Realidad Aumentada

Un Enfoque Práctico con ARToolKit y Blender

Carlos González Morcillo
David Vallejo Fernández
Javier A. Albusac Jiménez
José Jesús Castro Sánchez

IdenTIC consorcio

esi Escuela Superior de Informática

Paseo de la Universidad, 4
13071, Ciudad Real
Email: esi@uclm.es

Título: Realidad Aumentada.
Un Enfoque Práctico con ARToolKit y Blender.
Autores: Carlos González Morcillo,
David Vallejo Fernández,
Javier Alonso Albusac Jiménez y
José Jesús Castro Sánchez
ISBN: 978-84-686-1151-8 (*Edición en Papel*)
Edita: Bubok Publishing S.L.
Diseño: Carlos González Morcillo
Impreso en España

Este libro fue compuesto con LaTeX a partir de una plantilla de Carlos González Morcillo y Sergio García Mondaray. La portada y las entradillas fueron diseñadas con GIMP, Blender, InkScape y OpenOffice.

Escuela Superior de Informática

IdenTIC consorcio

Prefacio

La *Realidad Aumentada* (RA) es una variación de *Realidad Virtual*. Las tecnologías de Realidad Virtual sumergen al usuario dentro de un entorno completamente sintético, sin tener consciencia del mundo real que lo rodea. La RA, sin embargo, permite al usuario ver el mundo real, en el que se superponen o con el que se componen objetos virtuales. Así, la Realidad Aumentada no sustituye la realidad, sino que la complementa.

La RA ha experimentado un creciente interés en estos últimos años. En Septiembre de 2009, The Economist aseguró que «intentar imaginar cómo se utilizará la Realidad Aumentada es como intentar predecir el futuro de la web en 1994». Según la consultora *Juniper Research*, la RA en dispositivos móviles, generará más de 732 millones de dólares en 2014.

Este libro pretende proporcionar una visión práctica para desarrollar aplicaciones de RA, utilizando para ello la biblioteca ARToolkit para el desarrollo de este tipo de aplicaciones y la suite de modelado tridimensional Blender. Así mismo, el presente libro cubre aspectos esenciales en este ámbito, como por ejemplo la base de los fundamentos matemáticos necesarios para trabajar en el mundo 3D o el uso de APIs de programación gráfica como OpenGL. Finalmente, también se lleva a cabo un estudio del uso de las bibliotecas OpenCV y Ogre3D.

Sobre este libro...

Este libro que tienes en tus manos ha surgido de los apuntes del Curso de Realidad Aumentada impartido en las instalaciones del Consorcio Identic en Casar de Cáceres en Julio de 2011. Puedes obtener más información sobre las actividades del consorcio en www.identic.es.

La versión electrónica de este libro (y los ejemplos de código) pueden descargarse de la web: www.librorealidadaumentada.com. Salvo que se especifique explícitamente otra licencia, todos los ejemplos del libro se distribuyen bajo GPLv3. El libro «físico» puede adquirirse desde la página web de la editorial online *Bubok* en http://www.bubok.es.

Requisitos previos

Este libro tiene un público objetivo con un perfil principalmente técnico. Al igual que el curso del que surgió, está orientado a la capacitación de profesionales en el ámbito de la realidad aumentada basada en tracking con marcas.

Se asume que el lector es capaz de desarrollar programas de nivel medio en C/C++. El libro está orientado principalmente para titulados o estudiantes de últimos cursos de Ingeniería en Informática.

Agradecimientos

Los autores del libro quieren agradecer en primer lugar a los alumnos de las dos ediciones del *Curso de Realidad Aumentada* impartido en 2010 y 2011 en las instalaciones el Consorcio Identic (Casar de Cáceres), así como al personal técnico y de administración de Identic, por la ilusión en el curso y el fantástico ambiente en clase.

Por otra parte, este agradecimiento también se hace extensivo a la Escuela de Informatica de Ciudad Real y al Departamento de Tecnologías y Sistema de Información de la Universidad de Castilla-La Mancha.

Autores

Carlos González (2007, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Titular de Universidad e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en asignaturas relacionadas con Informática Gráfica, Síntesis de Imagen Realista y Sistemas Operativos desde 2002. Actualmente, su actividad investigadora gira en torno a los Sistemas Multi-Agente, el Rendering Distribuido y la Realidad Aumentada.

Javier Albusac (2009, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor e imparte docencia en la Escuela de Ingeniería Minera e Industrial de Almadén (EIMIA) en las asignaturas de Informática, Ofimática Aplicada a la Ingeniería y Sistemas de Comunicación en Edificios desde 2007. Actualmente, su actividad investigadora gira en torno a la Vigilancia Inteligente, Robótica Móvil y Aprendizaje Automático.

David Vallejo (2009, Doctor Europeo en Informática, Universidad de Castilla-La Mancha) es Profesor Ayudante Doctor e imparte docencia en la Escuela de Informática de Ciudad Real (UCLM) en asignaturas relacionadas con Informática Gráfica, Programación y Sistemas Operativos desde 2007. Actualmente, su actividad investigadora gira en torno a la Vigilancia Inteligente, los Sistemas Multi-Agente y el Rendering Distribuido.

José Jesús Castro es Profesor Titular de Universidad en el área de Lenguajes y Sistemas Informáticos, desde 1999 imparte docencia en la Escuela Superior de Informática de la UCLM. Sus temas de investigación están relacionados con el uso y desarrollo de métodos de IA para la resolución de problemas reales, donde cuenta con una amplia experiencia en proyec-tos de investigación, siendo autor de numerosas publicaciones.

Índice general

Capítulo 1

Introducción

La Realidad Aumentada ofrece diversas posibilidades de interacción que pueden ser explotadas en diferentes ámbitos de aplicación. En este capítulo definiremos qué se entiende por Realidad Aumentada, así como algunas de las soluciones tecnológicas más empleadas.

1.1. Introducción

La informática gráfica es un área en continuo desarrollo motivado, entre otros factores, por los grandes intereses comerciales que la impulsan desde el ámbito de los videojuegos, el cine y la televisión, las aplicaciones médicas e industriales, simulación, etc... En algunos entornos puede ser necesario integrar la representación de imagen sintética con imágenes del mundo real.

La Realidad Aumentada se encarga de estudiar las técnicas que permiten integrar en tiempo real contenido digital con el mundo real. Según la taxonomía descrita por Milgram y Kishino [16], los entornos de Realidad Mixta son aquellos en los que "se presentan objetos del mundo real y objetos virtuales de forma conjunta en una única pantalla". Esto abre un abanico de definiciones en la que se sitúan las aplicaciones de Realidad Aumentada (ver Figura 1.2).

A diferencia de la Realidad Virtual donde el usuario interactúa en un mundo totalmente virtual, la Realidad Aumentada se ocupa de generar capas de información virtual que deben ser correctamente ali-

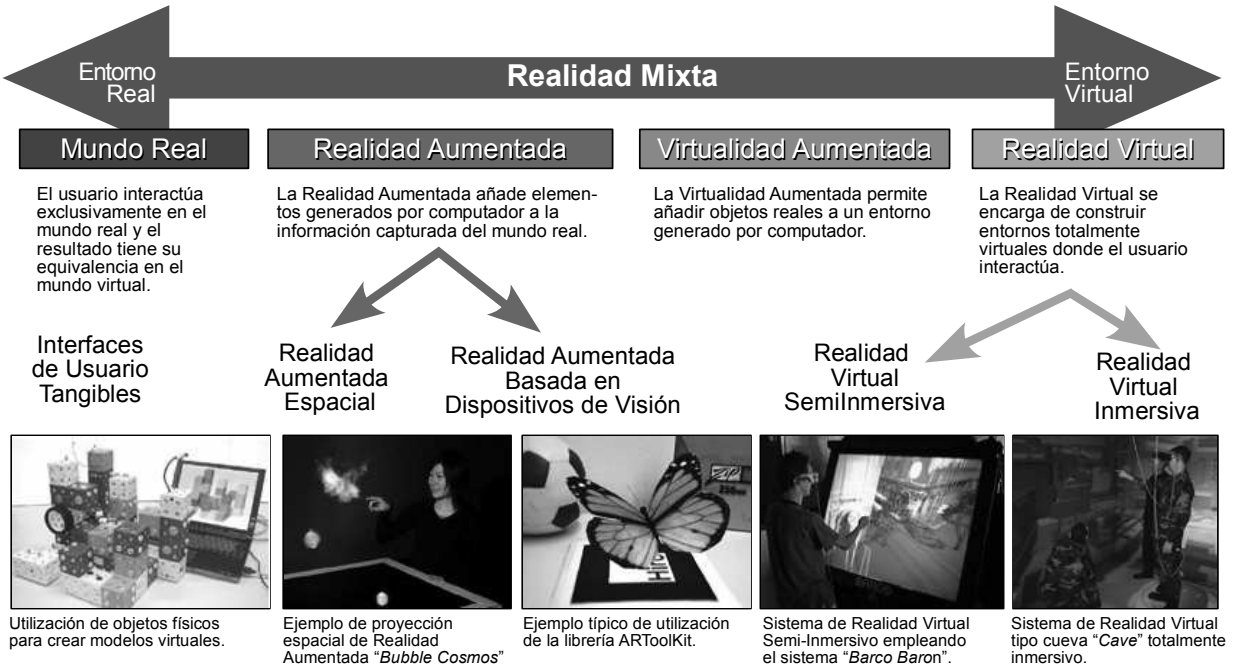


Figura 1.2: Taxonomía de Realidad Mixta según Milgram y Kishino.

neadas con la imagen del mundo real para lograr una sensación de correcta integración.

El principal problema con el que deben tratar los sitemas de Realidad Aumentada es el denominado **registro**, que consiste en calcular la posición relativa de la cámara real respecto de la escena para poder generar imágenes virtuales correctamente alineadas con esa imagen real. Este registro debe ser preciso (errores de muy pocos milímetros son muy sensibles en determinadas aplicaciones, como en medicina o en soporte a las aplicaciones industriales) y robusto (debe funcionar correctamente en todo momento). En la figura 1.1 puede verse un ejemplo de una imagen obtenida con un registro correcto (centro) e incorrecto (abajo). Este registro puede realizarse empleando diferentes tipos de sensores y técnicas (las más extendidas son mediante el uso *tracking visual*). La problemática del registro y las técnicas de tracking se estudiarán en el capítulo 2.

Así, la Realidad Aumentada se sitúa entre medias de los entornos reales y los virtuales, encargándose de construir y alinear objetos virtuales que se integran en un escenario real.

Figura 1.1: ***Arriba***: Escena real. El problema del registro trata de calcular la posición de la cámara real para poder dibujar objetos virtuales correctamente alineados. ***Centro***: Registro correcto. ***Abajo***: Registro incorrecto.

1.1.1. Un poco de historia

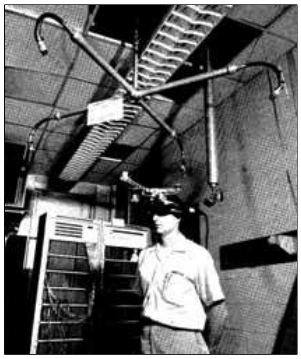


Figura 1.3: Primer Sistema de Realidad Aumentada de Sutherland.

El primer sistema de Realidad Aumentada fue creado por Ivan Sutherland [25] en 1968, empleando un casco de visión que permitía ver sencillos objetos 3D renderizados en *wireframe* en tiempo real. Empleaba dos sistemas de tracking para calcular el registro de la cámara; uno mecánico y otro basado en ultrasonidos (ver Figura 1.3).

Sin embargo no fue hasta 1992 cuando se acuñó el término de Realidad Aumentada por Tom Caudell y David Mizell, dos ingenieros de *Boeing* que proponían el uso de esta novedosa tecnología para mejorar la eficiencia de las tareas realizadas por operarios humanos asociadas a la fabricación de aviones.

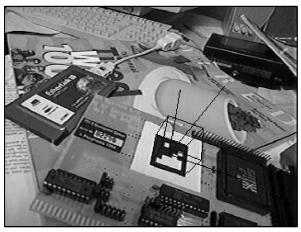


Figura 1.4: Marcas matriciales de Rekimoto.

En 1997, investigadores de la Universidad de Columbia presentan *The Touring Machine* el primer sistema de realidad aumentada móvil (MARS). Utilizan un sistema de visión de tipo *see-through* que combina directamente la imagen real con gráficos 2D y 3D proyectados en una pantalla transparente. [7]

En 1998, el ingeniero de Sony Jun Rekimoto [22] crea un método para calcular completamente el tracking visual de la cámara (con 6 grados de libertad) empleando marcas 2D matriciales (códigos de barras cuadrados, ver Figura 1.4). Esta técnica sería la precursora de otros métodos de tracking visuales en los próximos años. Un año más tarde en 1999, Kato y Billinghurst presentan ARToolKit [10], una librería de tracking visual de 6 grados de libertad que reconoce marcas cuadradas mediante patrones de reconocimiento. Debido a su liberación bajo licencia GPL se hace muy popular y es ampliamente utilizada en el ámbito de la Realidad Aumentada.

Figura 1.5: AR-Quake.

En 2000, un grupo de investigadores de la *University of South Australia* [4] presentan una extensión de Quake (AR-Quake, Figura 1.5) que permite jugar en primera persona en escenarios reales. El registro se realizaba empleando una brújula digital, un receptor de GPS y métodos de visión basados en marcas. Los jugadores debían llevar un sistema de cómputo portátil en una mochila, un casco de visión estereoscópica y un mando de dos botones.

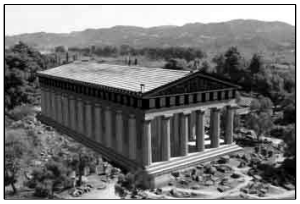


Figura 1.6: Archeoguide.

En 2001 se presenta *Archeoguide* [28] un sistema financiado por la Unión Europea para la creación de guías turísticas electrónicas basadas en Realidad Aumentada (ver Figura 1.6). El sistema proporciona información personalizada basada en el contexto, y muestra reconstrucciones de edificios y objetos mediante una base de datos multimedia adaptada al problema. La comunicación se realiza mediante Wifi, y el sistema es altamente escalable permitiendo diferentes dispositivos de visualización (portátiles, PDAs, etc).

En el 2003, Siemens lanza al mercado *Mozzies*, el primer juego de Realidad Aumentada para teléfonos móviles. El juego superpone mosquitos a la visión obtenida del mundo mediante una cámara integrada en el teléfono. Este juego fue premiado como el mejor videojuego para teléfonos móviles en dicho año.

Figura 1.10: Zapatillas Adidas Originals: Augmented Reality.

En 2004 investigadores de la Universidad Nacional de Singapur presentan *Human Pacman* [3], un juego que emplea GPS y sistemas inerciales para registrar la posición de los jugadores. El PacMan y los fantasmas son en realidad jugadores humanos que corren por la ciudad portando ordenadores y sistemas de visión, percibiendo el mundo como se muestra en la Figura Figura 1.7.

También en el 2004, la Universidad Técnica de Viena presenta el proyecto *Invisible Train* (ver Figura 1.8, el primer juego multi-usuario para PDAs. Esta aplicación se ejecutaba totalmente en las PDAs, sin necesidad de servidores adicionales para realizar procesamiento auxiliar. Los jugadores controlan trenes virtuales y deben intentar evitar que colisione con los trenes de otros jugadores. El proyecto utiliza la biblioteca *Studierstube* desarrollada en la misma universidad.

En 2005 A. Henrysson adapta la biblioteca ARToolKit para poder funcionar en Symbian, y crea un juego de Tenis (ver Figura 1.9) que gana un premio internacional el mismo año.

En 2007, Klein y Murray presentan en ISMAR (uno de los principales congresos de Realidad Aumentada) el algoritmo PTAM [11]; una adaptación del SLAM que separa el *tracking* y el *mapping* en dos hilos independientes, consiguiendo en tiempo real unos resultados muy robustos. Con este trabajo consiguieron el premio al mejor artículo del congreso.

En 2008, Mobilizy crea *Wikitude* (ver Figura 1.11) una aplicación que aumenta la información del mundo real con datos obtenidos de entradas de Wikipedia. Originalmente sólo estaba disponible para te-

Figura 1.7: H-Pacman.

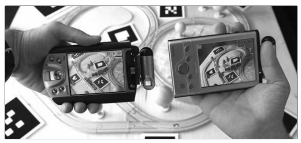


Figura 1.8: Interfaz de Invisible Train, de la Universidad de Viena.

Figura 1.9: AR-Tennis.

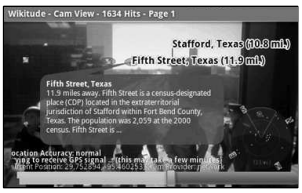


Figura 1.11: Wikitude.

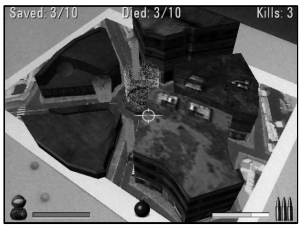


Figura 1.12: ARhrrrr!

Fotografía de ElPais.com 11-10-2009

Figura 1.13: Invizimals.

léfonos Android, aunque actualmente puede descargarse para otras plataformas (como iPhone entre otras).

En 2009, SPRXmobile lanza al mercado una variante de *Wikitude* llamada *Layar*, que utiliza el mismo mecanismo de registro que Wikitude (GPS + Brújula electrónica). Layar define un sistema de capas que permite representar datos de diversas fuentes globales (como Wikipedia o Twitter) además de servicios locales (como tiendas, estaciones de transporte público o guías turísticas). En la actualidad Layar cuenta con más de 100 capas distintas de contenido.

El mismo año se presenta *ARhrrrr!* (ver Figura 1.12), el primer juego con contenido de alta calidad para smartphones con cámara. El teléfono utiliza la metáfora de ventana virtual para mostrar un mapa 3D donde disparar a Zombies y facilitar la salida a los humanos que están atrapados en él. El videojuego emplea de forma intensiva la GPU del teléfono delegando en la tarjeta todos los cálculos salvo el tracking basado en características naturales, que se realiza en la CPU.

El videojuego de PSP *Invizimals* (ver Figura 1.13), creado por el estudio español Novorama en 2009, alcanza una distribución en Europa en el primer trimestre de 2010 superior a las 350.000 copias, y más de 8 millones de copias a nivel mundial, situándose en lo más alto del ránking de ventas. Este juego emplea marcas para registrar la posición de la cámara empleando tracking visual.

A principios de 2010 Adidas lanza al mercado un juego de 5 zapatillas originales de Realidad Aumentada (ver Figura 1.10). Los propietarios de estos modelos podrán enseñar la lengüeta de la zapatilla a una cámara y aparecerá un mundo 3D de la marca, con posibilidad de jugar a contenido exclusivo y obtener premios especiales (a modo de objetos 3D).

1.2. Características Generales

Según Azuma [1], un sistema de Realidad Aumentada debe cumplir las siguientes características:

1. **Combina mundo real y virtual**. El sistema incorpora información sintética a las imágenes percibidas del mundo real.
2. **Interactivo en tiempo real**. Así, los efectos especiales de películas que integran perfectamente imágenes 3D fotorrealistas con imagen real no se considera Realidad Aumentada porque no son calculadas de forma interactiva.
3. **Alineación 3D**. La información del mundo virtual debe ser tridimensional y debe estar correctamente alineada con la imagen del mundo real. Así, estrictamente hablando las aplicaciones que superponen capas gráficas 2D sobre la imagen del mundo real no son consideradas de Realidad Aumentada.

Siendo estrictos según la definición de Azuma, hay algunas aplicaciones vistas en la sección 1.1.1 que no serían consideradas de Realidad Aumentada, ya que el registro del mundo real no se realiza en 3D (como en el caso de *Wikitude* o *Layar* por ejemplo).

La combinación de estas tres características hacen que la Realidad Aumentada sea muy interesante para el usuario ya que complementa y mejora su visión e interacción del mundo real con información que puede resultarle extremadamente útil a la hora de realizar ciertas tareas. De hecho la Realidad Aumentada es considerada como una forma de *Amplificación de la Inteligencia* que emplea el computador para facilitar el trabajo al usuario.

Figura 1.14: W. Ross Ashby (1903-1972), el padre de la Cibernética moderna.

Amplificación Inteligencia

El término **Amplificación de la Inteligencia** comenzó a utilizarse desde la *Introducción a la Cibernética* de William Ross Ashby, refiriéndose al uso de las tecnologías de la información para aumentar la inteligencia humana. También se denomina habitualmente *Aumentación Cognitiva*.

1.3. Aplicaciones

La importancia de la Realidad Aumentada queda patente con el enorme interés que ha generado en los últimos meses. La prestigiosa publicación británicac *The Economist* aseguró en Septiembre de 2009 que *"intentar imaginar como se utilizará la Realidad Aumentada es como intentar predecir el futuro de la tecnología web en 1994"*. Según la consultora *Juniper Research*, la Realidad Aumentada en dispositivos móviles generará más de 700 millones de dólares en el 2014, con más de 350 millones de terminales móviles con capacidad de ejecutar este tipo de aplicaciones.

Un indicador que puede ser significativo es la tendencia de búsqueda en la web. Desde Junio de 2009 la búsqueda por "*Augmented Reality*" supera el número de búsquedas realizadas con la cadena "*Virtual Reality*".

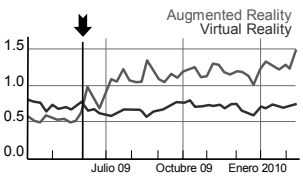


Figura 1.15: Porcentaje de búsquedas en Google por las cadenas *Augmented Reality* y *Virtual Reality*.

Una de las principales causas de este crecimiento en el uso de la Realidad Aumentada es debido a que mediante esta tecnología se amplían los espacios de interacción fuera del propio ordenador. Todo el mundo puede ser un interfaz empleando Realidad Aumentada sobre dispositivos móviles.

Existen multitud de ámbitos de aplicación de la realidad aumentada. En su revisión del estado del arte de 1997 [1], Azuma ya identificaba diversas áreas de aplicación de la Realidad Aumentada. Sólo por mencionar algunas, aunque prácticamente en cualquier área de trabajo puede tener sentido desarrollar aplicaciones de Realidad Aumentada.

- **Medicina**: La medicina puede beneficiarse del uso de la Realidad Aumentada en quirófanos y entrenamiento de doctores. Actualmente es posible obtener datos 3D en tiempo real mediante resonancias magnéticas o tomografías que pueden superponerse en la imagen real del paciente, dando una visión de *rayos X* al especialista.

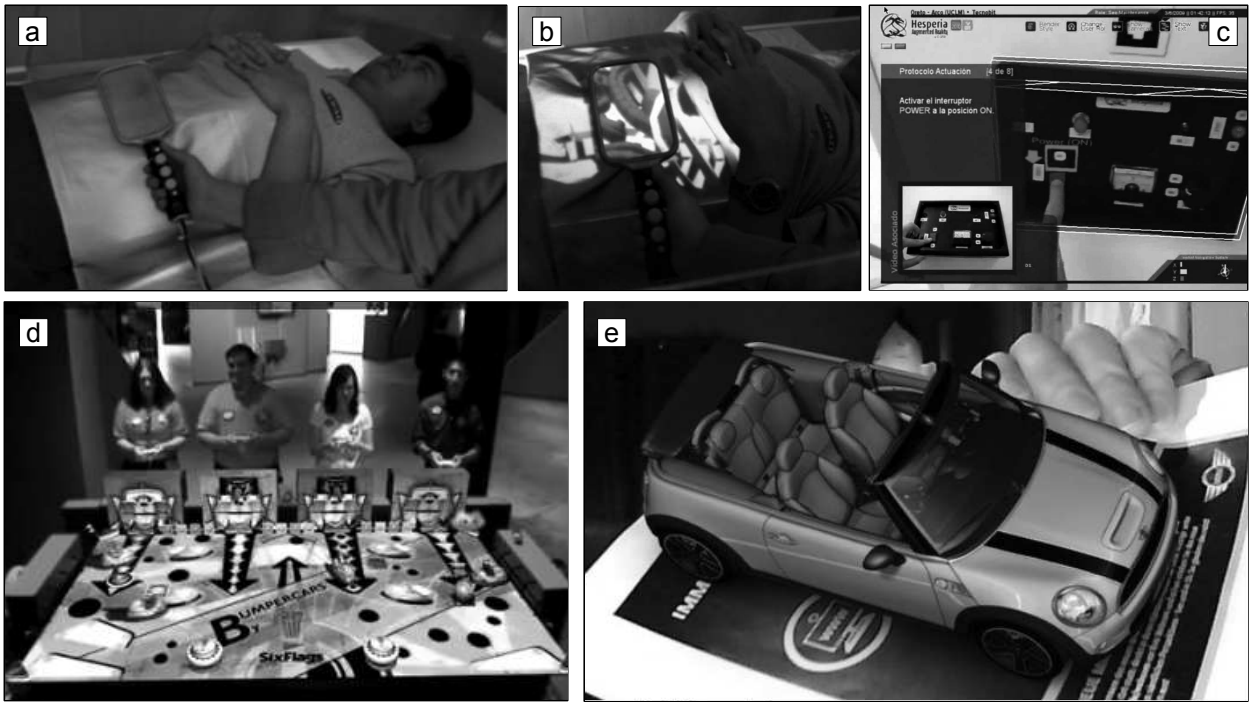


Figura 1.16: Ejemplo de aplicaciones de Realidad Aumentada. a) y b) Ejemplos de visualización médica con Realidad Aumentada del *3D Visualization and Imaging System Lab* de Arizona. c) Ejemplo de mantenimiento de un dispositivo con instrucciones 3D superpuestas del *Grupo de Investigación Oreto* de la Universidad de Castilla-La Mancha. d) Juego desarrollado por Total Immersion (Six Flags). e) Anuncio de Mini.

- **Fabricación**: Otro uso de la Realidad Aumentada es en el ámbito de la fabricación, mantenimiento y reparación de maquinaria compleja. Los pasos a seguir en la reparación son mucho más intuitivos y fáciles de seguir si aparecen directamente superpuestos sobre la imagen real.
- **Entretenimiento**: La industria del ocio ya ha comenzado a sacar partido del enorme potencial de interacción que ofrece la Realidad Aumentada (como Sony, Ubisoft o SixFlags).
- **Publicidad**: Cada vez más empresas utilizan la Realidad Aumentada como reclamo publicitario. Desde 2008, empresas como Adidas, Ford, BMW, Lego, FOX, Paramount, Doritos, Ray Ban y MacDonalds forman parte de una largísima lista de entidades que han utilizado estas técnicas, definiendo una curva creciente exponencialmente.

1.4. Alternativas tecnológicas

En el ámbito de la Realidad Aumentada existen varios *toolkits* que facilitan la construcción de aplicaciones. Sin embargo, para sacar el máximo partido a la tecnología es necesario dominar ciertos conceptos teóricos y de representación gráfica. La mayoría de sistemas están destinados a programadores con experiencia en desarrollo gráfico. A continuación enumeraremos algunas de las bibliotecas más famosas.

- **ARToolKit**: Es probablemente la biblioteca más famosa de Realidad Aumentada. Con interfaz en C y licencia libre permite desarrollar fácilmente aplicaciones de Realidad Aumentada. Se basa en marcadores cuadrados de color negro.
- **ARTag**: Es otra biblioteca con interfaz en C. Está inspirado en ARToolKit. El proyecto murió en el 2008, aunque es posible todavía conseguir el código fuente. El sistema de detección de marcas es mucho más robusto que el de ARToolKit.
- **OSGART**: Biblioteca en C++ que permite utilizar varias librerías de tracking (como ARToolKit, SSTT o Bazar).
- **FLARToolKit**: Implementación para Web (basada en Flash y ActionScript) del ARToolKit portado a Java *NyARToolKit*.
- **Otros ports de ARToolKit**: Existen multitud de versiones de ARToolKit portados en diferentes plataformas, como AndAR (para teléfonos Android), SLARToolkit, etc...

En este documento nos basaremos en ARToolKit por ser actualmente el principal referente en tracking basado en marcas, y por existir multitud de versiones portadas a diferentes plataformas móviles.

Capítulo 2

Métodos de Registro

El cálculo del registro requiere posicionar la cámara (posición y rotación) relativamente a los objetos de la escena. Existen varias tecnologías para realizar el registro, como sensores mecánicos, magnéticos, ultrasónicos, inerciales y basados en visión. En este capítulo estudiaremos diferentes métodos de tracking con sus ventajas y problemas asociados. Los métodos basados en visión, por ser los extendidos, baratos y fáciles de desplegar serán a los que dedicaremos la mayor parte del capítulo.

Los métodos de tracking tratan de obtener una estimación de la trayectoria en el espacio realizada por un objeto o sensor. Se pueden emplear diferentes tipos de sensores como basados en campos magnéticos, ondas sonoras o cámaras de visión. Las cámaras están actualmente integradas en multitud de dispositivos portátiles y permiten realizar tracking a un coste reduccido.

El tracking basado en cámaras de vídeo es un subcampo del tracking 3D en el que se emplean técnicas de visión por computador para obtener el posicionamiento de seis grados de libertad de la cámara (tres grados de la posición y tres de la orientación).

Para calcular la posición de la cámara con respecto al mundo real se necesitan un conjunto de referencias tridimensionales. Algunos ejemplos de estas referencias pueden ser marcas con una descripción geométrica previamente conocida u objetos previamente modelados. Realizando una comparación con lo que percibe la cámara en el mundo real es posible obtener el posicionamiento relativo a estas referencias.

2.1. Problemática

El tracking basado en visión es ampliamente utilizado debido a la gran cantidad de información que puede obtenerse de flujos de vídeo, y al bajo coste de este tipo de dispositivos. Sin embargo, las aplicaciones que emplean estas técnicas de tracking deben enfrentarse a tres clases de problemas:

1. **Problemas de identificación**. Debidos al propio uso de cámaras de vídeo, tales como oclusiones, problemas debidos al punto de vista o a las condiciones de iluminación.
2. **Problemas de seguimiento**. En este grupo de inconvenientes asociados al uso de flujos de vídeo se incluye todo lo referente a la inicialización automática o la recuperación del tracking cuando se pierden las referencias.
3. **Problemas en entornos no controlados**. El trabajo en esta línea se centra en realizar el tracking sin referencias previamente conocidas.

2.2. Métodos de Tracking

Según Marimón [15], en una taxonomía general de métodos de tracking distinguimos las aproximaciones *Bottom-Up* que tratan de obtener la posición a partir de lo que percibe la cámara, mientras que las aproximaciones *Top-Down* tratan de estimar si desde la posición actual se está percibiendo lo que se esperaba (primero se estima la posición y luego se trata de identificar las referencias que se esperaban obtener).

Ambas aproximaciones tienen sus ventajas y desventajas asociadas. Por ejemplo, en las aproximaciones *Bottom-Up* la inicialización y recuperación cuando se pierde el tracking es automática. Sin embargo, el principal problema se encuentra cuando no se localiza una referencia completa, ya que el tracker no es capaz de obtener una posición válida.

En este caso, las aproximaciones *Top-Down* funcionan mejor, incluso con referencias parciales (en el caso de oclusión). Incluso cuando en algún momento no se obtiene ninguna referencia válida, el tracker proporciona una estimación de la posición. Sin embargo, las aproximaciones *Top-Down* requieren una posición inicial (no permiten inicialización automática), y acumulan errores parciales de las estimaciones.

En la última década ha aumentado el interés en métodos de tracking basados en visión debido al aumento de la precisión y capacidad de cómputo de los dispositivos portátiles dotados de cámaras.

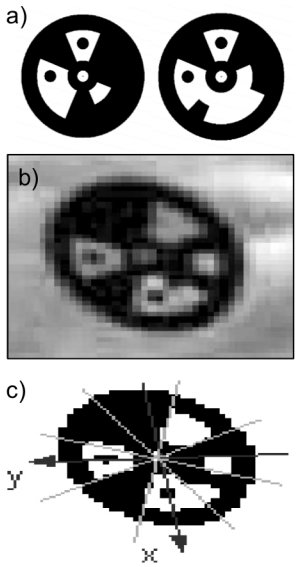


Figura 2.1: Ejemplo de marcas circulares usadas en [18] **a)** Dos marcas circulares **b)** Marca a detectar (baja resolución) **c)** Marca binarizada con ejes y sectores marcados para ser leida por el sistema de código de barras circular.

2.2.1. Aproximaciones Bottom-Up

Para las aproximaciones *Bottom-Up* los seis grados de libertad se calculan a partir de la obtención de características geométricas conocidas de objetos y sus relaciones geométricas 3D (por ejemplo, un cuadrado, una circunferencia, etc...). Dependiendo del tipo de características utilizadas, distinguimos tracking basado en marcas (empleo de patrones específicos) o tracking sin marcas que detecta características naturales de la escena.

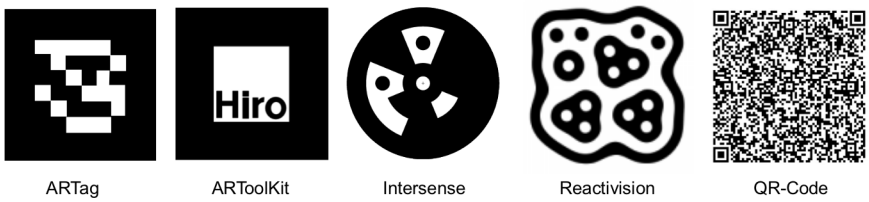


Figura 2.2: Algunas marcas empleadas en diferentes sistemas de tracking.

Tracking basado en marcas: Es el método más utilizado actualmente. Se emplean marcas cuadradas o circulares que pueden detectarse fácilmente gracias a su alto contraste. El framework más ampliamente utilizado es ARToolkit, aunque existen multitud de entornos de tracking basados en marcas (ver Figura 2.2). Otras aproximaciones similares como la de Liu et al. [13] emplean marcas cuadras con diferentes colores. Naimark [18] utiliza marcas circulares que codifican su identificador mediante un sistema de código de barras. En este trabajo se realiza una comparación con las marcas cuadradas, mostrando su mayor robustez cuando aumenta el número de marcas empleadas. Algunos trabajos recientes se centran en evitar el principal problema debido a la oclusión, como Fiala [8] que utiliza imágenes en escala de gris para segmentar líneas y completar marcas parcialmente ocultas.

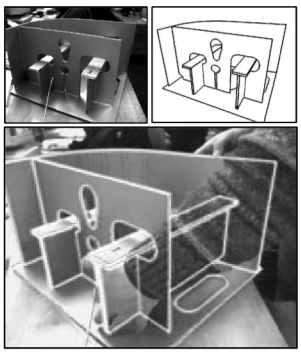
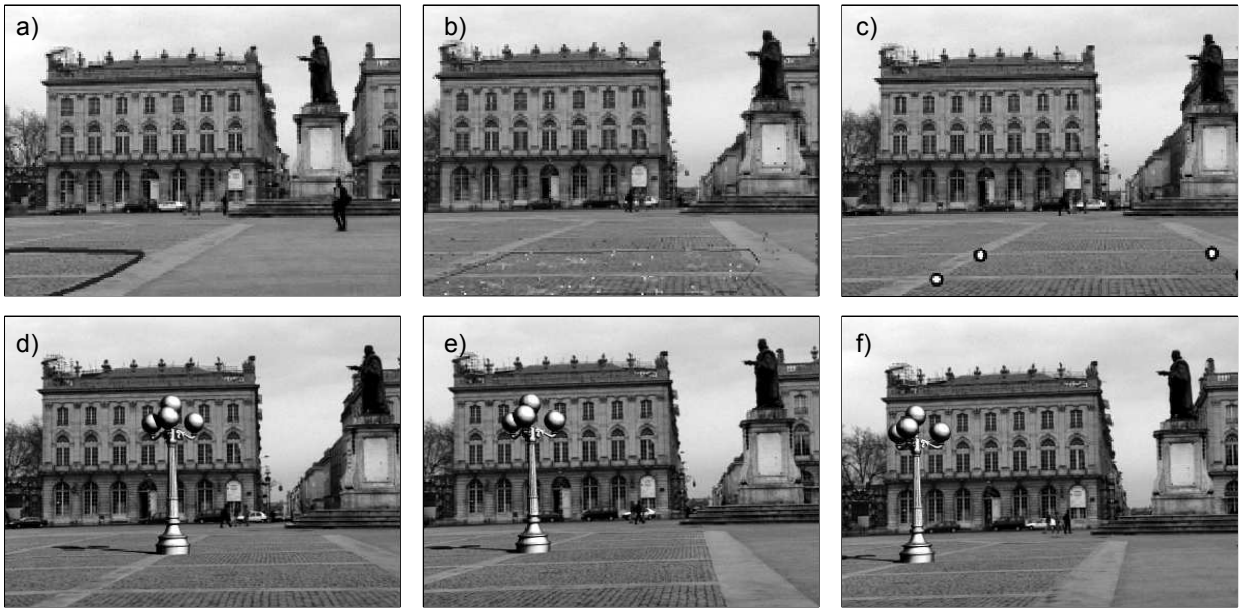


Figura 2.3: Ejemplo de tracking basado en Modelos (imágenes de Drummond y Cipolla [6]). **Arriba Izquierda**: Fotograma del vídeo **Derecha**: Modelo CAD **Abajo**: Ejemplo de tracking con oclusión del modelo físico.

Tracking sin marcas: Esta familia de métodos emplean únicamente características naturales de la escena (estructuras físicas que son altamente detectables desde el punto de vista de los métodos de visión por computador; como por ejemplo los bordes de una mesa). A pesar de no requerir entornos totalmente preparados, es necesario definir algunas restricciones que deben cumplirse o contar con modelos geométricos esperados. A continuación describiremos brevemente estas técnicas.

- **Estructuras Planas**: Este grupo de técnicas emplean información sobre áreas planas detectadas en la escena (por ejemplo, información sobre la región de la fachada de un edificio, carteles en paredes, etc...). El principal inconveniente de estas técnicas es su altos requisitos computacionales. Algunso ejemplos relevantes son el trabajo de Simon et al [24] que emplea dos o más planos del entorno para el tracking sin marcas (ver Figura 2.4) y



a) Selección de una región del plano sobre el que se realizará el tracking. **b)** Detección automática de características del Entorno. **c)** Establecimiento del sistema de coordenadas (4 puntos en un plano, en la imagen sólo aparecen 3 de ellos). **d,e,f)** Ejemplo de objeto virtual insertado en un vídeo empleando esta técnica.

Figura 2.4: Tracking basado en estructuras planas (ejemplo de [24]).

el trabajo de Jurie [9] que emplea una aproximación basada en Optical Flow.

- **Basadas en Modelos**: El tracking de la escena se basa en utilizar un modelo CAD de partes de la escena (habitualmente un objeto) y detectar aristas y puntos de interés en el vídeo de entrada. Se establece la posición de la cámara relativa a ese objeto. El trabajo de Drummond et al [6] establecen una función de peso para cada grupo de aristas detectados en el espacio de búsqueda. El trabajo de Vacchetti [26] utiliza varias hipótesis de detección asociando a cada una de ellas una valoración.

- **Escenas sin Restricciones**: En algunos escenarios de aplicación no es posible contar con conocimiento a priori sobre estructuras planas o modelos. En general este tipo de técnicas emplean restricciones epipolares de las cámaras en movimiento. El gran problema de estas técnicas es el alto coste computacional y sólo recientemente se están empezando a utilizar en tiempo real [19] [12].

2.2.2. Aproximaciones Top-Down

Como se ha comentado anteriormente, las aproximaciones *Top-Down* se basan en el uso del contexto y estiman la geometría de la escena de este contexto. Se utilizan modelos del movimiento basados en filtros bayesianos para predecir la posición de la cámara. A partir de esa posición de la cámara se buscan referencias en la escena que corrijan la predicción y ayuden a la creación de un modelo del entorno. En este ciclo de predicción y corrección se necesitan gestionar por un lado el filtrado de datos (uso del modelo de movimiento y sus limitaciones asociadas), y por otro lado la asociación de datos (localización de las referencias según la posición predicha). Todas las aproximaciones Top-Down tienen que trabajar con filtros y modelos de asociación de datos.

Figura 2.5: El uso de un filtro de partículas permite combinar diferentes métodos de tracking y continuar registrando correctamente los objetos incluso con oclusión manual de la marca (arriba) y cuando el patrón se escapa del campo de visión (abajo). Imágenes extraídas de [14]

Los filtros Bayesianos pueden dividirse en dos grandes familias; aquellas que trabajan con modelos de movimiento gausianos, que explotan los beneficios de los Filtros de Kalman y sus variantes (como por ejemplo [17] y [2]) y aquellos que, por las características del ruido no pueden ser modelados mediante modelos gausianos y se ajustan mejor al uso de Filtros de Partículas (como por ejemplo [21] y [20]).

Los algoritmos de asociación de datos tienen que trabajar con diferentes medidas candidatas para emparejar cada caraccterísticas detectadas. La aproximación de Cox [5] utiliza una aproximación mejorada del filtro MHT para el seguimiento de puntos de interés. Veenman [27] se basan en la idea de asignar las características empleando métodos clásicos.

Existen multitud de alternativas de tracking óptico (basado en visión por computador). Dependiendo de las características del dispositivo (capacidad de cómputo, resolución de la(s) cámara(s) asociada(s), precisión de las lentes) y del contexto de aplicación pueden emplearse una o varias de las familias de técnicas estudiadas empleando métodos de fusión de percepciones para obtener un posicionamiento preciso y coherente en el tiempo.

Capítulo 3

Introducción a ARToolKit

En este capítulo se introducirá el ciclo de desarrollo de aplicaciones empleando ARToolKit. Se estudiarán los pasos a seguir para calcular el registro de la cámara, y los parámetros de las principales llamadas a la biblioteca.

Como se ha estudiado en el capítulo 2, los métodos de tracking ópticos se utilizan ampliamente debido a su precisión y bajo coste de elementos hardware. A continuación se describirá una de las bibliotecas de tracking óptico más extendidas en el ámbito de la realidad aumentada.

Figura 3.1: Uso de ARToolKit en un dispositivo de visión estereo.

3.1. Qué es ARToolKit

ARToolKit es una biblioteca de funciones para el desarrollo rápido de aplicaciones de Realidad Aumentada. Fue escrita originalmente en C por H. Kato, y mantenida por el HIT Lab de la Universidad de Washington, y el HIT Lab NZ de la Universidad de Canterbury (Nueva Zelanda).

ARToolKit facilita el problema del registro de la cámara empleando métodos de visión por computador, de forma que obtiene el posicionamiento relativo de 6 grados de libertad haciendo el seguimiento de marcadadores cuadrados en tiempo real, incluso en dispositivos de baja capacidad de cómputo. Algunas de las características más destacables son:

- **Tracking de una cámara**. ARToolKit en su versión básica soporta de forma nativa el tracking de una cámara, aunque puede utilizarse para tracking multicámara (si el programador se hace cargo de calcular el histórico de percepciones). La biblioteca soporta gran variedad de modelos de cámaras y modelos de color.
- **Marcas negras cuadradas**. Emplea métodos de tracking de superficies planas de 6 grados de libertad. Estas marcas pueden ser personalizadas, siempre que el patrón no sea simétrico en alguno de sus ejes.
- **Rápido y Multiplataforma**. Funciona en gran variedad de sistemas operativos (Linux, Mac, Windows, IRIX, SGI...), y ha sido portado a multitud de dispositivos portátiles y smartphones (Andorid, iPhone, PDAs...).
- **Comunidad Activa**. A través de los foros[1] y listas de correo se pueden resolver problemas particulares de uso de la biblioteca.
- **Licencia libre**. Esto permite utilizar, modificar y distribuir programas realizados con ARToolKit bajo la licencia GPL v2.

3.2. Instalación y configuración

En esencia ARToolKit es una biblioteca de funciones para el desarrollo de aplicaciones de Realidad Aumentada, que utiliza a su vez otras bibliotecas. Por tanto, primero deberemos satisfacer esas dependencias. Para ello, en Debian instalaremos los paquetes necesarios para su compilación:

```
# apt-get install freeglut3-dev libgl1-mesa-dev libglu1-mesa-dev
libxi-dev libxmu-dev libjpeg-dev
```

A continuación ejecutamos `./Configure` para obtener un *Makefile* adaptado a nuestro sistema. Elegimos Video4Linux2 en el driver de captura de video porque disponemos de una cámara integrada de este tipo (la versión de ARToolKit con la que vamos a trabajar está parcheada para soportar este tipo de dispositivos), en la segunda pregunta no utilizaremos las instrucciones de ensamblador en ccvt (por disponer de una arquitectura ia64), habilitaremos los símbolos de depuración, y activaremos el soporte hardware para `GL_NV_texture_rectangle`, ya que la tarjeta gráfica del equipo los soporta:

```
carlos@kurt:ARToolKit$ ./Configure
Select a video capture driver.
  1: Video4Linux
  2: Video4Linux+JPEG Decompression (EyeToy)
  3: Video4Linux2
  4: Digital Video Camcoder through IEEE 1394 (DV Format)
  5: Digital Video Camera through IEEE 1394 (VGA NC Image Format)
```

Figura 3.2: Aunque las bibliotecas con las que trabajaremos en este documento son libres, sólo describiremos el proceso de instalación y configuración bajo GNU/Linux. *Imagen original de sfgate.com.*

[1] Foros de ARToolKit: `http://www.hitlabnz.org/wiki/Forum`

```
  6: GStreamer Media Framework
Enter : 3

Color conversion should use x86 assembly (not working for 64bit)?
Enter : n
Do you want to create debug symbols? (y or n)
Enter : y
Build gsub libraries with texture rectangle support? (y or n)
GL_NV_texture_rectangle is supported on most NVidia graphics cards
and on ATi Radeon and better graphics cards
Enter : y
  create ./Makefile
  create lib/SRC/Makefile
  ...
  create include/AR/config.h
Done.
```

Finalmente compilamos las librerías desde el mismo directorio, ejecutando `make`:

```
$ make
```

Si todo ha ido bien, ya tenemos compiladas las bibliotecas de ARToolKit. Estas bibliotecas no requieren estar instaladas en ningún directorio especial del sistema, ya que se compilan como bibliotecas estáticas, de forma que están incluidas en cada ejecutable que se construye. Los ficheros `makefile` que utilizaremos en los ejemplos tendrán definido un camino (relativo o absoluto) hasta la localización en disco de estas bibliotecas. A continuación veremos un ejemplo de funcionamiento básico.

3.3. El esperado “Hola Mundo!”

Figura 3.3: La impresionante salida en pantalla del *“Hola Mundo”* de Realidad Aumentada.

Aunque todavía quedan muchos aspectos que estudiar, comenzaremos con una aplicación mínima que dibuja una tetera 3D localizada en el centro de una marca de ARToolKit. El listado siguiente, que muestra el código completo (¡menos de 120 líneas!) del *Hola Mundo!*. El listado de código de la sección 3.3.4 muestra el fichero `makefile` necesario para compilar este ejemplo.

Listado 3.1: “Hola Mundo!” con ARToolKit

```
 1 #include <GL/glut.h>
 2 #include <AR/gsub.h>
 3 #include <AR/video.h>
 4 #include <AR/param.h>
 5 #include <AR/ar.h>
 6
 7 // ==== Definicion de constantes y variables globales =====
 8 int    patt_id;            // Identificador unico de la marca
 9 double patt_trans[3][4];   // Matriz de transformacion de la marca
10
11 void print_error (char *error) {  printf(error); exit(0); }
12 // ======== cleanup ========================================
13 static void cleanup(void) {
```

```
14   arVideoCapStop();            // Libera recursos al salir ...
15   arVideoClose();
16   argCleanup();
17 }
18 /* La funcion draw se encarga de obtener la matriz de
       transformacion de OpenGL y dibujar la tetera 3D posicionada en
       el centro de la marca.*/
19 // ======== draw ==========================================
20 static void draw( void ) {
21   double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL
22   GLfloat mat_ambient[]     = {0.0, 0.0, 1.0, 1.0};
23   GLfloat light_position[]  = {100.0,-200.0,200.0,0.0};
24
25   argDrawMode3D();              // Cambiamos el contexto a 3D
26   argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
27   glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
28   glEnable(GL_DEPTH_TEST);
29   glDepthFunc(GL_LEQUAL);
30
31   argConvGlpara(patt_trans, gl_para);   // Convertimos la matriz de
32   glMatrixMode(GL_MODELVIEW);           // la marca para ser usada
33   glLoadMatrixd(gl_para);               // por OpenGL
34   // Esta ultima parte del codigo es para dibujar el objeto 3D
35   glEnable(GL_LIGHTING);  glEnable(GL_LIGHT0);
36   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
37   glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
38     glTranslatef(0.0, 0.0, 60.0);
39     glRotatef(90.0, 1.0, 0.0, 0.0);
40     glutSolidTeapot(80.0);
41   glDiable(GL_DEPTH_TEST);
42 }
43 /* La funcion init inicializa la camara (cargando el fichero con
       sus parametros intrinsecos) y el patron que se reconocera en el
        ejemplo. */
44 // ======== init ==========================================
45 static void init( void ) {
46   ARParam  wparam, cparam;   // Parametros intrinsecos de la camara
47   int xsize, ysize;          // Tamano del video de camara (pixels)
48
49   // Abrimos dispositivo de video
50   if(arVideoOpen("") < 0) exit(0);
51   if(arVideoInqSize(&xsize, &ysize) < 0) exit(0);
52
53   // Cargamos los parametros intrinsecos de la camara
54   if(arParamLoad("data/camera_para.dat", 1, &wparam) < 0)
55     print_error ("Error en carga de parametros de camara\n");
56
57   arParamChangeSize(&wparam, xsize, ysize, &cparam);
58   arInitCparam(&cparam);   // Inicializamos la camara con çparam"
59
60   // Cargamos la marca que vamos a reconocer en este ejemplo
61   if((patt_id=arLoadPatt("data/simple.patt")) < 0)
62     print_error ("Error en carga de patron\n");
63
64   argInit(&cparam, 1.0, 0, 0, 0, 0);   // Abrimos la ventana
65 }
66 /* La funcion mainLoop se llama automaticamente (se registra un
       callback en el main). Se encarga de obtener un frame, detectar
       la marca y llamar a la funcion de dibujado del objeto 3D. */
67 // ======== mainLoop ======================================
68 static void mainLoop(void) {
69   ARUint8 *dataPtr;
```

```
70   ARMarkerInfo *marker_info;
71   int marker_num, j, k;
72
73   double p_width     = 120.0;         // Ancho del patron (marca)
74   double p_center[2] = {0.0, 0.0};    // Centro del patron (marca)
75
76   // Capturamos un frame de la camara de video
77   if((dataPtr = (ARUint8 *)arVideoGetImage()) == NULL) {
78     // Si devuelve NULL es porque no hay un nuevo frame listo
79     arUtilSleep(2);  return;  // Dormimos el hilo 2ms y salimos
80   }
81
82   argDrawMode2D();
83   argDispImage(dataPtr, 0,0);    // Dibujamos lo que ve la camara
84
85   // Detectamos la marca en el frame capturado (return -1 si error)
86   if(arDetectMarker(dataPtr, 100, &marker_info, &marker_num) < 0) {
87     cleanup(); exit(0);   // Si devolvio -1, salimos del programa!
88   }
89
90   arVideoCapNext();      // Frame pintado y analizado... A por
         otro!
91
92   // Vemos donde detecta el patron con mayor fiabilidad
93   for(j = 0, k = -1; j < marker_num; j++) {
94     if(patt_id == marker_info[j].id) {
95       if (k == -1) k = j;
96       else if(marker_info[k].cf < marker_info[j].cf) k = j;
97     }
98   }
99
100  if(k != -1) {   // Si ha detectado el patron en algun sitio...
101    // Obtenemos transformacion entre la marca y la camara real
102    arGetTransMat(&marker_info[k], p_center, p_width, patt_trans);
103    draw();       // Dibujamos los objetos de la escena
104  }
105
106  argSwapBuffers(); // Cambiamos el buffer con lo que tenga
         dibujado
107 }
108
109 // ======== Main ========================================
110 int main(int argc, char **argv) {
111   glutInit(&argc, argv);     // Creamos la ventana OpenGL con Glut
112   init();                    // Llamada a nuestra funcion de inicio
113   arVideoCapStart();         // Creamos un hilo para captura de
         video
114   argMainLoop( NULL, NULL, mainLoop );   // Asociamos callbacks...
115   return (0);
116 }
```

El ciclo de desarrollo puede resumirse en tres grandes etapas: 1. **Inicialización**: Consiste en leer los parámetros asociados a la cámara y la descripción de las marcas que se van a utilizar. 2. **Bucle Principal** (*Main Loop*): Es la etapa principal y está formada por un conjunto de subetapas que veremos a continuación. 3. **Finalización**: Libera los recursos requeridos por la aplicación. La etapa del Bucle Principal está formada por 4 subetapas funcionales que se realizan repetidamente hasta que el usuario decide finalizar la aplicación:

1. **Captura**. Se obtiene un *frame* de la cámara de vídeo. En el *"Hola Mundo!"* se realiza llamando a la función `arVideoGetImage` en la línea 77.
2. **Detección**. Se identifican las marcas en el *frame* anterior. En el ejemplo se llama a la función `arDetectMarker` en la línea 86.
3. **Transformación**. Se calcula la posición relativa entre las marcas detectadas y la cámara física. Se realiza llamando a la función `arGetTransMat` en la línea 102.
4. **Dibujado**. Se dibujan los objetos virtuales situando la cámara virtual en la posición relativa anteriormente calculada. En el *Hola Mundo* se ha creado una función propia `draw` que es llamada desde el mainLoop en la línea 103.

Marcas Personalizadas

ARToolKit permite crear fácilmente marcadores propios. En este primer ejemplo utilizaremos una marca previamente entrenada. Veremos en la sección 3.4.3 cómo se almacenan internamente los marcadores y los pasos a seguir para definir nuestros propios marcadores.

En los ejemplos de este documento nos basaremos en el uso de las bibliotecas GLUT (*OpenGL Utility Toolkit*) que nos facilitan el desarrollo de aplicaciones OpenGL proporcionando sencillas funciones de *callback* para el manejo de eventos de teclado y ratón, así como la fácil apertura de ventanas multiplataforma. Estas bibliotecas están pensadas para el aprendizaje de OpenGL, y para la construcción de pequeñas aplicaciones. Para facilitar la escritura de código se utilizarán algunas variables globales (por la imposibilidad de pasar parámetros adicionales a las funciones de callback de GLUT), aunque podrían utilizarse mecanismos para evitarlas. Se recomienda el uso de otras librerías multiplataforma para la construcción de proyectos de mayor tamaño, como SDL.

En un primer estudio, se pueden identificar algunos de los bloques funcionales descritos anteriormente. Por ejemplo, en las líneas 45–65 se encuentra definido el bloque de inicialización (función `init`), que es llamado desde `main` (línea 112). El bucle principal está definido en las líneas 68–107 y la liberación de recursos se realiza con la llamada a funciones propias de ARToolKit en las líneas 13–17.

Uso de **/dev/video***

Si tuviéramos dos cámaras en el sistema, y quisiéramos abrir la segunda (con interfaz V4L) llamaríamos a la función *arVideoOpen* con la cadena *"-dev=/dev/video1"*. Podemos ver una descripción completa de los parámetros soportados por esta función en el directorio de documentación *doc/video/* de ARToolKit.

3.3.1. Inicialización

El bloque de inicialización (función `init`), comienza abriendo el dispositivo de video con `arVideoOpen`. Esta función admite parámetros de configuración en una cadena de caracteres. A continuación, línea 51, se obtiene el tamaño de la fuente de vídeo mediante `arVideoInqSize`. Esta función escribe en las direcciones de memoria reservadas para dos enteros el ancho y alto del vídeo. En las siguientes líneas cargamos los parámetros intrínsecos de la cámara obtenidos en la etapa de calibración. En este ejemplo utilizaremos un fichero genérico *camera_para.dat* válido para la mayoría de webcams. Sin embargo, para

obtener resultados más precisos, lo ideal es trabajar con un fichero adaptado a nuestro modelo de cámara concreto. Veremos cómo crear este fichero específico para cada cámara en la sección 3.4.2.

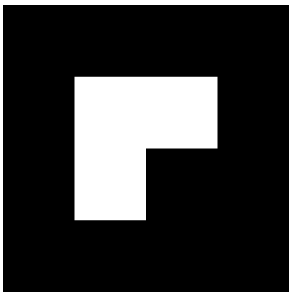


Figura 3.4: Aunque es posible definir marcas personalizadas, las que ofrecen mejores resultados son las basadas en patrones sencillos, como la empleada en este ejemplo.

Así, con la llamada a `arParamLoad` se rellena la estructura `ARParam` especificada como tercer parámetro con las características de la cámara. Estas características son independientes de la resolución a la que esté trabajando la cámara, por lo que tenemos que instanciar los parámetros concretamente a la resolución en píxeles. Para ello, se utiliza la llamada a `arParamChangeSize` (línea 57) especificando la resolución concreta con la que trabajará la cámara en este ejemplo. Una vez obtenidos los parámetros de la cámara instanciados al caso concreto de este ejemplo, se cargan en las estructuras de datos internas de ARToolKit, mediante la llamada a `arInitCparam`.

En la última parte cargamos el patrón asociado a la marca (línea 61). Estudiaremos en la sección 3.4.3 la estructura de esos ficheros. Finalmente abrimos la ventana de OpenGL (mediante la librería auxiliar Gsub de ARToolKit) `argInit` en la línea 64, pasándole como primer parámetro la configuración de la cámara. El segundo parámetro indica el factor de *zoom* (en este caso, sin zoom).

3.3.2. Bucle Principal

El primer paso a realizar en el bucle principal es recuperar un *frame* de la cámara de vídeo. Mediante la función `arVideoGetImage` obtenemos una imagen (la llamada devuelve un puntero a un buffer donde se encuentra la imagen capturada). La llamada devuelve `NULL` si no hay una nueva imagen (en el caso de que llamemos de nuevo muy pronto; con mayor frecuencia que la soportada por la cámara). Si esto ocurre, simplemente dormimos el hilo 2ms (línea 79)y volvemos a ejecutar el *mainLoop*.

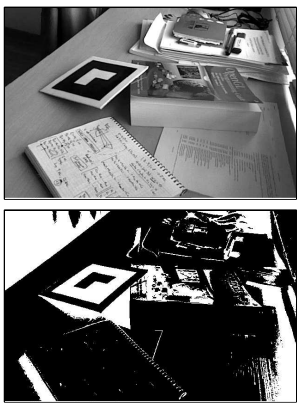


Figura 3.5: Ejemplo de proceso de binarización de la imagen de entrada para la detección de marcas.

A continuación dibujamos en la ventana (en modo 2D, línea 82) el buffer que acabamos de recuperar de la cámara (línea 83).

La llamada a `arDetectMarker` localiza las marcas en el buffer de entrada. En la línea 86 el segundo parámetro de valor 100 se corresponde con el valor umbral de binarización (a blanco y negro, ver Figura 3.5) de la imagen. El propósito de este valor umbral (*threshold*) se explicará en detalle en la sección 3.4.1. Esta función nos devuelve en el tercer parámetro un puntero a una lista de estructuras de tipo `ARMarkerInfo`, que contienen información sobre las marcas detectadas (junto con un grado de fiabilidad de la detección), y como cuarto parámetro el número de marcas detectadas.

De esta forma, ARToolKit nos devuelve *“posibles”* posiciones para cada una de las marcas detectas. Incluso cuando estamos trabajando con una única marca, es común que sea detectada en diferentes posiciones (por ejemplo, si hay algo parecido a un cuadrado negro en la escena). ¿Cómo elegimos la correcta en el caso de que tengamos varias detecciones? ARToolKit asocia a cada percepción una probabilidad de

que lo percibido sea una marca, en el campo `cf` (*confidence value*). En la tabla 3.1 se muestra la descripción completa de los campos de esta estructura. Como se puede comprobar, todos los campos de la estructura ARMarkerInfo se refieren a coordenadas 2D, por lo que aún no se ha calculado la posición relativa de la marca con la cámara.

Tabla 3.1: Campos de la estructura *ARMarkerInfo*

Tipo	Campo	Descripción
`int`	**`area`**	Tamaño en píxeles de la región detectada.
`int`	**`id`**	Identificador (único) de la marca.
`int`	**`dir`**	Dirección. Codifica mediante un valor numérico (0..3) la rotación de la marca detectada. Cada marca puede tener 4 rotaciones distintas.
`double`	**`cf`**	Valor de confianza. Probabilidad de ser una marca (entre 0 y 1).
`double`	**`pos[2]`**	Centro de la marca (en espacio 2D).
`double`	**`line[4][3]`**	Ecuaciones de las 4 aristas de la marca. Las aristas se definen con 3 valores (a,b,c), empleando la ecuación implícita de la recta $ax + by + c = 0$.
`double`	**`vertex[4][2]`**	Posición de los 4 vértices de la marca (en espacio 2D).

Así, en las líneas 93–98 guardamos en la variable `k` el índice de la lista de marcas detectadas aquella percepción que tenga mayor probabilidad de ser la marca (cuyo valor de fiabilidad sea mayor).

Mediante la llamada a `arGetTransMat` (línea 102) obtenemos en una matriz la transformación relativa entre la marca y la cámara (matriz 3x4 de doubles); es decir, obtiene la posición y rotación de la cámara con respecto de la marca detectada. Para ello es necesario especificar el centro de la marca, y el ancho. Esta matriz será finalmente convertida al formato de matriz homogenea de 16 componentes utilizada por OpenGL mediante la llamada a `argConvGlpara` en la línea 31. Veremos en el capítulo 5 una introducción a los aspectos básicos de OpenGL para su uso en aplicaciones de Realidad Aumentada.

3.3.3. Finalización y función Main

En la función `cleanup` se liberan los recursos al salir de la aplicación. Se hace uso de funciones de ARToolKit para detener la cámara de vídeo, y limpiar las estructuras de datos internas de ARToolKit.

En la función `main` se registran los *callbacks* en la línea 114 mediante la función `argMainLoop`. En este ejemplo, se pasa como primer y segundo parámetro `NULL` (correspondientes a los manejadores de ratón y teclado respectivamente). Veremos en posteriores ejemplos cómo

utilizarlo. Por su parte, se asocia la función que se estará llamando constantemente en el bucle principal. En el caso de este ejemplo se llama `mainLoop`.

3.3.4. Compilación con Make

Sobre el **enlazado**...

La fase de enlazado en ARToolKit se realiza de forma estática, por lo que no es necesario distribuir ninguna librería adicional con el programa compilado para su ejecución.

En lo referente al `makefile` del ejemplo (ver listado siguiente), se deberá cambiar el valor de la variable `ARTOOLKITDIR` para que apunte al directorio donde se encuentra ARToolKit. Esta ruta puede ser absoluta (como en el ejemplo) o relativa al *path* donde se encuentre el programa.

Listado 3.2: El makefile basico del "Hola Mundo!"

```
1  # Ruta absoluta o relativa (sin espacios al final del path!)
2  ARTOOLKITDIR = ../../ARToolKit
3  INC_DIR      = $(ARTOOLKITDIR)/include
4  LIB_DIR      = $(ARTOOLKITDIR)/lib
5
6  LIBS   = -lARgsub -lARvideo -lAR -lglut
7
8  NAMEEXEC     = helloWorld
9
10 all: $(NAMEEXEC)
11
12 $(NAMEEXEC): $(NAMEEXEC).c
13     cc -I $(INC_DIR) -o $(NAMEEXEC) $(NAMEEXEC).c -L$(LIB_DIR) $(
           LIBS)
14 clean:
15     rm -f *.o $(NAMEEXEC) *~ *.*~
```

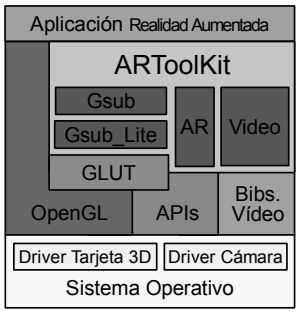


Figura 3.6: Arquitectura de ARToolKit.

3.4. Las Entrañas de ARToolKit

En este apartado veremos algunos detalles sobre cómo funciona internamente *ARToolKit*. Completaremos a lo largo del documento algunos detalles más avanzados que no tiene sentido estudiar en este momento.

ARToolKit está formado por tres módulos (ver Figura 3.6):

- **Video**: Este módulo contiene las funciones para obtener frames de vídeo de los dispositivos soportados por el Sistema Operativo. El prototipo de las funciones de este módulo se encuentran el fichero de cabecera `video.h`.
- **AR**: Este módulo principal contiene las funciones principales de *tracking* de marcas, calibración y estructuras de datos requeridas por estos métodos. Los ficheros de cabecera `ar.h`, `arMulti.h` (subrutinas para gestión multi-patrón) y `param.h` describen las funciones asociadas a este módulo.

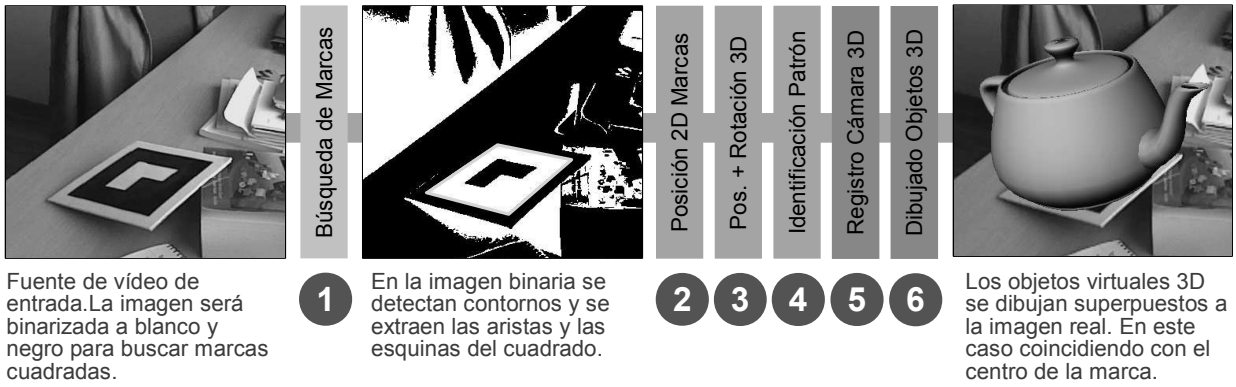


Figura 3.7: Esquema funcional de ARToolKit.

- **Gsub y Gsub_Lite**: Estos módulos contienen las funciones relacionadas con la etapa de representación. Ambas utilizan GLUT, aunque la versión "_Lite" es más eficiente y no tiene dependencias con ningún sistema de ventanas concreto. En estos módulos se describen los ficheros de cabecera `gsub.h`, `gsub_lite.h` y `gsubUtil.h`.

Estos módulos están totalmente desacoplados, de forma que es posible utilizar ARToolKit sin necesidad de emplear los métodos de captura de vídeo del primer módulo, o sin emplear los módulos de representación *Gsub* o *Gsub_Lite*.

3.4.1. Principios Básicos

ARToolKit está basado en un algoritmo de detección de bordes y un método rápido de estimación de la orientación. La figura 3.7 resume el principio básico de funcionamiento de ARToolKit. Inicialmente las funciones de ARToolKit nos aislan de la complejidad de tratar con diferentes dispositivos de vídeo, por lo que la captura del *frame* actual es una simple llamada a función. Con esta imagen se inicia el primer paso de búsqueda de marcas. La imagen se convierte a blanco y negro para facilitar la detección de formas cuadradas (en realidad este paso se realiza en dos etapas (ver Figura 3.8); primero se convierte a escala de grises (b), y después se binariza (c) eligiendo un parámetro de umbral "*threshold*" que elige a partir de qué valor de gris (de entre 256 valores distintos) se considera blanco o negro (ver Figura 3.9).

A continuación el algoritmo de visión por computador extrae componentes conectados de la imagen previamente binarizada, Figura 3.8 (d), cuya area es suficientemente grande como para detectar una marca. A estas regiones se les aplica un rápido algoritmo de detección de contornos (e), obteniendo a continuación los vértices y aristas que definen la región de la marca en 2D (f).

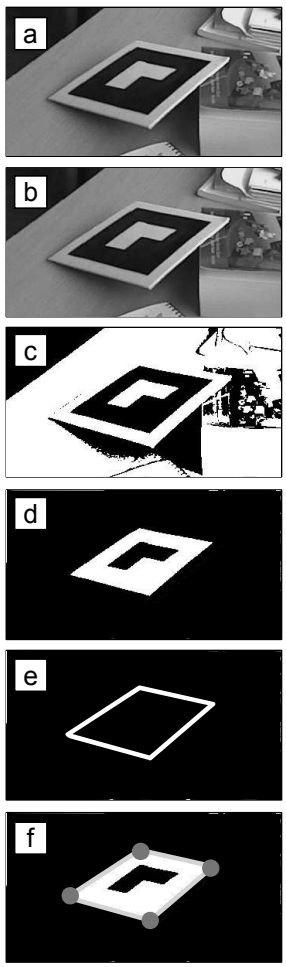


Figura 3.8: Pasos seguidos por ARToolKit para la detección e identificación de marcas.

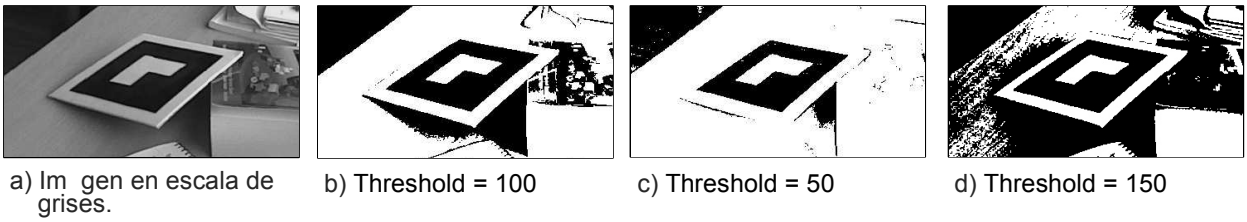


Figura 3.9: Ejemplo de binarización con diferentes valores de *Threshold*.

Con la región definida, se procede a una fase de normalización en la que se extrae el *contenido* de la marca (la zona central que la diferencia de otras marcas) y se compara con los patrones de las marcas conocidas (etapa 4 de la Figura 3.7).

Conociendo las posiciones 2D de las aristas y vértices que definen el marcador 2D, y el modelo de proyección de la cámara es posible estimar la posición y rotación 3D de la cámara relativamente a la marca.

No entraremos en detalles ahora sobre qué implica la obtención de la posición y rotación de la cámara (estos aspectos serán discutidos en los capítulos 4 y 5). El uso de marcas cuadradas de un tamaño previamente conocido nos permite definir un sistema de coordenadas local a cada marca, de modo que empleando métodos de visión por computador obtengamos la matriz de transformación 4x4 del sistema de coordenadas de la marca al sistema de coordenadas de la cámara T_{cm} (Ecuación 3.1).

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_x \\ R_{21} & R_{22} & R_{23} & T_y \\ R_{31} & R_{32} & R_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix} = T_{cm} \times \begin{bmatrix} X_m \\ Y_m \\ Z_m \\ 1 \end{bmatrix} \tag{3.1}$$

De este modo, conociendo la proyección de cada esquina de la marca (e) sobre las coordenadas de pantalla (xc, yc), y las restricciones asociadas al tamaño de las marcas y su geometría cuadrada, es posible calcular la matriz T_{cm}. La Figura 3.10 resume esta transformación.

El cálculo aproximado de la matriz de transformación que representa la rotación R y la traslación T desde las coordenadas de la marca a las coordenadas de la cámara se denominan en inglés *pose estimation* y *position estimation*.

Finalmente es posible dibujar cualquier objeto 3D correctamente alineado con la escena. Si conocemos la posición absoluta de las marcas respecto de algún sistema de coordenadas global, es posible representar los objetos posicionados globalmente. Veremos más detalles

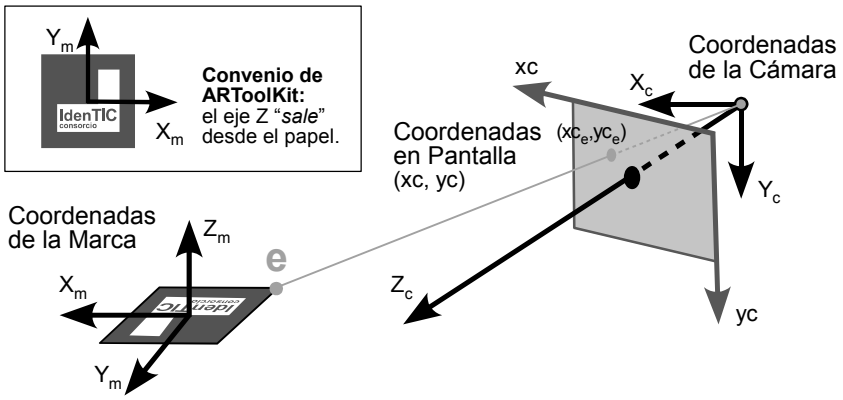


Figura 3.10: Sistema de coordenadas de ARToolKit

sobre el posicionamiento global en los próximos capítulos.

3.4.2. Calibración de la Cámara

Como vimos en el ejemplo del *"Hola Mundo!"*, los parámetros de la cámara se cargan en tiempo de ejecución de un archivo, mediante la llamada a la función `arParamLoad` (ver Listado de sección 3.3, línea 54). Aunque en el primer ejemplo trabajamos con un fichero de descripcion genérico de cámaras (válido para multitud de webcams), es preferible realizar un proceso previo de calibración que obtenga los parámetros intrínsecos de la cámara.

ARToolKit proporciona dos etapas de calibración; la primera (que es la que estudiaremos en esta sección) nos permite obtener un fichero de cámara válido para la mayoría de aplicaciones que contiene información sobre el **punto central de la imagen** y el **factor de distorsión** de las lentes. Si se quiere una mayor precisión (que contenga la **distancia focal** de la cámara), es necesario realizar la segunda etapa (ver documentación oficial de ARToolKit).

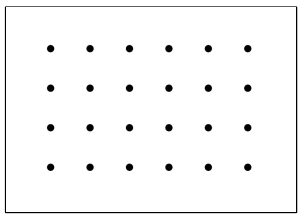


Figura 3.11: Patrón de calibración de la cámara.

Para realizar la calibración de una etapa de la cámara, necesitaremos imprimir el patrón de 24 puntos separados entre sí 4cm (ver Figura 3.11). Hecho esto, ejecutaremos la aplicación `calib_camera2` de la distribución de ARToolKit. La aplicación nos pedirá la longitud en milímetros entre cada punto del patrón de calibración (deberá ser 40 si hemos imprimido el patrón a tamaño 1:1, en otro caso mediremos el espacio físico entre el centro de los puntos).

```
carlos@kurt:calib_camera2$ ./calib_camera2
Input the distance between each marker dot, in millimeters: 40
-----------
Press mouse button to grab first image,
or press right mouse button or [esc] to quit.
```

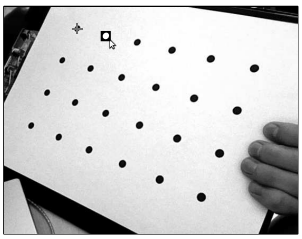


Figura 3.12: Marcado de círculos del patrón.

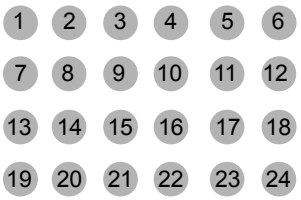


Figura 3.13: Orden de marcado de los círculos del patrón de calibración.

Hecho esto nos aparecerá una ventana con la imagen que percibe la cámara. Moveremos el patrón para que todos los puntos aparezcan en la imagen y presionaremos el botón izquierdo del ratón una vez sobre la ventana para congelar la imagen. Ahora tenemos que definir un rectángulo que rodee cada círculo del patrón (ver Figura 3.12 empleando el siguiente orden: primero el círculo más cercano a la esquina superior izquierda, y a continuación los de su misma fila. Luego los de la segunda fila comenzando por el de la izquierda y así sucesivamente. Es decir, los círculos del patrón deben ser recorridos en orden indicado en la Figura 3.13.

El programa marcará una pequeña cruz en el centro de cada círculo que hayamos marcado (ver Figura 3.12), y aparecerá una línea indicando que ha sido señalada como se muestra a continuación. Si no aparece una cruz en rojo es porque el círculo no se ha detectado y tendrá que ser de nuevo señalado.

```
Grabbed image 1.
-----------
Press mouse button and drag mouse to rubber-bound features (6 x 4),
or press right mouse button or [esc] to cancel rubber-bounding & retry
     grabbing.
Marked feature position   1 of  24
Marked feature position   2 of  24
Marked feature position   3 of  24
...
Marked feature position  24 of  24
-----------
Press mouse button to save feature positions,
or press right mouse button or [esc] to discard feature positions &
    retry grabbing.
```

Una vez que se hayan marcado los 24 puntos, se pulsa de nuevo el botón izquierdo del ratón sobre la imagen. Esto almacenará la posición de las marcas para la primera imagen, y descongelará la cámara, obteniendo una salida en el terminal como la siguiente.

```
### Image no.1 ###
 1,  1: 239.50, 166.00
 2,  1: 289.00, 167.00
 ...
 6,  4: 514.00, 253.50
-----------
Press mouse button to grab next image,
or press right mouse button or [esc] to calculate distortion param.
```

Precisión en calibración

Como es obvio, a mayor número de imágenes capturadas y marcadas, mayor precisión en el proceso de calibración. Normalmente con 5 ó 6 imágenes distintas suele ser suficiente.

Como se indica en el manual de ARToolKit, es necesario capturar entre 5 y 10 imágenes siguiendo el mismo proceso, variando el ángulo y la posición en la que se presenta el patrón de calibración. En la Figura 3.14 se muestran 6 ejemplos de diferentes imágenes capturadas para la calibración de una cámara.

Cuando te haya realizado un número de capturas adecuado pulsaremos el botón derecho del ratón (o la tecla ESC) para calcular el parámetro de distorsión. En el terminal aparecerá una salida como la

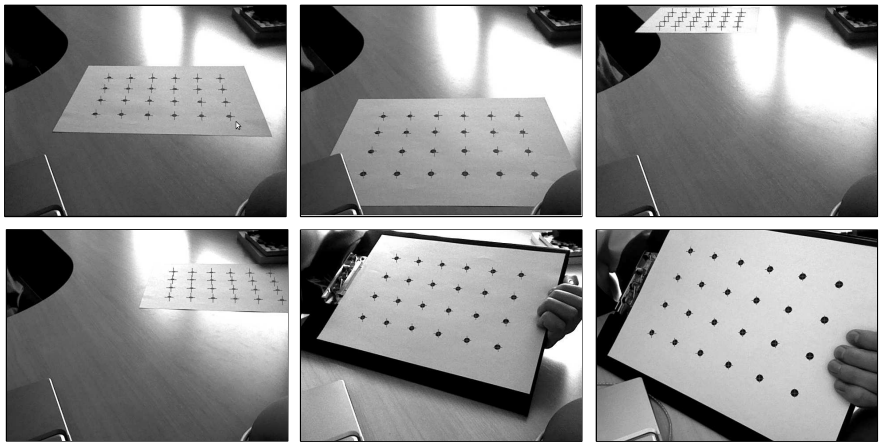


Figura 3.14: Ejemplo de posicionamiento de patrones de calibración.

siguiente, indicando al final el centro (X,Y) de la cámara, y el factor de distorsión. El cálculo de estos parámetros puede requerir varios segundos.

```
Press mouse button to grab next image,
or press right mouse button or [esc] to calculate distortion param.
[320.0, 240.0, -13.5] 0.459403
[370.0, 190.0, -15.3] 0.457091
...
[375.0, 211.0, -16.4] 0.456635
--------------
Center X: 375.000000
       Y: 211.000000
Dist Factor: -16.400000
Size Adjust: 0.978400
--------------
```

A continuación la aplicación nos muestra el resultado del cálculo de estos parámetros, para que el usuario realice una comprobación visual. Cada vez que pulsemos con el botón izquierdo del ratón se mostrará una imagen con líneas de color rojo que deben pasar por el centro de cada círculo de calibración (como las dos imágenes mostradas en la Figura 3.15).

```
Checking fit on image   1 of   6.
Press mouse button to check fit of next image.
```

Finalmente, tras aceptar los resultados mostrados por la aplicación, se calcularán todos los parámetros que se guardarán en el fichero que le indiquemos al final del proceso.

```
-- loop:-50 --
F = (816.72,746.92), Center = (325.0,161.0): err = 0.843755
-- loop:-49 --
F = (816.47,747.72), Center = (325.0,162.0): err = 0.830948
...
```

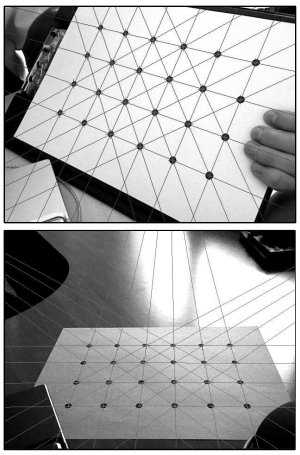


Figura 3.15: Dos imágenes resultado del cálculo del centro y el factor de distorsión.

```
Calibration succeeded. Enter filename to save camera parameter.
-------------------------------------
SIZE = 640, 480
Distortion factor = 375.000000 211.000000 -16.400000 0.978400
770.43632 0.00000 329.00000 0.00000
0.00000 738.93605 207.00000 0.00000
0.00000 0.00000 1.00000 0.00000
-------------------------------------
Filename: logitech_usb.dat
```

Este nuevo fichero creado será el que le indicaremos a ARToolKit que utilice en la llamada a la función `arParamLoad`. En los capítulos 4 y 5 estudiaremos la importancia de la etapa de calibración de la cámara (sobre todo en aquellas que tengan un valor elevado de distorsión).

3.4.3. Detección de Marcas

En la sección 3.4.1 hemos visto que ARToolKit extrae los vértices que definen las cuatro esquinas de una marca. En la etapa de detección de la marca, es necesario además identificar qué patrón está siendo detectado. Para ello ARToolKit primero normaliza la imagen detectada (eliminando la distorsión debida a la perspectiva) para posteriormente comparar el patrón con las plantillas de los patrones que puede reconocer la aplicación (aquellas que hayan sido cargadas con la función `arLoadPatt`, ver línea [61] del listado de la sección 3.3).

El proceso de normalización de los patrones requiere mucho tiempo, por lo que es un problema cuando se utilizan muchos marcadores. Tras la normalización, se reduce la resolución del patrón normalizado antes de pasar a la fase de comparación. Cuanto mayor es la resolución del patrón reescalado, mayor es la precisión de ARToolKit, pero require más capacidad de cómputo para realizar las operaciones.

En el fichero `config.h` de ARToolKit pueden defirse algunos parámetros relacionados con la detección de marcas. A continuación se indican los valores que trae por defecto la distribución de ARToolKit:

- `#define AR_SQUARE_MAX 50`: Este parámetro define el número máximo de marcadores que serán detectados en cada imagen.
- `#define AR_PATT_NUM_MAX 50`: Número máximo de patrones que pueden ser cargados por el usuario.
- `#define AR_PATT_SIZE_X 16`: Resolución (número de píxeles) en horizontal del patrón cuando es resampleado.
- `#define AR_PATT_SIZE_Y 16`: Resolución (número de píxeles) en horizontal del patrón cuando es resampleado.
- `#define AR_PATT_SAMPLE_NUM 64`: Número máximo de pasos empleados para resamplear el patrón.

De esta forma, el patrón reconocido incialmente es normalizado y resampleado para obtener una imagen como se muestra en la Figura

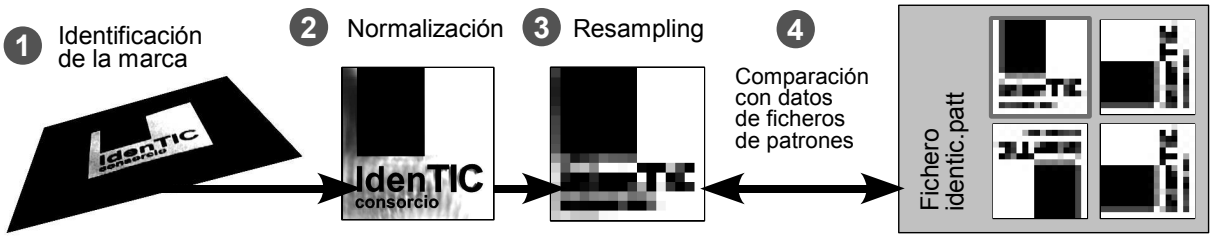


Figura 3.16: Pasos para la identificación de patrones.

3.16. Por defecto, esta representación se realizará en 64 pasos, generando una imagen de 16x16 píxeles. Esta matriz de 16x16 píxeles se compará con los datos contenidos en los ficheros de patrones. Estos ficheros simplemente contienen 4 matrices con el valor de gris de cada píxel del patrón. Cada matriz se corresponde con el patrón rotado 90º (ver Figura 3.16). Podemos abrir cualquier fichero .patt (como el que utilizamos en el *Hola Mundo!*) y veremos los valores en ASCII correspondientes a las 4 matrices.

Limitaciones

Si utilizamos únicamente métodos de registro basados en marcas como en el caso de ARToolKit, la principal limitación viene cuando ninguna marca es visible en un momento dado, ya que no es posible obtener la posición de la cámara virtual. Además, si se oculta alguna parte de la marca, el método de detección, tal y como hemos visto, no será capaz de identificar las 4 esquinas de la región y el método de identificación fallará.

Entre los factores que afectan a la detección de los patrones podemos destacar su tamaño, complejidad, orientación relativa a la cámara y condiciones de iluminación de la escena (ver Figura 3.17).

El **tamaño físico de la marca** afecta directamente a la facilidad de detección; a mayor tamaño de marca, mayor distancia puede ser cubierta. Por ejemplo, marcas de 7 cm de lado pueden ser detectadas hasta una distancia máxima de 40 cm (a una resolución de 640x480 píxeles). Si aumentamos el tamaño de la marca a 18cm, ésta será detectada hasta una distancia de 125cm.

Este rango de detección se ve también afectado por la **complejidad de la marca**. Los patrones simples (con grandes áreas de color blanco o negro) son detectados mejor.

La **orientación relativa** de la marca con respecto a la cámara afecta a la calidad de la detección. A mayor perpendicularidad entre el eje Z de la marca y el vector *look* de la cámara, la detección será peor.

Finalmente las **condiciones de iluminación** afectan enormemente a la detección de las marcas. El uso de materiales que no ofrezcan

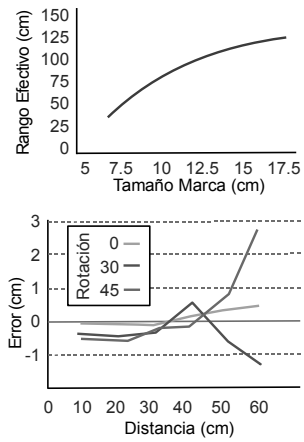


Figura 3.17: Arriba Relación entre el tamaño de la marca y el rango máximo de detección efectiva. **Abajo** Relación entre el error cometido, la distancia de detección, y el ángulo relativo entre la marca y la cámara.

brillo especular, y que disminuyan el reflejo en las marcas mejoran notablemente la detección de las marcas.

Creación de Marcas

ARToolKit dispone de una llamada a función `arSavePatt` que nos permite crear patrones personalizados. Utilizaremos esta función para construir un sencillo ejemplo que nos permitirá entrenar nuestras propias marcas. Modificaremos el *Hola Mundo!* de la sección anterior (ver listado siguiente). Este programa recibirá un argumento, que será el nombre del fichero donde queremos guardar el patrón entrenado. Así, la utilidad se llamará desde el terminal como:

```
carlos@kurt:02_patron$ ./makepatt identic.patt
```

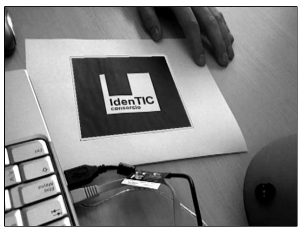


Figura 3.18: Ejemplo de salida en pantalla de la aplicación para entrenar patrones.

Cuando se detecta una marca, el programa dibuja sus límites con líneas de color verde (ver Figura 3.18). Mientras la marca está siendo detectada, el usuario podrá pulsar la tecla `s` para guardar el fichero del patrón (hecho esto, automáticamente liberará los recursos asignados y saldrá). Las teclas `q` o `ESC` servirán para cerrar la aplicación sin guardar el patrón.

Listado 3.3: Generador de patrones

```
1  #include <string.h>
2  #include <GL/glut.h>
3  #include <AR/gsub.h>
4  #include <AR/video.h>
5  #include <AR/param.h>
6  #include <AR/ar.h>
7
8  // ==== Definicion de constantes y variables globales ====
9  ARMarkerInfo *gPatt;          // Patron a guardar
10 char         gPattName[128];  // Nombre del fichero (patron)
11 ARUint8      *gImage=NULL;    // Ultima imagen capturada
12 int          imgSize=0;       // Tamano en bytes de la imagen
13 int          xsize, ysize;    // Tamano del video de camara
       (pixels)
14
15 void print_error (char *error) {  printf(error); exit(0); }
16 /* La funcion draw_line dibuja un segmento 2D desde el vertice (x1,
       y1) al (x2, y2).*/
17 void draw_line (double x1, double y1, double x2, double y2) {
18   glBegin(GL_LINES);
19   glVertex2f(x1, ysize - y1);
20   glVertex2f(x2, ysize - y2);
21   glEnd();
22 }
23 // ======== cleanup ========================================
24 static void cleanup(void) {
25   arVideoCapStop();            // Libera recursos al salir ...
26   arVideoClose();
27   argCleanup();
28   free(gImage);
29   exit(0);
30 }
```

```
31 /* La funcion draw unicamente dibuja las lineas de color verde en 2
       D. Utiliza la funcion auxiliar propia draw_line.*/
32 // ======== draw ==========================================
33 static void draw( void ) {
34   glLineWidth(5);
35   glColor3d(0, 1, 0);
36   draw_line(gPatt->vertex[0][0], gPatt->vertex[0][1],
37         gPatt->vertex[1][0], gPatt->vertex[1][1]);
38   draw_line(gPatt->vertex[1][0], gPatt->vertex[1][1],
39         gPatt->vertex[2][0], gPatt->vertex[2][1]);
40   draw_line(gPatt->vertex[2][0], gPatt->vertex[2][1],
41         gPatt->vertex[3][0], gPatt->vertex[3][1]);
42   draw_line(gPatt->vertex[3][0], gPatt->vertex[3][1],
43         gPatt->vertex[0][0], gPatt->vertex[0][1]);
44 }
45 /* La funcion keyboard es una funcion de retrollamada. Cuando el
       usuario pulse una tecla, sera invocada por GLUT, pasandole la
       tecla en key y la posicion del raton (en x,y).*/
46 // ======== keyboard ======================================
47 static void keyboard(unsigned char key, int x, int y) {
48   switch (key) {
49   case 0x1B: case 'Q': case 'q':
50     cleanup(); break;
51   case 's': case'S':
52     if (gPatt != NULL) {
53       if (arSavePatt(gImage, gPatt, gPattName)<0)
54         printf ("Error guardando patron %s\n", gPattName);
55       else printf ("Patron %s guardado correctamente\n", gPattName)
             ;
56       cleanup();
57     } else printf ("Patron actualmente no detectado\n");
58     break;
59   }
60 }
61 (*@\newpage@*)
62 // ======== init ==========================================
63 static void init( void ) {
64   ARParam  wparam, cparam;   // Parametros intrinsecos de la camara
65
66   // Abrimos dispositivo de video
67   if(arVideoOpen("") < 0) exit(0);
68   if(arVideoInqSize(&xsize, &ysize) < 0) exit(0);
69
70   // Cargamos los parametros intrinsecos de la camara
71   if(arParamLoad("data/camera_para.dat", 1, &wparam) < 0)
72     print_error ("Error en carga de parametros de camara\n");
73
74   arParamChangeSize(&wparam, xsize, ysize, &cparam);
75   arInitCparam(&cparam);   // Inicializamos la camara con çparam"
76
77   imgSize = cparam.xsize * cparam.ysize * AR_PIX_SIZE_DEFAULT;
78   arMalloc(gImage, ARUint8, imgSize); // Reservamos memoria Imagen
79
80   argInit(&cparam, 1.0, 0, 0, 0, 0);   // Abrimos la ventana
81 }
82
83 // ======== mainLoop ======================================
84 static void mainLoop(void) {
85   ARUint8 *dataPtr;
86   ARMarkerInfo *marker_info;
87   int marker_num, i, maxarea;
88
```

```
89    // Capturamos un frame de la camara de video
90    if((dataPtr = (ARUint8 *)arVideoGetImage()) == NULL) {
91      // Si devuelve NULL es porque no hay un nuevo frame listo
92      arUtilSleep(2);  return;  // Dormimos el hilo 2ms y salimos
93    }
94
95    argDrawMode2D();
96    argDispImage(dataPtr, 0,0);    // Dibujamos lo que ve la camara
97
98    // Detectamos la marca en el frame capturado (return -1 si error)
99    if(arDetectMarker(dataPtr, 100, &marker_info, &marker_num) < 0) {
100     cleanup(); exit(0);   // Si devolvio -1, salimos del programa!
101   }
102
103   // Nos quedamos con el patron detectado de mayor tamano
104   for(i = 0, maxarea=0, gPatt=NULL; i < marker_num; i++) {
105     if(marker_info[i].area > maxarea){
106       maxarea = marker_info[i].area;
107       gPatt = &(marker_info[i]);
108       memcpy(gImage, dataPtr, imgSize);
109     }
110   }
111
112   if(gPatt != NULL) draw();
113
114   argSwapBuffers(); // Cambiamos el buffer con lo dibujado
115   arVideoCapNext(); // Frame pintado y analizado... A por otro!
116 }
117
118 // ======== Main ========================================
119 int main(int argc, char **argv) {
120   glutInit(&argc, argv);    // Creamos la ventana OpenGL con Glut
121
122   if (argc == 2)                    // El primer argumento es el
123     strcpy (gPattName, argv[1]);    // nombre del patron
124   else {printf ("Llamada %s <nombre_patron>\n", argv[0]); exit(0);}
125
126   init();                   // Llamada a nuestra funcion de inicio
127
128   arVideoCapStart();        // Creamos hilo para captura de video
129   argMainLoop( NULL, keyboard, mainLoop );  // Asociamos callbacks.
130   return (0);
131 }
```

El listado anterior se estructura en los mismos bloques de funciones que el *Hola Mundo!*. En la función `main`, líneas (119-131) obtenemos el nombre del archivo donde guardaremos el patrón (en caso de no tener un parámetro cuando se llama al programa, se muestra una pequeña ayuda). Este nombre se guarda en una variable global `gPattName` (línea (10)). En la línea (129) se asocia el callback de pulsación de teclado, en la función `keyboard` (definida en las líneas (47-60)).

La función `init` se encarga, al igual que en el ejemplo anterior, de inicializar el dispositivo de vídeo, cargar los parámetros de la cámara, y además, de reservar memoria para una variable global (línea (78)) llamada `gImage`. En esta posición de memoria se volcará una copia del frame actual, en el caso de que detectemos una marca. Esta reserva de memoria no se puede hacer *a priori*, porque hasta el momento de la

ejecución del programa, no sabemos el tamaño de vídeo que cargaremos de la cámara.

Las principales modificaciones del `mainLoop` (líneas 84–116) se localizan en las líneas 104–110. En este caso, si se ha detectado marca, nos quedamos con aquella que tenga un área de detección mayor. Esto se comprueba con una variable llamada `maxarea`. En el caso de que detectemos alguna marca, guardamos el puntero a la marca de mayor área en `gPatt` (línea 107), y hacemos una copia del frame en `gImage` (línea 108).

De esta forma, cuando el usuario pulse la tecla `s` y se llame a la función de callback de teclado (líneas 47–60), se podrá guardar el patrón. La función `arSavePatt` (línea 53) requiere como primer parámetro el puntero al frame (almacenado en `gImage`), como segundo parámetro la información `ARMarkerInfo` de la marca detectada, y finalmente el nombre del fichero donde se quiere guardar el patrón.

Para finalizar, la función de dibujado `draw` (líneas 33–44) muestra en 2D los bordes del patrón detectado. Se apoya en una función auxiliar `draw_line` que toma como parámetros los 2 vértices de las línea a dibujar. El convenio que toma OpenGL para especificar el origen de la pantalla es en la esquina inferior izquierda. Debido a que ARToolKit toma como origen la esquina superior izquierda, hay que realizar una conversión para especificar la posición final del vértice (una simple resta con el tamaño en vertical de la pantalla, líneas 19,20).

3.5. Ejercicios Propuestos

Se recomienda la realización de los ejercicios de esta sección en orden, ya que están relacionados y su complejidad es ascendente.

Figura 3.19: Marca que puede ser entrenada.

1. Utilice el programa de calibración de cámara de ARToolKit para crear un fichero adaptado a su dispositivo de captura. Modifique el programa estudiado en la sección 3.3 para usar el fichero.

2. Entrene con el programa de la sección 3.4.3 una marca propia (por ejemplo, la marca de la Figura 3.19). Modifique el programa de la sección 3.3 para que utilice esa marca.

3. Modifique el programa de la sección 3.4.3 para que trabaje con dos marcas; las definidas en la figuras 3.4 y 3.19. En caso de detectar ambas marcas, el programa se quedará como *marca activa* aquella que tenga mayor valor de confianza (campo `cf` de la estructura `ARMarkerInfo`). Cambiaremos el color[2] de la tetera entre azul y verde, según la marca activa sea una u otra.

[2]Para cambiar el color del objeto en el ejemplo del listado de la sección 3.3, es necesario modificar las primeras componentes del vector `mat_ambient` (línea 22), que se corresponden con los valores RGB (entre 0 y 1). El cuarto componente es el valor de transparencia Alpha.

Capítulo 4

Fundamentos Matemáticos

En este capítulo estudiaremos algunos conceptos teóricos básicos para el trabajo con objetos 3D en el ámbito de la Realidad Aumentada. En concreto veremos cómo transformar posición, rotación y tamaño de objetos empleando transformaciones geométricas en su representación matricial. Igualmente, y por su importancia en el ámbito de la realidad aumentada estudiaremos en detalle la proyección en perspectiva, por ser ésta la que mejor se ajusta a dispositivos de vídeo reales.

En este capítulo comenzaremos a estudiar las transformaciones afines más básicas que serán necesarias para el desarrollo de aplicaciones de Realidad Aumentada. Las transformaciones son herramientas imprescindibles para cualquier aplicación que manipule geometría o, en general, objetos descritos en el espacio 3D. La mayoría de APIs que trabajan con gráficos 3D implementan funciones auxiliares para trabajar con matrices.

Para introducir los conceptos generales asociados a las transformaciones, comenzaremos con una discusión sobre las operaciones en 2D para pasar a la notación matricial 2D y, posteriormente, a la generalización tridimensional empleando coordenadas homogeneas.

4.1. Transformaciones Geométricas

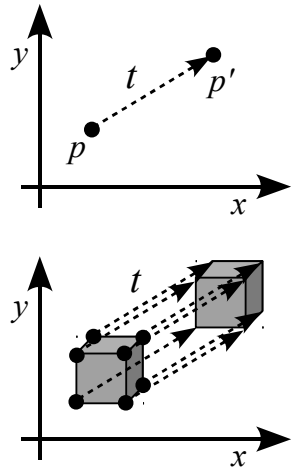


Figura 4.1: Arriba. Traslación de un punto p a p' empleando el vector t. **Abajo**. Es posible trasladar un objeto poligonal completo aplicando la traslación a todos sus vértices.

En la representación de gráficos 3D es necesario contar con herramientas para la transformación de los objetos básicos que compondrán la escena. A menudo, estas primitivas son conjuntos de triángulos que definen mallas poligonales. Las operaciones que se aplican a estos triángulos para cambiar su posición, orientación y tamaño se denominan **transformaciones geométricas**. En general podemos decir que una transformación toma como entrada elementos como vértices y vectores y los convierte de *alguna manera*.

La transformación básica bidimensional más sencilla es la **traslación**. Se realiza la traslación de un punto mediante la suma de un vector de desplazamiento a las coordenadas iniciales del punto, para obtener una nueva posición de coordenadas. Si aplicamos esta traslación a *todos* los puntos del objeto, estaríamos desplazando ese objeto de una posición a otra. De este modo, podemos definir la traslación como la suma de un vector libre de traslación t a un punto original p para obtener el punto trasladado p' (ver Figura 4.1). Podemos expresar la operación anterior como:

$$p'_x = p_x + t_x \qquad p'_y = p_y + t_y \tag{4.1}$$

De igual modo podemos expresar una **rotación** de un punto $p = (x, y)$ a una nueva posición rotando un ángulo θ respecto del origen de coordenadas, especificando el eje de rotación y un ángulo θ. Las coordenadas iniciales del punto se pueden expresar como (ver Figura 4.2):

$$p_x = d\,cos\alpha \qquad p_y = d\,sen\alpha \tag{4.2}$$

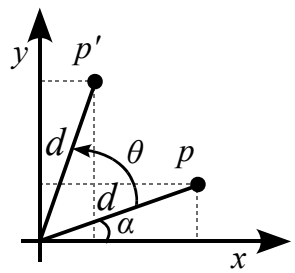


Figura 4.2: Rotación del punto p un ángulo θ respecto del origen de coordenadas.

Siendo d la distancia entre el punto y el origen del sistema de coordenadas. Así, usando identidades trigonométricas se pueden expresar las coordenadas transformadas como la suma de los ángulos del punto original α y el que queremos rotar θ como:

$$p'_x = d\,cos(\alpha + \theta) = d\,cos\alpha\,cos\theta - d\,sen\alpha\,sen\theta$$
$$p'_y = d\,sen(\alpha + \theta) = d\,cos\alpha\,sen\theta - d\,sen\alpha\,cos\theta$$

0mm

Que sustituyendo en la ecuación 4.2, obtenemos:

$$p'_x = p_x\,cos\theta - p_y\,sen\theta \qquad p'_y = p_x\,sin\theta - p_y\,cos\theta \tag{4.3}$$

De forma similar, un **cambio de escala** de un objeto bidimensional puede llevarse a cabo multiplicando las componentes x, y del objeto por el factor de escala S_x, S_y en cada eje. Así, como se muestra en la Figura 4.3 un cambio de escala se puede expresar como:

$$p'_x = p_x S_x \qquad p'_y = p_y S_y \tag{4.4}$$

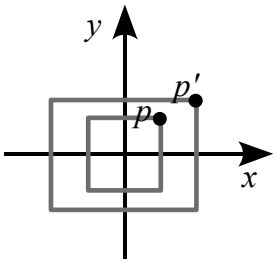


Figura 4.3: Conversión de un cuadrado a un rectángulo empleando los factores de escala $S_x = 2, S_y = 1{,}5$.

Cuando queremos cambiar la localización de un objeto, habitualmente necesitamos especificar una **combinación de traslaciones y rotaciones** en el mismo (por ejemplo, cuando cogemos el teléfono móvil de encima de la mesa y nos lo guardamos en el bolsillo, sobre el objeto se aplican varias traslaciones y rotaciones). Es interesante por tanto disponer de alguna representación que nos permita combinar transformaciones de una forma eficiente.

4.1.1. Representación Matricial

En muchas aplicaciones gráficas los objetos deben transformarse geométricamente de forma constante (por ejemplo, en el caso de una animación, en la que en cada *frame* el objeto debe cambiar de posición. En el ámbito de la Realidad Aumentada, aunque un objeto asociado a una marca permanezca inmóvil, deberemos cambiar la posición de la cámara virtual para que se ajuste a los movimientos del punto de vista del observador.

Coord. homogéneas

Gracias al uso de coordenadas homogéneas es posible representar las ecuaciones de transformación geométricas como multiplicación de matrices, que es el método estándar en gráficos por computador (soportado en hardware por las tarjetas aceleradoras gráficas).

De este modo resulta crítica la eficiencia en la realización de estas transformaciones. Como hemos visto en la sección anterior, las ecuaciones 4.1, 4.3 y 4.4 nos describían las operaciones de traslación, rotación y escalado. Para la primera es necesario realizar una *suma*, mientras que las dos últimas requieren *multiplicaciones*.

Sería conveniente poder **combinar las transformaciones** de forma que la posición final de las coordenadas de cada punto se obtenga de forma directa a partir de las coordenadas iniciales. Si reformulamos la escritura de estas ecuaciones para que todas las operaciones se realicen multiplicando, podríamos conseguir homogeneizar estas transformaciones.

Si añadimos un término extra (parámetro homogéneo h) a la representación del punto en el espacio (x, y), obtendremos la **representación homogénea** de la posición descrita como (x_h, y_h, h). Este *parámetro homogéneo* h es un valor distinto de cero tal que $x = x_h/h$, $y = y_h/h$. Existen, por tanto infinitas representaciones homogéneas equivalentes de cada par de coordenadas, aunque se utiliza normalmente $h = 1$. Veremos en la sección 4.2.2 que no siempre el parámetro $h = 1$.

De este modo, la operación de traslación, que hemos visto anteriormente, puede expresarse de forma matricial como:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{4.5}$$

Al resolver la multiplicación matricial se obtienen un conjunto de ecuaciones equivalentes a las enunciadas en 4.1. Las operaciones de rotación T_r y escalado T_s tienen su equivalente matricial homogéneo.

$$T_r = \begin{bmatrix} cos\theta & -sen\theta & 0 \\ sen\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad T_s = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

Las transformaciones inversas pueden realizarse sencillamente cambiando el signo en el caso de la traslación y rotación (distancias y ángulos negativos), y obteniendo el parámetro recíproco en el caso de la escala $1/S_x, 1/S_y$.

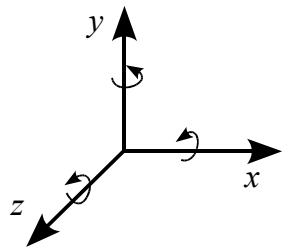


Figura 4.4: Sentido de las rotaciones positivas respecto de cada eje de coordenadas.

Las **transformaciones en el espacio 3D** requieren simplemente añadir el parámetro homogéneo y describir las matrices (en este caso 4x4). Así, las traslaciones T_t y escalados T_s en 3D pueden representarse de forma homogénea mediante las siguientes matrices:

$$T_t = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad T_s = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

Las rotaciones requieren distinguir el eje sobre el que se realizará la rotación. Las **rotaciones positivas** alrededor de un eje se realizan en sentido opuesto a las agujas del reloj, cuando se está mirando a lo largo de la mitad positiva del eje hacia el origen del sistema de coordenadas (ver Figura 4.4). Las expresiones matriciales de las rotaciones son las siguientes:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sen\theta & 0 \\ 0 & sen\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} cos\theta & 0 & sen\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sen\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z = \begin{bmatrix} cos\theta & -sen\theta & 0 & 0 \\ sen\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.8}$$

Las tres transformaciones estudiadas (traslación, rotación y escalado) son ejemplos de **transformaciones afines**, en las que cada una de las coordenadas transformadas se pueden expresar como una función lineal de la posición origen, y una serie de constantes determinadas por el tipo de transformación.

Una de las características interesantes de este tipo de transformaciones es que conservan la *colineridad*, por lo que las líneas paralelas seguirán siendo paralelas.

4.1.2. Transformaciones Inversas

En muchas situaciones resulta interesante calcular la inversa de una matriz. Un ejemplo típico es en la resolución de ecuaciones, como

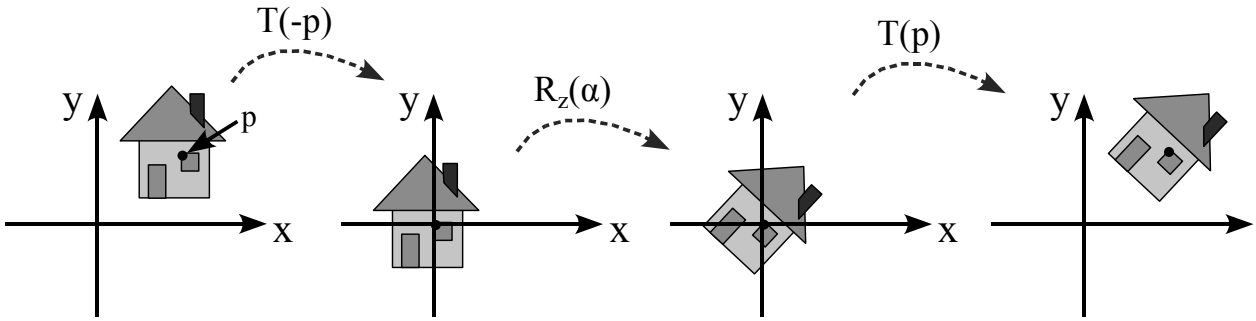


Figura 4.5: Secuencia de operaciones necesarias para rotar una figura respecto de un origen arbitrario.

en el caso de la expresión $A = Bx$. Si queremos obtener el valor de B, tendríamos $B = A/x$. Por desgracia, las matrices no tienen asociado un operador de división, por lo que debemos usar el concepto de *matriz inversa*.

Para una matriz A, se define su inversa A^{-1} como la matriz que, multiplicada por A da como resultado la matriz identidad I:

$$A \cdot A^{-1} = A^{-1} \cdot A = I \tag{4.9}$$

Matrices Inversas

No todas las matrices tienen inversa (incluso siendo cuadradas). Un caso muy simple es una matriz cuadrada cuyos elementos son cero.

Tanto la matriz A como su inversa deben ser *cuadradas* y del mismo tamaño. Otra propiedad interesante de la inversa es que la inversa de la inversa de una matriz es igual a la matriz original $(A^{-1})^{-1} = A$.

En la ecuación inicial, podemos resolver el sistema utilizando la matriz inversa. Si partimos de $A = B \cdot x$, podemos multiplicar ambos términos a la izquierda por la inversa de B, teniendo $B^{-1} \cdot A = B^{-1} \cdot B \cdot x$, de forma que obtenemos la matriz identidad $B^{-1} \cdot A = I \cdot x$, con el resultado final de $B^{-1} \cdot A = x$.

En algunos casos el cálculo de la matriz inversa es directo, y puede obtenerse de forma intuitiva. Por ejemplo, en el caso de una traslación pura (ver ecuación 4.7), basta con emplear como factor de traslación el mismo valor en negativo. En el caso de escalado, como hemos visto bastará con utilizar $1/S$ como factor de escala. Cuando se trabaja con matrices compuestas el cálculo de la inversa tiene que realizarse con métodos generales como por ejemplo el método de eliminación de Gauss, o el uso de la traspuesta de la matriz adjunta.

4.1.3. Composición

Como hemos visto en la sección anterior, una de las principales ventajas derivadas del trabajo con sistemas homogéneos es la **composición de matrices**. Matemáticamente esta *composición* se realiza multiplicando las matrices en un orden determinado, de forma que es posible obtener la denominada **matriz de transformación neta** M_N

resultante de realizar sucesivas transformaciones a los puntos. De este modo, bastará con multiplicar la M_N a cada punto del modelo para obtener directamente su posición final. Por ejemplo, si P es el punto original y P' es el punto transformado, y $T_1 \cdots T_n$ son transformaciones (rotaciones, escalados, traslaciones) que se aplican al punto P, podemos expresar la transformación neta como:

$$P' = T_n \times \cdots \times T_2 \times T_1 \times P$$

Este orden de multiplicación de matrices es el habitual empleado en gráficos por computador, donde las transformaciones se premultiplican (la primera está más cerca del punto original P, más a la derecha.

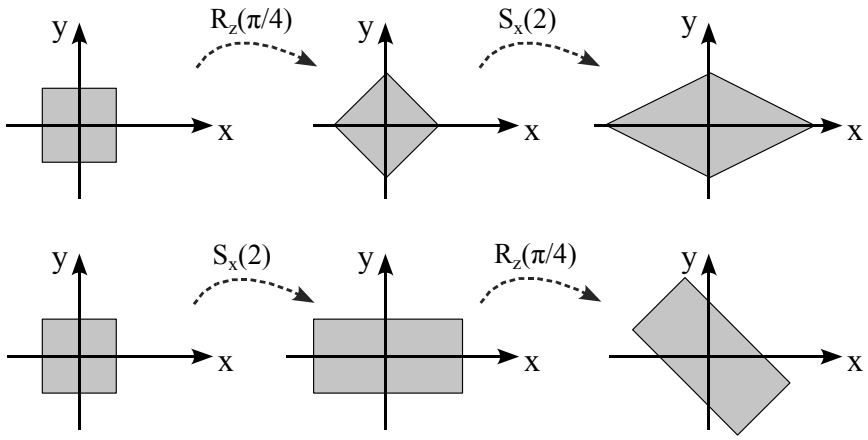


Figura 4.6: La multiplicación de matrices no es conmutativa, por lo que el orden de aplicación de las transformaciones es relevante para el resultado final. Por ejemplo, la figura de arriba primero aplica una rotación y luego el escalado, mientras que la secuencia inferior aplica las dos transformaciones en orden inverso.

La matriz de transformación neta M_N se definiría en este caso como $M_N = T_n \times \cdots T_2 \times T_1$, de tal forma que sólo habría que calcularla una vez para todos los puntos del modelo y aplicarla a todos vértices en su posición original para obtener directamente su posición final. De este modo, si un objeto poligonal está formado por V vértices, habrá que calcular la matriz de transformación neta M_N y aplicarla una vez a cada vértice del modelo.

$$P' = M_N \times P$$

Conmutatividad. Recordemos que la multiplicación de matrices es asociativa, pero *no* es conmutativa, por lo que el orden de aplicación de las transformaciones es importante (ver Figura 4.6).

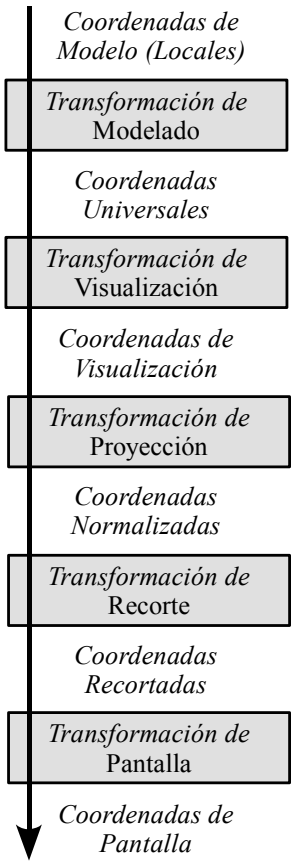


Figura 4.7: Pipeline general tridimensional.

Otro aspecto a tener en cuenta es que la expresión de las transformaciones para trabajar con coordenadas homogeneas, que se han comentado en las ecuaciones 4.7 y 4.8 se refieren al **Sistema de Referencia Universal** (*SRU*) o Sistema de Referencia Global.

Esto implica que si se quiere realizar una transformación respecto de un punto distinto a ese origen del sistema de referencia universal, habrá que hacer coincidir primero el punto con el origen del sistema de referencia, aplicar la transformación y devolver el objeto a su posición original. Así, en el ejemplo de la Figura 4.5 si queremos rotar el objeto respecto del punto p es necesario, primero trasladar el objeto para que su origen quede situado en el origen del *SRU*, luego aplicar la rotación, y finalmente aplicar la traslación inversa. De este modo, la Matriz Neta quedaría definida como $M_N = T_{-p} \times R_z \times T_p$.

4.2. Visualización 3D

En esta sección se describirán algunas características que deben tenerse en cuenta a la hora de visualizar gráficos 3D por computador. Como no puede ser de otro modo, será únicamente un breve resumen de las características más importantes que deben conocerse para desarrollar aplicaciones de Realidad Aumentada, sin abarcar toda la teoría relacionada.

Para obtener una imagen de una escena 3D definida en el *Sistema de Referencia Universal*, necesitamos definir un sistema de referencia de coordenadas para los **parámetros de visualización** (también denominados *parámetros de cámara*). Este sistema de referencia nos definirá el plano de proyección, que sería el equivalente de la zona de la cámara sobre la que se registrará la imagen[1]. De este modo se transfieren los objetos al sistema de coordenadas de visualización y finalmente se proyectan sobre el plano de visualización (ver Figura 4.8).

En gráficos por computador es posible elegir entre diferentes modelos de proyección de los objetos sobre el plano de visualización. Un modo muy utilizado en aplicaciones de CAD es la proyección de los objetos empleando líneas paralelas sobre el plano de proyección, mediante la denominada **proyección paralela**. En este modo de proyección se conservan las proporciones relativas entre objetos, independientemente de su distancia.

Mediante la **proyección en perspectiva** se proyectan los puntos hasta el plano de visualización empleando trayectorias convergentes en un punto. Esto hace que los objetos situados más distantes del plano de visualización aparezcan más pequeños en la imagen. Las escenas generadas utilizando este modelo de proyección son más realistas, ya que ésta es la manera en que el ojo humano y las cámaras físicas forman imágenes.

[1] En el mundo físico, la *película* en antiguas cámaras analógicas, o el sensor de imagen de las cámaras digitales.

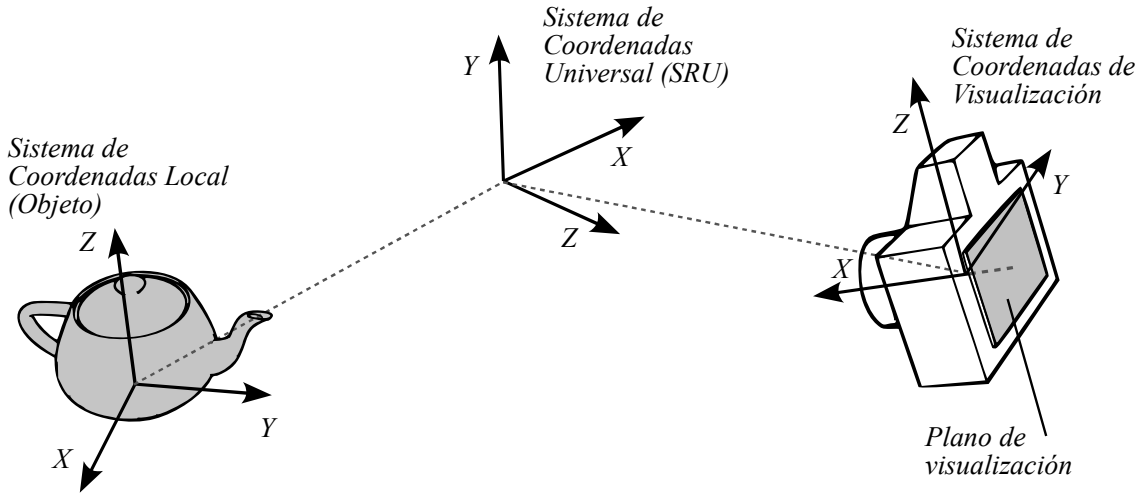


Figura 4.8: Sistema de coordenadas de visualización y su relación con otros sistemas de coordenadas de la escena.

4.2.1. Pipeline de Visualización

El proceso de visualizar una escena en 3D mediante gráficos por computador es similar al que se realiza cuando se toma una fotografía real. En primer lugar hay que situar el *trípode* con la cámara en un lugar del espacio, eligiendo así una posición de visualización. A continuación, rotamos la cámara eligiendo si la fotografía la tomaremos en vertical o en apaisado, y apuntando al motivo que queremos fotografiar. Finalmente, cuando disparamos la fotografía sólo una pequeña parte del mundo queda representado en la imagen 2D final (el resto de elementos son *recortados* y no aparecen en la imagen).

La Figura 4.7 muestra los pasos generales de procesamiento para la creación y transformación de una escena 3D a coordenadas dependientes del dispositivo de visualización (típicamente una pantalla con una determinada resolución).

En su tortuoso viaje hasta la pantalla, cada objeto 3D se transforma en diferentes sistemas de coordenadas. Originalmente como vimos en la Figura 4.8, un objeto tiene su propio *Sistema de Coordenadas Local* que nos definen las **Coordenadas de Modelo**, por lo que *desde su punto de vista* no está transformado. A los vértices de cada modelo se le aplican la denominada **Transformación de Modelado** para posicionarlo y orientarlo respecto del *Sistema de Coordenadas Universal*, obteniendo así las denominadas **Coordenadas Universales** o *Coordenadas del Mundo*. Como este sistema de coordenadas es único, tras aplicar la transformación de modelado a cada objeto, ahora todas las coordenadas estarán expresadas en el mismo espacio.

Perspectiva

La Realidad Aumentada emplea cámaras de visión para capturar imágenes del mundo real. Como estas cámaras se basan en modelos de proyección en perspectiva, estudiaremos este modelo en detalle en la sección 4.2.2.

Instancias

Gracias a la separación entre *Coordenadas de Modelo* y *Transformación de Modelado* podemos tener diferentes instancias de un mismo modelo para construir una escena a las que aplicamos diferentes transformaciones. Por ejemplo, para construir un templo romano tendríamos un único objeto de columna y varias *instancias* de la columna a las que hemos aplicado diferentes traslaciones.

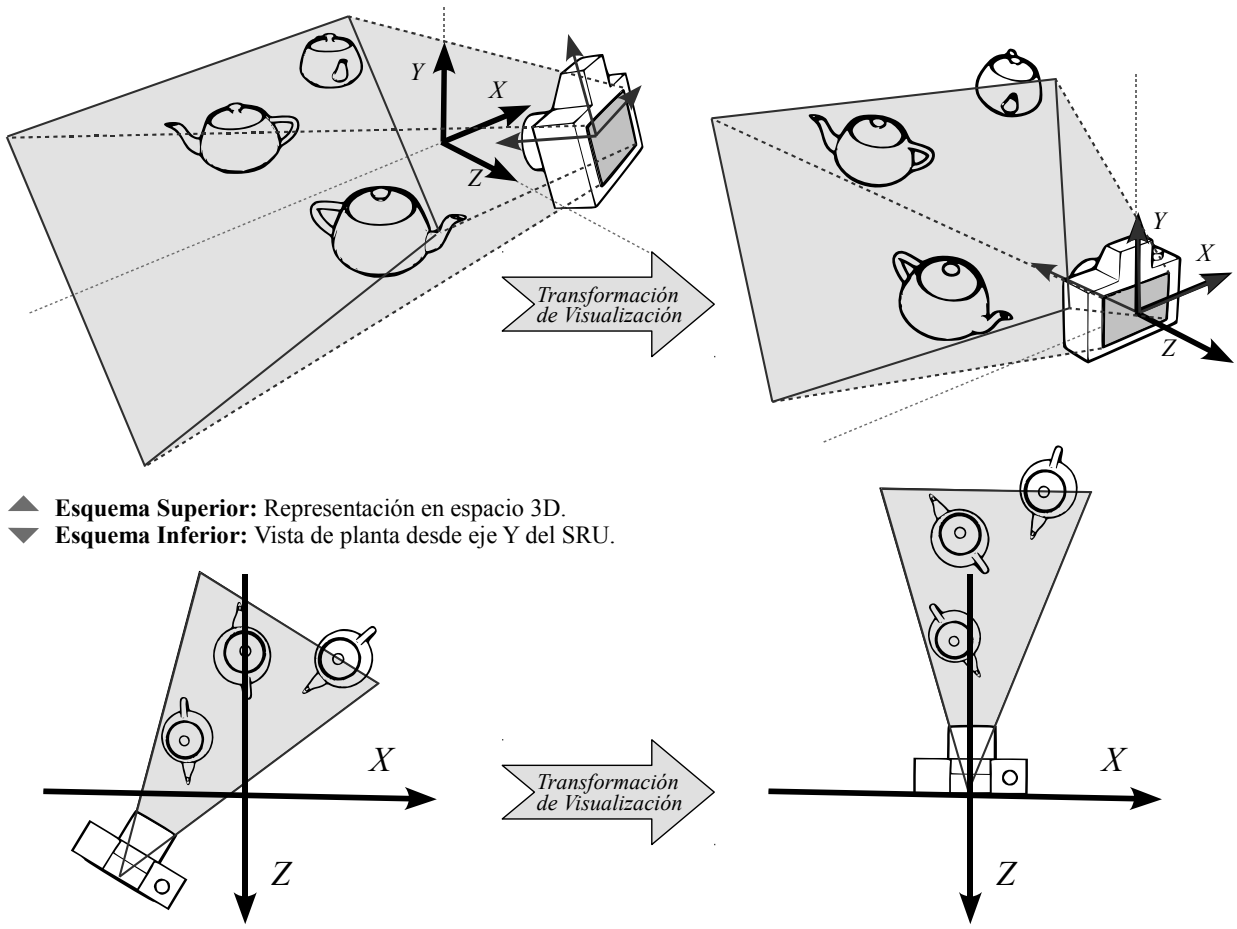


Figura 4.9: El usuario especifica la posición de la cámara (izquierda) que se transforma, junto con los objetos de la escena, para posicionarlos a partir del origen del SRU y mirando en la dirección negativa del eje Z. El área sombreada de la cámara se corresponde con el volumen de visualización de la misma (sólo los objetos que estén contenidos en esa pirámide serán representados).

La posición y orientación de la cámara nos determinará qué objetos aparecerán en la imagen final. Esta cámara tendrá igualmente unas coordenadas universales. El propósito de la **Transformación de Visualización** es posicionar la cámara en el origen del *SRU*, apuntando en la dirección negativa del eje Z y el eje Y hacia arriba. Obtenemos de este modo las **Coordenadas de Visualización** o *Coordenadas en Espacio Cámara* (ver Figura 4.9).

Habitualmente el pipeline contiene una etapa adicional intermedia que se denomina *Shading* que consiste en obtener la representación del material del objeto modelando las transformaciones en las fuentes de luz, utilizando los vectores normales a los puntos de la superficie, información de color, etc. Es conveniente en muchas ocasiones transformar las posiciones de estos elementos (fuentes de luz, cámara, ...) a otro espacio para realizar los cálculos.

La **Transformación de Proyección** convierte el **volumen de visualización** en un cubo unitario. Este *volumen de visualización* se define mediante planos de recorte 3D y define todos los elementos que serán visualizados. En la figura 4.9 se representa mediante el volumen sombreado. Existen multitud de métodos de proyección, aunque como hemos comentado anteriormente los más empleados son la ortográfica (o paralela) y la perspectiva. Ambas proyecciones pueden especificarse mediante matrices 4×4. Tras la proyección, el *volumen de visualización* se transforma en **Coordenadas Normalizadas** (obteniendo el *cubo unitario*), donde los modelos son proyectados de 3D a 2D. La coordenada Z se guarda habitualmente en un buffer de profundidad llamado *Z-Buffer*.

Únicamente los objetos que están dentro del *volumen de visualización* deben ser generados en la imagen final. Los objetos que están *totalmente* dentro del volumen de visualización serán copiados íntegramente a la siguiente etapa del *pipeline*. Sin embargo, aquellos que estén parcialmente incluidas necesitan ser recortadas, generando nuevos vértices en el límite del recorte. Esta operación de **Transformación de Recorte** se realiza automáticamente por el hardware de la tarjeta gráfica. En la Figura 4.10 se muestra un ejemplo simplificado de recorte.

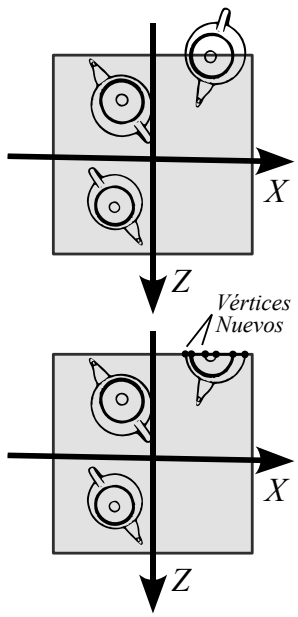


Figura 4.10: Los objetos que intersectan con los límites del *cubo unitario* (arriba) son recortados, añadiendo nuevos vértices. Los objetos que están totalmente dentro del cubo unitario se pasan directamente a la siguiente etapa. Los objetos que están totalmente fuera del *cubo unitario* son descartados.

Finalmente la **Transformación de Pantalla** toma como entrada las coordenadas de la etapa anterior y produce las denominadas **Coordenadas de Pantalla**, que ajustan las coordenadas x e y del cubo unitario a las dimensiones de ventana finales. En una posterior etapa de *rasterización* se emplean estas coordenadas de pantalla, junto con la información del *Z-Buffer* para calcular el color final de cada píxel de la ventana. Esta etapa de rasterización se divide normalmente en otra serie de etapas funciones para lograr mayor paralelismo.

En la siguiente sección estudiaremos en detalle cómo se realiza la proyección en perspectiva, por la que los objetos de la escena se proyectan en un volumen simple (el *cubo unitario*) antes de proceder al recorte y su posterior *rasterización*.

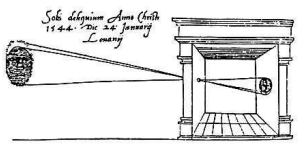


Figura 4.11: Descripción de la primera cámara estenopeica (*pinhole camera* ok *camera obscura*) por Gemma Frisius.

4.2.2. Proyección en Perspectiva

En la proyección en perspectiva, las líneas paralelas convergen en un punto, de forma que los objetos más cercanos se muestran de un tamaño mayor que los lejanos. Desde el 500aC, los griegos estudiaron el fenómeno que ocurría cuando la luz pasaba a través de pequeñas aberturas. La primera descripción de una cámara estenopeica se atribuye al atrónomo y matemático holandés *Gemma Frisius* que en 1545 publicó la primera descripción de una *cámara oscura* en la observación de un eclipse solar (ver Figura 4.11). En las cámaras esteneopeicas la luz pasa a través de un pequeño agujero para formar la imagen en la película fotosensible, que aparece invertida. Para que la imagen sea nítida, la abertura debe ser muy pequeña.

Siguiendo la misma idea y desplazando el plano de proyección delante del origen, tenemos el modelo general proyección en perspectiva. En la sección 4.1.1 estudiamos la representación homogénea, en la que habitualmente el parámetro $h = 1$. En otro caso, era necesario dividir cada componente del punto transformado por el valor del parámetro h, teniendo $p' = p/h$. En el resto de la sección, vamos a considerar que ya se ha realizado la *transformación de visualización* alineando la cámara y los objetos de la escena mirando en dirección al eje negativo Z, que el eje Y está apuntando hacia arriba y el eje X positivo a la derecha (como se muestra en la Figura 4.9).

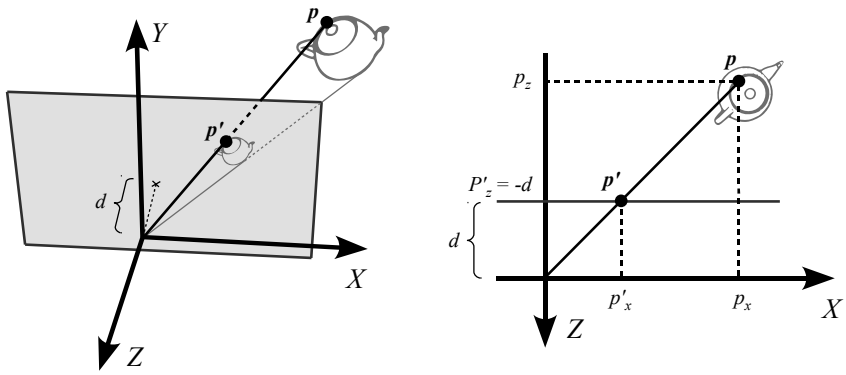


Figura 4.12: Modelo de proyección en perspectiva simple. El plano de proyección infinito está definido en $z = -d$, de forma que el punto p se proyecta sobre p'.

En la Figura 4.12 se muestra un ejemplo de proyección simple, en la que los vértices de los objetos del mundo se proyectan sobre un plano infinito situado en $z = -d$ (con $d > 0$). Suponiendo que la *transformación de visualización* se ha realizado, proyectamos un punto p sobre el plano de proyección, obteniendo un punto $p' = (p'_x, p'_y, -d)$.

Empleando triángulos semejantes (ver Figura 4.12 derecha), obtenemos las siguientes coordenadas:

$$\frac{p'_x}{p_x} = \frac{-d}{p_z} \quad \Leftrightarrow \quad p'_x = \frac{-d\,p_x}{p_z} \tag{4.10}$$

De igual forma obtenemos la coordenada $p'_y = -d\,p_y/p_z$, y $p'_z = -d$. Estas ecuaciones se pueden expresar fácilmente de forma matricial como se muestra en la siguiente expresión (siendo M_p la matriz de proyección en perspectiva).

$$p' = M_p\,p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{bmatrix} = \begin{bmatrix} -d\,p_x/p_z \\ -d\,p_y/p_z \\ -d \\ 1 \end{bmatrix} \tag{4.11}$$

El último paso de la ecuación 4.11 corresponde con la normalización de los componentes dividiendo por el parámetro homogéneo $h = -p_z/d$. De esta forma tenemos la matriz de proyección que nos *aplasta* los vértices de la geometría sobre el plano de proyección. Desafortunadamente esta operación no puede *deshacerse* (no tiene inversa). La geometría una vez *aplastada* ha perdido la información sobre su componente de profundidad. Es interesante obtener una trasnformación en perspectiva que proyecte los vértices sobre el *cubo unitario* descrito previamente (y que sí puede deshacerse).

De esta forma, definimos la *pirámide de visualización* o *frustum*, como la pirámide truncada por un plano paralelo a la base que define los objetos de la escena que serán representados. Esta *pirámide de visualización* queda definida por cuatro vértices que definen el plano de proyección (left l, right r, top t y bottom b), y dos distancias a los planos de recorte (near n y far f), como se representa en la Figura 4.13. El *ángulo de visión* de la cámara viene determinado por el ángulo que forman l y r (en horizontal) y entre t y b (en vertical).

La matriz que transforma el *frustum* en el cubo unitario viene dada por la expresión de la ecuación 4.12.

$$M_p = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.12}$$

Un efecto de utilizar este tipo de proyección es que el valor de profundidad normalizado no cambia linealmente con la entrada, sino que se va perdiendo precisión. Es recomendable situar los planos de recorte cercano y lejano (distancias n y f en la matriz) lo más juntos posibles para evitar errores de precisión en el *Z-Buffer* en distancias grandes.

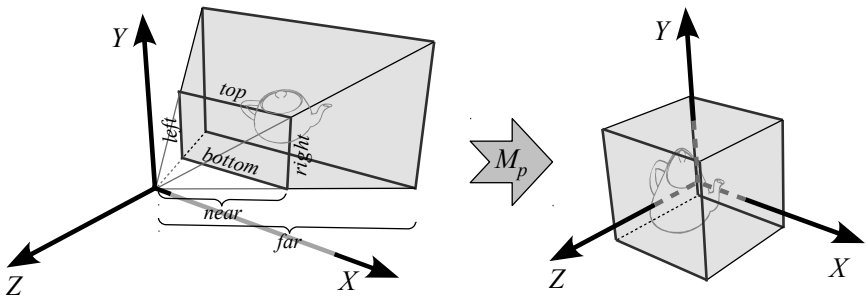


Figura 4.13: La matriz M_p se encarga de transformar la *pirámide de visualización* en el cubo unitario.

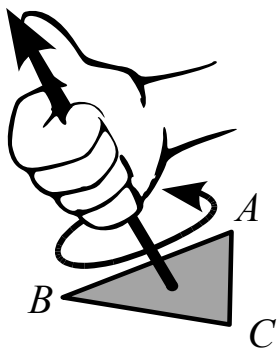


Figura 4.14: Regla de la mano derecha para calcular la normal de una superficie; los dedos *rodean* los vértices en el orden de definición y el pulgar indica el sentido del vector normal.

4.3. Ejercicios Propuestos

Se recomienda la realización de los ejercicios de esta sección en orden, ya que están relacionados y su complejidad es ascendente. Estos ejercicios propuestos se deben realizar sin implementación en el ordenador. 0mm

1. Dado el triángulo ABC, definido por $A = (1,1,3)$, $B = (3,1,3)$ y $C = (1,1,1)$, calcular el resultado de aplicar una traslación de -2 unidades en el eje X y 1 unidad en el eje Z, y después una rotación de $\pi/2$ radianes respecto del eje Z. Las transformaciones se realizan sobre el *SRU*.

2. Dado el cuadrado definido por los puntos $ABCD$, siendo $A = (1,0,3)$, $B = (3,0,3)$, $C = (3,0,1)$ y $D = (1,0,1)$, realizar un escalado del doble en el eje X con respecto a su centro geométrico.

3. ¿Cuáles serían las expresiones matriciales homogeneas correspondientes a la operación de reflejo (espejo) sobre los planos ortogonales $X = 0$, $Y = 0$ y $Z = 0$?. Deberá calcular una matriz para cada plano.

4. Dado el triángulo ABC, siendo $A = (1,-1,1)$, $B = (-1,-1,-1)$ y $C = (1,1,-1)$, desplace las cara poligonal 2 unidades en la dirección de su vector normal, suponiendo que el vector normal sale de la superficie según la regla de la mano derecha (ver Figura 4.14)[2].

[2] **Ayuda:** Utilice el producto escalar de dos vectores definidos por los lados del triángulo para calcular el vector normal de la superficie

Capítulo 5

OpenGL para Realidad Aumentada

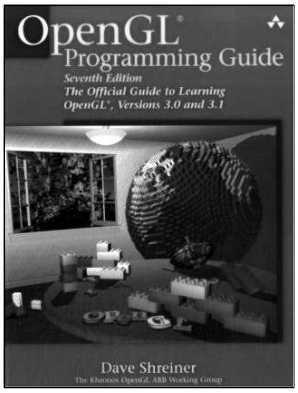


Figura 5.1: La última edición del libro oficial de OpenGL, la referencia imprescindible de más de 900 páginas.

En esta sección se introducirán los aspectos más relevantes de OpenGL. La visualización 3D de los ejemplos de este documento han sido generados con OpenGL. Entre otros aspectos, en este capítulo se estudiará la gestión de pilas de matrices y los diferentes modos de transformación de la API.

OpenGL es la biblioteca de programación gráfica más utilizada del mundo; desde videojuegos, simulación, CAD, visualización científica, y un largo etecétera de ámbitos de aplicación la configuran como la mejor alternativa en multitud de ocasiones. En esta sección se resumirán los aspectos básicos más relevantes para comenzar a utilizar OpenGL en aplicaciones de Realidad Aumentada.

Obviamente se dejarán gran cantidad de aspectos sin mencionar. Se recomienda al lector la guía oficial de OpenGL [23] (también conocido como *El Libro Rojo de OpenGL* para profundizar en esta potente biblioteca gráfica. Otra fantástica fuente de información, con ediciones anteriores del *Libro Rojo* es la página oficial de la biblioteca[1].

[1] http://www.opengl.org/

5.1. Sobre OpenGL

El propio nombre *OpenGL* indica que es una *Biblioteca para Gráficos Abierta*[2]. Una de las características que ha hecho de OpenGL una biblioteca tan famosa es que es independiente de la plataforma sobre la que se está ejecutando, además del Sistema Operativo. Esto implica que *alguien* tendrá que encargarse de abrir una ventana gráfica sobre la que OpenGL pueda dibujar los preciosos gráficos 3D a todo color.

Para facilitar el desarrollo de aplicaciones con OpenGL sin preocuparse de la creación de ventanas, gestión de eventos, etc... *M. Kilgard* creó GLUT, una biblioteca independiente de OpenGL (no forma parte de la distribución oficial de la biblioteca) que facilita el desarrollo de pequeños prototipos. Esta biblioteca auxiliar es igualmente multiplataforma. En este documento trabajaremos con FreeGLUT, la alternativa con licencia GPL totalmente compatible con GLUT. Si se desea mayor control sobre los eventos, y la posibilidad de extender las capacidades de la aplicación, se recomienda el estudio de otras APIs multiplataforma compatibles con OpenGL, como por ejemplo SDL[3]. Existen alternativas específicas para sistemas de ventanas concretos (ver Figura 5.2, como *GLX* para plataformas Unix, *AGL* para Macintosh o *WGL* para sistemas Windows.

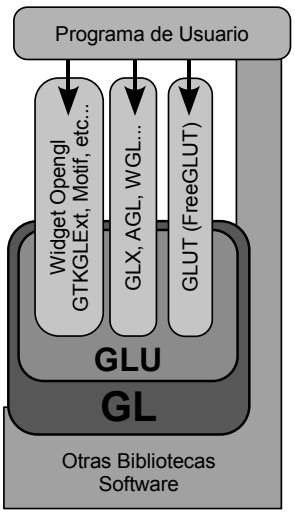


Figura 5.2: Relación entre los módulos principales de OpenGL.

De este modo, el núcleo principal de las biblioteca se encuentra en el módulo *GL* (ver Figura 5.2). En el módulo *GLU (OpenGL Utility Library* se encuentran funciones de uso común para el dibujo de diversos tipos de superficies (esferas, conos, cilindros, curvas...). Este módulo es parte oficial de la biblioteca.

Uno de los objetivos principales de OpenGL es la representación de imágenes de alta calidad a alta velocidad. OpenGL está diseñado para la realización de **aplicaciones interactivas**, como las de Realidad Aumentada. En la sección 3.3 estudiamos un sencillo ejemplo que redibujaba un objeto 3D en la posición de una marca. Para esto, se asociaba a la función del bucle principal de la aplicación una serie de llamadas a OpenGL que dibujaban en cada instante el objeto 3D correctamente alineado a la marca.

5.2. Modelo Conceptual

Las llamadas a funciones de OpenGL están diseñadas para aceptar diversos tipos de datos como entrada. El nombre de la función identifica además los argumentos que recibirá. Por ejemplo, en la Figura 5.3 se llama a una función para especificar un nuevo vértice en coordenadas homogéneas (4 parámetros), con tipo de datos *double* y en formato vector. Se definen tipos enumerados como redefinición de tipos básicos (como *GLfloat*, *GLint*, etc) para facilitar la compatibilidad con otras

[2]Las siglas de *GL* corresponden a *Graphics Library*
[3]http://www.libsdl.org/

plataformas. Es buena práctica utilizar estos tipos redefinidos si se planea compilar la aplicación en otros sistemas.

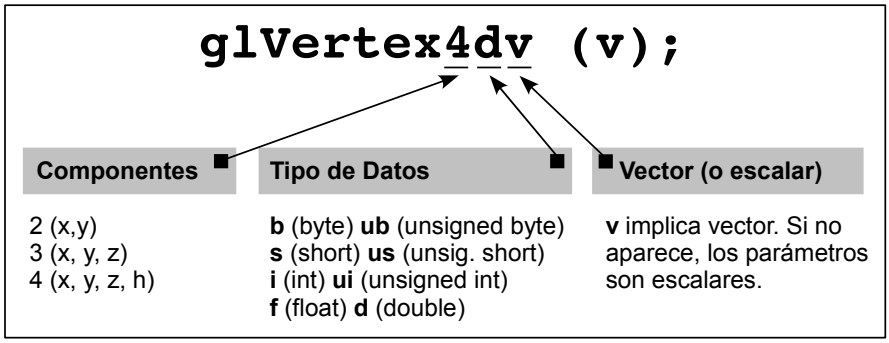


Figura 5.3: Prototipo general de llamada a función en OpenGL.

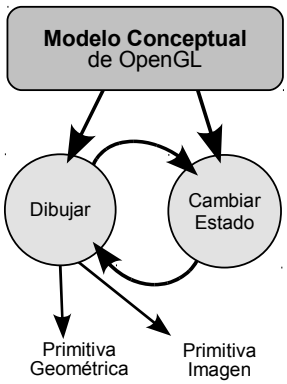


Figura 5.4: Modelo conceptual general de OpenGL.

El **Modelo Conceptual General** de OpenGL define dos operaciones básicas que el programador puede realizar en cada instante; 1) dibujar algún elemento o 2) cambiar el estado de cómo se dibujan los elementos. La primera operación de *dibujar algún elemento* tiene que ser a) una primitiva geométrica (puntos, líneas o polígonos) o b) una primitiva de imagen.

La Figura 5.4 resume el modelo conceptual de OpenGL. Así, la aplicación que utilice OpenGL será simplemente una colección de ciclos de cambio de estado y dibujado de elementos.

5.2.1. Cambio de Estado

Cambio de Estado

A modo de curiosidad: OpenGL cuenta con más de 400 llamadas a función que tienen que ver con el cambio del estado interno de la biblioteca.

La operación de **Cambiar el Estado** se encarga de inicializar las variables internas de OpenGL que definen cómo se dibujarán las primitivas. Este cambio de estado puede ser de mútiples tipos; desde cambiar el color de los vértices de la primitiva, establecer la posición de las luces, etc. Por ejemplo, cuando queremos dibujar el vértice de un polígono de color rojo, primero cambiamos el color del vértice con `glColor()` y después dibujamos la primitiva en ese nuevo estado con `glVertex()`.

Algunas de las formas más utilizadas para cambiar el estado de OpenGL son:

1. **Gestión de Vértices**: Algunas llamadas muy utilizadas cuando se trabaja con modelos poligonales son `glColor()` que ya fue utilizada en el programa del *Hola Mundo!* para establecer el color con el que se dibujarán los vértices[4], `glNormal()` para especificar

[4]Como ya vimos en el listado de la sección 3.3, los colores en OpenGL se especifican en punto flotante con valores entre 0 y 1. Las primeras tres componentes se corresponden con los canales *RGB*, y la cuarta es el valor de transparencia *Alpha*.

las normales que se utilizarán en la iluminación, o `glTexCoord()` para indicar coordendas de textura.

2. **Activación de Modos**: Mediante las llamadas a `glEnable` y `glDisable` se pueden activar o desactivar características internas de OpenGL. Por ejemplo, en la línea [28] del listado de la sección 3.3 se activa el Test de Profundidad que utiliza y actualiza el Z-Buffer. Este test no se utilizará hasta que de nuevo se cambia el *interruptor* activando esta funcionalidad en la línea [41].

3. **Características Especiales**: Existen multitud de características particulares de los elementos con los que se está trabajando. OpenGL define valores por defecto para los elementos con los que se está trabajando, que pueden ser cambiadas empleando llamadas a la API.

5.2.2. Dibujar Primitivas

La operación de **Dibujar Primitivas** requiere habitualmente que éstas se definan en coordenadas homogéneas. Todas las primitivas geométricas se especifican con vértices. El tipo de primitiva determina cómo se combinarán los vértices para formar la superfice poligional final. La creación de primitivas se realiza entre llamadas a `glBegin(PRIMITIVA)` y `glEnd()`, siendo `PRIMITIVA` alguna de las 10 primitivas básicas soportadas por OpenGL[5]. No obstante, el módulo GLU permite dibujar otras superficies más complejas (como cilindros, esferas, discos...), y con GLUT es posible dibujar algunos objetos simples. Como se puede ver, OpenGL no dispone de funciones para la carga de modelos poligonales creados con otras aplicaciones (como por ejemplo, en formato OBJ o MD3). Es responsabilidad del programador realizar esta carga y dibujarlos empleando las primitivas básicas anteriores.

Figura 5.5: Salida por pantalla del ejemplo del listado siguiente.

Una vez estudiados los cambios de estado y el dibujado de primitivas básicas, podemos modificar la función `draw` del listado de la sección 3.3 para que dibuje un cuadrado (primitiva `GL_QUADS`). Las líneas cambiadas del nuevo código se corresponden con el intervalo [15-24]. Especificamos la posición de cada vértice del cuadrado modificando entre medias el color (el estado interno). OpenGL se encargará de calcular la transición de color entre cada punto intermedio del cuadrado. Como la marca se definió de 120 mm de lado (ver línea [73] del listado siguiente, podemos dibujar el cuadrado justo encima de la marca posicionando los vértices en los extremos -60, 60.

[5]Las 10 primitivas básicas de OpenGL son `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINES`, `GL_LINE_LOOP`, `GL_POLYGON`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLES`, `GL_TRIANGLE_FAN`, `GL_QUADS` y `GL_QUAD_STRIP`

Listado 5.1: Ejemplo de cambio de estado y dibujo de primitivas

```
1  static void draw( void ) {
2    double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL
3
4    argDrawMode3D();              // Cambiamos el contexto a 3D
5    argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
6    glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
7    glEnable(GL_DEPTH_TEST);  glDepthFunc(GL_LEQUAL);
8
9    argConvGlpara(patt_trans, gl_para);   // Convertimos la matriz de
10   glMatrixMode(GL_MODELVIEW);           // la marca para ser usada
11   glLoadMatrixd(gl_para);               // por OpenGL
12
13   // Esta ultima parte del codigo es para dibujar el objeto 3D
14   // El tamano del patron definido es de 120 unidades de lado
15   glBegin(GL_QUADS);
16   glColor3f(1.0, 0.0, 0.0);
17   glVertex3f(60.0, 60.0, 0.0);
18   glColor3f(0.0, 1.0, 0.0);
19   glVertex3f(60.0, -60.0, 0.0);
20   glColor3f(0.0, 0.0, 1.0);
21   glVertex3f(-60.0, -60.0, 0.0);
22   glColor3f(1.0, 1.0, 1.0);
23   glVertex3f(-60.0, 60.0, 0.0);
24   glEnd();
25   glDisable(GL_DEPTH_TEST);
26 }
```

5.3. Pipeline de OpenGL

Como vimos en la sección 4.2.1, los elementos de la escena sufren diferentes transformaciones en el *pipeline* de gráficos 3D.

- **Transformación de Modelado**: En la que los modelos se posicionan en la escena y se obtienen las Coordenadas Universales.
- **Transformación de Visualización**: Donde se especifica la posición de la cámara y se mueven los objetos desde las coordenadas del mundo a las Coordenadas de Visualización (o coordenadas de cámara).
- **Transformación de Proyección**: Obteniendo Coordenadas Normalizadas en el cubo unitario.
- **Transformación de Recorte y de Pantalla**: Donde se obtienen, tras el recorte (o *clipping* de la geometría), las coordenadas 2D de la ventana en pantalla.

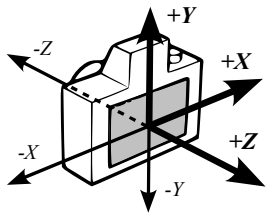


Figura 5.6: Matriz de Visualización por defecto en OpenGL.

OpenGL combina la Transformación de Modelado y la de Visualización en una Transformación llamada **"Modelview"**. De este modo OpenGL transforma directamente las Coordenadas Universales a Coordenadas de Visualización empleando la matriz *Modelview*. La posición inicial de la cámara en OpenGL sigue el convenio estudiado en el capítulo 4 y que se resume en la Figura 5.6.

5.3.1. Transformación de Visualización

La transformación de Visualización debe especificarse **antes** que ninguna otra transformación de Modelado. Esto es debido a que **OpenGL aplica las transformaciones en orden inverso**. De este modo, aplicando en el código las transformaciones de Visualización antes que las de Modelado nos aseguramos que ocurrirán después que las de Modelado. Por ejemplo, en el listado de la sección 3.3 primero posicionábamos la cámara virtual (línea 33) y luego posicionábamos los objetos (líneas 38–40).

Para comenzar con la definición de la Transformación de Visualización, es necesario *limpiar* la matriz de trabajo actual. OpenGL cuenta con una función que carga la matriz identidad como matriz actual `glLoadIdentity()`.

Matriz Identidad

La *Matriz Identidad* es una matriz 4x4 con valor 1 en la diagonal principal, y 0 en el resto de elementos. La multiplicación de esta matriz I por una matriz M cualquiera siempre obtiene como resultado M. Esto es necesario ya que OpenGL siempre multiplica las matrices que se le indican para modificar su estado interno.

Una vez hecho esto, podemos posicionar la cámara virtual de varias formas:

1. **glulookat**. La primera opción, y la más comunmente utilizada es mediante la función `gluLookAt`. Esta función recibe como parámetros un punto (*eye*) y dos vectores (*center* y *up*). El punto *eye* es el punto donde se encuentra la cámara en el espacio y mediante los vectores libres *center* y *up* orientamos hacia dónde mira la cámara (ver Figura 5.7).
2. **Traslación y Rotación**. Otra opción es especificar *manualmente* una secuencia de traslaciones y rotaciones para posicionar la cámara (mediante las funciones `glTraslate()` y `glRotate()`, que serán estudiadas más adelante.
3. **Carga de Matriz**. La última opción, que es la utilizada en aplicaciones de Realidad Aumentada, es la carga de la matriz de Visualización calculada *externamente*. En el listado de la sección 3.3, esto se realizaba mediante la llamada a la función `glLoadMatrixd()` (línea 33) cargando la matriz que previamente había calculado ARToolKit y convertido al formato de OpenGL (línea 31).

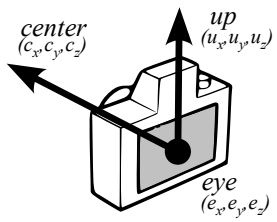


Figura 5.7: Parámetros de la función glulookat.

5.3.2. Transformación de Modelado

Las transformaciones de modelado nos permiten modificar los objetos de la escena. Existen tres operaciones básicas de modelado que implementa OpenGL con llamadas a funciones. No obstante, puede especificarse cualquier operación aplicando una matriz definida por el usuario.

- **Traslación**. El objeto se mueve a lo largo de un vector. Esta operación se realiza mediante la llamada a `glTranslate(x,y,z)`.
- **Rotación**. El objeto se rota en el eje definido por un vector. Esta operación se realiza mediante la llamada a `glRotate(`α`,x,y,z)`,

siendo α el ángulo de rotación en grados sexagesimales (en sentido contrario a las agujas del reloj).

- **Escalado**. El objeto se escala un determinado valor en cada eje. Se realiza mediante `glScale(x,y,z)`.

5.3.3. Transformación de Proyección

Como hemos visto, las transformaciones de proyección definen el volumen de visualización y los planos de recorte. OpenGL soporta dos modelos básicos de proyección; la proyección ortográfica y la proyección en perspectiva.

El núcleo de OpenGL define una función para definir la pirámide de visualización (o *frustum*) mediante la llamada a `glFrustum()`. Esta función require los seis parámetros (t, b, r, l, f y n) estudiados en la sección 4.2.2.

Otro modo de especificar la transformación es mediante la función de GLU `gluPerspective(fov, aspect, near, far)`. En esta función, *far* y *near* son las distancias de los planos de recorte (igual que los parámetros f y n del *frustum*. *fov* especifica en grados sexagesimales el ángulo en el eje Y de la escena que es visible para el usuario, y *aspect* indica la relación de aspecto de la pantalla (ancho/alto).

5.3.4. Matrices

Sobre el uso de Matrices

Como el convenio en C y C++ de definición de matrices bidimensionales es ordenados por filas, suele ser una fuente de errores habitual definir la matriz como un array bidimensional. Así, para acceder al elemento superior derecho de la matriz, tendríamos que acceder al `matriz[3][0]` según la notación OpenGL. Para evitar errores, se recomienda definir el array como unidimensional de 16 elementos `GLfloat matriz[16]`.

OpenGL utiliza matrices 4x4 para representar todas sus transformaciones geométricas. Las matrices emplean coordenadas homogéneas, como se estudió en la sección 4.1.1. A diferencia de la notación matemática estándar, OpenGL especifica por defecto las matrices por columnas, por lo que si queremos cargar nuestras propias matrices de transformación debemos tener en cuenta que el orden de elementos en OpenGL es el siguiente:

$$\begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix} \tag{5.1}$$

OpenGL internamente maneja *pilas de matrices*, de forma que únicamente la matriz de la cima de cada pila es la que se está utilizando en un momento determinado. Hay cuatro pilas de matrices en OpenGL:

1. Pila *Modelview* (`GL_MODELVIEW`). Esta pila contiene las matrices de Transformación de modelado y visualización. Tiene un tamaño mínimo de 32 matrices (aunque, dependiendo del sistema puede haber más disponibles).

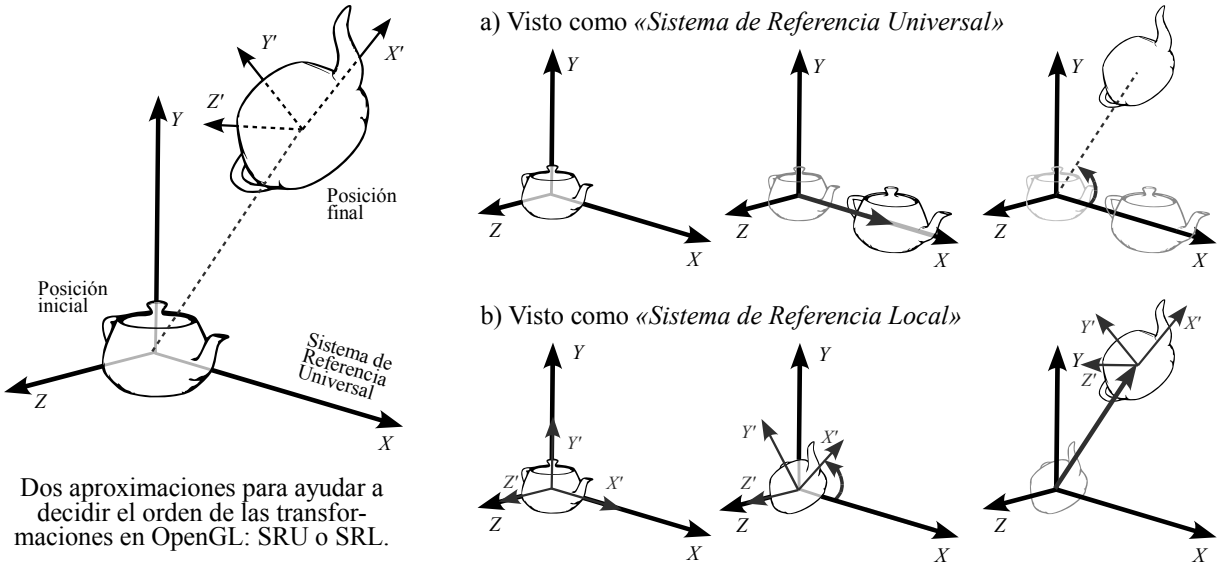


Figura 5.8: Cómo decidir el orden de las operaciones en OpenGL.

2. Pila *Projection* (`GL_PROJECTION`). Esta pila contiene las matrices de proyección. Tiene un tamaño mínimo de 2 elementos.

3. Pila *Color* (`GL_PROJECTION`). Utilizada para modificar los colores.

4. Pila *Texture* (`GL_TEXTURE`). Estas matrices se emplean para transformar las coordenadas de textura.

Es posible cambiar la pila sobre la que especificaremos las transformaciones empleando la llamada a `glMatrixMode()`. Por ejemplo, para utilizar la pila de *Modelview* utilizaríamos `glMatrixMode(GL_MODELVIEW)`.

El uso de *pilas* de matrices nos facilita la construcción de modelos jerárquicos, donde es posible utilizar modelos simples y ensamblarlos para construir modelos más complejos. Por ejemplo, si queremos dibujar un coche, tendríamos que dibujar las 4 ruedas con su posición relativa al chasis del vehículo. Así, dibujaríamos el chasis, y luego desplazándonos *cada vez desde el centro del coche* dibujaríamos cada una de las ruedas. Esta operación de *"volver al centro del coche"* se realiza fácilmente empleando pilas de matrices.

Recordemos que únicamente trabajamos con la matriz que está en la cima de la pila. La función `glPushMatrix()` añade una copia de la matriz de trabajo actual a la parte superior de la pila. Por su parte, la función `glPopMatrix()` elimina la matriz superior de la pila, descartando su información. La siguiente matriz de la pila pasará a ser la matriz *activa*. El efecto de esta función es el de "volver" al último punto que guardamos en la pila.

Supongamos ahora que queremos rotar un objeto 45º respecto del eje Z y trasladarlo 5 unidades respecto del eje X, obteniendo una determinada posición final (ver Figura 5.8 izquierda). El objeto inicialmente está en el origen del *SRU*. ¿En qué orden debemos aplicar las transformaciones de OpenGL? Como la transformación es de modelo, tendremos que aplicarla sobre la pila de matrices *Modelview*, con el siguiente código resultante:

Listado 5.2: Ejemplo de transformaciones

```
1 glMatrixMode(GL_MODELVIEW);
2 glLoadIdentity();
3 glRotatef(45,0,0,1);
4 glTraslatef(5,0,0);
5 dibujar_objeto();
```

El código anterior dibuja el objeto en la posición deseada, pero ¿cómo hemos llegado a ese código y no hemos intercambiado las instrucciones de las líneas 3 y 4?. Existen dos formas de imaginarnos cómo se realiza el dibujado que nos puede ayudar a plantear el código fuente. Ambas formas son únicamente aproximaciones conceptuales, ya que el resultado en código debe ser exactamente el mismo.

- **Idea de Sistema de Referencia Universal Fijo**. La composición de movimientos aparece en orden inverso al que aparece en el código fuente. Esta idea es como ocurre realmente en OpenGL. Las transformaciones se aplican siempre respecto del *SRU*, y en orden inverso a como se indica en el código fuente. De este modo, la primera transformación que ocurrirá será la traslación (línea 4) y después la rotación *respecto del origen del sistema de referencia universal* (línea 3). El resultado puede verse en la secuencia de la Figura 5.8 a).
- **Idea del Sistema de Referencia Local**. También podemos imaginar que cada objeto tiene un sistema de referencia local *interno* al objeto que va cambiando. La composición se realiza en el mismo orden que aparece en el código fuente, y siempre respecto de ese sistema de referencia local. De esta forma, como se muestra en la secuencia de la Figura 5.8 b), el objeto primero rota (línea 3) por lo que su sistema de referencia local queda rotado respecto del *SRU*, y respecto de ese sistema de referencia local, posteriormente lo trasladamos 5 unidades respecto del eje X' (local).

Como hemos visto, es posible además cargar matrices de transformación definidas por nuestros propios métodos (como en el caso de ARToolKit). Esto se realiza con la llamada a función `glLoadMatrix()`. Cuando se llama a esta función se reemplaza la cima de la pila de matrices activa con el contenido de la matriz que se pasa como argumento.

Si nos interesa es multiplicar una matriz definida en nuestros métodos por el contenido de la cima de la pila de matrices, podemos utili-

zar la función `glMultMatrix` que postmultiplica la matriz que pasamos como argumento por la matriz de la cima de la pila.

5.3.5. Dos ejemplos de transformaciones jerárquicas

Ejemplo Planetario

Ejemplo sencillo para afianzar el orden de las transformaciones.

Veamos a continuación un ejemplo sencillo que utiliza transformaciones jerárquicas. Volvemos a utilizar el código del *Hola Mundo!* como base. En el listado siguiente sólo se muestra el código modificado.

Listado 5.3: Sistema Planetario

```
1  // ==== Definicion de constantes y variables globales
       ===============
2  long   hours = 0;   // Horas transcurridas (para calculo
       rotaciones)
3  // ======== draw
       ====================================================
4  static void draw( void ) {
5    double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL
6    GLfloat mat_ambient[]     = {0.0, 0.0, 0.0, 1.0};
7    GLfloat light_position[]  = {100.0,-200.0,200.0,0.0};
8    float RotEarth = 0.0;     // Movimiento de traslacion de la
         tierra
9    float RotEarthDay = 0.0;  // Movimiento de rotacion de la tierra
10
11   argDrawMode3D();              // Cambiamos el contexto a 3D
12   argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
13   glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
14   glEnable(GL_DEPTH_TEST);
15   glDepthFunc(GL_LEQUAL);
16
17   argConvGlpara(patt_trans, gl_para);   // Convertimos la matriz de
18   glMatrixMode(GL_MODELVIEW);           // la marca para ser usada
19   glLoadMatrixd(gl_para);               // por OpenGL
20
21   // Esta ultima parte del codigo es para dibujar el objeto 3D
22   glEnable(GL_LIGHTING);  glEnable(GL_LIGHT0);
23   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
24
25   hours++;                      // Incrementamos la variable global
26   RotEarthDay = (hours % 24) * (360/24.0);   // Cuanto rota x hora
27   RotEarth = (hours / 24.0) * (360 / 365.0) * 10;  // x10 rapido!
28
29   mat_ambient[0] = 1.0; mat_ambient[1] = 0.9; mat_ambient[2] = 0.0;
30   glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
31   glTranslatef(0,0,80);    // Dibujamos 80mm flotando sobre la
         marca
32   glutWireSphere (80, 32, 32); // Sol (Radio 8cm y 32 divisiones)
33   glRotatef (-RotEarth, 0.0, 0.0, 1.0);
34   glTranslatef(150, 0.0, 0.0);   // Dejamos 15cm entre sol y tierra
35   glRotatef (-RotEarthDay, 0.0, 0.0, 1.0);
36   mat_ambient[0] = 0.1; mat_ambient[1] = 0.1; mat_ambient[2] = 1.0;
37   glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
38   glutWireSphere (20, 16, 16); // Tierra (Radio 2cm y 16
         divisiones)
39   glDisable(GL_DEPTH_TEST);
40 }
```

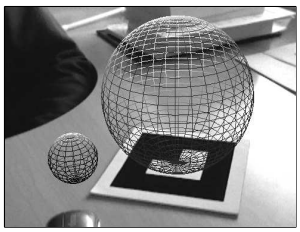


Figura 5.9: Salida por pantalla del ejemplo del planetario (ver listado del planetario).

Como puede verse, se ha incluido una variable global `hours` que se incrementa cada vez que se llama a la función `draw` (es decir, cada vez que se detecta la marca). Esa variable modela el paso de las horas, de forma que la traslación y rotación de la *Tierra* se calculará a partir del número de horas que han pasado (líneas [26] y [27]). En la simulación se ha acelerado 10 veces el movimiento de traslación para que se vea más claramente.

Empleando la *Idea de Sistema de Referencia Local* podemos pensar que desplazamos el sistema de referencia para dibujar el sol. En la línea [31] nos desplazamos 80 unidades en $+Z$ (nos elevamos de la marca), para dibujar una esfera alámbrica en la línea [32] (el primer argumento es el radio, y los dos siguientes son el número de *rebanadas* en que se dibujará la esfera).

En ese punto dibujamos la primera esfera correspondiente al Sol. Hecho esto, rotamos los grados correspondientes al movimiento de traslación de la tierra (línea [33]) y nos desplazamos 150 unidades respecto del eje X local del objeto (línea [34]). Antes de dibujar la *Tierra* tendremos que realizar el movimiento de rotación de la tierra (línea [35]). Finalmente dibujamos la esfera en [38].

Ejemplo Brazo Robot

Un segundo ejemplo que utiliza las funciones de añadir y quitar elementos de la pila de matrices.

Veamos ahora un segundo ejemplo que utiliza `glPushMatrix()` y `glPopMatrix()` para dibujar un brazo robótico sencillo. El ejemplo simplemente emplea dos cubos (convenientemente escalados) para dibujar la estructura jerárquica de la figura 5.10.

Figura 5.10: Salida por pantalla del ejemplo del robot (ver listado del brazo robótico).

En este ejemplo, se asocian manejadores de *callback* para los eventos de teclado en [62], que se corresponden con la función `keyboard` (en [5-14]). Esta función simplemente incrementa o decrementa los ángulos de rotación de las dos articulaciones del robot, que están definidas como variables globales en [2] y [3].

La parte más *interesante* del ejemplo se encuentra definido en la función `draw` en las líneas [36-51].

Listado 5.4: Brazo Robótico

```
1  // ==== Definicion de constantes y variables globales
       ===============
2  GLfloat rot1 = 0.0;        // Rotacion para el hombro del robot
3  GLfloat rot2 = 0.0;        // Rotacion para el codo del robot
4  // ======== keyboard
       ================================================
5  static void keyboard(unsigned char key, int x, int y) {
6    switch (key) {
7    case 0x1B: case 'Q': case 'q':
8      cleanup(); break;
9    case 'a': case'A': rot1+=2; break;
10   case 'z': case'Z': rot1-=2; break;
11   case 's': case'S': rot2+=2; break;
12   case 'x': case'X': rot2-=2; break;
13   }
14 }
15 // ======== draw
       ================================================
```

```
16 static void draw( void ) {
17   double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL
18   GLfloat light_position[]  = {100.0,-200.0,200.0,0.0};
19   GLfloat matGreen[] = {0.0, 1.0, 0.0, 1.0};
20   GLfloat matBlue[] = {0.0, 0.0, 1.0, 1.0};
21   GLfloat linksize = 80;  // Tamano del enlace
22
23   argDrawMode3D();              // Cambiamos el contexto a 3D
24   argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
25   glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
26   glEnable(GL_DEPTH_TEST);
27   glDepthFunc(GL_LEQUAL);
28
29   argConvGlpara(patt_trans, gl_para);   // Convertimos la matriz de
30   glMatrixMode(GL_MODELVIEW);           // la marca para ser usada
31   glLoadMatrixd(gl_para);               // por OpenGL
32
33   glEnable(GL_LIGHTING);  glEnable(GL_LIGHT0);
34   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
35
36   glRotatef(rot1, 0.0, 1.0, 0.0);
37   glTranslatef(linksize/2.0, 0.0, 0.0);  // Nos vamos a la mitad
38   glPushMatrix();
39   glScalef (1.0, 0.7, 0.1);
40   glMaterialfv(GL_FRONT, GL_DIFFUSE, matGreen);
41   glutSolidCube(linksize);
42   glPopMatrix();
43
44   glTranslatef(linksize/2.0,0.0,0.0);  // Avanzamos al extremo
45   glRotatef(rot2, 0.0, 1.0, 0.0);
46   glTranslatef(linksize/2.0,0.0,0.0);  // Hasta la mitad
47   glPushMatrix();
48   glScalef (1.0, 0.7, 0.1);
49   glMaterialfv(GL_FRONT, GL_DIFFUSE, matBlue);
50   glutSolidCube(linksize);
51   glPopMatrix();
52
53   glDisable(GL_DEPTH_TEST);
54 }
55
56 // ======== Main
       ==================================================
57 int main(int argc, char **argv) {
58   glutInit(&argc, argv);     // Creamos la ventana OpenGL con Glut
59   init();                    // Llamada a nuestra funcion de inicio
60
61   arVideoCapStart();         // Creamos un hilo para captura de
         video
62   argMainLoop( NULL, keyboard, mainLoop );  // Asociamos
         callbacks...
63   return (0);
64 }
```

Aplicamos de nuevo la idea de trabajar en un sistema de referencia local que se desplaza con los objetos según los vamos dibujando. De este modo nos posicionaremos en la mitad del trayecto para dibujar el cubo *escalado*. El cubo siempre se dibuja en el centro del sistema de referencia local (dejando mitad y mitad del cubo en el lado positivo y negativo de cada eje). Por tanto, para que el cubo *rote* respecto del extremo tenemos que rotar primero (como en la línea 36) y luego des-

plazarnos hasta la mitad del trayecto (línea 37), dibujar y recorrer la otra mitad 44 antes de dibujar la segunda parte del robot.

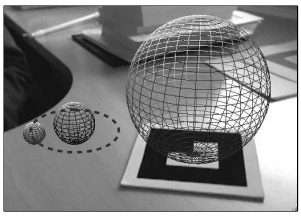


Figura 5.11: Ejemplo de salida del ejercicio propuesto.

5.4. Ejercicios Propuestos

Se recomienda la realización de los ejercicios de esta sección en orden, ya que están relacionados y su complejidad es ascendente.

1. Modifique el ejemplo del listado del planetario para que dibuje una esfera de color blanco que representará a la *Luna*, de radio 1cm y separada 4cm del centro de la *Tierra* (ver Figura 5.11). Este objeto tendrá una rotación completa alrededor de la *Tierra* cada 2.7 días (10 veces más rápido que la rotación real de la *Luna* sobre la *Tierra*. Supondremos que la luna no cuenta con rotación interna[6].

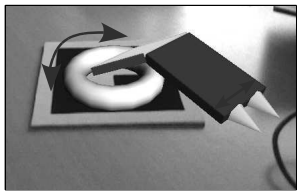


Figura 5.12: Ejemplo de salida del ejercicio del robot.

2. Modifique el ejemplo del listado del brazo robótico para que una base (añadida con un toroide `glutSolidTorus`) permita rotar el brazo robótico en el eje Z mediante las teclas d y c. Además, añada el códdigo para que el extremo cuente con unas pinzas (creadas con `glutSolidCone`) que se abran y cierren empleando las teclas f y v, tal y como se muestra en la Figura 5.12.

[6]Nota: Para la resolución de este ejercicio es recomendable utilizar las funciones de `glPushMatrix()` y `glPopMatrix()` para volver al punto donde se dibujó la *Tierra* antes de aplicar su rotación interna.

Capítulo 6

Uso Avanzado de ARToolKit

En el capítulo 3 se introdujeron algunas características básicas de uso de ARToolKit. En este capítulo se estudiará el uso de algunas capacidades avanzadas, como el uso del histórico de percepciones para estabilizar la captura, el uso de las capacidades multipatrón, y estudiaremos en detalle los sistemas de coordenadas asociados a las marcas.

En los ejemplos del capítulo 3 hemos utilizado algunas de las capacidades básicas de la biblioteca. El lector observador se habrá dado cuenta que en algunas ocasiones, especialmente cuando la marca está situada a una distancia considerable de la cámara, la detección empieza a fallar, mostrando un comportamiento *tembloroso*. En la siguiente sección estudiaremos una característica basada en un vector histórico de percepciones que estabiliza enormemente el registro y el tracking de las marcas.

6.1. Histórico de Percepciones

ARToolKit incorpora una función de tratamiento del histórico de percepciones para estabilizar el tracking. Este histórico se implementa en una función alternativa a `arGetTransMat` que, en realidad, utiliza únicamente la percepción anterior, llamada `arGetTransMatCont`. Mediante el uso de esta función se elimina gran parte del efecto de

registro *tembloroso* (ver Figura 6.1). En esta figura se muestran diferentes valores de la componente de la posición en X en 40 frames de captura. Lo interesante de esta figura no son los valores de estas posiciones (intencionadamente distintos), sino la *forma* de la trayectoria. Empleando el histórico, la trayectoria resultante es mucho más suave.

En el listado siguiente se muestra un ejemplo de utilización de esta función de histórico. La función de *callback* de teclado (líneas 8-16) permite activar el uso del histórico (mediante la tecla *h*).

Como el histórico requiere como parámetro la percepción anterior, no podrá utilizarse hasta que no dispongamos (al menos) de una percepción. Por esta razón, es necesario tener en cuenta este aspecto para llamar a la función de histórico `arGetTransMatCont` (cuando tengamos una percepción) o a la función sin histórico `arGetTransMat` la primera vez. Para esto, se utiliza otra variable llamada `useCont` (línea 4), que nos indica si ya podemos utilizar la función con histórico o no. Esta variable de comportamiento booleano se tendrá que poner a *false* (valor 0) cuando no se detecte la marca (línea 65).

Histórico y precisión

La utilización del histórico suaviza la captura y consigue mayor estabilidad, aunque el resultado del tracking cuenta con menor precisión.

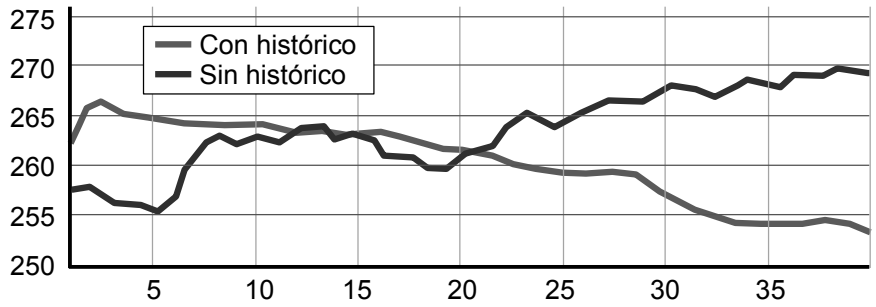


Figura 6.1: Comparativa de la trayectoria (en eje X) de 40 frames sosteniendo la marca manualmente activando el uso del histórico de percepciones. Se puede comprobar cómo la gráfica en el caso de activar el uso del histórico es mucho más suave.

Ver Ejercicios Propuestos

Aunque ARToolKit implementa esta versión del histórico, es muy interesante implementar nuestra propia versión del vector histórico, muy útil en multitud de ocasiones. Se propone, tras estudiar este ejemplo, resolver el primer ejercicio propuesto de este capítulo.

Listado 6.1: Uso del histórico de percepciones

```
1  // ==== Definicion de constantes y variables globales =============
2  int    patt_id;            // Identificador unico de la marca
3  double patt_trans[3][4];   // Matriz de transformacion de la marca
4  int    useCont = 0;        // Inicialmente no puede usar historico!
5  int    contAct = 0;        // Indica si queremos usar el historico
6
7  // ======== keyboard ============================================
8  static void keyboard(unsigned char key, int x, int y) {
9    switch (key) {
10   case 'H': case 'h':
11     if (contAct) {contAct = 0; printf("Historico Desactivado\n");}
12     else {contAct = 1; printf("Historico Activado\n");} break;
13   case 0x1B: case 'Q': case 'q':
14     cleanup(); exit(1); break;
15   }
16 }
17
```

```
18 // ======== mainLoop ============================================
19 static void mainLoop(void) {
20   ARUint8 *dataPtr;
21   ARMarkerInfo *marker_info;
22   int marker_num, j, k;
23
24   double p_width     = 120.0;        // Ancho del patron (marca)
25   double p_center[2] = {0.0, 0.0};   // Centro del patron (marca)
26
27   // Capturamos un frame de la camara de video
28   if((dataPtr = (ARUint8 *)arVideoGetImage()) == NULL) {
29     // Si devuelve NULL es porque no hay un nuevo frame listo
30     arUtilSleep(2);  return;  // Dormimos el hilo 2ms y salimos
31   }
32
33   argDrawMode2D();
34   argDispImage(dataPtr, 0,0);    // Dibujamos lo que ve la camara
35
36   // Detectamos la marca en el frame capturado (return -1 si error)
37   if(arDetectMarker(dataPtr, 100, &marker_info, &marker_num) < 0) {
38     cleanup(); exit(0);   // Si devolvio -1, salimos del programa!
39   }
40
41   arVideoCapNext();      // Frame pintado y analizado... A por
         otro!
42
43   // Vemos donde detecta el patron con mayor fiabilidad
44   for(j = 0, k = -1; j < marker_num; j++) {
45     if(patt_id == marker_info[j].id) {
46       if (k == -1) k = j;
47       else if(marker_info[k].cf < marker_info[j].cf) k = j;
48     }
49   }
50
51   if(k != -1) {   // Si ha detectado el patron en algun sitio...
52     // Obtener transformacion relativa entre marca y la camara real
53     if (useCont && contAct) {
54       arGetTransMatCont(&marker_info[k], patt_trans, p_center,
55             p_width, patt_trans);
56       printf ("Usando historico!!!\n");
57     }
58     else {
59       useCont = 1;  // En la siguiente iteracion lo podemos usar!
60       arGetTransMat(&marker_info[k], p_center, p_width, patt_trans)
             ;
61       printf ("Sin historico...\n");
62     }
63     draw();       // Dibujamos los objetos de la escena
64   } else {
65     useCont = 0; printf ("Reset Historico (fallo de deteccion)\n");
66   }
67
68   argSwapBuffers(); // Cambiamos el buffer con lo que tenga
         dibujado
69 }
```

De este modo, si el usuario activó el uso del histórico (y ya tenemos al menos una percepción previa de la marca, línea 53), utilizamos la función con histórico 54. En otro caso, llamaremos a la función de detección sin histórico 60 y activaremos el uso del historico para la siguiente llamada 59.

6.2. Utilización de varios patrones

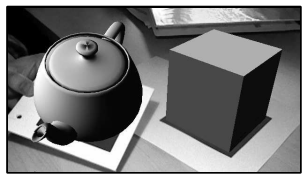


Figura 6.2: Asociación de diferentes modelos a varios patrones.

En muchas ocasiones querremos trabajar con más de un patrón. Un ejemplo típico es asociar cada patrón con un objeto 3D, o utilizarlos (como veremos en la siguiente sección) como mecanismos de interacción para realizar otras operaciones.

ARToolKit no proporciona ningún mecanismo directo para trabajar con varios patrones, por lo que es responsabilidad del programador mantener alguna estructura para guardar las matrices asociadas a cada marca detectada. A continuación veremos un sencillo ejemplo que define un tipo de datos propio `TObject` para guardar este tipo de información.

Listado 6.2: Utilización de varios patrones

```
1  // ==== Definicion de estructuras ====
2  struct TObject{
3    int id;                       // Identificador del patron
4    int visible;                  // Es visible el objeto?
5    double width;                 // Ancho del patron
6    double center[2];             // Centro del patron
7    double patt_trans[3][4];      // Matriz asociada al patron
8    void (* drawme)(void);        // Puntero a funcion drawme
9  };
10
11 struct TObject *objects = NULL;
12 int nobjects = 0;
13 /* La funcion addObject se encarga de cargar un patron (con ruta
       especificada en p), con un ancho de w, centro en c, y un
       puntero a funcion de dibujado drawme. Los patrones se guardan
       en una listado de objetos (objects) de tipo TObject. */
14 // ==== addObject (Anade objeto a la lista de objetos) ===========
15 void addObject(char *p, double w, double c[2], void (*drawme)(void)
       )
16 {
17   int pattid;
18
19   if((pattid=arLoadPatt(p)) < 0) print_error ("Error en patron\n");
20   nobjects++;
21   objects = (struct TObject *)
22     realloc(objects, sizeof(struct TObject)*nobjects);
23
24   objects[nobjects-1].id = pattid;
25   objects[nobjects-1].width = w;
26   objects[nobjects-1].center[0] = c[0];
27   objects[nobjects-1].center[1] = c[1];
28   objects[nobjects-1].drawme = drawme;
29 }
30 // ==== draw****** (Dibujado especifico de cada objeto) ==========
```

```
31 void drawteapot(void) {
32   GLfloat material[]     = {0.0, 0.0, 1.0, 1.0};
33   glMaterialfv(GL_FRONT, GL_AMBIENT, material);
34   glTranslatef(0.0, 0.0, 60.0);
35   glRotatef(90.0, 1.0, 0.0, 0.0);
36   glutSolidTeapot(80.0);
37 }
38
39 void drawcube(void) {
40   GLfloat material[]     = {1.0, 0.0, 0.0, 1.0};
41   glMaterialfv(GL_FRONT, GL_AMBIENT, material);
42   glTranslatef(0.0, 0.0, 40.0);
43   glutSolidCube(80.0);
44 }
45
46 // ======== cleanup ================================================
47 static void cleanup(void) {   // Libera recursos al salir ...
48   arVideoCapStop();  arVideoClose();  argCleanup();
49   free(objects);  // Liberamos la memoria de la lista de objetos!
50   exit(0);
51 }
52
53 // ======== draw ====================================================
54 void draw( void ) {
55   double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL
56   GLfloat light_position[]  = {100.0,-200.0,200.0,0.0};
57   int i;
58
59   argDrawMode3D();              // Cambiamos el contexto a 3D
60   argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
61   glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
62   glEnable(GL_DEPTH_TEST);
63   glDepthFunc(GL_LEQUAL);
64
65   for (i=0; i<nobjects; i++) {
66     if (objects[i].visible) {   // Si el objeto es visible
67       argConvGlpara(objects[i].patt_trans, gl_para);
68       glMatrixMode(GL_MODELVIEW);
69       glLoadMatrixd(gl_para);   // Cargamos su matriz de transf.
70
71       // La parte de iluminacion podria ser especifica de cada
72       // objeto, aunque en este ejemplo es general para todos.
73       glEnable(GL_LIGHTING);  glEnable(GL_LIGHT0);
74       glLightfv(GL_LIGHT0, GL_POSITION, light_position);
75       objects[i].drawme();      // Llamamos a su funcion de dibujar
76     }
77   }
78   glDisable(GL_DEPTH_TEST);
79 }
80
81 // ======== init ====================================================
82 static void init( void ) {
83   ARParam  wparam, cparam;   // Parametros intrinsecos de la camara
84   int xsize, ysize;          // Tamano del video de camara (pixels)
85   double c[2] = {0.0, 0.0};  // Centro de patron (por defecto)
86
87   // Abrimos dispositivo de video
88   if(arVideoOpen("") < 0) exit(0);
89   if(arVideoInqSize(&xsize, &ysize) < 0) exit(0);
90
91   // Cargamos los parametros intrinsecos de la camara
92   if(arParamLoad("data/camera_para.dat", 1, &wparam) < 0)
```

```
93      print_error ("Error en carga de parametros de camara\n");
94
95    arParamChangeSize(&wparam, xsize, ysize, &cparam);
96    arInitCparam(&cparam);   // Inicializamos la camara con çparam"
97
98    // Inicializamos la lista de objetos
99    addObject("data/simple.patt", 120.0, c, drawteapot);
100   addObject("data/identic.patt", 90.0, c, drawcube);
101
102   argInit(&cparam, 1.0, 0, 0, 0, 0);   // Abrimos la ventana
103 }
104 /* El bucle principal se encarga de guardar la informacion
       relevante de cada marca en la lista de objetos objects.
       Finalmente llama a la funcion draw que se encargara de utilizar
        el puntero a funcion de dibujado de cada objeto. */
105 // ======== mainLoop
       ================================================
106 static void mainLoop(void) {
107   ARUint8 *dataPtr;
108   ARMarkerInfo *marker_info;
109   int marker_num, i, j, k;
110
111   // Capturamos un frame de la camara de video
112   if((dataPtr = (ARUint8 *)arVideoGetImage()) == NULL) {
113     // Si devuelve NULL es porque no hay un nuevo frame listo
114     arUtilSleep(2);  return;  // Dormimos el hilo 2ms y salimos
115   }
116
117   argDrawMode2D();
118   argDispImage(dataPtr, 0,0);    // Dibujamos lo que ve la camara
119
120   // Detectamos la marca en el frame capturado (return -1 si error)
121   if(arDetectMarker(dataPtr, 100, &marker_info, &marker_num) < 0) {
122     cleanup(); exit(0);   // Si devolvio -1, salimos del programa!
123   }
124
125   arVideoCapNext();      // Frame pintado y analizado... A por
          otro!
126
127   // Vemos donde detecta el patron con mayor fiabilidad
128   for (i=0; i<nobjects; i++) {
129     for(j = 0, k = -1; j < marker_num; j++) {
130       if(objects[i].id == marker_info[j].id) {
131         if (k == -1) k = j;
132         else if(marker_info[k].cf < marker_info[j].cf) k = j;
133       }
134     }
135
136     if(k != -1) {   // Si ha detectado el patron en algun sitio...
137       objects[i].visible = 1;
138       arGetTransMat(&marker_info[k], objects[i].center,
139             objects[i].width, objects[i].patt_trans);
140     } else { objects[i].visible = 0; }  // El objeto no es visible
141   }
142
143   draw();           // Dibujamos los objetos de la escena
144   argSwapBuffers(); // Cambiamos el buffer con lo que tenga
          dibujado
145 }
```

La estructura de datos base del ejemplo es `TObject`, definida en las líneas 2–9. En esta estructura, el último campo `drawme` es un puntero a función, que deberá ser asignado a alguna función que no reciba ni devuelva argumentos, y que se encargará de dibujar el objeto.

Mantendremos en `objects` (línea 11) una lista de objetos reconocibles en la escena. La memoria para cada objeto de la lista se reservará en la función `addObject` (líneas 15–29) mediante una llamada a `realloc` en 21–22. Esta memoria será liberada cuando finalice el programa en `cleanup` (línea 49).

De este modo, en la función de inicialización `init`, se llama a nuestra función `addObject` para cada objeto que deba ser reconocido, pasándole la ruta al fichero del patrón, el ancho, centro y el nombre de la función de dibujado asociada a esa marca (líneas 99–100). En este ejemplo se han definido dos sencillas funciones de dibujado de una tetera 31–37 y un cubo 39–44. En el caso de que una marca no tenga asociado el dibujado de un objeto (como veremos en los ejemplos de la sección 6.3), simplemente podemos pasar `NULL`.

Orientación a Objetos. Como se comentó al principio de este manual, los ejemplos están realizados en C sin orientación a objetos por continuar con el lenguaje base de las bibliotecas empleadas en los ejemplos. Obviamente, en el desarrollo de aplicaciones más complejas sería conveniente realizar una clase de Objeto Dibujable, y utilizar métodos y atributos de clase que encapsulen la información necesaria.

En el bucle principal se han incluido pocos cambios respecto de los ejemplos anteriores. Únicamente en las líneas de código donde se realiza la comprobación de las marcas detectadas 129–142, se utiliza la lista de objetos global y se realiza la comparación con cada marca. Así, el bucle `for` externo se encarga de recorrer todos los objetos que pueden ser reconocidos por el programa 129, y se compara el factor de confianza de cada objeto con el que está siendo estudiado 133. A continuación se obtiene la matriz de transformación asociada al objeto en 139–140.

La función de dibujado (llamada en 144 y definida en 54–79) utiliza igualmente la información de la lista de objetos. Para cada objeto visible 65–66 se carga su matriz de transformación 67–69 y se llama a su función propia de dibujado 75.

6.3. Relación entre Coordenadas

Como hemos visto en los ejemplos de los capítulos anteriores, ARToolKit calcula la posición de la marca en coordenadas de la cámara.

Puede entenderse como un posicionamiento de la cámara en una posición del espacio (correspondiente a la distancia y rotación relativa entre la marca y la cámara real), conservando el origen del mundo en el centro de la marca. Esta *idea* es únicamente una ayuda para entender el posicionamiento real, porque como vimos en el capítulo 5, es una transformación que se realiza en la pila *ModelView*, siendo equivalente mover la cámara y mantener el objeto (la marca) estática o a la inversa.

Coordenadas Marca

Como se ha visto en el capítulo 5, el sistema de coordenadas de la marca sigue el mismo convenio que OpenGL (ver Figura 6.3), por lo que si utilizamos la idea intuitiva de que el origen del SRU está en la marca, podemos posicionar fácilmente los objetos con OpenGL, ya que las transformaciones se aplican siguiendo la misma notación.

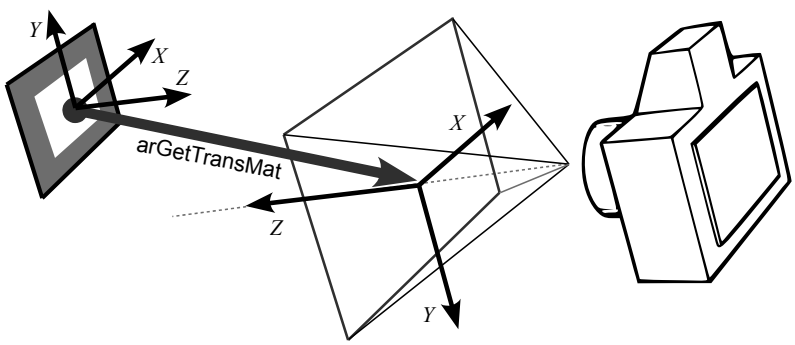


Figura 6.3: Relación entre el sistema de coordendas de la marca y el sistema de coordenadas de la cámara.

De este modo, como se representa en la Figura 6.3, la llamada a `arGetTransMat` nos posiciona la cámara en relación a la marca. Es como si aplicáramos la transformación marcada por la flecha de color rojo, con origen del sistema de referencia en la marca.

De hecho, si en el ejemplo del *Hola Mundo* imprimimos la columna de más a la derecha de la matriz de transformación de la marca, podemos comprobar cómo si movemos la marca hacia arriba (en dirección a su eje Y), el valor de posición en ese Y decrecerá (debido a que el sentido está invertido respecto de la cámara). De forma análoga ocurrirá con el eje Z. Por el contrario, si desplazamos la marca hacia la derecha, el eje X de la matriz crecerá, debido a que ambos sistemas de coordenadas emplean el mismo convenio (ver Figura 6.3).

Continuaremos trabajando con la idea mucho más intuitiva de considerar que la marca es estática y que lo que se desplaza es la cámara (aunque, como veremos, la marca puede en realidad desplazarse y la idea de considerar que la cámara es la móvil seguirá siendo últil).

Veamos a continuación un ejemplo que trabajará con las relaciones entre la cámara y la marca. Tener claras estas transformaciones nos

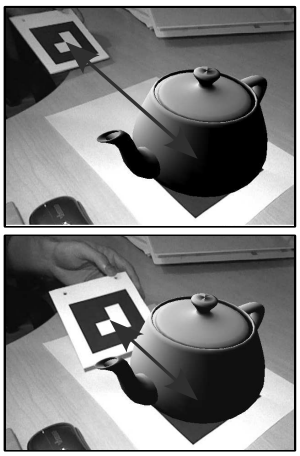


Figura 6.4: Salida del ejemplo de cálculo de distancias entre marcas. La intensidad de la componente roja del color varía en función de la distancia entre marcas.

permitirá cambiar entre sistemas de coordenadas y trabajar con las relaciones espaciales entre las marcas que no son más que transformaciones entre diferentes sistemas de coordenadas. En este programa (listado siguiente) queremos variar la intensidad de color rojo de la tetera según la distancia de una marca auxiliar (ver Figura 6.4).

Listado 6.3: Ejemplo de cálculo de distancias entre marcas

```
1  // ======== drawteapot ============================================
2  void drawteapot(void) {
3    GLfloat material[]     = {0.0, 0.0, 0.0, 1.0};
4    float value = 0.0;            // Intensidad del gris a dibujar
5
6    // La intensidad se asigna en funcion de la distancia "dist01"
7    // dist01 es la distancia entre el objeto 1 y 2 (es var. global)
8    // Mapear valor intensidad linealmente entre 160 y 320 ->(1..0)
9    value = (320 - dist01) / 160.0;
10   if (value < 0) value = 0;  if (value > 1) value = 1;
11   material[0] = value;
12
13   glMaterialfv(GL_FRONT, GL_AMBIENT, material);
14   glTranslatef(0.0, 0.0, 60.0);
15   glRotatef(90.0, 1.0, 0.0, 0.0);
16   glutSolidTeapot(80.0);
17 }
18
19 // ======== draw ================================================
20 void draw( void ) {
21   double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL
22   GLfloat light_position[]  = {100.0,-200.0,200.0,0.0};
23   double m[3][4], m2[3][4];
24   int i;
25
26   argDrawMode3D();              // Cambiamos el contexto a 3D
27   argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
28   glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
29   glEnable(GL_DEPTH_TEST);
30   glDepthFunc(GL_LEQUAL);
31
32   if (objects[0].visible && objects[1].visible) {
33     arUtilMatInv(objects[0].patt_trans, m);
34     arUtilMatMul(m, objects[1].patt_trans, m2);
35     dist01 = sqrt(pow(m2[0][3],2)+pow(m2[1][3],2)+pow(m2[2][3],2));
36     printf ("Distancia objects[0] y objects[1]= %G\n", dist01);
37   }
38
39   for (i=0; i<nobjects; i++) {
40     if ((objects[i].visible) && (objects[i].drawme != NULL)) {
41       argConvGlpara(objects[i].patt_trans, gl_para);
42       glMatrixMode(GL_MODELVIEW);
43       glLoadMatrixd(gl_para);   // Cargamos su matriz de transf.
44
45       glEnable(GL_LIGHTING);  glEnable(GL_LIGHT0);
46       glLightfv(GL_LIGHT0, GL_POSITION, light_position);
47       objects[i].drawme();      // Llamamos a su funcion de dibujar
48     }
49   }
50   glDisable(GL_DEPTH_TEST);
51 }
52
53 // ======== init ================================================
```

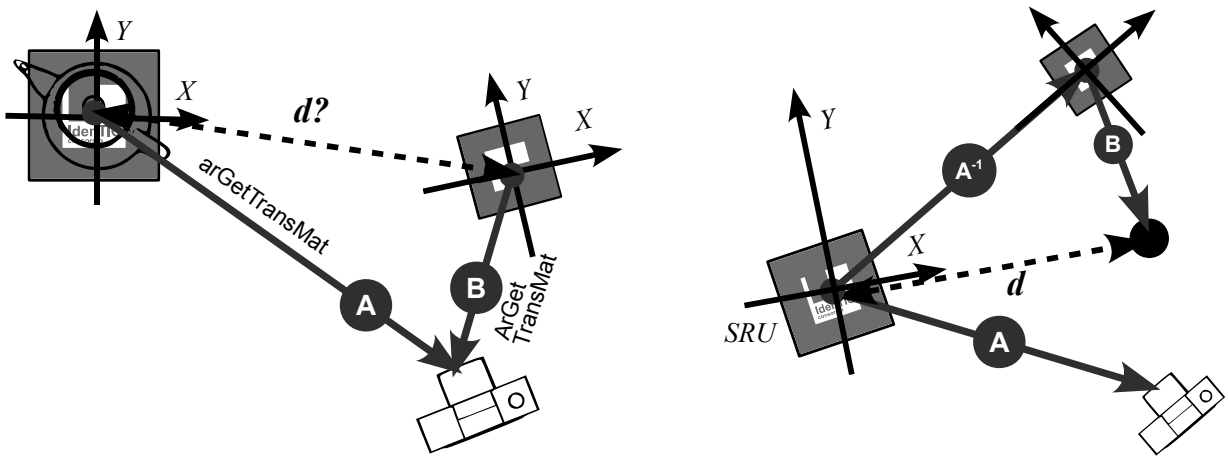


Figura 6.5: Utilización de la transformación inversa a la marca de Identic A^{-1} para calcular la distancia entre marcas. Primero *cargamos* la matriz asociada a la marca de marca de Identic, (estableciendo implícitamente el origen del SRU ahí) y de ahí aplicamos la transformación inversa A^{-1}. A continuación aplicamos la matriz B asociada a la segunda marca. La distancia al origen del SRU es la distancia entre marcas.

```
54 static void init( void ) {
55   // La parte inicial de init es igual que en ejemplos
         anteriores...
56
57   // Inicializamos la lista de objetos
58   addObject("data/identic.patt", 120.0, c, drawteapot);
59   addObject("data/simple.patt", 90.0, c, NULL);
60
61   argInit(&cparam, 1.0, 0, 0, 0, 0);   // Abrimos la ventana
62 }
```

El problema nos pide calcular la distancia entre dos marcas. ARToolKit, mediante la llamada a la función `arGetTransMat` nos devuelve la transformación relativa entre la marca y la cámara. Para cada marca podemos obtener la matriz de transformación *relativa* hacia la cámara. Podemos verlo como la transformación que nos posiciona la cámara en el lugar adecuado para que, dibujando los objetos en el origen del SRU, se muestren de forma correcta. ¿Cómo calculamos entonces la distancia entre ambas marcas?

En el diagrama de la Figura 6.5 se resume el proceso. Con la llamada a `arGetTransMat` obtenemos una transformación relativa al sistema de coordenadas de cada cámara (que, en el caso de una única marca, podemos verlo como si fuera el origen del *SRU* donde dibujaremos los objetos). En este caso, tenemos las flechas que parten de la marca y posicionan relativamente la cámara señaladas con A y B.

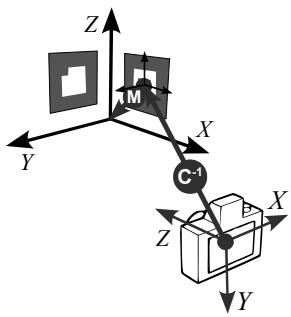


Figura 6.6: Esquema de posicionamiento global utilizando transformaciones inversas.

Podemos calcular la transformación entre marcas empleando la inversa de una transformación, que equivaldría a *viajar* en sentido contrario. Podemos imaginar realizar la transformación contraria; partiendo del origen del *SRU*, viajamos hasta la marca de Identic (aplicando A^{-1}, y de ahí aplicamos la transformación B (la que nos posicionaría

desde la segunda marca hasta la cámara). Así, llegamos al punto final (señalado con el círculo de color negro en la Figura 6.5.derecha). Su distancia al origen del *SRU* será la distancia entre marcas.

Posicionamiento Global

El uso de transformaciones inversas nos permiten obtener coordenadas de posicionamiento global, como se muestra en la Figura 6.6. En ese ejemplo, bastará con conocer la matriz de transformación entre la marca y el origen del SRU (expresada como M), para obtener el posiconamiento global de la cámara respecto del SRU, multiplicando la inversa de la transformación relativa de la marca C^{-1} con la matriz M.

La codificación de esta operación es directa. En la línea [33] del listado anterior, si ambos objetos son visibles [32] se calcula la inversa de la transformación a la marca de Identic (que está en la posición 0 de la lista de objetos). Esta nueva matriz `m` se multiplica a continuación con la matriz de transformación de la segunda marca [34].

Recordemos que la última columna de las matrices netas de transformación codifican la traslación. De este modo, basta con calcular el módulo del vector que une el punto trasladado con el origen para calcular la distancia, que se guarda en la variable global `dist01` [35] (distancia entre los objetos 0 y 1). Para cambiar el color de la tetera en función de esta distancia usamos una función lineal que asigna el nivel de rojo (entre 0 y 1) según la distancia en las líneas [9–11] (si $d \leq 16cm, r = 1$, si $d > 32cm, r = 0$, y linealmente los intermedios).

6.4. Tracking Multi-Marca

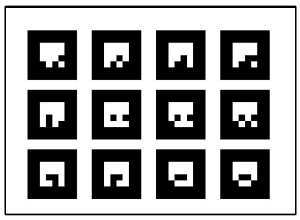


Figura 6.7: Ejemplo de patrón de 12 marcas.

El módulo de tracking Multi-Marca de ARToolKit (definido en el fichero de cabecera `arMulti.h`), nos permite emplear un conjunto de marcas como si fueran una única entidad. De esta forma, mientras *alguna* de las marcas del conjunto sea totalmente visible, ARToolKit será capaz de obtener la posición del resto (en relación a las visibles). Además, si más de una marca es visible, ARToolKit calculará el posicionamiento global empleando la información de cada una de ellas, consiguiendo así realizar un *registro* mucho más estable y preciso.

En la Figura 6.7 podemos ver un ejemplo de patrón Multi-Marca. Este tipo de patrones requiren un fichero de configuración donde se especifique la relación entre marcas y su sistema de coordenadas. A continuación se muestra un fragmento del fichero de configuración (llamado `marker.dat` en este caso) asociado al patrón de la Figura 6.7.

```
#Numero de patrones que seran reconocidos
12

# Marca 1
data/10.patt
50.0
0.0 0.0
1.0000 0.0000 0.0000 50.0000
0.0000 1.0000 0.0000 -50.0000
0.0000 0.0000 1.0000 0.0000

# Marca 2
data/11.patt
50.0
0.0 0.0
1.0000 0.0000 0.0000 115.0000
0.0000 1.0000 0.0000 -50.0000
0.0000 0.0000 1.0000 0.0000
```

```
# A continuacion se definen los 10 patrones que faltan
# etc...
```

Los comentarios comienzan con almohadilla. La primera línea útil del fichero debe definir el número de marcas que contiene el patrón (en este caso 12). A continuación, para cada marca se define el nombre del fichero que contiene el patrón, a continuación el ancho en milímetros (50mm en este caso), la posición del centro como tercer parámetro del fichero (en nuestro caso 0,0), y finalmente la matriz de transformación de la marca en relación al sistema de coordenadas del patrón. Esta matriz 3×4 tiene los siguientes elementos:

$$\begin{bmatrix} R_x & R_y & R_z & P_x \\ U_x & U_y & U_z & P_y \\ F_x & F_y & F_z & P_z \end{bmatrix} \tag{6.1}$$

Siendo R (*Right*) el vector de correspondencia del eje X de la marca en relación al eje X del Sistema de Referencia del patrón multimarca, U (*Up*) el vector de correspondencia del eje Y dela marca en relación al eje Y del Sistema de Referencia del patrón, y análogamente el F (*Front*) se define para el eje Z. En este primer fichero de ejemplo, los ejes de las marcas están alineados con el del patrón, por lo que esa parte izquierda de la matriz es igual a la matriz identidad. La Figura 6.8 muestra un ejemplo de definición de matriz con diferencias en el sistema de coordenadas.

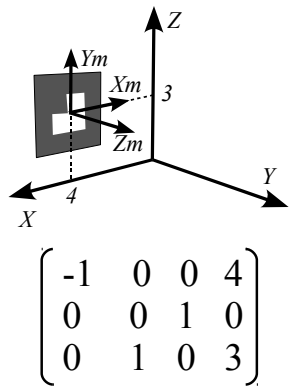


Figura 6.8: Ejemplo de definición de la matriz de transformación de una marca con un sistema de coordenadas orientado diferente al del patrón.

Las componentes P se corresponden con la posición de la marca. En este caso, como están todas sobre el mismo plano Z, esa componente es igual a cero en todas las marcas. La posición del centro cambia. El centro de la primera marca por ejemplo, está situado a 5cm de cada margen del patrón (ver Figura 6.9). Con la definición de este fichero de configuración se puede emplear la llamada a una función llamada `arMultiReadConfigFile`, que nos devuelve un puntero a una estructura de tipo `ARMultiMarkerInfoT` (ver Tabla 6.1).

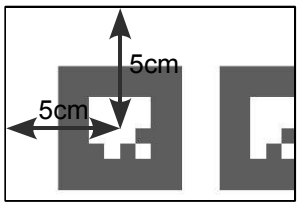


Figura 6.9: Definición del centro de la marca en el patrón MultiMarca.

Tabla 6.1: Campos de la estructura *ARMultiMarkerInfoT*

Tipo	Campo	Descripción
`ARMEMIT*`	**`marker`**	Lista de marcas del patrón MultiMarca.
`int`	**`marker_num`**	Número de marcas utilizadas actualmente.
`double`	**`trans[3][4]`**	Matriz de transformación de la cámara en relación al sistema de referencia del patrón (calculada a partir de todas las marcas visibles del patrón).
`int`	**`prevF`**	Flag (booleano) de visibilidad del patrón.
`double`	**`transR[3][4]`**	Última posición (por si se quiere utilizar histórico).

Como campos más relevantes de esta estructura podemos destacar `trans[3][4]`, que define la matriz de transformación del patrón completo. Esta matriz es calculada por ARToolKit empleando las marcas

del patrón que son visibles. A mayor número de marcas visibles, mejor estabilidad en el registro. El campo `marker` contiene la lista de marcas del patrón (del tipo `ARMultiEachMarkerInfoT`, ver Tabla 6.2).

ARMEMIT

Se ha contraido el nombre del tipo, para facilitar la escritura en la tabla. El nombre completo es: `ARMultiEachMarkerInfoT` (ver Tabla 6.2). Así, la estructura `ARMultiMarkerInfoT` contiene una lista de elementos de este tipo, que contendrán la información relativa a cada marca del patrón.

Tabla 6.2: Campos de la estructura *ARMultiEachMarkerInfoT* (ARMEMIT)

Tipo	Campo	Descripción
int	**patt_id**	Identificador de la marca.
double	**width**	Ancho de la marca (mm).
double	**center[2]**	Centro del patrón (mm).
double	**trans[3][4]**	Posición *estimada* de la marca.
double	**itrans[3][4]**	Posicion relativa de la marca (en relación al sistema de referencia del patrón).
double	**pos3d[4][3]**	posición final de la marca.
int	**visible**	Flag (booleano) de visibilidad de la marca.
int	**visibleR**	Estado de visibilidad anterior.

Esta estructura de datos, además de los campos habituales en marcas de ARToolKit contiene la matriz de transformación `trans`, que puede estar calculada de forma directa (porque la marca es visible) o bien estimada en relación a las otras marcas que forman parte del patrón Multimarca. El usuario puede saber si la matriz ha sido calculada de forma directa (o ha sido estimada por sus marcas vecinas) empleando el campo `visible`. El campo `itrans` contiene la posición relativa de la marca en relación al sistema de referencia del patrón (la información introducida en el fichero de configuración).

Listado 6.4: Ejemplo de uso de un patrón MultiMarca

```
1  // ==== Definicion de constantes y variables globales =============
2  ARMultiMarkerInfoT  *mMarker;       // Estructura global Multimarca
3  int dmode = 0;   // Modo dibujo (objeto centrado o cubos en marcas)
4  // ======== keyboard ================================================
5  static void keyboard(unsigned char key, int x, int y) {
6    switch (key) {
7    case 0x1B: case 'Q': case 'q':  cleanup(); break;
8    case 'D': case'd': if (dmode == 0) dmode=1; else dmode=0; break;
9    }
10 }
11 (*@\newpage@*)
12 // ======== draw ====================================================
13 static void draw( void ) {
14   double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL
15   GLfloat material[]        = {1.0, 1.0, 1.0, 1.0};
16   GLfloat light_position[]  = {100.0,-200.0,200.0,0.0};
17   int i;
18
19   argDrawMode3D();              // Cambiamos el contexto a 3D
20   argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
21   glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
22   glEnable(GL_DEPTH_TEST);
23   glDepthFunc(GL_LEQUAL);
24
25   argConvGlpara(mMarker->trans, gl_para);
```

```
26   glMatrixMode(GL_MODELVIEW);
27   glLoadMatrixd(gl_para);
28
29   glEnable(GL_LIGHTING);  glEnable(GL_LIGHT0);
30   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
31
32   if (dmode == 0) {  // Dibujar cubos en marcas
33     for(i=0; i < mMarker->marker_num; i++) {
34       glPushMatrix();   // Guardamos la matriz actual
35       argConvGlpara(mMarker->marker[i].trans, gl_para);
36       glMultMatrixd(gl_para);
37       if(mMarker->marker[i].visible < 0) {  // No se ha detectado
38         material[0] = 1.0; material[1] = 0.0; material[2] = 0.0; }
39       else {           // Se ha detectado (ponemos color verde)
40         material[0] = 0.0; material[1] = 1.0; material[2] = 0.0; }
41       glMaterialfv(GL_FRONT, GL_AMBIENT, material);
42       glTranslatef(0.0, 0.0, 25.0);
43       glutSolidCube(50.0);
44       glPopMatrix();   // Recuperamos la matriz anterior
45     }
46   } else { // Dibujar objeto global
47     glMaterialfv(GL_FRONT, GL_AMBIENT, material);
48     glTranslatef(150.0, -100.0, 60.0);
49     glRotatef(90.0, 1.0, 0.0, 0.0);
50     glutSolidTeapot(90.0);
51   }
52   glDisable(GL_DEPTH_TEST);
53 }
54 // ======== init ================================================
55 static void init( void ) {
56   // Las lineas anteriores del codigo son iguales a las empleadas
57   // en otros ejemplos (se omiten en este listado).
58   if( (mMarker = arMultiReadConfigFile("data/marker.dat")) == NULL
         )
59     print_error("Error en fichero marker.dat\n");
60   argInit(&cparam, 1.0, 0, 0, 0, 0);   // Abrimos la ventana
61 }
62 // ======== mainLoop ============================================
63 static void mainLoop(void) {
64   ARUint8 *dataPtr;
65   ARMarkerInfo *marker_info;
66   int marker_num;
67
68   if((dataPtr = (ARUint8 *)arVideoGetImage()) == NULL) {
69     arUtilSleep(2);  return;  // Dormimos el hilo 2ms y salimos
70   }
71   argDrawMode2D();   argDispImage(dataPtr, 0,0);
72   // Detectamos la marca en el frame capturado (return -1 si error)
73   if(arDetectMarker(dataPtr, 100, &marker_info, &marker_num) < 0) {
74     cleanup(); exit(0);
75   }
76   arVideoCapNext();      // Frame pintado y analizado... A por
         otro!
77   if(arMultiGetTransMat(marker_info, marker_num, mMarker) > 0)
78     draw();       // Dibujamos los objetos de la escena
79   argSwapBuffers();
80 }
```

Veamos el ejemplo del listado 6.4 que utiliza un patrón MultiMarca. En la función `init` se llama a la función `arMultiReadConfigFile` (línea 58), pasándole como parámetro el fichero de configuración que hemos definido con las marcas del patrón y las matrices de transformación.

En el bucle principal (líneas 63–80) se utiliza la llamada de detección de marcas igual que en los ejemplos anteriores 73, pero se utiliza una función especial para obtener la matriz de transformación 77 que rellena una estructura de tipo `ARMultiMarkerInfoT` (en el caso de este código es una variable global llamada `mMarker` 2). ARToolKit se encarga de reservar memoria para la estructura.

Esa estructura nos permite utilizar la matriz de transformación *global* de todo el patrón, o también podemos acceder a las matrices de transformación de cada marca dentro del patrón. En el caso del ejemplo del Listado 6.4, se utilizan ambas. Inicialmente cargamos la matriz global del patrón, que se encuentra en el campo `trans` de la variable `mMarker` 25–27 (ver Cuadro 6.1). Cargando esta matriz, si dibujamos lo haremos en el origen del sistema de coordenadas del patrón (en nuestro caso situado en la esquina superior izquierda).

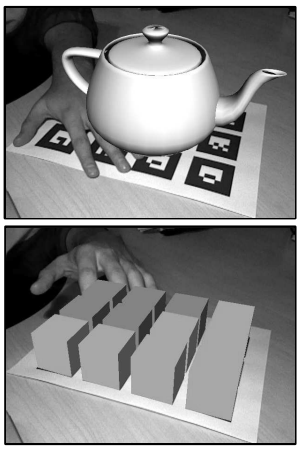


Figura 6.10: Arriba. Gracias al uso de un patrón MultiMarca es posible realizar el registro correctamente únicamente mientras alguna de las marcas del patrón sean visibles. **Abajo**. En este momento la posición de 5 de las marcas (ocultas por la mano) está siendo estimada a partir de las otras marcas.

Si por el contrario queremos dibujar en el centro de cada marca, utilizaremos la lista de marcas `marker` de tipo `ARMultiEachMarkerInfoT`. Como hemos explicado antes, esta estructura tiene su propia matriz de transformación (calculada de forma directa si la marca es visible o estimada si no lo es). Recordemos que esta matriz es *relativa* al origen del sistema de coordenadas del patrón, por lo que es necesario **multiplicarlas** (líneas 35–36) para obtener la posición del centro de la marca.

En el ejemplo, si se ha pulsado la tecla *d*, se recorren todas las marcas del patrón 33, y se obtiene la posición en el centro de la marca multiplicando con la matriz global del patrón (que se cargó anteriormente en las líneas 25–27). Para recuperar fácilmente la matriz de transformación del patrón cargada anteriormente, se realiza una copia de la matriz de la cima de la pila con `glPushMatrix` en 34. Se dibuja un cubo de color rojo (línea 38) si la marca no es visible (su posición ha sido estimada) y de color verde en otro caso 40 (ver Figura 6.10).

6.5. Ejercicios Propuestos

Se recomienda la realización de los ejercicios de esta sección en orden, ya que están relacionados y su complejidad es ascendente.

1. Implemente su propia versión del histórico de percepciones, definiendo un array de percepciones de un tamaño N. El valor final de la matriz la calculará como una combinación lineal de los elementos del vector, deciendo qué peso asigna a cada percepción. Implemente una versión sencilla del histórico que calcule la media de los elementos del vector por parejas, comenzando en las

Figura 6.11: Resultado del ejercicio 2. Según se aplica una rotación a la marca auxiliar (respecto de su eje Z), se aplicará al objeto situado sobre la marca principal.

percepciones más antiguas. De este modo, las percepciones más recientes tendrán más peso en el resultado final.

2. Desarrolle una aplicación que utilice dos marcas. Sobre la primera, dibujará una tetera (similar al primer ejemplo "*Hola Mundo*" del documento). Utilizará la rotación de la segunda marca relativa a su eje Z para aplicar la misma rotación a la tetera que se dibuja sobre la primera marca, tal y como se muestra en la figura 6.11.

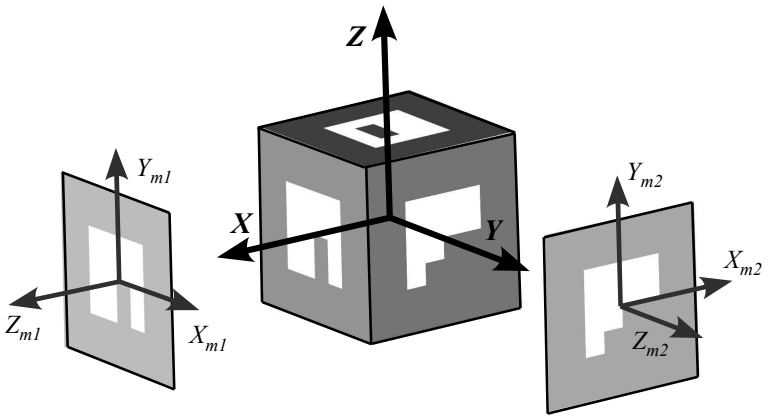


Figura 6.12: Esquema de patrón cúbico. Descripción de los sistemas de coordenadas locales de cada marca en relación a los sistemas de referencia universal del patrón multimarca.

3. Empleando el patrón cúbico suministrado (ver Figura 6.12), y sabiendo que el cubo tiene de lado 6cm (la marca del interior de cada cara tiene 5cm de lado), defina un patrón multimarca y utilícelo para dibujar un objeto correctamente alineado siempre que *alguna* de sus marcas sea visible. el origen del sistema de referencia universal se situará en el interior del cubo (en el centro

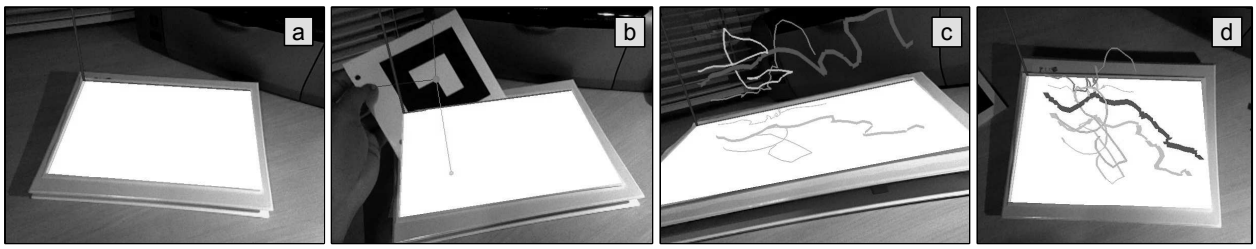


Figura 6.13: Posible salida del ejercicio del sistema de dibujo 3D. **a)** Primera etapa; dibujo de los ejes sobre el patrón multimarca, y un plano (`GL_QUADS`) en la base. **b)** Segunda etapa; dibujado del puntero 3D, con proyección sobre cada plano del sistema de coordenadas. **c)** y **d)** Ejemplo de resultado con líneas de diferentes colores y grosores.

geométrico), con el eje X positivo saliendo por la marca 1, el eje Y positivo saliendo por la marca 2, y el eje Z positivo saliendo por la marca 5.

4. Realice una aplicación de dibujo en 3D que cumpla los siguientes requisitos. Utilice el patrón multimarca del ejemplo 6.4 como base para definir la *tabla* sobre la que dibujará. Utilice el patrón sencillo como puntero 3D. Cuando pulse la tecla *d*, el sistema deberá comenzar a dibujar líneas hasta que presione de nuevo la tecla *d* o la marca del puntero 3D no sea detectada. El puntero 3D admitirá cambio de grosor (para dibujar líneas de diferentes grosores) mediante las teclas + y -, o cambiar entre una lista de colores mediante la tecla *c*. Como recomendación, realice la implementación en etapas:

   - **Primera etapa:** Utilice el patrón multimarca y céntrese en dibujar el sistema de referencia y la base de la *tabla de dibujo* (obtenga un resultado similar a la figura 6.13.a). Utilice las primitivas `GL_LINES` y `GL_QUADS`.
   - **Segunda etapa:** Dibuje la posición del puntero 3D en la posición correcta, así como las guías proyectadas sobre los planos del patrón multimarca (ver figura 6.13.b).
   - **Tercera etapa:** Finalmente cree una estructura de datos sencilla para almacenar las líneas que se van dibujando. Será necesario almacenar las coordenadas de la línea, el color de dibujado y el grosor. No se preocupe de la gestión dinámica de memoria; simplemente reserve memoria estática para un número máximo de líneas. Las 6.13.c y .d muestran un resultado de dibujado de algunas líneas con diferentes colores y grosores. Las coordenadas de las líneas deben ir asociadas a las coordenadas del patrón multimarca.

Capítulo 7

Producción de Contenidos para Realidad Aumentada

Como hemos visto en los capítulos anteriores, ARToolKit nos da soporte para calcular la posición de la cámara relativa a cada marca. Mediante el uso de bibliotecas gráficas (como OpenGL) podemos representar primitivas en el espacio 3D. Nos falta terminar el proceso mediante la generación de contenidos para sistemas de Realidad Aumentada. En este capítulo estudiaremos cómo exportar contenido desde Blender, la suite de gráficos 3D más instalada del mundo.

El propósito de este capítulo es estudiar brevemente las características principales que hacen de Blender una herramienta muy potente en el ámbito de la producción de contenidos para Realidad Aumentada. Esta sección no pretende ser un manual de Blender; se supone que el lector está familiarizado con la herramienta, por lo que únicamente se expondrán aquellas características que son relevantes al problema que nos ocupa[1].

En concreto estudiaremos el soporte de texturas mediante mapas UV (ampliamente utilizados en gráficos interactivos), la capacidad de especificar el centro y las dimensiones exactas de objetos, así como algunos scripts incluidos en la distribución de Blender que nos permiten *solidificar* el alambre asociado a un modelo poligonal.

[1]Se recomienda la lectura del libro Fundamentos de Síntesis de Imagen 3D, un enfoque práctico a Blender, disponible online en `http://www.esi.uclm.es-/www/cglez/fundamentos3D`

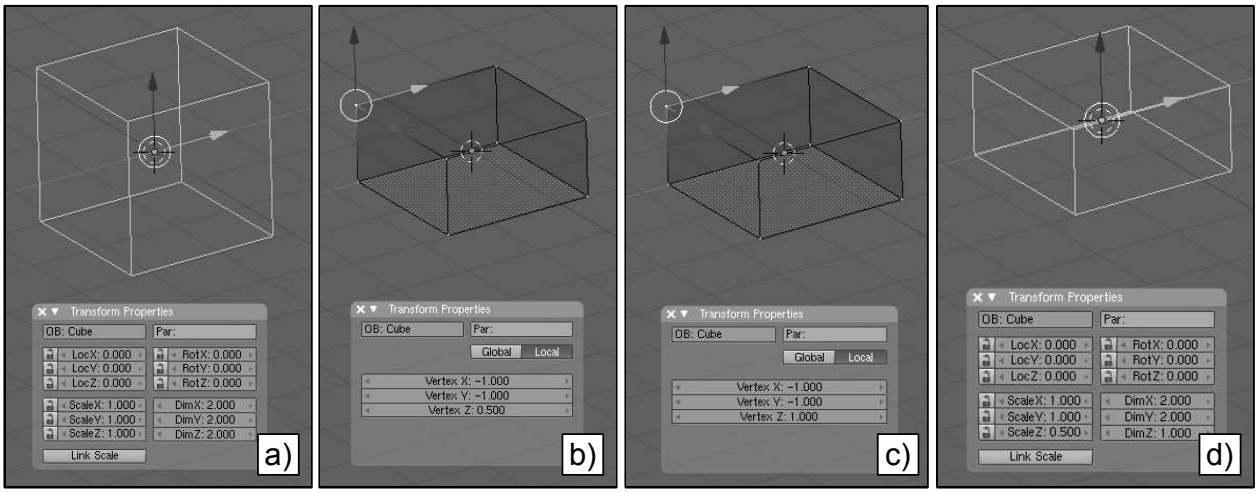


Figura 7.3: Aplicación de transformaciones geométricas en modo Edición. a) Situación inicial del modelo al que le aplicaremos un escalado de 0.5 respecto del eje Z. b) El escalado se aplica en modo Edición. El campo Vertex Z muestra el valor correcto. c) Si el escalado se aplicó en modo objeto, el campo Vertex Z no refleja la geometría *real* del objeto. Esto puede comprobarse fácilmente si alguno de los campos *Scale* no son igual a uno (d).

7.1. Modelado 3D

Blender es una herramienta de modelado de contorno *B-Rep (Boundary Representation)*. Esto implica que, a diferencia de herramientas de CAD, trabaja con *modelos huecos* definidos por vértices, aristas y caras. Estos modelos tienen asociado un centro que define el origen de su sistema de referencia local.

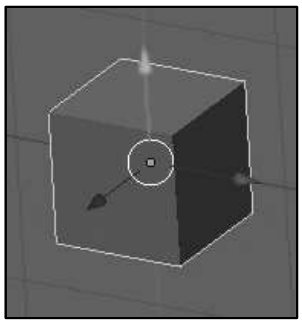


Figura 7.1: El centro del objeto define la posición del sistema de referencia local. Así, la posición de los vértices del cubo se definen según este sistemas local.

En Blender el centro del objeto se representa mediante un punto de color rosa (ver Figura 7.1). Si tenemos activos los manejadores en la cabecera de la ventana 3D , será el punto de donde comienzan los manejadores. Este centro define el sistema de referencia local, por lo que resulta crítico poder cambiar el centro del objeto ya que, tras su exportación, se dibujará a partir de esta posición.

Con el objeto seleccionado en Modo de Objeto se puede indicar a Blender que recalcule el centro geométrico del objeto en los botones de edición , en la pestaña Mesh ver Figura 7.2, pulsando en el botón Center New. Se puede cambiar el centro del objeto a cualquier posición del espacio situando el puntero 3D (pinchando con) y pulsando posteriormente en Center Cursor.

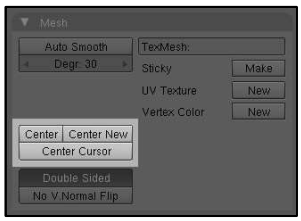


Figura 7.2: Botones para el cambio del centro del objeto, en la pestaña *Mesh*.

Si queremos situar un objeto virtual correctamente alineado con un objeto real, será necesario situar con absoluta precisión el centro y los vértices que lo forman.

Para situar con precisión numérica los vértices del modelo, basta con pulsar la tecla N en modo edición. En la vista 3D aparecerá una ventana como la que se muestra en la Figura 7.4, en la que podremos

especificar las coordenadas específicas de cada vértice. Es interesante que esas coordenadas se especifiquen *localmente* (botón Local activado), porque el exportador que desarrollaremos trabajará con esas coordenadas (al menos, con modelos estáticos).

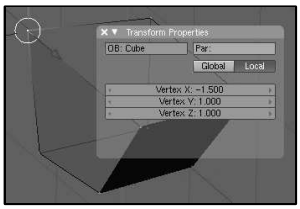


Figura 7.4: Ventana emergente al pulsar la tecla *N* para indicar los valores numéricos de coordenadas de los vértices del modelo.

El centro del objeto puede igualmente situarse de forma precisa, empleando la posición del puntero 3D. El puntero 3D puede situarse en cualquier posición del espacio con precisión, accediendo al menú *View/ View Properties* de la cabecera de una ventana 3D. En los campos de *3D Cursor* podemos indicar numéricamente la posición del cursor 3D (ver Figura 7.5) y posteriormente utilizar el botón Center Cursor (ver Figura 7.2).

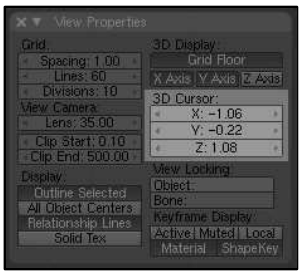


Figura 7.5: Posicionamiento numérico del cursor 3D.

Es muy importante que las transformaciones de modelado se realicen siempre en modo de edición, para que se apliquen de forma efectiva a las coordenadas de los vértices. Es decir, si después de trabajar con el modelo, pulsando la tecla N en modo Objeto, el valor *ScaleX*, *ScaleY* y *ScaleZ* es distinto de 1 en alguno de los casos, implica que esa transformación se ha realizado en modo objeto, por lo que los vértices del mismo no tendrán esas coordenadas asociadas. La Figura 7.3 muestra un ejemplo de este problema; el cubo de la izquierda se ha escalado en modo edición, mientras que el de la derecha se ha escalado en modo objeto. El modelo de la izquierda se exportará correctamente, porque los vértices tienen asociadas sus coordenadas locales. El modelo de la derecha sin embargo aplica una transformación a nivel de objeto, y se exportará como un cubo perfecto.

En el caso de trabajar con animaciones, las transformaciones se realizan habitualmente sobre el Sistema de Referencia Universal (*SRU*). De este modo, y como convenio para la realización del exportador con el que trabajaremos en este capítulo, es importante que el centro del objeto se sitúe en el origen del *SRU*. Esta operación se puede realizar fácilmente con el objeto seleccionado, en modo Objeto, pulsamos la tecla N y establecemos los campos *LocX*, *LocY* y *LocZ* a 0. Así, el centro del objeto coincidirá con el origen del *SRU*.

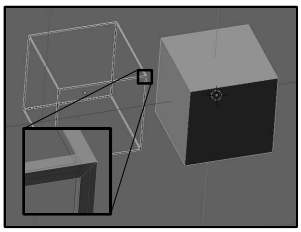


Figura 7.6: Ejemplo de solidificado de aristas de un modelo 3D.

7.1.1. Solidificado de Alambres

Blender incorpora un interesante *script* que permite crear una representación sólida de las aristas de una malla poligonal. Esto resulta muy interesante en el ámbito de la Realidad Aumentada, debido a que en muchas ocasiones es interesante mostrar un modelo de alambre sobre el que queremos tener un control muy preciso sobre cómo se mostrará, asociar diferentes materiales en sus aristas, etc.

Con el objeto elegido en modo *Edición*, accedemos al menú *Mesh/ Scripts/ Solid Wireframe*. El script nos permite modificar el ancho de las aristas que van a *solidificarse* (parámetro *Thick*). La Figura 7.6 muestra un ejemplo de modelo obtenido con este script, así como un detalle de la malla resultado.

7.2. Materiales y Texturas

Los materiales definen propiedades que son constantes a lo largo de la superficie. En los ejemplos de los capítulos anteriores, siempre hemos definido materiales básicos en las superficies, con colores constantes.

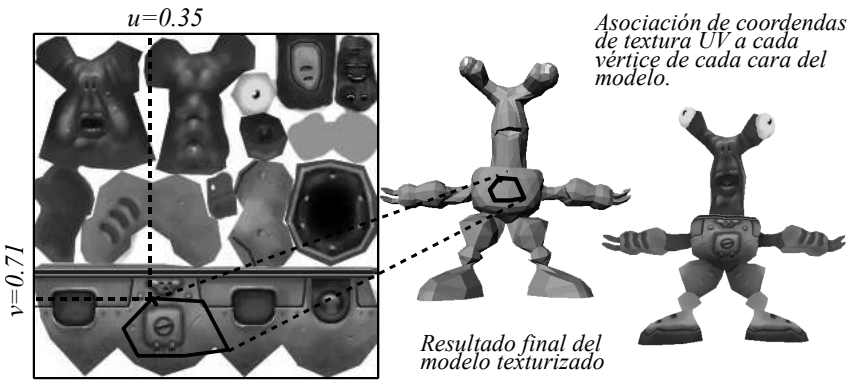


Figura 7.7: Asignación de coordenadas UV a los vértices de un modelo poligonal.

Las texturas permiten variar estas propiedades, determinando en cada punto cómo cambian concretamente estas propiedades. Básicamente podemos distinguir dos tipos de texturas:

- **Texturas Procedurales**: Cuyo valor se determina mediante una ecuación. Estas texturas se calculan rápidamente y no tienen requisitos de espacio en disco, por lo que son ampliamente utilizadas en síntesis de imagen realista para simular ciertos patrones existentes en la naturaleza (madera, mármol, nubes, etc). La Figura 7.8 muestra un ejemplo de este tipo de texturas.
- **Texturas de Imagen**: Almacenan los valores en una imagen, típicamente bidimensional.

```
color func(p3d p){
 if (sin(p_z)>0)
   return C_1
 else
   return C_2
}
```

Figura 7.8: Definición de una sencilla textura procedural que define bandas de color dependiendo del valor de coordenada Z del punto 3D.

Para utilizar las *Texturas de Imagen* es necesario indicar cómo queremos aplicar esa textura (2D) a la geometría del modelo 3D. Es decir, debemos espeicifcar cómo se *recubrirá* el objeto con ese mapa de textura. Existen varias alternativas para realizar este *mapeado*. Empleando proyecciones ortogonales es posible describir esta correspondencia, describiendo cuatro modos básicos de proyección (ver Figura 7.9).

Un método de proyección de texturas de imagen muy empleado en gráficos interactivos es el mapeado paramétrico, también denominado mapeado UV. En este método se definen dos coordenadas paramétricas (entre 0 y 1) para cada vértice de cada cara del modelo (ver Figura 7.7). Estas coordenadas son independientes de la resolución de

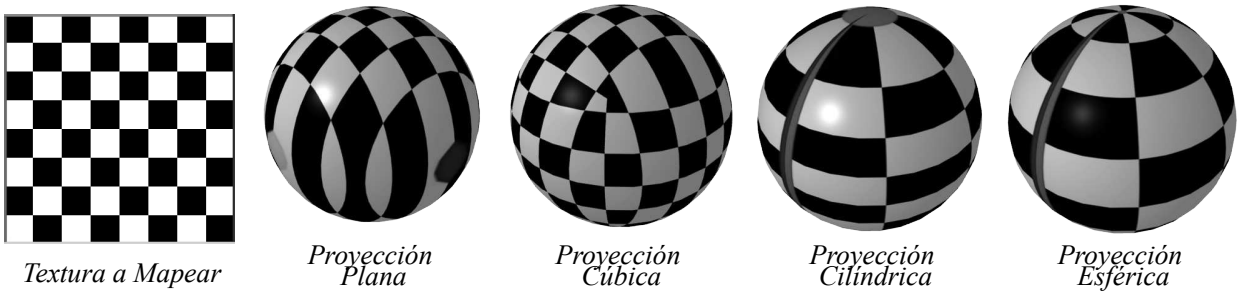


Figura 7.9: Métodos básicos de mapeado ortogonal. Nótese los bordes marcados con líneas de colores en la textura a mapear (izquierda) en las diferentes proyecciones.

la imagen, por lo que permite cambiar en tiempo de ejecución la textura teniendo en cuenta ciertos factores de distancia, importancia del objeto, etc.

El mapeado UV permite *pegar* la textura al modelo de una forma muy precisa. Incluso si se aplica sobre el modelo deformación de vértices (*vertex blending*), el mapa seguirá aplicándose correctamente. Por esta razón y su alta eficiencia es una técnica ampliamente utilizada en gráficos por computador.

7.2.1. Mapas UV en Blender

Blender da soporte a este tipo de proyección. Aunque Blender no impone ninguna restricción a la hora de trabajar con tamaños de texturas, las tarjetas gráficas están optimizadas para trabajar con texturas cuyas dimensiones seanmúltiplos de 2^n píxeles. Será recomendable en Blender trabajar igualmente con ese tipo de texturas para facilitar su posterior exportación.

Exportando Texturas

En muchas ocasiones crearemos imágenes de textura desde el propio interfaz de Blender. En el menú *Image* de la ventana UV/Image editor se encuentran las opciones de exportación de estos mapas de textura.

Una vez tengamos la geometría construida, pasaremos a texturizar el modelo con UV Mapping. Dividiremos la pantalla y en una parte tendremos el archivo de imagen con la texturas (ventana de tipo UV/Image editor). Una vez cargada la imagen, asignaremos a cada cara del modelo una porción de la textura. Para ello, en modo *Edición* pulsaremos la tecla U y utilizaremos algún método de proyección UV que nos resulte cómodo (por ejemplo, para modelos complejos podemos utilizar Unwrap que nos calcula automáticamente las mejores proyecciones). Para el caso del cubo sencillo de la Figura 7.10 se empleó *Reset* y se posicionó manualmente el mapa en la imagen.

Recordemos que para ver el resultado, activaremos el modo de sombreado con texturas Textured en la ventana 3D.

En el menú *Image* de la cabecera de la ventana UV/Image editor , podemos guardar el mapa de imagen (opción *Save As...*) con el que estamos trabajando en diferentes formatos. También podemos crear imágenes nuevas con la opción *New*, especificando el tamaño de la

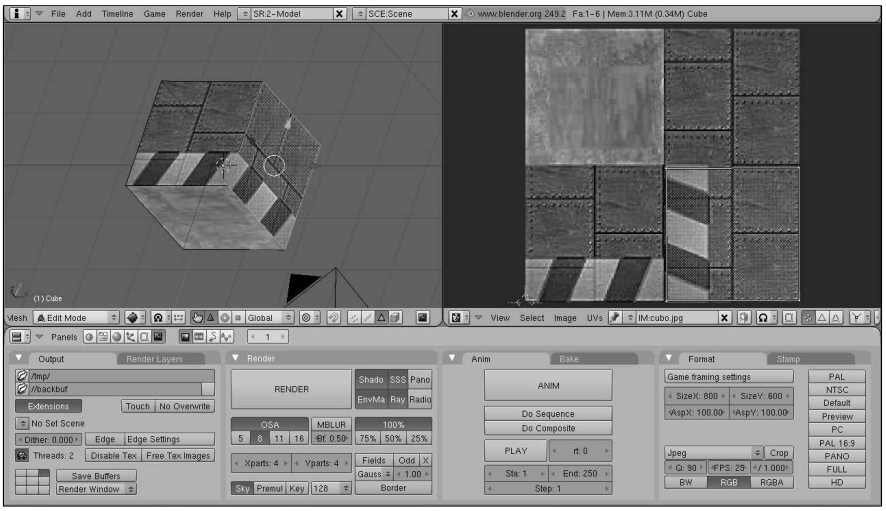


Figura 7.10: Ejemplo de utilización de mapas UV en Blender.

textura que queremos crear. Esto es muy interesante, debido a que el formato con el que vamos a trabajar en el resto del capítulo únicamente soporta Texturas UV (no permite trabajar con materiales o con otro tipo de proyección).

7.3. Precálculo del Render (Baking)

Desde hace unos años, Blender soporta asignar a texturas el resultado completo del Render, en un proceso de *precocinado* del rendering (*Baking*). De este modo se calcula la interacción de la luz sobre la superficie del modelo, almacenando el resultado en un mapa de imagen (con sus correspondientes coordenadas de UV-Mapping). Esta técnica se utiliza ampliamente en gráficos interactivos, y la Realidad Aumentada no es una excepción. La pestaña asociada al Render Baking está situada en el grupo de botones de Render, en la pestaña *Bake* (ver Figura 7.11).

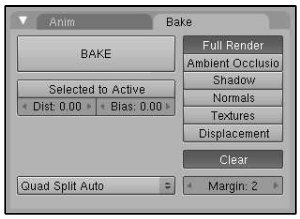


Figura 7.11: Menu para Render Baking.

A continuación veremos un ejemplo de cómo utilizar esta opción de Blender. Partiendo de un modelo con los materiales asignados, entramos en modo *Edición* y seleccionamos toda la geometría [A]. A continuación pulsamos [U] y elegimos *Unwrap (Smart Projections)* como modo de proyección de la textura. Debemos obtener una representación como la mostrada en la Figura 7.12 (superior), en la que la geometría se ha desplegado sobre un plano.

Ahora crearemos una nueva imagen en el menú de UV/Image editor, *Image New* (por ejemplo de 512x512 píxeles. Hecho esto, ya tenemos la imagen (inicialmente de color negro) sobre la que calcular

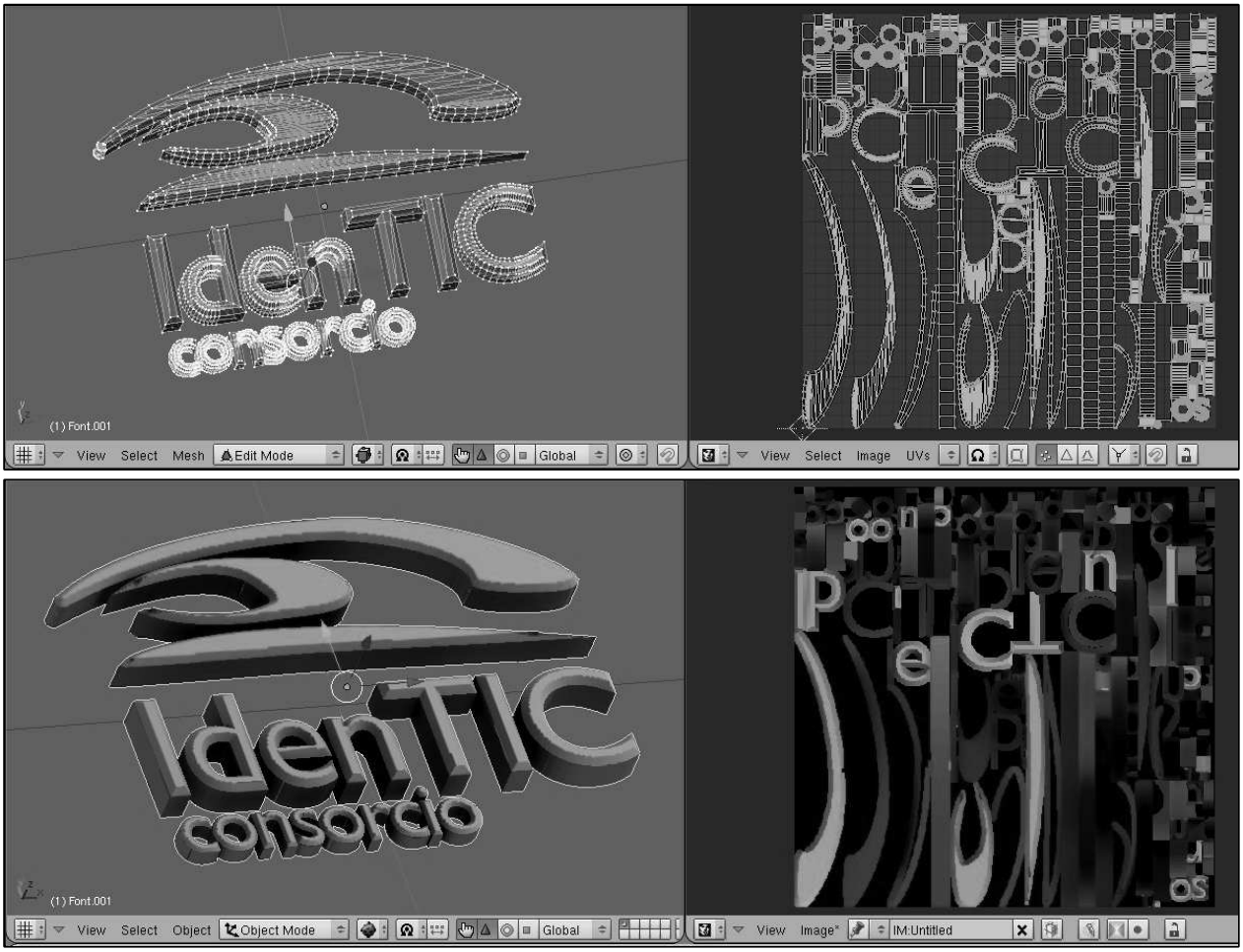


Figura 7.12: Utilización de Render Baking. **Arriba**. El modelo tras calcular la proyección de forma automática. **Abajo**. Resultado de precalcular el resultado del render sobre la textura.

el Baking. Dejamos seleccionado el botón **Full Render** de la pestaña *Bake* y pulsamos el botón **Bake**. Hecho esto, irá apareciendo sobre la imagen del UV/Image editor el resultado del proceso de precálculo de la textura, obteniendo un resultado final como el que se muestra en la figura 7.12 (inferior).

Gracias a este proceso podemos asignar texturas realistas que simulen el comportamiento de la luz a modelos de baja poligonalización, especialmente interesantes en aplicaciones de Realidad Aumentada.

7.4. Un exportador sencillo

A continuación estudiaremos cómo crear un exportador que soporte las siguientes características:

- **Soporte de modelos poligonales**. El formato soporta modelos poligonales. Aunque el exportador soporta polígonos de cualquier número de vértices, el importador está limitado a trabajar con triángulos, por lo que debemos convertir previamente los modelos a caras triangulares.
- **Texturas**. Soporte de texturas con mapas UV. No se dan soporte para otro tipo de materiales. La implementación actual ignora cualquier información sobre los modelos de iluminación (no se exportan normales de vértice ni de cara).
- **Animaciones**. Soporte de una animación de cuerpo sólido por modelo, y diferentes modos de reproducción.
- **Representación**. Sencillo modo de representación alámbrica empleando modo de transparencia.

Como se ha comentado al inicio del capítulo, estos módulos de exportación e importación están creado con propósitos docentes. Sería conveniente realizar una mejor estructuración del código, reserva de memoria dinámica, soporte de materiales, texturas con canal *alpha*, comprobación de errores, etc.

7.4.1. Formato OREj

Un ejemplo de fichero en formato OREj, lo podemos ver en el siguiente listado.

```
# Objeto OREj: cubo
# Vertices Totales: 8
# Caras Totales: 12
v 1.000000 1.000000 -1.000000
v 1.000000 -1.000000 -1.000000
v -1.000000 -1.000000 -1.000000
...
f 4 1 5
f 8 4 5
f 8 7 3
...
t 0.01171875 0.4921875 0.01171875 0.01171875 0.4921875 0.01171875
t 0.01171875 0.4921875 0.4921875 0.01171875 0.4921875 0.4921875
t 0.49609375 0.0078125 0.5 0.5 0.0078125 0.50390625
...
# FRAME  0
m  1.000000 0.000000 0.000000 0.000000 0.000000 -0.000000
   1.000000 0.000000 0.000000 -1.000000 -0.000000 0.000000
   -0.010914 0.020751 -0.015152 1.000000
# FRAME  1
m  1.000000 0.000000 0.000000 0.000000 0.000000 -0.000000
   1.000000 0.000000 0.000000 -1.000000 -0.000000 0.000000
   -0.010913 0.020749 -0.015155 1.000000
```

Figura 7.13: El formato OREj en realidad es una adaptación del formato de texto OBJ de Lightwave, para soportar únicamente las características necesarias en desarrollos internos del grupo ORETO de la Universidad de Castilla-La Mancha. De ahí la evolución de su nombre: Oreto + O-Be-Jota = Ore-Jota.

Los comentarios comienzan con almohadilla. La descripción de las coordenadas de los vértices (que serán definidos en un mismo bloque del archivo), comenzarán con la letra `v` y a continuación sus coordenadas locales *x*, *y*, *z* relativas al centro del objeto. A continuación se definen las caras del objeto (de cualquier número de vértices, siempre superior a 3), que comenzarán con la letra `f`, seguido de los índices de los vértices descritos anteriormente. Por ejemplo, f 3 2 1 definirá una cara formada por los vértices número 3, 2 y 1. El orden en que se dan los vértices es importante a la hora de calcular el vector normal.

Definición de vértices

El orden de definición de vértices es tal y como será necesario en OpenGL; el vector normal saliendo hacia afuera, si describimos los vértices en contra del sentido de giro de las agujas del reloj.

Después se describen las coordenadas de textura UV (comenzando con la letra `t`). Hay dos coordenadas por cada vértice (la primera corresponde al parámetro *U* y la segunda al *V*). El orden en que aparecen las coordenadas de textura (`t`) es el mismo que el orden en que aparecieron las caras (`f`).

Finalmente, asociados a la letra `m` se describen los 16 valores de la matriz de transformación neta del objeto para cada frame de la animación. El orden de los elementos de la matriz es el mismo que define OpenGL (las matrices se definen por columnas).

7.4.2. Exportación desde Blender

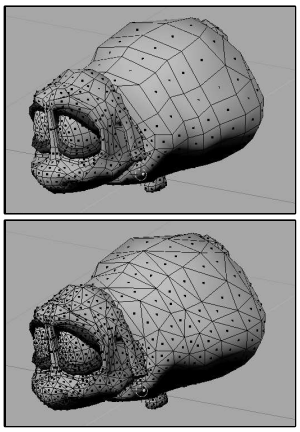


Figura 7.14: Modelo convertido a Malla de Triángulos.

La exportación desde Blender se realiza empleando el script en Python mostrado en el Listado 7.1. Para su correcta ejecución, se debe editar el valor de las variables definidas en las líneas 7 y 8 con el *path* de exportación y el número de frames a guardar en cada caso. Este script puede mejorarse en muchas líneas de trabajo; incorporando información sobre los vectores normales de las caras y vértices (necesarias para iluminación), color base del material, etc.

Para su utilización, es necesario abrir una ventana de tipo Text Editor, cargar el script (en la cabecera de la ventana, *Text/ Open*) y ejecutarlo situando el puntero del ratón en la ventana con Alt P. Si no arroja ningún error, es porque se ha ejecutado correctamente y tendremos en la ruta especificada en el script el fichero .orj (que tendrá el mismo nombre que el objeto a exportar).

Aunque el exportador del formato OREj está preparado para trabajar con caras triangulares y cuadradas, el importador sólo soporta las primeras. Así, es necesario convertir los modelos previamente a conjuntos de triángulos. Para ello, en modo de *Edición*, con todas las caras seleccionadas A pulsamos Control T (o accedemos al menú de la cabecera de la ventana 3D *Mesh/ Faces/ Convert Quads to Triangles*). La Figura 7.14 muestra un ejemplo de un modelo de *Yo Frankie* convertido a malla de triángulos.

Listado 7.1: Script de Exportación OREj

```
import Blender
import math, sys
from Blender import NMesh, Object, Scene
from math import *

# Edita las siguientes variables
PATH ="/home/carlos/"  # La ruta donde se guardara el OREj
    exportado
nframes = 50           # Numero de frames a exportar (animaciones)

object = Object.GetSelected()  # Se exporta el primer objeto selec.
objname = object[0].name
meshname = object[0].data.name
mesh = NMesh.GetRaw(meshname)

filename = PATH + objname + ".orj"
file = open(filename, "w")
std = sys.stdout               # Todo lo escrito en stdout se
    guarda
sys.stdout = file

print "# Objeto OREj:", objname
print "# Vertices Totales:", len(mesh.verts)
print "# Caras Totales:", len(mesh.faces)

# Escribimos toda la lista de vertices (v x y z)
for vertex in mesh.verts:
    x, y, z = vertex.co
    print "v %f %f %f" % (x, y, z)
# Escribimos toda la lista de caras (f v1 v2 v3 ...)
for face in mesh.faces:
    print "f",
    face.v.reverse()    # Invertimos las normales
    for vertex in face.v:
        print vertex.index + 1,
    print
# Lista de parametros uv de cada cara (t u1 v1 u2 v2...)
for face in mesh.faces:
    nvertex = 0
    face.v.reverse()
    print "t",
    for vertex in face.v:
        print face.uv[nvertex][0], face.uv[nvertex][1],
        nvertex = nvertex + 1
    print

# Exportamos para cada frame de la lista de frames la animacion
scn= Scene.GetCurrent()
context=scn.getRenderingContext()

for f in range(nframes):
    context.cFrame=f
    matrix = object[0].getMatrix()
    print "# FRAME ",f
    # Escribimos la matriz de transformacion para ese frame
    print "m  %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f" %
    (matrix[0][0],matrix[0][1],matrix[0][2],matrix[0][3],
     matrix[1][0],matrix[1][1],matrix[1][2],matrix[1][3],
     matrix[2][0],matrix[2][1],matrix[2][2],matrix[2][3],
     matrix[3][0],matrix[3][1],matrix[3][2],matrix[3][3])

sys.stdout = std
file.close()
```

7.4.3. Texturas: Formato PPM

El importador que se define en la sección 7.5 permite cargar un formato de imágenes muy sencillo con especificación ASCII llamado PPM. Este formato, como se define en las páginas *man* es una especificación altamente ineficiente para especificar imágenes. No incorpora ningún modo de compresión de datos. Sin embargo es muy sencillo leer y escribir imágenes en este formato, por lo que es la alternativa que utilizaremos en nuestro cargador.

El formato puede ser importado y exportado desde GIMP. Bastará con abrir la imagen que hemos exportado desde Blender con GIMP, y guardarla en PPM especificando ASCII como modo de *Formateado de datos*.

```
P3
# Creado con GIMP
256 256
255 113 113
113 111 111
111 113 111 ...
```

Figura 7.15: Ejemplo de fichero PPM.

En la Figura 7.15 se muestra un fragmento de formato PPM, correspondiente a la textura aplicada en el cubo que se utilizó en la Figura 7.10. La primera línea define el *número mágico* que identifica al fichero PPM (siempre es P3). A continuación hay un comentario con el nombre del creador del programa. Los dos números de la siguiente fila se corresponden con la resolución en horizontal y vertical en píxeles de la imagen (en este caso 256x256). Finalmente se almacena la lista de valores de color RGB (de 0 a 255) para cada píxel (por ejemplo, el primer píxel de la imagen tiene el valor 255 en R, y 113 para G y B).

7.5. Importación y Visualización

Para la carga y visualización del formato anterior se ha desarrollado una pequeña biblioteca que da soporte para la carga y visualización del formato OREj. El objeto de esta implementación es crear una base sobre la que el lector pueda construir su propio formato. Las funciones de carga incorporan mecanismos muy básicos de control de errores en la entrada (se suponen ficheros bien formados).

Para la carga de ficheros en formato OREj (ver Listado 7.2) simplemente hay que llamar a la función `cargarObjeto` (línea 37), pasándole como primer argumento un puntero a una zona de memoria reservada de tipo OrjObjeto (línea 10), a continuación las rutas a los ficheros de geometría `.orj` y de textura `.ppm`. Los dos últimos parámetros se corresponden con el tipo de animación (que estudiaremos a continuación) y el factor de escala.

El **factor de escala** permite utilizar un modelo asociado a diferentes tamaños de marca, o simplemente trabajar en unidades de Blender más cómodamente. De este modo, un objeto modelado en unidades normalizadas (entre 0 y 1), es fácilmente adaptable a cualquier tamaño de representación final únicamente especificando el factor de escala adecuado.

Figura 7.16: Ejemplo de representación sólida.

Listado 7.2: Ejemplo de uso de importador OREj

```
#include <GL/glut.h>
#include <AR/gsub.h>
#include <AR/video.h>
#include <AR/param.h>
#include <AR/ar.h>

#include "orej.h"     // Incluimos la cabecera para cargar orej
// ==== Definicion de constantes y variables globales =========
OrjObjeto orej;
// ======== draw ================================================
static void draw( void ) {
  double  gl_para[16];   // Esta matriz 4x4 es la usada por OpenGL

  argDrawMode3D();              // Cambiamos el contexto a 3D
  argDraw3dCamera(0, 0);        // Y la vista de la camara a 3D
  glClear(GL_DEPTH_BUFFER_BIT); // Limpiamos buffer de profundidad
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LEQUAL);

  argConvGlpara(patt_trans, gl_para);   // Convertimos la matriz de
  glMatrixMode(GL_MODELVIEW);           // la marca para ser usada
  glLoadMatrixd(gl_para);               // por OpenGL

  // Mostramos el objeto con representacion solida (texturas)
  desplegarObjeto(&orej, ESOLIDO);

  glDisable(GL_DEPTH_TEST);
}
// ======== Main ================================================
int main(int argc, char **argv) {
  glutInit(&argc, argv);    // Creamos la ventana OpenGL con Glut
```

```
32   init();                    // Llamada a nuestra funcion de inicio
33
34   cargarObjeto(&orej, "mod/cubo.orj", "mod/cubo.ppm", NOANIM, 100);
35
36   arVideoCapStart();         // Creamos un hilo para captura de
         video
37   argMainLoop( NULL, keyboard, mainLoop );   // Asociamos
         callbacks...
38   return (0);
39 }
```

La función de dibujado (línea 25) permite desplegar fácilmente el objeto, especificando el tipo de representación deseada. La versión actual de la biblioteca soporta representación sólida (con texturas, ver Figura 7.16) y wireframe (con un color base transparente, como el ejemplo mostrado en la Figura 7.17).

Figura 7.17: Ejemplo de representación wireframe.

A continuación estudiaremos el tipo de datos asociado al importador del formato OREj. En el Listado 7.3 se muestra el fichero de cabecera donde se define la estructura base `OrjObjeto` (líneas 38–56). En este listado se describen brevemente las propiedades de cada uno de los campos.

Listado 7.3: Fichero de cabecera del importador OREj

```
1  #include <GL/gl.h>
2  #include <GL/glu.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  // ======== definicion de constantes ============================
8  #define MAXPUTOS 50000    // Numero maximo de vertices del modelo
9  #define MAXLINEA 2000     // Numero maximo de aristas del modelo
10 #define MAXFRAMES 200     // Numero maximo de frames animacion
11
12 #define SDIRECTO  1       // Sentido de reproduccion anim. normal
13 #define SINVERSO -1       // Sentido de reproduccion anim. inverso
14
15 #define NOANIM    0       // Sin animacion
16 #define MSIMPLE   1       // Una iteracion en la reproduccion
17 #define MBUCLE    2       // Bucle infinito con sentido normal
18 #define MPINGPONG 3       // Reproduccion efecto "ping-pong"
19
20 #define EWIREFRAME 1      // Estilo de representacion alambrico
21 #define ESOLIDO    0     // Estilo de representacion solido
22
23 typedef struct{  // ======== OrjVertice ==========================
24   float x;       // Define coordenadas para un vertice del modelo
25   float y;
26   float z;
27 } OrjVertice;
28
29 typedef struct{  // ======== OrjCara ==============================
30   int v1;        // Indices de vertices (indexado!), coordenadas UV
31   float uv1, vv1;
32   int v2;
33   float uv2, vv2;
34   int v3;
```

```
35   float uv3, vv3;
36 } OrjCara;
37
38 typedef struct{  // ======== OrjObjeto ============================
39   // GEOMETRIA Propiedades de Geometria (vertices, caras...)
40   OrjCara    caras[MAXPUTOS];         // Vector de caras
41   OrjVertice vertices[MAXPUTOS];      // Vector de vertices
42   int        ncaras;                  // Numero de caras
43   int        nvertices;               // Numero de vertices
44   int        listmesh;                // Para usar glCallList
45   float      escala;                  // Escala de dibujado
46   // TEXTURA Todos los modelos tienen asociada una textura
47   int        textura;                 // Identificador de Textura
48   int        anchotex;                // Ancho (pixeles) Textura
49   int        altotex;                 // Alto (pixeles) Textura
50   // ANIMACION Propiedades de animacion
51   float      anim[MAXFRAMES][16];     // Vector de matrices 4x4
52   int        nframes;                 // Numero de frames anim.
53   int        frame;                   // Frame actual
54   int        modo;                    // Modo de animacion
55   int        sentido;                 // Sentido de reproduccion
56 } OrjObjeto;
57
58 // ======== cargarObjeto ========================================
59 // obj: Puntero a zona de memoria reservada de tipo OrjObjeto
60 // rutaorj: Ruta del fichero .orj
61 // rutatex: Ruta del fichero de textura .ppm de textura asociado
62 // modo: Modo de reproducir la animacion. Puede ser:
63 // NOANIM (Sin Animacion). Se ignora el valor de "sentido"
64 // MSIMPLE (Simple). Reproduce una unica vez la animacion.
65 // MBUCLE (Bucle). Bucle infinito de reproduccion.
66 // MPINGPONG. Como MBUCLE pero vuelve en sentido inverso
67 // escala: Escala de representacion (uniforme en los 3 ejes).
68 // ==============================================================
69 void cargarObjeto(OrjObjeto *obj, char *rutaorj, char *rutatex,
70           int modo, float escala);
71
72 // ======== desplegarObjeto =====================================
73 // obj: Puntero a zona de memoria reservada de tipo OrjObjeto
74 // estilo: Estilo de representacion. Puede ser:
75 // EWIREFRAME: Muestra el objeto en modo alambrico.
76 // ESOLIDO: Muestra el objeto con texturas en modo solido
77 // ==============================================================
78 void desplegarObjeto(OrjObjeto *obj, int estilo);
```

Finalmente se muestra el código completo del cargador del formato OREj, que rellena los campos de la estructura `OrjObjeto`. Hay multitud de caminos para mejorar este cargador: reserva de memoria dinámica, comprobación de errores, soporte de nuevos modos de representación, carga de varias animaciones por objeto, definición de jerarquías de objetos, y un largo etcétera.

Queda como ejercicio propuesto al intrépido programador mejorar y ampliar este formato y, naturalmente, publicar los resultados bajo la misma licencia que todos los ejemplos de este documento (GPL v3 o posterior).

Listado 7.4: Implementación del cargador de formato OREj

```
#include "orej.h"
#include <AR/ar.h>

// ======== getNewTime y resetTime ==========================
// Funciones de temporizacion (parte dependiente de ARToolKit)
double getNewTime() { return (arUtilTimer()); }
void resetTime() {  arUtilTimerReset(); }

// ======== cargarTextura ====================================
void cargarTextura(OrjObjeto *obj, char *ruta){
  GLubyte *imagen;
  FILE *fich;
  char buf[256];
  int i, j, c, w, h;

  if(obj==NULL) return;

  if(!(fich = fopen (ruta, "r"))){
    printf("no se encuentra el archivo %s\n", ruta);
    exit(-1);
  }

  fgets (buf, 255, fich);  fgets (buf, 255, fich);
  fscanf (fich, "%d %d\n", &obj->anchotex, &obj->altotex);
  fgets (buf, 255, fich); // Saltamos la linea del limite
  imagen = malloc (sizeof(GLubyte)*obj->anchotex*obj->altotex*3);

  for (i=0, w=obj->anchotex, h=obj->altotex; i<(w*h*3); i++)
    imagen[(w*h*3)-(w*3*((i/(w*3))+1))+(i%(w*3))] =
      (GLubyte) atoi(fgets(buf, 255, fich));

  glGenTextures(1, &(obj->textura));
  glPixelStorei(GL_UNPACK_ALIGNMENT,1);
  glBindTexture(GL_TEXTURE_2D, obj->textura);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
      ;
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
      ;
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGB,
               GL_UNSIGNED_BYTE, imagen);

  fclose (fich);
  free(imagen);
}

// ======== cargarObjeto =====================================
void cargarObjeto(OrjObjeto *obj, char *rutaorj, char *rutatex,
          int modo, float escala)
{
  int   i=0, linea=0;  char  cadaux[MAXLINEA];    FILE  *fich;
  float a,b,c,d,e,f;   float m[16];

  // Inicializacion de algunos parametros de la estructura ===
  obj->nframes = 0;         obj->frame = 0;        obj->modo = modo
      ;
  obj->sentido = SDIRECTO;  obj->escala = escala;  resetTime();

  if(!(fich=fopen(rutaorj, "r")))        // Abrimos el fichero .orj
    { printf("Error: %s\n", rutaorj);  exit(-1); }
```

```
59
60  // Parseamos el fichero de entrada y guardamos los datos en .ºbj"
61  while (fgets(cadaux, MAXLINEA, fich)!=NULL) {
62    linea++;
63    switch(cadaux[0]){
64    case '#': break;
65    case 'v': sscanf(&cadaux[2], "%f %f %f",   // Vertices
          (posicion)
66              &obj->vertices[obj->nvertices].x,
67              &obj->vertices[obj->nvertices].y,
68              &obj->vertices[obj->nvertices].z);
69      obj->nvertices++;
70      break;
71    case 'f': a = b = c = -1;          // Caras (indices de
          vertices)
72      sscanf(&cadaux[2], "%f %f %f",  &a, &b, &c);
73      obj->caras[obj->ncaras].v1 = (int) --a;
74      obj->caras[obj->ncaras].v2 = (int) --b;
75      obj->caras[obj->ncaras].v3 = (int) --c;
76      obj->ncaras++;
77      break;
78    case 't': a = b = c = d = e = f = -1;    // Coordenadas
          UVMapping
79      if(i>obj->ncaras) {
80    printf("Error formato en linea: '%d'\n", linea);
81    printf("valor uv no asociado a ninguna cara\n");
82    exit(-1);
83      }
84      sscanf(&cadaux[2], "%f %f %f %f %f %f", &a,&b,&c,&d,&e,&f);
85      obj->caras[i].uv1 = a;   obj->caras[i].vv1 = b;
86      obj->caras[i].uv2 = c;   obj->caras[i].vv2 = d;
87      obj->caras[i].uv3 = e;   obj->caras[i].vv3 = f;
88      i++;
89      break;
90    case 'm':             // Matrices de transformacion
          (animaciones)
91       sscanf(&cadaux[2],
92          "%f %f %f %f %f %f %f %f %f %f %f %f %f %f %f %f",
93          &m[0],&m[1],&m[2],&m[3],&m[4],&m[5],&m[6],&m[7],&m[8],
94          &m[9],&m[10],&m[11],&m[12],&m[13],&m[14],&m[15]);
95       memcpy (&obj->anim[obj->nframes], m, sizeof(float)*16);
96       obj->nframes++;
97       break;
98    }
99  }
100 fclose (fich);
101
102 cargarTextura (obj, rutatex);   // Asociar la textura al objeto
103
104 obj->listmesh=glGenLists(1);    // Utilizamos listas de
        despliegue
105 glNewList(obj->listmesh,GL_COMPILE);
106 // Para cada triangulo cargamos su coordenada de UVMapping y sus
107 // coordenadas del Sistema de Referencia Local.
108 for (i=0; i<obj->ncaras; i++) {
109   glBegin(GL_TRIANGLES);
110   glTexCoord2f(obj->caras[i].uv3, obj->caras[i].vv3);
111   glVertex3f(obj->vertices[obj->caras[i].v1].x,
112          obj->vertices[obj->caras[i].v1].y,
113          obj->vertices[obj->caras[i].v1].z);
114   glTexCoord2f(obj->caras[i].uv2, obj->caras[i].vv2);
115   glVertex3f(obj->vertices[obj->caras[i].v2].x,
```

```
116             obj->vertices[obj->caras[i].v2].y,
117             obj->vertices[obj->caras[i].v2].z);
118     glTexCoord2f(obj->caras[i].uv1, obj->caras[i].vv1);
119     glVertex3f(obj->vertices[obj->caras[i].v3].x,
120             obj->vertices[obj->caras[i].v3].y,
121             obj->vertices[obj->caras[i].v3].z);
122     glEnd();
123   }
124   glEndList();
125 }
126
127 // ======== desplegarObjeto ================================
128 void desplegarObjeto(OrjObjeto *obj, int estilo){
129   long i;
130
131   glPushMatrix();
132
133   glScalef (obj->escala, obj->escala, obj->escala);   // Escala?
134
135   // Si tiene animacion asociada, cambiamos el frame a reproducir
136   if (obj->modo != NOANIM) {
137     glMultMatrixf(obj->anim[obj->frame]);
138
139     if (getNewTime() > (1/25.0)) {   // Consideramos 25FPS siempre!
140       obj->frame += obj->sentido;
141       resetTime();
142     }
143
144     if (obj->frame == obj->nframes) {
145       switch (obj->modo) {
146       case MPINGPONG: obj->sentido *= -1; obj->frame -= 2; break;
147       case MSIMPLE: obj->frame--; break;
148       case MBUCLE: obj->frame = 0; break;
149       }
150     }
151     if (obj->frame == -1) {  // Solo posible en modo PINGPONG
152       obj->frame = 1;  obj->sentido *= -1;
153     }
154   }
155
156   // Finalmente llamamos a la lista de despliegue estableciendo
157   // las propiedades del estilo de dibujado.
158   if (estilo == EWIREFRAME) {
159     glDisable(GL_TEXTURE_2D);
160     glDisable(GL_LIGHTING);
161     glDisable(GL_DEPTH_TEST);
162     glEnable(GL_BLEND);
163     glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
164     glColor4f(0.0, 0.0, 0.8, 0.2);
165     glCallList(obj->listmesh);
166
167     glDisable(GL_BLEND);
168     glEnable(GL_DEPTH_TEST);
169     glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
170     glLineWidth(1);
171     glColor3f(1.0, 1.0, 1.0);
172     glCallList(obj->listmesh);
173     glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
174   }
175   if (estilo == ESOLIDO) {
176     glEnable(GL_TEXTURE_2D);
177     glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

```
178      glBindTexture(GL_TEXTURE_2D, obj->textura);
179      glPolygonMode(GL_FRONT, GL_FILL);
180      glCallList(obj->listmesh);
181      glDisable(GL_TEXTURE_2D);
182    }
183
184    glPopMatrix();
185  }
```

Capítulo 8

Integración con OpenCV y Ogre

En este capítulo estudiaremos primero cómo realizar una captura de vídeo y despliegue en el fondo de una ventana de Ogre para, posteriormente, convertir el tipo de datos relativos a un *Frame* en OpenCV para ser utilizados en ARToolKit (ver Figura 8.1).

8.1. Integración con OpenCV y Ogre

La clase *VideoManager* se encarga de la gestión de las fuentes de vídeo. Podrían instanciarse tantos objetos de esta clase como fueran necesarios, pero habría que modificar la implementación de la llamada a *createBackground* y a *DrawCurrentFrame*. El *VideoManager* abstrae a la aplicación de Ogre del uso de OpenCV. Únicamente el *VideoManager* conoce los dispositivos de captura (línea 10 del archivo de cabecera). El puntero al *sceneManager* de Ogre es necesario como veremos más adelante para la creación y actualización del plano que utilizaremos para desplegar la salida del dispositivo de vídeo en la escena.

El *VideoManager* trabaja internamente con dos tipos de datos para la representación de los *frames* (ver líneas 11–12), que pueden ser convertidos entre sí fácilmente. El IplImage ha sido descrito anteriormente en la sección. cv::Mat es una extensión para trabajar con matrices de cualquier tamaño y tipo en C++, que permiten acceder cómodamente a los datos de la imagen. De este modo, cuando se capture un fra-

Figura 8.1: Ejemplo de integración que se describirá en esta sección.

me, el *VideoManager* actualizará los punteros anteriores de manera inmediata.

Listado 8.1: VideoManager.h

```
1  #include <Ogre.h>
2  #include <iostream>
3  #include "cv.h"
4  #include "highgui.h"
5
6  class VideoManager {
7   private:
8    void createBackground(int cols, int rows);
9    void ReleaseCapture();
10   CvCapture* _capture;
11   IplImage* _frameIpl;
12   cv::Mat* _frameMat;
13   Ogre::SceneManager* _sceneManager;
14
15  public:
16   VideoManager(int device, int w, int h,
17                Ogre::SceneManager* sm);
18   ~VideoManager();
19   void UpdateFrame();
20   IplImage* getCurrentFrameIpl();
21   cv::Mat* getCurrentFrameMat();
22   void DrawCurrentFrame();
23 };
```

El constructor de la clase (líneas 4-8 del siguiente listado) se encarga de obtener el dispositivo de captura e inicializarlo con las dimensiones especificadas como parámetro al constructor. Esto es necesario, ya que las cámaras generalmente permiten trabajar con varias resoluciones. En el destructor se libera el dispositivo de captura creado por OpenCV (línea 12).

Listado 8.2: VideoManager.cpp

```
1  #include "VideoManager.h"
2  VideoManager::VideoManager(int device, int w, int h,
3                 Ogre::SceneManager* sm){
4    _sceneManager = sm;
5    _capture = cvCreateCameraCapture(device);
6    cvSetCaptureProperty(_capture, CV_CAP_PROP_FRAME_WIDTH, w);
7    cvSetCaptureProperty(_capture, CV_CAP_PROP_FRAME_HEIGHT, h);
8    createBackground (w, h); _frameIpl = NULL; _frameMat = NULL;
9  }
10
11 VideoManager::~VideoManager(){
12   cvReleaseCapture(&_capture); delete _frameIpl; delete _frameMat;
13 }
14 // ================================================================
15 // createBackground: Crea el plano sobre el que dibuja el video
16 void VideoManager::createBackground(int cols, int rows){
17   Ogre::TexturePtr texture=Ogre::TextureManager::getSingleton().
18     createManual("BackgroundTex", // Nombre de la textura
19     Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
20     Ogre::TEX_TYPE_2D,   // Tipo de la textura
21     cols, rows, 0,       // Filas, columas y Numero de Mipmaps
22     Ogre::PF_BYTE_BGRA,
23     Ogre::HardwareBuffer::HBU_DYNAMIC_WRITE_ONLY_DISCARDABLE);
24
25   Ogre::MaterialPtr mat = Ogre::MaterialManager::getSingleton().
26     create("Background",
27           Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
28   mat->getTechnique(0)->getPass(0)->createTextureUnitState();
29   mat->getTechnique(0)->getPass(0)->setDepthCheckEnabled(false);
30   mat->getTechnique(0)->getPass(0)->setDepthWriteEnabled(false);
31   mat->getTechnique(0)->getPass(0)->setLightingEnabled(false);
32   mat->getTechnique(0)->getPass(0)->getTextureUnitState(0)->
33     setTextureName("BackgroundTex");
34
35   // Creamos un rectangulo que cubra toda la pantalla
36   Ogre::Rectangle2D* rect = new Ogre::Rectangle2D(true);
37   rect->setCorners(-1.0, 1.0, 1.0, -1.0);
38   rect->setMaterial("Background");
39
40   // Dibujamos el background antes que nada
41   rect->setRenderQueueGroup(Ogre::RENDER_QUEUE_BACKGROUND);
42
43   Ogre::SceneNode* node = _sceneManager->getRootSceneNode()->
44     createChildSceneNode("BackgroundNode");
45   node->attachObject(rect);
46 }
47 // ================================================================
48 // UpdateFrame: Actualiza los punteros de frame Ipl y frame Mat
49 void VideoManager::UpdateFrame(){
50   _frameIpl = cvQueryFrame(_capture);
51   _frameMat = new cv::Mat(_frameIpl);
```

```
52 }
53 // = IplImage* getCurrentFrameIpl ================================
54 IplImage* VideoManager::getCurrentFrameIpl(){ return _frameIpl; }
55 // = IplImage* getCurrentFrameMat ================================
56 cv::Mat* VideoManager::getCurrentFrameMat(){  return _frameMat; }
57 // ===============================================================
58 // DrawCurrentFrame: Despliega el ultimo frame actualizado
59 void VideoManager::DrawCurrentFrame(){
60   if(_frameMat->rows==0) return;
61   Ogre::TexturePtr tex = Ogre::TextureManager::getSingleton().
62      getByName("BackgroundTex",
63      Ogre::ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME);
64   Ogre::HardwarePixelBufferSharedPtr pBuffer = tex->getBuffer();
65
66   pBuffer->lock(Ogre::HardwareBuffer::HBL_DISCARD);
67   const Ogre::PixelBox& pixelBox = pBuffer->getCurrentLock();
68
69   Ogre::uint8* pDest = static_cast<Ogre::uint8*>(pixelBox.data);
70   for(int j=0;j<_frameMat->rows;j++) {
71     for(int i=0;i<_frameMat->cols;i++) {
72       int idx = ((j) * pixelBox.rowPitch + i )*4;
73       pDest[idx] = _frameMat->data[(j*_frameMat->cols+i)*3];
74       pDest[idx+1] = _frameMat->data[(j*_frameMat->cols+i)*3+1];
75       pDest[idx+2] = _frameMat->data[(j*_frameMat->cols+i)*3+2];
76       pDest[idx+3] = 255;
77     }
78   }
79   pBuffer->unlock();
80   Ogre::Rectangle2D* rect = static_cast<Ogre::Rectangle2D*>
81     (_sceneManager->getSceneNode("BackgroundNode")->
82         getAttachedObject(0));
83 }
```

El método *createBackground* se encarga de crear un plano sobre el que se actualizará el vídeo obtenido de la cámara. Además de las propiedades relativas a desactivar el uso del *depthbuffer* o la iluminación (líneas 29-31), es importante indicar que el buffer asociado a ese material va a ser actualizado muy frecuentemente, mediante la constante de uso del buffer definida en la línea 23). Posteriormente crearemos un rectángulo 2D (coordenadas paramétricas en la línea 37), que añadiremos al grupo que se dibujará primero (línea 41). Finalmente se añade el nodo, con el plano y la textura, a la escena (líneas 43-45) empleando el puntero al *SceneManager* indicado en el constructor.

La actualización de la textura asociada al plano se realiza en el método *DrawCurrentFrame*, que se encarga de acceder a nivel de píxel y copiar el valor de la imagen obtenida por la webcam. Para ello se obtiene un puntero compartido al *pixel buffer* (línea 64), y obtiene el uso de la memoria con exclusión mutua (línea 67). Esa región será posteriormente liberada en la línea 79. La actualización de la región se realiza recorriendo el frame en los bucles de las líneas 70-77, y especificando el color de cada píxel en valores RGBA (formato de orden de componentes configurado en línea 22).

El frame obtenido será utilizado igualmente por ARToolKit para detectar las marcas y proporcionar la transformación relativa. Igualmente se ha definido una clase para abstraer del uso de ARToolKit a la

aplicación de Ogre. El archivo de cabecera del *ARTKDetector* se muestra a continuación.

Listado 8.3: ARTKDetector.h

```
#include <AR/ar.h>
#include <AR/gsub.h>
#include <AR/param.h>
#include <Ogre.h>
#include <iostream>
#include "cv.h"

class ARTKDetector {
 private:
  ARMarkerInfo *_markerInfo;
  int _markerNum;
  int _thres;
  int _pattId;
  int _width; int _height;
  double _pattTrans[3][4];
  bool _detected;

  int readConfiguration();
  int readPatterns();
  void Gl2Mat(double *gl, Ogre::Matrix4 &mat);

 public:
  ARTKDetector(int w, int h, int thres);
  ~ARTKDetector();
  bool detectMark(cv::Mat* frame);
  void getPosRot(Ogre::Vector3 &pos, Ogre::Vector3 &look,
         Ogre::Vector3 &up);
};
```

Complétame!

Sería conveniente añadir nuevos métodos a la clase ARTKDetector, que utilice una clase que encapsule los marcadores con su ancho, alto, identificador de patrón, etc...

En este sencillo ejemplo únicamente se han creado dos clases para la detección de la marca *detectMark*, y para obtener los vectores de posicionamiento de la cámara en función de la última marca detectada (*getPosRot*).

Listado 8.4: ARTKDetector.cpp

```
#include "ARTKDetector.h"

ARTKDetector::ARTKDetector(int w, int h, int thres){
  _markerNum=0;   _markerInfo=NULL;  _thres = thres;
  _width = w;     _height = h;       _detected = false;
  readConfiguration();
  readPatterns();
}
ARTKDetector::~ARTKDetector(){  argCleanup();  }
// ==========================================================
// readPatterns: Carga la definicion de patrones
int ARTKDetector::readPatterns(){
  if((_pattId=arLoadPatt("data/simple.patt")) < 0) return -1;
  return 0;
}
// ==========================================================
// readConfiguration: Carga archivos de configuracion de camara...
int ARTKDetector::readConfiguration(){
  ARParam  wparam, cparam;
  // Cargamos los parametros intrinsecos de la camara
```

```
21   if(arParamLoad("data/camera_para.dat",1, &wparam) < 0) return -1;
22
23   arParamChangeSize(&wparam, _width, _height, &cparam);
24   arInitCparam(&cparam);   // Inicializamos la camara con çparam"
25   return 0;
26 }
27 // ============================================================
28 // detectMark (FIXME): Ojo solo soporta una marca de tamano fijo!
29 bool ARTKDetector::detectMark(cv::Mat* frame) {
30   int j, k;
31   double p_width     = 120.0;        // Ancho de marca... FIJO!
32   double p_center[2] = {0.0, 0.0};   // Centro de marca.. FIJO!
33
34   _detected = false;
35   if(frame->rows==0) return _detected;
36   if(arDetectMarker(frame->data, _thres,
37                     &_markerInfo, &_markerNum) < 0){
38     return _detected;
39   }
40   for(j=0, k=-1; j < _markerNum; j++) {
41     if(_pattId == _markerInfo[j].id) {
42       if (k == -1) k = j;
43       else if(_markerInfo[k].cf < _markerInfo[j].cf) k = j;
44     }
45   }
46   if(k != -1) {   // Si ha detectado el patron en algun sitio...
47     arGetTransMat(&_markerInfo[k], p_center, p_width, _pattTrans);
48     _detected = true;
49   }
50   return _detected;
51 }
52 // ============================================================
53 // Gl2Mat: Utilidad para convertir entre matriz OpenGL y Matrix4
54 void ARTKDetector::Gl2Mat(double *gl, Ogre::Matrix4 &mat){
55   for(int i=0;i<4;i++) for(int j=0;j<4;j++) mat[i][j]=gl[i*4+j];
56 }
57 // ============================================================
58 // getPosRot: Obtiene posicion y rotacion para la camara
59 void ARTKDetector::getPosRot(Ogre::Vector3 &pos,
60                    Ogre::Vector3 &look, Ogre::Vector3 &up){
61   if (!_detected) return;
62
63   double glAuxd[16]; Ogre::Matrix4 m;
64   argConvGlpara(_pattTrans,glAuxd);
65   Gl2Mat(glAuxd, m);   // Convertir a Matrix4 de Ogre
66
67   m[0][1]*=-1; m[1][1]*=-1;  m[2][1]*=-1; m[3][1]*=-1;
68   m = m.inverse();
69   m = m.concatenate(Ogre::Matrix4(
70       Ogre::Quaternion(Ogre::Degree(90), Ogre::Vector3::UNIT_X)));
71   pos  = Ogre::Vector3 (m[3][0],         m[3][1],        m[3][2]);
72   look = Ogre::Vector3 (m[2][0]+m[3][0], m[2][1]+m[3][1],
73                                                m[2][2]+m[3][2]);
74   up   = Ogre::Vector3 (m[1][0],         m[1][1],        m[1][2]);
75 }
```

Esta clase está preparada para trabajar con una única marca (cargada en el método privado *readPatterns*), con sus variables asociadas definidas como miembros de la clase ARTKDetector (líneas 11-16). El siguiente listado implementa el *ARTKDetector*.

El método *getPosRot* es el único que vamos a comentar (el resto son una traducción directa de la funcionalidad estudiada anteriormente). En la línea (65) se utiliza un método de la clase para transformar la matriz de OpenGL a una *Matrix4* de Ogre. Ambas matrices son matrices columna!, por lo que habrá que tener cuidado a la hora de trabajar con ellas (el campo de traslación viene definido en la fila inferior, y no en la columna de la derecha como es el *convenio habitual*).

Las operaciones de la línea (67) sirven para invertir el eje Y de toda la matriz, para cambiar el criterio del sistema de ejes de *mano izquierda* a *mano derecha* (Ogre). Invertimos la matriz y aplicamos una rotación de 90 grados en X para tener el mismo sistema con el que hemos trabajado en sesiones anteriores del curso.

Finalmente calculamos los vectores de *pos*, *look* y *up* (líneas (71–74)) que utilizaremos en el bucle de dibujado de Ogre.

El uso de estas clases de utilidad es directa desde el FrameListener. Será necesario crear dos variables miembro del *VideoManager* y *ARTKDetector* (líneas (7–8)). En el método *frameStarted* bastará con llamar a la actualización del frame (líneas (15–16)). La detección de la marca se realiza pasándole el puntero de tipo Mat del *VideoManager* al método *detectMark* del *Detector*.

Listado 8.5: MyFrameListener.cpp

```
1  #include "MyFrameListener.h"
2
3  MyFrameListener::MyFrameListener(Ogre::RenderWindow* win,
4                 Ogre::Camera* cam, Ogre::SceneNode *node,
5                 Ogre::OverlayManager *om, Ogre::SceneManager *sm) {
6    // Omitido el resto del codigo...
7    _videoManager = new VideoManager(1, 640, 480, _sceneManager);
8    _arDetector = new ARTKDetector(640, 480, 100);
9  }
10
11 bool MyFrameListener::frameStarted(const Ogre::FrameEvent& evt) {
12   // Omitido el resto del codigo...
13   Ogre::Vector3 pos;  Ogre::Vector3 look;   Ogre::Vector3 up;
14
15   _videoManager->UpdateFrame();
16   _videoManager->DrawCurrentFrame();
17   if (_arDetector->detectMark(_videoManager->getCurrentFrameMat()))
         {
18     _arDetector->getPosRot(pos, look, up);
19     _camera->setPosition(pos);
20     _camera->lookAt(look);
21     _camera->setFixedYawAxis(true, up);
22   }
23 }
```

Bibliografía

[1] Ronald Azuma. A survey of augmented reality. *Presence*, 6(4):355–385, 1997.

[2] Denis Chekhlov, Mark Pupilli, Walterio Mayol, and Andrew Calway. Robust real-time visual slam using scale prediction and exemplar based feature description. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 0:1–7, 2007.

[3] Adrian David Cheok, Kok Hwee Goh, Wei Liu, Farzam Farbiz, Sze Lee Teo, Hui Siang Teo, Shang Ping Lee, Yu Li, Siew Wan Fong, and Xubo Yang. Human pacman: a mobile wide-area entertainment system based on physical, social, and ubiquitous computing. In *ACE '04: Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 360–361, New York, NY, USA, 2004. ACM.

[4] Ben Close, John Donoghue, John Squires, Phillip De Bondi, and Michael Morris. Arquake: An outdoor-indoor augmented reality first person application. In *In 4th International Symposium on Wearable Computers*, pages 139–146, 2000.

[5] Ingemar J. Cox and Sunita L. Hingorani. An efficient implementation of reid's multiple hypothesis tracking algorithm and its evaluation for the purpose of visual tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18:138–150, 1996.

[6] Tom Drummond and Roberto Cipolla. Real-time visual tracking of complex structures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:932–946, 2002.

[7] Steven Feiner, Blair MacIntyre, Tobias Hollerer, and Anthony Webster. A touring machine: Prototyping 3d mobile augmented reality systems for exploring the urban environment. *Wearable Computers, IEEE International Symposium*, 0:74, 1997.

[8] Mark Fiala. Artag, a fiducial marker system using digital techniques. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 2:590–596, 2005.

[9] Frederic Jurie and Michel Dhome. Hyperplane approximation for template matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:996–1000, 2002.

[10] Hirokazu Kato and Mark Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. *Augmented Reality, International Workshop on*, 0:85, 1999.

[11] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. *Mixed and Augmented Reality, IEEE / ACM International Symposium on*, 0:1–10, 2007.

[12] R. Koch, K. Koeser, and B. Streckel. Markerless image based 3d tracking for real-time augmented reality applications. In *International Workshop of Image Analysis for Multimedia Interactive Services*, pages 45–51, 2005.

[13] Yong Liu, Moritz Starring, Thomas B. Moeslund, Claus B. Madsen, and Erik Granum. Computer vision based head tracking from re-configurable 2d markers for ar. *Mixed and Augmented Reality, IEEE / ACM International Symposium on*, 0:264, 2003.

[14] D. Marimon, Y. Maret, Y. Abdeljaoued, and T. Ebrahimi. Particle filter-based camera tracker fusing marker and feature point cues. In *Proc. of IS&T/SPIE Conf. on Visual Communications and Image Processing*. Citeseer, 2007.

[15] David Marimon. *Advances in Top-Down and Bottom-Up Approaches to Video-Based Camera Tracking*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2007.

[16] Paul Milgram and Fumio Kishino. A taxonomy of mixed reality visual displays. In *IEICE Trans. Information Systems*, number 12 in E77-D, pages 1321–1329, 1994.

[17] N. Molton, A.J. Davison, and I.D. Reid. Locally planar patch features for real-time structure from motion. In *British Machine Vision Conference*, pages 91–101, 2004.

[18] Leonid Naimark and Eric Foxlin. Circular data matrix fiducial system and robust image processing for a wearable vision-inertial self-tracker. *Mixed and Augmented Reality, IEEE / ACM International Symposium on*, 0:27, 2002.

[19] Hesam Najafi, Nassir Navab, and Gudrun Klinker. Automated initialization for marker-less tracking: A sensor fusion approach. *Mixed and Augmented Reality, IEEE / ACM International Symposium on*, 0:79–88, 2004.

[20] Mark Pupilli and Andrew Calway. Real-time camera tracking using known 3d models and a particle filter. *Pattern Recognition, International Conference on*, 1:199–203, 2006.

[21] Gang Qian and Rama Chellappa. Structure from motion using sequential monte carlo methods. *Computer Vision, IEEE International Conference on*, 2:614, 2001.

[22] J. Rekimoto. Matrix: A realtime object identification and registration method for augmented reality. *Asia-Pacific Computer and Human Interaction*, 0:63, 1998.

[23] Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition)*. Addison-Wesley, 2009.

[24] Gilles Simon, Andrew W. Fitzgibbon, and Andrew Zisserman. Markerless tracking using planar structures in the scene. *Augmented Reality, International Symposium on*, 0:120, 2000.

[25] Ivan E. Sutherland. A head-mounted three dimensional display. In *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 757–764, New York, NY, USA, 1968. ACM.

[26] Luca Vacchetti, Vincent Lepetit, and Pascal Fua. Combining edge and texture information for real-time accurate 3d camera tracking. *Mixed and Augmented Reality, IEEE / ACM International Symposium on*, 0:48–57, 2004.

[27] Cor J. Veenman, Marcel J.T. Reinders, and Eric Backer. Resolving motion correspondence for densely moving points. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:54–72, 2001.

[28] Vassilios Vlahakis, John Karigiannis, Manolis Tsotros, Michael Gounaris, Luis Almeida, Didier Stricker, Tim Gleue, Ioannis T. Christou, Renzo Carlucci, and Nikos Ioannidis. Archeoguide: first results of an augmented reality, mobile computing system in cultural heritage sites. In *VAST '01: Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage*, pages 131–140, New York, NY, USA, 2001. ACM.

Este libro fue maquetado mediante el sistema de composición de textos LaTeX utilizando software del proyecto GNU.

Ciudad Real, a 25 de Julio de 2012.