Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Realidad aumentada utilizando un iPad

Tesis que presenta

César David Corona Arzola

para obtener el Grado de

Maestro en Ciencias en Computación

Director de la Tesis

Dr. Luis Gerardo de la Fraga

México, Distrito Federal **Septiembre, 2013**

Resumen

La realidad aumentada combina elementos virtuales generados en una computadora en escenas físicas reales, esto con la finalidad de complementar o facilitar el entendimiento de la escena real.

En este trabajo se establecen algunos criterios para el desarrollo de sistemas de realidad aumentada en el iPad, un dispositivo restringido tanto en poder de cómputo como en memoria. El iPad es una tableta de uso muy intuitivo y posee la ventaja de incorporar en un solo dispositivo una pantalla de despliegue –de adecuadas dimensiones para desarrollar aplicaciones de realidad aumentada–, una cámara y los sensores para interactuar tocando la pantalla, un giroscopio y un acelerómetro.

Para lograr llevar al iPad un sistema de realidad aumentada es indispensable resolver el problema que representa migrar una aplicación demandante en recursos de un entorno completo a uno limitado, empleando marcadores cuadrados para la detección de la posición de la cámara y dibujando los objetos virtuales en la escena, mediante la biblioteca *OpenGL ES*, que está optimizada para dispositivos móviles.

Se realizaron tres aplicaciones con un enfoque educativo para demostrar la metodología de diseño: (1) un fascículo de especies para niños, donde el modelo de la especie aparece de forma virtual sobre un marcador, (2) una aplicación para el aprendizaje del lenguaje japonés mediante tarjetas con marcadores, donde el significado real de los símbolos se indica empleando modelos virtuales y (3) una aplicación que simula el movimiento de una pelota virtual sobre un laberinto que además usa el giroscopio y el acelerómetro del iPad.

Abstract

Augmented reality combines computer-generated virtual elements within real-world physical scenes in order to complement them or to make easier to understand the scene itself.

In this project we set out some criteria for the development of augmented reality systems on iPad, a restricted device in both computing power and memory storing. iPad is a tablet with a very intuitive use and has the advantage of incorporating a display screen –with the proper size that fits the development of augmented reality applications–, a camera and the necessary sensors to interact by touching the screen, a gyroscope and an accelerometer.

In order to program an augmented reality system on iPad, it is essential to solve the problem of migrating an application that uses lots of resources in a non-constrained environment to a limited one applying square markers for detecting the camera position and drawing virtual objects in scene using *OpenGL ES*, a library which is optimized for mobile devices.

We made three applications with an educational approach to demonstrate our design methodology: (1) a booklet of species for children, where the model of the species virtually appears on a marker, (2) an application for japanese language learning through cards with a marker on it, where the meaning of the symbols is shown using virtual models and (3) an application that simulates the movement of a virtual ball on a maze, which additionaly uses iPad's gyroscope and accelerometer.

Agradecimientos

Al CONACyT

Pues si no fuera por su apoyo y por el proyecto CB2011/168357, no hubiera ingresado al programa de Maestría ni realizado este proyecto. Admiro su interés en el fomento a la ciencia.

Al CINVESTAV

Por invitarme cordialmente a formar parte de su programa de Maestría. Este fascinante y agradable lugar que conocí desde 2003 y en el cual deseaba algún día estudiar. Quién diría que pasarían 10 años desde ese momento hasta ahora.

Al Dr. Luis Gerardo de la Fraga

Por proponerme este interesante proyecto, por sus comentarios, su apoyo al brindarme todo el material necesario así como un espacio propio para poder realizar este trabajo, por entenderme y sobre todo por su gran paciencia, a pesar de mis errores.

A los profesores del Departamento de Computación

Por su ardua labor día a día y su interés en nuestra formación. En especial, gracias a quienes revisaron la presente tesis: el ***Dr. José Gpe. Rodríguez*** *y la* ***Dra. Sonia Mendoza****.*

Al personal del Departamento de Computación

A las tres agradables señoritas que nos facilitan las labores administrativas y a aquellos que forman la infraestructura del departamento y mantienen limpias nuestras áreas.

A mi familia

*Por su apoyo incondicional y por creer en mí en todo momento. Gracias a mis hermanos cuyos conocimientos en biología y buen gusto en diseño (****MoYoW C.C.****) influyeron bastante en las aplicaciones desarrolladas.*

A mis amigos

Aquellos que amenizaron mi estancia en el CINVESTAV, en especial Iván y Charly. Y en general a mis amigos de toda la vida: Angélica, Ana, Dianita, Alma, Toño, Tere, Karmen, Silver, Arturo, PP, Alex y Ricardito. Nunca los olvidaré, aun si perdemos contacto.

A mis otros amigos

Cuyos maullidos y ladridos que junto con mi música son indispensables en mi día a día.

A todos ustedes les dejo esta frase que habla sobre lo que representa una graducación:
un nuevo comienzo, un viaje.
卒業は終わりではなく、始まりだ。
だから、新しい世界を見つけて新しい扉を開くために旅を今始めよう。

Índice general

Índice de figuras x

Índice de tablas xiii

1. Introducción 1
- 1.1. Diferencias (RA, RV) . . . 1
- 1.2. Breve historia de la RA . . . 2
- 1.3. Motivación . . . 2
- 1.4. Planteamiento del problema . . . 3
  - 1.4.1. Especificaciones de la cámara del iPad . . . 4
  - 1.4.2. Módulo de gráficos . . . 5
- 1.5. Retos para los dispositivos móviles ante la RA . . . 6
- 1.6. Objetivos del proyecto . . . 7
- 1.7. Organización de la tesis . . . 8

2. Marco Teórico 9
- 2.1. Tipos de RA . . . 9
- 2.2. Aplicaciones de la RA . . . 11
  - 2.2.1. Medicina . . . 12
  - 2.2.2. Construcción, asistencia y mantenimiento . . . 12
  - 2.2.3. Educación . . . 13
  - 2.2.4. Entretenimiento y publicidad . . . 14
- 2.3. Trabajos relacionados . . . 16
- 2.4. Tipos de patrones . . . 20
  - 2.4.1. Marcadores de plantilla . . . 20
  - 2.4.2. Códigos de barras uni y bidimensionales . . . 20
  - 2.4.3. Marcadores de referencia . . . 22
  - 2.4.4. Marcas topológicas . . . 22
- 2.5. Arquitectura de *iOS* . . . 22
- 2.6. Rasgos característicos de *Objective-C* . . . 23
  - 2.6.1. Manejo de memoria en *Objective-C* . . . 24
  - 2.6.2. Mezclando *Objective-C* y *OpenGL ES* . . . 26

3. Sistema de RA **29**
3.1. Prácticas recomendadas para RA en dispositivos móviles 29
3.2. Espacios de coordenadas 31
3.2.1. Sistemas de coordenadas en *OpenGL* 31
3.2.2. Tipos de coordenadas del sistema de RA 34
3.3. Acciones principales del sistema 36
3.3.1. Esquema general del sistema de RA 36
3.3.2. Agregar un patrón 37
3.3.3. Detectar un marcador 40
3.3.4. Calibración de la cámara y estimación de la pose 50
3.4. Manejo de información previa 59

4. Aplicación 1: Fascículo de especies **61**
4.1. Objetos 3D 62
4.1.1. Dibujando el objeto 65
4.2. Conexión con el sistema de RA 68
4.3. Controlador principal de la vista 69
4.3.1. Captura de video 70
4.3.2. Reproduciendo audio en la aplicación 71
4.4. Modo de uso de la aplicación 72

5. Aplicación 2: Kanjirama **79**
5.1. Detección de marcadores de referencia 80
5.2. Nociones básicas de la escritura japonesa 85
5.3. Interfaz de juego 87
5.4. Modo de uso de la aplicación 89

6. Aplicación 3: Laberinto **97**
6.1. Sensores del iPad 98
6.1.1. Orientación del dispositivo 99
6.1.2. Movimiento de la pelota 102
6.2. Modo de uso de la aplicación 107

7. Conclusiones **111**
7.1. Conclusiones 111
7.2. Trabajo a futuro 113

A. Fascículo de especies **115**

Bibliografía **122**

Índice de figuras

2.1. Ejemplo de RA en dispositivos móviles. 9
2.2. Ejemplos de RA basada en proyecciones. 10
2.3. Ejemplo de RA que simula profundidad. 11
2.4. RA en sala de cirugía. 12
2.5. RA en las salas escolares auxiliada por libros. 13
2.6. Ejemplo de RA en sitios públicos. 14
2.7. RA en revistas. 14
2.8. Ejemplos de aplicaciones de RA en prendas de vestir. 15
2.9. Uso de RA en Nintendo 3DS™. 15
2.10. Ejemplo de marcador de plantilla. 20
2.11. Ejemplo de código de barras 1D. 20
2.12. Ejemplo de código de barras 2D (QR). 21
2.13. Ejemplo de código de matriz de datos. 21
2.14. Ejemplo de marcador de referencia. 22
2.15. Ejemplo de marca topológica. 22
2.16. Arquitectura *iOS*. 23

3.1. Sistemas de coordenadas (*OpenGL*). 32
3.2. Sistema de coordenadas en un monitor y en *OpenGL*. 33
3.3. Tipos de coordenadas más importantes para el sistema de RA. 35
3.4. Diagrama general del sistema de RA. 36
3.5. Ejemplo de diseño de marcador de plantilla. 37
3.6. Ejemplos de matrices de intensidades contenidas en un archivo de tipo patrón. 38
3.7. Ejemplo gráfico de imagen con componentes conectadas. 41
3.8. Dos posibles casos en la estimación del segundo vértice del cuadrado. 44
3.9. Procesos para estimar el tercer y/o cuarto vértice. 44
3.10. Modelos de calibración empleados para generar archivos con parámetros intrínsecos de la cámara. 53
3.11. Dos vectores perpendiculares unitarios: $\boldsymbol{v_1}$ y $\boldsymbol{v_2}$ calculados a partir de $\boldsymbol{u_1}$ y $\boldsymbol{u_2}$. 58

4.1. Diagrama general de la primera aplicación. 61

4.2. Ejemplo del uso de *Blender* para tratar un modelo 3D utilizado en la aplicación. . . . 63
4.3. Ejemplo de imagen de textura de un modelo. . . . 67
4.4. Icono de la aplicación *Fascículo de especies*. . . . 72
4.5. Fascículo de especies. . . . 72
4.6. Animales del fascículo de especies. . . . 73
4.7. Ejemplos de etiquetas con la descripción de la especie. . . . 74
4.8. Ejemplos de la animación de un modelo 3D en la aplicación. . . . 75
4.9. Diferencia entre animación de un modelo 3D y cambio de posición del fascículo. . . . 76
4.10. Ejemplos de escalamiento de modelos 3D en la aplicación. . . . 77
4.11. Ejemplos de pruebas al sistema deformando levemente el marcador o alterando su posición. . . . 77
4.12. Ejemplos de pruebas al sistema alejando el fascículo de especies de la cámara. . . . 78

5.1. Diagrama general de la segunda aplicación. . . . 79
5.2. Ejemplo de patrón de bits. . . . 80
5.3. Icono de la aplicación *Kanjirama*. . . . 89
5.4. Ejemplo del reverso de una tarjeta de RA. . . . 90
5.5. Pantalla principal de la aplicación *Kanjirama*. . . . 90
5.6. Ejemplo del icono que aparece en la esquina superior derecha de la pantalla. . . . 91
5.7. Tarjetas de RA colocadas con el marcador boca abajo sobre una superficie plana. . . . 92
5.8. Ejemplo de información que aparece al voltear una tarjeta. . . . 92
5.9. Modelos 3D de las tarjetas de RA. . . . 93
5.10. Ejemplo de tarjeta correcta volteada. . . . 94
5.11. Ejemplos del cambio de color en la etiqueta que muestra la cuenta regresiva. . . . 94
5.12. Ejemplo de tarjeta incorrecta volteada. . . . 95
5.13. Ejemplo de ventana con los resultados de la partida. . . . 95

6.1. Diagrama general de la tercera aplicación. . . . 97
6.2. Ejes de coordenadas del acelerómetro y giroscopio. . . . 99
6.3. Icono de la aplicación *Laberinto*. . . . 107
6.4. Tarjeta de RA de la aplicación *Laberinto*. . . . 107
6.5. Ejemplos de la pelota dibujada sobre el laberinto. . . . 108
6.6. Ejemplos de la etiqueta con todas las posibles orientaciones del dispositivo. . . . 109
6.7. Selector para cambiar la imagen del laberinto. . . . 109
6.8. Laberintos distintos en la aplicación. . . . 109
6.9. Ejemplos del movimiento de la pelota en distintos laberintos. . . . 110

A.1. Fascículo de especies (cubierta frontal y portada). 115
A.2. Fascículo de especies (páginas 1 y 2). 115
A.3. Fascículo de especies (páginas 3 y 4). 116
A.4. Fascículo de especies (páginas 5 y 6). 116
A.5. Fascículo de especies (páginas 7 y 8). 116
A.6. Fascículo de especies (páginas 9 y 10). 117
A.7. Fascículo de especies (páginas 11 y 12). 117
A.8. Fascículo de especies (páginas 13 y 14). 117
A.9. Fascículo de especies (páginas 15 y 16). 118
A.10.Fascículo de especies (páginas 17 y 18). 118
A.11.Fascículo de especies (páginas 19 y 20). 118
A.12.Fascículo de especies (páginas 21 y 22). 119
A.13.Fascículo de especies (páginas 23 y 24). 119
A.14.Fascículo de especies (páginas 25 y 26). 119
A.15.Fascículo de especies (páginas 27 y 28). 120
A.16.Fascículo de especies (páginas 29 y 30). 120
A.17.Fascículo de especies (páginas 31 y 32). 120
A.18.Fascículo de especies (páginas 33 y 34). 121
A.19.Fascículo de especies (contraportada y cubierta trasera). 121

Índice de tablas

3.1. Campos de la estructura que representa a un marcador. 40

4.1. Campos de la estructura que representa un objeto 3D. 65

5.1. Tipos de máscaras y sus usos. 81
5.2. Valores de las máscaras para filtrar el patrón. 82
5.3. Lista de los *kanji* empleados en la aplicación *Kanjirama*. 91

6.1. Parámetros capturados por el giroscopio. 102

Capítulo 1

Introducción

La realidad aumentada (referida en esta tesis simplemente como RA) es un término que se asocia al resultado del uso de tecnologías que superponen imágenes generadas por computadora sobre ambientes físicos pertenecientes al mundo real, que el usuario puede percibir de manera natural [1]. Este proceso se lleva a cabo en tiempo real, permitiendo a los usuarios interactuar con contenidos digitales a través de la manipulación de objetos reales o bien, tener una experiencia meramente visual al complementar el mundo real mediante objetos virtuales en un entorno de realidad mixto [2], creando la sensación de que el ambiente ha sido *aumentado*.

1.1. Diferencia entre RA y Realidad Virtual (RV)

Al igual que la RA, la RV es capaz de dar la impresión al usuario de que se encuentra en un ambiente diferente al que está acostumbrado, sin embargo, en el caso de la RV, dicho entorno es 100 % sintético [1], *i.e.*, todos los objetos son virtuales. Mientras el usuario se encuentre en dicho ambiente, no es capaz de ver el mundo real sino sólo el entorno virtual que lo rodea, generalmente desde una perspectiva en primera persona. Si el usuario es proyectado en el ambiente desde una perspectiva en tercera persona, también será mostrado como un objeto virtual.

Otra diferencia destacable es que la RV se auxilia de aditamentos generalmente de tipo visuales, tales como lentes, aunque también puede emplear otros medios para que la simulación parezca más real, ya sean audífonos, interfaces hápticas o inclusive trajes sensoriales, mientras que la RA tiene la opción de utilizar o no dichos aditamentos.

Además, cuando se realiza el proceso inverso al de la RA, *i.e.*, utilizar tecnologías para superponer imágenes del mundo real sobre entornos 100 % virtuales, estaremos refiriéndonos a ello como virtualidad aumentada (VA) [2].

Finalmente, una mezcolanza entre la realidad, la RA, la RV y la VA puede ser denominada realidad mixta (RM) [2].

1.2. Breve historia de la RA

A pesar de que las ideas, investigaciones y trabajos pioneros relacionados con la RA se encuentran ubicados entre las décadas de los 50 y 60, las tecnologías relacionadas con ella no rindieron frutos sino hasta más de 50 años después, a pesar de que aún no se ha logrado desarrollarlas al grado deseado.

Las primeras ideas de RA surgieron en 1950 gracias a Heilig [3], un cinematógrafo quien pensaba que el cine debía hacer sentir al observador que era parte de la escena, a través del uso de sus sentidos, por lo que desarrolló un prototipo de nombre *Sensorama* en 1955.

Posteriormente, Sutherland fue quien marcó los inicios de la RA, al diseñar un sistema auxiliado de un dispositivo óptico de despliegue de imágenes que se montaba en la cabeza y por el cual, muchas veces se asocia un mecanismo *HMD* (*Head-Mounted Display* por sus siglas en inglés) a la RA como elemento necesario.

En 1975, Krueger creó una sala que permitía al usuario la interacción con objetos virtuales denominada *Videoplace*, sin embargo el término *per se* fue acuñado por Caudell y Mizell. De igual forma (en ese mismo año), a pesar de que Rosenberg desarrolló el primer sistema de RA funcional (*Virtual Fixtures*), fueron Feiner, MacIntyre y Seligmann quienes presentaron el primer documento relacionado con el tema, mediante la propuesta de su sistema de nombre *KARMA*.

Posteriormente, en 1997, Azuma realizó el primer estudio serio en materia de RA [1] y proporcionó la definición más ampliamente conocida sobre el término. Unos años después (en el 2000), se desarrolló el primer juego de RA (*ARQuake*) que fue presentado en el simposio ISWC (*International Symposium on Wearable Computers*) de ese mismo año.

Con el desarrollo de las cámaras durante el nuevo milenio (específicamente en el 2005), se permitió comenzar el desarrollo de aplicaciones de RA para dispositivos móviles enfocadas en sistemas guía (2008). Además, la RA comenzó a utilizarse en el área de la medicina en el 2007. Hoy en día, el número de aplicaciones y dispositivos que soportan RA ha ido en aumento.

1.3. Motivación

Debido a que hoy en día, los dispositivos móviles tienen una gran influencia en nuestra vida diaria, al grado de volverse herramientas indispensables para los usuarios sin importar su edad, es destacable notar la flexibilidad que muchos modelos presentan en la realización de aplicaciones diversas, pese a sus limitaciones.

Uno de estos dispositivos es claramente el iPad, cuyas dimensiones lo vuelven muy práctico para las aplicaciones de RA, además de que la generación más reciente no tiene mucho tiempo de haber salido al mercado (incluyendo versiones miniatura que cuentan con buenos recursos).

Además, se rumora que pronto habrá una nueva versión, lo cual implica que poco a poco se realicen mejoras con respecto a sus entregas pasadas, al grado de que llegará un momento en que las limitaciones del iPad con respecto a un dispositivo no portátil se vuelvan despreciables, para el desarrollo de aplicaciones demandantes en recursos.

Por otra parte, las crecientes innovaciones realizadas en distintas áreas, así como la familiaridad que los usuarios tienen hacia la tecnología en el presente, provocan el desarrollo y la inclusión de nuevas y distintas formas de interacción. Tal es el caso de las técnicas de RA como parte de nuestras actividades, resaltándose la medicina, educación y sobre todo, la industria del entretenimiento. Por tal motivo, es importante aprovechar este hecho y modernizarnos para formar parte de un mercado creciente y competitivo.

Además, tomando en consideración que en el presente centro de investigación se ha tenido el interés de desarrollar aplicaciones y sistemas de RA, es interesante asimismo retomar estas investigaciones y desarrollos para generar aplicaciones (con un enfoque educativo) que sean portables. Sin embargo, se destaca que dichos trabajos deben ser optimizados, con la finalidad de que se ajusten a los requerimientos y recursos limitados (principalmente el poder de cálculo) de los que dispone el iPad.

1.4. Planteamiento del problema

Una aplicación de RA generalmente consta de cinco procesos fundamentales [4]:

1. Lectura de la cámara para obtener una imagen.
2. Estimación y seguimiento de la posición.
3. En algunos casos, comunicación entre dispositivos (aplicaciones multiusuario).
4. Optimización del cálculo para garantizar una interacción en tiempo real.
5. Despliegue de la escena virtual sobre la imagen obtenida de la cámara.

Cada uno de ellos depende del proceso anterior, *i.e.*, el seguimiento y detección de la posición requieren de una imagen extraída de la cámara, la aplicación no puede trabajar sin el proceso de seguimiento y finalmente los cálculos necesarios para el dibujo e inclusive las comunicaciones de red (en caso de ser una aplicación multiusuario) dependen del seguimiento y de los resultados de la posición de los marcadores obtenidos

por la aplicación, por lo que es necesario conocer los recursos con los que se cuenta para determinar cuáles son las prácticas más recomendables para optimizar el sistema.

Es muy importante en un sistema de RA contar con la transformación geométrica que relaciona el mundo virtual y el real. En la actualidad existen aplicaciones donde se sobrepone información textual sobre imágenes y aún así se le llama RA. En esta tesis se entiende que el término RA se debe aplicar en situaciones donde es necesario contar (y calcular) con la transformación geométrica entre el mundo real y el virtual, para poder llamarse RA.

1.4.1. Especificaciones de la cámara del iPad

Para poder realizar una aplicación de RA, es necesario conocer a fondo los dispositivos con los cuales se trabajará para desarrollar el proyecto. En el caso del iPad, cabe resaltar que las aplicaciones de RA no pudieron desarrollarse sino hasta la llegada del iPad 2, puesto que el primer modelo del iPad que salió a la venta no contaba con cámara alguna, un elemento indispensable para los sistemas de RA.

Con la salida del iPad 2 fue posible realizar las primeras aplicaciones de RA para iPad, sin embargo, desafortunadamente, la cámara no fue en lo absoluto el fuerte de este dispositivo, ya que tiene una resolución de 0.92 megapixeles (1280×720), aunque se dice que realmente es de 0.69 megapixeles (969×720). Sin embargo, el nuevo iPad (el de tercera generación) posee una cámara con una resolución de 5 megapixeles denominada *iSight* [5], aun cuando el iPhone 4S ya es capaz de brindar una resolución de hasta 8 megapixeles (3264×2448).

Otras características del iPad de tercera generación son [6]:

- Sensor de iluminación trasero (ideal para ambientes con baja iluminación).
- Apertura de f/2.4
- Lente de cinco elementos.
- Capacidad de grabación de video en alta definición a 1080 p (30 cuadros por segundo con sonido).
- Reducción de ruido.
- Detección automática de rostros.
- Balance automático de blancos.
- Autofoco.

Adicionalmente, este dispositivo incluye una cámara frontal VGA (denominada *FaceTime* por estar planeada principalmente para usarse con una aplicación del mismo nombre) que permite la grabación de video de hasta 30 cuadros por segundo. Desafortunadamente, la configuración de la cámara del iPad no puede modificarse.

En cuanto al aumento (*zoom*), la aplicación de la cámara provee cierta funcionalidad para ello, sin embargo, al tomar la fotografía, simplemente se ve como si se redujeran las dimensiones de la foto al área del acercamiento, no como un aumento propiamente dicho. Para realizar un verdadero aumento de 6x o inclusive 9x, se venden aplicaciones como *Camera Zoom 3* [7] e inclusive para modificar la apertura, se venden aplicaciones como *Aperture*.

Otra posibilidad de mejorar las fotografías tomadas por medio del iPad es con las lentes especializadas para el mismo [8], las cuales se venden por separado.

1.4.2. Módulo de gráficos

Una característica de vital importancia en una aplicación de RA es el módulo encargado de graficar los objetos 2D o 3D que se superpongan sobre la cámara, para ello, es necesario conocer qué alternativas ofrecen bibliotecas como *OpenGL*. Además, cuando se trata de dispositivos móviles, se requiere que este módulo sea un proceso optimizado, el cual varía en función del dispositivo.

OpenGL ofrece una versión optimizada para dispositivos embebidos, tales como consolas, teléfonos inteligentes e inclusive vehículos, ya que trata de reducir el uso de memoria. Dicha versión lleva por nombre *OpenGL ES* (*ES* hace referencia al inglés *Embedded Systems*, *i.e.*, sistemas embebidos) y es multiplataforma. En la versión 1.x era posible programar mediante operaciones de punto fijo y punto flotante, sin embargo, a partir de la versión 2.x, se dejó de dar soporte a la sección de punto fijo, siendo reemplazada por el uso de *shaders* (unidades de código escritas en lenguaje de sombreado, utilizadas principalmente para crear efectos como fuego, niebla, etc). El lenguaje de *shaders* es muy similar al código de C [9].

De acuerdo a Wagner y Schmalstieg [10], *OpenGL ES* 1.x puede ser ejecutado tanto en software como en hardware, sin embargo, de elegirse la segunda opción, los fondos se dibujan utilizando texturas, lo cual representa un problema puesto que colocar texturas es uno de los procesos más lentos en la parte de visualización, por lo que se recomienda cargar los fondos previamente en un búfer para no alterar el desempeño de la aplicación.

Otra consideración al momento de utilizar *OpenGL ES* es que dependiendo de las restricciones específicas de cada dispositivo, en ocasiones puede ser mejor emplear formatos de punto fijo o de punto flotante para los objetos de la escena, por ello,

se recomienda utilizar objetos con búfer para vértices, pues permiten convertir datos estáticos en cualquier formato que el dispositivo requiera internamente.

Finalmente, alguna práctica recomendada para aquellos que no dominan o no terminan de familiarizarse con los cambios significativos de esta versión de *OpenGL* con respecto a la de escritorio, ya sea por el lenguaje de sombreado o por otra causa (primitivas modificadas, definiciones sin bloques, etc.), es crear objetos con cualquier otro software de diseño y modelado de objetos 3D y utilizar formatos como *.obj* para la definición de los mismos y posteriormente transformarlos en archivos de cabecera *.h* para importarlos y utilizarlos mediante *OpenGL ES*.

1.5. Retos para los dispositivos móviles ante la RA

Hoy en día, en cuanto a RA se refiere, las *tablets* y los *smartphones* han representado una alternativa al uso de *HMDs* puesto que son livianos y están equipados inclusive con pantallas de buena resolución y cámaras de alta definición, además del hardware necesario (sensores) y un procesador poderoso en un solo sistema. Sin embargo, tres son los retos principales que hay que tomar en consideración [11]:

1. **Seguimiento:** uno de los retos más significativos en dispositivos móviles para la RA es el seguimiento de la posición y orientación de la cámara, en relación con los objetos relevantes de la escena. Si hablamos de un marco de referencia global, llamaremos a este proceso ubicación. Varios dispositivos móviles ya cuentan con GPS, brújula, acelerómetro y giroscopio, sin embargo se encuentran algo limitados, *e.g.*, en el caso del GPS, no actúa como un GPS especializado y no funciona en interiores.

   Por ello, una alternativa para determinar la ubicación y posición del dispositivo, es emplear técnicas de visión por computadora mediante el uso de patrones de referencia en 2D, que son comparados con los existentes en una base de datos o en memoria para que al coincidir se dibuje el objeto. Otros tipos de patrones son los de superficies, de los que se extraen puntos con ciertas características [11].

   De igual forma, se puede sugerir una aplicación cliente-servidor para que sea una computadora más potente la que realice el proceso fuerte de calibración de la escena, sin embargo se pierde portabilidad y se desaprovechan un poco los recursos con los que cuenta el dispositivo móvil. A causa de ello, generalmente se utiliza este enfoque cliente-servidor cuando se tienen bastantes patrones en la escena con los que la aplicación puede reaccionar (cientos o miles).

   Otra técnica a utilizar en estos casos es la reconstrucción “al vuelo” por medio del Mapeo y Ubicación Simultánea (*SLAM* [12] por sus siglas en inglés), el cual

es un proceso de inferencia recursivo y probabilístico utilizado para constantemente obtener la ubicación de un objeto, mientras se crea un mapa del ambiente que se está recorriendo.

Una combinación de las técnicas de visión por computadora con la información que provean los sensores del dispositivo es la opción adecuada en RA móvil, para crear aplicaciones impactantes. Alternativamente (según se describe en [13]) se pueden utilizar *frameworks* ya existentes para el desarrollo de las mismas.

2. **Interacción:** al realizar aplicaciones de RA en dispositivos móviles, es importante tomar en cuenta el tipo de interacción que se establece, ya sea: a) *interacción incorporada*, la cual se enfoca en la interacción del dispositivo con el objeto virtual (pantalla táctil, zoom, etc.), b) *interacción tangible*, la cual se basa en la manipulación directa de objetos en la escena por parte del usuario, c) *selección de rayos*, por medio de la cual se hace un mapeo entre los objetos virtuales de la escena y los pixeles de la pantalla del dispositivo o inclusive d) *capas de menús*, por medio de las cuales, mediante acciones tales como agitar el dispositivo o inclinarlo, se despliegan menús de interacción.

3. **Visualización:** la interacción entre objetos reales y virtuales conlleva a problemas de visualización, pues deben considerarse factores tales como el registro del objeto virtual, la oclusión (del patrón y entre objetos) y la colocación de sombras. El registro comprende el posicionamiento y orientación del objeto virtual en la escena, lo cual depende de la calibración. La oclusión refiere al buen manejo del posicionamiento de tal forma que, entre los objetos virtuales y reales, se visualicen sólo las partes que no sean obstruidas por otros objetos. Así mismo, la posición adecuada de las sombras no sólo depende de un mapeo estándar, sino que en ocasiones requiere información del proceso de calibración o directamente del usuario.

1.6. Objetivos del proyecto

El objetivo general de esta tesis es realizar un sistema de RA tomando como base los trabajos desarrollados por Vázquez del Ángel [14] y Serna Rodríguez [15]. Dicho sistema debe ser adaptado a las limitaciones de un dispositivo móvil, en este caso, un iPad. Para ello, dichos proyectos deben ser optimizados mediante el uso de distintas marcas de referencia y de la biblioteca optimizada para dispositivos móviles en cuanto a la graficación de objetos 3D. Así mismo, se pretende probar el sistema mediante el desarrollo de tres aplicaciones.

De igual forma, otros objetivos que se tienen contemplados son:

- Realizar aplicaciones enfocadas a educación, pues de esta manera resaltan su utilidad y obtienen un valor mayor.

- Programar tres aplicaciones que utilicen RA en el iPad ya que el tiempo designado para este proyecto se ajusta a esta cantidad.
- Evaluar las ventajas y desventajas de utilizar los sensores del iPad con el fin de saber si es adecuado emplearlos en aplicaciones de RA en este dispositivo.
- Evaluar la viabilidad de realizar deformaciones en los objetos virtuales de las aplicaciones desarrolladas para así realizar nuevas propuestas capaces de modelar algún problema diferente o emprender nuevos temas de investigación.

1.7. Organización de la tesis

La presente tesis está organizada en 7 capítulos:

- Desde luego el presente capítulo, donde a manera de introducción se dio la definición de RA y antecedentes de la misma. También se comentó sobre el planteamiento del problema en la presente tesis, la motivación, objetivos y los retos que implica realizar un sistema de RA en dispositivos móviles.
- En el capítulo 2 se definen los conceptos necesarios para elaborar el sistema y entender el trabajo desarrollado.
- En el capítulo 3 se describe el procedimiento que se realizó para desarrollar el sistema de RA, así como las modificaciones que se tuvieron que realizar sobre los sistemas tomados como base para adaptarlos al iPad.
- En el capítulo 4 se entra en detalle sobre todo lo relativo a la aplicación *Fascículo de especies* (la primera aplicación desarrollada para probar el presente trabajo de tesis) y sobre su realización.
- En el capítulo 5 se describen todas las consideraciones y métodos utilizados sobre los patrones de la aplicación para el aprendizaje del *kanji*, la segunda aplicación desarrollada para probar este trabajo de tesis.
- En el capítulo 6 se explica el procedimiento seguido para el desarrollo de la aplicación relativa al desplazamiento de una pelota a través de un laberinto, la última aplicación desarrollada.
- En el último capítulo se presentan las conclusiones de este trabajo de tesis y se describen algunas posibles sugerencias para continuar este proyecto.

Capítulo 2

Marco Teórico

Previo al desarrollo del sistema de RA, es necesario conocer algunos aspectos importantes (los cuales serán tratados a lo largo de este capítulo) como los tipos de RA (sección 2.1), las aplicaciones que suelen realizarse (sección 2.2), así como los trabajos relacionados con el tema (sección 2.3). Más concretamente debemos adentrarnos en el estudio de patrones de referencia (sección 2.4), de la arquitectura *iOS* (sección 2.5) y de *Objective-C* (sección 2.6), el lenguaje de programación empleado para realizar aplicaciones en el iPad.

2.1. Tipos de RA

Con base en lo descrito por Vázquez del Ángel [14] y Bimber [16], podemos realizar una clasificación de los sistemas de RA en dos tipos principales:

- **RA móvil**: este tipo de sistemas se desarrollan para dispositivos móviles y tienen la ventaja de que no requieren de aditamentos como cámaras, puesto que están integradas en el dispositivo. De igual forma, el procesamiento necesario se realiza internamente [11]. Un ejemplo de este tipo de RA se muestra en la Figura 2.1.

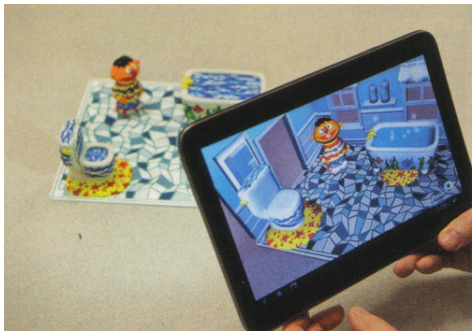


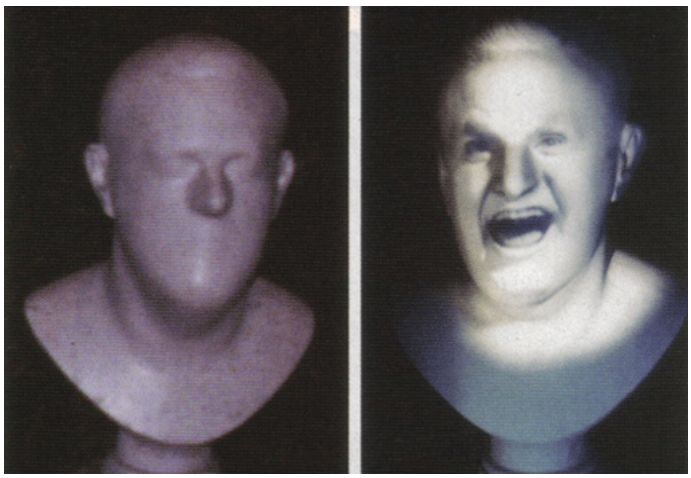
Figura 2.1: Ejemplo de RA en dispositivos móviles.

- **RA estacionaria**: también llamada RA espacial. Este tipo de RA se puede subdividir a su vez en tres tipos principales:

    1. *RA basada en proyecciones*: este tipo de RA no necesariamente se limita a superponer imágenes generadas por computadora, sino que puede inclusive sólo realizar cambios de luces sobre alguna escena para crear el efecto que la RA pretende. Utiliza proyectores como herramienta principal y está limitada al lugar donde se ubiquen los mismos [17].

       La Figura 2.2 muestra algunos ejemplos de este tipo de RA. Mientras que en 2.2(a) se muestra un ejemplo de todo un elaborado escenario, al que se le ha realizado una proyección y juego de iluminación para crear un efecto de RA, 2.2(b) refleja una demostración más directa y sencilla de este tipo de RA al aplicarse sobre un objeto, en este caso una figura decorativa.

(a) Proyecciones y luces en un escenario de un parque temático.



(b) Aplicación de RA basada en proyecciones sobre una figura decorativa (sin y con proyección).

Figura 2.2: Ejemplos de RA basada en proyecciones.

Realizar la calibración de instrumentos en este tipo de escenarios es más difícil, pues los proyectores pueden estar ubicados en lugares difíciles de maniobrar y la mala calibración de alguno de ellos puede provocar que el efecto se pierda. Se necesita un gran equipo de soporte técnico, aunque no

necesariamente especializado en temas de visión por computadora. Un uso frecuente de este tipo de RA se ve en los parques temáticos como *Disneylandia* ® [17].

2. *RA que utiliza aditamentos*: se auxilia del uso de dispositivos *HMD*, tales como lentes, en los cuales puede observarse el ambiente virtual combinado con el real, evitando así el uso de computadoras para proyectarlo [14].

3. *RA libre de aditamentos*: a pesar de ser menos portátil, no requiere utilizar dispositivos en el cuerpo y puede realizarse con cualquier computadora que cumpla los requisitos necesarios. A pesar de ello, demandará una cámara que capture la escena del mundo real [14].

Sea cual sea el tipo de RA que se emplee, cabe resaltar que ésta puede sobreponer gráficos en la parte externa de los objetos, *i.e.*, encima de ellos o bien mediante imágenes simular profundidad en objetos, tal como si se estuviese viendo en el interior de los mismos, como se ejemplifica en la Figura 2.3 (tomada del artículo citado en [11]).

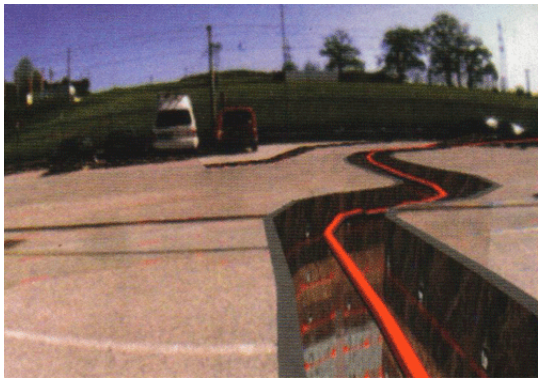


Figura 2.3: Ejemplo de RA que simula profundidad.

De igual modo, puede haber interacción directa con el usuario o éste puede ser sólo un espectador.

Cabe resaltar que la presente tesis se enfoca en el uso de RA en dispositivos móviles, en particular por medio del iPad.

2.2. Aplicaciones de la RA

La RA tiene aplicaciones bastante importantes en campos como los videojuegos, mercadeo, publicidad, educación, asistencia técnica y médica, construcción, mantenimiento, navegación, decoración, etc. Sin embargo, todas ellas podemos englobarlas en cuatro principales.

2.2.1. Medicina

En las salas de cirugía en países desarrollados, se utiliza la RA para superponer imágenes al momento de realizar biopsias y tomografías, permitiendo inclusive delimitar de manera más precisa el área en la que se encuentra un tumor o bien, simular un efecto de visión de rayos X, como se muestra en la Figura 2.4. Se auxilia de un dispositivo que cabe en la palma de la mano, el cual emite cierta radiación sobre el área del cuerpo que se desea analizar, mientras en tiempo real se superponen imágenes del tumor o del interior del cuerpo humano, según sea el caso.

Figura 2.4: RA en sala de cirugía.

Este tipo de RA muchas veces sufre de limitaciones de profundidad, sin embargo dichos problemas pueden solucionarse añadiendo más elementos de RV a la escena [18].

Utilizar RA durante estos procedimientos resulta bastante benéfico, puesto que la radiación emitida al cuerpo humano es menos dañina y las simulaciones de rayos X permiten realizar cirugías con el menor número de incisiones o inclusive puede evitarlas por completo, puesto que puede ayudar a refinar diagnósitcos.

Esta área ha demandado bastante el desarrollo de aplicaciones de RA a lo largo de los años, sin embargo el área donde más desarrollo existe sigue siendo la de entretenimiento.

2.2.2. Construcción, asistencia y mantenimiento

Las aplicaciones de este tipo se utilizan en la construcción para anteceder cómo se verá el prototipo de algún proyecto previo a su edificación. También se utilizan en el área de ensamblaje de piezas y asistencia técnica en reparaciones, pues se considera una buena práctica proveer un sistema de RA que modele las piezas del objeto a ensamblar (o reparar) con la finalidad de que el usuario pueda practicar hasta lograr realizar el procedimiento correctamente antes de utilizar las piezas originales, a pesar de que se cuente con manuales físicos [19].

Resulta muy útil en casos en los que no se pueda dar marcha atrás en los pasos (soldaduras, remaches, etc.). De igual forma, las aplicaciones de RA se utilizan para simular el interior de algún dispositivo y facilitar su mantenimiento. Algunas de ellas pueden considerarse como aplicaciones de aprendizaje y muchas veces son desarrolladas a manera de videojuegos. Las aplicaciones de navegación y decoración de interiores también son consideradas en este rubro.

2.2.3. Educación

En las aulas, la RA trae grandes beneficios en particular a los niños, puesto que les permite entender fenómenos físicos y químicos por medio de experiencias interactivas y visuales más enriquecedoras, generalmente presentadas a manera de juegos. Además, estudios demuestran que los niños que aprenden temas de alta dificultad por medio del uso de técnicas de RA y aplicaciones colaborativas, retienen más la información que aquellos niños que estudian las mismas temáticas por medios tradicionales tales como la lectura, las clases presenciales y las imágenes estáticas.

La mayoría de estas aplicaciones se presentan a los niños por medio de libros *aumentados*, junto con los softwares necesarios para la completa utilización del sistema [20]. Ejemplos de ello se muestran en la Figura 2.5. Por si fuera poco, la experiencia se facilita dado el hecho de que, cada vez más, el uso e interacción con las tecnologías de la información se presenta a más temprana edad en los niños.

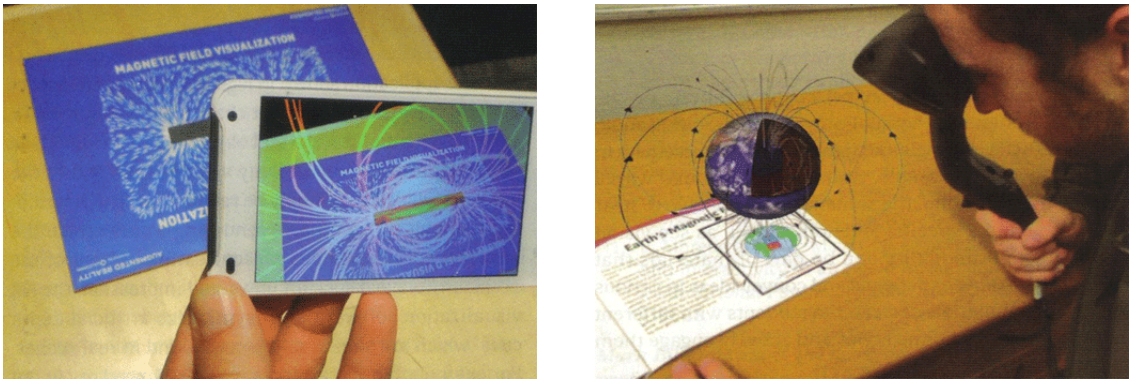


Figura 2.5: RA en las salas escolares auxiliada por libros.

Otro ejemplo de este tipo de aplicaciones son aquellas para enseñar a estudiantes de medicina sobre el cuerpo humano por medio de RA, en donde éstos ven imágenes generadas (con esencia de profundidad) relativas al interior del cuerpo humano (huesos, músculos, órganos, etc.) cada vez que el usuario pasa su mano a través de su cuerpo. Se auxilia del dispositivo *Kinect*™ de Microsoft ® [18].

Fuera de las escuelas, la RA con fines educativos puede verse en áreas públicas tales como museos, en los cuales las aplicaciones ya no están dirigidas meramente a niños sino al público en general, mediante experiencias interactivas con filmes, modelos virtuales y dispositivos táctiles ubicados en salas que, a pesar de que a simple vista se perciban vacías, revelan su contenido si se interactúa con ellas mediante el uso de tecnologías.

En dichos lugares se dispone de equipo tal como dispositivos táctiles empotrados sobre bases fijas, sensores, cámaras, etc., y funcionan con base en imágenes que sirven

como marcos de referencia (patrones) para calibrar las cámaras y poder graficar los objetos. Es muy útil sobre todo para informar al público acerca de especies extintas o fenómenos y lugares a los cuales es difícil tener acceso (un dinosaurio [21], el interior del núcleo de la Tierra, etc.).

La Figura 2.6 ejemplifica el uso de RA en el Museo de Historia Natural de Londrés.

Figura 2.6: Ejemplo de RA en sitios públicos.

2.2.4. Entretenimiento y publicidad

Una de las áreas que sin duda ha sobresalido y se ha visto beneficiada es la del entretenimiento y la publicidad, pues se han desarrollado varias aplicaciones de RA con este propósito. Sin duda, auxiliarse de tecnologías como RA puede llevar a la venta de un nuevo producto mediante una campaña publicitaria exitosa, agregando aplicaciones de RA en revistas (Figura 2.7), carteles, espectaculares, juguetes e inclusive prendas de vestir (Figura 2.8) [16]. También se utilizan para promocionar películas y videojuegos. Este tipo de aplicaciones pueden ser sólo visuales o interactivas y son presentadas generalmente a manera de juegos.

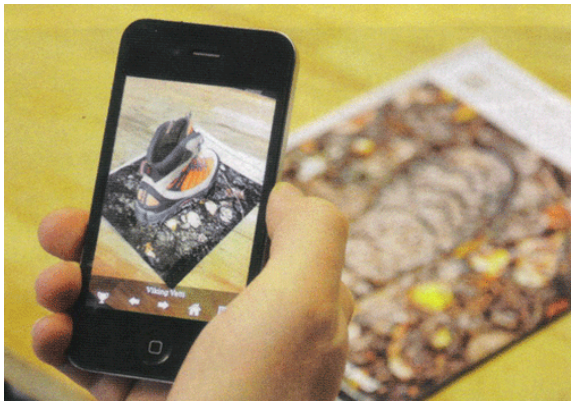


Figura 2.7: RA en revistas.

Funcionan comúnmente por medio de marcas de referencia (patrones) o a través de códigos QR que activan la aplicación o permiten descargarla.

En el área de los videojuegos, los dispositivos portátiles están sobresaliendo en la actual generación de consolas, mediante la portátil de SONY (PSVITA) y el Nintendo 3DS™. Como se ha mencionado anteriormente, la mayoría de las aplicaciones de RA se auxilian de patrones de referencia y estos dispositivos no son la excepción, pues los juegos enfocados en RA incluyen tarjetas de RA (*AR Cards* por sus siglas en inglés)

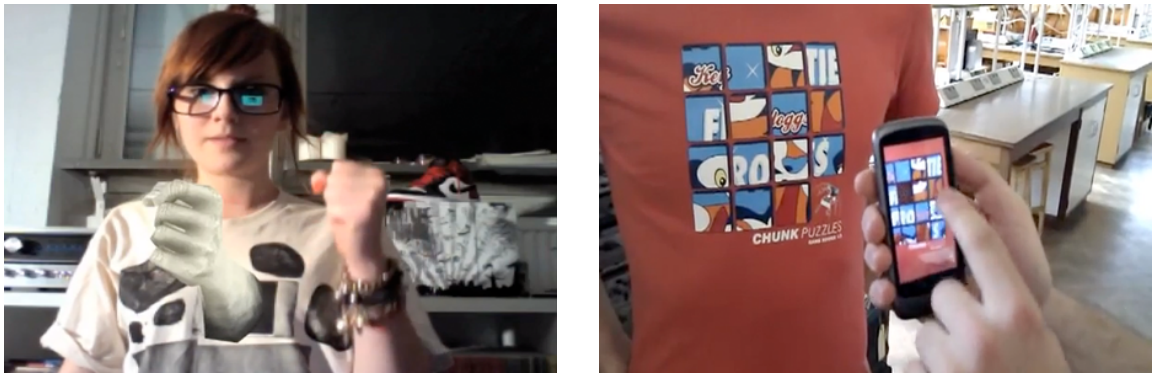


Figura 2.8: Ejemplos de aplicaciones de RA en prendas de vestir.

o libros con patrones (Figura 2.9(a)) que proyectan algún objeto sobre la vista previa de la cámara del dispositivo una vez que la tarjeta o libro son detectados, como se muestra en la Figura 2.9(b). Sin embargo, si estos aditamentos se encuentran sucios o doblados, difícilmente son reconocidos por el sistema, además de que las luces del lugar pueden interferir con la cámara en la detección de los mismos. De igual forma, puede que interactúen con el entorno deformándolo.

Nintendo 3DS™ además, se auxilia de su vista 3D por medio de la cual añade profundidad a la escena, dando la sensación de que existe una inmersión en un ambiente 3D que se combina con el mundo real, pues aquellos juegos que no incluyen aditamentos de RA, proponen modos de juego más interactivos con la cámara del dispositivo, superponiendo imágenes sobre la vista previa de la misma independientemente del lugar donde se encuentre [22].

(a) Aditamentos para RA en Nintendo 3DS™. (b) Nintendo 3DS™ interactuando con las AR Cards.

Figura 2.9: Uso de RA en Nintendo 3DS™.

2.3. Trabajos relacionados

Cuando pensamos en proyectos sobre RA que puedan servir como referencia para el desarrollo de un proyecto propio, más que pensar en aplicaciones similares o en conceptos innovadores que resuelvan problemáticas como detectar la posición del patrón, el seguimiento de un objeto o la captura de la imagen desde la cámara, es más factible comenzar la búsqueda por aquellas herramientas que proporcionen un entorno en el cual aplicar la RA sea más sencillo, ya sea que se trate de simples bibliotecas, *APIs*, *SDKs* o inclusive *frameworks*.

Por ello, a continuación se citan algunos de estos elementos, enfocándonos solamente en aquellos que puedan ser utilizados en dispositivos móviles, a pesar de que tal vez no todos estén centrados en *iOS*, el sistema operativo que utiliza el iPad, dispositivo que será utilizado en este proyecto. Esto con la finalidad de que cualquier otro desarrollador que tenga la intención de trabajar RA en dispositivos móviles, pueda tomar este documento como referencia para informarse sobre el entorno más adecuado que se ajuste a sus necesidades, independientemente del sistema operativo del dispositivo móvil que vaya a utilizar.

- **ARToolKitPlus [23]:** nació a partir de *ARToolKit*, una biblioteca para RA desarrollada por Hirokazu Kato en 1999 en el Instituto de Ciencia y Tecnología de Nara, Japón. Se considera una biblioteca de visión por computadora para seguimiento, que utiliza las habilidades de rastreo de video para calcular la posición real de la cámara y su orientación, en relación con un marcador o patrón de referencia.

  ARToolKitPlus es una extensión de *ARToolKit* que permite migrar todos los beneficios de ésta hacia los dispositivos móviles, inclusive con mejoras en algunos aspectos. Es muy popular, tiene soporte para varias plataformas y es de código abierto, aunque en la actualidad ya no sigue en desarrollo. Sin embargo, no es capaz de leer imágenes desde la cámara ni proporciona módulos para la graficación de objetos, sólo auxilia al programador en la tarea más difícil de la RA, el seguimiento y detección de la cámara, *i.e.*, las técnicas de visión por computadora. Los módulos para la lectura de la imagen y graficación de objetos deben ser implementados por separado.

- **Vuforia [24]:** un *SDK* de la mano de *Qualcomm*, industria más conocida por la fabricación de *chipsets* para dispositivos móviles. *Vuforia* está demostrando sus potencialidades sobre RA en dispositivos móviles, utiliza algoritmos de visión por computadora capaces de reconocer objetos 2D y 3D. Es multiplataforma y cuenta con una buena documentación tanto en *Android* como en *iOS*, aunque la primera está más completa que la segunda. Puede actuar en conjunto con *Unity 3D* para facilitar el desarrollo de aplicaciones y es código abierto.

Gervautz y Schmalstieg [11] muestran algunas aplicaciones desarrolladas con *Vuforia* (aunque no se indica explícitamente), además, la revista citada en [16] provee una aplicación de RA para interactuar con ella, la cual fue desarrollada con *Vuforia*.

Sobre su desempeño, podemos decir que es capaz de utilizar patrones enmarcados o imágenes cualesquiera, además de objetos 3D simples para realizar el seguimiento, es flexible en cuanto a la creación de patrones, soporta varios marcadores en la escena (no más de 5) y funciona si el marcador es parcialmente visible. Además, es más estable que otros *SDK*, sin embargo, para realizar las pruebas, sólo se puede hacer directamente por medio del dispositivo, *i.e.*, no se puede probar con cámaras auxiliares conectadas a la computadora.

- **String [25]:** un *SDK* exclusivo para *iOS* con el cual es posible desarrollar aplicaciones de RA bastante impresionantes. Reconoce imágenes como patrones, pero éstos deben tener un marco, lo cual lo vuelve poco flexible ya que casi todo el marco debe estar visible, sin embargo entiende dónde se ubican dichos patrones en un espacio 3D. Promete ser el más fácil y rápido para el desarrollo de aplicaciones de RA, pues requiere pocas líneas de código para funcionar y detecta los objetos en la escena rápidamente, facilitando el seguimiento de los mismos.

  Puede ser utilizado en conjunto con *OpenGL* y *Unity 3D*, con los cuales inclusive se puede facilitar la iluminación de objetos para hacer la escena más realista. También facilita la creación de interfaces al usarlo en conjunto con *Cocoa Touch*. Permite actuar con hasta 10 imágenes de referencia simultáneamente y si se utiliza una cámara, pueden realizarse pruebas de escritorio sin necesidad de migrar la aplicación al dispositivo. Desafortunadamente, se requiere una costosa licencia para su uso.

- **NyARToolkit [26]:** es una biblioteca visual de RA derivada de *ARToolKit*. Tiene soporte para distintos formatos de imagen y realiza el procesamiento de ajuste de una manera rápida, más veloz inclusive que el mismo *ARToolKit*. Existen diversas variantes de esta biblioteca, ya sea para plugins de *Flash*, *Silverlight*, etc., o para lenguajes como C++, AS3 y C#, por lo que en dichos casos el nombre de la biblioteca cambia un poco. Inicialmente fue desarrollada para Java, lo cual facilitaría el uso de ésta en aplicaciones para *Android*.

- **ARTag [27]:** al igual que la anterior, está basada en *ARToolKit*, pero a diferencia de ésta, provee módulos para auxiliar en la captura de video y el despliegue de objetos en el entorno. La RA puede realizarse de dos maneras, por medio de dispositivos móviles de manera individualizada o por medio de un sistema de una sola cámara colocada a distancia y una pantalla grande, donde los usuarios vean reflejados los objetos 3D.

- **Studierstube ES [28]:** la familia de *Studierstube* provee bibliotecas y *frameworks* para el desarrollo de aplicaciones de RA, es considerado un sucesor de *ARToolKit* y *ARToolKitPlus*. Auxilia en la detección y estimación de la pose de los marcos de referencia 2D. Sus conceptos son similares a los de *ARToolKit*, *ARToolKitPlus* y *ARTag*, pero su código y modo de uso son diferentes. Los requisitos de procesamiento son pocos y la biblioteca es rápida al procesar las imágenes. *Studierstube ES* es la versión de *Studierstube* especializada en dispositivos móviles y es multiplataforma, sin embargo, para *iOS* aún se encuentra en fase experimental.

- **FLARToolKit [29]:** se trata de una adaptación de *NyARToolKit* hacia *ActionScript*, pero es importante mencionarla ya que se ha vuelto muy popular sobre todo en esta versión, puesto que permite aplicaciones de RA vía web. Fue desarrollada por Tomohiko Koyama y cuenta con una licencia mixta.

  Posee la característica de que los objetos 3D son descritos mediante archivos XML para facilitar al sistema web el entendimiento de los mismos. Para generar dichos objetos, se pueden utilizar herramientas como *Flash* (de ahí el nombre de *FLARToolKit*) mediante la biblioteca *Papervision 3D*, *Blender*, *COLLADA/DAE*, etc., siempre que se exporte el gráfico como objeto de *COLLADA*, pues éste genera un archivo de extensión *.dae* (*digital asset exchange*) que describe el contenido digital. En cuanto a la forma de usar los patrones de referencia, lo hace de un modo similar a *ARToolKit*.

- **Popcode [30]:** es una aplicación, pero a la vez presenta un *SDK* mediante el cual es posible añadir contenido de RA a una imagen, sin la necesidad de tener patrones en blanco y negro como con muchas otras bibliotecas. Es gratuito, pero ya no sigue en desarrollo, pues el proyecto ha evolucionado como *Zappar*. A partir de las últimas versiones se dio soporte para *iOS*.

- **Nestor [31]:** es un sistema de reconocimiento y seguimiento de la posición en 3D de figuras planas. Su objetivo es brindar una solución para aplicaciones de RA que permiten crear figuras aumentadas con contenido virtual. Se caracteriza principalmente por permitir una libertad total de las figuras a reconocer, aun cuando éstas tengan varias curvas. A pesar de que las formas a reconocer sean planas, una vez identificadas, la superficie no se limita a estar 100 % plana, *i.e.*, puede curvearse un poco y el objeto virtual aún se encuentra en seguimiento.

- **Layar [32]:** es una plataforma de desarrollo para aplicaciones de RA que se enfoca en el reconocimiento de imágenes y la inclusión de contenidos digitales simples como vídeos, imágenes, vínculos a sitios web, etc. Está disponible para *iOS* y *Android*, pero su versión gratuita está soportada a base de anuncios publicitarios.

- **Wikitude [33]:** es un navegador que provee RA e interactúa con diversos elementos del mundo real. Es multiplataforma y además provee un *SDK* para desarrollar aplicaciones propias.

- **VRToolKit [34]:** es una aplicación que ayuda en el desarrollo de proyectos de RA. Realiza la detección de patrones por medio de una serie de funciones envueltas (hechas en *Objective-C*) que llaman a *ARToolKitPlus*. Para cargar los objetos 3D se auxilia de archivos *.xml* y *.h* e inclusive *.obj*, pero para estos últimos tiene una funcionalidad muy lenta. Para desarrollar aplicaciones se requiere: *OpenGL ES*, *QuartzCore*, *AVFoundation*, *CoreVideo*, *MessageUI*, *CoreMedia* y *libxml2*.

- **ARWin [35]:** es una aplicación que pretende (por medio de marcadores y patrones de referencia) convertir un entorno laboral en un ambiente de RA.

- **3DAR [36]:** un *SDK* enfocado en sobreponer marcadores a mapas y a la vista previa de la cámara para enfatizar sitios de interés. Éste es su enfoque de RA.

- **AR23D [37]:** la empresa creadora de este *SDK* tiene una postura más enfocada en desarrollar y vender aplicaciones que resuelvan las necesidades de sus clientes, sin embargo sí proveen un *SDK* para desarrolladores.

- **Metaio [38]:** otro *SDK* para realizar aplicaciones de RA en *iOS* y *Android*. Sin embargo, la versión gratuita añade marcas de agua a la aplicación.

- **D'Fusion [39]:** trabaja en conjunto con *Unity 3D* para crear aplicaciones completas de RA.

- **Droidar [40]:** se trata de un *framework* exclusivo para *Android* que permite realizar RA con y sin patrones de referencia.

- **Cocos2D [41]:** esta biblioteca se especializa en gráficos 2D y se utiliza primordialmente para realizar videojuegos 2D en dispositivos móviles. Sin embargo, propone la posibilidad de realizar aplicaciones de RA sencillas, pero sólo superponen imágenes 2D en la vista previa de la cámara, por lo que las aplicaciones que se pueden lograr son muy básicas y no pueden realizar los efectos de profundidad requeridos por la RA.

Se han citado los principales entornos para desarrollar RA, sin embargo a veces se puede prescindir de ellos y sólo tomarlos como guía para saber resolver problemas con código propio. En cuanto a las aplicaciones, actualmente existen varias de ellas que utilizan RA en diversas formas, desde las más sencillas hasta algunas bastante elaboradas y con efectos muy impresionantes. Sin embargo, las aplicaciones que más se apegan a lo que se pretende realizar en la presente tesis, son las descritas en los trabajos de Serna Rodríguez [15] y el de Wagner y Barakonyi [42]. Se entrará en detalle sobre las mismas en secciones posteriores.

2.4. Tipos de patrones

Los patrones o marcadores (en inglés *fiducial markers*) son símbolos específicos diseñados para ser fácilmente reconocidos por las máquinas [13]. Las técnicas de detección de la posición y seguimiento de la cámara basadas en marcadores, son comúnmente utilizadas en interiores y más aún en los dispositivos móviles, donde el GPS no posee un buen desempeño, según comentan Gervautz y Schmalstieg [11]. Los marcadores se usan porque emplean algoritmos demasiado simples para reconocerlos, los cuales no necesitan de un alto procesamiento ni de sensores especializados.

Los marcadores más comunes que podemos encontrar son los siguientes:

2.4.1. Marcadores de plantilla

Unos de los primeros tipos de marcadores de referencia, son básicamente cuadrados con un marco blanco o negro y un fondo contrastante del color opuesto, los cuales pueden tener en su interior cualquier dibujo [13]. El cuadrado completo y el patrón de su interior son reconocidos como marcadores. Un ejemplo de este tipo de marcador se muestra en la Figura 2.10, es uno de los marcadores más utilizados, con una abreviatura del nombre del creador de *ARToolKit*, pues esta herramienta puede utilizar este tipo de marcadores.

Figura 2.10: Ejemplo de marcador de plantilla.

Sin embargo, sufren de un pequeño inconveniente, cuando el marcador es detectado por primera vez, su patrón es extraído y correlacionado con todos los patrones conocidos, así, conforme más marcadores y patrones se utilicen en la escena, la aplicación se vuelve cada vez más lenta. Además, a parte de que los patrones deben ser diseñados, es necesario entrenar al algoritmo de detección para poder utilizarlos posteriormente. Finalmente, la complejidad del patrón afecta la eficiencia del proceso, por ello, los marcadores con regiones grandes de blanco o de negro son preferibles.

2.4.2. Códigos de barras uni y bidimensionales

Figura 2.11: Ejemplo de código de barras 1D.

Los códigos de barras (Figura 2.11) se volvieron muy importantes desde la década de los 80 a pesar de que fueron inventados muchos años atrás, ya que podían almacenar cierta información que no era descifrable a simple vista y mediante la cual era posible identificar algún elemento de manera única [43]. Raramente son utilizados en las aplicaciones de RA, mas que para aplicaciones sencillas de etiquetado.

Posteriormente, estos códigos unidimensionales (1D) tuvieron una evolución que devino en los códigos bidimensionales QR, los cuales son códigos de respuesta rápida (*Quick Response* por sus siglas en inglés) [13]. Son fáciles de detectar, *i.e.*, no por humanos, sino por las máquinas, detección que puede realizarse con un costo de procesamiento muy bajo, además de que al estar en dos dimensiones pueden almacenar mucho más información, como si se tratase de una matriz de datos (información en forma horizontal y vertical). Se ha vuelto muy popular el uso de este tipo de códigos en las aplicaciones de RA. Un ejemplo de código QR se muestra en la Figura 2.12.

Figura 2.12: Ejemplo de código de barras 2D (QR).

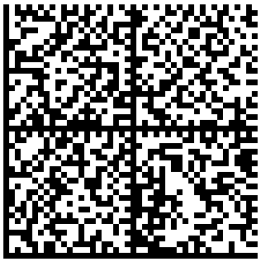


Figura 2.13: Ejemplo de código de matriz de datos.

Existen otro tipo de códigos bidimensionales, los cuales ya son propiamente conocidos con el nombre de "matrices de datos" (como el mostrado en la Figura 2.13), los cuales pueden almacenar mucho más información, alcanzando hasta máximo 2KB [13]. Sin embargo, en los ambientes donde no se requiere decodificar tanta información, es preferible utilizar marcadores más simples.

En el caso particular de los códigos QR, existen tres formas principales de utilizarlos en una aplicación de RA:

- Contenerlos dentro de un marco negro similar a los de las marcas de referencia, para que de esta manera, la estimación de la posición de la cámara se realice de la misma forma que con una marca de referencia cualquiera y el QR sirva sólo como identificador único de contenidos digitales.
- Anexando en la parte inferior de la hoja, una pequeña marca con forma simple para identificar el origen del sistema de coordenadas para dibujar el objeto virtual y utilizar el QR como identificador de contenidos.
- Utilizar el QR tanto para estimar la posición de la cámara como para identificar contenidos [44]. En estos casos, la imagen se filtra varias veces para obtener el contorno de los tres cuadrados más grandes ubicados en las esquinas de un código QR común y poder determinar a su vez, tres de las cuatro esquinas que poseería una marca de referencia característica de los sistemas de RA. Asimismo, se forma un triángulo entre los centros de los tres cuadrados previamente encontrados. Dicho triángulo permite (por medio de distancias y puntos centrales) estimar la cuarta esquina que tendría la marca de referencia y así, con estos puntos calcular la matriz de homografía. Además, para evitar problemas de oclusión temporal, se puede añadir un filtro *Kalman* [45].

2.4.3. Marcadores de referencia

Además de los códigos antes mencionados, existen otro tipo de patrones conocidos como marcadores de referencia (Figura 2.14), los cuales pueden ser utilizados en un ambiente consciente de identificadores, en los cuales el número ID está codificado en bits dentro del marcador. Existen muchos beneficios de utilizar este tipo de marcas, para empezar, la detección de este tipo de marcadores es más rápida que la de plantillas, pues no se requieren coincidencias con otras imágenes.

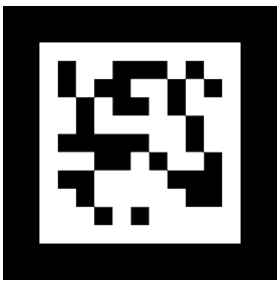


Figura 2.14: Ejemplo de marcador de referencia.

En segunda, el usuario no debe proveer al sistema con imágenes, sino que puede elegir cualquier marcador de entre un conjunto de patrones fijo. Finalmente, no es necesario realizar un entrenamiento del algoritmo sobre los marcadores, ya que cada marcador válido implícitamente es conocido por el sistema. Sin embargo, la mayoría de las herramientas que utilizan este tipo de marcadores no son gratuitas.

2.4.4. Marcas topológicas

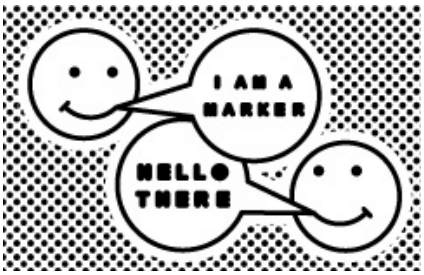


Figura 2.15: Ejemplo de marca topológica.

Las marcas topológicas (Figura 2.15) son una clase de marcadores reconocidos con base en características específicas del marcador, *i.e.*, se extraen ciertos puntos de interés que determinan curvas o líneas, pero no existe una forma específica, sino que más bien la guía de estos marcadores es la relación existente entre las zonas negras y blancas. Esto provoca que el diseño sea menos restringido, al grado de que es posible colocar figuras que tengan un significado para los seres humanos, pero deben ser únicos, lo cual provoca un esfuerzo de procesamiento adicional y costo en tiempo, reduciendo la escalabilidad de la aplicación.

2.5. Arquitectura de *iOS*

La arquitectura de *iOS* está basada en capas, donde las más altas comprenden los servicios y tecnologías más importantes para el desarrollo de aplicaciones, mientras que las más bajas se encargan de controlar los servicios básicos [46].

Las capas que conforman la arquitectura de *iOS* se describen a continuación:

- **Cocoa Touch:** es la capa más importante, puesto que permite el desarrollo de aplicaciones en *iOS*. Posee un conjunto de *Frameworks* que proporcionan la *API* de Cocoa, además, permite acceder a los sensores y controles táctiles por

medio de eventos empleando la jerarquía de vistas. Entre los *frameworks* más destacables se encuentra *UIKit* (para desarrollo de interfaces de ususario).

- **Media:** permite la ejecución de multimedia (audio, *OpenGL*, imágenes, PDF, etc.) y provee estos servicios a la capa superior.
- **Core Services:** comprende todos los servicios fundamentales del sistema, los cuales pueden ser empleados por cualquier aplicación. Ejemplos de este tipo de servicios son: la agenda, bases de datos, preferencias del sistema, conexiones de red, etc. Además, incluye al *framework Foundation* (especial en el acceso y manejo de objetos).
- **Core OS:** contiene todas aquellas características de bajo nivel. Gestiona la memoria, el sistema de archivos, seguridad, interacción con el hardware, etc.

Un diagrama con las capas de esta arquitectura se muestran en la Figura 2.16.

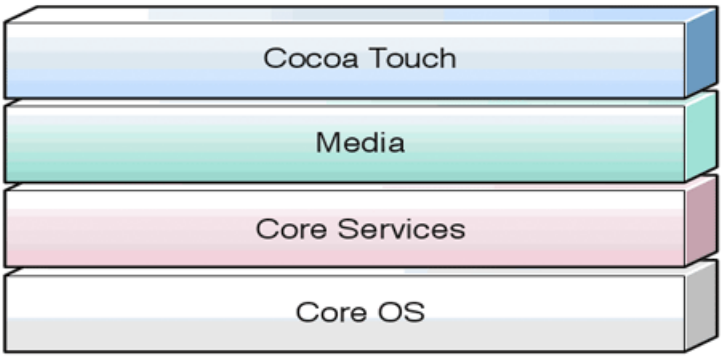


Figura 2.16: Arquitectura *iOS*.

2.6. Rasgos característicos de *Objective-C*

Para programar en *Objective-C* (el lenguaje de programación para la plataforma *iOS*), es necesario conocer ciertos detalles sobre el entorno con el que se trabaja, entre ellos se enlistan:

- **Entorno de desarrollo:** comúnmente se utiliza el *IDE Xcode* para el desarrollo de aplicaciones en *Objective-C*.
- **Archivos comunes:** *.plist* (descriptores similares a los *.xml*), archivos de cabecera (*.h*), implementación de las clases definidas en un archivo de cabecera (*.m*), archivos de cabecera precompilados (*.pch*) y para la descripción de interfaces de usuario, se utilizan archivos *.storyboard* o *.xib*, según la versión de *Xcode* que se utilice.
- **Clases especializadas:** al programar en *Objective-C* se puede hacer uso de clases especializadas para la realización de un proceso. Dichas clases son conocidas como *delegados*.

- **Instanciación de objetos:** la palabra reservada para realizar esta labor es `self`, la cual es similar a `this` (empleada en otros lenguajes de programación orientados a objetos) en cuanto a su forma de uso.
- **Llamada a un método:** para acceder a los métodos propios de una clase, se hace uso de corchetes, dentro de los cuales se escriben dos parámetros (primero el objeto y después el nombre del método).
- **Cadenas:** las cadenas inician con @.
- **Acceso a atributos:** para generar automáticamente los métodos de acceso a los atributos de una clase, se utiliza `@property` y para poder utilizar las propiedades declaradas en el archivo de cabecera, se emplea `@synthesize`. Dichas propiedades deben ser inicializadas en el archivo de implementación.

2.6.1. Manejo de memoria en *Objective-C*

Al trabajar con *Objective-C*, los errores más frecuentes se centran en el manejo de memoria. Si se está acostumbrado a lenguajes de programación que se encargan de manejar la memoria automáticamente (como *Java* o *C#*), entonces trabajar con este lenguaje será algo complicado a menos que se entiendan algunos conceptos básicos.

Los objetos en *Objective-C* son referenciados por medio de un contador, una vez creados. Esto significa que cada objeto mantiene un seguimiento de todos los demás objetos que apuntan hacia él [47]. Cuando el contador de referencia llega a 0, la memoria reservada para dicho objeto puede ser liberada sin problema alguno.

El programador es quien debe asegurarse de conocer el valor del contador de referencia en todo momento, durante la ejecución del programa. Cuando se crea un apuntador a un objeto en alguna parte del código (como al instanciar una variable), el contador debe ser incrementado; y cuando la variable no se necesite más, dicho contador debe ser decrementado.

Al crear un objeto en *Objective-C* se utiliza el método `alloc` para reservar espacio de memoria para dicho objeto. Posteriormente, se llama al método `init` para inicializar el objeto y si este método no requiere parámetro alguno para la inicialización, se puede hacer uso de la palabra reservada `new` al momento de crear el objeto (como en muchos otros lenguajes de programación orientados a objetos) en lugar de utilizar la combinación de `alloc` seguida de `init`.

Sea cual sea el caso, al crearse un objeto de cualquiera de estas dos formas, el contador de referencia del objeto tendrá un valor de 1; y deberá ser decrementado en algún momento en el que el objeto no se necesite más. El método encargado de decrementar este contador es `release`, pero debe ser llamado explícitamente.

El problema realmente es saber cuándo debe realizarse la llamada a `release`. La respuesta es simple: cuando el objeto ya no vaya a necesitarse más. Sin embargo, saber cuándo es el momento apropiado puede ser difícil para algunos programadores.

Se recomienda realizar esta llamada en el método `dealloc`, el cual está presente en cualquier archivo de implementación de una clase en *Objective-C* y su comportamiento puede ser modificado. Pero si el objeto fue inicializado dentro del método *viewDidLoad*, entonces se recomienda llamar a `release` en el método *viewDidUnload*.

Como sea, después de un `release` es necesario establecer un valor nulo (palabra reservada `nil`) al objeto, pues una llamada a un método de un objeto nulo es ignorada. En cambio, si dicho objeto no fue asignado a un valor nulo y el método perteneciente a dicho objeto liberado es llamado, entonces el programa presentará una especie de fallo de segmentación.

Alternativamente, al crear objetos se puede utilizar el método `autorelease`, el cual indica que deseamos que el objeto sea liberado en algún punto del código (*e.g.*, en la siguiente iteración de un ciclo), pero por el momento deseamos utilizarlo.

Cuando se utiliza `autorelease`, comúnmente no se hace ningún llamado a `release`. Sin embargo si se desea almacenar la variable en algún fragmento del código, utilizar `autorelease` no es lo más conveniente. Por lo que, cuando se trabaja con un enfoque orientado a eventos como cuando se utilizan botones que interactúan con las propiedades de algún objeto, quizás `autorelease` no sea la mejor opción para las variables.

Algunos métodos de las clases por defecto de *Objective-C* implícitamente utilizan `autorelease`, para saberlo basta con saber que, si el nombre del método comienza con `init` o `copy`, entonces el objeto que devuelve no llamará en ningún momento a `autorelease` y será el programador el que deberá hacer el llamado a `release` en el momento pertinente.

Ahora bien, si la variable que se está creando pertenece a una clase por defecto de *Objective-C* que implementa `autorelease`; y se requiere que dicho objeto sea almacenado para su posterior uso, *i.e.*, que su valor no se pierda, entonces se puede hacer uso del método `retain`.

Este método se encarga de incrementar el contador de referencia para dicha variable, por lo que cuando `autorelease` sea llamado en algún momento de la ejecución (que no se pueda controlar explícitamente), el contador del objeto será decrementado pero su valor no será 0, puesto que `retain` lo habría incrementado previamente. Sin embargo, debe tenerse en cuenta que utilizar `retain` implica que se deba hacer algún llamado a `release` en algún momento para que el objeto no permanezca en memoria, lo cual generalmente se lleva a cabo en el método `dealloc`.

En general, estas son las consideraciones que deben tomarse en cuenta para gestionar la memoria al programar en el lenguaje *Objective-C*.

2.6.2. Mezclando *Objective-C* y *OpenGL ES*

Como se mencionó en la sección 1.4.2 (p. 5) existen distintas versiones de *OpenGL ES*, además de que la curva de aprendizaje de éste se ve algo afectada debido a los cambios representativos con respecto a la versión de *OpenGL* de escritorio, *e.g.*, se mencionó que algunas primitivas habían sido modificadas y en general la forma de realizar algunos procesos, tan sólo dibujar; comúnmente en *OpenGL* se hace mediante una función en la cual se colocan todas las primitivas que se deseen utilizar y se pinta todo lo especificado al terminar la función, mientras que en *OpenGL ES* se debe llamar a la función *glDrawArrays* o *glDrawElements* cada vez que se desee dibujar un objeto diferente, pues en esta función se definen los vértices a dibujar y las primitivas encontradas entre cada llamada a alguna de estas funciones se aplica al último objeto que se dibuja.

En *OpenGL ES* se pueden especificar arreglos de vértices indicando las posiciones de cada uno en el espacio virtual de *OpenGL ES* y al realizar la llamada a alguna de las funciones para pintar, se indica el arreglo de vértices a dibujar y sobre el cual se aplicarán las primitivas indicadas en todas las líneas previas a la llamada. Esto no ocurre en la versión de escritorio puesto que se emplean las definiciones por bloques, *i.e.*, comúnmente en *OpenGL* se acostumbra declarar elementos que serán dibujados de una misma manera (mismo tipo de línea) por medio de bloques (donde se especifica la posición de cada vértice), los cuales se delimitan con las instrucciones *glBegin* y *glEnd*, mas en *OpenGL ES* no se emplea este concepto.

Algoritmo 1 Algoritmo que dibuja mediante *OpenGL ES* en *Objective-C*.

Entrada: Ninguna.

Salida: Dibuja un objeto 3D mediante *OpenGL ES*.

1: Crear y establecer el contexto.
2: Crear el búfer para dibujar y el búfer para frames.
3: glClearColor(0.0f,0.0f,0.0f,0.0f) ▷ Limpiar la pantalla
4: glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
5: Establecer los límites de la pantalla y colocar la capa de *OpenGL ES* al frente.
6: Crear una ventana gráfica (*viewport*)
7: Se establecen los *shaders* a utilizar.
8: Llamar a la función de dibujo *glDrawElements*
9: Colocar el búfer de dibujo en el contexto.

Para utilizar *OpenGL ES* en *Objective-C* hace falta de una capa especial sobre la cual se pintará, un contexto y un búfer para dibujar. Es indispensable crear este

contexto y establecer que es él quien funge como el contexto actual, pues el contexto lidia con toda la información que *iOS* requiere para dibujar con *OpenGL ES*. Además, el búfer para dibujar es un objeto que almacena la imagen a desplegar en la pantalla. Adicionalmente se requiere un búfer para *frames* (marcos), pues éste contiene al búfer para dibujar.

El Algoritmo 1 muestra un ejemplo de cómo dibujar un objeto en la capa de *OpenGL ES* que se despliega sobre la pantalla de algún dispositivo que utilice *iOS*. Cabe resaltar que para dibujar cualquier figura geométrica es necesario emplear el lenguaje de *shaders*, describiendo en archivos separados los efectos que se aplicarán sobre el objeto y/o los colores de los pixeles en dicha figura. De igual forma, se debe escribir una función que compile y ejecute en tiempo real el código de los *shaders* [48] para finalmente indicar cuáles de ellos utilizar en la escena y sobre qué objetos antes de dibujarla (línea 7).

A su vez es necesario crear el arreglo de vértices que indica las coordenadas de los mismos y el acomodo de las caras del objeto, recordando que es estrictamente necesario que todas las caras sean triangulares. Asimismo, se crea el arreglo de índices para las normales. Posteriormente se crea un búfer al cual se añaden el arreglo de vértices y de índices (búfer para vértices), marcando así que el elemento que se dibuje en la escena corresponde al objeto descrito en el arreglo de vértices y de índices.

Finalmente, si se desea que el objeto esté visto en perspectiva o se desea aplicarle transformaciones geométricas, deben establecerse (previo a la definición del objeto) una matriz de proyección y una para la vista del modelo 3D a dibujar.

Capítulo 3

Sistema de RA

A lo largo de este capítulo se exponen varios aspectos del sistema de RA, pues funge como núcleo para el desarrollo de todas las aplicaciones que prueban esta tesis. Aspectos como las consideraciones que se hicieron para realizar el sistema en un entorno móvil (sección 3.1) los espacios de coordenadas que se emplean (sección 3.2), las acciones principales que el sistema puede realizar (sección 3.3) y el manejo de información previa del marcador detectado y de la matriz de homografía para tratar de mantenerlos el mayor tiempo posible y evitar realizar operaciones de manera redundante (sección 3.4) son tratados en este capítulo.

3.1. Prácticas recomendadas para la implementación de RA en dispositivos móviles

En primer lugar, se puede realizar un procesamiento multi-hilo o intercalar procedimientos para acelerar las operaciones, como los accesos a la cámara, puesto que son operaciones de E/S. Generalmente la lectura de la cámara se realiza directamente por medio del chip de la misma, y se realiza en un módulo separado del resto de la implementación.

En este caso, se utilizó un delegado para la interfaz de la cámara. El tiempo de la lectura de la cámara se puede reducir al utilizar la imagen disponible de la cámara, procurando no capturar una nueva imagen hasta que pase un intervalo de tiempo considerable para que ocurra un cambio significativo en la escena, aunque esto aumente la latencia [4]. De igual forma, en la imagen no se filtra un canal de color específico como en los sistemas tomados como base, sino que se buscarán marcadores negros, para que de esta manera no sea necesario condicionar el entorno de trabajo para la detección de marcadores.

Para el caso de la estimación de la posición de la cámara, a diferencia de Serna Rodríguez [15] y Vázquez del Ángel [14], se estudiaron distintos tipos de marcadores. Los primeros a tomarse en consideración fueron figuras básicas similares a las de los

trabajos base, pero aplicadas sólo en alguna de las esquinas para determinar el origen del sistema de coordenadas para poder graficar el objeto. Sin embargo, una sola marca sencilla utilizada, dificulta la estimación de la orientación del marcador.

Otros tipos de patrones estudiados fueron los códigos de barras o códigos bidimensionales (QR), por su capacidad de almacenamiento de información. Al mismo tiempo, se analizaron los marcadores de plantilla, las marcas de referencia y las marcas topológicas.

Finalmente, se pensó en utilizar marcadores libres en cuanto a formas y figuras, pues pueden asociarse a objetos del mundo real y ser entendibles para los seres humanos. Sin embargo, su detección conlleva a un mayor procesamiento, el cual es posible pero no tan recomendable para dispositivos móviles y menos aún si la aplicación será utilizada en interiores.

Los tipos de marcadores que se eligieron como los más adecuados para el proyecto son los marcadores de plantilla y los de referencia. Posteriormente (en las secciones 3.3.3 (*p.* 40), 4.4 (*p.* 72), 5.1 (*p.* 80), 5.4 (*p.* 89) y 6.2 (*p.* 107)) se explicará la razón por la que se eligieron este tipo de marcadores, así como la forma en la que se trabaja con cada uno de ellos.

Para el caso del seguimiento de los objetos, se usa el esquema de las aplicaciones desarrolladas en las tesis tomadas como base. Asimismo, no se pretende utilizar una comunicación de red ni un esquema cliente-servidor para el procesamiento de la imagen ni para los algoritmos de visión por computadora, debido a que se reduce la portabilidad de las aplicaciones.

Para realizar un sistema de RA empleando los principios de Serna Rodríguez [15] y Vázquez del Ángel [14] aplicados al iPad, se siguen los pasos citados a continuación:

1. Iniciar la vista previa de la cámara del iPad para leer el video: se logra mediante un delegado en *Objective-C*.

2. Capturar las imágenes (marcos) de la cámara: para iniciar el proceso importante, se capturan varias imágenes de la cámara por segundo. De acuerdo a las especificaciones, el iPad proporciona 30 marcos por segundo.

3. Procesar los marcos para encontrar la posición de los patrones de referencia: a partir de los procesamientos realizados en la imagen de los marcos, se obtienen los patrones de referencia.

4. Calibrar la cámara, *i.e.*, encontrar los parámetros extrínsecos e intrínsecos de ésta: la calibración de la cámara es uno de los pasos más importantes, pues nos especifica las transformaciones geométricas que se deben realizar sobre el objeto, para colocarlo en la posición adecuada y no se pierda el efecto de seguimiento de

acuerdo a la posición de la cámara. Dicha calibración nos provee de los siguientes parámetros [14]:

- Matriz de proyección.
- Matriz de rotación.
- Vector de traslación.

5. Una vez conocidos los parámetros, se dibuja el objeto asociado a dicho patrón mediante *OpenGL ES*.

En cuanto a la sección del despliegue de objetos virtuales, es más recomendable utilizar un búfer de vértices para los objetos y cargarlos con *OpenGL ES* mediante archivos *.obj* transformados.

En el manual de referencia de *OpenGL ES* [49] podemos conocer a profundidad las funciones que ofrece esta biblioteca en sus versiones 2.0 y 3.0. Cabe resaltar que el número de funciones de esta versión optimizada para dispositivos móviles es considerablemente menor con respecto a su versión estándar. Por ejemplo, primitivas tales como *GL_QUADS* no son soportadas en *OpenGL ES*, por lo que los objetos deben crearse a partir de triángulos [50]. Es importante tener esto en cuenta al momento de exportar modelos 3D en formato *.obj*.

3.2. Espacios de coordenadas

Para entender cómo funciona el sistema de RA es necesario previamente aclarar que se deben considerar varios sistemas de coordenadas. En la sección 3.2.1 se describen los tipos de coordenadas que maneja *OpenGL* para posteriormente describir las coordenadas del sistema de RA en la sección 3.2.2.

3.2.1. Sistemas de coordenadas en *OpenGL*

OpenGL utiliza dos matrices principales que se encargan de posicionar el objeto en el espacio virtual y de hacer que dicho espacio sea visible para el observador a una distancia apropiada. Estas matrices son denominadas la matriz modelo-vista (o matriz del modelo para abreviar) y la matriz de proyección, denominadas en inglés *ModelView matrix* y *Projection matrix* respectivamente [51].

En la Figura 3.1 se muestra un bosquejo de cómo funcionan estas matrices en *OpenGL*, así como los sistemas de coordenadas que surgen en cada paso. De igual forma se muestra un bosquejo de cómo deberían ser conceptualmente estos sistemas de coordenadas para un mayor entendimiento, a pesar de que *OpenGL* no utilice este esquema.

- Ahora bien, las primeras coordenadas con las que nos toparemos serán, desde luego, las coordenadas del objeto (*Object coordinates*) que queramos dibujar mediante *OpenGL*. Las unidades no son de importancia, puesto que *OpenGL* no utiliza un tipo específico de unidades para el sistema. Estas coordenadas representan la ubicación de un objeto dentro del espacio de dibujo de *OpenGL*. La matriz del modelo transforma estas coordenadas para escalar el objeto de manera adecuada para que se encuentre dentro de la vista del observador, *i.e.*, se encarga de transformar las coordenadas de un modelo que representa a dicho objeto, hacia las coordenadas que simbolizan un área visible del entorno virtual (en la cual se encuentra el objeto) para un observador; de ahí que carezcan de importancia las unidades que se pretenda utilizar para los objetos.

- Una vez que estas coordenadas son transformadas, se producirán las coordenadas del observador (*Eye coordinates*). Como se mencionó, estas coordenadas indican el área que será visible para la aplicación que se está desarrollando, *i.e.*, los puntos visibles de entre todos los puntos que conforman el espacio virtual existente para *OpenGL*. Cabe resaltar que el observador se ubica en el origen del sistema con una dirección alineada hacia el eje Z negativo.

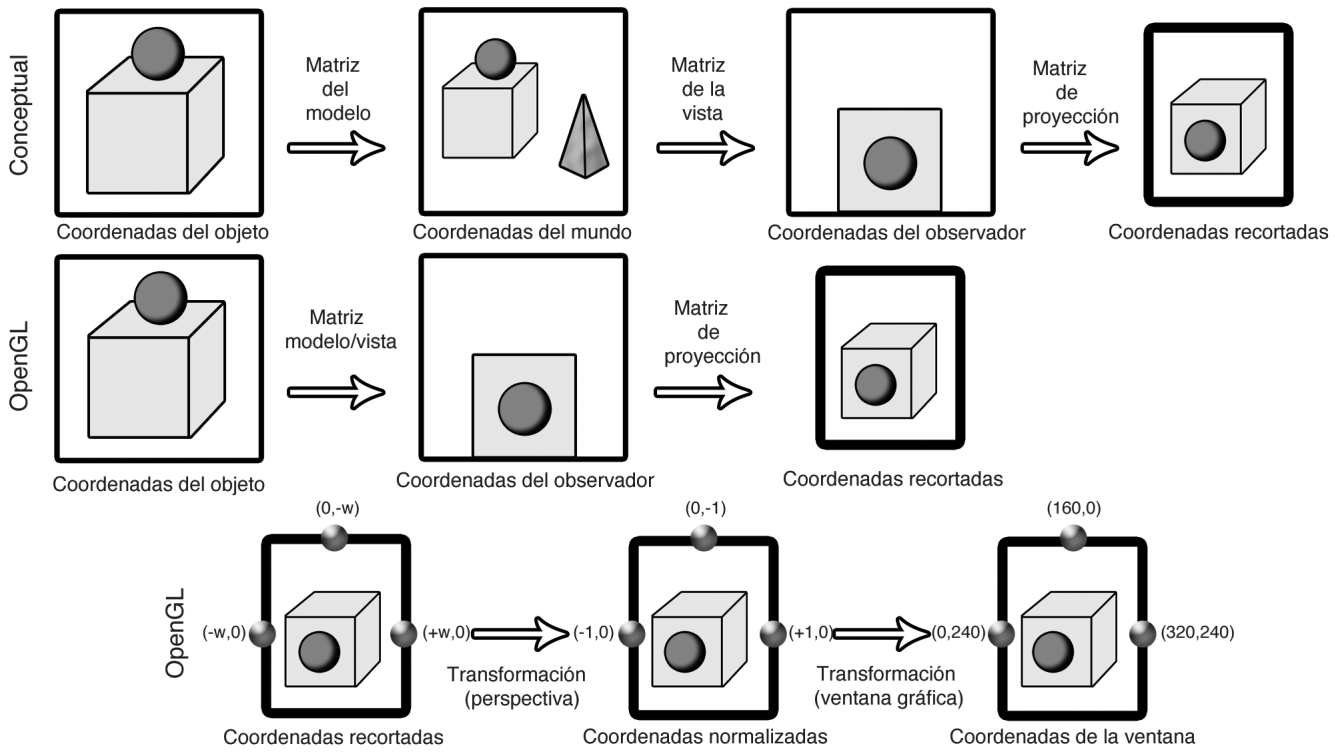


Figura 3.1: Sistemas de coordenadas (*OpenGL*).

- Posteriormente se utiliza la otra matriz, la matriz de proyección para continuar transformando las coordenadas del sistema. Las coordenadas resultantes

se denominan coordenadas recortadas (*Clip coordinates*). Estas coordenadas comprenden un área más limitada del área generada por las coordenadas del observador, sin embargo los objetos están proyectados en perspectiva para dar esencia de profundidad a la escena.

No obstante, si las coordenadas se mantienen así, se tendría una perspectiva aún muy burda y poco práctica para poder ajustarlas a las coordenadas del dispositivo en el que se ejecute la aplicación, por lo que las coordenadas recortadas son normalizadas en todas sus componentes (X, Y y Z) al dividirse entre la componente W del sistema (recordemos que en visión por computadora se suele agregar una coordenada W con la finalidad de homogeneizar el sistema).

- Las coordenadas resultantes son simplemente llamadas coordenadas normalizadas (*Normalized device coordinates*). Las coordenadas normalizadas provocan que el objeto esté centrado en el origen del sistema (coordenada $(0, 0)$) y que el rango esté entre -1 y 1 en todos los ejes. Por lo tanto, se sobrentiende que, previo a la normalización, el rango de las coordenadas recortadas comprende entre $-W_c$ y W_c.

- Una vez que las coordenadas han sido normalizadas, se escalan y trasladan mediante los parámetros de la ventana gráfica (*viewport*) en la que se visualizará el objeto sobre la pantalla, con la finalidad de producir las coordenadas de la ventana (*Window coordinates*).

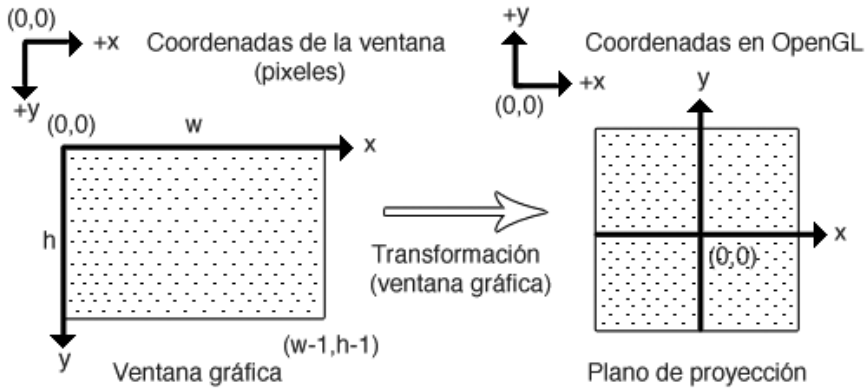


Figura 3.2: Sistema de coordenadas en un monitor y en *OpenGL*.

Como podemos observar, en cada transformación, los límites del espacio visible se ven reducidos y en el paso final ocurre la verdadera transformación hacia pixeles. Además, la mayoría de veces en que se manejan las coordenadas de un monitor o imagen, el centro del sistema se encuentra en la esquina superior izquierda, mientras que en la última transformación que se realiza para obtener las coordenadas de la ventana, el origen del sistema se encuentra en el centro de la ventana (Figura 3.2).

Por ello, la última transformación es bastante importante.

Muchos desarrolladores utilizan el término coordenadas del mundo (*World coordinates*) para designar a las coordenadas de todo el espacio virtual creado (tanto las del área visible y no visible para *OpenGL*), las cuales serían coordenadas que resultarían de aplicar al objeto las transformaciones asociadas al modelo, contenidas en la matriz del modelo; sin considerar las transformaciones asociadas a la vista del observador, las cuales también se encuentran en la matriz del modelo. Sin embargo *OpenGL* no considera las coordenadas del área que no es visible. Al final, este tipo de coordenadas pueden considerarse algo meramente conceptual para entender mejor el procedimiento, según se observa en la Figura 3.1.

Finalmente, cabe mencionar que *OpenGL ES* posee este mismo esquema.

3.2.2. Tipos de coordenadas del sistema de RA

Una vez conocida la forma en la que *OpenGL* lidia con el sistema de coordenadas y a la vez, dejándola a un lado de momento, podemos mencionar al menos cinco tipos de coordenadas diferentes que el sistema debe tomar en cuenta:

- Las coordenadas del objeto.
- Las coordenadas del marcador.
- Las coordenadas de la lente de la cámara del iPad.
- Las coordenadas de la pantalla del iPad.
- Las coordenadas del ojo del observador.

Si hacemos una analogía con los sistemas de coordenadas que maneja *OpenGL*, las coordenadas del objeto que pretendemos dibujar se corresponden con las coordenadas del objeto, tal como las maneja *OpenGL*.

Las coordenadas del marcador representan la posición en la que físicamente se encuentra el marcador en el mundo real. Se puede pensar que las coordenadas del objeto y las coordenadas del marcador son las mismas, puesto que el objeto debe dibujarse con base en la ubicación y orientación actual del marcador, pero dado el hecho de que podemos ser capaces de escalar, rotar o trasladar el objeto a nuestro antojo (incluso modificando su centro) independientemente de la posición del marcador gracias a las primitivas de *OpenGL*, es mejor considerarlas como otro tipo de coordenadas.

Además, si en algo ayuda al fácil entendimiento del sistema, incluso podemos decir que las coordenadas del marcador son reales, mientras que las coordenadas del objeto son virtuales, ya que éste no se encuentra presente en la escena del mundo real que

es capturada por la cámara.

Por otra parte, las coordenadas de la lente de la cámara representan la ubicación de ésta en el espacio físico perteneciente al mundo real, pues debemos considerar que existe una distancia que separa al marcador de la cámara. Además, este sistema de coordenadas limita el área visible sobre la cual se aplicará el efecto de RA en el mundo real. El análogo de este tipo de coordenadas en *OpenGL* son las coordenadas del observador, puesto que representan sólo al área visible por la lente de la cámara, factor que es influenciado por los valores de apertura y otras características de la misma.

Así como *OpenGL* maneja un sistema de coordenadas para representar los pixeles que tiene el dispositivo sobre el cual se dibujará la ventana gráfica, de igual forma el sistema de RA desarrollado considera el sistema de coordenadas existente en la pantalla del iPad, pues claramente se sabe que la cámara está ubicada en un lugar distinto al de la pantalla, además de ser de diferente resolución.

Finalmente, las coordenadas del ojo del observador simbolizan la ubicación de los ojos del usuario que utiliza el iPad con cualquiera de las aplicaciones que implementa este sistema de RA.

En la Figura 3.3 se muestran tres de los cinco sistemas de coordenadas mencionados anteriormente. En esta imagen se señalan los tipos de coordenadas más importantes para el sistema de RA, *i.e.*, sin considerar las coordenadas del objeto ni las del ojo del observador, para un mayor entendimiento.

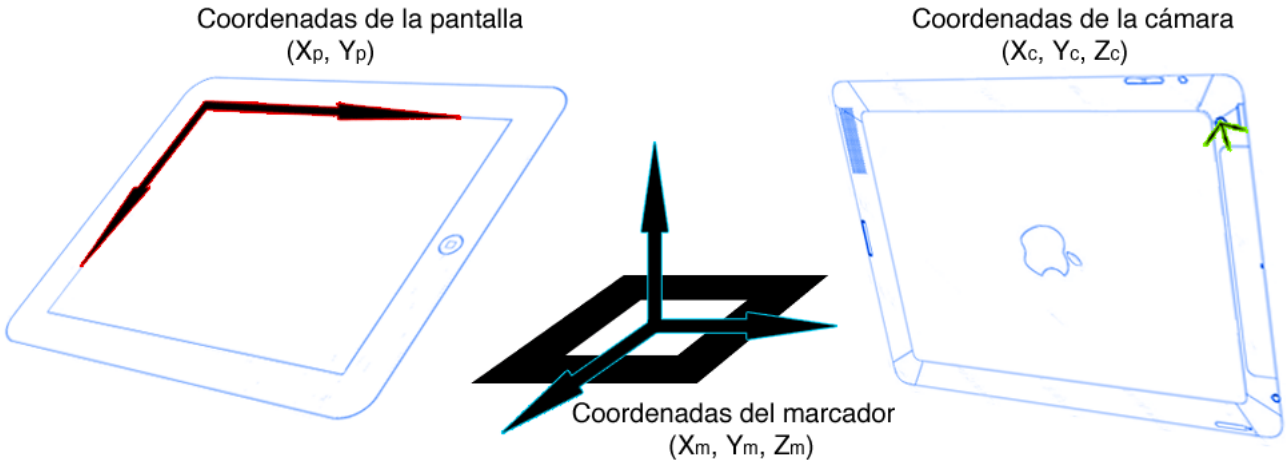


Figura 3.3: Tipos de coordenadas más importantes para el sistema de RA.

Es claro que lo que el individuo es capaz de ver en el mundo real, abarca un área mucho mayor a la que la cámara del iPad puede capturar. Sin embargo, la tarea del sistema de RA termina en la serie de transformaciones que se requiera realizar para que en la pantalla del iPad se despliegue un objeto virtual, posicionado en el lugar

donde se encuentre el marcador dentro de la imagen que proyecta el iPad, a través de la captura de la escena del mundo real por medio de la cámara que posee el mismo. Por lo anterior, las coordenadas del ojo del usuario no intervienen directamente en la serie de cálculos necesarios, pero es importante mencionar su existencia dentro del planteamiento de este sistema como modelo.

3.3. Acciones principales del sistema

La presente sección pretende dar conocimiento al lector de las acciones principales que se pueden realizar con el sistema de RA desarrollado y así mismo familiarizarse con él, puesto que es un sistema común a todas las aplicaciones de esta tesis. En el apartado 3.3.1 se muestra un esquema general del sistema, mientras que en los apartados subsecuentes se tratan a detalle las tres acciones principales del sistema: agregar un patrón (sección 3.3.2), detectar un marcador (sección 3.3.3) y estimar la posición del marcador (sección 3.3.4).

3.3.1. Esquema general del sistema de RA

En la Figura 3.4 se muestra un esquema general del sistema de RA empleado.

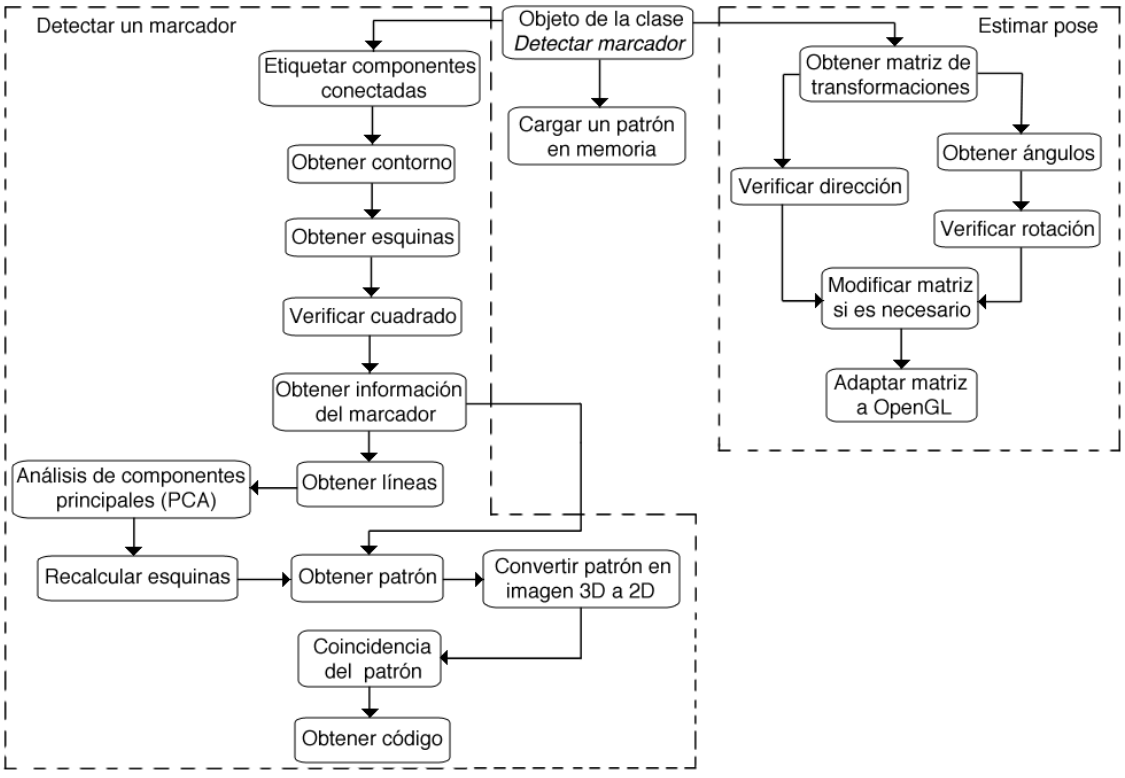


Figura 3.4: Diagrama general del sistema de RA.

Para un mayor entendimiento, se omitieron algunas cuestiones básicas como la inicialización del sistema (establecimiento de las dimensiones del marco capturado por la cámara, definición del tipo de patrones a utilizar, etc.) y ciertas operaciones fundamentales entre matrices y vectores (multiplicación de matrices, inversa de una matriz, producto punto, entre otras) que son utilizadas de manera implícita en algunos métodos del procedimiento general, obteniéndose así tres acciones principales (de las cuales dos de ellas están delimitadas por rectángulos trazados con línea punteada en la figura).

Se puede tener acceso a cualquiera de estos tres procesos principales del sistema por medio de un objeto *Detectar marcador* (el cual se estudia en la sección 4.2, en la página 68) en cualquiera de las aplicaciones de RA desarrolladas.

3.3.2. Agregar un patrón

Agregar un patrón (imagen contenida en el interior de un marcador) es una acción que sólo tiene sentido en el caso en el que se trabaje con marcadores de tipo plantilla, puesto que otros patrones como los de referencia o los códigos QR, están diseñados de tal modo que al aplicar ciertos algoritmos sobre la imagen, pueda obtenerse el identificador único que los diferencia del resto de los demás códigos del mismo tipo. Así que en ese sentido, un patrón de referencia o código QR es conocido implícitamente por el sistema.

Sin embargo, en el caso de los marcadores de plantilla, dado el hecho de que están diseñados de una manera más libre y pueden incluirse algunas curvas, el sistema debe ser entrenado para brindarle información previa sobre los marcadores de este tipo que pueden ser reconocidos durante la ejecución de la aplicación.

Cuando se usan este tipo de marcadores se procede de la siguiente manera: primero se debe diseñar un patrón con cualquier figura encerrada sobre un marco negro, tal como el que se muestra en la Figura 2.10 de la página 20 o el de la Figura 3.5.

Figura 3.5: Ejemplo de diseño de marcador de plantilla.

Una vez diseñado el patrón y encerrado en un marco negro, este marcador debe ser transformado en un archivo del tipo patrón (extension *.pat*). Estos archivos contienen 4 matrices (de $(n \times 3) \times n$, donde n es la longitud del marcador en pixeles) que representan las intensidades de color para cada pixel del marcador original, según la orientación a la que se encuentre, *i.e.*, la primera matriz contiene las intensidades del marcador rotado a 0°, la segunda posee los valores de las intensidades a 90°, la tercera a 180° y finalmente la cuarta a 270°.

La Figura 3.6 muestra dos de las matrices pertenecientes al marcador de la Figura 3.5. Estas matrices son la matriz rotada a 0° (Figura 3.6(a)) y la segunda, la matriz rotada a 180° (Figura 3.6(b)). Los valores en ceros corresponden a áreas totalmente negras en el marcador y ligeramente se puede apreciar la figura del patrón representada en valores numéricos, repetida tres veces en la dirección correspondiente (por ello, el número de filas de la matriz es $n \times 3$). Estas repeticiones se deben a los tres canales de color que posee la imagen.

143	191	191	191	191	191	191	184	159	191	191	191	191	191	191	191
191	255	255	255	255	199	77	0	0	30	132	243	255	255	255	255
191	255	255	255	169	5	0	0	0	0	0	67	239	255	255	255
191	255	255	224	9	0	0	0	0	0	0	0	121	255	255	255
191	255	255	122	0	0	0	0	0	0	0	0	17	234	255	255
191	255	255	75	117	57	0	0	0	0	22	117	43	188	255	255
191	255	255	129	233	192	3	16	7	10	120	213	170	165	255	255
191	255	255	133	125	40	170	255	255	244	8	10	190	131	255	255
191	255	227	96	245	177	17	221	244	62	131	227	174	122	255	255
191	255	180	63	255	255	170	7	0	88	244	255	132	68	255	255
191	255	119	18	255	255	255	244	223	255	255	255	95	9	234	255
191	255	54	25	178	255	255	255	255	255	255	255	53	0	174	255
191	246	3	159	58	255	255	255	255	255	255	149	115	53	127	255
191	255	186	255	58	145	255	255	255	255	218	20	189	206	215	255
191	255	255	255	197	5	134	253	255	200	40	79	254	255	255	255
191	255	255	255	255	190	150	197	239	158	138	242	255	255	255	255
143	191	191	191	191	191	191	184	159	191	191	191	191	191	191	191
191	255	255	255	255	199	77	0	0	30	132	243	255	255	255	255
191	255	255	255	169	5	0	0	0	0	0	67	239	255	255	255
191	255	255	224	9	0	0	0	0	0	0	0	121	255	255	255
191	255	255	122	0	0	0	0	0	0	0	0	17	234	255	255
191	255	255	75	117	57	0	0	0	0	22	117	43	188	255	255
191	255	255	129	233	192	3	16	7	10	120	213	170	165	255	255
191	255	255	133	125	40	170	255	255	244	8	10	190	131	255	255
191	255	227	96	245	177	17	221	244	62	131	227	174	122	255	255
191	255	180	63	255	255	170	7	0	88	244	255	132	68	255	255
191	255	119	18	255	255	255	244	223	255	255	255	95	9	234	255
191	255	54	25	178	255	255	255	255	255	255	255	53	0	174	255
191	246	3	159	58	255	255	255	255	255	255	149	115	53	127	255
191	255	186	255	58	145	255	255	255	255	218	20	189	206	215	255
191	255	255	255	197	5	134	253	255	200	40	79	254	255	255	255
191	255	255	255	255	190	150	197	239	158	138	242	255	255	255	255
143	191	191	191	191	191	191	184	159	191	191	191	191	191	191	191
191	255	255	255	255	199	77	0	0	30	132	243	255	255	255	255
191	255	255	255	169	5	0	0	0	0	0	67	239	255	255	255
191	255	255	224	9	0	0	0	0	0	0	0	121	255	255	255
191	255	255	122	0	0	0	0	0	0	0	0	17	234	255	255
191	255	255	75	117	57	0	0	0	0	22	117	43	188	255	255
191	255	255	129	233	192	3	16	7	10	120	213	170	165	255	255
191	255	255	133	125	40	170	255	255	244	8	10	190	131	255	255
191	255	227	96	245	177	17	221	244	62	131	227	174	122	255	255
191	255	180	63	255	255	170	7	0	88	244	255	132	68	255	255
191	255	119	18	255	255	255	244	223	255	255	255	95	9	234	255
191	255	54	25	178	255	255	255	255	255	255	255	53	0	174	255
191	246	3	159	58	255	255	255	255	255	255	149	115	53	127	255
191	255	186	255	58	145	255	255	255	255	218	20	189	206	215	255
191	255	255	255	197	5	134	253	255	200	40	79	254	255	255	255
191	255	255	255	255	190	150	197	239	158	138	242	255	255	255	255

(a) Matriz a 0°.

255	255	255	255	242	138	158	239	197	150	190	255	255	255	255	191
255	255	255	254	79	40	200	255	253	134	5	197	255	255	255	191
255	215	206	189	20	218	255	255	255	255	145	58	255	186	255	191
255	127	53	115	149	255	255	255	255	255	255	58	159	3	246	191
255	174	0	53	255	255	255	255	255	255	255	178	25	54	255	191
255	234	9	95	255	255	255	223	244	255	255	255	18	119	255	191
255	255	68	132	255	244	88	0	7	170	255	255	63	180	255	191
255	255	122	174	227	131	62	244	221	17	177	245	96	227	255	191
255	255	131	190	10	8	244	255	255	170	40	125	133	255	255	191
255	255	165	170	213	120	10	7	16	3	192	233	129	255	255	191
255	255	188	43	117	22	0	0	0	0	57	117	75	255	255	191
255	255	234	17	0	0	0	0	0	0	0	0	122	255	255	191
255	255	255	121	0	0	0	0	0	0	0	9	224	255	255	191
255	255	255	239	67	0	0	0	0	0	5	169	255	255	255	191
255	255	255	255	243	132	30	0	0	77	199	255	255	255	255	191
191	191	191	191	191	191	191	159	184	191	191	191	191	191	191	143
255	255	255	255	242	138	158	239	197	150	190	255	255	255	255	191
255	255	255	254	79	40	200	255	253	134	5	197	255	255	255	191
255	215	206	189	20	218	255	255	255	255	145	58	255	186	255	191
255	127	53	115	149	255	255	255	255	255	255	58	159	3	246	191
255	174	0	53	255	255	255	255	255	255	255	178	25	54	255	191
255	234	9	95	255	255	255	223	244	255	255	255	18	119	255	191
255	255	68	132	255	244	88	0	7	170	255	255	63	180	255	191
255	255	122	174	227	131	62	244	221	17	177	245	96	227	255	191
255	255	131	190	10	8	244	255	255	170	40	125	133	255	255	191
255	255	165	170	213	120	10	7	16	3	192	233	129	255	255	191
255	255	188	43	117	22	0	0	0	0	57	117	75	255	255	191
255	255	234	17	0	0	0	0	0	0	0	0	122	255	255	191
255	255	255	121	0	0	0	0	0	0	0	9	224	255	255	191
255	255	255	239	67	0	0	0	0	0	5	169	255	255	255	191
255	255	255	255	243	132	30	0	0	77	199	255	255	255	255	191
191	191	191	191	191	191	191	159	184	191	191	191	191	191	191	143
255	255	255	255	242	138	158	239	197	150	190	255	255	255	255	191
255	255	255	254	79	40	200	255	253	134	5	197	255	255	255	191
255	215	206	189	20	218	255	255	255	255	145	58	255	186	255	191
255	127	53	115	149	255	255	255	255	255	255	58	159	3	246	191
255	174	0	53	255	255	255	255	255	255	255	178	25	54	255	191
255	234	9	95	255	255	255	223	244	255	255	255	18	119	255	191
255	255	68	132	255	244	88	0	7	170	255	255	63	180	255	191
255	255	122	174	227	131	62	244	221	17	177	245	96	227	255	191
255	255	131	190	10	8	244	255	255	170	40	125	133	255	255	191
255	255	165	170	213	120	10	7	16	3	192	233	129	255	255	191
255	255	188	43	117	22	0	0	0	0	57	117	75	255	255	191
255	255	234	17	0	0	0	0	0	0	0	0	122	255	255	191
255	255	255	121	0	0	0	0	0	0	0	9	224	255	255	191
255	255	255	239	67	0	0	0	0	0	5	169	255	255	255	191
255	255	255	255	243	132	30	0	0	77	199	255	255	255	255	191
191	191	191	191	191	191	191	159	184	191	191	191	191	191	191	143

(b) Matriz a 180°.

Figura 3.6: Ejemplos de matrices de intensidades contenidas en un archivo de tipo patrón.

El sitio citado en [52] ofrece la opción de generar las matrices del marcador a partir de una imagen proporcionada por el usuario o bien, por medio de una imagen capturada directamente de una cámara web. Los marcadores a generarse pueden ser de 4×4, 8×8, 16×16, 32×32 o 64×64. En el caso de seleccionar 16×16, se generan matrices de 48×16 como las de la Figura 3.6. Así mismo, se proporciona la opción de seleccionar el porcentaje que se desee tomar del marcador, lo cual añade o no segmentos del marco que encierra a la figura para ser considerados en la matriz. Lo recomendable es dejarlo al 50 % para evitar añadir a las matrices información del

borde del marcador (si el porcentaje es mayor al 50 %) o para evitar que se pierda información de la imagen contenida en el interior del marcador (si el porcentaje es menor al 50 %).

Mientras más grandes sean las dimensiones seleccionadas del marcador a generar, la semejanza entre el diseño en el interior del marcador y la imagen que se forma mediante los valores de la matriz también será mayor.

Una vez que se tienen los patrones a utilizar en el sistema de RA (como archivos *.pat*), se procede a cargarlos en memoria para que sean reconocidos durante la ejecución de la aplicación, utilizando el método descrito en el Algoritmo 2.

Como se puede observar en este algoritmo, sólo cuando se cargue el primer patrón se inicializarán los marcadores a 0 (línea 1), labor que es posible por medio de un arreglo de banderas, en el cual, cada vez que se agregue un patrón al sistema la bandera cambiará su valor a 1, según se puede entender en el paso 15. Hacer uso de este arreglo de banderas nos permitirá determinar la primera posición actual del arreglo en la cual su valor no sea 1, otorgándonos así el identificador único que se le asignará al patrón que se pretende agregar al sistema.

Algoritmo 2 Algoritmo para cargar un patrón en memoria dentro del sistema.

Entrada: Un archivo del tipo patrón.
Salida: Un identificador para el patrón, -1 en caso de fallo.

1: Inicializar patrones a 0 en caso de que $numPatronesCargados = -1$
2: Buscar la última posición i del patrón cuyo valor sea 0
3: **if** $i = MAX\_PATRONES\_A\_CARGAR$ **then**
4: **return** -1
5: $numPatron \leftarrow i$
6: Verificar existencia del archivo.
7: **for** $h = 0$ to 4 **do**
8: **for** $i_3 = 0$ to 3 **do**
9: **for** $i_2 = 0$ to $ALTO$ **do**
10: **for** $i_1 = 0$ to $ANCHO$ **do**
11: $j \leftarrow$ Valor numérico en el archivo
12: $j \leftarrow 255 - j$
13: $patron[numPatron][h][(i_2 \times ANCHO + i_1) \times 3 + i_3] = j$
14: Obtener la desviación estándar de los valores del patrón.
15: Establecer que la posición del patrón encontrada en el paso 2 ya no es 0
16: $numPatronesCargados \leftarrow numPatronesCargados + 1$
17: **return** $numPatron$

Además, cada vez que se añade un patrón al sistema y se lee un archivo *.pat*, se realiza un ciclo por cada matriz de orientación (0°, 90°, 180° y 270°), uno por cada

canal (R, G y B), uno con respecto al alto del patrón y finalmente uno con respecto al ancho del mismo (líneas 7 a 10). A pesar de que el patrón se repita tres veces en cada matriz de orientación, no siempre se cumple que los valores numéricos sean todos idénticos, algunos de los archivos *.pat* generados (mediante el sitio citado en [52]) presentaron variaciones en algunas entradas de las matrices, sobre todo aquellas que se encuentran en el contorno de la figura, *i.e.*, la periferia donde ocurre una transición entre el color negro de la figura y el blanco de fondo en el patrón.

El valor numérico leído del archivo en cada iteración es almacenado en j, y posteriormente, al valor máximo que puede tomar el pixel (255) se le resta este valor para crear una matriz completa llamada *patron* (que incluye las cuatro orientaciones) que representa al patrón (líneas 11 a 13). Esta matriz será utilizada en procesos que se describirán posteriormente, al igual que la desviación estándar descrita en el paso 14.

3.3.3. Detectar un marcador

Para ser capaces de detectar un marcador, primero debemos representarlo de una manera adecuada para su detección. La estructura que representa a un marcador en el sistema de RA, se muestra en la Tabla 3.1 [53].

Tipo	Campo	Descripción
`int`	**area**	Cantidad de pixeles de la región detectada.
`int`	**id**	Identificador único del marcador.
`int`	**direccion**	Rotación del marcador detectado (valores de 0 a 3, dado que existen 4 posibles direcciones).
`double`	**confianza**	Probabilidad de ser un marcador (entre 0 y 1).
`double`	**centro[2]**	Centro del marcador.
`double`	**lineas[4][3]**	Cuatro aristas del cuadrado del marcador (tres valores por línea de acuerdo a la ecuación de la recta $ax + by + c = 0$).
`double`	**vertices[4][2]**	Posición de los cuatro vértices que conforman las esquinas del contorno (dos coordenadas por punto).

Tabla 3.1: Campos de la estructura que representa a un marcador.

Una vez conocida la estructura del marcador, es posible detectarlo al realizar los pasos que se enlistan en seguida [50] (claro, en el supuesto de que realmente se tiene un marcador en la escena):

1. Etiquetar las componentes conectadas en la imagen.
2. Obtener el contorno.
3. Verificar que el contorno sea un cuadrado obteniendo las esquinas.
4. Obtener la información del marcador.

5. Obtener las líneas del cuadrado.
6. Ajustar las líneas y recalcular las esquinas.
7. Obtener la homografía y quitar la transformación proyectiva.
8. Obtener el código del patrón.

Etiquetar componentes

Previo al etiquetado de componentes, es necesario realizar un proceso de binarización, por medio del cual, la imagen capturada por la cámara del iPad pasará a tener sólo dos intensidades de color, blanca y negra. Este sencillo proceso se describe en la Ecuación 3.1.

$$r + g + b < umbral \times 3 \tag{3.1}$$

Así, los pixeles que satisfacen el umbral son válidos para hacer el etiquetado, mientras que los demás son ignorados. Estos pixeles son agrupados en conjuntos conocidos como *pixeles conectados*.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	0	0	3	3	0	0	3	3	0
0	1	1	1	1	1	1	1	1	0	0	3	3	3	3	0	0
0	0	0	1	1	1	1	0	0	0	3	3	3	3	0	0	0
0	0	1	1	1	1	0	0	0	3	3	3	0	0	3	3	0
0	1	1	1	0	0	1	1	0	0	0	3	3	3	0	0	0
0	0	1	1	0	0	0	0	0	3	3	0	0	0	3	3	0
0	0	0	0	0	0	3	3	3	3	0	0	3	3	3	3	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 3.7: Ejemplo gráfico de imagen con componentes conectadas.

A cada grupo de pixeles conectados se le asigna una etiqueta, la cual consta de un entero mayor que cero [54]. Por medio de iteraciones sobre la imagen binarizada desde la esquina superior izquierda hasta la esquina inferior derecha se calculan y asignan las etiquetas, almacenándolas en una segunda imagen. Algunas regiones con etiquetas diferentes se unen para formar parte de una misma región y evitar redundancia. El resultado del procedimiento permite determinar regiones similares en la imagen.

La Figura 3.7 muestra un ejemplo gráfico de una imagen en la cual se han etiquetado sus componentes, además, tras varias iteraciones permanecen sólo tres etiquetas distintas. De igual forma, el Algoritmo 3 muestra cómo se etiquetan los pixeles.

Algoritmo 3 Algoritmo que etiqueta las componentes conectadas de una imagen.

Entrada: Una imagen capturada por la cámara del iPad.
Salida: La misma imagen con las componentes conectadas.

```
1: if el pixel anterior está etiquetado then
2:     Colocar la misma etiqueta.
3: else if el pixel de la esquina superior derecha está etiquetado then
4:     if al menos uno de los pixeles de la izquierda está etiquetado then
5:                                  ▷ Dos regiones etiquetadas están conectadas.
6:         Registrar que ambas etiquetas son equivalentes.
7: else if el pixel de la esquina superior izquierda está etiquetado then
8:     Colocar esta etiqueta.
9: else if el pixel de la izquierda está etiquetado then
10:    Colocar esta etiqueta.
11: else                                ▷ Ninguno de los casos anteriores.
12:    Asignar una nueva etiqueta.
```

Mientras se realiza este procedimiento, se recopila cierta información, entre ella:

- El número de pixeles asignados a cada etiqueta.
- La suma de las coordenadas x y suma de las coordenadas y de cada pixel de la etiqueta.
- Los valores mínimos y máximos de las coordenadas x y y para el pixel.

Esta información permite determinar el centro de la región etiquetada y es utilizada en pasos posteriores del proceso de detección.

Detección de contornos

En el contexto de detección de marcadores, la detección de un contorno corresponde a encontrar los vértices (pixeles) que conforman el borde más externo del marco negro perteneciente al marcador. En palabras simples, el procedimiento comienza en un vértice y se avanza en el sentido de las manecillas del reloj sobre el marcador, encadenando los pixeles descubiertos hasta que se vuelva al vértice inicial. Lo importante es saber cómo recorrer el marcador. El Algoritmo 4 (tomado de [55]) describe este procedimiento.

Primero (en la línea 1) se realiza una búsqueda lineal en la imagen para encontrar el primer pixel negro ($V_{comienzo}$), *i.e.*, el primer vértice del contorno. Posteriormente se toma el pixel vecino a éste ($P_{actual}\prime$) y se verifica su color, en caso de que no sea negro, se toma el siguiente vecino de manera radial hasta que se encuentre un pixel negro (línea 4), pues éste formará parte del contorno detectado (línea 7). En caso de que el pixel hallado sea el vértice inicial, el proceso se detiene ya que los vértices

añadidos al contorno detectado forman una figura cerrada donde todos ellos están conectados. De lo contrario, si el pixel encontrado no es el inicial, se toma el vecino del último pixel añadido al contorno y se repite el proceso desde la línea 4.

Algoritmo 4 Algoritmo para detectar un contorno.

Entrada: Una imagen con las componentes conectadas etiquetadas.
Salida: El contorno del marcador.

1: Realizar una búsqueda lineal de izquierda a derecha sobre los pixeles del área recortada y encontrar el vértice perteneciente al marcador ($V_{comienzo}$).
2: $P_{actual} \leftarrow V_{comienzo}$
3: $P_{actual}\prime \leftarrow$ pixel que está antes de P_{actual}
4: **while** $P_{actual}\prime$ no es negro **do** ▷ Pixel con etiqueta 0
5: $\quad P_{actual}\prime \leftarrow$ pixel vecino
6: $P_{actual} \leftarrow$ pixel negro encontrado.
7: Agregar pixel actual a contorno.
8: **if** $P_{actual} = V_{comienzo}$ **then**
9: $\quad$ **return**
10: **else**
11: $\quad P_{actual}\prime \leftarrow$ pixel anterior añadido al contorno.
12: $\quad$ Avanzar un pixel en sentido de las manecillas del reloj sobre P_{actual}
13: $\quad$ Volver al paso 4

Debemos considerar que cada pixel tiene ocho vecinos, asimismo, el procedimiento de etiquetado nos provee de una región cortada de la imagen original, la cual contiene el área con el marcador.

Además, ya que el procedimiento de etiquetado almacena los valores mínimos y máximos de las coordenadas x y y del pixel, se puede ser más preciso al saber cuál pixel pertenece al marcador (en caso de que el marcador esté rotado o incluso distorsionado), pues no es posible asumir que el pixel más a la izquierda corresponde a un vértice del marcador. El procedimiento a seguir es determinar las esquinas del cuadrado de este contorno.

Obtener esquinas y verificar si el contorno es un cuadrado

Mediante el procedimiento descrito anteriormente, se obtiene un contorno (en el supuesto de que la región etiquetada realmente contiene un marcador) y dentro de éste, tenemos asegurada solamente una de las cuatro esquinas del cuadrado, por lo que es necesario encontrar las otras tres [56].

Estimar la ubicación de la segunda esquina es relativamente sencillo, pues comenzando a partir del primer punto en la cadena que conforma el contorno (*i.e.*, el primer vértice) se busca el punto que esté más alejado de él aplicando el teorema de

Pitágoras. En esta situación sólo existen dos posibilidades (mismas que se ejemplifican gráficamente en la Figura 3.8):

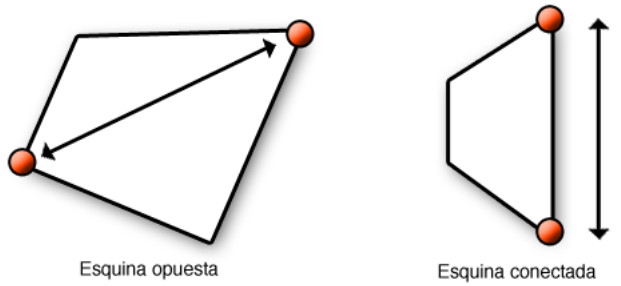


Figura 3.8: Dos posibles casos en la estimación del segundo vértice del cuadrado.

1. El vértice se encuentra en la esquina opuesta, o
2. El vértice se conecta directamente con el punto inicial.

Sin embargo, para encontrar las dos esquinas restantes, se deben seguir dos procedimientos distintos, pero similares. Si tenemos una línea a la cual llamamos l_n (línea norte) y ésta es perpendicular a otra línea denotada l_{se} (línea sureste), entonces, tendremos que maximizar de principio a fin la longitud de l_{se} por cada segmento de la cadena que forma el contorno.

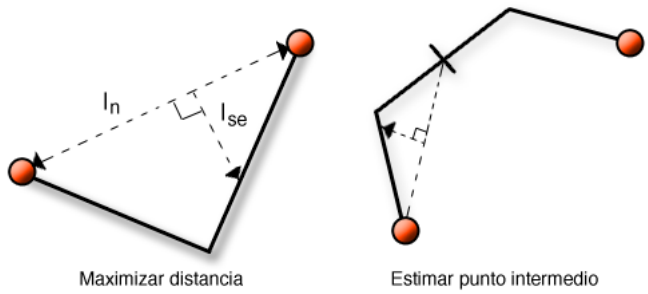


Figura 3.9: Procesos para estimar el tercer y/o cuarto vértice.

Para cada punto entre el comienzo y fin del segmento, se calcula la longitud de l_{se} desde dicho punto hasta la intersección con l_n. Un ejemplo de l_n y l_{se} se muestran en la parte izquierda de la Figura 3.9. Cada vez que se calcula la longitud de l_{se} se registra el punto con la longitud máxima, pero si esta longitud excede un umbral, dicho punto es considerado esquina.

Retomando los dos casos para encontrar la esquina V_2, si se trata del primero (en donde V_2 está en la esquina opuesta) entonces se aplica el método de maximizar la longitud a los puntos en la cadena que se forma desde V_1 hasta V_2. Posteriormente se aplica de V_2 a V_1, al final de la cadena.

No obstante, si ocurre el segundo caso (en donde V_2 se conecta con V_1) es necesario estimar un punto intermedio en el lado en el que falte encontrar las otras dos esquinas (lado derecho de la Figura 3.9). Este punto intermedio se utiliza posteriormente como índice para el método de encontrar vértices (esquinas).

Sin embargo, no es sencillo saber cuál de los dos casos se está produciendo, por lo que el método debe ser recursivo (aunque limitando la recursión) al encontrar el punto perpendicular cuya distancia sea mayor (como se muestra en el Algoritmo 5). Si la distancia está bajo un umbral de tolerancia, el método debe aplicarse nuevamente entre el punto inicial y el punto identificado como el de mayor distancia y posteriormente entre este punto y el punto final, contando el número de esquinas encontradas en el proceso.

Algoritmo 5 Algoritmo que obtiene las esquinas de un contorno.

Entradas: Las coordenadas de los vértices del contorno así como un punto inicial y uno final en los que se realiza la búsqueda.

Salida: Las coordenadas de un contorno.

1: $a \leftarrow Y_{fin} - Y_{inicio}$
2: $b \leftarrow X_{inicio} - X_{fin}$
3: $c \leftarrow (X_{fin} \times Y_{inicio}) - (Y_{fin} \times X_{inicio})$
4: $distanciaMaxima \leftarrow 0$
5: **for** $i = inicio$ to fin **do**
6: $\quad distanciaMaxima \leftarrow a \times X_i + b \times Y_i + c$
7: $\quad$ **if** $d \times d > distanciaMaxima$ **then**
8: $\quad\quad distanciaMaxima \leftarrow d \times d$
9: $\quad\quad v \leftarrow i$
10: **if** $\frac{distanciaMaxima}{a^2 \times b^2} > umbral$ **then**
11: $\quad$ **if** $obtenerEsquinas(X, Y, inicio, v) < 0$ **then**
12: $\quad\quad$ **return** -1
13: $\quad$ **if** $numeroDeVertices > 5$ **then**
14: $\quad\quad$ **return** -1
15: $\quad vertices[numeroDeVertices] = v$
16: $\quad numeroDeVertices \leftarrow numeroDeVertices + 1$
17: $\quad$ **if** $obtenerEsquinas(X, Y, v, fin) < 0$ **then**
18: $\quad\quad$ **return** -1
return 0

En el primer caso, cada mitad encontrará un solo vértice, mientras que en el segundo caso, una mitad encontrará cero vértices y la otra posiblemente dos vértices.

De ser así, esta mitad es dividida en dos segmentos y se aplica nuevamente el método hasta que se encuentre sólo un vértice en cada mitad. Si en algún punto, el número de vértices encontrado es mayor a cuatro, claramente no es un cuadrado (línea 14). De esta manera es como se verifica si el contorno es o no un cuadrado, con base en el número de esquinas encontradas al realizar el proceso anterior.

Hasta este punto se estima la orientación de los vértices, lo que permite saber el orden de los mismos si estos fueran numerados. El paso a seguir es la obtención del interior del marcador, sin embargo, la estimación de los vértices debe ser refinada.

Análisis de Componentes Principales

Basar la estimación de la posición de un marcador en la ubicación de un pixel no es preciso. Considérese la situación en la que exista un objeto oscuro en el fondo de la escena detrás del marcador a tal grado que haga parecer que alguna de las esquinas está más alejada del centro del marcador de lo que realmente está. En este tipo de casos, es posible utilizar el resto de los pixeles en cada lado para mejorar la estimación de manera significativa.

Un método de regresión lineal sobre los puntos de cada lado producirá una línea que se ajuste mejor a los puntos. La intersección de estas líneas puede ser empleada como la ubicación de las esquinas con un mayor nivel de confianza. La técnica para realizar este ajuste es conocida como *Análisis de Componentes Principales* (*PCA* por sus siglas en inglés).

El *PCA* es una técnica importante utilizada para reducir la dimensionalidad y ha sido aplicada en el área de clasificación de patrones y de representación de datos [57]. Trabaja con un conjunto de entrenamiento $X = [x_1, x_2, \ldots, x_n] \in \mathbb{R}^{d \times n}$, donde d representa la dimensionalidad del conjunto de datos. Además, no requiere etiquetas de clase para cada vector x_j.

En el procedimiento convencional se forma una matriz de covarianza definida como $\Sigma_x = LL^T$ (donde $L = \frac{1}{\sqrt{n}}[x_1 - \mu, x_2 - \mu, \ldots, x_n - \mu]$ y $\mu = \frac{1}{n}\sum_{j=1}^{n} x_j$ es el centroide de los datos entrenados) y se realiza su descomposición en valores propios o *eigenvalores* para extraer $h \leq t$ (donde $t = \text{rango}(L)$) vectores propios o *eigenvectores* correspondientes a h eigenvalores principales (*i.e.*, aquellos eigenvectores asociados a los eigenvalores más grandes de Σ_x de L). El valor de h se encuentra entre $[1, t]$ y representa la dimensionalidad del espacio de dimensiones reducido.

La transformación del *PCA* ($\Phi \in \mathbb{R}^{d \times h}$) convierte los vectores de dimensionalidad d a vectores de dimensionalidad h donde $h < d$. También puede regresar de dimensionalidad h a d con un mínimo error. La matriz de transformaciónes Φ para el error satisface $\Sigma_x \phi_i = \lambda_i \phi_i$, donde $\Sigma_x \in \mathbb{R}^{d \times d}$ es la matriz de covarianza, λ_i son los eigenvalores y $\phi_i \in \mathbb{R}^{d \times 1}$ los eigenvectores correspondientes a λ_i (desde $1 \ldots h$). Los

vectores columna de Φ son los eigenvectores principales ϕ_i, *i.e.*, los que corresponden a los eigenvalores más grandes.

Si la dimensionalidad de la matriz de covarianza Σ_x es extremadamente grande ($d \gg n$), entonces realizar la descomposición en eigenvalores se vuelve un procedimiento muy lento. En estos casos, se realiza esta descomposición para $L^T L$ en lugar de hacerla para LL^T, para ello, se realiza el proceso denominado *Descomposición en Valores Singulares* (comúnmente conocido como *SVD* por sus siglas en inglés) para obtener los eigenvectores y raíces cuadradas de los eigenvalores de Σ_x.

Sin embargo, la *SVD* es un procedimiento con una complejidad computacional considerable si hablamos de su uso en dispositivos móviles, por lo que alternativamente se puede hacer uso de la descomposición *QR* para lograr el mismo trabajo y con una complejidad menor (computacionalmente hablando), tal como se muestra en el trabajo de Sharma, Paliwal, Imoto y Miyano [57]. Ellos trabajaron con conjuntos de datos de alta dimensionalidad, sin embargo, ajustar un conjunto de puntos en 2D representa un problema bidimensional que puede ser resuelto sin ninguna función numérica especial.

El Algoritmo 6 muestra el procedimiento usado en el sistema para realizar el *PCA* a un conjunto de n puntos almacenados en una matriz de $n \times 2$. La matriz de entrada se forma a partir de los vectores de coordenadas X y Y de los puntos que forman el contorno, alternando las coordenadas en las columnas de la matriz, *i.e.*, colocando las coordenadas X en la primer columna y las coordenadas Y en la segunda.

Entre los pasos 3 y 5 del algoritmo se obtiene la matriz L necesaria para el método. Posteriormente se construye la matriz de covarianza Σ_x como $\begin{bmatrix} a & b \\ b & c \end{bmatrix}$ (líneas 6 a 9). Los dos eigenvalores se calculan resolviendo el determinante $\det(\Sigma_x - \lambda I) = 0$ (línea 13) y el eigenvector es calculado aplicando el teorema de Cayley-Hamilton.

El *PCA* calcula la dirección con mayor variación para los puntos dados y así refina cada una de las cuatro líneas que conforman el contorno del marcador encontrado en pasos previos, por lo que el procedimiento se lleva a cabo cuatro veces (una por cada línea) siendo los valores de los eigenvectores quienes reemplazan a las constantes a, b y c de la ecuación de la línea estimadas previamente. Así mismo, las cuatro esquinas del cuadrado del marcador son de igual forma refinadas.

Realizar el *PCA* resulta muy práctico puesto que si el papel sobre el que se encuentra impreso el marcador es flexionado, la detección sigue en pie y el objeto virtual que se coloque sobre éste puede aún dibujarse. Esto se debe a que el proceso de refinación estima un cuadrado perfecto que puede ser un poco menor al del marcador original, conservándolo el mayor tiempo posible hasta que la deformación de la imagen provoque que sea imposible considerar como cuadrado al marcador.

Algoritmo 6 Algoritmo para realizar el *PCA* a una matriz.

Entrada: Una matriz con las coordenadas X y Y de los puntos de una línea.
Salidas: Los valores y vectores propios de los datos de entrada, -1 en caso de fallo.

1: **if** *renglones* || *columnas* de la matriz son < 2 **then**
2: **return** -1
3: Crear (a partir de la matriz de entrada) un vector donde cada posición i contiene la media de los valores de la fila i. ▷ Obtener el centroide de los datos.
4: Restar a cada valor de la matriz, cada uno de los valores del vector generado en el paso 3. ▷ Centrar los datos.
5: Dividir cada valor de la matriz entre $\sqrt{renglones}$ ▷ Obtener L
6: **for** $i = 1$ to n **do** ▷ Obtener Σ_x
7: $a \leftarrow L_{i0} \times L_{i0}$ ▷ x^2
8: $b \leftarrow L_{i0} \times L_{i1}$ ▷ xy
9: $c \leftarrow L_{i1} \times L_{i1}$ ▷ y^2
10: $v \leftarrow \sqrt{(a-c)^2 + 4 \times b^2}$
11: $\lambda_{min} \leftarrow \frac{((c+a)-v)}{2}$
12: $\lambda_{max} \leftarrow \frac{((c+a)+v)}{2}$
13: $A \leftarrow \Sigma_x - \lambda_{min} I$
14: $\delta \leftarrow 8 \times \lambda_{max} \times EPSILON$
15: **if** norma$(A[1,;]) < \delta$ **then**
16: $r \leftarrow [0, 1]$
17: **else if** norma$(A[2,;]) < \delta$ **then**
18: $r \leftarrow [1, 0]$
19: **else**
20: $r \leftarrow \frac{A[1,;]}{\text{norma}(A[1,;])}$

De igual forma, utilizando el *PCA* es posible seguir detectando el marcador a una distancia más lejana con respecto a la cámara y de no hacerlo, es muy probable que la más mínima deformación del marcador en la imagen (*i.e.*, una distorsión que provoque que el marcador se vea rectangular y no cuadrado) provoque que el objeto virtual aparezca y desaparezca constantemente.

Obtención y coincidencia del patrón

Una vez que se ha realizado el ajuste a las líneas del contorno se obtiene la región del interior del marcador, para lo cual se considera el ancho del borde del marco, dato que se conoce desde el momento en el que se inicia el sistema.

Además, una vez que se tienen los vértices del contorno, se consultan los parámetros de la cámara para transformar el segmento de la imagen original (considerando tres canales) que contenga sólo al interior del marcador en una imagen en un plano

2D, *i.e.*, de un marcador encontrado en una escena 3D a una imagen 2D rectificada para poder verificar la coincidencia del patrón con otros patrones, disminuyendo también la resolución [58], pues es necesario considerar que la matriz de cada patrón almacenada en memoria (con la que se comparará el patrón obtenido en este paso) contiene valores de un marcador de baja resolución descrito en el archivo de tipo patrón (*.pat*) que fue cargado en el sistema.

Algoritmo 7 Algoritmo para verificar la coincidencia entre patrones.

Entrada: Una imagen donde posiblemente se encuentre un marcador.
Salidas: El identificador, orientación y nivel de confianza del marcador encontrado.

1: **for** $i = 0$ to $(ALTO \times ANCHO \times 3)$ **do**
2: $\quad promedio \leftarrow promedio + (255 - imgEntrada[i])$
3: $promedio \leftarrow \frac{promedio}{ALTO \times ANCHO \times 3}$
4: **for** $i = 0$ to $(ALTO \times ANCHO \times 3)$ **do**
5: $\quad imgAux[i] \leftarrow (255 - imgEntrada[i]) - promedio$
6: $\quad suma \leftarrow suma + imgAux[i] \times imgAux[i]$
7: $raiz \leftarrow \sqrt{suma}$
8: $resultOrientacion \leftarrow -1$
9: $resultCodigo \leftarrow -1$
10: $k \leftarrow -1$
11: $max \leftarrow 0.0$
12: **for** $l = 0$ to $numPatronesCargados$ **do**
13: $\quad k \leftarrow k + 1$
14: $\quad$ **while** $arregloDeBanderas[k] = 0$ **do**
15: $\quad\quad k \leftarrow k + 1$
16: $\quad$ **for** $j = 0$ to 4 **do**
17: $\quad\quad suma \leftarrow 0$
18: $\quad\quad$ **for** $i = 0$ to $(ALTO \times ANCHO \times 3)$ **do**
19: $\quad\quad\quad suma \leftarrow suma + imgAux[i] \times patron[k][j][i]$
20: $\quad\quad suma_2 \leftarrow \frac{suma}{\frac{desviacionEstandarPatron}{raiz}}$
21: $\quad\quad$ **if** $suma_2 > max$ **then**
22: $\quad\quad\quad max \leftarrow suma_2$
23: $\quad\quad\quad resultOrientacion \leftarrow j$
24: $\quad\quad\quad resultCodigo \leftarrow k$
25: $codigo \leftarrow resultCodigo$
26: $orientacion \leftarrow resultOrientacion$
27: $confianza \leftarrow max$

Resumiendo, se recorre la imagen original en el área delimitada por los vértices, eliminando el borde del marcador, se transforma y se almacenan los pixeles en una nueva imagen, disminuyendo en cada paso la resolución. Una vez que se tiene el área

con el patrón, se procede a compararlo con los existentes en memoria empleando el procedimiento descrito en el Algoritmo 7.

En este algoritmo se hace uso de algunas variables descritas en el Algoritmo 2, entre ellas el arreglo de banderas para definir el identificador único del marcador, el arreglo que almacena la matriz que representa al patrón, la variable que almacena el número de patrones cargados actualmente en el sistema y la desviación estándar de los valores del patrón.

El algoritmo comienza calculando la media de la imagen de entrada (línea 2), *i.e.*, la imagen que posiblemente contiene un marcador que puede coincidir con alguno cargado previamente en el sistema. Esta media es restada a los valores originales de los pixeles (línea 5), almacenando los resultados en una nueva imagen ($imgAux$). También se calcula la raíz de la media de los pixeles al cuadrado de esta nueva imagen (línea 7). Asimismo, algunas variables se inician en -1 (líneas 8 a 10), para que si no se detecta ningún marcador, se tenga conocimiento de ello mediante este valor.

Posteriormente se inicia el ciclo principal que determina el código del marcador identificado, la orientación a la que se encuentra y un valor de confianza para dicho marcador (línea 12). El ciclo se realiza por todos los marcadores que se encuentren cargados en el sistema, incrementando la variable k para que el ciclo pueda proseguir desde un valor posterior al que se asigne en cada iteración (línea 13). Después, se busca la posición para la cual el arreglo de banderas contenga el identificador de un marcador (línea 14).

Un nuevo ciclo que va por cuatro distintas orientaciones posibles y otro por las dimensiones y repeticiones del patrón en cada matriz de orientación se inician (líneas 16 a 18). En el ciclo más interno se multiplican los valores de $imgAux$ por los valores del patrón y se suman (considerando las posiciones correspondientes). Esta suma se divide entre la raíz calculada en el paso 7 de este algoritmo y la desviación estándar del Algoritmo 2, guardando el resultado en $suma_2$ (línea 20). Este valor es crucial para determinar el nivel de confianza del marcador y en caso de que se encuentre un mejor candidato para este valor, se determina el identificador del marcador (almacenado en k) y la orientación del mismo (almacenada en j), entre los pasos 21 y 24.

Una vez que los parámetros importantes del marcador son determinados, la función termina y estos valores retornan sucesivamente hasta la llamada inicial al proceso de detección del marcador, dando fin al procedimiento.

3.3.4. Calibración de la cámara y estimación de la pose

La calibración de la cámara es un proceso que consiste en determinar los parámetros intrínsecos de ésta, cuando se modela como una cámara obscura [15, 14].

Estos parámetros representan la distancia focal, factores de distorsión, factores de escalamiento, así como el punto central en el plano de la imagen y pueden englobarse en una matriz de 3×3 conocida como la matriz de calibración, representada generalmente como la matriz K.

La matriz de calibración K define la proyección en perspectiva como:

$$K = \begin{pmatrix} f_x & \alpha = 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.2}$$

donde f_x y f_y representan factores de escalamiento en x y en y respectivamente, α es un parámetro que se asocia al nivel de sesgado y en casos ideales es considerado cero, pues generalmente añade ruido al sistema. Finalmente u_0 y v_0 expresan las coordenadas del pixel que cruza el eje óptico con el plano formado por la imagen.

Aunado a los parámetros intrínsecos de la cámara, será necesario hallar una matriz que especifique la orientación (en todos sus ejes) así como un vector que describa la posición en X, Y y Z de la misma (parámetros extrínsecos). La Ecuación 3.3 expresa la proyección del espacio tridimensional al plano de la imagen bidimensional

$$\lambda \boldsymbol{p} = M \boldsymbol{P} \tag{3.3}$$

en donde M es la matriz de transformaciones necesarias para establecer una correspondencia entre los puntos $\boldsymbol{P}$ (puntos del modelo tridimensional) y los puntos $\boldsymbol{p}$ (coordenadas de los puntos en el plano de la imagen) escalados por λ. Dichas transformaciones corresponden a la proyección en perspectiva y a las rotaciones y traslaciones mencionadas anteriormente. Así, M se define como:

$$M = K[R \,|\, \boldsymbol{t}] \tag{3.4}$$

donde $[R|\boldsymbol{t}]$ es de tamaño 3×4 formada a su vez por una matriz R de 3×3 y un vector $\boldsymbol{t}$ de 3×1. Esta matriz se aplica cuando se trabaja con imágenes que no consideran distorsión. Conociendo la estructura interna de M, es posible expresar la proyección completa entre espacios tridimensionales y bidimensionales (Ecuación 3.3), como se muestra en la Ecuación 3.5.

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & \alpha & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3.5}$$

Uno de los métodos para obtener la matriz K es el denominado *calibración simple* o *autocalibración* descrito por Zhang [59], en el cual se utiliza una sola imagen y un modelo que funge como plano a proyectar, además de que algunos parámetros intrínsecos se asumen como conocidos, los cuales son u_0 y v_0 considerándolos las coordenadas del centro de la imagen.

El proceso de autocalibración requiere determinar a su vez la matriz de homografía H (la cual realiza la correspondencia entre los puntos del modelo y los puntos del marco capturado por la cámara), la cual es una matriz de 3×3 similar a la matriz de transformaciones M descrita en la Ecuación 3.4, pero su definición está dada por:

$$H = K[\boldsymbol{r_1}\ \boldsymbol{r_2}\ \boldsymbol{t}] \tag{3.6}$$

Debido a que el patrón es un modelo bidimensional, se toman solamente las primeras dos columnas de la matriz R [14]. La matriz H puede obtenerse al resolver un sistema de la forma $Ax = b$, donde A es una matriz de $2N \times 8$ entradas, *i.e.*, por cada punto en la imagen/modelo se tienen dos renglones de la matriz A, formándose así el siguiente sistema:

$$\begin{bmatrix} u_1 & v_1 & 1 & 0 & 0 & 0 & -u_1u'_1 & -v_1u'_1 \\ 0 & 0 & 0 & u_1 & v_1 & 1 & -u_1v'_1 & -v_1v'_1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -u_2u'_2 & -v_2u'_2 \\ 0 & 0 & 0 & u_2 & v_2 & 1 & -u_2v'_2 & -v_2v'_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ u_n & v_n & 1 & 0 & 0 & 0 & -u_nu'_n & -v_nu'_n \\ 0 & 0 & 0 & u_n & v_n & 1 & -u_nv'_n & -v_nv'_n \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \end{bmatrix} \begin{bmatrix} u'_1 \\ v'_1 \\ u'_2 \\ v'_2 \\ \vdots \\ u'_n \\ v'_n \end{bmatrix} \tag{3.7}$$

Al resolverse el sistema, se obtiene un vector $\boldsymbol{h}$ cuyas entradas forman parte de la matriz H, sólo faltando una entrada (pues H es de tamaño 3×3), la entrada h_{33} que tiene un valor de 1. Sin embargo, debido a que la solución del sistema de la Ecuación 3.7 puede resultar inestable, cada punto debe ser normalizado, *i.e.*, se obtiene la media y desviación estándar de todas las coordenadas en x y en y para posteriormente a cada coordenada restarle la media correspondiente ($\overline{x}$ o $\overline{y}$ según sea el caso) y dividirlo entre la desviación estándar correspondiente (σ_x o σ_y). Finalmente, debido a esta normalización, la matriz H obtenida con las entradas del vector $\boldsymbol{h}$ y la entrada h_{33} debe ser multiplicada por:

$$\begin{bmatrix} \sigma_x & 0 & \overline{x} \\ 0 & \sigma_y & \overline{y} \\ 0 & 0 & 1 \end{bmatrix} \tag{3.8}$$

Una vez obtenida H se utiliza la siguiente relación para continuar con el procedimiento de autocalibración:

$$\boldsymbol{h_1^T} W \boldsymbol{h_2} = 0 \tag{3.9}$$

donde $W = K^{-T}K^{-1}$ y es una matriz simétrica, pues:

$$W = \frac{1}{f}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -u_0 & -v_0 & f \end{bmatrix}\frac{1}{f}\begin{bmatrix} 1 & 0 & -u_0 \\ 0 & 1 & -v_0 \\ 0 & 0 & f \end{bmatrix} = \frac{1}{f^2}\begin{bmatrix} 1 & 0 & -u_0 \\ 0 & 1 & -v_0 \\ -u_0 & -v_0 & u_0^2 + v_0^2 + f^2 \end{bmatrix} \tag{3.10}$$

por lo que $w_{13} = w_{31} = -u_0$, $w_{23} = w_{32} = -v_0$ y $w_{33} = u_0^2 + v_0^2 + f^2$. De esta manera se requiere obtener f y por ende w_{33}, lo cual se logra al resolver la relación establecida en la Ecuación 3.9 una vez conocidas la mayoría de las entradas de W y los valores de H, pues se obtiene:

$$\begin{aligned} a_1 &= h_{12}(h_{11} + h_{31}w_{13}) \\ a_2 &= h_{22}(h_{21} + h_{31}w_{23}) \\ a_3 &= h_{32}(h_{11}w_{13} + h_{21}w_{23}) \\ w_{33} &= \frac{-a_1 - a_2 - a_3}{h_{31}h_{32}} \\ f &= \sqrt{w_{33} - u_0^2 - v_0^2} \end{aligned} \tag{3.11}$$

Y así se consigue f que es el último valor desconocido en la matriz K mediante el proceso de autocalibración, sin embargo, también existe el proceso de calibración completa de Zhang [59], en el cual implícitamente se utilizan las ecuaciones descritas hasta el momento para obtener la matriz de homografía y la matriz K de una sola imagen, más el proceso completo requiere de al menos cinco imágenes.

La calibración de la cámara en estos casos, se logra a partir del uso de los modelos mostrados en la Figura 3.10. Como se puede observar, el primer modelo es un patrón de puntos de 6×4 (ver Figura 3.10(a)) similar a uno de los utilizados por Vázquez del Ángel [14], mientras que el segundo consta de una rejilla formada a partir de siete líneas horizontales y nueve líneas verticales (ver Figura 3.10(b)). Ambos modelos deben presentar una separación de 40 mm entre cada línea o punto según sea el caso, tanto horizontal como verticalmente [60].

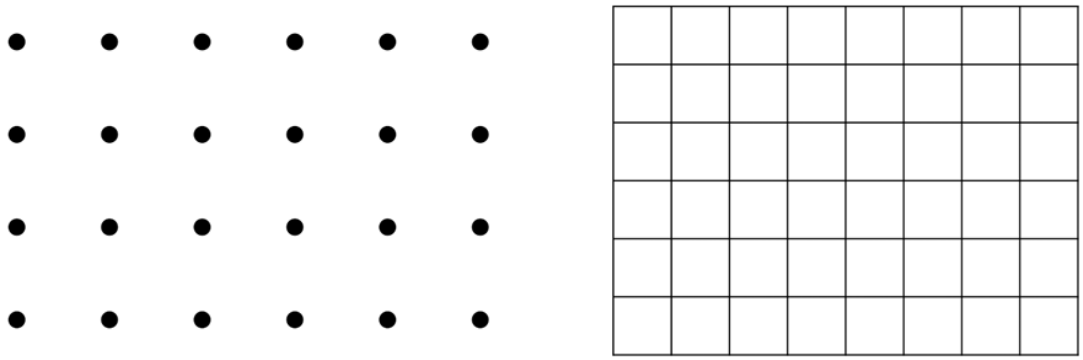


Figura 3.10: Modelos de calibración empleados para generar archivos con parámetros intrínsecos de la cámara.

El patrón de la Figura 3.10(a) determina el centro de la imagen (u_0, v_0) y la distorsión de la lente calculando el espaciado entre puntos, mientras que el modelo de la Figura 3.10(b) auxilia en el cálculo de la distancia focal de la cámara empleando la distancia entre las líneas. Ambos procesos emplean al menos cinco imágenes cada uno

(debido a que se requieren imágenes en distintos ángulos en el primer caso e imágenes a diferentes distancias con respecto a la cámara en el segundo caso), por lo que se puede entender que se trata de un proceso de calibración completa, además, puesto que el primer patrón auxilia en la obtención del centro de la imagen, se asume que es requisito para llevar a cabo el segundo proceso.

Se requiere calibrar la cámara cada vez que se cambia la escala (cada vez que se realiza un zoom). Estudiar cómo cambian los parámetros internos de la cámara según el zoom es un muy buen tema de investigación. Sobra decir que realizar este proceso sería algo muy costoso computacionalmente: se cambia el zoom, se necesitan cinco imágenes de un patrón con muchos puntos sin cambiar el zoom y se calibra la cámara.

Realizando estos procesos de calibración es como se obtienen las dimensiones de la imagen, el centro de la misma, factores de escalamiento y otros parámetros de distorsión y se guardan en un archivo. En el sistema de RA, se lee este archivo de parámetros de la cámara y se almacenan sus valores en la matriz K. Posteriormente se utiliza ésta en conjunto con *OpenGL* empleando la matriz descrita en la Ecuación 3.12, donde los valores de cerca ($zNear$) y lejos ($zFar$) se establecen al inicializar el sistema de RA.

$$M_{Proyeccion} = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2 \cdot zFar \cdot zNear}{zNear-zFar} \\ 0 & 0 & -1 & -1 \end{pmatrix} \tag{3.12}$$

El uso de estos archivos de calibración sin duda ahorra bastantes cálculos para el desarrollo de sistemas de RA en dispositivos móviles, aspecto que debe considerarse en estas plataformas, además de que acelera el flujo de las aplicaciones ya que éstas pueden iniciar ejecutando las tareas principales para las cuales fueron desarrolladas, evitando así un proceso inicial largo de captura de múltiples imágenes para lograr el procedimiento de calibración completo.

En este sistema no se consideran los factores de distorsión, por lo que sólo se aprovechan los valores correspondientes a las dimensiones del marco, los factores de escalamiento y por último, los valores del centro de la imagen.

Una vez aclarado el uso de K, hace falta comentar acerca de la obtención de las transformaciones geométricas.

Transformaciones geométricas y seguimiento de esquinas

Continuando con el proceso de autocalibración, para obtener las transformaciones geométricas (matriz de rotación y vector de traslación) se parte de una modificación a

la Ecuación 3.6, en la cual se considera un factor λ de escala en uno de sus miembros y K pasa al otro lado de la ecuación:

$$\lambda[\boldsymbol{r_1}\ \boldsymbol{r_2}\ \boldsymbol{t}] = K^{-1}H = [\boldsymbol{m_1}\ \boldsymbol{m_2}\ \boldsymbol{m_3}] \tag{3.13}$$

A partir de cualquiera de las igualdades descritas en la ecuación anterior, es posible determinar λ y así sucesivamente los vectores:

$$\begin{aligned} \lambda &= \frac{1}{\| \boldsymbol{m_1} \|} \\ \boldsymbol{r_1} &= \frac{\boldsymbol{m_1}}{\| \boldsymbol{m_1} \|} \\ \boldsymbol{r_2} &= \frac{\boldsymbol{m_2}}{\| \boldsymbol{m_2} \|} \\ \boldsymbol{r_3} &= \boldsymbol{r_1} \times \boldsymbol{r_2} \\ \boldsymbol{t} &= \frac{\boldsymbol{m_3}}{\lambda} \end{aligned} \tag{3.14}$$

Los vectores $\boldsymbol{r_1}$, $\boldsymbol{r_2}$ y $\boldsymbol{r_3}$ son las componentes de la matriz de rotación (R), sin embargo, los ángulos de rotación (o ángulos de Euler) se obtienen al realizar una SVD sobre R de la forma:

$$R = UIV^T \tag{3.15}$$

donde U y V son ortogonales. Una vez realizado este proceso, se aplican sobre las componentes de esta nueva R las ecuaciones descritas en 3.16:

$$\begin{aligned} \alpha &= \tan^{-1}\left(-\frac{r_{32}}{r_{31}}\right) \\ \beta &= \cos^{-1}(r_{33}) \\ \gamma &= \tan^{-1}\left(-\frac{r_{23}}{r_{13}}\right) \end{aligned} \tag{3.16}$$

Finalmente, la matriz R se conforma como:

$$R = R_z(\alpha) \cdot R_y(\beta) \cdot R_z(\gamma) \tag{3.17}$$

donde $R_z(\alpha)$, $R_y(\beta)$ y $R_z(\gamma)$ corresponden a las matrices:

$$\begin{aligned} R_z(\alpha) &= \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ R_y(\beta) &= \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix} \end{aligned} \tag{3.18}$$

$$R_z(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Ahora bien, los marcadores cuadrados cuyas dimensiones son conocidas se utilizan como la base para iniciar el procedimiento de obtención de transformaciones en el sistema [2], pues contienen las coordenadas importantes que se encuentran más lejos de las coordenadas de la pantalla. Posterior a ellas, se encuentran las coordenadas de la lente de la cámara, la cual captura la imagen del mundo real.

La matriz de transformaciones entre las coordenadas del marcador y las de la lente de la cámara (denotada por T_{cm}) se muestra en la Ecuación 3.19 y es estimada por medio de un análisis a la imagen (marco) capturada por la cámara del iPad.

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} & R_{3\times3} & & t_{3\times1} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix} = \boldsymbol{T_{cm}} \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix} \tag{3.19}$$

Retomando el procedimiento de la detección del marcador, tras haber realizado el proceso de umbralización sobre la imagen capturada por la cámara, se extraen las regiones cuyo contorno más externo puede ser ajustado por cuatro segmentos de línea que formarían el cuadrado. Los parámetros (a, b y c) de estas cuatro líneas, así como las coordenadas de los cuatro vértices de cada una de estas regiones encontradas a partir de las intersecciones de los segmentos de línea, se almacenan en la estructura del marcador (Tabla 3.1) para su posterior utilización.

Las regiones encontradas son normalizadas (por el método *PCA*) y el fragmento de imagen que contiene la región cuadrada es comparado (en el caso de los marcadores de plantilla) con los patrones agregados al sistema previamente, para hallar el identificador único del marcador.

En el momento en el que se pretende obtener la matriz de transformaciones geométricas, la estructura del marcador ya posee toda la información necesaria (el identificador, la dirección del patrón, las coordenadas de las esquinas y del centro del patrón, etc.) gracias al proceso descrito anteriormente por lo que ya es viable obtener los puntos de la imagen (en este caso, las esquinas) y los puntos rectificados (puntos que describen cuatro esquinas del marcador en una posición ideal) para estimar la matriz de homografía. El Algoritmo 8 muestra cómo obtener estos puntos.

Algoritmo 8 Algoritmo para obtener los puntos para la homografía.

Entrada: La información contenida en una estructura de tipo marcador.

Salidas: Las coordenadas de las esquinas y de los puntos rectificados de las mismas.

1: $direccion \leftarrow infoMarcador \rightarrow direccion$
2: $esquina_{1(x)} \leftarrow infoMarcador \rightarrow vertices[(4 - direccion)\,\%4][0]$
3: $esquina_{1(y)} \leftarrow infoMarcador \rightarrow vertices[(4 - direccion)\,\%4][1]$
4: $esquina_{2(x)} \leftarrow infoMarcador \rightarrow vertices[(5 - direccion)\,\%4][0]$
5: $esquina_{2(y)} \leftarrow infoMarcador \rightarrow vertices[(5 - direccion)\,\%4][1]$
6: $esquina_{3(x)} \leftarrow infoMarcador \rightarrow vertices[(6 - direccion)\,\%4][0]$
7: $esquina_{3(y)} \leftarrow infoMarcador \rightarrow vertices[(6 - direccion)\,\%4][1]$
8: $esquina_{4(x)} \leftarrow infoMarcador \rightarrow vertices[(7 - direccion)\,\%4][0]$
9: $esquina_{4(y)} \leftarrow infoMarcador \rightarrow vertices[(7 - direccion)\,\%4][1]$
10: $puntoRectificado_{1(x)} \leftarrow \frac{centro_x - ancho}{2}$
11: $puntoRectificado_{1(y)} \leftarrow \frac{centro_y + ancho}{2}$
12: $puntoRectificado_{2(x)} \leftarrow \frac{centro_x + ancho}{2}$
13: $puntoRectificado_{2(y)} \leftarrow \frac{centro_y + ancho}{2}$
14: $puntoRectificado_{3(x)} \leftarrow \frac{centro_x + ancho}{2}$
15: $puntoRectificado_{3(y)} \leftarrow \frac{centro_y - ancho}{2}$
16: $puntoRectificado_{4(x)} \leftarrow \frac{centro_x - ancho}{2}$
17: $puntoRectificado_{4(y)} \leftarrow \frac{centro_y - ancho}{2}$

Como se observa en dicho algoritmo, la dirección del marcador permite decidir rápidamente el orden de los vértices (recordando que *direccion* es un valor entero que comprende entre 0 y 3 debido a las 4 posibles orientaciones del marcador), mientras que el ancho del marcador (variable conocida desde la inicialización del sistema) permite determinar las coordenadas de un cuadrado ideal a partir de las coordenadas del centro del marcador restando o sumando este valor en x o en y según sea el caso.

Dados estos puntos es posible realizar el proceso de obtención de transformaciones geométricas, sin embargo, es importante considerar los planos que se forman debido a los lados paralelos del cuadrado del marcador [2], pues ayudan a que la matriz de transformaciones obtenida sea más precisa.

Considerando los valores de los parámetros de las ecuaciones de las líneas (a, b y c) encontradas en la estructura de tipo marcador (mediante el proceso de ajuste) y conociendo el orden de las esquinas para saber la correspondencia entre dichos vértices y dichos parámetros, entonces, cuando dos lados paralelos del cuadrado son proyectados en la imagen, las ecuaciones de esos dos segmentos de línea en las coordenadas de la pantalla son las siguientes (Ecuacion 3.20):

$$\begin{aligned} a_1x + b_1y + c_1 &= 0 \\ a_2x + b_2y + c_2 &= 0 \end{aligned} \tag{3.20}$$

Las ecuaciones de los planos que incluyen estos dos lados paralelos pueden representarse en coordenadas de la pantalla por medio de la matriz de homografía M descrita en la Ecuación 3.21, al sustituir x_c y y_c por x y y de la Ecuacion 3.20.

$$\boldsymbol{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & 0 \\ 0 & m_{22} & m_{23} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \lambda x_c \\ \lambda y_c \\ \lambda \\ 1 \end{bmatrix} = \boldsymbol{M} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} \tag{3.21}$$

$$\begin{aligned} a_1 m_{11} X_c + (a_1 m_{12} + b_1 m_{22}) Y_c + (a_1 m_{13} + b_1 m_{23} + c_1) Z_c = 0 \\ a_2 m_{11} X_c + (a_2 m_{12} + b_2 m_{22}) Y_c + (a_2 m_{13} + b_2 m_{23} + c_2) Z_c = 0 \end{aligned} \tag{3.22}$$

Estos dos planos tendrán cada uno un vector normal ($\boldsymbol{n_1}$ y $\boldsymbol{n_2}$ respectivamente) y el vector de dirección de los dos lados paralelos del cuadrado estará dado por el producto cruz $\boldsymbol{n_1} \times \boldsymbol{n_2}$.

Además, considerando los dos pares de planos paralelos que tiene el cuadrado, tendremos dos vectores de dirección y por tanto, dos vectores unitarios ($\boldsymbol{u_1}$ y $\boldsymbol{u_2}$ respectivamente), por lo que tendremos que asegurarnos de que estos vectores sean perpendiculares. Sin embargo, los errores producidos durante el procesamiento de la imagen indicarán que los vectores no son exactamente perpendiculares, por lo que para compensar este error se definen otros dos vectores ($\boldsymbol{v_1}$ y $\boldsymbol{v_2}$) que son perpendiculares y unitarios en el mismo plano que $\boldsymbol{u_1}$ y $\boldsymbol{u_2}$ (como se muestra en la Figura 3.11).

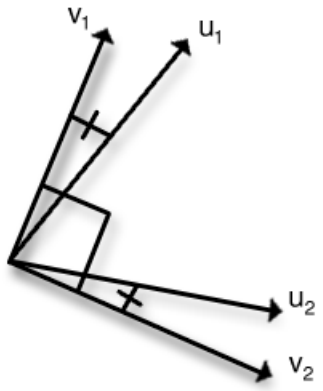


Figura 3.11: Dos vectores perpendiculares unitarios: $\boldsymbol{v_1}$ y $\boldsymbol{v_2}$ calculados a partir de $\boldsymbol{u_1}$ y $\boldsymbol{u_2}$.

Estos vectores agregados deberán tener un vector unitario que sea perpendicular a ambos (denotado como $\boldsymbol{v_3}$) y son estos vectores los que se corresponden con la componente de rotación $R_{3\times3}$ en la matriz de transformación T_{cm} definida en la Ecuación 3.19, *i.e.*, $[\boldsymbol{v_1^t} \quad \boldsymbol{v_2^t} \quad \boldsymbol{v_3^t}]$.

Al conocer a la componente $R_{3\times3}$ en la matriz de transformaciones, si se utiliza la Ecuación 3.19 y la 3.21 junto con las coordenadas de las cuatro esquinas del marcador

en el marco actual capturado y esas mismas coordenadas pero en su representación en coordenadas de la pantalla del dispositivo (puntos rectificados), se generarán 8 ecuaciones incluyendo las componentes de traslación (t_x, t_y y t_z) y los valores de dichas componentes pueden ser obtenidos mediante estas ecuaciones. De esta manera, considerar los dos planos que incluyen los lados paralelos del cuadrado permiten realizar una estimación previa de la matriz de rotación y del vector de traslación.

Una vez logrado esto, es posible determinar los ángulos de Euler (Ecuación 3.16) y obtener la matriz de rotación empleando su forma completa (Ecuación 3.17), evitando realizar una *SVD* sobre la matriz de rotación. Finalmente, al obtener esta matriz en su forma completa, se modifica la matriz de homografía.

Y es de esta manera como se estima la posición del marcador en la escena. Posteriormente, la matriz de transformaciones se adapta al formato utilizado por *OpenGL*, concluyendo así las tres operaciones principales que realiza el sistema de RA.

3.4. Manejo de información previa

Sin duda dos aspectos sumamente importantes a tomar en cuenta durante el seguimiento de los marcadores con la finalidad de evitar cálculos innecesarios son:

- Mantener el identificador del marcador el mayor tiempo posible.
- Evitar recalcular la matriz de homografía hasta que sea necesario.

Las razones son sencillas, en cuanto al identificador del marcador, si se mantiene información previa sobre este valor, una vez que el proceso de detección del marcador halle uno nuevo en la escena, éste debe ser comparado con el que se tenía anteriormente y es entonces que el nivel de confianza (ver Tabla 3.1) toma importancia, pues al ser un valor probabilístico (entre 0 y 1) se acepta el de mayor valor numérico.

Por lo tanto, si el nivel de confianza del marcador detectado es menor que el del marcador que se había detectado previamente, entonces el sistema indicará que el marcador no ha cambiado y así, la aplicación puede valerse de esta información para evitar la carga de un modelo 3D en la escena que podría resultar ser el mismo objeto que ya se había cargado y dibujado en el marco inmediato anterior o tal vez un poco antes. Las aplicaciones basan su lógica de carga y generación de objetos 3D con base en el identificador devuelto por el sistema de RA como se verá más adelante.

Con respecto a la matriz de homografía, también es posible determinar el momento en el que surge la necesidad de volver a estimarla, empleando un valor numérico correspondiente a un nivel de error el cual actúa de manera similar a como lo hace el nivel de confianza en el caso del identificador del marcador.

La matriz de transformaciones encontrada a partir del método descrito a lo largo de este capítulo puede generar algún error, pero dicho error puede reducirse al aplicar el siguiente procedimiento:

- Las coordenadas de los vértices del marcador en el mundo real se transforman a coordenadas de la pantalla del dispositivo mediante la matriz de transformaciones obtenida en un estado previo.
- Luego, la matriz de transformaciones se optimiza por medio de la suma de las diferencias entre estas coordenadas transformadas y las coordenadas obtenidas a partir de la imagen, hasta llegar a un valor mínimo o despreciable.

Este valor mínimo se emplea para decidir si la matriz debe recalcularse o no, lo cual es vital en casos en los que el marcador no se haya movido de manera considerable, *i.e.*, fácilmente perceptible.

A pesar de que la matriz de transformaciones tiene seis grados de libertad, sólo las componentes que representan a las rotaciones se optimizan y las componentes de traslación vuelven a estimarse utilizando el procedimiento mencionado en la sección 3.3.4 (p. 54). Conforme se va iterando, la matriz de transformaciones se vuelve más precisa.

Capítulo 4

Aplicación 1: Fascículo de especies

En la Figura 4.1 se muestra un diagrama que resume el funcionamiento de la aplicación. El *Delegado de la aplicación* es un objeto necesario en toda aplicación de *Objective-C* para lidiar con eventos concernientes a la misma y no tanto al comportamiento deseado, *e.g.*, pasar el proceso a segundo plano, volver a primer plano, realizar acciones al finalizar la carga de la aplicación, etc. Este objeto debe crear a su vez otro objeto conocido como *Controlador principal de la vista*, cuya clase actúa como la principal del programa, pues en ella se deben integrar todos los métodos necesarios para lograr el comportamiento esperado de la aplicación.

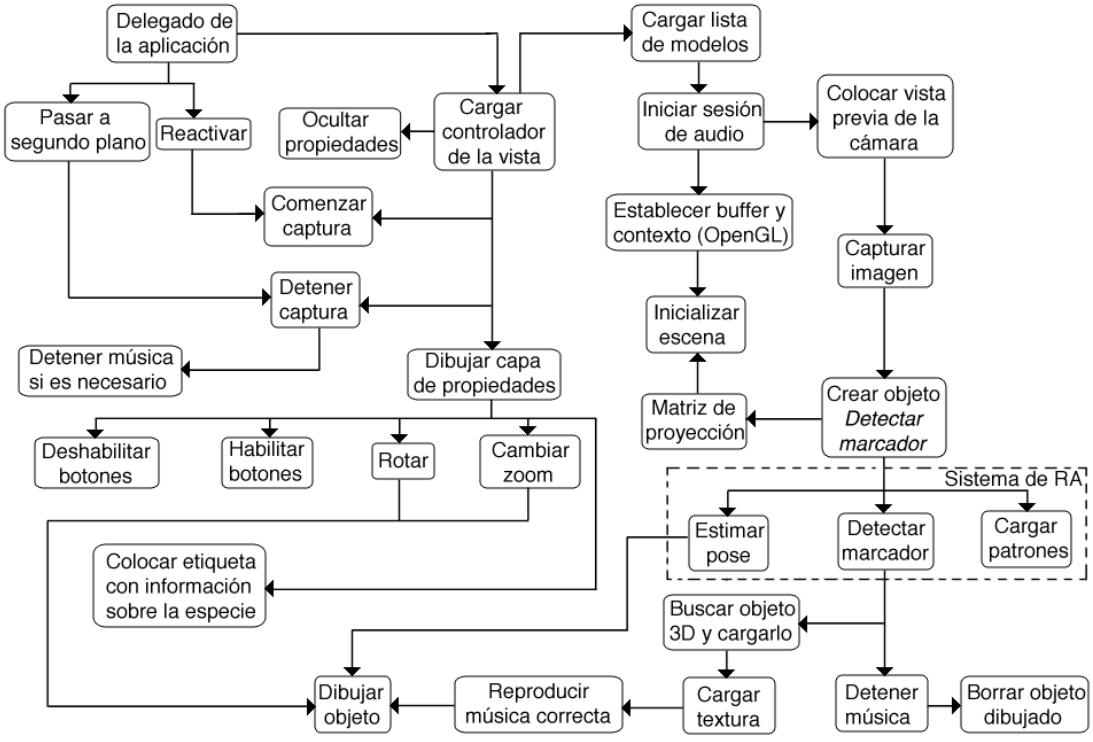


Figura 4.1: Diagrama general de la primera aplicación.

En el caso de *Fascículo de especies*, el controlador es capaz de comenzar o detener las sesiones de audio y captura de video (colocando la vista previa de la cámara) por medio de delegados exclusivos de *Objective-C*, además de que crea la capa necesaria para dibujar con *OpenGL ES* (iniciando la escena) y despliega la capa de propiedades de la especie dibujada en pantalla. Además, crea el objeto *Detectar marcador*, el cual se conecta con el sistema de RA (en la parte inferior derecha del diagrama), accediendo a sus acciones principales (ver Figura 3.4 (p. 36)).

Una vez que este objeto obtiene el identificador único asociado a un marcador encontrado en la escena, se busca (en una lista de modelos 3D previamente cargada) a qué modelo 3D le pertenece dicho identificador para cargar dicho objeto junto con una textura asociada al mismo y así dibujarlo en la escena.

Mediante esta aplicación es posible pasar el iPad cerca de uno de los marcadores de la libreta para que una especie animal se dibuje sobre la imagen de la vista previa de la cámara en la pantalla. Adicionalmente, se puede tocar la pantalla para que aparezca un menú con información sobre la especie dibujada, además, posee un botón y una barra deslizante mediante los cuales es posible animar (rotar sobre el eje Z el modelo) o escalar la figura. Estas características son manejadas por la capa de propiedades del objeto 3D.

4.1. Objetos 3D

Realizar un modelo 3D de calidad directamente mediante primitivas de *OpenGL ES* es claramente una tarea compleja y más si se trata de modelos referentes a especies animales, por lo que es más adecuado emplear modelos gratuitos y de libre distribución que hayan sido desarrollados por diseñadores y especialistas en el modelado 3D.

Estos modelos se encuentran definidos en formato *.obj*. Este tipo de archivos indican la posición de cada uno de los vértices del modelo, las coordenadas de la textura empleada considerando su mapeo con respecto a los vértices del modelo y se indican las caras, en un formato $v_x/vt_x/vn_x$, donde v_x corresponde al vértice, vt_x representa el mapeo de la textura para v_x y vn_x la normal en dicho punto.

Sin embargo, debido al formato en que se presentan los datos y a que se pueden describir caras con más de tres vértices (además de poder definirse efectos), *OpenGL ES* no puede cargar algún objeto definido mediante estos archivos de manera directa, por lo que es necesario colocar la información en arreglos de vértices, normales, etc., lo cual se puede hacer en un archivo de cabecera (*.h*) idéntico a los del lenguaje *C* en el que se escriban las variables que representen dichos arreglos.

Opcionalmente podría realizarse un intérprete en tiempo real de la información descrita en los archivos *.obj* para adaptar su contenido a *OpenGL ES*, sin embargo,

realizar este proceso disminuiría la velocidad de ejecución de la aplicación y más si se implementa de forma que se interprete el archivo *.obj* cada vez que se detecte un marcador.

Los modelos 3D de especies que se manejan en esta aplicación fueron descargados de distintos sitios en la red [61], de entre los cuales destacan *TurboSquid* [62], *top3Dmodels* [63], *archive3D* [64], entre otros.

Una vez que se haya descargado un modelo en formato *.obj*, debe leerse e interpretarse correctamente para convertir la información ahí descrita en un archivo de cabecera con variables que describan vértices, normales y coordenadas para la textura pertenecientes al modelo. El *script* de *Heiko Behrens* [65] es uno de los más utilizados para lograr este cometido ya que está enfocado en el uso de *OpenGL ES* en *iOS*.

Este *script* genérico y de uso gratuito desarrollado en *Perl* lee el archivo en formato *.obj* y estructura los arreglos de vértices, normales y coordenadas para la textura, que son necesarios para el modelo. Además, ubica las coordenadas en el rango $[-1, 1]$ (tal como trabaja *OpenGL*), por lo que carece de importancia si el modelo fue trabajado de una manera distinta mediante el software de modelado. Sin embargo, el *script* se limita al uso exclusivo de caras triangulares dado que así es como *OpenGL ES* trabaja, por lo que los modelos originales deben cumplir con esta restricción. De igual forma, se limita al uso de una sola textura para todo el modelo y no acepta el uso de archivos de materiales que comúnmente se emplean en los programas de modelado 3D, para evitar así el uso de algunas texturas. Desafortunadamente, hallar un modelo que cumpla con todas estas restricciones no ocurre tan fácilmente, por lo que será necesario aprender un poco sobre algún software de modelado 3D, preferentemente uno de software libre, como *Blender* [66]. La Figura 4.2 muestra un ejemplo del uso de *Blender* para tratar un modelo 3D.

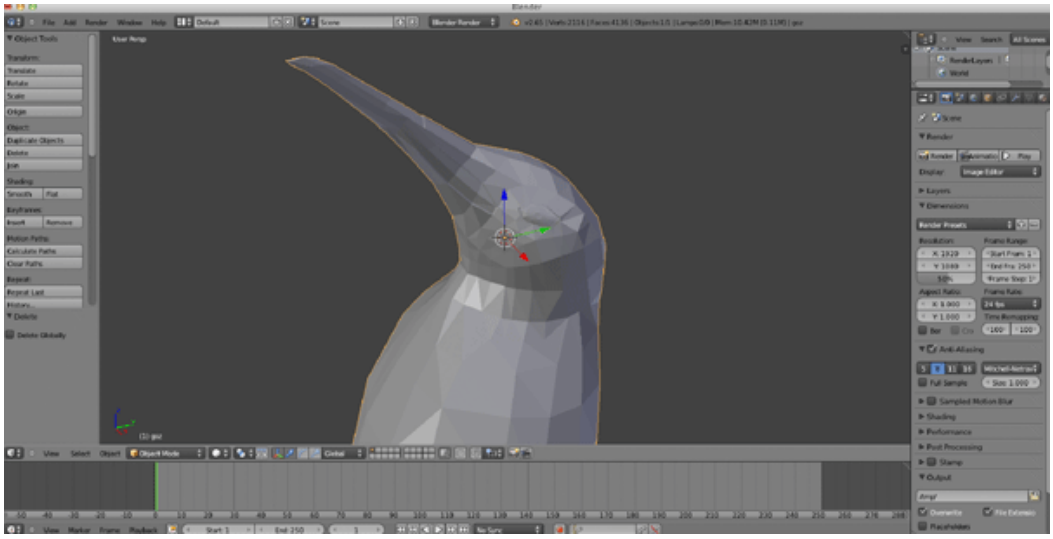


Figura 4.2: Ejemplo del uso de *Blender* para tratar un modelo 3D utilizado en la aplicación.

Será necesario al menos conocer lo básico en el uso de estos softwares para solucionar imprevistos (que no se pueden saber *a priori* cuando se descarga el modelo) tales como los que se describen a continuación:

- El modelo no tiene las caras formadas por triángulos, por lo que tendrá que volverse a exportar como *.obj* seleccionando la opción de triangular las caras.
- El modelo no está en formato *.obj* sino en otro formato, pero puede ser abierto por el programa y vuelto a exportar como *.obj*.
- El modelo fue generado incluyendo archivos de materiales aunque no utilice ninguno: en este caso tendrá que volverse a exportar como *.obj* excluyendo la creación de archivos de materiales.
- El modelo descargado, por alguna razón contiene la textura y las coordenadas para la textura, sin embargo el mapeo de dichas coordenadas con respecto al modelo no se realizó o se perdió dicha información al exportarlo como *.obj*, por lo que habrá que indicarle cuál es la imagen de textura que le corresponde para que ésta se coloque sobre el modelo y posteriormente debe exportarse.
- El modelo contiene normales, pero dicha información no se incluyó en el archivo *.obj*, por lo que es necesario volverlo a exportar incluyendo las normales.
- El modelo fue rotado y no está del modo que desearíamos que se encontrase: en este caso, podemos rotarlo a nuestro gusto por medio de comandos del software y volverlo a exportar. Se trata de un caso opcional, sin embargo el *script* de *Perl*, a pesar de solucionar problemas de escalamiento al ubicar los vértices en el rango $[-1, 1]$, conserva la posición del modelo con respecto al origen del sistema de coordenadas, por lo que si está rotado de alguna manera que no deseáramos, conserva esta posición.
- El modelo no está suavizado y los polígonos son muy notorios: en este caso deberá volverse a exportar utilizando la opción para suavizar. Lo que sucede en este caso es que se promedian los vértices para espaciarlos de manera más uniforme, no se reducen ni agregan vértices pero al moverlos para distribuirlos, algunos de ellos cambiarán sus coordenadas [67].
- El modelo presenta más de una textura, pero existe la posibilidad de mapearlo sobre una sola: este caso requerirá un conocimiento más avanzado del software, además de la modificación de la textura original para agregar los segmentos pertenecientes a otras texturas en una única, empleando algún otro programa de edición de imágenes.
- El modelo utiliza además de una textura, algún material, pero es posible mapear esa parte del modelo hacia la textura y eliminar el material: este último caso es similar al anterior y requerirá un uso más avanzado del software de modelado.

Adicionalmente, será aconsejable reducir las dimensiones de la imagen de textura para que el tamaño del archivo también se reduzca, pues muchas veces los modelos incluyen texturas de hasta 25MB que pueden ser reducidas a menos de 512Kb sin que se perciba a simple vista, además, no afecta al mapeo de coordenadas de textura. A algunos modelos utilizados en este trabajo tuvieron que realizárseles las correcciones correspondientes para ser empleados en la aplicación.

4.1.1. Dibujando el objeto

Una vez que mediante el *script* se crea el archivo de cabecera (*.h*) y se incluye entre los archivos que conforman la aplicación, puede utilizarse en el momento en el que se detecte un marcador en la escena, sin embargo es necesario describir una estructura que modele lo necesario para representar un objeto 3D en la aplicación. La Tabla 4.1 muestra los campos necesarios para ello.

Tipo	Campo	Descripción
`NSString *`	**nombre**	Nombre de la especie
`float`	**factorDeEscala**	Escala del modelo
`float`	**traslacionZ**	Traslación del objeto en el eje Z
`NSString *`	**archivoTextura**	Nombre de la imagen de textura del modelo
`NSString *`	**descripcion**	Descripción de la especie
`unsigned int`	**numeroDeVertices**	Número total de vértices del modelo
`GLfloat *`	**vertices**	Coordenadas de los vértices del modelo
`GLfloat *`	**normales**	Coordenadas de las normales del modelo
`GLfloat *`	**coordenadasParaTextura**	Coordenadas de la imagen de textura que se aplican sobre el modelo

Tabla 4.1: Campos de la estructura que representa un objeto 3D.

De los campos descritos en dicha tabla, los últimos cuatro son los que tienen relación directa con el archivo de cabecera del modelo, pues representan los vértices, normales y coordenadas para la textura. Los demás por su parte, reflejan propiedades del objeto que se obtienen a partir de un archivo (en formato *.plist*) que enlista los modelos existentes en el sistema. En este archivo se indica cuál es la llave única que se asocia a cada modelo.

Al detectar un marcador en la escena se obtiene el identificador único, el cual es buscado en la lista de modelos existentes para conocer todas las propiedades del objeto (nombre, factor de escala, descripción, etc.) y cargar una instancia del objeto 3D. El Algoritmo 9 describe los pasos a seguir para cargar un objeto 3D en la aplicación una vez detectado un marcador en la imagen.

Lo primero que se debe hacer es deshabilitar el cliente de *OpenGL ES* y dejar de pintar la textura de un objeto si es que hay uno en escena para que no se superponga

un objeto 3D sobre otro. Posteriormente se habilita el cliente de *OpenGL ES* para aceptar normales y texturas. En este procedimiento se coloca la textura del objeto para que pueda cargarse por completo antes de ser dibujado en la escena. Para colocar una textura al objeto 3D se siguen los pasos mostrados en el Algoritmo 10.

Algoritmo 9 Algoritmo que carga un objeto 3D.

Entrada: Un objeto 3D.
Salida: El dibujo del objeto sobre la capa de *OpenGL ES*.

```
1: Deshabilitar el cliente de OpenGL ES.
2: Borrar la textura actual.
3: self.objeto ← objeto
4: Habilitar cliente de OpenGL ES.
5: if el objeto tiene normales then
6:     Habilitar normales.
7:     glNormalPointer(GL_FLOAT,0,self.objeto.normales)
8: if el objeto presenta coordenadas para textura then
9:     Colocar textura.
10:    Habilitar texturas.
11:    glTexCoordPointer(2,GL_FLOAT,0,self.objeto.coordenadasParaTextura)
```

El nombre de la imagen que contiene la textura del objeto se obtiene en el controlador principal de la vista y se envía a la capa de *OpenGL ES* por medio del atributo *archivoTextura* del objeto 3D (Tabla 4.1).

Algoritmo 10 Algoritmo que coloca una textura a un objeto 3D.

Entrada: Ninguna.
Salida: Colocar una textura en el arreglo general de texturas.

```
1: Leer imagen de textura a partir de un objeto 3D.
2: if no se encuentra la imagen de textura then
3:     return
4: anchoTextura ← CGImageWidth(imagenDeTextura)
5: altoTextura ← CGImageHeight(imagenDeTextura)
6: contenidoTextura ← malloc(anchoTextura × altoTextura × 4)
7: Crear contexto.
8: Dibujar textura en el contexto.
9: Agregar textura al arreglo general de texturas.
10: Definir propiedades de textura.
11: Liberar memoria del contexto.
12: Habilitar texturas en el cliente OpenGL ES.
```

Ya que sólo se permite una textura por modelo, ésta deberá contener todos los elementos necesarios para que el modelo se vea lo más completo posible, en ocasiones, aún con segmentos que tal vez ni sean visibles en la escena, como en el caso de las

patas o el interior de la boca de algunas especies. La imagen generalmente muestra al modelo extendido o de perfil como si se tratase de una figura armable. Un ejemplo de textura se muestra en la Figura 4.3. Mediante el arreglo de coordenadas de textura, se mapean correctamente los vértices de acuerdo a las coordenadas de la imagen y las caras del modelo 3D.

Figura 4.3: Ejemplo de imagen de textura de un modelo.

Después de colocar la textura del objeto, se obtiene la matriz del modelo a partir del sistema de RA y se envía a la capa de *OpenGL ES*, la cual es necesaria para dibujar el modelo sobre la misma, tal como se muestra en el Algoritmo 11.

Algoritmo 11 Algoritmo que dibuja el objeto 3D en la capa de *OpenGL ES*.

Entrada: Ninguna.

Salida: Dibuja el objeto 3D en la capa de *OpenGL ES*.

```
1: if hay matriz del modelo-vista y objeto 3D then
2:     Establecer el contexto y búfer para frames.
3:     glClearColor(0.0f,0.0f,0.0f,0.0f)
4:     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
5:     Cargar la matriz del modelo-vista.
6:     Escalar el objeto.
7:     Rotar y trasladar el objeto.
8:     Dibujar el objeto (glDrawArrays) y establecer búfer para dibujar.
9:     [contexto presentRenderbuffer:GL_RENDERBUFFER_OES]
```

4.2. Conexión con el sistema de RA

En la sección 3.3 se describieron tres acciones principales que puede realizar el sistema de RA, lo necesario en este punto es llamar a esas tres acciones mediante una clase dentro de la aplicación.

Dos métodos serán necesarios para dicha tarea, el primero se encarga de inicializar el sistema de RA (Algoritmo 12) y el segundo de detectar marcadores en la escena (Algoritmo 13). Estos métodos se hayan en la clase *Detectar marcador* y por ende, se debe crear un objeto de dicha clase para emplearlos (ver diagrama de la Figura 4.1). En la Figura 4.1 se omitió el método para inicializar el sistema de RA para no dejar de resaltar sólo las tres acciones principales de éste.

Algoritmo 12 Algoritmo que inicializa el sistema de RA.

Entrada: Una imagen capturada desde la cámara del iPad.
Salida: Una instancia del sistema inicializada.

1: $ancho \leftarrow$ `CVPixelBufferGetWidth`$(imagen)$
2: $alto \leftarrow$ `CVPixelBufferGetHeigth`$(imagen)$
3: **if** ya existe instancia del sistema **then**
4: Eliminar instancia.
5: $sistema \leftarrow new \;\; sistema(ancho, alto, 1, 16, 16, 64, 17)$
6: Buscar archivo con parámetros de la cámara.
7: Crear arreglo con los nombres de las especies.
8: $sistema \rightarrow inicializar(rutaArchivoCamara, 0.1f, 10000.0f)$
9: $sistema \rightarrow cambiarElTamDelMarcoDeLaCamara(ancho, alto)$
10: $sistema \rightarrow establecerTipoDeMarcador(PLANTILLA)$
11: $\bigvee especie \in especies$
12: Buscar archivo de tipo patrón
13: $sistema \rightarrow agregarPatron(rutaArchivoPatron)$
14: **for** $i = 0 \rightarrow 16$ **do**
15: $matrizDeProyeccion$ `add`:$sistema \rightarrow obtenerMatrizDeProyeccion()[i]$
16: [`self.delegado detectarMarcador:self inicializaConMatriz:` matrizDeProyeccion]

La clase *Detectar marcador* sólo consta de cuatro métodos, entre los cuales se encuentran `init` y `dealloc`, los otros dos métodos son los que realmente se conectan con el sistema de RA. A ambos métodos se envía el búfer con la imagen de la vista previa de la cámara.

En el primer método se obtienen las dimensiones de la imagen de la vista previa de la cámara para ser enviadas al sistema de RA y se crea una instancia del sistema. Para ello, se requiere conocer las dimensiones de la imagen de la vista previa, así como las dimensiones del marcador en pixeles (en este caso 16×16).

También se requiere saber el número máximo de marcadores que se esperan en escena (un marcador máximo para esta aplicación) y el valor máximo que puede tomar el número de muestras en el proceso de obtención del patrón interior a partir de la imagen (en este caso 64), pues en ese paso de la detección se disminuye la resolución del marcador para lograr la coincidencia con los almacenados en memoria. Finalmente, también se requiere conocer el número máximo de marcadores que se cargarán en el sistema (17 especies).

Posteriormente se busca el archivo con los parámetros de la cámara (matriz K) y se indican las distancias mínima y máxima para considerar la detección ($zNear$ y $zFar$ para *OpenGL*), además del tipo de marcador a utilizar (de plantilla para esta aplicación). Se agregan los patrones al sistema y se obtiene la matriz de proyección para enviarla al delegado del controlador principal de la vista.

Algoritmo 13 Algoritmo que llama al sistema de RA para detectar un marcador.

Entrada: Una imagen capturada desde la cámara del iPad.
Salidas: El id del marcador detectado y la matriz del modelo.

1: $idMarcador \leftarrow -1$
2: $marcadoresDetectados \leftarrow sistema \rightarrow detectarMarcador(imagen)$
3: $sistema \rightarrow seleccionarMarcadorPorConfianza()$
4: **for** $i = 0 \rightarrow 16$ **do**
5: $\quad matrizDelModelo$ `add`:$sistema \rightarrow obtenerMatrizDelModelo()[i]$
6: [`self.delegado detectarMarcador:self detectaMarcadorConID`: idMarcador `yMatrizDelModelo`:matrizDelModelo]

Por su parte, el método que conecta con la detección de marcadores, es llamado constantemente a diferencia del anterior, que sólo es llamado una vez. El marcador detectado por el sistema se selecciona de acuerdo al nivel de confianza y se obtiene la matriz del modelo para enviarlos al controlador principal de la vista.

4.3. Controlador principal de la vista

La clase *controlador de la vista* en la Figura 4.1 es el núcleo de la aplicación, puesto que el *delegado de la aplicación* es una clase por omisión (que se crea para poder conectar la aplicación con la ventana principal necesaria para que el programa pueda iniciarse), sin embargo es necesario establecer un control sobre cada vista y dado que existe una vista principal, la clase asociada a ella conforma el núcleo de la aplicación.

El controlador de la vista se encarga entre otras funciones de conectarse con la cámara para obtener la imagen de la vista previa (ver sección 4.3.1), interactúa con la vista para modificar las propiedades del objeto 3D, se comunica con la capa de *OpenGL ES* y con el delegado capaz de detectar marcadores en la escena, cargando

los objetos 3D en caso de que se detecte alguno. Además, enriquece con pistas de audio a la aplicación (ver sección 4.3.2).

4.3.1. Captura de video

Para crear un delegado que se encargue de la captura de la vista previa de la cámara, se requiere establecer una sesión de captura, por lo que la aplicación hace uso de las clases por defecto AVCaptureSession y AVCaptureVideoPreviewLayer de *Objective-C*.

En esta sesión se establece el formato de pixeles (orden de los canales de color en cada pixel) de los marcos de video, se descartan los marcos que toman mucho tiempo en ser capturados, además de que al ser un proceso independiente, se agrega a una cola de procesos separada del hilo principal y la sesión se inicia y detiene mediante los métodos startRunning y stopRunning pertenecientes a la clase.

Algoritmo 14 Algoritmo para obtener el objeto 3D de acuerdo al id detectado.

Entradas: El id de un marcador y la matriz del modelo.
Salida: El objeto 3D a dibujar en pantalla.

```
1: if idMarcador ≠ −1 then
2:     if idMarcador ≠ idMarcadorPrevio then
3:         Leer el archivo de modelos y tomar la casilla cuyo id = idMarcador.
4:         Guardar propiedades del objeto en NSString, NSURL, float, etc.
5:         if la música se está reproduciendo then
6:             Detener la música y liberar memoria.
7:         Crear un objeto 3D con las propiedades obtenidas en el paso 4.
8:         Reproducir archivo de música asociado al objeto.
9:         Cargar objeto 3D.
10:        if la vista que muestra las propiedades del objeto está visible then
11:            Volver a dibujar la vista actualizando las propiedades del objeto.
12:    Establecer la matriz del modelo-vista.
13:    if la capa de OpenGL ES no está hasta el frente then
14:        Añadir capa a la vista.
15: else                                   ▷ El marcador vuelve a ser −1
16:    if la música se está reproduciendo then
17:        Detener la música y liberar memoria.
18:    if la vista que muestra las propiedades del objeto está visible then
19:        Volver a dibujar la vista actualizando las propiedades del objeto.
20:    if la capa de OpenGL ES está hasta el frente then
21:        Removerla del frente.
```

En cuanto a la vista previa, se define la posición sobre el eje Z de la imagen capturada para permitir alejarla o acercarla. Esta vista se sobrepone sobre la vista

principal, *i.e.*, sobre aquella asociada a la clase *Controlador de la vista*.

Definido lo anterior, se utiliza el método *captureOutput* para modificar el comportamiento de la aplicación una vez recibido un marco de la cámara, para lo cual simplemente si la sesión de captura está disponible, se envía la imagen al delegado de la detección de marcadores de inmediato, en caso de que éste haya sido inicializado. En caso contrario, se trataría del primer marco y se inicializa el delegado, *i.e.*, se decide a cuál de los dos algoritmos descritos en la sección 4.2 se debe llamar.

Una vez que el delegado de la detección de marcadores devuelve un identificador, se llama al método principal de la clase *Controlador de la vista*, el cual se describe en el Algoritmo 14. En este algoritmo se decide cuál es el modelo que debe dibujarse sobre la vista previa de la cámara y se dibuja de acuerdo a los algoritmos descritos en la sección 4.1.1.

Como se puede observar, el algoritmo busca a través del archivo de modelos (*.plist*) el modelo asociado al marcador entregado, en caso de no ser -1 (el identificador por omisión) para luego crear un objeto 3D y cargar el mismo.

En el momento en el que se obtienen las propiedades del objeto 3D (paso 4), se obtiene también el atributo *nombre* del objeto 3D, según los campos que éste puede almacenar (Tabla 4.1). Dicho atributo es enviado a la clase *Objeto 3D*, en donde se busca el archivo de cabecera correspondiente y se terminan de almacenar los últimos atributos del objeto 3D (normales, coordenadas para textura, etc.) para cargarlo.

Este algoritmo también se encarga de controlar la vista de *OpenGL ES* y la vista que muestra las propiedades del objeto. Para esta última se encargará de indicar si es momento de mostrar u ocultar esta vista de propiedades de acuerdo a si la pantalla del iPad es tocada. Ya será el controlador propio de la vista de propiedades del objeto quien se encargue de habilitar o deshabilitar el botón para animar el objeto y la barra de escalamiento según haya o no un objeto 3D en escena, así como de mostrar un texto con la descripción de la especie si es el caso.

4.3.2. Reproduciendo audio en la aplicación

Para reproducir audio en la aplicación se requiere instanciar un objeto de la clase AVAudioPlayer y utilizar una sesión de audio mediante AVAudioSession (clases por defecto en *Objective-C*), la cual debe activarse o desactivarse en funciones creadas por defecto, tales como *viewDidLoad*, *viewWillDisappear*, etc.

El AVAudioPlayer debe controlar la reproducción de audio empleando los procedimientos señalados en los pasos 5 y 6 del Algoritmo 14 en los momentos pertinentes, *i.e.*, solamente si el objeto 3D está visible en pantalla se reproduce la pista de audio.

Todo el procedimiento ocurre en la clase *Controlador de la vista*.

De igual forma, mediante el atributo *nombre* del objeto 3D obtenido en el paso 4 del Algoritmo 14, se busca un archivo de audio (en formato *.mp3*) que será reproducido después de crear el objeto 3D (paso 8).

Cabe mencionar que cada especie del fascículo tiene asociado un archivo de audio distinto que se reproduce hasta cinco veces continuas mientras el modelo asociado esté visible. Se utilizaron pistas de menos de un minuto de duración, editadas a partir de pistas de audio más largas mediante un software de edición de audio.

4.4. Modo de uso de la aplicación

El *Fascículo de especies* es una aplicación dirigida a un público infantil preferentemente entre los 8 y 13 años de edad, más no se restringe a éstos. Debido a ello se decidió utilizar los marcadores de plantilla (explicados en la sección 2.4.1, en la página 20), aun sabiendo que mientras más marcadores se incluyan, el sistema debe hacer más comparaciones. Como estos marcadores encierran una figura libre, ésta puede asociarse en este caso a la especie animal dibujada sobre la pantalla del dispositivo, aunado a ello, la audiencia a la que se dirige el proyecto tiende a ser más visual.

Figura 4.4: Icono de la aplicación *Fascículo de especies*.

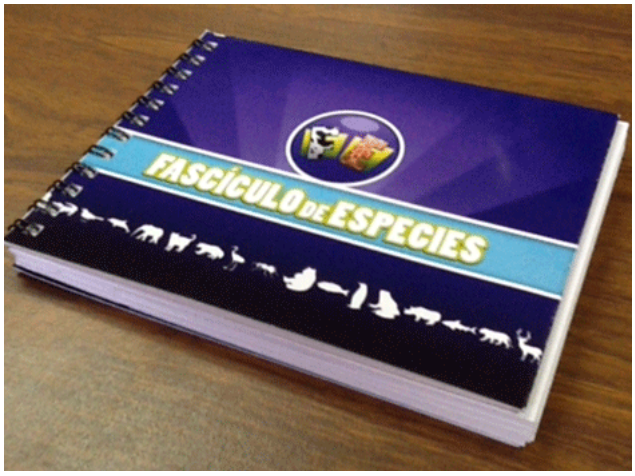


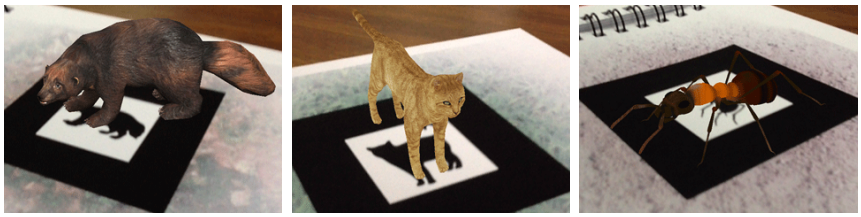
Figura 4.5: Fascículo de especies.

La aplicación inicia al presionar en el iPad el icono mostrado en la Figura 4.4 y una vez iniciada se visualizará en la pantalla del iPad la imagen capturada por la cámara, pero la verdadera interacción comienza al abrir el fascículo y pasar el iPad sobre la libreta. El fascículo de especies se muestra en la Figura 4.5. Esta pequeña libreta de 14×10 *cm*. aproximadamente contiene 18 hojas, entre las cuales (omitiendo la portada y contraportada) se tienen 17 especies.

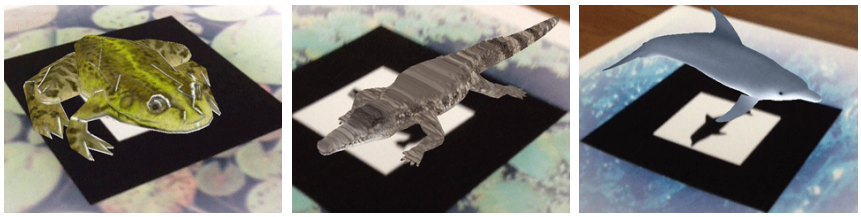
(a) cuervo. (b) búho nival. (c) grulla.

(d) pingüino rey. (e) elefante. (f) rinoceronte.

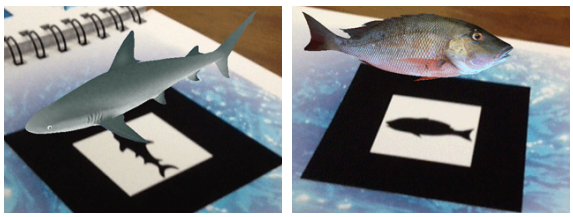
(g) oso panda. (h) tigre. (i) ciervo.



(j) glotón. (k) gato. (l) hormiga roja.



(m) rana. (n) cocodrilo. (ñ) delfín.



(o) tiburón. (p) pez.

Figura 4.6: Animales del fascículo de especies.

En las páginas impares se muestra el nombre y una imagen sobre el animal, además de un pequeño recuadro con información curiosa sobre dicha especie, mientras que en las páginas pares se encuentra el marcador asociado al animal expuesto en la página que le precede. En el apéndice de este documento se muestran todas las páginas que conforman el fascículo.

El fascículo comienza a partir de algunos animales aéreos, posteriormente varios terrestres y finalmente algunos marítimos, entre ellos se encuentran: el cuervo, el búho nival, la grulla, el pingüino rey, el elefante, el rinoceronte, el oso panda, el tigre, el ciervo, el glotón, el gato, la hormiga roja, la rana, el cocodrilo, el delfín, el tiburón y el pez. En la Figural 4.6 se muestran los modelos 3D de todos los animales del fascículo de especies al pasar el iPad sobre la libreta, en el área donde es visible el marcador de la especie correspondiente.

Al tocar la pantalla del iPad aparece la interfaz para modificar las propiedades del modelo 3D o desaparece en caso de que estuviese visible, independientemente de si está dibujado un objeto 3D en pantalla o no.

Al aparecer un modelo 3D en pantalla es posible animarlo de una manera sencilla al hacerlo rotar sobre su propio eje (el eje Z del objeto 3D, el cual apunta hacia arriba, igual que en las coordenadas del marcador) al presionar un botón con el símbolo de dos flechas que simbolizan rotación, además, se puede modificar una de las propiedades principales del modelo: el escalamiento, lo cual nos permitirá hacer más grande o más pequeño el modelo aun cuando esté siendo animado.

Figura 4.7: Ejemplos de etiquetas con la descripción de la especie.

A su vez, mientras sea visible la interfaz de propiedades, aparecerá una etiqueta con ciertos datos relativos a la especie (los cuales pueden variar de especie en especie). Esta etiqueta corresponde a una pequeña descripción del modelo 3D, la cual se encuentra en el archivo que enlista los modelos y es obtenida en el Algoritmo 14 para ser almacenada en el atributo *descripción* del objeto 3D (Tabla 4.1).

En caso de que ningún marcador se haya detectado, la etiqueta con la descripción muestra el texto "No se ha detectado ningún marcador". En la Figura 4.7 se muestran algunos ejemplos de la información que muestra la etiqueta de descripción de la especie dibujada al detectar el marcador. Como se puede apreciar en la imagen más a la izquierda no se ha detectado ningún marcador.

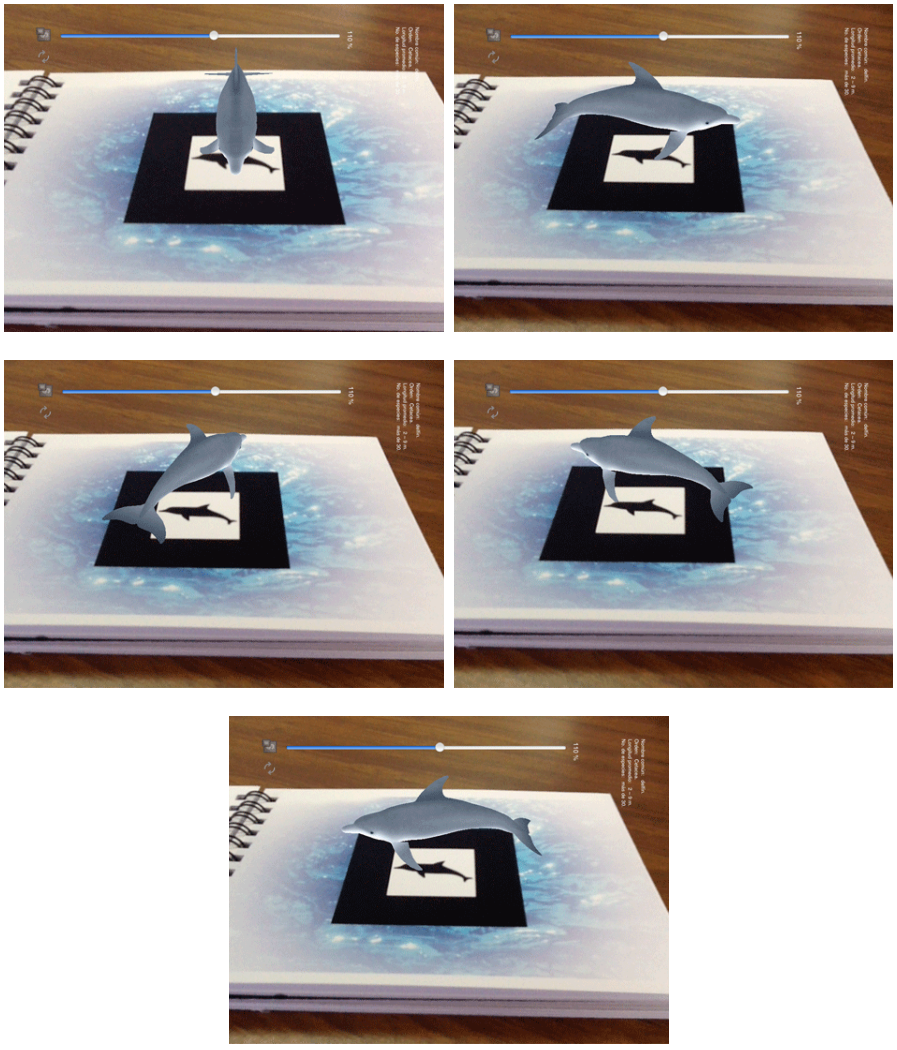


Figura 4.8: Ejemplos de la animación de un modelo 3D en la aplicación.

Ahora bien, para rotar el modelo 3D para animarlo, se presiona el botón con la imagen de dos flechas ubicado en la parte inferior izquierda de la pantalla si es que

el iPad se encuentra en posición vertical (*portrait*), pues la interfaz de propiedades no se adapta a la orientación del dispositivo al no ser relevante para la aplicación, ya que nos interesa el objeto virtual colocado en la escena al detectar un marcador y no los elementos gráficos de la interfaz de propiedades.

Por lo que al cambiar la orientación del iPad a horizontal (*landscape*), el botón, la etiqueta con la descripción de la especie y la barra deslizante para modificar la escala del modelo no cambiarán de ubicación.

La rotación que realiza el modelo es en sentido contrario al de las manecillas del reloj y no es posible aumentar o disminuir la velocidad a la que se realiza la animación. La Figura 4.8 da una perspectiva de cómo un modelo 3D rota sobre su propio eje en la aplicación.

Como se puede observar en cada una de las imágenes que conforman la Figura 4.8, la libreta se mantiene fija mientras que el modelo va cambiando constantemente de ubicación, a diferencia de si el modelo estuviese fijo y fuese el fascículo el que cambiara de posición (Figura 4.9), pues en este último caso, se apreciaría otra perspectiva del modelo 3D (el reverso, perfil, $\frac{3}{4}$, etc.) dependiendo de la ubicación de la libreta.

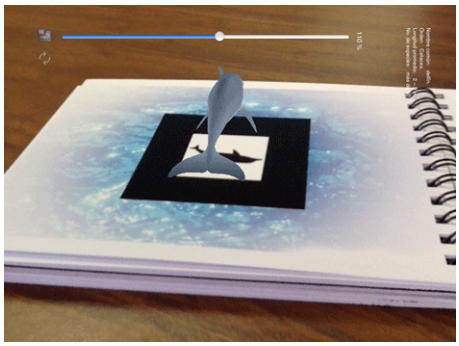


Figura 4.9: Diferencia entre animación de un modelo 3D y cambio de posición del fascículo.

En el caso del escalamiento, bastará con pasar el dedo sobre la barra deslizante de la interfaz en la dirección deseada para aumentar o reducir el tamaño de la especie. Algunos ejemplos de escalamiento se muestran en la Figura 4.10.

Cerca de la barra deslizante aparece una etiqueta indicando el porcentaje del tamaño del objeto (que va desde 0 % hasta 200 %). En el primer ejemplo el modelo se reduce hasta el 35 % y posteriormente se aumenta hasta un 200 %, mientras que en el segundo caso, la especie se muestra en el 100 % de su tamaño y se aumenta hasta el valor máximo permitido (200 %).

Como se observa, el escalamiento no altera la posición del modelo, no importa la orientación en que se encuentre, al escalarlo mantiene su orientación.

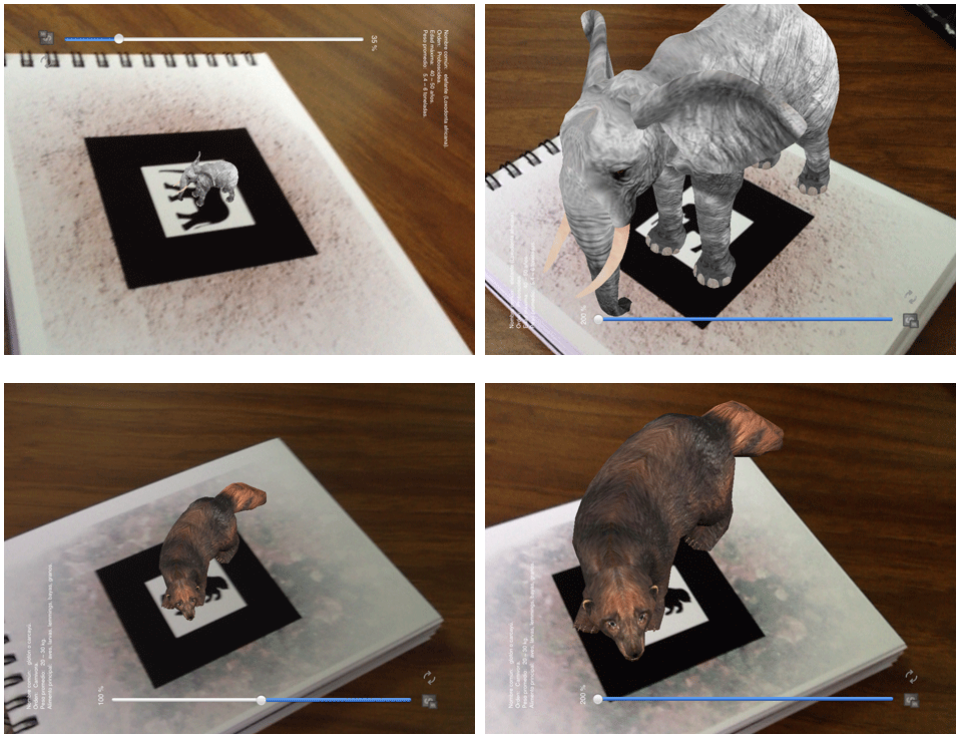


Figura 4.10: Ejemplos de escalamiento de modelos 3D en la aplicación.

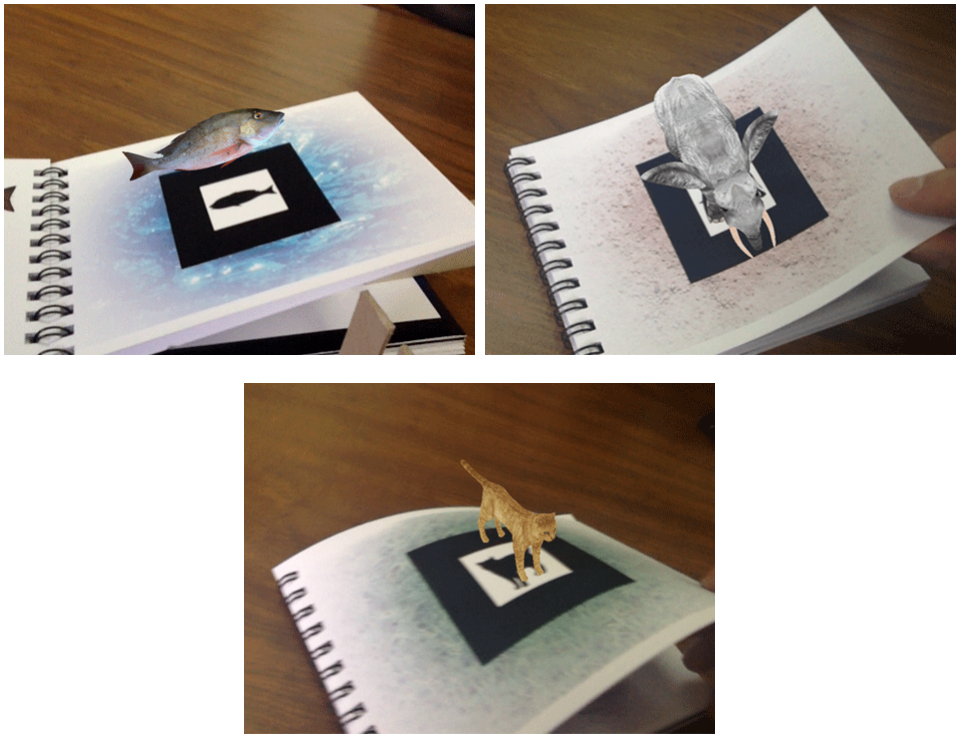


Figura 4.11: Ejemplos de pruebas al sistema deformando levemente el marcador o alterando su posición.

Así mismo, al probar el sistema se puede observar que se da un correcto seguimiento del objeto, *i.e.*, se dibuja el modelo 3D en la dirección en la que el marcador se encuentre, además, si éste se deforma un poco aún puede ser identificado hasta cierto punto. Ejemplos de ello se muestran en la Figura 4.11.

Por otra parte, si el fascículo se aleja de la cámara, el marcador podrá ser captado hasta cierta distancia, dibujando y posicionando el modelo de manera satisfactoria (Figura 4.12), mientras que en el caso del acercamiento, podrá ser detectado siempre y cuando sea completamente visible.

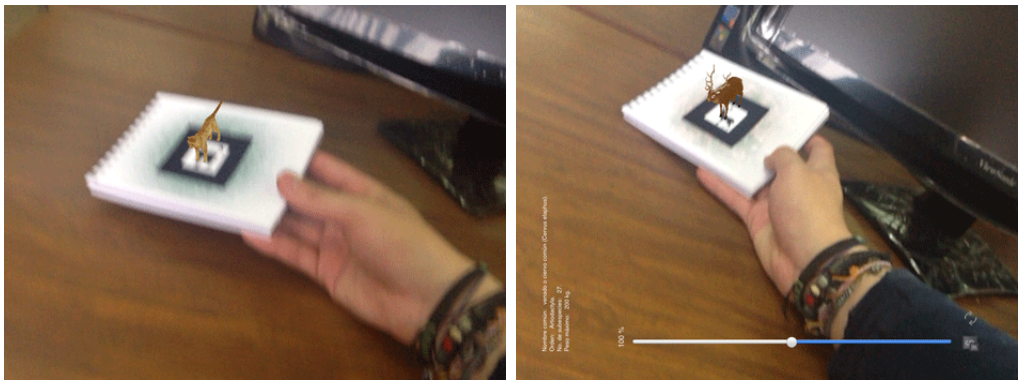


Figura 4.12: Ejemplos de pruebas al sistema alejando el fascículo de especies de la cámara.

Sin embargo si la deformación del marcador o el alejamiento de la libreta es demasiado, el marcador no logra ser identificado y en ocasiones, el sistema de RA puede interpretarlo como un marcador distinto puesto que la deformación lo vuelve un tanto irreconocible.

Así, figuras similares como la del pez y el pingüino o la de la hormiga roja y el tigre, pueden ser interpretadas como la misma en casos en los que el marcador esté demasiado lejos, pero sea aún reconocible.

La iluminación del entorno también puede afectar la detección del marcador, pues la cámara del iPad tiende a realizar continuamente un auto ajuste con la intención de enfocar mejor. En ocasiones (dependiendo de las condiciones de iluminación) tiende a realizar muy seguido esta acción, por lo que el modelo se vuelve a dibujar una y otra vez al grado de que pareciera que parpadea. Si esto ocurre, se sugiere acercar más el iPad al fascículo o pasar el dedo a través de la cámara del iPad para que vuelva a reconocer toda la escena hasta que se adapte a ella.

Básicamente, las acciones aquí descritas son las que se pueden realizar con la aplicación *Fascículo de especies*.

Capítulo 5

Aplicación 2: Kanjirama

La aplicación *Kanjirama* tiene una estructura similar a la de la aplicación *Fascículo de especies*, esto es, se requiere una conexión al sistema de RA mediante la clase *Detectar marcador*, el manejo de la cámara y del audio se realizan de la misma forma, se utiliza una vista para la conexión con *OpenGL ES* en la cual se dibujan los objetos 3D y éstos se buscan y cargan de la misma manera.

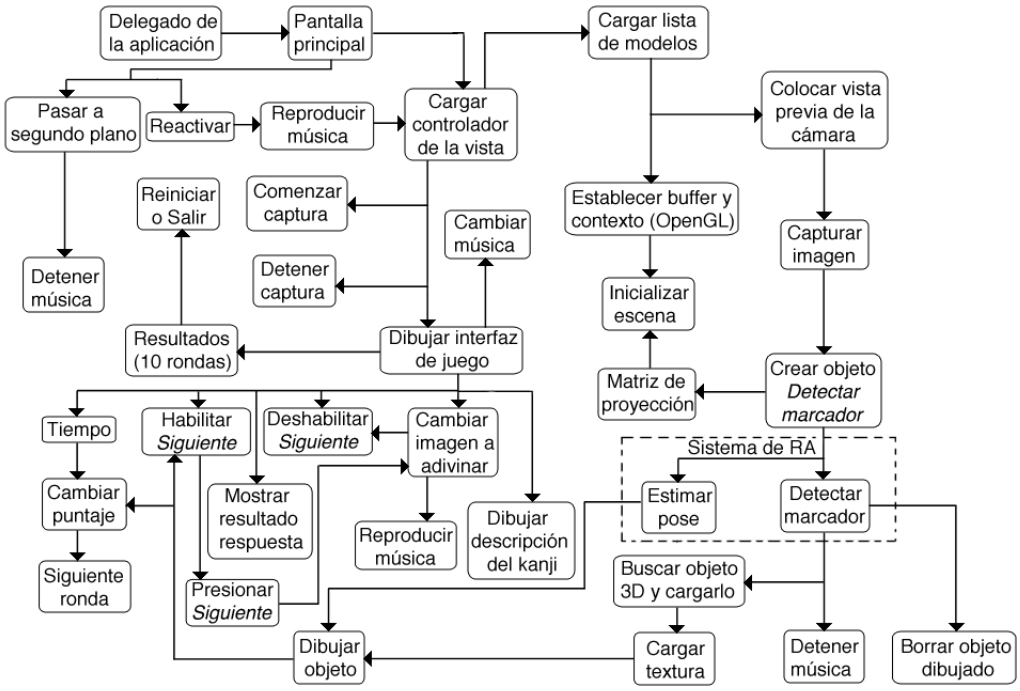


Figura 5.1: Diagrama general de la segunda aplicación.

Sin embargo, existen ciertos cambios que se deben realizar en algunos procesos para adaptar las clases al comportamiento deseado en esta aplicación, además de que se utiliza una interfaz de juego (cuyas acciones se describen en la sección 5.3) y una pantalla de inicio (por medio de clases). La Figura 5.1 muestra un esquema general del funcionamiento de esta aplicación.

Como se puede observar, el manejo de audio se realiza por medio de la interfaz de juego y no por el controlador principal de la vista, además de que el delegado principal de la aplicación se conecta con una clase de nombre *Pantalla principal*, en lugar de conectarse con el controlador principal de la vista, por lo que la aplicación no comienza directamente con la vista previa de la cámara, sino que emplea una pantalla de presentación inicial antes de mostrar dicha vista previa.

5.1. Detección de marcadores de referencia

En la sección 2.4 (p. 20) se resaltó cuáles son los diversos tipos de marcadores que se pueden emplear para proyectos de RA, entre ellos, los marcadores de referencia. Además, se explicó brevemente cuáles son sus características, pero ahora nos enfocaremos en el método que se puede utilizar para determinar el identificador único que se asocia a cada uno de ellos, pues la aplicación *Kanjirama* hace uso de este tipo de marcadores.

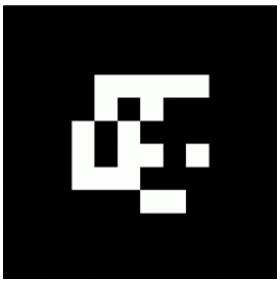


Figura 5.2: Ejemplo de patrón de bits.

Otras formas de designar a estos marcadores son patrones de bits o patrones de identificador simple e inclusive, algunos tienen su propio método de decodificación como los patrones *BCH*, aunque su apariencia sea muy similar a la de un patrón de bits común [68]. Además, pueden presentar diferente ancho del marco negro que los envuelve. Un ejemplo de un patrón de bits como los que utiliza la aplicación *Kanjirama* se puede apreciar en la Figura 5.2.

Existe la posibilidad de variar las dimensiones de este tipo de marcadores, aunque lo más común es utilizar imágenes de 6×6 pixeles e imprimirlos a diferente escala, a manera de que no se pierda la resolución en la muestra impresa. Mientras menor sea la resolución, el procesamiento para determinar el identificador tardará menos tiempo, aunque se limite el número de posibles marcadores a generar, que en este caso es de 2^9 posibles combinaciones. En cambio, otro tipo de marcadores como los *BCH* pueden generar hasta 4096 imágenes distintas, pero se reitera que el método para determinar el identificador que poseen es diferente.

Para utilizar marcadores de referencia en el sistema, el Algoritmo 12 (p. 68) debe ser modificado desde su inicialización, la cual debe realizarse de la siguiente manera: $sistema \leftarrow new\ \ sistema(ancho, alto, 1, 6, 6, 6, 1)$, donde (como se puede apreciar) las dimensiones de los marcadores se establecen de 6×6. Además, es necesario señalar que el tipo de marcador a utilizar es ID_SIMPLE y no $PLANTILLA$ (línea 10). Finalmente, no es necesario cargar ningún tipo de patrón ya que en dicho algoritmo, todos los archivos con las matrices de las especies debían agregarse al sistema.

El procedimiento de detección se sigue tal cual se explicó en el capítulo 3 (pero ahora las dimensiones del patrón son 6×6), excepto en el momento en el que se realiza la coincidencia del patrón (Algoritmo 7, en la página 49) con los existentes en memoria, pues este procedimiento es diferente y se utilizan operadores binarios para determinar el identificador, así como la orientación y nivel de confianza del marcador, para lo cual será necesario conocer algunas máscaras básicas entre bits.

Una máscara es un pequeño conjunto de datos binarios para realizar operaciones bit a bit sobre un frente de bits mayor y es útil para activar, desactivar o invertir el estado en el que se encuentra un bit, modificando su valor. Algunos tipos de máscaras se resumen en la Tabla 5.1.

Tipo	**Operador**	**Resultados**	**Uso común**
Enmascarar bits a 1	OR	$X \mid 1 = 1$ $X \mid 0 = X$	Activar un bit, asegurarse de que un bit está activo, etc.
Enmascarar bits a 0	AND	$X \,\&\, 1 = X$ $X \,\&\, 0 = 0$	Desactivar un bit, verificar el estado de un bit, etc.
Alternar valor del bit	XOR	$X \oplus X = 0$ $X \oplus Y = 1$	Invertir el valor de un bit.

Tabla 5.1: Tipos de máscaras y sus usos.

Ahora bien, empleando máscaras, se procede de la siguiente manera: primero se pasa de un patrón encontrado en un segmento de una imagen a color capturada por el dispositivo a una imagen en escala de grises del mismo tamaño (en este caso de 6×6) empleando los pasos descritos en el Algoritmo 15, pues no se asocian colores en las comparaciones que siguen al procedimiento.

Algoritmo 15 Algoritmo que crea un patrón de 6×6 en escala de grises.

Entrada: Una imagen con información y una imagen vacía del mismo tamaño.
Salida: La versión en escala de grises de la imagen con información.

1: **for** $i = 0$ to $ANCHO \times ALTO \times 3$ step 3 **do**
2: $\quad salida++ \leftarrow (entrada[i+0] + (entrada[i+1] << 1) + entrada[i+2]) >> 2$
return 0

Algoritmo 16 Algoritmo que aplica una mascara XOR al patrón.

Entrada: Un patrón de bits.
Salida: El mismo patrón tras aplicarle una máscara.

1: $patron \leftarrow patron \oplus mascaraFinal$

Se realiza un proceso de umbralización a este patrón en escala de grises para crear uno nuevo sobre el que se buscará la información solicitada (identificador, dirección y nivel de confianza) y una vez creado se verifica qué valores encierra, primero aplicando una máscara de tipo *XOR* para invertir el valor de los bits (ver Tabla 5.1) empleando el Algoritmo 16.

En este algoritmo, $mascaraFinal$ representa una máscara formada a partir de cuatro máscaras de menor tamaño. Recordando que sólo se tienen 6×6 pixeles (esto es 36 pixeles) y al mismo tiempo, teniendo en cuenta que una imagen es al final un conjunto de bits, entonces, si se coloca una máscara pequeña cada nueve bits realizando los corrimientos necesarios y se aplica el operador OR, tendremos una máscara completa de 36 bits. Los valores de las máscaras de menor tamaño y su colocación en la máscara final se especifican en la Tabla 5.2.

Nombre	**Valor**	**Posición**
$Mascara_0$	0×0027	0
$Mascara_1$	$0 \times 014e$	9
$Mascara_2$	0×0109	18
$Mascara_3$	$0 \times 00db$	27

Tabla 5.2: Valores de las máscaras para filtrar el patrón.

Una vez aplicada la máscara sobre el patrón, se emplea el Algoritmo 17 para examinar los bits y conocer su estado por medio del operador AND (ver Tabla 5.1).

Algoritmo 17 Algoritmo que verifica el estado de un bit.

Entradas: Un patrón de bits y un bit específico.

Salidas: El estado del bit (1 ó 0) y un valor auxiliar para obtener el nivel de confianza.

```
1: b_0 ← patron >> (Mascara_0 + bit)&1
2: b_1 ← patron >> (Mascara_1 + bit)&1
3: b_2 ← patron >> (Mascara_2 + bit)&1
4: b_3 ← patron >> (Mascara_3 + bit)&1
5: suma ← b_0 + b_1 + b_2 + b_3
6: switch suma do
7:     case 0
8:         valorBit ← 0
9:         return 1.0f
10:    case 1
11:        valorBit ← 0
12:        return 0.5f
13:    case 2
14:        valorBit ← 0
15:        return 0.0f
16:    case 3
17:        valorBit ← 1
18:        return 0.5f
19:    case 4
20:        valorBit ← 1
21:        return 1.0f
   return 0.0f
```

En este algoritmo también se emplean las máscaras descritas en la Tabla 5.2 (líneas 1 a 4) y nos auxilia en el cálculo del valor de confianza (líneas 9, 12, 15, 18 y 21) y posteriormente del identificador del patrón, como se muestra en el Algoritmo 18, donde se aplica la máscara *XOR* del Algoritmo 16 (línea 3). La variable $idBits$ en este algoritmo posee el valor de la longitud (en bits) de una máscara pequeña, *i.e.*, 9 y se utiliza para realizar un promedio con los valores de los bits para determinar el nivel de confianza del marcador encontrado (línea 8).

Algoritmo 18 Algoritmo que busca el identificador y valor de confianza del patrón de bits (verificar patrón) de acuerdo a la imagen de entrada.

Entrada: Un patron de bits.
Salidas: El identificador y valor de confianza del patrón de entrada.

```
1: confianza ← 0.0f
2: id ← 0
3: aplicarMascara(patron)
4: for i = 0 to idBits do
5:     valorBit ← 0
6:     confianza ← confianza + verificarBitEnPatron(patron, i, &valorBit)
7:     id ← id | (valorBit << i)
8: confianza ← confianza / idBits
9: if confianza < 0.9f then
10:    confianza ← 0.0f
```

Algoritmo 19 Algoritmo que rota un patrón a la derecha.

Entrada: Un patrón de bits.
Salida: El mismo patrón rotado 90° en el sentido de las manecillas del reloj.

```
1: Patron_ID uno ← 1, patronTmp ← patron
2: patron ← 0
3: for i = 0 to bitsPatron do
4:     if bitActivado(patronTmp, matriz90[i]) then
5:         patron ← patron | (uno << i)
```

Algoritmo 20 Algoritmo que verifica si un bit está activado.

Entradas: Un patrón de bits y un bit específico.
Salida: Un valor booleano con base en si el bit está activado o no.

```
1: return ((patron >> bit)&1) ≠ 0
```

Lamentablemente verificar el patrón, *i.e.*, obtener su nivel de confianza e identificador no es suficiente para aseverar que se trata de cierto marcador, es necesario hallar la dirección (orientación) a la que el marcador se encuentra y en el proceso

Algoritmo 21 Algoritmo para verificar la coincidencia entre patrones de bits.

Entrada: Una imagen donde posiblemente se encuentre un marcador.
Salidas: El identificador, orientación y nivel de confianza del patrón de bits.

$imagenPatron[ANCHO \times ALTO] \leftarrow imgEntrada$
if $convertirAEscalaDeGrises(imgEntrada, imagenPatron) < 0$ **then**
    $codigo \leftarrow 0$
    $direccion \leftarrow 0$
    $confianza \leftarrow -1.0f$
    return -1
PATRON_ID $patron = 0, uno = 1$
for $i = 0$ to $bitsPatron$ **do**
    if $imagenPatron[bitsPatron - 1 - i] > umbral$ **then**
        $patron \leftarrow patron \mid uno << i$
PATRON_ID $patron_0, patron_{90}, patron_{180}, patron_{270}$
Crear un id e inicializarlo con -1 por cada patrón
Crear un valor de $confianza$ por cada patrón e inicializarlo a $-1.0f$
$patron_0 \leftarrow patron$
$verificarPatron(patron_0, id_0, confianza_0)$
$patron_{90} \leftarrow patron_0$
$rotar_{90}(patron_{90})$
$verificarPatron(patron_{90}, id_{90}, confianza_{90})$
$patron_{180} \leftarrow patron_{90}$
$rotar_{90}(patron_{180})$
$verificarPatron(patron_{180}, id_{180}, confianza_{180})$
$patron_{270} \leftarrow patron_{180}$
$rotar_{90}(patron_{270})$
$verificarPatron(patron_{270}, id_{270}, confianza_{270})$
if $patron_0$ es el mayor de todos **then**
    $direccion \leftarrow 0$
    $confianza \leftarrow confianza_0$
    $codigo \leftarrow id_0$
else if $patron_{90}$ es el mayor de todos **then**
    $direccion \leftarrow 1$
    $confianza \leftarrow confianza_{90}$
    $codigo \leftarrow id_{90}$
else if $patron_{180}$ es el mayor de todos **then**
    $direccion \leftarrow 2$
    $confianza \leftarrow confianza_{180}$
    $codigo \leftarrow id_{180}$
else
    $direccion \leftarrow 3$
    $confianza \leftarrow confianza_{270}$
    $codigo \leftarrow id_{270}$
return 0

puede encontrarse que el marcador realmente es otro diferente al hallado previamente, por ello el valor de confianza es vital en este proceso.

Para verificar otras orientaciones del patrón, será necesario rotarlo 90° en el sentido de las manecillas del reloj (como se describe en el Algoritmo 19) para volver a realizar el proceso de verificación al menos otras tres veces (una por cada posible orientación) y al final verificar cuál de los valores de confianza obtenidos en el proceso es el más alto.

En este algoritmo, *bitsPatron* es el número total de bits del patrón (línea 3), *i.e.*, 36, además, se verifica el estado de un bit usando el operador *AND* (ver Algoritmo 20) como se mencionó en la Tabla 5.1 empleando el patrón de bits y un bit específico, que en este caso es una de las entradas que conforman una matriz que permite la rotación a 90°, ya que define el corrimiento que debe hacerse sobre el patrón para hallar el bit específico a verificar. Esta matriz de rotación es de 6×6 entradas y posee valores entre 0 y 35 ordenados de forma tal que los corrimientos sean los adecuados. Así, en caso de que el bit esté activado, se rota dicha entrada en el patrón (línea 5). Este proceso se realiza por cada entrada de la matriz que representa al patrón.

Una vez que el patrón ha sido rotado tres veces y que se han obtenido los identificadores y valores de confianza de cada uno de los posibles casos, se procede a comparar cuál de ellos posee el nivel de confianza mayor y con base en dicha información se determina la orientación, identificador único y el valor de confianza final. Estos valores son devueltos y el proceso de detección termina. El Algoritmo 21 engloba todo el proceso de coincidencia de un patrón de bits.

5.2. Nociones básicas de la escritura japonesa

El idioma japonés se compone de diversos tipos de escritura, los silabarios *Hiragana* (平仮名 o ひらがな) y *Katakana* (片仮名 o カタカナ) y de los *Kanji* (漢字). Ambos silabarios se componen de cuarenta y seis caracteres diferentes divididos en cuarenta sílabas, cinco vocales y una consonante independiente (aunque se pueden añadir elementos que suavizan el sonido y formar combinaciones que dan origen a nuevas sílabas) que poseen las mismas pronunciaciones.

El *Hiragana* se emplea para la escritura japonesa en general, mientras que el *Katakana* se emplea para la escritura de palabras de origen extranjero [69]. Otros tipos de escritura derivadas a partir de las anteriores son: *Okurigana* (送り仮名 u おくりがな), que es la combinación de *Kanji* con *Hiragana* en un texto (útil para identificar verbos) y *Furigana* (振り仮名 o ふりがな) que es la colocación de pequeños caracteres en *Hiragana* o *Katakana* arriba o al lado de los *kanji* para saber su lectura. Finalmente, cabe mencionar que también existe el *Rōmaji* (ローマ字) que no es más que la forma en que se designa a la escritura occidental y que es utilizada en

menor medida, pero que es de gran utilidad para los extranjeros ya que mediante él es posible escribir japonés con el alfabeto que utilizamos.

Sin embargo, a causa de diversos factores como el tamaño de los caracteres, la similitud del sonido en varias palabras que conforman el idioma y aunado al hecho de que en la escritura japonesa no se utilizan espacios, el uso exclusivo del *Hiragana* dificulta bastante la comprensión de un texto, puesto que no se reconoce el inicio y fin de una palabra o no son fácilmente entendibles los términos utilizados cuyas pronunciaciones sean equivalentes, aun cuando se conozca el contexto. Por ello se utilizan los *kanji*, los cuales son ideogramas, *i.e.*, caracteres que representan ideas, además, sus pronunciaciones se forman con base en la combinación de sílabas.

Se estima que existen alrededor de 50000 caracteres chinos a partir de los cuales se originaron los *kanji*, incluyendo aquellos que han caído en desuso. Sin embargo, el japonés actual contempla el uso del *Kanji* por categorías, entre ellas, la lista de *kanji* de uso común (1945 caracteres) que corresponde a los ideogramas que un estudiante japonés aprende desde primaria hasta el nivel medio superior, la lista de los *kanji* para nombres propios (284 caracteres) y el estándar *JIS* dividido en dos categorías, la primera que agrega aproximadamente 3000 caracteres entre los cuales están algunos *kanji* de campos específicos como medicina y leyes, ideales para poder leer el periódico con fluidez y la segunda clase, que abarca alrededor de otros 3000 caracteres, para los cuales, la persona que domine éstos y todos los anteriores podría ser considerada un experto en materia [70], mas no un especialista. Además, una computadora con soporte para japonés, tiene a disposición al menos 11436 *kanji* diferentes.

El estudio del *Kanji* es tomado como algo serio pues se puede saber su origen, orden de trazos, radicales (otros *kanji* que componen al que se está estudiando), significado, grado de similitud con otros *kanji*, etc., pero la dificultad se eleva un poco más al incluir la lectura, la cual depende del contexto, combinación y localización. La mayoría de los *kanji* poseen dos lecturas a pesar de que algunos (muchos de ellos de uso diario) pueden tener hasta diez o más posibles lecturas. Las lecturas de un *kanji* se categorizan dependiendo de si derivan del chino original o fueron adaptadas a la lengua japonesa, para lo cual, la lectura recibe el nombre de 音読み (*Onyomi*) en el primer caso y 訓読み (*Kunyomi*) en el segundo. Para saber cuál de las dos se utiliza, existen dos reglas generales:

- Si se trata de una palabra de un solo *kanji* o de un vocablo escrito en *Okurigana* en el que sólo hay un *kanji*, éste se leerá en *Kunyomi*.
- Generalmente, una combinación de *kanji* se lee mediante *Onyomi*, aunque hay excepciones para las cuales se debe consultar un diccionario de *Kanji* [71].

Es importante tener en cuenta esto último, ya que la aplicación *Kanjirama* sólo muestra una lectura (en *Kunyomi* para la mayoría de los *kanji* empleados).

5.3. Interfaz de juego

La aplicación *Kanjirama* actúa de una manera muy similar a como lo hace el *Fascículo de especies* (descrita en el capítulo anterior), *i.e.*, los modelos 3D se obtuvieron, utilizan y dibujan de la misma manera, la conexión con el sistema de RA se realiza empleando el mismo procedimiento, los marcos de video se capturan como en dicha aplicación y la sesión de audio se obtiene de igual forma, simplemente es una clase diferente la que hace uso de esta última.

Además, como se muestra en el esquema de la Figura 5.1, el *Delegado de la aplicación* se conecta con otra clase de nombre *Pantalla principal*, la cual tiene el objetivo de actuar como una interfaz de inicio y no comenzar directamente la vista previa de la cámara, además de dar presentación a la aplicación. Esta interfaz hace uso de la clase de manejo de audio por defecto de *Objective-C*, pero se omitió en el diagrama por considerarse un aspecto irrelevante.

Esta pantalla dispone de un botón para realizar una transición entre vistas y colocar en pantalla la vista principal empleando el Algoritmo 22 para ello, comenzando así el juego de *Kanjirama*. En este algoritmo, se especifica claramente hacia qué vista se desea cambiar y mediante qué tipo de animación hacerlo para finalmente remover a la vista actual de la pantalla. Hasta este punto, self.controladorPrincipal debe ser una instancia ya iniciada de la clase *Controlador principal de la vista*.

Algoritmo 22 Algoritmo que realiza la transición entre vistas.

Entrada: Una vista.
Salida: Otra vista distinta.

```
1: if la música se está reproduciendo then
2:     Detener la música y liberar memoria.
3: [UIView  transitionFromView:self.view
            toView: self.controladorPrincipal.view
            duration:0.5
            options:UIViewAnimationOptionTransitionFlipFromLeft
            completion:^(BOOL finished)
            {
             [self:view removeFromSuperview]
            }
       ]
```

Ya que *Kanjirama* pretende ser un juego de destreza, el factor de tiempo influye bastante en determinar la habilidad de respuesta rápida que un jugador puede tener para reaccionar ante una situación propuesta en pantalla. En este caso, dado a que se debe tomar una tarjeta de RA con base en una imagen que aparezca en pantalla, se añade cierta presión sobre el jugador al incluir un reloj a la modalidad de juego.

Para agregar este atributo de tiempo, se hace uso de un objeto del tipo temporizador, el cual es gestionado por la clase *Interfaz de juego*, la cual entra en escena al mismo tiempo que lo hace la clase *Controlador de la vista*, después de la transición realizada a partir de la clase *Pantalla principal*. La manera de iniciar dicho temporizador se muestra en el Algoritmo 23.

Algoritmo 23 Algoritmo que crea e inicia el temporizador.

Entrada: Ninguna.
Salida: Inicia el temporizador.

```
1: [self deshabilitarSiguiente]
2: Buscar archivo de audio.
3: Reproducir archivo de audio.
4: if self.timer ≠ nil then
5:     [self.timer invalidate]
6: segundos ← 30
7: Instanciar un objeto del tipo temporizador.
8: Dar formato a la etiqueta de segundos.
9: Agregar temporizador al hilo principal.
```

El método propio de la clase llamado en la línea 1 de este algoritmo se encarga de deshabilitar un botón de nombre *Siguiente* (que se encarga de reiniciar el temporizador) para que mientras el temporizador se encuentre en ejecución, no sea posible detenerlo manualmente sino sólo hasta que éste finalice o sucedan ciertas acciones características del juego que serán descritas posteriormente.

Por otra parte, tal como se mencionó previamente, la clase *Interfaz de juego* hace uso de la sesión de audio para reproducir archivos de música, por lo que cuando se inicia el temporizador, el archivo de una pista de audio específica es buscado y reproducido, además, de existir una instancia para el temporizador, éste se invalida y se crea una nueva. El temporizador es de 30 segundos, por lo que se destina una variable para manejar el tiempo y se da formato a una etiqueta que mostrará este valor en pantalla.

Comúnmente un temporizador emplea una función auxiliar que describe las acciones a realizar cada intervalo de tiempo deseado, por lo que se especifica la función del temporizador en el Algoritmo 24. Como se puede observar, los intervalos corresponden a un segundo, por ello esta variable es decrementada cada que se inicia esta función y con base en su valor se destinan las acciones a realizar. Si los segundos se han terminado, el juego incrementa un turno, *i.e.*, el jugador no fue capaz de voltear la tarjeta de RA solicitada y por tanto se termina la ronda. Además, si se han alcanzado los 10 turnos, el juego termina y se realizan las acciones pertinentes (detener el temporizador y abrir una ventana de alerta que muestra los resultados de la partida). De no ser el caso, el juego continúa y *segundos* vuelve a iniciarse en 30, además de que se busca una nueva imagen aleatoria (de entre 27 posibles) para desplegarse en

pantalla y el jugador pueda adivinar nuevamente. Finalmente, si $segundos < 10$, el color de la etiqueta que muestra el tiempo restante en pantalla cambia de color para ejercer una presión mayor sobre el jugador.

Algoritmo 24 Algoritmo que describe la función del temporizador.

Entrada: Un temporizador.
Salida: Ninguna.

```
1:  segundos ← segundos − 1
2:  if segundos = 0 then
3:      turnos ← turnos + 1
4:      if turnos ≥ 10 then
5:          [self detenerTimer]
6:          [self abrirVentanaAlerta]
7:      segundos ← 30
8:      idAadivinar ← arc4random() %27
9:      Obtener el nombre de la imagen cuyo id sea idAadivinar.
10:     Crear una imagen temporal con el nombre anterior.
11:     imagenAadivinar ← imagen
12: if segundos < 10 then
13:     Colocar la etiqueta de segundos en rojo.
14: else
15:     Colocar la etiqueta de segundos en blanco.
16: Dar formato a la etiqueta de segundos
```

En la sección siguiente se describe más a detalle la forma de interactuar con la aplicación, para una mayor comprensión de las acciones que es posible realizar y de cómo está estructurado el juego.

5.4. Modo de uso de la aplicación

La aplicación *Kanjirama* no está dirigida a un público con un rango de edad específico, sino a aquellas personas interesadas en el aprendizaje del idioma japonés, en lo que respecta a algunos de los miles de ideogramas que conforman su escritura, según se menciona en la sección 5.2.

La aplicación inicia al presionar en el iPad el icono mostrado en la Figura 5.3. La idea básica del juego es similar a la de un *memorama* común, en el cual se tienen una serie de tarjetas con imágenes iguales presentadas en pares, *e.g.*, si el *memorama* es de 64 tarjetas, se tendrían 32 pares diferentes. Las cartas se colocan boca abajo y deben ser volteadas en pares hasta lograr voltear dos tarjetas iguales para avanzar en el juego, regresando las cartas a

Figura 5.3: Icono de la aplicación *Kanjirama*.

su posición original en caso de voltearse dos tarjetas distintas. Se cree que el juego estimula la memoria, pues conforme se revela la ubicación de alguna de las tarjetas que forma un par, el jugador memoriza dichas posiciones, de ahí el nombre de *memorama*.

Figura 5.4: Ejemplo del reverso de una tarjeta de RA.

Partiendo de esta iniciativa y tomando las modificaciones de Wagner y Barakonyi [42], *Kanjirama* emplea una serie de tarjetas diferentes, donde los pares se forman a partir de una imagen 2D en la pantalla del dispositivo y otra imagen 3D que se dibuja sobre la pantalla al voltear alguna de las tarjetas. Las tarjetas son de 9×9 *cm*. aproximadamente y fueron impresas por ambos lados. En el anverso poseen un marcador de referencia como el de la Figura 5.2, mientras que en el reverso muestran una imagen como la de la Figura 5.4, en la que se encuentra un *kanji*.

La aplicación cuenta con 27 tarjetas distintas, *i.e.*, sirve para aprender el significado y lectura (aunque no se especifica si en *Kunyomi* u *Onyomi*) de 27 *kanji* distintos y no se necesita tener ninguna tarjeta repetida ya que (como se mencionó anteriormente) el par se forma al mostrar en pantalla una imagen alusiva al significado de un *kanji*, con la intención de que el jugador voltee la tarjeta de RA con el símbolo correcto, para desplegar un modelo 3D alusivo al significado del *kanji* también y así formar el par.

Los *kanji* utilizados en la aplicación corresponden a algunos de los más básicos y de uso común, encontrados en las listas de vocabularios sugeridas a estudiar para acreditar el 日本語能力試験 (*Nihongo Nōryoku Shiken*) o Examen de Aptitud del Idioma Japonés [72], en sus niveles N5 y algunos incluso aplican para N4. La lista completa de los caracteres utilizados por esta aplicación se encuentran en la Tabla 5.3.

Se buscaron símbolos con significados triviales y fáciles de entender por medio de algún icono y modelo 3D equivalente, además de considerarse sólo *kanji* que tuviesen significado por sí mismos, *i.e.*, sin la necesidad de una combinación con otros *kanji* para formar una palabra.

Figura 5.5: Pantalla principal de la aplicación *Kanjirama*.

Al comenzar la aplicación se presenta una pantalla en la que se describen los pasos a seguir para jugar, entre ellos, se anticipa que para comenzar el juego es necesario colocar las tarjetas boca abajo sobre una superficie plana a manera de que los *kanji* sean visibles. Esta pantalla se muestra en la Figura 5.5. Se cuenta con un botón con la

leyenda *Comenzar* y al presionarlo aparece la imagen con la vista previa de la cámara y comienza la captura.

Kanji	**Lectura**	**Significado**
海(うみ)	*Umi*	Mar/playa.
女(おんな)	*Onna*	Mujer.
心(こころ)	*Kokoro*	Corazón/mente/espíritu.
石(いし)	*Ishi*	Piedra/gema.
月(つき)	*Tsuki*	Luna.
雪(ゆき)	*Yuki*	Nieve.
家(いえ)	*Ie*	Casa/familia.
本(ほん)	*Hon*	Libro/regalo/verdad.
目(め)	*Me*	Ojo/vista.
鏡(かがみ)	*Kagami*	Espejo.
水(みず)	*Mizu*	Agua.
男(おとこ)	*Otoko*	Hombre.
傘(かさ)	*Kasa*	Paraguas.
犬(いぬ)	*Inu*	Perro/can.
花(はな)	*Hana*	Flor.
手(て)	*Te*	Mano/manual.
山(やま)	*Yama*	Montaña/monte.
車(くるま)	*Kuruma*	Carro/auto/vehículo.
一(いち)	*Ichi*	Uno/único.
二(に)	*Ni*	Dos.
三(さん)	*San*	Tres.
四(よん)	*Yon*	Cuatro.
五(ご)	*Go*	Cinco.
六(ろく)	*Roku*	Seis.
七(なな)	*Nana*	Siete.
八(はち)	*Hachi*	Ocho.
九(きゅう)	*Kyū*	Nueve.

Tabla 5.3: Lista de los *kanji* empleados en la aplicación *Kanjirama*.

Asimismo, inicia la primera ronda del juego y en la esquina superior derecha de la pantalla aparece una imagen aleatoria alusiva al significado de un *kanji* (como se muestra en la Figura 5.6). La palabra *Adivina* aparece por encima de la imagen y por debajo se visualizan dos números que corresponden a un temporizador, ya que se cuenta con sólo 30 segundos para tratar de adivinar la tarjeta acorde a la imagen, de no hacerlo, el contador se reinicia y una nueva imagen aleatoria vuelve a aparecer.

Figura 5.6: Ejemplo del icono que aparece en la esquina superior derecha de la pantalla.

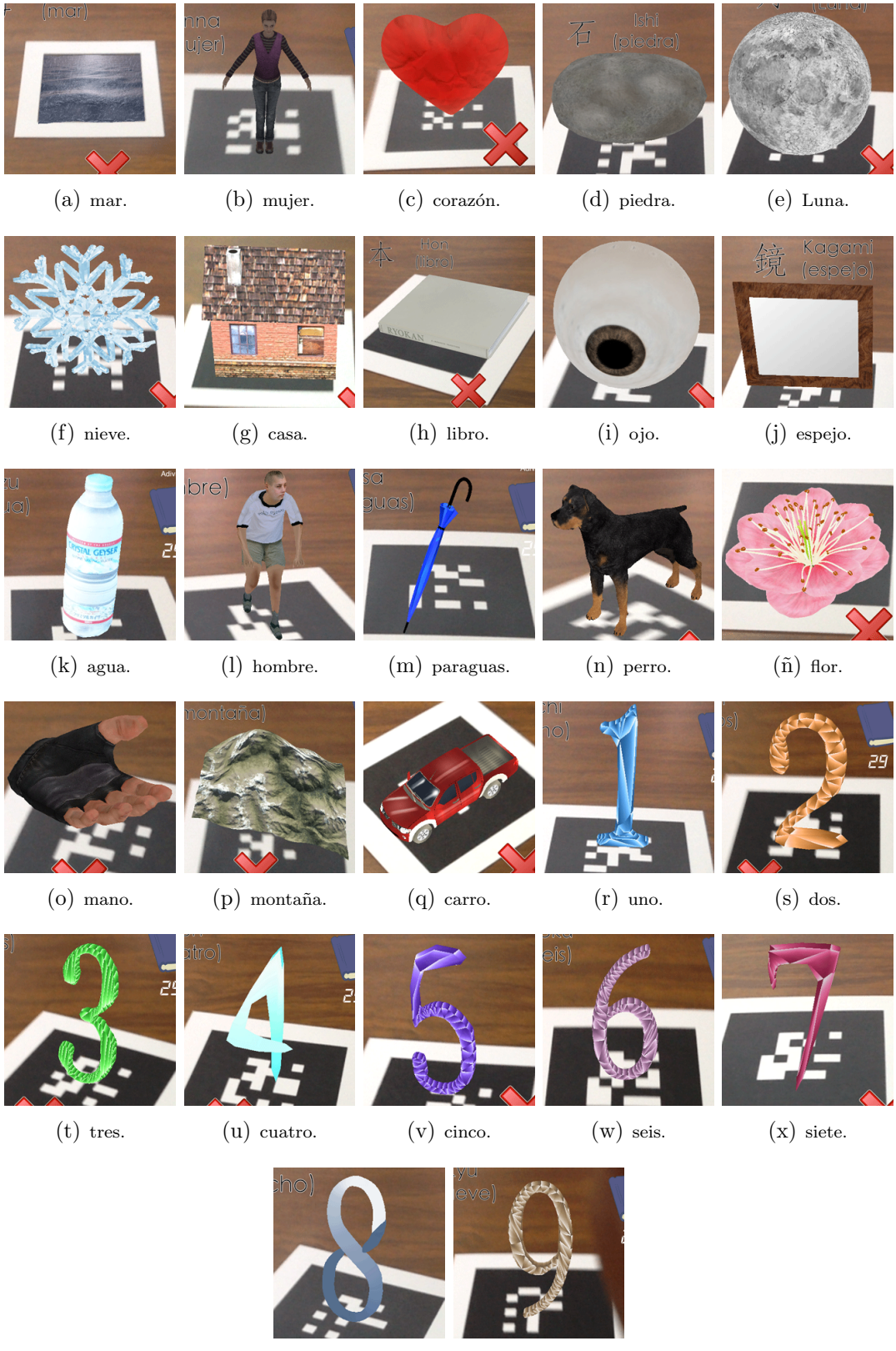
El iPad debe pasarse entonces por donde se ubiquen las tarjetas, como se observa en la Figura 5.7. Esta fotografía fue tomada empleando la cámara del iPad colocado de manera horizontal, por ello es que la imagen que indica el *kanji* a adivinar (en este caso el del espejo) se encuentra en la esquina superior izquierda y está rotado. De manera similar, un botón con la etiqueta *Siguiente* se encuentra en la esquina superior derecha pero rotado y deshabilitado.

Figura 5.7: Tarjetas de RA colocadas con el marcador boca abajo sobre una superficie plana.

Al voltear una tarjeta (sin importar de cuál se trate), en la esquina superior izquierda de la pantalla se muestra un texto que indica cuál es el *kanji* que le pertenece a la tarjeta que fue volteada, incluyendo su lectura y significado. Un ejemplo de ello puede verse en la Figura 5.8, en el cual la tarjeta que tiene el dibujo del *kanji* del agua fue volteada y aparece el texto asociado a ella.

Figura 5.8: Ejemplo de información que aparece al voltear una tarjeta.

Además de este texto, aparece un modelo en 3D puesto que al voltear la tarjeta de RA, el marcador de referencia se hace visible para la cámara del iPad. Así, se elige entre una de las 27 tarjetas y se dibuja el modelo sobre la vista previa de la cámara. La Figura 5.9 muestra todos los posibles modelos que pueden aparecer al voltear una tarjeta, cada uno se corresponde con el significado del *kanji* que tienen al reverso, según se especificó en la Tabla 5.3. Así, en el ejemplo de la Figura 5.8, la tarjeta de la Figura 5.9(k) fue volteada.



(a) mar. (b) mujer. (c) corazón. (d) piedra. (e) Luna.

(f) nieve. (g) casa. (h) libro. (i) ojo. (j) espejo.

(k) agua. (l) hombre. (m) paraguas. (n) perro. (ñ) flor.

(o) mano. (p) montaña. (q) carro. (r) uno. (s) dos.

(t) tres. (u) cuatro. (v) cinco. (w) seis. (x) siete.

(y) ocho. (z) nueve.

Figura 5.9: Modelos 3D de las tarjetas de RA.

Al voltear cualquier tarjeta, el temporizador se detiene. Si la tarjeta que se voltea es la solicitada, la interfaz de juego indica que la respuesta fue correcta (como se muestra en la Figura 5.10). Además, la ronda termina y se añade un punto al marcador, aunque el jugador desconoce esta información hasta que el juego termina. Como se puede ver en la Figura 5.10, el par se forma al coincidir la imagen de un copo de nieve con el modelo 3D del mismo objeto dibujado al voltearse la tarjeta. También aparece el texto referente al *kanji* y una imagen alusiva a que la respuesta fue correcta.

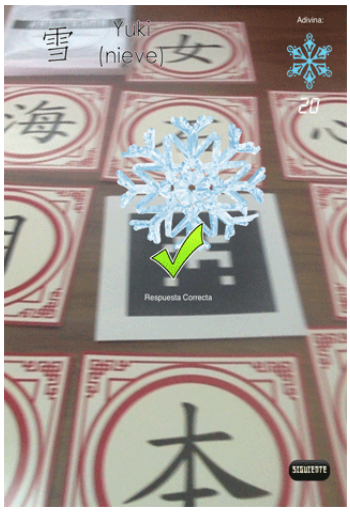


Figura 5.10: Ejemplo de tarjeta correcta volteada.

Por último, el botón *Siguiente* (ubicado en la esquina inferior derecha de la pantalla) se habilita, por lo que el juego se detiene hasta que el usuario presione este botón. Esta opción permite al usuario regresar la tarjeta a su posición original o voltear otras tarjetas para practicar, memorizar y aprender, además de que evita que el usuario falle en la ronda siguiente si es que aún no ha colocado la tarjeta volteada con el marcador boca abajo.

Por otra parte, si la cuenta regresiva se halla por debajo de los 10 segundos, el color del texto que indica el tiempo restante cambiará de color a rojo para añadir presión sobre el jugador, aunado al hecho de que existe una melodía de fondo que también agrega cierta tensión cada que está por finalizar una ronda. Un ejemplo del cambio de color en esta etiqueta se muestra en la Figura 5.11.

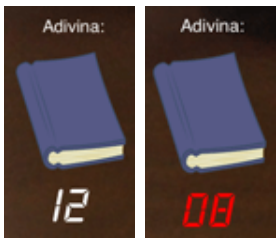


Figura 5.11: Ejemplos del cambio de color en la etiqueta que muestra la cuenta regresiva.

En caso de que el tiempo se termine, la imagen del *kanji* a adivinar cambia de manera aleatoria nuevamente y se incrementa el número de turnos jugados, sin afectar esto de ninguna manera el puntaje acumulado por el jugador hasta el momento.

El marcador tampoco se ve afectado si por el contrario, el jugador voltea una tarjeta errónea, pues en dicho caso, simplemente la ronda termina y se muestra una imagen que indica que se cometió un error, añadiendo el texto del *kanji* solicitado,

como se muestra en la Figura 5.12, en donde se indica que el caracter de la tarjeta de RA esperado era aquel cuyo significado es libro y no el que el jugador volteó.

Figura 5.12: Ejemplo de tarjeta incorrecta volteada.

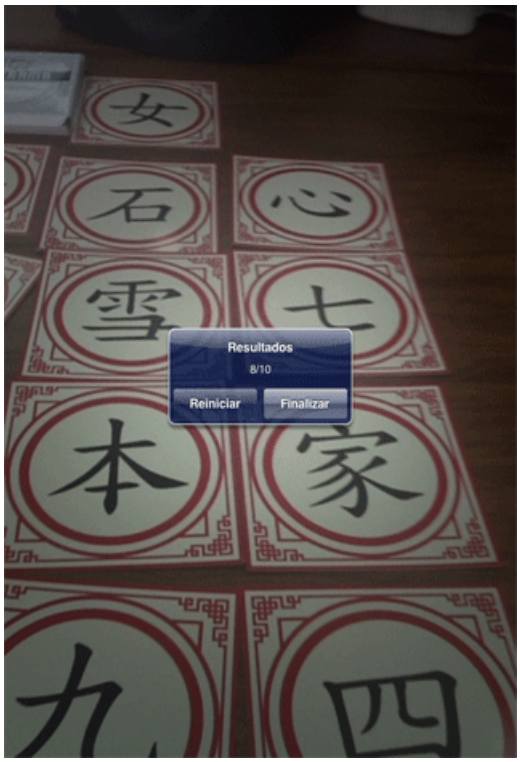


Figura 5.13: Ejemplo de ventana con los resultados de la partida.

Una vez que se cumplen 10 rondas, ya sea por aciertos, fallos o por tiempo finalizado, se muestra al jugador el marcador final de la partida (ver Figura 5.13) por medio de una ventana de alerta característica de las aplicaciones móviles. En dichos casos, se detiene de inmediato la detección, a pesar de que se evite al usuario ver cuál era la respuesta correcta en la última ronda en caso de que éste fallara. Esta ventana muestra dos botones con las opciones *Reiniciar* y *Finalizar* respectivamente, en cuyo caso, el primero permitirá repetir el juego mientras que el segundo finalizará la aplicación. En general, la detección del marcador en estos casos es más estable, pues no existen problemas por figuras similares como ocurría en el *Fascículo de especies* gracias al proceso de detección de patrones de bits.

Básicamente, las acciones aquí descritas son las que se pueden realizar con la aplicación *Kanjirama*.

Capítulo 6

Aplicación 3: Laberinto

En la Figura 6.1 se encuentra un esquema que muestra el funcionamiento general de la tercera aplicación. El diagrama de esta aplicación es más parecido al del *Fascículo de especies* que al de *Kanijirama*, *i.e.*, no existe una *Pantalla principal* que se conecte directamente con el *Delegado de la aplicación*, además, se conservan los objetos de las clases *Detectar marcador*, *Capa OpenGL ES*, *Objeto 3D* y *Controlador principal de la vista*.

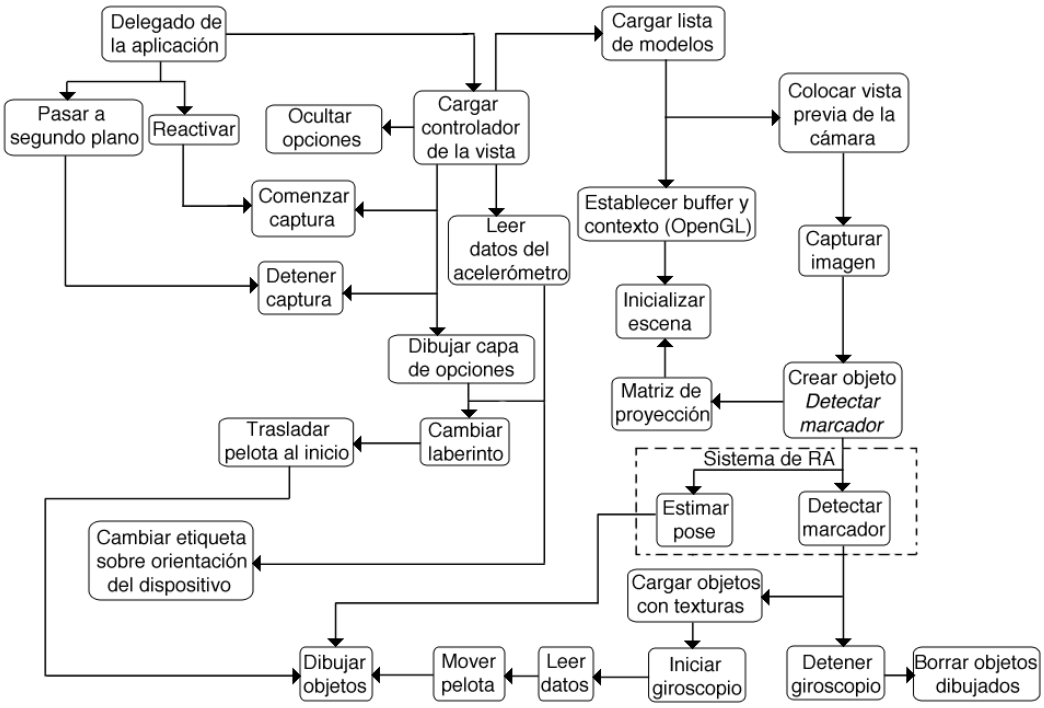


Figura 6.1: Diagrama general de la tercera aplicación.

Sin embargo, el objeto *Propiedades del objeto 3D* (ver Figura 4.1) es reemplazado por la capa *Opciones*, la cual tiene una funcionalidad similar. Por último, el *Controlador principal de la vista* incluye procedimientos que crean objetos de clases por omisión de *Objective-C* para el manejo del giroscopio y del acelerómetro, pues la

aplicación *Laberinto* hace uso de los sensores del iPad para mover la pelota virtual a través del laberinto y para detectar la orientación del dispositivo, con la finalidad de mapear correctamente las coordenadas (ver sección 6.1.2).

6.1. Sensores del iPad

El *iPhone* de *Apple* revolucionó el mundo de la telefonía celular al incorporar por primera vez tecnología no existente hasta ese momento como el acelerómetro, el cual permite conocer la orientación del dispositivo (en relación a la superficie de la Tierra) con base en una serie de parámetros que representan la aceleración lineal.

El acelerómetro de los dispositivos con *iOS* mide la fuerza en seis direcciones (tres ejes) utilizando un acelerómetro LIS302DL con escalas completas de $\frac{\pm 2}{\pm 8}$ dinámicamente seleccionables por el usuario y es capaz de medir aceleraciones con una tasa de salida de datos de 100 o 400 Hz [73] y una tasa de entrega de datos de 10 a 100 Hz.

Además de la orientación del dispositivo, el acelerómetro es capaz de medir el movimiento relativo de un dispositivo, *i.e.*, la traslación del mismo, por lo que sería viable con él crear aplicaciones capaces de dibujar una trayectoria seguida por el usuario al mover el dispositivo.

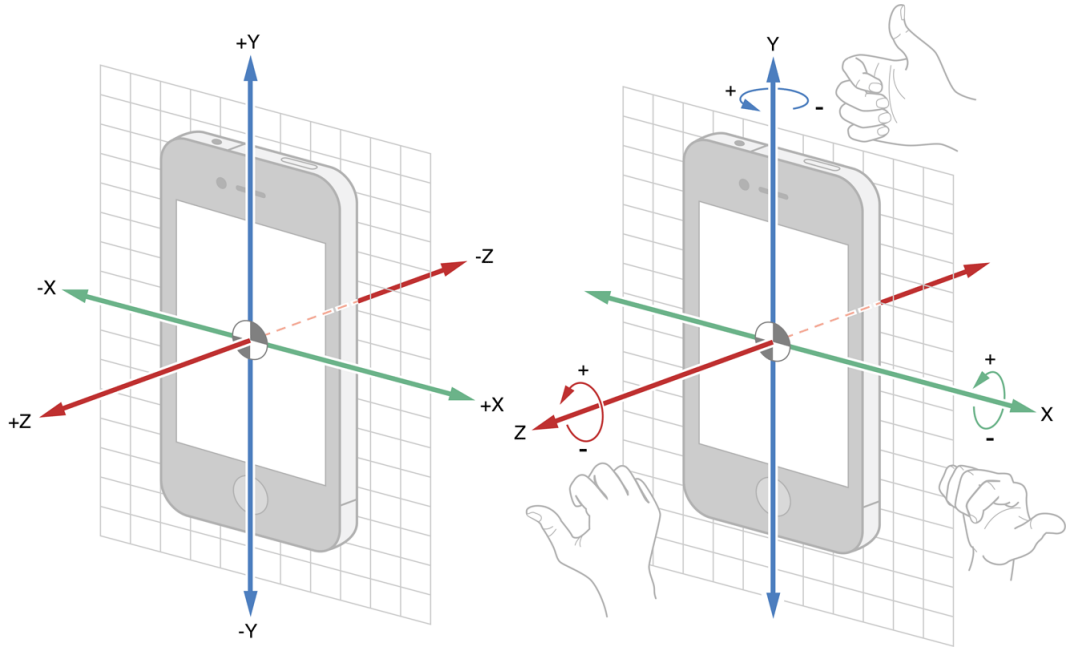
Por su parte, el giroscopio nos provee de la información de giro del dispositivo en una dirección en particular, *i.e.*, la rotación en cualquiera de los tres ejes distintos, rotación que no es posible medir empleando el acelerómetro.

El acelerómetro y el giroscopio se utilizan para propósitos similares, pero hay una sutil diferencia entre ambos, además de que muchas aplicaciones confían en la combinación de las lecturas de ambos para lograr una mayor eficiencia [41], *Laberinto* es una de esas aplicaciones.

Para hacer uso de estos sensores en *Objective-C*, el acelerómetro y giroscopio pueden ser utilizados mediante el *framework Core Motion* [74], el cual es responsable de acceder a los datos de ambos sensores cuando el usuario genera un evento de moción al mover, agitar o inclinar el dispositivo. Dichos eventos de movimiento son detectados por el hardware y *Core Motion* es capaz de pasar los datos leídos a la aplicación para que ésta lidie con ellos y los utilice para realizar alguna tarea específica.

La Figura 6.2 muestra los distintos ejes sobre los cuáles se mide la aceleración (en el caso del acelerómetro) y la rotación (en el caso del giroscopio) del dispositivo. *Core Motion* realiza las lecturas con base en estos ejes de coordenadas.

Como se puede observar en la imagen, los ejes son los mismos y estos no cambian sin importar la orientación a la que se encuentre el dispositivo, por lo que para mover la pelota virtual con ayuda del giroscopio, es necesario mapear las coordenadas correctamente, según se explicará en la sección 6.1.1.



(a) Acelerómetro. (b) Giroscopio.

Figura 6.2: Ejes de coordenadas del acelerómetro y giroscopio.

Core Motion emplea algoritmos únicos para procesar los datos que colecta para presentar información refinada de movimiento a la aplicación que lo haya solicitado. Dicho procesamiento ocurre en el hilo específico del *framework*. Cabe resaltar que se requiere un intervalo de actualización para la captura de datos, el cual mientras mayor sea, menores serán las capturas que se entreguen a la aplicación, lo cual es importante considerar si se desea no consumir demasiado la batería.

En los apartados siguientes se describe la manera en la que se utiliza el acelerómetro (sección 6.1.1) y giroscopio (sección 6.1.2) para lograr la funcionalidad deseada en la aplicación *Laberinto*.

6.1.1. Orientación del dispositivo

Una característica que causa conflicto al momento de desarrollar aplicaciones para el iPad es que no importa cuánto se trate de cuidar la detección correcta de la orientación del dispositivo, eventualmente se utilizará éste boca arriba mientras esté en posición horizontal, siendo que en estos casos el dispositivo marcará que se está utilizando en posición vertical y adaptará la interfaz a dicha orientación.

La razón es que mientras el dispositivo se encuentre boca arriba, el objeto *UIDevice* de *Objective-C* (comúnmente utilizado para determinar la orientación en una aplicación) no indica la orientación de la interfaz, sino la del dispositivo, *i.e.*, indica la orientación como *UIDeviceOrientationFaceUp* [75]. El problema viene cuando realmente debe considerarse la orientación de la interfaz para realizar una acción, puesto que los métodos que automáticamente la rotan no son llamados cuando se utiliza el dispositivo boca arriba.

En estos casos sólo se pueden aplicar dos soluciones (las cuales mejoran la detección de la orientación del dispositivo en cualquier situación):

1. Verificar la orientación de la barra de estado de la aplicación.
2. Emplear el acelerómetro para detectar un ángulo que sirve como discriminante para la detección de la orientación de la interfaz.

En el primer caso, gracias a que la barra de estado (barra que muestra el nivel de batería, conexión a la red, etc.) siempre sigue la orientación de la pantalla, se puede obtener la orientación de la interfaz a través de ésta independientemente de si la aplicación fue programada para que escondiera o no dicha barra, pues ésta siempre actualiza su estado sin importar la situación.

Sin embargo, en las aplicaciones desarrolladas, debido a que permitir que la interfaz se adaptara a la orientación del dispositivo provocaría que la vista previa de la cámara del iPad deba ser rotada (así como todos los elementos dibujados en la capa de *OpenGL ES*), se optó por programar las aplicaciones a manera de que sólo soportaran un tipo de orientación (en este caso la posición vertical), lo cual provoca que inclusive la barra de estado se conserve en un mismo lugar siempre, por lo que emplear el primer método para determinar la orientación de la interfaz no es viable.

En la segunda solución, se emplean las capturas del acelerómetro para calcular un ángulo (por medio de la función trigonométrica *arcotangente*) que sirve de discriminante para determinar la orientación de la interfaz, ya que si el ángulo se encuentra en un intervalo específico, se verifica si la orientación de la interfaz corresponde a la esperada con base en dicho ángulo y de no coincidir con la orientación ideal, se reasigna manualmente. De esta manera, los casos en los que por omisión, la orientación de la interfaz devuelve un valor que no coincide con las expectativas (como cuando se utiliza el dispositivo boca arriba sobre una superficie) no ocurren ya que la orientación es modificada gracias a este ángulo que funge como discriminante.

El Algoritmo 25 resume la forma en la que la aplicación *Laberinto* utiliza los valores del acelerómetro para determinar la orientación del dispositivo en la clase *Controlador de la vista*.

Algoritmo 25 Algoritmo que determina la orientación del dispositivo.

Entradas: Un objeto del tipo acelerómetro y los valores de aceleración (X, Y, Z).
Salida: La orientación a la que se encuentra el dispositivo.

1: $x \leftarrow -aceleracion_x$
2: $y \leftarrow aceleracion_y$
3: $angulo \leftarrow \tan^{-1}\left(\frac{y}{x}\right)$
4: **if** $angulo \geq -2.25$ && $angulo \leq -0.25$ **then**
5: **if** $self.orientacion \neq UIInterfaceOrientationPortrait$ **then**
6: $self.cambiarPosicion \leftarrow 1$
7: $self.orientacion \leftarrow UIInterfaceOrientationPortrait$
8: Cambiar la etiqueta de orientación del dispositivo en la capa de opciones.
9: **else if** $angulo \geq -1.75$ && $angulo \leq 0.75$ **then**
10: **if** $self.orientacion \neq UIInterfaceOrientationLandscapeRight$ **then**
11: $self.cambiarPosicion \leftarrow 2$
12: $self.orientacion \leftarrow UIInterfaceOrientationLandscapeRight$
13: Cambiar la etiqueta de orientación del dispositivo en la capa de opciones.
14: **else if** $angulo \geq 0.75$ && $angulo \leq 2.25$ **then**
15: **if** $self.orientacion \neq UIInterfaceOrientationPortraitUpsideDown$ **then**
16: $self.cambiarPosicion \leftarrow 3$
17: $self.orientacion \leftarrow UIInterfaceOrientationPortraitUpsideDown$
18: Cambiar la etiqueta de orientación del dispositivo en la capa de opciones.
19: **else if** $angulo \leq -2.25$ || $angulo \geq 2.25$ **then**
20: **if** $self.orientacion \neq UIInterfaceOrientationLandscapeLeft$ **then**
21: $self.cambiarPosicion \leftarrow 4$
22: $self.orientacion \leftarrow UIInterfaceOrientationLandscapeLeft$
23: Cambiar la etiqueta de orientación del dispositivo en la capa de opciones.

En este algoritmo, *self.cambiarPosicion* es una variable que reserva alguno de los siguientes valores: 1, 2, 3 o 4, para cuatro posibles orientaciones del dispositivo. Este valor es empleado por la clase *Opciones* para modificar el texto de una etiqueta que indica la orientación actual del dispositivo. Dicha etiqueta es útil para que el usuario entienda cómo se están tomando las coordenadas del giroscopio al mover la pelota virtual (ver sección 6.1.2). Cabe resaltar que en esta clase, las lecturas provenientes del acelerómetro no son capturadas por *Core Motion*, sino por un delegado por defecto de *Objective-C*, el cual trabaja de una manera similar a como lo hace *CoreMotion* al manejar estos datos.

Las lecturas del acelerómetro pueden ser refinadas aplicando un filtro de paso bajo o de paso alto, los cuales permiten el paso de frecuencias bajas o altas respectivamente [41]. Para aplicar dichos filtros, es necesario utilizar las expresiones (6.1) o (6.2) (donde $factor$ tiene un valor comúnmente de 0.1) según el filtro deseado [73].

$$bajo = aceleracion_{eje} \times factor + aceleracion_{previa} \times (1.0 - factor) \quad (6.1)$$

$$alto = aceleracion_{eje} - aceleracion_{eje} \times factor + aceleracion_{prev} \times (1.0 - factor) \quad (6.2)$$

El uso de estos filtros depende mucho de la aplicación, pero generalmente se emplea el filtro de paso bajo cuando la aceleración cambia lentamente y el filtro de paso alto cuando la aceleración cambia rápidamente. En el caso del filtro de paso bajo, se remueve todo el ruido en los datos recibidos, obteniendo únicamente las lecturas derivadas de la gravedad.

6.1.2. Movimiento de la pelota

Para realizar el movimiento de la pelota virtual tridimensional a través del laberinto bidimensional se hace uso del giroscopio para determinar la dirección (magnitud y sentido) en la cual se realiza el movimiento. El giroscopio realmente es más fácil de utilizar que el acelerómetro, auxiliándose esta vez de *Core Motion*, pues no se requiere de una clase delegada ni de métodos que gestionen las actualizaciones de los datos que éste captura.

Una vez que se ha iniciado el giroscopio, simplemente se solicitan las lecturas a través de la clase *CMMotionManager*, la cual actúa como una puerta de enlace hacia el hardware del dispositivo, además, para no realizar las lecturas manual y periódicamente, se puede hacer uso de un temporizador que dado un intervalo de tiempo, active o desactive las lecturas del giroscopio. La Figura 6.2(b) muestra los tres ejes de coordenadas que detecta el giroscopio al mover el dispositivo, así, para obtener la posición del dispositivo, se emplearían las coordenadas de estos ejes. La Tabla 6.1 resume los parámetros que componen a dicha posición.

Propiedad	**Eje**	**Descripción**
inclinación	`X`	Representa la inclinación del dispositivo en radianes, *i.e.*, una rotación alrededor del eje lateral que pasa a través del dispositivo de lado a lado.
rodaje	`Y`	Representa el nivel de rodaje en radianes, *i.e.*, una rotación alrededor del eje longitudinal que pasa a través del dispositivo desde el punto más alto hasta el más bajo.
derrape	`Z`	Representa el derrape en radianes, *i.e.*, una rotación alrededor del eje que corre verticalmente a través del dispositivo y es perpendicular al mismo, con su origen en el centro de gravedad y dirigido hacia el punto más bajo del dispositivo.

Tabla 6.1: Parámetros capturados por el giroscopio.

Comúnmente, los valores leídos son convertidos de radianes a grados, sin embargo, *Laberinto* no requiere esta conversión, pues dado que los valores en radianes son

pequeños se pueden aprovechar estas magnitudes y enviarse a la capa de dibujo, para incrementar o decrementar la posición de la pelota en un rango pequeño del espacio de coordenadas de *OpenGL ES*. El Algoritmo 26 muestra cómo se toman los valores del giroscopio en la clase *Controlador de la vista*.

Algoritmo 26 Algoritmo que captura los datos del giroscopio.

Entrada: Ninguna.

Salidas: La cantidad de movimiento en X y en Y obtenidas del giroscopio, así como la orientación actual del dispositivo.

1: $movimiento_x \leftarrow self.motionManager.gyroData.rotationRate.x$
2: $movimiento_y \leftarrow self.motionManager.gyroData.rotationRate.y$
3: $self.posicion_{xy} \leftarrow [NSNumber\ arrayWithCapacity\ :3]$
4: $self.posicion_{xy}[0] \leftarrow movimiento_x$
5: $self.posicion_{xy}[1] \leftarrow movimiento_y$
6: $self.posicion_{xy}[2] \leftarrow self.cambiarPosicion$
7: Llamar al método para mover la pelota en la capa de *OpenGL ES*

En este algoritmo, se toman sólo las lecturas de los giros en X y en Y, pues la pelota sólo puede moverse en dos direcciones al desplazarse sobre un laberinto $2D$ (líneas 1 y 2), además, el arreglo de posiciones $self.posicion_{xy}$ debe establecer sus dimensiones en cada ocasión por tratarse de un arreglo mutable y así evitar posibles problemas de memoria (línea 3).

Por su parte, *self.cambiarPosicion* corresponde al valor determinado en el Algoritmo 25 y debe ser enviado también a la capa de *OpenGL ES* para (con base en su valor) mapear correctamente las coordenadas (línea 6).

Una vez enviados los valores leídos por el giroscopio a la capa de *OpenGL ES*, se siguen los pasos del Algoritmo 27 para mover la pelota alrededor del laberinto. En la línea 2 se verifica que el valor proporcionado por el giroscopio tanto en X como en Y no sobrepasen un umbral y en caso de hacerlo, se les asigna el valor del umbral. Este paso debe realizarse o de lo contrario, se corre el riesgo de que la cantidad de movimiento proporcionada por el giroscopio sea tal que la pelota sobrepase alguna de las paredes del laberinto.

Posteriormente (a partir de la línea 3 y hasta la línea 14) se verifica la orientación actual del dispositivo, *i.e.*, *self.cambiarPosicion*, para determinar en qué orden o con qué signo se acomodan los valores de X y Y obtenidos del giroscopio para asignarlos a las posiciones siguientes de la pelota, pues como se dijo con anterioridad, los ejes del giroscopio no cambiarán y por ello se deben invertir los valores en algunos casos.

Una vez determinadas las nuevas posiciones, se verifica que no sobrepasen los límites del espacio de *OpenGL ES* designado para la pelota (línea 15) y de hacerlo, se les

asigna el valor mínimo o máximo según sea el caso. Posteriormente se determina el color del pixel en la imagen del laberinto en la posición sugerida por la pelota, para lo que se debe realizar el mapeo entre las coordenadas de la pelota y las de la imagen.

Conocida la intensidad del color, se verifica si ésta sobrepasa un umbral que muestra si se trata de un pixel negro o blanco. En caso de que el pixel sea blanco, se suman los valores del giroscopio a las coordenadas y se permite el movimiento y en caso contrario, se evita el movimiento y las coordenadas de la pelota se reasignan a sus coordenadas previas. De igual forma, *self.movimientoRapido* cambia de valor de acuerdo a esta misma condición. Esta variable sirve para saber si al dibujar la pelota en movimiento ésta rueda rápida o lentamente, creando un efecto más realista del movimiento de acuerdo a si la pelota está tocando una pared del laberinto o no.

Algoritmo 27 Algoritmo que mueve la pelota alrededor del laberinto.

Entradas: La cantidad de movimiento en X y en Y obtenidas del giroscopio, así como la orientación actual del dispositivo.

Salida: Ninguna.

1: Copiar en un nuevo arreglo los valores de entrada.
2: Verificar que la cantidad de movimiento no sobrepase un umbral determinado (tanto en X como en Y)
3: **if** $posicion_{xy}[2] = 1$ **then**
4: $\quad posicionSiguiente_x \leftarrow posicionSiguiente_x + posicion_{xy}[1]$
5: $\quad posicionSiguiente_y \leftarrow posicionSiguiente_y + posicion_{xy}[0]$
6: **else if** $posicion_{xy}[2] = 2$ **then**
7: $\quad posicionSiguiente_x \leftarrow posicionSiguiente_x + posicion_{xy}[0]$
8: $\quad posicionSiguiente_y \leftarrow posicionSiguiente_y - posicion_{xy}[1]$
9: **else if** $posicion_{xy}[2] = 3$ **then**
10: $\quad posicionSiguiente_x \leftarrow posicionSiguiente_x - posicion_{xy}[1]$
11: $\quad posicionSiguiente_y \leftarrow posicionSiguiente_y - posicion_{xy}[0]$
12: **else**
13: $\quad posicionSiguiente_x \leftarrow posicionSiguiente_x - posicion_{xy}[0]$
14: $\quad posicionSiguiente_y \leftarrow posicionSiguiente_y + posicion_{xy}[1]$
15: Verificar que ninguna de las coordenadas está fuera de los límites del espacio de *OpenGL ES*
16: $color \leftarrow determinarColorEnPosicion(posicionSiguiente_x, posicionSiguiente_y)$
17: **if** $color \neq -1.0$ && $color > umbral$ **then**
18: $\quad posicionActual_x \leftarrow posicionSiguiente_x$
19: $\quad posicionActual_y \leftarrow posicionSiguiente_y$
20: $\quad self.movimientoRapido \leftarrow YES$
21: **else**
22: $\quad posicionSiguiente_x \leftarrow posicionActual_x$
23: $\quad posicionSiguiente_y \leftarrow posicionActual_y$
24: $\quad self.movimientoRapido \leftarrow NO$

Ahora bien, el espacio de *OpenGL ES* sobre el cual se puede mover la pelota es un pequeño campo comprendido entre $[-10, +10]$ tanto en X como en Y, por lo que es necesario mapear las coordenadas comprendidas en ese espacio a coordenadas en la imagen del laberinto. La imagen del laberinto es de 512×512 pixeles, por lo que su espacio de coordenadas es de $[0, 511]$ en ambos ejes. La transformación de coordenadas es lineal (ver Ecuación 6.3), en este caso se utilizan los puntos $(-10, 0)$ y $(10, 511)$ en ambos ejes, *i.e.*, que en la coordenada -10 del espacio de *OpengGL ES* se obtendrá la coordenada 0 de la imagen del laberinto, y en la coordenada 10 se tendrá la coordenada 511.

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} \tag{6.3}$$

Así, si se sustitiuyen dichos puntos en la expresión (6.3) se obtiene la ecuación para el mapeo (Ecuación 6.4). Esta ecuación debe aplicarse en ambos ejes, *i.e.*, primero se obtiene la coordenada de la imagen para la coordenada en X de la pelota y posteriormente se hace lo mismo para Y y una vez que se obtengan estas dos nuevas coordenadas, se crea un punto con ellas y se busca la intensidad del color en la imagen del laberinto para dicho punto. El Algoritmo 28 resume este procedimiento.

$$\begin{aligned} \frac{x - (-10)}{10 - (-10)} &= \frac{y - 0}{511 - 0} \\ \frac{x + 10}{20} &= \frac{y}{511} \\ 511x + 5110 &= 20y \\ y &= \frac{511x + 5110}{20} \end{aligned} \tag{6.4}$$

Algoritmo 28 Algoritmo que determina el color en la imagen del laberinto.

Entradas: Las coordenadas de un punto (x, y) en el espacio virtual de *OpenGL ES*.
Salida: La intensidad de color en un punto de la imagen del laberinto.

1: $y_x \leftarrow \frac{511p_x+5110}{20}$
2: $y_y \leftarrow \frac{511p_y+5110}{20}$
3: $punto \leftarrow (floor(y_x), floor(y_y))$
4: $color \leftarrow obtenerColorEnPosicion(punto)$
5: **return** *color*

En este algoritmo se aprecia cómo se aplica dos veces la expresión (6.4) con base en las coordenadas de la pelota que se reciben como entrada, posteriormente se crea un punto con estas nuevas coordenadas y se obtiene la intensidad del color de la imagen del laberinto en dicho punto (línea 4). El algoritmo que determina la intensidad del color en una imagen dadas las coordenadas de un punto es el Algoritmo 29.

Algoritmo 29 Algoritmo que determina el color en una posición determinada de la imagen del laberinto.

Entradas: Las coordenadas de un punto en la imagen del laberinto.
Salida: La intensidad de color en ese mismo punto.

```
1: negro ← −1.0
2: laberinto ← [UIImage imageNamed :self.laberinto.archivoTextura.CGImage]
3: if laberinto es nulo then
4:     return −1.0
5: Obtener dimensiones de la imagen de textura.
6: Crear un contexto para la imagen.
7: Dibujar la imagen en el contexto creado.
8: Obtener los datos contenidos en la imagen del contexto creado.
9: if contexto es nulo then
10:    return −1.0
11: if datosImagen ≠ nulo then
12:    offset ← 4 × ((ancho × punto_y) + punto_x)
13:    alfa = datosImagen[offset]
14:    rojo = datosImagen[offset + 1]
15:    verde = datosImagen[offset + 2]
16:    azul = datosImagen[offset + 3]
17:    negro ← rojo + verde + azul + alfa
18: Liberar contexto.
19: Liberar memoria usada por los datos de la imagen.
20: return negro
```

En este algoritmo, se establece un valor inicial de -1 para el color negro (línea 1) el cual es devuelto en caso de no existir imagen del laberinto. Sin embargo, si la imagen existe, se crea un contexto temporal para poder ser dibujada sobre él y una vez obtenidos los datos de la imagen dibujada sobre el contexto (línea 8), se obtiene la información de los canales en el punto recibido como entrada y finalmente, se suma la información de cada uno de los canales para obtener el nivel de negro existente en dicho punto (línea 17). Este valor es devuelto por el algoritmo.

El valor de negro regresado por este algoritmo es fundamental para determinar si la pelota debe moverse o no, ya que en el Algoritmo 27 se verifica la condición de la línea 17 con base en dicho valor, *i.e.*, se comprueba que el valor sea diferente de -1 y que sobrepase un umbral para considerarlo negro.

Finalmente, una vez que se ha determinado si la pelota debe moverse o no, al momento de dibujarla sobre la capa de *OpenGL ES* se traslada en las coordenadas siguientes establecidas o bien, se mantiene en las mismas coordenadas. Por ello es de suma importancia que los valores de dichas coordenadas en el espacio de *OpenGL ES*

estén en el intervalo $[-10, 10]$, lo cual siempre se cumple ya que inicialmente la pelota se encuentra en el punto $(-10, -10)$ y los valores provenientes del giroscopio que se suman no son tan grandes y de serlo, son limitados. Además, si la pelota debe moverse rápidamente, se rota en X y en Y y si debe moverse lentamente también se rota, pero el ángulo es mucho menor. Cabe resaltar que si se cambia el laberinto sobre el que se desplaza la pelota virtual, entonces, las coordenadas de la pelota automáticamente vuelven a ser las coordenadas iniciales.

6.2. Modo de uso de la aplicación

La aplicación *Laberinto* tiene por objetivo emplear los sensores del iPad para mover una pelota 3D virtual sobre un laberinto 2D y son dibujados sobre la vista previa de la cámara una vez que se detecta un marcador de identificador simple en la escena.

La aplicación inicia al presionar en el iPad el icono mostrado en la Figura 6.3 y una vez iniciada se desplegará la vista previa de la cámara tal como ocurre con la aplicación *Fascículo de especies*, pero para que se dibuje el laberinto junto con la pelota, es necesario utilizar la tarjeta de RA de esta aplicación, misma que se muestra en la Figura 6.4 y cuyas dimensiones coinciden con las de las tarjetas de RA de la aplicación *Kanjirama*. Al voltear esta tarjeta, se visualiza un marcador del tipo patrón de bits sobre el cual se dibuja el laberinto y la pelota virtual.

Figura 6.3: Icono de la aplicación *Laberinto*.

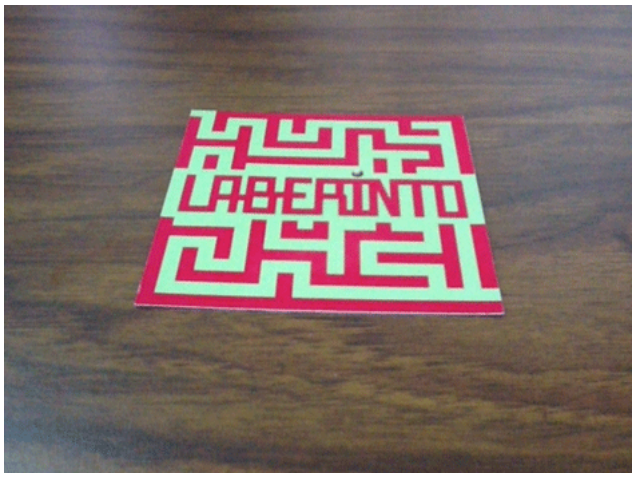
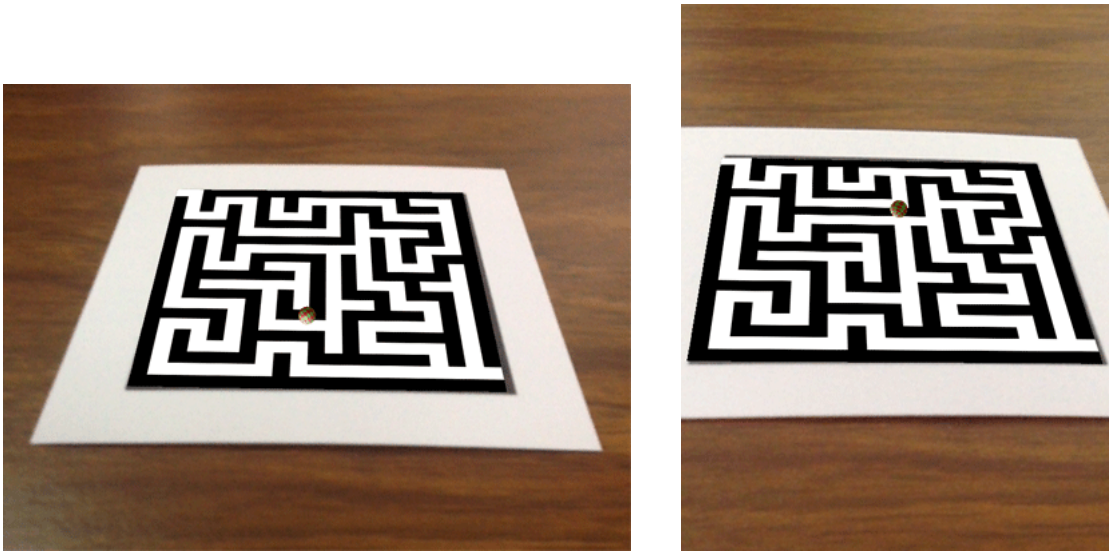


Figura 6.4: Tarjeta de RA de la aplicación *Laberinto*.

Como se puede apreciar en la imagen, la tarjeta de RA de esta última aplicación lleva en el reverso la misma figura que su icono, pero al ser volteada, el iPad puede visualizar el marcador y es entonces que se dibuja el laberinto junto con la pelota. Sin importar la orientación del dispositivo, la pelota se moverá en cualquiera de las cuatro direcciones posibles (arriba, abajo, derecha e izquierda) y aunque exista movimiento

tanto en X como en Y sin importar hacia dónde se mueva el iPad, generalmente el movimiento con una magnitud mayor sólo ocurre en una dirección, *e.g.*, si el iPad se mueve a la derecha, la magnitud con un valor numérico mayor se dará en X y en Y habrá movimiento pero con una magnitud menor, lo cual facilita la visualización del desplazamiento de la pelota a través del laberinto.

Algunos ejemplos de la pelota desplazándose sobre el laberinto utilizando el dispositivo en posición horizontal y vertical respectivamente se muestran en las Figuras 6.5(a) y 6.5(b).



(a) Dispositivo en posición horizontal.

(b) Dispositivo en posición vertical.

Figura 6.5: Ejemplos de la pelota dibujada sobre el laberinto.

En estas imágenes se puede observar la pelota en las partes media baja y media alta del laberinto respectivamente. En el primer caso (Figura 6.5(a)) el dispositivo se encuentra en posición horizontal y el movimiento del dispositivo es hacia la derecha, mientras que en el segundo caso (Figura 6.5(b)) el dispositivo se encuentra en posición vertical y el movimiento del dispositivo es hacia arriba.

En esta aplicación al igual que en *Fascículo de especies*, al tocar la pantalla del iPad aparece una interfaz de opciones. Ésta posee una etiqueta que muestra la orientación actual del dispositivo (de cuatro posibles), así como un elemento *UIPickerView* que sirve para cambiar la imagen del laberinto de manera instantánea y devolver la pelota a sus coordenadas iniciales. Sin importar si la pelota y el laberinto están dibujados sobre la vista previa de la cámara del iPad, la vista de opciones aparece al tocar la pantalla, pero si esta capa ya se encuentra dibujada y se toca la pantalla, desaparece.

La etiqueta con la orientación del dispositivo cambia de acuerdo al uso del mismo, auxiliada por el Algoritmo 25. Ejemplos de la etiqueta mostrando las cuatro posibles orientaciones del dispositivo según como éste se use, se muestran en la Figura 6.6.

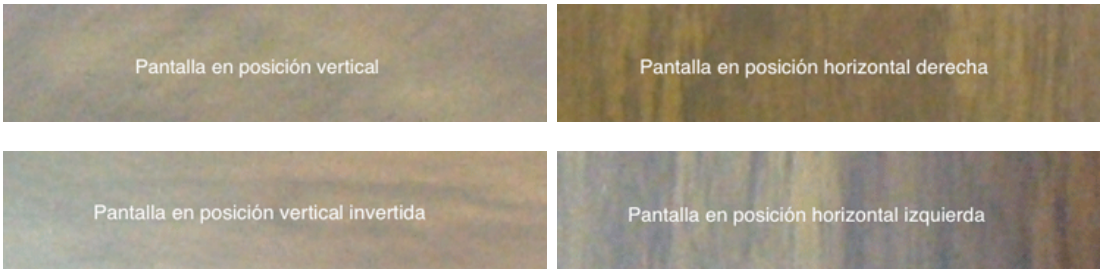
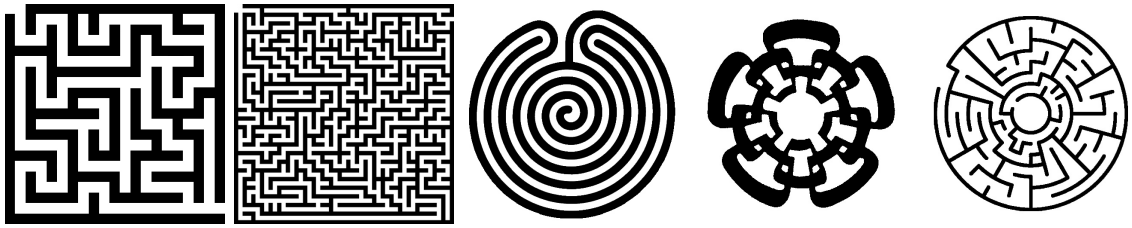


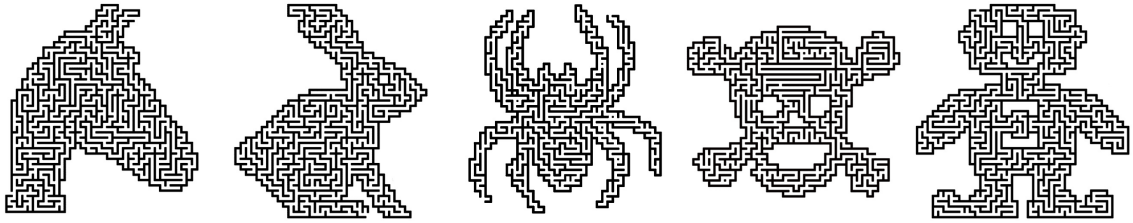
Figura 6.6: Ejemplos de la etiqueta con todas las posibles orientaciones del dispositivo.

Figura 6.7: Selector para cambiar la imagen del laberinto.

Además, gracias al *UIPickerView* (ver Figura 6.7), es posible cambiar la imagen del laberinto en la vista de opciones. Se puede elegir de entre diez laberintos distintos: laberinto 1, laberinto 2, espiral, logotipo CINVESTAV, laberinto circular, caballo, conejo, araña, pirata y robot. Cada vez que se cambia la imagen del laberinto, ésta lo hace de manera inmediata y la pelota vuelve a la posición inicial (punto $(-10, -10)$). Las imágenes de los laberintos se muestran en la Figura 6.8.



(a) laberinto 1. (b) laberinto 2. (c) espiral (d) CINVESTAV. (e) circular.



(f) caballo. (g) conejo. (h) araña. (i) pirata. (j) robot.

Figura 6.8: Laberintos distintos en la aplicación.

Como se puede observar, los grosores de las paredes de los laberintos no son iguales, por lo que los valores del umbral para limitar la cantidad de movimiento agregada por el giroscopio para que no se sobrepasen las paredes del laberinto variarán en función de la imagen actual del laberinto seleccionada. Algunos ejemplos más sobre el uso de la aplicación empleando distintos laberintos se pueden apreciar en la Figura 6.9

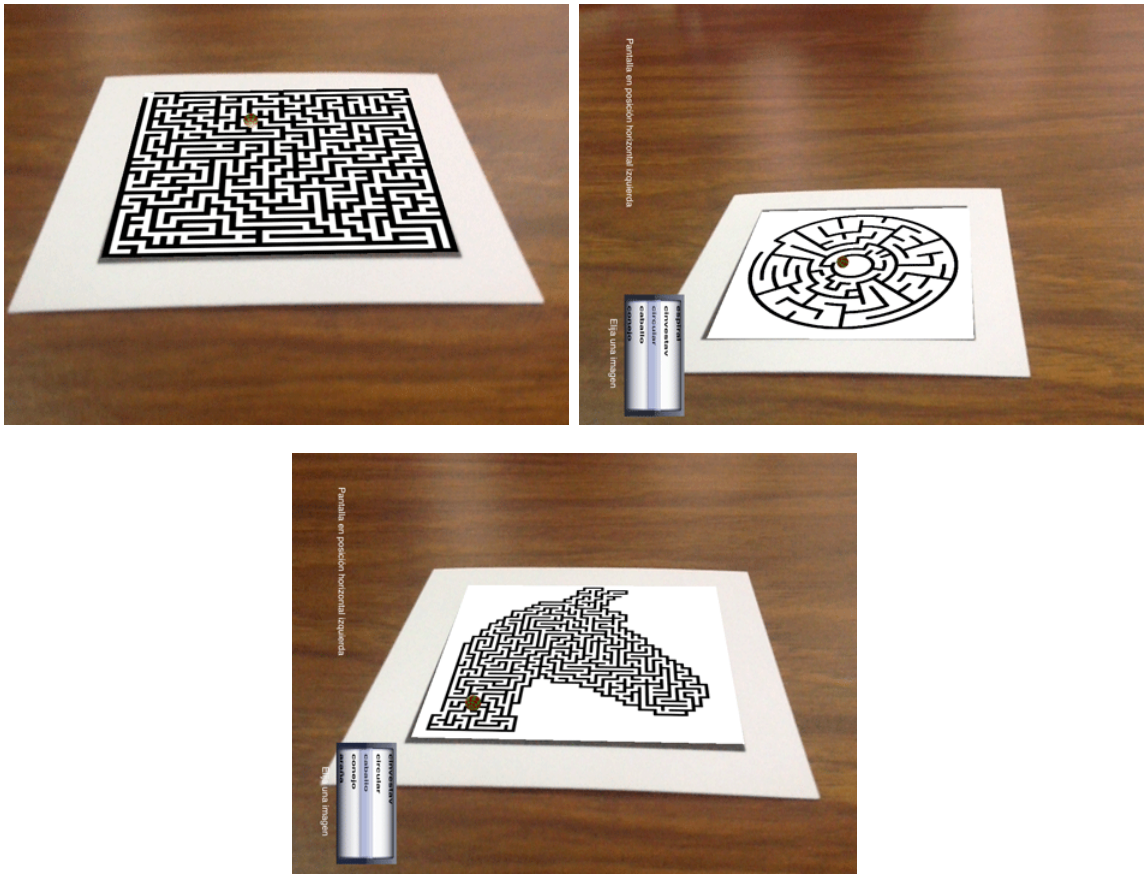


Figura 6.9: Ejemplos del movimiento de la pelota en distintos laberintos.

Cabe resaltar que a menos que el usuario cambie el laberinto al iniciar la aplicación, la imagen por defecto es la de *laberinto 1* (Figura 6.8(a)).

Al igual que *Kanjirama*, al usar un marcador del tipo patrón de bits en la aplicación *Laberinto*, la detección del mismo es más estable y aunado al hecho de que es el único marcador asociado a un modelo 3D, no hay posibilidad de que se dibuje algún modelo 3D diferente del laberinto y la pelota. En caso de presentar ante la vista previa de la cámara algún patrón diferente, no se dibujará ningún elemento virtual. Además, cuando se dibuja la escena con los objetos virtuales, se crean dos instancias de la clase *Objeto 3D*, una para el laberinto y otra para la pelota y cuando la imagen del laberinto es cambiada, se crea nuevamente la instancia del objeto laberinto y por ello puede ser dibujado de inmediato en la escena.

Básicamente, las acciones aquí descritas son las que se pueden realizar con la aplicación *Laberinto*.

Capítulo 7

Conclusiones

Se desarrolló un sistema de RA que se considera genérico y móvil, acoplable a cualquier aplicación de RA en dispositivos que utilicen *iOS*, pues los procesos principales (acceso a la cámara, procesamiento de los marcos y generación de objetos virtuales) y la estructura de las aplicaciones no cambian, *i.e.*, se ha propuesto la organización de cualquier aplicación de RA en *iOS* en al menos seis clases: el delegado de la aplicación, un controlador principal de la funcionalidad de la aplicación y la conexión de las vistas, una clase para modelar objetos 3D, una capa para dibujar dichos objetos mediante *OpenGL ES*, una clase para conectarse a la cámara del dispositivo y una última para la comunicación con el sistema de RA.

7.1. Conclusiones

El problema principal resuelto en este proyecto fue la migración de una aplicación demandante en recursos proveniente de un entorno completo a uno limitado, *i.e.*, limitantes en el procesamiento y en el manejo de memoria.

Para validar el sistema, se realizaron tres aplicaciones de RA:

- **Fascículo de especies:** esta pequeña libreta contiene un marcador por cada hoja y tras visualizarlos con la cámara del iPad, se despliega un objeto 3D referente a algún animal, desplegándose a su vez información relativa al mismo si se toca la pantalla. Se espera que se puedan añadir contenidos adicionales a cada especie (imágenes, sonidos, vídeo, etc). La idea surge dado el hecho de que la realidad aumentada se está presentando no sólo a base de marcadores impresos en tarjetas, sino también sobre libros, prendas de vestir, etc., y se están aplicando sobre todo para publicidad y videojuegos.
- **Aplicación para el aprendizaje de *kanji*:** similar a la aplicación desarrollada por Wagner y Barakonyi [42], se realizó una serie de tarjetas de RA que

contienen en la parte trasera un ideograma del idioma japonés y en la parte frontal un marcador que puede ser detectado al pasar la cámara del iPad cerca de él. La interacción de esta aplicación con el usuario es mediante un juego en el cual las tarjetas se encuentran sobre alguna superficie plana colocadas de modo que el símbolo sea visible, mientras que en alguna sección de la pantalla, se despliega una pequeña imagen alusiva al significado o pronunciación del *kanji* para que el usuario voltee la tarjeta correcta y así acertar y obtener puntos, con la intención de que memorice los símbolos de las tarjetas con su pronunciación o significado. Esta aplicación tiene especial cuidado al momento de voltear varias tarjetas incorrectas a la vez.

- **Desplazamiento de una pelota a través de un laberinto:** tomando como base una de las aplicaciones realizadas por Serna Rodríguez [15], en la cual se desplazaba una pelota virtual en el interior de una caja considerando que ésta no podía salir de los límites de la caja, de igual forma, la pelota de la aplicación desarrollada cruza a través de un laberinto que limita el movimiento de la misma. El desplazamiento de la pelota se realiza por medio del movimiento del iPad, auxiliándose de los sensores que éste posee.

Sin duda la cámara integrada en el iPad facilita bastante su uso en las aplicaciones, pues el delegado de *Objective-C* permite deshacerse de los marcos que son detectados tarde, además de que este delegado opera en otra cola de procesos independiente. Cabe resaltar que no hay necesidad de preocuparse por el acercamiento y alejamiento de la cámara.

Se emplearon marcadores de referencia basados en un marco cuadrado, los cuales son de uso más característico en aplicaciones de realidad aumentada para dispositivos móviles, pues son más flexibles y permiten formas libres en su interior, lo cual facilita el uso de diferentes diseños ya que al momento de la detección, antes que nada se espera encontrar dicho cuadrado en toda ocasión para seguir el proceso. Esto evita el diseño de múltiples marcadores de diferente tipo, *i.e.*, todas las formas están encerradas por un marco cuadrado, además de que usan el color negro sin importar la cantidad de éste en la escena.

Sin embargo, se probó con dos tipos de marcadores distintos, los marcadores del tipo plantilla y los patrones de bits y aunque ambos pudieron ser detectados con éxito, es más recomendable utilizar los patrones de bits, ya que la complejidad de la detección nunca cambia y los marcadores de plantilla pueden afectar su detección debido a su complejidad, además de que mientras más marcadores de este tipo se utilicen en la aplicación, el proceso de detección se vuelve más lento ya que se deben realizar más comparaciones y existen casos en los que marcadores muy parecidos entre sí, pueden ser detectados erróneamente a largas distancias. Por ello, se considera recomendable utilizar este tipo de marcadores cuando existan pocos en el sistema o todos los utilizados sean significativamente diferentes.

El manejo de información previa auxilió en la exitosa detección de marcadores, ya que se conserva el identificador del marcador detectado hasta que el marcador realmente cambie. Esto evita la carga constante de objetos virtuales en la escena. Además, dado a que se siguen las esquinas del marcador, se conserva la matriz de transformaciones el mayor tiempo y de esa manera no es necesario estar recalculando dicha matriz hasta que sea estrictamente necesario. Asimismo, la aplicación *Laberinto* guarda la posición previa de la pelota virtual hasta que el movimiento infringido a la pelota sea válido y la imagen capturada de la cámara se conserva el mayor tiempo posible en vez de estar capturando una nueva imagen tan frecuentemente.

Además, se realizan procesamientos multi-hilo, se intercalan procedimientos o bien, se establecen tiempos específicos al realizar ciertas tareas para acelerar las operaciones, *e.g.*, en el caso de la aplicación *Laberinto*, las lecturas provenientes del giroscopio se realizan en intervalos de $\frac{1}{30}$ de segundo para evitar que la captura de valores del giroscopio retrase o impida que se dibujen los objetos virtuales en la escena, pues este retraso ocurre si el tiempo asignado es menor.

En cuanto a la generación de objetos virtuales en la escena, es destacable que pueden dibujarse en escena y con mapeo de texturas desde 4 hasta 25000 vértices sin presentar ningún retraso al momento de seguir al objeto virtual ni al dibujarlo. Sin embargo, se probó con algunos objetos de casi 30000 o 40000 vértices presentándose un leve retraso al momento de dibujar el objeto, pero no al realizar el seguimiento del mismo. Aún así, este retraso no es muy significativo.

7.2. Trabajo a futuro

Algunas mejoras que se pueden hacer al sistema de RA o a las aplicaciones desarrolladas con la finalidad de tener un sistema más robusto o mejorar la funcionalidad de las aplicaciones son:

- Investigar sobre métodos alternativos en la detección de patrones del tipo plantilla para que no se requieran tantas comparaciones y el proceso no se vea afectado a mayor número de patrones agregados al sistema, además debe ser mejorado de forma tal que no haya detección errónea con plantillas similares.

- Aplicar métodos para la oclusión de objetos, *i.e.*, para que el objeto virtual permanezca en la escena un poco más de tiempo en caso de que el marcador sea total o parcialmente obstruido por algún objeto.

- Procurar la inclusión de más marcadores de tipos distintos a los utilizados, *e.g.*, los códigos bidimensionales (QR) que pueden ser detectados con o sin la inclusión del marco negro del patrón [44].

- Buscar o implementar un *script* alternativo al de *Heiko Behrens* [65] con la finalidad de que se puedan mapear objetos con más de una textura en las aplicaciones, o bien aplicar los principios propuestos en la sección 4.1 de manera tal que se puedan descargar objetos con más de una textura y mapearlas en una sola (como se hizo con varios modelos de las aplicaciones).

- Todos los objetos virtuales en esta tesis son rígidos y no se interactúa con ellos. Puede ser posible extender el trabajo para que se usen objetos articulados por ejemplo, u objetos deformables. A su vez, éstos podrían animarse reaccionando a algún evento ocurrido en el entorno.

- Pueden agregarse más especies al *Fascículo de especies* o bien realizarse nuevas propuestas para libros de RA.

- En la aplicación *Kanjirama*, a pesar de que no se agregaron símbolos de los más básicos en la enseñanza de esta lengua como 木 (ki ∼ árbol) por no encontrarse un modelo adecuado, esta aplicación podría mejorar considerablemente su funcionalidad si fuese útil para aprender palabras formadas por dos o más *kanji*, pues la mayoría de las palabras del idioma se forman de más de un *kanji*. Así, *e.g.*, una palabra como 飛行機 (*hikōki* ∼ avión) podría ser adivinada sólo al voltear las tres tarjetas en el orden correcto, para ello, no sería necesario dibujar modelos virtuales intermedios al voltear alguna de las tarjetas que forman el símbolo, ya que en su mayoría, los *kanji* intermedios no poseen un significado fácilmente representable, sólo se dibujaría el avión en la escena al voltear las tres tarjetas, lo cual requeriría un sistema capaz de detectar varios marcadores en la escena.

- Una mejora sugerible para la aplicación *Laberinto* es la creación de laberintos por parte del usuario, *i.e.*, para no limitarse al uso de un número determinado de laberintos preestablecidos, el usuario debería ser capaz de dibujar y guardar sus propios laberintos, aun cuando se limitara el grosor de la pluma de dibujo a uno mismo para todas las imágenes dibujadas.

- Finalmente, ya que se ha demostrado que mediante un sistema de RA generalizado y bien organizado es posible realizar distintas aplicaciones de la misma categoría con enfoques diferentes y con funcionalidades distintas en una plataforma tan prometedora como el iPad, asimismo, basándose en este proyecto y siguiendo una metodología similar es viable pensar en la migración de este sistema de RA a otras plataformas distintas, ya sean móviles o no.

Apéndice A

Figura A.1: Fascículo de especies (cubierta frontal y portada).

Figura A.2: Fascículo de especies (páginas 1 y 2).

Figura A.3: Fascículo de especies (páginas 3 y 4).

Figura A.4: Fascículo de especies (páginas 5 y 6).

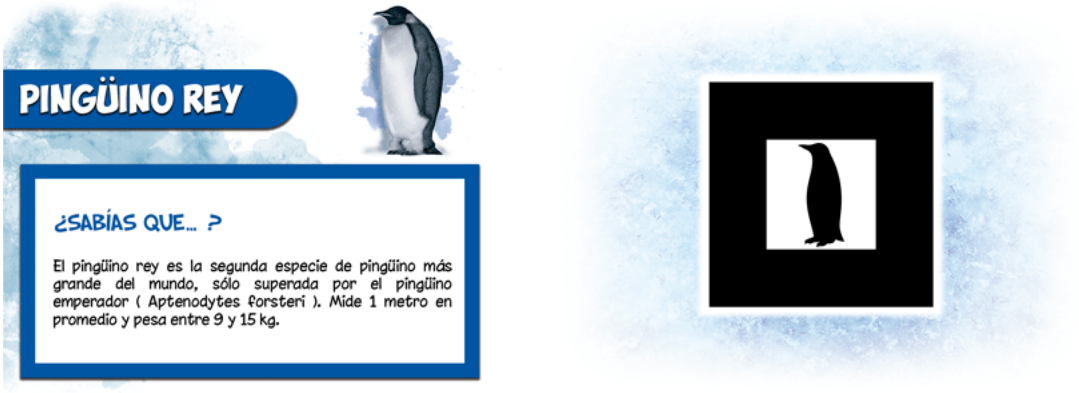


Figura A.5: Fascículo de especies (páginas 7 y 8).

Figura A.6: Fascículo de especies (páginas 9 y 10).

Figura A.7: Fascículo de especies (páginas 11 y 12).

Figura A.8: Fascículo de especies (páginas 13 y 14).

Figura A.9: Fascículo de especies (páginas 15 y 16).

Figura A.10: Fascículo de especies (páginas 17 y 18).

Figura A.11: Fascículo de especies (páginas 19 y 20).

Figura A.12: Fascículo de especies (páginas 21 y 22).

Figura A.13: Fascículo de especies (páginas 23 y 24).

Figura A.14: Fascículo de especies (páginas 25 y 26).

Figura A.15: Fascículo de especies (páginas 27 y 28).

Figura A.16: Fascículo de especies (páginas 29 y 30).

Figura A.17: Fascículo de especies (páginas 31 y 32).

Figura A.18: Fascículo de especies (páginas 33 y 34).

Figura A.19: Fascículo de especies (contraportada y cubierta trasera).

Bibliografía

[1] Ronald T. Azuma. *A survey of augmented reality. Presence: Teleoperators and Virtual Environments*, 6(4):355–385, August 1997.

[2] Mark Billinghurst, Hirokazu Kato, Suzanne Weghorst, and Tom Furness. *A Mixed Reality 3D Conferencing Application.* Technical report, Seattle: Human Interface Technology Laboratory at University of Washington, 1999. `http://www.hitl.washington.edu/publications/r-99-1/`.

[3] Julie Carmigniani, Borko Furht, Marco Anisetti, Paolo Ceravolo, Ernesto Damiani, and Misa Ivkovic. *Augmented reality technologies, systems and applications.* Technical report, Department of Computer and Electrical Engineering and Computer Sciences at Florida Atlantic University and Dipartimento di Tecnologie dell'Informazione, Universita' degli studi di Milano, Milano, Italy and University of Novi Sad, Serbia, 2010.

[4] Daniel Wagner and Dieter Schmalstieg. *Making Augmented Reality Practical on Mobile Phones, Part 1. Computer Graphics and Applications, IEEE*, 29(3):12 –15, May-June 2009.

[5] *Comparativa entre el iPad 2 y el de tercera generación.* `http://www.geek.com/articles/gadgets/new-ipad-vs-ipad-2-how-the-specs-compare-2012038/`. Consultada el 26 de Septiembre del 2012.

[6] *Especificaciones del iPad.* `http://www.apple.com/ipad/specs/`. Consultada el 26 de Septiembre del 2012.

[7] *Camera Zoom 3 and Aperture.* `http://itunes.apple.com/mx/app/`. Consultada el 26 de Septiembre del 2012.

[8] *Accesorios para el iPad.* `http://www.ipadcameralens.com/`. Consultada el 26 de Septiembre del 2012.

[9] *Sitio Oficial de OpenGL ES.* `http://www.khronos.org/opengles/`. Consultada el 26 de Septiembre del 2012.

[10] Daniel Wagner and Dieter Schmalstieg. *Making Augmented Reality Practical on Mobile Phones, Part 2. Computer Graphics and Applications, IEEE*, 29(4):6 –9, May-June 2009.

[11] Michael Gervautz and Dieter Schmalstieg. *Anywhere Interfaces Using Handheld Augmented Reality. IEEE Computer: Innovative Technology for Computer Professionals*, 45(7):26–31, July 2012. Published by IEEE Computer Society.

[12] John Zelek, Ehsan Fazl, Daniel Asmar, and Adel Fakih. *Computer Vision GeoLocation, Awareness and Detail.* In *COM. Geo 2010*, Waterloo, Canada, June 21-23 2010. University of Waterloo, Canada, ACM.

[13] Mohamed Ashraf Nassar and Fatma Meawad. *An Augmented Reality Exhibition Guide for the iPhone.* In *2010 International Conference on User Science Engineering (i-USEr).*

[14] Rosa Atzín Vázquez del Ángel. *Sistema de Desarrollo para Aplicaciones de Realidad Aumentada.* Tesis de Maestría, Centro de Investigación y de Estudios Avanzados del IPN, CINVESTAV-IPN, Unidad Zacatenco, Octubre 2010.

[15] Norma Irene Serna Rodríguez. *Sistema de Realidad Aumentada.* Tesis de Maestría en Ciencias en Ingeniería Eléctrica, Centro de Investigación y de Estudios Avanzados del IPN, CINVESTAV-IPN, Unidad Guadalajara, Noviembre 2006.

[16] Oliver Bimber. *What's Real About Augmented Reality? IEEE Computer: Innovative Technology for Computer Professionals*, 45(7):24–25, July 2012. Published by IEEE Computer Society.

[17] Mark Mine, David Rose, Bei Yang, Jeroen van Baar, and Anselm Grundhöfer. *Projection-Based Augmented Reality in Disney Theme Parks. IEEE Computer: Innovative Technology for Computer Professionals*, 45(7):32–40, July 2012. Published by IEEE Computer Society.

[18] Nassir Navab, Tobias Blum, Lejing Wang, Asli Okur, and Thomas Wendler. *First Deployments of Augmented Reality in Operating Rooms. IEEE Computer: Innovative Technology for Computer Professionals*, 45(7):48–55, July 2012. Published by IEEE Computer Society.

[19] IEEE, editor. *Augmented reality in a serious game for manual assembly processes*, Berlin, Germany, October 26-29 2011. Fraunhofer Institute for Production Systems and Design Technology, Technische Universität Berlin, Springer.

[20] Mark Billinghurst and Andreas Dünser. *Augmented Reality in the Classroom. IEEE Computer: Innovative Technology for Computer Professionals*, 45(7):56–63, July 2012. Published by IEEE Computer Society.

[21] Ailsa Barry, Jonathan Trout, Paul Debenham, and Graham Thomas. *Augmented Reality in a Public Space: The Natural History Museum, London. IEEE Computer: Innovative Technology for Computer Professionals*, 45(7):42–47, July 2012. Published by IEEE Computer Society.

[22] *Manual del sistema Nintendo 3DS™*. `http://www.nintendo.com/consumer/manuals/es_la/3ds.jsp`. Consultada el 8 de Septiembre del 2012.

[23] *ARToolKitPlus*. `https://launchpad.net/artoolkitplus`. Consultada el 12 de Septiembre del 2012.

[24] *Vuforia*. `https://developer.qualcomm.com/mobile-development/mobile-technologies/augmented-reality`. Consultada el 12 de Septiembre del 2012.

[25] *String*. `http://www.poweredbystring.com/product`. Consultada el 19 de Septiembre del 2012.

[26] *NyARToolKit*. `http://www.poweredbystring.com/product`. Consultada el 12 de Septiembre del 2012.

[27] *ARTag*. `http://www.artag.net/`. Consultada el 12 de Septiembre del 2012.

[28] *Studierstube ES*. `http://studierstube.icg.tugraz.at/handheld_ar/stbtracker.php`. Consultada el 19 de Septiembre del 2012.

[29] *FLARToolKit*. `http://mlab.taik.fi/mediacode/archives/1939`. Consultada el 19 de Septiembre del 2012.

[30] *Popcode*. `http://popcode.info/whatispopcode.html`. Consultada el 21 de Septiembre del 2012.

[31] Nate Hagbi, Oriel Bergig, Jihad El-Sana, and Mark Billinghurst. *Shape Recognition and Pose Estimation for Mobile Augmented Reality*. In *Mixed and Augmented Reality, 2009. ISMAR 2009. 8th IEEE and ACM International Symposium on*, pages 65 –71, October 2009.

[32] *Layar*. `http://www.layar.com/what-is-layar/`. Consultada el 21 de Septiembre del 2012.

[33] *Wikitude*. `http://www.wikitude.com/tour/wikitude-world-browser`. Consultada el 19 de Septiembre del 2012.

[34] *VRToolKit*. `http://www.benjaminloulier.com/projects/VRToolKit/doc/index.html`. Consultada el 12 de Septiembre del 2012.

[35] Stephen DiVerdi, Daniel Nurmi, and Tobias Höllerer. *ARWin - A Desktop Augmented Reality Window Manager*. In *Mixed and Augmented Reality, 2003. ISMAR 2003. 2^{th} IEEE and ACM International Symposium on*, page 298, May 2003.

[36] *3DAR*. `http://3dar.us/`. Consultada el 21 de Septiembre del 2012.

[37] *AR2 3D*. `http://ar23d.com/augmented-reality-sdk.html`. Consultada el 21 de Septiembre del 2012.

[38] *Metaio*. `http://www.metaio.com/software/mobile-sdk/`. Consultada el 21 de Septiembre del 2012.

[39] *D'Fusion*. `https://community.t-immersion.com/`. Consultada el 21 de Septiembre del 2012.

[40] *droidar*. `http://code.google.com/p/droidar//`. Consultada el 21 de Septiembre del 2012.

[41] Kyle Roche. *Pro iOS 5 Augmented Reality*, chapter 4 and 7 and 8. Apress, 2011.

[42] Daniel Wagner and Istvan Barakonyi. *Augmented Reality Kanji Learning*. In *Mixed and Augmented Reality, 2003. Proceedings. The Second IEEE and ACM International Symposium on*, pages 335 – 336, October 2003.

[43] *Historia del código de barras*. `http://www.codigodebarras.pe/codigo-de-barras-historia/`. Consultada el 22 de Septiembre del 2012.

[44] Tai-Wei Kan, Chin-Hung Teng, and Wen-Shou Chou. *Applying QR code in augmented reality applications*. In *Proceedings of the 8th International Conference on Virtual Reality Continuum and its Applications in Industry*, VRCAI '09, pages 253–257, New York, NY, USA, 2009. ACM.

[45] Jian-tung Wang, Chia-Nian Shyi, T.-W. Hou, and C.P. Fong. *Design and implementation of augmented reality system collaborating with QR code*. In *Computer Symposium (ICS), 2010 International*, pages 414 –418, December. 2010.

[46] *Arquitectura iOS*. `http://disanji.net/iOS_Doc/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/IPhoneOSOverview/IPhoneOSOverview.html`. Consultada el 31 de Octubre del 2012.

[47] *Manejo de memoria en Objective-C*. `http://www.raywenderlich.com/2657/memory-management-in-objective-c-tutorial`. Consultada el 30 de Enero del 2013.

[48] *Ejemplo de OpenGL ES en iOS*. `http://www.raywenderlich.com/3664/opengl-es-2-0-for-iphone-tutorial`. Consultada el 30 de Enero del 2013.

[49] *Manual de referencia de OpenGL ES 2.0 y 3.0.* `http://www.khronos.org/opengles/sdk/docs/`. Consultada el 3 de Octubre del 2012.

[50] Kim Jaeyoung and Jun Heesung. *Implementation of Image Processing and Augmented Reality Programs for Smart Mobile Device.* In *2011 The* 6^{th} *International Forum on Strategic Technology.*

[51] *Sistema de coordenadas en OpenGL.* `http://www.opengl.org/archives/resources/faq/technical/transformations.htm`. Consultada el 25 de Enero del 2013.

[52] *Sitio para generar archivos de tipo patrón a partir de imágenes de marcadores.* `http://flash.tarotaro.org/blog/2008/12/14/artoolkit-marker-generator-online-released/`. Consultada el 26 de Enero del 2013.

[53] Francisco Jurado, Javier A. Albusac, José J. Castro, David Vallejo, Luis Jiménez, Félix J. Villanueva, David Villa, Carlos González, and Guillermo Simmross. *Desarrollo de Videojuegos: Desarrollo de Componentes.* Bubok.

[54] *Desmantelando ARToolKit, parte 1: etiquetado.* `https://xgoat.com/wp/2011/05/08/dismantling-artoolkit-part-1-labelling/`. Consultada el 10 de Enero del 2013.

[55] *Desmantelando ARToolKit, parte 2: detección de contornos.* `http://chriskirkham.co.uk/2011/06/27/dismantling-artoolkit-part-2-contour-detection/`. Consultada el 12 de Enero del 2013.

[56] *Desmantelando ARToolKit, parte 3: localizando vértices.* `http://chriskirkham.co.uk/2011/07/14/dismantling-artoolkit-part-3-locating-vertices/`. Consultada el 12 de Enero del 2013.

[57] Alok Sharma, Kuldip K. Paliwal, Seiya Imoto, and Satoru Miyano. *Principal component analysis using QR decomposition. International Journal of Machine Learning and Cybernetics*, pages 1–5, 2012.

[58] *Consultar parámetros de la cámara para transformar una imagen en una escena 3D a 2D.* `http://www2.teu.ac.jp/clab/kondo/research/cadcgtext/Chap5/Chap503.html`. Consultada el 28 de Enero del 2013.

[59] Zhengyou Zhang. *A flexible new technique for camera calibration. Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(11):1330–1334, 2000.

[60] *Modelos de calibración para archivos genéricos.* `http://www.hitl.washington.edu/artoolkit/documentation/usercalibration.htm`. Consultada el 26 de Enero del 2013.

[61] *Sesenta excelentes sitios para descargar modelos 3D*. http://www.hongkiat.com/blog/60-excellent-free-3d-model-websites/. Consultada el 12 de Enero del 2013.

[62] *TurboSquid*. http://www.turbosquid.com/. Consultada el 12 de Enero del 2013.

[63] *Top3Dmodels*. http://www.top3dmodels.com/. Consultada el 12 de Enero del 2013.

[64] *Archive3D*. http://archive3d.net/. Consultada el 12 de Enero del 2013.

[65] *Script de Heiko Behrens para convertir modelos en formato .obj a archivos de cabecera (.h) para utilizarlos en OpenGL ES*. http://heikobehrens.net/2009/08/27/obj2opengl/. Consultada el 8 de Enero del 2013.

[66] *Blender, software para el modelado 3D*. http://www.blender.org/. Consultada el 18 de Enero del 2013.

[67] *Opción de suavizado en Blender*. http://blenderartists.org/forum/showthread.php?258708-what-is-smooth-vertex. Consultada el 18 de Enero del 2013.

[68] Daniel Wagner and Dieter Schmalstieg. *ARToolKitPlus for Pose Tracking on Mobile Devices*. Technical report, Institute for Computer Graphics and Vision, Graz University of Technology, February 2007.

[69] *Nociones básicas del idioma japonés*. http://www.aprendejapones.com/. Consultada el 3 de Mayo del 2013.

[70] *Categorías de Kanji*. http://www.sljfaq.org/afaq/how-many-kanji.html. Consultada el 3 de Mayo del 2013.

[71] *Diccionario de Kanji*. http://jisho.org/. Consultada el 3 de Mayo del 2013.

[72] *Japanese-Language Proficiency Test*. http://www.jlpt.jp/. Consultada el 3 de Mayo del 2013.

[73] *Manejo del acelerómetro en iOS*. http://homepages.ius.edu/RWISMAN/C490/html/chapter22.htm. Consultada el 12 de Mayo del 2013.

[74] *Documentación Core Motion*. http://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/motion_event_basics/motion_event_basics.html. Consultada el 12 de Mayo del 2013.

[75] *¿Cómo determinar correctamente la orientación del dispositivo en iOS?* http://ddeville.me/2011/01/getting-the-interface-orientation-in-ios/. Consultada el 15 de Mayo del 2013.

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL INSTITUTO POLITÉCNICO NACIONAL

Departamento de Computación

Después de haber efectuado la revisión del trabajo de tesis titulado

"Realidad aumentada utilizando un iPad".

Realizado por el alumno César David Corona Arzola, bajo la dirección del Doctor Luis Gerardo de la Fraga.

1 El documento final de tesis cumple con los requisitos para el Programa de Maestría en Ciencias.

2 Solicitamos que la fecha de Examen de Grado se lleve a cabo el día 25 de septiembre a las 12:00 del medio día.

3 Firman la presente el día 10 de septiembre del año 2013:

Dr. Luis Gerardo de la Fraga

Dra. Sonia Guadalupe Mendoza Chapa

Dr. José Guadalupe Rodríguez García

Av. Instituto Politécnico Nacional # 2508 Col. San Pedro Zacatenco México, D.F. C.P. 07360
Tel.: 5747-3800 Fax: 5747-7002