\begin{pmatrix} \tilde{u}_0 \\ \tilde{v}_0 \end{pmatrix} = L(r) \cdot \begin{pmatrix} u_0 \\ v_0 \end{pmatrix},$$

donde r es la distancia radial $\sqrt{u_0^2 + v_0^2}$ y $L(r)$ es un factor de distorsión que depende únicamente del radio r . Si se desarrolla la ecuación anterior, y se expresa en píxeles, respecto del sistema de coordenadas normalizadas de la imagen; se obtiene lo siguiente:

$$\begin{aligned} \tilde{u}'_0 &= u'_P + L(r)(u'_0 - u'_P) \\ \tilde{v}'_0 &= v'_P + L(r)(v'_0 - v'_P) \end{aligned}$$

donde $(\tilde{u}'_0, \tilde{v}'_0)$ son las coordenadas reales de la proyección medidas en píxeles, (u'_0, v'_0) son las coordenadas ideales de la proyección medidas también en píxeles y (u'_P, v'_P) son las coordenadas del punto principal. Véase que en este caso $r = \sqrt{(u'_0 - u'_P)^2 + (v'_0 - v'_P)^2}$.

La función $L(r)$ es definida sólo para valores positivos de r y $L(0) = 1$. Una aproximación a la función arbitraria $L(r)$ puede ser una expansión de Taylor: $L(r) = 1 + k_1 r + k_2 r^2 + k_3 r^3 + \dots$. Finalmente, a la hora de calcular los parámetros intrínsecos de una cámara, también deben ser estimados sus coeficientes de distorsión radial $\{k_1, k_2, k_3, k_4, \dots\}$.

5.4. Métodos para la calibración de cámara

Como se vió algunos párrafos atrás, el proceso mediante el cual se calculan los parámetros intrínsecos reales de una cámara es denominado “calibración de cámara”. Existen varios métodos para calibrar una cámara; sin embargo, los tres algoritmos, basados en modelos planos, más ampliamente utilizados alrededor del mundo [?] son el método de Zhang [?], el método de R.Y. Tsai [?] y un método llamado “Direct Linear Transform” (DLT) [?]. Para calibrar las cámaras utilizadas en este proyecto, se trabajó con una implementación en *Matlab* basada en el método de Zhang ([?]), que afortunadamente dió resultados muy buenos. Por eso, se explicará a continuación, de forma breve, cómo funciona este método. Por dudas respecto de cualquier resultado matemático expuesto sin los cálculos intermedios, siempre se recomienda leer el artículo original.

El método de Zhang es muy sencillo y flexible. Sólo requiere de la cámara a calibrar, una computadora y una imagen patrón (plana), de tipo damero; a la que se le tomarán al menos dos fotografías desde orientaciones distintas. En la figura 5.3 se ve una de las imágenes utilizadas para



Figura 5.3: Imagen de un damero, utilizada para calibrar la cámara del *iPad* durante el proyecto.

calibrar la cámara del *iPad* durante el proyecto. Ni las posiciones de la cámara en cada caso, ni el movimiento entre estas posiciones tienen por qué ser conocidos. Este método devuelve los parámetros intrínsecos de la cámara correspondientes al modelo *pin-hole* visto anteriormente, sus parámetros extrínsecos para cada fotografía utilizada para la calibración y la distorsión radial de sus lentes.

Recuérdese que la relación entre un punto 3D \mathbf{M} expresado respecto de los ejes de coordenadas del mundo y su proyección en el plano imagen \mathbf{m} , expresada respecto de los ejes normalizados de la imagen, viene dada por:

$$\mathbf{m} = I \cdot E \cdot \mathbf{M}$$

donde E representa a la matriz de parámetros extrínsecos e I representa a la matriz de parámetros intrínsecos de la cámara. Además:

$$I = \begin{pmatrix} \alpha & s & u'_P \\ 0 & \beta & v'_P \\ 0 & 0 & 1 \end{pmatrix}$$

con $\alpha = d_u \cdot f$ y $\beta = d_v \cdot f$.

Se asume en este método que el sistema de coordenadas del mundo “reposa” sobre la imagen patrón; o lo que es lo mismo, que esta se encuentra en $Z = 0$. Se obtiene entonces la siguiente simplificación:

$$\begin{pmatrix} u'_0 \\ v'_0 \\ 1 \end{pmatrix} = I \cdot \begin{pmatrix} r_1 & r_2 & r_3 & t \end{pmatrix} \cdot \begin{pmatrix} X_{m0} \\ Y_{m0} \\ Z_{m0} \\ 1 \end{pmatrix} = I \cdot \begin{pmatrix} r_1 & r_2 & t \end{pmatrix} \cdot \begin{pmatrix} X_{m0} \\ Y_{m0} \\ 1 \end{pmatrix}$$

donde $(X_{m0}, Y_{m0}, Z_{m0}, 1)^T$ denota las coordenadas homogéneas del punto \mathbf{M} respecto de los ejes del mundo y $(u'_0, v'_0, 1)^T$ representa las coordenadas homogéneas de su proyección en el plano imagen, \mathbf{m} , respecto de los ejes normalizados de la imagen. Se le llamó r_i a la i -ésima columna de la matriz rotación de los parámetros extrínsecos de la cámara.

Dada una fotografía de la imagen patrón plana (figura 5.3), es posible estimar una homografía que relacione a los puntos de la imagen con sus correspondientes en la fotografía. Si se toma en cuenta que dicha homografía vale $H = (h_1, h_2, h_3) = I \cdot (r_1, r_2, t)$, con h_i la i -ésima columna de la matriz, y que las columnas r_1 y r_2 son ortonormales entre sí, realizando algo de matemática se llega

a que:

$$\begin{aligned} h_1^T \cdot (I^{-1})^T \cdot I^{-1} \cdot h_2 &= 0 \\ h_1^T \cdot (I^{-1})^T \cdot I^{-1} \cdot h_1 &= h_2^T \cdot (I^{-1})^T \cdot I^{-1} \cdot h_2 \end{aligned}$$

Las anteriores son las únicas dos relaciones básicas entre parámetros intrínsecos que se pueden obtener a partir de una única homografía. Esto es porque una homografía tiene 8 grados de libertad y existen 6 parámetros extrínsecos (3 para la traslación y 3 para la rotación).

Si se define la matriz B como sigue:

$$B = (I^{-1})^T \cdot I^{-1} = \begin{pmatrix} B_{11} & B_{21} & B_{31} \\ B_{12} & B_{22} & B_{32} \\ B_{13} & B_{23} & B_{33} \end{pmatrix} = \begin{pmatrix} \frac{1}{\alpha^2} & -\frac{s}{\alpha^2 \cdot \beta} & \frac{s \cdot v'_P - u'_P \cdot \beta}{\alpha^2 \cdot \beta} \\ -\frac{s}{\alpha^2 \cdot \beta} & \frac{s^2}{\alpha^2 \cdot \beta^2} + \frac{1}{\beta^2} & -\frac{s(s \cdot v'_P - u'_P \cdot \beta)}{\alpha^2 \cdot \beta^2} - \frac{v'_P}{\beta^2} \\ \frac{s \cdot v'_P - u'_P \cdot \beta}{\alpha^2 \cdot \beta} & -\frac{s(s \cdot v'_P - u'_P \cdot \beta)}{\alpha^2 \cdot \beta^2} - \frac{v'_P}{\beta^2} & \frac{(s \cdot v'_P - u'_P \cdot \beta)^2}{\alpha^2 \cdot \beta^2} + \frac{v'^2_P}{\beta^2} + 1 \end{pmatrix}$$

se ve fácilmente que esta es simétrica, por lo que quedará absolutamente definida por un vector de 6 dimensiones:

$$b = (B_{11}, B_{12}, B_{22}, B_{13}, B_{23}, B_{33})^T$$

Si además se define el vector variable v_{ij} de la siguiente manera:

$$v_{ij} = (h_{i1} \cdot h_{j1}, h_{i1} \cdot h_{j2} + h_{i2} \cdot h_{j1}, h_{i2} \cdot h_{j2}, h_{i3} \cdot h_{j1} + h_{i1} \cdot h_{j3}, h_{i3} \cdot h_{j2} + h_{i2} \cdot h_{j3}, h_{i3} \cdot h_{j3})^T,$$

se tiene que:

$$h_i^T \cdot B \cdot h_j = V_{ij}^T \cdot b$$

Las dos relaciones básicas entre parámetros intrínsecos obtenidas de una única homografía, vistas anteriormente, pueden ser reescritas como:

$$\begin{pmatrix} v_{12}^T \\ (v_{11} - v_{22})^T \end{pmatrix} \cdot b = V \cdot b = 0$$

Utilizando n fotografías distintas de la imagen patrón, y por lo tanto n homografías distintas se obtiene una matriz V de tamaño $2.n \times 6$. Es sabido que si $n \geq 3$, el sistema matricial anterior tendrá una solución b única, que varía según cierto factor de escala. Sin embargo, si $n = 2$, es posible imponer la condición $s = 0$ y así también calcular al vector b de forma única, sin mayores problemas.

Una vez estimado b es posible reconstruir la matriz de parámetros intrínsecos I , para luego utilizando I y las homografías H obtener los parámetros extrínsecos de la cámara para cada fotografía utilizada para la calibración.

El artículo de Zhang afirma que la solución obtenida hasta el momento no es del todo buena, pues se obtuvo minimizando una distancia algebraica y eso no tiene mucho sentido. Lo que se hace entonces es, utilizando las n fotografías tomadas para la calibración y los k puntos seleccionados en cada una de ellas, minimizar la siguiente ecuación:

$$\sum_{i=1}^n \sum_{j=1}^k \|m_{ij} - \hat{m}(I, E_i, M_j)\|^2$$

donde $\hat{m}(I, E_i, M_j)$ es la proyección del punto M_j en la imagen i utilizando la homografía $H_i = I \cdot E_i$. El resultado de dicha minimización no lineal será el resultado final. Este método requiere de valores

inicales para I y para los $E_i|_{i=1..n}$; que serán los obtenidos en los cálculos anteriores.

Finalmente se realiza una estimación de la distorsión radial utilizando un modelo muy similar al visto en la sección 5.3. Cabe destacar que cuando se estimaron los parámetros intrínsecos de las cámaras utilizadas en este proyecto, la distorsión radial no se tomó en cuenta y aún así los resultados obtenidos fueron realmente muy precisos.

CAPÍTULO 6

POSIT: *POS* with *ITerations*

6.1. Introducción

En este capítulo se explica el algoritmo utilizado para el cálculo de la pose a partir de una imagen capturada por la cámara. Como lo dice el nombre de algoritmo se utiliza una técnica llamada *POS* (*Pose from Orthography and Scaling*), esta técnica consiste en aproximar la pose de la cámara a partir de la proyección *SOP*(*Scaled Orthographic Projection*). Se comienza el capítulo explicando en que consiste la proyección *SOP* y como se estima la pose a partir ella. Con esto como fundamento teórico se explican las diferentes variantes de *POSIT* y finalmente se explica la implementación utilizada en la aplicación.

6.2. POSIT clásico

La primera versión de *POSIT* presentada por *Daniel DeMenthon* y *Larry Davis* en [?] resuelve el problema de calcular la pose de la cámara dados 4 o más puntos detectados en la imagen y sus correspondientes en el mundo real, con la condición de que estos puntos no sean coplanares. Si bien no es la versión final que se utilizó vale la pena ser explicada ya que ayuda a sentar las bases de la implementación utilizada.

6.2.1. Notación y definición formal del problema de estimación de pose

En la figura 6.1 se puede ver un modelo de cámara pinhole, donde el centro es el punto O , G es el plano imagen ubicado a una distancia focal f de O . O_x y O_y son los ejes que apuntan en las direcciones de las filas y las columnas del sensor de la cámara respectivamente. O_z es el eje que está sobre el eje óptico de la cámara y apunta en sentido saliente. Los versores para estos ejes son \mathbf{i} , \mathbf{j} y \mathbf{k} .

Se considera ahora un objeto con puntos característicos $M_0, M_1, \dots, M_i, \dots, M_n$, cuyo eje de coordenadas está centrado en M_0 y está compuesto por los versores (M_0u, M_0v, M_0w) . La geometría del objeto se asume conocida, por lo tanto las coordenadas de los puntos característicos del objeto en el eje de coordenadas del mismo son conocidas. Por ejemplo (U_i, V_i, W_i) son las coordenadas del punto M_i en el marco de referencia del objeto. Los puntos correspondientes a los puntos del objeto M_i en la imagen son conocidos y se identifican como m_i , (x_i, y_i) son las coordenadas de este punto en la imagen. Las coordenadas de los puntos M_i en el eje de coordenadas de la cámara, identificadas como (X_i, Y_i, Z_i) , son desconocidas ya que no se conoce la pose del objeto respecto a la cámara.

Se busca computar la matriz de rotación y el vector de traslación del objeto respecto a la cámara. La matriz de rotación \mathbf{R} del objeto, es la matriz cuyas filas son las coordenadas del los versores i , j y k , expresados en el sistema de coordenadas del objeto (u , v , w), se puede ver como la matriz de cambio de base que pasa coordenadas en la base del objeto a coordenadas en la base de la cámara.

La matriz \mathbf{R} queda:

$$\mathbf{R} = \begin{pmatrix} i_u & i_v & i_w \\ j_u & j_v & j_w \\ k_u & k_v & k_w \end{pmatrix}$$

Para obtener la matriz de rotación solo es necesario obtener los versores \mathbf{i} y \mathbf{j} , el vessor \mathbf{k} se obtiene de realizar el producto vectorial $\mathbf{i} \times \mathbf{j}$. El vector de traslación es el vector que va del centro del objeto M_0 a el centro del sistema de coordenadas de la cámara O . Por lo tanto las coordenadas del vector de traslación son (X_0, Y_0, Z_0) . Si este punto M_0 es uno de los puntos visibles en la imagen, entonces el vector \mathbf{T} esta alineado con el vector Om_0 y es igual a $(Z_0/f)Om_0$. Por lo tanto la pose queda determinada si se conocen \mathbf{i} , \mathbf{j} y Z_0 .

Como detalle a tener en cuenta, se observa que para calcular la traslación es necesario conocer el punto de referencia del objeto M_0 , esta es la principal diferencia con la versión moderna de POSIT en la que para calcular la traslación no es necesario suponer nada acerca del punto de referencia del objeto.

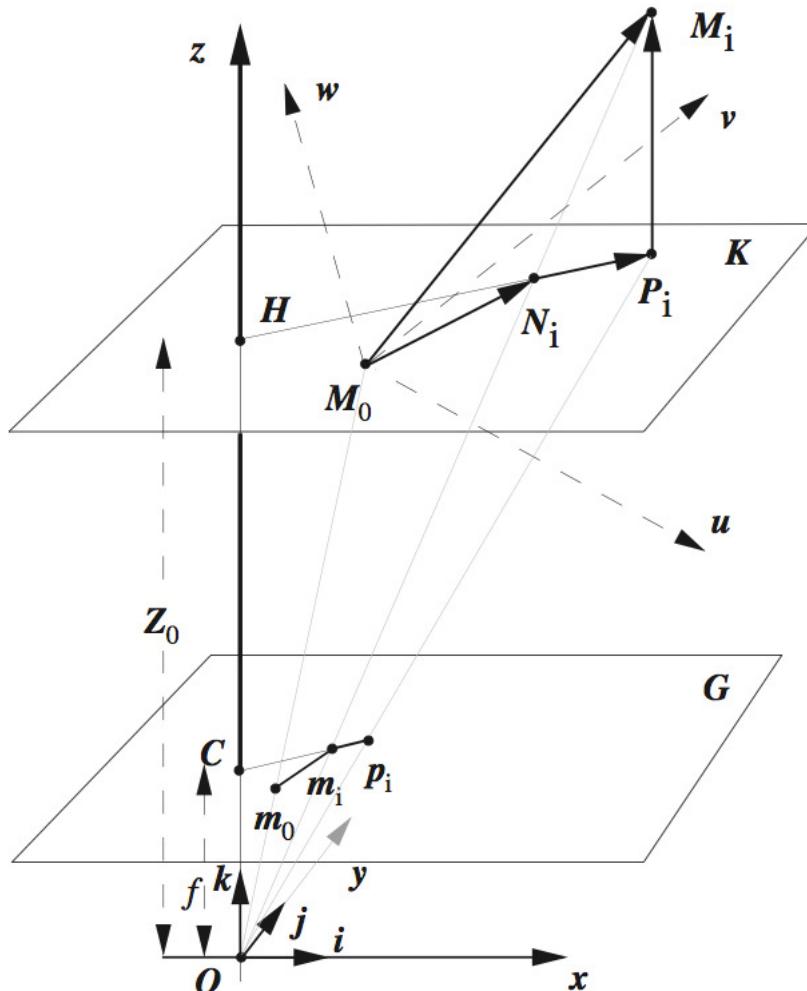


Figura 6.1: Proyección en perspectiva (m_i) y SOP (p_i) para un punto del modelo 3D M_i y un punto de referencia del modelo M_0 . Fuente: [?].

6.2.2. SOP: Scaled Orthographic Projection

La proyección ortogonal escalada(SOP) es una aproximación a la proyección perspectiva. En esta aproximación se supone que las profundidades Z_i de diferentes puntos M_i en el eje de coordenadas de la cámara no difieren mucho entre sí, y por lo tanto se asume que todos los puntos M_i tienen la misma profundidad que el punto M_0 .

Para un punto M_i la proyección perspectiva sobre el plano imagen estaría dada por:

$$x_i = fX_i/Z_i, \quad y_i = fY_i/Z_i,$$

mientras que la proyección SOP esta dada por:

$$x'_i = fX_i/Z_0, \quad y'_i = fY_i/Z_0.$$

De aquí en más las proyecciones SOP de los puntos M_i se identificaran como p_i , mientras que las proyecciones perspectivas, que son los puntos que se detectan en la imagen, se identifican como m_i . Al término $s = f/Z_0$ se lo conoce como el factor de escala de la SOP. Se puede ver que para el caso particular del punto M_0 la proyección perspectiva m_0 y la SOP p_0 coinciden.

En la figura 6.1 se puede ver como se construye la SOP. Primero se realiza la proyección ortogonal de todos los puntos M_i sobre K , el plano paralelo al plano imagen que pasa por el punto M_0 . Las proyecciones de los puntos M_i sobre K se llaman P_i . El segundo paso consiste en hacer la proyección perspectiva de los puntos P_i sobre el plano imagen G para obtener finalmente los puntos p_i . En la figura también se puede ver que el tamaño del vector $m_0 p_i$ es s veces el tamaño de $M_0 P_i$. Teniendo esto en cuenta se pueden expresar las coordenadas de p_i como:

$$\begin{aligned} x'_i &= fX_0/Z_0 + f(X_i - X_0)/Z_0 = x_0 + s(X_i - X_0) \\ y'_i &= y_0 + s(Y_i - Y_0) \end{aligned} \tag{6.1}$$

6.2.3. Ecuaciones para calcular la proyección perspectiva

Como se mencionó anteriormente la pose queda determinada si se conocen los vectores \mathbf{i}, \mathbf{j} y la coordenada Z_0 del vector de translación. Las ecuaciones que vinculan estas variables son:

$$M_0 M_i \frac{f}{Z_0} \mathbf{i} = x_i(1 + \varepsilon_i) - x_0 \tag{6.2}$$

$$M_0 M_i \frac{f}{Z_0} \mathbf{j} = y_i(1 + \varepsilon_i) - y_0 \tag{6.3}$$

donde ε_i se define como

$$\varepsilon_i = \frac{1}{Z_0} M_0 M_i \mathbf{k} \tag{6.4}$$

Se puede ver que los términos $x_i(1 + \varepsilon_i)$ y $y_i(1 + \varepsilon_i)$ son las coordenadas (x'_i, y'_i) de la SOP, en el caso en que la pose esta determinada. En la expresión de ε_i en 9.7, el producto escalar da la coordenada z de $M_0 M_i$, $Z_i - Z_0$, entonces se tiene que

$$(1 + \varepsilon_i) = \frac{Z_i - Z_0}{Z_0} + 1 = \frac{Z_i}{Z_0}$$

ademas se tiene la proyección perspectiva $x_i = fX_i/Z_i$, combinando las dos expresiones se tiene

$$x_i(1 + \varepsilon_i) = f \frac{X_i}{Z_i} \frac{Z_i}{Z_0} = f \frac{X_i}{Z_0}$$

que es la coordenada x'_i del punto p_i .

6.2.4. Algoritmo

Las ecuaciones 9.2 y 9.3 se puede reescribir como:

$$M_0 M_i \mathbf{I} = x_i(1 + \varepsilon_i) - x_0 \quad (6.5)$$

$$M_0 M_i \mathbf{J} = y_i(1 + \varepsilon_i) - y_0 \quad (6.6)$$

en donde

$$\mathbf{I} = \frac{f}{Z_0} \mathbf{i} = s \cdot \mathbf{i}, \quad \mathbf{J} = \frac{f}{Z_0} \mathbf{j} = s \cdot \mathbf{j} \quad (6.7)$$

Si se conociera el valor de ε_i , las ecuaciones 6.5 y 6.6 representan un sistema de ecuaciones en que las incógnitas son los vectores \mathbf{I} y \mathbf{J} . Una vez obtenidos estos vectores es si pueden obtener los versores \mathbf{i} y \mathbf{j} normalizando, y Z_0 se obtiene de la norma de cualquiera de los vectores \mathbf{I} o \mathbf{J} . A esta parte del algoritmo se le llama *POS* (*Pose from Orthography and Scaling*), ya que estima la pose a partir de las proyecciones SOP de los puntos M_i .

Si se conocieran los valores exactos de los ε_i la pose obtenida de resolver el sistema de ecuaciones sería la pose real del objeto, como no se conocen los valores exactos de ε_i se utiliza un método iterativo que tiende a la solución buscada. En la primera iteración se le toma $\varepsilon_i = 0$, es decir, $p_i = m_i$. Esta suposición es razonable si se tiene que la relación distancia cámara objeto - profundidad del objeto es grande. La ecuación para un punto cualquiera está dada por:

$$\begin{aligned} M_0 M_i \cdot \mathbf{I} &= x'_i - x_0 \\ M_0 M_j \cdot \mathbf{J} &= y'_i - y_0 \end{aligned} \quad (6.8)$$

Si se escribe la ecuación 6.8 para los n puntos del modelo, se tiene un sistema de n ecuaciones con \mathbf{I} y \mathbf{J} como incógnitas

$$\begin{aligned} A\mathbf{I} &= x' - x_0 \\ A\mathbf{J} &= y' - y_0 \end{aligned} \quad (6.9)$$

\mathbf{A} es una matriz $n \times 3$ con las coordenadas de los puntos del modelo M_i en el marco de coordenadas del objeto. Si se tienen mas de 4 puntos y no son coplanares, la matriz \mathbf{A} es de rango 3, y las soluciones al sistema están dadas por

$$\begin{aligned} I &= \mathbf{B}x' - x_0 \\ J &= \mathbf{B}y' - y_0 \end{aligned} \quad (6.10)$$

donde \mathbf{B} es la pseudo inversa de la matriz \mathbf{A} . Se debe notar que la matriz \mathbf{B} depende únicamente de la geometría del modelo que se asume conocida, por lo tanto solo es necesario calcular la matriz \mathbf{B} una sola vez.

Una vez obtenidos \mathbf{I} y \mathbf{J} se calculan s y los versores \mathbf{i} , \mathbf{j} y \mathbf{k}

$$s = (|\mathbf{I}| |\mathbf{J}|)^{1/2} \quad (6.11a)$$

$$i = \frac{\mathbf{I}}{s} \quad (6.11b)$$

$$j = \frac{\mathbf{J}}{s} \quad (6.11c)$$

$$k = i \times j \quad (6.11d)$$

El vector traslación del centro del objeto al centro de la cámara es el vector OM_0

$$OM_0 = \frac{Z_0}{f} Om_0 = \frac{Om_0}{s} \quad (6.12)$$

El vector Om_0 es conocido ya que se conocen las coordenadas de los puntos m_i , en particular m_0 .

Una vez que se calcularon \mathbf{i} , \mathbf{j} , \mathbf{k} y \mathbf{T} se calculan los valores actualizados de ε_i según la ecuación 9.7. Si la variación de los ε_i es mayor a un determinado umbral, se repite el procedimiento actualizando las proyecciones SOP en 6.9, si es menor al umbral se deja de iterar y se guarda la pose calculada.

6.2.5. POSIT para puntos coplanares

Como se mencionó anteriormente, el algoritmo POSIT no funciona en el caso en que los puntos pertenecen a un mismo plano. Como los marcadores utilizados son planos, se buscó una versión de POSIT que resuelve el problema de la estimación de pose para este caso. El algoritmo fue escrito por *Denis Oberkampf, Daniel DeMenthon y Larry Davis* en [?].

Para entender cual es el problema de trabajar con puntos coplanares se explica la situación desde un punto de vista geométrico. Como se vio anteriormente

$$M_0M_i \cdot \mathbf{I} = x'_i - x_0.$$

Esto quiere decir que si se toma que la base de \mathbf{I} en M_0 , la punta del vector \mathbf{I} se proyecta sobre el vector M_0M_i en un punto H_{x_i} , entonces todas las posibles puntas del vector \mathbf{I} se encuentran en el plano perpendicular a M_0M_i que pasa por el punto H_{x_i} . Si se tuvieran 4 puntos no coplanares M_0, M_1, M_2 y M_3 , el vector \mathbf{I} quedaría determinado. La base de \mathbf{I} estaría en M_0 y la punta estaría en la intersección de los planos perpendiculares a M_0M_1, M_0M_2 y M_0M_3 por los puntos H_{x_1}, H_{x_2} y H_{x_3} respectivamente. Para este caso el sistema definido en 6.9 es de rango 3.

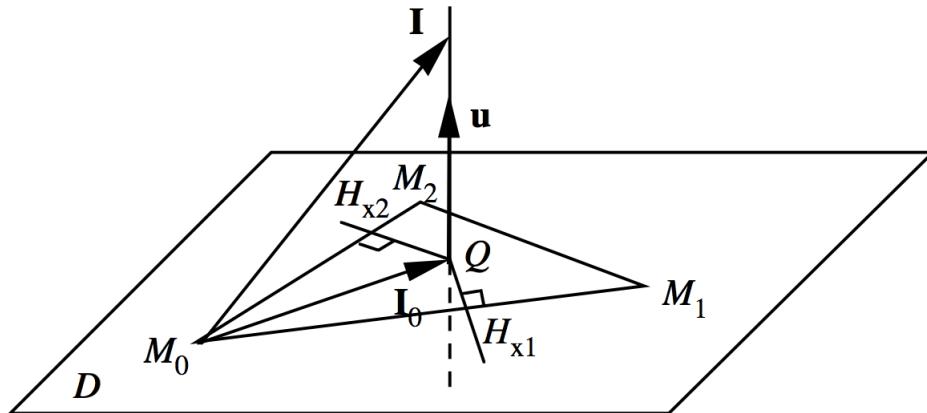


Figura 6.2: Configuración de puntos coplanares pertenecientes al plano D . Los planos perpendiculares que pasan por los puntos H_{x_1} y H_{x_2} se intersectan en un recta que pasa por el punto Q . Se puede ver que si hubiera un 4^{to} punto, el plano perpendicular correspondiente haría aparecer 2 rectas paralelas. Fuente: [?].

Si los puntos son coplanares, los vectores M_0M_1, M_0M_2 y M_0M_3 son todos coplanares y los planos perpendiculares que pasan por los puntos H_{x_1}, H_{x_2} y H_{x_3} , se intersectan todos en una línea o en dos líneas paralelas por lo tanto hay infinitas soluciones para el vector \mathbf{I} . En este caso el sistema de ecuaciones 6.9 queda de rango 2. El vector solución que se obtiene al realizar la pseudo inversa

de \mathbf{A} es el que está a menor distancia de los planos, en la figura 6.2 es el vector \mathbf{I}_0 . Esta la solución no es la solución al problema de los vectores de rotación, las soluciones se pueden expresar como

$$\begin{aligned}\mathbf{I} &= \mathbf{I}_0 + \lambda \mathbf{u} \\ \mathbf{J} &= \mathbf{J}_0 + \mu \mathbf{u}\end{aligned}\quad (6.13)$$

donde \mathbf{u} es un versor perpendicular al plano de los puntos, \mathbf{J}_0 se calcula de manera análoga a \mathbf{I}_0 y λ y μ son las coordenadas de \mathbf{I} y \mathbf{J} según el versor \mathbf{u} . Para encontrar las soluciones hay que calcular el versor \mathbf{u} y los valores de λ y μ .

Como el vector \mathbf{u} es perpendicular al plano de los puntos característicos se cumple $M_0 M_i \cdot \mathbf{u} = 0$, se puede hallar entonces como la base del núcleo de la matriz \mathbf{A} . En la práctica este vector se halla a partir de la descomposición SVD de la matriz \mathbf{A} . La descomposición en valores singulares de la matriz \mathbf{A} queda:

$$\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^T \quad (6.14)$$

donde $\mathbf{U} \in \Re^{n \times n}$ es ortogonal, $\Sigma \in \Re^{n \times 3}$ es diagonal, con los valores singulares en la diagonal y $\mathbf{V} \in \Re^{3 \times 3}$ es ortogonal. Como la matriz \mathbf{A} es de rango 2, los dos primeros vectores columna de la matriz \mathbf{V} corresponden a la base de todos los puntos que pertenecen al plano del modelo, mientras que el último vector de \mathbf{V} es la base del núcleo de \mathbf{A} , o sea el vector \mathbf{u} . El cálculo \mathbf{u} se realiza junto al cálculo de la matriz \mathbf{B} , ya que para ambos es necesario hacer la descomposición SVD de \mathbf{A} .

Para calcular los valores de λ y μ se utilizan las condiciones de que \mathbf{I} y \mathbf{J} tienen que ser perpendiculares entre sí y del mismo largo. Como tienen que ser perpendiculares se tiene que

$$\mathbf{I} \cdot \mathbf{J} = (\mathbf{I}_0 + \lambda \mathbf{u}) \cdot (\mathbf{J}_0 + \mu \mathbf{u}) = 0$$

entonces se tiene que

$$\lambda \mu = -\mathbf{I}_0 \cdot \mathbf{J}_0 \quad (6.15)$$

Como tienen que ser del mismo largo se tiene que

$$(\mathbf{I}_0 + \lambda \mathbf{u}) \cdot (\mathbf{I}_0 + \lambda \mathbf{u}) = (\mathbf{J}_0 + \mu \mathbf{u}) \cdot (\mathbf{J}_0 + \mu \mathbf{u}) \Leftrightarrow \lambda^2 - \mu^2 = \mathbf{J}_0^2 - \mathbf{I}_0^2 \quad (6.16)$$

Se define el número complejo $C = \lambda + i\mu$, si se eleva al cuadrado queda $C^2 = \lambda^2 - \mu^2 + i\lambda\mu$. Utilizando 6.15 y 6.16 se llega a que

$$C^2 = \mathbf{J}_0^2 - \mathbf{I}_0^2 - 2i\mathbf{I}_0 \cdot \mathbf{J}_0 \quad (6.17)$$

por lo que λ y μ pueden calcularse como las partes real e imaginaria del complejo C^2 . Para hallar la raíces de C^2 , se expresa en forma polar:

$$\begin{aligned}C^2 &= [R, \Theta], \text{ donde} \\ R &= \left((\mathbf{J}_0^2 - \mathbf{I}_0^2)^2 + 4(\mathbf{I}_0 \cdot \mathbf{J}_0)^2 \right)^{1/2} \\ \Theta &= \arctan \left(\frac{-2\mathbf{I}_0 \cdot \mathbf{J}_0}{\mathbf{J}_0^2 - \mathbf{I}_0^2} \right), \text{ si } \mathbf{J}_0^2 - \mathbf{I}_0^2 > 0, \text{ y} \\ \Theta &= \arctan \left(\frac{-2\mathbf{I}_0 \cdot \mathbf{J}_0}{\mathbf{J}_0^2 - \mathbf{I}_0^2} \right) + \pi, \text{ si } \mathbf{J}_0^2 - \mathbf{I}_0^2 < 0 \\ \text{si } \mathbf{J}_0^2 - \mathbf{I}_0^2 &= 0 \text{ se toma } \Theta = -\text{signo}(\mathbf{I}_0 \cdot \mathbf{J}_0) \frac{\pi}{2}, \text{ y } R = |2\mathbf{I}_0 \cdot \mathbf{J}_0|\end{aligned}$$

Se obtienen 2 raíces, $C = [\rho, \theta]$, y $C = [\rho, \theta + \pi]$, donde

$$\rho = \sqrt{R}, \text{ y } \theta = \frac{\Theta}{2}$$

como se mencionó anteriormente λ y μ son las partes real e imaginaria de C , por lo tanto

$$\lambda_1 = \rho \cos \theta, \quad \mu_1 = \rho \sin \theta \quad (6.18a)$$

$$\lambda_2 = -\rho \cos \theta, \quad \mu_2 = -\rho \sin \theta \quad (6.18b)$$

Esto quiere decir que se obtienen dos soluciones para \mathbf{I} y \mathbf{J}

$$\mathbf{I}_1 = \mathbf{I}_0 + \rho \cos \theta \mathbf{u}, \quad \mathbf{J}_1 = \mathbf{J}_0 + \rho \sin \theta \mathbf{u} \quad (6.19a)$$

$$\mathbf{I}_2 = \mathbf{I}_0 - \rho \cos \theta \mathbf{u}, \quad \mathbf{J}_2 = \mathbf{J}_0 - \rho \sin \theta \mathbf{u} \quad (6.19b)$$

Como el vector \mathbf{u} es perpendicular la plano del objeto, la solución encontrada en 6.19a es simétrica a 6.19b. Desde el punto de vista de la cámara, se puede ver que las dos posibles soluciones son aquellas que tienen la misma proyección SOP. Esto es equivalente a decir que para una misma proyección SOP hay dos posibles poses que que verifican las ecuaciones 6.9.

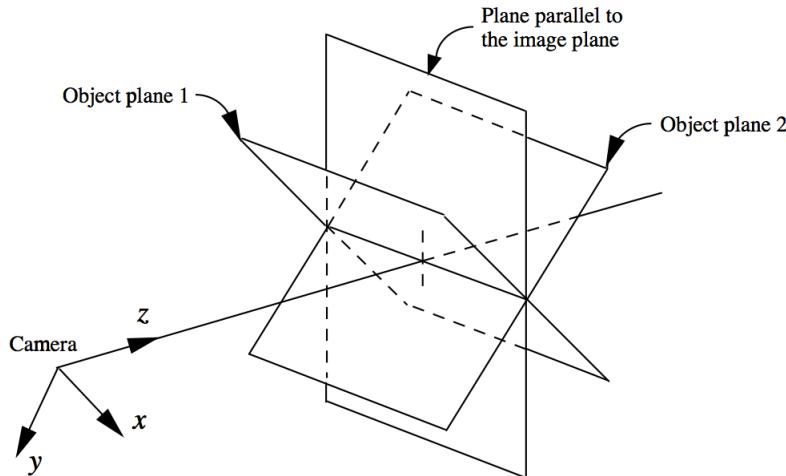


Figura 6.3: Dos objetos dando la misma proyección SOP. Fuente: [?].

Por lo tanto se toman las soluciones $(\mathbf{I}_1, \mathbf{J}_1)$ y $(\mathbf{I}_2, \mathbf{J}_2)$ y se calculan las poses. Como las dos poses son simétricas respecto a un plano paralelo al plano imagen, puede pasar que una pose de una solución en la que los los puntos del objeto queden ubicados detrás de la cámara. Por lo tanto previo a dar las dos soluciones como válidas hay que verificar esto.

En el caso en que las dos soluciones sean validas para todas las iteraciones, el número de poses posibles sería 2^n a lo largo de n iteraciones. En la práctica se manejan menos soluciones posibles. Se diferencian dos casos:

- . Si se tiene que solo una de las dos primeras poses calculadas es válida, en las siguientes iteraciones se da mismo comportamiento, por lo que hay solo un camino a seguir.
- . Si se tiene que las dos primeras poses calculadas son válidas, se abren dos posibles ramas. En la segunda iteración cada rama da lugar a dos nuevas poses, pero en este caso se toma la pose que da menos error de reproyección.

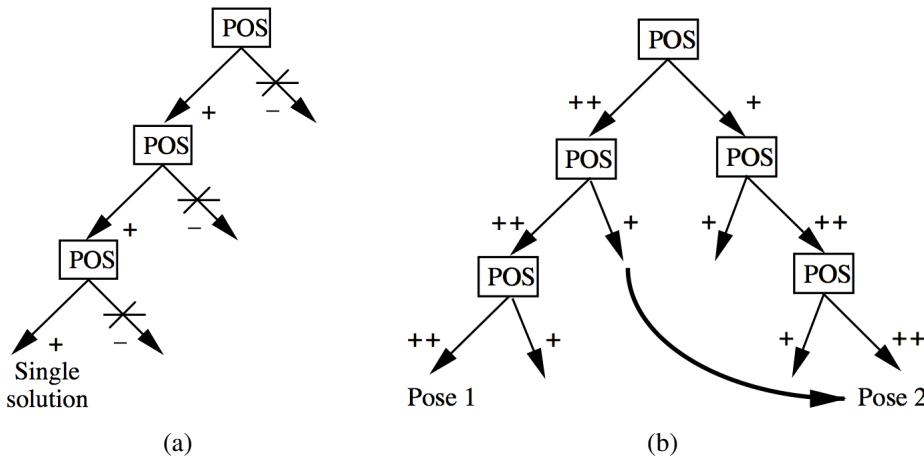


Figura 6.4: (a):Caso en el que solo una pose de las dos iniciales es coherente, también en las siguientes iteraciones solo una de las dos poses es posible, se tiene un única solución. (b): Caso en el que en cada paso hay dos posibilidades, se opta por la mejor pose(++) mejor pose, + peor pose) en cada rama. Fuente: [?].

6.3. SoftPOSIT

Hasta aquí se vio el algoritmo POSIT que permite obtener la pose de un modelo respecto a la cámara para el caso en que se tienen correspondencias entre puntos del modelo y puntos característicos en la imagen. Como se vio en el capítulo 2 obtener correspondencias entre puntos detectados en una imagen y el modelo real puede ser complicado. Por este motivo se estudio el algoritmo SoftPOSIT desarrollado por *Philip David, Daniel DeMenthon, Ramani Duraiswami y Hanan Samet* presentado en [5]. Este algoritmo recibe como entrada el modelo 3D y una lista de puntos detectados en la imagen para los cuales no se sabe como se relacionan con los puntos del modelo. Utiliza un método llamado *softassign* para resolver las correspondencias y luego que tiene las correspondencias utiliza una versión modificada de POSIT.

6.3.1. Modern POSIT

Como se mencionó anteriormente, SoftPOSIT utiliza una versión modificada de POSIT llamada Modern POSIT. POSIT clásico requiere que se conozca cual es el punto de referencia en el modelo y en la imagen, ya que de estos datos se calcula el vector de traslación. Para el caso de SoftPOSIT no es posible saber de antemano cual es el punto de referencia del modelo ya que no se tienen las correspondencias. Modern POSIT calcula la pose, sabiendo las correspondencias, pero sin utilizar ningún punto en particular como referencia. Ademas la pose se calcula minimizando una función que mide la distancia entre la proyección SOP y los puntos estimados en cada iteración.

El punto M_0 origen del sistema de coordenadas del objeto no es conocido, por lo tanto tampoco se conoce su correspondiente m_0 en el plano imagen. En la ecuación 6.8 que se presentó en la sección 6.2.4 se conocían las coordenadas del punto m_0 , por lo que las incógnitas de esta ecuación eran solamente los vectores \mathbf{i}, \mathbf{j} .

$$\begin{aligned} M_0 M_i \cdot \mathbf{I} &= x'_i - x_0 \\ M_0 M_j \cdot \mathbf{J} &= y'_i - y_0 \end{aligned}$$

En este caso no se conocen las coordenadas de m_0 , por lo que también hace falta calcularlas para

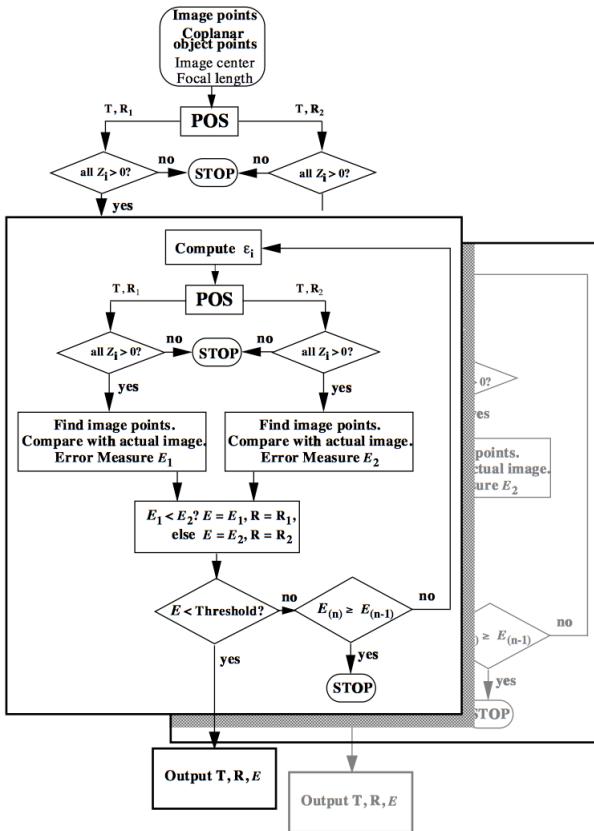


Figura 6.5: Diagrama de flujo del algoritmo para puntos coplanares, E es el error de reproyección, la condición $Z_i > 0$ verifica que los puntos reproyectados están por delante del plano imagen. Fuente: [?].

obtener el vector de traslación. Sabiendo que

$$X_0 = x_0/s$$

$$Y_0 = y_0/s$$

se puede reescribir la ecuación 6.8 como

$$\begin{aligned} x'_i &= M_0 M_i \cdot s\mathbf{i} + sX_0 \\ y'_i &= M_0 M_j \cdot s\mathbf{j} + sY_0 \end{aligned} \tag{6.20}$$

El sistema a resolver queda

$$A \cdot \begin{bmatrix} \mathbf{I} & \mathbf{J} \\ sX_0 & sY_0 \end{bmatrix} = \begin{bmatrix} x' & y' \end{bmatrix} \tag{6.21}$$

donde la matriz A son los puntos del modelo 3D en coordenadas homogéneas. Este sistema se puede resolver, utilizando mínimos cuadrados, como se vio en la sección 6.2.4. Se calculan la pseudo inversa de la matriz A y luego se obtienen \mathbf{i} , \mathbf{j} y \mathbf{k} como se vio en 6.11. Finalmente el vector de traslación se obtiene como.

$$X_0 = \frac{(sX_0)}{s} \quad Y_0 = \frac{(sY_0)}{s} \quad Z_0 = \frac{f}{s} \tag{6.22}$$

Sin embargo se propone un método que busca minimizar la distancia al cuadrado entre las proyecciones SOP de los puntos M_i y las proyecciones SOP calculadas en cada iteración. En la

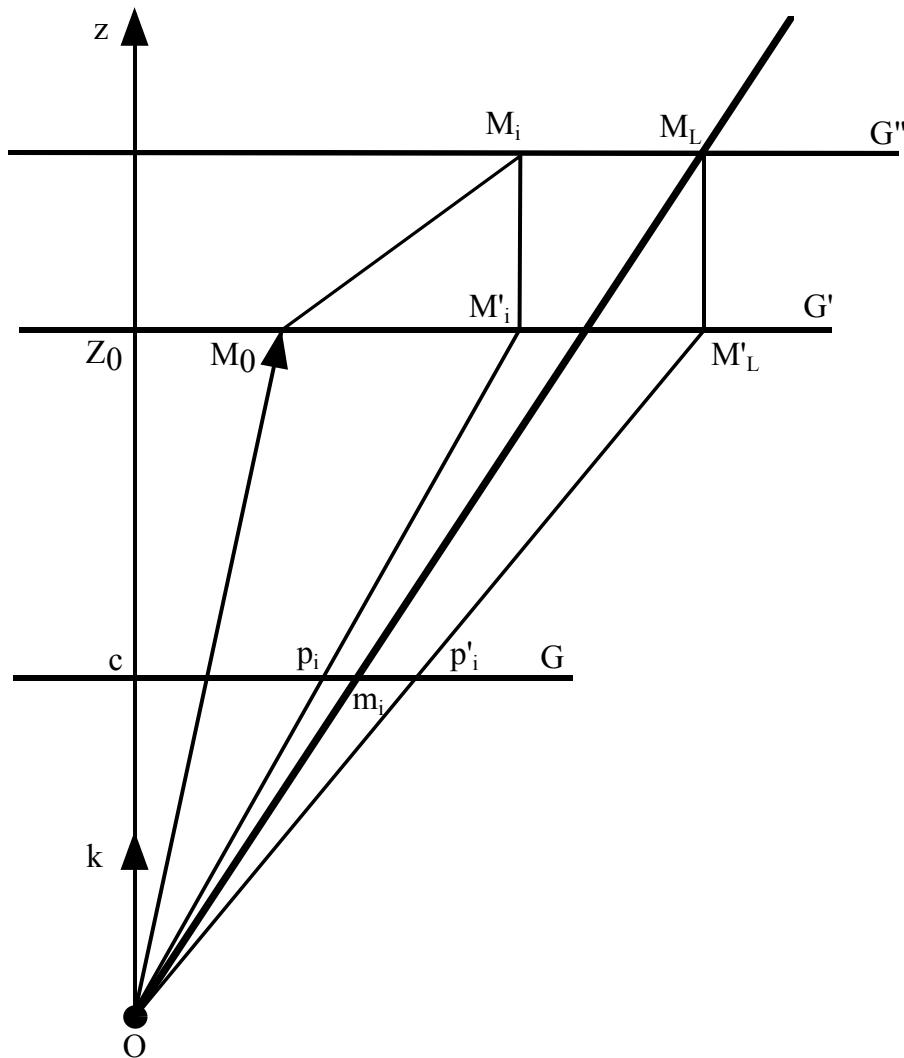


Figura 6.6: Interpretación geométrica de POSIT. El punto p_i es la proyección SOP de M_i que es el término de la derecha de la ecuación 6.20. El punto p'_i es la proyección SOP de M_L , ubicado en la línea de vista de m_i , corresponde al término de la izquierda de la ecuación 6.20. Para que la ecuación se satisfaga se tiene que cumplir que p_i y p'_i sean iguales. Fuente: [5].

figura 6.3.1 se puede ver geométricamente cual es la distancia que se busca minimizar. En el término de la derecha de 6.20, se tiene la proyección SOP de M_i , p_i . Las coordenadas de este punto son

$$p_i = s(M_0M_i \cdot \mathbf{i} + X_0, M_0M_i \cdot \mathbf{j} + Y_0).$$

Por otro lado, en el término de la izquierda de 6.20, se tienen las coordenadas del punto p'_i

$$p'_i = (1 + \varepsilon_i)(x_i, y_i).$$

que es la proyección SOP de la intersección de la línea de vista del punto m_i con el plano G'' , esto esta demostrado en [5]. La pose encontrada es correcta cuando ambos lados de 6.20 son iguales. Por lo tanto la ecuación que se busca minimizar es la siguiente:

$$E = \sum_i \left((\mathbf{Q}_1 \cdot M_0M_i - (1 + \varepsilon_i)x_i)^2 + (\mathbf{Q}_2 \cdot M_0M_i - (1 + \varepsilon_i)y_i)^2 \right) \quad (6.23)$$

donde

$$\begin{aligned}\mathbf{Q}_1 &= s(i, X_0) \\ \mathbf{Q}_2 &= s(j, Y_0)\end{aligned}\quad (6.24)$$

y M_0M_i se toma en coordenadas homogéneas.

Los vectores \mathbf{Q}_1 y \mathbf{Q}_2 son aquellos que minimizan el valor E , por lo tanto se despejan de derivar la expresión de E e igualarla a cero. La expresión para calcular \mathbf{Q}_1 y \mathbf{Q}_2 queda:

$$\mathbf{Q}_1 = \left(\sum_i M_0 M_i^T M_0 M_i \right)^{-1} \left(\sum_i (1 + \varepsilon_i) x_i M_0 M_i \right) \quad (6.25)$$

$$\mathbf{Q}_2 = \left(\sum_i M_0 M_i^T M_0 M_i \right)^{-1} \left(\sum_i (1 + \varepsilon_i) y_i M_0 M_i \right) \quad (6.26)$$

La matriz $L = (\sum_i M_0 M_i^T M_0 M_i)$ es una matriz 4×4 y como solo depende de los puntos del modelo, puede ser calculada previamente.

Para calcular la pose se procede como sigue:

- 1 Se calculan los vectores \mathbf{Q}_1 y \mathbf{Q}_2 asumiendo que se conocen los valores de ε_i , para el paso inicial se supone que $\varepsilon_i = 0$.
- 2 Con los vectores \mathbf{Q}_1 y \mathbf{Q}_2 calculados se calculan los ε_i corregidos.

Cuando E es menor a determinado umbral el algoritmo se detiene y se obtiene la pose calculada.

6.3.2. Calculo de pose sin correspondencias

Se tienen N puntos detectados en la imagen y M puntos en el modelo. Cuando no se conocen las correspondencias cada punto detectado p_j es candidato a corresponderse con cualquier punto del modelo M_i . La distancia que se busca minimizar es

$$d_{ji}^2 = (\mathbf{Q}_1 \cdot M_i M_0 - (1 + \varepsilon_i) x_j)^2 + (\mathbf{Q}_2 \cdot M_i M_0 - (1 + \varepsilon_i) y_j)^2$$

Se puede ver que para cada punto de modelo hay N candidatos, la distancia d_{ji}^2 da una idea de que tan cerca esta de ser el correspondiente. Para resolver el problema de estimar la pose y las correspondencias simultáneamente se busca minimizar la siguiente función:

$$\begin{aligned}E &= \sum_{j=1}^N \sum_{i=1}^M m_{ji} (d_{ji}^2 - \alpha) \\ &= \sum_{j=1}^N \sum_{i=1}^M m_{ji} \left((\mathbf{Q}_1 \cdot M_i M_0 - (1 + \varepsilon_i) x_j)^2 + (\mathbf{Q}_2 \cdot M_i M_0 - (1 + \varepsilon_i) y_j)^2 \right)\end{aligned}\quad (6.27)$$

donde m_{ji} son pesos para cada una de las distancias d_{ji}^2 . Los pesos m_{ji} forman lo que se llama matriz de asignación, en esta matriz se puede ver el grado de correspondencia de cualquier punto detectado con cualquier punto del modelo. El valor α es la tolerancia que se le da a la medida de distancia.

Las expresiones para los vectores \mathbf{Q}_1 y \mathbf{Q}_2 se modifican

$$\mathbf{Q}_1 = \left(\sum_{i=1}^M m'_i M_0 M_i^T M_0 M_i \right)^{-1} \left(\sum_{j=1}^N \sum_{i=1}^M m_{ji} (1 + \varepsilon_i) x_j M_0 M_i \right) \quad (6.28)$$

$$\mathbf{Q}_2 = \left(\sum_{i=1}^M m'_i M_0 M_i^T M_0 M_i \right)^{-1} \left(\sum_{j=1}^N \sum_{i=1}^M m_{ji} (1 + \varepsilon_i) y_j M_0 M_i \right) \quad (6.29)$$

donde $m'_i = \sum_{j=1}^N m_{ji}$. El termino $L = \sum_{i=1}^M m'_i M_0 M_i^T M_0 M_i$ es una matriz 4×4 , para este caso L no se puede calcular previamente porque la matriz de asignación cambia en cada iteración.

Para minimizar E se procede como sigue:

- 1 Se calculan las variables de la matriz de asignación asumiendo todo lo demás conocido.
- 2 Se calculan los vectores \mathbf{Q}_1 y \mathbf{Q}_2 asumiendo que se conocen los valores de ε_i , para el paso inicial se supone que $\varepsilon_i = 0$.
- 3 Con los vectores \mathbf{Q}_1 y \mathbf{Q}_2 calculados se calculan los ε_i corregidos.

Esto se repite hasta que la pose converge.

6.3.3. Matriz de asignación

Se busca tener una matriz m que indique las correspondencias entre los N puntos detectados y los M puntos en del modelo, y ademas minimice E . La matriz de asignación tiene las siguientes características:

- . Tiene $N+1$ filas y $M+1$ columnas.
- . $m_{ji} \in [0, 1]$. Si $m_{ji} = 1$ quiere decir que el punto detectado p_j se corresponde con el punto del modelo M_i .
- . La fila $N+1$ y la columna $M+1$ se utilizan para ver si alguna correspondencia en esa fila o columna. Por ejemplo si el elemento j de la columna $M+1$ es 1, significa que el punto detectado p_j no se corresponde con ningún punto del modelo.
- . La suma de los elementos a lo largo de cualquier fila o columna es siempre 1.

Para obtener una matriz m que cumpla con las características mencionadas se utiliza una técnica llamada *softassign*. Se comienza con una matriz m^0 en la que los elementos están dados por

$$m_{ji}^0 = \exp^{-\beta(d_{ji}^2 - \alpha)}$$

en donde β es una constante muy pequeña y la fila $N+1$ y la columna $M+1$ son inicializadas con constantes pequeñas. Luego se itera utilizando los siguientes pasos hasta obtener la matriz m .

- 1 Se normaliza cada fila y columna por la suma de los elementos de esa fila o columna respectivamente hasta que $\|m^i - m^{i-1}\|$ sea pequeña. La matriz resultante cumple que todas las filas y columnas suman 1.
- 2 Se incrementa el valor de β a medida que se itera. A medida que se agranda β cada fila y columna de m^0 es renormalizada, los términos m_{ji}^0 correspondientes a las d_{ji}^2 convergen a 1, mientras que los demás convergen a 0.

Al final del algoritmo se observa que la matriz m está muy cerca de ser una matriz de ceros y unos.

6.4. Modern POSIT Coplanar

La implementación que se usó en la aplicación es el modern POSIT adaptado para trabajar con puntos coplanares. Inicialmente se quiso desarrollar una versión de SoftPOSIT que trabajara con puntos coplanares, para ello previamente se desarrollo el modern POSIT coplanar a modo de prueba.

Como se vio en la sección de POSIT coplanar 6.2.5 cuando los puntos son coplanares, al resolver el sistema 6.9 se obtienen las proyecciones de los vectores \mathbf{i} y \mathbf{j} sobre el plano del objeto. Se utilizó el enfoque de POSIT moderno para hallar las proyecciones de \mathbf{i} y \mathbf{j} sobre el plano del modelo, así como los componentes en x e y del vector de traslación. Luego aplicando lo visto en POSIT coplanar se termino de calcular la pose.

Se definen los puntos $M_0M_i^*$ como los puntos M_0M_i sin la coordenada z , ya que la coordenada z es función de x e y . A su vez se definen los vectores \mathbf{Q}_1^* y \mathbf{Q}_2^* como los vectores \mathbf{Q}_1 y \mathbf{Q}_2 sin la componente según el eje w en el sistema de coordenadas del modelo. Teniendo esto en cuenta se tiene

$$E^* = \sum_i \left((\mathbf{Q}_1^* \cdot M_0M_i^* - (1 + \varepsilon_i)x_i)^2 + (\mathbf{Q}_2^* \cdot M_0M_i^* - (1 + \varepsilon_i)y_i)^2 \right)$$

Los vectores \mathbf{Q}_1^* y \mathbf{Q}_2^* se calculan de

$$\begin{aligned} \mathbf{Q}_1^* &= \left(\sum_{i=1}^M m_i' M_0M_i^{*T} M_0M_i^* \right)^{-1} \left(\sum_{j=1}^N \sum_{i=1}^M m_{ji} (1 + \varepsilon_i) x_j M_0M_i^* \right) \\ \mathbf{Q}_2^* &= \left(\sum_{i=1}^M m_i' M_0M_i^{*T} M_0M_i^* \right)^{-1} \left(\sum_{j=1}^N \sum_{i=1}^M m_{ji} (1 + \varepsilon_i) y_j M_0M_i^* \right) \end{aligned}$$

En este caso el término $L = \sum_{i=1}^M m_i' M_0M_i^{*T} M_0M_i^*$ es una matriz 3×3 . Una vez que se tienen los vectores \mathbf{Q}_1^* y \mathbf{Q}_2^* se procede como se vio en la sección 6.2.5

6.5. Resultados

Se realizo un comparación entre la implementación en C de POSIT clásico para puntos coplanares, obtenida de la sitio web de *Daniel DeMenthon*¹, y una versión desarrollada para esta aplicación de POSIT moderno para puntos coplanares.

Se utilizaron imágenes de prueba obtenidas con el iPad e imágenes sintéticas. Para ambos grupos de imágenes se midió el error de proyección obtenido entre los puntos del modelo y los puntos detectados. Para las imágenes sintéticas, como se cuenta con la información de la pose, se midió el error obtenido en cada ángulo y en la traslación.

Para el caso de las imágenes de prueba del iPad se eligieron 9 posiciones y en cada posición se sacaron 50 fotos. Con estas 450 fotos se obtuvo la estadística del funcionamiento de los algoritmos. En la figura 6.5 se puede ver una de las imágenes utilizadas para cada posición. También se utilizaron imágenes sintéticas en posiciones similares a las de las imágenes de prueba, se utilizaron 9 casos con 50 fotos por caso. Se partió de una posición base y se vario la posición muy poco, intentando simular el movimiento que se tiene tuvo al sacar las fotos con el iPad. En total se probaron 900 imágenes.

A las imágenes se les aplica todo el proceso, se realiza la detección y filtrado de segmentos, se calculan las correspondencias y luego se estima la pose. Para cada imagen se calcula el error de

¹http://www.cfar.umd.edu/~daniel/Site_2/Code.html

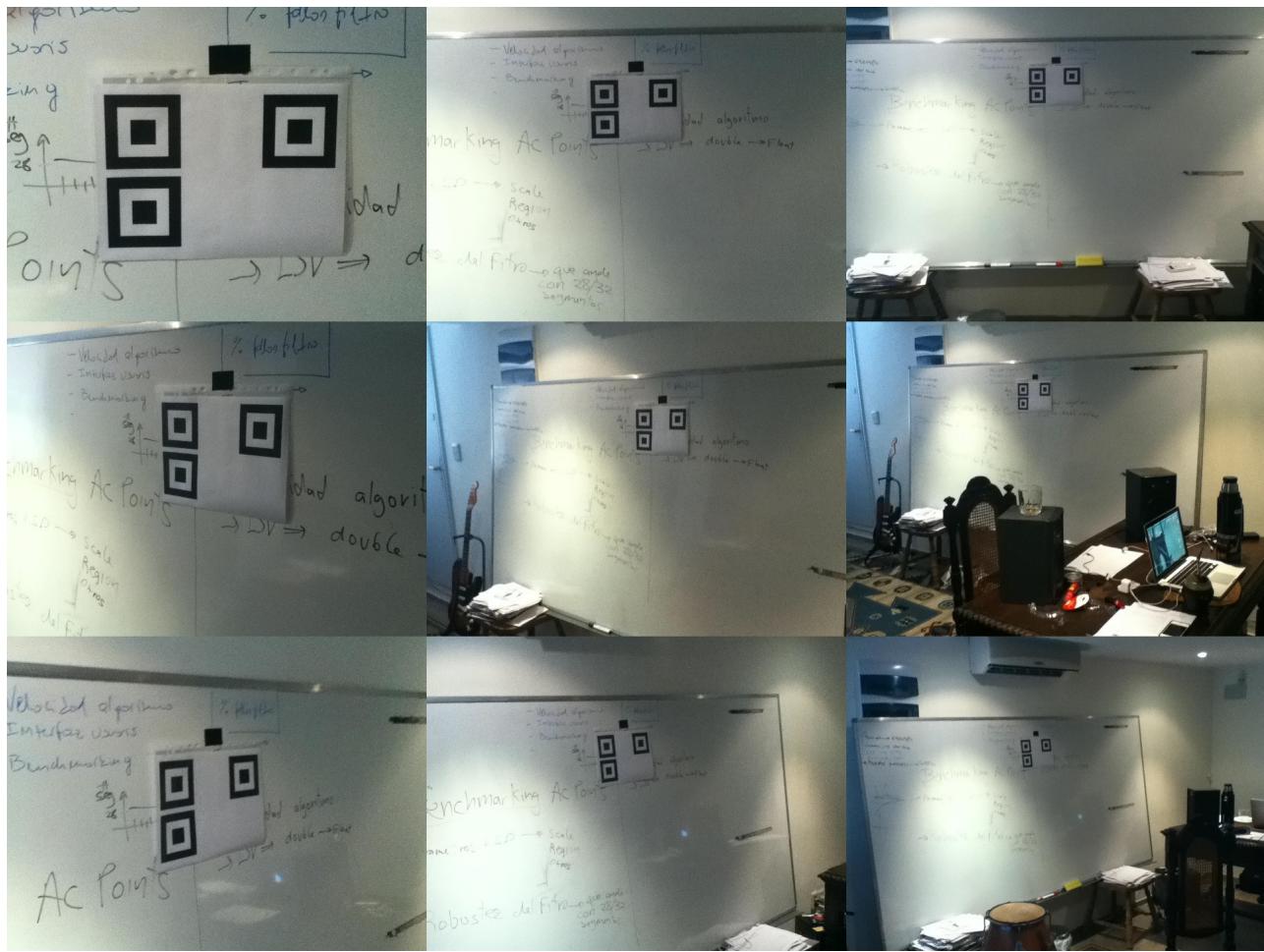


Figura 6.7: Posiciones que se utilizaron para las imágenes de prueba

proyección de cada punto, luego se promedian obteniendo una sola medida de error por imagen, esta medida es a su vez promediada con los errores obtenidos de la imágenes para un mismo caso. En las tablas hay valores que no pudieron ser calculados debido a que el filtro de segmentos no pudo detectar todos los segmentos. Estos comportamientos fueron discutidos en 2. En general se puede ver que el error de proyección y la varianza son un poco menores para el caso de modern POSIT.

Para otro grupo de imágenes sintéticas se comparó la pose original con la pose obtenida luego de aplicar el procesamiento, se relevo el desempeño de los algoritmos para rotaciones segun los tres ejes. En general se vio que la implementación de modern POSIT dio mejores resultados.

	Modern POSIT	Varianza	Classic POSIT	Varianza
Caso1	3.6136	0.8104	4.5979	1.1392
Caso2	0.8449	0.4153	0.9275	0.4415
Caso3	-	-	-	-
Caso4	1.5894	0.2600	1.1081	0.1696
Caso5	-	-	-	-
Caso6	-	-	-	-
Caso7	0.6742	0.1468	0.5416	0.1022
Caso8	-	-	-	-
Caso9	-	-	-	-

Tabla 6.1: Error de proyección de imágenes de pruebas

	Modern POSIT	Varianza	Classic POSIT	Varianza
Caso1	4.2712	0.3192	5.7352	0.4525
Caso2	1.0831	0.0375	1.0889	0.0358
Caso3	-	-	-	-
Caso4	0.7975	0.0169	0.9778	0.0185
Caso5	-	-	-	-
Caso6	-	-	-	-
Caso7	0.3796	0.0121	0.4761	0.0077
Caso8	-	-	-	-
Caso9	-	-	-	-

Tabla 6.2: Error de proyección en imágenes sintéticas

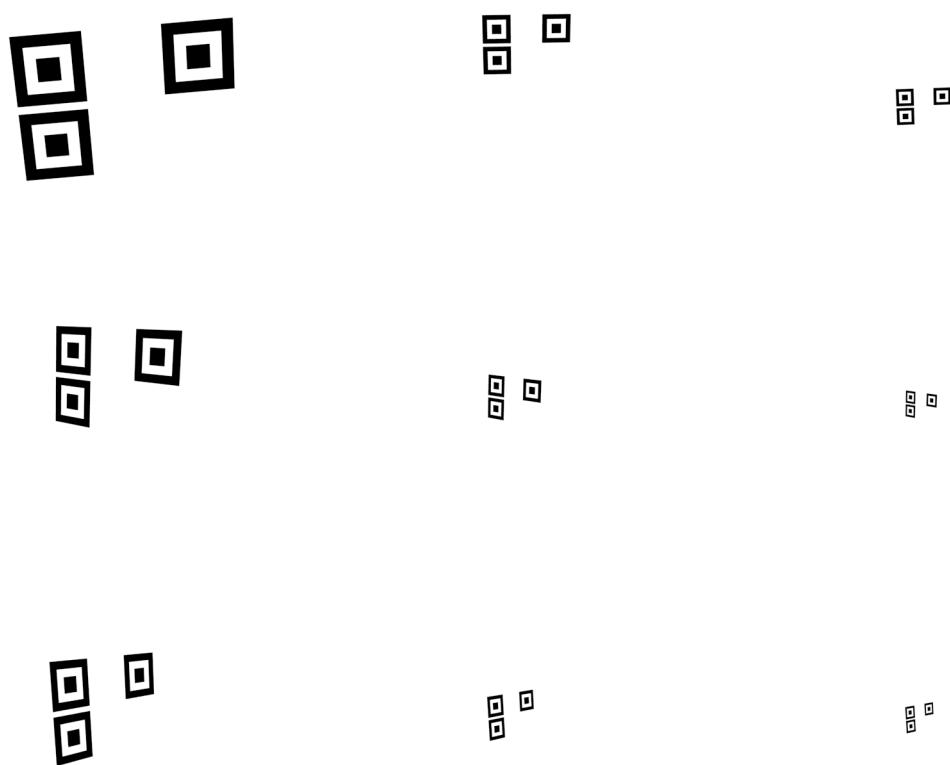


Figura 6.8: Posiciones que se utilizaron para las imágenes sintéticas

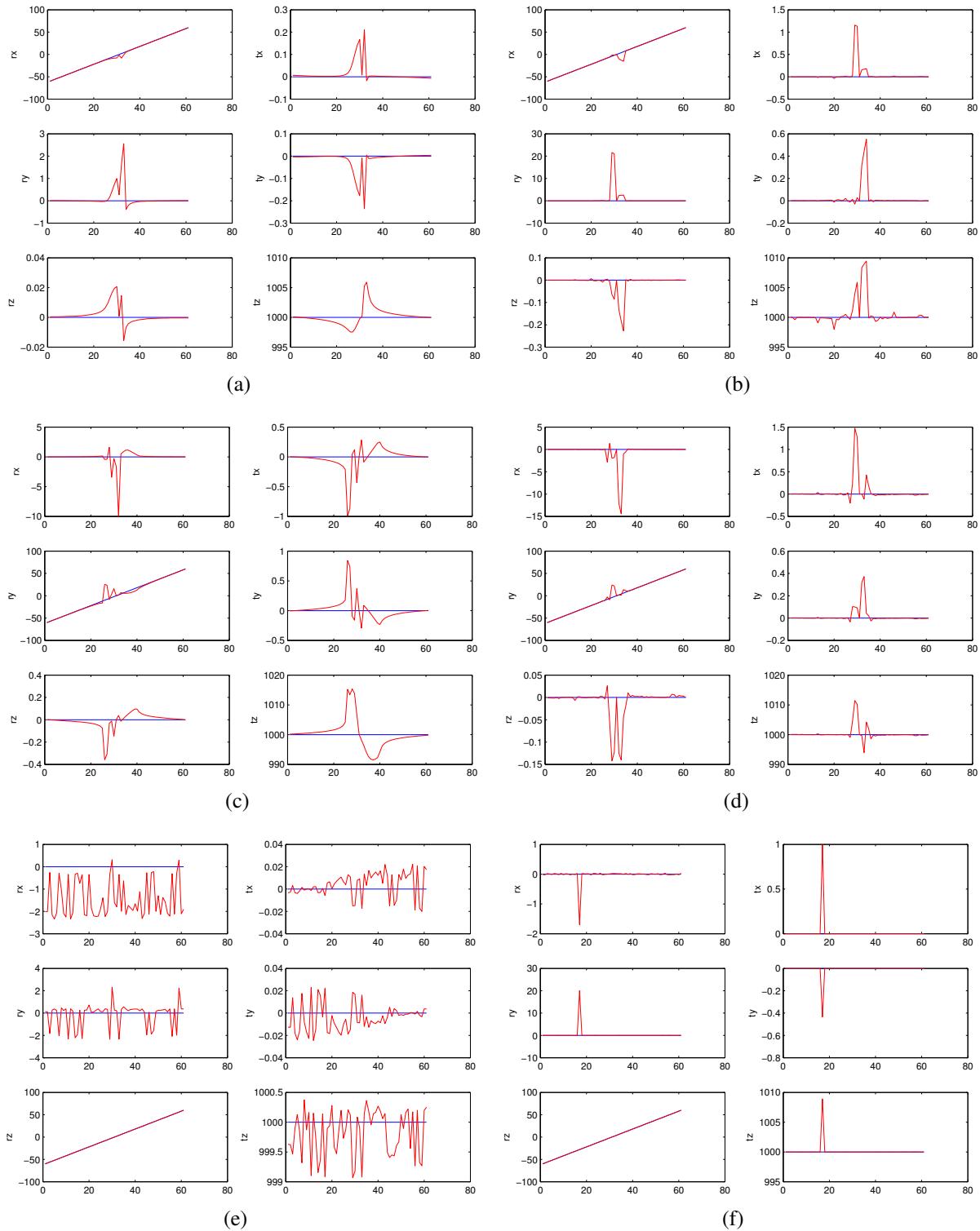


Figura 6.9: Rotación según eje x para Modern POSIT (a) y para Classic POSIT (b), rotación según eje y para Modern POSIT (c) y para Classic POSIT (d) y rotación según eje z para Modern POSIT (e) y para Classic POSIT (f)

CAPÍTULO 7

Filtrado de Kalman para estimación de pose

7.1. Kalman clásico

7.2. Kalman con sensores

7.3. Kalman robusto

CAPÍTULO 8

Herramientas de *rendering*

8.1. Introducción

Rendering es un término en inglés que denota el proceso de generar una imagen 2D a partir de un modelo digital 3D o un conjunto de ellos, a los que se les llama “escena”. Puede ser comparado con tomar una foto o filmar una escena en la vida real. Afortunadamente, existen varias herramientas de *rendering*, también llamadas “ motores de juego 3D”, para plataformas móviles, en especial que funcionen sobre iOS. Algunas de ellas son Unity 3D, ISGL3D, Cocos3D, Open GL ES y Shi-Va3D. A continuación serán comentadas tan sólo las consideradas durante el presente proyecto por ser populares y gratuitas.

La primera en ser tomada en cuenta fue “Open Graphics Library Embedded Systems” (Open GL ES), que es un subconjunto de las herramientas de gráficos 3D de Open GL. Fue diseñada para ser utilizada sobre sistemas embebidos (dispositivos móviles, consolas de video juegos, etc.); Open GL es el estándar más ampliamente usado alrededor del mundo para la creación de gráficos 2D y 3D, es gratis y multiplataforma. Como la programación en Open GL y en particular en Open GL ES es de muy bajo nivel y por lo tanto bastante complicada, se optó por investigar otras herramientas. Se descubrió entonces ISGL3D un *framework* escrito en Objective-C que trabaja sobre Open GL ES y que busca facilitar la tarea del programador al momento de crear y manipular escenas 3D mediante una “Application Program Interface” (API) sencilla e intuitiva. Es un proyecto gratis y en código abierto. Luego de algunas semanas de trabajo con la herramienta e importantes avances desde el punto de vista del manejo de la misma se descubrió la existencia de otro *framework* de idénticas características llamado “Cocos3D”. Cocos3D es una extensión de “Cocos2D”, una herramienta para la generación de gráficos 2D, muy popular entre los desarrolladores de aplicaciones para iOS. Como no se identificaron diferencias significativas entre ISGL3D y Cocos3D, se priorizó el tiempo dedicado a ISGL3D y se decidió continuar trabajando de forma inalterada. Al día de hoy, sobre el final del proyecto, se cree que si bien técnicamente ambos *frameworks* son muy buenos y a la vez similares entre sí, ISGL3D parece estar algo más avanzado en cuanto a su desarrollo. Sin embargo debido a la gran popularidad de Cocos2D, Cocos3D ha heredado muchos usuarios y cuenta con una comunidad mucho más activa, lo que facilita mucho su uso y hace pensar que en un futuro cercano resulte en un *framework* más desarrollado.

En este capítulo se comentarán algunas características y conceptos de ISGL3D que fueron importantes para el proyecto; por detalles de algunos temas en particular referirse a la referencia de la ya mencionada API en: www.isgl3d.com/resources/api. Además se trazará una hoja de ruta para todo aquel que quiera iniciarse en el manejo de la herramienta.

8.2. ISGL3D

8.2.1. Conceptos básicos de ISGL3D

ISGL3D es un motor de juegos 3D para *iPad*, *iPhone* y *iPod touch* escrito en *Objective-C*, que sirve para crear escenas y *renderizarlas* de forma sencilla. Es un proyecto en código abierto y gratis. En su sitio web oficial: www.isgl3d.com, se puede descargar su código, y de forma sencilla ISGL3D puede ser agregado como un complemento de *Xcode*. Además se pueden encontrar tutoriales, detalles de su API y un acceso a un grupo de *Google* donde la comunidad pregunta y responde dudas propias y ajena. Una buena manera de iniciarse en manejo de la herramienta es siguiendo los tutoriales en: www.isgl3d.com/resources/tutorials; al menos este fue el camino elegido por el grupo. Los tutoriales son 6, y abarcan distintos tópicos:

- **Tutorial 0:** primer paso en el creado de una aplicación ISGL3D. Cubre algunos conceptos básicos y muestra cómo integrar la herramienta a *Xcode*.
- **Tutorial 1:** muestra cómo crear una escena bien simple, con tan sólo un cubo en rotación continua.
- **Tutorial 2:** enseña cómo agregar luz a una escena. Se ven las distintas fuentes de luz que existen en el *framework*.
- **Tutorial 3:** se ve cómo hacer para mapear texturas en los objetos 3D con el objetivo de hacerlos más realistas.
- **Tutorial 4:** muestra cómo crear interacción entre el usuario y los distintos objetos ISGL3D, cuando este los toca a través de la pantalla.
- **Tutorial 5:** se ven algunas nuevas primitivas (modelos básicos) y se muestra cómo agregar transparencia a los objetos.

Al descargar e instalar ISGL3D, se puede ver que la herramienta incluye un proyecto *Xcode* integrado por varios ejemplos para ejecutar y a la vez ver su código, otra buena forma de aprender cómo realizar distintas tareas de interés. Entre los ejemplos se encuentra la solución a cada uno de los tutoriales.

Cuando se crea una aplicación ISGL3D, el núcleo de la misma es la llamada “view” (“vista” en Español). Una *view* está compuesta principalmente por una escena y una cámara:

- Una **escena** (*Isgl3dScene3D*) es donde los objetos o modelos 3D son agregados como nodos. Todos los nodos pueden ser tanto trasladados como rotados y pueden tener otros nodos hijos; los nodos hijos son trasladados y rotados con sus padres. Así como objetos 3D, se pueden agregar luces de distinto tipo, que generarán en la escena efectos de sombra que luego serán adecuadamente *renderizados* en función de dónde se encuentre y hacia dónde este mirando la cámara.
- Una **cámara** que es utilizada para ver la escena desde una posición y un ángulo en particular. La cámara se manipula como cualquier otro objeto o nodo en la escena, se puede trasladar, rotar y hasta indicar hacia dónde quiere uno que esta apunte. Es importante ajustar

la cámara de manera que su arquitectura sea la que uno busca. Se pueden entonces ajustar ciertos parámetros intrínsecos a esta como por ejemplo su campo visual, su distancia focal, la altura y la anchura del plano imagen, etc.

Es importante entender que el llamado *render* se realiza sumando la información de la escena, objetos 3D y sus hijos, luces, etc.; más la información de dónde se encuentra la cámara, sus características y hacia dónde esta apunta.

8.2.2. FOV y ejes de ISGL3D

Una particularidad de la cámara de ISGL3D es que el parámetro intrínseco “distancia focal” visto en la sección 5.2, no es directamente configurable. En cambio, el valor que sí se puede alterar es el llamado “FOV”, acrónimo de “Field Of View”. El *field of view* de una cámara no es más que su campo visual, y se mide como la extensión angular máxima mapeable en el plano imagen, medida desde el centro óptico C . Puede ser medido de forma horizontal o de forma vertical; sin embargo, en ISGL3D es definido verticalmente. Ver figura 1.1.

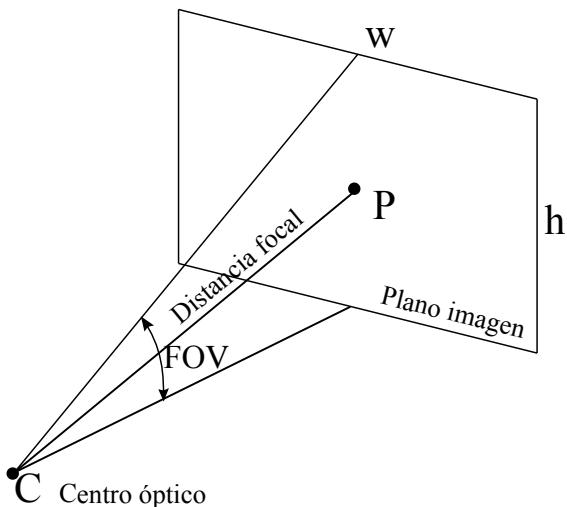


Figura 8.1: Definición gráfica del FOV.

Realizando algo de geometría se ve que la relación entre la distancia focal y el FOV es:

$$FOV = 2 \cdot \text{arctg} \left(\frac{h}{2 \cdot f} \right)$$

donde h denota la altura del plano imagen y f la distancia focal de la cámara.

Otra particularidad de ISGL3D es el sistema de coordenadas que difiere del que se usa habitualmente en el área de procesamiento de imágenes. Para ISGL3D los ejes están orientados como en la Figura 1.2 con el origen en el centro del plano de la pantalla del dispositivo que coincide con el plano $X-Y$ y el eje Z saliente de la misma.

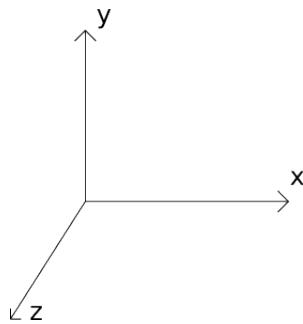


Figura 8.2: Sistema de coordenadas de ISGL3D.

8.2.3. Primitivas de ISGL3D

ISGL3D cuenta con algunas estructuras primitivas que pueden ser usadas como modelos, o incluso combinadas de manera de formar modelos algo más complejos. Las principales estructuras primitivas de ISGL3D son:

- **Isgl3DArrow:** modelo correspondiente a una flecha. Tiene 4 parámetros configurables:
 - *headHeight*: altura de la punta.
 - *headRadius*: radio de la punta.
 - *height*: altura total de la flecha.
 - *radius*: radio de la base.
- **Isgl3DCone:** modelo correspondiente a un cono. Tiene 3 parámetros configurables:
 - *bottomRadius*: radio de la base inferior.
 - *height*: altura del cono.
 - *topRadius*: radio de la base superior.
- **Isgl3DCube:** modelo correspondiente a un cubo. Tiene 3 parámetros configurables:
 - *depth*: profundidad del cubo.
 - *height*: altura del cubo.
 - *width*: anchura del cubo.
- **Isgl3DCylinder:** modelo correspondiente a un cilindro. Tiene 3 parámetros configurables:
 - *height*: altura del cilindro.
 - *radius*: radio del cilindro.
 - *openEnded*: indica si el cilindro cuenta con sus extremos abiertos o no.
- **Isgl3DEllipsoid:** modelo correspondiente a una elipsoide. Cuenta con 3 parámetros configurables:
 - *radiusX*: radio de la elipsoide en la dirección *x*.
 - *radiusY*: radio de la elipsoide en la dirección *y*.
 - *radiusZ*: radio de la elipsoide en la dirección *z*.

- **Isgl3DOvoid:** modelo ovoide. Cuenta con 3 parámetros configurables:
 - a : radio del ovoide en la dirección x .
 - b : radio del ovoide en la dirección y .
 - k : factor que modifica la forma de la curva. Cuando toma el valor 0, el modelo se corresponde con el de una ellipsoide.
- **Isgl3DSphere:** modelo correspondiente a una esfera. Tiene un único parámetro configurable:
 - $radius$: radio de la esfera.
- **Isgl3DTorus:** modelo correspondiente a un toroide. Cuenta con 2 parámetros configurables:
 - $radius$: radio desde el origen del toroide hasta el centro del tubo.
 - $tubeRadius$: radio del tubo del toroide.

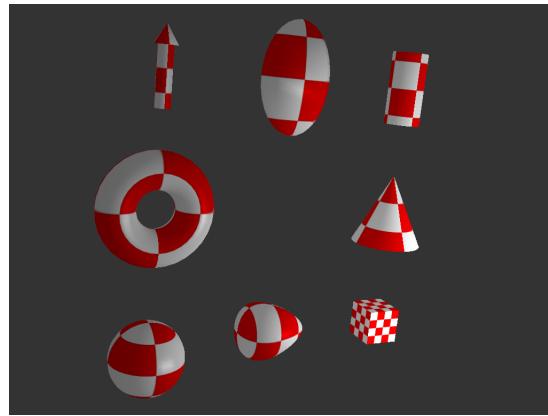


Figura 8.3: Principales primitivas en ISGL3D.

Para la creación de cada primitiva, se debe especificar además, la cantidad de segmentos que la forman en las distintas dimensiones. En la figura 1.3 se pueden ver todas las primitivas anteriores. Es fácil ver que dichas primitivas cuentan con cierta textura cuadriculada de colores rojo y blanco, que fue lograda mapeando una imagen sobre cada una de ellas. La porción de código que se usó para realizar tal mapeo se muestra a continuación:

```
Isgl3dTextureMaterial * material = [Isgl3dTextureMaterial
    materialWithTextureFile:@"red_checker.png" shininess:0.9];

Isgl3dTorus * torusMesh = [Isgl3dTorus meshWithGeometry:2 tubeRadius:1 ns:32 nt:32];

Isgl3dMeshNode * _torus = [self.scene createNodeWithMesh:torusMesh andMaterial:material];
```

En la primera línea de código se crea el material. Dicho material es del tipo *Isgl3dTextureMaterial*; y la imagen con la que este se crea es la de la figura 1.4. Luego, se crea el toroide asignándole los parámetros vistos más atrás en esta sección; y finalmente, se crea y se agrega a la escena el nodo asociado al toroide, con el material creado al principio.

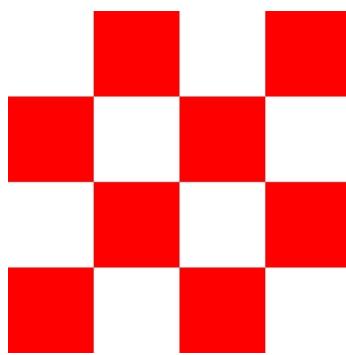


Figura 8.4: Imagen *red_checker.png*, utilizada para crear la textura asociada a las primitivas de la figura 1.3.

8.2.4. Importación de modelos a ISGL3D.

A veces lo que se quiere no es agregar a la escena una primitiva sino un modelo previamente creado. Los modelos son realizados en herramientas de creado y animación de gráficos 3D como por ejemplo *Blender*, *MeshLab*, *Autodesk Maya* o *Autodesk 3ds Max*. Luego deben ser exportados en un formato llamado *COLLADA*, acrónimo de “COLLABorative Design Activity”, que sirve para el intercambio de contenido digital 3D entre distintas aplicaciones de modelado. Por su parte, ISGL3D permite importar modelos pero en un formato llamado “POD”. Se usó entonces, una aplicación llamada *Collada2POD* que lo que hace es convertir modelos tridimensionales en formato *COLLADA* al formato POD. *Collada2POD* puede ser descargada gratuitamente de la página oficial de *Imagination Technologies*, su desarrollador: <http://www.imgtec.com>.

Una vez que se tiene al objeto 3D en el formato requerido, este puede ser importado en ISGL3D de forma sencilla:

```
Isgl3dPODImporter * podImporter = [Isgl3dPODImporter podImporterWithFile:@“modelo.pod”];  
Isgl3dNode * _model = [self.scene createNode];  
  
[podImporter addMeshesToScene:_model];  
  
_model.position = iv3(2, 6, 0);
```

En la primera línea de código se instancia la clase *Isgl3dPODImporter* que sirve para transformar modelos POD a objetos ISGL3D, y se le asigna a la misma el modelo “modelo.pod”. Luego, se crea un nodo llamado “_model”, al que se le asigna el modelo; y se agrega a la escena. Finalmente, se le asigna al nodo una posición. En la figura 1.5 se puede ver un modelo de José Artigas, agregado dos veces a una misma escena, pero visto desde ángulos distintos.

Si lo que se quiere es que los modelos sean animados, o lo que es lo mismo, que tengan movimiento, hay dos soluciones posibles a tomar en consideración:

- **Modelo animado:** muchas veces lo que se tiene es un modelo 3D animado desde su construcción. Estos pueden ser creados, al igual que los modelos 3D inanimados, con las herramientas para el creado y la animación de gráficos 3D antedichas. Existe mucha bibliografía al respecto, además de haber múltiples sitios en internet de donde bajar los modelos, incluso en forma gratuita. Luego de obtenido el modelo animado, lo que se tiene es precisamente al modelo, pero con una línea de tiempo con las animaciones. Nuevamente, hay que convertirlo a formato POD para ser usado en ISGL3D. El código para poder visualizar al modelo es:

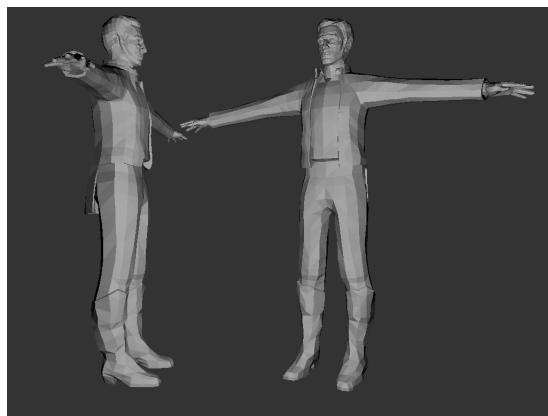


Figura 8.5: Modelo de José Artigas agregado dos veces en una misma escena, pero visto desde ángulos distintos.

```
Isgl3dPODImporter * podImporter = [Isgl3dPODImporter
    podImporterWithFile:@"animated_model.pod"];

Isgl3dSkeletonNode *_model = [self.scene createSkeletonNode];

[podImporter addMeshesToScene:_model];

Isgl3dAnimationController *_animationController = [[Isgl3dAnimationController alloc]
    initWithSkeleton:_model andNumberOfFrames:[podImporter numberOfFrames]];

[_animationController start];
```

En la primera línea de código se instancia la clase *Isgl3dPODImporter*, y se le asigna a la misma el modelo animado “animated_model.pod”. Luego, se crea y se agrega a la escena un nodo del tipo *Isgl3dSkeletonNode* llamado “_model” que contiene al modelo animado. La clase *Isgl3dSkeletonNode* provee una interfaz sencilla para animar al ahora objeto ISGL3D, que con la ayuda de la clase *Isgl3dAnimationController*, logra automatizar el movimiento del mismo. Finalmente, se instancia y configura la clase *Isgl3dAnimationController* y se le da inicio a la animación en la última línea.

- **Múltiples modelos inanimados:** otra forma de animar un modelo 3D es usando múltiples modelos inanimados. Estos pueden ser cargados en ISGL3D como un único objeto o nodo y mediante ciertas instrucciones sencillas, se le dice al *framework* que presente uno a continuación del otro, interpolando entre posiciones contiguas, lo que genera una sensación de movimiento. Este fue el método utilizado en el presente proyecto para animar al modelo de José Artigas. En la figura 1.6 se puede ver al mismo en 3 posiciones distintas.

El código para realizar la interpolación mencionada, aplicado por ejemplo a dos modelos, será:

```
Isgl3dPODImporter * podImporter = [Isgl3dPODImporter
    podImporterWithFile:@"model_1.pod"];

[podImporter buildSceneObjects];

Isgl3dPODImporter * podImporter2 = [Isgl3dPODImporter
```



Figura 8.6: Modelo de José Artigas en 3 posiciones distintas, utilizadas para generar en ISGL3D una sensación de movimiento.

```

podImporterWithFile:@"model_2.pod"];

[podImporter2 buildSceneObjects];

Isgl3dGLMesh* _modelMesh = [podImporter meshAtIndex:0 ];

Isgl3dGLMesh* _modelMesh2 = [podImporter2 meshAtIndex:0 ];

Isgl3dKeyframeMesh * _mesh = [Isgl3dKeyframeMesh keyframeMeshWithMesh:_modelMesh];

[_mesh addKeyframeMesh:_modelMesh2];

[_mesh addKeyframeAnimationData:0 duration:1.0f];
[_mesh addKeyframeAnimationData:0 duration:2.0f];
[_mesh addKeyframeAnimationData:1 duration:1.0f];
[_mesh addKeyframeAnimationData:1 duration:2.0f];

[_mesh startAnimation];

Isgl3dNode * node = [_container createNodeWithMesh:_mesh
                    andMaterial:[podImporter materialWithName:@"material_0"]];

node.position = iv3(-90, -60, -150);

[podImporter addMeshesToScene:node];

```

En las primeras 4 líneas de código se instancia en dos oportunidades la clase *Isgl3dPODImporter*, y se les asigna a las instancias los modelos inanimados “model_1.pod” y “model_2.pod”. La instrucción *buildSceneObjects* crea todos los objetos de la escena del modelo POD, pero no los agrega a la escena ISGL3D. Luego se obtienen los modelos indexados de cada uno de los PODs (cada POD puede tener una escena con más de un modelo). Mediante un índice se referencia qué modelo se quiere obtener) y se almacenan en “_modelMesh” y “_modelMesh2” respectivamente. Se genera a continuación un nuevo modelo al que se le asignan los dos modelos anteriores, luego se programa la animación y se le da inicio mediante la instrucción *startAnimation*. Finalmente, se genera un nuevo objeto o nodo ISGL3D al que se le asigna el modelo, y un material también cargado desde el archivo POD; se le asigna además una posición y se lo agrega a la escena.

8.2.5. Características de Luz en ISGL3D

Un tema importante al momento de proyectar modelos 3D en una escena es la luz. La visualización de un modelo puede cambiar significativamente en función de las características de luz que tenga una escena. Básicamente existen tres tipos de luz: *ambiente*, *difusa* y *especular*. La luz ambiente es no direccional y está presente en toda la escena. La luz difusa ya implica la reacción que tiene la luz proveniente de las distintas fuentes de luz direccionales sobre las superficies de los objetos que existen en la escena generando haces de luz en direcciones aleatorias. La luz especular modela el comportamiento de la luz reflejada sobre las distintas superficies en ciertas direcciones particulares (no aleatorias) que dependen del coeficiente de reflexión de los materiales. Cada uno de estos tres tipos de luz que modelan el mundo real, existen en ISGL3D y son representados como características configurables de los objetos de luz. A su vez existen fuentes lumínicas de distintos tipos: *puntual*, *direccional* y *cónica*. Cada tipo tiene una función distinta de la atenuación con respecto a la distancia. Un ejemplo de la creación de un elemento lumínico para una escena se ve en el siguiente código:

```
_redLight = [ISGL3DLight lightWithHexColor:@“FF0000” diffuseColor:@“FF0000” specularColor:@“FFFFFF” attenuation:0.02];
_redLight.renderLight = YES;
[self.scene addChild:_redLight];
```

donde se crea una luz de color rojo (notación hexadecimal de las componentes RGBA), del tipo especular y con la atenuación dada.

8.2.6. Método - (*void*) *tick:(float)dt*

Cuando se instancia la clase encargada de generar el render (clase *Helloworld*), la misma ejecuta la configuración básica de inicialización del objeto. Entre otras cosas, dentro del código de inicialización, se agrega el siguiente código:

```
[self schedule:@selector(tick:)];
```

Esto lo que hace es “agendarse” una invocación del método *tick* en forma periódica. Dentro de dicho método es que se hace la actualización periódica de reproyección que se obtiene como resultado de la estimación de pose. Este método tiene principal importancia por tratarse de uno de los dos *callbacks* que toda aplicación de realidad aumentada tiene (el otro naturalmente es la captura y procesamiento de la imagen para obtener la pose).

8.2.7. ISGL3D en la aplicación

En el capítulo ?? se explican algunos detalles sobre el uso de ISGL3D dentro de la aplicación final. En particular se dan detalles constructivos sobre cómo utilizar esta herramienta para generar *renders* sobre un fondo que sea la captura de la cámara y de la convivencia de los dos *callbacks* de la aplicación.

CAPÍTULO 9

Casos de Uso

9.1. Introducción

En este capítulo se describen los distintos casos de uso que se implementaron con el fin de aplicar los algoritmos desarrollados en los capítulos anteriores. Se buscó generar distintos casos de uso que funcionaran como muestra de las funcionalidades que son posibles de realizar mediante la resolución de los algoritmos mencionados.

9.2. Caso de Uso 01

9.2.1. Comentarios sobre el caso de uso

9.2.2. Detalles constructivos

9.3. Caso de Uso 02

9.3.1. Comentarios sobre el caso de uso

Este caso de uso básicamente busca desplegar un video en una superficie dada del mundo real. Esto puede ser de gran interés como complemento de contenido para un cuadro o cualquier obra si se piensa en aplicarlo para museos. Es posible por ejemplo, generar un video que sea reproducido dentro de los marcos del propio cuadro, en un extremo o en una superficie cualquiera que resulte interesante desde el punto de vista artístico. A continuación se explican brevemente algunos detalles para lograr la implementación de este caso de uso.

PONER FOTO MOSTRANDO EL CASO DE USO CON VIDEO PROYECTADO.

9.3.2. Detalles constructivos

Para lograr lo propuesto para este caso de uso se implementó un proyecto que proyecta el video en uno de los cuadrados del marcador. De esta manera, de toda la lógica de estimación de pose, solamente se hace uso de la detección y filtrado. En particular no se hace uso de los resultados del posit. Teniendo entonces detectados los cuatro puntos en los que se quiere reproducir el video parecería que el problema está resuelto. Sin embargo, xcode no permite posicionar en forma directa una vista de video en cualquier conjunto de cuatro puntos.

Si simplemente se quiere reproducir un video, y no se quiere procesar el contenido, lo más cómodo para hacerlo es utilizar la clase *MPMoviePlayerController* que hereda de *NSObject*. Una alternativa similar es haciendo uso de la clase *MPMoviePlayerViewController* que hereda de *UIViewController* y tiene como única propiedad una del tipo *MPMoviePlayerController*.

MPMoviePlayerController tiene un atributo *view* del tipo *UIView* que es la vista y es este atributo el que se quiere posicionar en los cuatro puntos detectados por el filtro. Un atributo del tipo *UIView* tiene un atributo *frame* que es del tipo *CGRect*

```
theMovie.view.frame = CGRectMake(0, 0, 60, 60);
```

En el código anterior *theMovie* es del tipo *MPMoviePlayerController*. De esta manera, se tiene el inconveniente de que en principio cualquier video parecería que solamente puede ser reproducido sobre rectángulos y no en cualquier polígono de cuatro puntos por ejemplo. Sin embargo algo que sí se puede hacer a las instancias de la clase *UIView* es una transformación afín o incluso, de manera más genérica, una homografía.

9.3.3. *CGAffineTransform* y *CATransform3D*

La clase *UIView* tiene una propiedad llamada *transform* que es del tipo *CGAffineTransform*. Las primeras letras de esta clase (CG) refieren a la API **Core Graphics** utilizada ampliamente como herramienta para resolver *rendering* y cualquier tipo de transformación en 2D.

La clase *UIView* también tiene una propiedad llamada *layer* que es del tipo *CALayer* y que permite realizar transformaciones del tipo *CATransform3D*. Las primeras letras de estas dos clases (CA) refieren a la API **Core Animation** que es utilizada para generar animaciones y transformaciones sobre objetos 3D solamente indicando un punto inicial y final para el objeto (también es posible agregar efectos para la transición). En definitiva para resolver el problema del caso de uso existen a priori dos alternativas posibles: *CGAffineTransform* y *CATransform3D*.

Se pueden generar fácilmente instancias transformaciones afines invocando la siguiente función:

```
CGAffineTransform CGAffineTransformMake (
    CGFloat a,
    CGFloat b,
    CGFloat c,
    CGFloat d,
    CGFloat tx,
    CGFloat ty
);
```

que toma 6 *CGFloats* y crea una *CGAffineTransform*, donde cada uno de los valores anteriores se corresponde con los elementos de una matriz transformación afín de la siguiente manera:

$$\begin{pmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{pmatrix}$$

Así entonces, de los 9 valores de la matriz, 2 de ellos son nulos por tratarse de una transformación afín y otro de ellos es unitario como valor de escala. Resolviendo el sistema como se muestra en la sección 9.3.4 y obteniendo los restantes 6 valores, se le puede asignar transformaciones a la propiedad *transform* y realizar la trasnformación deseada. Este método tuvo como inconveniente el hecho de que

EXPLICAR POR QUE NO FUNCIONÓ

Por su parte también es sencillo generar instancias de transformaciones 3D debido a que existe el tipo de dato definido para generar la matriz *CATransform3D* como:

```
struct CATransform3D
{
    CGFloat m11, m12, m13, m14;
    CGFloat m21, m22, m23, m24;
    CGFloat m31, m32, m33, m34;
    CGFloat m41, m42, m43, m44;
};

typedef struct CATransform3D CATransform3D;
```

donde m_{ij} corresponde al elemento de la matriz ubicado en la fila i columna j . Así entonces también es posible, conociendo los valores de la homografía, completar los elementos de esta matriz 4x4 y asignársela a la propiedad *layer* de la *UIView*. Esta opción de generar una transformación 3D permite incluir transformaciones más generales que una homografía o una transformación afín. Si lo que se busca es que esta matriz represente una homografía (2D), es necesario entonces que la coordenada z sea nula, es decir

$$\begin{pmatrix} m_{11} & m_{12} & 0 & m_{14} \\ m_{21} & m_{22} & 0 & m_{24} \\ 0 & 0 & 1 & 0 \\ m_{41} & m_{42} & 0 & m_{44} \end{pmatrix}$$

donde a su vez m_{44} se asume de valor unitario por ser un factor de escala. De la misma manera que para la transformación afín, resolviendo la homografía como se ve en la sección 9.3.4 se obtienen los 8 valores restantes de la matriz y es posible asignarle una homografía a un objeto *UIView* para resolver el problema presente.

9.3.4. Resolución de Homografía

A continuación se hace el desarrollo de la resolución del sistema de ecuaciones que se tuvo que resolver para hallar los parámetros de la homografía que transforma una imagen de referencia en la imagen que se tiene en cada momento como resultado de la captura de la cámara. Se asume entonces que se conocen los puntos de referencia y los puntos de referencia transformados (los detectados luego del filtrado de segmentos) y lo que se quiere averiguar es la matriz h que logra dicha transformación. Esta homografía 2D-2D se puede expresar en forma matricial, en coordenadas homogéneas de la siguiente manera:

$$\begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} i \\ j \\ k \end{pmatrix}$$

donde la matriz h_{3x3} representa la transformación homográfica, el vector $(x, y, z)^t$ representa los puntos de referencia a ser transformados y el vector $(i, j, k)^t$ respresenta los puntos detectados cuadro a cuadro como las esquinas del marcador.

Asumiendo un valor unitario para las coordenadas z y k la resolución del sistema se simplifica mucho y no se pierde generalidad. Imponiendo esto entonces, el sistema anterior se puede expresar de la siguiente forma:

$$xh_{11} + yh_{12} + h_{13} = i \quad (9.1)$$

$$xh_{21} + yh_{22} + h_{23} = j \quad (9.2)$$

$$xh_{31} + yh_{32} + h_{33} = 1 \quad (9.3)$$

Multiplicando la ecuación (9.3) por i e igualándola a la ecuación (9.1) se obtiene lo siguiente:

$$xh_{11} + yh_{12} + h_{13} = ixh_{31} + iyh_{32} + ih_{33} \quad (9.4)$$

o lo que es lo mismo:

$$xh_{11} + yh_{12} + h_{13} - ixh_{31} - iyh_{32} - ih_{33} = 0 \quad (9.5)$$

Procediendo de manera análoga y multiplicando la ecuación (9.3) por j e igualándola a la ecuación (9.2) se obtiene lo siguiente:

$$xh_{21} + yh_{22} + h_{23} = jxh_{31} + jyh_{32} + jh_{33} \quad (9.6)$$

o lo que es lo mismo:

$$xh_{21} + yh_{22} + h_{23} - jxh_{31} - jyh_{32} - jh_{33} = 0 \quad (9.7)$$

Las ecuaciones (9.3) y (9.7) se pueden expresar en forma matricial, de la siguiente manera:

$$\begin{pmatrix} x & y & 1 & 0 & 0 & 0 & -ix & -iy & -i \\ 0 & 0 & 0 & x & y & 1 & -jx & -jy & -j \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Teniendo entonces 4 parejas de puntos referencia y puntos transformados y asumiendo h_{33} de valor unitario se tiene entonces 8 ecuaciones y 8 incógnitas, lo que lo vuelve un sistema compatible determinado que se puede expresar de la siguiente manera:

$$\begin{pmatrix} x_0 & y_0 & 1 & 0 & 0 & 0 & -i_0x_0 & -i_0y_0 \\ 0 & 0 & 0 & x_0 & y_0 & 1 & -j_0x_0 & -j_0y_0 \\ x_1 & y_1 & 1 & 0 & 0 & 0 & -i_1x_1 & -i_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -j_1x_1 & -j_1y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -i_2x_2 & -i_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -j_2x_2 & -j_2y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -i_3x_3 & -i_3y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -j_3x_3 & -j_3y_3 \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{pmatrix} = \begin{pmatrix} i_0 \\ j_0 \\ i_1 \\ j_1 \\ i_2 \\ j_2 \\ i_3 \\ j_3 \end{pmatrix}$$

Así entonces, lo que se hace para resolver la homografía es cuadro a cuadro tener detectados los puntos en los que se quiere presentar la vista del video que se corresponden con cuatro puntos detectados por el filtro y tener las correspondencias con el marcador real, se posiciona la vista en la posición de referencia y se le aplica la homografía hallada que vincula la posición referencia con los puntos detectados.

9.4. Caso de Uso 03

9.4.1. Comentarios sobre el caso de uso

9.4.2. Detalles constructivos

9.5. Caso de Uso 04

9.5.1. Comentarios sobre el caso de uso

9.5.2. Detalles constructivos

CAPÍTULO 10

Implementación

10.1. 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.

10.2. 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

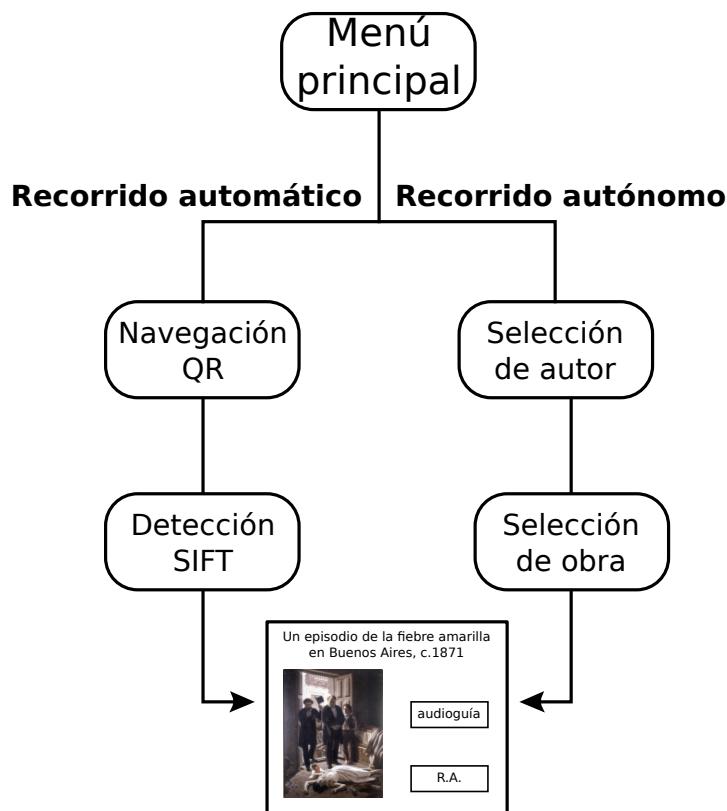


Figura 10.1: Diagrama global de la aplicación

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 referenciarlos 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

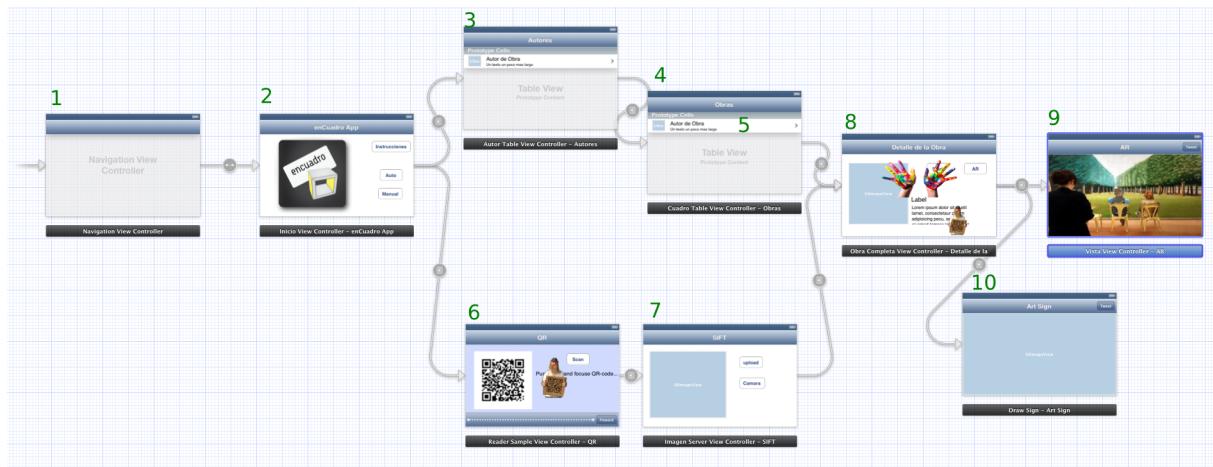


Figura 10.2: *Storyboard* de la aplicación

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

10.2.1. NavigationViewController

Esta clase se ve en la Figura 1.2, identificada con el número 1. La aplicación está embebida dentro de un *UINavigationController*. 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: *Portrait*, *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 *UINavigationController* 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
```

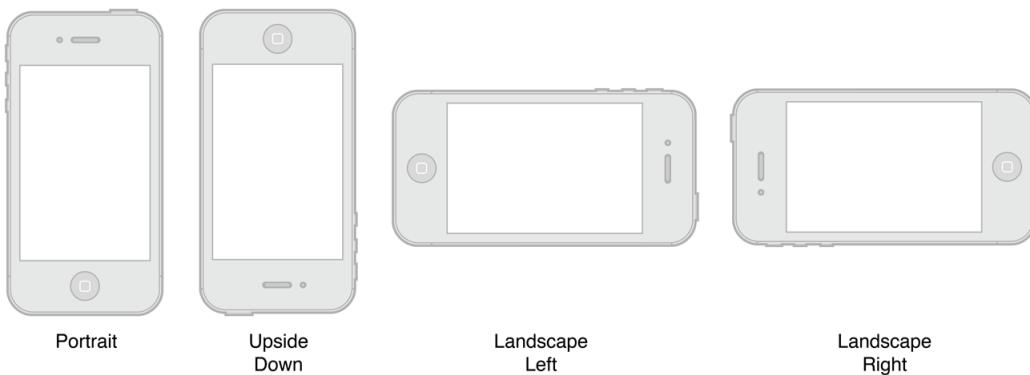


Figura 10.3: Orientaciones posibles del dispositivo.

```
{
    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 ítem *SupportedInterfaceOrientations* y completando 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.

10.2.2. 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*.

10.2.3. 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

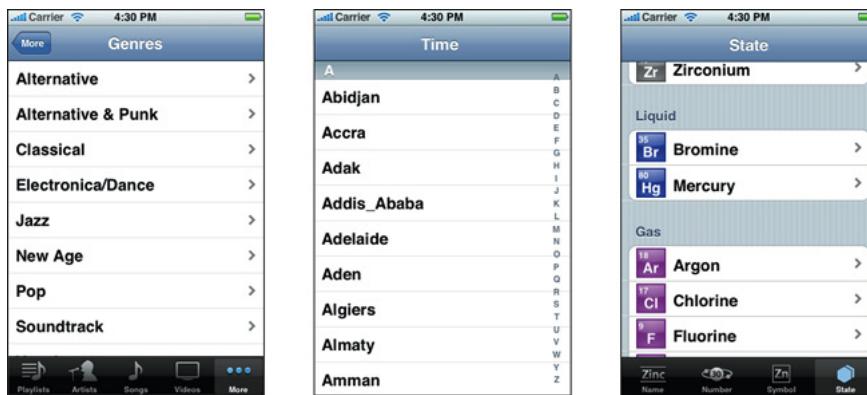


Figura 10.4: Ejemplos de TableViewControllers con distintos tipos de tablas.

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.

10.2.3.1. 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.

10.2.3.2. CuadroTableViewController

Esta clase (identificada con el número 4 en la Figura 1.2) es muy similar a la clase *AutorTableViewController* recién descrita 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.

10.2.3.3. 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.

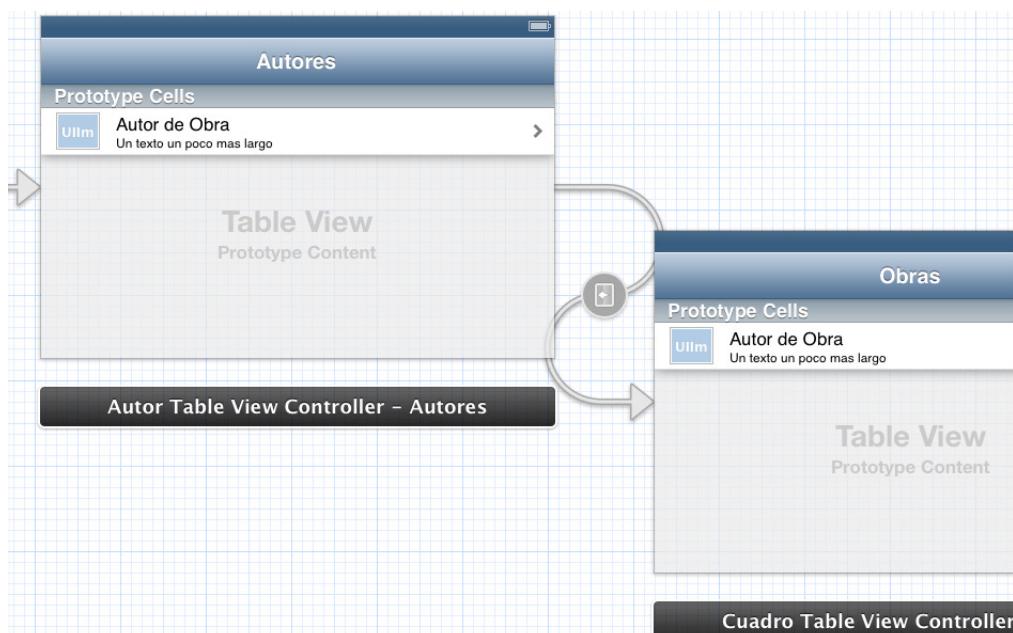


Figura 10.5: Autor y Cuadro TableViewControllers

10.2.4. 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.

10.2.5. 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 10.6: 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.

10.2.6. 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.

10.2.7. 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];
}
```

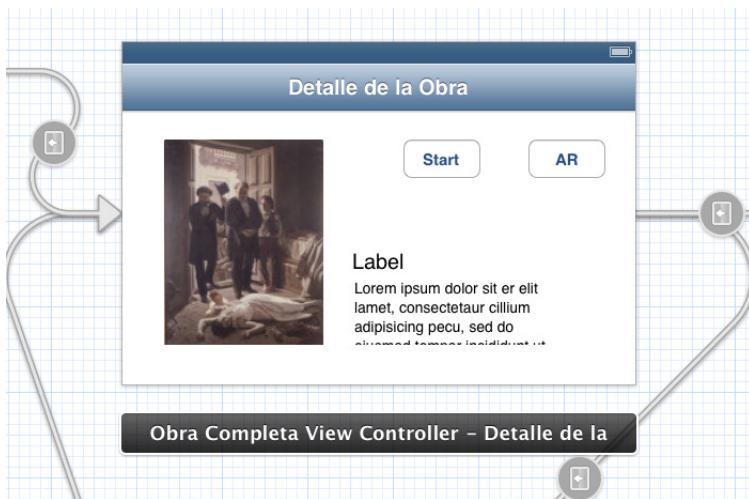


Figura 10.7: Pantalla con la obra completa

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*.

10.2.8. 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 10.8: 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 levantarla 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*.

10.2.9. 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 tamaño 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*.

10.2.10. 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 error:&error];

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

self.frameOutput.videoSettings = [NSDictionary dictionaryWithObject:[NSNumber numberWithInt:kCVPixelFormatType_32BGRA] forKey:kAVVideoOutputColorSpace];

/*Ahora conectamos todos los objetos*/
/*Primero le agregamos a la sesión 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 *multi-threading*. 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:(CMSSampleB
```

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.

10.3. QR

10.3.1. 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.

10.3.2. 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.

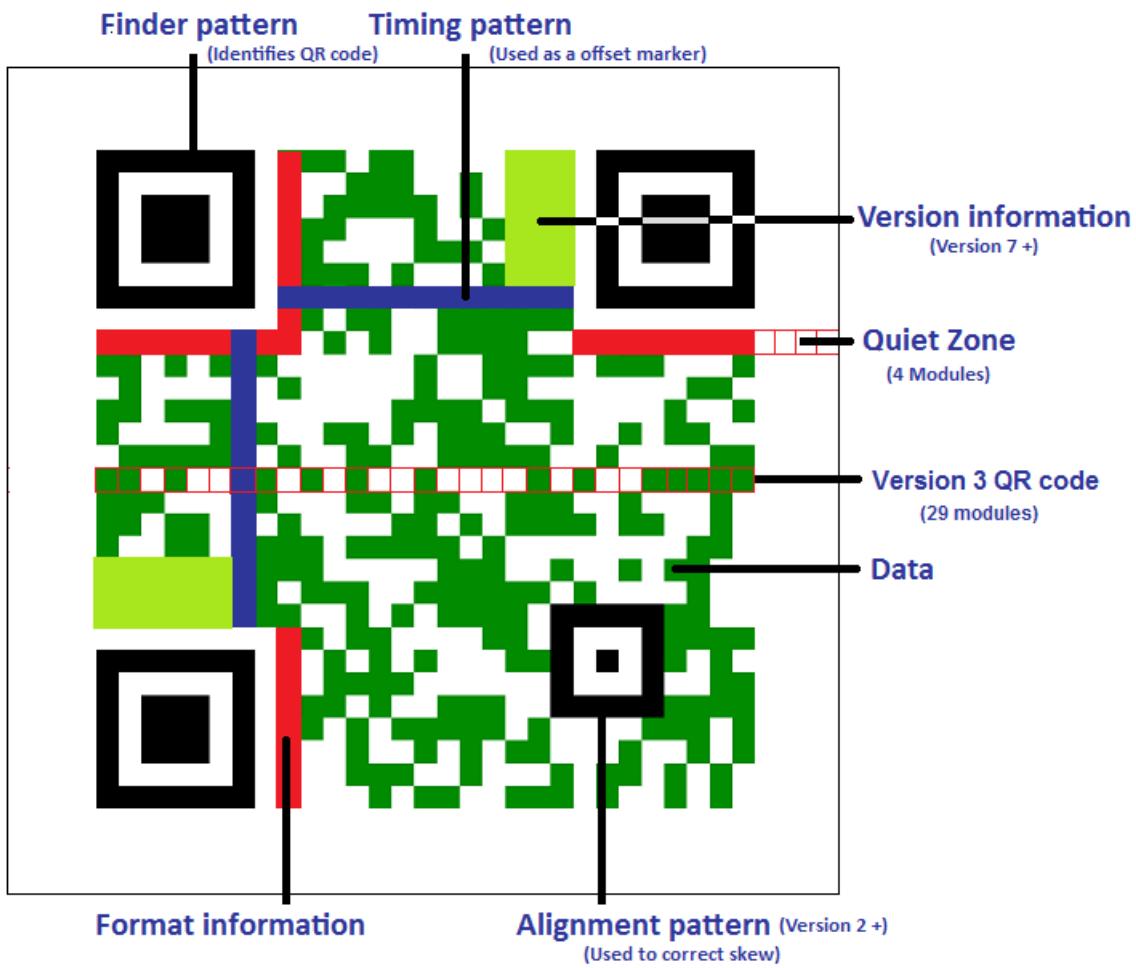


Figura 10.9: Las distintas componentes de un QR. Fuente [7].

10.3.3. 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++, Objective 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.

10.3.4. 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.

10.3.5. 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 modificando 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.

10.4. 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 servidor. 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 quizás 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.



Figura 10.10: Ejemplo de un QR artista. Fuente [7].

10.4.1. 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.

10.4.1.1. Servidor iOS

redactar pasos...

10.4.1.2. 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 intérprete php instalado. A partir de este momento todo se reduce a comprender bien el lenguaje php y poder



Figura 10.11: Impresión de pantalla al ingresar a <http://localhost>.

realizar pequeños módulos de programación que puedan tomar entradas, procesarlas y arrojar una salida.

10.4.2. 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 action.php

MOU SIGUE ESTO...

10.5. SIFT

10.6. Incorporación de la realidad aumentada a la aplicación

asdfasdfasdfsdf

[8].

Bibliografía

- [1] J. García Ocón. Autocalibración y sincronización de múltiples cámaras plz. 2007.
- [2] B. Furht. *The Handbook of Augmented Reality*. 2011.
- [3] C. Avellone and G. Capdehourat. Posicionamiento indoor con señales wifi. 2010.
- [4] Philip David, Daniel Dementhon, Ramani Duraiswami, and Hanan Samet. Simultaneous pose and correspondence determination using line features. pages 424–431, 2003.
- [5] Philip David, Daniel Dementhon, Ramani Duraiswami, and Hanan Samet. Softposit: Simultaneous pose and correspondence determination. pages 424–431, 2002.
- [6] Daniel F. DeMenthon and Larry S. Davis. Model-based object pose in 25 lines of code. *International Journal of Computer Vision*, 15:123–141, 1995.
- [7] R. Grompone von Gioi, J. Jakubowicz, J. M. Morel, and G. Randall. Lsd: A fast line segment detector with a false detection control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(4):722–732, April 2010.
- [8] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.
- [9] V. Lepetit and P. Fua. Monocular model-based 3d tracking of rigid objects: A survey. *Foundations and Trends in Computer Graphics and Vision*, 1(1):1–89, 2005.