Índice general

CAPÍTULO 1

Hardware y Software

1.1. Introducción

Introducción

1.2. Software de procesamiento de imágenes

Software de procesamiento de imágenes

1.3. Dispositivos móviles

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 tama?o de la pantalla entre otras puedan verse afectadas. A continuación se relata el surgimiento de cada uno de lo 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 a?os. 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 desbloquar 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 a?os. 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.
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. Sistemas operativos y entorno de desarrollo

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 a?o 1984). En los últimos 28 a?os 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* en todas sus versiones. La versión más reciente de este SO es el *iOS 6.1*.

Entorno de desarrollo

1.4.1. Objective-C y xCode

1.4.2. Simulador

1.4.3. Instruments

1.4.3.1. Time Profiler

1.4.3.2. Memory Leak

1.4.4. Librerías

1.4.4.1. AvFoundation

1.4.4.2. MediaPlayer

1.4.4.3. CoreMotion

1.4.4.4. Tweeter y mensajería

1.5. Herramientas

Herramientas

CAPÍTULO 2

Detección

Hola

2.1. Típos de características

Intro y mas breves definiciones?.
Bordes, esquinas, líneas, segmentos de línea, regiones (blobs).

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

2.4. Regiones y puntos de interés

2.4.1. FAST

2.4.2. Blobs

2.5. Detección sin primitivas markerless

SIFT (puntero a capitulo que tiene SIFT para reconocimiento) SURF, ETC ETC.

2.6. Marcadores

Marcadores que se usan. Limitaciones

La inclusión de *marcadores*, *marcas de referencia* o *fiduciales*, en inglés *markers*, *landmarks* o *fiducials*, en la escena ayuda al problema de extracción de características y por lo tanto al problema de estimación de pose [10]. 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 facilmente utilizables para la estimación de la pose.

Se distinguen dos tipos de *fiduciales*. El primer tipo son los que se llaman puntos *fiduciales* por que proveen una correspondencia de puntos entre la escena y la imagen. El segundo tipo, *fiduciales planares*, se pueden obtener mediante la construcción en una geometría coplanar de una serie de *puntos fiduciales* identificables como esquinas. Un único *fiducial planar* puede contener por si solo todas las seis restricciones espaciales necesarias para definir el marco de coordenadas.

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.

2.7. Marcador QR

El enfoque inicial elegido para la detección de *puntos fiduciales* para marcadores parte del trabajo de fin de curso de Matías Tailanian para el curso *Tratamiento de imágenes por computadora* de Facultad de Ingeniería, Universidad de la República[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[8] por su buena *performance* y aparente bajo costo computacional.

El marcador utilizado está basado en la estructura de detección incluida en los códigos *QR* y se muestra en la figura 1.1. 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 colo 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.

2.7.1. 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.

Los elementos mas 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.

[1] Autoposicionamiento 3D - http://sites.google.com/site/autoposicionamiento3d/

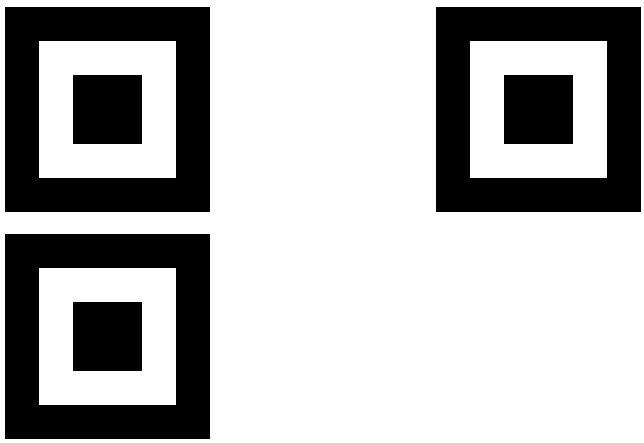


Figura 2.1: Marcador propuesto basado en la estructura de detección de códigos QR.

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 $(Ql[0], Ql[1], \ldots, Ql[M-1])$ en donde el orden viene dado por el valor de *perímetro* de cada Ql. Se define el *centro del grupo de cuadriláteros* 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 $(QlSet[0], QlSet[1], \ldots, QlSet[N-1])$ 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 fiduciales para proveer las correspondencias $\mathbf{M}_i \leftrightarrow \mathbf{m}_i$ al algoritmo de estimación de pose.

2.7.2. 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 1.1.

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 1.2. Adicional-

mente 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 ambiente del *tratamiento 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,

$$v_{j_0} = (a/2, a/2) \qquad v_{j_2} = (-a/2, -a/2)$$
$$v_{j_1} = (a/2, -a/2) \qquad v_{j_3} = (-a/2, a/2)$$

con $a = (j+1) \times l$. El orden aquí explicado se puede ver también junto con el sistema de coordenadas en la figura 1.3.

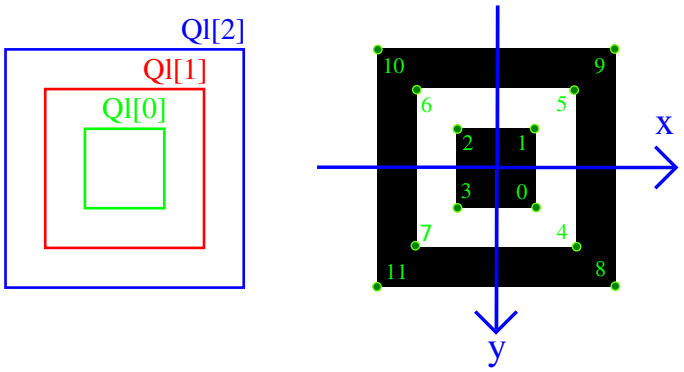


Figura 2.2: Detalle de un QlSet. A la izquierda se muestra el resultado de la detección de un *QlSet* y el orden interno de sus *cuadriláteros* y a la derecha el orden de los vértices respecto al sistema de coordenadas local.

Un detalle del marcador completo se muestra en la figura 1.3 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,

$$\mathbf{x} = \frac{\mathbf{c}_1 - \mathbf{c}_0}{||\mathbf{c}_1 - \mathbf{c}_0||} \quad \mathbf{y} = \frac{\mathbf{c}_2 - \mathbf{c}_0}{||\mathbf{c}_2 - \mathbf{c}_0||} \tag{2.1}$$

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.: 1.7.3.3).

2.7.2.1. 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*.

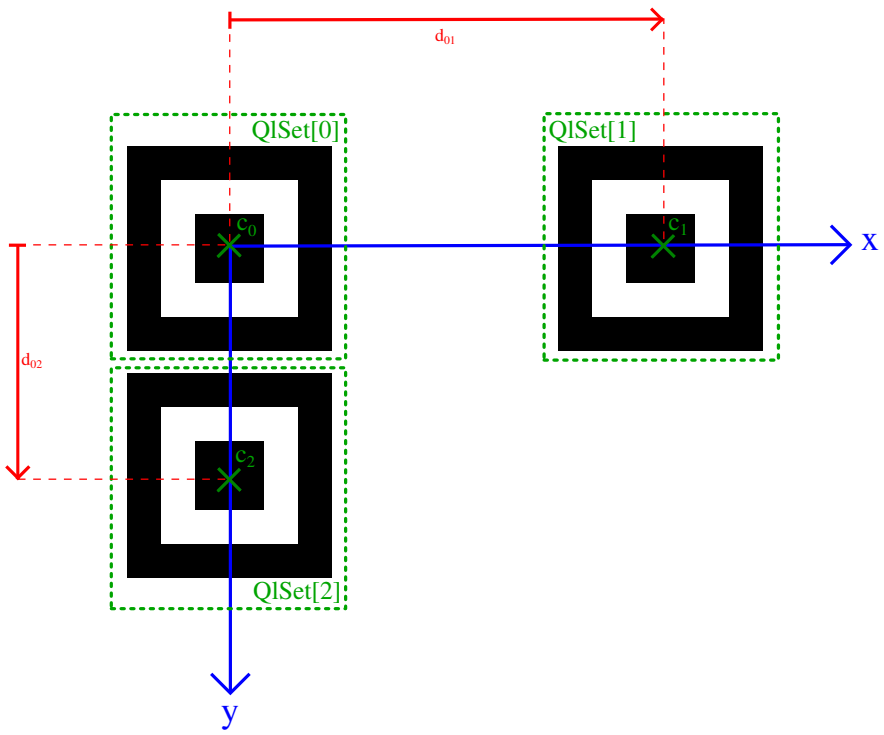


Figura 2.3: Detalle del marcador propuesto formando un sistema de coordenadas.

- 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 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 1.3.

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.

2.7.2.2. Diseños utilizados

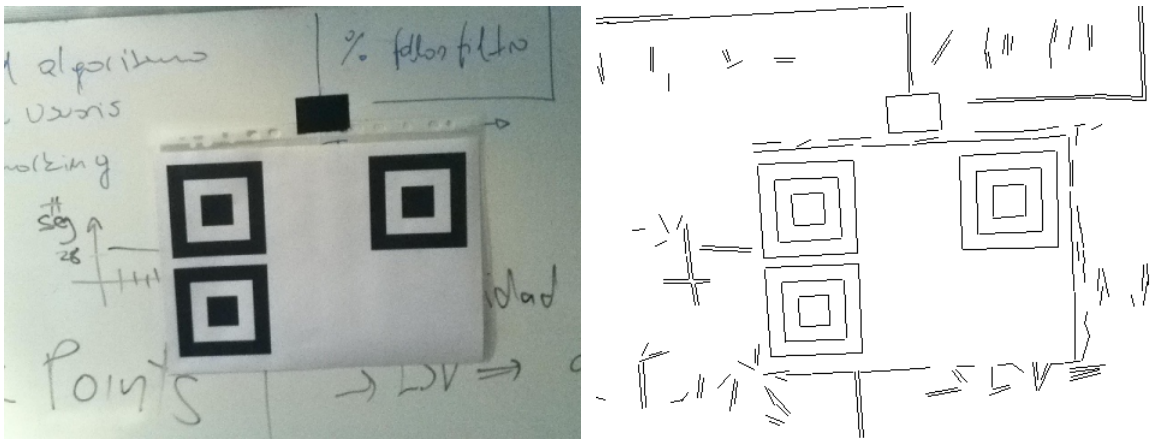
- **Test**: Durante el desarrollo de los algoritmos de detección e identificación de los *vértices* del *marcador QR* se trabajó con determinados parámetros de diseño de dimensiones apropiadas para posibilitar el traslado y las pruebas domésticas.
- **Da Vinci**
- **Artigas**
- **Mapa**

2.7.3. 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.

2.7.3.1. 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.



(a) Entrada: imagen conteniendo al marcador (b) Salida: segmentos de línea detectados por LSD

Figura 2.4: Resultados del algoritmo de detección de segmentos de línea LSD.

2.7.3.2. 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,

$$\mathbf{L} = \begin{pmatrix} \mathbf{s}_0 & \mathbf{s}_1 & \dots & \mathbf{s}_{m-1} \end{pmatrix}^t \tag{2.2}$$

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 1.4. La tercer 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 euclidiana 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 euclidiana d_{ij} 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 tercer 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 $\sim_k$.

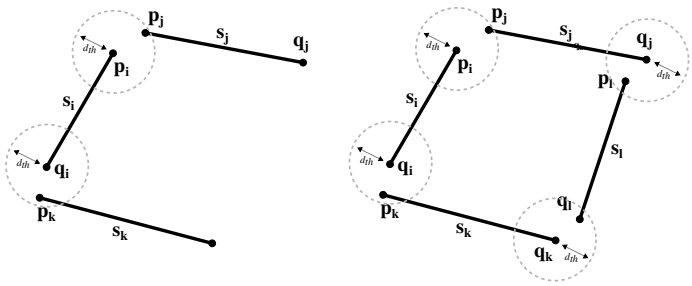
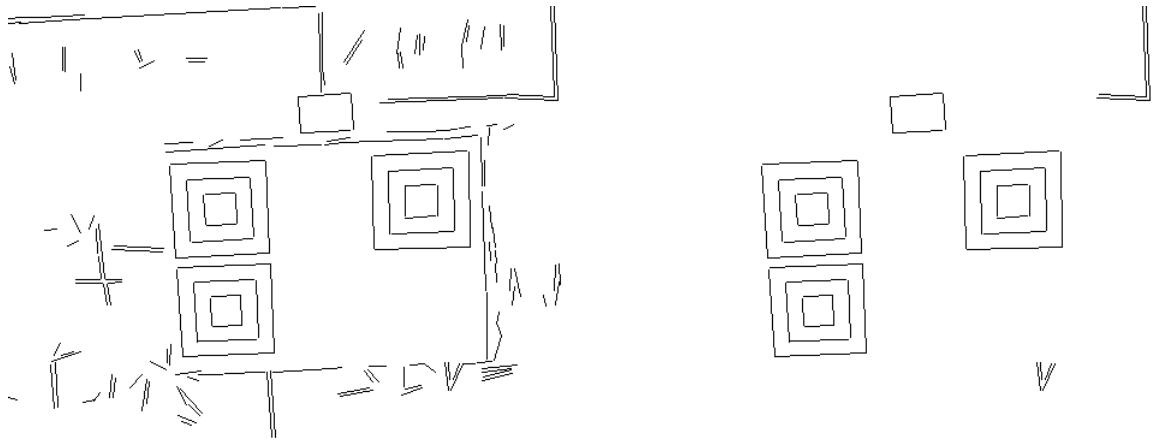


Figura 2.5: Conjunto de cuadriláteros conexos. A la izquierda la primera y segunda etapa del filtrado completadas para el segmento $\sim_i$ en donde se busca una "U". A la derecha la última etapa en donde se cierra la "U" con el segmento $\sim_l$.

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 **??** se muestran lo 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 sostenía el marcador (ver figura **??**(a)). También sobreviven otro tipo de elementos indeseados que se explican a continuación.

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



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

Figura 2.6: Resultados del algoritmo de filtrado y agrupamiento de segmentos de línea.

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 1.5 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 un condición de vecindad que tome en cuenta el punto de intersección entre los segmentos y la distancia de este punto a los puntos **p**, **q** mas cercanos de cada segmento. Como se explicará en le 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.

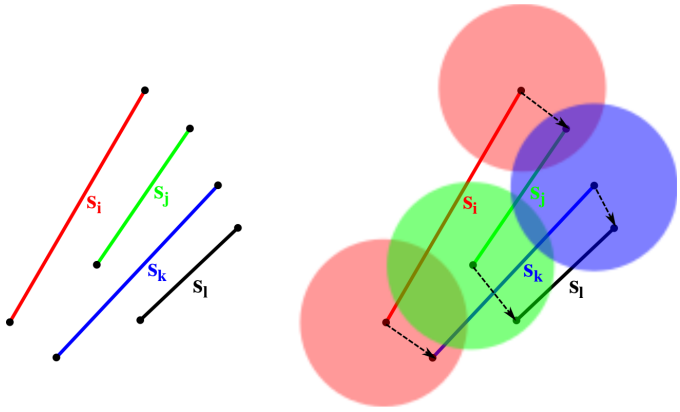


Figura 2.7: Posible configuración de segmentos paralelos que "sobreviven" al filtrado. A la izquiera el grupo de segmentos, a la derecha se muestra como se desarrolla el filtrado de $\mathbf{s}_i$.

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.

2.7.3.3. 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.: 1.7.1), de forma de desarrollar un algoritmo modular, escalable y simple.

Se toma como entrada la lista de segmentos filtrados y agrupados

$$\mathbf{S} = \begin{pmatrix} \mathbf{s}_0 & \mathbf{s}_1 & \dots & \mathbf{s}_i & \mathbf{s}_{i+1} & \mathbf{s}_{i+2} & \mathbf{s}_{i+3} & \dots & \mathbf{s}_{n-1} \end{pmatrix}^t \tag{2.3}$$

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 es múltiplo de cuatro. Si i también lo es, entonces el sub-conjunto, $\mathbf{S}_i = \begin{pmatrix} \mathbf{s}_i & \mathbf{s}_{i+1} & \mathbf{s}_{i+2} & \mathbf{s}_{i+3} \end{pmatrix}^t$, 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, $\mathbf{V}_i = \begin{pmatrix} \mathbf{v}_i & \mathbf{v}_{i+1} & \mathbf{v}_{i+2} & \mathbf{v}_{i+3} \end{pmatrix}^t$. Si $\mathbf{r}_i$ es la recta que pasa por los puntos $\mathbf{p}_i$ y $\mathbf{q}_i$ del segmento $\mathbf{s}_i$, 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 1.6 en donde una de ellas tiene sentido horario y la otra antihorario partiendo de v_i.

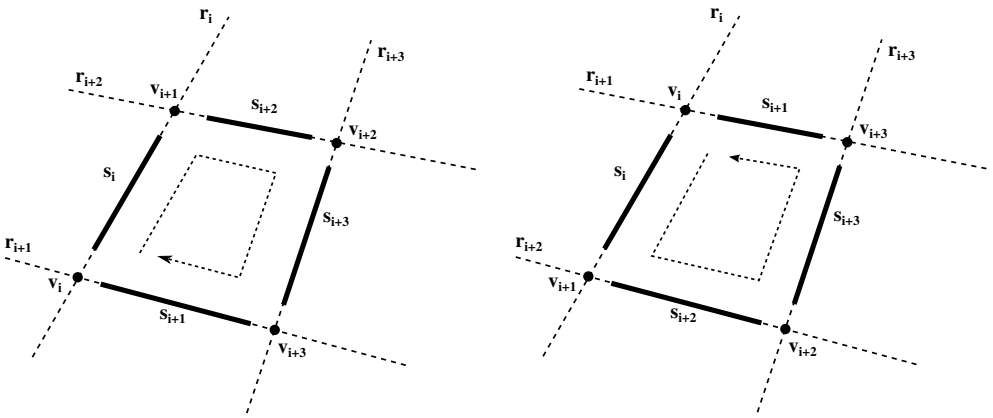


Figura 2.8: Posibles configuraciones de vértices posterior a la intersección de conjuntos de segmentos pertenecientes a un cuadrilátero.

Posterior a la intersección se realiza un chequeo sobre el valor de las coordenadas de lo 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 filtado 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

$$QlList = \begin{pmatrix} Ql[0] & Ql[1] & \dots & Ql[i] & \dots & Ql[\frac{n}{4}] \end{pmatrix}^t$$

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 d_{th},

$$d_{ij} = ||\mathbf{c}_i - \mathbf{c}_j|| < d_{th}, \quad d_{ik} = ||\mathbf{c}_i - \mathbf{c}_k|| < d_{th}. \tag{2.4}$$

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

$$QlSet[l] = \begin{pmatrix} Ql[0] & Ql[1] & Ql[2] \end{pmatrix}$$

con $l = (0,1,2)$. Esta busqueda 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*,

$$QlSetList = \begin{pmatrix} QlSet[0] & QlSet[1] & QlSet[2] \end{pmatrix}$$

ésta se ordena de forma que su dispoción 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 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 1.7.

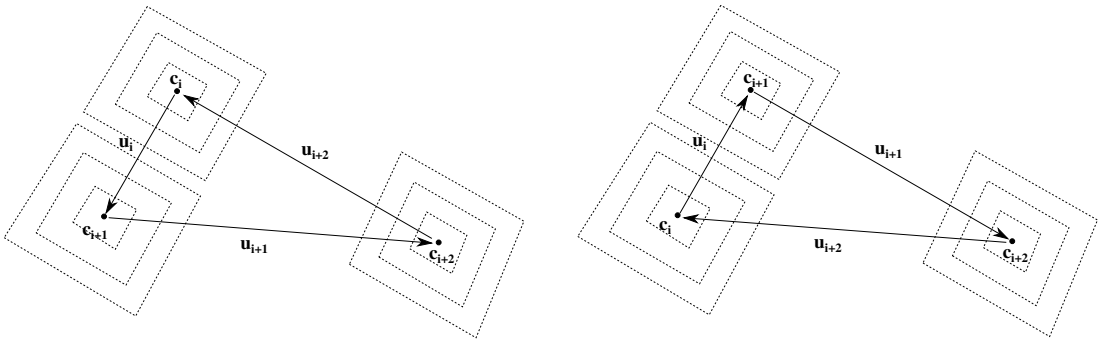


Figura 2.9: Vértices de cada *Ql* ordenados respecto al signo de sus proyecciones contra el sistema de coordenadas local a cada *QlSet*.

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,

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

o de lo contratio se ordena como,

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

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 1.1. Los ejes de este sistema de coordenadas permiten, para cada *Ql* de cada *QlSet*, proyectar los vértices sobre sus ejes y según su signo ordenarlos como se muestra en la figura 1.8. De esta forma, recorriendo ordenadamente los elementos del marcador, se ordenan

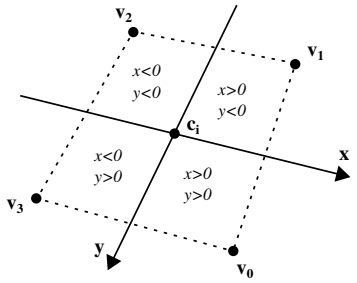


Figura 2.10: Posibles configuraciones de centros resultan en la orientación de los vectores $\mathbf{u}_{i+k}$.

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,

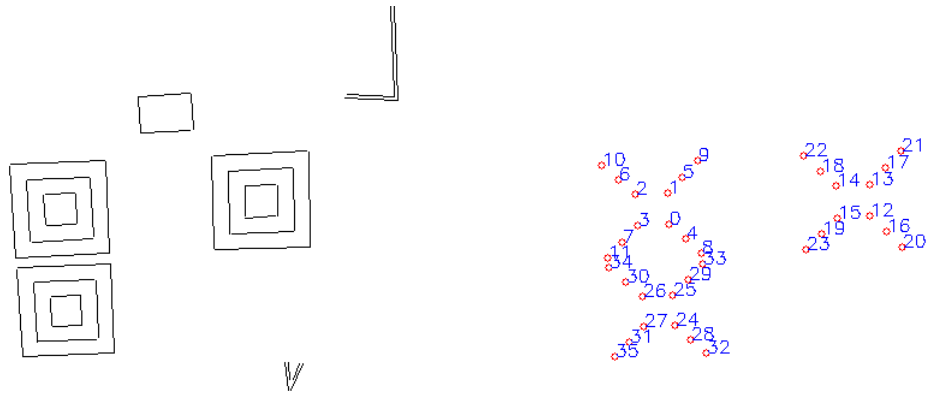
---

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 **??**. 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.

2.7.3.4. Robustificando la detección

El algorítmo 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



(a) Entrada: segmentos de línea filtrados y agrupados (b) Salida: puntos vértices ordenados.

Figura 2.11: Resultados del algoritmo de determinación de correspondencias.

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 ordenandolos 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.

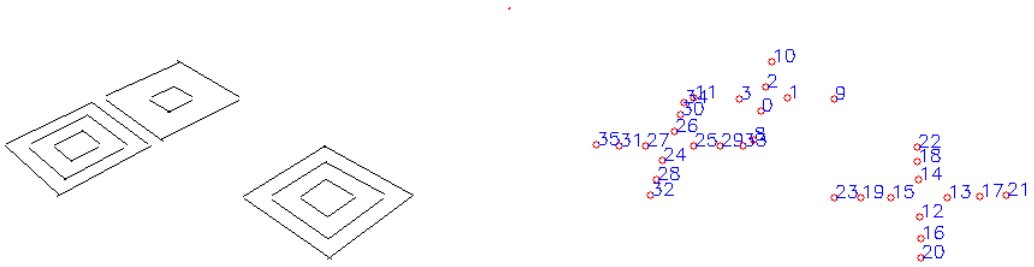
Al realizar la busqueda de conjuntos de cuadriláteros concéntricos se buscan en primer lugar los *QlSet*'s completos y luego en caso de que estos no lleguen a ser tres, se intenta completar buscando *QlSet*'s incompletos o sea conjuntos de dos cuadriláteros que comparten un mismo centro. Estos se agrupan 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 **??**(a) se muestran imagen en la que falla el filtrado de segmentos para uno de los cuadriláteros mientras que en la figura **??**(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 2.12: Resultados del algoritmo de determinación de correspondencias robustificado para una falla en el filtrado de segmentos.

2.7.3.5. Resultados

Oh sí!

CAPÍTULO 3

LSD: "Line Segment Detection"

3.1. 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 del 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.

3.2. *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.

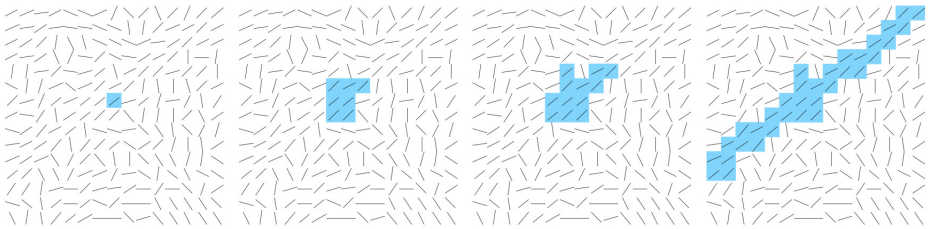


Figura 3.1: Proceso de crecimiento de una región. El ángulo asociado cada píxel de la imagen está representado por los pequeños segmentos y los píxeles coloreados representan la formación de la región. Fuente [8].

Los píxels 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 3.2. Este potencial problema es heredado del algoritmo de Burns, Hanson y Riseman.

Figura 3.2: Potencial problema heredado del algoritmo de Burns, Hanson y Riseman. Izq.: Imagen original. Ctro.: Segmento detectado. Der.: Segmentos que deberían haberse detectado. Fuente [8].

Sin embargo, LSD plantea un método para ahorraste 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.

3.3. 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 cubir el 99 % de la masa de la región.

Figura 3.3: 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 [8].

3.4. 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 distribuídos 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 esta 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:

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

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 menera, 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 imgen analizada de la sigiuente manera:

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

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

3.5. 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 disminuír su valor NFA y así entonces validarlos. Esta 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.

3.6. 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 algorimos 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, simpre 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.

3.6.1. 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, disminuír el tamaño de la imagen de entrada con el objetivo de disminuír 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 imgen se hace en dos pasos, primero a lo ancho y luego a lo largo. Se utiliza el núceo Gaussiano normalizado de la figura 3.4.

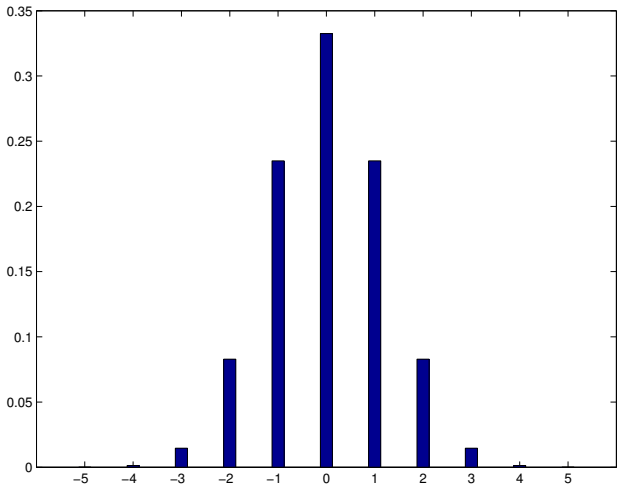


Figura 3.4: Núcleo Gaussiano utilizado por LSD. $\sigma = 1,2$.

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 $\frac{x}{escala}$ en la imagen original y sus vecinos, todos ponderados por el núcleo Gaussiano centrado en $\frac{x}{escala}$. 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 $\frac{y}{escala}$ en la imagen auxiliar y sus vecinos, todos ponderados por el núcleo Gaussiano centrado en $\frac{y}{escala}$. En la figura 3.5 se muestra la relación entre las imágenes.

Véase que cuando en el submuestreo $\frac{1}{escala}$ 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 le agrega precisión al algoritmo, también le agrega un gran costo computacional, ya que lo que se

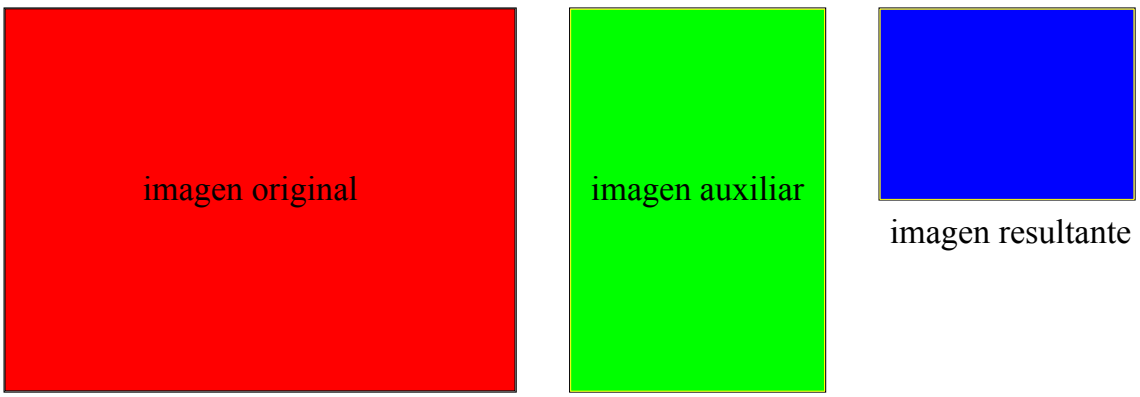


Figura 3.5: Relación entre las imágenes en consideradas en el filtro Gaussiano. Escala: $0,5$.

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 utlizados 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 3.4); 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 $\frac{1}{escala} = n$, 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 requeire 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 3.6.

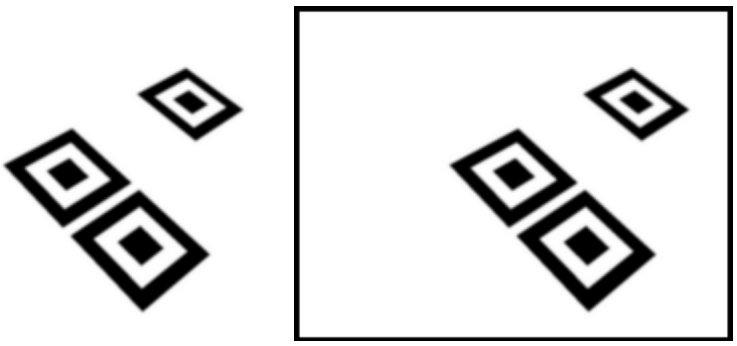


Figura 3.6: Imagen artificial del marcador trasladado y rotado, filtrada con el filtro Gaussiano. Izq.: Filtro Original. Der.: Filtro sin las condiciones de borde.

3.6.2. *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 grandiente 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 3.6), que de no ser así se detectarían.

3.6.3. 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 concentricos, 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 desempeõ 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 inferioes.

3.6.4. 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.

3.6.5. Resultados

3.6.5.1. Filtro Gaussiano

Figura 3.7: 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 imagen de prueba fue la de la figura 3.7; sépase que por cómo es el algoritmo, el contenido de la imagen en independiente del tiempo de procesamiento en cualquiera de los casos. Los valores relevantes del experimento se muestran en las tablas 3.1 y 3.2:

- **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 3.1: Comparación entre los tiempos de ejecución del filtro Gaussiano optimizado y el original. Ambos comprecisión 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 3.2: Comparación entre los tiempos de ejecución del filtro Gaussiano optimizado y el original. Ambos comprecisión simple.

3.6.5.2. *Line Segment Detection*

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

Figura 3.8: Imagen *zebras.png*.

ya que el algoritmo original está implementado con éstos integrados. Las imágenes de prueba fueron la del marcador sintético (figura 3.7) y *zebras.png* mostrada en la figura 3.8. Los valores relevantes de los experimentos se muestran en las tablas 3.3, 3.4, 3.5 y 3.6.

- **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 3.3: Comparación entre los tiempos de ejecución del filtro Gaussiano más LSD optimizados y los originales, para la imagen 3.7. En todos los casos con comprecisión doble.

	Algoritmo original	Algoritmo optimizado
Imagen urilizada	*zebras.png*	*zebras.png*
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 3.4: Comparación entre los tiempos de ejecución del filtro Gaussiano más LSD optimizados y los originales, para la imagen 3.8. 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 3.5: Comparación entre los tiempos de ejecución del filtro Gaussiano más LSD optimizados y los originales, para la imagen 3.7. En todos los casos con comprecisión simple.

	Algoritmo original	Algoritmo optimizado
Imagen urilizada	*zebras.png*	*zebras.png*
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 3.6: Comparación entre los tiempos de ejecución del filtro Gaussiano más LSD optimizados y los originales, para la imagen 3.8. En todos los casos con comprecisión simple.

CAPÍTULO 4

Modelo de cámara y estimación de pose monocular

4.1. Introducción

Se le llama "estimación de pose" al proceso mediente 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 comunmente 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.**

4.2. Modelo de cámara *pin-hole* [1]

4.2.1. 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 1.1.

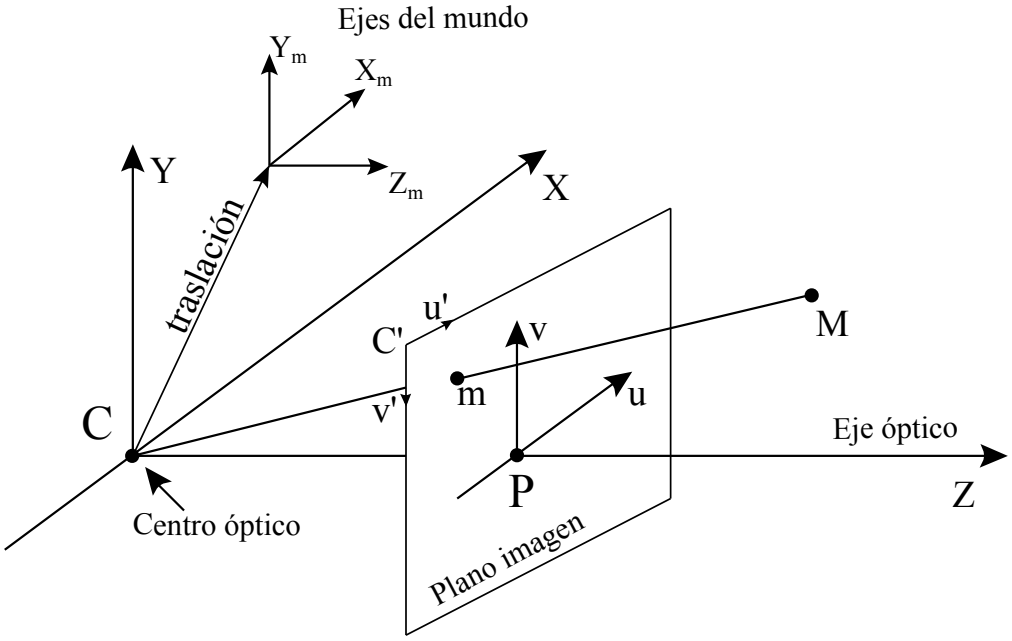


Figura 4.1: Modelo de cámara pin-hole.

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:

$$m = H.M,$$

entonces:

$$H = I.E$$

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 (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 (**C**) y el punto principa (**P**). Su unidad es el milímetro.
  - Factor de proporción (s): indica la proporción entre las dimensiones horizontal y vertical de un píxel.

4.2.2. 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 se sus parámetros extrínsecos:

$$m = H.M$$

donde **M** y **m** son los puntos ya definidos y vienen expresados en "coordenadas homogéneas". Por más información acerca de este tipo de coordenadas ver [9].
Para determinar la forma de la matriz de proyección se estudia cómo se relacionan las coordenadas de **M** con las coordenads de **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 **M** expresadas en el sistema de coordenadas de la cámara (X_0, Y_0, Z_0, T_0), a las coordenadas homogéneas del punto **m** expresadas en el sistema de coordenadas de la imagen (u_0, v_0, s_0):

Se desprende de la imagen 1.1 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):

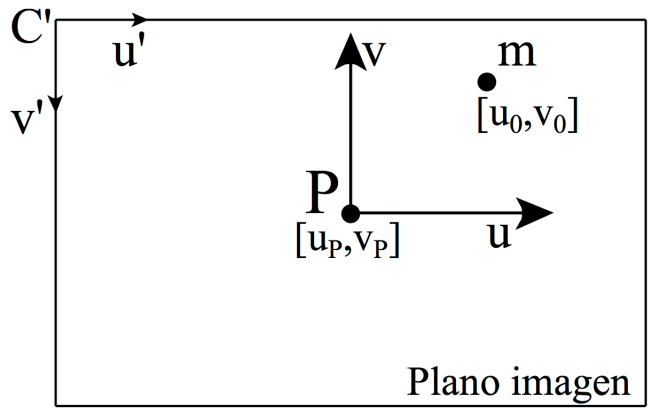


Figura 4.2: Relación entre el sistema de coordenadas de la imagen y el sistema de coordenandas normalizadas de la imagen.

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 imgen (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:

$$u'_0 = d_u.u_0 + u'_P$$
$$v'_0 = d_v.v_0 + v'_P$$

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 **M** expresadas en el sistema de coordenadas de la cámara (X_0, Y_0, Z_0, T_0), a las coordenadas homogéneas del punto **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:

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

  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:

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

- **Matriz de parámetros extrínsecos** (E)**:** de las coordenadas homogéneas del punto **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:

$$\begin{pmatrix} X_0 \\ Y_0 \\ Z_0 \\ T_0 \end{pmatrix} = \begin{pmatrix} R & T \\ 0 & 1 \end{pmatrix} \begin{pmatrix} X_{m0} \\ Y_{m0} \\ Z_{m0} \\ T_{m0} \end{pmatrix}$$

  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 **M** expresadas respecto del sistema de coordenadas del mundo $(X_{m0}, Y_{m0}, Z_{m0}, T_{m0})$, a las coordenadas homogéneas del punto **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:

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

4.3. 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 **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:

$$\begin{pmatrix} \tilde{u_0} \\ \tilde{v_0} \end{pmatrix} = L(r).\begin{pmatrix} u_0 \\ v_0 \end{pmatrix},$$

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:

$$\tilde{u_0}' = u'_P + L(r)(u'_0 - u'_P)$$
$$\tilde{v_0}' = v'_P + L(r)(v'_0 - v'_P)$$

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: $L(r) = 1 + k_1 r + k_2 r^2 + k_3 r^3 + ...$ Finalmente, a la hora de calcular los parámetros intrínsecos de una cámara, también deben ser estimados sus coeficientes de distorsión radial $\{k_1, k_2, k_3, k_4, ...\}$.

4.4. 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 [14] son el método de Zhang [11], el método de R.Y. Tsai [15] y un método llamado "Direct Linear Transform" (DLT) [16] . Para calibrar las cámaras utilizadas en este proyecto, se se trabajó con una implementación en *Matlab* basada en el método de Zhang ([13]), 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

Figura 4.3: Imagen de un damero, utilizada para calibrar la cámara del *iPad* durante el proyecto.

desde orientaciones distintas. En la figura 1.3 se ve una de las imágenes utilizadas para 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 **M** expresado respecto de los ejes de coordenadas del mundo y su proyección en el plano imagen **m**, expresada respecto de los ejes normalizados de la imagen, viene dada por:

$$m = I.E.M$$

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

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:

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

donde $(X_{m0}, Y_{m0}, Z_{m0}, 1)^T$ denota las coordenadas homogéneas del punto **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, **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 1.3), 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 $H = (h_1, h_2, h_3) = I.(r_1, r_2, t)$, 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:

$$h_1^T.(I^{-1})^T.I^{-1}.h_2 = 0$$
$$h_1^T.(I^{-1})^T.I^{-1}.h_1 = h_2^T.(I^{-1})^T.I^{-1}.h_2$$

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 libartad y existen 6 parámetros extrínsecos (3 para la traslación y 3 para la rotación).

Si se define la matriz B como sigue:

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

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

$$b = (B11, B_{12}, B_{22}, B_{13}, B_{23}, B_{33})^T$$

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

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

se tiene que:

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

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

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

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 recontruír 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 homografíau $H_i = I.E_i$. 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 1.3. 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 5

POSIT: *POS* with *IT*erations

5.1. 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* (*P*ose from *O*rtography and *S*caling), esta técnica consiste en aproximar la pose de la cámara a partir de la proyección *SOP*(*S*caled *O*rtographic *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.

5.2. POSIT clásico

La primera versión de POSIT presentada por *Daniel DeMenthon* y *Larry Davis* en [6] 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.

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

En la figura 5.1 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 esta sobre el eje óptico de la cámara y apunta en sentido saliente. Los versores para estos ejes son **i**, **j** y **k**.

Se considera ahora un objeto con puntos característicos M_0, M_1, ..., M_i, ..., M_n, cuyo eje de coordenadas esta 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 lo 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 **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 **R** queda:

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

Para obtener la matriz de rotación solo es necesario obtener los versores **i** y **j** , el versor **k** se obtiene de realizar el producto vectorial **i** $\times$ **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 **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 **i**, **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.

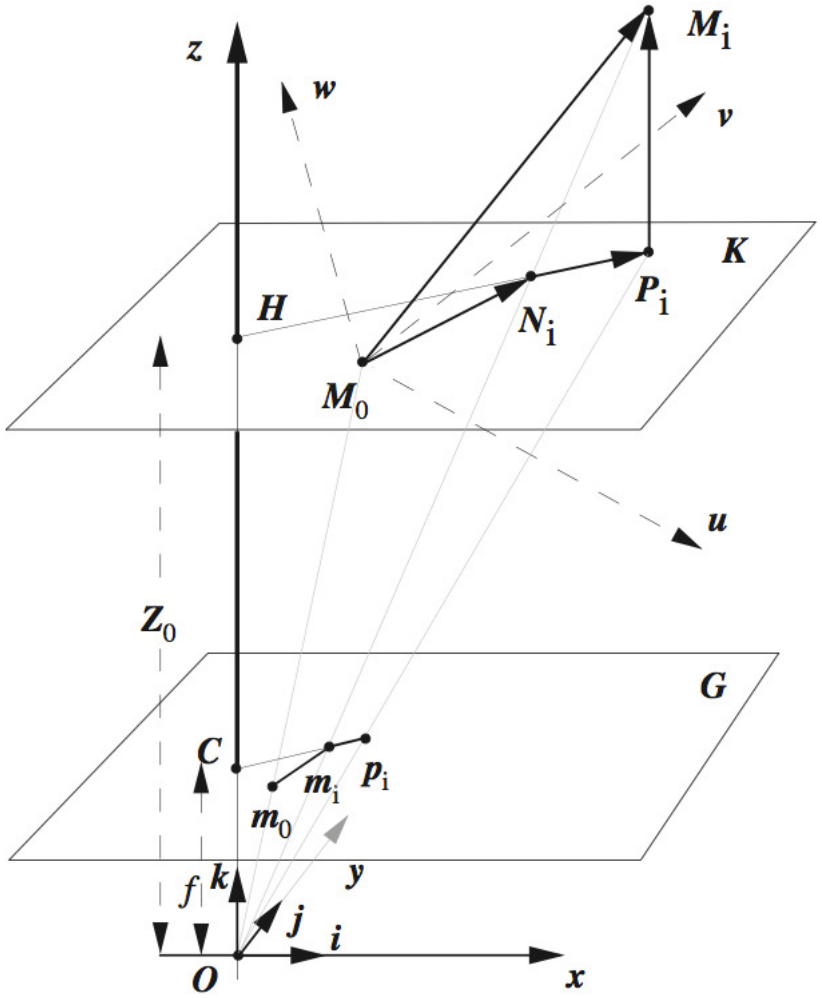


Figura 5.1: Proyección en perspectiva (m_i) y SOP (p_i) para un punto del modelo 3D M_i y un punto de referencia del modelo M_O. Fuente: [6] .

5.2.2. SOP: *S*caled *O*rtographic *P*rojection

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:

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

mientras que la proyección SOP esta dada por:

$$x_i^{'} = fX_i/Z_0, \qquad y_i^{'} = fY_i/Z_0.$$

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 5.1 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_0p_i es s veces el tamaño de M_0P_i. Teniendo esto en cuenta se pueden expresar las coordenadas de p_i como:

$$\begin{aligned} x_i^{'} &= fX_0/Z_0 + f(X_i - X_0)/Z_0 = x_0 + s(X_i - X_0) \\ y_i^{'} &= y_0 + s(Y_i - Y_0) \end{aligned} \tag{5.1}$$

5.2.3. Ecuaciones para calcular la proyección perspectiva

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

$$M_0M_i\frac{f}{Z_0}\mathbf{i} = x_i(1+\varepsilon_i) - x_0 \tag{5.2}$$

$$M_0M_i\frac{f}{Z_0}\mathbf{j} = y_i(1+\varepsilon_i) - y_0 \tag{5.3}$$

donde ε_i se define como

$$\varepsilon_i = \frac{1}{Z_0}M_0M_i\mathbf{k} \tag{5.4}$$

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 7.7, el producto escalar da la coordenada z de M_0M_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.

5.2.4. Algoritmo

Las ecuaciones 7.2 y 7.3 se puede reescribir como:

$$M_0 M_i \mathbf{I} = x_i(1+\varepsilon_i) - x_0 \tag{5.5}$$

$$M_0 M_i \mathbf{J} = y_i(1+\varepsilon_i) - y_0 \tag{5.6}$$

en donde

$$\mathbf{I} = \frac{f}{Z_0}\mathbf{i} = s \cdot \mathbf{i}, \qquad \mathbf{J} = \frac{f}{Z_0}\mathbf{j} = s \cdot \mathbf{j} \tag{5.7}$$

Si se conociera el valor de ε_i, las ecuaciones 5.5 y 5.6 representan un sistema de ecuaciones en que las incógnitas son los vectores **I** y **J**. Una vez obtenidos estos vectores es si pueden obtener los versores **i** y **j** normalizando, y Z_0 se obtiene de la norma de cualquiera de los vectores **I** o **J**. A esta parte del del algoritmo se le llama *POS* (*P*ose from *O*rthography 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:

$$\begin{aligned} M_0 M_i \cdot \mathbf{I} &= x_i^{'} - x_0 \\ M_0 M_j \cdot \mathbf{J} &= y_i^{'} - y_0 \end{aligned} \tag{5.8}$$

Si se escribe la ecuación 5.8 para los n puntos del modelo, se tiene un sistema de n ecuaciones con **I** y **J** como incógnitas

$$\begin{aligned} AI &= x^{'} - x_0 \\ AJ &= y^{'} - y_0 \end{aligned} \tag{5.9}$$

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 **A** es de rango 3, y las soluciones al sistema están dadas por

$$\begin{aligned} I &= \mathbf{B}x^{'} - x_0 \\ J &= \mathbf{B}y^{'} - y_0 \end{aligned} \tag{5.10}$$

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

Una vez obtenidos **I** y **J** se calculan s y los versores **i**, **j** y **k**

$$s = (|\mathbf{I}||\mathbf{J}|)^{1/2} \tag{5.11a}$$

$$i = \frac{\mathbf{I}}{s} \tag{5.11b}$$

$$j = \frac{\mathbf{J}}{s} \tag{5.11c}$$

$$k = i \times j \tag{5.11d}$$

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

$$OM_0 = \frac{Z_0}{f} Om_0 = \frac{Om_0}{s} \tag{5.12}$$

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 **i**, **j**, **k** y **T** se calculan los valores actualizados de ε_i según la ecuación 7.7. Si la variación de los ε_i es mayor a un determinado umbral, se repite el procedimiento actualizando las proyecciones SOP en 5.9, si es menor al umbral se deja de iterar y se guarda la pose calculada.

5.2.5. POSIT para puntos coplanares

Como se mencionó anteriormente, el algoritmo POSIT no funciona en el caso en que los puntos puntos del modelo 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 [7].

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

$$M_0 M_i \cdot \mathbf{I} = x_i^{'} - x_0.$$

Esto quiere decir que si se toma que la base de **I** en M_0 , la punta del vector **I** se proyecta sobre el vector M_0M_i en un punto H_{xi}, entonces todas las posibles puntas del vector **I** se encuentran en el plano perpendicular a M_0M_i que pasa por el punto H_{xi}. Si se tuvieran 4 puntos no coplanares M_0, M_1, M_2 y M_3, el vector **I** quedaría determinado. La base de **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_{x1}, H_{x2} y H_{x3} respectivamente. Para este caso el sistema definido en 5.9 es de rango 3.

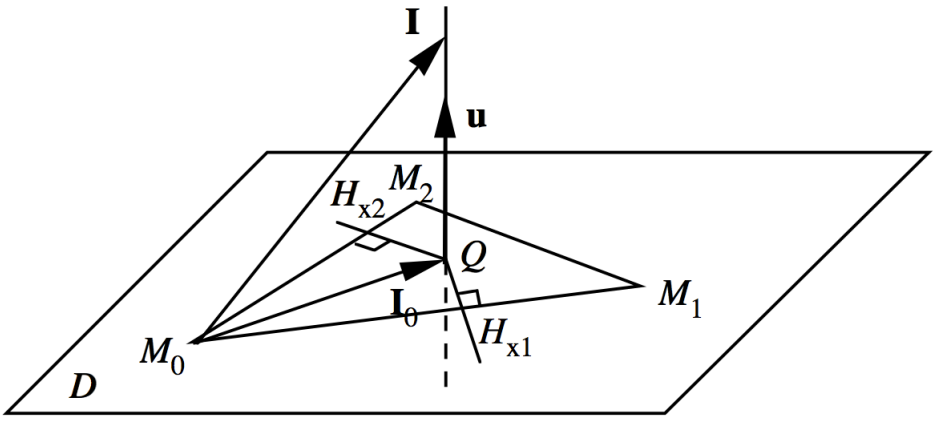


Figura 5.2: Configuración de puntos coplanares pertenecientes al plano D. Los planos perpendiculares que pasan por los puntos H_{x1} y H_{x2} se intersectan en un recta que pasa por el punto Q. Se puede ver que si hubiera un 4^{to} punto, el plano perpendicular correspondiente haría aparecer 2 rectas paralelas. Fuente: [7] .

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_{x1}, H_{x2} y H_{x3}, se intersectan todos un una línea o en dos líneas paralelas por lo tanto hay infinitas soluciones para el vector **I**. En este caso el sistema de ecuaciones 5.9 queda de rango 2. El vector solución que se obtiene al realizar la pseudo inversa

de **A** es el que está a menor distancia de los planos, en la figura 5.2 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

$$\begin{aligned} \mathbf{I} &= \mathbf{I}_0 + \lambda \mathbf{u} \\ \mathbf{J} &= \mathbf{J}_0 + \mu \mathbf{u} \end{aligned} \tag{5.13}$$

donde **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 **I** y **J** según el versor **u**. Para encontrar las soluciones hay que calcular el versor **u** y los valores de λ y μ.

Como el vector **u** es perpendicular al plano de los puntos característicos se cumple $M_0M_i \cdot \mathbf{u} = 0$, se puede hallar entonces como la base del núcleo de la matriz **A**. En la práctica este vector se halla a partir de la descomposición *SVD* de la matriz **A**. La descomposición en valores singulares de la matriz **A** queda:

$$A = U\Sigma V^T \tag{5.14}$$

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

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

$$\lambda\mu = -\mathbf{I}_0 \cdot \mathbf{J}_0 \tag{5.15}$$

Como tienen que ser del mismo largo se tiene que

$$(\mathbf{I}_0 + \lambda \mathbf{u}) \cdot (\mathbf{I}_0 + \lambda \mathbf{u}) = (\mathbf{J}_0 + \mu \mathbf{u}) \cdot (\mathbf{J}_0 + \mu \mathbf{u}) \Leftrightarrow \lambda^2 - \mu^2 = \mathbf{J}_0^2 - \mathbf{I}_0^2 \tag{5.16}$$

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 5.15 y 5.16 se llega a que

$$C^2 = \mathbf{J}_0^2 - \mathbf{I}_0^2 - 2i\mathbf{I}_0 \cdot \mathbf{J}_0 \tag{5.17}$$

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:

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

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

$$\lambda_1 = \rho\cos\theta, \qquad \mu_1 = \rho\sin\theta \tag{5.18a}$$
$$\lambda_2 = -\rho\cos\theta, \qquad \mu_2 = -\rho\sin\theta \tag{5.18b}$$

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

$$\mathbf{I}_1 = \mathbf{I}_0 + \rho\cos\theta\mathbf{u}, \qquad \mathbf{J}_1 = \mathbf{J}_0 + \rho\sin\theta\mathbf{u} \tag{5.19a}$$
$$\mathbf{I}_2 = \mathbf{I}_0 - \rho\cos\theta\mathbf{u}, \qquad \mathbf{J}_2 = \mathbf{J}_0 - \rho\sin\theta\mathbf{u} \tag{5.19b}$$

Como el vector $\mathbf{u}$ es perpendicular la plano del objeto, la solución encontrada en 5.19a es simétrica a 5.19b. 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 5.9.

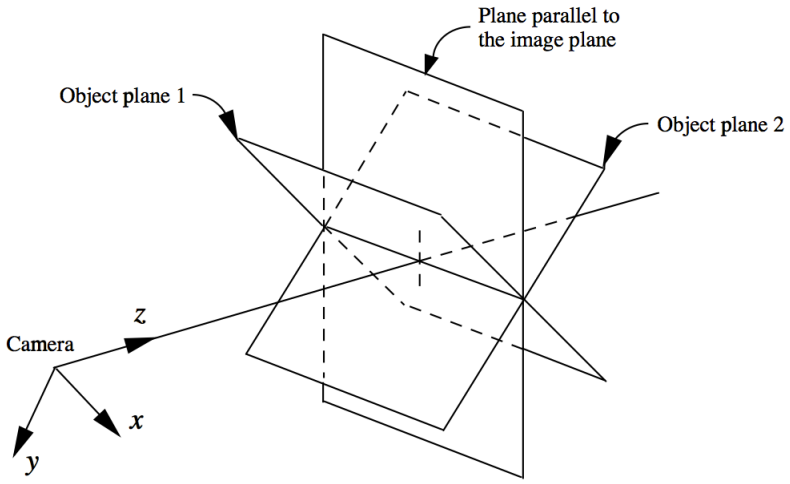


Figura 5.3: Dos objetos dando la misma proyección SOP. Fuente: [7].

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.

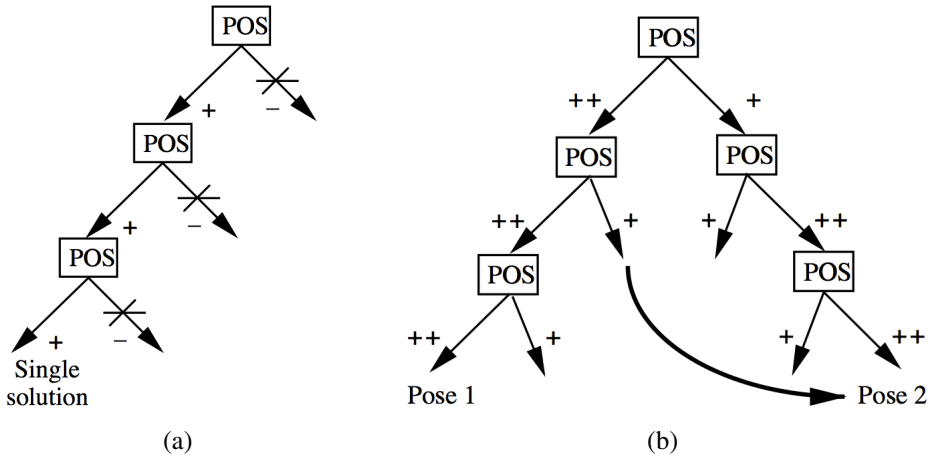


Figura 5.4: (a):Caso en el que solo una pose de las dos iniciales es coherente, también en las siguientes iteraciones solo una de las dos poses es posible, se tiene un única solución. (b): Caso en el que en cada paso hay dos posibilidades, se opta por la mejor pose(++ mejor pose, + peor pose) en cada rama. Fuente: [7].

5.3. 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 capitulo 1 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.

5.3.1. 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 5.8 que se presentó en la sección 5.2.4 se conocían las coordenadas del punto m_0, por lo que las incógnitas de esta ecuación eran solamente los vectores **i**, **j**.

$$M_0M_i \cdot \mathbf{I} = x_i^{'} - x_0$$
$$M_0M_j \cdot \mathbf{J} = y_i^{'} - y_0$$

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

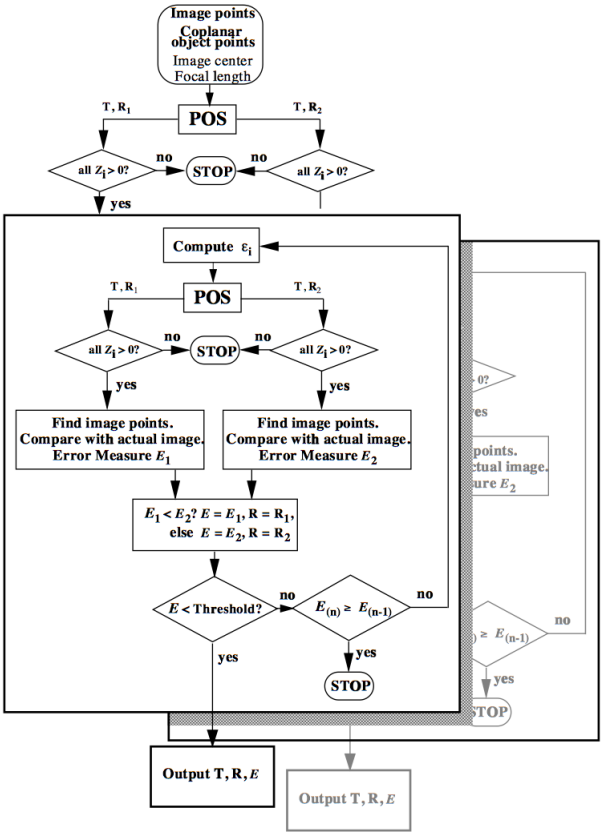


Figura 5.5: Diagrama de flujo del algoritmo para puntos coplanares, E es el error de reproyección, la condición $Z_i > 0$ verifica que los puntos reproyectados están por delante del plano imagen. Fuente: [7].

obtener el vector de traslación. Sabiendo que

$$X_0 = x_0/s$$
$$Y_0 = y_0/s$$

se puede reescribir la ecuación 5.8 como

$$\begin{aligned} x_i^{'} &= M_0 M_i \cdot s\mathbf{i} + sX_0 \\ y_i^{'} &= M_0 M_j \cdot s\mathbf{j} + sY_0 \end{aligned} \tag{5.20}$$

El sistema a resolver queda

$$A \cdot \begin{bmatrix} \mathbf{I} & \mathbf{J} \\ sX_0 & sY_0 \end{bmatrix} = \begin{bmatrix} x^{'} & y^{'} \end{bmatrix} \tag{5.21}$$

donde la matriz $\mathbf{A}$ son los puntos del modelo 3D en coordenadas homogéneas. Este sistema se puede resolver, utilizando mínimos cuadrados, como se vio en la sección 5.2.4. Se calculan la pseudo inversa de la matriz $\mathbf{A}$ y luego se obtienen $\mathbf{i}$, $\mathbf{j}$ y $\mathbf{k}$ como se vio en 5.11. Finalmente el vector de traslación se obtiene como.

$$X_0 = \frac{(sX_0)}{s} \qquad Y_0 = \frac{(sY_0)}{s} \qquad Z_0 = \frac{f}{s} \tag{5.22}$$

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

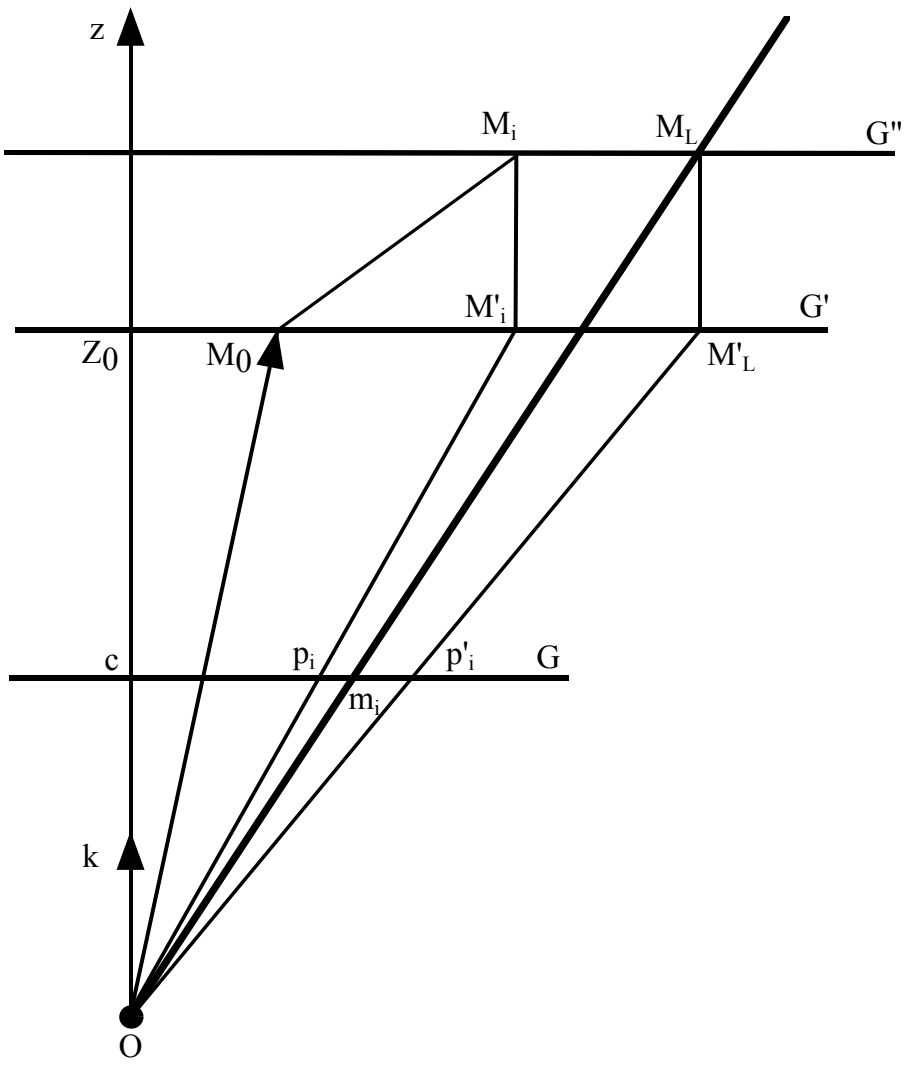


Figura 5.6: Interpretación geométrica de POSIT. El punto p_i es la proyección SOP de M_i que es el término de la derecha de la ecuación 5.20. El punto $p_i^{'}$ es la proyección SOP de M_L, ubicado en la línea de vista de m_i, corresponde al término de la izquierda de la ecuación 5.20. Para que la ecuación se satisfaga se tiene que cumplir que p_i y $p_i^{'}$ sean iguales. Fuente: [5].

5.3.1 se puede ver geométricamente cual es la distancia que se busca minimizar. En el término de la derecha de 5.20,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 5.20, 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 5.20 son iguales. Por lo tanto la ecuación que se busca minimizar es la siguiente:

$$E = \sum_i \left((\mathbf{Q}_1 \cdot M_0M_i - (1+\varepsilon_i)x_i)^2 + (\mathbf{Q}_2 \cdot M_0M_i - (1+\varepsilon_i)y_i)^2 \right) \tag{5.23}$$

donde

$$\begin{aligned} \mathbf{Q}_1 &= s(i, X_0) \\ \mathbf{Q}_2 &= s(j, Y_0) \end{aligned} \tag{5.24}$$

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:

$$\mathbf{Q}_1 = \left(\sum_i M_0M_i^T M_0M_i\right)^{-1}\left(\sum_i (1+\varepsilon_i)\, x_i M_0M_i\right) \tag{5.25}$$

$$\mathbf{Q}_2 = \left(\sum_i M_0M_i^T M_0M_i\right)^{-1}\left(\sum_i (1+\varepsilon_i)\, y_i M_0M_i\right) \tag{5.26}$$

La matriz $L = \left(\sum_i M_0M_i^T M_0M_i\right)$ 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.

5.3.2. 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

$$d_{ji}^2 = \left(\mathbf{Q}_1 \cdot M_iM_0 - (1+\varepsilon_i)\, x_j\right)^2 + \left(\mathbf{Q}_2 \cdot M_iM_0 - (1+\varepsilon_i)\, y_j\right)^2$$

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:

$$\begin{aligned} E &= \sum_{j=1}^{N}\sum_{i=1}^{M} m_{ji}(d_{ji}^2 - \alpha) \\ &= \sum_{j=1}^{N}\sum_{i=1}^{M} m_{ji}\left(\left(\mathbf{Q}_1 \cdot M_0M_i - (1+\varepsilon_i)x_j\right)^2 + \left(\mathbf{Q}_2 \cdot M_0M_i - (1+\varepsilon_i)y_j\right)^2\right) \end{aligned} \tag{5.27}$$

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

$$\mathbf{Q}_1 = \left(\sum_{i=1}^{M} m_i' M_0M_i^T M_0M_i\right)^{-1}\left(\sum_{j=1}^{N}\sum_{i=1}^{M} m_{ji}\,(1+\varepsilon_i)\, x_j M_0M_i\right) \tag{5.28}$$

$$\mathbf{Q}_2 = \left(\sum_{i=1}^{M} m_i' M_0M_i^T M_0M_i\right)^{-1}\left(\sum_{j=1}^{N}\sum_{i=1}^{M} m_{ji}\,(1+\varepsilon_i)\, y_j M_0M_i\right) \tag{5.29}$$

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.

5.3.3. 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 los 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

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

en donde β es una constante muy peque/ na y la fila $N+1$ y la columna $M+1$ son inicializadas con constantes peque/ nas. 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 peque/ no. La matriz resultante cumple que todas la filas y columnas suman 1.

2 Se incrementa el valor de β a media 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 esta muy cerca de ser un matriz de ceros y unos.

5.4. 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 5.2.5 cuando los puntos son coplanares, al resolver el sistema 5.9 se obtienen las proyecciones de los vectores **i** y **j** sobre el plano del objeto. Se utilizó el enfoque de POSIT moderno para hallar las proyecciones de **i** y **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

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

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

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 5.2.5

5.5. 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.

Figura 5.7: Posiciones que se utilizaron para las imágenes de prueba

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 **??** 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

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

	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 5.1: Error de proyección de imágenes de pruebas

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 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 1. En general se puede ver que el error de proyección y la varianza son un poco menores para el caso de modern POSIT.

	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 5.2: Error de proyección en imágenes sintéticas

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.

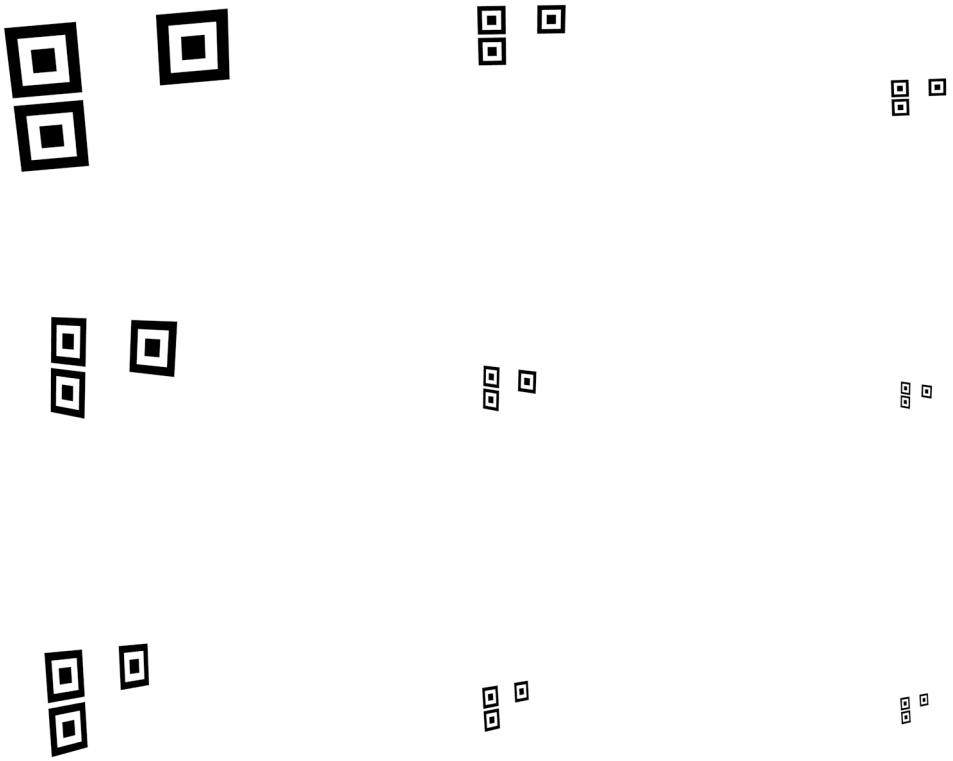


Figura 5.8: Posiciones que se utilizaron para las imágenes sintéticas

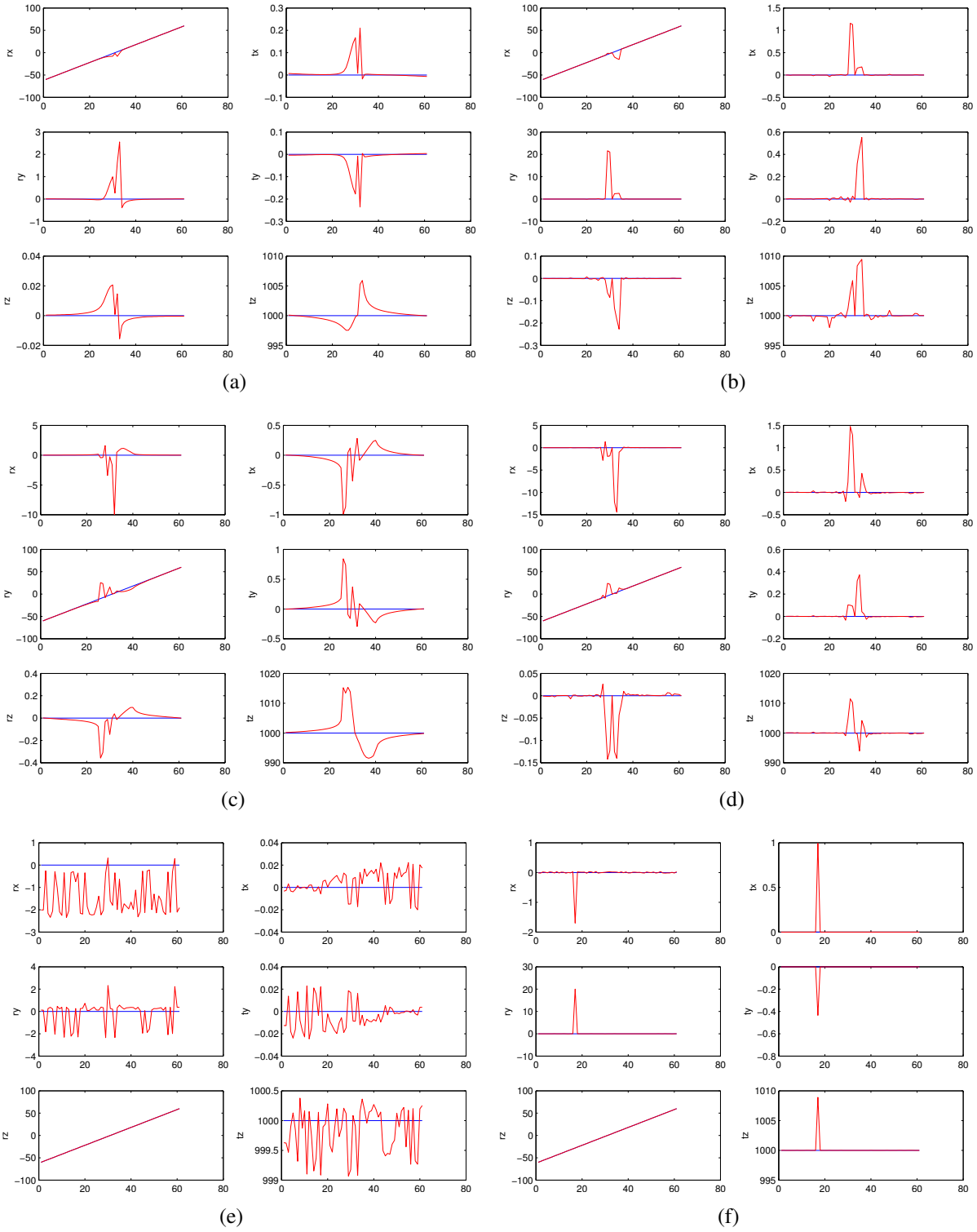


Figura 5.9: Rotación según eje *x* para Modern POSIT **??** y para Classic POSIT **??**, rotación según eje *y* para Modern POSIT **??** y para Classic POSIT **??** y rotación según eje *z* para Modern POSIT **??** y para Classic POSIT **??**

CAPÍTULO 6

Herramientas de *rendering*

6.1. 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* para plataformas móviles, en especial que funcionen sobre iOS, algunas de ellas serán comentadas a continuación.

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 diseñado para ser utilizado 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 grátis 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. La primera que se encontró fue "ISGL3D". ISGL3D es un *framework* (marco de trabajo) escrito en Objective-C que trabaja sobre Open GL ES y que busca facilitar la tarea del programador a la hora de crear y manipular escenas 3D mediante una "*Application Program Interface*" (API) sencilla e intuitiva. Es un proyecto grátis 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 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, Cocos3D cuenta con una comunidad mucho más activa, lo que facilita mucho su uso.

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.

6.2. ISGL3D

ISGL3D es un *framework* 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 ajenas. Una buena manera de iniciarse en manejo de la herramienta es siguiendo los tutoriales en: `www.isgl3d.com/resources/tutorials`; al menos este fué 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:** mustra 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 hacrlos 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 y se muestra cómo *renderizar* objetos con transparencia.

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* esta 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 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 la cámara 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.

Un particularidad de la cámara de ISGL3D es que el parámetro intrínseco "distancia focal" visto la sección 1.2, no es directamente configurable. En cambio, el valor que sí se puede alterar es el llamado "FOV", acónimo de "Fiel 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.

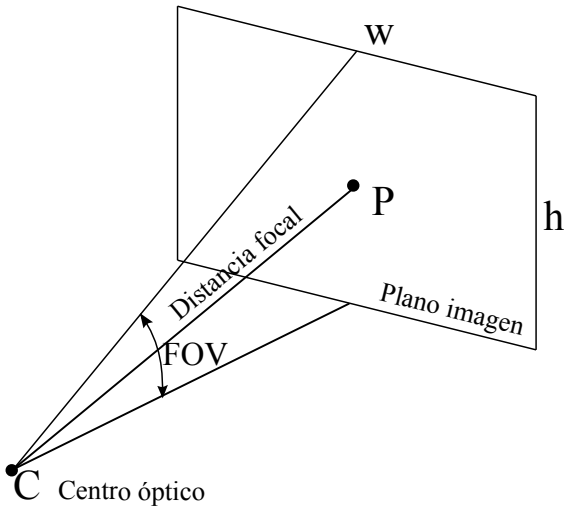


Figura 6.1: Definición gráfica del FOV.

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

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

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

ISGL3D cuanta 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 cilindio.
  - *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:** modello 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.

Para la creación de cada primitiva, se debe especificar además, la cantidad de segmentos utilizados en las distintas dimensiones. En la figura 1.2 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];
```

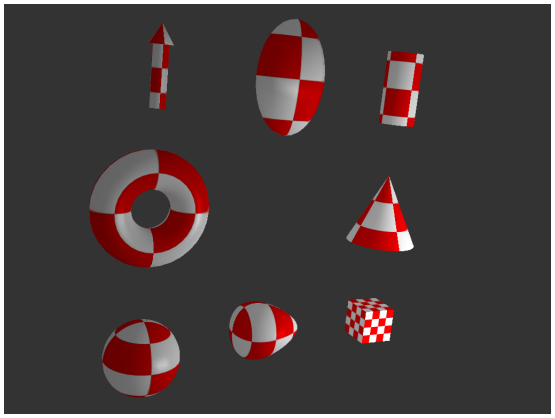


Figura 6.2: Principales primitivas en ISGL3D.

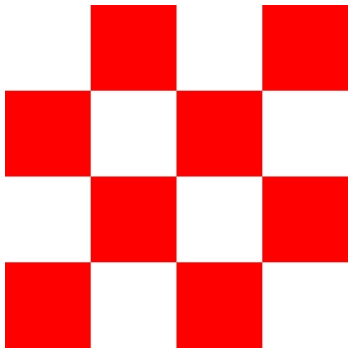


Figura 6.3: Imagen *red_checker.png*, utilizada para crear la textura asociada a las primitivas de la figura 1.2.

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 1.3. 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.

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 *POD*, 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 asignará el modelo; y se agrega a la escena. Finalmente, se le asigna al modelo una posición en la escena. En la figura 1.4 se puede ver un modelo de José Artigas, agregado dos veces a una misma escena, pero visto desde ángulos distintos.

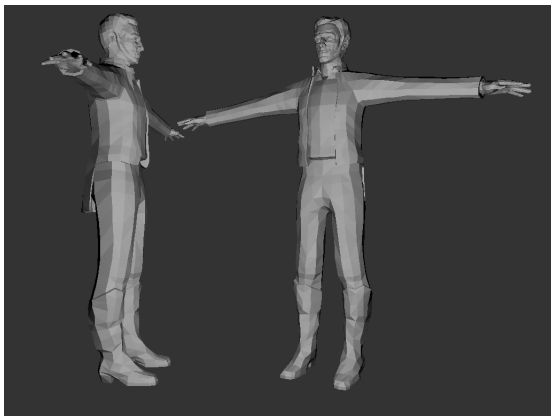


Figura 6.4: Modelo de José Artigas agregado dos veces en una misma escena, pero visto desde ángulos distintos.

CAPÍTULO 7

Casos de Uso

7.1. 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.

7.2. Caso de Uso 01

7.2.1. Comentarios sobre el caso de uso

7.2.2. Detalles constructivos

7.3. Caso de Uso 02

7.3.1. Comentarios sobre el caso de uso

Este caso de uso básicamente busca desplegar un video en una superfice 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.

7.3.2. 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 afin o incluso, de manera más genérica, una homografía.

7.3.3. *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 **??** 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 **??** se obtienen los 8 valores restantes de la matriz y es posible asignarle una homografía a un objeto *UIView* para resolver el problema presente.

7.3.4. 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:

$$xh_{11} + yh_{12} + h_{13} = i \tag{7.1}$$

$$xh_{21} + yh_{22} + h_{23} = j \tag{7.2}$$

$$xh_{31} + yh_{32} + h_{33} = 1 \tag{7.3}$$

Multiplicando la ecuación (7.3) por i e igualándola a la ecuación (7.1) se obtiene lo siguiente:

$$xh_{11} + yh_{12} + h_{13} = ixh_{31} + iyh_{32} + ih_{33} \tag{7.4}$$

o lo que es lo mismo:

$$xh_{11} + yh_{12} + h_{13} - ixh_{31} - iyh_{32} - ih_{33} = 0 \tag{7.5}$$

Procediendo de manera análoga y multiplicando la ecuación (7.3) por j e igualándola a la ecuación (7.2) se obtiene lo siguiente:

$$xh_{21} + yh_{22} + h_{23} = jxh_{31} + jyh_{32} + jh_{33} \tag{7.6}$$

o lo que es lo mismo:

$$xh_{21} + yh_{22} + h_{23} - jxh_{31} - jyh_{32} - jh_{33} = 0 \tag{7.7}$$

Las ecuaciones (7.3) y (7.7) 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.

7.4. Caso de Uso 03

7.4.1. Comentarios sobre el caso de uso

7.4.2. Detalles constructivos

7.5. Caso de Uso 04

7.5.1. Comentarios sobre el caso de uso

7.5.2. Detalles constructivos

CAPÍTULO 8

Implementación

8.1. Introducción

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

(1) Detección QR

(2) Navegación por listas de cuadros

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

(4) Detección SIFT para identificar el cuadro.

(5) Diferentes realidades aumentadas según la obra.

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

8.2. Diagrama global de la aplicación

Para que sea más sencilla la comprensión de los bloques que componen la aplicación a continuación se muestra el *Storyboard* de la aplicación que es la interfaz de usuario y que sirve para visualizar bastante cada una de las clases que intervienen y cómo es el flujo de la aplicación.
A continuación se pasará a explicar algunas de las clases implementadas en la aplicación y que tienen cierta relevancia. Se mostrarán sus principales características y su rol dentro de la aplicación.

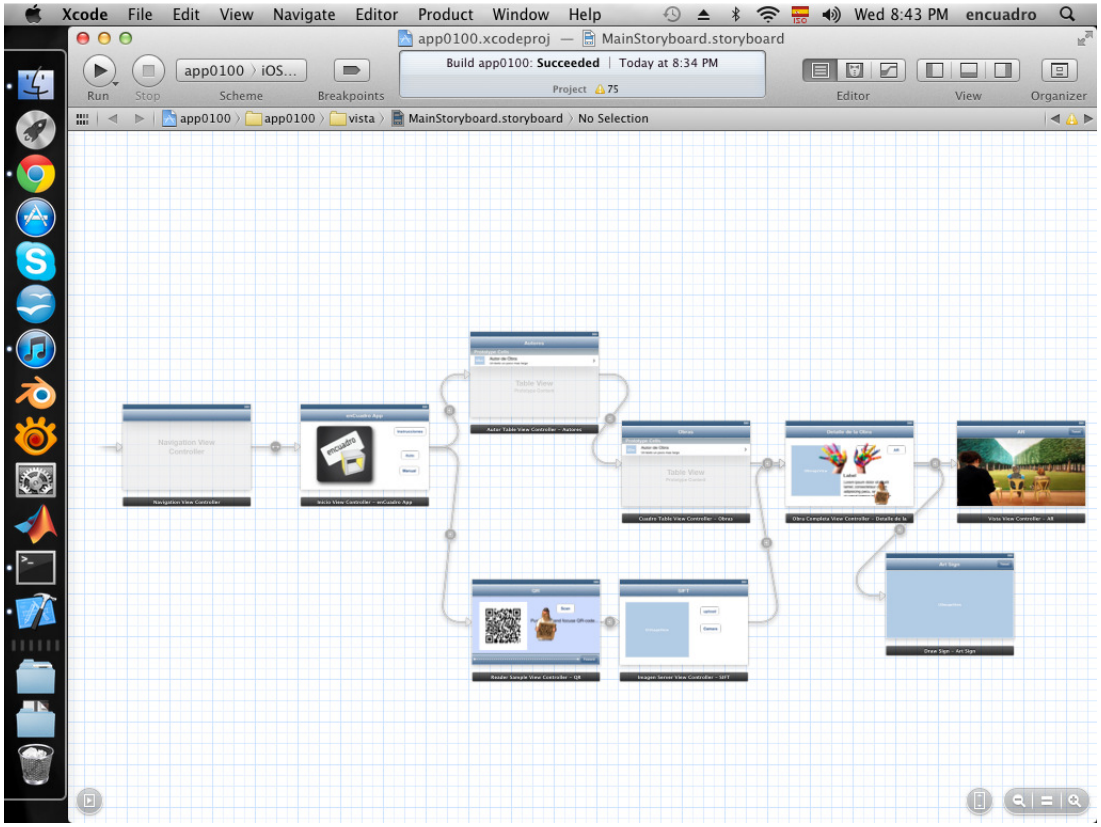


Figura 8.1: Diagrama global de la aplicación

8.2.1. NavigationViewController

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

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

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

Esto lo que hace es fijar la orientación de la interfaz de usuario a modo *LandscapeRight*. También es posible lograr esto editando el archivo *Info.plist* que toda aplicación de xcode tiene y agregando el

item *SupportedInterfaceOrientations* y completado las opciones que se desean. Las autorotaciones es algo que tiene bastante relevancia en las aplicaciones. En particular se optó por esta forma, es decir, bloquear las autorotaciones y dejar solamente la vista en un sentido, para facilitar la reproyección de la realidad aumentada. De no haberlo hecho de esta manera, con cada rotación de la interfaz se tendrían que intercambiar los ejes de coordenadas en función del sentido de la rotación. Esto es posible de hacer ya que con cada rotación se ejecuta una serie de métodos en forma automática entre los cuales se encuentra el siguiente

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

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

8.2.2. InicioViewController

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

8.2.3. TableViewControllers

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

8.2.3.1. AutorTableViewController

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

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

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

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

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

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

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

8.2.3.2. CuadroTableViewController

Esta clase es muy similar a la clase *AutorTableViewController* recién descripta difiriendo simplemente en su contenido. Los conceptos utilizados y métodos implementados son básicamente los mismos pero su contenido es un listado de obras en lugar de autores. Una especificación extra es que al ejecutrase el método *viewDidLoad* se completa una lista de cuadros diferente en función del autor seleccionado. Así como en la clase *AutorTableViewController* en esta también se implementa el método *prepareForSegue* para poder completar los datos de la instancia de la clase con la que se está conectando, completando los datos de la obra seleccionada (autor, obra, imagen, descripción, audio, ARid). El ARid es un identificador de realidad aumentada que asocia una realidad aumentada a cada cuadro. Esto se verá más adelante en la sección 8.6.

8.2.3.3. CuadroTableViewCell

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

8.2.4. ReaderSampleViewController

Este *ViewController* es el encargado de hacer la lectura de los códigos QR y de invocar los métodos necesarios para realizar la búsqueda de la zona del museo en la que se encuentra el usuario. Esto es, existe un código QR asociado a cada autor (Blanes, Figari y Torres García) y en base al código QR leído se despliega un texto e imagen asociadas al mismo. El funcionamiento de la decodificación se explica un poco más en detalle en la sección 8.3.

8.2.5. ImagenServerViewController

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

Figura 8.2: Autor y Cuadro TableViewControllers

Figura 8.3: Ejemplo de captura para reconocimiento SIFT

CompletaViewController. Dentro del método *prepareForSegue*, encargado de preparar todo previo a la invocación de *ObraCompletaViewController* se invoca el método *uploadImage*. Este método genera un mensaje HTTP del tipo POST y se lo envía a la IP del servidor. En el cuerpo del mensaje se agrega la foto sacada previamente y se le agrega un string llamado *room*. Este string es completado previamente en el *ReaderSampleViewController* en base al QR detectado, dando la información sobre en qué sala/región del museo se encuentra (sala Figari, sala Blanes o sala Torres García). Este string lo que permite es tener un identificador para poder hacer la búsqueda de la imagen sacada (luego de procesar la imagen con el algoritmo SIFT), en una base de datos más peque?a, que contenga solamente los cuadros de esa región del museo y no todos los cuadros del museo. En caso que el usuario no haya detectado ningún QR y haya seleccionado directamente la opción de sacar una foto para comenzar la comunicación con el servidor, entonces este string estará vacío y la búsqueda de la imagen más parecida se hace en toda la base de datos. En un primer caso, claramente los tiempos son mejores (del orden de 3s en una LAN) que en el segundo (del orden de 6s en una LAN).
Luego de establecida la conexión y enviada la consulta POST, el servidor responde con un string

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

(1) autor

(2) obra

(3) texto

(4) imagen

(5) audio

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

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

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

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

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

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

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

8.2.6. ObraCompletaViewController

Este *ViewController* simplemente es la presentación de la obra, mostrando el cuadro, título, autor, descripción y distintas opciones para interactuar con el mismo. Tiene dos botones y una animación que funciona como botón. El primero de los botones dispara un audio donde se explican detalles de la obra que el usuario está contemplando. El otro botón conecta con el *VistaViewController*, encargado de mostrar la realidad aumentada, explicado en la sección 8.2.7. La animación que aparece funciona como *segue* hacia otro *ViewController*, llamado *DrawSign* que se explica más adelante en la sección 8.2.8.

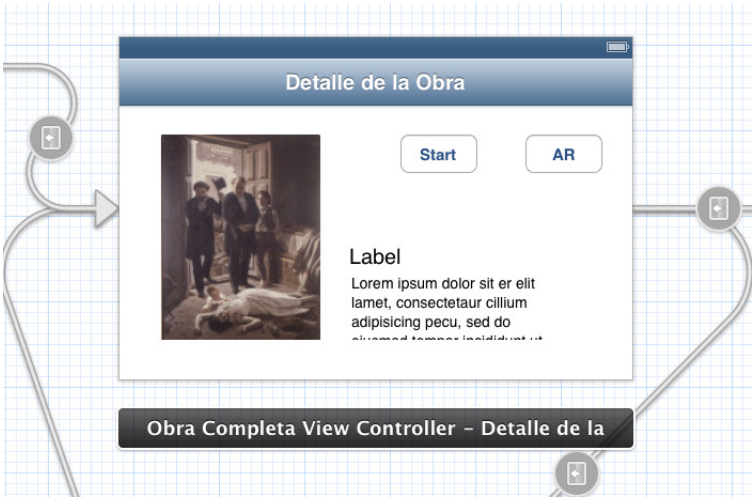


Figura 8.4: Pantalla con la obra completa

8.2.7. VistaViewController

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

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

    [self hacerRender];

}
```

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

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

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

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

8.2.8. DrawSign

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

Figura 8.5: Ejemplo de dibujo libre

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

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

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

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

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

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

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

```
[touch tapCount];
```

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

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

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

8.2.9. TouchVista

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

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

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

8.2.10. Isgl3dViewController y app0100AppDelegate

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

8.3. QR

8.3.1. 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.

8.3.2. 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 8.1 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.

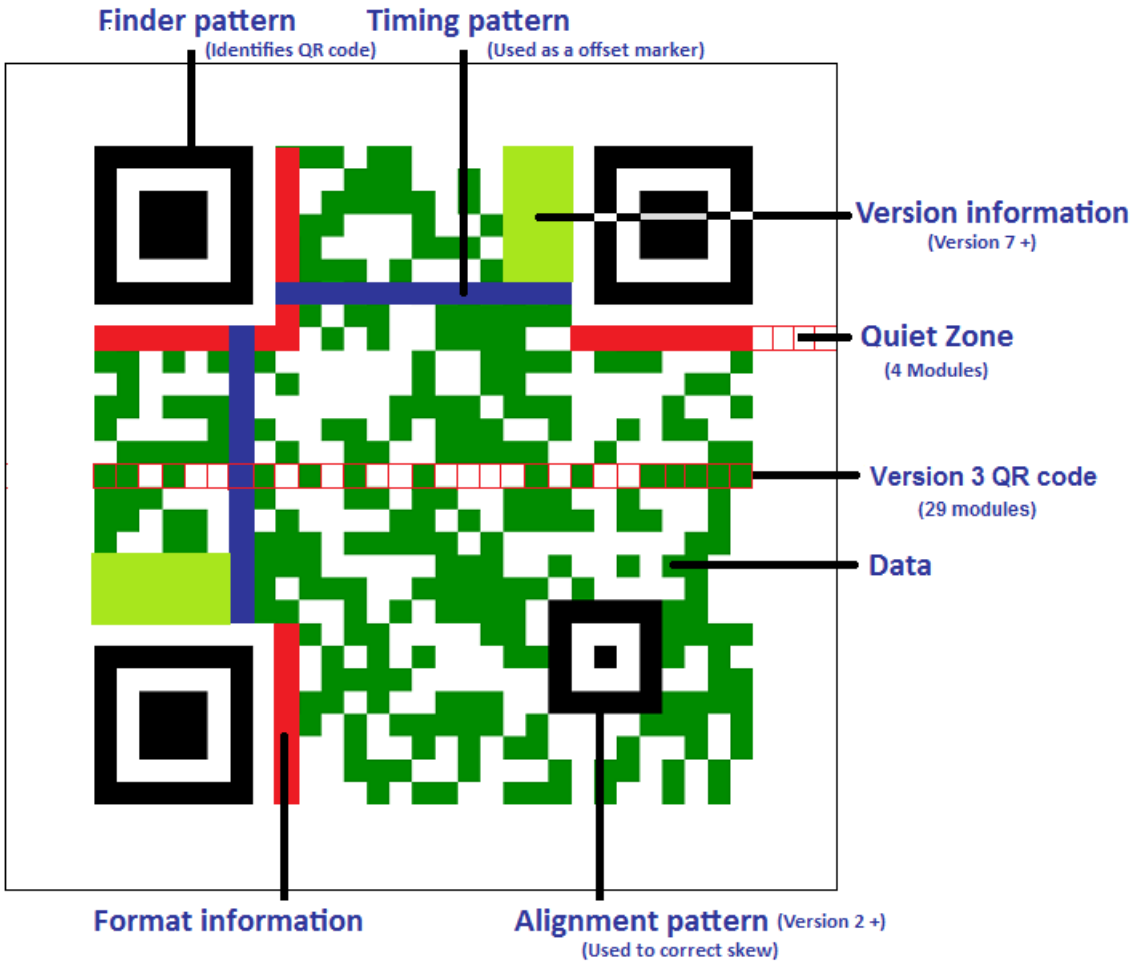


Figura 8.6: Las distintas componentes de un QR. Fuente [8].

8.3.3. Codificación y decodificación de QRs

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

8.3.4. 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.

8.3.5. 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 mo-

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

Figura 8.7: Ejemplo de un QR artista. Fuente [8].

8.4. Servidor

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

8.4.1. 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.

8.4.1.1. Servidor iOS

redactar pasos...

8.4.1.2. Servidor LAMP

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

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

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

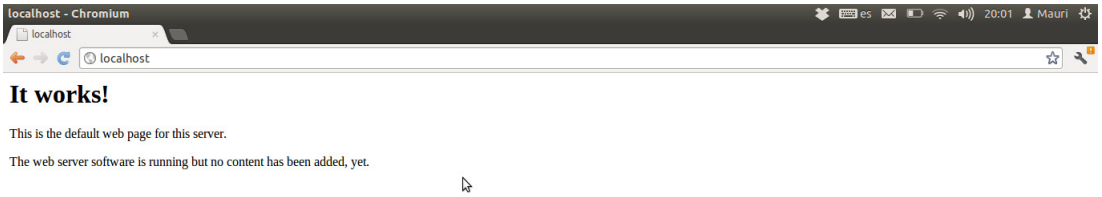


Figura 8.8: Impresión de pantalla al ingresar a http://localhost.

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

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

8.4.2. Lenguaje php y principales scripts

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

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

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

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

8.5. SIFT

8.6. Incorporación de la realidad aumentada a la aplicación

asdfasdfasdf

[9].

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] Denis Oberkampf, Daniel F. DeMenthon, and Larry S. Davis. Iterative pose estimation using coplanar feature points. *Comput. Vis. Image Underst.*, 63(3):495–511, may 1996.

[8] 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.

[9] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[10] 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.

[11] Zhengyou Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *ICCV*, pages 666–673, 1999.

[12] Jane Heikkilä and Olli Silvén. A four-step camera calibration procedure with implicit image correction. In *1997 Conference on Computer Vision and Pattern Recognition (CVPR 97), June 17-19, 1997, San Juan, Puerto Rico*, page 1106. IEEE Computer Society, 1997.

[13] Jean-Yves Bouguet. Camera calibration toolbox for matlab. `http://www.vision.caltech.edu/bouguetj/calib_doc/`, November 2012.

[14] Helmut Zollner and Robert Sablatnig. Comparision of methods for geometric camera calibration using planar calibration targets. In W. Burger and J. Scharinger, editors, *Digital Imaging in Media and Education, Proc. of the 28th Workshop of the Austrian Association for Pattern Recognition (OAGM/AAPR)*, volume 179, pages 237–244. Schriftenreihe der OCG, 2004.

[15] R.Y. Tsai. An efficient and accurate camera calibration technique for 3d machine vision. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 364–374, 1986.

[16] Y. I. Abdel-Aziz and H. M. Karara. Direct linear transformation from comparator coordinates into object space coordinates in close-range photogrammetry. In *Proceedings of the Symposium on Close-Range photogrammetry*, volume 1, pages 1–18, 1971.