UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

MARS: Sistema de Tracking Híbrido Modular para Realidad Aumentada

Sergio Pérez Camacho

Febrero, 2011

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Tecnologías y Sistemas de Información

PROYECTO FIN DE CARRERA

MARS: Sistema de Tracking Híbrido Modular para Realidad Aumentada

Autor: Sergio Pérez Camacho

Director: Carlos González Morcillo

Febrero, 2011

Sergio Pérez Camacho

E-mail: sergio.perez.camacho@gmail.com
Teléfono: (+34) 902204100 - Ext. 3746
Web site: `http://oreto.esi.uclm.es`

TRIBUNAL:

Presidente:

Vocal 1:

Vocal 2:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL 1	VOCAL 2	SECRETARIO
Fdo.:	Fdo.:	Fdo.:	Fdo.:

A mis padres y Eglė, mi realidad aumentada

Resumen

La Realidad Aumentada (RA) es un campo en auge tanto a nivel de investigación como de explotación comercial. La diversidad de los dispositivos hardware y los diferentes métodos de *tracking* utilizados en RA revelan un problema de tratamiento de la heterogeneidad. Las arquitecturas y *frameworks* existentes para el desarrollo de aplicaciones de RA habitualmente ofrecen un soporte limitado a cierto tipo de dispositivos y están enfocados a determinados métodos de *tracking*. Además, no son libres, lo que imposibilita su ampliación o adaptación para ciertos ámbitos de aplicación, como la combinación de varios métodos de *tracking*, el filtrado o el uso de diferentes fuentes de vídeo.

Como solución a este problema y con motivo de facilitar la creación de aplicaciones de RA de manera rápida, se propone el presente proyecto fin de carrera: MARS, una arquitectura modular, híbrida y libre, que homogeneiza el acceso a los dispositivos de vídeo y proporciona interfaces genéricos y homogéneos para la creación de filtros y métodos de *tracking*.

MARS implementa un sistema de representación de objetos sintéticos 3-D y de interfaz de usuario y suministra un conjunto de clases matemáticas para realizar transformaciones de cuerpo rígido en el espacio tridimensional. Igualmente facilita la reproducción de sonidos y de voz sintetizada, y permite el uso de un lenguaje de *script* para ampliar la funcionalidad de una aplicación o cambiar su comportamiento sin la necesidad de recompilarla.

Abstract

Augmented Reality (AR) is a booming field at both research and commercial exploitation. The diversity of hardware devices and tracking methods used in AR reveal a heterogeneity treatment problem. Existing architectures and frameworks used for the development of AR applications tipically offer support only for specific devices and certain tracking methods. Also, they are not free (as in freedom), thus preventing their expansion and adaptation to some application areas such as combining various tracking methods, filtering or the use of different video sources.

This Final Project (MARS) is proposed to solve this problem and to allow the quick creation of AR applications. MARS is a modular, hybrid and free architecture designed to provide homogenized access to video devices and generic and uniform interfaces for the creation of filters and tracking methods.

MARS implements a real-time rendering system for the representation of synthetic 3D objects and graphic interfaces, and provides a set of mathematical classes to facilitate the transformation of rigid bodies in three-dimensional space. It also facilitates sound and synthesized speech reproduction, and allows the use of a script language to extend the funcionality or to modify the behavior of an application without recompiling it.

Índice general

Índice de figuras

Índice de listados

Listado de acrónimos

RA	Realidad Aumentada
RV	Realidad Virtual
HMD	*Head Mounted Display*
2-D	Dos Dimensiones
3-D	Tres Dimensiones
FSF	Free Software Foundation
API	*Application Programming Interface*
POSIX	*Portable Operating System Interface*
STL	*Standard Template Library*
XML	*Extensible Markup Language*
DOM	*Document Object Model*
IP	*Internet Protocol*
RTSP	*Real Time Streaming Protocol*
DOM	*Document Object Model*
GUI	*Graphic User Interface*
XUL	*XML User interface Language*

Agradecimientos

Quisiera agradecer al director de este proyecto, Carlos G. Morcillo, y al resto del grupo ORETO y ARCO, por su ayuda y amistad.

Gracias a mis padres y a Eglė por su paciencia y apoyo continuo.

1

Introducción

En los últimos cinco años, el interés por la Realidad Aumentada (RA) ha crecido exponencialmente, tanto en el ámbito de la investigación, con el desarrollo de nuevos métodos de *tracking* y paradigmas de interacción, como en el desarrollo comercial de videojuegos y sistemas de información.

Una característica común en los sistemas de RA es el uso de diferentes técnicas de *tracking*, la capacidad de utilización de diferentes dispositivos hardware, y la inclusión de métodos de integración y corrección de errores. El tratamiento de esta heterogeneidad requiere del diseño de arquitecturas modulares que ofrezcan soporte a sistemas híbridos (en términos software y hardware) y que aislen al programador de la complejidad subyacente.

Con el propósito de paliar estas dificultades, se propone el presente proyecto fin de carrera, MARS, una arquitectura modular portable a diferentes plataformas, que facilita la construcción de aplicaciones de RA mediante el uso de diversos algoritmos y métodos de *tracking*.

1.1. Realidad Aumentada

Según Azuma [A+97], la RA es una variación de Realidad Virtual (RV). Las tecnologías de RV sumergen al usuario dentro de un entorno completamente sintético, sin tener consciencia del mundo real que lo rodea. 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 RA no sustituye la realidad, sino que la complementa.

Mientras que una aplicación de la RV introduciría al usuario en un entorno de visualización arqueológica, la RA complementaría la visión real de las mismas con objetos virtuales.

Las características que a definen un sistema de RA son:

1. Combina lo real y lo virtual.
2. Es interactivo en tiempo real.
3. Alinea y compone (registra[1]) de forma creíble los objetos sintéticos con la realidad en 3-D.

Esto descarta al cine como RA ya que puede componer escenas en 3-D de forma muy creíble pero ni son interactivas ni se generan en tiempo real. También descarta las aplicaciones que añaden sobreimpresiones 2-D encima del vídeo, puesto que no están compuestas con el mundo real en 3-D.

Sin embargo, esta definición de RA abarca aplicaciones no sólo basadas en HMDs (sistemas de visualización montados en la cabeza), sino además las basadas en pantallas, en sistemas monoculares, en *see-through*[2] HMDs y en otras tecnologías híbridas de visualización.

1.1.1. Impacto socio-económico

La RA ha experimentado un creciente interés en los últimos meses. En Septiembre de 2009, *The Economist* aseguró que "intentar imaginar cómo se utilizará la Realidad Aumentada es como internar 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 (figura 1.1). *ABI Research* hace unas previsiones más modestas, estimando esta cifra en 350 millones. Los ingresos generados por aplicaciones móviles que utilizan RA ha sido de dos millones en 2010, lo que muestra el impacto esperado de esta tecnología en los próximos cuatro años.

Según *Juniper Research*, las aplicaciones de RA se enfocarán principalmente en la búsqueda basada en localizaciones, juegos, estilo de vida y salud, educación y referencia,

[1] En inglés esta caracterísitca se conoce como *Registration*, así que se empleará el término *registro* indistintamente a partir de ahora

[2] *see-through* se refiere a sistemas de visualización que permiten ver el mundo real a través de una pantalla semi-transparente en la que se sobreimpresiona la parte sintética.

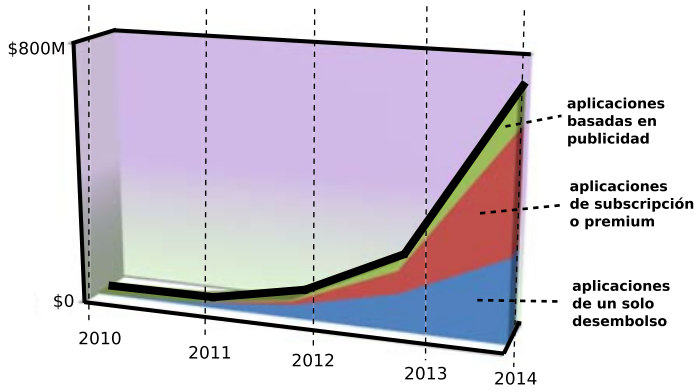

FIGURA 1.1: Previsión de ingresos de aplicaciones móviles de RA (JUNIPER RESEARCH)

multimedia y entretenimiento, redes sociales, con ganancias basadas en descargas pagadas, en cuotas de subscripción o en publicidad.

El desprendimiento de la tecnología de los equipos informáticos de sobremesa y su salto a los dispositivos móviles ha hecho de la RA una tecnología accesible y de la que se puede hacer uso en la vida cotidiana. Como dato curioso, desde Junio de 2009, las búsquedas en *Google* del los términos "*Augmented Reality*" supera a las realizadas de los términos "*Virtual Reality*".

1.1.2. Problemática

El creciente auge de esta tecnología está en parte motivado por el bajo coste del hardware que la apoya. Entre estos dispositivos se encuentran las videocámara, que cada vez están incluidas en más dispositivos móviles y también en más hogares, con motivo del uso de aplicaciones de videoconferencia.

Los métodos utilizados para extraer información de este vídeo o de dispositivos de apoyo como acelerómetros son dispares, utilizando cada uno de ellos un tipo de abstracción diferente para tratar los datos de entrada que provienen del mundo real. Además, estos métodos no están libre de errores o imprecisiones, haciendo que el uso único de uno de ellos pueda conllevar problemas en el *registro* del mundo real con los objetos sintéticos.

Todo esto, junto con la diversidad de formatos para la representación gráfica y la falta de un consenso para el desarrollo de aplicaciones de RA hace que uno de los problemas a resolver en la misma, sea la heterogeneidad de este tipo de sistemas.

1.1.3. Objetivo principal

El problema creado por los sistemas heterogéneos hace que sea conveniente el desarrollo de algún tipo de arquitectura que unifique la forma de acceso al hardware de vídeo, que unifique la forma de acceso y filtrado de los datos que provienen del mundo real a través de diferente métodos, y que facilite la representación de objetos sintéticos alineados con el mundo real.

Aunque los sistemas de representación 3-D lo suficientemente precisos (con la exactitud que un número de doble precisión proporciona) para ser utilizados en RA, los métodos de *tracking* frecuentemente conllevan de errores de medición o de problemas intrínsecos a la propia técnica utilizada. Esto ha motivado el uso conjunto de diferentes técnicas de seguimiento (*tracking*), como se sugiere y se justifica por ejemplo en el trabajo de G.Klein [Kle06], donde este aporta una solución compuesta de varios métodos de *tracking* visuales.

En este proyecto de fin de carrera se propone MARS, una arquitectura para la creación y el prototipado rápido de aplicaciones de RA, que unifica el acceso de las fuentes de vídeo, que facilita la escritura de métodos de procesado y la creación de filtros que unifiquen los resultados y que proporciona una forma de representación 3-D basada en tecnologías y formatos estándar.

1.2. Estructura del documento

Este documento se ha redactado respetando la normativa del proyecto de fin de carrera de la Escuela Superior de Informática (UCLM) [dIU07]. El contenido está dividido en los siguientes capítulos:

Capítulo 2: Antecedentes

En este capítulo se realiza un estudio del arte de los sistemas existentes, de las aplicaciones y de las tecnologías relacionadas con la RA.

Capítulo 3: Objetivos

En este capítulo se recoge el objetivo general y los objetivos concretos de este proyecto fin de carrera.

Capítulo 4: Metodología de desarrollo y herramientas

En este capítulo se describe la metodología elegida para llevar a cabo el desarrollo de este proyecto y de cuáles han sido la herramientas utilizadas en su realización.

Capítulo 5: Arquitectura de MARS

En este capítulo se detallan los resultados obtenidos de aplicar la metodología descrita en el capítulo 4. En concreto se describen los aspectos más relevantes de la arquitectura de MARS, así como los módulos y subsistemas que la componen, prestando especial atención a las decisiones de diseño e implementación tomadas.

Capítulo 6: Evolución y costes

En este capítulo se resume la evolución del proyecto y se analizan los costes asociados al mismo.

Capítulo 7: Conclusiones y Propuestas

En este capítulo se analiza el trabajo realizado y se comentan algunas propuestas acerca del trabajo futuro en relación con el mismo.

2

Antecedentes

En este capítulo se introducirán los campos y las tecnologías relacionadas con este proyecto, realizando una revisión de las mismas. Se realizará un estudio del estado del arte de los sistemas existentes.

2.1. Introducción general

En este primer punto de los antecedentes se tratarán las aplicaciones principales de la RA, se abordará el estudio de algunos de los sistemas que dan soporte a esta tecnología (sección 2.1.2) y se analizarán las tecnologías relacionadas (sección 2.1.3) extraídas de su revisión.

2.1.1. Aplicaciones de la RA

Los campos de aplicación de la RA son inumerables. Cualquier tarea que se pueda ver beneficiada por la inclusión de información en el contexto visual, es candidata para ser apoyada por RA.

Algunos de estos campos son:

- **Medicina**. Ayudando a la diagnosis, añadiendo imágenes sintéticas a una vista del paciente en tiempo real. Se pueden utilizar métodos no *intrusivos* (ultrasonidos, resonancia magnética, tomografías computerizadas, ...) para recopilar datos. Posterior-

mente estos datos se representarán en 3-D correctamente alineados sobre la imagen real del paciente, facilitando así el trabajo del personal sanitario.

- **Construcción, reparación y mantenimiento**. La capacidad de utilizar la composición de imágenes tridimensionales sobreimpresas encima de la maquinaria o del equipamiento, y la capacidad de mostrar los pasos necesarios para completar una tarea, hacen de la RA una tecnología idónea para su aplicación en este campo (ver figura 2.1).

- **Anotaciones y Visualización**. Se podría utilizar la RA para realizar cualquier tipo de anotación asociada a un objeto real. Por ejemplo, se podría reconocer el objeto que se está visualizando y añadirle información sintética a modo de etiquetas. De esto modo, una aplicación de la RA en este ámbito podría ser la inclusión de ciertos *metadatos* en la vista de un libro. Incluso se podría mostrar información más general de la biblioteca cuando el interés del usuario no se centrase en un ejemplar concreto. Las aplicaciones de visualización son múltiples. Algunos ejemplos concretos son:

  - Previsualización de una acción de restauración incluyendo en la vista real una representación sintética del resultado previsto.
  - Ayuda a la identificación y a la comprensión de objetos durante condiciones de baja visibilidad, mediante la representación de información que facilite su orientación y localización mediante trazado de líneas.
  - Identificación de personas y visualización de datos asociados a las mismas.
  - Visualización de protocolos de actuación.

- **Simulación y Planificación**. Existen tareas cuya realización sin una planificación previa pueden conllevar consecuencias inesperadas. Para realizar una planificación es a menudo necesario llevar a cabo una simulación previa, y si en esta simulación intervienen objetos reales, la RA podrá mostrar esa simulación, recreada sobre la vista del mundo real.

- **Entretenimiento**.La mayoría de los videojuegos modernos son aplicaciones software en tiempo real. Esto hace que sean claros candidatos para utilizar la RA. Usando una videocámara para capturar el mundo real, se pueden introducir elementos del juego en dicha vista. De esta manera es posible representar una escena interactiva compuesta del vídeo real y de los modelos sintéticos, creando la ilusión de, por ejemplo, dos personajes luchando encima de nuestra mesa.

En la actualidad Sony comercializa un videojuego llamado *Invizimals* [1] para su sistema portable *PSP*, en el que hace uso de la RA.

En el proyecto *ARQuake* [PT02] se utiliza un HMD con un espejo semi-transparente donde se proyectan las imágenes sintéticas 3-D del juego *Quake* (ver figura 2.1).

- **Aplicaciones Móviles**. Gracias a las *PDA*s y a los teléfonos móviles con capacidades multimedia, la RA se ha introducido en el ámbito de las aplicaciones móviles. Así, aplicaciones como *Layar* [2] utilizan la RA para incluir información en el contexto de las imágenes capturadas por un móvil en tiempo real. *Layar* superpone la localización de lugares de interés encima del vídeo. Cuando el usuario mueve su teléfono, las imágenes sintéticas cambian de acuerdo a la nueva vista, ayudando al usuario a orientarse de forma completamente intuitiva.

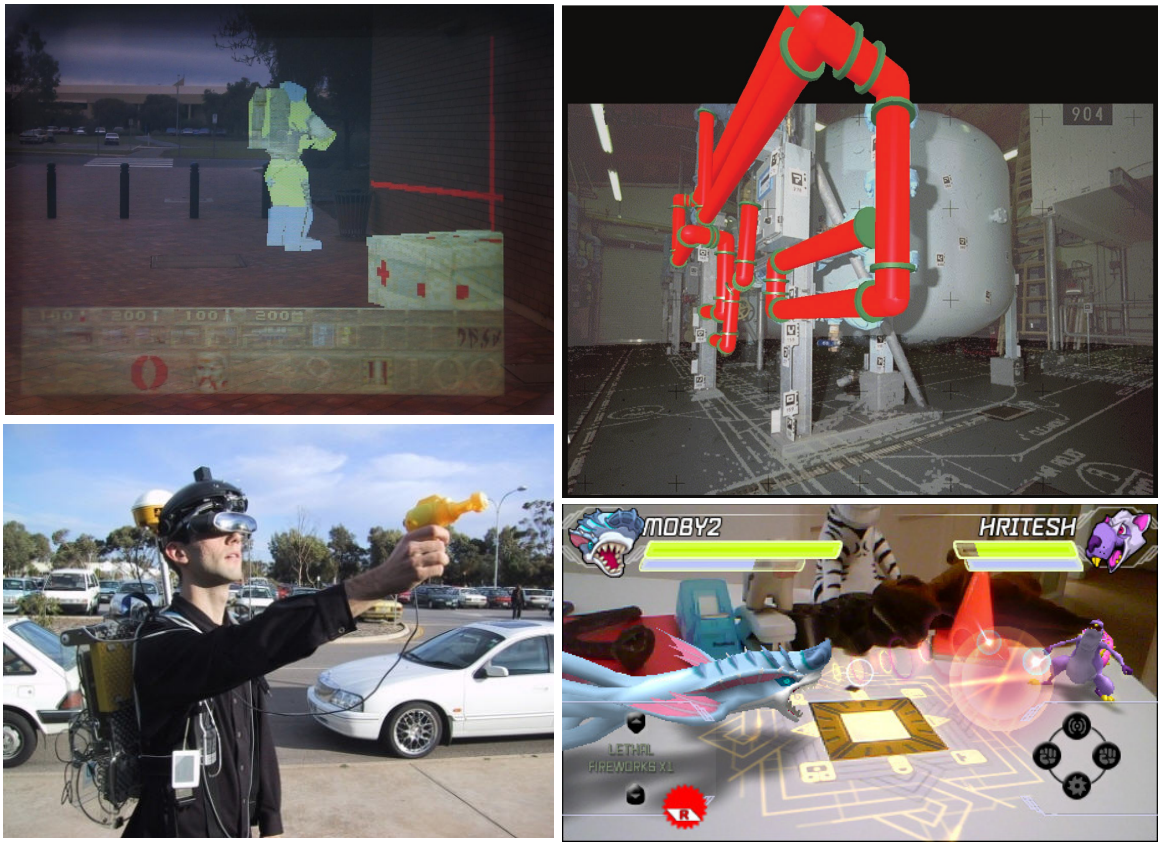

FIGURA 2.1: Visualización a través del *HDM* del proyecto *ARQuake* (arriba izquierda). Usuario de *ARQuake* utilizando el sistema (abajo izquierda). Superposición de tuberías y del plano de la planta en una fábrica (arriba derecha). Escena del videojuego *Invizimals* compuesta con vídeo real (abajo derecha).

[1] http://www.invizimals.com
[2] http://www.layar.com/

2.1.2. Arquitecturas y *frameworks* de RA

Vallino [Val98] describe los sistemas de RA como arquitecturas software. En su tesis, Vallino propone una arquitectura basada en varios subsistemas que interactúan entre sí. Estos subsistemas son el módulo responsable de recoger las imágenes, el encargado de procesarlas (realizar el *tracking* de las mismas), el encargado de generar la imágenes sintéticas y de componerlas, el subsistema de eventos y el de interacción con los objetos sintéticos a través de un dispositivo apuntador táctil (*haptic*). Es una arquitectura modular pero no extensible, que se basa en un conjunto cerrado de hardware.

Feiner et al [FMHW97] proponen un *framework* hardware y software para la realidad aumentada. Este *framework*, al que llaman *Mobile Augmented Reality System* se compone en su parte hardware de una mochila que alberga una computadora, de un casco con una cámara y unas gafas para *realidad virtual*, de un GPS y de un *modem* de radio. El software se llama COTERIE (Columbia Object-oriented Testbed for Exploratory Research in Interactive Environments) [MF96] y está basado en MODULA-3. Este software da soporte a la representación gráfica basada en estándares de objetos 3-D y 2-D sobreimpresos, a la representación interna de los datos y filtrado de los mismos y al desarrollo de prototipos usando un lenguaje de *script*. El *framework* se completa con una aplicación que hace uso de COTEIRE para representar datos obtenidos de la parte hardware. Este sistema, aunque desacopla el sistema de representación del resto de la aplicación, no presta un especial interés al diseño de una arquitectura modular ni extensible. Además, el sistema es completamente dependiente del hardware para el que se diseñó.

MacIntyre et al [MGDB04] proponen un *toolkit* de autoría de aplicaciones de RA basado en Macromedia Director. Este *toolkit* se compone de una aplicación de diseño y de un *plugin* (DART Xtra) que proporciona la interfaz con los dispositivos físicos. DART es un sistema cerrado, no extensible, monolítico y ligado a los sistemas operativos en los que funciona Director, limitando completamente su portabilidad a otras plataformas.

Todas las aplicaciones y estos sistemas tienen en común el uso de métodos y técnicas, que se enunciarán en el apartado siguiente.

2.1.3. Áreas y campos relacionados

Para diseñar e implementar un sistema para la RA es necesario el estudio de ciertas áreas y campos de conocimiento relacionados en las que se basa su funcionamiento. Las materias más relevantes en este ámbito son:

- La **Matemática**, en concreto un subconjunto del álgebra lineal. En la sección 2.2 se proporcionará una pequeña revisión de las estructuras y conceptos básicos necesarios para situar y orientar objetos en el espacio, de cómo realizar transformaciones de cuerpos rígidos y en qué consiste una proyección.
- La generación de **Gráficos 3-D por Computador**, que estudia la representación de objetos en el espacio. En la sección 2.3 se repasan algunos conceptos básicos relacionados con esta tecnología y en la sección 2.4 algunas bibliotecas 3-D.
- La **Visión por Computador**, que estudia el proceso de imágenes obtenidas a partir de una fuente de vídeo y su interpretación automática. En la sección 2.7 se introduce este campo.

Otras tecnologías que pueden resultar interesantes para la RA son la reproducción de audio, de voz sintetizada y de vídeos con compresión. También es importante disponer de alguna tecnología para la captura de eventos. En la sección 2.9 se verán algunas bibliotecas que implementan las funciones necesarias para la utilización de las mismas.

En la sección 2.6 se realiza una aproximación interfaces gráficas de usuario como forma para interactuar con una aplicación de RA.

En la sección 2.10 se investigan algunos lenguajes de *script* así como la posibilidad de facilitar su utilización en aplicaciones escritas en lenguajes compilados.

2.2. Introducción al marco matemático

En una aplicación de RA es muy importante disponer de las herramientas matemáticas necesarias para abstraer la información y poder tratarla mediante algoritmos. Dado que una aplicación de RA hace uso de información relativa a la posición y orientación, se hace necesario revisar algunos conceptos relacionados con el espacio euclídeo $\Re^n$, en concreto con uno de tres dimensiones $\Re^3$. Pero antes de comenzar a revisar conceptos se localizará la función del álgebra en un proceso de representación gráfica, ya que este campo hace uso directo de la misma durante algunas de sus etapas.

2.2.1. *Pipeline* gráfico

La función principal del *pipeline*[3] es la de generar una imagen 2-D a partir de una cámara virtual, objetos 3-D, fuentes de luz, texturas y otras entidades. El *pipeline* se divide en tres etapas principales (figura 2.2): la de aplicación, la de geometría y la de *rasterizado*[4].

FIGURA 2.2: Las tres etapa principales del pipeline gráfico

Los dispositivos gráficos con aceleración 3-D implementan un *pipeline*. El programador tiene control sobre la etapa de aplicación pero normalmente[5] no lo tiene sobre el resto de etapas. Aun así, se requiere conocerlas para comprender cómo utilizar la de aplicación, especialmente los tipos de datos que maneja la etapa de geometría, y cuáles de ellos y de qué manera afectan al resultado final.

La etapa de geometría (figura 2.3) se divide a su vez en otras seis.

FIGURA 2.3: La etapa de geometría dividida en un pipeline de etapas funcionales

Las etapas del *pipeline* de geometría son las siguientes:

- **Transformación de modelos y vista**. En esta etapa se generan las coordenadas globales de todos los modelos (sección 2.3.4 y 2.3.6), que en un principio sólo disponían de coordenadas locales, aplicando las transformaciones pertinentes (sección 2.2.3), conocidas como la *transformación de modelos*. Sólo se representan los modelos que la cámara ve. La cámara tiene una localización y una orientación (sección 2.2.4). Para facilitar la proyección y el *clipping* se realiza una *transformación de vista* que afecta a todos los modelos y a la cámara. Esta transformación consiste en colocar la cámara en el origen de coordenadas, apuntándola en el sentido del eje z negativo, con el eje y apuntando hacia arriba y el x hacia la derecha.

[3] *Pipeline* se puede traducir como *cauce*, aunque su traducción exacta en *tubería de distribución*.

[4] *Rasterizar* es una adaptación del verbo *rasterize*, sin una traducción concreta al español. Se utiliza para nombrar a la acción consistente en crear un mapa de *bits* a partir de otros datos, normalmente expresados como funciones matemáticas.

[5] En la actualidad, gran cantidad del hardware gráfico tiene una parte de la etapa de geometría programable (*vertex shadders*) y también de la etapa de *resterizado* (*fragment shadders*). Esta programación se utiliza para obtener efectos especiales o implementar funciones gráficas para las que no fue diseñado el hardware originalmente

- **Iluminación y sombreado**. Para brindar a los modelos de una apariencia más realista, una escena puede incluir una o más fuentes de luz. Se puede elegir si la iluminación afecta o no a un modelo en concreto. Los modelos geométricos pueden tener un color asociado a cada vértice o incluso una textura (sección 2.3.5). Cuando un modelo se ve afectado por alguna fuente de luz, se utiliza una ecuación de iluminación para calcular el color de cada vértice, utilizando el vector normal (sección 2.2.2) del mismo. El color entre los vértices de una cara (sección 2.3.4) se interpolará utilizando una técnica conocida como *Gouraud Shadding* (sección 2.3.3).

- **Proyección**. En esta etapa se realiza una transformación del volumen de la vista en un *cubo unidad* con puntos extremos $(-1,-1,-1)$ y $(1,1,1)$ que se conoce como *proyección* (sección 2.2.5). En esta etapa se consigue saber qué posición 2-D de un plano situado frente a la cámara corresponde a cada parte visible de la escena.

- ***Clipping*** o recortado. Sólo se dibujarán las caras que están dentro del volumen del cubo unidad. Las que están fuera se descartan, pero las que quedan entre el interior y el exterior tienen que ser recortadas. Es esta etapa, todas esas caras se procesan, generando nuevos vértices y aristas que hagan que se ajusten al volumen del cubo.

- **Mapeado de Pantalla**. En esta fase se hace coincidir la proyección del cubo unidad con la pantalla o ventana, pasando de una coordenadas tridimensionales a otras 2-D.

En las siguientes secciones se estudiarán las entidades matemáticas pertinentes junto con las propiedades y operaciones necesarias en la generación de gráficos 3-D.

2.2.2. Vectores y Puntos

Un vector geométrico es una entidad con magnitud (longitud) y dirección, que se representa gráficamente como un segmento con una punta de flecha en uno de sus extremos. Se representa un vector por una letra en negrita ($\mathbf{v}$) o por una letra con una flecha encima ($\vec{v}$). De un vector de magnitud 1, se dice que es unitario o que está normalizado ($\hat{\mathbf{v}}$).

Nótese que aunque los vectores se dibujan habitualmente en posiciones determinadas para facilitar la visualización de algunas operaciones, éstos carecen de localización.

Una de las características de un vector 3-D ($\in \Re^3$) es que se puede expresar con una 3-*tupla* de números reales, lo que facilita su representación en una computadora y por consiguiente su uso.

$$\vec{v} = (v_1, v_2, v_3) = v_1\,\mathbf{i} + v_2\,\mathbf{j} + v_3\,\mathbf{k} \tag{2.1}$$

donde **i** , **j**, **k** son vectores linearmente independientes. Por comodidad, estos vectores suelen corresponder con la base estándar (**i** = $(1,0,0)$, **j** = $(0,1,0)$, **k** = $(0,0,1)$).

En los motores gráficos y aplicaciones multimedia en tiempo real [VB04], el uso de los vectores suele remitirse a expresar una dirección en el espacio, por ejemplo, hacia dónde mira una cámara. También se utiliza para expresar un cambio de posición de un objeto, que puede por ejemplo, estar determinado por su velocidad.

La longitud d de un vector $\vec{v}$ (v_1, v_2, v_3), que se nota como $\|\vec{v}\|$ es:

$$\|\vec{v}\| = d = \sqrt{v_1^2 + v_2^2 + v_3^2} \tag{2.2}$$

Los vectores cumplen la propiedad conmutativa y la asociativa con la suma, existiendo un elemento neutro (identidad $\equiv \vec{0}$) y un elemento inverso ($\forall\ \vec{v} \in \Re^3 \exists(-\vec{v})\ |\ \vec{v} + (-\vec{v}) = \vec{0}$) de la misma. Con el producto por escalares, los vectores cumplen la asociativa, las dos distributivas y existe el elemento neutro ($1 \in \Re$).

Aparte de la suma, que se hace componente a componente, y del producto de un escalar por un vector, en el que se multiplica dicho escalar por cada componente del vector, los vectores tienen otras dos operaciones: el producto escalar y el vectorial.

El *productor escalar*[6] de dos vectores $\vec{v}$ y $\vec{w}$ es, como su nombre indica, un escalar y se calcula como

$$\vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{v} = \|\vec{v}\|\|\vec{w}\| \cos\theta = v_1 w_1 + v_2 w_2 + v_3 w_3 \tag{2.3}$$

siendo θ el ángulo que forman. Este ángulo se puede calcular como $\theta = \arccos \frac{\vec{v} \cdot \vec{w}}{\|\vec{v}\|\|\vec{w}\|}$. Esta es una de las utilidades más importantes del *producto escalar* de dos vectores.

El *productor vectorial* de dos vectores es otro vector perpedicular al plano que forman , cuya magnitud es

$$\|\vec{v} \times \vec{w}\| = \|\vec{v}\|\|\vec{w}\| \sin\theta \tag{2.4}$$

De manera analítica este vector se calcula así:

$$\vec{v} \times \vec{w} = (v_2 w_3 - v_3 w_2, v_3 w_1 - v_1 w_3, v_1 w_2 - v_2 w_1) \tag{2.5}$$

[6]En inglés, el *productor escalar* se conoce como *dot product* refiriéndose al símbolo que se utiliza para notar la operación $(\cdot)$

Un punto tiene una representación interna similar, aunque es una entidad matemática diferente. Un punto tiene localización, pero no tiene magnitud. La relación entre un punto y un vector son sus coordenadas. La distancia entre dos puntos a y b ($\overline{ab}$) es igual a la magnitud del vector que los une, y ese vector se calcula restando las coordenadas de los mismos.

$$\overline{ab} = \|b \vec{-} a\| \tag{2.6}$$

En los motores gráficos se suele utilizar una representación de cuatro elementos. Un punto p se representaría por $(p_x, p_y, p_z, 1)$ y un vector $\vec{v}$ por $(v_x, v_y, v_z, 0)$, como notación interna para diferenciar unos de otros. Esta notación es conveniente, como se verá en el apartado siguiente.

2.2.3. Matrices y transformaciones en el espacio

Las matrices matemáticas pueden ser utilizadas para manipular vectores y puntos. Una matriz está compuesta por elementos, distribuidos en m filas y n columnas. Si el número de columnas es igual al número de filas, la matriz es cuadrada. Si la matriz tiene sólo una fila ($1 \times n$), se dice que representa a un *vector fila* y si sólo tiene una columna ($m \times 1$) a un *vector columna*.

Una matriz es una entidad matemática con la que se pueden realizar operaciones. Las más importantes para la representación gráfica son la suma, el producto, la transposición, el cálculo del determinante, la matriz adjunta y la inversa de una matriz.

Las operaciones tienen una aplicación limitada. Sólo se pueden sumar matrices con las mismas dimensiones y sólo se pueden multiplicar aquellas cuyo número de columnas coincida con el de número de filas de la otra, no siendo una operación conmutativa. El determinante de una matriz sólo se puede calcular si ésta es cuadrada, y para calcular la inversa de una matriz se hace uso del determinante.

Calcular la inversa de una matriz es importante para resolver ecuaciones que las incluyan. La forma más común de calcular la inversa, si existe, de una matriz es la regla de Cramer (2.7).

$$\mathbf{M}^{-1} = \frac{1}{|\mathbf{M}|} adj(\mathbf{M}) \tag{2.7}$$

El elemento neutro de la suma es una matriz compuesta por ceros, y el del producto la *matriz identidad* (**I**), rellena de ceros excepto en la diagonal principal, donde todo elemento es 1.

En los motores de representación, la matrices utilizadas son cuadradas con una dimensión de 4×4. Aparte de para representar la orientación y la posición, estas matrices también se utilizan para aplicar transformaciones [AMHH08]. Las transformaciones básicas son la translación, la rotación y el escalado.

Traslación

Un cambio de una posición a otra se representa a través de una matriz **T** de traslación. Esta matriz traslada una entidad según el vector $\vec{t} = (t_x, t_y, t_x)$.

$$T(\vec{t}) = T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.8}$$

Cuando se aplica esta transformación $(\mathbf{T}(t) \cdot p)$ se produce una traslación que se corresponde con el vector $\vec{v}$. Si se aplica a un punto $p = (p_x, p_y, p_z, 1)$ se obtiene uno nuevo $p' = (p_x + t_x, p_y + t_y, p_x + y_z)$. Nótese que si esta transformación se aplica a un vector $\vec{v} = (v_x, v_y, v_x, 0)$, éste no se traslada, lo que es coherente con el concepto de vector. La inversa de una matriz de traslación corresponde al vector $\vec{t}$ negado.

$$\mathbf{T}^{-1} = \mathbf{T}(-\vec{t})$$

Rotación

Las matrices de rotación permiten rotar una entidad en el espacio. Existen tres matrices de rotación diferentes $\mathbf{R}_x(\theta), \mathbf{R}_y(\theta), \mathbf{R}_z(\theta)$ (2.9), una por cada eje del espacio tridimensional, rotando θ radianes alrededor de cada uno de ellos.

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 1 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.9}$$

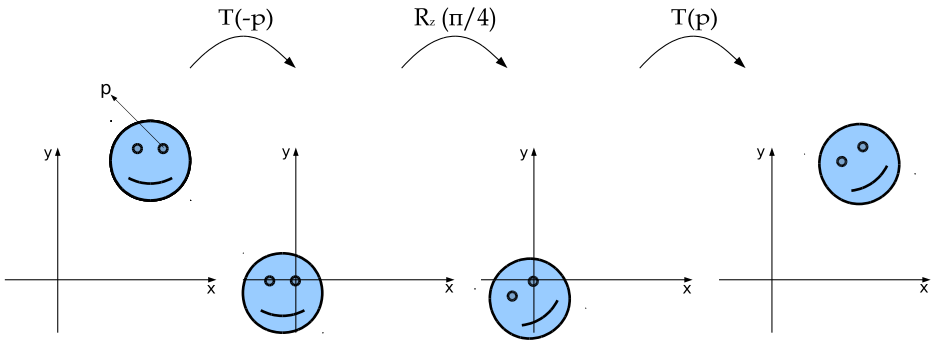

FIGURA 2.4: Ejemplo de una rotation alrededor de un punto **p**

La matrices de rotación son ortonormales ($R\,R^T = R^T R = \mathbf{I}$), por lo que $\mathbf{R}^{-1} = \mathbf{R}^{\mathbf{T}}$.

Como ejemplo, imagínese que se quiere rotar ϕ radianes alrededor de su eje z una entidad que está situada en la posición $p = (p_x, p_y)$ (figura 2.4). Lo primero que se hará será posicionar la entidad sobre el eje z, después se rotará y se volverá a desplaza a su posición original. Esta transformada **Z** viene dada por:

$$\mathbf{Z} = \mathbf{T}(\vec{p})\mathbf{R}_z(\phi)\mathbf{T}(-\vec{p})$$

Para representar una rotación arbitraria, girando ϕ, ω y δ radianes alrededor de los ejes x, y, z respectivamente se podrían componer estas tres rotaciones en una sola matriz **R** (2.10).

$$\mathbf{R} = \mathbf{R}_x(\phi)\mathbf{R}_y(\omega)\mathbf{R}_z(\delta) \tag{2.10}$$

Escalado

Una matriz de escalado $\mathbf{S}(\vec{s}) = \mathbf{S}(s_x, s_y, s_z)$ (ecuación 2.11) escala una entidad con los factores s_x, s_y, s_z a lo largo de sus direcciones x, y, z respectivamente. Esto significa que se puede utilizar una matriz de escalado para hacer más grande o más pequeño un objeto. Cuando más grande sea $s_i (i \in \{x, y, z\})$, mayor se escalará el objeto en dicha dimensión.

$$\mathbf{S}(\vec{s}) = \mathbf{S}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.11}$$

La transformación de escalado se dice que es *uniforme* si $s_x = s_y = s_z$, en otro caso se dice que es *no-uniforme*. La inversa es $\mathbf{S}^{-1}(\vec{s}) = \mathbf{S}(1/s_x, 1/s_y, 1/s_z)$.

Si alguno o los tres valores de **S** son negativos, se dice que es una matriz espejo. Es motivo es que la entidad gira empujada por el eje de la dimensión que se está escalando. Esto es, en $(0,1)$ el objeto sufre una escalo a un tamaño menor y en $(1,+\infty)$ el objeto se agranda. En $(-\infty,-1)$ y $(-1,0)$ sucede lo mismo, pero con el objeto reflejado en el plano al que dicho eje de dimensión es perpendicular.

Concatenación de transformaciones

Debido a que el producto de matrices no es conmutativo, el orden en el que se apliquen las transformaciones es relevante. Por motivos de eficiencia se puede desear concatenar las transformaciones que debe de sufrir un objeto, y aplicar la transformación dada por esa matriz **M** (2.12) resultante. El orden que se debe seguir el primero aplicar el escalado, luego la rotación y después la traslación.

$$\mathbf{M} = \mathbf{TRS} \tag{2.12}$$

2.2.4. Representación de la orientación

La representación de la orientación se puede llevar a cabo con un par de vectores perpendiculares entre sí. Uno de ellos indicaría la dirección de un objeto, el otro, cuánto gira este objeto en torno al primer vector. Esta representación es útil para representar cámaras, pero se hacen necesarias otras representaciones que permitan realizar cálculos de manera rápida y sencilla.

En el campo de la representación en tiempo real, las dos formas más populares son las *matrices de rotación* y los *cuaterniones*.

Matrices de orientación

En $\Re^3$, una matriz de orientación es una matriz de 3×3 elementos que contiene tres vectores de tres elementos. Dependiendo de si se elige una notación de columna o de fila, los vectores coincidirán con las columnas o las filas de la matriz.

El primer vector corresponde con el eje x local del objeto, el segundo con el eje y, el tercero con el z. En notación columna, esto correspondería con una matriz **O** (2.13) de orientación.

$$\mathbf{O} = \begin{pmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{pmatrix} \quad x' = (x_x, x_y, x_z) \;\; y' = (y_x, y_y, y_z) \;\; z' = (z_x, z_y, z_z) \tag{2.13}$$

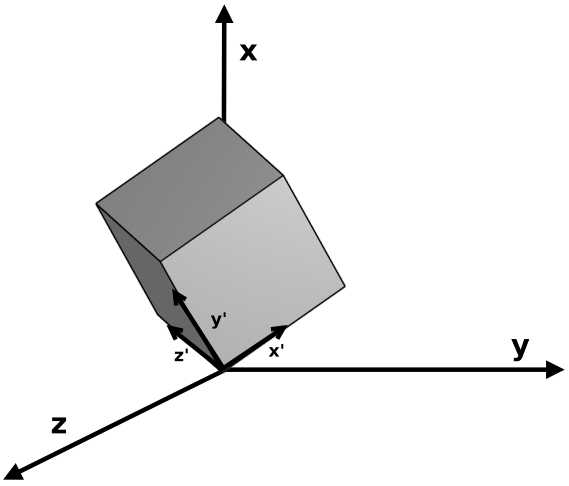

FIGURA 2.5: Ejemplo de orientación global usando ejes locales

Esta matriz se amplía con un nuevo vector que contiene la posición del objeto. Como la matriz resultante no es cuadrada, se amplía en una fila más, tomando la última fila de la matriz identidad.

$$\begin{pmatrix} x_x & y_x & z_x \\ x_y & y_y & z_x \\ x_z & y_z & z_z \end{pmatrix} \longrightarrow \begin{pmatrix} x_x & y_x & z_x & \mathbf{c_x} \\ x_y & y_y & z_y & \mathbf{c_y} \\ x_z & y_z & z_z & \mathbf{c_z} \end{pmatrix} \longrightarrow \begin{pmatrix} x_x & y_x & z_x & \mathbf{c_x} \\ x_y & y_y & z_y & \mathbf{c_y} \\ x_z & y_z & z_z & \mathbf{c_z} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.14}$$

A esta matriz se le pueden aplicar las transformaciones del apartado anterior, para modificar así la posición y orientación del objeto al que representa.

Cuaterniones

Una matriz de rotación está formada por 9 elementos, y los datos que contiene son redundantes. Aunque los cuaterniones fueron inventados en 1843 por Sir William Rowan, no fue hasta 1985 cuando Shoemake [Sho85] introdujo su uso en el campo de los gráficos por computador.

Un cuaternión tiene cuatro componentes, con lo que la representación interna en un motor gráfico es igual que la de un vector. Un cuaternión o cuaternio $\hat{\mathbf{q}}$ se puede definir de las siguientes formas:

$$\hat{\mathbf{q}} = (\vec{q}_v, q_w) = iq_x + jq_y + kq_z + q_w \tag{2.15}$$

$$\vec{q}_v = iq_x + jq_y + kq_z = (q_x, q_y, q_z) \tag{2.16}$$

$$i^2 = j^2 = k^2 = -1, jk = -kj = i, ki = -ik = j, ij = -ji = k \tag{2.17}$$

El vector que contiene un cuaternio se puede utilizar como tal, y sus operaciones se utilizan en las del cuaternio. Las definiciones de producto (2.18), de suma (2.19), conjugado (2.20), norma (2.21),identidad (2.22), e inversa (2.23) de un cuaternio se muestran a continuación.

$$\hat{\mathbf{q}}\hat{\mathbf{r}} = (iq_x + jq_y + kq_z + q_w)(ir_x + jr_y + kr_z + r_w) = (\vec{q}_v \times \vec{r}_v + q_w\vec{r}_v,\ q_w r_w - \vec{q}_v \cdot \vec{r}_v) \tag{2.18}$$

$$\hat{\mathbf{q}} + \hat{\mathbf{r}} = (\vec{q}_v, q_w) + (\vec{r}_v, r_w) = (\vec{q}_v + \vec{r}_v, q_w + r_w) \tag{2.19}$$

$$\hat{\mathbf{q}}^* = (\vec{q}_v, q_w)^* = (-\vec{q}_v, q_w) \tag{2.20}$$

$$n(\hat{\mathbf{q}}) = \hat{\mathbf{q}}\hat{\mathbf{q}}^* = \vec{q}_v \cdot \vec{q}_v + q_w^2 = q_x^2 + q_y^2 + q_z^2 + q_w^2 \tag{2.21}$$

$$\hat{\mathbf{i}} = (\vec{0}, 1) \tag{2.22}$$

$$\hat{\mathbf{q}}^{-1} = \frac{1}{n(\hat{\mathbf{q}})}\hat{\mathbf{q}}^* \tag{2.23}$$

Un *cuaternio unidad* es aquel que $n(\hat{\mathbf{q}}) = 1$, y se puede escribir como $\hat{\mathbf{q}} = \sin\phi\vec{u}_q + \cos\phi = e^{\phi\vec{u}_q}$, siendo $\vec{u}_q$ un vector unitario tridimensional. De aquí se derivan las definiciones de logaritmo (2.24) y potencia (2.25) de un cuaternio.

$$log(\hat{\mathbf{q}}) = log(e^{\phi\vec{u}_q}) = \phi\vec{u}_q \tag{2.24}$$

$$\hat{\mathbf{q}}^t = (\sin\phi\vec{u}_q + \cos\phi)^t = e^{\phi t\vec{u}q} = \sin(\phi t)\vec{u}_q + \cos(\phi t) \tag{2.25}$$

El hecho más relevante acerca de los cuaterniones unidad es que pueden representar cualquier rotación 3-D (figura 2.6) y que se representan con tan sólo cuatro elementos. Si se usan la coordenadas de un punto o vector $\mathbf{p} = (p_x, p_y, p_z, p_w)$ como las componentes de un cuaternio $\hat{\mathbf{p}}$ y se asume que $\hat{\mathbf{q}} = (\sin\phi\vec{u}_q, \cos\phi)$ es un cuaternio unidad, entonces

$$\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^{-1} \tag{2.26}$$

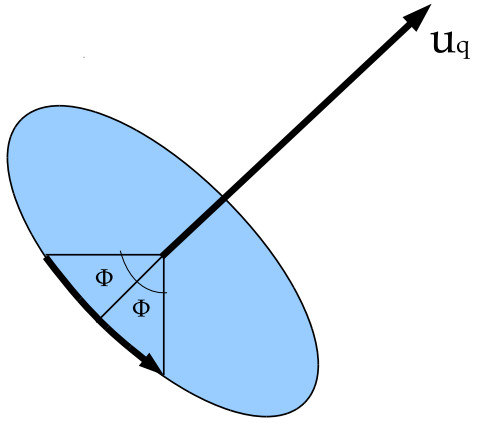

FIGURA 2.6: Transformación causada por un cuaternión unidad

rota $\hat{\mathbf{p}}$ y por tanto $\mathbf{p}$, alrededor de del eje representado por $\vec{u}_q$. Nótese que $\hat{\mathbf{q}}$ y $-\hat{\mathbf{q}}$ representan la misma rotación.

La concatenación de dos rotaciones, aplicando $\hat{\mathbf{q}}$ y luego $\hat{\mathbf{r}}$ a un cuaternio $\hat{\mathbf{p}}$ (que representa un punto o un vector), está dada por la ecuación 2.27.

$$\hat{\mathbf{r}}(\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^*)\hat{\mathbf{r}}^* = (\hat{\mathbf{r}}\hat{\mathbf{q}})\hat{\mathbf{p}}(\hat{\mathbf{r}}\hat{\mathbf{q}})^* = \hat{\mathbf{c}}\hat{\mathbf{p}}\hat{\mathbf{c}}^* \tag{2.27}$$

Las ventajas de usar cuaterniones para representar la orientación, aparte del tamaño de las estructuras necesarias, son la facilidad de interpolar entre dos rotaciones y que trabajando con ellos se evita el problema del *gimbal lock* [Han05].

Conversión entre matriz de orientación y cuaternio

En algunos sistemas, la multiplicación de matrices cuadradas de 4×4 está implementada de forma muy rápida, incluso se ofrece soporte hardware para llevar a cabo dicho producto. Por este motivo es conveniente poder realizar conversiones entre cuaterniones y matrices [SE91]. A continuación se proporcionan las ecuaciones para la conversión desde un cuaternio unidad a matriz (2.28) y desde una matriz a un cuaternio unidad (2.29).

$$\mathbf{M}^q = \begin{pmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y-q_wq_z) & 2(q_xq_z+q_wq_y) & 0 \\ 2(q_xq_y+q_wq_z) & 1-2(q_x^2+q_z^2) & 2(q_yq_z-q_wq_x) & 0 \\ 2(q_xq_z-q_wq_y) & 2(q_yq_z+q_wq_x) & 1-2(q_x^2+q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.28}$$

$$tr(\mathbf{M}^q) = 4-4(q_x^2+q_y^2+q_z^2) = 4(1-(q_x^2+q_y^2+q_z^2))$$

$$q_w = \tfrac{1}{2}\sqrt{tr(\mathbf{M}^q)} \quad q_x = \tfrac{m_{21}^q - m_{12}^q}{4q_w} \tag{2.29}$$

$$q_y = \tfrac{m_{02}^q - m_{20}^q}{4q_w} \qquad q_z = \tfrac{m_{10}^q - m_{01}^q}{4q_w}$$

2.2.5. Proyecciones

Antes de proceder a la representación de una escena, todos los objetos relevantes de la misma (aquellos que se visualizarán) han de ser proyectados en una especie de plano que hará de pantalla. Existen dos tipos de proyecciones: ortográfica y en perspectiva.

Proyección ortográfica

La característica principal de una proyección ortográfica es que la líneas paralelas se mantienen de ese modo tras la proyección (figura 2.7).

Una matriz de proyección ortográfica puede ser definida en los términos de una 6-*tupla* (l,r,b,t,n,f) que corresponden con los valores de izquierda, derecha, abajo, arriba, cerca y lejos. Estos valores representan los planos que delimitan una caja de proyección. Esencialmente lo que hace esta matriz es alinear ese caja con el origen de coordenadas y escalarla para darle forma de cubo. La matriz se muestra en la ecuación 2.30.

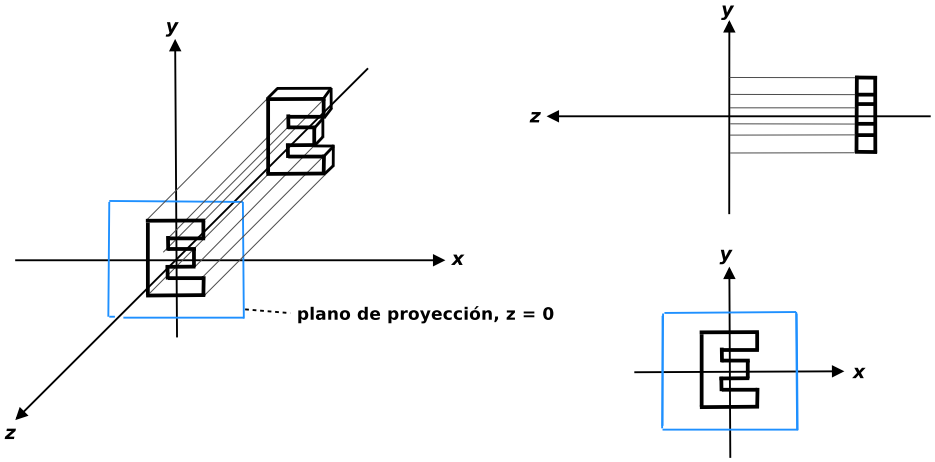

FIGURA 2.7: Tres vistas de una proyección ortográfica simple

$$\mathbf{P}_o = \mathbf{S}(\vec{s})\mathbf{T}(\vec{t}) = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.30}$$

Proyección en perspectiva

En una proyección en perspectiva las líneas paralelas ya no lo son tras la proyección sino que tienden a converger en un punto. El ser humano tiene una percepción del mundo con perspectiva, por eso este tipo de proyección es de gran interés en los gráficos por computador.

Una matriz de proyección transforma un *frustum*[7] en un *cubo unitario* (2.8).

Una matriz de proyección en perspectiva $\mathbf{P}_p$ se puede escribir como en la ecuación 2.31.

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.31}$$

[7]En el campo de gráficos por computador *frustum* es un cono piramidal truncado, que representa una cámara.

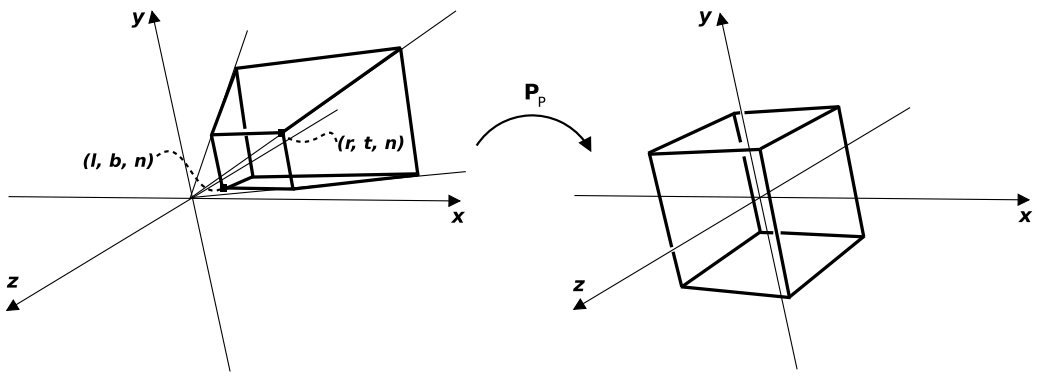

FIGURA 2.8: Conversión de un *frustum* a un cubo unidad

2.3. Gráficos 3-D y apariencia visual

La apariencia visual de una escena no sólo depende de los objetos visualizados sino también de otras entidades y propiedades, como las fuentes de luz, los materiales o el tipo de sombreado. A continuación se verán algunas de ellas y de qué manera están relacionadas.

2.3.1. Fuentes de luz

Una fuente de luz es cualquier entidad emisora de fotones. Existen tres tipos de fuentes de luz: *direccionales*, *puntuales* y *focales*; dependiendo del comportamiento de los fotones.

Los tres tipos tienen parámetros de intensidad comunes, con un color asociado. Estos parámetros se dividen en intensidad *ambiental*, *difusa* y *especular*. Normalmente, estas intensidades se dan como una 3-*tupla* RGB, que representa el color de cada una de las componentes de la fuente de luz.

Una fuente de luz tiene además una posición en el espacio y, en el caso de la direccionales y focales, una orientación. Según éstos, la luz incidirá o no sobre un objeto, que tendrá un material asociado.

2.3.2. Materiales

En los sistemas de representación gráfica en tiempo real, un material consiste en unos parámetros que lo describen: *ambiente*, *difuso*, *especular*, *brillo* y *emisión*. El color de un objeto dependerá de estos parámetros, de los parámetros de la fuente de luz que lo ilumine y del modelo de iluminación utilizado de entre los mencionados en el siguiente apartado.

2.3.3. Iluminación y sombreado

El teŕmino iluminación se utiliza para designar la interacción entre fuentes de luz y objetos. Sombreado es el proceso de realizar cálculos acerca de la iluminación para determinar el color de los píxeles a partir de la misma. Los tres tipos principales de sombreado son el sombreado plano[8], el Gouraud y el Phong.

En el sombreado plano, el color de una cara depende de su vector normal. El sombreado Gouraud [Gou71] interpola los valores de color de los píxeles resultantes utilizando las normales de cada vértice de la cara. El sombreado Phong [Pho75], calcula una serie de normales por toda la superficie, interpolando las normales de los vértices, y las utiliza para los cálculos de iluminación.

Puede que este color no sea el final, sino que se tomen datos de texturas para aumentar el realismo de los objetos, pero antes de hablar sobre esto, se hará un pequeño comentario sobre los vértices y las caras, que son los que reciben el proceso de iluminación y texturizado.

2.3.4. Vértices y caras

Un vértice se representa de la misma forma que un punto 3-D. Normalmente su posición se expresa de forma local al objeto que lo contiene.

Una cara es un plano delimitado por una serie de vértices coplanares. Así, la cara más sencilla se corresponde con un triángulo. Una cara triangular está formada por tres vértices; la unión de los mismos forman las aristas. El orden de unión de los vértices determina el vector normal asociado a dicha superficie (figura 2.9).

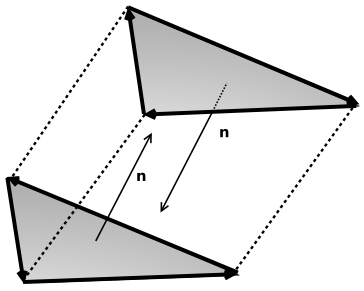

FIGURA 2.9: Vector normal a una cara dependiendo del orden de sus vértices

[8] *flat shading*

Nótese que para calcular el vector normal de un polígono con tres vértices v_1, v_2 y v_3 consecutivos en dicho orden se utilizan las propiedades y operaciones de los puntos y de los vectores. Primero se calculan dos vectores $\vec{u} = v_2 - v_1$ y $\vec{w} = v_3 - v_1$; después se utiliza el producto vectorial $\vec{n} = \vec{u} \times \vec{w}$ para calcular la normal. Por último, se suele normalizar dicho vector, obteniendo así la *normal unitaria* a dicha cara.

Para caras con un mayor número de vértices se aplica la misma regla.

2.3.5. Texturas y mapeado *UV*

En los gráficos por computador, el texturizado es un proceso por el cual se modifica la apariencia de una cara en cada uno de sus píxeles, utilizando alguna imagen, función u otra fuente de datos diferente. La texturas procedurales son las generadas por una función $f : \Re^2 \longrightarrow \Re^2$, y poseen una resolución infinita. Sin embargo, aunque las funciones procedurales tienen enormes ventajas, también tienen carencias importantes, como lo es la complejidad de representar matemáticamente ciertos patrones de la naturaleza, o el poder de cálculo que se requiere para obtener el valor correspondiente a un solo píxel.

El tipo de texturas en la representación en tiempo real son las basadas en mapas de bits. Para llevar a cabo el texturizado de una cara utilizando una imagen se realiza un proceso conocido como *mapeado de texturas*. El tipo de mapeado más común es el conocido como *mapeado UV*. Se llama así porque cada vértice de una cara tiene asociada un posición bidimensional $t = (u, v) \mid u, v \in [0, 1]$ que determina el píxel de la imagen que corresponde a este vértice (figura 2.10).

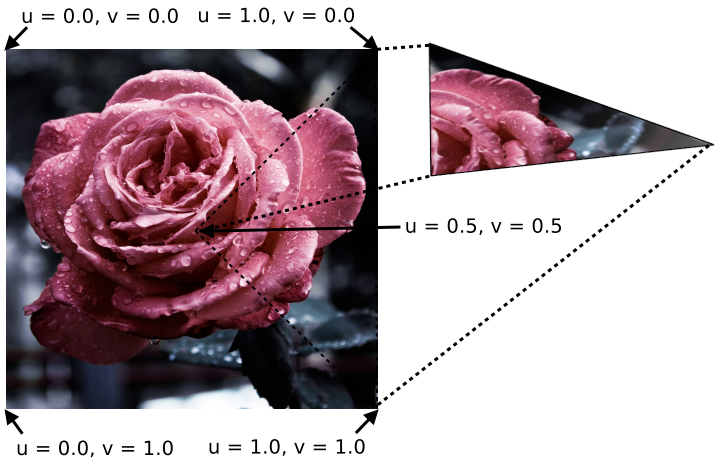

FIGURA 2.10: Mapeado UV

Si se tiene una textura de 256×256 píxeles y un vértice cuyas coordenadas UV son $(0.3, 0.8)$, su píxel asociado vendrá determinado por $(abs(256 \cdot 0.3), abs(256 \cdot 0.8)) = (76, 204)$. Nótese que la resolución viene determinada por la de la textura. Así la función de mapeado UV es de la forma $g : [0.0, 1.0] \longrightarrow \mathbb{N}^2$.

Los vértices, caras, materiales y texturas se utilizan de forma conjunta en los modelos 3-D, ya que por si solos no proporcionan capacidades de representación gráfica más allá que la de polígonos iluminados y con una imagen impresa.

2.3.6. Metaformatos de archivo para 3-D interactivo

En su forma más sencilla, un modelo 3-D es un conjunto de vértices y un conjunto de caras. Puede contener también información sobre materiales y texturizado. Un modelo puede estar definido no sólo como caras planas, sino como un conjunto de curvas o una mezcla de ambas. El hardware de aceleración 3-D trabaja mejor con objetos basados en caras, en concreto con aquellos basados en triángulos. De hecho, normalmente las caras de más de tres vértices se dividirán internamente en triángulos. A la información recogida por las caras y los vértices se le denomina malla.

Se han revisado varios formatos, encontrando relevancia en dos de ellos: m2d y Collada.

M2D

El formato de modelos 3-D *md2* [Hen02] es el utilizado en el motor gráfico del juego *Quake II* de *Id Software*, creado por esta empresa durante el desarrollo del mismo. Debido a que dicho motor gráfico se hizo público bajo licencia GPL, el uso de este formato se ha generalizado.

Es un formato que almacena información sobre vértices, caras triangulares, texturizado y animación. Las animaciones se basan en mutaciones de una misma malla. Es un formato binario, esto es, el archivo donde se guarda un modelo de este tipo no es editable por un humano de forma directa. En el listado 2.1 se muestra la cabecera de este formato en C.

Aunque cubre las necesidades de un pequeño motor gráfico, md2 es quizá un formato limitado para la actualidad y al ser un formato binario carece de capacidad de extensión.

```
/* MD2 header */
struct md2_header_t
{
  int ident;                  /* magic number: "IDP2" */
  int version;                /* version: must be 8 */

  int skinwidth;              /* texture width */
  int skinheight;             /* texture height */

  int framesize;              /* size in bytes of a frame */

  int num_skins;              /* number of skins */
  int num_vertices;           /* number of vertices per frame */
  int num_st;                 /* number of texture coordinates */
  int num_tris;               /* number of triangles */
  int num_glcmds;             /* number of opengl commands */
  int num_frames;             /* number of frames */

  int offset_skins;           /* offset skin data */
  int offset_st;              /* offset texture coordinate data */
  int offset_tris;            /* offset triangle data */
  int offset_frames;          /* offset frame data */
  int offset_glcmds;          /* offset OpenGL command data */
  int offset_end;             /* offset end of file */
};
```

LISTADO 2.1: Cabecera del formato m2d de Id Software

Collada

Collada [AB06] es el formato abierto sugerido por el grupo Khronos[9]. Este formato soporta modelos complejos, basados en mallas triangulares o poligonales, basados en curvas o superficies de Bézier, información de texturizado, *shadders* y propiedades físicas del modelo. Es un formato basado en XML y la extensión de los archivos es normalmente *.dae*. Así, es un formato editable directamente por un humano, además de estar basado en un estándar reconocido.

Collada fue creado originalmente por SONY y su versión más extendida es la especificación 1.4.1 [Bar06]. El grupo Khronos mantiene las revisiones de esa y de una versión superior del formato, que no guardan muchas diferencias entre sí.

El formato Collada no es tan simple como el m2d, pero se puede hacer uso únicamente de las características que interesen en cada sistema, reduciendo la complejidad de trabajar con este tipo de archivos.

Collada soporta la creación de escenas complejas, con cámaras, luces y varios modelos (basados en mallas o no), además de información sobre los materiales, texturas y *shadders*. Una escena contendrá ninguna o varias geometrías (`<geometry>`), que corresponden con

[9]El *Khronos Group* es un consorcio de empresas creado para proponer estándares abiertos en el campo de la autoría de aplicaciones paralelas, de gráficos y multimedia.

los modelos 3-D. Estas geometrías podrán contener, entre otras formas de representación obviadas, mallas (`<mesh>`), que podrán estar formadas por triángulos (`<triangles>`) o polígonos (`<polygons>`) entre otras entidades. Una malla contendrá una serie de vértices y otros datos como cuáles de ellos forman una cara determinada, a modo de *arrays* de números reales o enteros. En el listado 2.2 se muestra un ejemplo de código para guardar la información de los vértices y en la figura 2.11 se ofrece una explicación sobre el mismo.

```
<source id="vertices_source" name="Vertices">
    <float_array id="values" count="6"> 0.3 0.5 0.7 0.2 0.4 0.6 </float_array>
    <technique_common>
         <accessor source="#values" count="2" stride="3">
              <param name="X" type="float"/>
              <param name="Y" type="float"/>
              <param name="Z" type="float"/>
         </accessor>
   </technique_common>
</source>
```

LISTADO 2.2: Fragmento de código para definir vértices en Collada

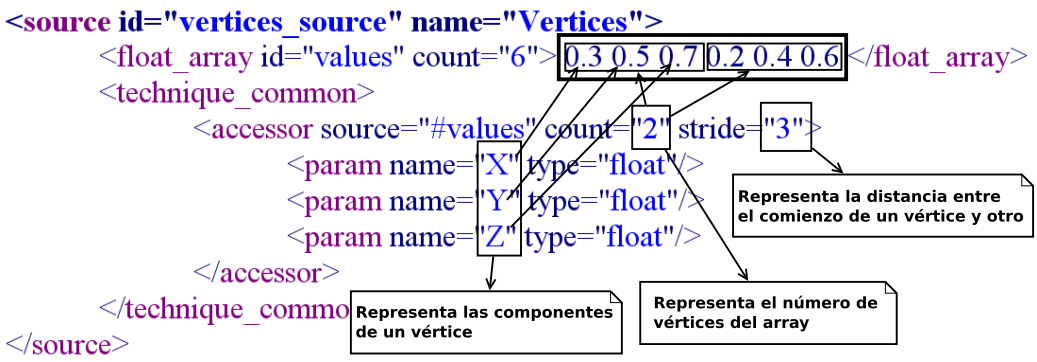

FIGURA 2.11: Array de vértices de un archivo Collada

En el listado 2.3 se muestra un ejemplo del código correspondiente a una malla, y en la figura 2.12 la explicación pertinente.

```
<mesh>
    <source id="position"/>
    <source id="normal"/>
    <source id="textureCoords"/>
    <vertices id="verts">
        <input semantic="POSITION" source="#position"/>
    </vertices>
    <triangles count="2" material="Bricks">
        <input semantic="VERTEX" source="#verts" offset="0"/>
        <input semantic="NORMAL" source="#normal" offset="1"/>
        <input semantic="TEXCOORD" source="#textureCoords" offset="2" set="1" />
        <p>
           0 0 0 1 3 2 2 1 3
           0 0 0 2 1 3 3 2 2
       </p>
    </triangles>
</mesh>
```

LISTADO 2.3: Fragmento de código para definir una malla en Collada

Nótese como en un mismo *array* de enteros está contenida la información sobre los índices de los vértices que forman dos caras, los índices de las componentes de sus normales y los índices de las coordenadas de mapeo.

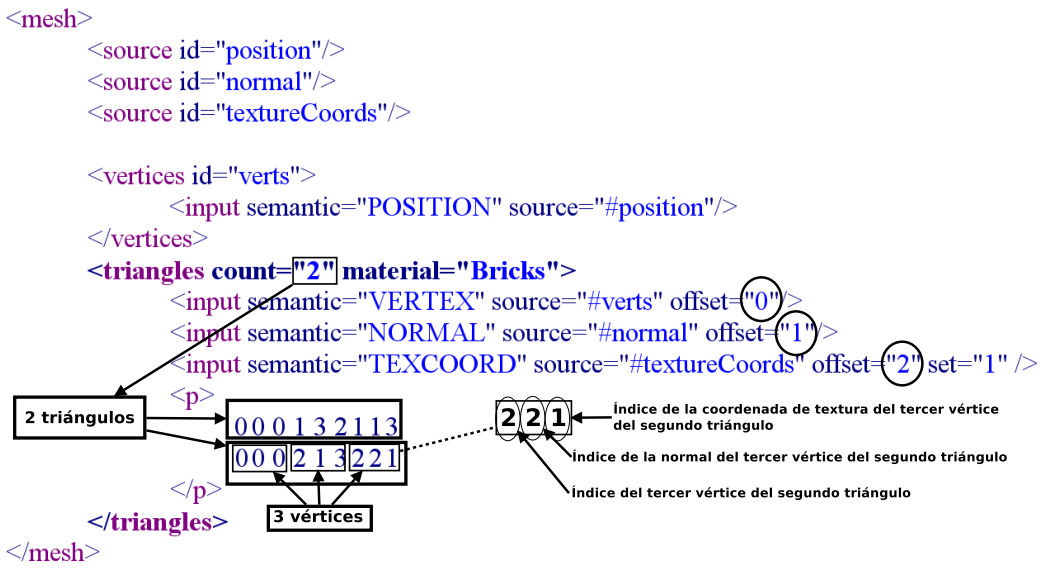

FIGURA 2.12: Una malla en Collada

2.3.7. Transparencia, Alpha y Composición

Como final de la sección, cabe mencionar cómo se procesa la transparencia en los gráficos por computador, ya que puede resultar útil el uso de objetos traslúcidos en la RA.

La transparencia en tiempo real está muy limitada. Si bien es cierto que se pueden obtener efectos de refracción y reflexión en tiempo real mediante técnicas que implican el uso de *shadders* o el falseamiento de dichos efectos, los sistemas de tiempo real suelen dar soporte sólo para representar una cara de forma semi-transparente.

Para llevar a cabo esta representación lo que se realiza es una composición de *buffers*. El *buffer* que representa los píxeles de debajo no contiene información extra, mientras que el *buffer* que se superpone ha de tener un valor asociado. Este valor se conoce como *alpha*, y va de 0.0 (100 % transparente) hasta 1.0 (100 % opaco).

Existen varias formas de realizar la composición, pero este concepto de transparencia es utilizado por la mayoría de las bibliotecas gráficas.

2.4. Bibliotecas de representación 3-D

Para la realización de este proyecto se han revisado dos de las principales bibliotecas de representación 3-D a bajo nivel: Direct3D y OpenGL. Estas dos bibliotecas son las que apoyan la mayoría de los dispositivos hardware de aceleración gráfica.

Direct3D forma parte del *framework* DirectX de Microsoft. Es una biblioteca muy extendida, y muchos motores 3-D comerciales están basados en ella. DirectX es una biblioteca exclusiva para el sistema operativo Windows de dicha empresa. Aunque existen formas de ejecutar un programa compilado para esta plataforma en otros sistemas, no es posible crear aplicaciones nativas utilizando Direct3D fuera de Windows.

2.4.1. OpenGL

OpenGL (*Open Graphic Library*[10]) [SWND04] es una biblioteca consolidada en la industria de gráficos por computador. La especificación de la misma es responsabilidad del grupo Khronos, que la mantiene abierta. Gracias a esto, existen varias implementaciones libres. Además, casi todos los sistemas no-libres (Windows y Mac OSX entre ellos) disponen de una implementación de OpenGL, lo que influye positivamente en la portabilidad de la aplicaciones basadas en esta biblioteca.

OpenGL dispone de funciones para dibujar polígonos texturizados, tiene soporte para fuentes de luz puntuales y de foco, y para transparencias, entre muchas otras funcionalidades. Su funcionamiento corresponde a una máquina de estados, que corresponden a características que afectarán a la representación. Para habilitar o deshabilitar un estado se utilizan las funciones `glEnable()/glDisable()`, pasándoselo como parámetro.

Para dibujar con OpenGL se hace uso de alguna de sus primitivas, que son entre otras: triángulos, cuadrángulos, polígonos con un número de vértices arbitrario y líneas. Una secuencia de comandos para dibujar en OpenGL empieza normalmente con la función `glBegin()`, que acepta un valor referido al tipo de primitiva a utilizar, y termina con `glEnd()`. Entre estas dos funciones se utilizan otras para determinar la posición de los vértices de la primitiva elegida (`glVertex__()`), sus normales (`glNormal__()`), su

[10] En español, biblioteca gráfica abierta

color (`glColor__()`) y sus coordenadas de texturizado (`glTexCoord__()`). Todas estas funciones poseen variaciones, dependiendo de los parámetros esperados. Por ejemplo, existe una función con prototipo

```
void glVertex3fv(const GLfloat *v)
```

donde el ***3*** se refiere a la cantidad de elementos que se esperan, ***f*** al tipo (`float` en este caso) y ***v*** a la forma de pasar estos datos (como un puntero/vector en este caso).

OpenGL proporciona funciones para realizar transformaciones de traslación, de rotación y de escalado (`glTranslate()`, `glRotate()` y `glScale()` respectivamente). Estas transformaciones se aplican sobre dos matrices especiales. OpenGL dispone de dos pilas, cada una asociada a una de esas matrices. Para guardar una matriz en estas pilas se utilizan las funciones `glPushMatrix()/glPopMatrix()`. Para cambiar de una pila a otra se usa la función `glMatrixMode()`.

Uno de esas matrices es la *MODELVIEW MATRIX*, que corresponde a la matriz de posicionamiento y orientación y que representa el sistema de referencia actual con respecto del origen de coordenadas. Es una matriz de 4×4 como las estudiadas en el apartado 2.2.4.

El otro tipo es la *PROJECTION MATRIX*, que corresponde como su nombre indica con la matriz de proyección utilizada para la representación. Como el sistema de referencia de OpenGL se basa en la regla de la mano derecha y no de la izquierda[11], la matriz de proyección de OpenGL ha de construirse con una transformación de escalado $\mathbf{S}(1,1,-1)$, para hacer que el eje Z cambie de sentido. El resultado se muestra en la ecuación 2.32.

$$\mathbf{P}_{OpenGL} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & \mathbf{-1} & 0 \end{pmatrix} \tag{2.32}$$

[11] En la mayoría del resto de los sistemas de representación, incluyendo *Renderman* y *Direct3D*, se utiliza un modo de representación *left-handed*, lo que supone que el eje Z tenga sus valores positivos apuntando hacia el interior de la pantalla. En OpenGL son los valores negativos los que van hacia dentro.

2.5. Motores gráficos

Después de estudiar las bibliotecas de bajo nivel se han revisado varios motores gráficos, todos ellos abiertos y gratuitos, con la premisa de encontrar alguno basado en OpenGL lo suficientemente simple para el desarrollo del proyecto. En esta sección se recogen los tres más interesantes que produjo dicha búsqueda y revisión.

2.5.1. OGRE

OGRE (*Object-Oriented Graphics Rendering Engine*)[12] es un motor gráfico programado en C++ cuyo enfoque principal es la creación de videojuegos y de aplicaciones en tiempo real. Su licencia es libre, de tipo MIT.

Alguna de sus principales características son: compatibilidad con varios sistemas operativos (GNU/Linux, Windows y Mac OSX), dispone de un *framework* extensible, soporte de Direct3D y OpenGL, carga de imágenes desde varios formatos, soporte de varios tipos de mallas y para superficies de Bézier, animación esquelética,y manejo de escenas y del orden de dibujado para las transparencias.

La característica que diferencia este motor gráfico de otros es que su diseño está dirigido por la sencillez y no por cubrir una lista determinada de características. Ogre dispone de una amplia documentación tanto de las herramientas como del API.

OGRE es utilizado en algunos videojuegos comerciales.

2.5.2. CrystalSpace

Crystal Space[13] es un motor gráfico escrito en C++, basado en OpenGL y con una licencia libre de tipo GPL.

La principal característica de este motor es su capacidad de integración Blender[14], utilizando esta aplicación como método de edición de modelos, de escenas e incluso de comportamientos.

[12] http://www.ogre3d.org
[13] http://www.crystalspace3d.org
[14] Blender es una aplicación libre de modelado, animación 3D y de composición.

Se proporcionan dos manera de trabajar con este motor gráfico, una de ellas como biblioteca y otra de ellas como aplicación, cuyo nombre es CEL. Para trabajar con la biblioteca hay que utilizar C++, mientras que con CEL se utiliza Python como lenguaje de *script* no siendo necesario compilar en ningún momento.

2.5.3. Blender GameKit

Blender GameKit [Roo03] (BGK)es el motor gráfico que proporciona Blender, con licencia GPL.

BGK hace uso de su sistema de nodos para editar reglas *ECA*[15] y utiliza Python como lenguaje de *script*. La distribución de una aplicación basada en este motor gráfico depende de Blender completamente, y no se puede generar una aplicación independiente del mismo.

Su ventaja principal es la integración total del motor con la utilidad de edición, dando así apoyo a cada una de las características disponibles en Blender, como son la edición de materiales con texturas procedurales, *shadders*, edición compleja de mallas y superficies de subdivisión y otra gran cantidad de facilidades.

Se proporciona la tabla 2.1 como resumen de los tres motores.

Motor	Soporte Collada	OpenGL	Extensible mediante *scripts*	API C/C++
OGRE	Sí, mediante un *plugin*	Sí	Sí	Sí
Crystal Space	No	Sí	Sí	Sí
BGK	No	Sí	Sí, todo se basa en Python	No

TABLA 2.1: Comparativa de los motores gráficos estudiados

2.6. Interfaces gráficas de usuario

Una interfaz gráfica de usuario (GUI) es un programa software que facilita la interacción entre un humano y una computadora de forma visual. Normalmente, la bibliotecas que proporcionan el apoyo a la creación y uso de las mismas proveen al programador de un conjunto de componentes que responden a eventos. Estos eventos son normalmente provocados por un ratón o un teclado.

[15]Evento-Condición-Acción

Según Opperman [Opp02] las características que debe tener un diálogo o componente de una interfaz gráfica son:

- Debe ser **auto-descriptivo**, esto es, utilizar una metáfora válida que describa su funcionalidad.
- Debe de ajustarse a su **comportamiento esperado**.
- Debe poder ser **adaptable**, esto es, se debe poder distinguir un control específico de otro de su misma clase.
- Debe **facilitar el aprendizaje del uso del sistema**, guiando al usuario dentro del mismo.

Bajo estos criterios, se han estudiado algunas bibliotecas libres de interfaces de usuario basadas en OpenGL, como GLUT, GLOW, GLUI y libUFO.

GLUT

GLUT [Kil96] es un *toolkit* para OpenGL que entre otras utilidades permite crear menús como interfaz gráfica. El uso de GLUT está muy extendido puesto que no sólo proporciona este tipo de utilidades, sino que unifica la entrada de teclado y de ratón. GLUT está portada a múltiples sistemas operativos, incluyendo GNU/Linux, Mac OSX y Windows.

Los menús que permite crear GLUT son sobrios y están basados en *pop-ups*. Cada menú puede contener opciones u otro submenú. No dispone de un componente botón, ni de listas de opciones y no cumple con las características deseables.

GLOW y GLUI

GLOW [Azu00] es un *toolkit* que no es más que una capa encima de GLUT, proporcionando un conjunto de más completo de componentes. GLOW no permite la integración de la interfaz gráfica en un contexto de OpenGL, tan sólo permite hacerlo en una contexto separado, en otra ventana o sobre la escena que se esté representando.

Aunque GLUI [RS06] es una biblioteca más completa que GLOW, tampoco permite la sobreimpresión de la interfaz gráfica sobre la escena 3-D.

libUFO

Una biblioteca que sí permite sobreimprimir la interfaz de usuario es *libUFO*[16]. El conjunto de componentes gráficos es variado e incluye botones, listas y menús. Se utiliza XUL (*XML User interface Language*) para la creación de interfaces y el posicionamiento de los controles. No incluye soporte para imágenes sobreimpresas ni para cambiar las texturas asociadas a los controles. Así, no es posible cambiar el estilo de ninguno de los controles.

En la tabla 2.2 se proporciona una breve comparativa de las cuatro bibliotecas.

Biblioteca	Auto-descriptivos	Adaptables	Imprimibles sobre la escena 3-D	API C/C++
GLUT	No	No	No	Sí
GLOW	Sí	No	No	Sí
GLUI	Sí	No	No	Sí
libUFO	Sí	No	Sí	Sí

TABLA 2.2: Comparativa de los controles de las de interfaces de usuario estudiadas

2.6.1. Fuentes de texto

Dado que los sistemas de interfaces de usuarios estudiados no cubrían las necesidades del MARS, se procedió a revisar algunas bibliotecas para la representación de texto con OpenGL, con vistas a implementar una interfaz de usuario propia.

Las características que se buscaban en una biblioteca fueron que tuviera soporte para fuentes *TrueType*, debido a su popularidad y a la gran cantidad de fuentes existentes en este formato; y que facilitase la representación rápida de texto 3-D y 2-D.

Las bibliotecas que se revisaron fueron GLC, GLTT, FTGL y WGL.

GLC

GLC es una biblioteca que utiliza fuentes Adobe Type1 para la representación de texto. Aunque se proporciona una utilidad para convertir otros tipos de fuente, entre ellos TrueType, GLC sólo permite la representación 2-D de texto.

Esta biblioteca sólo está disponible para estaciones de trabajo SGI.

[16]http://libufo.sourceforge.net/

GLTT

GLTT[17] es un *envoltorio*[18] de la biblioteca FreeType, que brinda soporte al uso de fuentes TrueType. GLTT utiliza esta biblioteca para obtener los contornos de las fuentes y para *rasterizarlas*. En el primer caso, construye fuentes poligonales planas o con extrusión; en el segundo, crea fuentes de mapas de bits, que puede utilizar como fuentes 2-D. GLTT permite realzar suavizado de fuentes, y es libre y multiplataforma.

GLTT depende de GLUT para el dibujado de las fuentes.

FTGL

FTGL[19] es la biblioteca sucesora de GLTT. Está basada en FreeType2 y no tiene ninguna dependencia con GLUT. Aun así, mantiene todas sus ventajas y brinda además la facilidad de trabajar con fuentes *unicode* y no *monoespaciadas*.

WGL

Esta biblioteca da soporte a cualquiera de los tipos de fuentes del sistema operativo Windows. Hay mucha información sobre ella en el sitio web MSDN[20] de Microsoft, donde se publicita su facilidad de uso. Al no ser multiplatarforma no se ha podido probar y se ha descartado su uso.

En la tabla 2.3 se muestra una pequeña comparativa de todas esta bibliotecas.

Biblioteca	Soporte TrueType	Libre	Portable	2-D	3-D	Desarrollo activo	API C/C++
GLC	No	Sí	No	Sí	No	No	Sí
GLTT	Sí	Sí	Sí	Sí	Sí	No	Sí
FLGL	**Sí**	**Sí**	**Sí**	**Sí**	**Sí**	**Sí**	**Sí**
WGL	Sí	No	No	Sí	No	No	Sí

TABLA 2.3: Tabla comparativa de las bibliotecas de fuentes estudiadas

[17] http://gltt.sourceforge.net/
[18] *wrapper*
[19] http://sourceforge.net/projects/ftgl/develop
[20] http://msdn.microsoft.com/es-es/

2.7. Visión por computador

Otra de la familia de técnicas implicadas en la RA es la *visión por computador*. El bajo coste de los dispositivos de vídeo y su cada vez más frecuente integración en móviles, *tablets* y ordenadores portátiles, hacen de la misma una disciplina imprescindible para el desarrollo de la RA. La biblioteca por excelencia en este campo es OpenCV [BK08], que fue inicialmente desarrollada por INTEL, haciendo su aparición en 1999. OpenCV es libre, bajo licencia BSD, y multiplataforma, pudiendo ser compilada tanto en Windows, como en Mac OSX o en GNU/Linux.

La visión por computador es la transformación de los datos proporcionados por una imagen estática o un vídeo, en una decisión o en una nueva representación para lograr un fin particular. Estas imágenes provienen de una fuente de vídeo.

2.7.1. Fuentes de vídeo y formatos de píxel

Una fuente de vídeo es cualquier entidad que proporcione imágenes capturadas por una cámara. Aparte de una cámara en sí, una fuente de vídeo puede ser un archivo con un vídeo en diferido o un *stream* en vivo desde una cámara remota. Al final, lo que se procesa es una imagen o un conjunto de frames consecutivos, sin importar de dónde provengan mientras hayan sido capturados por una cámara.

Una imagen o frame no es más que un mapa de bits con un formato determinado. Este formato varía según la cámara o el *codec* utilizado en un archivo de vídeo (local o remoto). Cuando una imagen está descomprimida en memoria, los colores de cada píxel se codifican con una serie de bits, normalmente consecutivos. Unos de los formatos de memoria de imágenes más extendido es el RGB24_8 por su sencillez y facilidad de uso. Este formato dedica 8 bits a cada una de las componentes de color de un píxel (roja, verde y azul) lo que significa que cada píxel puede tomar un valor de entre 256^3 posibles. El orden de las componentes y el número de bits dedicados a cada una de ellas se puede alterar, existiendo formatos de píxel como el modo 565, que dedica ese número de bits a las componentes *rgb* respectivamente.

OpenCV trabaja con imágenes con cualquier formato de píxel gracias a las funciones de conversión que dispone, aunque da soporte transparente para capturar vídeo de fuentes, devolviendo imágenes con un formato BGR24_8. Nótese el orden de las componentes.

OpenGL también tiene capacidad para utilizar este formato de píxel, con lo que resulta conveniente su uso en aplicaciones donde convivan estas dos bibliotecas.

Una vez que se dispone de una imagen descomprimida de la que se conoce su formato ya se puede comenzar el proceso de la misma. A veces la imagen proporciona información suficiente por sí sola, pero normalmente hacen falta datos no contenidos en la misma para poder llegar a la meta establecida (medir una distancia, calcular una velocidad,...). En RA es crucial conocer ciertos parámetros de la cámara, que se obtendrán a través de la calibración de la misma, para poder calcular su matriz de proyección, y utilizarla a la hora de representar los objetos sintéticos. Sino se utilizase esta matriz de proyección, dichos objetos no quedarían alineados con el vídeo del mundo real.

2.7.2. Parámetros intrínsecos de la cámara

Los parámetros intrínsecos de una cámara sin distorsión se pueden expresar en función de su punto principal [CWF98], de su distancia focal y de su torsión[21] (*skew*). Normalmente, el valor de torsión es cero y se ignora. El punto principal de una cámara tiene dos coordenadas, referidas a la posición del mismo en cualquier imagen obtenida por la cámara. Este punto corresponde a la instersección del eje óptico de la cámara con el plano ficticio de la imagen obtenida.

La distancia focal va desde la lente hasta el punto donde se cruzan los rayos de luz. En OpenCV, el punto principal (que normalmente corresponde al centro de la imagen) se representa por los valores c_x, c_y; y la distancia focal que se expresa en píxeles, en relación con el largo y el alto de la imagen, y que se representa por los valores f_x, f_y respectivamente, donde $f_x = f/w$, $f_y = f/h$ siendo f la distancia focal real, w la anchura, y h la altura de la imagen.

Estos parámetros forman parte de la *matriz intrínseca* **A** de la cámara. Esta matriz junto con la *extrínseca* **R** puede transformar un punto 3-D del mundo real en su correspondiente en la imagen.

[21] diferencia entre el ángulo real y el ideal que forman el eje X y el eje Y (en la imagen proyectada)

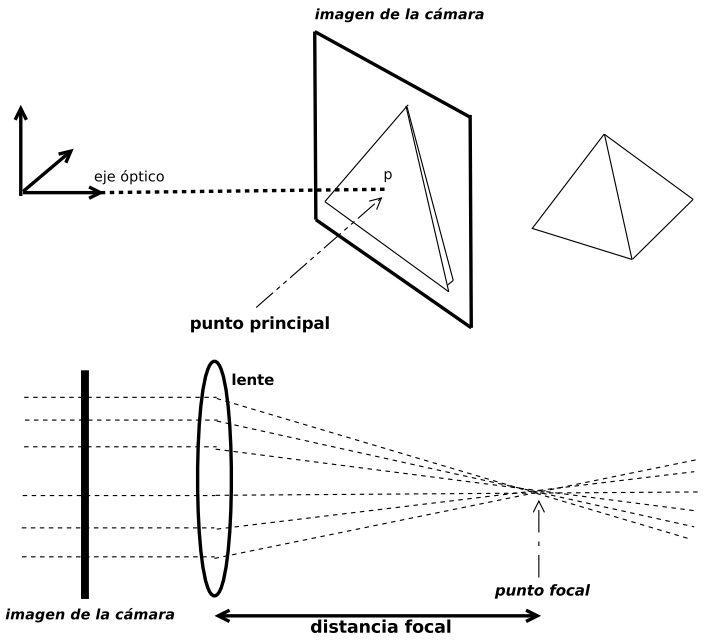

FIGURA 2.13: Parámetros intrínsecos de una cámara. Punto principal y distancia focal

La matriz que interesa para el cálculo de la de proyección en OpenGL es **A** (ec. 2.33), puesto que con los datos almacenados en ella es suficiente.

$$\mathbf{A} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{2.33}$$

Para obtener esta matriz hay que realizar un proceso conocido como calibración.

2.7.3. Calibración

El proceso de calibración estima la matriz intrínseca de una cámara a partir de imágenes recogidas por la misma. Normalmente las imágenes contienen algún tipo de patrón reconocible. El más común es un tablero de ajedrez (figura 2.14).

A través de una función de OpenCV (`cv::findChessBoardCorners()`) se pueden buscar las esquinas de un tablero en una imagen, con una precisión de píxeles. Estas posiciones deberán ser refinadas mediante otra función (`cv::FindCornerSubPix()`), obteniendo una precisión de *subpíxeles.*

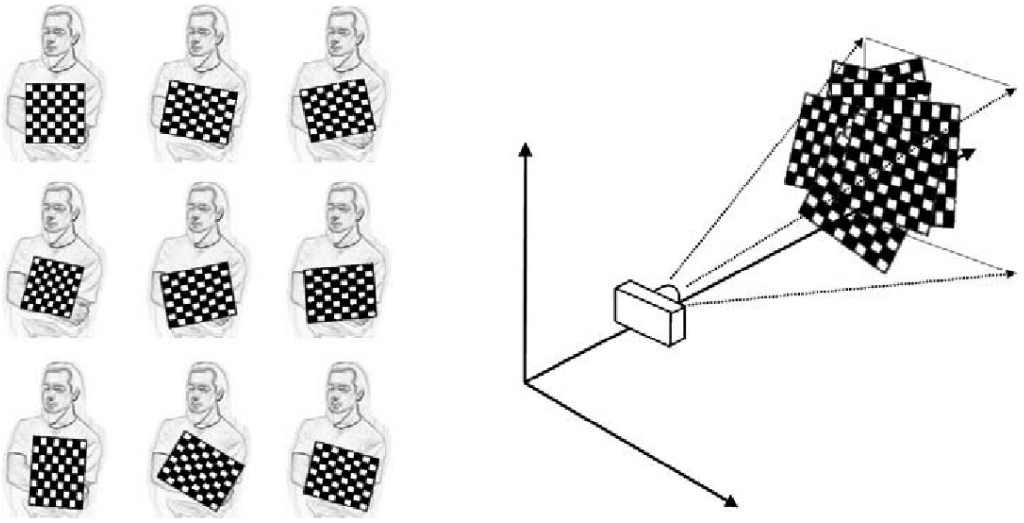

FIGURA 2.14: Calibración a través de un tablero de ajedrez (BRADSKI & KAEHLER)

El algoritmo que utiliza OpenCV para hallar el punto principal y la distancia focal, basándose en los puntos encontrados anteriormente, es el método de Zhang [Zha02]. La función utilizada en el API de C++ de OpenCV que aplica este método es `cv::calibrateCamera()`. A través del mismo se obtiene una matriz intrínseca de la cámara.

Una vez obtenida, ya se puede calcular la matriz de proyección que se usará en OpenGL para proyectar objetos virtuales sobre las imágenes capturas por la cámara calibrada.

2.7.4. Obtención de la matriz de proyección

Li [Li01] explica cómo obtener una matriz de proyección a partir de la intrínseca. La matriz obtenida al final de su artículo se muestra en la ecuación 2.34, donde H, W se refieren a la altura y anchura de la imagen *rasterizada*; y F, N a los planos *far* y *near* respectivamente.

$$\mathbf{M}_{po} = \begin{pmatrix} 2.0 \times f_x \times W & 2.0 \times (c_x/W) - 1.0 & 0 & 0 \\ 0 & 2.0 \times f_y \times H & 2.0 \times (c_y/H) - 1.0 & -2.0 \times F \times N/(F-N) \\ 0 & 0 & -(F+N)/(F-N) & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{2.34}$$

2.8. Métodos de *tracking*

En RA se conoce como *tracking* a la acción de localizar y seguir un objeto del mundo real. Para poder componer una escena real con una sintética previamente habrá que obtener información del contexto de la misma. Tratándose de una aplicación de RA la información necesaria tiene que ver con la localización de usuario que maneja el sistema (HMD, *Tablet PC*, *PDA*, teléfono móvil,...) y con los objetos que el sistema visualiza.

Mientras que existen diferentes tecnologías para posicionar y orientar al usuario, el campo de la visión por computador se hace indispinsable para analizar la vista capturada por el sistema. Esta vista será analizada. Se realizará un *tracking* visual de los objetos de interés en la misma, que servirá para componer la vista con la parte sintética gracias a la información obtenida.

2.8.1. Taxonomía general

Dado la importancia de los métodos de *tracking* visuales, la primera clasificación divide al conjunto total en dichos métodos y en métodos no-visuales.

2.8.2. Métodos de *tracking* no-visuales

Estos métodos de *tracking* realizan el seguimiento de objetos basándose en tecnologías que no requieren el uso de la visión por computador. Estos métodos proporcionan información de manera activa, es decir, están asociados a un objeto del mundo real del cual se desean obtener localización y/o orientación. Normalmente este objeto es el propio sistema, ya que habitualmente el cometido de una aplicación de RA es la localización del usuario.

Algunos ejemplos de métodos no visuales son los métodos inerciales, el GPS, métodos ópticos (lásers, diodos infrarrojos), métodos ultrasónicos o métodos mecánicos.

2.8.3. Métodos de *tracking* visuales

En entre proyecto se ha prestado especial interés en el soporte de los métodos visuales, ya que el coste de la potencia de proceso se decrementa con el tiempo y cada vez más dispositivos disponen de una cámara capaz de capturar vídeo. Esto convierte a este tipo de métodos en una solución barata para la RA.

Métodos basados en características naturales

Estos métodos extraen ciertas características de la imagen, como el color, la forma, la textura o el movimiento. La necesidad de procesar diferentes frames y de compararlos entre sí, hace que sea deseable la posibilidad de hacerlo lo más rápido posible, puesto que estamos tratando con sistemas en tiempo real y el usuario debe tener la sensación de respuesta inmediata de la aplicación. Es por esta razón por la que el uso de *descriptores visuales* se ha popularizado en nuestros días.

Un *descriptor visual* es, como indica su nombre, una descripción de las características de una imagen. Esta descripción ha de ser comparable con otra obtenida a partir del mismo algoritmo. Esta cualidad hace que sea posible clasificar imágenes con su *descriptor* como criterio. Un ejemplo de este tipo de uso es *Google Images*[22], que indexa y recupera imágenes basándose en *metadatos* y en *descriptores visuales*.

Existen ciertos *descriptores* que debido a su velocidad son aptos para el uso en aplicaciones de tiempo real. Un ejemplo es SURF (*Speeded up robust features*) [BTVG06], que extrae características robustas de la imagen a través de información localizada en crudo y de la distribución del gradiente. También proporciona la orientación de los puntos de interés localizados, como se puede apreciar a la derecha de la figura 2.15.

Gracias a estos *descriptores* y a la facilidad para ser comparados, se pueden reconocer formas en la imágenes y sus desplazamientos entre varios frames.

[22] http://www.google.es/imghp

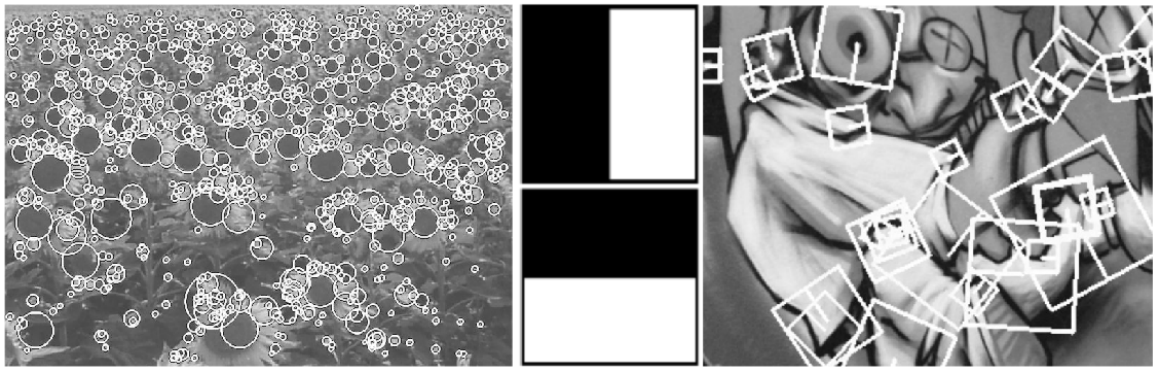

FIGURA 2.15: El *descriptor visual* SURF. A la izquierda se aprecian los puntos de interés de un campo de girasoles. A la derecha los de un *graffiti*. En el centro lo tipos del *wavelet* de Harr utilizados en SURF. (BAY, TUYTELAARS & VAN GOOL)

Métodos basados en marcas

Un caso especial de la extracción de características naturales es la localización de marcas (algunos autores como Klein hablan de *fiducials*). Aunque es posible utilizar texturas como marcas (existen bibliotecas como BazAR[23] con esta finalidad), lo más habitual es utilizar marcas sencillas con dos colores, blanco y negro, puesto que facilitan y simplifican su localización y procesamiento.

Un ejemplo de estas marcas son la utilizadas por ARToolkit[24] (figura 2.16).

FIGURA 2.16: Marcas de ARToolkit. Las mostradas se distribuyen con la propia biblioteca.

Con este tipo de métodos no se obtiene un *descriptor*; se obtiene información sobre qué marca se ha localizado y una matriz asociada que determina su posición y orientación relativa a la cámara de manera unívoca, como explica Kato [KB99]. Gracias a estos datos podremos componer imágenes sintéticas de manera precisa con la vista (en forma de imagen) donde ha sido localizada la marca (figura 2.17).

[23] http://cvlab.epfl.ch/software/bazar/
[24] http://www.hitl.washington.edu/artoolkit/

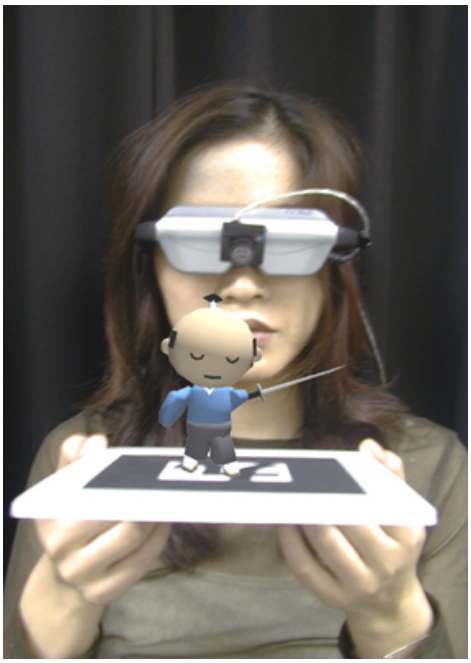

FIGURA 2.17: Ejemplo de imagen *aumentada* con ARToolkit

2.8.4. Métodos relativos y absolutos

Otra clasificación más simple consiste en dividir a los métodos de *tracking* en relativos y absolutos. Aunque la finalidad de estas métodos es la de localizar unívocamente un objeto en el espacio, algunos de ellos proporcionarán información relativa a alguna medida conocida.

Se podría plantear un sistema en el que una marca tiene asociada su posición en el espacio (será una marca conocida e inmóvil) y en el cual también se utiliza alguna técnica de análisis de movimiento entre varios frames usando características naturales. Cuando se encontrase la marca en la vista podríamos obtener la posición de los objetos visualizados o del usuario con respecto a ella. La técnica de análisis de movimiento complementaría esa información con los incrementos relativos de un frame a otro. Así, si en un momento la marca se encontrase fuera de la percepción visual del sistema, se podría seguir estimando la posición y orientación acumulando los incrementos a la última información válida recogida de la marca.

Un caso similar se daría con una brújula digital apoyada por sensores inerciales. Mientras que los cambios electromagnéticos afectarán a la brújula, no afectarán a los acelerómetros.

2.9. Bibliotecas de apoyo

La generación de audio, de voz sintetizada y la decodificación de vídeo comprimido está fuera de los cometidos de este proyecto, pero no así el uso de dichas tecnologías. Igualmente sucede con la recogida de los eventos de ratón y teclado desde el sistema operativo.

En esta sección se estudiarán tres bibliotecas de apoyo con licencia libre, candidatas a apoyar la implementación de MARS.

2.9.1. SDL

SDL[25] (*Simple DirectMedia Layer*) es una biblioteca de apoyo a la creación de aplicaciones multimedia. Es portable y proporciona una manera uniforma de crear ventanas, de obtener eventos y reproducir sonido. SDL proporciona apoyo a OpenGL desarrollando ciertas operaciones [WL04] que quedan fuera de su ámbito.

SDL ha sido utilizada con éxito en juegos libres como *Nexuiz* y en las versiones para GNU/Linux de muchos juegos comerciales como *Doom3* o *Return to Castle Wolfenstein*.

2.9.2. libVLC

VLC es un *framework* multimedia libre que se puede utilizar para decodificar y reproducir vídeo. Fue creado por VideoLan, una organización francesa sin ánimo de lucro que produce software libre, basado en la licencia GPL. Gracias al API para C++ proporcionado por libVLC[26], se puede integrar con aplicaciones propias, incluso se puede hacer uso de las primitivas de SDL para representar imágenes de manera muy sencilla.

2.9.3. Festival

Festival es sistema para la síntesis de voz, creado por Black y Taylor [BT97]. Tiene un API en C++ y su utilización es muy sencilla, ya que puede generar archivo de audio a partir de cadenas de texto. Entre otros lenguajes, Festival genera audio desde el inglés y el español.

[25] http://www.libsdl.org/
[26] http://wiki.videolan.org/Libvlc

2.10. Lenguajes de *script*

Un lenguaje de *script* es un lenguaje interpretado, cuyo intérprete se invoca desde un programa para variar su comportamiento de acuerdo al código proporcionado por el *script* en cuestión.

Ejecutar estos *scripts* desde un programa compilado permite cambiar su configuración y comportamiento sin tener que recompilarlo. Esto soslaya la necesidad de disponer de un compilador y de conocer la arquitectura en profundidad.

Se han estudiado dos de ellos, Python y Lua. El primero por su creciente uso en proyectos de todo tipo y el segundo por ser el más utilizado en videojuegos comerciales.

2.10.1. Python

Python [Lut06] es un lenguaje de programación de propósito general orientado a objetos. Python y su intérprete son libres, y han sido portados a gran cantidad de plataformas, incluyendo a Windows, OSX, GNU, iOS, Symbian, Windows CE y Android.

Python provee al programador de un API de C para poder ejecutar programas desde C o C++. Este API no ofrece una forma intuitiva de relacionar clases de C++ con clase de Python, puesto que no es su cometido. Para realizar este *binding* de clases se puede utilizar la biblioteca Boost.Python[27] [Com09] , que según sus autores facilita la interacción entre C++ y Python de manera sencilla y eficiente.

2.10.2. Lua

Lua [Ier06] es un lenguaje de *script* ligero y potente, que se puede empotrar de manera sencilla en cualquier aplicación. Su uso está muy extendido, y ha sido utilizado en proyectos de éxito como *Photoshop Lightroom* de Adobe, o *World of Warcraft* de Blizzard.

Lua dispone de una licencia libre (MIT) y según sus creadores es rápido, potente, empotrable, sencillo y pequeño (el archivo comprimido de todo el lenguaje ocupa 212 KB).

[27] http://www.boost.org/doc/libs/1_36_0/libs/python/doc/index.html

LuaBind

Luabind es una biblioteca creada por Rasterbar Software[28] que permite empotrar este lenguaje en C++. Permite relacionar clases, métodos y funciones de los dos lenguajes, además de constantes y variables de todo tipo. Luanbind es libre, con licencia MIT.

Hace uso de la metaprogramación con patrones de C++, evitando así tener que preprocesar varias veces los archivos donde se usa durante la compilación de un proyecto.

Su uso es sencillo y permite invocar *scripts* o utilizar cadenas de texto como código Lua. Inicialmente sólo carga el sistema base de Lua, pero permite importar las bibliotecas necesarias para el programador.

[28] http://www.rasterbar.com/products/luabind.html

3

Objetivos

En este capítulo se recogen los objetivos que se pretenden llevar a cabo con este proyecto de fin de carrera.

3.1. Objetivo general

El objetivo general del proyecto fin de carrera puede definirse como la *creación de una arquitectura software que facilite y acelere la creación de aplicaciones de RA basadas en diferentes métodos de* tracking. En base a este objetivo general se plantean una serie de subobjetivos específicos que se detallan en la siguiente sección.

3.2. Objetivos específicos

El objetivo general del presente proyecto se define en base a una serie de requisitos funcionales que se definen a continuación:

- **Arquitectura extensible**. La arquitectura debe dar soporte a diferentes métodos de *tracking*, facilitando en la medida de lo posible la inclusión de nuevos métodos y algoritmos.
- **Integración de *tracking***. El diseño de la arquitectura debe permitir utilizar simultáneamente diferentes métodos de *tracking*, combinando los resultados de cada método.

- **Tratamiento de fuentes de vídeo heterogéneas**. El sistema deberá permitir utilizar diversas fuentes de vídeo (con soporte hardware y software) de forma transparente al usuario. En este ámbito deberá dar soporte a la calibración de las fuentes hardware, así como la utilización de fuentes de vídeo basadas en archivos, *streams* remotos (como cámaras IP), etc.

- **Motor gráfico 3-D**. La arquitectura deberá desarrollarse con soporte específico 2D y 3D para aplicaciones de realidad aumentada, prestando especial atención al uso mínimo de recursos hardware (para garantizar su portabilidad y autonomía). El motor deberá soportar la carga de modelos 3D en algún formato estándar.

- **Prototipado Rápido**. La arquitectura deberá proporcionar mecanismos para la utilización de lenguajes interpretados (como Lua o Python) para la construcción de prototipos.

- **Bibliotecas Matemáticas**. El *framework* desarrollado en la arquitectura deberá dar soporte para los formalismos matemáticos empleados habitualmente en aplicaciones de Realidad Aumentada como representación matricial y vectorial, uso de cuaternios, etc.

- **Sonido y Audio**. El sistema deberá proporcionar utilidades para la reproducción y síntesis de voz, que permitirán su utilización en proyectos donde se requiera interacción *multimodal*.

- **Estándares Multiplataforma**. El desarrollo se basará en estándares y software libre para facilitar su posterior expansión y uso en diferentes arquitecturas hardware.

4

Método de trabajo

En este capítulo se exponen la metodología de desarrollo de software utilizada y las herramientas usadas para llevarla a cabo.

La elección de la metodología y de las herramientas se ha visto condicionada por la naturaleza del proyecto (construcción de una arquitectura multiplataforma), los requisitos funcionales (eficiencia, orientada a aplicaciones interactivas en tiempo real) y los objetivos concretos del proyecto (basada en estándares y software libre).

4.1. Metodología de desarrollo

La metodología utilizada ha sido el prototipado evolutivo (figura 4.1). Esta metodología es arriesgada, pero permite hacer una aproximación rápida al desarrollo de un prototipo funcional, que será descartado por el cliente si no está satisfecho, o que evolucionará en diferentes iteraciones gracias a la retroalimentación del mismo.

El director del proyecto propuso un concepto inicial del sistema y se realizó un análisis preliminar de los requisitos. Con este análisis de requisitos el autor realizó un diseño general del sistema, lo desarrolló y entregó como un primer prototipo, tras realizar las pruebas pertinentes. El “cliente” (el director del proyecto) aceptó el prototipo. Sobre ese

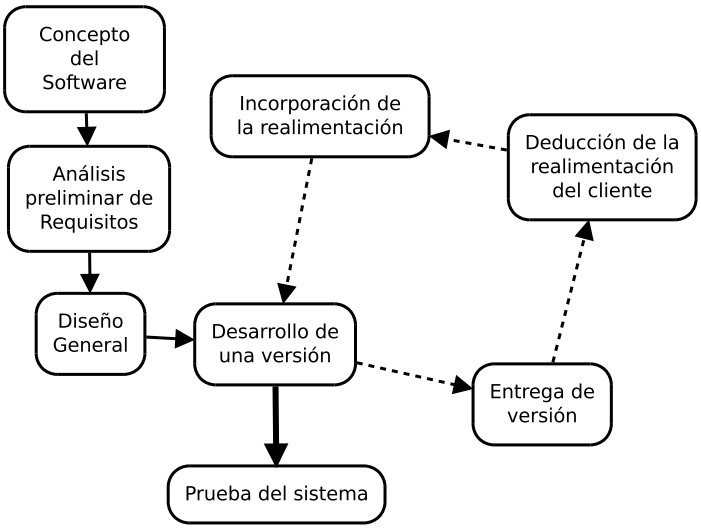

FIGURA 4.1: Prototipado Evolutivo

primer prototipo se empezó a construir el sistema, gracias a los nuevos requisitos que se proporcionaban en cada iteración (ver capítulo 6).

4.2. Herramientas

4.2.1. Lenguajes

- **C++** - El lenguaje principal utilizado ha sido C++ [Str99]. C++ que es un lenguaje de sistemas que permite una serie de abstracciones muy convenientes para el modelado de una arquitectura como la planteada en este proyecto. El polimorfismo y la herencia han sido cruciales en el diseño del mismo, y además es un lenguaje compilado. Esto último permite utilizar el compilador para comprobar los tipos y las estructuras del programa antes de la ejecución, ahorrando pruebas y esfuerzo de depurado.

- **Lua** - El lenguaje elegido para ampliar y modificar el comportamiento del sistema mediante *scripts* ha sido Lua, por su facilidad de integración con C++ y su sintaxis clara y sencilla.

4.2.2. Hardware

Las pruebas se han realizado en varias arquitecturas hardware: *i386* y *amd64*.

En las pruebas del sistema se han empleado diversas videocámaras, aunque la mayor parte de las mismas se han realizado utilizando principalmente dos modelos. Uno de ellos el modelo *Sphere AF* de Logitech[1] y otro el *USB uEye XS* de uEye[2].

Se ha hecho uso del servidor `devoreto.esi.uclm.es` de la Escuela Superior de Informática (UCLM). Algunas herramientas utilizadas en este proyecto están alojadas en una máquina virtual de dicho servidor.

4.2.3. Software

Todo el software utilizado en este proyecto es libre según la definición de la Free Software Foundation (FSF)[3] [Sta99].

Aplicaciones de desarrollo

- **Eclipse CDT** - Entorno integrado de desarrollo con soporte de C/C++. Depuración y entorno de pruebas.
- **GNU Emacs** [Sta07] - Entorno de desarrollo. Se ha utilizado para editar algunas pruebas y para escribir la documentación.
- **GNU Make** - Herramienta para la compilación incremental, con soporte multiproceso.
- **GCC** [CS03] - La colección de compiladores GNU. En concreto se ha utilizado el compilador de C++ (*g++*).

Aplicaciones de edición 3-D

- **Blender 3D** - Programa de modelado, animación, *renderizado* 3-D y post-producción.

[1] http://www.logitech.com
[2] http://www.ids-imaging.com
[3] http://www.fsf.org/about/what-is-free-software

Control de Versiones

- **Mercurial** - Sistema de control de versiones distribuido.

  Mercurial [O'S09] ha sido el sistema de control de versiones elegido por la facilidad de uso y la versatilidad que proporciona, muy parecida a la de *GIT* [Loe09].

  - **Mercurial Eclipse** - *Plugin* para Eclipse que actúa como cliente de Mercurial.

Control de Proyectos

- **Redmine**[4] -Sistema de control de proyectos basado en Ruby. Compatibilidad con control de versiones. Permiten la gestión de tareas y soporta la estimación del tiempo de realización. Esta estimación la realizará el creador de la tarea y se ajustará con el tiempo real utilizado para la misma.

  Redmine permite la comunicación sencilla de los nuevos requisitos a través de tareas.

  El cliente (el director del proyecto) proporcionaba la retroalimentación y los nuevos requisitos a través de esta plataforma.

  Redmine también soporta el reporte de *bugs*, el reporte de *fixes* (correcciones de *bugs* concretos) y la integración de comentarios realizados a través del control de versiones en las tareas asociadas a un proyecto.

  El cliente proporcionaba la retroalimentación y los nuevos requisitos a través de esta plataforma.

Documentación

- **LaTeX** [CLM+03] - Herramienta de maquetación con la que ha sido realizado este documento.
- **Doxygen** - Sistema de documentación de código fuente [dox10]. Compatible con C++. Se ha utilizado para realizar el manual de referencia de MARS.
- **InkScape** - Programa de edición de imágenes vectoriales.
- **Gimp** - Programa de edición de mapas de bits.

[4] http://www.redmine.org

Bibliotecas

- **OpenGL** - Biblioteca de representación 3D. Abierta, multiplataforma y con implementaciones libres. La mayoría del hardware con aceleración gráfica tiene controladores que permiten usar esta biblioteca como API 3-D.
- **libPThreads** - Biblioteca de apoyo al uso de hilos (*threads*) POSIX. Existen implementaciones para la gran mayoría de sistemas operativos que se utilizan hoy en día.
- **SDL** - *Simple DirectMedia Layer*. Bibliotecas para la creación de videojuegos y aplicaciones multimedia.
- **OpenCV** - Biblioteca de visión por computador que proporciona métodos de procesado de imágenes.
- **UEYE Libs/API** - Biblioteca/API privativa para el soporte de la cámara *UEye XS*. El proyecto se puede construir sin la necesidad de tener estas bibliotecas en el sistema, dando la opción al usuario de tener un sistema completamente basado en software libre.
- **colladaDOM** - Bibliotecas que implementan el modelo DOM [Mar02] del formato basado en XML de Collada.
- **FTGL** - Biblioteca de representación de fuentes de texto en OpenGL.
- **Festival Speech Synthesis System** - Biblioteca de síntesis de voz.
- **ARToolKit** - Biblioteca para el reconocimiento de marcas de RA.

Sistema Operativo

El sistema operativo utilizado ha sido GNU/Linux, en concreto la distribución Debian[5]. La *shell* usada ha sido BASH y el sistema de escritorio GNOME.

[5] http://www.debian.org

5

Arquitectura de MARS

En este capítulo se mostrarán los resultados obetenidos al aplicar la metodología elegida (sección 4.1).

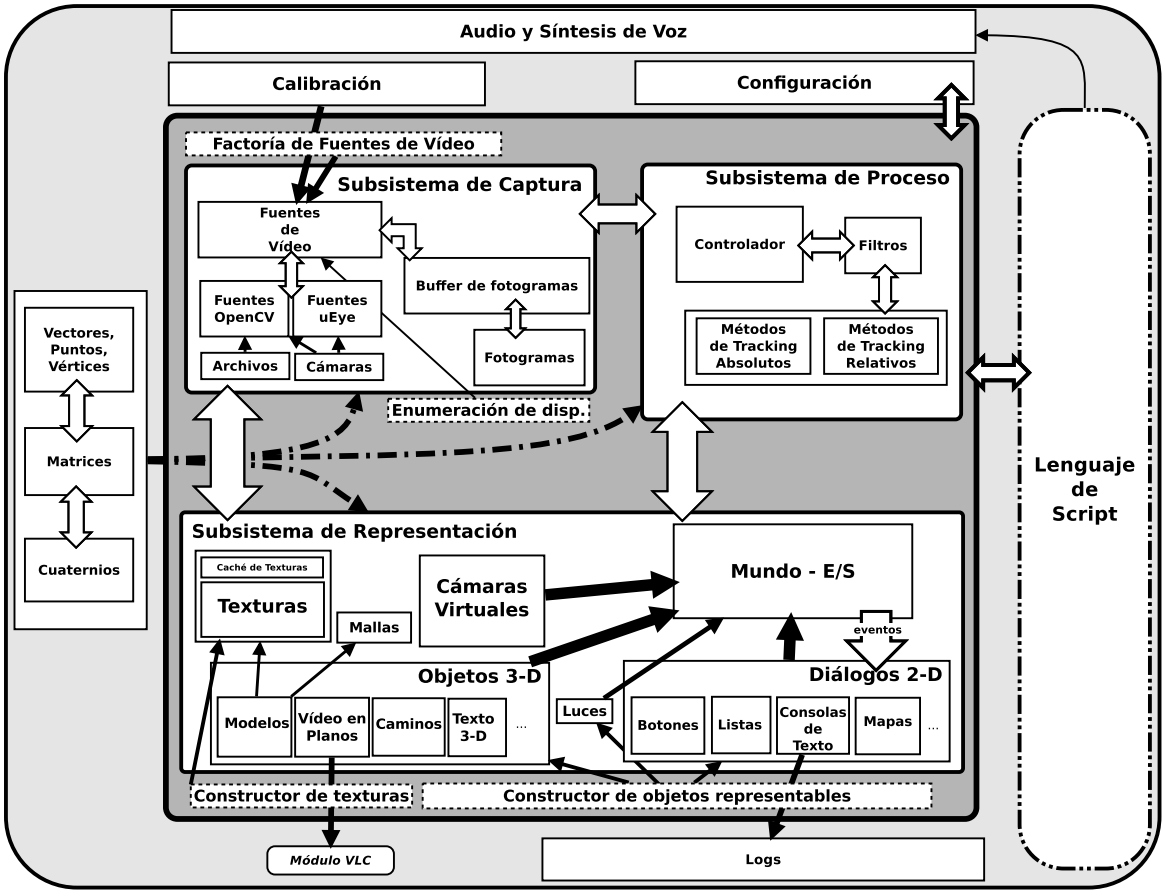

FIGURA 5.1: Estructura de MARS

MARS se diseñó como una arquitectura con tres subsistemas, como se muestra en la figura 5.1. Estos tres subsistemas son el de captura (sección 5.1), el de proceso (sección 5.2) y el de representación (sección 5.3). Además, en dicha figura se muestran las utilidades que brinda el sistema para la representación matemática, la calibración (sección 5.5), la configuración (sección 5.6.2), la reproducción de audio y de voz sintetizada (sección 5.6.5), y del lenguaje de *script* (sección 5.6.6).

MARS además proporciona un *framework* que facilita el desarrollo de aplicaciones de RA reuniendo en una sola biblioteca las utilidades necesarias. El programador sólo tiene que implementar las interfaces proporcionadas para crear de forma rápida nuevos métodos de *tracking* o filtros. Para facilitar la interacción entre el procesado y el sistema de filtros, se proporcionan las clases matemáticas adecuadas para la representación de la posición y de la orientación, así como sus operaciones más importantes. De esto modo, la estructura del subsistema de proceso permite un uso completamente modular del mismo, pudiéndose utilizar uno o varios métodos de *tracking* y aplicar al resultado del mismo el filtro más conveniente en cada caso.

MARS también facilita el uso de diferentes dispositivos de vídeo, tanto locales como remotos, y gracias a la estructura modular del subsistema de captura permite añadir soporte para nuevos dispositivos sin tener que cambiar ninguna otra parte de la arquitectura. En el momento de escritura de este documento, se daba soporte a todos los dispositivos compatibles con *Video4Linux* y a los dispositivos de la empresa *uEye*, cuyo uso está extendido en este área de investigación, por la calidad de imagen y las funcionalidades que ofrecen. Ya que los dispositivos pueden ser tan diversos, también se proporciona una utilidad de calibración de cámaras. Estos datos se procesarán para obtener la matriz de proyección correspondiente a cada uno de ellos y poder así componer de manera creíble el vídeo real con los objetos sintéticos. Estas matrices, la de proyección y la de los parámetros intrínsecos de la cámara pueden ser utilizadas internamente por cualquier método de *tracking* que los necesite para realizar cálculos sobre ellas. Esto homogeneiza la calibración y libera al programador tener que realizar una por cada método.

El diseñador puede hacer uso de la herramienta de modelado que domine para crear modelos 3-D compatibles con MARS, gracias al soporte nativo del estándar Collada. Desde este metaformato, se pueden importar mallas triangulares y poligonales, así como archivos de textura y los valores necesarios para realizar el mapeado de las mismas. Así mismo, el programador puede implementar la interfaz propuesta para añadir nuevos tipos de objetos representables, y hacer uso de ellos en la aplicación de forma transparente. Como conjunto inicial, para cubrir la mayoría de las necesidades de una aplicación de RA MARS proporciona objetos representables para dibujar rutas, planos con vídeo, texto en 3-D, objetos para el depurado y objetos para representar puntos WIFI.

Además, se incluye una implementación especial de los objetos representables, que corresponde a un conjunto de *widgets* 2-D. El programador podrá crear nuevos *widgets* implementando la interfaz correspondiente. MARS proporciona un conjunto propio compuesto por botones, listas, imágenes 2-D, mapas y consolas de texto. Todos ellos reciben eventos de teclado y de ratón, y pueden procesar dichos eventos según dicte el comportamiento que deban desempeñar.

Aparte de con las utilidades matemáticas y de calibración, MARS completa los tres subsistemas principales con utilidades para la creación de *logs*, para la reproducción de audio y síntesis de voz y para la configuración global del sistema.

Con motivo de poder realizar cambios en aplicaciones ya compiladas, MARS proporciona la posibilidad de utilizar LUA como lenguaje de *script*. De esto modo, se permite la ejecución de archivos que contengan programas escritos en este lenguaje y que utilicen los *bindings* de las clases de MARS. A través de estos *scripts* se permite la creación de modelos 3-D a partir de archivos Collada, así como su posicionamiento, la creación de *widgets* y la asociación de eventos de *click* y de selección. También se permite la reproducción de sonidos y la generación de voz sintetizada.

MARS es completamente modular y los subsistemas están desacoplados. La forma en la que interactúan los subsistemas es sencilla y transparente al programador, que se ha de limitar a implementar las interfaces de los objetos que desee crear, ya sean métodos de *tracking*, filtros o nuevas capacidades de representación o captura.

5.1. Subsistema de captura

En esta sección se describirá el subsistema de captura, su cometido, su funcionalidad y su implementación.

5.1.1. Fuentes de Vídeo

Una fuente de vídeo es cualquier recurso que proporcione una secuencia de frames. Normalmente esta secuencia se mantiene en el mismo orden que en el momento de la captura.

OpenCV es la biblioteca por excelencia para la visión por computador. Las estructuras que proporciona implementan todas la operaciones necesarias para el tratamiento de imágenes y su uso está ampliamente extendido y aceptado. OpenCV utiliza un tipo matriz propio (`cv::Mat`), adaptado a las necesidades de esta tecnología. Por este motivo, MARS utiliza esta estructura internamente para representar imágenes y para proporcionarlas a los diferentes métodos de *tracking*.

Los frames del sistema son matrices de OpenCV (`cv::Mat`) y para su almacenamiento se ideó una clase que contiene un *vector*[1] de `cv::Mat` con los últimos frames capturados. El número de frames puede ser variable. Por defecto se almacenan los últimos cinco. Estos fotogramas podrán ser utilizados en métodos que necesiten procesar una secuencia de frames para obtener datos acerca de los cambios que se producen entre ellos. Esta clase, llamada `FramesBuffer` encapsula el `vector` y proporciona los métodos de acceso a un frame determinado. También proporciona un método para añadir nuevos frames, que será utilizado por una fuente de vídeo cada vez que capture uno nuevo. En el caso de que el vector esté lleno, el frame más antiguo será eliminado. Para evitar que al eliminar un frame de esta estructura ya no pueda ser procesado, cuando se pide un frame a la clase esta devuelve una copia (una nueva `cv::Mat`) y no una referencia.

Una webcam, un archivo de vídeo y una cámara remota son consideradas fuentes de vídeo. Una videocámara remota (en el caso de este proyecto, un cámara IP) proporciona el acceso al archivo de vídeo capturado a través de *streaming*. Gracias a esta tecnología, no es necesario descargar el archivo completo para poder visualizarlo, sino que se puede descargar y visualizar a la vez. Así, la fluidez y el retardo de este vídeo con respecto a la captura original, dependerá de la conexión entre la videocámara y el cliente que haga uso

[1] *vector* es un *template* de la biblioteca estándar de C++ que gestiona de manera adecuada *arrays* variables de elementos de un tipo determinado.

de ella. El protocolo que se utiliza para este tipo de conexiones es RTSP [S+98], que utiliza TCP para el control y UDP para los datos.

Para poder dar soporte a estas dos posibles fuentes de vídeo (dispositivos hardware y archivos) se ha diseñado una jerarquía (figura 5.4) que se corresponde directamente con las clases diseñadas e implementadas.

Así, una fuente de vídeo cualquiera está representada por la clase `VideoSource`. `VideoSource` recoge las características de una fuente de vídeo que son importantes para la RA, como pueden ser su posición (punto 3-D *eye*), hacia dónde mira (vector 3-D *lookTo*) y cuál es la rotación (vector 3-D *up*). También almacena datos importantes como la matriz de calibración de la cámara y la cámara virtual relacionada. Esta clase contiene un `FramesBuffer` y lo encapsula, proporcionando métodos para acceder a un frame determinado, y utilizándolo internamente para almacenar las imágenes capturadas de la fuente de vídeo. También incluye un nombre, que se utilizará como identificador de la fuente de vídeo y los atributos necesarios para la creación de un hilo con PThreads.

Cabe destacar la información de calibrado asociada, que será crucial a la hora de componer de manera creíble los objetos sintéticos 3-D. Si el sistema contiene esta información sobre la fuente de vídeo utilizada (que se identifica a través de su nombre), esta se cargará y se procesará en la inicialización del `VideoSource`. Para calcular la matriz de proyección de la cámara virtual asociada, a partir de los datos de calibración, se utiliza la operación `calcProjectionMatrixFromCameraMatrix()`. Se puede obtener más información acerca de este proceso en el apartado 5.5.

`VideoSource` es una clase abstracta y proporciona el método virtual puro `ThreadCaptureFrame()`, que ha de ser implementado por cada una de las especializaciones concretas, según su forma de capturar frames. `ThreadCaptureFrame()` es ejecutado de manera concurrente en el sistema. De esto modo, cada fuente de vídeo ejecutará la parte encargada de capturar frames en un hilo independiente. Cada vez que se capture una imagen se añadirá al `FramesBuffer` de la fuente de vídeo (figura 5.2).

En la figura 5.3 se muestra el diagrama secuencial de la petición de un frame y la forma que ha de tener el método `ThreadCaptureFrame()` a modo de diagrama de flujo.

Las dos especializaciones de `VideoSource` son: `VideoFile`, que representa a un archivo de vídeo, y `VideoDevice`, que representa a un dispositivo físico de vídeo. La única información nueva que se añade a los archivos de vídeo es el número de frames que contiene. Un `VideoDevice` añade a su antecesor un método que devuelve si contiene frames o no. Un archivo de vídeo en buen estado siempre contiene algún frame. Sin embargo, puede que durante la inicialización del dispositivo hardware, la estructura que lo representa

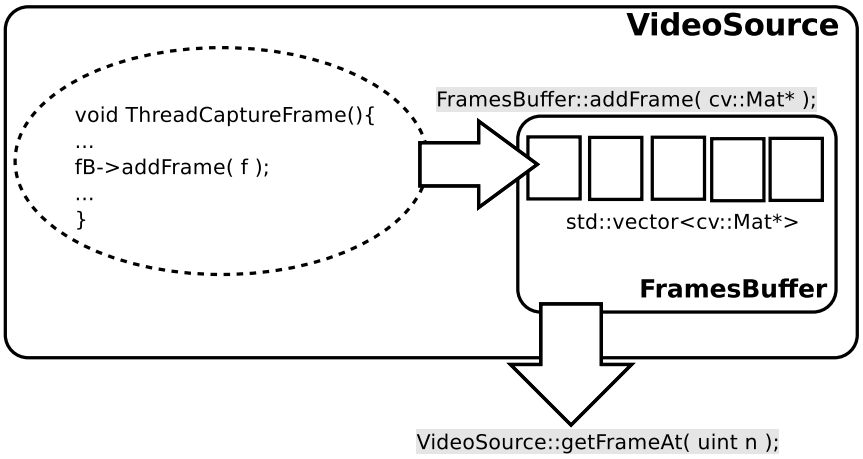

FIGURA 5.2: Captura de frames en un `VideoSource` concreto

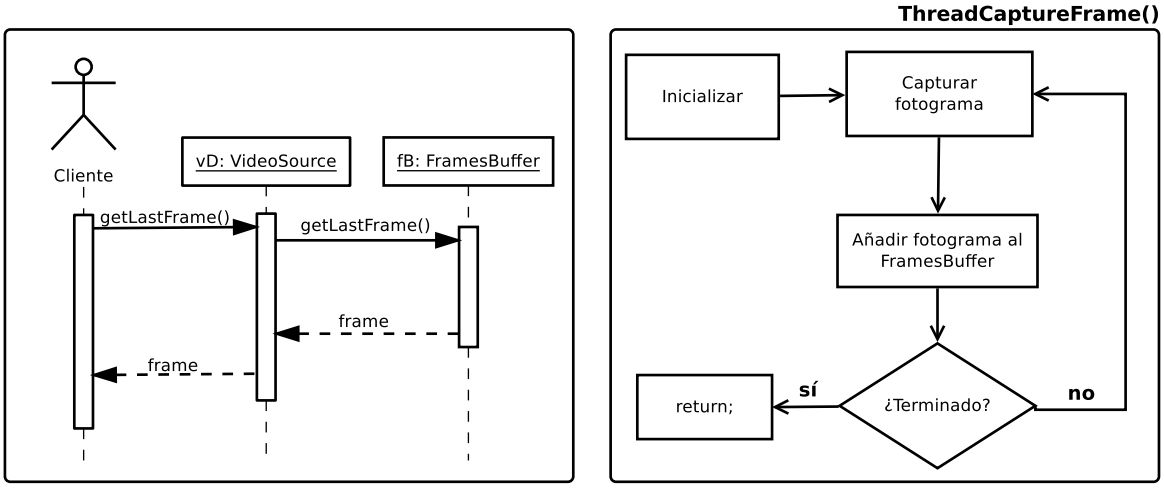

FIGURA 5.3: Petición de un frame y diagrama de flujo de `ThreadCaptureFrame()`

no contenga ningún frame aunque ya sea accesible por el resto del sistema. La solución para este caso concreto es devolver una imagen predeterminada (a modo de carta de ajuste) mientras el dispositivo no captura imágenes.

Este diseño, en el que las fuentes de vídeo concretas tienen que respetar un interfaz común, se creó con la idea de homogeneización. Así, el acceso a los recursos que proporciona cualquier fuente es común a todas ellas. Es muy probable que en un futuro sea necesaria la incorporación de nuevos tipos de fuentes, y de hecho, durante el desarrollo de este proyecto fue necesaria la incorporación de soporte para un tipo que no estaba soportado desde el principio. Gracias a este diseño, sólo hubo que implementar una nueva clase concreta para dicho tipo de fuente (una cámara *uEye XS*).

`VideoFile` y `VideoDevice`

La única implementación concreta de `VideoFile` en el momento de redacción de este documento es `VideoFileOpenCV`, que delega la responsabilidad de recuperación de frames en OpenCV. `VideoDevice` tiene dos implementaciones concretas: `VideoDeviceOpenCV`, que da soporte a todos los dispositivos compatibles con OpenCV; y `VideoDeviceUEye`, que da soporte a la cámara *UEye XS* (no compatible con OpenCV).

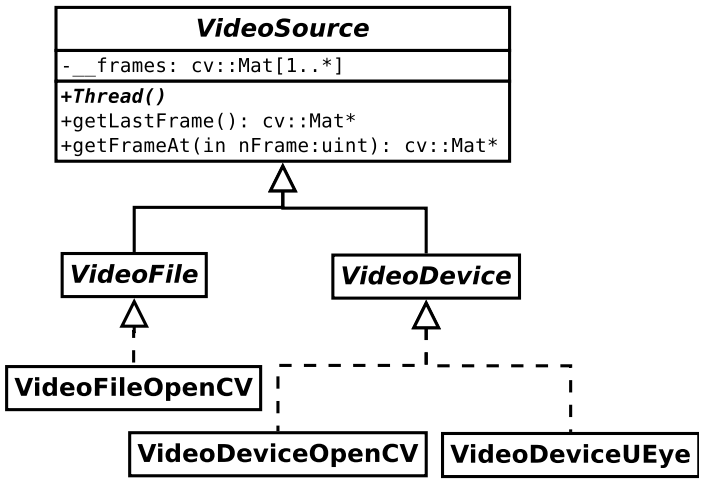

FIGURA 5.4: Jeraquía de fuentes de vídeo

OpenCV utiliza la clase `cv::VideoCapture` (diferente a `mars::Videocapture` que se verá más adelante) como abstracción de las de fuentes de vídeo a las que da soporte. Para la realización de este proyecto se ha compilado una versión de OpenCV con soporte de archivos de vídeo basado en FFMPEG y con soporte de dispositivos de vídeo basado en *Video4Linux 2*. Así, la idea inicial de utilizar OpenCV para encapsular el acceso a cualquier tipo de fuente de vídeo pareció tentadora, pero se desechó puesto que dos de los requisitos del sistema eran la modularidad y la capacidad de extensión.

En `VideoFileOpenCV` y `VideoDeviceOpenCV` se incluye un objeto de este tipo, que se utilizará para recoger los frames.

```
1 protected:
2     cv::VideoCapture* cap;
```

El constructor de esta clase requiere una cadena de texto que contenga la ruta del vídeo en el sistema de archivos. En la inicialización de la clase se abre el archivo de vídeo y se obtienen algunas propiedades del mismo, como son el tamaño del frame y el número frames por segundo (ver listado 5.1).

```
1  try{
2    this->cap = new cv::VideoCapture(path);
3   } catch (cv::Exception e) {
4    std::cout << "Error: " << e.err << std::endl;
5    throw e;
6   }
7  if (!cap->isOpened()){
8    __done = true; // This stops the thread
9    framesCount = 0;
10   return;
11  }

13 framesCount = (int)   cap->get(CV_CAP_PROP_FRAME_COUNT);
14 fps         = (float) cap->get(CV_CAP_PROP_FPS);
15 width       = (int)   cap->get(CV_CAP_PROP_FRAME_WIDTH);
16 height      = (int)   cap->get(CV_CAP_PROP_FRAME_HEIGHT);
```

LISTADO 5.1: Detalle del constructor de `VideoFileOpenCV`

Se calcula el número de milisegundos que se muestra cada frame y se utiliza en el método `VideoFileOpenCV::ThreadCaptureFrame()` para controlar la velocidad de reproducción del vídeo (Listado 5.2 líneas 20-23). Si no se realizase esta espera pasiva, el vídeo se reproduciría tan rápido como fuera posible, que no correspondería con la velocidad a la que se grabó. Tras la espera, el frame se añade al `FramesBuffer` (línea 25), que a partir de eso momento será accesible por un método de *tracking* cualquiera.

```
1  void VideoFileOpenCV::Thread(){
2    Mat* frame;

4    Uint32 ticks0;
5    Uint32 ticks1;

7    float ticksPerFrame = 1000.0 / fps;

9    ticks0 = SDL_GetTicks();
10   ticks1 = ticks0;

12   while(!__done && (nFrame < framesCount) ){

14     frame = new Mat();
15     *cap >> *frame;
16     nFrame++;

18     while (!__done &&  ( (ticks1 - ticks0) < ticksPerFrame)  ){
19       threadWait(ticksPerFrame*0.10);
20       ticks1 = SDL_GetTicks();
21     }

23     fBuffer->addFrame(frame);
24     ticks0 = ticks1;
25   }
26 }
```

LISTADO 5.2: Resumen de `VideoFileOpenCV::ThreadCaptureFrame()`

Gracias a la abstracción proporcionada por OpenCV, la manera de implementar `VideoDeviceOpenCV` es muy parecida. La diferencia fundamental entre estas dos implementaciones concretas es la forma de referirse a la fuente de vídeo. Un archivo necesita una ruta; un dispositivo necesita un número de identificación que depende del sistema opera-

tivo. En GNU/Linux este número de dispositivo coincide con el número de dispositivo de *Video4Linux* (por ejemplo, `/dev/video0` corresponde al 0). La diferencia entre un constructor y otro es mínima (listado 5.1 y listado 5.3 - línea 2).

```
1  try{
2    cap = new cv::VideoCapture(nVideoDevice);
3   } catch (cv::Exception e) {
4    std::cout << "ERROR: " << e.err << endl;
5    std::exit(-1);
6   }
7  if(!cap->isOpened()){  // check if we succeeded
8    std::cout << "ERROR: opening device number " << nVideoDevice << endl;
9    std::exit(-1);
10  }
```

LISTADO 5.3: Detalle del constructor de `VideoDeviceOpenCV`

El método `ThreadCaptureFrame()` es prácticamente igual al de *Video*, pero sin control de velocidad de reproducción. Este control no es necesario puesto que lo proporciona el propio dispositivo.

`VideoDeviceUEyeXS`

Esta clase proporciona la compatibilidad de MARS con las cámaras *UEye XS*. Un `VideoDeviceUEyeXS` «es un» `VideoDevice`. El diseño jerárquico queda justificado con la inclusión de esta implementación concreta, ya que el método de captura de OpenCV es incompatible con esta videocámara.

La forma de capturar frames de este dispositivo depende del *driver* y del API del vendedor de la cámara, que es privativo y no compatible con *Video4Linux*. Se pueden distinguir las funciones de este API en el código de MARS gracias a su prefijo «`is_`».

La construcción de estos objetos requiere un parámetro de tipo `UEYE_CAMERA_INFO` que recoge la información del dispositivo. Esta información la proporciona el propio API a través de la enumeración de dispositivos.

`VideoDeviceUEye::getUEyeCameraList()` proporciona esa enumeración.

La inicialización es completamente diferente a la de las otras clases (ver listado 5.4), y se realiza según la información proporcionada por el manual de programación del fabricante [Gmb09] de la cámara.

```
1  __hCam = (HIDS) __cameraInfo.dwDeviceID; // The enumarated cam
2  int ret = is_InitCamera(&__hCam, NULL);

4  is_EnableAutoExit(__hCam, IS_ENABLE_AUTO_EXIT);

6  if (ret == IS_SUCCESS){
7    SENSORINFO sInfo;                  // sensor info
8    is_GetSensorInfo(__hCam, &sInfo); // query it

10   __maxW = sInfo.nMaxWidth;  // Width
11   __maxH = sInfo.nMaxHeight; // Height
12   __bitsPerPixel = 24;
13   __colorMode = IS_CM_BGR8_PACKED;

15   is_SetColorMode(__hCam, __colorMode);
16   // Allocate memory for the image
17   is_AllocImageMem(__hCam, __maxW, __maxH, __bitsPerPixel, &__imageMem, &__memId);
18   is_SetImageMem(__hCam, __imageMem, __memId);

20   int xPos, yPos = 0;
21   is_SetAOI(__hCam, IS_SET_IMAGE_AOI, &xPos, &yPos, &__maxW, &__maxH);
22   is_SetImagePos(__hCam, 0, 0);
23   is_SetImageSize(__hCam, __maxW, __maxH);
24  }

26 fps = 15; width = __maxW; height =__maxH;
27 }
```

LISTADO 5.4: Detalle de la inicialización de `VideoDeviceUEye`

La captura de frames se realiza en formato crudo. Para ello se configura la cámara con los parámetros que determina el tipo de representación en memoria que se ha utilizado (figura 5.5).

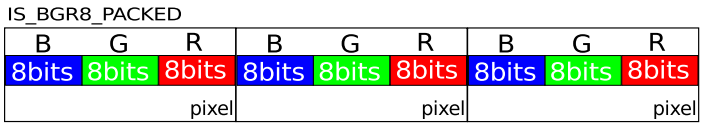

FIGURA 5.5: Formato de la imagen de uEye en memoria.

También es necesario reservar y proporcionar un espacio de memoria que será utilizado para volcar la imagen (listado 5.4 línea 23 - `__imagemem` es de tipo `char*`).

Para que el sistema sea homogéneo, esta clase ha de transformar las imágenes que captura en matrices de OpenCV (`cv::Mat`). Las matrices de OpenCV proporcionan un constructor que acepta una `cv::IplImage`, que es la estructura básica que proporciona para el almacenamiento de imágenes con un formato determinado. Para realizar la conversión desde memoria, primero se crea una imagen de este tipo y se configura con los parámetros deseados (bits por *pixel*, profundidad de color, ...) según el formato de memoria elegido.

Después se utiliza el constructor `cv::Mat(IplImage*, bool)` para construir la imagen (listado 5.5, línea 24).

```
1  if (is_FreezeVideo(__hCam, IS_WAIT) == IS_SUCCESS){ // Capture

3    IplImage i;
4    IplImage* img = &i;
5    img->nSize=112;
6    img->ID=0;
7    img->nChannels=3;
8    img->alphaChannel=0;
9    img->depth=8;
10   img->dataOrder=0;
11   img->origin=0;
12   img->align=4;
13   img->width=__maxW;
14   img->height=__maxH;
15   img->roi=NULL;
16   img->maskROI=NULL;
17   img->imageId=NULL;
18   img->tileInfo=NULL;
19   img->imageSize=3*__maxW*__maxH;
20   img->imageData=(char*)__imageMem; // pointer to image data
21   img->widthStep=3*__maxW;
22   img->imageDataOrigin=(char*)__imageMem;

24   frame = new Mat(img, true); // create cv::Mat

26   fBuffer->addFrame(frame);   // add frame
27  }
```

LISTADO 5.5: Transformación de la imagen en una `cv::Mat`

Esta conversión se lleva a cabo en `VideoDeviceUEye::ThreadCaptureFrame()` En este mismo método se realiza el almacenamiento de la matriz resultante (línea 26).

5.1.2. Creación y control de las fuentes de vídeo

Un `VideoSource` es la entidad del sistema que representa de manera unívoca una fuente de vídeo, pero carece de toda la funcionalidad necesaria para su ejecución. Se hace necesario el uso de otra clase que cree el hilo de ejecución de cada uno de ellos.

Para tal cometido sirve la clase `mars::VideoCapture`. Esta clase agrega fuentes de vídeo que correspondan a un mismo tipo de implementación. Esta decisión ha sido motivada por la necesidad de separar los diferentes tipos de formas de acceder a las fuentes de vídeo. Nótese que si un dispositivo es compatible con OpenCV de manera directa, no existe la necesidad de incluir soporte para otras formas de acceso al vídeo (que pueden ser privativos, como en el caso de los *drivers* de *uEye XS*).

Se hace uso de una especialización de nuevo. Cada una de las especializaciones agrega una lista de fuentes de video por cada tipo soportado. Este diseño se muestra en la figura 5.6.

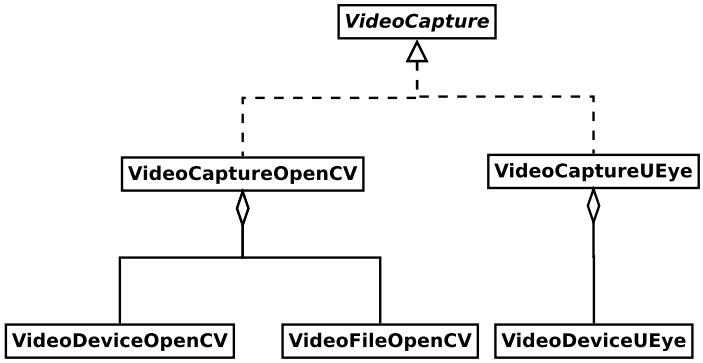

FIGURA 5.6: Jerarquía y agregación de `VideoCapture`

Un `VideoCapture` proporciona un método sobrecargado para la agregación de `VideoSources` de los dos tipos básicos:

```
1    void addVideoSource_(VideoDevice* vD);
2    void addVideoSource_(VideoFile* vD);
```

Este método añade una fuente de vídeo a una lista e inmediatamente después ejecuta su método `ThreadCaptureFrame()` en un hilo.

Cada especialización se encarga de crear un `VideoSource` según los parámetros que se le pasan a su método sobrecargado `addNewVideoSource()`. Los parámetros dependerán de la forma de creación específica del mismo. Se puede ver el caso concreto de un *VideoCaptureOpenCV* en el listado 5.6. Nótese que se crea el tipo adecuado dependiendo de estos parámetros, lo que facilitará que la creación de los mismos sea transparente al cliente.

```
1  void VideoCaptureOpenCV::addNewVideoSource(unsigned dev,const std::string& name){
2    VideoCapture::addVideoSource_(new mars::VideoDeviceOpenCV(dev, name));
3  }

5  void VideoCaptureOpenCV::addNewVideoSource(const std::string& path, const std::string&
       name){
6    VideoCapture::addVideoSource_(new mars::VideoFileOpenCV(path, name));
7  }
```

LISTADO 5.6: Detalle de `VideocaptureOpenCV`

En la figura 5.7 se muestra un esquema con el funcionamiento. Cuando se añade una fuente de vídeo a través de un `VideoCapture`, este se encarga de la su creación y de iniciar su hilo de captura de frames. En la figura 5.8 un diagrama secuencial de la creación. El cliente sólo que tiene pasar los parámetros adecuados al método `addNewVideoSource()` y este se encargará del resto.

La otra implementación de `VideoCapture` es `VideoCaptureUEye` y proporciona la forma de crear y de poner en ejecución objetos de la clase `VideoDeviceUEye`.

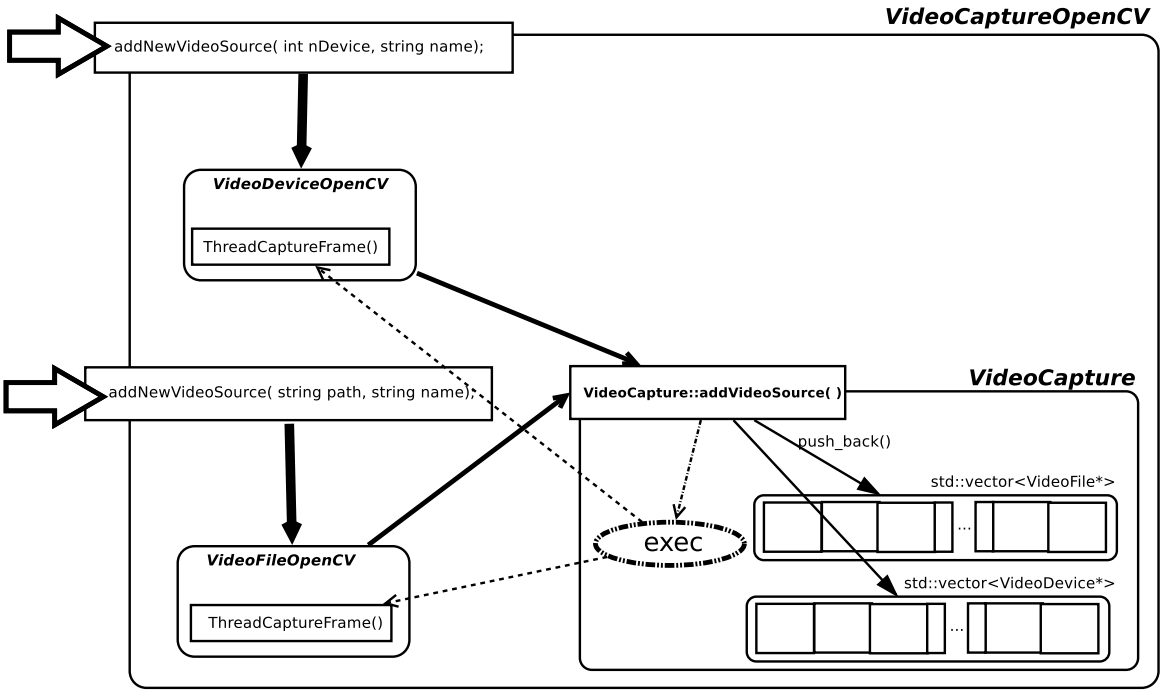

FIGURA 5.7: Esquema de funcionamiento de un VideoCaptureOpenCV

Cuando un `VideoCapture` termina, este detiene la ejecución de todos los `VideoSource` asociados y los destruye. Esto es importante puesto que la experiencia en las primeras iteraciones del desarrollo demostró que el cliente (programadores que utilizaron una primera versión de MARS) casi siempre olvidaba liberar memoria o no sabía cuándo hacerlo. También es importante puesto que se utiliza un lenguaje de *script* para instanciar objetos de nuestro sistema y es preferible liberarle de la responsabilidad de destruirlos.

5.1.3. Patrones utilizados

El diseño de este subsistema corresponde con el patrón «*factoría abstracta*» [GHJV95], donde `VideoCapture` desempeña el papel de fábrica abstracta, `VideoCaptureUEye` y `VideoCaptureOpenCV` de fábricas concretas, y `VideoFile` y `VideoDevice` harían de productos.

La instanciación de un `VideoCapture` se realiza con una *fachada* (patrón *facade*), que se corresponde con la clase `VideoCaptureFactory`. Esta *fachada* sirve además para mantener una lista de objetos `VideoCapture`, que serán destruidos al finalizar la ejecución de forma transparente.

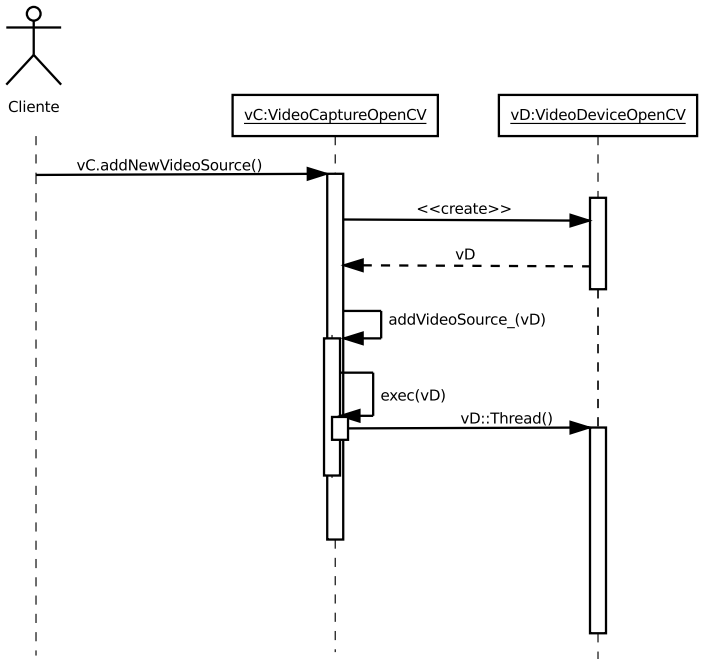

FIGURA 5.8: Diagrama de secuencia de la creación de un `VideoDeviceOpenCV`

Como no es deseable permitir la creación de varias instancias de la *fachada* o de la *factoría*, se ha utilizado el patrón *singleton* (ver anexo B) para asegurar que su existencia sea única.

Estos dos *singletons* (que corresponden a una instancia de un objeto) serán destruidos al finalizar el programa de manera automática, liberando al programador de la responsabilidad de hacerlo.

5.2. Subsistema de proceso

En esta sección se describe el subsitema de proceso. El subsistema de proceso es el encargado de proporcionar una manera de ejecutar métodos de seguimiento diferentes sobre el video capturado y de interpretar, agrupar y filtrar la información que proporcionan.

5.2.1. Métodos de *tracking*

Un método de *tracking* está representado por la clase `TrackingMethod` que encapsula una lista de *percepciones*. Uno de los cometidos principales de MARS es facilitar el seguimiento del usuario (en realidad de la cámara que maneja), esto es, localizarlo y orientarlo en el espacio. Una percepción es una estructura (`tUserL`) que almacena la posición (*eye*), el vector que indica hacia dónde mira ($\vec{lookTo}$) y el vector de rotación ($\vec{up}$, perpendicular a $\vec{lookTo}$), que indica la inclinación de la cabeza (figura 5.9).

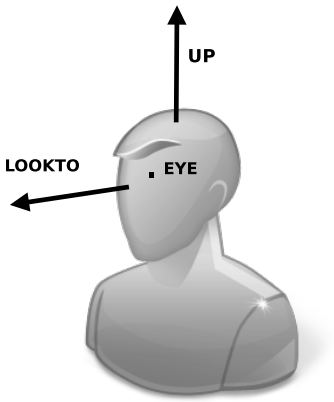

FIGURA 5.9: Posición y orientación de un usuario

Cada percepción tiene un peso asociado, que representa la calidad de la misma. El método de *tracking* que haya calculado la percepción es el que determina su peso, el cual será utilizado posteriormente en la etapa de filtrado. Un peso va desde 0 hasta 1. Una percepción con un peso de 0 será ignorada, y una percepción con peso 1 tendrá prioridad sobre otras, dependiendo del filtro utilizado.

```
1  typedef struct{
2    Point3D  __Eye;      ///< where the user is.
3    Vector3D __LookTo;   ///< where the user is looking to.
4    Vector3D __Up;       ///< up Vector
5    float    __Weight;   ///< The weight assigned to this perception
6    double   __clock;    ///< time of creation in milliseconds from program start
7  } tUserL;
```

LISTADO 5.7: Estructura `tUserL`

Un método de *tracking* óptico tiene asociado una lista de fuentes de vídeo, que serán procesadas por el mismo. En un principio se pensó en utilizar un `VideoCapture` como forma natural de almacenar las fuentes de vídeo, pero buscando la máxima versatilidad del sistema, se utiliza una lista de `VideoDevices`, ya que la primera opción limitaba el sistema a utilizar un sólo tipo de método de captura.

El programador del método de *tracking* será el encargado de preparar su constructor para recoger las fuentes de vídeo utilizadas. También, si así lo decide, podrá añadir fuentes durante la ejecución con los métodos `addVideoSource(VideoSource*)`. Para tal efecto, se recomienda incluir en la factoría de métodos (que es simplemente un *facade* que hace de *frontend* de la creación de estos) la forma explícita de crearlo, ya que cada uno de ellos requerirá de un tipo de fuente de datos diferente.

Los métodos de *tracking* se ejecutan de forma paralela en un hilo. Para ello, `TrackingMethod` implementa un método `Thread()`. Este método no tendrá que sobreescribirse como en las fuentes de vídeo, sino que para tal cometido se utiliza el método virtual `loopThread()` (ver figura 5.10). El método `Thread()` encapsula un bucle (llamadas continuas a `loopThread()`) y lo ejecuta dentro de una zona de exclusión mutua. `TrackingMethod` incluye la variable que se utilizará como *mutex* e implementa dos métodos privados para bloquearla o desbloquearla (`lock()` y `unlock()`). El hilo se crea y se pone en ejecución mediante la operación `ThreadMethod::startMethod()`.

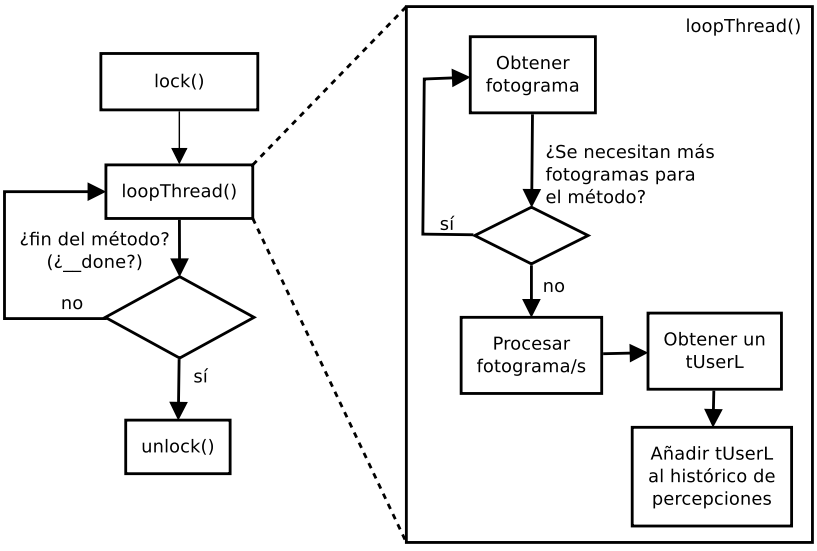

FIGURA 5.10: Ejecución del hilo asociado a un `TrackingMethod`

La razón para utilizar este diseño es aislar al programador de la necesidad de controlar la finalización de método de *tracking*. Es importante que la terminación se haga de manera transparente al usuario del *framework* puesto que al ser un entorno *multihilo* se evitarán

errores si se delega la inicialización y la finalización de los mismos y de su control al sistema de proceso. Así se evitarán errores de uso, y fundamentalmente, se evitará el acceso a recursos inexistentes en el momento de finalizar un método de *tracking*.

Para evitar que un método de *tracking* no pueda acceder a una fuente vídeo que se haya destruido, se sigue un orden estricto de destrucción de objetos en MARS. Este orden implica que los métodos de *tracking* que estén en ejecución siempre se destruirán antes que las fuentes de vídeo. Además, un método de *tracking* nunca se podrá destruir en mitad de una iteración de `Thread()`, ya que al empezar a destruir el objeto el hilo sigue en ejecución, y esto puede tener resultados no deseados. De este modo, se provee de una operación `TrackingMethod::endMethod()` que cambiará el valor de una variable privada que se utiliza como condición deparada del bucle de `Thread()` a verdadero. Con esto se consigue que termine la ejecución el hilo. Además, es importante que antes de dejar retornar a la operación, esperemos (`lock()`) a la finalización de la iteración que se esté llevando a cabo en ese instante. La destrucción del hilo se realiza en el destructor.

Cuando el programador sobreescriba el método `loopThread()`, él será el responsable de obtener los frames necesarios de una fuente de vídeo (con `getLastFrame()` o `getFrameAt(uint)`) y de procesarlos de manera acorde al método que esté implementando. Una vez que se hayan procesado los frames necesarios en esa iteración, el método de *tracking* tendrá que añadir los resultados, en forma de `tUserL` al histórico de percepciones, utilizando para ello su método heredado `addRecord(tUserL*)`. Estas acciones se repetirán hasta que se termine la aplicación o hasta que se termine de manera explícita este método de *tracking* (no recomendado, aunque factible). De este modo, en cada iteración se procesarán frames diferentes, que estarán siendo capturados en un hilo diferente.

MARS propone una jerarquía de métodos de *tracking* (figura 5.11) basada en si son visuales o no, y si son o no relativos. Aunque MARS incorporará en un futuro métodos inerciales, en el momento de escritura de este documento sólo incorpora métodos visuales, uno de ellos aun en desarrollo.

Uno de estos métodos (`TrackingMethodOAbsoluteARTK`) está basado en *ARToolkit* y determina la posición del usuario cuando la cámara visualiza alguna de las marcas preestablecidas. El método que está en desarrollo actualmente, se basa en el uso de puntos de interés y homografías. El autor ha proporcionado el esqueleto del código para los dos métodos, y participa en el desarrollo del último.

Uno de esos métodos es absoluto y otro es relativo. El método basado en marcas puede determinar la posición del usuario tan sólo con visualizar una de ellas. En el momento que alguno de los frames contenga una marca, el método de *tracking* , utilizando ARTOOLKIT

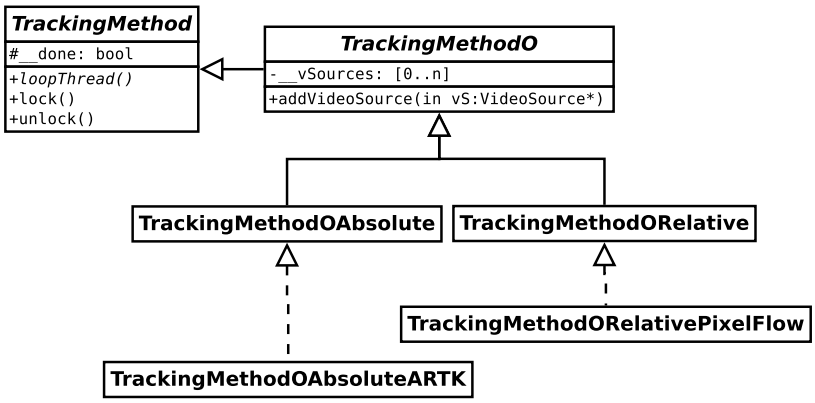

FIGURA 5.11: Jerarquía de métodos de *tracking*.

y mediante composición de matrices, podrá determinar la posición del usuario. Nótese que el sistema conoce a priori la posición y orientación de las marcas (mediante una matriz de 4 x 4 elementos).

5.2.2. Filtros

Un filtro está representado por la clase `TrackingFilter`. Esta clase contiene un atributo privado con su nombre (`std::string __name`), que debe identificar de manera unívoca al método, puesto que se utilizará para buscarlo en una tabla *hash* como se verá más adelante.

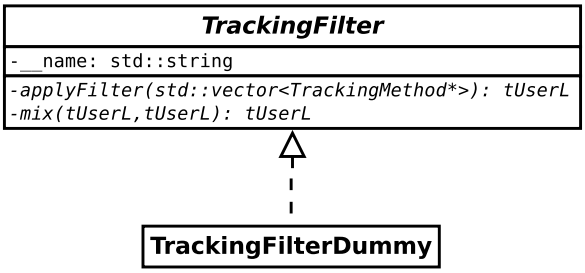

FIGURA 5.12: TrackingFilter

Esta clase contiene dos operaciones virtuales puras que deberán ser implementadas por cualquier filtro que se desarrolle. Estas operaciones son:

- `tUserL applyFilter(std::vector<TrackingMethod*>tMs)`. Esta operación acepta una lista de métodos de *tracking* que han de ser procesados por el filtro. El filtro deberá realizar las operaciones pertinentes sobre sus históricos de *percepciones* y devolver un `tUserL`.

- `tUserL mix(tUserL t1, tUserL t2)` - Esta operación determina cómo se mezclan dos `tUserL`. El programador del filtro es el responsable de decidirlo, en función de los parámetros que estime conveniente. Se recomienda que uno de ellos sea el peso asociado a cada una de las *percepciones*. Devolverá una nueva percepción que recoja la información fusionada.

Con MARS se incluye un filtro de ejemplo que no realiza ningún cálculo pero que se puede utilizar como esqueleto para la construcción de nuevos filtros, o para probar el funcionamiento del sistema sin hacer uso de ningún filtro. Este filtro se llama `TrackingFilterDummy` y la operación `applyFilter()` siempre devuelve la última percepción del primer método de *tracking* de la lista que se le pasa como parámetro. La operación `mix()` siempre devuelve la primera de las dos percepciones que se le pasa.

5.2.3. Control de métodos y uso de filtros

La puesta en marcha y la finalización de los métodos no depende de ellos mismos. La razón es poder controlar la puesta en marcha y la finalización de un hilo, para evitar conflictos con el constructor y el destructor de estos objetos. En las primeras aproximaciones el hilo se creaba en el constructor del padre, que se ejecuta antes del constructor del hijo. Esto hacía que se ejecutase el hilo de un objeto sin asegurar que estuviera completamente inicializado. De igual forma, la parada y destrucción del hilo se realizaba en el destructor (que es virtual), dándose las mismas circunstancias adversas que en el caso anterior.

La solución fue delegar el comienzo de la ejecución y la destrucción de los métodos de *tracking* en otro objeto: `TrackingController`. Esta clase, que es un controlador de métodos, se encarga de realizar las siguientes funciones:

- Puesta en marcha de los métodos de *tracking*
- Centralización y filtrado de las *percepciones*
- Actualización de los datos de la cámara virtual asociada
- Finalización y destrucción de los métodos *tracking*

Para llevar a cabo estas operaciones, un método ha de suscribirse al controlador a través de las operaciones `TrackingController::publishMethodRealative()` o `TrackingController::publishMethodAbsolute()`, que aceptan un `TrackingMethod*`

como parámetro. Se elegirá una u otra dependiendo de su tipo. Se ha optado por no sobrecargar una operación común para los dos puesto que la jerarquía se divide en métodos visuales y no visuales. Además así se permite utilizar los datos como absolutos o relativos sin cambiar el método de *tracking*, sólo la operación utilizada para *publicarlo.*

Cuando un método se suscribe al `TrackingController`, este se añade a una lista de métodos absolutos o relativos, dependiendo de la función utilizada, e inmediatamente se pone en ejecución. Las listas separan los métodos ya que los relativos acumulan el error total del sistema y puede resultar interesante procesarlos de forma independiente.

Para añadir filtros se utiliza la operación `addFilter()`, que acepta un `TrackingFilter`. La propiedad `__name` del filtro será utilizada como clave para almacenarlo en una tabla *hash* (`std::map`). Se puede elegir entre aplicar un filtro a todos los métodos, o discriminar entre relativos y absolutos. Por defecto no se discrimina. Con motivo de cambiar esta opción, se utiliza la operación `useRealtiveFilter()`, que determinará si se usa un filtro independiente para los métodos relativos, según el valor del booleano que acepta como parámetro.

Seleccionar el filtro general usado en el controlador supone utilizar `TrackingController:: selectFilter()`, que acepta una `string` como parámetro. Esta operación selecciona el filtro según el nombre con el que fue creado. Para seleccionar el filtro de los métodos relativos, se utiliza la operación `selectRelativeFiler()`. Si no se elige ningún filtro, será utilizado el de por defecto, que es `TrackingFilterDummy`.

El controlador ejecutará el filtro seleccionado, que hará uso de una lista de métodos de *tracking*; en particular de los históricos de percepciones que contienen. Ejecutará uno o dos filtros dependiendo de si se discriminan los tipos de métodos de *tracking* o no. Si se ejecutan dos filtros, los dos resultados serán mezclados con la operación `mix()` del filtro general seleccionado, obteniendo un único `tUserL`.

Esta salida se utilizará de manera directa para actualizar los parámetros de la cámara virtual asociada al controlador. Esta cámara está asociada al sistema de representación y desempeña el papel de relacionar la cámara real y la cámara virtual (ver apartado 5.4).

`TrackingController` es un *singleton*, que se destruye cuando finaliza la aplicación en la que se utiliza. En el destructor de este objeto se paran y destruyen todos los métodos de *tracking* publicados. Ya se justificó por qué se hace en este momento y no de manera autónoma.

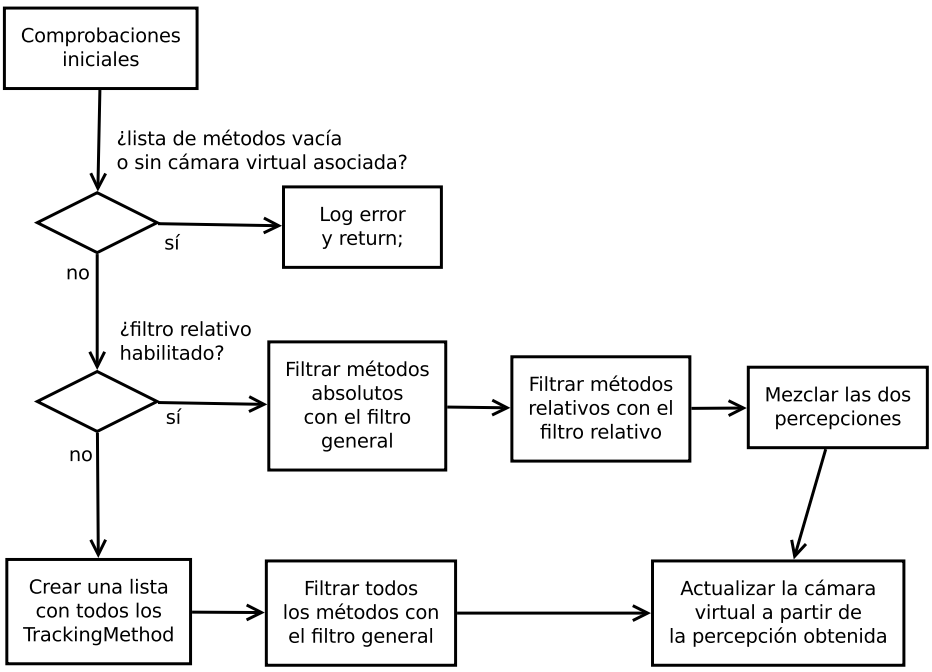

FIGURA 5.13: Diagrama de flujo de la operación de computa las percepciones

5.2.4. Creación de métodos

Con motivo de una posible inclusión de los métodos de *tracking* en la lista de elementos tratables por el lenguaje de *script* propuesto, se creó una clase *fachada*[2]. Esta clase se llama `TrackingMethodFactory`. Se eligió este nombre puesto que crea objetos de métodos de *tracking*. En el momento de la escritura de este documento, esta *fachada* da soporte a los dos métodos en los que se está trabajando para el proyecto ELCANO, que ya se han mencionado en un apartado anterior.

Esta clase es un *singleton*, y se puede utilizar para construir objetos de un determinado método de *tracking* dados un tipo (`enum trackingMethodType`), un nombre y una/s fuente/s de vídeo. También se puede utilizar para obtener por su nombre un objeto previamente creado. Para ello se utiliza internamente una tabla *hash*.

5.2.5. Patrones utilizados

Para la ejecución de los filtros por el controlador se ha utilizado el patrón *visitor*. Este patrón se utiliza para unificar las operaciones de filtrado. Las operaciones, que serán diferentes por cada filtro, tienen un interfaz común. Estas operaciones serán *visitadas* por el controlador cada vez que se computen y filtren los métodos de *tracking*.

[2]patrón *facade*

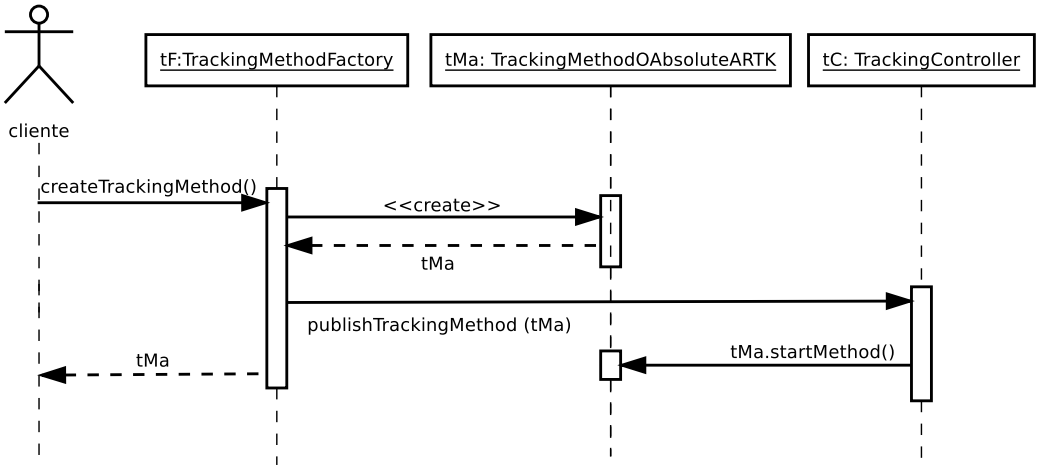

FIGURA 5.14: Diagrama de secuancia de la creación de un método de *tracking*

El patrón *singleton* también se puede encontrar en este subsistema. Las clases que sólo permiten una instancia son el controlador y la factoría de métodos, ya que sería redundante tener varios objetos de estos tipos a la vez en el sistema. Además dificultaría el ligado de las clases con un lenaguaje de *script* (ver apartado 5.6.6 y anexo A.6).

5.3. Subsistema de representación

El subsistema de representación es el más complejo de todos. Se ocupa de representar objetos sintéticos y de componerlos en 3-D según los datos obtenidos por los métodos de *tracking*. También se encarga de la carga y de la representación interna en memoria de los objetos, que en algunos casos habrán sido modelados y *texturizados* con una herramienta 3-D.

5.3.1. Configuración de SDL y OpenGL

La inicialización del subsistema de representación consiste en la creación de la ventana asociada al sistema y la configuración básica de OpenGL. De esto se encarga la clase `VideoOutputSDLOpenGL`, que contiene los métodos necesarios para llevarlo a cabo. SDL se encarga de la creación del contexto OpenGL, y de la creación y gestión de la ventana donde se representará el video real y los objetos sintéticos. Utilizar SDL facilita portar el sistema a otras arquitecturas o sistemas operativos, ya que SDL es altamente portable y evita al programador tener que conocer las particularidades de cada sistema operativo. Así,

SDL permite crear una ventana en GNU/Linux, en Windows y en MAC OSX sin tener que hacer uso de ninguna de las características específicas estos entornos.

La inicialización de SDL se realiza en `VideoOutputSDLOpenGL::SDLInit()`.

Para crear la ventana con SDL primero hay que inicializar internamente la parte relacionada con video de SDL, de lo que se encarga la función `SDL_Init()`, utilizando como parámetro `SDL_INIT_VIDEO`.

Una vez que SDL se ha configurado internamente, se debe configurar el modo de vídeo a utilizar. En el caso de MARS, una de las características indispensables del modo de vídeo es la capacidad para ser utilizado con OpenGL.

Otra característica deseable es utilizar un *buffer* doble para la representación. Mientras en un de los *buffers* se está realizando la síntesis de un frame, el otro está siendo mostrado. Cuando se termine de llevar a cabo la representación, los *buffers* se intercambiarán (con `SDL_Swap()`) y así lo harán sus funciones. Esta técnica permite ocultar la generación no inmediata de un frame. Si además se combina con una sincronización vertical [3], se conseguirá una animación sin fallos en la imagen.

Acorde con esto, se comienza la construcción de los *flags* de configuración SDL del siguiente modo:

```
int videoFlags  = SDL_OPENGL | SDL_GL_DOUBLEBUFFER | SDL_HWPALETTE;
```

El último parámetro (`SDL_HWPALETTE`) se refiere a la asociación de la paleta virtual con la paleta física del dispositivo hardware. Se debe utilizar esta opción puesto que la superficie SDL que se está creando se utilizará para escribir en la pantalla.

Posteriormente se consultará el dispositivo de vídeo para conocer sus características.

```
const SDL_VideoInfo *videoInfo = SDL_GetVideoInfo();
```

La estructura almacena información acerca de la posibilidad de utilizar aceleración 2-D y 3-D de vídeo, entre otras cosas. En el caso de que fuera posible, se añadirán dichos *flags*.

Finalmente se creará la superficie SDL, que se corresponderá directamente con la ventana, utilizando `SDL_SetVideoMode()`. Esta función devuelve un puntero a una superficie SDL, que se utilizará como lienzo para dibujar en esa ventana.

[3] Sincronización vertical se refiere a la acción de sincronizar el paso de un frame a otro, con el momento físico en el que los electrones terminan de recorrer verticalmente una pantalla CRT. Aunque la mayoría de las pantallas ya no utilizan esta tecnología, se sigue utilizando este término para referirse a los problemas relativos con la velocidad física de actualización de la pantalla. En las pantallas LCD, el tiempo de respuesta mide lo que tarda un *pixel* en cambiar de color

```
screen = SDL_SetVideoMode(screenWidth, screenHeigh, screenDepth,
                          videoFlags );
```

En el caso de MARS, las variables de anchura y altura de la ventana, y la variable de profundidad de color, se leen de un archivo de configuración.

Cabe destacar la posibilidad de utilizar MARS a pantalla completa, utilizando el *flag* `SDL_FULLSCREEN`. Para elegir esta característica se usa el archivo de configuración. Si no se utiliza el modo a pantalla completa, es posible cambiar el título de la ventana utilizando `VideoOutputSDLOpenGL::setWindowTitle()`, que acepta una cadena de la STL de C++ como entrada.

Tras inicializar SDL y la superficie donde se representarán los frames, se inicializa OpenGL utilizando `VideoOutputSDLOpenGL::OpenGLInit()`.

En este método se elige el tipo de sombreado, que en el caso de MARS será `GL_SMOOTH`. Este tipo de sombreado corresponde a uno por vértice y no por cara. Esto significa que las normales de los vértices determinaran cómo se sombrea el objeto.

También se elige el color con el que se borra un *buffer* antes de empezar a pintar de nuevo en él. En el caso de MARS, en negro, aunque como la imagen de fondo se refresca continuamente con el vídeo adquirido de una fuente, sólo se podrá apreciar su uso en el caso de que algún frame sea capturado de forma errónea.

Igualmente, se habilita la comprobación de profundidad, que hará que no se pinten píxeles que se encuentren detrás de otros, y se habilita la corrección de perspectiva más precisa que OpenGL permita. La primera opción acelera el rendimiento y la segunda mejora la fidelidad de la representación.

Una vez que se ha inicializado OpenGL se procede al ajuste de la vista. Este ajuste determinará la porción de espacio 3-D de OpenGL que se corresponde con la ventana y de la perspectiva con la que se proyectan los objetos del espacio 3-D en el plano de la misma.

La operación que realiza estos ajustes es `VideoOutputSDLOpenGL::resizeWindow()`. Tiene este nombre puesto que cada vez que cambia el tamaño de la ventana, será necesario llamarla para reajustar la representación a la misma. MARS sólo utiliza este sistema en la inicialización, por dos razones. La primera es que el interfaz de MARS es de tamaño fijo, la segunda es que la perspectiva dependerá de la fuente de vídeo utilizada (ver apartado 2.7.2).

`VideoOutputSDLOpenGL` también proporciona una operación para cambiar la matriz de proyección de OpenGL a una ortográfica que se ajuste a la ventana de MARS, *pixel* a *pixel*. Este cambio permitirá proyectar objetos 3-D de manera ortográfica, privándolos de perspectiva, una condición deseable a la hora de diseñar componentes con apariencia 2-D.

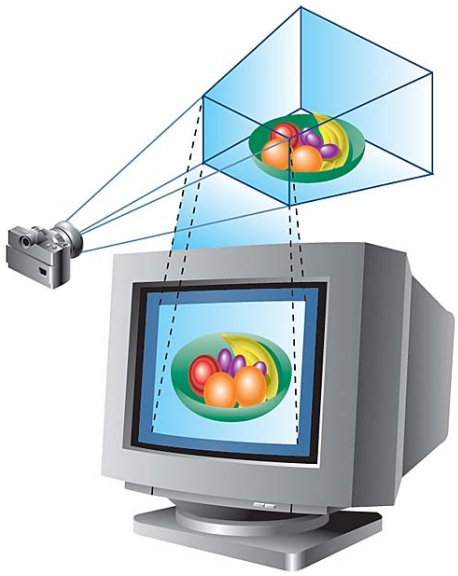

FIGURA 5.15: Correspondencia del espacio 3-D con la pantalla (viewport). (GAMEDEV.NET)

La operación `setOrtho()` guarda la matriz de proyección actual en una pila de matrices y configura la vista ortográfica en la ventana. Esta clase también proporciona la operación `setView()`, que permite recuperar la matriz de proyección anteriormente almacenada y reajustar de nuevo la vista.

5.3.2. El mundo 3-D

La clase que abstrae el concepto de mundo 3-D en MARS es `mars::World`. Esta clase se encarga de mantener una lista de objetos representables (`Renderables`) y una lista de *widgets* 2-D (`RenderableWidgets`). También se encarga de recoger los eventos de entrada y de repartirlos, de recoger el último frame de una fuente y pintarlo como fondo, de gestionar una lista de luces y de cámaras, de habilitar o deshabilitar la iluminación, de cambiar el tipo de representación y de gestionar la visualización de la consola de información.

Gestión de objetos *representables*

Para añadir un `Renderable` (5.3.6) se utiliza la operación `World::addRenderable()`, que acepta un puntero a uno de ellos como parámetro. Lo único que hace esta operación es añadir ese puntero a una lista de elementos *representables* (`std::vector<Renderable*>`). Paralelamente existe otro de punteros a `RenderableWidgets`. La razón de utilizar punteros es evitar la copia de los objetos. Si se utilizase una lista de objetos y no de punteros, se

tardaría más en añadir un elemento a la lista y existirían objetos duplicados. Además, en C++ se utiliza el constructor de copia cada vez que se añade un objeto a un contenedor[4] (excepto en la copia de los tipos básicos). Esto conlleva la posibilidad de que se produzca *truncado de objetos*[5]. El truncado se da cuando se copian objetos que contienen punteros de cualquier tipo. Un constructor de copia por defecto reproduciría el objeto y sus punteros, pero esos punteros estarían apuntando al mismo sitio que los punteros originales. Esto significa que si apuntan a objetos que se deberían haber copiado, esto no habrá sucedido. Para solucionar este problema se ha de implementar un constructor de copia a medida que copie también los datos apuntados. Si se optara por esta solución se estaría obligando al cliente de MARS (al programador), a programar un constructor de copia para cada nuevo tipo de objeto que quisiera incluir en el *framework*. Por este y por los otros dos motivos, en MARS casi siempre se utilizan *arrays* de punteros.

Para añadir un `RenderableWidget` al mundo se utiliza la operación `World::addWidget()`. Esta operación añade el *widget* a la lista de representables (ya que un `RenderableWidget` «es un» `Renderable`, como se verá más adelante) y lo añade también a una lista de *widgets*.

Gestión de luces

Las luces en OpenGL tienen una representación propia (apartado 5.3.3).`World::addLight()` acepta como parámetro una luz, que será añadida a un `std::vector` de luces y que será tras esto inicializada. Esta operación se asegura que no existan más de ocho luces en el mundo.

Cabe destacar que `World` es la clase encargada de actualizar las luces en OpenGL. Para realizar esta acción `World` contiene el método `updateLights()`, que actualiza OpenGL con la nueva información de las luces (ver listado 5.8).

```
for (unsigned i = 0; i < __lights.size(); i++){
    switch(i){
        case 0: light = GL_LIGHT0; break;
        case 1: light = GL_LIGHT1; break;
        case 2: light = GL_LIGHT2; break;
        case 3: light = GL_LIGHT3; break;
        case 4: light = GL_LIGHT4; break;
        case 5: light = GL_LIGHT5; break;
        case 6: light = GL_LIGHT6; break;
        case 7: light = GL_LIGHT7; break; } // end for
      glLightfv(light, GL_AMBIENT,   __lights.at(i)->getAmbient() );
      glLightfv(light, GL_DIFFUSE,   __lights.at(i)->getDiffuse() );
      glLightfv(light, GL_POSITION,  __lights.at(i)->getPosition() );
}
```

LISTADO 5.8: Actualización de las luces en `World::updateLigts()`

[4]Un contenedor en C++ es una clase especial *templatizada* cuya finalidad es contener elementos

[5]Conocido en inglés como *object slicing*

`World` contiene un atributo booleano que determina si la iluminación está habilitada o no. Se puede consultar a través de `World::isLightingEnabled()`. Se modifica a través de `enableLighting()` y `diableLigthing()`. Estás operaciones hacen uso de el estado de la variable `GL_LIGHTING` de OpenGL.

Gestión de cámaras virtuales

`World` contiene un `vector` de cámaras virtuales (`mars::Camera3D`, apartado 5.3.4) y un atributo que representa a la cámara seleccionada. De este modo, se pueden añadir varias cámaras, aunque sólo se hará uso de la seleccionada en ese momento.

Como `World` se encarga de gestionar las cámaras virtuales, es en esta clase donde se ha incorporado el mecanismo para ajustar la matriz de proyección de OpenGL según la cámara seleccionada. Así, cuando se selecciona una cámara según su posición en el `vector` usando la operación `World::selectCamera()`, se ejecutará de manera interna la operación `setUpprojection()` para cambiar la matriz de proyección actual de OpenGL por la de la cámara seleccionada. Esto se verá con más detalle en el apartado que trata sobre las cámaras virtuales (5.3.4).

Gestión del modo de representación

MARS ofrece la posibilidad de cambiar el modo de representación global del mundo virtual. Existen varios modos globales predefinidos. Estos modos corresponden a los modos de representación de los objetos `Renderable`s. El mundo cambiará el modo de cada uno dichos objetos, de acuerdo con el modo global elegido.

De este modo, `World` *visitará* los objetos del `vector` de `Renderable`s en el momento en el que se seleccione un nuevo modo.

```
1    vector<Renderable*>::iterator it = __renderables.begin();
2    while(it!=__renderables.end()){
3      (*it)->setRenderMode(0);
4      it++;
5    }
```

LISTADO 5.9: `World` visitando su lista de objetos `Renderable`

Esta funcionalidad se ha implementado ya que puede resultar interesante cambiar en tiempo de ejecución la forma de visualizar los objetos sintéticos. Por ejemplo, en una situación en la que no se distinga un objeto sintético oscuro sobre un frame de vídeo real oscuro, se puede activar el modo global de representación con trazado de líneas.

Gestión de eventos

`World` contiene una operación que procesa los eventos de entrada que genera SDL. Estos eventos, que proceden de una cola, se procesan mediante *polling*. Esto es, cada vez que se llama a la operación `eventsCall()` se procesan todos los eventos pendientes. Para ello, se realiza una consulta continuada al sistema de eventos de SDL y se procesa uno a uno hasta que ya no queden eventos por procesar. El esqueleto de dicha consulta es el siguiente:

```
while (SDL_PollEvent(&event)){ /* procesar event  */ }
```

La función `SDL_PollEvent()` devuelve falso cuando no quedan eventos en la cola. Cuando quedan eventos, selecciona el más antiguo y lo coloca en la estructura `SDL_Event` que se le pasa como parámetro. Este evento se procesará dentro del bloque *while*. Los eventos que procesa MARS son de tres tipos:

- EVENTOS DE PULSACIÓN DE TECLADO. Los eventos de teclado están conectados a acciones que se realizan sobre el mundo virtual como mostrar u ocultar la consola, cambiar el modo de representación global, o habilitar la iluminación; y también a acciones que se pueden realizar con los *widgets*.
- EVENTO DE MOVIMIENTO DEL RATÓN y EVENTOS DE «CLICK» DEL RATÓN. Estos eventos corresponden a las acciones del usuario del sistema sobre el ratón y se procesarán individualente por cada *widget*.

Los eventos de teclado referidos a mundo virtual son fijos, pero todos los demás dependen de los *widgets* que formen parte de la interfaz gráfica. Así, se visitará cada *widget* que contenga *World* a través de su operación `RenderableWidget::handleEvent()`, pasándole como parámetro el evento que se esté procesando (ver listado 5.10).

Cabe hacer una mención especial del evento de finalización de SDL y el de la pulsación de la tecla ESCAPE, que hacen que la operación `eventsCall()` devuelva falso. En los demás casos devuelve verdadero.

```
1 vector<RenderableWidget*>::iterator it;
2 for (it=__widgets.begin(); it!=__widgets.end(); ++it){
3   (*it)->handleEvent(event);
4 }
```

LISTADO 5.10: `World` visitando sus objetos `RenderableWidget` para que procesen el evento

Los detalles de cómo se procesa este evento se verán en el apartado 5.3.8.

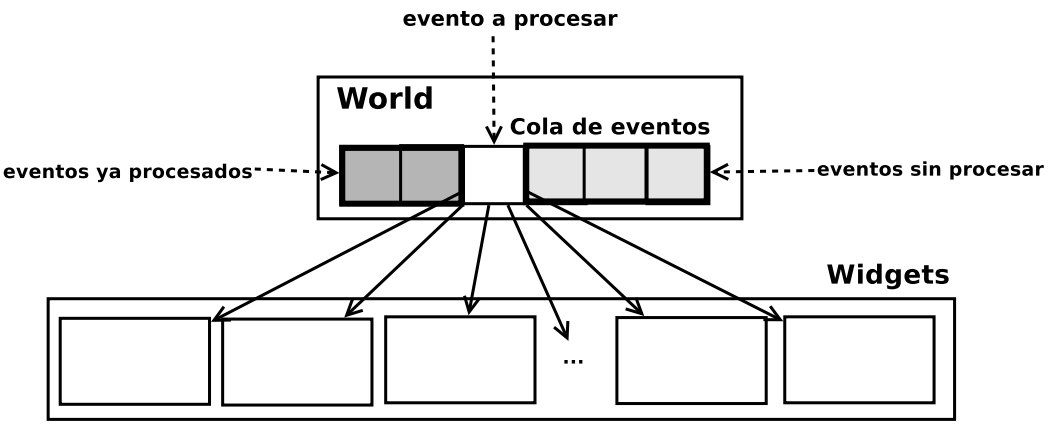

FIGURA 5.16: Reparto de eventos desde `World`

```
1    gluOrtho2D(0.0,__backgroundImage.cols,__backgroundImage.rows,0.0);
2    glMatrixMode(GL_MODELVIEW);
3    glPushMatrix();
4    glDisable(GL_DEPTH_TEST);
5    glDrawPixels(__backgroundImage.cols,__backgroundImage.rows,GL_BGR,  GL_UNSIGNED_BYTE,
         __backgroundImage.data);
```

LISTADO 5.11: Detalle de `World::drawBackground()`

Representación del vídeo de fondo

Ya que una de la características que definen la RA es la composición de vídeo real con objetos sintéticos, hay que proporcionar una manera de incluir frames en la representación. Para que `World` pueda dibujar los frames como fondo de la composición es necesario que conozca alguna fuente de vídeo.

Manteniendo la coherencia con el diseño del subsistema de captura, `World` contiene una fuente de vídeo en forma de `VideoSource`. Esta fuente de vídeo se añade con la operación `setBackgroundSource()`.

El método de `World` que se utiliza para dibujar el fondo es `drawBackground()`. Esta operación redimensionará el frame si así estuviera previsto, ajustará el *viewport* de OpenGL al tamaño de la misma preparando la matriz de proyección como ortográfica, deshabilitará la comprobación de profundidad, escribirá la imagen píxel a píxel en la superficie de representación y restaurará la matriz de proyección de OpenGL.

Los frames son matrices de OpenCV con un formato de imagen BGR8, esto es, un píxel está compuesto por tres bytes (24 bits), uno por cada componente (azul, verde y rojo). Nótese cómo este es el formato que se ha utilizado en la función `glDrawPixels()` (listado 5.11 - línea 4), que se encarga de pintar píxeles de forma directa en el *buffer* de vídeo de OpenGL.

Representación de la escena

Representar la escena completa supone llevar a cabo una serie de pasos ordenados:

1. **Cálculo de frames por segundo**. Esta operación, que realiza cada ciertos frames, se puede habilitar y deshabilitar. Para el cálculo se utilizan los llamados *ticks*[6] de SDL entre cada frame. Los *ticks* se miden en milisegundos. Si se calcula el tiempo que se tarda en dibujar un número de frames determinados (restando los *ticks* entre el primero y el segundo) podremos obtener este dato estadístico, que resultará útil para comprobar el rendimiento del sistema.

2. **Actualizar las luces**. Simplemente se invoca la operación textttupdateLights() de la que ya se ha hablado.

3. **Limpiar el fondo del frame**. El *buffer* de video actual[7] se pone en negro, eliminando su contenido anterior.

4. **Recuperación del frame de vídeo real**. Se recoge el último frame de la fuente de vídeo asociada.

5. **Dibujado de la imagen de vídeo real**. El frame recogido se dibuja como fondo (`World::drawBackground()`).

6. **Posicionamiento y orientación de la cámara seleccionada**. Se ejecuta la operación `use()` de la cámara seleccionada actualmente.

7. **Dibujado de todos los objetos sintéticos**. Se recorre la lista de representables visitando su método `draw()`.

8. **Intercambio de los *buffers* de OpenGL**. Se intercambian los *buffers* de vídeo, mostrándose la escena que se acaba de crear.

9. **Procesado de eventos**. Se llama a `eventsCall()` en último lugar para poder utilizar el valor que devuelve. Además, esto asegura que se realiza el *polling* de eventos en cada frame.

Todos estos pasos se llevan a cabo en el método `World::step()`, que representa el dibujado de un sólo frame compuesto. Este método habrá en un bucle, pero surge el problema de cómo determinar su fin. Para resolverlo, este método devuelve verdadero

[6] *Ticks* se refiere a tics de reloj.
[7] Recuérdese que se eligió una configuración de doble *buffer* para MARS.

mientras el usuario no elija terminar el programa , y devuelve falso en caso contrario (ha pulsado la tecla ESCAPE). Este valor booleano corresponde al mismo que devuelve `eventsCall()`.

El método `step()` se puede utilizar en una aplicación propia del siguiente modo:

```
while( world->step() ){ /* Rutinas propias del programador */ }
```

Dentro del bloque *while*, el programador podrá decidir qué rutinas ejecutar entre la representación de dos frames consecutivos.

5.3.3. Luces

MARS permite el uso de 8 luces, respetando la implementación de OpenGL utilizada en la realización del proyecto. Una luz está representada por la clase `Light`. Los atributos de la misma corresponden a los siguientes arrays de cuatro elementos:

- `GLfloat __ambient[4]` - Representa la componente ambiental de la luz. Los tres primeros elementos corresponden al rojo, al verde y al azul. La mezcla de estos determina el color. El cuarto elemento corresponde a la intensidad de esta componente.
- `GLfloat __diffuse[4]` y `GLfloat __specular[4]` - Igual con las componentes difusa y especular.
- `GLfloat __position[4]` - Representa la posición de la luz. Los tres primeras elementos corresponden con las coordenadas *x*, *y*, *z* del espacio tridimensional. El cuarto elemento tiene que ver con si la luz es puntual o dirigida.

Las operaciones que implementa esta clase proporcionan acceso a estos atributos para su lectura o modificación. `Light` es una clase muy ligera que podría haberse implementado como una estructura al estilo C, pero se decidió incluir operaciones para encapsular los datos que contiene.

5.3.4. Cámaras virtuales

Las cámaras virtuales están representadas por la clase `mars::Camera3D`. Esta clase contiene la misma información fundamental que un `tUserL`, esto es, un punto 3-D y dos

vectores que localizan y orientan a la cámara que representa. Esto no es casualidad, ya que una cámara virtual representa el punto de vista del usuario en el mundo virtual que será compuesto con la imagen real.

Las operaciones más importantes que implementa esta clase son las encargadas de establecer y devolver la matriz de proyección utilizada, y la de usar dicha cámara en una de las etapas de generación de la escena que se está representando.

El método que establece la matriz de proyección es utilizado por el `VideoSource` al que pertenezca esta `Camera3D`, si es que perteneciera a alguno. Éste es llamado tras el cálculo de la matriz de proyección a partir de los datos de calibración. De este modo, una cámara virtual pretende mimetizar en la medida de lo posible, las características de la fuente de vídeo externa.

La operación `Camera3D:use()` es la encargada de realizar las transformaciones en el sistema de representación de OpenGL para que los objetos que se representen a continuación parezca vistos desde la posición y orientación de la cámara. Para llevar a cabo dicha tarea se utiliza la función `gluLookAt()` de la biblioteca de utilidades de OpenGL.

5.3.5. Texturas

En MARS una textura está representada por la clase `mars::Texture`. Cuando se pensó en cómo abstraer el concepto de textura en el subsistema de representación, en un principio se evaluó la posibilidad de utilizar una lista de imágenes que se cargarían en lote al inicio de la ejecución. Aunque quizá no parezca una mala idea, esta opción no respetaría el paradigma de orientación a objetos, y mermaría la versatilidad que se necesita para la gestión de texturas.

Se pensó entonces en cómo encapsular una textura OpenGL dentro de una clase. Una textura de OpenGL está representada por un entero sin signo (un manejador[8]). Para llegar a obtener ese número hay que realizar los pasos necesarios para que OpenGL utilice un archivo de imagen como una textura.

De este modo, un objeto `Texture` contiene un `GLunit` (*unsigned de OpenGL*) que recogerá ese valor. Ya que una textura se carga desde un archivo, puede que se produzcan errores durante el proceso (el archivo no existe o no se puede abrir, por ejemplo). Para tal

[8]En inglés, *handler*

caso se utiliza un booleano, que tendrá un valor de falso cuando se haya producido un error en la carga. A este valor se accede desde la operación `isLoaded()`.

Un objeto `Texture` también tiene un nombre asociado, que corresponde al nombre del fichero con la imagen a cargar. Este nombre es útil para saber qué textura está contenida en un objeto de una forma rápida y legible.

Con todo esto, la clase se encarga de las siguientes tareas:

- **Cargar la imagen desde disco y transformarla en una textura de OpenGL**, usando la operación `loadFromFile()` a la que se le pasa una ruta del sistema de archivos. Esa ruta puede ser global al sistema de archivos o relativa al directorio de texturas que haya sido configurado.

  Para cargar la imagen se utiliza la función `IMG_Load()` que recibe como parámetro la ruta anterior y que devuelve una superficie SDL. Primero se utilizará la ruta como global y si no se encuentra el archivo se utilizará como local al directorio de texturas. Si se tiene éxito, lo primero que se comprueba es si las dimensiones de la imagen (ahora en forma de superficie SDL) son potencia de dos. La razón es que los dispositivos hardware que implementan soporte de texturas OpenGL rinden mucho mejor con texturas de este tipo. MARS soporta el uso de imágenes de otras dimensiones, pero avisa si se da esta situación durante la creación de una textura. Gracias a esto el desarrollador sabrá que puede optimizar esa imagen (y realmente debería hacerlo) para acelerar la ejecución. En algunos casos, la diferencia de rendimiento es de incluso superior a un orden de magnitud, puesto que algunos dispositivos hardware antiguos no soportan este tipo de texturas y han de ser gestionadas por software (por el driver o por la implementación OpenGL).

  El siguiente paso es determinar el formato de memoria de la imagen, que corresponderá con el formato de textura a utilizar. En MARS se da soporte a imágenes con cuatro tipo de representaciones en memoria: BGRA, BGR, RGBA y RGB. Es decir, se da soporte a imágenes planas de 24 bits y a imágenes con transparencia de 32 bits (los 8 bits extra se utilizan para el canal *alpha*). El número de bits por píxel lo determina el entero `BytesPerPixel` dentro de la estructura `format` de la superficie

de SDL. Para determinar el orden de las componentes de color se utiliza la propiedad `rmask` de la superficie, que representa la posición del byte del color rojo. Un valor de `0x000000FF` significa que el rojo está a la derecha (BG**R**).

Después se utiliza la función `glGenTextures()` de OpenGL que devuelve un `GLuint`. Este número representa un manejador de textura que esté libre. Este manejador se asocia a la creación de una textura 2-D nueva y se seleccionan algunos parámetros de calidad sobre la misma (el filtrado que sufre), tras lo que finalmente se procede a generar la textura OpenGL. Para esto se utiliza la función `glTexImage2D()` que recibirá como parámetros los datos relativos a la imagen (tamaño, profundidad de color, formato de textura,...) y un puntero a la zona de memoria donde está contenida (listado 5.12, línea 5).

```
1 glGenTextures(1, &__glTexture); // Generate Handler
2 glBindTexture( GL_TEXTURE_2D, __glTexture ); // bind it
3 glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR ); // Filter
4 glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR ); // Filter
5 // Create Texture
6 glTexImage2D( GL_TEXTURE_2D, 0, __sdlSurface->format->BytesPerPixel,
7 __sdlSurface->w, __sdlSurface->h, 0, __glTextureFormat,
8 GL_UNSIGNED_BYTE, __sdlSurface->pixels );
```

LISTADO 5.12: Detalle de la generación de texturas

- **Convertir una matriz de OpenCV en una textura OpenGL**. Para ello se utiliza `convertCVMatToTexture()`, que acepta la matriz y un puntero a donde se almacenará el manejador de la textura generada. Esta operación es una utilidad, y se ha implementado como un método estático. El proceso es casi idéntico al anterior, pero como se sabe que las imágenes contenidas en estas matrices tienen un formato BGR, se ahorran los pasos relativos a la identificación del mismo.

- **Convertir una superficie SDL en una textura OpenGL**. Igual que la anterior pero desde una superficie. La operación se llama `convertSurfaceToOpenGLTexture()`.

Una textura no implementa ningún mecanismo para evitar que se cargue un archivo ya cargado, o para evitar que dos modelos que comparten la misma textura la carguen varias veces. Por este motivo surgió la necesidad de gestionar de algún modo la texturas en MARS.

Gestión de texturas

No es deseable que el mismo objeto se encuentre duplicado en memoria sin que exista la necesidad y en el caso de las texturas no existe. Si dos objetos comparten la misma textura esta sólo debe cargarse una única vez. De esto modo se ahorrará espacio en memoria y se evitará el acceso a disco, que es varios órdenes de magnitud más lento que el acceso a memoria.

Para tal cometido se diseñó una clase llamada `TextureFactory`. Esta clase actúa como una *caché*. Contiene una tabla *hash* (un `map` de C++) cuya clave es el nombre del fichero que contiene la textura y cuyo valor es la textura (`mars::Texture`) resultante de su carga. Cuando el subsistema de representación necesite una textura, ya sea como un objeto `Texture` o como un manejador (`GLuint`) de OpenGL, utilizará esta clase para pedirla. Si la textura ya fue generada anteriormente se devolverá inmediatamente y si no se cargará desde disco.

Este diseño corresponde a un patrón *proxy* con dos aproximaciones distintas de acceso a una textura, basadas en el tipo de representación que se quiera obtener de la misma.

5.3.6. Objetos representables

Un objeto representable es, como indica su nombre, cualquier objeto que pueda ser representado en una escena. En MARS, un objeto representable es siempre un descendiente de la clase base `Renderable`. Esta clase contiene los atributos comunes a cualquier objeto de este tipo. Estos son:

- **La posición y la orientación**, que se pueden expresar de dos formas, una de ellas global y otra de ellas local. La global consiste en un punto 3-D y tres ángulos de rotación. Esta forma facilita la colocación y la rotación de los objetos en el espacio para un usuario inexperto en geometría y uso matrices. Si la posición global de un objeto no es visible por la cámara virtual seleccionada, este no se mostrará en la escena. Esta posición corresponde a la del objeto en el mundo real.

  La local sin embargo consiste en una matriz de 4x4 elementos `mars::Matrix16`, compatible con OpenGL. Cuando un objeto representable hace uso de esta matriz (`useMatrix(true)`) el objeto no se posiciona de manera global, esto es, esta matriz no se compone con las transformaciones de la cámara Esto significa que el valor de

esa matriz corresponderá a la matriz que será utilizada por OpenGL para dibujar en ese momento. Esto permite utilizar las matrices que devuelven algunas bibliotecas para la RA como ARTOLLKIT o como BAZAR. Estas bibliotecas devuelven la posición de las marcas como una matriz de 4x4 compatible con OpenGL que localiza a la misma en la vista actual. Así, se proporciona la facilidad de utilizar esta matriz directamente para poder dibujar un objeto sobre una marca con independencia de dónde se encuentre la cámara virtual. De esto modo, la posición del objeto representado es local a la posición de la fuente de vídeo real que visualiza la marca.

- **La forma de representarlo**, que podrá ser tal y como se modeló y texturizó, o mediante trazado de líneas (con o sin textura). Una clase que herede de `Renderable` debe determinar si puede dar soporte a las diferentes formas de representación. La forma de llevar a cabo la forma de representación elegida dependerá de cada clase en concreto. Para cambiar entre los diferentes modos se hace uso del método `Renderable::setRenderMode()`, que acepta como parámtro un entero sin signo que corresponde modo que se desee utilizar (0 - normal, 1 - *wireframe* sin *back-face culling*[9], 2 - *wireframe* con *back-face culling*).

- **La escala y la visibilidad**, que determinan el escalado del objeto (1.0 en todos los ejes por defecto) y si ese objeto debe ser mostrado o si está oculto actualmente. Un objeto se muestra u oculta con sus métodos `hide()` y `show()`.

`Renderable` proporciona las operaciones adecuadas para modificar y consultar estos atributos. Además, contiene un método virtual puro llamado `draw()` que tendrá que ser implementado por cada clase concreta de la jerarquía de representables. Este método será el encargado de dibujar el objeto en la escena dependiendo del tipo de objeto representable al que pertenezca. No es lo mismo representar un objeto desde un modelo poligonal que representar un texto desde una fuente *Truetype* o un plano con una película. Todos ellos son objetos representables, pero cada uno de una forma diferente.

Como ayuda a la implementación de `draw()`, `Renderable` proporciona una operación llamada `computePositionRotation()` (listado 5.13), que utiliza las funciones

[9]Técnica que hace que no representen los polígonos ocultos detrás de otros

de OpenGL para modificar la matriz *modelview* de OpenGL, según se utilice el posicionamiento global o local. Este método podrá ser llamado dentro de `draw()` antes de empezar a dibujar para configurar esta matriz con los datos correctos para el objeto representable a pintar.

```
1  void Renderable::computePositionRotation(){
2    if (__useMatrix){
3      glLoadIdentity();
4      glMultMatrixf(__internalMat.getData());
5    } else {
6      glTranslatef(__position.x, __position.y, __position.z);
7      glRotatef(__rotX,1.0f,0.0f,0.0f);
8      glRotatef(__rotY,0.0f,1.0f,0.0f);
9      glRotatef(__rotZ,0.0f,0.0f,1.0f);
10   }
11   glScalef(__scaleX, __scaleY, __scaleZ);
12 }
```

LISTADO 5.13: `Renderable::computePositionRotation()`

Gracias al polimorfismo, `World` recorrerá su `vector` de `Renderable`s asociado, visitando `draw()` sólo en el caso de que el objeto no esté oculto, como se muestra en el listado 5.14

```
1    vector<Renderable*>::iterator it;
2    it = __renderables.begin();
3    while(it!=__renderables.end()){
4      if (((Renderable*) *it)->isShown())
5      ((Renderable*) *it)->draw();
6      it++;
7    }
```

LISTADO 5.14: Fase de dibujado de los objetos representable desde `World`

El cliente de MARS deberá tener en cuenta que si desea implementar un nuevo `Renderable` tendrá que guardar (`glPushMatrix()`) y recuperar (`glPopMatrix()`) la matriz `MODELVIEW` de OpenGL antes y después de dibujar. El motivo es que antes de recorrer la lista de representables, el sistema ha configurado OpenGL para cambiar el sistema de referencia al adecuado según la posición y orientación de la cámara virtual.

MARS proporciona algunas implementaciones de objetos representables, que dan soporte directamente al proyecto ELCANO.

5.3.7. Objetos 3-D representables

En esta se sección de hablará de las implementaciones de `Renderable` que representan los objetos 3-D que se pueden sintetizar en una escena. En la figura 5.17 se muestra el diagrama de clases que conciernen a esta clase y su jerarquía asociada.

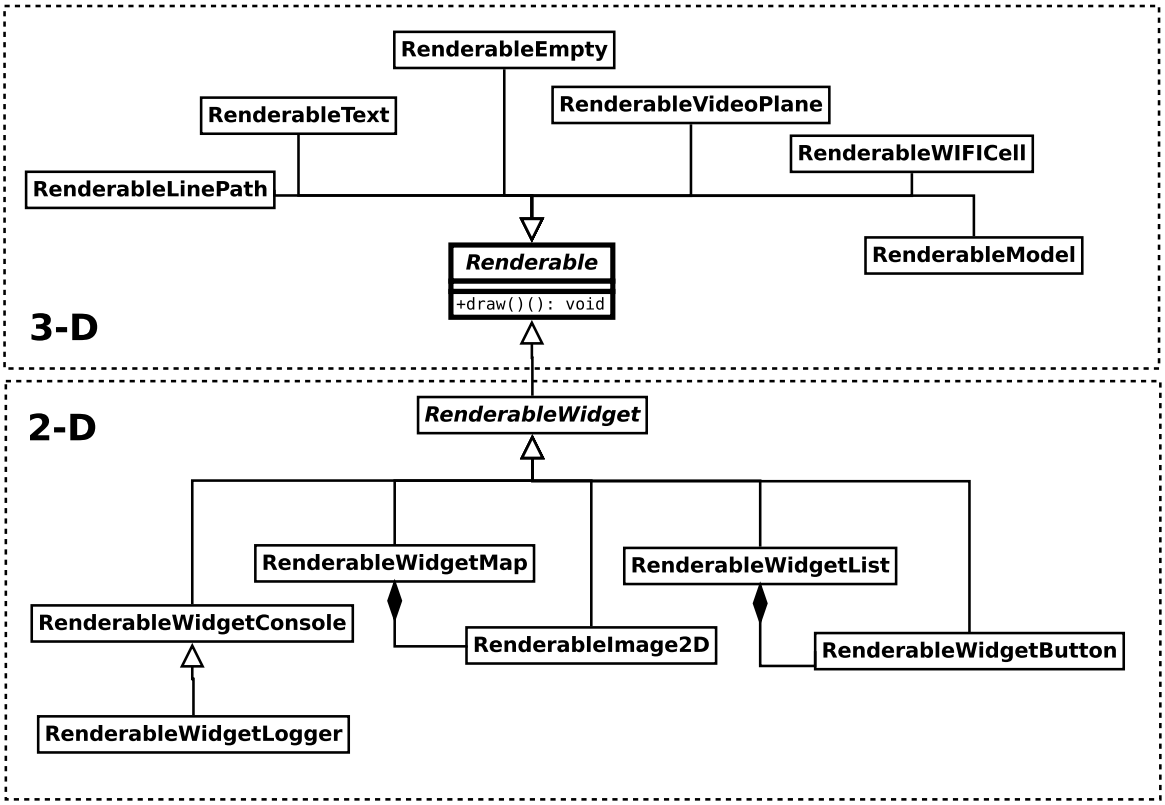

FIGURA 5.17: Jerarquía de `Renderables`

Modelos 3-D. `RenderableModel`

La implementación de `Renderable` que da soporte a la carga de modelos 3-D es `RenderableModel`. Cuando se diseñó esta clase lo primero que se hizo fue pensar en qué tipo de modelos podrían cargar. En un principio se dio sólo soporte a modelos basados en triángulos, puesto que son el tipo de modelos que se representan más rápidamente. Cuando ya se había realizado una pequeña implementación de la clase, surgió la necesidad de poder representar trazados de líneas de cuadriláteros. En una primera aproximación, tratando de cumplir con los requisitos sin modificar el diseño inicial, se realizó un algoritmo de

eliminado de aristas innecesarias que trasformaba la mallas de vértices en una mezcla de caras triangulares y rectangulares. Esto hacía que el diseñador de los modelos no controlase completamente cómo se representaban en MARS, así que se desechó este diseño y se comenzó con otro que diera soporte a la carga de modelos basados en cualquier tipo de polígonos.

Los tipos de archivos que puede cargar MARS son del tipo COLLADA (extensión *.dae*). Se eligió este formato por varias razones. La primera y más importante es que es un estándar *de facto* para OpenGL, puesto que es el grupo Khronos el que la propone. La segunda razón es que da soporte a distintas formas de representación de un modelo y entre algunas de ellas se encuentran la mallas triangulares y las basadas en polígonos de más de tres vértices. Otra de las razones es que los archivos COLLADA son archivos XML, que son legibles y editables por un ser humano, lo que facilita el depurado de las aplicaciones.

Como API para interactuar con este tipo de archivos se ha utilizado *ColladaDOM*, que es una biblioteca creada por SONY para el manejo de los mismos. Esta biblioteca, como sugiere su nombre, implementa el modelo DOM (*Document Object Model*) para los archivos *Collada*. Es libre, abierta y portable a diferentes arquitecturas.

A día de hoy *ColladaDOM* no está empaquetada en Debian y además contiene algunos fallos[10] que dificultan su uso y construcción, motivo por el cual el autor proporcionó dos archivos comprimidos (uno para *i386* y otro para *amd64*) con la bibliotecas compiladas y listas para su uso. De esto modo se evitó que el resto del equipo de desarrollo del proyecto ELCANO tuviera que descargar, configurar, parchear y construir el código fuente.

Para almacenar los datos contenidos en los archivos *.dae*, la clase `RenderableModel` posee los siguientes atributos:

- **Un lista de mallas**, que almacena todas las que contenga el archivo DAE. Estas mallas pueden ser de triángulos (`mars::TriangleMesh`) o de otro tipo de polígonos (`mars::PolyMesh`). Estas mallas se corresponden con las geometrías que contiene el fichero DAE.

[10] El autor ha notificado estos fallos al equipo de desarrollo de *ColladaDOM*, pero aun no han sido corregidos. Ver BUG *3017150* en el *Tracker* del proyecto: *http://sourceforge.net/projects/collada-dom/support*

- **Una lista de texturas** del tipo `mars::Texture`. Un modelo puede contener más de una textura y dos objetos pueden compartir la misma. Si no contiene ninguna, esta lista estará vacía. Cada textura tiene una imagen asociada, que está contenida en un fichero al que se accede con una ruta. Esta ruta está contenido en el archivo DAE que contiene el modelo.

- **Una lista de *call lists* de OpenGL**, que almacenará los listados OpenGL que se construyan durante la carga del modelo, y que se utilizarán para dibujarlo en el método `draw()`.

En la figura 5.18 se muestra un diagrama con el resumen de las agregaciones y dependencias que se dan en esta clase.

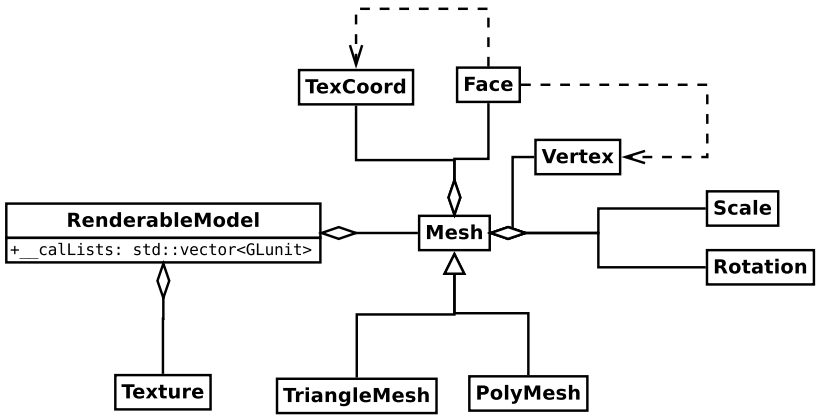

FIGURA 5.18: Esquema de un `RenderableModel`

`TriangleMesh` y `PolyMesh` son especializaciones de `mars::Mesh`, que representa una malla de cualquier tipo. Una malla está compuesta por una lista de caras y de vértices.

Atributo	Tipo	Qué representa
`__faces`	`std::vector<Faces*>`	La lista de caras
`__vertexes`	`std::vector<Vertex*>`	La lista de veŕtices
`__initialRotations`	`std::vector<Rotation*>`	La lista de rotaciones iniciales
`__scales`	`std::vector<Scale*>`	La lista inicial de escalados
`__initialTranslation`	`float[3]`	La translación inicial
`__hasTexCoords`	`boolean`	¿Contiene textura?

TABLA 5.1: Contenido de un objeto `mars::Mesh`

Los escalados iniciales (tabla 5.2) tienen que ver con el tamaño que se le ha dado a la malla al ser modelada. Este escalado se aplica de manera independiente a la posición de

los vértices (no es destructivo). De igual forma, las rotaciones iniciales (tabla 5.3) rotan la malla completa, pero no los vértices que la componen. Se ha dado soporte a estas dos características de COLLADA puesto que se pretende dar la máxima facilidad y libertad al creador de los modelos 3-D. Así, se pueden crear modelos compuestos por varias mallas, que tendrán desplazamientos y rotaciones diferentes y que se reflejarán en MARS.

Atributo	Tipo	Qué representa
x	`float`	Escalado en el eje X
y	`float`	Escalado en el eje Y
z	`float`	Escalado en el eje Z

TABLA 5.2: Contenido de `mars::Scale`

Atributo	Tipo	Qué representa
x	`float`	Componente X del eje de giro
y	`float`	Componente Y del eje de giro
z	`float`	Componente Z del eje de giro
r	`float`	Ángulo en grados que se rota

TABLA 5.3: Contenido de un objeto `mars::Rotation`.

La lista de vértices contiene los vértices sin duplicar de la malla. Una cara (tabla 5.4) está formada por tres o más vértices, y se referirá a estos según la posición que ocupen en el `vector` que los contiene. También contendrá la normal de esa cara y las coordenadas de mapeo de la textura para la misma.

Atributo	Tipo	Qué representa
normal	`float[3]`	La normal de la cara
vertex_count	`unsigned int`	El número de vértices que la forman
vertexes	`std::vector<int>`	Las posiciones de los vértices.
texcoords	`std::vector<TexCoord*>`	Las coordenadas de *UV-Mapping* de la cara.

TABLA 5.4: Contenido de un objeto `mars::Face`

Las coordenadas del mapeo están contenidas en una estructura `mars::TexCoord` (tabla 5.5). Estas coordenadas van de 0 a 1, correspondiendo 0.0 con el comienzo de la imagen y 1.0 la longitud máxima del ancho (s) o alto (t) de la misma. Puede que una malla no tenga ninguna textura asociada, con lo que tampoco contendrá coordenadas de texturizado.

Atributo	Tipo	Qué representa
s	`float`	Posición de la coordenada X de la textura.
t	`float`	Posición de la coordenada Y de la textura.

TABLA 5.5: Contenido de un objeto `mars::TexCoord`

Una malla de MARS contiene las operaciones necesarias para añadir nuevos vértices, caras, normales, coordenadas de texturizado, escalados y rotaciones, facilitando la creación de la estructuras internas desde la carga de modelos COLLADA.

`RederableModel` provee a MARS de la carga de un archivo COLLADA a través de `loadWithColladaDom()`, operación a la que se le pasa como parámetro la ruta del archivo a cargar. *ColladaDOM* proporciona una clase llamada `DAE` que apuntará inicialmente a la raíz de la jerarquía del XML y que se utilizará para su recorrido. La carga consiste en ir rellenando el objeto `RenderableModel` con los datos del modelo 3-D. Este proceso se resume en el diagrama de flujo de la figura 5.19.

La creación de un *call list* de OpenGL se realiza para acelerar el proceso de dibujado. Una vez que se crea uno, OpenGL lo mantiene en memoria y lo único que hay que hacer es ejecutarlo para pintar el objeto. Cuando se crea un *call list*, OpenGL genera un manejador para el mismo (al estilo del manejador de texturas pero con `glGenLists()`). Para dibujar se utilizará ese manejador y no habrá que invocar de nuevo cada una de las operaciones OpenGL que este contenga. `RenderableModel` proporciona el método interno `createCallLists()` que crea uno de estos listados por cada malla que contenga. Los manejadores creados se guardarán en una lista, que se recorrerá en el método `draw()`, ejecutándolo con `glCallList()`.

Para generar una *call list* se realizan los pasos que se llevarían a cabo para pintar el modelo directamente, pero contenidos entre las funciones de OpenGL `glNewList()` y `glEndList()` en vez de entre `glBegin()` y `glEnd()`. En la figura 5.20 se muestra un diagrama de flujo que resume el proceso.

Con todo esto, cada vez que se tenga que dibujar un objeto `RenderableModel`, su operación `draw()` realizará los pasos recogidos en la figura 5.21.

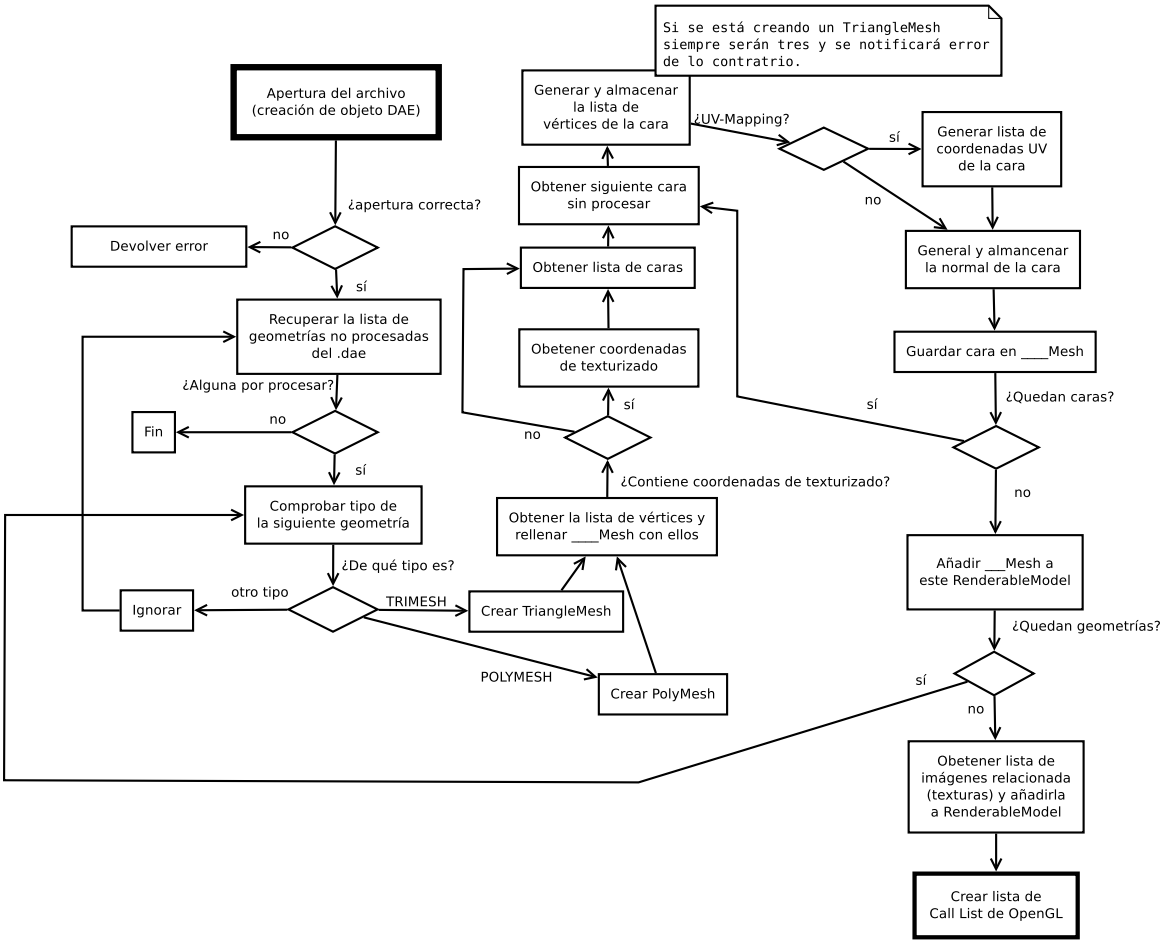

FIGURA 5.19: Diagrama de flujo de la carga de un modelo desde un archivo DAE

Texto en 3-D. `RenderableText` y `FontFactory`

La implementación concreta de un `Renderable` que representa a un texto 3-D es `RendrableText`. La necesidad de tener objetos de este tipo surge de la conveniencia de poder añadir etiquetas dentro de la escena compuesta. Por ejemplo, en el proyecto ELCANO es necesario poder nombrar las puertas de un edificio o una habitación concreta.

El diseño de esta clase se realizó con la idea de poder utilizar varios tipos de fuentes. Además, dependiendo del uso que se le diera al texto, es necesario que el mismo pueda ser plano o que pueda tener algo de profundidad. Para suplir estas necesidades se utiliza la biblioteca FTGL, que realiza la creación de texto a partir de una fuente *Truetype* y lo encapsula en un objeto. Este objeto contiene un método que se puede llamar para pintar la fuente en OpenGL. Sin embargo, esta biblioteca por sí sola con puede introducir texto en una escena. De esto se encarga `RenderableText`.

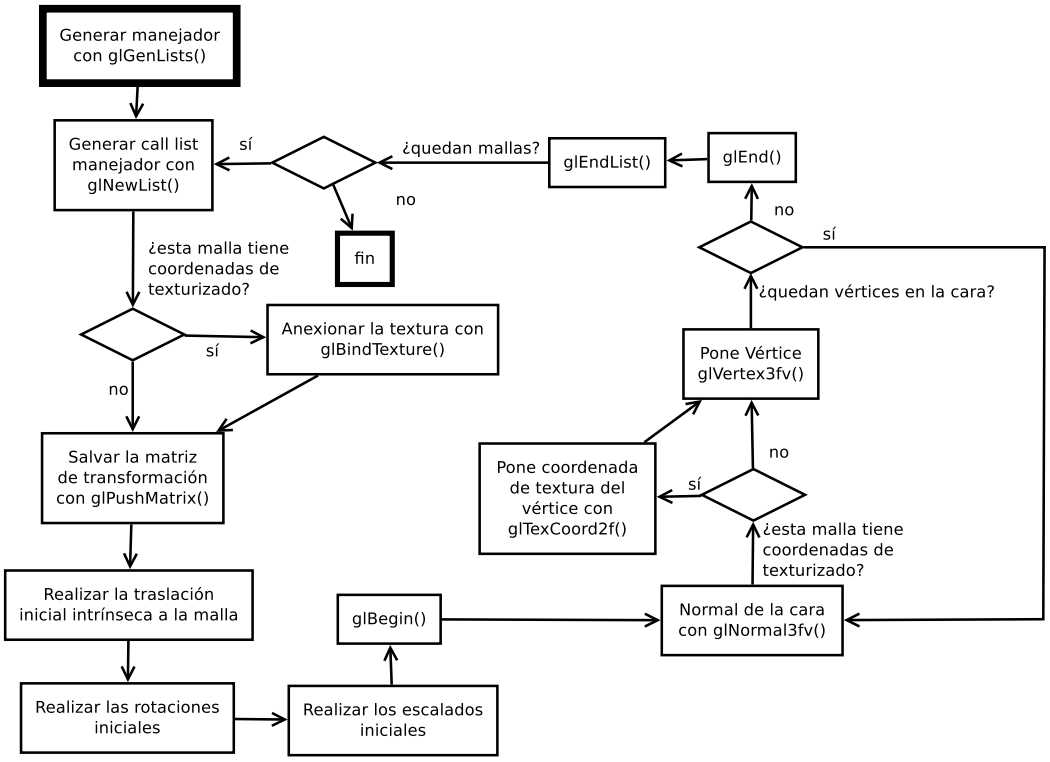

FIGURA 5.20: Generación de un *call list* para un `RenderableModel`

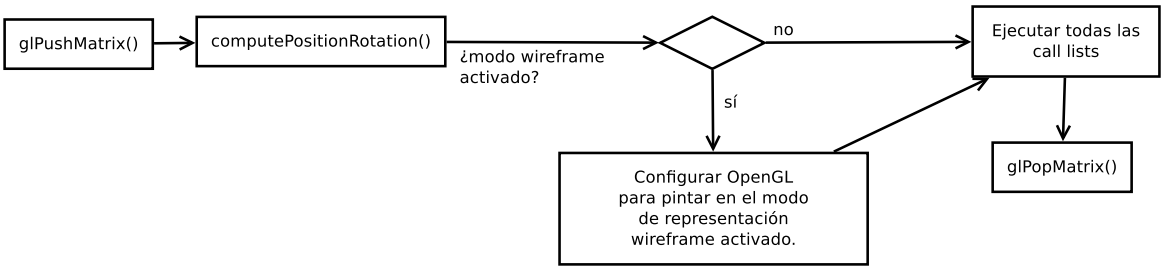

FIGURA 5.21: Diagrama de flujo de `RenderableModel::draw()`

Como interfaz para la creación de la fuentes, MARS proporciona la clase `FontFactory`. Su operación `addFont()` recibe como parámetros el nombre y el tipo de la fuente. El nombre corresponde al de archivo que la contiene (que se buscará en el directorio de las fuentes que haya sido configurado). El tipo de fuente corresponde a una de las siguientes opciones: *EXTRUDED*, *POLIGON* o *BUTTON*. La primera opción añadirá esa fuente con extrusión, la segunda como un polígono plano y la tercera la añadirá con la opción por defecto para los *widgets* de botón. Una fuente creada se recupera con la operación `getFont()` con su nombre como parámetro. Si la fuente no existe, se devuelve una por defecto de MARS. `FontFactoy` actúa como un patrón *Builder* construyendo la fuentes necesarias para el subsistema de representación y es además un *singleton*, puesto que la

factoría es única. Además, esta clase se encarga de destruir los objetos de fuentes (`FTFont`) que crea.

FIGURA 5.22: Relación entre RenderableText y FontFactory

`FontFactory` indexa las fuentes por su nombre en una tabla *hash* y contiene aparte otras tres fuentes especiales, que corresponden a la fuente por defecto del sistema, a la fuente por defecto de la consola y a la fuente por defecto de los botones. Esta clase implementa las operaciones necesarias para recuperar estas fuentes.

`RenderableText` hace uso de esta factoría en su constructor, al que se le tiene que pasar obligatoriamente la fuente a utilizar. Existe otro constructor al que se le pasa el nombre de la fuente, la cadena de texto que contiene y el tipo de fuente.

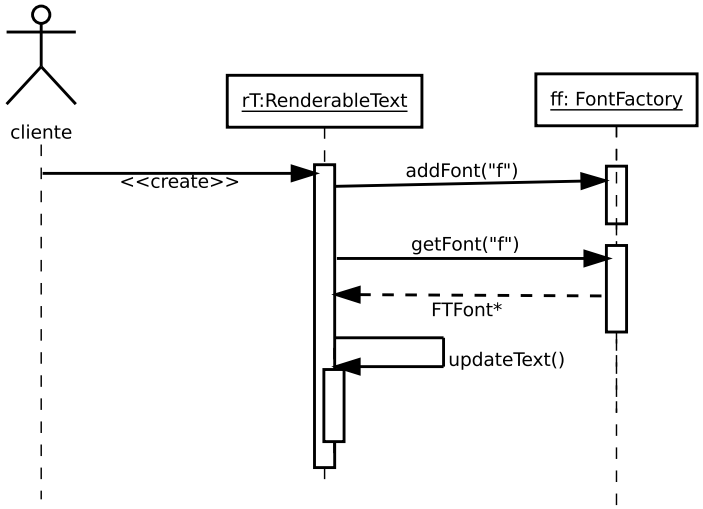

FIGURA 5.23: Constructor de `RenderableText`

Cada vez que se modifica la cadena de texto que se representa con estos objetos, se debe llamar al método `updateText()` que se encargar de actualizar los valores para la *bounding box*[11] que facilita el depurado. El dibujado de esta caja está deshabilitado por defecto, pero el programador puede hacer uso del mismo utilizando la operación `drawBox()`.

La cadena de texto se modifica con la operación sobrecargada `setText()` que acepta tanto una cadena de C (`char*`) como una `string` de C++. El tamaño se cambia con `setSize()` que acepta un entero sin signo, y el color con `setColor()`, que acepta tres parámetros, que corresponden a las tres componentes de color RGB.

[11] *Bounding Box* se traduce como *caja contenedora*, y representa la mínima caja 3-D que puede contener a ese texto.

```
1   if (__fontType == POLYGON){
2     glDisable(GL_LIGHTING);
3     glDisable(GL_TEXTURE_2D);
4     glEnable(GL_BLEND);
5   }

7   if (__fontType == BUTTON){
8     glEnable(GL_TEXTURE_2D);
9     glDisable(GL_DEPTH_TEST);
10    glEnable(GL_BLEND);
11    glNormal3f(0.0, 0.0, 1.0);
12  }

14  glEnable(GL_POLYGON_SMOOTH);

16  if (__wire){
17    glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
18  }

20  glPushMatrix();
21  glColor3fv(__color);
22  computePositionRotation();
23  glTranslatef(cx, cy, cz); // The text is drawn from its center.
24   __font->Render(__text.c_str()); // Draw
25  glPopMatrix();

27  if (__wire) glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);

29  if (__fontType == POLYGON)
30   if (World::getInstance()->isLightingEnabled())
31     glEnable(GL_LIGHTING);
```

LISTADO 5.15: Detalle de `Renderable::draw()`

En la operación `draw()` se discrimina entre texto extruído o de botón y texto poligonal. Cuando el texto que se pinta es de una fuente poligonal, la iluminación de OpenGL se deshabilita, al igual que el texturizado. Esto facilita la legibilidad y evita que alguna textura seleccionada con anterioridad en el proceso de pintado se asocie al texto. Si la fuente que se va a dibujar es de tipo *BUTTON*, entonces se activa el texturizado OpenGL, ya que este tipo de fuente puede estar basada en mapas de bits, que se pintarán como textura encima de un plano. Si se está utilizando la representación de trazado de líneas, se habilitará el modo de OpenGL correspondiente. Después se guardará la matriz de trasnformación actual de OpenGL, se pintará utilizando el método `Render()` de la fuente (línea 28, listado 5.15) y se restaurará la matriz.

En la línea 26 del listado 5.15 se realiza una operación de traslación. Esta operación se realiza para que la posición del texto se determine por su centro. Para ello, cada vez que se ejecuta la operación `updateText()`, se calcula la mitad del ancho, alto y largo de la *bounding box*, que corresponden con los valores que recibe `glTranslatef()`. Si no se realizase esta transformación, el texto se pintaría desde su esquina superior izquierda, lo que dificultaría enormemente su centrado en un punto específico del mundo virtual.

Vídeo en la escena. `RenderableVideoPlane`

Un de los requisitos fundamentales del subsistema de representación fue que se pudieran visualizar vídeos en un polígono rectangular dentro de la escena. Este polígono se comportaría como otro objeto representable, con la característica añadida de que su textura cambiaría para reproducir los frames de un archivo de vídeo. En un principio se intentó abordar este problema utilizando una fuente de vídeo de MARS y convirtiendo cada uno de sus frames en texturas. Recuerdesé que existe una operación para tal efecto. De hecho, así se hizo en una primera aproximación, que fue desechada de inmediato. Los motivos de su rechazo fueron dos: OpenCV no maneja el audio de los archivos de vídeo y el sistema de sincronización de los objetos `VideoSource` no es lo suficientemente potente para mantener un flujo de frames constante.

La solución fue utilizar la biblioteca *libVLC* para su implementación. Esta biblioteca proporciona la carga y el acceso al vídeo de un fichero. Para ello hace falta configurar algunas funciones que serán llamadas por el hilo que crea *libVLC* para reproducirlo. Esta funciones son tres:

- **`lock()`**. Una función que bloquee el acceso a la estructura que se utilice para almacenar el frame. El prototipo de esta función lo determina *VLC*:
  `void* lock(void* data, void** pixels)`
  data es una forma de encapsular el frame junto con el *mutex* y *pixels* habrá que apuntarlo a la zona de memoria que contenga la imagen del frame.

- **`unlock()`**. Una función que libere el acceso a esa estructura. Su prototipo es igual que el de `lock()`.

- **`display()`**. Una función que llama `libVLC` cuando se pueda dibujar el vídeo. MARS no lo utiliza, puesto que primero hay que convertir un frame en una textura. Aun así, el ajuste de la llamada de *callbacks* de *libVLC* requiere de esta función.

Para controlar todos estos detalles específicos de VLC en MARS existe en archivo `VLCPlayer.cpp` y su cabecera. Dentro de este archivo se encuentran la clase `OneMedia` y la clase `VLCPlayer`. `OneMedia` representa un archivo de vídeo reproducible. En la tabla 5.6 se muestra su contenido. Además de esto, `OneMedia` contiene una operación `update()` que para el vídeo si ha terminado y actualiza a falso el valor de `playing`.

`Ctx` (tabla 5.7) es una estructura que se utiliza para pasar sus datos entre las funciones de *callback* de VLC. Así, cada archivo que se reproduzca tendrá una superficie SDL y un *mutex* asociados, que serán para su uso exclusivo.

Atributo	Tipo	Qué representa
ctx	`mars:Ctx`	Una estructura Ctx
media	`libvlc_media_player_t*`	Un puntero a un manejador de VLC
width	`unsigned`	La anchura del vídeo
heigh	`unsigned`	La altura del vídeo.
playing	`bool`	¿Está en reproducción?

TABLA 5.6: Contenido de `mars::OneMedia`

Atributo	Tipo	Qué representa
suface	`SDL_Surface*`	Un puntero a una superficie SDL
mutex	`SDL_Mutex*`	Un puntero a un *mutex* de SDL

TABLA 5.7: Contenido de `mars::Ctx`

`VLCPlayer` se encarga de inicializar y configurar VLC. Con una sola instancia de VLC (`libvlc_instance_t`) es suficiente para reproducir los archivos que fueran necesarios, por eso esta clase sólo tiene un atributo de este tipo. De hecho, esta clase es un *singleton*, por el mismo motivo. VLC se inicializa en el constructor de la clase.

La clase contiene una tabla *hash* de vídeos, indexados por un nombre elegido por el cliente. Cuando se añade un archivo con `VLCPlayer::addMedia()`, que recibe una cadena con el nombre y otra con la ruta del archivo, se almacena una estructura `OneMedia` en esta tabla. Para recuperarla, se puede utilizar el método `VLCPlayer::getMedia()` con el nombre asociado como parámetro.

Añadir un nuevo archivo implica la realización de los siguientes pasos:

1. **Crear un reproductor de media VLC** asociado a ese archivo.
2. **Crear una superficie SDL** con la configuración adecuada para albergar un frame.
3. **Crear la variable de *mutex***.
4. **Configurar las llamadas de retorno (*callbacks*)** para *libVLC*. Estas llamadas se configuran con la siguiente función:

```
libvlc_video_set_callbacks(structMedia->media, lock, unlock, display, &(structMedia
    ->ctx));
```

   El último parámetro será pasado por VLC a las funciones `lock()`, `unlock()` y `display()` cuando sean llamadas por el mismo, y es la forma de intercambiar datos entre MARS y VLC.

5. **Configurar el formato de vídeo** de manera coherente con el utilizado para la superficie SDL creada en el paso 2.

Una vez que se tiene un medio preparado para su reproducción esta se comenzar con `VLCPLayer::playMedia()` y su nombre asociado como parámetro. De manera análoga, se puede parar un vídeo con el método `stopMedia()`.

Para destruir uno de estos medios se utiliza `destroyMedia()` con su nombre como parámetro.

Cabe entrar en detalle de cómo se produce la conversión desde un vídeo que reproduce la biblioteca por su cuenta y la textura que irá en el rectángulo de la escena. La clave está en la llamada interna que se produce a la función `lock()` (listado 5.16).

```
1  void* VLCPlayer::lock(void* data, void** pixels){

3    Ctx *ctx = static_cast<Ctx*>(data); // Cast

5    SDL_LockMutex(ctx->mutex);
6    SDL_LockSurface(ctx->surface);

8    *pixels = ctx->surface->pixels;

10   return NULL;
11 }
```

LISTADO 5.16: `VLCPlayer::lock()`

La función `lock()` recibe desde VLC un puntero (`void** pixels`). Este puntero indica a VLC en qué parte de la memoria ha de almacenar el frame que tiene listo para ser escrito. En la configuración de los *callbacks* se decidió pasar una estructura `Ctx` que contenía una superficie SDL. Lo único que hay que decirle a VLC es que escriba en la parte adecuada de esta superficie (línea 8). Así, cada vez que VLC escribe un frame, MARS tendrá acceso a el mismo a través de la superficie SDL asociada a un `OneMedia`.

Para reproducir sonido VLC utiliza ALSA (en el caso de tener configurado este sistema) o el dispositivo de audio configurado por defecto si estamos en un sistema operativo diferente a *GNU/Linux*. La reproducción de audio y su sincronización es transparente al usuario y la realiza VLC de manera interna.

Para representar en plano donde se visualiza el vídeo, MARS brinda la clase `RenderableVideoPlane`, que es una implementación concreta de `Renderable`. Esta clase está encargada de añadir el medio deseado a `VLCPlayer`, de convertir cada frame del mismo en una textura OpenGL y de dibujar el polígono rectangular con dicha textura en la escena. Es en su método `draw()` donde esta clase realiza esta funciones. En la figura 5.24 se puede ver un diagrama de flujo de la operación concreta para esta clase.

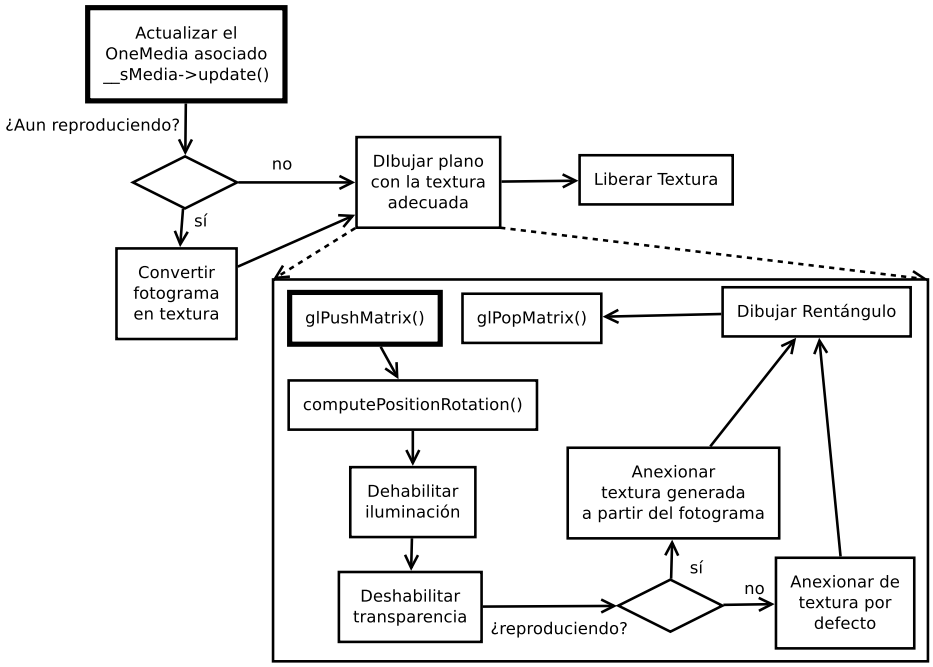

FIGURA 5.24: Diagrama de flujo de `RenderableVideoPlane::draw()`

Como este método se ejecuta cada vez que se genera una escena, la textura siempre contendrá el frame actual del vídeo.

`RenderableEmpty`

MARS proporciona una implementacion especial de `Renderable` para facilitar el depurado de las aplicaciones. Este objeto consiste en tres segmentos perpendiculares con tres colores diferentes. El centro del objeto corresponde al punto de intersección. El eje X está representado por el segmento rojo, el eje Y por el verde y el eje Z por el azul. La clase se llama `RenderableEmpty`.

Este objeto es útil para determinar la posición y la rotación de un renderable de forma rápida. Por ejemplo, si un programador que utiliza MARS añade un modelo 3-D a una escena y no se representa en la posición o en la orientación deseada, se recomienda sustituirlo por un objeto de este tipo para comprobar si el fallo está relacionado con el modelo (está rotado o tiene un desplazamiento inicial no deseado en el archivo DAE) o si correponde a una mala actualización de sus atributos.

`RenderableLinePath`

Con motivo de la representación de rutas entre varios puntos de interés, se creó la clase `RenderableLinePath`. Esta clase se encarga de dibujar esta ruta, que está compuesta de segmentos que unen varios puntos de control.

`RenderableLinePath` contiene un `vector` de puntos 3-D (`mars::Point3D`). Estos puntos se unirán entre sí para formar el camino.Para añadir un punto se utiliza el método `addPoint()` que acepta uno de estos puntos como parámetro. Se debe tener en cuenta que hacen falta un mínimo de dos puntos para poder dibujar un segmento. Si no, `RenderableLinePath()` no pintará nada en la escena.

Esta clase permite elegir el color del camino y su anchura. Para lo primero se proporciona la operación sobrecargada `setColor()`, que acepta tres o cuatro parámetros. Los tres primeros corresponden a las componentes roja, verde y azul del camino, la cuarta al valor *alpha* de transparencia. Para establecer la anchura se utiliza la operación `setWidth()`, que acepta un `float`.

En el proyecto ELCANO, los caminos hacia los puntos de interés se comunican como un conjunto de puntos en un instante determinado. Esto quiero decir que este representable ha de poder construir un camino a partir de nuevos puntos continuamente, desechando los anteriores. Para llevar esto a cabo se proporciona la operación `reset()`, que borra todos los puntos de una ruta.

`RenderableWIFICell`

Otro de los requisitos concretos para MARS en relación con el proyecto ELCANO, es proporcionar alguna forma de representar la localización de puntos WIFI en una escena. La clase `RenderableWIFICell` brinda al desarrollador un objeto representable en forma de cilindro para tal cometido. Aparte de la posición, se puede modificar la altura y el radio del mismo. La correspondencia entre intensidad del punto WIFI y el tamaño de este objeto se deja a elección del programador.

Para dibujar este cilindro se ha utilizado un *Quadratic* de la biblioteca de utilidades de OpenGL. Este *Quadratic* se crea en el constructor de la clase y se utiliza en el método `draw()` a través de la función `gluCylinder()` de OpenGL.

5.3.8. Componentes de la interfaz gráfica

MARS incluye un conjunto de objetos representables para ser utilizados en la construcción de una interfaz gráfica. Estos componentes están creados con primitivas de OpenGL y no pertenecen a ninguna biblioteca, están creados a partir de cero para suplir las necesidades de MARS.

Estos componentes son representables, por esa razón heredan de la clase `Renderable`, pero necesitan algo más de especialización. Cuando se diseñó esta jerarquía se pensó en qué atributos y métodos debería tener un *widget* que no estuvieran ya contemplados en un `Renderable`, para crear una nueva clase abstracta llamada `RenderableWidget`.

Un *widget* ha de tener una altura y una anchura, que corresponderán con los píxeles que ocupe en la superficie de representación. Los píxeles son indivisibles por lo que para representar estas dimensiones se utilizan enteros sin signo. Un *widget* puede estar o no seleccionado, un booleano se encarga de guardar esta información y se proporcionan las operaciones `select()` y `unselect()` para cambiar su valor.

Eventos

La diferencia más importante con un objeto representable común es que un *widget* debe poder recibir y procesar eventos. Por ejemplo, cuando un control de botón reciba un evento de movimiento de ratón deberá resaltar su texto (poner su estado en seleccionado) siempre y cuando la posición del puntero esté dentro de sus límites.

En el subapartado de *gestión de eventos* del apartado 5.3.2 (*mundo 3-D*), ya se mostró cómo se mandaban los eventos a los *widgets*. Cada vez que se obtenía un evento este se repartía a los *widgets* que estuvieran publicados utilizando un bucle *for*. Este evento SDL lo recibe como parámetro un `RenderableWidget` a través de su método `handleEvent()`. Este método discrimina el tipo de evento y dependiendo del mismo delega su proceso a diferentes métodos (listado 5.17)

```
1  void RenderableWidget::handleEvent(SDL_Event event){
2    switch (event.type){
3      case SDL_MOUSEMOTION:
4        handleMouseMotion(event.motion.x, event.motion.y); break;
5      case SDL_MOUSEBUTTONDOWN:
6        handleMouseClick(event.button.button, event.button.x, event.button.y); break;
7      case SDL_MOUSEBUTTONUP:
8        handleMouseRelease(event.button.button, event.button.x, event.button.y); break;
9      case SDL_KEYDOWN:
10       handleKey(event.key.keysym.sym); break;
11   }
12 }
```

LISTADO 5.17: `RenderableWidget::handleEvent()`

Así, cuando recibe un evento de movimiento del ratón (`SDL_MOUSEMOTION`), la operación encargada de procesarlo es **`handleMouseMotion()`**, que recibe la posición del puntero (`event.motion.x, event.motion.y`). Cuando recibe un evento de pulsado de un botón del ratón (`SDL_MOUSEBUTTONDOWN`) es **`handleMouseClick()`** la que lo procesa, que recibe el número de botón pulsado, y las coordenadas de la pantalla donde estaba el puntero del ratón en ese momento. Se actúa de la misma forma cuando se deja de pulsar ese botón (`SDL_MOUSEBUTTONUP`), aunque será **`handleMouseRelease()`** la operación que procese este evento. MARS utiliza estos dos últimos eventos para determinar cuándo se ha realizado un *click* con el ratón, que se produce cuando se suelta un botón que fue previamente pulsado en las coordenadas adecuadas.

Estas tres operaciones son abstractas, esto es, cualquier *widget* concreto que implemente a `RenderableWidget` deberá implementar estas tras operaciones. De este modo, cualquier especialización se ve obligada a responsabilizarse de cómo tratar estos eventos.

Cabe mencionar que MARS da soporte a ratones con hasta `MAX_BUTTON` botones. `RenderableWidget` contiende un *array* de booleanos con un tamaño que determina esa constante. Cuando un botón esté siendo pulsado, el elemento correspondiente de este vector será verdadero. Cuando no esté pulsado, será falso.

El cuarto tipo de evento SDL procesable es la pulsación de una tecla. Este evento se manda a la operación **`handleKey()`**, que recibe la tecla pulsada. Este método no es abstracto, es concreto y por lo tanto común para todos los *widgets*. El motivo es que la pulsación de una tecla no implica un procesamiento diferente dependiendo del tipo de *widget*. Por ejemplo, mientras que en un botón hay que determinar si el puntero del ratón está en su interior cuando se realiza un *click*, si a este botón se le asigna la tecla *a*, sólo habrá que esperar a que sea pulsada para activarlo, sin comprobar la posición del puntero.

Hasta ahora se ha visto cómo se discriminan los eventos para que sean procesados por un `RenderableWidget` concreto. Falta explicar cómo se asocia una acción a un evento determinado, es decir, cómo se asocia, por ejemplo, la pulsación de un botón con la reproducción de un sonido.

Los tipos de eventos propios de MARS son `CLICK`, `SELECT` y `KEY` (de tipo `wEventType`), que corresponden con la realización de un *click* de ratón sobre un *widget*, con la selección del *widget* y con la pulsación de una tecla. La consecución correcta de un tipo de evento de MARS se asocia con una lista de funciones a ejecutar. De este modo, existen dos listas de funciones, una para los eventos `CLICK` y otra para los eventos `SELECT`. Cuando se produzca algún evento `CLICK`, se llamará a todas las funciones que contenga su lista. Lo mismo sucede

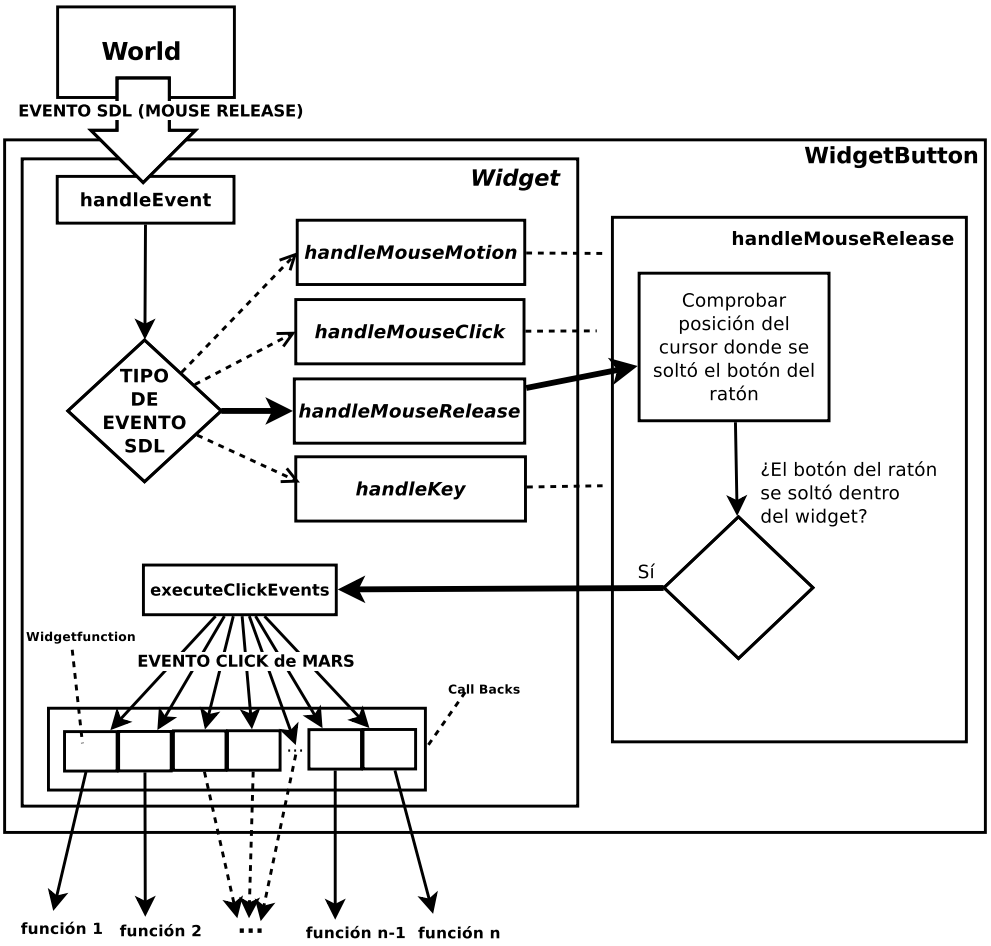

FIGURA 5.25: Un evento desde su propagación hasta la ejecución de las función asociadas. En este caso se trata de un evento SDL producido al soltar un botón del ratón, que termina disparando los eventos de *click*.

con los eventos `SELECT`. Para los eventos `KEY` se utiliza una tabla *hash* que utiliza como clave la tecla (`SDLKey`) que dispara su función asociada.

En el diseño se tuvo en cuenta la necesidad de utilizar funciones con parámetros. Como a priori no se conoce el tipo de parámetros que necesita recibir una función, MARS propone el siguiente prototipo de función: `void func(void* data)`. Cualquier función que requiera ser ejecutada por un *widget* en respuesta a un evento ha de ser de este tipo.

Para asociar un evento a una función determinada se realiza lo que MARS conoce como *publicación de un evento manejado*. Esto se realiza con el método `publishHandledEvent()` (listado 5.18). Este método recibe el tipo de evento, el puntero a la función a ejecutar, el parámetro de la función a ejecutar y en caso de que el evento fuera de tipo `KEY`, la tecla que dispara la ejecución de la función. El último parámetro toma

el valor `SDLK_LAST` por defecto, así, si se publica un evento de tecla y no se proporciona ninguna, MARS puede mostrar un aviso.

```
1  void RenderableWidget::publishHandledEvent(wEventType eType, void (*fptr) (void*), void*
       data, SDLKey k){

3    switch (eType) {
4      case CLICK:
5        __callBacksClick.push_back(WidgetFunction(fptr, data));
6        break;
7      case SELECT:
8        __callBacksSelect.push_back(WidgetFunction(fptr, data));
9        break;
10     case KEY:
11       if (k == SDLK_LAST) {
12         Logger::getInstance()->warning("RenderableWidget::"
13         "publishHandledEvent :: cannot publish a key event w/o"
14         " a key");
15         return;
16       }
17       __callBacksKey.insert( pair<SDLKey, WidgetFunction>(k, WidgetFunction(fptr, data)));
18       break;
19   }
20 }
```

LISTADO 5.18: `RenderableWidget::publishHandledEvent()`

Nótese en las líneas 5, 8 y 17 del listado 5.18 que los `vector` (listas) y el `map` (tabla *hash*) de funciones, lo que contienen es una estructura de tipo `mars::WidgetFunction` (tabla 5.8). El motivo es que los contenedores STL de C++ sólo pueden almacenar un tipo de objeto. Además se descartó inmediatamente el uso de varias listas (una de funciones y otra de parámetros) para evitar posibles inconsistencias.

Atributo	Tipo	Qué representa
f	`void (*) (void*)`	Puntero a una función que acepta un `void*` y no devuelve nada
data	`void*`	Datos que se le pasarán a la función

TABLA 5.8: Contenido de una estructura `mars::WidgetFunction`

Por último, con objeto de ejecutar todas las funciones publicadas, `RenderableWidget` proporciona las operaciones `executeClickEvents()` y `executeSelectEvents()`, que recorren la lista correspondiente a los eventos `CLICK` y `SELECT` ejecutando una a una cada función. La ejecución de un evento de tecla se realizará llamando a la función que devuelve la tabla *hash* al utilizarla como clave de la misma.

```
1  vector<WidgetFunction>::iterator it;

3  for (it=__callBacksSelect.begin(); it!=__callBacksSelect.end(); ++it){
4    void (*f)(void*) = (*it).f;
5    void* data = (*it).data;
6    f(data);
7  }
```

LISTADO 5.19: Ejecutando todas la funciones asociadas a un evento `SELECT` de un *widget*

En el listado 5.19 se muestra cómo se recorre el `vector` de funciones que se ejecutan en el evento `SELECT` de un*widget*. El proceso se realiza utilizando un *iterador* que contiene

la estructura `WidgetFunction` de la posición por la que se pasa de la lista. En la línea 4, se asocia la función contenida en esta estructura a una función temporal `f()`. En la 5 se asocian los datos que se pasarán como parámetro. En la 6 se realiza la llamada a la función.

Como ejemplo de uso de la publicación de funciones asociadas a eventos, si se quisiera asociar un evento `CLICK` de una instancia `wK` imaginaria de un *widget*, con la impresión del valor de su atributo público `n` (supóngase que es de tipo entero), se haría lo siguiente:

```
2  void imprime(void* data){
3    int* i = reinterpret_cast<int> data;

5    cout << ''Valor ='' << *i << endl;
6  }


9  wk->publishHandledEvent(CLICK, f, (void*) &(wK->n) );
```

LISTADO 5.20: Ejemplo de publicación de función asociada a un evento de *click* de ratón

En los siguientes subapartados se verán implementaciones concretas de `RenderableWidget` y cómo procesa los eventos que le llegan alguna de ellas.

Botones. `RenderableWidgetButton`

La implementación de `RenderableWidget` que representa a un botón dentro de MARS es `RenderableWidgetButton`.

Con motivo de poder crear interfaces gráficas atractivas a la vista del usuario, un botón utiliza una textura como fondo para su representación. Encima de dicha textura se dibujará el texto del botón. Este texto cambiará de color cuando el botón esté seleccionado. Para esta implementación de un botón, se entenderá que está seleccionado cuando el ratón pase por encima.

Con todo esto, un `RenderableWidgetButton` tendrá un objeto `Texture` asociado, y, ya que MARS da soporte al uso de texto, también tendrá asociado un objeto `RenderableText`. La textura se puede cambiar con `setTexture()`, con una ruta hacia un archivo de imagen como parámetro, aunque existe una textura por defecto para todos los botones del sistema. Esta textura se carga por el primer botón que la usa, a través de la factoría de texturas.

Como *widget* que es, ha de implementar las tres operaciones virtuales correspondientes. En su método `handleMouseMotion()`, esta clase determina si el cursor del ratón está dentro de los límites de este componente gráfico y ejecuta el método `select()` si así fuera. Los límites se calculan en el método `draw()`, donde también se utilizan y que corresponden a las componentes x e y de la pantalla donde comienza el botón y las componentes x e y donde termina.

En su método `handleMouseClick()`, primero comprueba que el botón pulsado no es mayor que el número de botones soportados y después comprueba que la pulsación se ha realizado dentro de los límites del *widget*. Si es así, activa ese botón como pulsado en el array de booleanos y reproduce un sonido de *click*.

En el `handleMouseRelease()` se comprueba si el botón que se ha dejado de pulsar es el izquierdo del ratón y si el cursor estaba dentro de los límites cuando esto sucedió. Si es así, se procederá a ejecutar las funciones asociadas al evento `CLICK`. El motivo de implementar así este evento es brindar al usuario la posibilidad de arrepentirse de ejecutar una acción. Si se presiona un botón, pero antes de soltar se arrastra el cursor fuera del *widget*, las funciones asociadas no serán ejecutadas.

Como `Renderable` que son, cualquier *widget* concreto ha de implementar el método `draw()`. En el caso de `RenderableWidgetButton`, el dibujado se divide en cuatro pasos: habilitar la proyección ortográfica en OpenGL, dibujar el fondo, dibujar el texto y restaurar la configuración de proyección previa.

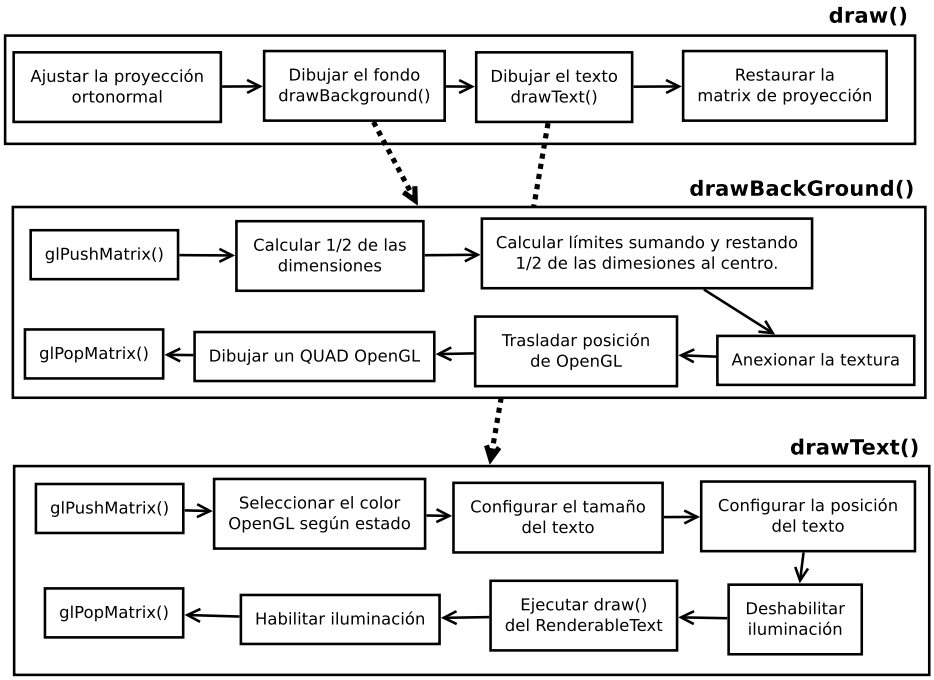

FIGURA 5.26: Diagrama de flujo de `RenderableWidgetButton::draw()`

Para dibujar el fondo se utiliza la operación privada `drawBackground()`, que calcula los límites del botón a partir de su centro y de sus dimensiones. Con esos datos dibuja un rectángulo aplicando la textura elegida.

El texto lo dibuja la operación `drawText()`, que selecciona el color adecuado según el estado del botón e invoca al método `draw()` del `RenderableText` asociado.

Lista de opciones. `RenderableWidgetList`

Una lista de opciones seleccionables corresponde a la clase `RenderableWidgetList`. Este componente gráfico permite al usuario seleccionar una opción pulsando en ella. En un principio este componente se diseñó para mantener una lista fija de opciones. Para dar soporte al proyecto ELCANO la lista debía poder albergar un número de opciones cambiante. La pantalla tiene un tamaño fijo y sólo es posible mostrar un número de opciones que dependerá del tamaño de estas. Por este motivo un *widget* lista debía restringir las opciones mostradas y proporcionar alguna forma de navegar por las ocultas. Se eligió el desplazamiento vertical como método para recorrer las opciones.

Aprovechando la infraestructura que ya proporcionaba MARS en el momento de diseñar este componente, se decidió utilizar el *widget* de botón existente para representar una opción. Así se delega en este objeto la responsabilidad de ejecutar las acciones asociadas a una opción, y se exime al componente de lista de dibujar cada una de las opciones desde cero.

La lista se puede desplazar verticalmente con la ruleta del ratón, pero no hay ruleta en una pantalla. Por este motivo el *widget* consta de un par de botones de desplazamiento. Uno hacia arriba y otro hacia abajo. De este modo un *widget* lista tiene una representación gráfica como la que se muestra en la figura 5.27.

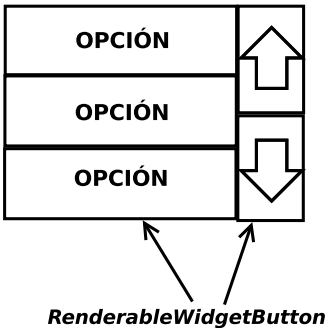

FIGURA 5.27: Aspecto visual de una lista desplazable (`RenderableWidgetList`)

Como los componentes que forman este *widget* se pueden representar a ellos mismos, `RenderableWidgetList` sólo se encargará de crearlos y asociar las opciones, y de gestionar su posición y visibilidad.

Para añadir una opción a la lista, primero se crea un botón con el texto de la misma, luego se le añade la acción que se quiera ejecutar con su selección. Esta acción se publica en el botón como respuesta a un evento `CLICK`. Se añade el botón a la lista (`vector`) interna de opciones (que es un `vector` de `RenderableWidgetButton`) y se añade a la lista de *widgets* del mundo virtual. Estas tarea las realiza la operación `addAction()` (listado 5.21),

que admite como parámetros el texto a utilizar en la opción y la función que se asocia a su selección, en forma de estructura `WidgetFunction`.

```
1  void RenderableWidgetList::addAction(std::string text, const
2                                       WidgetFunction& wFunc ){

4    RenderableWidgetButton* temp;
5    temp = new RenderableWidgetButton(__optionButtonWidth, __optionButtonHeight);
6    temp->useAutoScale(false);
7    temp->setText(text);
8    temp->setTexture("option-button.png");
9    temp->publishHandledEvent(CLICK, wFunc.f, wFunc.data );
10   __optionButtons.push_back(temp);

12   World::getInstance()->addWidget(temp);

14   __totalOptions = __optionButtons.size();

16   updateOptionButtons();
17 }
```

LISTADO 5.21: `RenderableWidgetList::addAction()`

Los tamaños de los botones de opción y de los de navegación se calculan en el constructor del *widget*, al que se le puede pasar como parámetro el porcentaje de anchura utilizada por los de navegación (0.2 por defecto).

La posición de los botones se calculará cada vez que se posicione el *widget* que los contiene. Por esto motivo fue necesario sobreescribir el método `setPostion()` de `Renderable`. Cuando se ejecuta este método, se calcula y actualiza la posición de los botones de navegación, y se llama a la operación interna `updateOptionButtons()`, que se encarga de actualizar los botones de opción.

La operación `updateOptionButtons()` recorre la lista de botones de opción, y ejecuta sobre ellos la operación `calculateVisibility()` que determinará los botones visibles y procederá a situarlos en la posición correcta y a mostrarlos. Los botones que determine como no visibles, los ocultará. Para calcular esto se utiliza la posición en la lista de la opción que se debe mostrar en primer lugar.

En el inicio se muestran siempre las primeras opciones, con un límite establecido en el constructor. Si hay cinco opciones y el número de visibles son tres, se mostrarán las tres primeras. Para mostrar las opciones siguientes habrá que pulsar el botón de *navegación abajo*, y para las anteriores, el botón de *navegación arriba*. Esto funciona gracias a que en el constructor (listado 5.22) se crean estos botones de navegación a los que se les asocia las funciones `goUp()` y `goDown()`, estáticas y que admiten un `RenderableWidgetList` como parámetro.

```
1  __upButton->publishHandledEvent(CLICK, RenderableWidgetList::goUp, (void*) this);
2  __downButton->publishHandledEvent(CLICK, RenderableWidgetList::goDown, (void*) this);
```

LISTADO 5.22: Detalle del constructor de `RenderableWidgetList`

A estas dos funciones se les pasa el `RenderableWidgetList` que las invoca, puesto que deberán actuar según su contenido. Además, estas funciones se encargarán de ejecutar las operaciones de actualización anteriormente explicadas. En el listado 5.23 se muestra el método `goUp()`.

```
1  void RenderableWidgetList::goUp(void* rWL){
2
3    RenderableWidgetList* myThis = reinterpret_cast<RenderableWidgetList*>(rWL);
4
5    // if there's no point of moving through the list ... return
6    if ( (myThis->__totalOptions <= myThis->__nVisibleOptions) || (myThis->__listPos == 0) )
7    return;
8
9    // __totalOptions > RWL_nOptions
10   if (  myThis->__listPos != 0  ){
11     myThis->__listPos--;
12     myThis->updateOptionButtons();
13   }
14 }
```

LISTADO 5.23: `RenderableWidgetList::goUp()`

Cuando se pulsa el botón *navegación arriba*, `goUp()` recibe el *widget* al que pertenece dicho botón y cambia el valor de la posición de la lista (le quita uno siempre y cuando se pueda), y ejecuta la función de actualización (línea 12). Cuando se pulsa el botón *navegación abajo* se ejecuta la función `goDown()` que funciona de manera similar, pero sumando uno a la posición actual.

Para que se pueda realizar este desplazamiento con la ruleta del ratón, se implementó la operación `handleMouseRelease()` de tal modo que ejecute las funciones `goUp()`/`goDown()` cuando se mueve la ruleta hacia arriba o hacia abajo (ver listado 5.24).

```
1  void RenderableWidgetList::handleMouseRelease(int button, int x, int y){
2    if (button >= MAX_BUTTONS) return;
3
4    if (__mouseButtonClicked[4]){
5      RenderableWidgetList::goUp(this);
6    }
7    if (__mouseButtonClicked[5]){
8      RenderableWidgetList::goDown(this);
9    }
10   __mouseButtonClicked[button] = false;
11 }
```

LISTADO 5.24: `RenderableWidgetList::handleMouseRelease()`

Consolas de texto. `RenderableWidgetConsole`

Este *widget* representa una lista de lineas de texto que sirven para mostrar mensajes de manera secuencial. Las líneas representan mensajes de notificación, de aviso o de error, cada uno de ellos con un color diferente.

Como abstracción de una línea de texto se utiliza la clase `mars::Line`, que contendrá el texto en sí y el color de la misma. Se utiliza su método `setColor()` para cambiar su color,

y `setText()` para cambiar su texto. Hay cuatro colores predeterminados: *RED*, *YELLOW*, *GREEN*, *WHITE*. Corresponden a valores RBG prefijados.

Las funciones de manejo de eventos de ratón están deshabilitadas, esto es, están implementadas pero están vacías. El motivo es que este componente no necesita procesar ningún evento de este tipo.

Para añadir líneas a este *widget* se usan sus operaciones `addNote()`, `addWarning()` y `addError()`, pasándoles una cadena de texto de C++. Estas funciones crearán los objetos `Line` con el color adecuado y los guardará en la lista interna de líneas.

El método `draw()` utiliza dos operaciones internas para dibujar el *widget*, una que dibuja el fondo (`drawBackground()`), similar a la de un botón pero sin utilizar una textura, y otra que dibuja las líneas visibles (`drawLines()`). La última operación calculará la posición de cada línea dependiendo de la posición del *widget* y de su orden en la lista.

Su funcionamiento es parecido a otros *widgets* y por eso no se entrará en más detalle.

Imágenes estáticas. `RenderableWidgetImage2D`

Con motivo de poder mostrar imágenes estáticas como logotipos o fotografías en la interfaz de usuario, se creó la clase `RenderableWidgetImage2D`.

Esta clase no es más que un rectángulo con una textura y funciona de manera similar a cómo lo hace el fondo de un botón. La característica que la diferencia es que posee un método especial para poder rotar ese rectángulo en 2-D.

Otra de las utilidades de este *widget* es poder formar parte de otros como un componente.

Mapas. `RenderableWidgetMap`

Para poder mostrar información global en la pantalla sobre la ubicación de un usuario se añadió el requisito funcional de poder utilizar un mapa. En dicho mapa debería mostrarse la posición y orientación de usuario. Además, se debería dar la posibilidad de representar los datos que devolvían diferentes métodos de *tracking*, para así visualizar los datos proporcionados por los mismos antes del filtrado.

Con estas premisas se diseñó e implementó la clase `RenderableWidgetMap`. Esta clase está compuesta de una imagen 2-D que contiene el mapa en sí, y por una lista de imágenes que representan los diferentes métodos visualizados. Se usa una lista de imágenes para que cada método tenga la suya propia y se puedan distinguir unas de otras encima de un mapa.

Por ejemplo, la posición computada del usuario se podría mostrar con una flecha negra, los valores que devuelve el método de *tracking* con marcas con una flecha roja y los valores que devuelve el método de localización por WIFI con un punto verde.

Un *widget* tiene unas dimensiones basadas en píxeles y por tanto habrá que realizar una conversión desde la longitud en el mundo real a su píxel correspondiente en la pantalla. Para realizar esta conversión, `RenderableWidgetMap` propone el uso de una escala. La escala corresponde a los píxeles por unidad real. No hay una unidad para la medida real, esto es, puede ser metros, centímetros u otra unidad de longitud. Si el mapa ocupa en pantalla 100 píxeles que representan 100 metros en el mundo real, la escala sería de 1.0, obteniendo una resolución de 1 píxel por metro.

Es muy probable que el origen que se toma en el mapa no coincida con el origen de la imagen del mapa. Por ejemplo, el origen está situado en el interior de un edificio pero el mapa recoge parte exteriores del mismo. Por este motivo se proporciona la posibilidad de configurar el punto de origen, que corresponde con el píxel del *widget* donde se sitúa el punto (0, 0) del mapa.

Haciendo uso de la escala (`setScale()`) y del punto de origen (`setStartPoint()`), se puede calcular (listado 5.25) la posición de un método en el *widget* conociendo su situación en el mundo real.

```
1 int __pixelX = int ( x  * __scale) + __realStartX;
2 int __pixelY = int ( y  * __scale) + __realStartY;
```

LISTADO 5.25: Conversión de unidades reales a píxeles en `RenderableWidgetMap`

En la figura 5.28 se muestra un diagrama de las acciones realizadas para cambiar la posición de un método dentro del mapa.

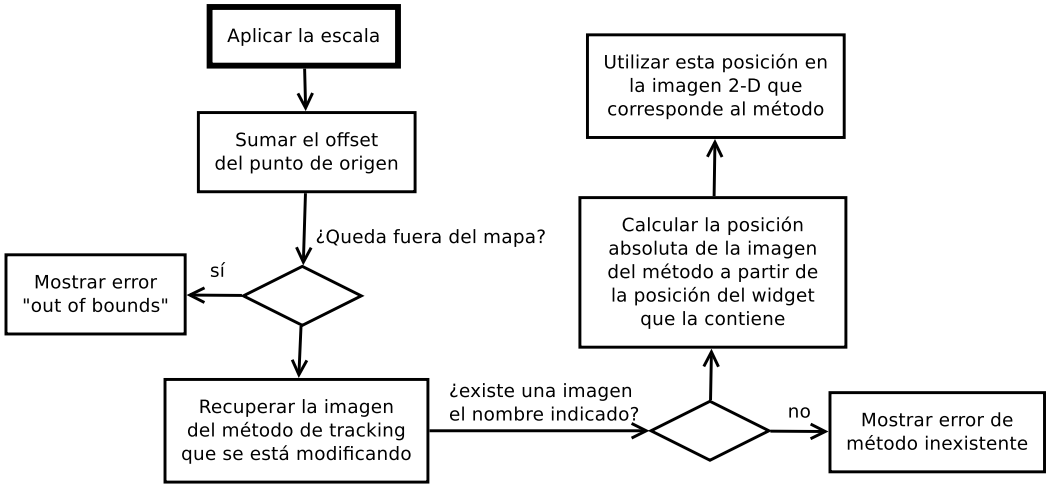

FIGURA 5.28: Diagrama de flujo de `RenderableWidgetMap::setMethodPosition()`

Para asociar un método con su imagen (que debería tener un tamaño de icono, aunque MARS no impone ninguna restricción) `RenderableWidgetMap` contiene una tabla *hash* con un nombre y un objeto `RenderableWidgetImage2D` asociado. Este hombre identifica unívocamente al método y no podrá estar duplicado. Para añadir un método de *tracking* a la representación del mapa se utiliza la operación `addMethod()` que acepta su nombre y un *widget* de imagen.

Un método de *tracking* tiene asociada una posición y una rotación. Para cambiar estos valores se utilizan las operaciones `setMethodPosition()` y `setMethodAngle()`. Las dos operaciones aceptan como parámetro el nombre que se le asignó (como cadena de C++). La primera operación también acepta dos números en punto flotante que representan las coordenadas reales que devuelve ese método de *tracking*. La segunda, acepta también otro número en punto flotante que determina la rotación en grados de la imagen. Esta rotación servirá para indicar la orientación del método.

`RenderableWidgetMap` sobreescribe el método `setPosition()` de su clase padre para poder cambiar la posición de la imagen de fondo de forma acorde. Además, se calcula el diferencia con la posición antigua, y se le suma a la posición de los métodos de *tracking* en el mapa, para que aparezcan en el lugar correcto.

Como es importante que la imagen de fondo se represente detrás de los métodos de *tracking* , esta clase se encarga de ejecutar el método `draw()` de sus componentes, sin publicarlos en el mundo virtual. Así en el método `draw()` de este *widget*, lo primero que se hace es ejecutar el método `draw()` de la imagen 2-D que corresponde al fondo (a la imagen que hace de mapa) y después se recorrerá la tabla *hash* de imágenes que corresponden a los métodos, ejecutando también su método `draw()`.

Los eventos de ratón se ignoran, así que las funciones que implementan su tratamiento están vacías.

5.3.9. Creación de representables

El proceso de creación de objetos representables directamente por el usuario de MARS suponía que el control de la memoria se confiaba completamente al mismo. La experiencia de las primeras iteraciones, en las que ya se utilizaba una versión temprana del sistema para elaborar algunas demos técnicas como hitos para ELCANO, determinó que debía ser MARS el que se encargase de gestionar la creación, la publicación y la liberación de los representables. Así, el usuario no tendrá que preocuparse por liberar recursos,

ni por publicarlos en el mundo virtual, puesto que para tal propósito se creó la clase `RenderableCreator`.

Esta clase contiene una tabla *hash* por cada uno de los tipos de `Renderable` que implementa MARS. La clave de dicha tabla es una cadena de C++, que identifica unívocamente ese objeto. Este nombre servirá para recuperar ese objeto. Así, un programador puede crear un objeto en un momento y recuperarlo posteriormente simplemente utilizando ese nombre, sin necesidad de almacenar o guardar los objetos creados en alguna lista o en variables especiales. Además, debido a que `RenderableCreator` es un *Singleton*, se puede hacer uso de la misma instancia desde cualquier parte del código, unificando el acceso a los objetos representables de la escena. Esto también facilita la creación de objetos con el lenguaje de *script*, eximiendo al usuario del mismo de la gestión. Esta clase se encarga de destruir los objetos que crea cuando se termina la ejecución.

Para evitar que el cliente cree de forma directa los objetos representables, todos sus constructores son privados, y todos declaran a la clase `RenderableCreator` como *clase amiga*. De este modo, la única forma de crear representables es a través de esta clase, que funciona como una fachada para la gestión de los mismos en el sistema.

En la figura 5.29 se puede ver un diagrama de secuencia con el ejemplo de la creación de un *widget* botón. Nótese como se añade el objeto al mundo, haciendo que sea incluido en la representación de la escena.

Las operaciones de creación y de recuperación están documentadas en el manual de referencia de MARS (ver anexo D).

5.3.10. Patrones utilizados

El patrón que más ha sido utilizado en este subsistema ha sido el *Visitor*. `World` visita a sus objetos representables delegando en ellos la forma de dibujarse. Lo hace igualmente con los *widgets* justo después de obtener algún evento.

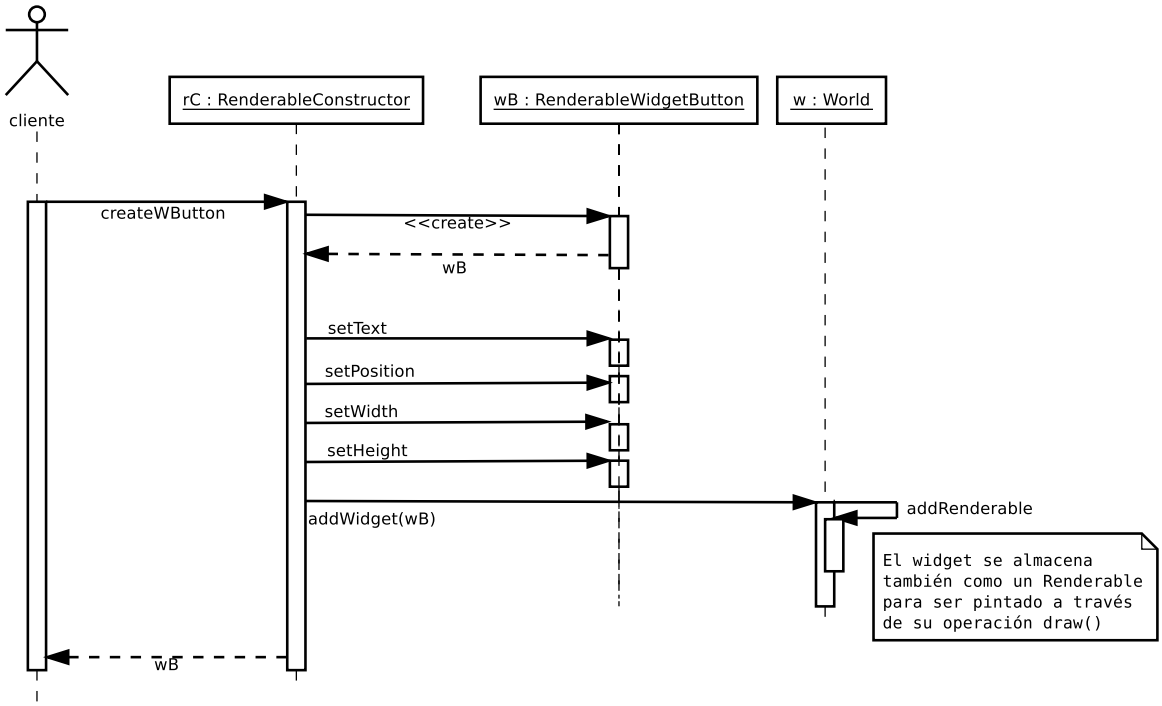

FIGURA 5.29: Creación de un widget botón utilizando `mars::RenderableCreator`

5.4. Los tres subsistemas como un todo

El funcionamiento en conjunto del sistema depende de las cámaras virtuales y de los datos que devuelven los métodos de *tracking*.

Una fuente de vídeo tiene una cámara virtual asociada y toma de ella alguno de sus datos a través del proceso de calibración (apartado 5.5). Estos datos son los valores de la matriz de proyección, que permiten que se alineen los objetos sintéticos con el vídeo del mundo real.

Los frames que captura una fuente de vídeo son procesados por un método de *tracking*. Un método de *tracking* contiene una o varias fuentes vídeo de las que toma frames que serán procesados. El resultado de procesar estas imágenes se filtra junto con los datos provenientes de otros métodos. El controlador que realiza este filtrado se encarga de modificar una cámara virtual asociada, que corresponde a la fuente de vídeo que se muestra al usuario. Aunque existan varias fuentes de vídeo en el sistema, sólo una de ellas se utiliza en la composición. Esta fuente corresponde con la visión del usuario del mundo real. Normalmente esta fuente es una cámara que porta el usuario, compartiendo así su misma localización y orientación. La cámara virtual asociada a esta fuente será modificada por el controlador de los métodos de *tracking*, permitiendo así que se use como cámara virtual principal en la representación del mundo.

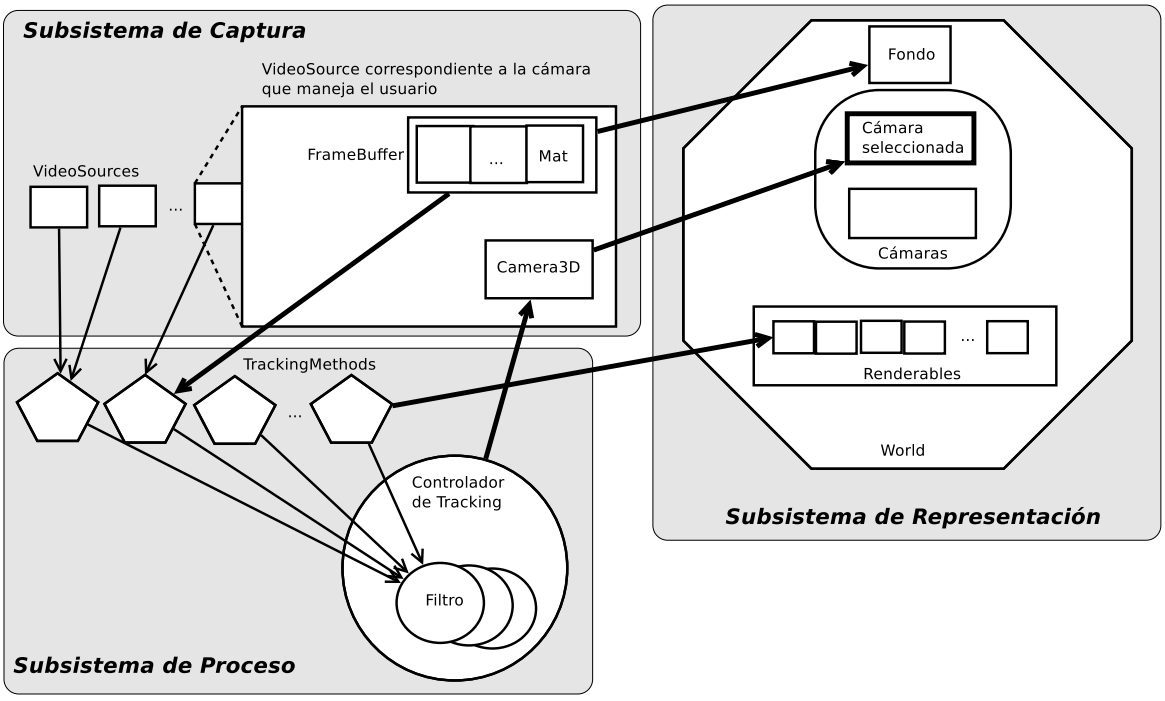

FIGURA 5.30: Relación entre los tres subsistemas

Si se revisa el proceso desde la representación, se podría decir que el mundo virtual utiliza una cámara. Esa cámara, también virtual, es una analogía de la cámara real, tomando de ella todos sus valores. Estos valores se toman de un controlador que devuelve una posición y una orientación. Estos datos se obtienen de filtrar otros que son proporcionados por métodos de *tracking*. Los métodos de *tracking* procesan imágenes para obtener esos datos. Las imágenes provienen de fuentes de vídeo. Esas fuentes de vídeo poseen una cámara virtual modelada para imitar el comportamiento la real. Esa cámara virtual, cuyos valores modifica el controlador de *tracking*, se corresponde a la seleccionada para la representación del mundo virtual.

5.5. Calibración y matriz de proyección asociada

En MARS, la calibración de las fuentes de vídeo tiene dos finalidades. La primera es la obtención de una matriz con los parámetros intrínsecos de la cámara y otra con los factores de distorsión asociados. La segunda es obtener la matriz de proyección que representa a esa cámara a partir los parámetros intrínsecos. Esa matriz de proyección es la que utilizará OpenGL para hacer creíble la composición del mundo real y los objetos sintéticos. Si no se realizase este proceso, los objetos no quedarían alineados, puesto que la perspectiva ofrecida por la cámara sería diferente a la utilizada para la representación del mundo virtual.

El objeto utilizado para estimar su pose es un tablero de ajedrez (véase la sección 2.7.3). Este es el objeto recomendado por OpenCV puesto que el contraste entre las casillas facilita su reconocimiento.

La calibración se realiza a través de la utilidad **`marsCalibrate`**. Esta aplicación se encarga de recoger frames de una fuente de vídeo y de procesarlos para obtener los datos concernientes. Actualmente puede calibrar fuentes de vídeo soportadas por OpenCV (tanto dispositivos físicos como basados en archivos) y dispositivos de la empresa *uEye* (a través del API que proporciona la misma).

`marsCalibrate` acepta una serie de parámetros. Para mostrar información acerca de éstos, basta con ejecutar el comando sin ninguno (figura 5.31)

Los parámetros son:

- `camera_name` - El nombre de la cámara / fuente de vídeo.
- `wBoard` - El número de esquinas que tiene el tablero a lo largo.
- `hBoard` - El número de esquinas a lo alto.

```
Usage: ./build/marsCalibrate camera_name wBoard hBoard squareSize nSnapShots <nDevice> <WUEYE>

       camera_name -  camera's name. It should be use to load the calibration
       wBoard      -  # of corners (width. )          (int greater than 2)
       hBoard      -  # of corners (height.)          (int greater than 2)
       squareSize  -  size of a square
       nSnapShots  -  # of snapshots                (int greater than 0)
       nDevice     -  # of device or VALID URL (optional - defaults to 0)
       WUEYE       -                            (if set, use UEYE driver)
```

FIGURA 5.31: Salida de la ejecución de `marsCalibrate` sin parámetros

- `squareSize` - El tamaño del lado de una casilla.
- `nSnapShots` . El número de capturas correctas que se tienen que realizar para proceder a la calibración.
- `nDevice` - El número de dispositivo hardware a utilizar o una URL de un archivo de vídeo válida. Por defecto, el valor es 0.
- `WUEYE` - Si se utiliza esta cadena como último parámetro se hará uso del API de *uEye* para utilizar una cámara de dicha empresa.

Un ejemplo de invocación de este comando para una cámara de nombre *test*, un tablero de 9×6 con casillas de 24 mm de lado, diez frames capturados y un dispositivo de vídeo con identificador *0*, es el siguiente:

```
marsCalibrate test 9 6 0.024 10 0
```

`marsCalibrate` muestra en pantalla la captura de la cámara. El usuario tendrá que elegir una serie de frames (pulsando la tecla *espacio* cuando se desee utilizar uno) que contengan diferentes poses del tablero de ajedrez. Cada uno de estos frames será procesado con la función de OpenCV `cv::findChessboardCorners()`, que devuelve una lista de puntos que representan la posición de las esquinas de las casillas del tablero. La posición de estos puntos se refina utilizando la función `cv::cornerSubPix()`. En el momento que se han capturado suficientes frames se procede a procesar los puntos encontrados. Esto se realiza con la función `cv::calibrateCamera()` a la que se le pasan los puntos junto con el número de esquina al que representan y que devuelve las matrices de parámetros intrínsecos de la fuente de vídeo y de distorsión.

Esas dos matrices se almacenan como un archivo XML con un nombre del tipo `<cam>-calib.xml`, donde `<cam>` debería ser el nombre elegido para esa fuente de vídeo en el sistema.

En el momento en el que se crea una fuente de vídeo en MARS, se buscará su archivo XML asociado y se cargarán esas dos matrices, que estarán a disposición de cualquier método de *tracking* que las requiera para sus cálculos. Además, a partir de la matriz de parámetros se calculará la matriz de proyección asociada. Para llevar a cabo esa tarea se utiliza la operación interna `VideoSource::calcProjectionMatrixFromCameraMatrix()`.

```
1  void VideoSource::calcProjectionMatrixFromCameraMatrix(){ //FIXME
       calcProjectionMatrixFromCameraMatrix
2    double fx = __cameraMatrix.at<double>(0,0);
3    double fy = __cameraMatrix.at<double>(1,1);
4    double cx = __cameraMatrix.at<double>(0,2);
5    double cy = __cameraMatrix.at<double>(1,2);

7    Matrix16 m;
8    GLfloat* data = m.getData();

10   float near  = 0.1f;   float far = 1000.0f;

12   float w = (float) (Configuration::getConfiguration()->openGLScreenWidth() );
13   float h = (float) (Configuration::getConfiguration()->openGLScreenHeight() );

15   data[0]  = 2.0*fx/w;
16   data[4]  = 0;
17   data[8]  = 0;
18   data[12]  = 0;

20   data[1]  = 0;
21   data[5]  = 2.0*fy/h;
22   data[9]  = 0;
23   data[13]  = 0;

25   data[2]  = 2.0*(cx/w)-1.0;
26   data[6]  = 2.0*(cy/h)-1.0;
27   data[10] = -(far+near)/(far-near);
28   data[14] = -1;

30   data[3] = 0;
31   data[7] = 0;
32   data[11] = -2.0*far*near/(far-near);
33   data[15] = 0;

35   __openGLCamera->setProjectionMatrix(m);
36 }
```

LISTADO 5.26: `VideoSource::calcProjectionMatrixFromCameraMatrix()`

Esta operación utiliza los parámetros de configuración de altura y anchura de la pantalla y los datos de la matriz intrínseca (c_x, c_y, f_x, f_y). El cálculo de la matriz de proyección se realiza de la forma que se sugiere en el *wiki*[12] de la biblioteca OpenCV, que corresponde con la estudiada en los antecedentes (sección 2.7.4).

La matriz obtenida, será utilizada en el sistema de representación, y será añadida al estado de OpenGL en el momento que se seleccione el uso de esa cámara virtual en `World`.

[12]http://opencv.willowgarage.com/wiki/Posit

5.6. Funcionalidad de soporte adicional

Aparte de las clases contenidas en los tres subsistemas y la aplicación de calibración, MARS proporciona algunas otras con la finalidad de ayudar a la creación de aplicaciones de RA. Estas clases son las correspondientes a los vectores, matrices y cuaterniones, a la configuración inicial del sistema, al sistema de registro de incidencias (*log*) y a la consola de depurado, y al sistema de sonido y de síntesis de voz.

5.6.1. Utilidades algebraicas

MARS provee de las tres estructuras matemáticas necesarias para posicionar y orientar objetos en 3-D, así como sus operaciones más importantes. Cabe destacar que se ha tenido especial cuidado con las comparaciones de igualdad de los números en punto flotante. De este modo, se ha programado una pequeña función de igualdad que acepta un error *épsilon*, que corresponde a la precisión que se puede perder manejando esta representación [Gol91].

Vectores

Las clase que MARS proporciona para representar un vector euclídeo es **`Vector3D`**. Además de contener las tres componentes que lo forman, la clase proporciona las operaciones más comunes como son la suma, la multiplicación por un escalar, el producto escalar y vectorial, la normalización y el cálculo de la longitud.

Matrices

Para representar una matriz de 4×4 MARS proporciona la clase **`Matrix16`**. Esta clase se construye a partir de una matriz de OpenCV o a partir de un *array* de dieciséis números en punto flotante.

Las operaciones que se pueden realizar con esta clase son la adición y el producto (por un escalar, por un vector o por otra matriz). Además se proporciona la capacidad de obtener la matriz inversa (`Matrix16::inv()`).

Utilizando esta operación, se proporciona otra más (`Matrix16::transform()`) para resolver la ecuación (5.1), esto es, para hallar la matriz que transforma **A** en **B**.

$$\mathbf{XA} = \mathbf{B} \tag{5.1}$$

Además se proporciona una operación para extraer los ángulos de *pitch*, *roll* y *head* de una matriz (`Matrix16::getEulerAngles()`).

Cuaterniones

La clase `Quaternion` brinda soporte para tratar este tipo matemático. Se proporcionan las operaciones de suma y producto, además de una para hallar el cuaternio inverso. También se incluye la operación `Quaternion::slerp()`, que acepta como entrada dos cuaternios y un número entre 0 y 1. Esta operación devuelve el cuaternión resultante de interporlar los otros dos.

5.6.2. Configuración

La configuración inicial del sistema se lleva a través del archivo descrito en la definición de `CONFIG_FILE`, que se encuentra en `Configuration.h`. En ese archivo se encuentra también la definición de la clase `Configuration` que proporciona el acceso a dicho archivo y que rellena una serie de atributos con los datos recogidos del mismo.

El archivo de configuración (*mars.cfg*) contiene una serie de líneas y en cada una de ellas cambia la configuración de una propiedad.

En la tabla 5.9 se muestran las variables de MARS que se pueden modificar a través del archivo de configuración.

Estas variables cuentan con un valor por defecto que se les asigna durante la inicialización de la clase (listado 5.27) y por tanto, la configuración de las mismas es opcional. Igualmente lo es la existencia del archivo *mars.cfg*.

`addStreamURL` es una variable especial, que añade a MARS una URL para la obtención de un vídeo remoto. Puede haber varias líneas con esta variable en el archivo de configuración, puesto que actúa como un comando que no modifica ningún valor, sino que lo añade.

Variable	Tipo	Qué representa
defaultImage	cadena de texto	Ruta a la imagen por defecto de un `VideoSource`
defaultVideoTexture	cadena de texto	Textura por defecto en un `RenderableVideoPlane`
defaultTexturePath	cadena de texto	Directorio de texturas
defaultVideoPath	cadena de texto	Directorio de vídeos
defaultFontPath	cadena de texto	Directorio de fuentes de texto
defaultSoundPath	cadena de texto	Directorio de sonidos
defaultFont	cadena de texto	Fuente por defecto de MARS
buttonsFont	cadena de texto	Fuente por defecto para un botón
consoleFont	cadena de texto	Fuente por defecto para una consola
defaultButtonTexture	cadena de texto	Textura por defecto de un botón
openGLFullScreen	`YES` / `NO`	Uso de MARS a pantalla completa (SI / NO)
openGLScreenWidth	entero	Tamaño horizontal de la ventana / Resolución en X
openGLScreenHeight	entero	Tamaño vertical de la ventana / Resolución en Y
openGLScreenDepth	entero	Profundidad de color
addStreamURL	cadena de texto	URL, nombre y descripción de un *stream* de vídeo

TABLA 5.9: Variables modificables a través del archivo de configuración

```
_defaultImage =  "./images/defaultImage.png";
_defaultVideoTexture =  "./textures/defaultVideoTexture.png";
_defaultTexturePath = "./textures/";
_defaultVideoPath = "./videos/";
_defaultFontPath = "./fonts/";
_defaultSoundPath = "./sounds/";
_defaultFont = "LiberationSans-Regular.ttf";
_buttonsFont = "Buttons.ttf";
_consoleFont = "Console.ttf";
_defaultButtonTexture = "ButtonTexture.png";
_openGLFullScreen = false;
_openGLScreenWidth = 640;
_openGLScreenHeight = 480;
_openGLScreenDepth = 32;
```

LISTADO 5.27: Inicialización de la variables configurables de MARS (`Configuration`)

En un principio se pensó en utilizar *flex* y *yacc* para construir un pequeño *parser* de una supuesta gramática para el archivo de configuración pero al ver el número de variables y la sencilla de la gramática necesaria, de descartó.

En la figura 5.32 se muestra contenido de ejemplo del archivo de configuración. Como se puede apreciar, los comentarios están soportados ya sea al principio de una línea o al final de la misma.

Para el proceso del archivo de configuración, `mars::Configuration` proporciona la operación `parseConfigFile()`, que *parsea* el archivo configuración, previamente abierto en el constructor de la clase. Para la implementación de ésta se utilizan las operaciones que proporcionan las cadenas de C++. Se procesa cada línea buscando el carácter "#", eliminándolo de la misma junto con el resto de la línea a su derecha.

```
# Creado por Sergio Perez el 21/04/2010
[defaultImagePath] := ./images  # Local al directorio actual.
[defaultTexturePath] := ./textures
[openGLFullScreen] := NO   # Usar YES para el deploy final
[openGLScreenWidth] := 640
[openGLScreenHeight] := 480
[openGLScreenDepth] := 32
[dafaultButtonTexture] := ButtonTexture.png
# [addStreamURL] := http://192.168.0.23/videoTest.mpg
```

FIGURA 5.32: Ejemplo de contenido del archivo de configuración

Lo que queda de cadena se procesa buscando una expresión del tipo `[VAR] := VALUE`. De ella se extrae el valor de `VAR` y el de `VALUE` que se pasan a la operación `addProperty()` que según la variable que sea, almacena una cadena o procede a su conversión a entero. En el caso de que `VAR` sea "addStreamURL", se crea una estructura nueva de tipo `urlStreamConfig` (tabla 5.10) y se rellenan sus valores con los obtenido de procesar `VAL` como una cadena con valores separados por comas.

Variable	Tipo	Qué representa
`name`	`std::string`	Nombre asociado al *stream*
`URL`	`std::string`	URL del *stream*
`description`	`std::string`	Descripción asociada al *stream*

TABLA 5.10: Contenido de la estructura `urlStreamConfig`

La clase `mars::Configuration` es un *Singleton*, accesible por todo el sistema. Las operaciones que proporcionan acceso a las variables sólo permiten la lectura de las mismas y jamás su modificación.

5.6.3. Enumeración de dispositivos

MARS provee de tres funciones de uso exclusivo en sistemas basados en el núcleo Linux. Estas tres funciones son `mars::getDevVideoNames()`, `mars::printDevices()`, y `mars::getDevVideoCaptureNames()`.

La primera devuelve un `vector` de cadenas con los nombres de los posibles dispositivos de vídeo. Esta función utiliza las funciones de listado de archivos del sistema operativo. La segunda función consulta los dispositivos e imprime su información asociada, utilizando *Video4Linux*. La tercera devuelve un `vector` de cadenas de C++. Cada cadena contiene la ruta de un archivo que corresponde a un dispositivo de vídeo que admite la captura de frames.

5.6.4. *Logs* del sistema y consola de depurado

MARS proporciona una clase llamada `Logger` para mantener un archivo de registro. El nombre de archivo de registro viene determinado por la definición de `LOG_FILENAME` que se encuentra en el archivo `Logger.h`.

La clase es un *Singleton* y su instancia es accesible por todo el sistema. En su constructor se abre el archivo en modo escritura, borrando su contenido anterior. También se inicializa el tiempo interno de este objeto, recogiendo el tiempo en milisegundos del sistema (con `getTimeofDay()`). Ese tiempo corresponderá al tiempo de registro inicial.

`Logger` proporciona tres operaciones para automatizar el registro. Esta operaciones son `note()`, `warning()` y `error()`, que aceptan una cadena como entrada. La única diferencia entre las operaciones es el color en el que imprimen el mensaje y el texto con el que preceden el mensaje (*NOTE*, *WARNING* o *ERROR*). En el momento de llamar a una de estas operaciones, se recoge el tiempo del sistema y se resta con el tiempo inicial. De este modo se consigue el tiempo en milisegundos entre el momento en el que se creó `Logger` y dicha llamada. De este modo se obtiene el momento preciso de la ejecución en el que se registra la información. Ese tiempo, junto con el tipo de notificación y la cadena usada como parámetro, se guardan en una línea del archivo *mars.log*.

Los errores siempre se muestran por la salida estándar, en rojo. Sin embargo, los mensaje de aviso y las notas sólo se muestran en construcciones de MARS con información de depurado. En cualquier caso, siempre quedan guardadas en el archivo de *log*.

Uno de los requisitos para MARS era poder utilizar una consola visual de texto como un sistema de *logs* visual. Fue un requisito de las últimas iteraciones, cuando el sistema ya tenía un diseño sólido. `Logger` es utilizado por todo el sistema, y por consiguiente, su cabecera se incluye en cada una de las clases que requiere hacer *log* de algún tipo. La clase `RenderableWidget` y sus derivadas no son una excepción. De este modo se hace imposible utilizar la cabecera de `RenderableWidgetConsole` dentro de la de `Logger`, puesto que se formaría un ciclo de inclusiones y MARS jamás compilaría. Así, no es posible utilizar un objeto *widget* de consola dentro de la clase `Logger`.

Para resolver este problema se hace uso de llamadas a funciones, en vez de llamadas a objetos. `Logger` no hará uso de un *widget* consola sino que cada vez que sea notificado a través de algunas de sus operaciones, llamará a su vez a otra función. Esta función será la encargada de ejecutar la llamada al objeto *widget* consola pertinente.

`Logger` contiene como atributos a tres punteros a función, que serán configurados a través de la operación `setConsoleFunctions()`, que acepta los tres punteros como parámetro. Los prototipos de función son la misma: una función que devuelve `void` y que acepta una cadena de C++. Estas funciones actuarán como *proxy* entre `Logger` y `RenderableWidgetConsole`, sin hacer uso explícito de uno de estos objetos.

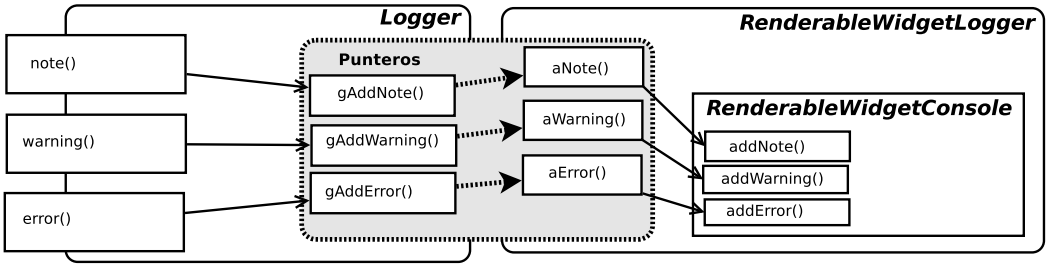

FIGURA 5.33: Propagación de las llamadas de *log*

La consola de depurado ha de ser única en el sistema, por eso se planteó la implementación de un `RenderableWidgetConsole` especial: `RenderableWidgetLogger`. Este nueva clase es un *Singleton* que contiene tres operaciones especiales. Estas tres operaciones corresponden a tres métodos estáticos que harán de funciones de consola en un `Logger`. Son `aError()` , `aWarning` y `aNote()`, que aceptan una cadena de C++.

Aprovechando el constructor de esta clase, se conectan las funciones (listado 5.28).

```
1  RenderableWidgetLogger::RenderableWidgetLogger() : RenderableWidgetConsole(640, 150) {
2      setPosition(320,75); // Center & Top of screen
3      hide();              // So far, hidden
4      Logger::getInstance()->setConsoleFunctions(RenderableWidgetLogger::aError,
5                                                 RenderableWidgetLogger::aWarning,
6                                                 RenderableWidgetLogger::aNote);
7      World::getInstance()->setConsole(this);
8  }
```

LISTADO 5.28: Conectando las funciones de log en `RenderableWidgetLogger`

De este modo, siempre que se crea un objeto del tipo `RenderableWidgetLogger`, este se utilizará como interfaz gráfico del *log* del sistema, de forma transparente al programador y al usuario, eximiéndolos de la responsabilidad de su gestión.

5.6.5. Sonido y síntesis de voz

Para la reproducción de sonidos se utiliza la clase `SoundTool`, de nuevo un *Singleton* puesto que sólo es necesaria una instancia de la misma. Esta clase inicializa el sistema de sonido de SDL y el sistema de síntesis de voz Festival.

`SoundTool` permite la carga y la reproducción de sonidos, así como la síntesis de voz. Los sonidos se almacenan en estructuras `Mix_Music` de SDL. Cada ver que se carga un sonido desde un archivo de audio, este se guarda en una lista *hash* que actuará como caché, de manera similar a la carga de texturas. La clave de esta tabla es el nombre de dicho archivo, evitando así que un mismo archivo se cargue más de una vez. Para tal cometido se proporciona la operación `getSound()` que devuelve una de esas estructuras de SDL a partir de una ruta a un archivo. Una de esas estructuras se puede reproducir con la operación `playSound()`.

Para sintetizar voz se utilizar la operación `createWAVEfromText()`, que hace uso de la función de Festival `festival_text_to_wave()` para sintetizar una cadena de C++ en un archivo *WAVE* de audio. Este archivo se podrá cargar y reproducir como cualquier otro.

En MARS se utiliza por defecto el archivo de voces español de Festival, pero se podría utilizar cualquier otro idioma.

5.6.6. *Bindings* de LUA

MARS realiza los *bindings* de algunas de sus clases con LUA, proporcionando una manera de cargar objetos y de asociar eventos.

Estos *bindings* se llevan a cabo en la clase `ScriptingLUA`, concretamente en su constructor, donde se asocian clases de LUA a las clases de C++ de MARS, utilizando la biblioteca *LuaBind*. También aquí se inicializa Lua y sus bibliotecas, y se crea la variable `__luaState`, que corresponde al estado de Lua, utilizado como manejador. Este manejador se publica en `mars::Configuration` para que pueda ser usado por otras partes del sistema.

Para dar soporte a los eventos en Lua, `WidgetFunction` implementa un constructor extra, que acepta como parámetro una cadena de C++. Esta cadena representa al nombre de una función de Lua. Cuando uno de estos objetos se construye de este modo, se llamará a esa función en el momento de su ejecución.Esto se consigue con un método estático en `WidgetFunction` (listado 5.29, línea 7), al que se le pasará la cadena, y que ejecutará la función pertinente, que ha de estar definida en el archivo de *script*.

```
1  struct WidgetFunction{
2    void (*f) (void*);
3    void* data;

5    WidgetFunction(void (*f_) (void*), void* data_) : f(f_), data(data_) {}

7    WidgetFunction(std::string name) {
8      __name = name;  f = &__f; data = (void*) __name.c_str();
9      Logger::getLogger()->note("WidgetFunction:: LUA function "
10                               + toString(reinterpret_cast<char*>(data))
11                               + " published");}
12   private:

14   static void __f(void* data) {
15     char* cTemp = reinterpret_cast<char*>(data);
16     try{
17       luabind::call_function<void>( Configuration::getConfiguration()->lState , cTemp );
18     } catch (...) {
19       Logger::getInstance()->warning("Error calling lua function:: " + toString(cTemp) + "
             : does it exist?");
20     }
21   }

23   std::string __name;
24   WidgetFunction();
25 };
```

LISTADO 5.29: `WidgetFunction` con soporte de funciones Lua

El método estático definido en la línea 17, será la función que se llamará siempre que se cree un `WidgetFunction` de este modo. En la línea 9 se asocia la función interna con la estática. Como se puede ver, en este caso, los datos asociados (`void* data`) son la propia cadena que representa el nombre de la función de Lua. Hay que hacer *cast* del tipo de dato a un `char*` (línea 18). Esa cadena de C se usará en la función `call_function()` de LuaBind (línea 20). Ésta ejecuta una función que deberá haber sido definida en el estado de Lua utilizado como parámetro.

Se permite la publicación de eventos `CLICK` y `SELECT` en botones y en listas. Para ello, tanto `RenderableWidgetButton` como `RenderableWidgetList` implementan dos operaciones especiales, `pusblishHandledEventLUA()` y `addActionLUA()`, respectivamente. Estas operaciones funciones de manera análoga a las originales, pero en vez de utilizar un `WidgetFunction`, utilizan una cadena de C++ para referirse a la función que debe ser ejecutada. El motivo de utilizar métodos diferentes es hacer sencillo el interfaz con Lua.

En el listado 5.30 se muestra un *script* de ejemplo. En este *script* se definen dos funciones (líneas 1-8), se ejecuta la función `init()` (línea 10), que corresponde con el método `ScriptingLUA::init()`, se obtiene el *Singleton* de `RenderableConstructor` (línea 12), y con él se crea y publica un modelo (línea 16) y se modifica su posición (línea 17). Más adelante se crea y se posiciona un botón (líneas 21-23) y se añade un evento `CLICK`, asociándolo a la función `test()` definida en el inicio. Después se añade una lista, se modifica su posición (líneas 25-26) y se publican varios eventos `CLICK` (líneas 27-31). Uno de ellos (línea 30) asocia un de los eventos a la función `saludo()`, que reproduce una frase utilizando `SoundTool.h`.

```
1  function test()
2      print ("test!")
3  end

5  function saludo()
6      sT = getSoundTool()
7      sT:playText("Hola amigo, bienvenido a MARS")
8  end
9  -------------------------------------
10 init() -- Init Scripting
11 -------------------------------------
12 rF = getRenderableConstructor() -- Singleton
13 -------------------------------------
14 -- Let's add the ESI
15 -------------------------------------
16 esi = rF:createModel("ESI", "daes/Plano.dae")
17 esi:setPosition(0, 0, 0) -- at 0,0,0
18 -------------------------------------
19 -- Adding a button
20 -------------------------------------
21 b = rF:createWButton("testButton", "CLICK HERE2", 300, 300, 230, 30) -- adds a button
22 b:setPosition(200, 300)
23 b:publishEvent("CLICK", "test")
24 -------------------------------------
25 w = rF:createWList("wListrr", 300, 300, 0.2, 5)
26 w:setPosition(300,300)
27 w:addAction("testing1", "test")
28 w:addAction("testing2", "test")
29 w:addAction("testing3", "test")
30 w:addAction("saludo", "saludo")
31 w:addAction("testing5", "test")
32 -------------------------------------
33 -- Adding a Light.
34 -------------------------------------
35 l = rF:createLight("Light zero")
36 l:setAmbient(0.5, 0.5, 0.5, 1.0)
37 l:setDiffuse(1.0, 1.0, 1.0, 10.0)
38 l:setPosition(0.0, 20.0, 20.0, 1.0)
39 -------------------------------------
```

LISTADO 5.30: Ejemplo de *script mars.lua*

6

Evolución y costes

En este capítulo se hablará de la evolución del proyecto en relación con la metodología elegida (ver capítulo 4). Mientras que en el capítulo anterior se recogía el resultado de su aplicación, en este se dará una visión global de los pasos realizados desde el comienzo del proyecto hasta su finalización.

6.1. Evolución del proyecto

Es esta sección se mostrará la evolución del proyecto según la metodología elegida. Antes de comenzar, se realizó un pequeño estudio sobre las herramientas necesarias para llevarla a cabo y se procedió a implantarlas. A continuación se procedió a estudiar los antecedentes y se llevó a cabo el análisis preliminar de requisitos. A partir de ese momento, comenzaron las iteraciones, que se recogen juntos con el resto de las etapas en los apartados siguientes.

6.1.1. Infraestructura para la metodología

MARS es un *framework* que se utiliza en un proyecto real, y que será ampliado y utilizado en el futuro. Este fue el motivo de la elección de la metodología utilizada, ya que los requisitos cambiaban según las necesidades del desarrollo de los mismos.

Para la aplicación correcta del prototipado evolutivo se llevó a cabo la revisión de varios sistemas de control de versiones y de control de proyectos. Tras la elección de los mismos (ver capítulo 4), se procedió a su instalación en `devoreto.esi.uclm.es`, en una máquina virtual proporcionada por la Escuela Superior de Informática (UCLM).

6.1.2. Concepto del software

En esta fase del proyecto se planteó la idea de un sistema modular para la RA, que resolviera el problema de la heterogeneidad en los dispositivos de vídeo, en los métodos de *tracking* y en el filtrado de datos proveniente de los mismos. También se estableció la funcionalidad mínima que debería tener el mismo, a modo de objetivo general del proyecto.

6.1.3. Análisis preliminar de requisitos

En esta etapa del proyecto se realizó el estudio del arte (capítulo 2), del que se extrajeron los requisitos iniciales. Gracias a esto también se refinaron los objetivos concretos del proyecto de fin de carrera.

Lo primero que se revisaron fueron los conceptos matemáticos necesarios para llevar a cabo un proyecto como este. Para acotar el estudio de los mismos, se realizó una revisión de las etapas del proceso de representación. Los conceptos y las estructuras matemáticas necesarias durante este proceso, también lo son en el resto del proyecto.

Tras el estudio de los sistemas de RA existentes y de las tecnologías relacionadas, se determinó realizar un motor gráfico desde cero (sección 5.3). El motivo fue que los motores analizados son demasiados complejos y no facilitaban la portabilidad del sistema a plataformas con poca potencia de cálculo. Así, se optó por escribir uno con el soporte mínimo para el tipo de aplicaciones que requiere la RA. El motor se basaría en OpenGL, ya que a día de hoy es la biblioteca 3-D libre con un mayor de dispositivos aceleradores compatibles, y con un mayor número de implementaciones en diferentes plataformas y arquitecturas. También se eligió Collada (sección 2.3.6) como el formato de modelos 3-D, puesto que es libre y está lo suficientemente respaldado como estándar *de facto*. Además de esto, cumple con creces las necesidades del proyecto.

También en esta etapa de determinó la escritura de un conjunto de *widgets* (controles gráficos) y de algún sistema de eventos para manejarlos. No se utilizó ninguna biblioteca con soporte de interfaz gráfica puesto que las analizadas carecían de las necesidades de MARS.

Se eligió OpenCV como biblioteca para la visión por computador y se analizaron sus capacidades para recoger vídeo de diferentes fuentes. El motivo de elegir OpenCV fue su calidad y su licencia, siendo además la biblioteca por excelencia para dicho campo.

A partir de esta etapa y de la anterior surgieron los requisitos preliminares siguientes:

- Soporte para diferentes tipos de fuentes de vídeo y calibración de las mismas.
- Motor gráfico ligero y con soporte para Collada.
- Conjunto mínimo de *widgets*.
- Integración de métodos de *tracking*.

La implementación se debía llevar a cabo utilizando estándares y software libre.

6.1.4. Diseño general

MARS se ideó desde el comienzo como tres subsistemas interconectados que se pudieran extender de manera independiente, uno de captura, uno de proceso y otro de representación. En este diseño general se incluyeron también las clases necesarias para dar soporte a vectores, matrices y cuaterniones.

6.1.5. Iteraciones

En este apartado se muestra un resumen de las partes implementadas en cada una de las iteraciones. Para conocer el funcionamiento de las clases mencionadas se puede revisar el capítulo 5 o el manual de referencia (ver anexo D).

Iteración 0

En esta iteración se realizó el primer prototipo inicial.

- **Desarrollo** - Se implementó la jerarquía de fuentes de vídeo (`VideoSources`) y la clase `FrameBuffer`, una primera implementación de `World` y un sistema rudimentario para el soporte de métodos de *tracking* en hilos independientes, al igual que la captura de frames de fuentes de vídeo de OpenCV.

- **Pruebas** - Se realizaron pruebas de integración entre los componentes implementados. Se detectaron y aislaron errores que serían corregidos en el caso de que este primer prototipo fuera aprobado.

- **Entrega de versión** - El prototipo capturaba frames y sobreimprimía un cubo sobre los mismos, además, un método de *tracking* de prueba procesaba la entrada, cambiando el color del vídeo capturado, aunque sin ofrecer ninguna información sobre la posición.

- **Análisis de los nuevos requisitos** - En esta etapa los requisitos se mantuvieron sin cambio, ya que el cometido de la misma era la aprobación del primer prototipo y del diseño inicial del sistema.

Iteración 1

En esta iteración se hizo una *refactorización* de las clases necesarias, se corrigieron los errores detectados en la iteración anterior y se terminaron de implementar los requisitos iniciales.

- **Desarrollo** - Aparte del *refactorizado*, se implementó el sistema de representables (3-D y *widgets*) y de cámaras del mundo, dando soporte para la carga de modelos Collada y creando las estructuras necesarias para alojar los datos de los mismos (mallas y texturas). Se incorporó soporte para fuentes de texto 2-D. También se implementó la jerarquía base para los métodos de *tracking* y se incorporó una cámara virtual a cada fuente de vídeo. En esta iteración se incorporaron las clases de vectores, matrices y cuaterniones; al igual que el sistemas de *logs*.

  Se creó la aplicación de calibrado y se añadió la opción de carga de los parámetros intrínsecos a las fuentes de vídeo. De igual forma, se añadió la posibilidad de utilizar una matriz de proyección calculada a partir de dichos parámetros.

- **Pruebas** - Para las clases de soporte matemático se realizaron pruebas unitarias con CppUnit, para el resto de clases se realizaron pruebas de integración. Estas

pruebas consisten en probar las operaciones de las clases, verificando algunas de sus propiedades. También se realizan estimaciones de velocidad, realizando operaciones sobre un millón de objetos. Gracias a esto, se ajustaron los parámetros de optimización del compilador para conseguir aumentar la velocidad en la versión de distribución.

En uno de los test se muestra cómo afecta la representación de punto flotante a las comparaciones de números y cómo dos cadenas de 32 bits diferentes corresponden a la misma cifra. Se justifica así el uso de una función de comparación con un error ε, para comparar números en esta representación.

- **Entrega de versión** - Esta versión podía manejar archivos Collada, obtener frames de fuentes de vídeo OpenCV y ejecutar métodos de *tracking* de manera independiente. Además, ya se podía hacer uso de las cámaras virtuales. El conjunto de *widgets* contenía sólo un botón.

- **Análisis de los nuevos requisitos** - Los nuevos requisitos para MARS fueron: una forma de integrar los métodos y de filtrar los datos obtenidos.

Iteración 2

En esta iteración se diseñó y se incorporó el sistema de filtros.

- **Desarrollo** - Se creó la clase `TrackingController` y la jerarquía de filtros (`TrackingFilter`), se implementó uno de ejemplo y se añadió la posibilidad de modificar una cámara virtual utilizando el resultado proporcionado por los mismos.

- **Pruebas** - Se realizaron pruebas de integración. En esta iteración MARS ya se estaba utilizando para el desarrollo de algunas demostraciones internas relacionadas con el proyecto ELCANO.

- **Entrega de versión** - Tras esta iteración se entregó una versión funcional de MARS con un diseño global casi final.

- **Análisis de los nuevos requisitos** - Los nuevos requisitos fueron: el soporte para la creación de representables de manera más sencilla, la creación de un *widget* consola y la implementación de un representable para reproducir vídeo sobre planos y de otro para utilizar etiquetas de texto 3-D, así como la incorporación de un sistema de configuración.

FIGURA 6.1: Tarea en Redmine correspondiente a la implementación del *log* visual.

Iteración 3

- **Desarrollo** - Se implementaron las clases `RenderableText`, el sistema de caché de fuentes, la clase para la creación de representables (`RenderableCreator`), se incorporó el soporte de VLC y su representable asociado (`RenderableVideoPlane`). También se diseñó el lenguaje de entrada para la configuración y se añadió la clase `Configuration` para procesarlo. La consola de texto (`RenderableWidgetConsole`) se asoció al sistema de *logs*.

- **Pruebas** - Se realizaron pequeños programas para probar las nuevas funcionalidades del sistema y se realizaron pruebas de integración de todos los componentes.

- **Entrega de versión** - Se entregó el *framwork* con las nuevas funcionalidades añadidas.

- **Análisis de los nuevos requisitos** - Se planteó la necesidad de incorporar un *widget* de lista y otro de mapa, un representable para realizar pruebas de orientación, otro para representar celdas WIFI y otro más para representar rutas 3-D.

Iteración 4

- **Desarrollo** - Se crearon los representables `RenderableWIFICell` y `RenderableLinePath` para dar soporte al dibujado de celdas WIFI y de caminos 3-D basados en líneas. Para las pruebas de orientación se creó `RenderableEmpty`. Por último se crearon las implementaciones `RenderableWidgetList` y `RenderableWidgetMap`, como *widgets* de lista y de mapa.

- **Pruebas** - Se probaron los nuevos componentes de manera aislada y se realizaron pruebas de integración.

- **Entrega de versión** - Coincidiendo con esta entrega, se realizó una demostración interna para el proyecto ELCANO, mostrando las nuevas funcionalidades de MARS.

- **Análisis de los nuevos requisitos** - Se decidió llevar a cabo la incorporación de un lenguaje de *script* a MARS, optando por LUA. También se tuvo que añadir soporte para cámaras de la empresa *uEye*, en concreto para su modelo *XS*.

Iteración 5

- **Desarrollo** - Se incorporó el soporte de *scripts* a través de la clase `ScriptingLUA`. Se relacionaron las clases necesarias y se rediseñó mínimamente el sistema de eventos para dar soporte a este lenguaje. Para añadir un tipo nuevo de cámara se implementó el `VideoDevice` concreto llamado `VideoDeviceUEye`.

- **Pruebas** - Se escribieron algunos programas de ejemplo en LUA para probar las funcionalidades añadidas. Se realizaron pruebas de integración con el resto del sistema.
- **Entrega de versión** - Igual que la anterior añadiendo el soporte de *scripts* en LUA.
- **Análisis de los nuevos requisitos** - Añadir soporte de audio y de síntesis de voz.

Iteración 6

- **Desarrollo** - Se implementaron las clases de soporte a la reproducción de audio y la responsable de la caché de sonidos. Igualmente se incorporó el sistema de síntesis de voz Festival.

  En esta última iteración se crearon los archivos `Makefile` de construcción de MARS para que pudiera ser utilizado como una biblioteca o como una aplicación.
- **Pruebas** - Individuales y de integración.
- **Entrega de versión** - Primera versión del *framework* MARS.

En la tabla 6.1 se muestran las fechas de entrega de la revisión de cada iteración.

It.	Fecha de entrega
0	19 - 03 - 2010
1	07 - 05 - 2010
2	11 - 07 - 2010
3	05 - 08 - 2010
4	21 - 09 - 2010
5	05 - 11 - 2010
6	03 - 12 - 2010

TABLA 6.1: Fechas de entrega según iteración

6.1.6. Estadísticas del repositorio

El sistema de control de versiones utilizado proporciona el número de revisiones y de cambios por mes en el código fuente del proyecto. En la figura 6.2 se muestra el gráfico de barras correspondiente a estas estadísticas.

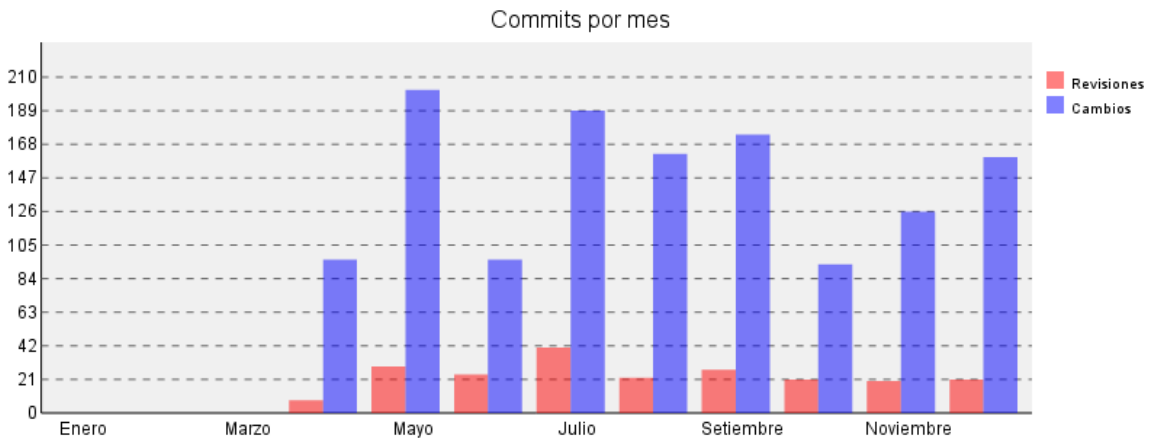

FIGURA 6.2: Estadísiticas de commits en `devoreto.esi.uclm.es`

6.2. Recursos y costes

El desarrollo de este proyecto ha durado 9 meses. El coste de los recursos utilizados se recoge en la tabla 6.2.

Recurso	Cantidad	Coste (euros / unidad)
Recursos humanos	1	11997,00[1]
Cámara UEye XS	1	563,20
Cámara Logitech AF-Sphere	2	117,16
Equipo de sobremesa[2]	1	532,20
Tablet HP	1	640,00
Servidor Virtual	1	180,00[3]
		14146,72

TABLA 6.2: Tabla de costes del proyecto

El *framework* ha sido implementando en C/C++ de manera exclusiva. En la tabla 6.3 se muestran las líneas de código que corresponden a cada subsistema.

En la tabla 6.4 se muestran estadísticas del código fuente según el tipo de archivo. Nótese que parte de la implementación se ha realizado en archivos de cabecera, puesto que C++ no permite la definición de funciones `inline` fuera de ellos.

[1] Sobre 16002 euros/año brutos.
[2] Pentium Celeron, 2GB RAM, NVidia 8400GS, Monitor 19”
[3] 20 euros/mes. 20GB de disco y 512MB de RAM (Datos: 1AND1)

Subsistema	Número de archivos	Líneas de código
Captura	30	1275
Proceso	22	1032
Representación	54	5005
Otros	27	1803
Total	133	9115

TABLA 6.3: Estadísticas del código fuente según subsistema

Tipo de archivo	Archivos	Líneas en blanco	Comentarios	Código	Todas las líneas
C++	65	2926	1320	**6533**	10779
Cabecera C/C++	66	1746	2538	**2452**	6736
Makefile	2	52	16	**130**	198
Total	133	4724	3874	**9115**	17713

TABLA 6.4: Estadísticas del código fuente según el tipo de archivo

7

Conclusiones y propuestas

En este capítulo se muestran los objetivos alcanzados durante el desarrollo de este proyecto de fin de carrera y se realizan una serie de propuestas en relación con el mismo.

7.1. Objetivos alcanzados

Tal y como se ha justificado en el capítulo 5, el objetivo principal ha sido cumplido. Se ha desarrollado una arquitectura que facilita y acelera en gran medida la construcción de aplicaciones de RA. El diseño del *framework* permite que un programador realice la incorporación sencilla de nuevos métodos de *tracking* , eximiéndole de responsabilidades como el control del vídeo de entrada, o la inicialización y destrucción de recursos.

La homogeneización de los datos procesados, la centralización de los mismos y el sistema de filtros implementado, hace que se puedan utilizar diferentes métodos de *tracking* para estimar la posición del usuario, ayudando de este modo a resolver el problema del registro (sección 1.1.2).

El motor gráfico es portable y da soporte a la carga de modelos 3-D de un estándar reconocido por la industria, permitiendo al diseñador utilizar el software que crea más adecuado para la creación de contenido. Estos modelos pueden estar optimizados, maximizando así el rendimiento del sistema de representación. Además, las texturas se cargan una sola vez, siendo accesibles por cualquier modelo, minimizando así el número de accesos a disco.

Las fuentes de vídeo que pueden ser utilizadas por el sistema van desde cámaras físicas conectadas de manera local hasta archivos de *stream* remotos. La jerarquía de vídeo permite la incorporación futura de nuevos tipos no soportados, y así se demostró en la iteración 5 del proyecto cuando hubo que añadir un tipo de videocámara basada en controladores privativos.

MARS da soporte a los tipos matemáticos más comunes en sistemas 3-D, prestando especial cuidado a la representación interna de la biblioteca 3-D elegida (OpenGL). De este modo se proporcionan estructuras y operaciones para vectores, puntos, matrices y cuaterniones.

Se incluyen también utilidades para reproducir sonido y para la síntesis de voz, así como una forma simple de configuración del sistema basada en un archivo de texto.

También se facilita el uso de MARS como un sistema de prototipado rápido de aplicaciones de RA, utilizando LUA como lenguaje de *script*. Esto permite a un programador crear la base de una aplicación que podrá ser modificada y adaptada más tarde sin tener que recompilarla. Esto facilita también la labor del diseñador, que no necesita utilizar un compilador para cambiar algunos comportamientos del sistema.

Además, ampliando los objetivos concretos del proyecto, se ideó un sistema de propagación y procesado de eventos de teclado y ratón para dar soporte al uso de *widgets* de una interfaz de usuario. La implementación de estos diálogos se realizó de tal modo que se pudieran adaptar a las necesidades de una aplicación cualquiera, tanto a la hora de cambiar su aspecto como a la hora de añadir eventos de respuesta asociados a los mismos.

7.2. Propuestas de trabajo futuro

MARS se puede utilizar en la investigación de nuevos métodos de *tracking* y de filtrado, así como en proyectos de aplicaciones completas de RA. En el momento de escritura de este documento MARS se utilizaba como plataforma de soporte y de interacción con el usuario en el proyecto ELCANO de la cátedra Indra-UCLM de tecnologías accesibles.

Como trabajo futuro relacionado con el *framework*, se realizan las siguientes propuestas de ampliación:

- **Sistema de reglas evento-condición-acción**. Ampliar MARS con un sistema de reglas ECA. Utilizando un de editor de nodos como el de Blender, se podría programar el comportamiento del sistema sin hacer uso de ningún lenguaje de programación.

Para llevar a cabo esta propuesta habría que crear una jerarquía de eventos, de condiciones y de acciones. Los eventos podrían estar provocados por un teclado o un ratón, pero también por los objetos 3-D de MARS (colisiones, posiciones y orientaciones relativas), por el cumplimiento de una lista de hitos, etc. Las condiciones comprobarían el estado del sistema y lo contrastarían con el requerido para que el evento dispare la acción. Las acciones ejecutarían una función (para esto ya está preparado MARS) y además podrían cambiar el estado del sistema de manera explícita.

Así, habría que ampliar la lista de eventos de MARS (añadiendo los tipos nuevos a la enumeración). También habría que crear una nueva clase evento, que representase a uno genérico, y que actuase como interfaz para la creación de nuevos tipos (asociados a la enumeración). Cada objeto que pudiera provocar eventos contendría una lista de estos. Por ejemplo, si un modelo 3-D tuviera que provocar un evento cuando se acercase a otro, cada vez que se cambiase la posición de uno de ellos, el objeto evento asociado debería comprobar la distancia entre los mismos y dispararse cuando la distancia sea la adecuada. Las condiciones se dividirían en simples y compuestas. Las simples serían propiedades o estados del sistema, y las compuestas serían la unión de varias simples con los operadores `not`, `and`, `or` y `xor`. Estas propiedades o estados se podrían encapsular junto con las operaciones en una clase estática, sin modificar la arquitectura actual de MARS. Para representar las acciones, se podría crear una clase base acción, parecida a la actual `WidgetFunction`, con los métodos adecuados para ejecutar funciones internas y externas, o para modificar el estado del sistema de manera directa.

- **Implementación de animaciones de modelos 3-D**. Dar soporte a la animación jerárquica y al uso de esqueletos (*vertex blending*) en los modelos importados desde Collada.

  Para realizar esta propuesta habría que implementar las clases relativas a la representación de curvas parametrizadas (de Bézier por ejemplo), que servirían para representar los cambios entre frames clave de la animación. Estos frames corresponderían a la posición y a la rotación de los huesos de los modelos. Para cada instante de tiempo entre un frame y otro habría que interpolar (mediante el valor de la curva utilizada por ejemplo) sus posiciones y rotaciones (utilizando la interpolación esférica de los cuaterniones, `slerp`). Finalmente habría que realizar el *vertex blending* de la malla con los huesos, cada vez que se pintase el modelo. La jerarquía de huesos o de objetos determinaría en torno a cuál de ellos se posicionan o

sea rotan sus descendientes. Esto es, el objeto padre determina el sistema de referencia del objeto hijo.

Para integrar este cambio en MARS se podría heredar de `RenderableModel` y especializarlo o simplemente añadir nuevos métodos y atributos a la clase para la gestión de animaciones y de jerarquía de mallas. Se sugiere el uso del patrón *composite* para la implementación de la jerarquía de objetos.

- **Optimizaciones concretas**. Realizar optimizaciones dependientes de la plataforma, para mejorar su rendimiento en aquellas que lo permitan. Ya que el sistema es modular, no habría que crear un nuevo proyecto.

  La mayor parte de estas optimizaciones son relativas a la implementación de OpenGL ES (OpenGL *for Embedded Systems*) y consistirían en hacer uso de las funciones de dibujado de mallas basadas en arrays de vértices, triángulos y texturas. Para maximizar la eficiencia de carga de objetos se sugiere utilizar un metaformato 3-D binario propio.

7.3. Conclusiones personales

El campo de la ingeniería informática no es sólo el de la ciencia de la computación y el de las disciplinas relacionadas con la misma. La ingeniería del software aporta las metodologías y las herramientas necesarias para poder realizar con éxito proyectos de cierta envergadura, que serían imposibles de llevar a cabo sin su ayuda.

Esto no excluye, en mi opinión, la necesidad de mantener el pensamiento científico durante la ejercitación de nuestra profesión. Después de todo, las matemáticas son la base de la programación y de la algoritmia. Comprender los procesos que se llevan a cabo para realizar una representación en tiempo real, ha hecho que se despierten dentro de mí sentimientos muy positivos hacia el álgebra lineal y la geometría. Espero sinceramente seguir profundizando en las mismas.

Conocer la arquitectura de un computador y de los sistemas de representación de punto flotante ha sido indispensable para diseñar algunas de las pruebas unitarias de las estructuras matemáticas. Aunque se tiende, desde mi criterio, a separar de forma radical la arquitectura de computadores y la ingeniería del software, este caso concreto demuestra que no se encuentran tan alejadas.

MARS es un proyecto muy completo donde han convergido matemáticas, programación, ingeniería del software, interacción persona-computador, arquitectura de computa-

dores, informática gráfica y visión por computador. La realización del mismo no sólo me ha supuesto un esfuerzo para mejorar como diseñador y programador, sino que además ha conseguido enfocar mi interés profesional en un campo tan extenso como la representación en tiempo real y la creación de videojuegos.

ANEXOS

A

Utilización de MARS

En este anexo se describe la forma de utilizar MARS, desde su construcción hasta algunos ejemplos de uso concretos. En el anexo D se muestra la manera de obtener un manual de referencia de MARS.

A.1. Construcción de MARS

Para construir MARS se proporcionan dos archivos *Makefile*. Uno de ellos se puede utilizar para construir MARS como una aplicación desde el código fuente que se proporciona, haciendo uso de un archivo `main.cpp`. El otro *Makefile* permite construir MARS como un *framework* compuesto por la biblioteca *libmars* y por la aplicación de calibrado `marsCalibrate`.

Estas dos formas de construcción se corresponden, respectivamente, con las invocaciones de `make` siguientes:

```
make -f Makefile
make -f Makefile.sharedlibs
```

Para compilar MARS para su uso con las cámaras *uEye* se utiliza el argumento `wueye=yes` al invocar `make`.

A.2. Orden de instanciación

MARS ha sido programado para eximir al usuario de la gestión de memoria y del control de los recursos e hilos generados. Para ello, la instanciación de objetos se realiza a través de factorías o de fachadas, cuya existencia será única. Esto se garantiza puesto que son clases *Singleton*.

Los objetos representados por estas clases liberan los recursos creados de manera automática. Algunos de ellos son utilizados por hilos en ejecución, así que es necesario terminar los hilos antes de liberar los recursos de los que hacen uso.

El orden de destrucción es inverso al orden de creación, el primer *Singleton* instanciado será el último destruido. `Logger` y `Configuration` no necesitan ser instanciados de manera explícita puesto que serán creados la primera vez que se usen, aun así, se pueden inicializar obteniendo su instancia (`getInstance()`) en cualquier momento. El orden sugerido de instanciación es el siguiente:

1. Instanciar un `VideoCapture`.
2. Instanciar un `VideoSource` en ese `VideoCapture`.
3. Inicializar SDL y OpenGL instanciando `VideoOutputSDLOpengl`.
4. Instanciar el mundo, añadirle el VideoSource como cámara y como fuente de frames para el fondo de la escena.
5. Instanciar el `TrackingController`.
6. Instanciar un filtro o dos y publicarlos en el `TrackingController`. Este paso es opcional, si no se realiza, se utilizará el filtro `dummy` por defecto.
7. Instanciar un método de *tracking* y publicarlo en el `TrackingController`.
8. Inicializar el sistema de audio, instanciando `SoundTool`.
9. Opcionalmente, ejecutar un *script* LUA de inicialización.
10. Ejecutar el *loop* de dibujado, incluyendo dentro la llamada para que `TrackingController` realice los cálculos pertinentes (`TrackingController::compute()`).

Si se sigue este orden, MARS se encargará de finalizar y de liberar los recursos de manera automática cuando se destruyan las instancias generadas por los *Sigleton*s, esto es, al finalizar la ejecución de la aplicación.

A.3. Esqueleto genérico de una aplicación

En el listado A.1 se muestra el esqueleto de una aplicación C++ basada en MARS, que utiliza un método de *tracking* visual absoluto basado en ARToolKit (creado para el proyecto ELCANO por otro miembro del grupo ORETO). Son en total 35 líneas, sin contar con las que están en blanco y con los comentarios.

```
1  #include "mars.h"

3  int main(){

5    // Inicializar el Logger y guardar mensaje de comienzo de ejecucion
6    mars::Logger::getLogger()->note("Beginning Execution");
7    mars::VideoCapture* vC =  mars::VideoCaptureFactory::getFactory()->createVideoCapture(
         OPENCV); // Crear un VideoCapture
8    mars::VideoDevice* vD_1= NULL;

10   vector<string> deviceNames = mars::getDevVideoCaptureNames();   // Listar los
         dispositivos

12   if (deviceNames.empty()){   // Si no hay ninguno, avisar.
13     mars::Logger::getLogger()->error("No video devices found... trying to "
14     "continue using a video file");
15   }
16   else // Si hay alguno, intentar usar el primero como un VideoSource
17   {
18     unsigned firstVideoDevice = atoi(&deviceNames.at(0)[mars::getDevVideoCaptureNames().at
           (0).size()-1]); // Obtener el numero al final de la cadena /dev/videoX
19     vC->addNewVideoSource(firstVideoDevice, "test"); //Crear un VideoSource llamado "test"
20     vD_1 = vC->getVideoDevice("test"); // Recuperarlo para manipularlo
21   }

23   if (vD_1 == NULL) {   // Salir si no hay un VideoSource disponible
24     mars::Logger::getLogger()->error("No device or video loaded. HALTING.");
25     return 0;
26   }

28   mars::VideoOutputSDLOpenGL::getInstance();   //Crear inicializar SDL y OpenGL
29   mars::World* theWorld = mars::World::getInstance();   // Crear y obtener World
30   theWorld->addCamera(vD_1);  // Adjuntar el VideoSource a World, pasar usar
31                               // su Camera3D interna.

33   theWorld->setBackgroundSource(vD_1);  //Usar ese VideoSource como fondo de la escena

35   // Crear y obtener el TrackingController
36   mars::TrackingController* tC = mars::TrackingController::getController();

38   // Crear y publicar un TrackingMethod que use el VideoSourceCreado
39   mars::TrackingMethodFactory::getInstance()->createTrackingMethod("ARToolKit", ARTK, vD_1
         , NULL);

41   mars::SoundTool::getInstance(); // AUDIO
42   mars::ScriptingLUA::getInstance()->execLUAFile("mars.lua"); // Ejecutar un archivo LUA

44   // Configurar la consola visual
45   mars::RenderableWidgetLogger* console = mars::RenderableWidgetLogger::getInstance();
46   theWorld->showFPS(true); // Mostrar frames por segundos

48   while(theWorld->step()){  // Bucle principal
49     // Computar los datos obtenidos por los metodos de tracking
50     tC->compute();
51   }
52   return 0;
53 }
```

LISTADO A.1: Esqueleto de una aplicación basada en MARS

Nótese que el programador puede utilizar el bucle `while` (línea 48) para incluir código que se ejecutará entre la generación de un *frame* de la escena y el siguiente. En este espacio es posible utilizar funciones OpenGL, no así en el resto del programa, donde habrá que utilizar las entidades representables de MARS (que siempre deberán crearse en los constructores de los métodos o en el `main` de la aplicación, nunca en fragmentos de código ejecutados de manera paralela).

A.4. Ampliación de MARS

A continuación se muestra cómo ampliar MARS con nuevos métodos de *tracking* y filtros. Todo el código que sigue se supone incluido en el espacio de nombres `mars`.

A.4.1. Implementación de un nuevo método de *tracking*

Para añadir un método de *tracking* a la arquitectura hay que realizar una implementación concreta de la clase `TrackingMethod`. En la sección 5.2.1 se mostró la jerarquía de los mismos (figura 5.11).

En el caso de querer crear un nuevo método de *tracking* visual que proporcionase datos de posicionamiento y orientación absolutos, habría que implementar la clase abstracta `TrackingMethodoAbsolute`, heredando públicamente de ella. Si el nombre de la nueva clase fuera por ejemplo `TMNew`, el esqueleto de la cabecera (archivo `TMNew.h`) sería el mostrado en el listado A.2.

```
#include "VideoSource.h"

namespace mars{

class TMNew : public TrackingMethodOAnsolute {
public:
    TMNew(VideoSource* vS);
    ~TMNew();

    void loopThread();

private:
    TMNew(){};

    VideoSource* _vS;
};

}
```

LISTADO A.2: Esqueleto de la cabecera de un `TrackingMethod` nuevo

El método de este ejemplo hará uso de sólo una fuente de vídeo (un `VideoSource`), y así se refleja en su constructor. Nótese que el constructor por defecto se oculta, declarándose como privado, para evitar que se instancie un método sin tener previamente una fuente de vídeo.

El esqueleto de la implementación (archivo `TMNew.cpp`) sería el mostrado en el listado A.3.

```
// Constructor
TMNew::TMNew(VideoSource* vS) : _vS(vS) {

  // Comprobar si el VideoSource es NULL
  if (_vS == NULL){
    Logger::getInstance()->error("Invalid VideoSource");
    __done = true; // Nunca se ejecuta loopThread()
    return;
  }

  // Aqui se pueden crear objetos Renderable o realizar cualquier tipo de inicializaciones
  // relativas al metodo de Tracking y a sus atributos. Se recomienda hacer todas las
  // inicializaciones e instanciacion de recursos basados en el heap aqui, con motivo de
  // preservar el rendimiento de bucle loopThread()
}

TMNew::~TMNew(){
  // Destruir los recursos del heap instanciados en el constructor
}

void TMNew::loopThread(){

  // Recoger un frame
  cv::Mat* frame;
  frame = _vS->getLastFrame();

  tUserl tempUserL;

  // Procesar el frame para obtener un tUserL o una matriz de orientacion y posicion, y
  // calcular el peso de esa percepcion
  ...
  ... Funciones OpenCV + Calcular + ...
  ...

  // Liberar la memoria del frame. MUY IMPORTANTE.
  delete frame;

  // Si obtiene una matriz, convertirla en tUserL
  tempUserL = TrackingMethod::Mat2TUserL(matriz, peso);

  // Publicar el resultado en el registro de percepciones
  addRecord(tempUserL);
}
```

LISTADO A.3: Esqueleto de la implementación de un `TrackingMethod` nuevo

Un método de *tracking* puede modificar las posiciones globales y relativas de un `Renderable` durante el método `loopThread()`. Esto es muy útil para representar objetos de actuación[1], independientes de la posición global del usuario, relativos a una marca o a algún conjunto de características naturales de la imagen.

[1]Un objeto de actuación sería por ejemplo un `VideoPlane` asociado a una marca de ARToolKit que admitiese algún tipo de interacción basada en el contexto.

Es muy conveniente añadir este nuevo tipo de método a la fábrica, para que el usuario final del mismo no tenga que instanciar el método y liberarlo, eximiéndolo así de esta responsabilidad. Para llevar esto a cabo, hay que añadir un nuevo tipo a la enumeración del archivo `TrackingMethodFactory`, en este ejemplo, `NEWT` (listado A.4).

```
enum trackingMethodType{
        ...,
        NEWT
};
```

LISTADO A.4: Nuevo tipo en la enumeración de la fábrica de métodos

A continuación, en el `switch` de `createTrackingMethod()` dentro del archivo `TrackingMethodFactory.cpp`, hay que incluir la nueva opción (listado A.5).

```
switch (type){

  // Otros metodos
  case ...
  case ...

  // El nuevo
  case NEWT: {
    TMNew* temp;
    temp = new TMNew(vS1);
    __tMethods[name] = temp;
    tC->publishMethodAbsolute(temp); // Publicarlo en el controlador
    return temp;
  }
}
```

LISTADO A.5: Modificación de `createTrackingMethod()`

De este modo, el método de *tracking* ya está preparado para ser usado, dándole un nombre, y pasándole un `VideoSource`, a través de `createTrackingMethod()` (listado A.6).

```
mars::TrackingMethodFactory::getInstance()->createTrackingMethod("Nuevo", NEWT, vD_1,
    NULL);
```

LISTADO A.6: Instanciando el nuevo método de *tracking*

Esta línea de código se situaría en el esqueleto sustituyendo a la número 57 del listado A.1, o a continuación de la misma.

A.4.2. Implementación de un nuevo filtro

La implementación de un filtro es similar a la de un método de *tracking* , pero heredando públicamente de `TrackingFilter`. Se deberán implementar las operaciones `mix()` y `applyFilter()`.

En el listado A.7 se muestra la cabecera de un filtro de ejemplo llamado `NewF` (archivo `NewF.h`).

```
namespace mars {

class NewF: public mars::TrackingFilter {
public:
    tUserL mix(const tUserL& t1, const tUserL& t2);
    tUserL applyFilter(std::vector<TrackingMethod*> tMs);

    TrackingFilterDummy();
    virtual ~TrackingFilterDummy();
  };

}
```

LISTADO A.7: Esqueleto de la cabecera de un filtro

La implementación del filtro se muestra en el listado A.8.

```
NewF::NewF(){
  // Instanciar los objetos necesarios en el heap
  ...
  __name = "FiltroNewF";   // IMPORTANTE. Nombrar el filtro.
}

NewF::~NewF(){
 // Liberar los objetos instanciados en el constructor
}

tUserL NewF::mix(const tUserL& t1, const tUserL& t2){
  tUserL temp;
  // Calcular el tUserL que se produce al mezclar dos de ellos
  ...
  ...
  ...

  return temp;
}

tUserL NewF::applyFilter(std::vector<TrackingMethod*> tMs){

  // Recorrer los TrackingMethod ...
  for (unsigned i = 0; i < tMs->size(); ++i){
    // Procesar los registros de la manera deseada...
    // Por ejemplo, para mezclar con mix() la ultima percepcion de cada metodo (temp
        deberia
    // contener la ultima percepcion del primer metodo de la lista antes entrar en el for)
    temp = mix(temp, tMs[i]->getLastUserL());
  }

  // Devolver el tUserL
  return temp;

}
```

LISTADO A.8: Esqueleto de la implementación de un filtro

Para utilizar el nuevo filtro, se podría incluir en el esqueleto (listado A.1), justo después de la instanciación del `TrackingController`, las líneas de código del listado A.9. De esta forma MARS utilizaría el nuevo filtro para unir las percepciones de todos métodos de *tracking* creados.

```
// Instancia en la stack
mars::NewF filtro;

// Publicar el filtro en el controlador
tC->addFilter(&filtro);

// Seleccionar el uso de este filtro
tC->selectFilter("FiltroNewF");
```

LISTADO A.9: Instanciación y publicación de un nuevo filtro

A.5. Ejemplos de uso de los representables

La forma de utilizar los objetos representables que utiliza MARS es muy sencilla, y está descrita en la sección 5.3.6. Básicamente consiste en crearlos con un nombre único asociado para poder recuperarlos desde cualquier parte de la aplicación, y en posicionarlos. En el caso de los *widgets* también existe la posibilidad de añadir eventos y funciones asociadas.

A.5.1. Ejemplos de uso de objetos 3-D

Los objetos representables 3-D se posicionan de manera absoluta a todo el mundo virtual o de manera relativa a la cámara utilizada. El flujo de trabajo es el que sigue: supóngase que un diseñador proporciona un modelo 3-D en un archivo *.dae* de Collada junto con las texturas utilizadas; esas texturas se copiarán al directorio elegido (véase la sección 5.6.2), y el archivo *dae* se cargará en el mundo virtual utilizando el *Builder* de objetos representables `RenderableConstructor` (listado A.10).

```
mars::RenderableConstructor::getInstance()->createModel("ESI", "daes/esi.dae");
```

LISTADO A.10: Añadiendo un modelo *dae* a la aplicación

Esta creación sólo se realiza una vez, al igual que la carga del objeto y de las texturas desde disco. El usuario de MARS puede crear un `Renderable` en cualquier momento, menos en los fragmentos de programa que se ejecutan de forma paralela (`TrackingMethod::loopThread()`). Esto es debido a que es necesario garantizar el uso secuencial y ordenado de las funciones de OpenGL, para no provocar un error interno en dicha biblioteca. Sin embargo, es posible utilizar los métodos proporcionados por estos objetos (excepto `draw()`) durante toda la ejecución.

Para obtener un objeto `Renderable` se puede hacer de nuevo uso de la clase `RenderableConstructor`, en concreto de los métodos `get__()` implementados para cada tipo de objeto representable de MARS.

En el caso del objeto `esi` anterior, se podría recuperar de la forma mostrada en el listado A.11.

```
RenderableModel* esiNuevo =  mars::RenderableConstructor::getInstance()->getModel("ESI");
```

LISTADO A.11: Recuperando un modelo previamente cargado

Este objeto representable sería utilizado como un objeto con una posición global, en concreto, como su posición no ha sido modificada, estaría situado en el punto $(0,0,0)$. Si así se requiere, esta posición se puede cambiar a través una función disparada por un evento o por un método de *tracking* , según convenga.

Si se desea posicionar un objeto de manera relativa a la cámara (por ejemplo porque un método de *tracking* decide que lo quiere colocar encima de una marca que se está visualizando en ese preciso instante), es necesario configurar el `Renderable` para que utilice la matriz interna de posicionamiento (`useMatrix(true)`) y actualizar dicha matriz cada vez que se desee cambia la posición y orientación del mismo (cuando se mueva la marca, por ejemplo. En el listado A.12 se muestra una mezcla de pseudocódigo y de C++ para ilustrar esta situación.

```
// Constructor de un TrackingMethod
MetodoTK::MetodoTK(VideoSource *vS){
  ...
  // Crear un plano con video
  mars::RenderableConstructor::getInstance()->createVideoPlane("VID", "intro.avi", 320,
      240);
  ...
}

// Proceso
void MetodoTK::loopThread(){

  ...

  if (MARCA_DE_VIDEO_DETECTADA){
    mars::RenderableVideoPlane* rV = mars::RenderableConstructor::getInstance()->
        getVideoPlane("VID"); // Recuperar el objeto
    rV->show(); // Mostar el objeto

    mars::Matrix16 temp = OBTENER_MATRIZ_DE_LA_MARCA;
    rV->setMatrix(temp);
  } else {
    // Ocultar el VideoPlane
    mars::RenderableConstructor::getInstance()->getVideoPlane("VID")->hide();
  }

  ...

}
```

LISTADO A.12: Usando un modelo de forma relativa a la cámara en un `TrackingMethod`

La implementación anterior es sólo un ejemplo para ilustrar el uso de esta característica de MARS. Se recomienda que el usuario que implemente un método de *tracking* y haga uso de la misma, guarde en uno de los atributos de la clase un puntero a este objeto representable. De esto modo se logrará evitar tener que realizar la petición continua a `RenderableConstructor`, iteración tras iteración, con la consiguiente penalización de rendimiento que supone recuperar la instancia del *singleton* y buscar en la tabla *hash* (`std::map`) el nombre del objeto.

Cabe destacar que un modelo se puede representar con trazado de líneas y con una animación de dicho trazado que lo destaque de los demás (ver listado A.13).

```
// Modo normal
r->setRenderMode(0);

// En wireframe
r->setRenderMode(1);

// En wireframe ocultando las caras traseras
r->setRenderMode(2);

// Habilitar la animacion del wireframe
r->enableWireAnim()

//Desahilitar la animacion del wireframe
r->disableWireAnim();
```

LISTADO A.13: Algunos modos de representación de un modelo

A.5.2. Ejemplos de uso de *widgets*

Los *widgets* de MARS sirven para crear una interfaz de usuario 2-D, superpuesta a la escena 3-D. Los detalles de cómo se lleva a cabo la implementación de los componentes de la misma se muestran en la sección 5.3.8. A continuación se mostrarán ejemplos de uso de algunos de estos componentes.

El componente más sencillo es `RenderableWidgetImage2D`, que lo único que hace es mostrar una imagen 2-D superpuesta. Este *widget* es útil para mostrar un logotipo, elementos de información basados en imágenes en el GUI de la aplicación o para ser utilizado en la construcción de otros *widgets* (véase la sección 5.3.8 donde se muestra cómo se utiliza en el *widget* mapa).

Para crear uno de estos *widgets* (al igual que para cualquiera de los otros), se utiliza de nuevo `RenderableConstructor`. En el caso de que el diseñador proporcione una imagen (`logo.png`) para su uso como logotipo en la interfaz gráfica, el programador la movería al directorio de texturas y si no fuera de dimensiones potencia de dos, podría escalarla de

manera no uniforme hasta que lo fuera, para aumentar así el rendimiento de la aplicación. Suponiendo que la imagen midiera originalmente 200 x 100 píxeles, y que se desease mostrar pegada a la esquina superior izquierda, habría que situarla en el píxel $(100, 50)$. Esto se puede realizar en el momento de su creación (y publicación, por consiguiente), como se muestra en el listado A.14. Nótese que esa posición corresponde al centro del *widget*.

```
mars::RenderableConstructor::getInstance()->createImage2D("LOGO",
"logo.png", 320, 240)->setPosition(100,50);
```

LISTADO A.14: Creación de una imagen como logotipo en el GUI

Un escenario más interesante para el uso de un *widget* es la necesidad de utilizarlo para realizar cambios en la ejecución de la aplicación. Por ejemplo, supóngase que se desea asociar una serie de posibles acciones a un *widget* lista. Las opciones serán mostrar un modelo representable (con un nombre único `TESTOBJ`) u ocultarlo. Suponiendo que ese objeto ya está creado, lo siguiente ha realizar sería crear dos funciones (estáticas y con el prototipo adecuado mencionado en la sección 5.3.8) que muestren y oculten ese representable. Después se creará el *widget* y se añadirán dos opciones, asociadas con las funciones previamente creadas. La forma de llevar esto a cabo se muestra en el listado A.15.

```
void ocultarModelo(void* data){
  std::string name(static_cast<char*>(data));
  mars::RenderableConstructor::getInstance()->getModel(s)->hide();
}

void mostrarModelo(void* data){
  std::string name(static_cast<char*>(data));
  mars::RenderableConstructor->getModel(s)->show();
}

...

RenderableWidgetList* createWList(const std::string& name,
                        const unsigned& w, const unsigned& h,
                        const float& widthPortionUpDown = 0.2,
                        const unsigned& nVisibleOptions = 5);


// Creando el widget de lista de opciones
mars::RenderableWidgetList* rL = mars::RenderableConstructor::getInstance()->createWList("
    LISTA", 200, 300);

// Creando las opciones y publicando las funciones
rL->addAction("Ocultar!", mars::WidgetFunction(ocultarModelo,
                                               (void*)(char*) "TESTOBJ"));
rL->addAction("Mostrar!", mars::WidgetFunction(mostrarModelo,
                                               (void*)(char*) "TESTOBJ"));
```

LISTADO A.15: Asociando un *widget* con algunas acciones

A.6. Uso de *script*s

La utilización de *scripts* no lleva asociada ninguna restricción. El usuario de MARS es libre de ejecutar tantos archivos de *script* como quiera, en el momento que desee. Para ello basta con utilizar la clase `ScriptingLUA` tal y como se muestra en el listado A.16.

```
ScriptingLUA::getInstance()->execLUAFile("archivo.lua");
```

LISTADO A.16: Ejecución de un archivo con un *script* de LUA

En un *script* se puede utilizar libremente el lenguaje LUA [Ier06] para interactuar con las clases asociadas de MARS. En el listado 5.30 (apartado 5.6.6) se muestra un ejemplo de *script*.

La forma de obtener los objetos *singleton* se muestra en el listado A.17.

```
-- El singleton de RenderableConstructor
rC = getRenderableConstructor()

-- El singleton de SoundTool
sT = getSoundTool()

-- El singleton de World
w = getWorld()
```

LISTADO A.17: Obtención de los *singletone*s de MARS desde LUA

Se permite la creación de objetos representables utilizando el constructor obtenido con `getRenderableConstructor()`, de forma similar a cómo se hace en C++. Por ejemplo, para crear un botón se utiliza el método `createWButton` con los mismos parámetros que aceptan dicho método en C++. Para publicar un evento se utiliza `publishEvent`, que se asocia a uno de los tipos permitidos (`CLICK` o `SELECT`) y a una función de LUA.

Además, se permite cambiar la posición de cualquiera de los objetos representables a través de su operación `setPostiion()`, como en C++.

B

Plantilla *Singleton*

En MARS se utiliza una plantilla de C++ para la creación de clases *Singleton*. A partir de esta plantilla el compilador generará el código necesario para cada clase de la que se quiera tener una única instancia.

Esta plantilla se basa en una clase que no se puede copiar. El motivo es no permitir duplicar un objeto *singleton*.

B.1. `uncopyable`

Esta clase (listado B.1) representa un objeto que no se puede copiar. La idea está tomada de S.Meyers [Mey05].

```
#ifndef UNCOPYABLE_H_
#define UNCOPYABLE_H_

namespace mars{

class uncopyable{
protected:
  uncopyable() {}
  ~uncopyable() {}
private:
  uncopyable(const uncopyable &);
  uncopyable & operator=(const uncopyable &);
};

}
#endif
```

LISTADO B.1: Declaración e implementación de la clase `uncopyable` (`uncopyable.h`)

Esta clase declara como privados el constructor de copia y el operador de asignación, evitando así que el usuario de clases derivadas haga uso de los mismos.

B.2. Singleton

La clase *templatizada* `Singleton` se muestra en el listado B.2. Nótese cómo la clase `Singleton` hereda públicamente de `uncopyable`.

```
1  #ifndef __SINGLETON_H__
2  #define __SINGLETON_H__

4  #include "Uncopyable.h"
5  #include <memory>
6  #include <cstdlib>

8  namespace mars{

10 class Singleton: public uncopyable {
11 public:
12   static T* getInstance(){
13     if (__instance == NULL){
14       __instance = new T;
15       std::atexit(destroy);
16     }
17     return __instance;
18   }

20 private:
21   static void destroy(){
22     delete __instance;
23   }

25   static T* __instance;

27 };

29 template<typename T>
30 T* Singleton<T>::__instance = NULL;


33 }
34 #endif
```

LISTADO B.2: Plantilla utilizada para la generación de clases *singleton* (`Singleton.h`)

Obsérvese que en la línea 16 se gestiona la ejecución del método privado `__destroy()` al finalizar el programa, que realizará un `delete` de la instancia. Esto provocará la ejecución del *destructor* de la clase instanciada, y liberará la memoria ocupada por esta.

B.2.1. Uso de la clase *templatizada* `Singleton`

Para crear una clase utilizando esta plantilla hay que heredar de la clase que se obtiene de hacer uso de la plantilla `Singleton` en la propia clase. También es necesario declarar esta clase *templatizada* como amiga (`friend`) de la clase que se esté creando. El listado B.3 muestra el código necesario.

```
#include "Singleton.h"

class NuevoSingleton : public Singleton<NuevoSingleton> {
public:
    ...

private:
  friend class Singleton<NuevoSingleton>;
};
```

LISTADO B.3: Ejemplo de uso de la plantilla `Singleton`

Para obtener la instancia de `NuevoSingleton`, tan solo hay que ejecutar su método estático `NuevoSingleton::getInstance()`, que devuelve un puntero a la misma.

C

Diagrama de clases

En la figura C.1 se muestra el diagrama de clases general de MARS (se recogen las más importantes y se omiten algunas dependencias). Para un mayor detalle, se recomienda la generación del manual de referencia en Doxygen (ver anexo D, donde se detallan los pasos para su generación). Este manual se incluye en su versión digital en el CD anexo a este documento.

Dada la extensión del código fuente (más de 9000 líneas), no se incluye en los anexos del presente documento el código de ninguna de las clases que forman MARS. Los fuentes completos y algunos ejemplos de uso se encuentran en el CD adjunto.

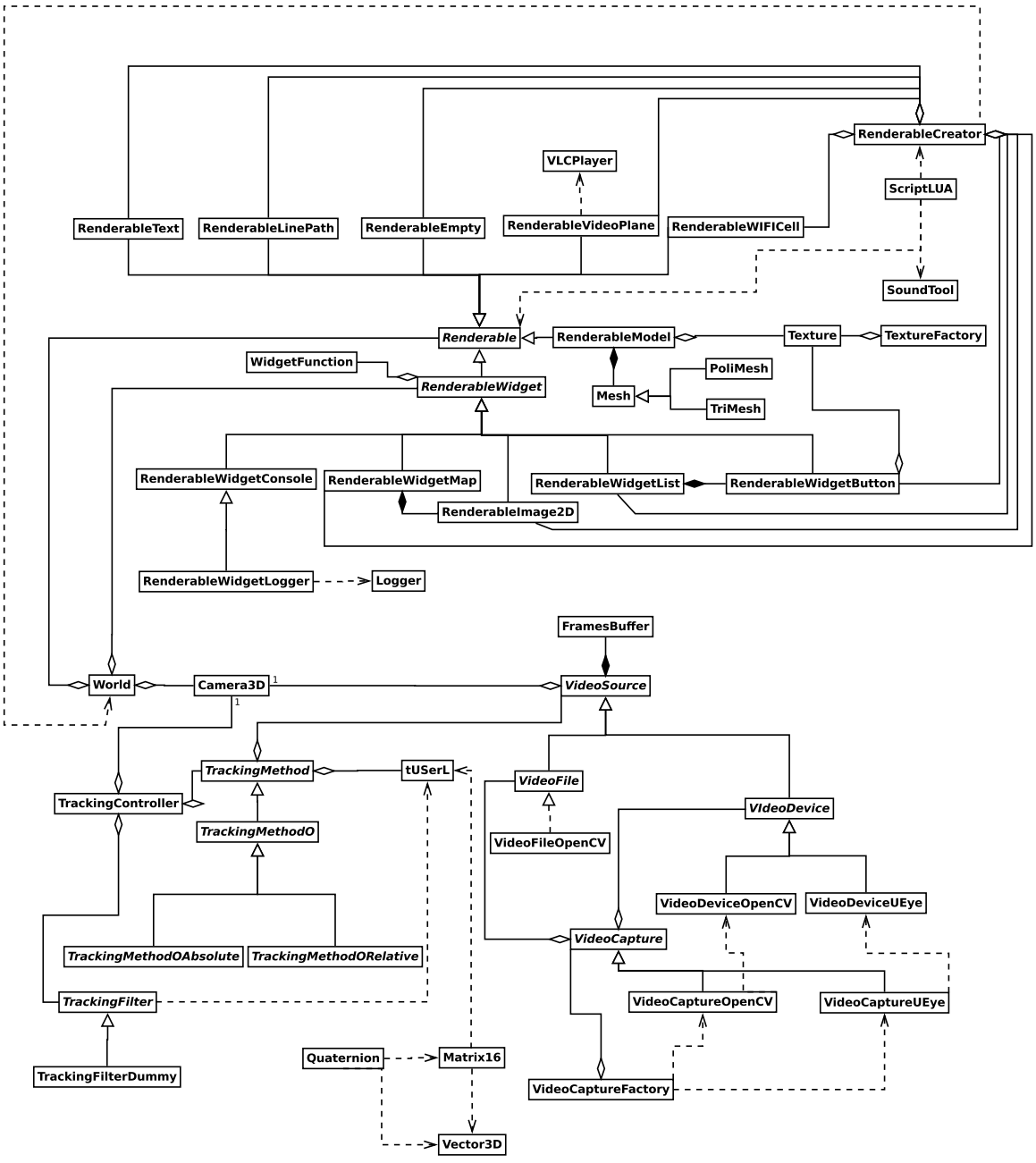

FIGURA C.1: Diagrama de clases

D

Manual de referencia

En el CD que acompaña a este documento se incluye el código fuente de MARS y el manual de referencia de la biblioteca. Es posible generar este manual de referencia a partir del código fuente. Este manual (escrito en inglés por motivos prácticos) contiene la descripción de las clases y métodos de MARS, y también información sobre la herencia con pequeños diagramas de clases en cada una de ellas. Dada su extensión (124 páginas en su versión en PDF), se ha optado por no incluirlo en este documento impreso.

D.1. Generación del manual de referencia

El manual de referencia de MARS se puede obtener a partir del código fuente, ya que este ha sido documentado utilizando Doxygen. Para ello, lo único que hay que hacer es invocar el comando `make` de la siguiente forma:

```
make doc
```

Esto genera la documentación dentro del subdirectorio `doc`. Se genera en formato HTML y LaTeX. Para visualizar la documentación en un navegador web, basta con abrir el archivo `doc/html/indel.html`. Para convertir la documentación generada en LaTeX en un archivo PDF, primero hay que compilarla:

```
cd doc/latex
make
```

Esto genera un archivo llamado `refman.pdf` en ese directorio, que corresponde con el manual de referencia en su versión PDF generada con LaTeX.

E

GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall

directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

5. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

6. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

7. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

8. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

9. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

10. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

Bibliografía

[A+97] R.T. Azuma et al. A survey of augmented reality. *Presence-Teleoperators and Virtual Environments*, 6(4):355–385, 1997.

[AB06] R. Arnaud and M.C. Barnes. *COLLADA: sailing the gulf of 3D digital content creation*. AK Peters, Ltd., 2006.

[AMHH08] T Akenine-Möller, E Haines, and N Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[Azu00] D. Azuma. The GLOW Toolkit, 2000. Available from: `http://glow.sourceforge.net/`.

[Bar06] M. Barnes. COLLADA-Digital Asset Schema Release 1.4. 1 Specification. *Sony Computer Entertainment Inc*, 2006.

[BK08] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, 2008.

[BT97] Alan W. Black and Paul A. Taylor. The Festival Speech Synthesis System: System documentation. Technical Report HCRC/TR-83, Human Communciation Research Centre, University of Edinburgh, Scotland, UK, 1997. Avaliable at http://www.cstr.ed.ac.uk/projects/festival.html.

[BTVG06] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer Vision–ECCV 2006*, pages 404–417, 2006.

[CLM+03] B. Cascales, P. Lucas, J.M. Mira, A.J. Pallarés, and S. Sánchez-Pedreño. *El libro de LaTeX*. Prentice Hall, 1ª edition, 2003.

[Com09] Boost Community. *Boost 1.41.0 Library Documentation*, 2009. Available from: http://www.boost.org/doc/libs/1_41_0.

[CS03] GCC Developer Community and R.M. Stallman. *Using GCC: The GNU Compiler Collection (Version 3.3.1)*. Free Software Foundation, 2003.

[CWF98] TA Clarke, X. Wang, and JG Fryer. The principal point and CCD cameras. *The Photogrammetric Record*, 16(92):293–312, 1998.

[dIU07] Escuela Superior de Informática (UCLM). *Normativa del Proyecto Fin de Carrera*, 2007. Available from: http://www.esi.uclm.es:8081/www/documentos/Normativas/NormativaPFC2007.pdf.

[dox10] *Doxygen Manual*, 2010. Available from: http://www.stack.nl/~dimitri/doxygen/manual.html.

[FMHW97] S. Feiner, B. MacIntyre, T. Höllerer, and A. Webster. A touring machine: Prototyping 3D mobile augmented reality systems for exploring the urban environment. *Personal and Ubiquitous Computing*, 1(4):208–217, 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissided. *Design Patterns*. Addison Wesley, 1995.

[Gmb09] IDS Imaging Development Systems GmbH. *uEye Software Development Kit. Programming Manual*, 2009. Available from: http://www.ids-imaging.de/frontend/files/uEye_Programming_Manual.pdf.

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[Gou71] H. Gouraud. Continuous shading of curved surfaces. *IEEE transactions on computers*, pages 623–629, 1971.

[Han05] Andrew J. Hanson. Visualizing quaternions. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. Available from: http://doi.acm.org/10.1145/1198555.1198701, doi:http://doi.acm.org/10.1145/1198555.1198701.

[Hen02] D. Henry. The quake II's md2 file format, 2002. Available from: http://tfc.duke.free.fr/coding/md2-specs-en.html.

[Ier06] Roberto Ierusalimschy. *Programming in Lua, Second Edition*. Lua.Org, 2006.

[KB99] H. Kato and M. Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *iwar*, page 85. Published by the IEEE Computer Society, 1999.

[Kil96] M.J. Kilgard. The OpenGL utility toolkit (GLUT) programming interface API version 3, 1996.

[Kle06] G. Klein. Visual tracking for augmented reality. *University of Cambridge, PhD Thesis*, 2006.

[Li01] M. Li. «Correspondence Analysis Between The Image Formation Pipelines of Graphics and Vision». In *Proceedings of the IX Spanish Symposium on Pattern Recognition and Image Analysis, Benicasim (Castellón), Publications de la Universitat Jaume I, Spain, Universitat Jaume I*, pages 187–192, 2001.

[Loe09] J. Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, 2009.

[Lut06] Mark Lutz. *Programming Python*. O'Reilly Media, Inc., 2006.

[Mar02] J. Marini. *Document Object Model : Processing Structured Documents*. McGraw-Hill/OsborneMedia, 2002.

[Mey05] S. Meyers. *Effective C++: 55 specific ways to improve your programs and designs*. Addison-Wesley Professional, 2005.

[MF96] B. MacIntyre and S. Feiner. Language-level support for exploratory programming of distributed virtual environments. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 83–94. ACM, 1996.

[MGDB04] B. MacIntyre, M. Gandy, S. Dow, and J.D. Bolter. DART: a toolkit for rapid design exploration of augmented reality experiences. In *Proceedings of the 17th annual ACM symposium on User Interface Software and Technology*, pages 197–206. ACM, 2004.

[Opp02] R. Oppermann. User interface design. *Handbook on information technologies for education and training*, pages 233–248, 2002.

[O'S09] B. O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, 2009.

[Pho75] B.T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[PT02] W. Piekarski and B. Thomas. ARQuake: the outdoor augmented reality gaming system. *Communications of the ACM*, 45(1):36–38, 2002.

[Roo03] Ton Roosendaal. *The Official Blender Game Kit: Interactive 3d for Artists*. No Starch Press, San Francisco, CA, USA, 2003.

[RS06] P. Rademacher and N. Stewart. Glui user interface library. *GLUI User Interface Library, v1*, 4, 2006.

[S+98] H. Schulzrinne et al. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), apr 1998. Available from: `http://www.ietf.org/rfc/rfc2326.txt`.

[SE91] K. Shoemake and O. Enterprises. Quaternions and 4 x 4 matrices. *Graphic Gems II*, page 351, 1991.

[Sho85] K. Shoemake. Animating rotation with quaternion curves. *ACM SIGGRAPH computer graphics*, 19(3):245–254, 1985.

[Sta99] R. Stallman. The GNU operating system and the free software movement. 1999.

[Sta07] R.M. Stallman. *GNU Emacs manual (Version 22)*. Free Software Foundation, 16ª edition, 2007.

[Str99] B. Stroustrup. *C++ Programming Language*. Addison Wesley, 3ª edition, 1999.

[SWND04] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL programming guide*. Addison-Wesley, 2004.

[Val98] J.R. Vallino. *Interactive augmented reality*. PhD thesis, Citeseer, 1998.

[VB04] James Van Verth and Lars Bishop. *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[WL04] R.S. Wright and B. Lipchak. *OpenGL superbible*. Sams Indianapolis, IN, USA, 2004.

[Zha02] Z. Zhang. A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(11):1330–1334, 2002.