MINERVA: SISTEMA DE ESPECIFICACIÓN LÓGICA BASADO EN SENSORES, CONTROLADORES Y ACTUADORES PARA APLICACIONES DE REALIDAD AUMENTADA

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

Minerva: Sistema de especificación lógica basado en Sensores, Controladores y Actuadores para aplicaciones de Realidad Aumentada

César Mora Castro

Julio, 2011

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

Minerva: Sistema de especificación lógica basado en Sensores, Controladores y Actuadores para aplicaciones de Realidad Aumentada

Autor: César Mora Castro
Director: Carlos González Morcillo

Julio, 2011

César Mora Castro

E-mail: Cesar.Mora@alu.uclm.es, cesarmoracastro@gmail.com
Telefono: 926 295 300 Ext:96677
Web site: `http://theminervaproject.wordpress.com`

TRIBUNAL:

Presidente:

Vocal 1:

Vocal 2:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

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

A mi familia y amigos,
por su apoyo y amistad.

Resumen

La informática gráfica es un área de la informática que ha venido evolucionando de forma vertiginosa durante las últimas tres décadas, con aplicación directa en multitud de mercados como la industria cinematográfica, los videojuegos, la medicina o la educación. La Realidad Aumentada supone un nuevo enfoque de esta área que está cambiando la forma en que nos comunicamos con los ya imprescindibles dispositivos informáticos.

La gran heterogeneidad de arquitecturas hardware, la existencia de multitud de técnicas de *registro* de la realidad, la fuerte base matemática subyacente y el amplio compendio de conocimientos informáticos necesarios, hacen que desarrollar aplicaciones de Realidad Aumentada sea, hoy en día, un complejo trabajo de ingeniería.

El presente proyecto fin de carrera surge con el objetivo de proporcionar un lenguaje de alto nivel (denominado *Minerva Specification Language*) mediante el cual definir la lógica de aplicaciones de Realidad Aumentada, además de una plataforma software para su interpretación y posterior ejecución. De esta forma, el conocimiento técnico requerido se reduce drásticamente, abstrayendo al usuario de numerosas cuestiones como la captura de vídeo, la representación de gráficos o el *registro* de la realidad.

Minerva ofrece multitud de características multimedia, como la reproducción de sonidos, representación de modelos tridimensionales o simulación física. Incluso para usuarios más avanzados, se permite extender su funcionalidad utilizando el lenguaje de *scripting* Python.

Gracias a la potencia descriptiva del enfoque basado en Sensores, Controladores y Actuadores, y al acceso de bajo nivel basado en el lenguaje Python, con Minerva se pueden crear aplicaciones de Realidad Aumentada en una gran variedad de dominios con una baja complejidad de desarrollo. El eslógan de Minerva resume el principal objetivo seguido en su construcción: *Minerva: building Augmented Reality apps has never been so easy!*.

Abstract

Computer graphics is an area of computer science that has had a striking evolution during the last three decades, and has a direct application in several markets such as film industry, videogames, medicine or education. Augmented Reality is a new approach of this technology that is changing the way used to communicate with the indispensables.

The wide heterogeneity of hardware architectures, the existence of several tracking methods, and the extense number of computer skills needed, make the development of Augmented Reality applications a complex engineering task nowadays.

The current Final Project arise with the goal of provide a high-level language (called *Minerva Specification Language*) whereby the logic of Augmented Reality applications can be defined, as well as a software platform to interpret and run it. Thus, the technical knowledge is decreased dramatically, releasing the user of several issues like video capturing, graphic rendering or reality tracking.

Minerva offers a lot of multimedia features as sound playing, the representation of tridimensional objects or physics simulation. Even for more advanced users, the functionality can be extended through the Python scripting language.

Thanks to the descriptive power of the Sensors, Controllers and Actuators based approach, and the low level access provided by the Python language, several Augmented Reality applications can be made with a low development complexity. The Minerva's slogan resumes its main aim: *Minerva: building Augmented Reality apps has never been so easy!*.

Índice general

Índice de figuras

Listado de acrónimos

MSL	Minerva Specification Language
MAO	Minerva Augmenter Object
API	Application Programming Interface
MLB	Minerva Logic Brick
CD	Compact Disc
MGE	Minerva Game Engine
CPU	Central Processing Unit
GPU	Graphics Processing Unit
HDM	Head Mounted Display
SCA	Sensor, Controlador, Actuador
MIT	Massachusetts Institute of Technology
XML	Extensible Markup Language
RA	Realidad Aumentada
RM	Realidad Mixta
VE	Virtual Environments

Agradecimientos

Son muchísimas las personas a las que me gustaría agradecer el que yo haya llegado hasta aquí.

En primer lugar, gracias a mi familia, en especial a mis padres y mis hermanos, por apoyarme tanto en los buenos como en los malos ratos que me ha dado esta carrera.

A los amigos que he conocido en Ciudad Real, tanto dentro como fuera del ámbito académico, que a pesar de conocernos durante poco tiempo, ya son de las personas más importantes de mi vida. En especial a Paco, Jorge y Tafa, por ser con los que más tiempo y dolores de cabeza he compartido.

No puedo olvidar a todos mis amigos de *Oreto*, que siempre han estado ahí para cualquier cosa que he necesitado o simplemente para alegrarme el día. Gracias a Sergio, Paco, Vane, Nines, Javi, Gaby, Manu, Cayetano, Luis, José Carlos... y todos los demás.

Especial mención merece Carlos González Morcillo, por brindarme la idea y la oportunidad de realizar este proyecto, por su compromiso, trabajo y amistad. Profesional brillante y mejor persona.

Muchísimas gracias a todos. Estos cinco años no habrían sido lo mismo sin vosotros.

César Mora Castro.

CAPÍTULO 1

INTRODUCCIÓN

EL campo de la informática gráfica ha tenido, desde su nacimiento, un gran crecimiento y evolución debido al interés suscitado en industrias como la de los videojuegos, cine, televisión o educación. Las formas en la que se explota son muy variadas, aunque quizá una de las más innovadoras y que está cobrando especial relevancia en los últimos años es la composición de imagen sintética por computador con imagen obtenida del mundo real. Así es como surge el término de *Realidad Aumentada*.

La técnica de integrar objetos ficticios con imagen del mundo real no es nueva. En producciones cinematográficas es común introducir imágenes generadas por computador para conseguir espectaculares efectos especiales. La mayor parte de las películas actuales utilizan gráficos 3D por computador debido a su bajo coste y alto nivel de realismo conseguido con la tecnología actual. Sin embargo, como se verá a continuación, esta integración no puede considerarse *Realidad Aumentada*.

A continuación se definirá qué se entiende por Realidad Aumentada (RA), además de dar una visión tecnológica e histórica de este nuevo paradigma de interacción con el usuario.

1.1. Qué es Realidad Aumentada

Realidad Aumentada es, según Ronald T. Azuma [A+97], una *variación de los entornos virtuales*, en la cual el usuario percibe el mundo real con objetos virtuales integrados, dando la sensación de que están correctamente compuestos y alineados.

Para que una aplicación pueda ser considerada de Realidad Aumentada, debe cumplir las siguientes características definidas por Azuma:

- Debe combinar el mundo real con el virtual, es decir, el resultado debe mostrar imagen captada del mundo real e imagen sintetizada por computador.

Figura 1.1: Taxonomía de la Realidad Mixta.

- Debe ser *interactivo* en *tiempo real*. Esto significa que todo cálculo necesario debe ser realizado en un tiempo lo suficientemente pequeño como para dar la sensación que la composición de los objetos virtuales está sucediendo en *ese mismo momento*. Para realizar los efectos especiales de las producciones cinematográficas se graban imágenes del mundo real, se procesa la integración y, tras un largo intervalo de tiempo dedicado a postproducción, que pueden ser unas horas, se consigue la combinación del mundo real y el virtual.

- La alineación de los objetos sintéticos debe realizarse en el espacio 3D. De este modo, aplicaciones móviles como *Layar* no son estrictamente Realidad Aumentada, ya que la combinación se realiza sobre un plano virtual bidimensional.

El paradigma de la Realidad Aumentada es considerado como una especialización de la denominada Realidad Mixta (RM) [MK94]. El objetivo principal de la Realidad Mixta es conseguir integrar objetos del mundo real con elementos virtuales (ver Figura 1.1). Dependiendo del nivel de integración de objetos reales y virtuales se distinguen diferentes subcampos de la Realidad Mixta.

En los entornos completamente virtuales (Virtual Environments (VE)), el usuario únicamente percibe información generada por ordenador, sin tener realimentación visual del mundo real (ver Figura 1.1). El objetivo principal es conseguir una sensación de *inmersión* en el mundo real.

La Realidad Aumentada se encuentra a medio camino entre los entornos reales y los virtuales, tratando de integrar de una forma *real* e *interactiva* ambos mundos.

A continuación se definirán los elementos básicos necesarios en cualquier aplicación de Realidad Aumentada.

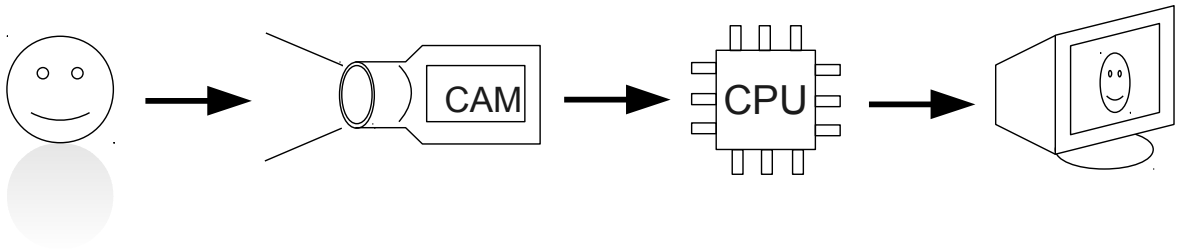

Figura 1.2: Elementos básicos de una aplicación de Realidad Aumentada.

1.2. Elementos Estructurales

En una aplicación de Realidad Aumentada son necesarios al menos tres elementos tecnológicos básicos, como se muestra en la Figura 1.2:

- **Dispositivo de vídeo:** es necesario para capturar información del mundo real, mediante la cual se resuelve el problema del *registro*, que se abordará en la Sección 1.5. Estos dispositivos pueden ser desde *webcams* integradas de bajo coste hasta cámaras de alta resolución.

- **Unidad de proceso:** Dependiendo de las necesidades de la aplicación, se pueden utilizar desde ordenadores de sobremesa, portátiles o tablets, hasta dispositivos móviles como *Smartphone's* o PDA's. Si se dispone de suficientes recursos hardware, se suele dividir el trabajo de cálculo entre la CPU (registro de la realidad, lógica de la aplicación) y la GPU (renderizado de los objetos virtuales).

- **Dispositivo de visualización:** la imagen resultado de la composición del mundo real y los objetos virtuales debe mostrarse a través de algún dispositivo de visualización. Existen dos opciones básicas, de las que se derivan una serie de alternativas concretas de representación. A continuación se muestra una breve descripción y comparativa de dichas opciones:

  - *Tecnología óptica:* esta tecnología, cuyo esquema puede verse en la Figura 1.3, se basa en la utilización de un *combinador* óptico situado en frente de los ojos del usuario. Este combinador es *transparente*, por lo que el usuario puede recibir información visual del mundo real, y es capaz de reflejar el mundo virtual proyectado por un monitor. De este modo el dispositivo óptico únicamente representa los datos sintetizados (ya que la imagen del mundo real se percibe por transparencia de las pantallas).

    El principal inconveniente de esta técnica es la reducción de luz que atraviesa la lente combinadora, por lo que las imágenes se oscurecen. El HDM descrito en [Hol92], por ejemplo, es capaz de transmitir al usuario únicamente el 30 % de luz que atraviesa la lente.

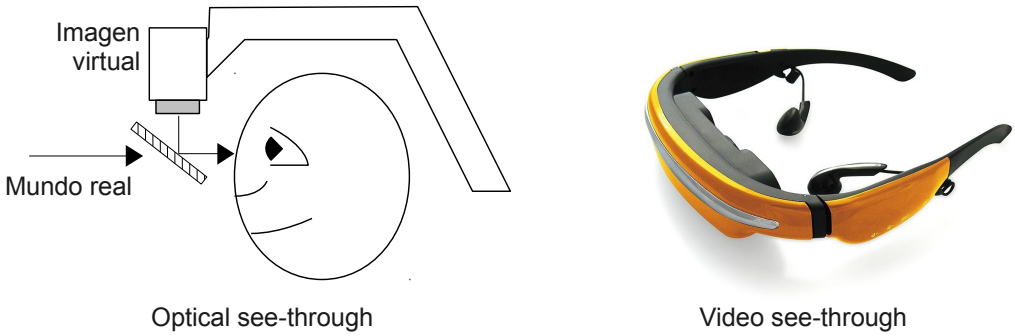

Figura 1.3: Esquema de sistema see-through y dispositivo HDM comercializado por *Microchip*.

- *Tecnología de vídeo:* por contraposición, la tecnología basada en vídeo combina la imagen percibida del mundo real a través de una videocámara con los objetos sintetizados mediante un computador. El resultado se visualiza utilizando monitores convencionales. Estos monitores representan la imagen capturada del mundo real y la imagen virtual superpuesta. En la Figura 1.3 se muestra el caso concreto de uso de monitores en un HDM, aunque uno de los usos más extendidos es emplear la pantalla de un computador convencional.

  La combinación de imagen real y objetos virtuales puede realizarse de numerosas formas. Una técnica muy extendida es el *chroma key*. Consiste en determinar un color para el fondo de las imágenes generadas por computador (este color no debe estar contenido en los objetos sintéticos). Este color se reemplaza en una siguiente fase por la imagen del mundo real. Se trata de una técnica muy eficiente en tiempo real usada en numerosos medios de comunicación.

El enorme potencial que ofrece la Realidad Aumentada hoy en día viene determinado por su desarrollo histórico durante las últimas dos décadas. En la siguiente sección se estudiarán diferentes hitos alcanzados a lo largo de la evolución de recursos hardware y técnicas software relacionados directamente con Realidad Aumentada.

1.3. Introducción Histórica

El primer sistema de Realidad Aumentada fue desarrollado por Ivan Sutherland en 1968, utilizando un prototipo de Head Mounted Display aún primitivo (ver Figura 1.4). A través de él podían verse objetos simples tridimensionales renderizados en *wireframe* en tiempo real, lo que supuso un gran logro hasta la fecha.

En 1992 Tom Caudell y David Mizell, ambos ingenieros de *Boeing*, acuñaron el término de Realidad Aumentada. Proponían el uso de esta novedosa tecnología para mejorar la ex-

Figura 1.4: De izquierda a derecha, primer sistema de Realidad Aumentada de Sutherland, aplicación funcionando con la biblioteca ARToolKit, y *Wikitude* ejecutándose en un dispositivo móvil.

periencia y proporcionar información adicional a los operadores humanos en las fábricas de aviones.

Años más tarde, en 1997, investigadores de la Universidad de Columbia presentan *The Touring Machine* [FMHW97]. Se trataba de un sistema de Realidad Aumentada móvil que utilizaba un dispositivo de visualización del tipo *optical see-through*, el cual componía los objetos tridimensionales usando una lente semi-transparente.

En 1999, Kato y Billinghurst desarrollan *ARToolKit* [KB99], una biblioteca de tracking visual con 6 grados de libertad que reconoce marcas cuadradas en blanco y negro. Su uso está hoy en día muy extendido gracias a su liberación bajo licencia GPL.

En 2003 se lanza al mercado *Mozzies*, un juego pionero de Realidad Aumentada para teléfonos móviles desarrollado por Siemens. El juego se basa en la superposición mosquitos al mundo real y el objetivo es tratar de aniquilarlos.

En 2007 se presenta en ISMAR el algoritmo PTAM [KD03], desarrollado por Klein y Murray. PTAM es un innovador método de *tracking* visual que no requiere marcas, desarrollado como una ampliación del algoritmo SLAM, consiguiendo unos resultados muy eficientes en *tiempo real* mediante la paralelización de los procesos de *mapping* y *tracking*.

En 2008 Mobilizy lanza *Wikitude*, una aplicación que compone información obtenida de Wikipedia sobre imagen del mundo real capturada con un dispositivo móvil. Actualmente está disponible para *Android* e *iPhone* entre otras plataformas.

Este breve resumen de hitos históricos trae asociado un fuerte impacto de implantación social y crecimiento económico de los que se estudiarán algunos indicadores en la siguiente sección.

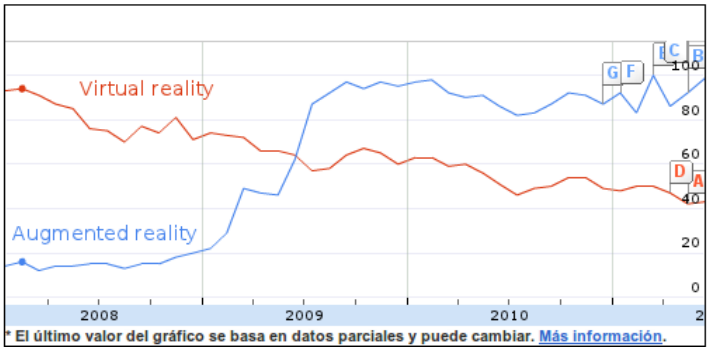

Figura 1.5: Estadísticas de búsqueda de las cadenas *Augmented Reality* y *Virtual Reality*. *Fuente:* Google.

1.4. Impacto Socio-Económico

La importancia de la Realidad Aumentada en la actualidad empieza a ser notable, y se augura que mantendrá su crecimiento durate los próximos años. Según previsiones de la consultora *Juniper Research* [jun], la Realidad Aumentada en dispositivos móviles generará más de 700 millones de dólares en 2014, con más de 350 millones de terminales capaces de ejecutar estas aplicaciones.

Consultando las estadísticas de búsquedas de *Google* (Figura 1.5), se puede observar cómo desde el año 2009 las búsquedas de la cadena *Augmented Reality* superaron a las de *Virtual Reality*, siguiendo la primera una tendencia al alza que se mantendrá durante los próximos años.

Las áreas en las que actualmente se aplica la Realidad Aumentada son muy numerosas, pudiendo destacar las siguientes:

- **Ocio:** en la industria del ocio supone un paradigma muy novedoso y atractivo para el desarrollo de videojuegos. *EyePet* [webi] es un juego comercializado por Sony para su plataforma Playstation 3, cuyo objetivo principal es criar una mascota inmersa en el mundo real mediante el uso de una marca. *Invizimals* [webk] es otro ejemplo de éxito para la misma plataforma desarrollado por la empresa española Novarama Technology S.L.

- **Seguridad:** una opción interesante es aplicar la Realidad Aumentada para complementar la información relacionada con la seguridad que se obtiene de un complejo (vídeo de cámaras remotas o logs diversos). En el proyecto Hesperia, se utilizaba la Realidad Aumentada como tecnología para mejorar la experiencia en labores de seguridad. En la universidad de Hokkaido [KT02] se realizaron igualmente investigaciones en esta materia. Estas investigaciones se centraron en aplicar la Realidad Aumentada a la monitorización en tiempo real de las instalaciones de un edificio.

- **Docencia:** la Realidad Aumentada puede ayudar durante el aprendizaje, haciéndolo de forma más amena e interactiva. También puede utilizarse para facilitar el entrenamiento de personal de mantenimiento en automóviles, fábricas, etc. En [KS03] se describe un sistema para la enseñanza de matemáticas ayudándose de la Realidad Aumentada, gracias a la cual se representan funciones y geometría en el espacio 3D.

- **Mantenimiento y reparación:** los mecánicos y operadores de maquinaria compleja pueden utilizar la Realidad Aumentada para llevar a cabo tareas de mantenimiento. El trabajador utilizaría un sistema en el que se superpondría información visual que indicase las instrucciones para realizar la tarea. BMW ha invertido en los últimos años en esta tecnología, siendo un ejemplo de empresa que emplea la Realidad Aumentada para mejorar la eficiencia de sus mecánicos [webn].

- **Medicina:** en trabajos que requieren una precisión y un cuidado tan delicado como puede ser una operación quirúrgica, la Realidad Aumentada puede añadir información adicional como la ubicación exacta para realizar el corte o las constantes vitales del paciente. Actualmente ya se han llevado a cabo investigaciones como las de la universidad de Carolina del Norte [R$^+$02].

1.5. Problemática

El desarrollo de aplicaciones de Realidad Aumentada requiere resolver problemas de diversa naturaleza. Por un lado, conseguir un correcto alineamiento de los objetos virtuales en la escena real necesita calcular con precisión el *Registro*.

Por otro lado, la diversidad de dispositivos hardware y tecnologías implicadas requiere el dominio de multitud de áreas de conocimiento tales como la visión por computador, la simulación física, síntesis de imagen 3D, animación por computador, geometría euclídea, etc. En el Capítulo de 2 se realizará una breve exposición de los aspectos más relevantes en los que se ha profundizado para la realización de este Proyecto de Fin de Carrera.

Esta heterogeneidad en términos de software y hardware, unidos a la necesidad de un profundo conocimiento de geometría computacional 3D, hace que el desarrollo de aplicaciones de Realidad Aumentada basadas en mecanismos avanzados de interacción esté condicionada a la disponibilidad de personal altamente cualificado.

Para mitigar estos problemas, se propone el desarrollo de este Proyecto de Fin de Carrera, Minerva, cuyo objetivo general puede ser descrito como la definición de un lenguaje de alto nivel para la especificación completa de una aplicación de Realidad Aumentada, así como la construcción de una paltaforma para su posterior ejecución. Este lenguaje debe permitir al menos, las siguientes características:

- Definir los elementos necesarios de funcionamiento de los métodos de tracking implementados (marcas o imágenes en el caso de métodos visuales).
- Añadir la componente virtual de la aplicación, como objetos tridimensionales primitivos y especificados en algún metaformato de descripción de geometría, o objetos bidimensionales como imágenes o texto.
- Especificación de la lógica de actuación de esos componentes basándose en los sistemas Sensor, Controlador, Actuador (SCA) (Sensores, Controladores y Actuadores) descritos en la Sección 2.1.1.
- Proporcionar acceso sencillo a fuentes de vídeo como webcams.
- Dar la posibilidad de añadir características multmedia como la reproducción de audio, o la detección de eventos por parte de periféricos de entrada como teclados o ratones.
- Incorporación de simulación física para la componente virtual permitiendo la detección de colisiones o el efecto de la fuerza de la gravedad.
- Añadir soporte para un lenguaje de script que extienda la funcionalidad del lenguaje de alto nivel de especificación, dando acceso directo a las características de la arquitectura de Minerva.
- El presente de Proyecto de Fin de Carrera debe basarse en una arquitectura *modular* y *extensible* para añadir características nuevas o mejorar las ya presentes.
- El sistema debe ser *multiplataforma*, pudiendo ejecutarse en al menos dos plataformas: *Microsoft Windows* y *GNU/Linux*.

Estos objetivos son descritos con mayor nivel de detalle en el Capítulo 3.

1.6. Estructura del documento

Este documento se ha estructurado según las indicaciones de la normativa de proyectos de fin de carrera de la Escuela Superior de Informática de la Universidad de Castilla-La Mancha, empleando los siguientes capítulos:

- **Capítulo 2. Antecedentes:** en este capítulo se hace un repaso a los conocimientos y áreas que ha sido necesario estudiar para el desarrollo de Minerva, junto a los principales sistemas existentes relacionados.
- **Capítulo 3. Objetivos:** en este capítulo se desglosa y se describen la lista de objetivos y subobjetivos planteados para este Proyecto de Fin de Carrera.

- **Capítulo 4. Método de trabajo:** en este capítulo se explica y se justifica la metodología escogida para el desarrollo de este sistema. Además se describen los recursos empleados, tanto hardware como software.

- **Capítulo 5. Arquitectura:** en este capítulo se describe el diseño e implementación del sistema, detallando los inconvenientes surgidos y las soluciones aportadas. El capítulo se centra en la clasificación de los módulos y submódulos que componen Minerva, describiendo desde un enfoque funcional hasta un nivel de detalle técnico cada uno de ellos.

- **Capítulo 6. Evolución y costes:** en este capítulo se describe la evolución del sistema durante su periodo de desarrollo, detallando las etapas e iteraciones realizadas. Igualmente se aporta información relacionada con el coste económico, el análisis de rendimiento (*profiling*), y una serie de comparativas con aplicaciones homólogas que no utilizan Minerva, además de una encuesta realizada sobre su uso.

- **Capítulo 7. Conclusiones y propuestas:** en este capítulo se hace un resumen a modo de conclusión del desarrollo y las metas alcanzadas. También se enumera una serie de líneas de trabajo futuro planteadas para continuar su desarrollo, y una conclusión perosonal.

CAPÍTULO 2

ANTECEDENTES

LA realización de los objetivos propuestos en Minerva requieren un conocimiento específico en diversas áreas de la informática, tales como procesadores de lenguajes, visión por computador, representación 3D o la simulación física.

El objetivo de este capítulo es resumir los principales conceptos estudiados y afianzados en la realización del Proyecto Fin de Carrera. La novedosa aproximación de Minerva al desarrollo de aplicaciones de Realidad Aumentada lleva asociada la falta de plataformas y sistemas similares para su comparación. De este modo, el estudio del estado del arte se realizará atendiendo a determinados aspectos funcionales de la arquitectura tales como la simulación física, lenguajes de alto nivel, etc. La exposición de los antecedentes del capítulo se realiza agrupando áreas temáticas. El mapa conceptual de la Figura 2.1 describe la estructura de las secciones y subsecciones del presente capítulo. Así, en la sección de Plataformas y *Frameworks* se estudiarán distintas aplicaciones realizadas anteriormente que proporcionan características relacionadas con la simulación física, lenguajes de alto nivel y *toolkits* para el desarrollo de Realidad Aumentada. En la sección de Lenguajes de Alto Nivel se analiza el funcionamiento de los procesadores de lenguajes, profundizando en conceptos como analizadores léxicos, sintácticos y semánticos, y el uso de lenguajes de *script* para ampliar la funcionalidad de un sistema. Para la sección de Informática Gráfica se realizará un repaso sobre la base matemática subyacente, las diferentes técnicas de representación 3D (relacionadas con iluminación, texturas, materiales, etc) y la implementación de simulación física. En la sección de Visión por Computador se muestran los conceptos que determinan el comportamiento de las lentes ópticas, y se estudiarán los diferentes tipos de métodos para realizar el *registro* de la realidad (*tracking*), dando ejemplos concretos de implementaciones. Por último, en la sección de Diseño Software se analizan las necesidades para la creación de código multiplataforma, y se detallan distintas bibliotecas auxiliares requeridas por los objetivos de Minerva.

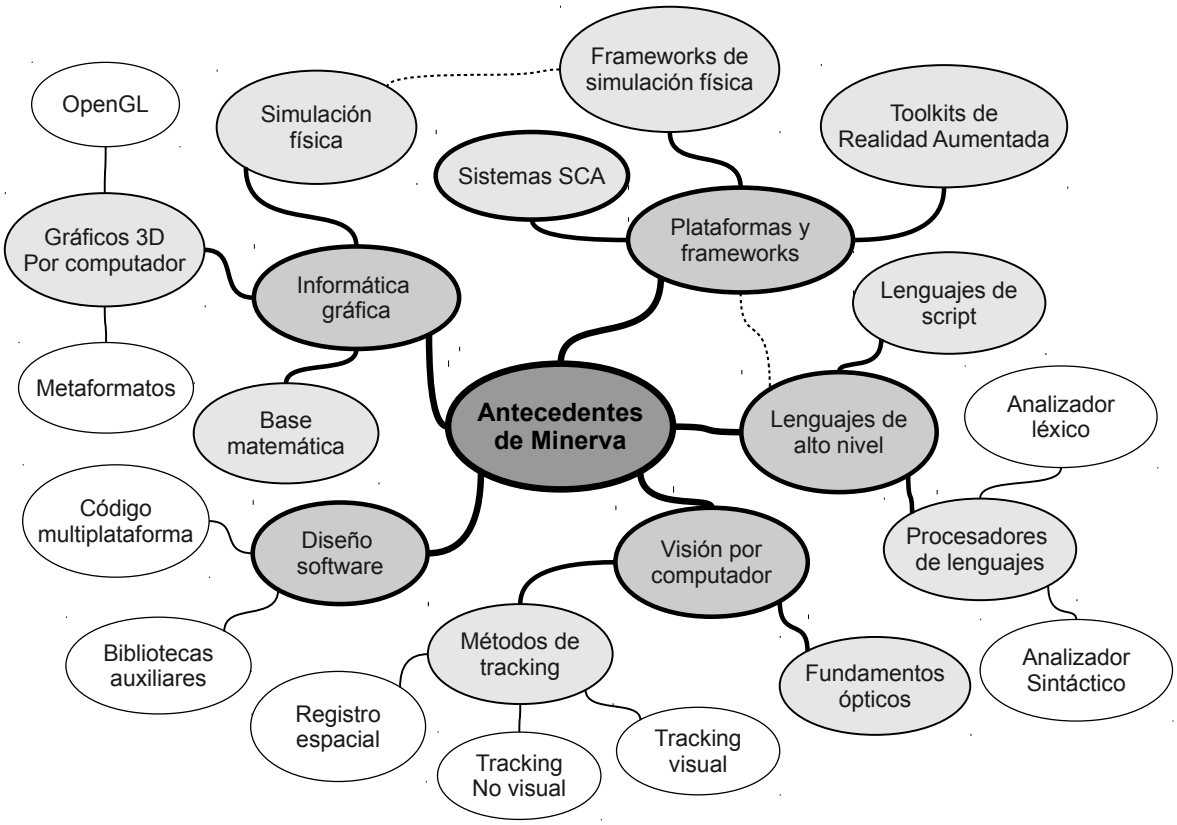

Figura 2.1: Mapa conceptual de los contenidos del presente capítulo.

En la Figura 2.1 se muestra un mapa conceptual de los contenidos descritos en este capítulo.

2.1. Plataformas y *frameworks*

En esta sección se hace un repaso sobre diferentes plataformas y *frameworks* desarrollados hasta la fecha, cuya funcionalidad está relacionada de alguna manera con el presente Proyecto de Fin de Carrera. No existe ninguna plataforma que pueda ser equiparada en todos los sentidos con Minerva, por lo que se describirán aquellas que implementen características como:

- Sistemas SCA.
- Simulación física.
- Lenguajes de alto nivel.
- *Frameworks* para el desarrollo de Realidad Aumentada.

2.1.1. Sistemas SCA

Minerva basa la especificación de la lógica de las aplicaciones de Realidad Aumentada en los sistemas SCA. Estos sistemas utilizan tres componentes básicos: *Sensores*, *Controladores* y *Actuadores*.. Sin embargo, este concepto ni es nuevo ni se reduce al ámbito del software.

Los sistemas SCA surgieron en el campo de la Ingeniería Industrial, para la especificación del comportamiento de *robots* en las cadenas de montaje, y se extendió a otras áreas. A continuación se describen ejemplos de aplicación:

- **Fabricación industrial:** en el área de la automatización industrial se utiliza una gran variedad de *robots* para la fabricación. El comportamiento de estos robots se especifica mediante *Sensores*, *Controladores* y *Actuadores*.

  Sirva como ejemplo un brazo mecánico que apila piezas de un determinado tipo en una torre a medida le llegan mediante una cinta transportadora. El robot dispone de un *Sensor* para detectar si le ha llegado una pieza nueva, un *Controlador* que le permite realizar la acción si se cumplen ciertas condiciones (por ejemplo, que el tamaño de la pieza sea el correcto), y un *Actuador* correspondiente al brazo con la capacidad de sujetar la pieza y trasladarla a la torre.

  La tendencia actual es el uso de sistemas de monitorización de un sistema SCA completo mediante Wi-Fi [LWE05].

- **Lego Mindstorms:** se trata de un producto fruto de la colaboración de la compañía Lego con el Massachusetts Institute of Technology (MIT) [webm]. Es un juego de robótica orientado a un público infantil compuesto por elementos básicos como motores, sensores de presión, luminosos o de temperatura, entre otros.

  Mediante un microcontrolador programable a través del ordenador, puede especificarse la lógica de comportamiento de los *Sensores* y *Actuadores*. Esta programación puede realizarse mediante un lenguaje visual (basado en *árboles de decisión*) o mediante lenguajes más avanzados como C o ensamblador.

- **Domótica:** este campo está cobrando mucha importancia en los últimos años. De una forma simplificada, se puede ver la demótica como la aplicación de los conocimientos adquiridos en la automatización industrial a un entorno más cotidiano. La mayoría de los dispositivos utilizados se sustentan también en sistemas SCA. Un ejemplo es una luz exterior que mediante un *sensor* luminoso decide (*controlador)* si debe estar encendida o apagada (*actuador*). En [JPV04]. se puede ver un estudio del estado actual de esta tecnología.

- **Blender GameKit:** la *suite* de modelado 3D *open source* por excelencia dispone de una herramienta para la creación de juegos profesionales basados en la especificación

de *Sensores, Controladores* y *Actuadores* denominado *Blender GameKit*[webg]. Esta herramienta proporciona una amplia variedad de estos elementos, así como la posibilidad de extender su funcionalidad mediante *scripting* con el lenguaje de programación Python. Un ejemplo de la potencia de este tipo de motores es *YoFrankie!*[webq], un juego *open source* realizado exclusivamente con *Blender*.

El proyecto Minerva se ha inspirado en esta implementación adaptándola a las necesidades de la Realidad Aumentada.

2.1.2. Frameworks de simulación física

Minerva proporciona una forma muy sencilla para permitir simulación física tridimensional a los elementos que componen la aplicación. Los objetos pueden estar sometidos a gravedad, colisiones e inercia de forma configurable.

A continuación se describe un ejemplo de plataforma que permite simular diferentes tipos de objetos sometidos a leyes físicas. Se trata de *Algodoo*.

Algodoo[webc] es una aplicación que permite realizar simulación física 2D mediante interacción visual. En ella el usuario puede dibujar directamente objetos y asociarle propiedades físicas (peso o puntos de bloqueo) y realizar la simulación. Proporciona elementos motrices básicos como poleas, motores, engranajes o cuerdas de transmisión. Es utilizado para ocio y para fines educativos.

Minerva permite realizar estas simulaciones en un espacio 3D en vez de 2D. El usuario puede determinar la fuerza, dirección y sentido de la gravedad o configurar teclas para aplicar impulsos a los objetos. La simulación física se puede utilizar para realizar aplicaciones educativas (explicación de las leyes físicas de forma interactiva) y de entretenimiento (destruir bloques o juegos de conducción).

2.1.3. Lenguajes de alto nivel

Minerva dispone de un lenguaje propio de alto nivel para la especificación de la lógica (*Minerva Specificacion Language - MSL*). Los lenguajes de alto nivel para realizar tareas complejas se han utilizado en diferentes campos a lo largo de los últimos años. Algunos ejemplos donde se utilizan son:

- **Asignatura de procesadores de lenguajes**: muchas universidades utilizan la implementación de un lenguaje propio con el fin de enseñar el funcionamiento de los procesadores de lenguajes a sus alumnos. La Escuela Superior de Informática de Ciudad Real [webl] propone en su asignatura *Procesadores de lenguajes* la realización de una práctica donde los alumnos creen un lenguaje de alto nivel para realizar una tarea avanzada (especificación de redes Ethernet, resolución de mallas eléctricas, generación de videojuegos, bibliotecas musicales, etc).

```
#sec:Main
REPEAT
  INPUT x
  IF x = 0 THEN
    EXIT LOOP
  ENDIF
  x0 = x
  f = 2
  p = 0
```

Figura 2.2: Ejemplo de código fuente en SmallBasic.

- **SmallBasic:** se trata de un lenguaje de alto nivel interpretado con una sintaxis muy sencilla orientada al prototipado rápido o para fines educativos para un público infantil [webo]. Proporciona funciones trigonométricas, matriciales y algebraicas, subsistemas de sonido y gráficos, además de su propio IDE.

  En la Figura 2.2 se puede ver un ejemplo mínimo de código en SmallBasic.

- **AGS:** *Adventure Game Studio* [webb] es un framework para realizar aventuras gráficas al más puro estilo de *Sierra* y *Lucasarts* de la década de los 90. Proporciona una interfaz gráfica para la especificación de escenarios, personajes y objetos. Además dispone de un lenguaje de alto nivel específico para la realización de scripts basado en Java/C#.

  Este concepto es muy similar al de Minerva, mientras que con AGS se realizan aventuras gráficas, con Minerva se consiguen aplicaciones de Realidad Aumentada.

2.1.4. *Toolkits* de Realidad Aumentada

En un plano más cercano al objetivo principal de Minerva, existen *toolkits* para la realización de aplicaciones de Realidad Aumentada. Estos *toolkits* se caracterizan por ofrecer características básicas, por lo que las aplicaciones desarrolladas no son de una alta complejidad. A continuación se describen los más importantes:

- **ARMedia Plugin:** [webd] es una extensión para el software de modelado tridimensional *Google Sketchup* [webj]. Con *Google Sketchup* se pueden realizar modelos tridimensionales sencillos, orientado a la realización de edificios para *Google Earth*. ARMedia Plugin permite utilizar dichos modelos para asociarlos a marcas del tipo *ARToolKit*.

- **Layar toolkits:** *Layar* es un navegador basado en Realidad Aumentada con el que poder superponer información contextual al lugar donde se encuentre el usuario con su

dispositivo móvil. Recientemente ha liberado la *API* para que terceros puedan realizar aplicaciones para su navegador. A esta *API* se le denomina *Layar Connect*. *Layar* vende su *toolkit* basado en el posicionamiento geográfico y en la interacción con su navegador. Sin embargo, estas aplicaciones no pasan de ser *apps* para un navegador, y no tienen entidad propia.

2.2. Lenguajes de alto nivel

La interfaz que Minerva ofrece a sus usuarios se basa en lenguajes de alto nivel: el lenguaje MSL y el lenguaje de *script*. En esta sección se explican los conocimientos básicos que han sido necesarios adquirir y dominar para el desarrollo del presente Proyecto de Fin de Carrera. Estos conocimientos se dividen en dos grupos principales:

- Procesadores de lenguajes.
- Soporte para lenguaje de *script*.

2.2.1. Procesadores de lenguajes

La teoría de lenguajes y gramáticas formales no fue tomada en serio hasta la década de 1950, cuando el lingüista norteamericano Avram Noam Chomsky revolucionó este área con *la teoría de las gramáticas transformacionales*. Esta teoría estableció las bases de la lingüística matemática y proporcionó una herramienta que facilitaría el estudio y la formalización de los lenguajes informáticos.

Los compiladores modernos basan su funcionamiento en conceptos como *gramáticas*, *teoría de autómatas* y *lenguajes formales*. A la hora de analizar un texto escrito para un compilador se suele dividir el trabajo en tres tipos de analizadores. En las siguientes secciones se estudiarán estos tres tipos de analizadores indicando la función de cada uno de ellos, y se describirán las bibliotecas utilizadas para su implementación en este proyecto. En [A+06] se puede encontrar un estudio más profundo sobre teoría de procesadores de lenguajes.

Analizador léxico

El analizador léxico, también conocido como *Scanner*, tiene como función el reconocimiento de *tokens*. Un *token* es una *palabra* perteneciente al lenguaje. Estos *tokens* se pueden indicar de forma explícita (por ejemplo, el token *while*) o definido mediante su expresión regular (un *identificador* es un *token* que empieza por letra, y el resto pueden ser caracteres alfanuméricos y el carácter guión bajo).

Los *tokens* reconocidos se pasan a los analizadores léxico y semántico. En caso de encon-

trar un *token* no válido el compilador lanzará un error léxico.

Analizador sintáctico

El analizador sintáctico, también conocido como *Parser*, es el corazón del compilador. Su objetivo es, partiendo de los *tokens* reconocidos por el analizador léxico, comprobar que la sintaxis es correcta. La sintaxis se refiera al *orden* de los *tokens* encontrados.

Existen dos formas básicas de implementar un analizador sintáctico: siguiendo una estrategia *descendente* (*top-down*, se parte del axioma inicial *S* y se realizan sucesivas derivaciones hasta llegar a la meta del análisis), o *ascendente* (*bottom-up*, se parte del objetivo *x* y se reconstruye en sentido inverso hasta llegar al símbolo inicial *S*).

La estrategia más común es la *descendente*, que se basa en la utilización de las gramáticas *LL(1)*.

Las características notables de una gramática *LL(1)* son:

- No existen dos reglas con la misma parte izquierda cuya parte derecha empiece por el mismo símbolo terminal.
- La parte derecha de todas las reglas empieza por un símbolo terminal, seguido opcionalmente por símbolos no terminales.
- Todas las producciones deben estar en *forma normal de Greibach*.

Analizador semántico

Esta es la fase en la que el compilador comprueba la corrección semántica del programa. Utiliza como entrada el *árbol* generado por el análisis sintáctico, y la salida es ese mismo árbol con notaciones semánticas. Algunas comprobaciones semánticas realiza este analizador son:

- Comprobar que todo identificador que se utilice haya sido previamente declarado.
- Asignar valor a las variables antes de usarlas.
- Los índices utilizados para acceder a vectores y *arrays* entran dentro de su rango.
- En las expresiones aritméticas, los operandos siguen correctamente las reglas sobre los tipos de datos permitidos por los operadores.
- Cuando se invoca un procedimiento o función, este ha sido declarado previamente de forma correcta.

Flex++ y Bison++

Es este apartado se describen brevemente las bibliotecas escogidas para la implementación de los analizadores léxico y sintáctico del lenguaje *MSL* de Minerva. Estas bibliotecas son *flex*++ y *Bison*++.

Flex++ es un generador de analizadores sintácticos, *open source* y multiplataforma. Su salida es normalmente redirigida a un analizador sintáctico y semántico, como puede ser el caso de *Bison* o *Yacc*. Se basa a su vez en *flex*, el generador de analizadores léxicos desarrollado originalmente para el lenguaje C.

Los *tokens* a reconocer en el lenguaje se especifican en un fichero con extensión *.l*.

Se ha escogido el uso de *flex*++ frente a *flex* ya que el desarrollo de Minerva se pretende realizar completamente en C++, como norma para mantener la coherencia y el orden del proyecto.

Bison++ es un generador de analizadores sintácticos y semánticos *open source* y multiplataforma. Suele ir acompañado por *flex* para la creación de compiladores e intérpretes. *Bison* convierte la descripción formal de un lenguaje, especificada en un fichero con extensión *.y* en forma de una gramática libre de contexto, en una clase C++ que realiza el análisis.

Bison mantiene compatibilidad con *Yacc*, para poder utilizar las especificaciones de uno con el otro indiferentemente. Richard Stallman colaboró para su implementación.

2.2.2. Lenguajes de *script*

El *scripting* es una técnica que consiste en utilizar un lenguaje de programación *interpretado* para proporcionar mecanismos avanzados para la especificación de la funcionalidad de una aplicación. Las principales ventaja de esta técnica es la sencillez del lenguaje de *script* y la ausencia de necesidad de compilación. Es necesario que el sistema al que se añade un lenguaje de *script* proporcione una interfaz (o *binding*) para poder acceder a su funcionalidad desde dicho lenguaje. Normalmente el lenguaje de programación con el que se desarrolla el sistema y el de *scripting* no tienen por qué ser el mismo.

Algunos lenguajes utilizados para añadir funcionalidad de scripting son *Lua* o *Python*. Para Minerva se escogió Python al tratarse de un lenguaje altamente extendido, con una sintaxis sencilla y *open source*.

Python es principalmente un lenguaje orientado a objetos, aunque también permite programación imperativa y funcional. Es interpretado, usa *tipado* dinámico y es multiplataforma. Fue desarrollado por Guido van Rossum, y la última versión estable es la 2.7. Actualmente se está trabajando para liberar la versión 3.

Boost-Python

Python proporciona una interfaz para poder utilizado desde lenguajes como C/C++ como lenguaje de *script*. Sin embargo, existe una biblioteca denominada *Boost-Python* que facilita en gran medida esta tarea. A continuación se describe esta biblioteca.

Boost es un conjunto de bibliotecas *open source* para extender la funcionalidad de C++. Está compuesta por multitud de sub-bibliotecas como por ejemplo:

- **Boost-Filesystem:** proporciona rutas portables, acceso a directorios y operaciones comunes sobre ficheros.
- **Boost-Interprocess:** pone a disposición del usuario mecanismos de memoria compartida, mútex compartidos, etc.
- **Boost-Math:** implementa numerosos algoritmos matemáticos para diferentes tareas.
- **Boost-Thread:** arquitectura multihilo portable para C++.

Una de las bibliotecas más interesante para el desarrollo de Minerva ha sido *Boost-Python*. Esta biblioteca permite ejecutar código Python desde C++, proporcionando un *intérprete* que puede comunicarse con funciones declaradas en C++. Facilita el manejo de punteros y el mapeo de estructuras de datos entre los dos lenguajes. En [weba] se encuentra la lista completa de las bibliotecas que componen *Boost*.

2.3. Informática gráfica

En esta sección se describen las áreas estudiadas relacionadas con la representación de gráficos por computador, desde las matemáticas subyacentes hasta técnicas más avanzadas. Estas áreas son:

- Base matemática
- Gráficos 3D por computador
- Simulación física

2.3.1. Base matemática

La informática gráfica tiene como base la rama de las matemáticas de la geometría espacial. Para la realización de Minerva se ha tenido que comprender conceptos como sistemas de coordenadas locales y globales, transformaciones espaciales y toda la base matemática asociada. En esta sección se hace un repaso de los conceptos más importantes que un usuario debe comprender a la hora de desarrollar aplicaciones con Minerva.

Los gráficos por computador tridimensionales se basan en un espacio euclídeo $\Re^3$, aunque la teoría se puede generalizar a espacios de cualquier dimension $\Re^n$.

Vectores espaciales

Un vector es, según [Lip92] son magnitudes que *tienen longitudes y direcciones apropiadas y parten de algún punto de referencia dado, O*.

Operaciones y propiedades básicas

Las operaciones básicas con vectores son:

- **Suma:** sean u y v vectores en $\Re^3$:

$$u = (u_1, u_2, u_3) y v = (v_1, v_2, v_3) \tag{2.1}$$

  La suma de u y v, escrito $u + v$, es el vector obtenido sumando las componentes correspondientes de éstos:

$$u + v = (u_1 + v_1, u_2 + v_2, u_3 + v_3) \tag{2.2}$$

- **Producto por un escalar:** el *producto* de un número real k por el vector u, escrito ku, es el vector obtenido multiplicando cada componente de u por k:

$$ku = (ku_1, ku_2, ku_3) \tag{2.3}$$

Nótese que la suma de dos vectores y la multiplicación de un escalar por un vector dan como resultado otro vector en $\Re^3$.

Se define, además:

$$-u = -1 u y u - v = u + (-v) \tag{2.4}$$

La suma de dos vectores pertenecientes a espacios vectoriales diferentes *no está definida*.

Producto escalar

Sean u y v vectores de $\Re^n$, se define el *producto escalar* o *interno*, denotado por $u \cdot v$, es el escalar obtenido multiplicando las componentes de los vectores una a una y sumando los productos resultantes:

$$u \cdot v = u_1 v_1 + u_2 v_2 + u_3 v_3 \tag{2.5}$$

Se dice que los vectores u y v son *ortogonales* (o *perpendiculares*) si su producto escalar es cero, esto es $u \cdot v = 0$.

Otra forma de calcular el producto escalar de dos vector u y v es la siguiente:

$$u \cdot v = |u||v| \cos(\theta) \tag{2.6}$$

Siendo θ el ángulo que forman los dos vectores.

Módulo de un vector

Sea u un vector en $\Re^3$, el *módulo* (*longitud*, o *norma*) del vector u, denotado por $|u|$, se define como la raíz cuadrada no negativa de $u \cdot u$:

$$|u| = \sqrt{u \cdot u} = \sqrt{u_1^2 + u_2^2 + u_3^2} \tag{2.7}$$

Intuitivamente, el módulo de un vector se ajusta a la longitud de la *flecha* del vector en la geometría euclídea.

Producto vectorial

Existe un producto especial denominado *producto vectorial* cuyo resultado es un vector, de ahí su nombre. Sean dos vectores u y v en $\Re^3$, denotado por $u \times v$, se define su producto vectorial de la siguiente forma:

$$\times v = (u_2 v_3 - u_3 v_2, u_3 v_1 - u_1 v_3, u_1 v_2 - u_2 v_1) \tag{2.8}$$

En notación de determinantes puede expresarse como sigue:

$$u \times v = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix} \tag{2.9}$$

Matrices

En el campo de la informática gráfica, las matrices se utilizan para representar *transformaciones* en el espacio. A continuación se describen las principales operaciones y propie-

dades matriciales, mientras que en la Sección 2.3.2 se detallaran en particular las *matrices de transformación*. Una matriz, según [Lip92], es *una tabla ordenada de escalares* a_ij de la forma:

$$\begin{pmatrix} a_11 & a_12 & ... & a_1n \\ a_21 & a_22 & ... & a_2n \\ ... & ... & ... & ... \\ a_m1 & a_m2 & ... & a_mn \end{pmatrix} \tag{2.10}$$

Propiedades básicas

Suma de matrices

Sean A y B dos matrices con el mismo tamaño (esto es, con el mismo número de filas y de columnas):

$$A = \begin{pmatrix} a_11 & a_12 & ... & a_1n \\ a_21 & a_22 & ... & a_2n \\ ... & ... & ... & ... \\ a_m1 & a_m2 & ... & a_mn \end{pmatrix} B = \begin{pmatrix} b_11 & b_12 & ... & b_1n \\ b_21 & b_22 & ... & b_2n \\ ... & ... & ... & ... \\ b_m1 & b_m2 & ... & b_mn \end{pmatrix} \tag{2.11}$$

Se define la suma de A y B, denotado como $A + B$, de la siguiente manera:

$$A + B = \begin{pmatrix} a_11 + b_11 & a_12 + b_12 & ... & a_b1n + b_1n \\ a_11 + b_21 & a_22 + b_22 & ... & a_2n + b_2n \\ ... & ... & ... & ... \\ a_m1 + b_m1 & a_m2 + b_m2 & ... & a_mn + b_mn \end{pmatrix} \tag{2.12}$$

Producto de un escalar por una matriz

El *producto* de un escalar k por una matriz A, denotado por $k \cdot A$, es la matriz obtenida multiplicando cada entrada de A por k:

$$A = \begin{pmatrix} ka_11 & ka_12 & ... & ka_1n \\ ka_21 & ka_22 & ... & ka_2n \\ ... & ... & ... & ... \\ ka_m1 & ka_m2 & ... & ka_mn \end{pmatrix} \tag{2.13}$$

Además, también se define:

$$-A = -1 \cdot AyA - B = A + (-B) \tag{2.14}$$

La suma de matrices de tamaños diferentes no está definida.

Producto de matrices

Se denota una fila cualquiera n de una matriz cualquiera A como A_n, y una columna cualquiera m como A^m.

El producto de dos matrices A y B, denotado por AB, se define como la matrix $m \times n$ cuyo elemento ij se obtiene multiplicando la fila i-ésima A_i de A por la columna j-ésima B^j de B:

$$AB = \begin{pmatrix} A_1B^1 & A_1B^2 & ... & A_1B^n \\ A_2B^1 & A_2B^2 & ... & A_2B^n \\ ... & ... & ... & ... \\ A_mB^1 & A_mB^2 & ... & A_mB^n \end{pmatrix} \tag{2.15}$$

El producto de matrices **no es conmutativo**.

Determinante de una matriz

Dada una matriz cuadrada A de dimensión $n \times n$ se define su determinante como la suma del producto de los elementos de una fila o columna cualquiera elegida por sus correspondientes adjuntos.

El siguiente ejemplo ilustra el cálculo del determinante de una matriz de 3×3:

$$A = \begin{pmatrix} -2 & 4 & 5 \\ 6 & 7 & -3 \\ 3 & 0 & 2 \end{pmatrix} \tag{2.16}$$

$$det(A) = 3 \begin{vmatrix} 4 & 5 \\ 7 & -3 \end{vmatrix} + 0 \left(- \begin{vmatrix} -2 & 5 \\ 6 & -3 \end{vmatrix} \right) + 2 \begin{vmatrix} -2 & 4 \\ 6 & 7 \end{vmatrix} = -217 \tag{2.17}$$

Existen métodos más específicos como *El método de Sarrus*, para calcular el determinante de una matriz de 3×3.

Matrices invertibles (no singulares)

En informática gráfica, la inversa de una matriz es una operación muy común y muy importante. La inversa de una matriz de transformación denota *la transformación inversa*.

Se dice que una matriz cuadrada A es *invertible* (o *no singular*) si existe una matriz B que cumpla la siguiente propiedad:

$AB = BA = I$

siendo I la matriz identidad. Tal matriz B es *única*.

No todas las matrices son invertibles. Existen métodos para comprobar la invertibilidad de una matriz. Sin embargo, las matrices de transformación utilizadas en informática gráfica (tipicamente de dimensiones 3×3 o 4×4) son siempre invertibles, por lo que no es necesario conocer tales métodos.

Existen múltiples formas de calcular la inversa de una matriz. El más general es calcularla *a través de la matriz de adjuntos*, mediante la siguiente fórmula:

$$A^{-1} = \frac{1}{|A|} Adj(A^t) \tag{2.18}$$

Siendo $|A|$ el determinante de la matriz, A^t la transpuesta de la matriz, y $Adj()$ su matriz de adjuntos.

2.3.2. Gráficos 3D por computador

A continuación se describen los conceptos básicos que se han utilizado para la representación de modelos tridimensionales. La sección comienza con una aproximación de cómo funciona de forma general una aplicación que sintetiza objetos tridimensionales, para luego detallar distintas técnicas a la hora de representar iluminación, textura y transformaciones entre otras.

El cauce de renderizado gráfico

Cuando se habla de *cauce de renderizado gráfico* se quiere referir al conjunto de acciones que se realizan a la hora de dibujar una determinada escena tridimensional. *Cauce* no es más que la traducción del término *pipeline*, que se refiere a una sucesión de etapas invariantes para realizar una determinada función.

Generalmente, una arquitectura de renderizado *en tiempo real* se subdivide en tres etapas, como se puede ver en la Figura 2.3: *aplicación, geometría* y *rasterizado* (*application, geometry* and *rasterizer*). Esta es la división de cualquier motor de renderizado. A su vez, cada etapa es en sí misma otro cauce con etapas específicas.

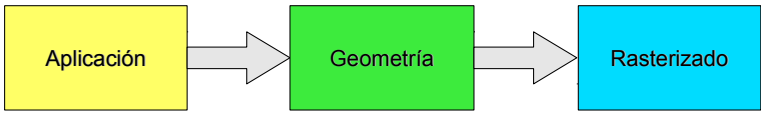

Figura 2.3: Etapas de un cauce típico de renderizado 3D.

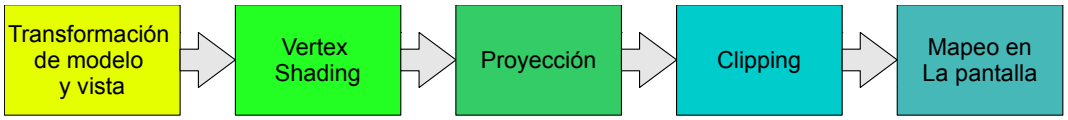

Figura 2.4: Subetapas de la etapa de Geometría.

Una de las características más atractivas de los cauces es la posibilidad de *paralelizar etapas de diferentes iteraciones*, por lo que el rendimiento se ve incrementado.

Etapa de aplicación

Esta es la etapa en la que el programador de la aplicación tiene *todo* el control. Típicamente se determina *qué* será hará aplicación, y de la forma en que lo haga afectará al rendimiento del resto de etapas y de la aplicación en general. Por ejemplo, un algoritmo que itere innecesariamente tendrá menos eficiencia que uno que lo haga de forma correcta.

En esta etapa se genera la información geométrica con la que se alimentará a la siguiente etapa, la geométrica. Esta información se da en forma de primitivas (puntos, líneas, triángulos). Cualquier algoritmo de *detección de colisiones* es implementado en esta etapa.

Etapa de geometría

Esta etapa es la principal responsable de la mayoría de las operaciones poligonales. Se divide a su vez en cinco subetapas:

- **Transformación de modelo y vista:** en primera instancia, la etapa geométrica parte de información de un modelo tridimensional (3D). Sin embargo, el resultado final se muestra por pantalla, que es bidimensional (2D). A cada uno de los vértices se les aplica transformaciones de modelo, lo que quiere decir que cada uno tiene su posición. En las sucesivas transformaciones pueden haber diferentes localizaciones y orientaciones para los elementos de la escena.

  El núcleo sólo dibuja los elementos que están comprendidos dentro del campo de visión de la cámara. La transformación de vista se utiliza para facilitar las etapas posteriores. Sitúa la cámara en el origen de coordenadas, y desplaza el resto del mundo para mantener la coherencia del modelo.

Figura 2.5: Cubo en proyección ortográfica (izquierda) y perspectiva (derecha).

- **Vertex shading:** para añadir realismo a la escena, no basta con dibujar la geometría y la localización de los objetos. Existe información adicional como el material o el efecto al impactar rayos de luz sobre él. La operación que determina el efecto de un material al recibir haces de luz se denomina *shading*. Esta operación se hace a nivel de vértice, y almacena información como la localización del vértice, su normal o el color.

- **Proyección:** la proyección determina *qué cantidad de volumen del espacio es capaz de observar la cámara*. Hay dos tipos básicos de proyecciones: *ortográfica* (o paralela) y perspectiva.

  La proyección ortográfica se caracteriza por que todas las líneas que parten del observador hacia cualquier punto del mundo *son paralelas*. Esta no es la proyección natural a la que está acostumbrado el ojo humano, pero permite apreciar la geometría tal y como es

  La proyección con perspectiva es más compleja. Se caracteriza por que los objetos más lejanos se aprecian más pequeños que los más cercanos (aunque sean del mismo tamaño). Es más afín a la proyección del ojo humano. Se representa como una pirámide cuya cúspide parte del ojo del observador. A esa pirámide se denomina *frustum* y de sus características depende el resultado final (ver Figura 2.6).

- **Clipping:** se trata de una técnica de optimización. Significa literalmente *recorte*. Se basa en la definición de un *near clip* y un *far clip*, que determinan un intervalo fuera del cual no se tendrá en cuenta ninguna geometría para renderizar. También consiste en hacer un *recorte de la geometría*. Cuando un objeto no aparece completo en la pantalla, se recorta su geometría (creando nuevos vértices si es necesario) para dibujar únicamente la parte visible del objeto.

- **Mapeo en la pantalla:** Sólo la geometría que entra dentro del *clipping* pasa a esta ultima subetapa. Las coordenadas del modelo son todavía tridimensionales. En esta sub-etapa se realiza la conversión a las *screen coordinates* (coordenadas de pantalla) bidimensionales. En este momento se sabe exactamente qué vértices se apreciarán por pantalla y en qué píxeles.

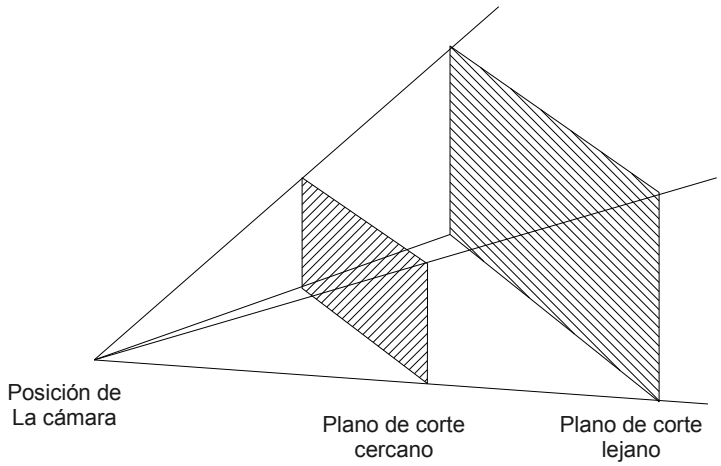

Figura 2.6: Representación gráfica del *frustum* de una lente.

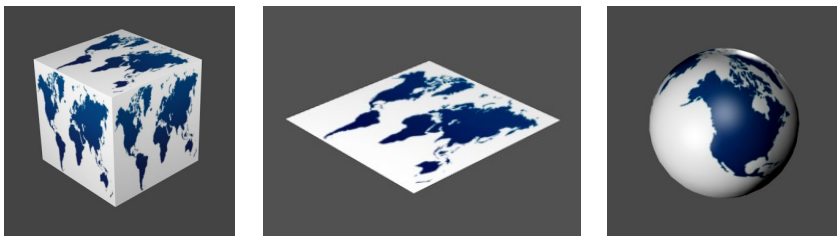

Figura 2.7: Diferentes funciones proyectoras de texturas.

Etapa de *rasterización*

Una vez que se dispone de los vértices transformados y proyectados en la pantalla y de su información de *shading* asociada, el último paso es calcular el color de cada uno de los píxeles. Este proceso se denomina *rasterización*.

Texturas

Según [AMHH08], el texturizado es el proceso que toma una superficie y modifica su apariencia en todos sus puntos usando una imagen, una función u otra fuente. Las texturas, además de definir el color del material puede añadir otros efectos como el brillo (*gloss effect*).

Las texturas suelen tratarse de imágenes bidimensionales que se aplican sobre un objeto tridimensional. El primer paso es hacer corresponder los píxeles de la textura en coordenadas 2D (u, v) con los vértices tridimensionales del objeto. Esto se hace mediante *la función proyectora* (*the projector function*). Esta función tiene numerosas implementaciones: esférica, cilíndrica, planar o natural entre otras.

Iluminación y sombras

En cuanto a las sombras, existe la opción de que los objetos las proyecten. Para ello se debe utilizar un punto hipotético para la fuente de luz. Existen multitud de métodos para el renderizado de sombras. En Minerva se implementan las denominadas *sombras planares* (*planar shadows*). Se trata en dibujar la geometría de nuevo pero *aplastada* contra el suelo y sin textura, dando la sensación de ser la propia sombra. Este tipo de métodos son más eficientes que los que se basan en lanzar rayos desde la fuente de luz.

Matrices de transformación

Una matriz de transformación es una matriz de dimensión 4×4 que almacena tres tipos de transformaciones:

1. Rotación.
2. Traslación.
3. Escala.

Una única matriz almacena los tres tipos de transformaciones simultáneamente.

Las transformaciones se pueden componer. Sea el siguiente ejemplo:

$$M_n ... M_3 M_2 M_1 V$$

Se realizarán sucesivamente las transformaciones sobre el vector V (también podría tratarse de un punto). Un detalle importante a tener en cuenta es el siguiente:

Si se aplican las transformaciones de derecha a izquierda, se realiza sobre el sistema local. Si se aplican de izquierda a derecha, se realiza sobre el sistema global.

La transformación resultante será la misma, pero se puede pensar localmente aplicándola de derecha a izquierda, o globalmente de izquierda a derecha. Para los principiantes en informática gráfica este suele ser uno de los mayores problemas a visualizar.

Metaformatos de modelos 3D

Los metaformatos de modelos 3D son formatos intermedios entre las bibliotecas gráficas y las aplicaciones de modelado. Estos formatos deben almacenar información como los vértices, caras y aristas del modelo, coordenadas *uv* para las texturas, animación o propiedades físicas. Existen numerosos formatos con diferentes características. A continuación se describen algunos de ellos:

- **Obj:** se trata de un formato sencillo y abierto adoptado por muchos desarrolladores de gráficos 3D. Se limita a representar la geometría, sin animación. Almacena la posición

```
#Esto es un comentario
v 1.230 2.340 3.450 #Esto es un vertice
vt 0.500 0.60 #Esto es una coordenada uv
vn 0.700 0.000 0.700 #Esto es un vector normal
```

Figura 2.8: Ejemplo de formato OreJ.

de cada vértice, las coordenadas *uv*, las normales y las caras de cada polígono. La extensión de los archivos es *.obj*

En la Figura 2.8 se puede ver un ejemplo del formato OreJ.

- **Collada:** es un formato creado inicialmente por Sony y actualmente mantenido por el grupo Khronos. Es un estándar abierto basado en Extensible Markup Language (XML) cuya extensión de archivos es *.dae*. Se caracteriza por ser un formato muy completo. A partir de la versión 1.4 del mismo se incluyó soporte para información relacionada con la simulación física. Algunas aplicaciones que lo utilizan son *Second Life*, *OpenSimulator* y *Google Earth*. La última versión liberada es la 1.5.

- **OreJ:** este formato fue creado por el grupo *Oreto* de la Escuela Superior de Informática de Ciudad Real. Está baso en el formato *Obj*, añadiendo soporte para animaciones simples. Actualmente existe un exportador escrito en Python para la suite de modelado Blender. El importador es una clase implementada en C++, que proporciona funciones para la utilización del modelo 3D.

  Minerva utiliza una implementación modificada de este formato para soportar información relacionada con la simulación física, como el cálculo de las formas de colisión. Estas formas pueden ser esféricas, cónicas, cilíndricas y de malla.

OpenGL

OpenGL (*Open Graphics Library*) es una API multiplataforma soportada por numerosos lenguajes para la producción de gráficos 2D y 3D [S$^+$05]. Fue desarrollada inicialmente por *Silicon Graphics*. Está compuesta por más de 700 funciones, de las cuales 670 son específicas de la plataforma y unas 50 de la *OpenGL Utility Library*.

Los nombres de las funciones de OpenGL siguen un convenio de nombrado para dar información sobre su funcionamiento.

Todas las funciones de OpenGL empiezan con el prefijo *gl*. A continuación viene el nombre de la función, y por último un sufijo que da información sobre el número y el tipo de argumentos que admite esa función. Por ejemplo, la siguiente función:

```
glColor3fv(color_array);
```

- Empieza conel prefijo *gl*.
- Su nombre es *Color*. Establece el color con el que se realizarán las siguientes funciones.
- Admite 3 valores, de tipo flotante (*f*), en forma de vector (*v*).

Por supuesto existen más combinaciones de sufijos en función de los tipos de los argumentos y el número.

OpenGL actúa como una máquina de estados. El programador lo configura en un estado con unas características (color, iluminación, etc) y no cambiarán hasta que se pase a otro estado.

Esta máquina de estados se basa en las *variables de estado* y otro tipo de información que determinan el color actual, las transformaciones de vista y de modelo, la proyección, características lumínicas o propiedades de los materiales.

Algunos estados hay que declararlos explícitamente si están o no desactivados con:

```
glEnable()
glDisable()
```

Otras bibliotecas gráficas

Existen muchas otras bibliotecas gráficas. Una de ellas es *Direct3D*, desarrollada por Microsoft, disponible para plataformas Windows, Xbox y Xbox360. No es libre ni está disponible para plataformas GNU/Linux. Otro inconveniente por el que no se ha escogido es que necesita más código para realizar la misma tarea que en OpenGL.

Minerva ha tenido como objetivo principal desde su nacimiento el ser *open source* y *multiplataforma*. Por ello la mejor elección como biblioteca gráfica ha sido *OpenGL*, pues las otras alternativas no cumplían con los requisitos necesarios.

2.3.3. Simulación física

En el desarrollo de gráficos 3D puede ser interesante que los objetos sintetizados se comporten bajo el influjo de las leyes físicas como la fuerza de la gravedad o colisiones con otros objetos. Existen bibliotecas que implementan con mayor o menor precisión este tipo de simulación, que puede además ser en *tiempo real*.

Una biblioteca de simulación física lleva a cabo estas tareas:

- Detección de colisiones.
- Dinámica de cuerpos rígidos (*rigid bodies)*.
- Soporte para cuerpos no rígidos (*soft bodies)* como prendas de ropa.

A continuación se describe *Bullet*, que es la biblioteca utilizada por Minerva y la suite de Blender.

Bullet Physics Library

Bullet Physics Library es una biblioteca profesional *open source* para la detección de colisiones y dinámica de cuerpos rígidos y no rígidos [webh]. Está disponible bajo la licencia *Zlib*.

Sus características principales son:

- Es *open source* y multiplataforma incluyendo PlayStation 3, XBox 360, Wii, PC, Linux, Mac OSX e iPhone.
- Detección de colisiones discreta y continua. Dispone de formas de colisión primitivas (esferas, cilindros) y mallas de triángulos cóncavas y convexas.
- Solucionador de restricciones de dinámica de cuerpos rígidos *rápida* y *estable*. Dinámica de vehículos, controladores de personajes.
- Dinámica para cuerpos no rígidos como ropa, cuerdas y volúmenes deformables.
- Soporte con *Maya Dynamica*, integración con Blender, soporte para importar/exportar propiedades físicas del metaformato *Collada*.

2.4. Visión por computador

El área de la visión por computador está muy relacionada con el objetivo de este Proyecto de Fin de Carrera y con la Realidad Aumentada en general. Los conocimientos adquiridos de esta área se dividen en dos grupos, y se describen en las siguientes secciones:

- Fundamentos ópticos
- Métodos de *tracking*

2.4.1. Fundamentos ópticos

A la hora de realizar aplicaciones de Realidad Aumentada, es necesario conocer cómo funcionan las lentes de los dispositivos de vídeo y cuáles son sus parámetros más importantes. A continuación se describen dichos parámetros y lo que se entiende por *calibrado* de una cámara.

Parámetros intrínsecos de las ópticas

El objetivo de la óptica de una cámara es captar los rayos luminosos para proyectarlos sobre el sensor óptico y poder recoger la información visual. Dependiendo de las características de la óptica la realidad se captará una forma u otra, lo cual es importante a la hora de representar los modelos tridimensionales para dar la sensación de pertenencia al mundo real.

Como se puede observar en la Figura 2.9, cuando los rayos pasan a través de una lente convexa, convergen hacia un punto denominado *punto focal*. La *distancia focal* es la distancia entre ese punto y el eje de la lente. Esta distancia es el parámetro principal a la hora de calcular la posición y el tamaño de los objetos en la imagen.

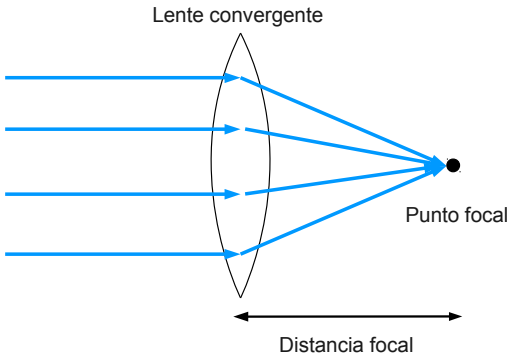

Figura 2.9: Esquema de una lente convexa.

Otro parámetro importante es el diámetro D del diafragma, que determinará la potencia lumínica con la que se ataca el sensor. De este se desprende otro parámetro, el número F, que indica la relación entre la distancia focal y el diámetro del diafragma:

$$F = \frac{f}{D} \tag{2.19}$$

Calibración

El objetivo de los algoritmos de *calibración* es la obtención, de forma empírica, de parámetros asociados a una cámara como la distancia focal y los coeficientes de distorsión. Estos parámetros se dividen en dos grupos:

1. **Parámetros intrínsecos:** son los que tienen que ver con la óptica y la cámara (centro de la imagen, distancia focal, distorsión y tamaño de los píxeles).

2. **Parámetros extrínsecos:** se refieren a la posición y orientación de la cámara respecto del mundo.

La idea subyacente a todos los métodos de calibración es la misma. Se trata de colocar objetos en posiciones perfectamente conocidas y calcular los valores de las proyecciones reales sobre la imagen obtenida.

Uno de los métodos más utilizados según [Esc01] es el *método de Tsai*. Con él se pueden obtener la distancia focal f y el coeficiente de distorsión radial K.

2.4.2. Métodos de *tracking*

La principal diferencia entre Realidad Virtual y Realidad Aumentada, reside en que la segunda debe *registrar* (*track*) el mundo real para adaptarse a él, mientras que en la primera este problema no existe. En los siguientes apartados se esboza una visión general sobre la problemática del *tracking* y los tipos generales que existen de cada uno de ellos, indicando ventajas e inconvenientes.

Problemática del tracking

La problemática del *tracking* consiste en, dada una fuente de vídeo, obtener la posición, orientación y movimiento del observador dentro del mundo real. Conocido esto, se puede dibujar un mundo virtual superpuesto sobre el mundo real, dando la sensación de ser un todo. Este es el punto de partida de la Realidad Aumentada.

Como se dijo en la Sección 1.1, para que una aplicación pueda considerarse de Realidad Aumentada debe cumplir tres características:

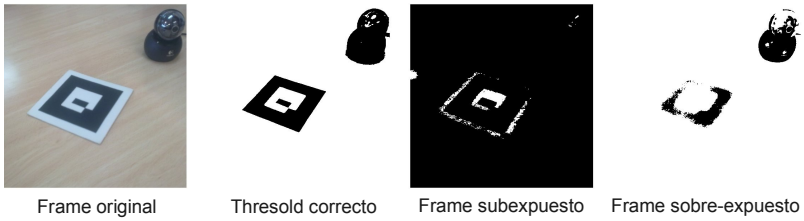

Figura 2.10: Distintos resultados de binarización dependiendo de la iluminación.

1. Debe combinar el mundo real con el virtual.
2. Debe ser interactivo y en tiempo real.
3. La integración de los objetos sintetizados en el espacio tridimensional debe ser lo más real posible.

De estas características extraemos que en un método de tracking tan importante es la precisión, como su *rendimiento*. El método debe ser lo suficientemente rápido como para que la aplicación pueda considerarse de tiempo real.

La iluminación de la escena real juega un papel muy importante a la hora de realizar el *registro*. En entornos donde sea posible, siempre es mejor adaptar la iluminación del entorno a un algoritmo que realizar la tarea inversa.

Muchos métodos de *tracking* implementan etapas en las que convierte las imágenes a escala de grises, y las *binariza* mediante un factor determinado. Este factor puede ser conveniente en la mayoría de los casos, pero si la iluminación es demasiado fuerte, o por el contrario demasiado débil, puede que el algoritmo del método de tracking no funcione.

Para solventar este problema se podría crear un *umbral adaptativo*, donde analice la luminosidad de la escena antes de *binarizarla*.

La Figura 2.10 muestra los resultados de la *binarización* en función de la iluminación de la escena.

Existen implementaciones de métodos de *tracking* muchos tipos: visuales, no visuales, basadas en marcas, absolutos, relativos, etc. En los siguientes apartados se describen algunos ejemplos de soluciones a la problemática del *registro*.

Tracking visual

Este tipo de *tracking* se basa en el análisis de una fuente de vídeo para llevar a cabo el *registro* del mundo real. Son los más económicos y accesibles porque no necesitan de ningún recurso hardware adicional, a parte de una cámara. Algunas implementaciones de este tipo de tracking son:

- **ARToolKit:** [webe] este método de tracking se basa en el reconocimiento de marcas cuadradas en blanco y negro. Es muy eficiente y preciso, aunque tiene como inconveniente el uso de las marcas. Es multiplataforma, y está escrito en C. Este es el método de tracking incorporado en Minerva, aunque la arquitectura está diseñada para soportar más de uno.

- **BazAR:** [webf] otro método visual parecido a *ARToolKit*, con la diferencia de que en vez de utilizar marcas, se utilizan imágenes pre-entrenadas. Se puede utilizar cualquier imagen (por ejemplo un cuadro), crear un archivo descriptor para que *BazAR* la pueda detectar y realizar el tracking. Es menos eficiente que *ARToolKit* pero tiene el añadido de no necesitar marcas externas.

- **Ptam:** [KD03] se trata de un método desarrollado por George Klein en la universidad de Oxford. Se basa en el funcionamiento de los *camera trackers* tradicionales, optimizado para funcionar en tiempo real. Según su página web, no necesita marcas, mapas preparados, plantillas conocidas o sensores inerciales.

Tracking no visual

Este tipo de métodos de tracking suelen estar basados en dispositivos hardware externos. Son muy precisos aunque más costosos. Algunos ejemplos de dispositivos que se utilizan para realizar *tracking* no visual son:

- **Giróscopos:** estos dispositivos miden, de forma absoluta, la orientación.

- **Acelerómetros:** estos dispositivos miden la aceleración del movimiento en las tres dimensiones.

Estos dispositivos suelen utilizarse en conjunto con métodos de tracking visuales para aumentar la eficiencia y la precisión del tracking.

2.4.3. Registro espacial

Otra clasificación de los métodos de tracking es dividirlos en métodos relativos y absolutos.

Los *métodos absolutos* son aquellos que no necesitan información anterior para localizarse en el espacio. Con la información actual es suficiente. Son ejemplos de ellos *ARToolKit* y *BazAR*.

Los *métodos relativos* sí necesitan ir almacenando información de instantes anteriores para poder localizarse. Deben llevar un histórico sobre las posiciones que ha pasado el usuario antes de llegar al momento actual. Algunos ejemplos son los acelerómetros o *Ptam*.

2.5. Diseño software

En esta sección se describen las nociones relacionadas con el diseño y la Ingeniería del Software, requeridos por objetivos del sistema Minerva como la portabilidad de la arquitectura. En los siguientes apartados se describirán las siguientes áreas:

- Código multiplataforma
- Bibliotecas auxiliares

2.5.1. Código multiplataforma

Uno de los objetivos primordiales de este Proyecto Fin de Carrera ha sido la *portabilidad*. Dado que pretende ser un sistema que sea fácil de usar, es interesante que pueda llegar al máximo de usuarios posible, y salvar inconvenientes como la plataforma en la que se ejecute es un punto crítico.

Se han tenido que adquirir conocimientos en la escritura de código portable, ya que diferentes compiladores de un mismo lenguaje no suelen comportarse exactamente de la misma forma. En [webp] se pueden encontrar algunos consejos sobre la escritura de código portable en C++ fruto de la experiencia de la *Mozilla Fundation*.

Esta es una de las razones por las que se ha tenido mucho cuidado en la elección de las bibliotecas. Era necesario que fuesen multiplataforma y, dada la naturaleza de Minerva, también *open source*.

GNU/Linux

GNU/Linux ha sido la principal plataforma de desarrollo del proyecto. Se ha querido facilitar su compilación y ejecución utilizando únicamente versiones de bibliotecas estables y que se encuentren en la mayoría de las distribuciones GNU/Linux.

Con el código fuente se adjunta el fichero *Makefile* que se encarga de la compilación completa del proyecto con tan solo ejecutar un comando.

Minerva está disponible para plataformas GNU/Linux mediante un paquete *.deb*.

Microsoft Windows

La compilación en sistemas Windows se realizó una vez terminado el desarrollo para sistemas GNU/Linux. Se eligió como compilador *Visual Studio 2008*, dado que es *gratuito*. Hubo que utilizar compilación condicional para acceder a funcionalidades relacionadas con los sistemas de ficheros. El resto del trabajo fue encontrar las versiones para esta plataforma de las bibliotecas utilizadas, y enlazarlas adecuadamente.

Minerva está disponible para plataformas Windows mediante un sencillo instalador.

2.5.2. Bibliotecas auxiliares

Además de todas las áreas mencionadas anteriormente que han debido de ser estudiadas para el desarrollo del presente Proyecto de Fin de Carrera, ha sido necesario la utilización de otras bibliotecas auxiliares que se describen a continuación.

SDL

SDL (*Simple DirectMedia Layer*) es un conjunto de bibliotecas desarrolladas para C que proporciona funcionalidad básica para el dibujado en 2D, gestión de ventanas, detección de eventos de entrada (teclado, ratón o joystick) y hasta reproducción de sonido. Es una biblioteca *open source* y multiplataforma.

Se ha escogido por su funcionalidad a la hora de detectar eventos de teclado e inicialización y gestión de las ventanas. Además proporciona funciones para la carga de archivos de imágenes que soporta un variado repertorio de formatos.

OpenCV

OpenCV (*Open Source Computer Vision*) es una biblioteca para visión por computador en tiempo real. Esta liberada bajo licencia *BSD* y es multiplataforma. Originalmente fue escrita en C aunque la última versión ha sido desarrollada completamente para C++. Implementa más de 2000 algoritmos optimizados para el tratamiento de imágenes y visión artificial como detección de contornos o de puntos de interés.

Es interesante gracias a que proporciona una implementación de entidades matriciales con un amplio conjunto de operaciones muy optimizadas. Para el tratamiento de transformaciones geométricas en informática gráfica es de mucha utilidad.

También dispone de una interfaz para capturar vídeo de distintas fuentes de vídeo (*clips* o *webcams*). Esta es la forma en la que Minerva accede a dispositivos de vídeo.

CAPÍTULO 3

OBJETIVOS

EL objetivo principal de Minerva es el desarrollo de un lenguaje de descripción de alto nivel para la definición de la lógica de aplicaciones de Realidad Aumentada, así como un framework completo para su posterior interpretación y ejecución.

A partir de este objetivo principal se definen una serie de subobjetivos funcionales específicos que detallan el amplio alcance de Minerva:

- El lenguaje de descripción *MSL* (**M**inerva **S**pecification **L**anguage) estará basado en los componentes básicos de los sistemas SCA: *Sensores, Controladores* y *Actuadores*. Estos sistemas son ampliamente utilizados en distintas áreas industriales. Gracias a esta amplia difusión se garantiza el potencial y la facilidad de uso para el usuario sin conocimientos avanzados en programación.

- Independencia del método de *registro*. Minerva abstrae al desarrollador de las particularidades del método de *tracking* utilizado.

- Módulo de representación 3D con independencia de la plataforma. El desarrollo de mecanismos de representación de alto nivel multiplataforma facilitan su futura portabilidad a nivel tanto hardware como software. El diseño de mecanismos avanzados de despliegue (como cálculo de sombras, carga de modelos con animación, etc) facilitan al programador la creación de aplicaciones visualmente eficaces, y le permite centrarse en su lógica.

- Soporte para la definición de entidades físicas. El lenguaje MSL y el framework de ejecución soporta la definición y simulación de entidades y leyes físicas, así como los mecanismos de interacción básicos (gravedad o colisiones de cuerpos rígidos). Para ello se ofrece una interfaz para la definición de propiedades de los objetos como masa, velocidad y formas de colisión.

- Soporte para lenguajes de script para el desarrollo a bajo nivel. Minerva soporta el acceso a sus métodos principales mediante un lenguaje de script, para facilitar el desarrollo de aplicaciones más complejas a programadores expertos.

- Biblioteca de Transformaciones Geométricas de alto nivel. El sistema SCA integrado en el presente Proyecto de Fin de Carrera permite la definición de sistemas de coordenadas para permitir transformaciones relativas a ellos. La biblioteca de transformaciones permite el desarrollo de componentes básicos de interacción empleando registro basado en marcas.

- Portabilidad. El desarrollo de Minerva se ha realizado siguiendo estándares y tecnologías libres, multiplataforma y consolidados, con el objetivo de que pueda ser portado al mayor número de plataformas posibles. Minerva podrá ser ejecutado al menos en dos plataformas: GNU/Linux y Microsoft Windows.

- Facilidad de uso. Ante todo, Minerva es un sistema orientado a usuarios con un bajo conocimiento en programación, por lo que una gran parte de los esfuerzos se han centrado en facilitar el trabajo. La creación de un buen manual de usuario y ofrecer la mayor información y soporte posible son objetivos primordiales.

CAPÍTULO 4

MÉTODO DE TRABAJO

A LO largo de este capítulo se describe la metodología utilizada durante el desarrollo de Minerva. En el Capítulo 6 se describen las iteraciones de este desarrollo, junto a la complejidad y resultados de cada una. Además se listan las herramientas, tanto hardware como software, utilizadas, detallando la versión de compilación.

4.1. Metodología de trabajo

La metodología de desarrollo escogida, dada la naturaleza de Minerva, seguía un modelo *iterativo* e *incremental*. Al tratarse de una arquitectura modular, se dividió la implementación de cada componente en iteraciones, y al final de cada una de ellas podía entregarse un prototipo funcional. El análisis de requisitos descrito en el Capítulo 6 se llevó a cabo en la etapa *Análisis del sistema* de la Figura 4.1.

En la Figura 4.1 se puede apreciar el esquema de un modelo de metodología como la escogida.

Esta metodología se aplica una estrategia basada en pequeñas iteraciones donde todas las fases se completan sobre cada módulo, permitiendo generar un prototipo que valide los requisitos iniciales. De este modo, es posible definir nuevos requisitos con el director del proyecto, que serán desarrollados en una nueva iteración incremental en el sistema.

El concepto inicial del software y el diseño de la arquitectura se definen igualmente empleando un enfoque en cascada, que finalizarán con el desarrollo del producto final. Así, este enfoque se adaptaba perfectamente a la especificación de requisitos dinámicos empleados en la definición del Proyecto de Fin de Carrera.

4.2. Herramientas

A continuación se enumeran y describen los recursos hardware y software utilizados durante el desarrollo, indicando las versiones específicas empleadas en la construcción de la versión de Minerva adjunta en el CD que acompaña a la presente memoria.

4.2.1. Lenguajes

Para la realización de Minerva se han utilizado diversos lenguajes de programación, que se listan a continuación:

- **C++:** es el lenguaje utilizado principalmente en el desarrollo del proyecto (ver Sección 6.2).
- **Flex:** es el lenguaje de especificación utilizado para el analizador léxico.
- **Bison:** es el lenguaje de especificación utilizado para el analizador sintáctico.
- **Python:** es el lenguaje utilizado para realizar *scripting*. La versión adjunta en el CD está compilada con soporte 2.7, aunque durante el desarrollo del proyecto se han realizado pruebas satisfactorias empleando la versión 2.6.

4.2.2. Hardware

La mayor parte del desarrollo de Minerva se ha realizado en un *Sony VAIO VGN-C1Z*, procesador Intel Core 2 T5500 1.66 GHz, 1 GB de memoria RAM, propiedad del laboratorio Oreto.

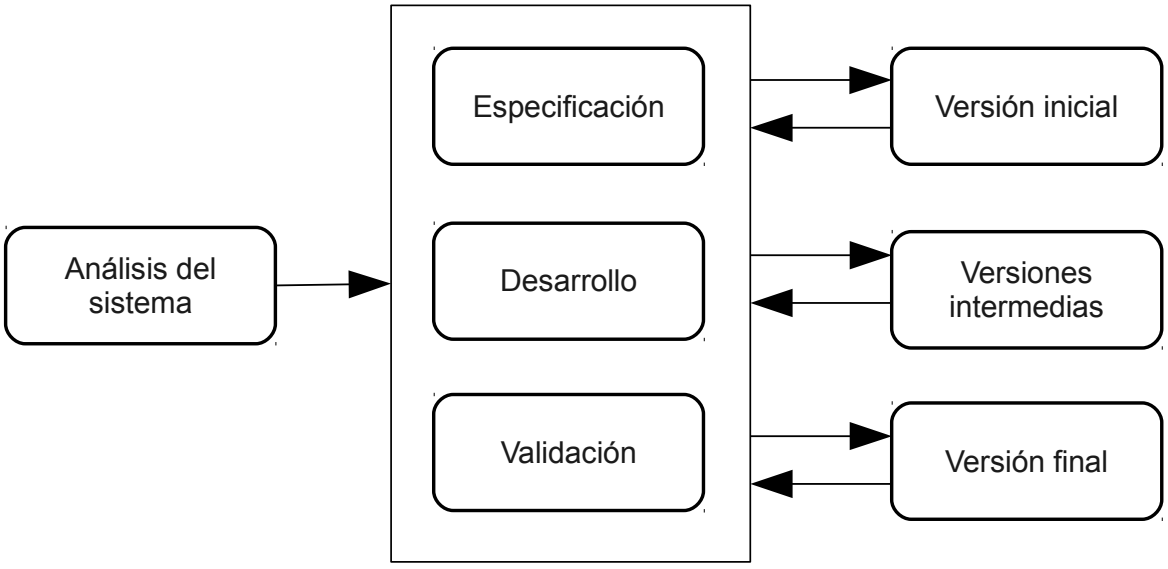

Figura 4.1: Esquema de una metodología iterativa e incremental.

Como dispositivo de vídeo se disponía de varias cámaras *Logitech Sphere AF* (óptica Carl Zeiss, 2 Megapíxeles, con enfoque automatizado), con plena compatibilidad para plataformas *GNU/Linux* y *Microsoft Windows*.

Para el control de versiones se disponía del servidor DEVORETO.ESI.UCLM.ES de la Escuela Superior de Informática de Ciudad Real (UCLM).

4.2.3. Software

A continuación se lista todas las herramientas y bibliotecas software utilizadas para el proyecto.

Sistema operativo

- **Ubuntu:** como sistema operativo principal se ha utilizado Ubuntu 11.04, al ser una de las distribuciones de GNU/Linux más usadas.

- **Debian:** se han realizado pruebas en otras distribuciones ampliamente extendidas como Debian *testing*, arrojando resultados satisfactorios.

- **Windows XP:** para portar la aplicación a plataformas Microsoft Windows se utilizó Windows XP SP2.

Software de desarrollo

- **Eclipse:** se utilizó la plataforma de desarrollo Eclipse Galileo 3.5-2, junto al plugin desarrollo para C++ Eclipse-CDT 6.0.2-1.

- **Emacs:** este editor de texto muy versátil y potente se ha utilizado para retoques mínimos del código de Minerva, y para el desarrollo de las demostraciones junto al *major-mode* de resaltado de sintaxis del lenguaje MSL. Se ha utilizado la version Emacs 23.2.

- **GCC:** el compildor de C++ utilizado ha sido *g++* (de la familia de compiladores *gcc*), en su versión 4.5.2.

- **Make:** se ha utilizado para crear el sistema de *Makefiles* de compilación que facilita el proceso. La versión instalada es la 3.81.

- **GDB:** se trata del depurador por excelencia de los sistemas GNU/Linux. Se ha utilizado la versión 7.2.

- **GPROF:** es una herramienta para hacer *profiling* (ver Sección 6.2) para compiladores de la familia *gcc*. Se ha utilizado la versión 2.21.

- **CMAKE:** se trata de una herramienta análoga a *make*, aunque de más alto nivel, para la automatización de generación de código. Se ha utilizado principalmente para la compilación de la biblioteca *Bullet*. La versión de CMake es la 2.8.3.

- **Visual C++:** compilador y entorno de desarrollo para C++ para plataformas Microsoft Windows. Se ha utilizado en su versión 2009.

Documentación y gráficos

- **Gimp:** potente herramienta de manipulación de gráficos utilizada para la documentación y generación de gráficos para las demostraciones. Se ha utilizado la versión 2.6.11.

- **Dia:** genera todo tipo de diagramas, utilizado para desarrollar el esquema de clases de los componentes MAO's y MLB's. Se ha utilizado la versión 0.97.1.

- **LibreOffice Draw:** herramienta de dibujo vectorial utilizada para la generación de figuras e imágenes para la documentación, en su versión 3.3.2.

- **LibreOffice Calc:** herramienta de creación y manipulación de hojas cálculo, utilizada para la generación de gráficos estadísticos de la documentación. Se ha aplicado la versión 3.3.2.

- **Latex:** herramienta para la creación de documentos profesional, utilizada para la generación de la misma de este Proyecto de Fin de Carrera. Se ha utilizado la versión 2.09 de TexLive.

- **Blender:** suite de modelado 3D para la importación y exportación de los modelos tridimensionales de las demostraciones. Se ha utilizado la versión 2.49a.

Bibliotecas

- **OpenGL:** OpenGl es una biblioteca para el dibujado de gráficos 2D y 3D multiplataforma y multilenguaje. Se ha utilizado su implementación *mesa* en la versión 7.10.2.

- **Flex++:** utilidad para la generación de analizadores léxicos. Se ha utilizado la versión 2.5.4a

- **Bison++:** utilidad para la generación de analizadores sintácticos, compatible con yacc. Se ha utilizado la versión 1.21.11-3.

- **SDL:** biblioteca para el desarrollo de aplicaciones multimedia que proporciona dibujado 2D, gestión de ventanas, eventos, sonido, fuentes, etc. Se ha utilizado la versión 1.2.14-6.

- **FreeGlut:** implementación libre de la *OpenGL Utility Toolkit*, que proporciona funciones útiles basadas en OpenGL. Se ha utilizado la versión 2.6.0.
- **Boost-python:** biblioteca que abstrae la tarea de embeber o empotrar el lenguaje Python en C++. Se ha utilizado la versión 1.42.0.
- **Boost-filesystem:** biblioteca que abstrae de la realización de operaciones relacionadas con el sistema de archivos como copiar, pegar o renombrar archivos, y proporcionar un acceso uniforme a las rutas del sistema de ficheros. Se ha utilizado la versión 1.42.0.
- **Python-dev:** biblioteca de desarrollo de Python para C/C++, utilizada para la escritura del intérprete del lenguaje de script. Se ha utilizado la versión 2.7.1.
- **ARToolKit:** biblioteca para la detección de marcas visuales. Se ha utilizado la versión 2.72.1.
- **Bullet:** biblioteca para la simulación física. Se ha utilizado la versión 2.77.

CAPÍTULO 5

ARQUITECTURA

En este capítulo se detalla la estructura *modular* de Minerva. El enfoque descriptivo utilizado se basa en una aproximación *Top-Down* donde se comenzará con una explicación funcional de cada módulo y componente. La especificación se irá refinando con un diseño más detallado hasta abordar, en los aspectos más complejos, las decisiones de diseño e implementación finales. Esta exposición se abordará de forma aislada para cada módulo indicando las relaciones con las clases implicadas, así como dependencias con otros módulos de la plataforma.

El sistema está compuesto por seis módulos, como se puede apreciar en el esquema de la Figura 5.1. Estos módulos son:

- *Módulo de componentes:* crea y gestiona la lógica de los componentes básico de Minerva: MAO's y MLB's.
- *Módulo de representación:* dibuja los elementos 3D y 2D de la aplicación, y gestiona su animación y simulación física.
- *Módulo de entrada-salida:* proporciona mecanismos de comunicación con el exterior.
- *Módulo de registro:* realiza el registro de la realidad mediante métodos de *tracking*.
- *Módulo de procesamiento de lenguajes:* procesa el lenguaje de alto nivel MSL y el de script.
- *Módulo de depuración:* proporciona métodos para controlar la depuración y gestionar errores del sistema.

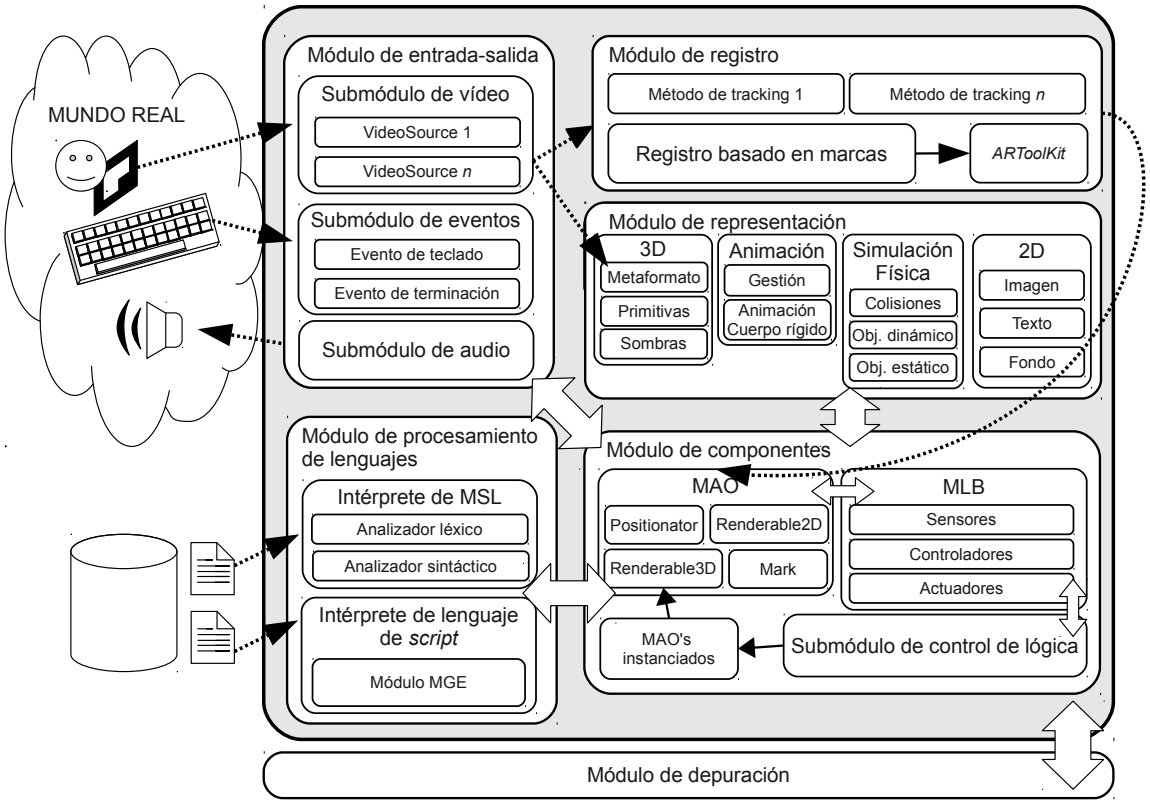

Figura 5.1: Esquema de la estructura modular de Minerva.

5.1. Descripción general

Minerva ha sido diseñada con una estructura *modular* y *extensible*. Gracias a este diseño el sistema puede ser extendido con nuevas funcionalidades sin tener que realizar cambios profundos. Algunas de las características más notables de esta arquitectura modular son:

- *Soporte multicámara:* la arquitectura puede soportar aplicaciones a las que se le proporcione imagen de más de una fuente de vídeo. La arquitectura clasifica estas fuentes entre *cámara principal* y *cámaras externas*.

- *Carga de modelos:* Minerva es capaz de cargar formatos 3D con textura desde cualquier metaformato que se implemente. La arquitectura es capaz de soportar tantos metaformatos como sea necesario.

- *Multimedia:* otra característica de Minerva es el soporte para reproducción de audio y detección de eventos de entrada. La arquitectura es capaz de soportar cualquier formato de audio y cualquier dispositivo de entrada, gracias a su estructura modular. Actualmente se admite el formato *.wav* para audio, y teclado como método de entrada. Sin embargo Minerva puede implementar mucha más variedad sin reescribir su arquitectura.

- *Métodos de registro:* Minerva ha sido intencionadamente diseñado para poder utilizar más de un método de registro simultáneamente. No importa la naturaleza del registro (tanto si es visual o no visual, relativo o absoluto), la arquitectura está preparada para soportar cualquiera.

- *Portabilidad:* la elección de las bibliotecas de las que depende este Proyecto de Fin de Carrera debían cumplir un requisito obligatorio: ser multiplataforma. Gracias a esta decisión, Minerva puede funcionar en tantas plataformas como sean soportadas por sus bibliotecas. Actualmente Minerva funciona en plataformas GNU/Linux y Microsoft Windows.

Un aspecto importante de la arquitectura modular de Minerva es el uso de los patrones *Factory* y *Controller*.

El patrón *Factory* proporciona una interfaz para crear de forma sencilla objetos que pueden ser de distintos tipos, aunque pertenecen a una misma jerarquía de clases. Este patrón es especialmente atractivo para Minerva dada su naturaleza basada en componentes (MAO's y MLB's), por lo que ayuda a mantener un orden y un diseño modular de la arquitectura.

El patrón *Controller* gestionar un módulo o submódulo completo mediante el uso de una interfaz sencilla y accesible desde todo el sistema. Minerva está orientada a módulos, por lo que se han implementado diferentes *Controller* como *GameLogicController* (ver Sección 5.2.3), *InputEventController* (ver Sección 5.4.2) o *PhysicsController* (ver Sección 5.3.2).

En las próximas secciones se realizará un análisis de cada uno de los módulos de Minerva (ver Fig. 5.1), haciendo una descripción funcional de cada módulo/submódulo, detallando dependencias y requisitos con otros submódulos y detalles de implementación relevantes que han sido necesarios.

5.2. Módulo de componentes

Este módulo es el encargado de proporcionar al resto del sistema los dos componentes básicos de Minerva, mediante su creación, gestión y eliminación.

La especificación de la lógica de las aplicaciones de Realidad Aumentada de Minerva está basada en los componentes MAO y MLB. Estos componentes básicamente son:

- *MAO (Minerva Augmenter Object):* se tratan de los *objetos* que componen la aplicación. Pueden ser una marca, un objeto tridimensional o un objeto bidimensional.

- *MLB (Minerva Logic Brick):* son los bloques lógicos que determinan *cómo* se comportan los MAO. Pueden ser de tres tipos: Sensores, Controladores y Actuadores. Cada MLB pertenece a un único MAO.

La colección de MAO's y MLB's son definidos por el usuario a través del lenguaje de especificación de lógica de alto nivel MSL (*Minerva Specification Language*). En la Sección 5.6.1 se describe el funcionamiento del submódulo de procesamiento de este lenguaje.

Al ser los componentes básicos de la aplicación, son los responsables de todo lo que pasa en ella. Pueden reproducir sonido o capturar eventos de entrada gracias al módulo de entrada-salida (ver Sección 5.4), son directamente accesibles gracias a la API de bajo nivel de *scripting* mediante un lenguaje de script (ver Sección 5.6.2). Los MAO que representan objetos tridimensionales pueden contener animaciones (ver Sección 5.3.3) y estar sometidos a simulación física (ver Sección 5.3.2). El módulo de registro (ver Sección 5.5) interactúa directamente con los MAO para determinar su posición en el espacio tridimensional. Por último, el submódulo de control de lógica establece las reglas de comportamiento de estos componentes para determinar el resultado final de la aplicación en cada momento (ver Sección 5.2.3).

El presente módulo está compuesto por los siguientes submódulos:

1. *Submódulo de componentes MAO:* se encarga de la creación y gestión de componentes MAO.

2. *Submódulo de componentes MLB:* se encarga de la creación y gestión de componentes MLB.

3. *Submódulo de control de lógica:* se encarga de arbitrar el comportamiento de los componentes y determina el resultado final de la aplicación.

A continuación se describen estos tres submódulos, proporcionando detalles de implementación en los aspectos más importantes.

5.2.1. Submódulo de componentes MAO

Este submódulo se encarga de crear y gestionar los MAO. Está compuesto por dos partes principales: los propios componentes MAO y la interfaz de creación y gestión de dichos componentes. A continuación se comenzará describiendo qué es un MAO, qué características soporta y qué distintos tipos existen. En última instancia se describirá la interfaz de creación y gestión.

Componentes MAO

Existen MAO's de diferentes tipos (Renderable2D, Renderable3D, Mark, etc), dependiendo del objeto que representen. En la Figura 5.2 se puede ver un esquema de todos los tipos disponibles, y las relaciones entre ellos. Cada uno tiene unas propias propiedades intrínsecas desde el momento de su creación (en el Anexo C se detallan todas ellas). A parte, el usuario

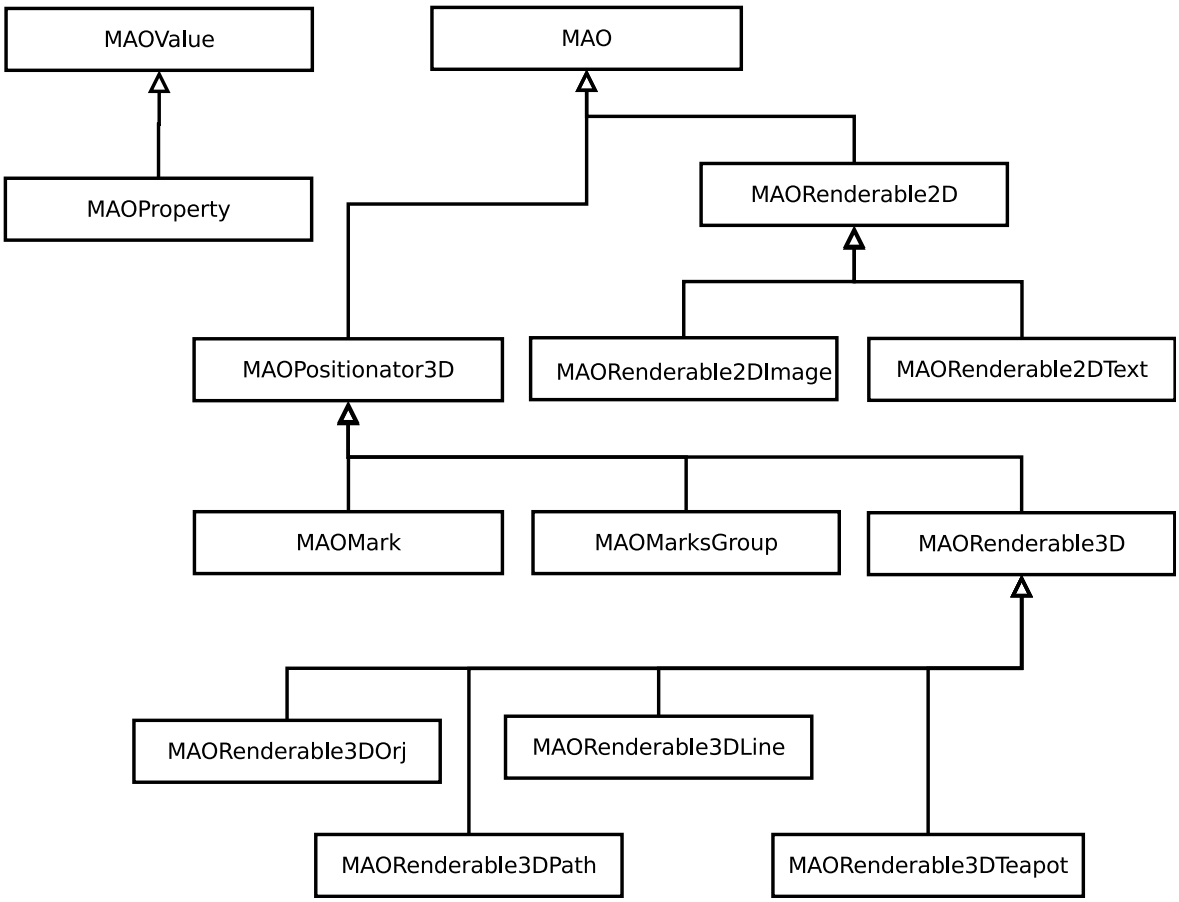

Figura 5.2: Jerarquía de clases de los MAO.

puede definir otras propias de distintos tipos (para más información sobre el uso de estas propiedades, ver el Anexo A).

La incorporación de propiedades a los componentes MAO de diversos tipos es un punto crítico de la arquitectura de Minerva. El diseño de estas propiedades es lo suficientemente flexible como para poder ser tratadas de forma homogénea, sin importar su tipo o naturaleza.

La clase base que implementa una propiedad es *MAOValue* (en el directorio *MAO*). Su objetivo es servir como contenedor para valores de distintos tipos en una sola clase. Los distintos tipos a los que puede pertenecer una propiedad son:

- *INT:* valor entero.
- *BOOLEAN:* admite los valores *True* o *False*.
- *FLOAT:* valor decimal.
- *STRING:* cadena de texto.
- *POSE:* matriz de transformación de 16 elementos decimales.

Cada tipo de propiedad es almacenada en un vector propio. La única diferencia de implementación entre las propiedades intrínsecas y las de usuario es que las primeras son añadidas en tiempo de compilación, en el constructor del componente, mientras que las últimas se añaden en tiempo de ejecución según lo indique el usuario.

Una función muy útil que proporciona la clase *MAO* es *hasProperty*, para saber si un determinado MAO dispone o no de una propiedad dado su nombre, independientemente de su tipo. Esta función se utiliza para implementar algunos MLB que necesitan saber si un MAO tiene o no una propiedad para su funcionamiento.

El funcionamiento se basa en la utilización de punteros a *void* para almacenar cualquier tipo de dato, más un campo *tipo* que indica la naturaleza del dato almacenado. *MAOValue* es una clase *template*, lo que facilita el acceso a sus datos conociendo su tipo. Se ha necesitado sobrescribir el constructor de copia, y el operador de asignación (=) para poder abstraerse de cuestiones de gestión de memoria.

Un *MAOValue* por sí solo no determina una propiedad de un MAO. Para ello se ha implementado la clase *MAOProperty*, basada en *MAOValue* (realmente hereda de ella) para poder tener la capacidad de ser identificada con un nombre. Además realiza unas definiciones auxiliares necesarias para el funcionamiento del *binding* entre Minerva y el lenguaje de script (ver Sección 5.6.2). Estas definiciones son:

```
typedef MPYProperty<int> MPYPropertyInt;
typedef MPYProperty<float> MPYPropertyFloat;
typedef MPYProperty<std::string> MPYPropertyStr;
typedef MPYProperty<bool> MPYPropertyBool;
typedef MPYProperty<cv::Mat> MPYPropertyPose;
```

A continuación se describe cada uno de los MAO de Minerva, detallando su funcionamiento e implementación.

- **MAOMark** Este MAO mantiene la información relacionada con la detección de una marca. La posición se la suministra directamente el módulo de registro (Sección 5.5). Las lista completa de características son:
  - Almacena la información básica para el reconocimiento y posicionamiento de una marca:: *size*, *center*, *path* e *id*.
  - Mantiene una matriz *offset* para poder trasladar el punto de referencia, que originalmente está en el centro (como se puede ver en la Figura 5.3). Cuando se trata de un MAOMarksGroup, los offset de todas las marcas que lo componen deben de apuntar al mismo centro de referencia. En la descripción de este último MAO se explica mejor su funcionamiento.

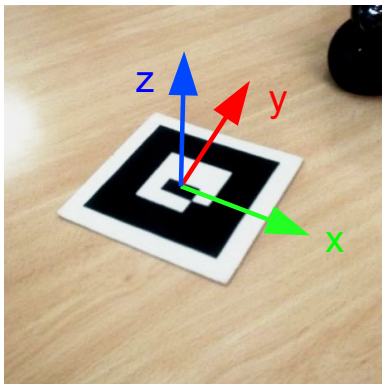

Figura 5.3: Centro de referencia por defecto en una marca de ARToolKit.

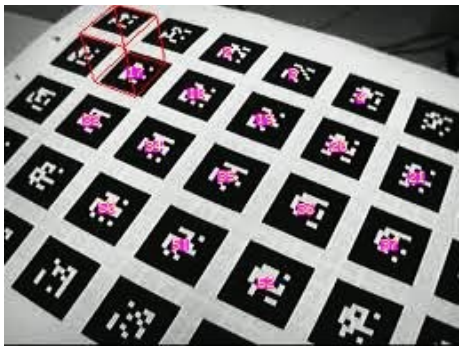

Figura 5.4: Patrón multimarca para declarar un MAOMarksGroup.

- Además cada marca almacena un histórico de, por defecto, cuatro *frames* para calcular la media ponderada y poder así suavizar el movimiento del registro.

- **MAOMarksGroup** Se trata de un conjunto de marcas cuya posición se mantiene constante (ver Figura 5.4). Actúa como una única marca. Esto facilita que el *tracking* pueda realizarse correctamente aunque no se detecten todas las marcas (bastaría con que se percibiera correctamente una sola).

  Se suelen utilizar para marcas que necesitan ser detectadas el mayor tiempo posible, como el *ground* de la simulación física.

  Funciona añadiendo MAOMark's ya definidos en la aplicación. Lo óptimo sería que todas las matrices *offset* de los MAOMark's apuntasen al mismo punto de referencia. Sin embargo, si se dejan con sus valores por defecto, el centro de referencia será la media de todos ellos. Si el centro de referencia exacto del MAOMarksGroup no es importante, funcionará sin especificarlo.

  La única función característica de este MAO es *addMarktoGroup*, que añade un MAOMark al grupo.

- **MAOPositionator3D** Representa un MAO que puede estar posicionado en el espacio. Almacena su información de posición (su matriz de transformación) y si está o no posicionado en ese momento. Esta última funcionalidad es especialmente útil para saber si se está realizando el *tracking* o no de un MAOMark o MAOMarksGroup, y por consecuencia de todos sus MAORenderable3D asociados.

  Como utilidad dispone de una función privada *setIdentityMat* que carga la matriz identidad en un *cv::Mat* que se le pase. Esta función se utiliza a modo de inicialización de los componentes MAOPositionator3D.

- **MAORenderable2D** Representa la interfaz para todos aquellos objetos representables en 2D. Todos los MAO de este tipo son proporcionados al submódulo de representación 2D (ver Sección 5.3.4) para su dibujado.

  Dispone un método *draw* común a todos los MAORenderable2D. El método virtual que utiliza *draw* es *getSDLSurface* que genera la imagen bidimensional a representar en función del tipo de MAORenderable2D implementado. En los tipos específicos de MAORenderable2D (imagen y texto) se describe la implementación de esta función.

  Además se almacena información como la posición (x e y), el ancho, el alto y su visibilidad.

  Se ha implementado *setOrtho2D* para cambiar la perspectiva a modo 2D, y *glGenTexture* que genera la textura OpenGL a partir de una *SDL_Surface*.

- **MAORenderable2DImage** Es un MAORenderable2D que carga una imagen de disco y la dibuja en la pantalla con el tamaño y en la posición que especifica el usuario.

  Sobrecarga la función *getSDLSurface* con una implementación que carga la imagen desde la ruta introducida mediante la función *IMG_Load*. La clase MAORenderable2D se encarga del resto.

- **MAORenderable2DText** Este MAO es capaz de renderizar texto con diferente color, tamaño y fuente de forma dinámica. La cadena del texto se puede cambiar en tiempo de ejecución.

  La información que almacena es color (r, g y b), tamaño (en puntos), estilo y fuente (en formato *ttf*).

  La función *getSDLSurface* se implementa utilizando la biblioteca *SDL_ttf* que renderiza texto. El código de esta función se puede ver en la Figura 5.5.

- **MAORenderable3D** Define la interfaz de los objetos representables en tres dimensiones. Estos objetos, junto a los MAO's instanciados, son utilizados por el submódulo de representación 3D (ver Sección 5.3.1). La información y funcionalidad que proporciona es:

```
SDL_Surface* MAORenderable2DText::getSDLSurface() {
    SDL_Surface* textSurface = NULL;
    SDL_Color color;

    color.r = getR();
    color.g = getG();
    color.b = getB();

    textSurface = TTF_RenderUTF8_Blended(_font, getProperty("text").
       getValue<std::string>().c_str(), color);

    setWidth(textSurface->w);
    setHeight(textSurface->h);

    return textSurface;
}
```

Figura 5.5: Código de la función *getSDLSurface* del MAORenderable2DText.

- **Dibujado:** proporciona funciones virtuales para el dibujado con y sin textura. Estas funciones son *drawGeometryWithTexture* y *drawGeometryWithoutTexture*. Además almacena su visibilidad, y el tamaño.
- **Posicionamiento:** almacena el MAO de referencia (MAOPositionator3D), y su matriz de posición.
- **Simulación física:** en cuanto a la simulación física almacena la *CollisionShape*, el tipo de forma de colisión (caja, cilíndrica, esférica y malla de triángulos) y la masa. Esta información es importante para el funcionamiento del submódulo de simulación física (ver Sección 5.3.2).

  Proporciona la función virtual *generateCollisionShape* que crea la forma de colisión dependiendo del tipo de MAORenderable3D del que se trate. En caso de los MAORenderable3DOrej es el propio importador el que se encarga de generarlo, aunque cualquier metaformato de descripción de modelos tridimensionales es capaz de generar la forma de colisión del objeto. La arquitectura de Minerva está diseñada para implementar cualquier tipo de formato para que proporcione estas características.
- **MAO's instanciados:** Los MAO's instanciados son un tipo especial de MAORenderable3D's. Estos MAO toman uno como referencia (*MAO Clase*), y se puede crear de él tantas copias como se desee. En el manual de usuario (ver Anexo A) se explica su uso. La información relativa a ellos que contiene MAORenderable3D son el tiempo que le queda de vida, y si están o no activos. Proporciona funciones para la gestión de la vida como *getTimeToExpire* o *decrementTimeToExpire*. El submódulo de control de lógica (ver Sección 5.2.3) es el encargado de eliminar estos MAO cuando su vida llega a su fin.

- **MAORenderable3DLine** Representa un único segmento en el espacio 3D. Para este segmento se puede especificar su color en formato RGB, y dos MAOPositionator3D que indiquen el inicio y el fin respectivamente del segmento.

  Para el renderizado de la línea se utiliza *GL_LINES* como se muestra a continuación:

```
glBegin(GL_LINES);
    glVertex3f(_x1, _y1, _z1);
    glVertex3f(_x2, _y2, _z2);
glEnd();
```

  La función privada *setPointsFromMao* obtiene la posición de los dos MAO de referencia indicados por el usuario y los almacena en las variables privadas *_x1, _y1, _z1* y *_x2, _y2, _z2*.

- **MAORenderable3DOrj** Dibuja la geometría, textura y animación definida por el metaformato OreJ. Proporciona la interfaz para gestionar las animaciones con las funciones *playAnim*, *pauseAnim* y *stopAnim*. Básicamente sirve de contenedor para los objetos proporcionados por el importador Orej, ya que uno de sus campos es un objeto de dicho metaformato.

  La función de generación de forma de colisión también accede a la proporcionada por el importador.

- **MAORenderable3DPath** Dibuja una ruta en el espacio 3D. Esta ruta está compuesta por varios tramos, que pueden ser de diferentes colores, e incluso algunos visibles y otros no.

  La implementación del MAO se basa en la utilización de un vector de objetos de *PathPoint's*. *PathPoint* es una clase auxiliar que contiene la información necesaria para representar un punto de un *MAORenderable3DPath*. La información que almacena cada objeto de esta clase es:

  - **Posición tridimensional:** coordenadas *x,y* y *z* del punto.
  - **Tamaño:** almacena el tamaño del punto. Este tamaño en realidad se refiere al grosor de la línea que une este punto con el siguiente de la lista. En la implementación concreta del submódulo de representación 3D, se utiliza la función de OpenGL *glLineWidth()*.
  - **Color:** valores *r, g* y *b* del color de la línea que comienza en este punto y continúa hasta el siguiente.
  - **Visibilidad:** es posible que ciertas partes de la ruta sean *invisibles*. En vez de implementar esto como varios MAORenderable3D distintos, se utiliza el mismo pero se puede indicar que a partir de ciertos puntos no haya visibilidad. El valor

```
1    glLineWidth(refPoint->getSize());
2    glBegin( GL_LINE_STRIP);

4    for (unsigned int i = 0; i < _vectorPathPoint.size(); i++) {
5        PathPoint& p = _vectorPathPoint.at(i);

7        if (hasChanged(p)) {
8            refPoint = &p;
9            glEnd();
10           glLineWidth(p.getSize());
11           glBegin(GL_LINE_STRIP);
12       }

14       if (p.getVisible()) {
15           glColor3f(p.getR() / 255.0f, p.getG() / 255.0f, p.getB() /
                 255.0f);
16           glVertex3f(p.getX(), p.getY(), p.getZ());
17       }
18   }
19   glEnd();
```

Figura 5.6: Código de la función de dibujado de MAORenderable3DPath.

de visibilidad de este punto representa la visibilidad de la línea que une este punto con el siguiente.

Una función importante de este MAO es *hasChanged*, para comprobar si las propiedades de un punto con otro consecutivo han cambiado, para poder realizar los cambios en la máquina de estados de OpenGL pertinentes.

El código de la función que dibuja la ruta se puede ver en la Figura 5.6.

La parte más importante del bucle *for* es la comprobación de si el punto actual ha cambiado con respecto al anterior (llamada *hasChanged(p))* para poder realizar los cambios oportunos.

- **MAORenderable3DTeapot** Es el MAORenderable3D más simple de todos. Únicamente dibuja una tetera con la función *glutSolidTeapot*. Fue el primero en implementarse con el objetivo de realizar pruebas sobre el resto del sistema.

Interfaz de creación

La gestión y creación de los MAO's y los MAO's instanciados la realiza la clase *MAOFactory* (que implementa el patrón *Factory*). Proporciona una interfaz para crearlos y añadirlos al sistema. Mantiene una lista con los MAO de cada tipo que se han creado. Esta clase tiene cuatro funciones principales:

1. **Interfaz de creación:** proporciona una o más funciones de creación del tipo *addMAOxxx* para cada tipo de MAO existente en el sistema (en el Anexo B se puede encontrar la lista completa junto a una breve descripción de cada una). Estas funciones, además de crearlo, hacen las gestiones pertinentes para que el sistema completo sea notificado de la incorporación del nuevo componente.

2. **Almacenamiento de los MAO:** funciona como almacén de estos componentes, pudiendo recuperarlos por tipo con funciones de la forma *getVectorMAOxxx*, o por el nombre con funciones como *getMAOxxx(string name)*.

3. **Gestión de MAO's instanciados:** los MAO's instanciados son un *caso aparte*. Al poder tener un tiempo de vida limitado, hace que el acceso a esos MAO deba ser diferente al resto. Esta clase proporciona funciones especiales para acceder a ellos. Estas funciones son:

   ```
   void addInstMAORenderable3D()
   vector getVectorInstMAORenderable3D()
   ```

4. **Gestión de propiedades:** muchos MLB's funcionan identificando MAO's comprobando si disponen o no de una determinada propiedad. Por ejemplo, MLBSensorNear se activa cualquier MAO con una propiedad determinada se acerca a menos de una distancia mínima al MAO padre del componente MLB. MAOFactory proporciona un método sencillo para comprobar si un MAO dispone de una determinada propiedad, independientemente de su tipo u otros detalles. Esta función es es *findProperty*.

5.2.2. Submódulo de componentes MLB

Este submódulo es el equivalente del submódulo de componentes MAO aplicado a los componentes MLB. Los MLB (*Minerva Logic Brick*) especifican la lógica individual de cada MAO, por lo que este submódulo da soporte al submódulo de componentes MAO. Además proporciona todos los componentes al submódulo de control de lógica (ver Sección 5.2.3) para determinar el comportamiento de la aplicación.

Al igual que el submódulo de componentes MAO, está compuesto de dos partes principales. La primera es la implementación de todos los MLB disponibles, y la segunda la interfaz de creación y gestión de estos.

A continuación se describen todos los componentes MLB, junto a su funcionamiento y detalles más complejos.

Componentes MLB

La interfaz de un MLB se implementa en la clase *MLB* (directorio *MLB*). Esta interfaz indica el tipo de MLB de que se trata el objeto, su nombre y el MAO al que pertenece (este MAO también es referido como *padre* o *MAO padre*).

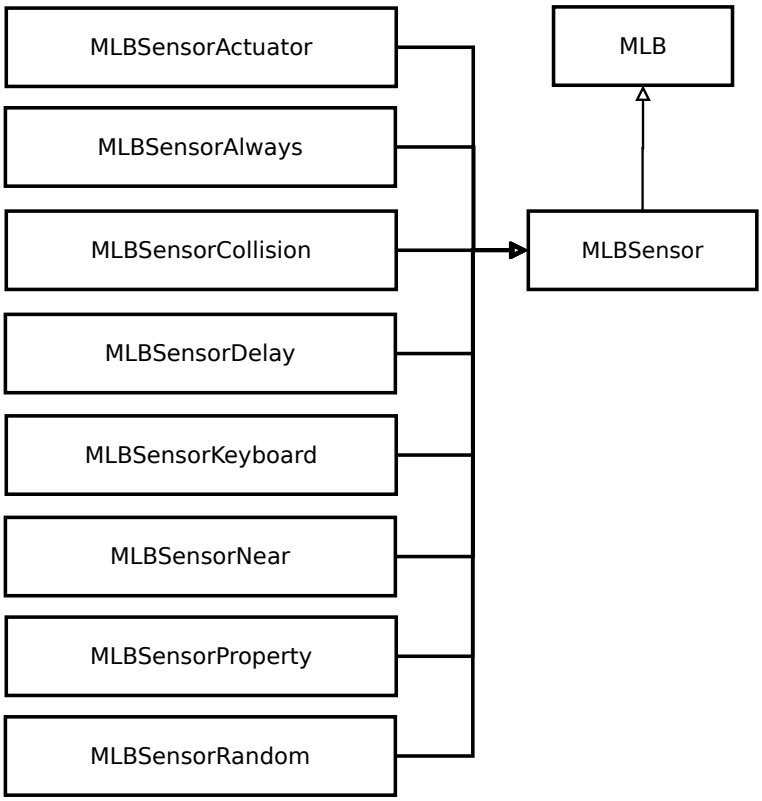

Figura 5.7: Jerarquía de clases de los MLB sensores.

Hay tres tipos de componentes MLB: Sensores, Controladores y Actuadores. En el Anexo A se explica el propósito y el funcionamiento de estos componentes. A continuación se detallan cada uno de los componentes según esta misma clasificación:

- **Sensores:** La interfaz de todo sensor está definida en la clase *MLBSensor* (directorio *MLB/Sensor*). Esta interfaz proporciona funciones para conocer si está o no activado (función *getState*), y para activarlo, entre otras.

  En el esquema de la Figura 5.7 se puede apreciar el esquema de estos componentes.

  - **MLBSensorActuator:** este sensor se activa cuando a su vez se activa un actuador determinado por el usuario. Esta funcionalidad se implementa mediante el patrón *Observador*. Este sensor no contiene nada, simplemente se subscribe al actuador que debe monitorizar mediante la función *addMLBSensorActuator* proporcionada por la interfaz *MLBActuator*.
  - **MLBSensorAlways:** siempre devuelve *True* mediante la función *getState*.
  - **MLBSensorCollision:** se activa cuando detecta la colisión entre el MAORenderable3D al que pertenece este sensor, y cualquier otro MAORenderable3D que tenga una propiedad especificada por el usuario. Para ello se apoya en el submódulo de simulación física (ver Sección 5.3.2). Este es un ejemplo de MLB

en el que es interesante saber si un MAO contiene una determinada propiedad, independientemente de su tipo o valor contenido.

- **MLBSensorDelay:** se activa periódicamente cada cierto tiempo especificado por el usuario. El tiempo se mide en *frames*, por lo que la medida temporal real está condicionada a los *fps* de la fuente de vídeo.
- **MLBSensorKeyboard:** detecta la pulsación de teclas de dos tipos: *KeyDown* (al presionar) y *KeyUp* (al liberar). La lista completa de teclas soportadas se encuentran en el Anexo E. Su funcionamiento se basa en el submódulo de eventos de entrada (ver Sección 5.4.2), mediante el patrón *Observador*. Cada sensor debe suscribirse al submódulo mediante la función *addMLBSensorKeyboard*.
- **MLBSensorNear:** se activa cuando un MAOPositionator3D (y toda su herencia) con una determinada propiedad especificada por el usuario se acerca a menos de una distancia al *MAO padre*. Para calcular la distancia utiliza las matrices de posición de ambos MAO, calculando la distancia euclídea entre los dos puntos de localización de dichas matrices.
- **MLBSensorProperty:** este sensor es capaz de detectar condiciones entre propiedades. Estas condiciones pueden ser de tres tipos:
  1. *EQUAL:* se activa cuando una propiedad es igual a un valor fijo proporcionado, o al valor de otra propiedad. Puede aplicarse a cualquier tipo de propiedad excepto el tipo *Pose*.
  2. *NOT_EQUAL:* se activa cuando una propiedad **no** es igual a un valor fijo proporcionado, o al valor de otra propiedad. Puede aplicarse a cualquier tipo de propiedad excepto el tipo *Pose*.
  3. *INTERVAL:* se activa cuando el valor de una propiedad está contenido en el intervalo cerrado comprendido por dos valores dados por el usuario. Puede aplicarse únicamente a los tipos *Int* y *Float*.

  La implementación de este sensor se basa en la utilización de *MAOProperty's* cuando el valor a comparar es el de otra propiedad, y de *MAOValue's* cuando el valor proporcionado es fijo.
- **MLBSensorRandom:** se activa de forma aleatoria en base a una probabilidad proporcionada por el usuario, cuyo valor puede ser [0,1]. Se utiliza la función *srand(time(NULL))* para inicializar la semilla, y la función *rand* para obtener el valor aleatorio. Si el valor aleatorio es menor o igual que la probabilidad determinada por el usuario, el sensor se activará.

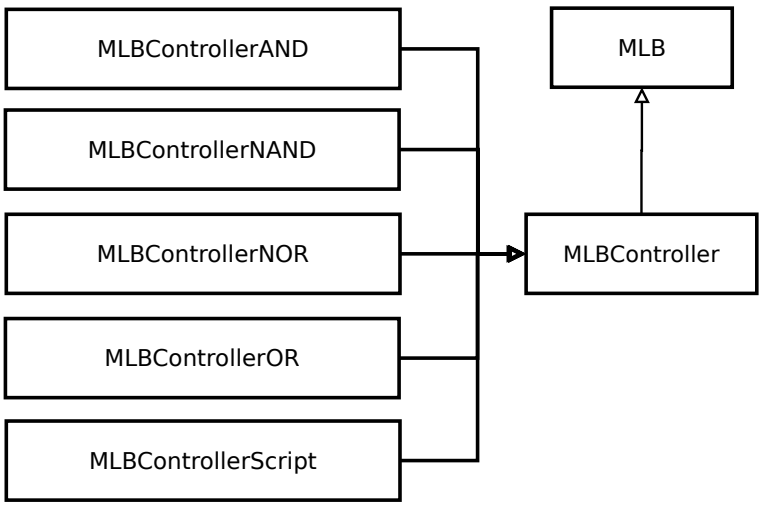

Figura 5.8: Jerarquía de clases de los MLB controladores.

- **Controladores:** La interfaz de los controladores se define en la clase *MLBController* (directorio *MLB/Controller*). Esta interfaz proporciona métodos para añadir sensores y controladores, y para evaluarlos. Además proporciona una función para forzar el activado de sus actuadores (función *activateActuators*).

  En la Figura 5.8 se aprecia el esquema de los controladores.

  - **MLBControllerAND:** realiza la función lógica AND (*&*) sobre el estado de todos sus sensores para determinar si debe activar todos sus actuadores.
  - **MLBControllerNAND:** realiza la función lógica NAND (*!&*) sobre el estado de todos sus sensores para determinar si debe activar todos sus actuadores.
  - **MLBControllerNOR:** realiza la función lógica AND (|) sobre el estado de todos sus sensores para determinar si debe activar todos sus actuadores.
  - **MLBControllerOR:** realiza la función lógica AND (*!*|) sobre el estado de todos sus sensores para determinar si debe activar todos sus actuadores.
  - **MLBControllerScript:** ejecuta un *script* en el lenguaje proporcionado para ello en Minerva. Se basa en el submódulo de procesamiento de lenguaje de *script* (ver Sección 5.6.2).

    En particular, la implementación de *scripting* de Minerva utiliza el lenguaje de programación Python. El sensor almacena un del tipo *PyObject* que almacena el código precompilado del script del usuario. Este mismo sensor se encarga de compilarlo dada la ruta del script. En la Figura 5.9 se puede ver el código, basado en *Boost-Python*, que compila el script. En primer lugar es necesario volcar todo el contenido del fichero del script a un búfer de caracteres. Después se llama a la función *Py_CompileString* mediante las macros de *Boost-Python* para propor-

```
file.open(_path.c_str());

//Calculating the length
int length = 0;
file.seekg(0, std::ios::end);
length = file.tellg();

char* buf = new char[length + 2]; //Allocating the memory!
file.seekg(0, std::ios::beg); //Rewind!

//Allocating the memory!
file.read(buf, length);
buf[length] = '\n';
buf[length + 1] = '\0';

file.close();

_compiledObj = boost::python::object(boost::python::handle<>(boost::
    python::borrowed(Py_CompileString((char*) buf,_path.c_str(),
    Py_file_input))));

delete[] buf;
_compiled = true;
```

Figura 5.9: Código de compilación de un script en *Python* basado en la biblioteca *Boost-Python*

cionar gestión dinámica de punteros. Una vez compilado se libera la memoria del búfer.

La ejecución del script lo lleva a cabo el submódulo de procesamiento de lenguaje de script, proporcionándoselo este sensor.

- **Actuadores:** La interfaz de los actuadores está definida en la clase *MLBActuator* (directorio *MLB/Actuator*). Esta interfaz proporciona una función para activarlo (función *actuate*), y otra para suscribir MLBSensorActuators (para implementar el funcionamiento de dichos sensores).

  En la Figura 5.10 se muestra el esquema de los actuadores del sistema.

  - **MLBActuatorAddDynamicObject:** este es el actuador que hace posible la existencia de MAO's instanciados en Minerva. Cada vez que se activa, se crea una copia del MAORenderable3D indicado por le usuario, en el lugar donde se encuentre el MAO de referencia (El MAOPositionator3D padre). En el Anexo A se explica más en detalle su uso.

    Está relacionado con el submódulo de simulación física, ya que el MAO indicado por el usuario de un MAO instanciado debe ser un objeto físico dinámico. El actuador tendrá como padre un MAOPositionator3D que actuará como *referencia de creación*. Esta referencia es el punto del espacio donde se creará el MAO

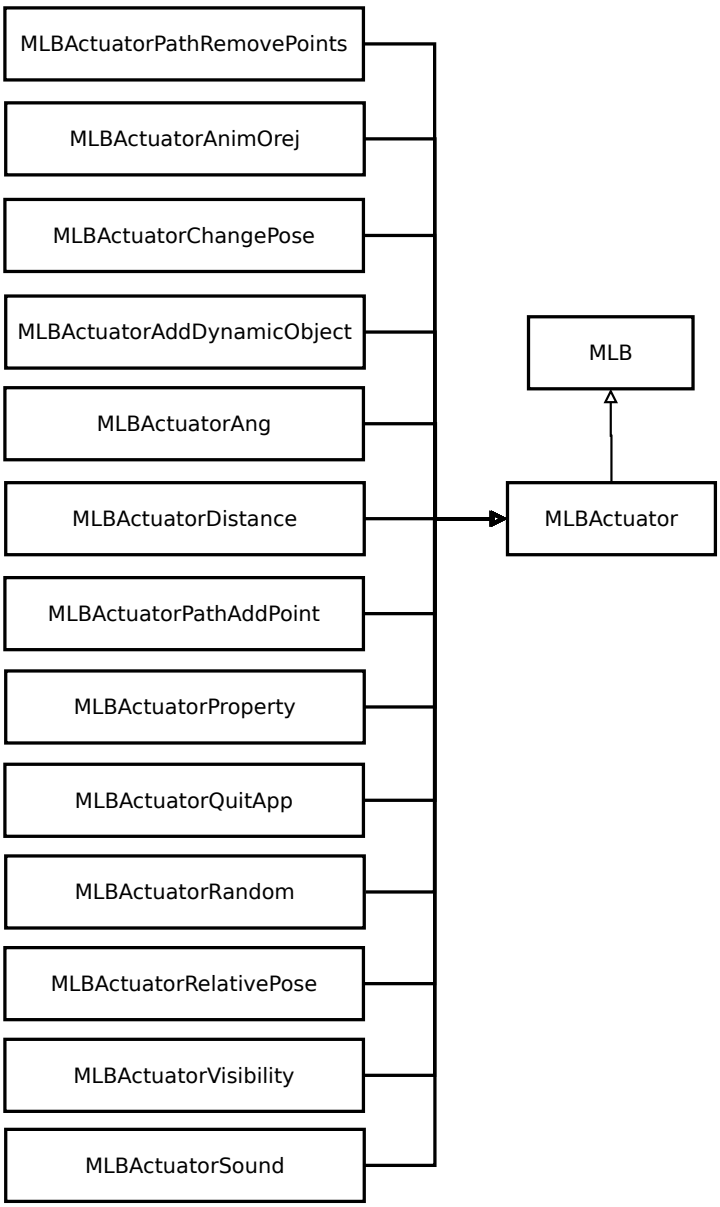

Figura 5.10: Jerarquía de clases de los MLB actuadores.

instanciado. Una vez creado, se independiza de su marca de referencia y queda sometido a las leyes físicas.

Los MAO's instanciados pueden disponer de una vida limitada o infinita. En caso de proporcionar una vida, esta se mide en *frames*. El submódulo de lógica de control (ver Sección 5.2.3) se encarga de decrementar esta vida, y eliminarlo cuando expire.

Estos MAO's pueden tener un impulso inicial a la hora de su creación, proporcionado en forma de vector 3D. La dirección y sentido del vector marcan la del impulso, y la magnitud la fuerza. También se puede indicar un *offset* desde su referencia de creación, para que el objeto se cree desde otro punto.

- **MLBActuatorAng:** calcula la diferencia de ángulo en uno de los tres ejes entre el MAO padre y otro MAO indicado por el usuario. El resultado se almacena en una variable del MAO padre también determinada por el usuario. El ángulo se obtiene calculando el producto escalar de los vectores de cada MAO que indican la dimensión requerida.
- **MLBActuatorAnimOrej:** proporciona un mecanismo para la gestión de las animaciones de los MAORenderable3DOrj. El MAO padre de este actuador debe ser un MAO de dicho tipo. Las operaciones que puede realizar son *play, pause* y *stop*. Directamente usa las funciones proporcionadas por el metaformato implementado en Minerva, OreJ. Para una descripción más detallada de dicho formato ver la Sección 5.3.1.
- **MLBActuatorChangePose:** aplica una transformación espacial (localización y rotación) a MAORenderable3D que *no* sean objetos dinámicos físicos. La transformación se puede aplicar de forma local o global. El ángulo de rotación debe ser proporcionado en *grados*.
- **MLBActuatorDistance:** calcula la distancia entre el MAO padre y otro MAO indicado por el usuario. Esta distancia se almacena en una propiedad *Int* o *Float* igualmente indicada por el usuario.
- **MLBActuatorPathAddPoint:** este actuador añade un punto más a la ruta del MAORenderable3DPath, que debe ser su padre. La información de este nuevo punto (posición, color, visibilidad, etc) se toma de los valores actuales que tengan esas variables en el MAORenderable3DPath. Para cambiar esos valores, es necesario acceder a los del MAO. En la demo *ARPaint* (ver Sección A) se implementa una aplicación simple de dibujado 3D utilizando un MAORenderable3DPath.
- **MLBActuatorPathRemovePoints:** elimina todos los puntos de la ruta del MAORenderable3DPath, que debe ser su padre. Simplemente limpia las estructuras de datos que almacena todos los PathPoint's.
- **MLBActuatorProperty:** actuador capaz de realizar operaciones simples sobre propiedades de los MAO. Estas operaciones son:
  - *ASSIGN:* asigna un valor fijo o el de una propiedad indicada.
  - *ADD:* suma al valor actual de la propiedad, un valor fijo o el de una propiedad indicada.
  - *MINUS:* resta al valor actual de la propiedad, un valor fijo o el de una propiedad indicada.
  - *DIVIDE:* divide el valor actual de la propiedad, por un valor fijo o el de una propiedad indicada.

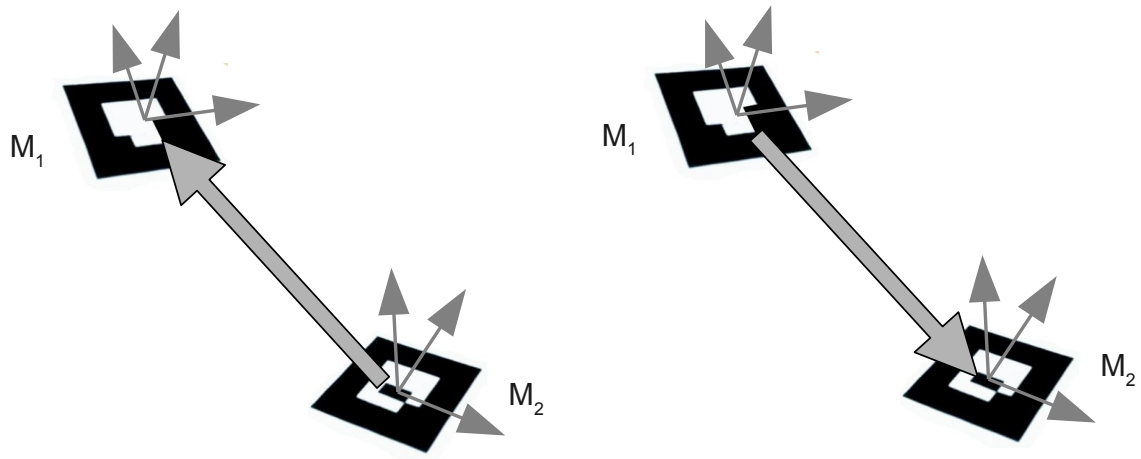

Figura 5.11: Transformaciones relativas entre dos MAOPositionator3D

- *MULTIPLY:* multiplica el valor actual de la propiedad, por un valor fijo o el de una propiedad indicada.

La operación de asignación puede ser aplicada a cualquier tipo de propiedad, mientras que el resto (suma, resta, multiplicación y división) puede ser aplicada únicamente a los tipos *Int* y *Float*.

- **MLBActuatorQuitApp:** provoca que finalize la aplicación. Para ello se basa en el *EndController*.

  El *EndController* (directorio *Kernel*) es una clase que implementa el patrón *Singleton*, y que mantiene una variable que indica si la aplicación ha finalizado o no. El objetivo de este controlador es el de poder propagar a todo el sistema cuándo termina la aplicación, para poder terminar correctamente todos los módulos y submódulos, y liberar todos los recursos liberados.

  Este sensor únicamente pide al *EndController* que finalice la aplicación.

- **MLBActuatorRandom:** este actuador genera un número aleatorio en el intervalo [0,1) y lo almacena en una propiedad el tipo *Float* indicada por el usuario. Al igual que el MLBSensorRandom, utiliza la función *srand(time(NULL))* para inicializar la semilla, y la función *rand* para generar el número aleatorio.

- **MLBActuatorRelativePose:** este sensor calcula la diferencia de transformación entre dos MAOPositionator3D. Se puede calcular la transformación directa o inversa. En la Figura 5.11 se puede ver la transformación directa entre los MAOPositionator3D M_1 y el M_2 (izquierda) y la inversa (derecha).

- **MLBActuatorSound:** este actuador reproduce un sonido utilizando el submódulo de audio (ver Sección 5.4.3). Los formatos de audio aceptados dependen de este subsistema.

- **MLBActuatorVisibility:** cambia la visibilidad de cualquier MAORenderable (2D o 3D). Simplemente establece el valor de la propiedad *visibility* a *True* o *False*, según establezca el usuario.

Interfaz de creación

La gestión y creación de componentes MLB, al igual que ocurría con los componentes MAO, lo realiza una clase *Factory* llamada *MLBFactory*. Esta clase tiene tres tareas principales:

1. **Interfaz de creación:** proporciona tres tipos de funciones de creación, para cada uno de los MLB (sensores, controladores y actuadores). Estas funciones son de la forma:

   ```
   MLBSensorxxx& addMLBSensorxxx()
   MLBControllerxxx& addMLBControllerxxx()
   MLBActuatorxxx& addMLBActuatorxxx()
   ```

   Crean los objetos y los añaden al sistema para que el subsitema de control de lógica (Sección 5.2.3) pueda tener buena cuenta de ellos.

2. **Almacenamiento de los MLB:** al igual que la MAOFactory, funciona como almacenamiento de los MLB que crea, proporcionando funciones para acceder a ellos por su tipo, o por su nombre y el de su MAO padre. Estas funciones son del tipo *getMLBxxx(string parentName, string mlbName)* o *getVectorMLBxxx*. Otro cometido de la MLBFactory es comprobar que no existan en el sistema dos MLB que pertenezcan al mismo padre y tengan el mismo nombre (puede suceder que pertenezcan a distintos MAO y se llamen igual) mediante la función privada *checkMLBName()*.

3. **Gestión de *links*:** en el momento de crear los componentes MLB no se indica los *links* que tienen con otros, es decir, no se indica a qué controladores se enlazará un sensor, o a qué actuadores se enlazará un controlador. Estos enlaces se definen más tarde mediante la función *addMLBLink(string parent, string mlb1, string mlb2)*. Sólo existe esta función, y además de crear el enlace realiza tareas de comprobación como que los nombres de los MLB's proporcionados existan y además sea un enlace válido (sensor-controlador y controlador-actuador únicamente).

```
checkInstMAO()

pollSensors()
pollControllers()
MPYWrapper::runScripts()

activateActuators()
```

Figura 5.12: Algoritmo de evaluación de la lógica SCA.

5.2.3. Submódulo de control de lógica

Este submódulo está implementado en la clase *GameLogicController* (directorio *Kernel*), y su objetivo es evaluar la lógica del juego especificada mediante los componentes MLB, por lo que debe tener acceso completo a estos mediante la *MLBFactory* (ver Sección 5.2.2). Esta tarea la lleva a cabo mediante su única función pública *pollLogic*. A esta función se le llama una vez por cada frame desde el hilo principal de la aplicación. El algoritmo que evalúa la lógica se muestra en la Figura 5.12.

Este submódulo también se ocupa de la vida de los MAO's instanciados. Como se explica en el manual de usuario (ver Anexo A), este tipo de MAO's pueden tener una vida limitada, y es el *GameLogicController* el encargado de comprobar si esta ha expirado.

El funcionamiento del algoritmo es el siguiente: se evalúan los sensores para determinar su estado actual. A continuación se evalúan los controladores y según estos, se activan los actuadores pertinentes. Los scripts se implementan como controladores, por lo que es en este momento cuando deben ser evaluados mediante el submódulo de procesamiento de lenguaje de script, en la Sección 5.6.2.

5.3. Módulo de representación

Este módulo comprende todas aquellas funcionalidades relacionadas con la representación de objetos 3D y 2D, su carga, animación, etc. Implementa la carga de metaformatos de geometría, carga de imágenes para texturas y renderable 2D, animaciones prefijadas y simulación física.

Accede a los componentes MAO gestionados por el submódulo de componentes (Sección 5.2) para renderizar aquellos que lo necesiten, y de acuerdo a sus configuraciones específicas (simulación física, colisiones, animaciones). Obtiene imagen del mundo real mediante el submódulo de vídeo (Sección 5.4.1).

La inicialización del sistema de ventanas y representación gráfica se hace en base a la biblioteca *SDL*. En la clase *World* (directorio *Kernel*) se encuentra la función *initWorld* en la que se inicializa varios subsistemas.

Después de dibujar los objetos (tanto 2D como 3D), se fuerza el renderizado mediante la llamada a la función *SDL_GL_SwapBuffers()*, ya que la representación de gráficos en Minerva se implementa mediante una técnica de doble búfer. Esta característica se activa activando el atributo con la siguiente función:

```
SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
```

Esta técnica consiste en tener dos búfers de pintado. Mientras que uno se utiliza para representar la información en pantalla, en el otro se realizan los cambios para el siguiente frame. Una vez terminado se *intercambian* (*swap* en inglés) los búfer y uno hace la función del otro. Así se consigue solapar la fase de representación en el dispositivo hardware y la escritura de los cambios del siguiente frame, ganando en velocidad.

El módulo de representación se divide a su vez en cuatro submódulos:

1. *Submódulo de representación 3D:* dibuja los objetos tridimensionales.
2. *Submódulo de simulación física:* gestiona la simulación física y colisiones entre componentes.
3. *Submódulo de animación:* gestiona las animaciones prefijadas.
4. *Submódulo de representación 2D:* dibuja los objetos bidimensionales.

5.3.1. Submódulo de representación 3D

Este submódulo se ocupa de la carga, gestión y dibujado de los objetos tridimensionales de Minerva. La arquitectura de Minerva proporciona tres cuatro de objetos tridimensionales:

- *Teapot:* dibuja una taza tridimensional. Este objeto principalmente tiene funciones de depuración y pruebas. Se implementa mediante el MAORenderable3DTeapot.
- *Line:* dibuja una única línea entre dos puntos del espacio, cuyo color y grosor pueden ser personalizados. Se implementa mediante el MAORenderable3DLine.
- *Path:* dibuja una ruta de puntos en el espacio, con distintos colores, grosores y visibilidad. Se implementa a través de un MAORenderable3DPath.
- *Triangle mesh:* Minerva es capaz de leer objetos tridimensionales cuya geometría esté definida como una malla triangular, con texturas y animación (para conocer en detalle el submódulo de animación, ver Sección 5.3.3). En concreto se soporta el metaformato *OreJ*, creado por el grupo de investigación *Oreto* de la Escuela Superior de Informática de Ciudad Real (UCLM).

A continuación se describe la implementación del renderizado de objetos tridimensionales, de sus sombras, y del metaformato *OreJ*.

Figura 5.13: Diferencia entre utilizar (izquierda)o no (derecha) GL_DEPTH_TEST.

Renderizado de objetos tridimensionales

El renderizado de los objetos tridimensionales se implementa en la clase *World* (directorio *Kernel*), ayudándose de la interfaz definida en la clase *MAORenderable3D*. En el bucle principal de dibujado de *World* se llama a las funciones de dibujado de los MAORenderable3D y los MAORenderable3D's instanciados. Estos dos tipos de MAO se gestionan de forma diferente, por lo que le dibujado es independiente.

Para ello, *World* pide al submódulo de componentes MAO (ver Sección 5.2.1) todos aquellos MAORenderable3D y MAO's instanciados, e invoca a su función *drawWithTexture*.

Un aspecto importante del dibujado tridimensional el *DEPTH_TEST* a la hora del renderizado. Esto se utiliza para evitar que el orden en que se dibujan los objetos no provoque errores de profundidad. Por ejemplo, si se tiene un objeto *MAO1* que se dibuja primero y está en primer plano, y un objeto *MAO2* que se dibuja después de *MAO1* pero está en segundo plano, al ser este último dibujado después aparecería *encima* de *MAO1*, aunque lo correcto sería que apareciese detrás. Este es el objetivo de habilitar *DEPTH_TEST*. La función específica que lo habilita es *glEnable(GL_DEPTH_TEST)*. En la Figura 5.13 se puede observar la diferencia entre usar o no usar esta técnica.

Sombras y *Ground*

Como parte de las características de la simulación física, se puede indicar al módulo de representación que calcule las sombras y las dibuje, junto al suelo del mundo físico.

Las sombras se implementan utilizando un algoritmo que duplica la geometría del cuerpo a proyectar y la *aplasta* sobre el suelo. Para ello se utiliza una matriz de perspectiva, calculada por el submódulo de simulación física (ver Sección 5.3.2), teniendo en cuenta la posición del suelo y del sol respecto del cuerpo. Este algoritmo se encuentra en el Anexo G.

Además también se puede indicar que el submódulo de representación 3D dibuje un plano sobre el suelo de la simulación física. El algoritmo que lleva a cabo la obtención de la normal del plano a partir de la matriz de transformación de la marca que actúa como referencia del suelo se puede ver en el Anexo K.

Metaformato OreJ

Minerva utiliza una versión modificada del importador del formato Orej. Básicamente se trata de un formato basado en el formato OBJ definido por *Alias Wavefront*, añadiendo animación de cuerpo rígido. Almacena la información en formato ASCII. Las principales características que soporta son:

- **Mallas triangulares:** el formato OreJ guarda la información geométrica del modelo como vértices (marca *v*) y caras triangulares formadas por tres vértices cada una (marca *f*). Dependiendo de la versión del importador puede almacenar también la normal de cada cara.

- **Carga de texturas en diferentes formatos:** la textura se almacena en un fichero imagen aparte de la información geométrica del fichero *.orj*. La carga de la imagen se realiza utilizando la función *IMG_Load* de la biblioteca *SDL_Image*, por lo que puede tener cualquier formato soportado por ella. Las coordenadas *uv* se almacenan en el fichero *.orj* con la marca *t*.

- **Animación:** en la Sección 5.3.3 se explica el soporte de animaciones prefijadas de Minerva.

- **Formas de colisión:** esta característica es específica de la versión para Minerva del importador. Una vez cargada la geometría y la textura, el importador puede calcular diferentes formas de colisión (*Collision Shapes*) necesarias para el subsistema de simulación física (ver Sección 5.3.2). Para generar estas formas el importador ofrece las siguientes funciones:

  ```
  generateBoxShape
  generateCylinderShape
  generateSphereShape
  generateConvexTriangleMeshShape
  ```

  El importador trata de generar la forma más óptima para el modelo según la geometría escogida (caja, cilíndrica, esférica o malla de triángulos). Esto es especialmente útil para la simulación física o la detección de colisiones.

- **Soporte de texturas:** otra característica especial del importador de metaformatos de Minerva es la posibilidad de dibujar la geometría con o sin textura. Esta funcionalidad es necesaria para el renderizado de sombras, ya que según el algoritmo implementado (ver Anexo G) una sombra no es más que duplicar la geometría *aplastada* según la matriz de transformación proporcionada por el algoritmo. La única diferencia es que la sombra *no* tiene textura, debe ser todo de un color grisáceo.

La otra parte del formato OreJ es el exportador. Se trata de un script escrito en Python para la suite de modelado 3D Blender que es capaz de generar los ficheros *.orj* a partir de un modelo *.blend*. En el Anexo H se explica paso a paso cómo crear el fichero de metaformato OreJ a partir de un objeto diseñado en blender.

A continuación se describe uno de los módulos más complejos e importantes de Minerva, el módulo de simulación física. Este módulo tiene relaciones con muchos otros, y añade mucho atractivo al presente Proyecto de Fin de Carrera.

5.3.2. Submódulo de simulación física

El submódulo de simulación física añade a Minerva el poder simular fuerza de gravedad y colisiones dinámicas entre objetos tridimensionales, indicando como referencia global un suelo (también denominado *Ground*). A parte de esto también es capaz de detectar colisiones sin que exista ninguna otra simulación física.

Para ello se basa en la información de formas de colisión que suministra el subsistema de componentes MAO (ver Sección 5.2.1). Toda esta gestión la lleva a cabo la clase *Physics-Controller* (directorio *Controllers*). Se trata de una clase que implementa el patrón *Singleton* y *Controller*.

Para ampliar la información física de los objetos MAORenderable3D, utiliza como clases auxiliares *PhysicObject* y *PhysicDynamicObject* (ambas en el directorio *Kernel*).

A continuación se describe el *Controller* y las clases auxiliares, junto a la funcionalidad y detalles de implementación de cada una.

Controlador de simulación física

PhysicsController es el controlador más complejo de todos. Gestiona todo lo relacionado con la simulación física, basándose en la biblioteca *Bullet*. Su funcionalidad se subdivide a su vez en:

- Detección de colisiones: la detección de colisiones entre dos MAORenderable3D debe poder realizarse sin el uso de simulación física explícita para el funcionamiento del MLBSensorCollision. Es capaz de funcionar con estructuras de datos diferentes a las de la simulación física. Para ello proporciona la función pública:

  ```
  bool collision(MAORenderable3D* m1, MAORenderable3D* m2)
  ```

  Esta función devuelve directamente los valores *True* o *False* para indicar si ha habido colisión.

- Referencia global: para la simulación física, un requisito indispensable es disponer de un MAOMark o un MAOMarksGroup que actúe como referencia global del mundo

```
1    _collisionConf = new btDefaultCollisionConfiguration();
2    _dispatcher = new btCollisionDispatcher(_collisionConf);
3    _broadphase = new btDbvtBroadphase();
4    _solver = new btSequentialImpulseConstraintSolver();
5    _world = new btDiscreteDynamicsWorld(_dispatcher, _broadphase,
        _solver, _collisionConf);
```

Figura 5.14: Objetos básicos de Bullet para la simulación física.

físico. Este MAO define el plano del suelo y la dirección, sentido y magnitud de la gravedad. Este controlador se encarga de mantener cuál es el MAO que realiza esta función, conocido como *Ground*.

- Leyes físicas: la función principal es la de proporcionar simulación física de objetos dinámicos y estáticos. Mantiene la lista de estos tipos de objetos que existen en la aplicación. Proporciona una interfaz para añadir y eliminar estos objetos de forma sencilla mediante las siguientes funciones:

  ```
  void addStaticRigidBody()
  void addDynamicRigidBody()
  void removeDynamicRigidBody()
  void removeStaticRigidBody()
  ```

  El controlador no utiliza directamente MAORenderable3D's, sino que se apoya en el uso de objetos *PhysicObject* y *PhysicDynamicObject* definidos en el directorio *Kernel*. Más adelante en esta misma sección se describe el funcionamiento y aspectos más importantes de estos objetos.

  Para utilizar la simulación es necesario la inicialización de las estructuras de datos de Bullet previamente. Los objetos necesarios básicos se muestran en la Figura 5.14.

  Una vez inicializado, el proceso principal llama a la función pública *pollPhysics* de este controlador que calcula las transformaciones pertinentes de cada objeto suscrito a la simulación física. El algoritmo principal de esta función se muestra en la Figura 5.15.

El algoritmo comienza recalculando las sombras (si éstas están activas). Después indica a Bullet que avance un *step* la simulación física, por lo que actualizaría la posición actual de todos los objetos que pertenezcan al objeto *world*. Una vez realizada la simulación, se actualizan cada uno de los objetos para que el subsistema de dibujado los represente correctamente.

En el caso de los objetos estáticos, se debe indicar al objeto *world* la posición del objeto, ya que estos no están sometidos a simulaciones físicas, es el propio usuario el que determina su posición, y es necesario informarla a Bullet para poder calcular las colisiones.

```
if(isActive() && ground->isPositioned(){
    calculateShadowsMatrix()

    //Avanzar un paso la simulacion de Bullet
    world->stepSimulation()

    //Actualizar los objetos estaticos
    for o in staticObjects{
        o->setWorldTransform()
    }

    //Actualizar los objetos dinamicos
    for o in dynamicObjects{
        if(!o->isCreated()){ //No ha sido creado todavia
            if(o->getCreationRef()->isPositioned()){ //Se puede crear
                o->setCreated(true) //Marcar como creado
                o->applyInitialImpulse() //Aplicar impulso inicial
                world->addRigidBody(o) //Anadir objeto a Bullet
            }
        }else{ //Ya ha sido creado
            t = o->getWorldTransform()
            o->getMAO()->setWorldTransform()
        }
    }
}
```

Figura 5.15: Pseudocódigo de la función *pollPhysics*.

En cuando a los objetos dinámicos, se obtiene la transformación del objeto según la simulación física y se actualiza en el MAO para que el submódulo de representación 3D (ver Sección 5.3.1) pueda dibujarlo en su posición correcta. Además estos objetos pueden ser creados mediante el actuador MLBActuatorAddDynamicObject, por lo que el controlador debe tener en cuenta esa creación. La creación solo se va a producir siempre y cuando la referencia de creación (el MAOMark o MAOMarksGroup asociado al MAORenderable3D) esté posicionado en ese momento.

PhysicObject

Se trata de una clase para extender un *MAORenderable3D* como un objeto estático de la simulación física. Intuitivamente es una *capa externa* que se añade a los MAO para completar su información con la necesaria para poder representar un objeto físico estático. La información exacta que almacena esta clase es:

- **MAO al que representa:** almacena un puntero al MAO que se adapta como objeto físico estático.

- **RigidBody:** se trata de una estructura que almacena la forma de colisión (proporcionada por el MAO por medio del importador del metaformato o indicado por el usuario),

```
1  //Creating the rigid body!
2  float mass = mao->getMass();
3
4  btVector3 localInertia(0,0,0);
5
6  btCollisionShape* cs = mao->getCollisionShape();
7
8  if(mass!=0.f){ //Is dynamic!
9      cs->calculateLocalInertia(mass,localInertia);
10 }
11
12 btTransform t;
13 t.setIdentity();
14
15 btDefaultMotionState* ms = new btDefaultMotionState(t);
16
17 btRigidBody::btRigidBodyConstructionInfo rbInfo(mass,ms,cs,localInertia);
18
19 _rigidBody = new btRigidBody(rbInfo);
20
21 //Kinematic object? (Static!)
22 if(mass==0.f){
23     _rigidBody->setCollisionFlags(_rigidBody->getCollisionFlags() |
          btCollisionObject::CF_KINEMATIC_OBJECT);
24     _rigidBody->setActivationState(DISABLE_DEACTIVATION);
25 }
```

Figura 5.16: Creación de un objeto físico de Bullet.

la masa (para los objetos estáticos debe ser siempre 0) y otras propiedades de implementación del submódulo de simulación física.

El algoritmo básico para la creación de un objeto físico que pueda manejar Bullet se muestra en la Figura 5.16.

Por último quedaría añadirlo al *world* de Bullet para que pueda tenerlo en cuenta. De esto se encarga el *PhysicsController* y lo realiza mediante la función *world->addRigidBody()*.

PhysicDynamicObject

Es una clase para extender un *MAORenderable3D* como un objeto dinámico de la simulación física. Realmente esta clase hereda de la clase *PhysicObject*, y añade información relativa a un objeto físico dinámico. Esta información adicional es:

- **Gestión de la creación:** los objetos físicos dinámicos tienen un momento de creación, por lo que es necesario gestionar si han sido o no creados. También necesitan una referencia de creación, que no es más que un MAOMark o un MAOMarksGroup.
- **Impulso inicial:** almacena y aplica un posible impulso inicial de estos objetos en el momento de la creación. El impulso se almacena en un vector tridimensional (tres elementos) de tipo decimal (*Float*).
- **Offset:** es posible que el usuario haya especificado un *desplazamiento* en forma de transformación espacial a partir de la referencia de creación. Esta clase también almacena este desplazamiento para que el *PhysicsController* pueda aplicarlo convenientemente.

5.3.3. Submódulo de animación

Minerva permite la incorporación de animaciones prefijadas de cuerpo rígido. Este tipo de animaciones se extrae del propio fichero de metaformato donde se especifica el objeto tridimensional.

La versión del importador OreJ implementado en Minerva soporta una animación de cuerpo rígido. La animación se especifica en el fichero *.orj*, enumerando la matriz de transformación (16 valores decimales) de cada uno de los frames. Los frames de animación se indican con la marca *m*.

El control de la animación se realiza utilizando la interfaz proporcionada por el importador mediante las siguientes funciones *playAnim, pauseAnim* y *stopAnim*.

También se realiza un control del *timing* de la animación, para que el *frame rate* se mantenga constante independientemente de la velocidad del hardware en el que se ejecute la aplicación. El *frame rate* lo controla la obtención de vídeo del dispositivo, por lo que el *fps* sera igual al proporcionado por dicho dispositivo.

En cuanto a las animaciones no prefijadas, la simulación de objetos dinámicos físicos se considera un tipo de animación. Este tipo de animación se detalla en el submódulo de simulación física, en la Sección 5.3.2.

5.3.4. Submódulo de representación 2D

Este módulo se encarga del dibujado de objetos 2D. Estos objetos actualmente pueden tratarse de imágenes o texto dinámico.

La lista de objetos 2D se la suministra el submódulo de componentes MAO (ver Sección 5.2.1). Estos objetos se corresponden con los MAORenderable2D. Además se encarga de dibujar la imagen captada con el submódulo de vídeo (ver Sección 5.4.1) de fondo (*background*).

A continuación se describe detalladamente cada una de estas tareas que lleva a cabo el submódulo de representación 2D.

Dibujado del *BackGround*

La primera de las acciones que lleva a cabo es dibujar de fondo (*background*) el frame de la cámara. Para ello carga el frame como una textura y lo dibuja en un polígono de 4 vértices. Al terminar de dibujar se libera la memoria ocupada por la textura. Únicamente para esta parte se cambia la matriz de proyección mediante *glOrtho* para dibujado en 2D, para luego retomar la matriz de perspectiva de la fuente de vídeo.

Dibujado de los objetos 2D

Para dibujar los MAORenderable2D hay que llevar a cabo primero un cambio de perspectiva para dibujado en 2D mediante la función *glOrtho* de la misma forma que ocurría la dibujar el background. Este cambio de perspectiva se devuelve a su estado anterior al terminar de dibujar los MAORenderable2D (imágenes y texto). Las funciones que cambian las perspectivas entre 2D y 3D de la clase *World* son:

```
1  void World::enable2D() {
2      glMatrixMode(GL_PROJECTION);
3      glPushMatrix();
4      glLoadIdentity();
5      glOrtho(0, _width, 0, _height, -1., 1.);
6  }
7
8  void World::disable2D() {
9      glMatrixMode(GL_PROJECTION);
10     glPopMatrix();
11 }
```

Realmente la perspectiva 2D es un estado *temporal* de OpenGL. Para volver a la perspectiva 3D simplemente hay que retirar la última matriz de la pila de *proyección* (con un *glPopMatrix*), ya que al cambiar a perspectiva 2D se hace un *glPushMatrix*.

En el caso específico de los MAORenderable2DText (objeto de texto dinámico), es necesario inicializar la biblioteca que lleva a cabo esta tarea. La inicialización la lleva a cabo la función *initWorld* de la clase *World* (directorio *Kernel*) con la siguiente llamada de la biblioteca *SDL_ttf*:

```
TTF_Init()
```

Para conocer más a fondo la implementación de los objetos 2D de imagen y texto, ver la descripción del submódulo de componentes MAO, más concretamente la de los MAORenderable2DImage y MAORenderable2DText (ver Sección 5.2.1).

5.4. Módulo de Entrada-Salida

Este módulo proporciona al resto del sistema aquellos eventos, información o características relacionadas con la comunicación con el exterior, es decir la *E/S* (Entrada-Salida).

Está compuesto por tres submódulos:

1. *Submódulo de captura de vídeo:* ofrece imagen de la realidad con soporte multicámara.
2. *Submódulo de eventos de entrada:* detecta eventos de entrada de distintos tipos y distintos dispositivos como teclados.
3. *Submódulo de audio:* proporciona una interfaz para reproducir sonidos.

El módulo de registro (ver Sección 5.5) depende totalmente del submódulo de captura de vídeo para realizar el *tracking* de los métodos visuales implementados. Además el submódulo de representación 2D (ver Sección 5.3.4) utiliza esta misma información visual para dibujarla de fondo en la ventana. Por otra parte, el submódulo de entrada y el de audio están relacionados con el módulo de componentes (ver Sección 5.2), ya que muchos de estos dependen de ellos para su funcionamiento, como MLBSensorKeyboard o MLBActuatorSound.

A continuación se explicará en detalle el funcionamiento de estos submódulos, junto a relaciones y dificultades que los caracterizan.

5.4.1. Submódulo de captura de vídeo

El módulo de captura de vídeo se encarga de crear y proporcionar fuentes de vídeo de diversa naturaleza para dar soporte al submódulo de registro (ver Sección 5.5), y al submódulo de representación 2D (ver Sección 5.3.4).

Es capaz de dar soporte multicámara. Se pueden crear tantas fuentes de vídeo como se disponga en el sistema, y el submódulo de vídeo las clasificará como:

- *Cámara principal:* es necesaria esta fuente de vídeo para ejecutar una aplicación Minerva. Realiza el registro de la realidad y se dibuja de fondo.
- *Cámaras externas:* no es necesario que exista ninguna. Sirven de soporte para realizar registro sobre ellas, pero *a priori* no se visualiza al usuario.

El submódulo de vídeo se implementa en dos componentes: el *VideoFactory*, que proporciona la interfaz de creación y gestión de fuentes de vídeo, y el *VideoSource*, que implementa una única fuente de vídeo basáda en la biblioteca de visión artificial *OpenCV*.

En las siguientes secciones se describe el funcionamiento de estos componentes.

```
cv::VideoCapture* c = new cv::VideoCapture(nDevice);
_cam = c;

if(!_cam->isOpened()){
    throw "Camera not found exception: "+name;
}

_name = name;
_lastFrame = new cv::Mat();
_width = width;
_height = height;
_cam->set(CV_CAP_PROP_FRAME_WIDTH,_width);
_cam->set(CV_CAP_PROP_FRAME_HEIGHT,_height);
_fps = _cam.get(CV_CAP_PROP_FPS);
```

Figura 5.17: Creación de un *VideoCapture* de OpenCV.

VideoFactory

Gestiona los distintos dispositivos de vídeo. Se implementa en la clase *VideoFactory* (directorio *Factories*).

Mantiene cuál de los dispositivos es el *principal*. Las funciones públicas que frece su interfaz son: *addVideoSource*, que añade un dispositivo de vídeo dado su identificador, *getCamera* que devuelve un dispositivo ya creado conocido su nombre, y *getMainCamera*, que devuelve el dispositivo principal.

VideoFactory da soporte multicámara para poder realizar *tracking* con varias de ellas simultáneamente. La arquitectura se implementó de esta forma para poder crear aplicaciones multicámara, por ejemplo una en la que el usuario disponga de una cámara y existan otras externas que lo detecten y puedan calcular su posición. La *main camera* es sobre la que se realizará la composición, es decir, de la que se muestra vídeo sobre la ventana gracias al submódulo de representación 2D (ver Sección 5.3.4).

Los dispositivos de vídeo se implementan utilizando la biblioteca *OpenCV*, implementado en la clase *VideoSource*. En la siguiente sección se explica el funcionamiento de esta clase.

VideoSource

VideoSource representa una fuente de vídeo para la obtención de *frames*, basada en la biblioteca OpenCV. Se encuentra implementado en la clase *VideoSource* (directorio *Kernel*). Supone una capa de abstracción sobre los *VideoCapture* de OpenCV, añadiendo la funcionalidad de poder recuperar el último frame sin tener que pedirlo al dispositivo de vídeo, o servir información como los frames por segundo o el tamaño del frame. La forma en la que Minerva crea un dispositivo de vídeo basándose en OpenCV se muestra en la Figura 5.17.

Para recuperar un frame del dispositivo de vídeo se utiliza la función *grabFrame* de Vi-

deoSource, basada a su vez en el operador >> del objeto *VideoCapture* de OpenCV. Esta función es *bloqueante*, por lo que funciona a la misma frecuencia que el dispositivo de vídeo. Al ser Minerva un sistema monohilo, esto sirve para controlar los frames por segundo de la aplicación, que serán los mismos que los que proporcione la fuente de vídeo.

5.4.2. Submódulo de eventos de entrada

El submódulo de eventos de entrada detecta la pulsación de teclas de diversos dispositivos de entrada como teclados o ratones. Estos eventos pueden ser de distintos tipos, como *pulsación de tecla* y *liberación de tecla*.

Se implementa en la clase *InputEventController* (directorio *Controllers*). Para ello utiliza la biblioteca SDL. Este submódulo se utiliza para implementar la funcionalidad de MLBSensorKeyboard, mediante el patrón *observador*. Normalmente los sensores implementan su propia lógica de evaluación, sin embargo los MLBSensorKeyboard carecen de dicha lógica. Funcionan *avisando* a este controlador de la tecla que deben monitorizar, y si se produce una pulsación de dicha tecla es el propio controlador quien activa el sensor. Por tanto este controlador mantiene una lista en forma de *std::vector* de los MLBSensorKeyboard definidos por el usuario. Para poder publicar un sensor de este tipo en el controlador se utiliza la función pública *addMLBSensorKeyboard*.

Minerva da soporte para eventos del tipo presión de una tecla (*Keydown*), la liberación (*Keyup*) y la terminación de la aplicación. El diseño está preparado para soportar otro tipo de eventos como pulsación de teclas de ratón o joystick.

La gestión de eventos de entrada se realiza mediante una cola implementada en la biblioteca SDL. En cada frame se analizan todos los elementos acumulados en dicha cola. Cada uno de estos elementos es un evento. En el análisis se comprueba el tipo y las características de cada evento, y se actúa en consecuencia. En la Figura 5.18 se muestra en pseudocódigo el bucle de consumo de eventos.

5.4.3. Submódulo de audio

Este submódulo da soporte para la reproducción de ficheros de audio almacenados en el disco duro. Sirve para la implementación de la funcionalidad del MLBActuatorSound (ver Sección 5.2).

Se basa en el uso de la biblioteca *SDL_Mixer*. Su funcionamiento está dividido en dos partes: *inicialización del subsistema de audio* y *carga y reproducción de audio*. A continuación se detallan estas dos partes:

```
SDL_Event event;
while(SDL_PollEvent(&event)){
    switch(event.type){
        case SDL_MOUSEBUTTONDOWN:
        /* Se ha presionado un boton del raton */
        break;
        case SDL_MOUSEBUTTONUP:
        /* Se ha liberado un boton del raton */
        break;
        case SDL_KEYDOWN:
        /* Se ha presionado una tecla */
        break;
        case SDL_KEYUP:
        /* Se ha liberado una tecla */
        break;
        case SDL_QUIT:
        /* Se ha pedido el cierre de la aplicacion */
        break;
    }
}
```

Figura 5.18: Pseudocódigo del bucle de consumo de eventos basado en la biblioteca SDL.

Inicialización

La inicialización del subsistema de audio basado en la biblioteca *SDL_Mixer* la realiza la clase *World* (directorio *Kernel*) mediante la función *initWorld*. La función que lo inicializa es:

```
Mix_OpenAudio(MIX_DEFAULT_FREQUENCY, MIX_DEFAULT_FORMAT,
MIX_DEFAULT_CHANNELS, 4096)}.
```

Esta biblioteca da soporte para audio en formato *.wav*, pero está planeado como trabajo futuro el incorporar soporte para formatos con mayor compresión como *.mp3* y *.ogg*.

Carga y reproducción de audio

Los ficheros de audio deben cargarse a una estructura del tipo *Mix_Chunk* de la biblioteca *SDL_Mixer* para poder ser utilizados por esta. Cada MLBActuatorSound se encarga de la carga de estos ficheros conocida su ruta. La función que realiza dicha carga es:

```
_chunk = Mix_LoadWAV(path.c_str());
```

Aparte hay que realizar algunas configuraciones como ajustar el volumen del *chunk*. Por defecto, Minerva siempre carga el máximo volumen disponible.

Para la reproducción del sonido simplemente hay que ejecutar la función:

```
Mix_PlayChannel(-1,_chunk,0);
```

Con los parámetros indicados, el subsistema de audio se encarga de buscar el primer canal libre para reproducir el sonido. Esto proporciona la reproducción *simultánea* de varios sonidos, sin que haya que realizar explícitamente ninguna gestión de los canales de audio.

5.5. Módulo de registro

El módulo de registro tiene como objetivo realizar el registro (también conocido como *tracking*) de la realidad. Este registro consiste en conocer en todo momento la posición de la cámara virtual en el mundo real, para que el ajuste de los objetos virtuales con la imagen real sea lo más fiel posible.

Este submódulo obtiene la información visual del submódulo de captura de vídeo (ver Sección 5.4.1) para llevar a cabo los métodos de tracking visual. Una vez realizado el registro, la información obtenida se suministra directamente a cada componente de la aplicación mediante el módulo de componentes (ver Sección 5.2).

La arquitectura de Minerva ha sido diseñada para soportar más de un método de *tracking*, ejecutándose simultáneamente. La interfaz que crea y gestiona estos métodos la proporciona la clase *TrackingFactory*, que implementa el patrón *Factory* como su nombre indica. Se encarga de conocer cada uno de los métodos implementados en el sistema, y poder desplegar todos ellos de manera conveniente.

Para poder soportar distintos métodos de tracking se ha implementado una interfaz general mediante la clase *TrackingMethod* (directorio *Kernel*). Se trata de una clase abstracta que representa un método de *tracking*. Sirve como interfaz para implementar métodos de tracking, proporcionando el método virtual *pollMethod* y otras características como conocer si el método está o no activo.

Minerva por defecto implementa un método de registro visual basado en marcas, mediante el *submódulo de detección de marcas*. Este submódulo se explica en la siguiente sección.

5.5.1. Submódulo de detección de marcas

La clase *TrackingMethodARTK* (directorio *Kernel*) implementa la funcionalidad de detección de marcas basándose en la biblioteca ARToolKit. Esta clase hereda de la anteriormente explicada *TrackingMethod*. Las tareas que lleva a cabo son:

- **Inicializar ARToolKit:** para utilizar la funcionalidad de detección de marcas de esta biblioteca es necesario inicializarla con los parámetros intrínsecos del dispositivo de vídeo y su resolución. El código que realiza esta tarea está implementado en la función *initARTK* de *TrackingMethodARTK*, y se puede ver en la Figura 5.19.

```
void TrackingMethodARTK::initARTK() {
    ARParam cparam, wparam;
    //Camera parameters
#ifdef WIN32
    if (arParamLoad("data\\camera_para.dat", 1, &wparam) < 0) {
#else
    if (arParamLoad("./data/camera_para.dat", 1, &wparam) < 0) {
#endif
        Logger::getInstance()->error("Unable to init ARToolKit");
        deactive();
        return;
    }

    arParamChangeSize(&wparam, 640, 480, &cparam);
    arInitCparam(&cparam);

    Logger::getInstance()->out("ARToolKit loaded successfully!");
}
```

Figura 5.19: Código de inicialización de la biblioteca ARToolKit.

- **Gestión de las marcas del sistema:** los MAOMark y MAOMarksGroup de la aplicación se almacenan también en este método de tracking, mediante la interfaz proporcionada por las siguientes funciones *addMAOMark* y *addMAOMarksGroup*.

- **Detección de marcas:** al invocar al método público *pollMethod* se ejecuta el algoritmo de detección de marcas de ARToolKit. La función principal que lleva a cabo esta tarea es *arDetectMarker*, que necesita como parámetros el *thresold* para binarizar la imagen, el frame del dispositivo de vídeo, y las estructuras de datos proporcionadas por ARToolKit para guardar información interna *ARMarkerInfo* y *markerNum*.

Este método accede a los componentes MAO suscritos al método por las funciones indicadas anteriormente, y actualizan su información de posicionamiento directamente. Así, cada MAO es *responsable* de conocer de forma correcta y actualizada su posición, para poder suministrarla al resto de módulos que la necesiten, como el módulo de representación (ver Sección 5.3).

5.6. Módulo de procesamiento de lenguajes

Minerva basa la especificación de la lógica de las aplicaciones de Realidad Aumentada en un lenguaje de alto nivel denominado MSL (Minerva Specification Language). Este módulo realiza todas aquellas tareas relacionadas con la interpretación de ese lenguaje, detección de errores, declaración de componentes, etc.

Para añadir funcionalidad más avanzada, Minerva también da soporte para un lenguaje de script que accede de forma directa a los componentes de la aplicación para poder modificar-

los. Este módulo también se encarga de dar soporte al *binding* entre Minerva y el lenguaje de script.

Toda la información recabada durante el procesamiento de ambos lenguajes de suministra al módulo de componentes (ver Sección 5.2) para realizar las modificaciones deseadas. De esta forma el resto del sistema es independiente de este módulo, ya que únicamente utilizan la información almacenada directamente por los componentes MAO y MLB.

Este módulo se divide a su vez en dos submódulos:

1. *Submódulo de procesamiento MSL:* se encarga de la interpretación del lenguaje de especificación de alto nivel MSL, y de realizar los cambios que en este se propone.

2. *Submódulo de procesamiento de lenguaje de script:* proporciona todas las estructuras de datos y subsistemas necesarios para poder ejecutar código de un lenguaje de script ajeno a Minerva.

5.6.1. Submódulo de procesamiento MSL

El submódulo de procesamiento de lenguaje de MSL se encarga de implementar los analizadores necesarios para interpretar el lenguaje MSL y declarar los componentes que en él se especifican. En el Anexo A se explica paso a paso la utilización y la funcionalidad de este lenguaje.

El análisis del lenguaje se divide en dos analizadores:

1. *Analizador léxico:* realiza el análisis de los *tokens* (palabras) del lenguaje, e indica si son o no válidas.

2. *Analizador sintáctico:* comprueba que el *orden* de los tokens es el adecuado y corresponde con la gramática del lenguaje.

A continuación se explica en profundidad el funcionamiento de estos dos analizadores.

Analizador léxico

La especificación del analizador léxico del lenguaje MSL se ha realizado mediante la biblioteca *flex++*, y está contenida en el fichero *MSLScanner.l* (directorio *Kernel*). En el Anexo I se muestra la especificación en *flex++* completa. Contiene las expresiones regulares para identificar los *token* del lenguaje. Los *tokens* que identifica se pueden clasificar en los siguientes grupos:

- **Identificadores:** se corresponde con la expresión regular de un identificador de cualquier lenguaje de programación convencional. El primer carácter debe ser una letra (mayúscula o minúscula) seguido de caracteres, dígitos y/o el guión bajo.

- **Números:** el lenguaje es capaz de reconocer tokens de números enteros (con o sin signo) y decimales (con o sin signo).
- **Cadenas:** en Minerva las cadenas de texto comienzan y terminan con dobles comillas, mientras que los identificadores no.
- **Operadores específicos:** algunos operadores específicos son el operador flecha (->) y el operador punto (.). En el Anexo A se explica el uso de cada uno de ellos.
- **Palabras reservadas:** las palabras reservadas pueden ser desde nombres de componentes (MAO's y MLB's), de parámetros de estos componentes (*path*, *name*, *size*, *int*, etc) o valores para variables como *True* o *False*.
- **Token de comienzo y fin de comentario:** en Minerva los comentarios son multilínea y los *token* de principio y fin de comentario son /* y */ respectivamente.

Para obtener el código en C++ del escáner (*Scanner* en inglés) a partir de la especificación se utiliza el comando:

```
flex++ -d -oKernel/MSLScanner.cpp Kernel/MSLScanner.l
```

Analizador sintáctico

El analizador sintáctico comprueba que el orden de los *tokens* sea coherente y acorde con la gramática del lenguaje. Se ha implementado utilizando la biblioteca *bison++*, y la especificación del analizador está contenida en el fichero *MSLParser.y* (directorio *Kernel*). En el Anexo J se muestra la especificación completa.

La especificación sintáctica de un lenguaje es una parte muy importante de la facilidad de uso de la aplicación, ya que en el caso de Minerva es la interfaz con la que el usuario interactúa directamente. Es una de las partes más complejas de este proyecto.

La principal carga de trabajo reside en describir sintácticamente cada uno de los numerosos componentes (MAO's y MLB's) de los que está compuesto Minerva, cada uno con sus parámetros y sus particularidades. En la Figura 5.20 se puede ver un ejemplo de cómo se describe en *bison++* un componente MAO.

Las producciones van a la izquierda seguidas del carácter dos puntos (:). A continuación se escribe la parte derecha de la producción, que puede ser un símbolo terminal (en mayúsculas o entre comillas simples para un único carácter) o uno no terminal (en minúsculas). Este ha sido el convenio seguido para implementar la especificación sintáctica de MSL, aunque no es obligatorio.

Entre cada uno de los símbolos puede intercalarse código C++ entre llaves que se ejecutará en esa parte del análisis. Su objetivo es principalmente añadir los componentes a las factorías,

```
1  def_maomark: MAOMARK identifier '{' param_maomark
2      {currentMAO = &MAOFactory::getInstance()->addMAOMark(*$2,*$4->string1
           ,$4->float1);
3      ((MAOMark*)currentMAO)->setOffsetMatrix($4->pose1);delete $4;}
4      mao_properties mlbs links  ground '}' {}
5      ;

7  param_maomark:  PARAM_PATH '=' string PARAM_SIZE '=' float
       optparam_maomark
8      {$$ = new MSLProperties(*$7); $$->string1 = $3; $$->float1 = $6;
           delete $7;}
9      ;

11 optparam_maomark: PARAM_OFFSET '=' pose { $$ = new MSLProperties(); $$->
       pose1 = $3;}
12     | /* empty */ {$$ = new MSLProperties();}
13     ;
```

Figura 5.20: Especificación en bison++ de las producciones necesarias para declarar un MAOMark

pasar los datos encontrados en las hojas del árbol sintáctico a través de él (para ello se utiliza la clase *MSLProperties* que se describe más adelante) o incluso realizar comprobaciones *semánticas*.

El nombre de los símbolos no terminales que definen un componente comienza con el prefijo *def_*, el que define los parámetros obligatorios es *param_* y el que define los parámetros opcionales es *optparam_*. Este método ha sido utilizado en toda la especificación.

Como trabajo futuro sería conveniente añadir más flexibilidad a este lenguaje (actualmente, el orden de los parámetros al declarar un componente es importante, no es lo mismo indicar primero el parámetro *path* y luego el *size* que hacerlo al revés, por ejemplo). También sería conveniente añadir producciones de error y análisis semántico. De momento era una carga de trabajo desproporcionada en comparación con la funcionalidad que se quería conseguir con este proyecto.

Para obtener el código C++ del *parser* especificado en el fichero *.y* se utiliza el siguiente comando:

```
bison++ -d -hKernel/MSLParser.h -o Kernel/MSLParser.cpp
Kernel/MSLParser.y
```

Otra parte importante es la forma en la que se pasan los datos encontrados en el código de la aplicación escrita en MSL a través de todo el árbol del analizador sintáctico. A continuación se explica la solución aportada, junto a una descripción de la clase que realiza esta función.

MSLProperties es una clase auxiliar (directorio *Kernel*) utilizada durante el transcurso del análisis sintáctico del código fuente. Se utiliza para pasar los valores de las propiedades declaradas a lo largo del árbol sintáctico. Intuitivamente se puede representar como un *camión* que recoge todos los datos especificados por el usuario en el lenguaje, sea del tipo que sea, para pasarlos todos juntos al componente que se está definiendo.

Está compuesto por los siguientes campos:

- Cuatro campos *string*.
- Tres campos *float*.
- Cuatro campos *int*.
- Un campo *bool*.
- Dos campos *btVector*.
- Dos campos *Mat*.
- Dos campos *MAOValue*.
- Dos campos *MAOProperty*.

Además ha sido necesaria la implementación de un constructor de copia para pasar una copia de la clase por a través del árbol. Otro método interesante es *fill*. Este método toma como parámetro otro objeto de la clase *MSLProperties*, obtiene sus datos, y los añade al objeto del que se está invocando la función *fill*.

En la siguiente sección se explica la implementación y funcionamiento del lenguaje de script soportado por Minerva para el acceso a funcionalidades avanzadas de la arquitectura.

5.6.2. Submódulo de procesamiento de lenguaje de *script*

Este submódulo da soporte para la interpretación y ejecución de código en un lenguaje de *script*, de proporcionar a este una API para acceder a la características avanzadas del funcionamiento de Minerva.

En la arquitectura de Minerva se ha implementado soporte para el lenguaje de programación Python. A continuación se hace una introducción sobre qué significa utilizar un lenguaje de script, para más adelante profundizar en la implementación concreta de la API de Minerva (*MGE* - Minerva Game Engine) para Python.

Utilizar dos lenguajes de programación (como en el caso de Minerva que se utiliza C++ y Python) en una misma aplicación se puede llevar a cabo de dos maneras:

1. **Extendiendo:** extender Python con C++ significa utilizar bibliotecas y funciones de este último desde un programa de Python. De esta forma el programa principal está hecho en Python, pero puede aprovechar bibliotecas de otros lenguajes sin perder tiempo en portarlo, o simplemente usarlas porque son más eficientes.

2. **Embebiendo:** embeber Python en otro lenguaje como C++ significa poder ejecutar código Python desde otro lenguaje. El programa principal en el caso de Minerva está escrito en C++ y puede, eventualmente, ejecutar código Python. El código ejecutado de Python no tiene por qué ser independiente de las estructuras de datos desarrolladas en C++. En Minerva, por ejemplo, se pueden llamar a funciones de C++ desde el código de Python, que es el objetivo principal del scripting. Minerva implementa esta opción. En [emb] se encuentra mucha documentación sobre cómo embeber Python.

Este submódulo se divide está compuesto por dos componentes principales:

1. *Módulo MGE:* implementación del módulo accesible por Python para usar la funcionalidad avanzada de Minerva.

2. *Binding:* componente que lleva a cabo todas aquellas necesarias para poder ejecutar código Python desde Minerva, y utilizar estructuras de datos comunes.

A continuación se describen la implementación concreta de cada uno de estos submódulos.

Módulo MGE

El módulo principal para la *API* de Python se denomina MGE y está implementado en la clase *MGEModule* (directorio *MPY*). Al igual que en lenguajes como C o C++ se añade nueva funcionalidad mediante el uso de #include, en Python se utilizan los denominados *módulos* para importar nuevas bibliotecas. En el caso de Minerva es este módulo el que permite a Python acceder a la funcionalidad de su arquitectura.

MGEModule proporciona una única función a la API de Python, la cual es:

```
1 getCurrentController() //Nombre para Python
2 mPyGetCurrentController() //Nombre para C++
```

A partir del objeto MLBController devuelto por esa función se puede acceder al resto de componentes y funcionalidades de la API. En el Anexo F se detalla la lista completa de objetos y funciones soportados.

```
Logger::getInstance()->out("Initializing python subsystem");
PyImport_AppendInittab("MGE", &initMGE);
Py_Initialize();
main_module = object((handle<> (
        borrowed(PyImport_AddModule("__main__")))));
main_namespace = main_module.attr("__dict__");
mge_module = object((handle<> (PyImport_ImportModule("MGE"))));
(main_namespace)["MGE"] = mge_module;

/* Inserting objects! */
scope(mge_module).attr("mge") = ptr(&mge);

/* This is the last thing to do! */
PyRun_SimpleString("from MGE import *");

/*-------- Importing constants! --------*/
/* Anim Orej! */
PyRun_SimpleString("MGE_PLAY=0");
PyRun_SimpleString("MGE_PAUSE=1");
PyRun_SimpleString("MGE_STOP=2");
PyRun_SimpleString("MGE_SIMPLE=0");
PyRun_SimpleString("MGE_LOOP=1");
PyRun_SimpleString("MGE_PINGPONG=2");
/* Change pose! */
PyRun_SimpleString("MGE_LOCAL=0");
PyRun_SimpleString("MGE_GLOBAL=1");
/*ActuatorProperty!*/
PyRun_SimpleString("MGE_ASSIGN=0");
PyRun_SimpleString("MGE_ADD=1");
PyRun_SimpleString("MGE_MINUS=2");
PyRun_SimpleString("MGE_MULTIPLY=3");
PyRun_SimpleString("MGE_DIVIDE=4");
```

Figura 5.21: Código de inicialización del intérprete de Python basado en la biblioteca *Boost-Python*.

Binding

Minerva implementa una interfaz general para acceder a la funcionalidad del intérprete de Python, llevar a cabo su inicialización y destrucción y ejecutar código en él. Esta clase es *MPYWrapper* (directorio *MPY*).

MPYWrapper se encarga de inicializar todo el subsistema de ejecución de código Python, crea el módulo de Python y ejecuta los scripts, todo ello basándose en la biblioteca *Boost-Python*. A continuación se describe más en detalle cada una de estas funciones.

La inicialización del subsistema de intérprete de Python se realiza al comienzo de la aplicación. MPYWrapper ofrece la función *initPython* que inicializa el intérprete embebido de Python y registra el módulo MGE. También registra algunas constantes utilizadas en la API. En la Figura 5.21 se puede ver el código de inicialización.

La primera parte del código registra la clase MGEModule con el nombre *mge* para su

utilización desde Python. Además ejecuta el código *from MGE import* * para que no sea necesario importarlo en el script.

Por último registra varias constantes utilizadas en componentes MLB, ejecutando código simple declarando esas variables.

Una cuestión importante es la correspondencia entre los tipos de datos de los dos lenguajes de programación (también denominado *mapeo* de datos). Boost-Python por defecto proporciona mapeados los tipos más comunes como *int, float, string,* etc., pero el resto deben ser mapeados.

A continuación se describe cómo se ha realizado el mapeo de la clase *MAO* en Python. Esta clase ofrece en Python la función *getName*. El código de esta función es el que sigue:

```
1 std::string MAO::getName() {
2     return _name;
3 }
```

Al tratarse de tipos de datos ya mapeados (en este caso *string*) no es necesario definir nada más. Sin embargo, si se devuelve un objeto de una clase no mapeada por Boost-Python (como puede ser una clase de Minerva) el código es más complejo.

El siguiente ejemplo es la implementación de la función *mPyGetProperty(string name)*, correspondiente con la función *getProperty(string name)* de la API de Python de la clase MAO. El valor devuelto es un objeto del tipo MAOProperty, no mapeado por Boost-Python. El siguiente es el código en C++ de esta función:

```
1 PyObject* pyObj = NULL;
2 MAOProperty& prop = getProperty(name);
3
4 MPYPropertyInt& p = getMPYPropertyInt(name);
5 boost::python::reference_existing_object::apply<MPYPropertyInt*>::type
6         converter;
7 pyObj = converter(&p);
8
9 return boost::python::object(boost::python::handle<>(pyObj));
```

Este es el caso específico de una propiedad del tipo *MAOPropertyInt*. Una vez obtenido el objeto de la propiedad (en este caso *p*), es necesario crear un *converter* de Boost-Python del tipo MPYPropertyInt para transformarlo en un objeto de Python (*PyObject*) y poder devolverlo. MPYPropertyInt es la equivalencia en Python del MAOPropertyInt de C++.

Es interesante disponer mapeados también tipos de dátos *estándar* como los *std::vector* de C++ en Python. Para ello hay que hacer el mapeo específico. En la clase *WrapperTypes* (directorio *MPY*) se define una clase *template* llamada *vector_item* que contiene el *binding* de los métodos más comunes de una lista de Python (*set, del, add*, etc) con las operaciones de la clase *std::vector*.

Una vez definida esta clase *template*, se particulariza en las listas más comunes que se utilizarán. Estas son:

```
typedef std::vector<std::string> VectorStr;
typedef std::vector<int> VectorInt;
typedef std::vector<float> VectorFloat;
```

Una clase *Vectorxxx* con las mismas operaciones que las listas de Python pueden ser registradas como una clase normal. El siguiente es el ejemplo de registro de la clase *VectorStr*:

```
object VectorStr_class = class_<VectorStr>("VectorStr")
.def("__len__",&VectorStr::size)
.def("clear",&VectorStr::clear)
.def("append",&vector_item<std::string>::add,
    with_custodian_and_ward<1,2>())
.def("__getitem__",&vector_item<std::string>::get,
    return_value_policy<copy_non_const_reference>())
.def("__setitem__",&vector_item<std::string>::set,
    with_custodian_and_ward<1,2>())
.def("__delitem__",&vector_item<std::string>::del)
.def("__iter__",boost::python::iterator<VectorStr>());
```

Una vez inicializado el intérprete y el módulo, es necesario registrar las clases y las funciones a las que podrá acceder la API. Para que una clase pueda ser accedida desde Python no es necesario hacer nada especial, simplemente definir de una determinada manera las funciones a las que podrá acceder. En el siguiente código se muestra el registro de la clase MAO junto a sus funciones:

```
BOOST_PYTHON_MODULE(MGE){
    /*--- MAO --*/
    object MAO_class = class_<MAO>("MAO",no_init)
    .def("getName",&MAO::getName)
    .def("getProperty",&MAO::mPyGetProperty)
    ;

    //Registro del resto de funciones
}
```

Como se puede apreciar, se utiliza la macro de Boost BOOST_PYTHON_MODULE para registrar clases y funciones. Además se usa la técnica de *method chaining*.

La ejecución de los scripts lo realiza esta clase por medio de la función *runScripts*. Esta función pide a la MLBFactory todos los MLBControllerScripts, obtiene el código de su script de Python y los ejecuta por medio de la siguiente función:

```
// "o" es el objeto compilado del script
handle<> ignore((PyEval_EvalCode((PyCodeObject*) o->ptr(), main_namespace
    .ptr(), main_namespace.ptr())));
```

```
1 void Logger::error(std::string msg) {
2     _logFile << "[ERROR]" << msg << std::endl;
3 #ifdef WIN32
4     std::cout<<"[ERROR] "<<msg<<std::endl;
5 #else
6     std::cout << "[\x1b[31m\x1b[1m" <<"ERROR" << "\x1b[0m] "<<msg<< std::
        endl;
7 #endif
8 }
```

Figura 5.22: Código de la función *error* de la clase Logger.

5.7. Módulo de depuración

Minerva dispone de un módulo de depuración para notificación de cualquier situación anómala o con un mínimo de interés que se dé durante la ejecución de una aplicación.

Cualquiera del resto de los módulos que componen la arquitectura de Minerva puede hacer uso de él. Se divide en dos subsistemas:

1. *Submódulo de log:* crea un registro (*log* en inglés) de situaciones anómalas que se dan en el sistema.

2. *Submódulo de excepciones:* gestiona excepciones para manejar errores durante la ejecución de la aplicación, e intentar tratarlas de modo que el sistema no pierda su estabilidad.

A continuación se detallan en profundidad estos dos subsistemas:

Submódulo de log

El submódulo de log se utiliza en Minerva para unificar todas aquellas situaciones de interés que se den durante la ejecución de la aplicación.

Se implementa mediante la clase *Logger* (directorio *Kernel*). Esta clase proporciona un mecanismo básico de *logging*, para registrar *salidas normales*, *advertencias* y *errores*. Estos mensajes se escriben en un fichero llamado *log*, y además las muestra por consola.

Tiene partes de código condicional multiplataforma como el mostrado en la Figura 5.22.

Esto se debe a que GNU/Linux soporta coloreado de texto en terminal utilizando códigos *ANSI C,* pero las plataformas Microsoft Windows no. Las salidas de errores se colorean en rojo y negrita, y las advertencias en amarillo y negrita, mientras que las salidas normales no reciben ningún tipo de formateo especial.

Submódulo de excepciones

La arquitectura de Minerva gestiona los errores producidos mediante el uso de excepciones de *C++*, en vez de utilizar códigos de error u otro tipo de mecanismos.

Las excepciones son del tipo *const char**, conteniendo un mensaje sobre la naturaleza del error. Estas excepciones se lanzan en caso de que la aplicación no se pueda recuperar. Si es un fallo menor, se lanzará un *warning* y se continuará la ejecución de la mejor forma posible.

CAPÍTULO 6

EVOLUCIÓN Y COSTES

En el presente capítulo se describirán las diferentes fases empleadas en el desarrollo de Minerva, especificando los hitos característicos de cada una, y aportando datos relacionados con la complejidad y el coste temporal de cada una. Igualmente se aportará información del rendimiento (*profiling*) del sistema en diferentes situaciones, así como algunas comparativas con aplicaciones demostrativas que no han sido desarrolladas utilizando Minerva.

6.1. Evolución del proyecto

Antes del comienzo de la etapa de desarrollo (*18 de Febrero*), se inició una etapa de análisis de requisitos, tanto tecnológicos como funcionales, de la arquitectura de Minerva. Se elaboraron una serie de documentos que describían de forma general la arquitectura orientada a componentes, una primera lista de su repertorio, y algunas aplicaciones cuya funcionalidad se quería cubrir. A continuación se consiguieron los recursos hardware para el desarrollo y pruebas, y se comenzó su elaboración mediante una serie de iteraciones. A continuación se describen en detalle cada una de estas fases.

6.1.1. Concepto del software

El laboratorio de *Oreto* ha trabajado con sistemas de Realidad Aumentada desde hace varios años, y ha participado en proyectos de cierta envergadura como HESPERIA (Proyecto CENIT 2007/2010) y ELCANO (cátedra Indra-UCLM 2011/2012). Fruto de esta experiencia surge la idea de crear un sistema que facilite el desarrollo de pequeños prototipos funcionales de RA, ya que se debía empezar prácticamente desde cero una aplicación cada vez que cambiaban sus requisitos.

Además sería deseable que el sistema pudiera extenderse a usuario con bajos conocimientos en programación e informática gráfica. Un perfil de usuario potencial de Minerva son los diseñadores gráficos y modeladores 3D, quienes podrían crear, además de los elementos de despliegue de este tipo de aplicaciones, el código necesario para la construcción de la lógica de aplicaciones.

Minerva se inspiró desde el comienzo en el *Blender GameKit*, capaz de crear juegos muy potentes como *YoFrankie!* con un intuitivo sistema basado en sensores, controladores y actuadores.

Tras comprobar que no existía ningún sistema con las características de Minerva, se comenzó la etapa de análisis de requisitos. Esta etapa se explica a continuación.

6.1.2. Análisis preliminar de requisitos

Después de determinar el concepto del sistema, se dedicaron dos semanas a analizar la lista de requisitos y objetivos que debía cumplir el sistema.

Para comenzar, se establecieron una serie de requisitos básicos generales que debían ser satisfechos. Estos objetivos eran:

1. Debe estar basado en bibliotecas y estándares libres.
2. Debe ser fácilmente compilable. Ello implicaba usar las versiones estables de las bibliotecas necesarias, y de uso más extendido entre las distribuciones GNU/Linux.
3. Minerva debe ser multiplataforma, contando en principio con GNU/Linux y Microsoft Windows. Esto implicaba utilizar únicamente bibliotecas que fuesen soportadas en ambas plataformas.

A continuación se elaboró una lista de componentes MLB (*Sensores, Controladores Actuadores*) inicial. Se partió del repertorio disponible del *Blender GameKit*, adaptándolos a las necesidades de una aplicación de Realidad Aumentada, y descartando los que no eran aplicables. Esta lista fue lo suficientemente detallada como para saber qué parámetros exactos necesita, a qué objetos se podía aplicar o su funcionalidad exacta.

El siguiente paso fue determinar las características multimedia de Minerva. Se comenzaron a esbozar los diferentes módulos y submódulos que conformarán la arquitectura. Minerva debe poder obtener *frames* de varias fuentes de vídeo simultáneamente, dibujar gráficos 3D importando objetos generados con herramientas de terceros y primitivas, gráficos 2D como imágenes y texto, detección de eventos de periféricos de entrada y la reproducción de audio.

Se elaboraron diversos documentos con LaTeXpara que sirvieron como retroalimentación para la etapa de desarrollo.

6.1.3. Diseño general

El diseño de Minerva se planteó de forma modular y extensible. Dada su naturaleza orientada a componentes, era interesante poder ampliar el repertorio de forma sencilla, proporcionando una interfaz que esté integrada en el resto de la plataforma. Para ello se elaboró la clasificación en módulos y submódulos (ver Capítulo 5), siendo cada uno de ellos lo más independiente posible del resto y de las bibliotecas en las que se basaban. Esto aumenta la complejidad del código pero tiene la ventaja añadida de que si en el futuro es necesario cambiar una biblioteca, el cambio sea lo más transparente posible para el resto de módulos.

Se han aplicado patrones de diseño ampliamente utilizados en *Ingeniería del Software*, como el patrón *Factory*, *Controller*, *Observador* o *Singleton*. Estos patrones han sido de mucha ayuda para crear una arquitectura modular y un código legible y estructurado.

6.1.4. Iteraciones

En esta sección se muestra la división en iteraciones seguidas durante el desarrollo de Minerva, describiendo los hitos conseguidos en cada una de ellas, la complejidad, el esfuerzo y detalles especialmente relevantes.

Iteración 1

En esta primera iteración se comenzó construyendo la estructura básica de directorios y *Makefile*'s para albergar el proyecto. Desde el principio se tuvo conciencia de que el proyecto era de una envergadura considerable, y se hacía necesario poder mantener el código fuente bien organizado para cumplir el objetivo de desarrollar una arquitectura modular. También se crearon los primeros documentos explicativos de cómo compilar y usar Minerva, de cara a tenerlo actualizado durante el desarrollo.

Se crearon las primeras versiones de las *Factories* (*TrackingMethodFactory, VideoFactory, MAOFactory* y *MLBFactory*), los *Controllers* (*InputEventController, GameLogicController*), se implementó el patrón *Singleton*, el *Logger*. También se implementó un primitivo submódulo de representación 2D para dibujar de fondo los frames del dispositivo de vídeo.

Se implementaron algunos componentes MAO para probar el enfoque del submódulo de componentes, junto a la base para la creación propiedades de usuario.

Además se añadió como biblioteca externa *ARToolKit*, y se implementó la interfaz *TrackingMethodFactory*.

Como resultado, la temprana arquitectura podía obtener vídeo, dibujarlo de fondo, y detectar marcas, todo ello de forma modular. Las marcas ya se añadían como componente MAO de forma correcta. Sin embargo, todo se hacía en tiempo de compilación, todavía no existía la interfaz de alto nivel accesible mediante MSL.

El objetivo general de esta primera iteración puede ser resumido como la creación del esqueleto básico necesario para poder desarrollar la funcionalidad de especificación lógica deseada de Minerva.

6.1.5. Iteración 2

Una vez establecidos los subsistemas básicos de una aplicación de Realidad Aumentada (vídeo y *registro* de la realidad), se procedió a implementar la lógica de funcionamiento de los componentes MLB. En esta segunda iteración se implementa un subconjunto reducido de Sensores, Controladores y Actuadores básicos, junto al cuerpo de *GameLogicController* para su evaluación y comprobar la validez de sus relaciones.

Igualmente en esta iteración, se implementa el dibujado de objetos 2D (*MAORenderable2D)*, y 3D (*MAORenderable3DOrej* y *MAORenderable3DTeapot).*

El sistema resultante es capaz de añadir a la escena objetos virtuales asociados a marcas, y también evalúa la lógica temprana de los componentes MLB.

La complejidad de esta segunda iteración residió en comprender e implementar la lógica de evaluación de sensores, de controladores y activación de actuadores. También fue necesario invertir esfuerzo en adaptar la versión del importador OreJ existente (escrito en C) a Minerva, por lo que hubo que reescribirlo en C++. Se corrigieron errores de optimización en el subsistema de representación para realizar las operaciones de carga y despliegue de un modo más eficiente desde el punto de vista del tiempo de cómputo.

6.1.6. Iteración 3

El esfuerzo en este momento se centró en dar soporte para la reproducción de ficheros de audio, para la implementación específica del *MLBActuatorSound*. Hubo que estudiar muchas bibliotecas, y aprender conceptos nuevos relacionados con audio digital. El soporte se dio para ficheros *.wav*, formato crudo sin compresión. Esta se consideró una característica muy interesante para la realización de aplicaciones con una fuerte componente multimedia.

También se concentró la atención en la implementación de los componentes necesarios para el cálculo del posicionamiento relativo entre sistemas de marcas que sería utilizado en uno de los demostradores de la plataforma: *ARPaint*. Se implementó el actuador *MAOActuatorRelativePose*, que calcula la posición relativa en referencia a un centro de coordenadas local. Este actuador indica al *lienzo* dónde estan los puntos de la aplicación.

La arquitectura en esta tercera iteración es capaz de ejecutar una de las demos de forma modular, y consigue reproducir audio utilizando la especificación de sensores, controladores y actuadores.

Estudiar los conceptos de audio fue la parte más compleja fue la comparación de las bibliotecas de audio, teniendo los conceptos básicos en mente, y proporcionar una forma sencilla e intuitiva de crear la demo *ARPaint*. También se siguió la implementación de otros componentes como el *MLBActuatorNear* o el *MLBActuatorDistance*.

6.1.7. Iteración 4

En esta cuarta iteración, la base de especificación lógica basada como sistema SCA está en una fase muy madura. La atención se centró en la implementación del soporte de simulación física. Desde el principio se consideró *Bullet* como una alternativa sostenible al ser multiplataforma, software libre y su calidad estaba más que respaldada (es el motor escogido por la suite de modelado 3D Blender).

Se dedicaron unos días únicamente al estudio de dicha biblioteca, de su funcionamiento, interfaz, y la manera de integrarlo en el sistema ya existente. Se comenzó el desarrollo del *PhysicsController*, dando soporte a poder realizar simulación física completa o parcial, únicamente habilitando la detección de colisiones.

Se implementó en el importador de OreJ la posibilidad de detectar automáticamente la forma de colisión como malla de triángulos (la más precisa posible) o de otras formas primitivas como la esférica, cilíndrica o en forma de caja rectangular (*bounding box*).

También se implementó la funcionalidad del *MLBSensorCollision*, que aprovechaba esta capacidad del *PhysicsController* de notificar colisiones sin activar la funcionalidad de la simulación física.

Minerva era capaz de establecer una marca (o en general un *MAOPositionator3D*) como *Ground* de la simulación física, y someter a ella los *MAORenderable3D*.

La parte más crítica fue idear la manera más sencilla de proporcionar al usuario la potencia de un motor de simulación física sin requerir un trabajo complejo. Para ello se ideó el concepto de *roles* en el ámbito de los componentes físicos. Un objeto podía actuar como objeto estático o dinámico, cubriendo las posibilidades de utilización de esta característica. Por otro lado también se pensó el rol de *Ground* por parte de un *MAOPositionator3D* del mundo físico.

6.1.8. Iteración 5

En este momento comienza el desarrollo del intérprete del lenguaje MSL. El sistema está en un estado bastante maduro, y tiene implementadas de forma robusta muchas de las características multimedia previstas.

Se estudia el uso de distintas bibliotecas de generación de analizadores, tanto léxicos como sintácticos. Se opta por *Bison++* y *Flex++*. Existen las versiones para C, pero se opta por mantener la coherencia y utilizar únicamente C++ siempre que esté disponible.

El esfuerzo en diseñar el lenguaje MSL persiguiendo su sencillez e intuitividad es considerable. En esta iteración fue indispensable repasar conceptos estudiados en la asignatura de *Procesadores de Lenguajes* relacionados con los analizadores léxicos y sintácticos.

Es necesario crear una estructura que consiga transportar los datos encontrados a lo largo del árbol sintáctico, y se estudia una forma que sea sencilla, elegante, pero sobretodo *eficiente*. Se implementa *MSLProperties* como solución a estos requisitos.

Se realizan multitud de para validar un amplio rango de situaciones posibles de entrada del intérprete, y cubrir el máximo posible de la especificación sintáctica.

Aprovechando que ya es posible la escritura de código MSL, se crea un *major-mode* para el marcado de sintaxis en *emacs*, que lo hace mucho más legible.

Minerva empieza a considerarse un sistema que cumple todos los objetivos básicos propuestos. Es capaz de añadir componentes a través de un lenguaje de alto nivel, especificar lógica con los MSL, crear propiedades de usuario y someterlos a simulación física.

6.1.9. Iteración 6

En esta iteración se quiso cubrir otros objetivos de las demos planeadas. Era necesario poder crear en tiempo de ejecución tantas copias de un MAORenderable3D como se quisiera, por lo que se diseñaron los MAO's instanciados. Era necesario proporcionar un mecanismo de *copia* y gestión de esas instancias, y proporcionar una interfaz sencilla para ello. Esta interfaz fue el MLBActuatorAddDynamicObject.

Fue necesario pensar los requisitos necesarios para que funcionasen este tipo de MAO's, como que el MAO padre debe ser un objeto físico dinámico, y que pueden tener un tiempo de vida (para no colapsar la memoria o que existan para siempre sin ningún propósito más).

También se implementó el algoritmo para el renderizado de sombras sobre el *Ground* de la simulación física.

Se siguió mejorando y depurando el lenguaje MSL, y testeando el sistema completo.

En ese momento se pudo implementar la demo *ARSheep*, utilizando la reciente incorporación de los MAO's instanciados y la simulación física. Además, era posible realizarlo a través de MSL.

6.1.10. Iteración 7

En esta iteración se propuso implementar completamente la integración de la arquitectura con un lenguaje de script. Una vez estudiadas las alternativas, se escogió Python por su sencillez y amplia difusión, frente a lenguajes como *Lua*. Se dedicó tiempo a conocer las diferentes bibliotecas y métodos que permiten esta incorporación, lo cual resultó un trabajo más complejo del inicialmente previsto.

Se crea el directorio *MPY* en la estructura para albergar todo lo relacionado con el *scripting* en Minerva. Se desarrolla el módulo Minerva Game Engine (MGE) con una función de prueba para acceder desde un script.

Es necesario mapear distintos tipos de datos de la arquitecura para poder utilizarlos en Python, como vectores de enteros o float. Además, la implementación de las propiedades de usuario requería realizar un *Proxy*. Se implementa la clase MPYProperty que realiza esta tarea.

Para poder utilizar el scripting desde MSL se crea el MLBControllerScript, que carga un script desde una ruta dada, lo compila y lo sirve al subsistema de intérprete de lenguaje de script.

En esta séptima iteración, Minerva ya es completamente funcional bajo sistemas GNU/-Linux. Se terminan de desarrollar el resto de demos, arreglando pequeños fallos detectados y mejorando aspectos del lenguaje y del resto de la arquitectura.

Esta iteración resultó ser una de las más complejas de todas, debido a la dificultad de poder *embeber* un lenguaje de programación en otro, y de la falta de experiencia realizando esta tarea. El hecho de existir muy diversos métodos de embeber un lenguaje agravó el problema. Además, la biblioteca escogida (*Boost-Python*) funciona utilizando diversas *macros* y objetos *proxy*, por lo que el uso no fue tan directo como el de otras bibliotecas.

6.1.11. Iteración 8

Una vez que el Minerva ya cumple toda la funcionalidad mediante el lenguaje MSL, el último paso es hacerlo compilable en la plataforma Microsoft Windows y añadir características para facilitar su uso.

Se escogió Visual Studio 2009 como compilador para Microsoft Windows, tras barajar posibilidades como *mingw* o *cygwin*. Hubo que conseguir las bibliotecas compiladas para la plataforma específica y aprender a utilizar el nuevo compilador, que funcionaba de forma ligeramente diferente. Los cambios en el código fueron mínimos gracias a la elección de bibliotecas que no fueran dependientes de la plataforma.

Se añadieron características para independizar el directorio en el que se ejecute y evitar problemas relacionados con las rutas relativas. Los ficheros importantes de Minerva se establecen en un directorio general, y se solucionan todos los bugs relacionados con esta cuestión.

Se crean también el instalador para *Windows* y el paquete *.deb* para facilitar la instalación en entornos *GNU* basados en *Debian*.

En esta iteración, el desarrollo de Minerva ya se considera finalizado y se continúa únicamente generando la documentación relativa a las demostraciones y al manual de usuario.

6.2. Recursos y costes

En esta sección se enumeran y describen los diferentes recursos utilizados, tanto temporales como económicos (para estos últimos se realiza una estimación). También se ofrece una comparativa con otras aplicaciones de Realidad Aumentada realizadas sin el apoyo de Minerva.

6.2.1. Coste económico

El periodo de implementación del presente Proyecto de Fin de C arrera comprende del 22 de Febrero, al 18 de Mayo principalmente. El trabajo de implementación se realizó durante un periodo de 15 semanas, con una estimación de dedicación diaria de 8 horas (y 6 días a la semana), lo que suma un total de 720 horas de implementación aproximadas [1]. Se ha considerado un precio de *30 €/hora* (tomando como referencia la web *http://www.infolancer.net*).

En la Taba 6.1 se muestra el desglose económico de todos los recursos utilizados en el desarrollo de Minerva.

Recurso	**Cantidad**	**Coste**
Sueldo programador	1	21600 €
Sony VAIO VGN-C1Z	1	789,01 €
Webcam Logitech Sphere AF	1	115 €
Total		22504,01 €

Cuadro 6.1: Tabla de desglose económico del precio de Minerva.

6.2.2. Estadísticas del repositorio

Para el control de versiones se ha utilizado un repositorio *Mercurial* ofrecido por un servidor de Oreto. En la Figura 6.1 se muestra un gráfico de la evolución de las líneas de código fuente durante toda la etapa de desarrollo.

Se ha utilizado la herramienta *cloc* para contabilizar las líneas de código fuente, asociado a los lenguajes en los que está programado Minerva. Hay que tener en cuenta que dicha herramienta contabiliza todas las líneas de código del proyecto, también las autogeneradas. Para la generación del analizador léxico (*Scanner*) y el analizador sintáctico (*Parser*), las herramientas para tal propósito (*flex++* y *bison++*, respectivamente) generan las clases en *C++* a partir de la especificación particular de cada una. Este código supone aproximadamente *4.500 líneas* del total.

En la Tabla 6.2 se observan los resultados arrojados por *cloc*.

[1] En estos cálculos no se ha tenido en cuenta el tiempo dedicado a la documentación del PFC, de unas 280 horas aproximadamente.

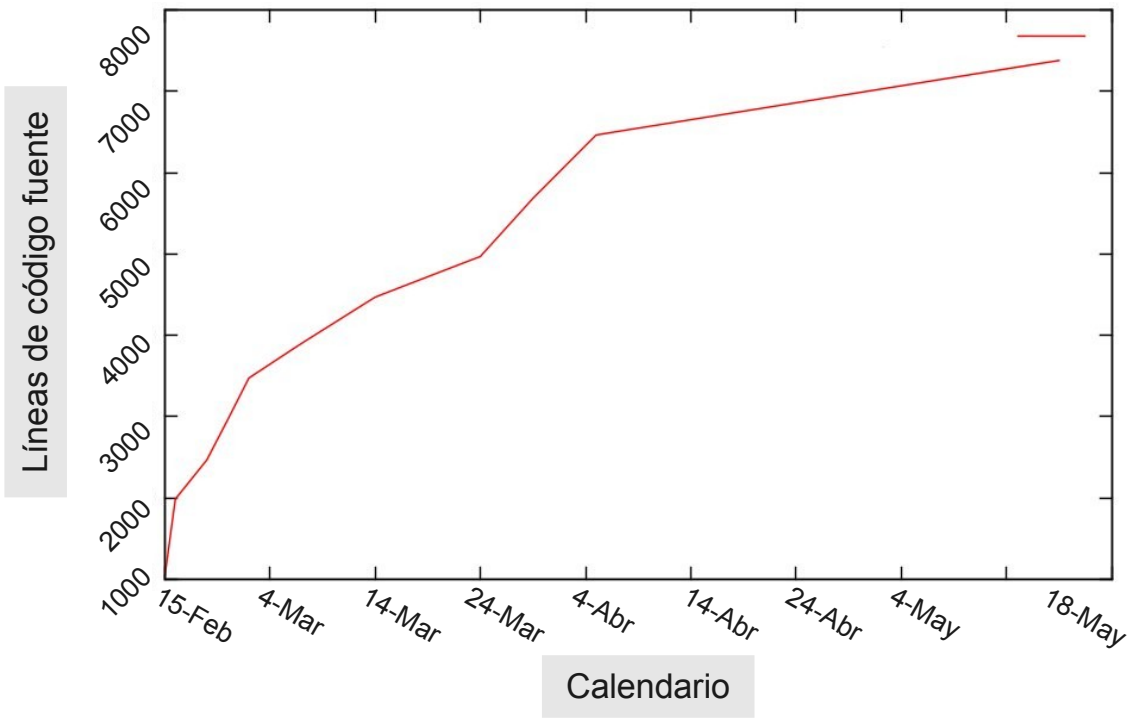

Figura 6.1: Estadísticas de actividad en el repositorio.

Lenguaje	Archivos	Espacios en blanco	Comentarios	Líneas de código
C++	68	1874	952	9257
Cabeceras C/C++	67	600	487	2238
bison	1	110	0	560
lex	1	15	1	109
make	1	18	10	59
Total:	**138**	**2617**	**1450**	**12223**

Cuadro 6.2: Número de líneas de código fuente de Minerva.

6.2.3. Comparativas

Se han realizado varias comparativas entre dos demos realizadas con Minerva y sus equivalentes programadas únicamente en C con *ARToolKit*.

Las dos demos escogidas han sido *ARSimple* y *ARPaint*. En la Tabla 6.3 se puede ver la comparativa de líneas entre las dos aplicaciones.

Se puede observar que en el *ARPaint* se reduce 10 veces el número de líneas, y en el caso del *ARPaint* son 4. Además, las aplicaciones de Minerva tienen más funcionalidad. En el caso de ARPaint añade unas teclas para poder parar el dibujado y eliminar todo el lienzo.

	ARSimple	**ARPaint**
Aplicación convencional	95	288
Minerva	9	72
Ratio	10.55	4

Cuadro 6.3: Comparativa entre dos aplicaciones realizadas con y sin Minerva.

Cabe destacar el hecho de que Minerva necesita un menor número de líneas, añade otras ventajas: no es necesario usar punteros, no es necesario compilar y el código es mucho más fácil de entender que una aplicación desarrollada directamente con *ARToolKit*.

6.2.4. *Profiling*

Profiling es como se conoce en ingeniería del software a la medida y análisis de rendimiento de una aplicación de forma dinámica, es decir, utilizando una ejecución concreta de dicha aplicación. El objetivo es determinar qué partes del código son *cuellos de botella* (consumen más recursos), y por tanto, son más susceptibles de ser optimizadas.

Para realizar *profiling* de Minerva, se ha utilizado *gprof*, herramienta libre para GNU/Linux que funciona con los compiladores de la familia *gcc*.

Para realizar el análisis es necesario compilar el programa con un flag *-pg*. A continuación se ejecuta con la configuración deseada para determinar su rendimiento. Fruto de la ejecución, se genera un fichero llamado *gmon.out*. Este fichero se ejecuta con *gprof* de la siguiente manera:

```
gprof ./ejecutable gmon.out
```

El *profiler* genera un documento a modo de informe del porcentaje de uso de CPU de *cada una de las funciones individuales* de la ejecución de la aplicación. En el Anexo L se puede ver un extracto de este código, en el que se han identificado los subsistemas a los que pertenecen las llamadas a función que más porcentaje de CPU emplean.

En la Figura 6.3 se muestran los porcentajes de uso de los subsistemas que más recursos consumen de Minerva ejecutando la aplicación *ARSheep*. La etiqueta *World&Logic* se refiere al dibujado, gestión de MAO's y evaluación de MLB's; la etiqueta *Physics* se refiere a los cálculos de simulación física, y la etiqueta *Tracking* a la ejecución del método de tracking basado en marcas.

Se han realizado ejecuciones en función de tres parámetros:

1. **Número de marcas:** se ha utilizado una ejecución con 2 marcas, y otra con 13.
2. **Número de modelos:** se han realizado ejecuciones con 5, 20 y 50 modelos simultáneos.
3. **Complejidad del modelo:** se han utilizado tres modelos: el *modelo simple* con 2.462 caras, el *modelo mediano* con 9.848 caras y el *modelo complejo* con 39.392 caras.

Además se ha hecho uso de la simulación física, tanto de objetos dinámicos (MAO's instanciados) como estáticos para provocar colisiones entre ellos.

6.2.5. Encuesta

Para poder probar la facilidad de uso de Minerva, se ha realizado una encuesta a 8 compañeros del grupo de investigación Oreto, proponiéndoles resolver un sencillo ejercicio y respondiendo después a 5 preguntas. En la Figura 6.4 se muestra la encuesta.

Los resultados se consideran un éxito rotundo. El sistema de Minerva basado en *Sensores, Controladores* y *Actuadores* fue entendido por los encuestados de forma muy rápido, y la totalidad consiguieron resolver el problema en muy poco tiempo. Todos ellos consideraron el manual de muy buena ayuda. En la Figura 6.2 se muestran los resultados de la encuesta.

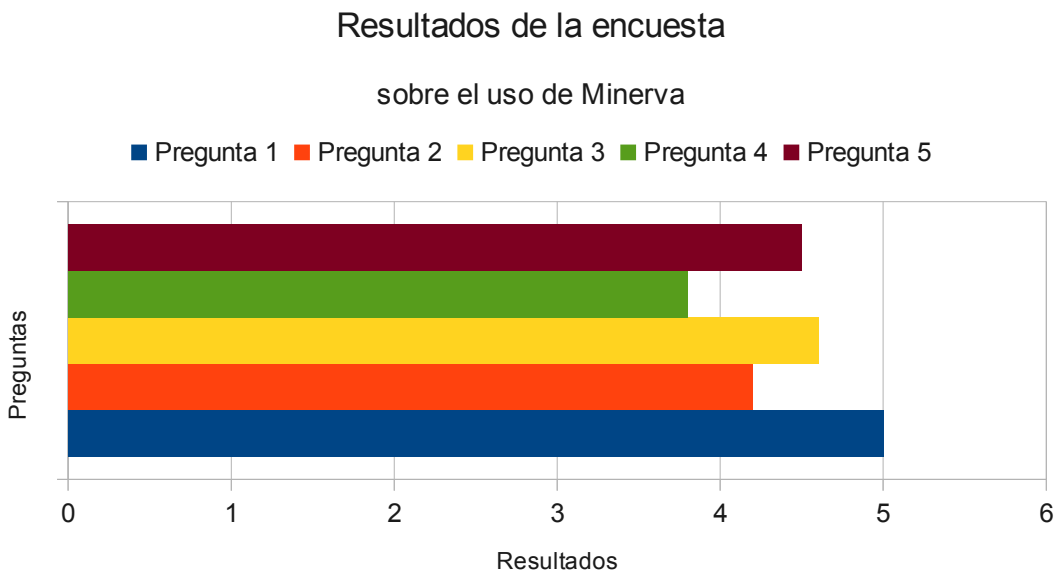

Figura 6.2: Resultados de la encuesta sobre el uso de Minerva.

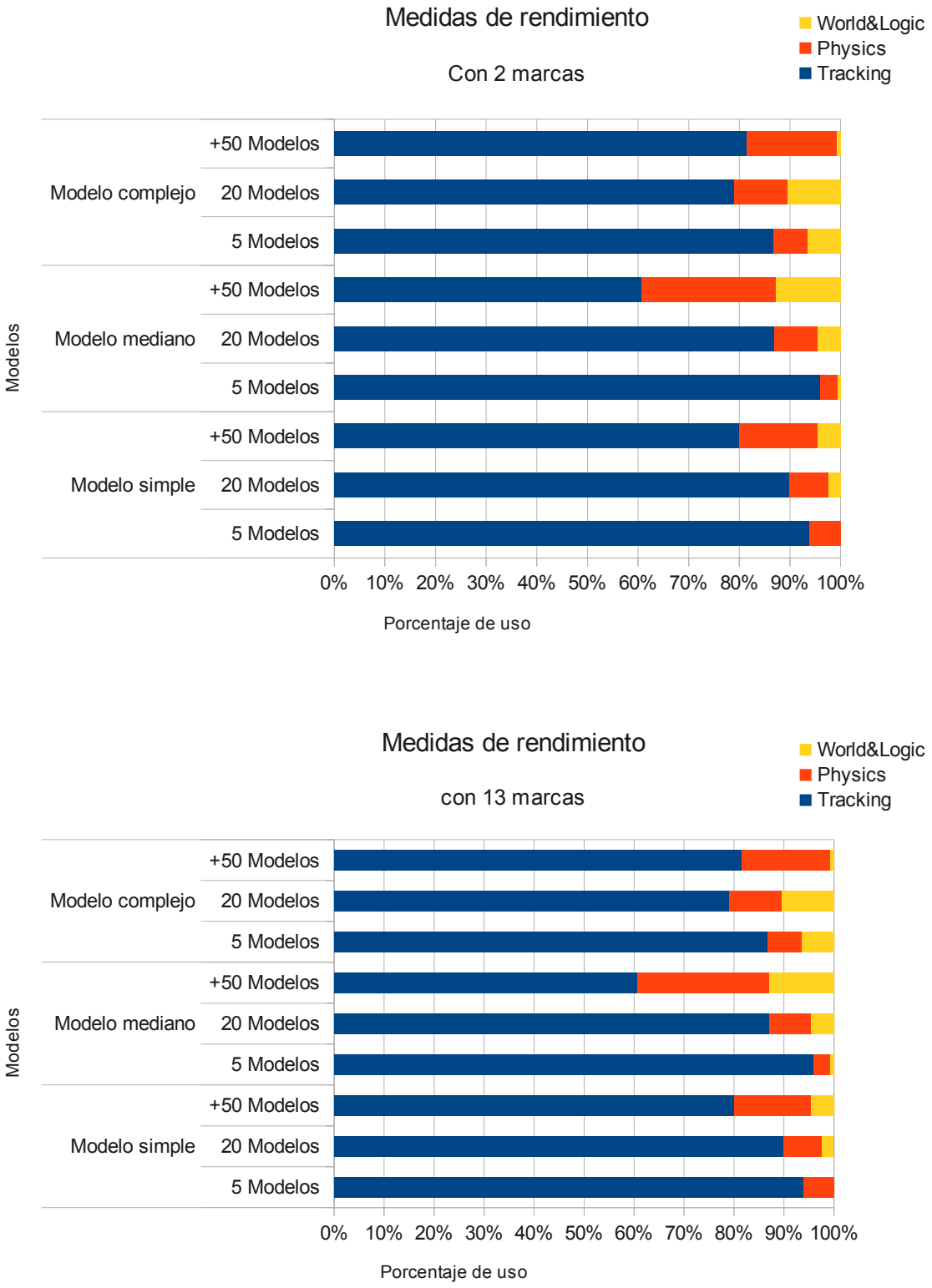

Figura 6.3: Resultados del *profiling* de Minerva.

Problema:

Desarrolle una aplicación de Realidad Aumentada en la que dada una marca, se dibuje una taza sobre ella. Además, cada 50 frames debe reproducirse un sonido cualquiera en formato *.wav*.

Encuesta:

1/ Puntúe del 1 al 5 la facilidad del problema propuesto.
(5) Lo he solucionado a la primera.
(4) Tras pensarlo un poco lo he solucionado.
(3) Le he dedicado más tiempo del esperado.
(2) Es un problema muy complejo.
(1) No he conseguido resolverlo.

2/ Puntúe del 1 al 5 el manual de usuario.
(5) Muy sencillo e intuitivo.
(4) Bien explicado pero demasiado técnico.
(3) Completo pero demasiado confuso.
(2) Faltan algunos aspectos por profundizar.
(1) Tiene muchos errores y está incompleto.

3/ Puntúe del 1 al 5 la facilidad de uso del lenguaje MSL.
(5) Muy fácil.
(4) Fácil.
(3) Un lenguaje cualquiera.
(2) Es necesario un tiempo para adaptarse.
(1) Demasiado confuso.

4/ Puntúe del 1 al 5 la facilidad que ha tenido para comprender el funcionamiento de la lógica basada en los MLB.
(5) Es la forma natural de pensar.
(4) Se entiende con el primer ejemplo.
(3) Con un par de ejemplos se entiende.
(2) Son necesarios muchos ejemplos para adaptarse.
(1) No he terminado de entenderlo.

5/ Puntúe del 1 al 5 su impresión general sobre el sistema Minerva.
(5) Muy buena.
(4) Buena.
(3) Normal.
(2) Mala.
(1) Muy mala.

Figura 6.4: Encuesta realizada para evaluar la facilidad de uso subjetiva de Minerva.

CAPÍTULO 7

CONCLUSIONES Y PROPUESTAS

TODOS los objetivos y subobjetivos explicados en el capítulo 5 han sido abordados para su implementación, durante el tiempo de desarrollo y las horas de dedicación (ver Capítulo 6). En este Capítulo se detallan los aspectos más relevantes asociados a los hitos cumplidos, las principales soluciones y decisiones aportadas, así como algunas de las líneas de trabajo asociadas al trabajo futuro.

7.1. Objetivos alcanzados

Los objetivos y subobjetivos descritos en el Capítulo 3 se han cumplido. Minerva es una arquitectura modular multiplataforma que proporciona un lenguaje de alto nivel denominado MSL (*Minerva Specification Language*) para el desarrollo, de una manera sencilla y rápida, de aplicaciones de *Realidad Aumentada*.

El objetivo principal que podía resumirse como el diseño de un lenguaje de alto nivel sencillo para la especificación de la lógica de las aplicaciones se ha desarrollado satisfactoriamente. Este lenguaje implementado en el submódulo de procesamiento de lenguaje MSL (ver Sección 5.6.1) permite definir de forma intuitiva los distintos componentes que forman parte de la aplicación, dando soporte a la resolución de errores de sintaxis, indicando la línea donde se detectan. Este lenguaje se ha diseñado teniendo en cuenta la facilidad de uso de cara a su utilización por parte de usuarios con escasos conocimientos de las materias que componen la Realidad Aumentada. Se han omitido la utilización de conceptos complejos o tecnicismos persiguiendo encontrar siempre la manera más sencilla de implementarlos.

La base en la que se sustenta este lenguaje es la definición de los componentes MAO y MLB descritos en la sección 5.2. Se dispone de un variado y rico repertorio de dichos componentes, que pueden dar lugar al desarrollo de aplicaciones de Realidad Aumentada de muy variada naturaleza. Este repertorio se ha diseñado de forma modular, para que la adición de nuevos componentes sea sencilla.

Por una parte, los MAO (*Minerva Augmenter Object*, ver Sección 5.2.1) permite la declaración de objetos tridimensionales, bidimensionales o elementos de los métodos de *tracking* entre otros.

Los MLB (*Minerva Logic Brick*, ver Sección 5.2.2), clasificados en *Sensores, Controladores* y *Actuadores*, permiten la definición de la lógica proporcionando diferentes formas de interacción entre los MAO.

El diseño de la lógica de control (ver Sección 5.2.3) es lo suficientemente extensible como para poder admitir nuevos tipos de MLB sin tener que cambiar su funcionamiento. Este submódulo permite además la definición de MAO's instanciados, lo que significa poder crear tantas copias de un único objeto tridimensional que facilita la incorporación de ciertas características a las aplicaciones.

El subsistema de representación 3D (ver Sección 5.3.1) permite el despliegue sencillo de modelos primitivos (como líneas simples o rutas más complejas) y de objetos tridimensionales representados en algún metaformato de representación con animación de cuerpos rígidos de forma transparente al usuario. Este subsistema es altamente eficiente para tratar de consumir el mínimo de recursos hardware y poder ampliar el rango de plataformas en las que Minerva pueda funcionar.

Completando el módulo de representación se encuentra el submódulo de representación 2D (ver Sección 5.3.4), que permite dibujar información bidimensional como imágenes (de un amplio repertorio de formatos) o texto dinámico, de diferente color, tamaño y fuente.

El módulo de entrada salida permite la declaración de múltiples dispositivos de vídeo para la realización de aplicaciones donde se utilice una cámara principal (también conocida como cámara de usuario o *Main Camera*), y otro número indefinido de cámaras externas, que sirvan como apoyo para realizar el *registro* de la posición y orientación del usuario. Estos dispositivos se basan en la utilización de una biblioteca multiplataforma con un uso muy extendido para dar soporte al mayor número de plataformas y fabricantes de dichos dispositivos, como se describe en la Sección 5.4.1.

La gestión de eventos de periféricos de entrada (ver Sección 5.4.2) como teclados y ratón se presenta al usuario de forma sencilla y extensible, de forma que se pueda configurar distintos tipos de eventos (como pulsación o liberación de teclas) y poder añadir dispositivos nuevos (ratones, *joysticks*) sin requerir mucho esfuerzo de codificación por parte del desarrollador.

El subsistema de audio (ver Sección 5.4.3) permite la reproducción sencilla de ficheros de formato de audio sin tener conocimientos previos de sonido digital ni conocer conceptos como *bitrate* o *número de canales*. Además el soporte de nuevos formatos de audio puede ampliarse de forma transparente al usuario.

Una de las partes clave que caracterizan las aplicaciones de Realidad Aumentada es la solución del *registro* de la realidad. Minerva implementa el submódulo de registro (ver Sección 5.5) que permite la implementación de varios métodos de tracking simultáneos de diversa naturaleza (visual o no visual, absolutos o relativos). Además se ha incorporado un método basado en marcas apoyado en la biblioteca *ARToolKit*, con soporte para el reconocimiento multimarca.

Una de las características más atractivas del sistema Minerva era la incorporación de simulación física (ver Sección 5.3.2). Esta característica se ha conseguido implantar superando las expectativas iniciales. Es posible añadir objetos a la simulación física con diferentes roles (objetos físicos estáticos o dinámicos), la detección de colisiones (incluyendo o no el resto de simulación física), y la definición de distintos tipos de formas de colisión (desde mallas de triángulos a formas primitivas). Todo esto basado en el motor eficiente y multiplataforma *Bullet*.

El submódulo de procesamiento de lenguaje de script (ver Sección 5.6.2), proporciona un acceso avanzado para usuarios experimentados a toda la funcionalidad de Minerva, consiguiendo implementar características avanzadas de una forma más eficiente. Se ha escogido el lenguaje de programación Python por ser uno de los lenguajes de script más extendidos en la actualidad.

El sistema se apoya en el módulo de depuración (ver Sección 5.7) que permite ayudar tanto al desarrollador del presente proyecto, como al usuario a la hora de codificar sus propias aplicaciones de Realidad Aumentada, informando de todo lo que ocurre en el sistema con distintos tipos de prioridad (mensaje normal, advertencia y error). El objetivo es proporcionar el mayor nivel de información posible que ayude en el desarrollo de aplicaciones basadas en la plataforma.

El código de Minerva se basa en buenos principios de diseño basados en el uso de patrones, implementando varios de ellos. Se ha conseguido crear un código portable, que funciona en plataformas *Microsoft Windows* y *GNU/Linux*, facilitando su instalación (mediante un instalador en el primer caso, y mediante paquetes en el segundo). El código MSL de cualquier aplicación funciona en ambas plataformas sin realizar ninguna modificación.

Por último, se ha hecho hincapié en realizar un manual de usuario (ver Anexo A) sencillo de entender, explicando las características del sistema paso a paso, empezando desde un sencillo programa *Hola Mundo* hasta complejas aplicaciones. Se adjuntan también el código de diferentes aplicaciones *Demo* para que el usuario pueda aprender *programando*.

7.2. Propuestas de trabajo futuro

Minerva pretende ser una plataforma de desarrollo con un uso extendido entre la comunidad de usuarios, pudiendo dar soporte y admitir propuestas y modificaciones para el rápido desarrollo de aplicaciones de Realidad Aumentada.

A continuación se describen las propuestas de trabajo futuro planeadas hasta la fecha:

- **Mejora y ampliación del lenguaje MSL:** el lenguaje MSL es la pieza principal de Minerva, ya que es el elemento base con el que el usuario interactúa para realizar las aplicaciones de Realidad Aumentada. Como trabajo futuro se propone añadir más flexibilidad a este lenguaje, como por ejemplo añadir nuevos mecanismos a la sintaxis para declarar los componentes de forma más sencilla e intuitiva.

  Otro aspecto relacionado es la información de error que proporciona el lenguaje ante un fallo de sintaxis. Poder indicar, a parte de la línea y el *token* que da el error, sugerencias que guíen al usuario para poder solucionar el fallo. Esto conllevaría el estudio de los analizadores semánticos y las producciones de error, para poder detectar errores relacionados con los tipos de datos. Hasta ahora, el intérprete del lenguaje detecta si una variable ha sido declarada o no al usarla, pero no comprueba que el tipo de la variable sea el indicado para su uso.

  Se debería estudiar más a fondo las bibliotecas utilizadas para la implementación del lenguaje, incluso tener en cuenta otras que proporcionen mayor modularidad a este componente de Minerva.

- **Ampliación del repertorio de componentes MAO y MLB:** hasta ahora se dispone un repertorio de componentes adecuado para realizar diversos tipos de aplicaciones, pero basta con imaginar otras tantas para poder darse cuenta de nuevos componentes MAO y MLB que podrían ser útiles.

  En cuanto a los componentes MAO, se pueden añadir más objetos tridimensionales primitivos, distintos tipos de animaciones o incluso aumentar el número de propiedades de los que dispone actualmente. Se puede incluir objetos bidimensionales con animaciones (como ficheros con formato GIF). Los objetos tridimensionales actualmente soportan animaciones de cuerpo rígido, pero pueden utilizarse animaciones avanzadas que puedan ser exportadas al metaformato.

  Los MLB definen la lógica, y son el *núcleo* de la lógica de la aplicación. Se pueden implementar sensores de tipo *mensaje*, que detecten eventos de más periféricos de entrada (*ratones* o *joysticks*). En cuanto a los actuadores las posibilidades son infinitas. Implementar un sistema de síntesis de voz, o de comunicación remota con otra aplicación desarrollada con Minerva para permitir aplicaciones *multijugador*.

- **Construcción de un sitio Web:** un hito importante para la expansión de este sistema es la construcción de una página web donde se pueda descargar el código fuente y los binarios para diversas plataformas de Minerva. Además se añadiría una sección de tutoriales para que nuevos usuarios tengan más facilidades en el aprendizaje de su uso. Otra Sección interesante sería un *bug tracker*, donde se puedan hacer sugerencias y notificar *bugs* sobre el funcionamiento de Minerva, para conseguir un desarrollo *comunitario* del proyecto.

- **Completar el soporte de formatos:** ampliar el soporte para distintos formatos de archivos es una tarea prioritaria para no limitar el funcionamiento de Minerva. Se pretende dar soporte a formatos de audio con más compresión como *mp3* u *ogg*, pues actualmente solo se soporta *wav*. Otro tipo de ficheros que se pretende implementar son metaformatos de geometría tridimensional, para no limitar únicamente al formato *OreJ*. Formatos como *Obj* o *Collada* son buenas alternativas para comenzar, ya que son de uso muy extendido y, en el caso del segundo, se trata de un formato muy completo que tiende a convertirse en un estándar ámpliamente soportado por multitud de aplicaciones de diseño 3D.

- **Añadir distintos métodos de tracking:** Dado que Minerva ha sido diseñada con un módulo de registro para soportar múltiples métodos de tracking simultáneos, se pretende estudiar otras alternativas (visuales y no visuales) para implementar. Algunas alternativas vibles son *Ptam* y *BazAR*, ambos métodos de tracking visuales no basados en marcas, lo que los hace más naturales al usuario. También es posible utilizar métodos no visuales como acelerómetros y giróscopos, que podrían incorporarse en forma de recurso hardware.

7.3. Conclusión personal

El desarrollo de un sistema como Minerva, y cualquier Proyecto de Fin de Carrera en general, supone una experiencia más cercana al trabajo real que va a desempeñar cualquier aspirante a ser Ingeniero en Informática durante el resto de su vida. Se debe estudiar y dominar un amplio rango de áreas, bibliotecas y disciplinas, pensando siempre en el usuario final, para que la aplicación sea robusta y fácil de usar. Todo ello supone un nuevo enfoque en comparación con lo que el estudiante ha realizado durante sus años de formación en la carrera, que le aporta madurez para poder enfrentarse a su futuro inmediato.

Cuando un estudiante realiza su Proyecto de Fin de Carrera, pone de manifiesto que los Ingenieros Informáticos no son *ingenieros de segunda* que formatean ordenadores, o arreglan sistemas operativos. Un Ingeniero Informático debe tener conocimiento de muchas áreas técnicas de muy diversa naturaleza, como pueden ser el álgebra lineal, geometría euclídea, procesadores de lenguajes, arquitectura de computadores, ingeniería del software o bases de datos. En mi opinión, *debemos defender* nuestra identidad con dignidad, sin caer en tópicos relacionados con nuestra profesión, pidiendo siempre un trato y un respeto acorde con nuestra formación, a la altura de otras ingenierías.

La Ingeniería Informática es una de las profesiones con más *potencial* y *versatilidad* de todas las que existen actualmente en el mercado, por lo que es para mí un orgullo haber realizado esta elección hace ya cinco años.

ANEXOS

APÉNDICE A

MANUAL DE USUARIO

En este anexo se explica el funcionamiento de Minerva de un modo didáctico, comenzando por las características más básicas hasta llegar a dominar las más avanzadas, mediante multitud de ejemplos.

En la Figura A.1 se muestran diferentes capturas de pantallas creadas con Minerva, y adjuntas en el CD con la presente documentación.

A.1. Primeros pasos

En Minerva las aplicaciones se realizan mediante su lenguaje de propósito específico llamado *MSL* (*Minerva Specification Language*). Este lenguaje es mucho más sencillo que cualquier lenguaje de programación imperativo, ya que se trata de un lenguaje *declarativo*. Carece de estructuras de control, funciones, etc. Se trata de un lenguaje cuyo propósito específico es *declarar* unos tipos de objetos especiales, y sus relaciones entre ellos. Hay dos tipos de objetos especiales:

- **MAO (Minerva Augmenter Object)**: son aquellos objetos que van a aparecer en la aplicación. Pueden ser marcas o un grupo de ellas, objetos tridimensionales en formato OreJ con o sin animación o imágenes y texto 2D.

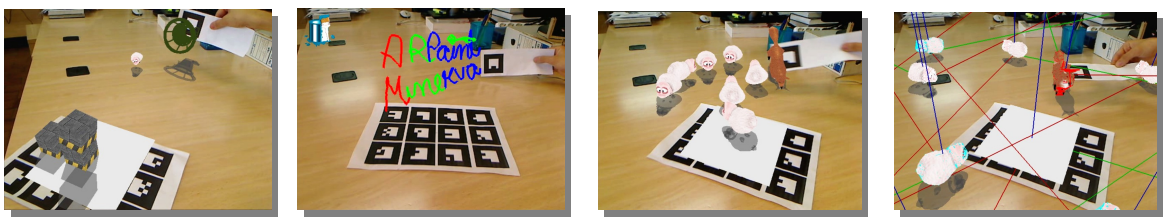

Figura A.1: *Screenshots* de varias aplicaciones creadas con Minerva.

Todos ellos comienzan con el prefijo *MAO* seguido de su nombre. Cada uno tiene una serie de parámetros obligatorios y opcionales para declararlos. Por ejemplo, una marca necesita saber la ruta a su archivo de marca y su tamaño en metros.

Además de los parámetros necesarios para su creación, cada *MAO* tiene por naturaleza unas *propiedades intrínsecas*. Los valores de estas propiedades definen su comportamiento y pueden ser modificadas en tiempo de *ejecución*. Por ejemplo, todo *MAO* tridimensional (objeto 3D) tiene un tamaño, que se puede cambiar cada vez que se pulse una tecla.

Básicamente, crear una aplicación de Realidad Aumentada en Minerva consiste en declarar los *MAO's* de esa aplicación. Para consultar en detalle la lista de *MAO's* disponible, su descripción, propiedades intrínsecas, sintaxis en MSL y más, consultar el Anexo C.

- **MLB (Minerva Logic Brick):** son los *bloques* de lógica de la aplicación. Cada Minerva Augmenter Object (MAO) tiene asociados unos Minerva Logic Brick (MLB) que definen cómo se comportará ante cualquier tipo de evento. Los hay de tres tipos: *sensores*, *controladores* y *actuadores*. Los sensores son los *ojos* del mundo, están pendientes de lo que todo lo que ocurre en él y avisan a los *controladores*, que se encargan de decidir si se da una situación apropiada o no para activar los *actuadores*, que son los que producen cambios en el mundo.

  Al igual que sucedía con los MAO's, los MLB's también tienen propiedades que los definen. Por ejemplo, un actuador que proporciona un impulso a un MAO tridimensional, dispone de una propiedad *fuerza*, que indica la magnitud del impulso. Este se puede utilizar para variar la velocidad con la que sale disparado una bala dependiendo del tiempo que se tenga pulsada la tecla de disparo. Estas propiedades se modifican a través de la Application Programming Interface (API) de Python.

  En los sucesivos ejemplos se comprenderá de forma práctica el uso de estos bloques lógicos, que son más sencillos e intuitivos de lo que pueda parecer en un principio. Una lista de todos los MLB's de Minerva, junto a su descripción, sintaxis MSL y propiedades intrínsecas, se encuentra en el Anexo D.

Para finalizar esta introducción, se muestra el esqueleto básico de una aplicación escrita en MSL en la Figura A.2.

El último concepto nuevo introducido ha sido el de *MAOWorld*, que es simplemente una forma de ponerle nombre a la aplicación.

```
/* Esto es un comentario de varias lineas.
   Lo que se escriba aqui no interfiere
   en la aplicacion.
*/
MAOWorld <nombre de la aplicacion>{
    [Declaracion MAO1
        [Declaracion MLB's de MAO1]
    ]

    [Declaracion MAO2
        [Declaracion MLB's de MAO2]
    ]
    [Declaracion MAO3
        [Declaracion MLB's de MAO3]
    ]
    ...
}
```

Figura A.2: Estructura básica de una aplicación de Minerva.

A.2. La primera aplicación

Esta es una aplicación muy sencilla que servirá para entender de forma práctica los conceptos y el funcionamiento de Minerva.

El objetivo de la aplicación es poder mostrar una tetera 3D en una marca y, además, que cuando se pulse la tecla de *ESCAPE* se cierre la aplicación. De esto se deduce que son necesarios dos MAO's:

1. Un MAO haciendo el papel de la marca que necesitamos. Para ello se utiliza *MAOMark*. Según su nombre se desprende que es un MAO, y una marca (*Mark*, en inglés).

2. Otro MAO que represente la tetera tridimensional. Minerva proporciona un *MAORenderable3DTeapot* que hace justamente eso. De su nombre se pueden deducir tres cosas: la primera, que es un MAO pues empieza por ese prefijo. La segunda, que es un objeto tridimensional, ya que contiene *Renderable3D* (todos los objetos tridimensionales tienen en su nombre ese identificador), y por último, que es una tetera (*Teapot*, en inglés).

Una vez identificados los MAO del sistema, se puede comenzar su desarrollo.

En el directorio donde se quiera tener, se puede crear otro denominado *resources* (o cualquier otro nombre) para organizar todos los ficheros que se van a necesitar (archivo de marcas, modelos OreJ, imágenes, sonidos). En este caso vamos a utilizar la marca 99, por lo que se guarda el archivo *4x4_99.patt*.

Ahora se crea el fichero de texto que contendrá el código de la aplicación (fuera del directorio *resources*). Hay que recordar que estos ficheros tienen extensión *.mrv*, por lo que se

puede llamar *App.mrv*. Según la estructura de una aplicación Minerva, la primera sentencia a escribir es:

```
MAOWorld App{
```

La aplicación se llama *App*, aunque no tiene por qué ser el mismo que el del fichero de código. A partir de ahora hay que comenzar a declarar MAO's. Primero se va a declarar el MAO marca, ya que el MAO tetera la va a necesitar para saber en qué marca tiene que dibujarse.

Viendo la documentación en el Anexo C, se observa que para declarar el MAO es necesario un *path* (ruta al archivo de la marca) y un *size* (tamaño de la marca en metros). Se pueden definir sus MLB's como a cualquier MAO, y además indicarle si actúa como *Ground*. Esta parte tiene que ver con la simulación física, por lo que en este ejemplo no tiene interés.

La declaración del MAOMark quedaría de la siguiente forma:

```
MAOMark marca99{
    path="resources/4x4_99.patt"
    size=0.12
}
```

Antes de continuar hay que tener en cuenta unos aspectos:

1. El tamaño es el lado de la marca. En este ejemplo se supone 12 centímetros.
2. El nombre *marca99* puede ser cualquier otro.
3. Un aspecto del lenguaje MSL es que el orden de los parámetros **es importante**. Deben escribirse tal y como especifica la sintaxis del componente.

Es el turno del *MAORenderable3DTeapot*. La declaración resultante es:

```
MAORenderable3DTeapot taza{
    size = 0.12
    reference = marca99
}
```

El parámetro *reference* hace alusión a la marca en la que se dibujará. Según su sintaxis puede tomar el valor *Null*, que se utiliza para los MAO instanciados. Esta característica se detalla en la Sección A.6.

Por último queda especificar que cuando se pulse la tecla *Escape* la aplicación se cierre. Esto se hace mediante los MLB's mencionados anteriormente. El análisis de los componentes MLB necesarios es el siguiente:

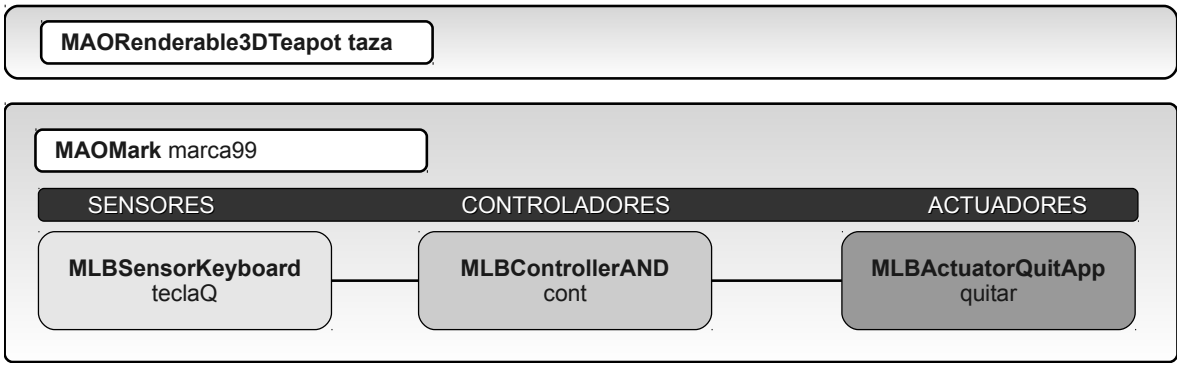

Figura A.3: Esquema de componentes de ARSimple.

1. Es necesario un sensor que detecte la pulsación de la tecla *Escape*. El MLB que implementa esta función es *MLBSensorKeyboard*.

2. Un controlador que se active únicamente cuando lo haga el sensor. Para ello se puede utilizar un *MLBControllerAND* o un *MLBControllerOR*. Para comprender mejor el funcionamiento de cada controlador leer la descripción en el Anexo C.

3. Se necesita un actuador que cierre la aplicación. Este actuador es *MLBActuatorQuitApp*.

En la Figura A.3 se puede observar un esquema general de todos los componentes necesarios para la aplicación.

A continuación se muestra la especificación de estos tres MLB's en MSL:

```
MLBSensorKeyboard teclaQ{type = "KEYDOWN" key = "ESCAPE"}

MLBControllerAND cont

MLBActuatorQuitApp quitar{}
```

El sensor de teclado se ha declarado para que se active cuando se presiona la tecla (*KEYDOWN*). Para ver qué más tipos de eventos y teclas admite este y otros sensores, ver el Anexo E.

Ahora queda conectar cada sensor con su controlador, y cada controlador con su actuador. Esto se realiza mediante el operador flecha (->). Especificar los enlaces se puede hacer de varias formas:

Hacerlo en una sola línea de la forma *sensores->controlador->actuadores*:

```
teclaQ -> cont -> quitar
```

O en dos líneas, de las formas *sensores ->controlador* y *controlador ->actuadores*:

```
MAOWorld App{
  MAOMark marca99 {
    path="resources/4x4_99.patt"
    size=0.12

    MLBSensorKeyboard teclaQ {type = "KEYDOWN" key = "ESCAPE"}

    MLBControllerAND cont

    MLBActuatorQuitApp quitar {}

    teclaQ -> cont -> quitar
  }

  MAORenderable3DTeapot taza{
    size = 0.12
    reference = marca99
  }

}
```

Figura A.4: Código de la primera aplicación en Minerva.

```
teclaQ -> cont
cont -> quitar
```

Las dos maneras se comportan exactamente igual. Otro aspecto importante es que un controlador pueden conectarse a varios sensores y varios actuadores, pero nunca un sensor con un actuador. De esta forma la lógica puede hacerse todo lo compleja que se quiera. Por ejemplo, si se quisiera que la aplicación se cerrase cuando se pulsasen simultáneamente las teclas *ESCAPE* y *A*, se crearía un sensor por cada tecla y se conectarían los dos a un controlador *AND* (este controlador sólo activa sus actuadores cuando **todos** sus sensores están activados).

Como se explicó al principio, los MLB's deben pertenecer a un MAO. En este caso es indiferente a cuál, por lo que se asocian, por ejemplo, a la marca.

El código completo de la aplicación se puede ver en la Figura A.4.

Para terminar, Minerva dispone de un *log* que informa de todos los fallos o advertencias que se producen en el sistema. Estos mensajes son mostrados por consola y escritos en un fichero de texto llamado *log*. Es de mucha utilidad recurrir a él cuando algo no funciona como se esperaba.

A.3. ARPaint_Lite

En esta sección se va a diseñar una aplicación simple de dibujado en tres dimensiones gracias a la Realidad Aumentada. Esta es una versión *Lite* porque en el Compact Disc (CD) adjunto con la documentación se facilita además una versión más completa y con más características. Primero se va a describir qué va a hacer esta aplicación.

ARPaint_Lite es una sencilla aplicación de dibujado que va a consistir en dos marcas: la primera servirá como lienzo, y la segunda como pincel. Dispondrá de una tecla para comenzar el dibujado (tecla D), otra para pararlo (tecla S) y una tercera para borrar el lienzo entero (tecla R). Dispondrá de tres colores (teclas 1, 2 y 3) y además se añadirá una para cerrar la aplicación (tecla ESC). En la Figura A.5 se muestra el esquema de componentes de esta aplicación.

A continuación se explica el funcionamiento general de la aplicación.

La aplicación está compuesta por tres elementos MAO básicos:

- **Lienzo:** es el MAOMark que actúa como referencia para la ruta y para la brocha. La única lógica que contiene es la de la pulsación de la tecla ESC para finalizar la aplicación.
- **Brocha:** la brocha capturará los puntos y los almacenará en la propiedad *relative* de el MAO ruta. Este último MAO añade un punto donde lo diga dicha variable (que es del tipo *pose*). Es necesario que la posición de los puntos sea relativa al lienzo, por lo que la lógica de la brocha es calcularla (mediante el actuador *MLBActuatorRelativePose*, y almacenarla en la ruta.
- **Ruta:** por último, la ruta está siempre añadiendo puntos, utilizando el valor almacenado en su variable *relative*. Además, se ha añadido algo de lógica para cambiar la visibilidad de los puntos (cambiar el valor de la propiedad *visiblePoint* mediante dos MLBActuatorProperty), eliminar los puntos (mediante un MLBActuatorPathRemovePoints) y cambiar el color del punto. Este color se debe cambiar componente a componente las tres posibles (*r,b* y *b*), por lo que por cada componente y por cada color es necesario un MLBActuatorProperty.

El código completo de la aplicación se muestra a continuación:

```
1 /* A simple Augmented Reality paint application */
2 /* How to!
3 Key 1: Red Color
4 Key 2: Green Color
5 Key 3: Blue Color
6 Key D: Start Drawing
7 Key S: Stop Drawing
8 Key R: Clean up!
```

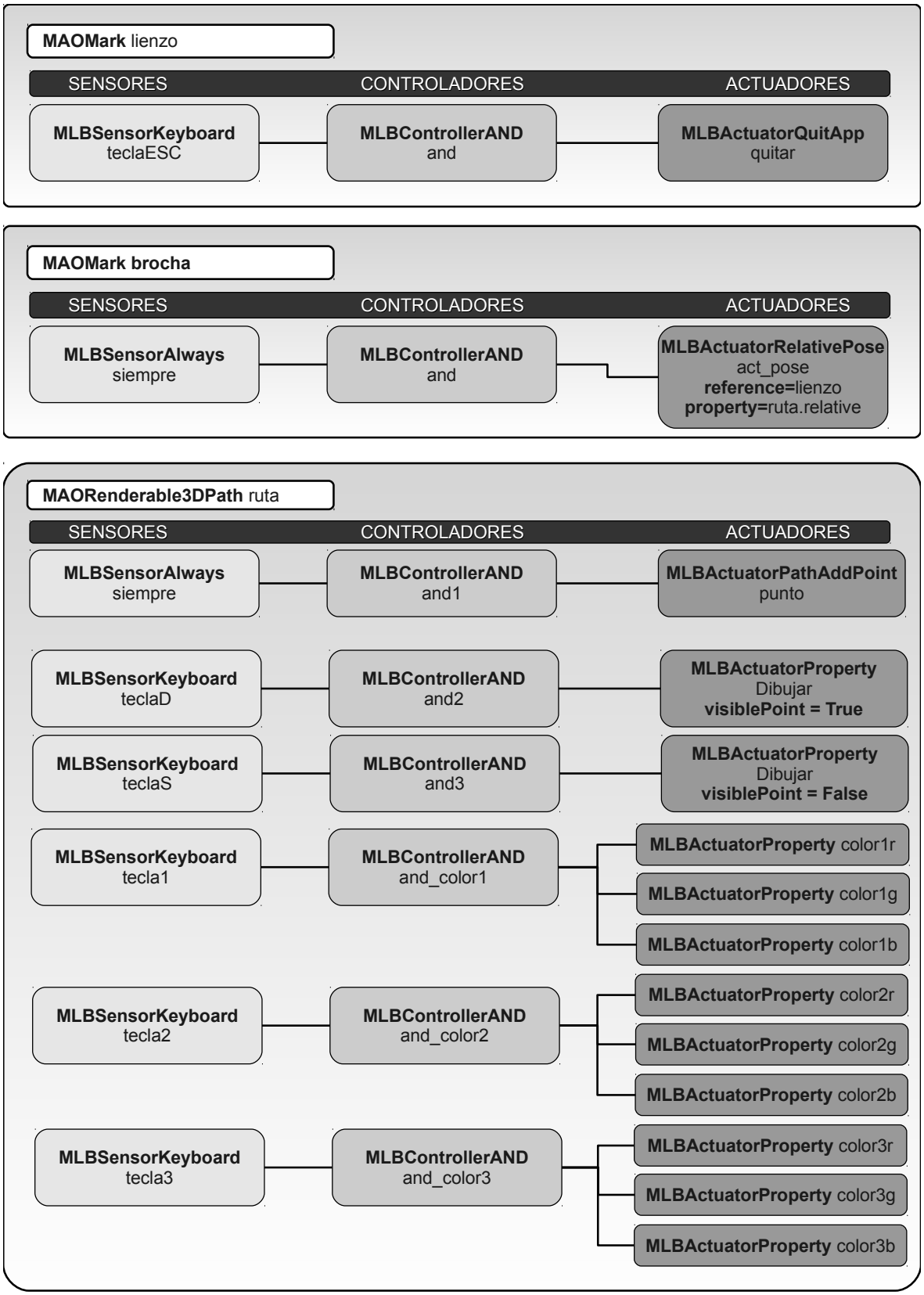

Figura A.5: Esquema de componentes de la aplicación ARPaint_Lite.

```
Key ESC: Exit the app.
Mark 1: Reference.
Mark 6: Brush.

~ Enjoy! ~
*/

MAOWorld ARPaint{
   /* Reference mark. */
   MAOMark lienzo{
      path = "resources/4x4_1.patt"
      size = .04

      MLBSensorKeyboard teclaESC {type = "KEYDOWN" key = "ESCAPE"}

      MLBControllerAND and

      MLBActuatorQuitApp quitar{}

      teclaESC -> and -> quitar
   }

   MAORenderable3DPath ruta{
      size = 5.
      color = (255,0,0)
      reference = lienzo

      MLBSensorAlways siempre{}

      MLBSensorKeyboard teclaS {type = "KEYDOWN" key = "S"}

      MLBSensorKeyboard teclaR {type = "KEYDOWN" key = "R"}

      /* Change the color */
      MLBSensorKeyboard tecla1 {type = "KEYDOWN" key = "1"}

      MLBSensorKeyboard tecla2 {type = "KEYDOWN" key = "2"}

      MLBSensorKeyboard tecla3 {type = "KEYDOWN" key = "3"}

      MLBSensorKeyboard teclaD {type = "KEYDOWN" key = "D"}

      MLBControllerAND and1, and2, and3
      MLBControllerAND and_color1, and_color2, and_color3
      MLBControllerAND and_tecla_r

      MLBActuatorPathAddPoint punto{}

      MLBActuatorPathRemovePoints borrar{}

      MLBActuatorProperty dibujar {type = "ASSIGN" property =
         visiblePoint value = True}

      /* Actuators to change the colors! */
      /* Color 1 */
      MLBActuatorProperty color1r {type = "ASSIGN" property = r  value =
         255}
      MLBActuatorProperty color1g {type = "ASSIGN" property = g  value =
```

```
            0}
65          MLBActuatorProperty color1b {type = "ASSIGN" property = b  value =
               0}

67          /* Se repiten los tres ultimos MLB's para el color2 y el color 3*/
68          /* ... */


71          MLBActuatorProperty nodibujar {type = "ASSIGN" property =
               visiblePoint value = False}

73          /* Always add a point to the path! */
74          siempre -> and1 -> punto
75          teclaD -> and2 -> dibujar
76          teclaS -> and3 -> nodibujar
77          teclaR -> and_tecla_r -> borrar

79          tecla1 -> and_color1 -> color1r, color1g, color1b
80          tecla2 -> and_color2 -> color2r, color2g, color2b
81          tecla3 -> and_color3 -> color3r, color3g, color3b
82      }

84      MAOMark brocha{
85          path = "resources/4x4_6.patt"
86          size = .04

88          MLBSensorAlways always {}

90          MLBControllerAND and

92          MLBActuatorRelativePose act_pose {reference = lienzo property =
               ruta.relative inverse = False}

94          always->and->act_pose
95      }
96  }
```

A.4. Propiedades de los MAO y tipos de datos

Como ya se ha explicado, los MAO tienen asociadas unas propiedades intrínsecas según su naturaleza. Además se le pueden añadir tantas propiedades como se desee. Por ejemplo, se puede diseñar para una aplicación de Realidad Aumentada un objeto tridimensional botón, y añadirle una propiedad que indique si está o no activado.

Estas propiedades pueden ser accedidas mediante la API de Python, o mediante los MLBS's *MLBSensorProperty* y *MLBActuatorProperty*. Con estos dos MLB's se pueden consultar y modificar explícitamente el valor de dichas propiedades, aunque existen otros MLB's que también pueden hacer uso de ellas.

Estas propiedades tienen asociado un tipo. Los tipos de propiedades soportados en Minerva son:

```
<tipo de la propiedad> <nombre de la propiedad> = <valor>
```

Figura A.6: Sintaxis para declarar una propiedad de un MAO en MSL.

```
int maximo = 9
```

Figura A.7: Ejemplo de declaración de una propiedad entera.

- **int**: representa un número entero.

  Ejemplo: 5

- **float**: representa un número decimal. La coma representa con un punto.

  Ejemplo: 0.25

- **boolean**: representa un valor binario (*True* o *False*)

  Ejemplo: True ó False

- **string**: representa una cadena de caracteres. La cadena va entre comillas (¨).

  Ejemplo: "Hola Mundo!"

- **pose:** matriz de elementos float de *4x4*, en forma de vector de 16 elementos. Representa una transformación espacial.

  Ejemplo: 1.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 1.0

En la Figura A.6 se muestra cómo declarar en MSL, una propiedad.

Estas declaraciones se hacen en la zona de *declaración de propiedades*. En la Figura A.7 se muestra un ejemplo.

Además de los tipos de datos de los que pueden ser las propiedades, existen otros utilizados para los parámetros en el momento de la declaración de los MAO's y los MLB's:

- **Identificador:** también denominado *identifier* o *name*. Se refiere al nombre de otro MAO, MLB o propiedad de MAO. Se escriben sin comillas.

  Ejemplo:

```
mao = marca99
```

- **vector2di:** vector de dos elementos (2D) de *int*.

  Ejemplo: (2, 1)

```
Ground{
    axis = Z
    gravity = -9.8
    shadows = true
    sun = (0.0, 0.0, 10.0)
}
```

Figura A.8: Ejemplo de declaración como *Ground* de un MAOMark o MAOMarksGroup

- **vector2df:** vector de dos elementos (2D) de *float*.

  Ejemplo: (1.5, 0.2)

- **vector3di:** vector de tres elementos (3D) de *int*.

  Ejemplo: (2, 1, 4)

- **vector3df:** vector de tres elementos (3D) de *float*.

  Ejemplo: (1.5, 0.2, 2.1)

A.5. Simulación física

Utilizar la *simulación física* en Minerva consiste en proporcionar un entorno a los MAORenderable3D donde existe un suelo, están sometidos al efecto de la gravedad y a colisiones entre ellos.

El primer paso para utilizar la simulación física es indicar que un *MAOMark* o un *MAOMarksGroup* actúe como suelo. Para ello, en su declaración hay que escribir la parte de la sintaxis que comienza por *Ground*. En la Figura A.8 aparece un ejemplo de esa parte, configurado para realizar la fuerza de la gravedad en el sentido contrario al eje Z con una fuerza de 9.8, habilitando el dibujado de sombras con un sol hipotético situado en el punto (0.0, 0.0, 10.0) según el eje de coordenadas del MAOMark o MAOMarksGroup.

En este momento la simulación física está activa. Ahora solo falta añadir los MAORenderable3D's que se quiera que estén sometidos a ella. Estos pueden estar sometidos a la simulación física de dos formas:

1. **Objeto físico dinámico:** de esta forma el objeto están sometidos completamente a las leyes físicas. Tanto si son instanciados como si no, en el momento de su creación se independizan de su referencia, es decir, cuando se detecta su MAOMark o su MAOMarksGroup caen por la acción de la gravedad y no vuelven a tener en cuenta dicha marca. Pueden sufrir colisiones con otros objetos físicos. La sintaxis para declararlo es el bloque que empieza por *DynamicObject*. En la Figura A.9 se muestra un ejemplo.

```
DynamicObject{
    mass = 1.0
    shape = TRIANGLE_MESH
}
```

Figura A.9: Ejemplo de sintaxis MSL para declarar un objeto físico dinámico

```
StaticObject{
    shape = BOX
}
```

Figura A.10: Ejemplo de sintaxis MSL para declarar un objeto físico estático

2. **Objeto físico estático:** estos objetos *no* están sometidos a la acción de la gravedad. Se dibujan siempre donde aparezca su MAOMark o MAOMarksGroup de *referencia*, pero puede provocar colisiones en el resto de objetos físicos dinámicos. El usuario puede de esta manera interactuar de forma directa en la simulación física de la aplicación. Se declara añadiendo la sintaxis que comienza por *StaticObject*. En la Figura A.10 se muestra un ejemplo de sintaxis para declarar un objeto estático.

Los parámetros más importantes a la hora de declarar objetos físicos son:

- *mass:* indica la masa del objeto en kilogramos.
- *shape:* indica la forma que tiene el objeto a la hora de calcular la colisión con otros objetos. Las opciones posibles son:
  - BOX: Supone que el objeto tiene el tamaño de una caja.
  - SPHERE: El objeto colisiona como si fuese una esfera.
  - CYLINDER: El objeto tiene forma cilíndrica.
  - TRIANGLE_MESH: toma la forma real del objeto, si se trata de una malla compuesta de polígonos triangulares.

En todos los casos Minerva calcula el tamaño óptimo de la forma de colisión, para que se ajuste lo mejor posible.

A.6. MAO's instanciados

Hasta ahora se ha visto que un MAO es algo que aparece explícitamente en la aplicación de Realidad Aumentada, como una marca o un objeto tridimensional.

En particular, los objetos tridimensionales (aquellos cuyo nombre empiezan por *MAO-Renderable3D*) necesitaban una referencia (parámetro *reference*) que puede ser cualquier otro *MAOPositionator3D* para poder ser representados, y aparecía solamente uno por declaración.

Existe otra forma de crear *MAORenderable3D's*. Ésta es mediante el uso de los MAO's instanciados. Esta técnica consiste en declarar un MAORenderable3D, con la única diferencia de que en el parámetro *reference* se le pasa *Null*. Esto quiere decir que el MAO está en el sistema, pero no se representará de ninguna forma. A este MAO se le llama *MAO Clase*.

A partir de este MAO Clase pueden crearse *copias* suyas (MAO's instanciados) mediante el actuador *MLBActuatorAddDynamicObject*. Este actuador necesita saber el nombre (parámetro *name*) del MAOPositionator3D que utilizará como referencia en el momento de su creación. Esto quiere decir que si se indica el nombre de un MAOMark, cuando se active el actuador se creará un MAO instanciado en el lugar donde se encuentre dicha marca.

Otros parámetros que se pueden configurar de este actuador son:

- *time:* indica la duración de la vida del MAO instanciado en *frames*.
- *offset:* se utiliza para dar un desplazamiento en localización y rotación a partir del MAO referencia. Se indica con un tipo de dato *pose*.
- *impulse:* proporciona un impulso inicial al MAO. Se indica como un tipo de datos *vector3df*, indicando la dirección, sentido y magnitud del impulso.

El actuador MLBActuatorAddDynamicObject debe pertenecer al MAO clase que se quiera instanciar. Otro detalle importante es que ese MAO Clase debe ser además un *objeto físico dinámico*. Para entender mejor los conceptos relacionados con la simulación física ver la Sección A.5.

En el ejemplo de ARSheep, en la Sección A.7, se utilizan MAO's instanciados para la creación de las ovejas y se ilustra el funcionamiento de la simulación física.

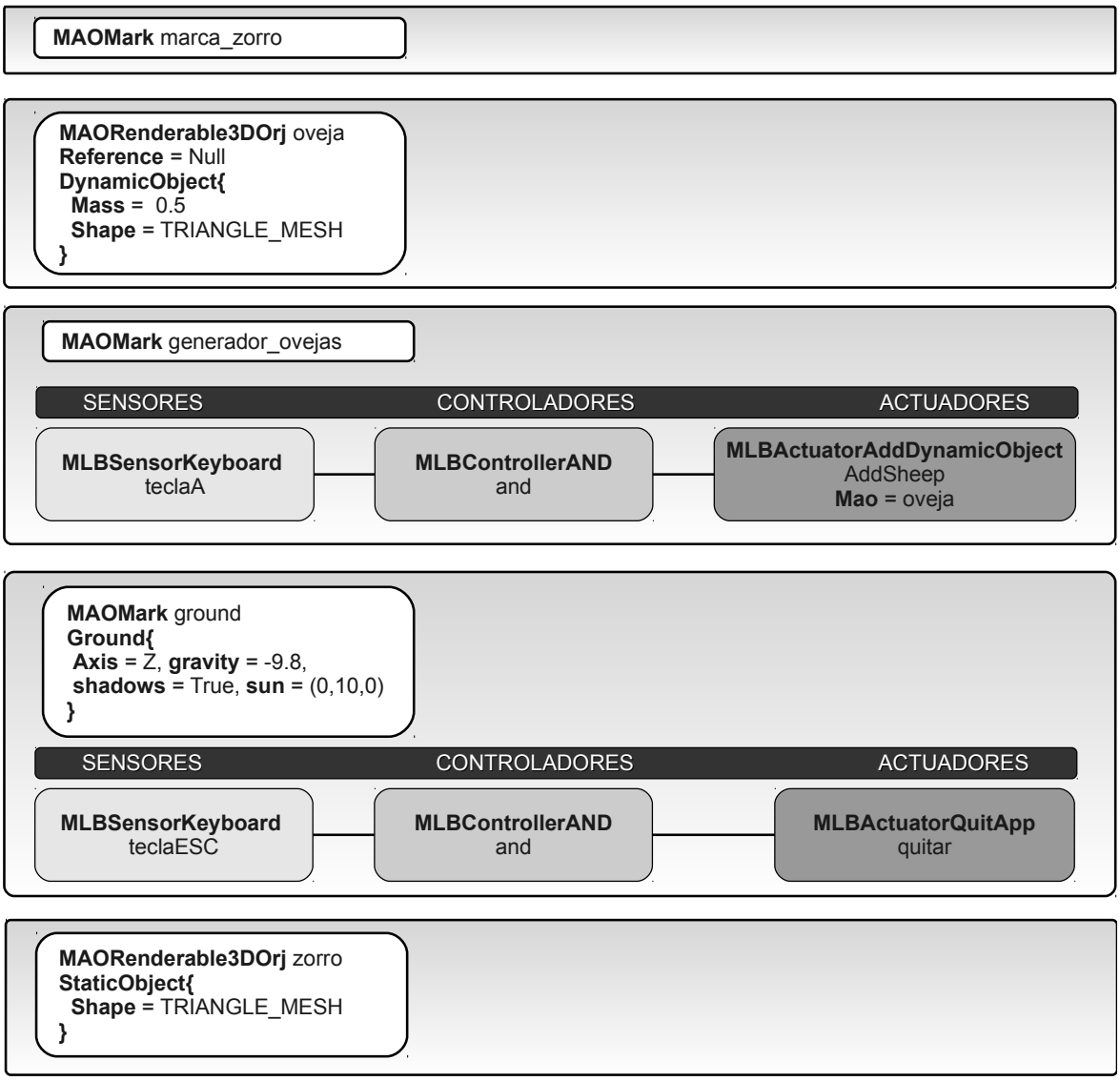

Figura A.11: Esquema de componentes de la aplicación ARSheep_Lite.

A.7. ARSheep

El objetivo de esta aplicación es demostrar el uso de la simulación física, los MAO's instanciados, y de paso ir cogiendo algo más de habilidad en la utilización de Minerva.

Básicamente la aplicación consiste en utilizar una marca como referencia del mundo físico, otra marca generará ovejas (MAO's instanciados) al pulsar la tecla A, y una última marca representará un zorro malvado (MAO estático) que podrá colisionar con las ovejas.

En la Figura A.11 se muestra el esquema de componentes de esta aplicación.

A continuación se explica paso a paso cada uno de los componentes que componen la aplicación:

- *Marca_zorro:* se declara un MAOMark convencional para usar como referencia para

el componente *zorro*.

- *Oveja:* es un MAORenderable3DOrej con la peculiaridad de que su parámetro *reference* es *Null*. Como se explicó en la Sección A.6, esto indica que el MAO no se dibujará directamente, pero se tomará como modelo para instanciar MAO's. Se le denomina *MAO Clase*.

  Además, para poder ser instanciado tiene que ser un objeto dinámico físico. Esto se indica con el bloque de sintaxis *DynamicObject*, donde se configura para tener una masa de 0.5 kilos, y calcular la forma de colisión como *TRIANGLE_MESH*.

- *Ground:* se declara una marca que se tomará como referencia para el suelo físico. Como se explicó en la Sección A.5, para activar la simulación física el primer paso es indicar un suelo. Esto se realiza con el bloque de sintaxis *Ground*, parametrizando la fuerza de la gravedad al eje X de la marca, en sentido contrario, con una magnitud de 9,8.

- *Zorro:* se trata de un MAORenderable3DOrej, cargando el modelo de un zorro y asociado a la marca *Marca_zorro*. Si no se añadiese más, simplemente se dibujaría con la marca, sin afectar lo más mínimo a la simulación física. Para poder añadirle la característica de provocar colisiones a los objetos físicos dinámicos es necesario declararlo como objeto físico estático. Esto se realiza con el bloque de sintaxis *StaticObject*, indicando que la forma de colisión es *TRIANGLE_MESH*.

A continuación se muestra el código completo de la aplicación:

```
/* ARSheep Lite! 2011
Key A: Generates a sheep! ;)
Key ESCAPE: Exits App.
Mark 1: Suelo!
Mark 6: Sheeps generator!
Mark 11: Zorro
*/

MAOWorld ARSheep{
   MAOMark marca_zorro{
      path = "resources/4x4_11.patt"
      size = .06
   }

   MAORenderable3DOrj oveja{
      size = .05
      path_orj = "resources/oveja.orj"
      path_tex = "resources/oveja.ppm"
      reference = Null

      DynamicObject{
        mass = .5
        shape = "TRIANGLE_MESH"
      }
```

```
25      }

27      MAOMark generador_ovejas{
28           path = "resources/4x4_6.patt"
29           size = .06

31           MLBSensorKeyboard teclaA {type = "KEYDOWN" key = "A"}

33           MLBControllerAND and

35           MLBActuatorAddDynamicObject addSheep {mao = oveja}

37           teclaA->and->addSheep
38      }

40      MAOMark ground{
41         path = "resources/4x4_1.patt"
42         size = .06

44         /* Exit app! */
45         MLBSensorKeyboard teclaESC {type = "KEYDOWN" key = "ESCAPE"}

47         MLBControllerAND and

49         MLBActuatorQuitApp quitar{}

51         teclaESC->and->quitar

53         Ground{
54           axis = "Z"
55            gravity = -9.8
56           shadows = True
57           sun = (0.,10.,0.)
58         }
59      }

61      MAORenderable3DOrj zorro{
62       size = 0.05
63       path_orj = "resources/zorro.orj"
64       path_tex = "resources/zorro.tga"
65       reference = marca_zorro

67       StaticObject{
68         shape = "TRIANGLE_MESH"
69       }
70    }
71  }
```

```
cont = mge.getCurrentController()

mao_padre = cont.getParent()

nombre_padre = mao_padre.getName()

print 'El nombre del MAO al que pertenece el script es: ',nombre_padre

sensor_tecla_q = cont.getSensor('teclaQ')

actuador_quitar = cont.getActuator('quitar')

if sensor_tecla_q.getState()==true:
    actuador_quitar.actuate()
```

Figura A.12: Ejemplo sencillo de scripting en Python.

A.8. Scripting

Para añadir funcionalidad más avanzada a una aplicación de Minerva, se puede utilizar *scripting* mediante Python. El scripting consiste en escribir código fuente en Python que Minerva ejecutará para evaluar la lógica de la aplicación.

En Minerva, el scripting se utiliza añadiendo un controlador denominado *MLBControllerScript*. Este controlador tendrá asociados, como cualquier otro, una lista de sensores y otra de actuadores. El script puede acceder al estado de los sensores, y puede activar los actuadores, además de poder cambiar sus propiedades. El módulo de Python que le permite acceder directamente a la funcionalidad de la arquitectura de Minerva se denomina MGE.

En la Figura A.12 se muestra un ejemplo de script que se explicará paso a paso.

Supóngase para el ejemplo de la Figura A.12 que el *MLBControllerScript* tiene asociados el sensor y el actuador del ejemplo de la Figura A.4. Lo primero es obtener el objeto *controller* mediante el módulo *mge* con la función *getCurrentController*.

Lo siguiente que realiza el ejemplo es obtener el *MAO Parent*, es decir, el MAO al que pertenece este controlador, e imprimir su nombre obtenido con la función *getName*.

Después se recuperan los objetos del sensor de la tecla Q y el actuador de cerrar la aplicación, mediante sendas funciones *getSensor* y *getActuator*. Con una condición *if* se puede comprobar el estado del sensor (es decir, si se ha pulsado la tecla Q) con la función *getState*, y en caso afirmativo activar el actuador mediante la función *actuate*.

Este código se guarda como un fichero con extensión *.py*, y se pasa al parámetro *path* en la sintaxis del *MLBControllerScript*.

Utilizar scripting en Minerva requiere conocimientos muy básicos del lenguaje de programación Python, pero añade mucha más potencia a la aplicación. En el Anexo F se encuentra toda la documentación detallada con relación a la API de Python.

A.9. ARCanyon_Lite

Para esa aplicación se va a ilustrar el uso del lenguaje de script para añadir funcionalidad avanzada a la aplicación. Además se utiliza la simulación física, propiedades de los MAO y MAO's instanciados explicados en secciones anteriores. En el CD adjunto se proporciona todo el material para ejecutar estas aplicaciones, y otras adicionales con mayor funcionalidad.

ARCanyon_Lite es un minijuego cuyo objetivo es derribar unas cajas disparando munición. En la Figura A.13 se puede ver el esquema de componentes de la aplicación.

A continuación se describen cada uno de estos componentes:

- *Municion:* este es el MAO Clase que representa la munición disparada. Al ser un MAO Clase, su parámetro *reference* es *Null*. Se trata de un MAORenderable3DOrej, que además es un objeto dinámico físico (requisito indispensable para poder ser un MAO instanciado, ver Sección A.6).

- *Enemigo1:* este es otro tipo de MAO Clase para añadir MAO's instanciados enemigos. Mediante el script se crearán varios de estos MAO's en una posición predeterminada. Con la munición el objetivo es intentar derribarlos.

- *Marca_objetivo:* este MAO tiene doble funcionalidad. Se usa como *Ground* para la utilización de la simulación física, mediante el bloque de sintaxis *Ground*. Además se utiliza como referencia para crear los objetos *Enemigo1*. Para ello se añade un sensor de teclado para la tecla *Enter*, y un actuador *MLBAddDynamicObject*. La intención es que cuando se pulse la tecla *Enter* se creen los objetos objetivo.

- *Canyon:* esta marca orienta los disparos de la munición. En realidad es la referencia de creación de los MAO's instanciados de los objetos *Municion*. Tiene dos sensores de teclado para la tecla *Space* (barra espaciadora), uno para la pulsación y otra para la liberación. Esto se utiliza para detectar cuánto tiempo ha estado pulsado la tecla, que se almacenará en la propiedad de usuario *tiempoDisparo*. En función de dicho tiempo se calculará la intensidad del impulso de creación del MAO instanciado, es decir, del disparo. *Script_objetivo:* este script crea cuatro objetos del tipo *enemigo1* cuando se pulsa la tecla *Enter*. El lugar donde se crean se controla modificando el offset del actuador. En la Figura A.14 se puede ver el código completo de dicho script.

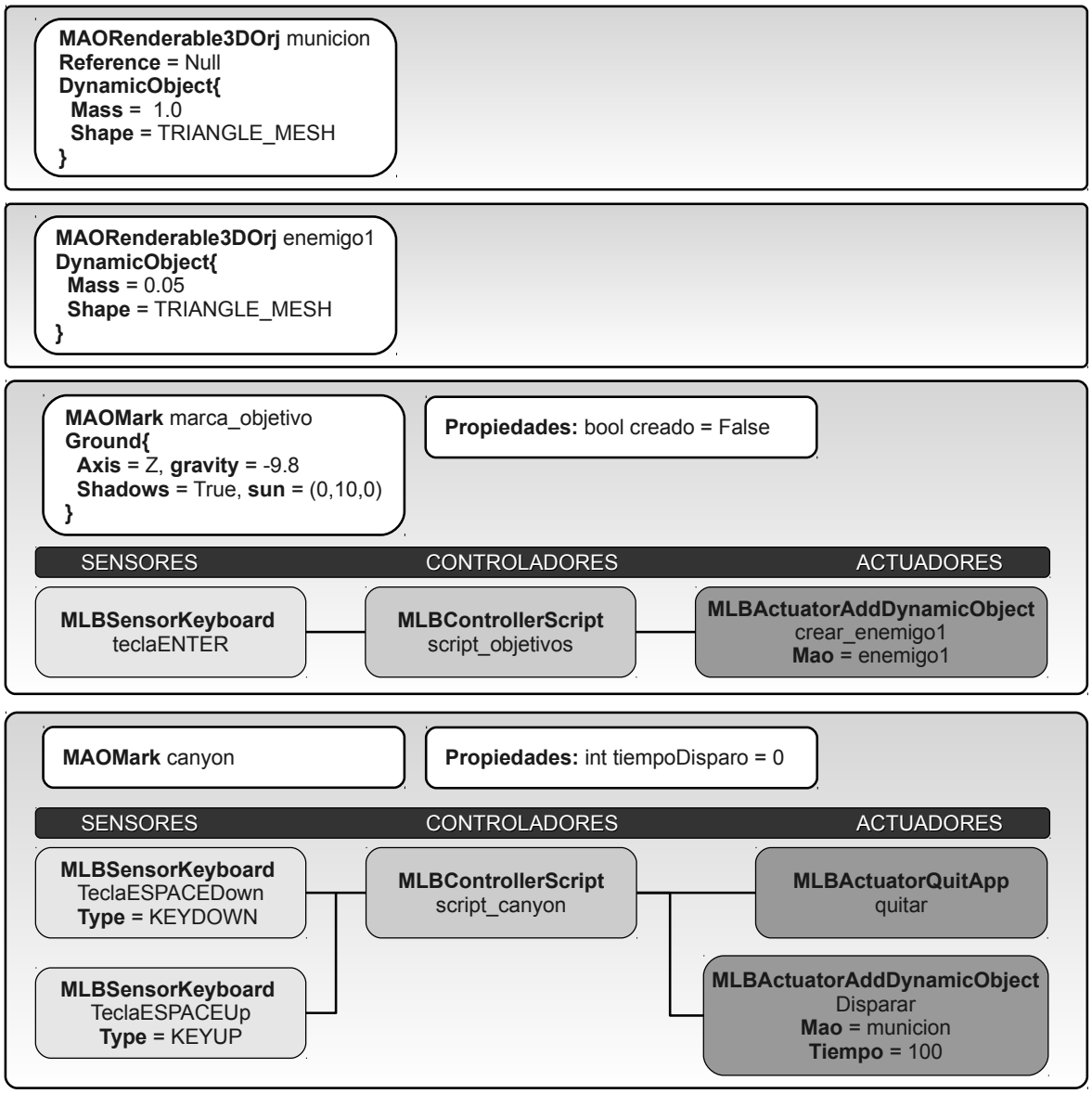

Figura A.13: Esquema de componentes de la aplicación ARCanyon_Lite.

- *Script_canyon:* este script controla el momento de pulsación y liberación de la tecla *Space*. Cuando se pulsa, empieza a aumentar el valor de la variable *tiempoDisparo*. Cuando se libera, calcula el impulso en función del valor de dicha variable, y la pone a 0. Además activa el actuador MLBAddDynamicObject para realizar el disparo. En la Figura A.15 se puede ver el código completo del script.

```
#!/usr/bin/python
#Script para MGE (Minerva Game Engine)
#Logica de control del canyon

controller = mge.getCurrentController()

#Recuperando la propiedad
padre = controller.getParent()
prop_tiempoDisparo = padre.getProperty('tiempoDisparo')

#Recuperando los sensores
teclaSpaceDown = controller.getSensor('teclaSPACEDown')
teclaSpaceUp = controller.getSensor('teclaSPACEUp')

#Recuperamos los actuadores
disparar = controller.getActuator('disparar')

#Se ha soltado la tecla... Disparamos!
if teclaSpaceUp.getState():
    tiempoDisparo = prop_tiempoDisparo.getValue()
    impulso = disparar.getImpulse()
    impulso[0]=0.0
    impulso[1]=0.0
    impulso[2]=0.5*float(tiempoDisparo)
    disparar.setImpulse(impulso)

    #Reiniciamos la cuenta
    prop_tiempoDisparo.setValue(0)

    #DISPARAMOS!
    disparar.actuate()

#Se presiona la tecla... contamos el tiempo!
elif teclaSpaceDown.getState():
    tiempoDisparo = prop_tiempoDisparo.getValue()
    tiempoDisparo = tiempoDisparo+1
    prop_tiempoDisparo.setValue(tiempoDisparo)
```

Figura A.14: Código del script de control de la lógica del objetivo.

```
#!/usr/bin/python
#ARSheep Lite 2011
#Script para el control de la logica del objetivo

#Recuperamos el controlador
controlador = mge.getCurrentController()

#Recuperamos los sensores!
teclaEnter = controlador.getSensor('teclaENTER')

#Recuperamos los actuadores!
crear_enemigo1 = controlador.getActuator('crear_enemigo1')

#Recuperamos las propiedades
prop_creado = controlador.getParent().getProperty('creado')
#print "Estado de la teclaS: "+teclaS.getState()
if teclaEnter.getState() and not prop_creado.getValue():
    prop_creado.setValue(True)
    offset = crear_enemigo1.getOffset()

    #Creamos un 1 enemigo1!
    offset[14]=0.05
    crear_enemigo1.setOffset(offset)
    crear_enemigo1.actuate()

    #Creamos un 2 enemigo1!
    offset[14]=0.11
    crear_enemigo1.setOffset(offset)
    crear_enemigo1.actuate()

    #Creamos un 3 enemigo1!
    offset[13]=0.07
    offset[14]=0.05
    crear_enemigo1.setOffset(offset)
    crear_enemigo1.actuate()

    #Creamos un 4 enemigo1!
    offset[13]=0.07
    offset[14]=0.11
    crear_enemigo1.setOffset(offset)
    crear_enemigo1.actuate()
```

Figura A.15: Código del script de control de la lógica del cañon.

A continuación se muestra el código completo de la aplicación en lenguaje MSL:

```
/* ARCanyon Lite! 2011
Objetivo: marca 6
Canyon: marca 1
Disparar: Barra espaciadora.
Crear Objetivo: ENTER.
*/

MAOWorld ARCanyon_Lite{
   /* Bola para disparar */
   MAORenderable3DOrj municion{
      size = 0.05
      path_orj = "resources/Orej/oveja.orj"
      path_tex = "resources/Orej/oveja.ppm"
      reference = Null

      DynamicObject{
         mass = 1.0
        shape = "TRIANGLE_MESH"
      }
   }

   MAORenderable3DOrj enemigo1{
      size = 0.05
      path_orj = "resources/Orej/cubo.orj"
      path_tex = "resources/Orej/cubo.ppm"
      reference = Null

      DynamicObject{
        mass = 0.005
         shape = "TRIANGLE_MESH"
      }
   }

   MAOMark marca_objetivo{
      path = "resources/4x4_6.patt"
      size = 0.058

      bool creado = False

      /* Sensores */
      MLBSensorKeyboard teclaENTER {type = "KEYDOWN" key = "RETURN"}

      /* Controladores */
      MLBControllerScript script_objetivos {path = "resources/objetivo.py
          "}

      /* Actuadores */
      MLBActuatorAddDynamicObject crear_enemigo1{mao = enemigo1}

      /* Links */
      teclaENTER->script_objetivos->crear_enemigo1

      /* Es Ground! */
      Ground{
        axis = "Z"
        gravity = -9.8
```

```
56            shadows = True
57           sun = (0.,10.,0.)
58        }
59     }

61     MAOMark canyon{
62        path = "resources/4x4_1.patt"
63        size = 0.06

65        /* Propiedades */
66        int tiempoDisparo = 0

68        /* Sensores */
69        MLBSensorKeyboard teclaESC {type = "KEYDOWN" key = "ESC"}

71        MLBSensorKeyboard teclaSPACEDown {type = "KEYDOWN" key = "SPACE"}

73        MLBSensorKeyboard teclaSPACEUp {type = "KEYUP" key = "SPACE"}

75        /* Controladores */
76        MLBControllerAND and
77        MLBControllerScript script_canyon{path = "resources/canyon.py"}

79        /* Actuadores */
80        MLBActuatorQuitApp quitar{}

82        MLBActuatorAddDynamicObject disparar{mao = municion time = 100}

84        /* Links */
85        teclaESC->and->quitar
86        teclaSPACEDown,teclaSPACEUp->script_canyon->disparar
87     }
88  }
```

APÉNDICE B

LISTA COMPONENTES MAO Y MLB

B.1. MAO's

- **MAOMark:** una única marca de ARToolKit.
- **MAOMarksGroup:** un grupo de marcas, cuyas posiciones relativas entre unas y otras son fijas.
- **MAORenderable2DImage:** una imagen en dos dimensiones que puede fijarse de forma absoluta en la pantalla.
- **MAORenderable2DText:** texto dinámico (se puede cambiar su contenido durante la ejecución de la aplicación) que puede fijarse de forma absoluta en la pantalla.
- **MAORenderable3DLine:** línea entre dos puntos del espacio.
- **MAORenderable3DPath:** ruta (conjunto de líneas) entre puntos del espacio.
- **MAORenderable3DOrj:** modelo de cuerpo rígido con posibilidad de animación en formato OreJ.
- **MAORenderable3DTeapot:** taza renderizada con glut.

B.2. MLB's

B.2.1. Sensores

- **MLBSensorActuator:** detecta la ejecución de un actuador.
- **MLBSensorAlways:** siempre está activo.
- **MLBSensorCollision:** detecta colisión entre dos MAORenderable3D.
- **MLBSensorDelay:** se activa cada cierto intervalo.
- **MLBSensorKeyboard:** detecta un evento de teclado.
- **MLBSensorNear:** detecta cuándo un MAORenderable3D con una cierta propiedad se acerca a menos de una determinada distancia al padre del MLB.
- **MLBSensorProperty:** detecta si una propiedad cumple una cierta condición.

- **MLBSensorRandom:** se activa de forma aleatoria de acuerdo a una determinada probabilidad.

B.2.2. Controladores

- **MLBControllerAND:** activa sus actuadores realizando la operación AND sobre todos sus sensores de entrada.
- **MLBControllerNAND:** activa sus sensores realizando la operación NAND sobre todos sus sensores de entrada.
- **MLBControllerOR:** activa sus sensores realizando la operación OR sobre todos sus sensores de entrada.
- **MLBControllerNOR:** activa sus sensores realizando la operación NOR sobre todos sus sensores de entrada.
- **MLBControllerScript:** ejecuta un determinado script de Python utilizando la API de Minerva. El usuario puede así controlar cuándo y cómo se activan sus actuadores, pudiendo variar los parámetros de estos.

B.2.3. Actuadores

- **MLBActuatorAddDynamicObject:** añade una copia del MAORenderable3D padre a la escena. El padre debe ser un objeto dinámico.
- **MLBActuatorAng:** calcula la diferencia de ángulo en cualquiera de los tres ejes entre el MAO padre y otro determinado.
- **MLBActuatorAnimOrej:** controla la animación del MAORenderable3DOrj padre. Puede realizar las acciones de play, pause y stop.
- **MLBActuatorChangePose:** realiza una transformación de localización (loc) y/o de rotación (rot).
- **MLBActuatorDistance:** calcula la distancia entre el MAO padre y otro determinado.
- **MLBActuatorPathAddPoint:** añade un punto al MAORenderable3DPath padre.
- **MLBActuatorPathRemovePoints:** elimina todos los puntos del MAORenderable3DPath padre.
- **MLBActuatorProperty:** cambia el valor de una determinada propiedad del padre.
- **MLBActuatorQuit:** finaliza la aplicación.
- **MLBActuatorRandom:** calcula un número aleatorio en el intervalo [0,1] y lo almacena en una propiedad determinada.
- **MLBActuatorRelativePose:** calcula la matriz de transformación entre un MAO de referencia determinado y el MAO padre. Lo almacena en una propiedad del tipo pose determinada.
- **MLBActuatorSound:** reproduce un determinado sonido.
- **MLBActuatorVisibility:** cambia la visibilidad del MAO padre.

APÉNDICE C

ESPECIFICACIÓN DE LOS MAO

LEYENDA

- El intérprete de MSL no tiene en cuenta los espacios en blanco o los saltos de línea, excepto para separar palabras.
- El orden de los parámetros de los componentes ***es importante***.
- Los términos encerrados por <*término*> deben ser sustituidos por un nombre del tipo indicado. Ejemplo: <*name*> por *marca1* ó <string>por *resources/imagen.jpg"*
- Los términos encerrados por *[término]* indica que son opcionales. Pueden aparecer o no.
- Los términos encerrados por *[término]**, indica que puede no aparecer, o aparecer un número indefinido de veces. Ejemplo: <*mark_name*>*[, <*mark_name*>]** por *marca1, marca2, marca3, marca4*.
- Dos términos separados por | indica que debe escogerse uno u otro, pero no los dos a la vez. Ejemplo: <*name*>| *Null* por *marca1* (si se opta por <*name*>) o *Null* (si se opta por la segunda opción).

MAOMark

DESCRIPCIÓN: Una única marca de *tracking* visual.
PROPIEDADES INTRÍNSECAS:
SINTÁXIS:

```
MAOMark <name>{
    path = <string>
    size = <float>
    [offset =  <pose>]

    [User Properties Definitions]
    [MLB Definitions}
    [MLB Links]

    [Ground {
        axis = <string:Constante de eje>
        gravity = <float>
        [shadows = <boolean>
        sun = <vector3f>]
    }]
}
```

MAOMarksGroup

DESCRIPCIÓN: grupo de marcas de registro visual, cuyas posiciones relativas unas de otras son fijas.
PROPIEDADES INTRÍNSECAS:
SINTÁXIS:

```
MAOMarksGroup <name>{
    marks = <mark name>[, <mark name>]*

    [User Properties Definitions]
    [MLB Definitions]
    [MLB Links]

    [Ground {
        axis = <string:Constante de ejes>
        gravity = <float>
        [shadows = <boolean>
        sun = <vector3f>]
    }]
}
```

MAORenderable2DImage

DESCRIPCIÓN: imagen en dos dimensiones que puede fijarse de forma absoluta en la pantalla.
PROPIEDADES INTRÍNSECAS:

- visible: *boolean*. Indica si el objeto es visible o no.
- width: *int*. Ancho de la imagen.
- height: *int*. Alto de la imagen.
- x: *int*. Posición absoluta del eje x.
- y: *int*. Posición absoluta del eje y.

SINTÁXIS:

```
MAORenderable2DImage <name>{
    path = <string>
    pos = <vector2i>
    width = <int>
    height = <int>

    [User Properties Definitions]
    [MLB Definitions]
    [MLB Links]
}
```

MAORenderable2DText

DESCRIPCIÓN: texto dinámico (su contenido puede cambiarse durante la ejecución de la aplicación) que puede fijarse de forma absoluta en la pantalla.
PROPIEDADES INTRÍNSECAS:

- visible: *boolean*
- width: *int*. (Sin uso).
- height: *int*. (Sin uso).
- x: *int*. Posición absoluta del texto en el eje x.
- y: *int*. Posición absoluta del texto en el eje y.
- text: *string*. Contenido del texto.
- ptSize: *int*. Tamaño del texto en puntos.
- r: *int*. Componente r del color del texto (entre 0 y 255).
- g: *int*. Componente g del color del texto (entre 0 y 255).
- b: *int*. Componente b del color del texto (entre 0 y 255).

SINTÁXIS:

```
MAORenderable2DText <name>{
    path = <string>
    size = <float>
    text = <string>
    pos = <vector2i>

    [User Properties Definitions]
    [MLB Definitions]
    [MLB Links]
}
```

MAORenderable3DLine

DESCRIPCIÓN: línea entre dos puntos del espacio tridimensional.
PROPIEDADES INTRÍNSECAS:

- size: *float*. Grosor de la línea.
- visible: *boolean*. Indica si la línea es visible.
- relative: *pose*. (Sin uso).
- mass: *float*. (Sin uso).
- r: *int*. Componente r del color de la línea.
- g: *int*. Componente g del color de la línea.
- b: *int*. Componente b del color de la línea.

SINTÁXIS:

```
MAORenderable3DLine <name>{
    size = <float>
    color = <vector3i>
    reference = <name point 1>
    reference = <name point 2>

    [User Properties Definitions]
    [MLB Definitions]
    [MLB Links]

    [DynamicObject{
        mass = <float>
        shape = <string:Constantes de forma de colision>
        [offset = <pose>]
        [impulse = <vector3f>]
    }
    |
    StaticObject{
        shape = <string>
    }]
}
```

MAORenderable3DPath

DESCRIPCIÓN: ruta (conjunto de líneas) entre puntos del espacio.
PROPIEDADES INTRÍNSECAS:

- size: *float*. Grosor de la línea de la ruta.
- visible: *boolean*. Indica si la ruta completa es visible.
- relative: *pose*. Posición del espacio donde se añadirá un punto.
- mass: *float*. (Sin uso).
- r: *int*. Componente r del color del siguiente punto.
- g: *int*. Componente g del color del siguiente punto.
- b: *int*. Componente b del color del siguiente punto.
- visiblePoint: *boolean*

SINTÁXIS:

```
MAORenderable3DPath <name>{
    size = <float>
    color = <vector3i>
    reference = <name> | Null

    [User Properties Definitions]
    [MLB Definitions]
    [MLB Links]

    [DynamicObject{
        mass = <float>
        shape = <string:Constantes de forma de colision>
        [offset = <pose>]
        [impulse = <vector3f>]
    }
    |
    StaticObject{
        shape = <string>
    }]
}
```

MAORenderable3DOrej

DESCRIPCIÓN: modelo de cuerpo rígido con posibilidad de animación en formato Orej.
PROPIEDADES INTRÍNSECAS:

- size: *float*. Tamaño (o escala) del objeto.
- visible: *boolean*. Indica si el objeto es visible.
- relative: *pose*. Posición en el espacio del objeto.
- mass: *float*. Masa del objeto.

SINTÁXIS:

```
MAORenderable3DOrj <name>{
    size = <float>
    path_orj = <string>
    path_tex = <string>
    reference = <name> | Null

    [User Properties Definitions]
    [MLB Definitions]
    [MLB Links]

    [DynamicObject{
        mass = <float>
        shape = <string:Constantes de forma de colision>
        [offset = <pose>]
        [impulse = <vector3f>]
    }
    |
    StaticObject{
        shape = <string>
    }]
}
```

MAORenderable3DTeapot

DESCRIPCIÓN: modelo en forma de taza.
PROPIEDADES INTRÍNSECAS:

- hola
- size: *float*. Tamaño (o escala) de la taza.
- visible: *boolean*. Indica si la taza es visible.
- relative: *pose*. Posición en el espacio de la taza.
- mass: *float*. Masa de la taza.

SINTÁXIS:

```
MAORenderable3DTeapot <name>{
    size = <float>
    reference = <identifier> | Null

    [User Properties Definitions]
    [MLB Definitions]
    [MLB Links]

    [DynamicObject{
        mass = <float>
        shape = <string:Constantes de forma de colision>
        [offset = <pose>]
        [impulse = <vector3f>]
    }
    |
    StaticObject{
        shape = <string>
    }]
}
```

APÉNDICE D

ESPECIFICACIÓN DE LOS MLB

LEYENDA

- El intérprete de MSL no tiene en cuenta los espacios en blanco o los saltos de línea, excepto para separar palabras.
- El orden de los parámetros de los componentes ***es importante***.
- Los términos encerrados por <*término*> deben ser sustituidos por un nombre del tipo indicado. Ejemplo: <*name*> por *marca1* ó <string>por *resources/imagen.jpg"*
- Los términos encerrados por *[término]* indica que son opcionales. Pueden aparecer o no.
- Los términos encerrados por *[término]**, indica que puede no aparecer, o aparecer un número indefinido de veces. Ejemplo: <*mark_name*>*[, <*mark_name*>]** por *marca1, marca2, marca3, marca4*.
- Dos términos separados por | indica que debe escogerse uno u otro, pero no los dos a la vez. Ejemplo: <*name*>| *Null* por *marca1* (si se opta por <*name*>) o *Null* (si se opta por la segunda opción).

D.1. Sensores

MLBSensorActuator

DESCRIPCIÓN: detecta la ejecución de un determinado actuador.
ÁMBITO: MAO.
SINTÁXIS:

```
1 MLBSensorActuator <name>{
2     actuator = <name>
3 }
```

MLBSensorAlways

DESCRIPCIÓN: sensor que siempre está activo.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBSensorAlways <name>{}
```

MLBSensorCollision

DESCRIPCIÓN: detecta si ha habido colisión entre el *MAORenderable3D* al que pertenece este MLB, y aquellos que tengan una determinada propiedad.
ÁMBITO: MAORenderable3D.
SINTÁXIS:

```
MLBSensorCollision <name>{
    property = <identifier>
}
```

MLBSensorDelay

DESCRIPCIÓN: se activa cada cierto intervalo, indicado en frames.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBSensorDelay <name>{
    time = <int>
}
```

MLBSensorKeyboard

DESCRIPCIÓN: detecta la pulsación o liberación de una tecla.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBSensorKeyboard <name>{
    type = <string: Constante de evento de teclado>
    key = <string:Constante de teclado>
}
```

MLBSensorNear

DESCRIPCIÓN: detecta si un *MAOPositionator3D* con una cierta propiedad está a menos distancia que una indicada del *MAOPositionator3D* al que pertenece este sensor.
ÁMBITO: MAOPositionator3D.
SINTÁXIS:

```
MLBSensorNear <name>{
    property = <name>
    distance = <float>
}
```

MLBSensorProperty

DESCRIPCIÓN: detecta si una propiedad cumple una determinada condición.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBSensorProperty <name>{
    type = <string:Constante comparacion aritmetica>
    property = <name>
    [value = <MAOValue>
    |
    value1 = <MAOValue>
    value2 = <MAOValue>
    |
    value = <name>
    ]
}
```

MLBSensorRandom

DESCRIPCIÓN: se activa de forma aleatoria de acuerdo a una determinada probabilidad.
ÁMBITO: MAO.
SINTÁXIS:

```
MLB <name>{
    probability = <float>
}
```

D.2. Controladores

MLBControllerAND

DESCRIPCIÓN: activa sus actuadores asociados realizando la operación AND sobre todos sus sensores de entrada.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBControllerAND <name>[, <name>]*
```

MLBControllerNAND

DESCRIPCIÓN: activa sus actuadores asociados realizando la operación NAND sobre todos sus sensores de entrada.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBControllerNAND <name>[, <name>]*
```

MLBControllerOR

DESCRIPCIÓN: activa sus actuadores asociados realizando la operación OR sobre todos sus sensores de entrada.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBControllerOR <name>[, <name>]*
```

MLBControllerNOR

DESCRIPCIÓN: activa sus actuadores asociados realizando la operación NOR sobre todos sus sensores de entrada.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBControllerNOR <name>[, <name>]*
```

MLBControllerScript

DESCRIPCIÓN: ejecuta un determinado script en Python utilizando la *API* de Minerva. El usuario puede así controlar cuándo y cómo se activarán sus actuadores asociados, pudiendo variar sus parámetros.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBControllerScript <name>{
    path = <string>
}
```

D.3. Actuadores

MLBActuatorAddDynamicObject

DESCRIPCIÓN: añade una copia de un *MAORenderable3D* (debe ser un objeto dinámico) indicado por el usuario a la escena. El *padre* debe ser un MAOPositionator3D que indica dónde se creará.
ÁMBITO: MAORenderable3D dinámico.
SINTÁXIS:

```
MLBActuatorAddDynamicObject <name>{
    mao = <name>
    [time = <int>]
    [offset = <pose>]
    [impulse = <vector3f>]
}
```

MLBActuatorAng

DESCRIPCIÓN: calcula la diferencia de ángulo en cualquiera de los tres ejes entre el MAO *padre* y otro determinado.
ÁMBITO: MAOPositionator3D.
SINTÁXIS:

```
MLBActuatorAng <name>{
    property = <name>
    ang_axis = <string:Constante de ejes>
}
```

MLBActuatorAnimOrej

DESCRIPCIÓN: controla la animación del *MAORenderable3DOrj padre*. Realiza acciones como *play, pause* o *stop*.
ÁMBITO: MAORenderable3DOrj.
SINTÁXIS:

```
MLBActuatorAnimOrej <name>{
    type = <string:Constante de operaciones de animacion>
    [anim_type = <string: Constante de tipo de animacion>]
}
```

MLBActuatorChangePose

DESCRIPCIÓN: realiza una transformación de localización (*loc*) y/o de rotación.
ÁMBITO: MAORenderable3D.
SINTÁXIS:

```
MLBActuatorChangePose <name>{
    [loc_type = <string:Constante de tipo de transformacion>
    location = <vector3f>
    |
    roc_type = <string: Constante de tipo de transformacion>
    rotation = <vector3f>
    ]+
}
```

MLBActuatorDistance

DESCRIPCIÓN: calcula la distancia entre el MAO *padre* y otro determinado.
ÁMBITO: MAOPositionator3D.
SINTÁXIS:

```
MLBActuatorDistance <name>{
    mao = <name>
    property = <name>
}
```

MLBActuatorPathAddPoint

DESCRIPCIÓN: añade un punto al MAORenderable3DPath padre. Este MLB no necesita ningún parámetro adicional. El color, tamaño y posición del punto que se añadirá dependerá del valor de las respectivas propiedades del MAORenderable3DPath. Estas propiedades son r, g, b, size y relative.
ÁMBITO: MAORenderable3DPath.
SINTÁXIS:

```
MLBActuatorPathAddPoint <name>{}
```

MLBActuatorPathRemovePoints

DESCRIPCIÓN: elimina todos los puntos del MAORenderable3DPath *padre*.
ÁMBITO: MAORenderable3DPath.
SINTÁXIS:

```
MLBActuatorPathRemovePoints <name>{}
```

MLBActuatorProperty

DESCRIPCIÓN: cambia el valor de una determinada propiedad del padre.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBActuatorProperty <name>{
    type = <string:Constante de operaciones aritmeticas>
    property = <name>
    value = <MAOValue> | <name>
}
```

MLBActuatorQuitApp

DESCRIPCIÓN: finaliza la aplicación.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBActuatorQuitApp <name>{}
```

MLBActuatorRandom

DESCRIPCIÓN: calcula un número aleatorio en el intervalo [0,1) y lo almacena en una propiedad determinada.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBActuatorRandom <name>{
    property = <name>
}
```

MLBActuatorRelativePose

DESCRIPCIÓN: calcula la matriz de transformación entre un MAO de referencia determinado y el MAO padre. Lo almacena en una propiedad del tipo pose.
ÁMBITO: MAOPositionator3D.
SINTÁXIS:

```
MLBActuatorRelativePose <name>{
    reference = <name>
    property = <name>
    inverse = <boolean>
}
```

MLBActuatorSound

DESCRIPCIÓN: reproduce un sonido utilizando el subsistema de audio.
ÁMBITO: MAO.
SINTÁXIS:

```
MLBActuatorSound <name>{
    path = <string>
}
```

MLBActuatorVisibility

DESCRIPCIÓN: cambia la visibilidad del MAO *padre*.
ÁMBITO: MAORenderable3D y MAORenderable2D.
SINTÁXIS:

```
MLBActuatorVisibility <name>{
    value = <boolean>
}
```

APÉNDICE E

CONSTANTES

Ejes

Parámetros que las admiten:

- *axis* de MAOMark
- *axis* de MAOMarksGroup
- *ang_axis* de MLBActuatorAng

Constantes

- X
- Y
- Z

Eventos de teclado

Parámetros que las admiten:

- *type* de MLBSensorKeyboard

Constantes

- KEYDOWN
- KEYUP

Comparaciónes aritméticas

Parámetros que las admiten:

- *type* de MLBSensorProperty

Constantes

- EQUAL
- NOTEQUAL
- INTERVAL

Formas de colisión

Parámetros que las admiten:

- *shape* de MAORenderable3DLine
- *shape* de MAORenderable3DPath
- *shape* de MAORenderable3DOrej
- *shape* de MAORenderable3DTeapot

Constantes

- BOX
- SPHERE
- CYLINDER
- TRIANGLE_MESH

Operaciones aritméticas

Parámetros que las admiten:

- *type* de MLBActuatorProperty

Constantes

- ASSIGN
- ADD
- MINUS
- MULTIPLY
- DIVIDE

Tipos de transformación

Parámetros que las admiten:

- *loc_type* de MLBActuatorChangePose
- *roc_type* de MLBActuatorChangePose

Constantes

- LOCAL
- GLOBAL

Operaciones de animación

Parámetros que las admiten:

- *type* de MLBActuatorAnimOrej

Constantes

- PLAY
- PAUSE
- STOP

Tipos de animación

Parámetros que las admiten:

- *anim_type* de MLBActuatorAnimOrej

Constantes

- SIMPLE
- LOOP
- PINGPONG

Teclas

Parámetros que las admiten:

- *key* de MLBSensorKeyboard

Constantes

- A
- B
- C
- D
- E
- F
- G
- H
- I
- J
- K
- L
- M
- N
- O
- P
- Q
- R
- S
- T
- U
- V
- W
- X
- Y
- Z
- UP
  Flecha arriba.
- DOWN:
  Flecha abajo.
- LEFT
  Flecha izquierda.
- RIGHT
  Flecha derecha.
- SPACE
  Barra espaciadora.
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- TAB
  Tabulador.
- RETURN
  Enter.
- ESCAPE | ESC
  Escape.
- F1
- F2
- F3
- F4
- F5
- F6
- F7
- F8
- F9
- F10
- F11
- F12
- RCTRL
  Tecla Control derecha.
- LCTRL
  Tecla Control izquierda.
- RSHIFT
  Tecla Shift derecha.
- LSHIFT
  Tecla Shift izquierda.

APÉNDICE F

API DE PYTHON

LEYENDA

- El formato de las cabeceras de las funciones aquí detalladas es el siguiente:
  `#Descripción`
  `nombreFuncion(tipo entrada): tipo devuelto.`
- El tipo *void* significa que no se devuelve ningún valor.
- Si no aparece nada entre los paréntesis de la función, significa que no tiene ningún parámetro de entrada.

F.1. Módulo MGE

El módulo MGE (*Minerva Game Engine*) es el que utiliza Python para poder acceder a la funcionalidad de Minerva. No hace falta importarlo, se realiza automáticamente.

Sólo proporciona una función, que es el punto de partida para comenzar el scripting:

```
1 #Devuelve el objeto MLBControllerScript.
2 getCurrentController(): MLBControllerScript.
```

En las sucesivas secciones se listan las funciones soportadas por cada uno de los componentes de Minerva.

F.2. Propiedades de los MAO

```
1 #Devuelve el nombre de la propiedad.
2 getName(): String
3
4 #Devuelve el nombre del valor.
5 getValue(): Value
6
7 #Establece el valor de la propiedad.
8 setValue(Value value): void
```

F.3. MAO

```
#Devuelve el nombre del MAO.
getName(): String

#Devuelve la propiedad con nombre <name> del MAO.
getProperty(String name): Property
```

F.4. MLB

F.4.1. MLBControllerScript

```
#Devuelve el nombre del MLB.
getName(): string

#Devuelve el MAO padre del MLB.
getParent(): MAO

#Devuelve el MLBSensor con nombre <name> del controlador.
getSensor(string name): MLBSensor

#Devuelve el MLBActuator con nombre <name> del controlador.
getActuator(string name): MLBActuator
```

F.4.2. Sensores

Todos los sensores disponen de las siguientes funciones:

```
#Devuelve el nombre del MLBSensor.
getName(): String

#Devuelve el estado (True o False) del MLBSensor.
getState(): Boolean
```

Aparte, hay sensores con funciones específicas:

- **MLBSensorCollision**

```
#Devuelve el nombre de la propiedad de colision.
getCollisionProperty(): String

#Establece el nombre de la propiedad de colision.
setCollisionProperty(String name): void
```

- **MLBSensorDelay**

```
#Devuelve el valor del delay.
getDelayFrames(): Int

#Establece el valor del delay.
setDelayFrames(Int delay): void
```

- **MLBSensorNear**

```
#Devuelve la distancia minima.
getMinDistance(): Float

#Establece la distancia minima.
setMinDistance(Float distance): void

#Devuelve el nombre de la propiedad de deteccion de cercania.
getNearProperty(): String

#Establece el nombre de la propiedad de deteccion de cercania.
setNearProperty(String name): void
```

- **MLBSensorProperty**

```
#Devuelve el tipo de comparacion aritmetica.
getType(): Int

#Establece el tipo de comparacion aritmetica.
setType(Int type): void
```

Las funciones *getType* y *setType* utiliza las siguientes constantes:

- MGE_EQUAL
- MGE_NOTEQUAL
- MGE_INTERVAL

- **MLBSensorRandom**

```
#Devuelve la probabilidad de activacion del MLBSensor.
getProbability(): Float

#Establece la probabilidad de activacion del MLBSensor.
setProbability(Float probability): void
```

F.4.3. Actuadores

Todos los actuadores disponen de las siguientes funciones:

```
#Devuelve el nombre del MLBActuator.
getName(): String

#Activa el MLBActuator.
actuate(): void
```

Aparte, hay actuadores con funciones específicas:

- **MLBActuatorAddDynamicObject**

```
#Devuelve el tiempo maximo de vida de los MAO's instanciados que
    crea.
getTimeToExpire(): Int

#Establece el tiempo maximo de vida de los MAO's instanciados que
    crea.
setTimeToExpire(Int time): void

#Devuelve el impulso inicial de los MAO's instanciados que crea.
getImpulse(): List

#Establece el impulso inicial de los MAO's instanciados que crea.
setImpulse(List impulse): void

#Devuelve el offset de los MAO's instanciados que crea.
getOffset(): List

#Establece el offset de los MAO's instanciados que crea.
setOffset(List offset): void
```

- **MLBActuatorAnimOrej**

```
#Devuelve el tipo de animacion.
getAnimType(): Int

#Establece el tipo de animacion.
setAnimType(Int type): void

#Devuelve la operacion a realizar con la animacion.
getAnimChoice(): Int

#Establece la operacion a realizar con la animacion.
setAnimChoice(Int choice): void
```

Las funciones *getAnimType* y *setAnimType* utilizan las siguientes constantes:

- MGE_SIMPLE
- MGE_LOOP
- MGE_PINGPONG

Las funciones *getAnimChoice* y *setAnimChoice* utilizan las siguientes constantes:

- MGE_PLAY
- MGE_PAUSE
- MGE_STOP

- **MLBActuatorChangePose**

```
#Devuelve el tipo de transformacion de localizacion.
getLocType(): Int

#Establece el tipo de transformacion de localizacion.
setLocType(Int type): void

#Devuelve el tipo de transformacion de rotacion.
getRotType(): Int

#Establece el tipo de transformacion de rotacion.
setRotType(Int type): void

#Devuelve la transformacion de localizacion.
getLoc(): List

#Establece la transformacion de localizacion.
setLoc(List loc) void

#Devuelve la transformacion de rotacion.
getRot(): List

#Establece la transformacion de rotacion.
setRot(List loc) void
```

Las funciones *getLocType*, *setLocType*, *getRotType* y *setRotType* utilizan las siguientes constantes:

- LOCAL
- GLOBAL

- **MLBActuatorProperty**

```
#Devuelve el tipo de operacion aritmetica.
getOpType(): Int

#Establece el tipo de opreacion aritmetica.
setOpType(Int type): void
```

Las funciones *getOpType* y *setOpType* utilizan las siguientes constantes:

- MGE_ASSIGN
- MGE_ADD
- MGE_MINUS
- MGE_MULTIPLY
- MGE_DIVIDE

- **MLBActuatorRelativePose**

```
#Devuelve si es o no una transformacion inversa.
getInverse(): Boolean

#Establece si realiza o no la transformacion inversa.
setInverse(Boolean inverse): void
```

- **MLBActuatorVisibility**

```
#Devuelve el valor de la visibilidad.
getVisibility(): Boolean

#Establece el valor de la visibilidad.
setVisibility(Boolean visibility): void
```

APÉNDICE G

ALGORITMO PARA EL RENDERIZADO DE SOMBRAS

El cálculo de las sombras se realiza mediante la obtención de la matriz de composición neta relativa a la proyección del objeto sobre un plano determinado. La implementación original está basada en un código proporcionado por Carlos González Morcillo.

```
1  void PhysicsController::calculateShadowsMatrix() {
2      cv::Mat& mGround = _maoGround->getPosMatrix();
3      btVector3 vz(mGround.at<float> (2, 0), mGround.at<float> (2, 1),
           mGround.at<float> (2, 2));
4      btVector3 p(mGround.at<float> (3, 0), mGround.at<float> (3, 1),
           mGround.at<float> (3, 2));

6      GLfloat Lx = _sun.x(), Ly = _sun.y(), Lz = _sun.z(); // Sun pos
7      GLfloat Nx = -vz.x(), Ny = -vz.y(), Nz = -vz.z();
8      GLfloat Cx = (GLfloat) p.x(), Cy = (GLfloat) p.y(), Cz = (GLfloat) p.
           z();

10     //Calculating the shadow
11     float a = Nx * Lx + Ny * Ly + Nz * Lz;
12     float b = Cx * Nx + Cy * Ny + Cz * Nz - a;

14     _shadowsMatrix[0] = Lx * Nx + b;
15     _shadowsMatrix[1] = Nx * Ly;
16     _shadowsMatrix[2] = Nx * Lz;
17     _shadowsMatrix[3] = Nx;
18     _shadowsMatrix[4] = Ny * Lx;
19     _shadowsMatrix[5] = Ly * Ny + b;
20     _shadowsMatrix[6] = Ny * Lz;
21     _shadowsMatrix[7] = Ny;
22     _shadowsMatrix[8] = Nz * Lx;
23     _shadowsMatrix[9] = Nz * Ly;
24     _shadowsMatrix[10] = Lz * Nz + b;
25     _shadowsMatrix[11] = Nz;
26     _shadowsMatrix[12] = -Lx * b - Lx * a;
27     _shadowsMatrix[13] = -Ly * b - Ly * a;
28     _shadowsMatrix[14] = -Lz * b - Lz * a;
29     _shadowsMatrix[15] = -a;
30 }
```

APÉNDICE H

EXPORTAR MODELOS EN FORMATO OREJ

En este anexo se explica cómo convertir un modelo tridimensional en formato *.blend* (creado por la suite de modelado Blender) a metaformato OreJ para ser utilizado en Minerva.

En la página www.blendswap.com se pueden encontrar cientos de modelos gratuitos licenciados bajo Creative Commons.

H.1. Aspectos a tener en cuenta

Es importante tener en cuenta el eje de coordenadas de una marca (ver Figura 5.3). El modelo tridimensional se situará en referencia a dicho eje, por lo que es importante tenerlo en cuenta a la hora de modelarlo en Blender.

Es recomendable centrar el modelo en el eje de coordenadas, para que el cálculo de las formas de colisión esféricas, cúbicas y cilíndricas funcionen mejor.

Otro aspecto es que OreJ sólo almacena un cuerpo. Esto es equivalente a que el modelo tridimensional en Blender debe estar compuesto por un único objeto, pues sólo se exportará uno.

Por último, el tamaño del modelo debe ser proporcional al utilizado en la aplicación de Minerva. Es una buena práctica utilizar la medida en **metros**, por lo que una unidad de blender equivale a un metro de la realidad. Hay que reducir el modelo hasta ajustarlo al tamaño deseado.

H.2. Exportar un modelo a OreJ

A continuación se enumeran los pasos a seguir para exportar correctamente un modelo al formato OreJ.

1. El primer paso es localizar el exportador. Este se encuentra en el CD, con el nombre de *ExportadorOreJ.py*. Se trata de un script en Python para Blender.

2. Se abre el modelo *.blend* que se quiere exportar con Blender. El script ha sido desarrollado para la versión 2.49 y anteriores.

3. **Reducir polígonos:** la computación de gráficos en tiempo real puede ser muy costosa. Es importante que los modelos tridimensionales tengan el mínimo posible de polígonos. Opcionalmente, se pueden reducir los polígonos antes de exportarlo, a cambio de perder calidad del modelo.

   La reducción de polígonos no es una herramienta integrada en Blender, y debe hacerse con otras externas.

4. **Convertir a malla triangular:** el formato OreJ almacena los modelos en forma de malla de polígonos triangulares. Si no está garantizado que el modelo lo sea, se debe transformar antes a malla triangular.

   Blender permite realizar esta conversión:

   a) Entrar al modo de edición (*Edit Mode*), pulsando la tecla *TAB* desde el modo objeto (*Object Mode*).

   b) Seleccionar todos los polígonos del objeto pulsando una vez la tecla *A*.

   c) Pulsar *Ctrl+T* para convertir de *Quads* a *Triangles*.

5. **Exportar texturas:** las texturas se cargan desde un fichero de imagen aparte del fichero de geometría. Para exportarlo desde el modelo de blender hay que seguir los siguientes pasos:

   a) Hacer click en el menú *File->External data->Unpack into files*

   b) Se despliega un nuevo menú. Seleccionar *Use files in current directory (create when neccesary)*.

   c) En el directorio donde se encuentra el *.blend* se ha creado un directorio nuevo llamado *textures*. En él se encuentra el fichero de imagen de la textura que se utiliza directamente en Minerva.

6. **Exportar modelo:** para exportar el modelo hay que seguir los siguientes pasos:

   a) Abrir una ventana en Blender del tipo *Text Editor*.

   b) Pulsar en el menú *Text->Open* y abrir el exportador *ExportadorOreJ.py*.

   c) En el script, cambiar el contenido de la variable *PATH* por el directorio donde quiere exportar el fichero *.obj*. Este valor puede ser su *home*.

   d) El nombre del fichero *.orj* será el del objeto de Blender a exportar.

   e) Estando en la ventana *Text Editor*, pulsar la combinación *ALT+P*.

   f) El modelo debe haberse exportado correctamente.

APÉNDICE I

ESPECIFICACIÓN MSLSCANNER.L

```
%option c++
%option noyywrap

%{
#include <Kernel/MSLParser.h>
using namespace std;
%}

%x COMMENT

DIGIT [0-9]
ID [A-Za-z][A-Za-z0-9_]*
SIGN {-|+}

%%

"MAOWorld"                      {return MSLParser::MAOWORLD;}
"MAOMark"                       {return MSLParser::MAOMARK;}
"MAOMarksGroup"                 {return MSLParser::MAOMARKSGROUP;}
"MAORenderable2DImage"          {return MSLParser::MAORENDERABLE2DIMAGE;}
"MAORenderable2DText"           {return MSLParser::MAORENDERABLE2DTEXT;}
"MAORenderable3DLine"           {return MSLParser::MAORENDERABLE3DLINE;}
"MAORenderable3DOrj"            {return MSLParser::MAORENDERABLE3DORJ;}
"MAORenderable3DPath"           {return MSLParser::MAORENDERABLE3DPATH;}
"MAORenderable3DTeapot"         {return MSLParser::MAORENDERABLE3DTEAPOT
    ;}

"MLBActuatorAddDynamicObject"   {return MSLParser::
    MLBACTUATORADDDYNAMICOBJECT;}
"MLBActuatorAng"                {return MSLParser::MLBACTUATORANG;}
"MLBActuatorChangePose"         {return MSLParser::MLBACTUATORCHANGEPOSE
    ;}
"MLBActuatorDistance"           {return MSLParser::MLBACTUATORDISTANCE;}
"MLBActuatorPathAddPoint"       {return MSLParser::
    MLBACTUATORPATHADDPOINT;}
"MLBActuatorPathRemovePoints"   {return MSLParser::
    MLBACTUATORPATHREMOVEPOINTS;}
"MLBActuatorProperty"           {return MSLParser::MLBACTUATORPROPERTY;}
```

```
34 "MLBActuatorQuitApp"             {return MSLParser::MLBACTUATORQUITAPP;}
35 "MLBActuatorRandom"              {return MSLParser::MLBACTUATORRANDOM;}
36 "MLBActuatorRelativePose"        {return MSLParser::
      MLBACTUATORRELATIVEPOSE;}
37 "MLBActuatorSound"               {return MSLParser::MLBACTUATORSOUND;}
38 "MLBActuatorVisibility"          {return MSLParser::MLBACTUATORVISIBILITY
      ;}
39 "MLBActuatorAnimOrej"            {return MSLParser::MLBACTUATORANIMOREJ;}

41 "MLBControllerAND"               {return MSLParser::MLBCONTROLLERAND;}
42 "MLBControllerNAND"              {return MSLParser::MLBCONTROLLERNAND;}
43 "MLBControllerNOR"               {return MSLParser::MLBCONTROLLERNOR;}
44 "MLBControllerOR"                {return MSLParser::MLBCONTROLLEROR;}
45 "MLBControllerScript"            {return MSLParser::MLBCONTROLLERSCRIPT;}

47 "MLBSensorActuator"              {return MSLParser::MLBSENSORACTUATOR;}
48 "MLBSensorAlways"                {return MSLParser::MLBSENSORALWAYS;}
49 "MLBSensorCollision"             {return MSLParser::MLBSENSORCOLLISION;}
50 "MLBSensorDelay"                 {return MSLParser::MLBSENSORDELAY;}
51 "MLBSensorKeyboard"              {return MSLParser::MLBSENSORKEYBOARD;}
52 "MLBSensorNear"                  {return MSLParser::MLBSENSORNEAR;}
53 "MLBSensorProperty"              {return MSLParser::MLBSENSORPROPERTY;}
54 "MLBSensorRandom"                {return MSLParser::MLBSENSORRANDOM;}

56 "Ground"                         {return MSLParser::GROUND;}
57 "DynamicObject"                  {return MSLParser::DYNAMICOBJECT;}
58 "StaticObject"                   {return MSLParser::STATICOBJECT;}


61 "name"                           {return MSLParser::PARAM_NAME;}
62 "path"                           {return MSLParser::PARAM_PATH;}
63 "size"                           {return MSLParser::PARAM_SIZE;}
64 "marks"                          {return MSLParser::PARAM_MARKS;}
65 "pos"                            {return MSLParser::PARAM_POS;}
66 "color"                          {return MSLParser::PARAM_COLOR;}
67 "width"                          {return MSLParser::PARAM_WIDTH;}
68 "height"                         {return MSLParser::PARAM_HEIGHT;}
69 "text"                           {return MSLParser::PARAM_TEXT;}
70 "path_orj"                       {return MSLParser::PARAM_PATH_ORJ;}
71 "path_tex"                       {return MSLParser::PARAM_PATH_TEX;}
72 "anim_type"                      {return MSLParser::PARAM_ANIM_TYPE;}
73 "reference"                      {return MSLParser::PARAM_REFERENCE;}
74 "mao"                            {return MSLParser::PARAM_MAO;}
75 "time"                           {return MSLParser::PARAM_TIME;}
76 "offset"                         {return MSLParser::PARAM_OFFSET;}
77 "impulse"                        {return MSLParser::PARAM_IMPULSE;}
78 "ang_axis"                       {return MSLParser::PARAM_ANG_AXIS;}
79 "property"                       {return MSLParser::PARAM_PROPERTY;}
80 "rot_type"                       {return MSLParser::PARAM_ROT_TYPE;}
81 "rotation"                       {return MSLParser::PARAM_ROTATION;}
82 "loc_type"                       {return MSLParser::PARAM_LOC_TYPE;}
83 "location"                       {return MSLParser::PARAM_LOCATION;}
84 "type"                           {return MSLParser::PARAM_TYPE;}
85 "value"                          {return MSLParser::PARAM_VALUE;}
86 "value1"                         {return MSLParser::PARAM_VALUE1;}
87 "value2"                         {return MSLParser::PARAM_VALUE2;}
88 "actuator"                       {return MSLParser::PARAM_ACTUATOR;}
89 "key"                            {return MSLParser::PARAM_KEY;}
```

```
90  "distance"                          {return MSLParser::PARAM_DISTANCE;}
91  "probability"                       {return MSLParser::PARAM_PROBABILITY;}
92  "inverse"                           {return MSLParser::PARAM_INVERSE;}
93  "gravity"                           {return MSLParser::PARAM_GRAVITY;}
94  "axis"                              {return MSLParser::PARAM_AXIS;}
95  "mass"                              {return MSLParser::PARAM_MASS;}
96  "shape"                             {return MSLParser::PARAM_SHAPE;}
97  "shadows"                           {return MSLParser::PARAM_SHADOWS;}
98  "sun"                               {return MSLParser::PARAM_SUN;}

100 "int"                               {return MSLParser::T_INTEGER;}
101 "float"                             {return MSLParser::T_FLOAT;}
102 "bool"                              {return MSLParser::T_BOOL;}
103 "string"                            {return MSLParser::T_STRING;}
104 "pose"                              {return MSLParser::T_POSE;}

106 "/*"                                {BEGIN(COMMENT);}
107 <COMMENT>"*/"                       {BEGIN(INITIAL);}
108 <COMMENT>[\n\t ]*                   {}
109 <COMMENT>.                          {}

111 "->"                                {return MSLParser::ARROW;}
112 "."                                 {return MSLParser::DOT;}
113 \"[^\n\t ]*\"                       {return MSLParser::STRING;}
114 "-"?{DIGIT}*"."{DIGIT}*     {return MSLParser::FLOAT;}
115 "-"?{DIGIT}*                        {return MSLParser::INTEGER;}
116 "False"                             {return MSLParser::BOOL;}
117 "True"                              {return MSLParser::BOOL;}
118 {ID}                                {return MSLParser::IDENTIFIER;}
119 [ \n\t]                             {/*Ignore every blank space*/}
120 .                                   {/*Any other literal, just return it! */
    return yytext[0];}
121 <<EOF>>                             {yyterminate();}

123 %%
```

APÉNDICE J

ESPECIFICACIÓN MSLPARSER.Y

```
%name MSLParser
%define LSP_NEEDED

%define MEMBERS\
    virtual ~MSLParser(){}\
    private:\
        yyFlexLexer lexer;\
        MAO* currentMAO;

%define LEX_BODY {return lexer.yylex();}

%define ERROR_BODY {std::cout <<std::endl<<"~ MSL Error Reporting ~"<<std
   ::endl;\
                    std::cout<< "-----------------------"<<std::endl;\
                    std::cout<< "Error encountered at line: "<<lexer.
                       lineno()<<std::endl;\
                    std::cout<< "Last symbol parsed: "<<lexer.YYText()<<
                       std::endl;\
                    std::cout<< "Exiting..."<<std::endl;\
                    exit(-1);\
                    }

//INCLUDES
//-----------------------------------------------------------------------
%header{
#include <cstdlib>
#include <FlexLexer.h>
#include <opencv/cv.h>
#include <btBulletDynamicsCommon.h>

#include <Kernel/MSLProperties.h>
#include <Factories/MAOFactory.h>
#include <Factories/MLBFactory.h>
#include <Kernel/Logger.h>

%}
```

```
36 //DATA TYPES
37 //------------------------------------------------------------------------

39 %union {
40     int int_type;
41     bool bool_type;
42     float float_type;
43     std::string* string_type;
44     cv::Mat* pose_type;
45     MSLProperties* param_type;
46     btVector3* vector3_type;
47     MAOValue* maovalue_type;
48     MAOProperty* maoproperty_type;
49     std::vector<std::string*>* vectorstr_type;
50 }



54 //TERMINALS
55 //------------------------------------------------------------------------

57 %token < param_type > INTEGER
58 %token < int_type > T_INTEGER
59 %token < float_type > FLOAT
60 %token < int_type > T_FLOAT
61 %token < string_type > STRING
62 %token < int_type > T_STRING
63 %token < bool_type > BOOL
64 %token < int_type > T_BOOL
65 %token < int_type > T_POSE
66 %token < string_type > IDENTIFIER
67 %token < int_type > ARROW
68 %token < int_type > DOT

70 %token < int_type > PARAM_NAME
71 %token < int_type > PARAM_PATH
72 %token < int_type > PARAM_SIZE
73 %token < int_type > PARAM_MARKS
74 %token < int_type > PARAM_POS
75 %token < int_type > PARAM_COLOR
76 %token < int_type > PARAM_WIDTH
77 %token < int_type > PARAM_HEIGHT
78 %token < int_type > PARAM_TEXT
79 %token < int_type > PARAM_PATH_ORJ
80 %token < int_type > PARAM_PATH_TEX
81 %token < int_type > PARAM_ANIM_TYPE
82 %token < int_type > PARAM_REFERENCE
83 %token < int_type > PARAM_MAO
84 %token < int_type > PARAM_TIME
85 %token < int_type > PARAM_OFFSET
86 %token < int_type > PARAM_IMPULSE
87 %token < int_type > PARAM_ANG_AXIS
88 %token < int_type > PARAM_PROPERTY
89 %token < int_type > PARAM_ROT_TYPE
90 %token < int_type > PARAM_LOC_TYPE
91 %token < int_type > PARAM_ROTATION
92 %token < int_type > PARAM_LOCATION
93 %token < int_type > PARAM_TYPE
```

```
94  %token < int_type > PARAM_VALUE
95  %token < int_type > PARAM_VALUE1
96  %token < int_type > PARAM_VALUE2
97  %token < int_type > PARAM_ACTUATOR
98  %token < int_type > PARAM_KEY
99  %token < int_type > PARAM_DISTANCE
100 %token < int_type > PARAM_PROBABILITY
101 %token < int_type > PARAM_INVERSE
102 %token < int_type > PARAM_GRAVITY
103 %token < int_type > PARAM_AXIS
104 %token < int_type > PARAM_MASS
105 %token < int_type > PARAM_SHAPE
106 %token < int_type > PARAM_SHADOWS
107 %token < int_type > PARAM_SUN


110 %token  MAOWORLD
111 %token  MAOMARK
112 %token  MAOMARKSGROUP
113 %token  MAORENDERABLE2DIMAGE
114 %token  MAORENDERABLE2DTEXT
115 %token  MAORENDERABLE3DLINE
116 %token  MAORENDERABLE3DORJ
117 %token  MAORENDERABLE3DPATH
118 %token  MAORENDERABLE3DTEAPOT

120 %token  MLBACTUATORADDDYNAMICOBJECT
121 %token  MLBACTUATORANG
122 %token  MLBACTUATORCHANGEPOSE
123 %token  MLBACTUATORDISTANCE
124 %token  MLBACTUATORPATHADDPOINT
125 %token  MLBACTUATORPATHREMOVEPOINTS
126 %token  MLBACTUATORPROPERTY
127 %token  MLBACTUATORQUITAPP
128 %token  MLBACTUATORRANDOM
129 %token  MLBACTUATORRELATIVEPOSE
130 %token  MLBACTUATORSOUND
131 %token  MLBACTUATORVISIBILITY
132 %token  MLBACTUATORANIMOREJ

134 %token  MLBCONTROLLERAND
135 %token  MLBCONTROLLERNAND
136 %token  MLBCONTROLLERNOR
137 %token  MLBCONTROLLEROR
138 %token  MLBCONTROLLERSCRIPT

140 %token  MLBSENSORACTUATOR
141 %token  MLBSENSORALWAYS
142 %token  MLBSENSORCOLLISION
143 %token  MLBSENSORDELAY
144 %token  MLBSENSORKEYBOARD
145 %token  MLBSENSORNEAR
146 %token  MLBSENSORPROPERTY
147 %token  MLBSENSORRANDOM

149 %token GROUND
150 %token DYNAMICOBJECT
151 %token STATICOBJECT
```

```
154 //NON TERMINALS
155 //------------------------------------------------------------------------
156 %type < int_type > integer
157 %type < bool_type > bool
158 %type < float_type > float
159 %type < string_type > string
160 %type < pose_type > pose
161 %type < string_type > identifier;
162 %type < vector3_type > vector2di;
163 %type < vector3_type > vector2df;
164 %type < vector3_type > vector3di;
165 %type < vector3_type > vector3df;
166 %type < maovalue_type > maovalue;
167 %type < maoproperty_type > maoproperty;

169 %type < int_type > begin
170 %type < int_type > maos
171 %type < int_type > mlbs
172 %type < int_type > links
173 %type < int_type > link
174 %type < vectorstr_type> list_mlbidentifiers

176 %type < int_type > mao_properties
177 %type < int_type > mao_property

179 %type < int_type > ground
180 %type < param_type > optparam_ground
181 %type < int_type > physicobj

183 %type < param_type > param_dynamicobj
184 %type < param_type > optparam_dynamicobj

186 %type < int_type > mao
187 %type < int_type > def_maomark
188 %type < int_type > def_maomarksgroup
189 %type < int_type > def_maorenderable2dimage
190 %type < int_type > def_maorenderable2dtext
191 %type < int_type > def_maorenderable3dorj
192 %type < int_type > def_maorenderable3dline
193 %type < int_type > def_maorenderable3dpath
194 %type < int_type > def_maorenderable3dteapot
195 %type < param_type > param_maomark
196 %type < param_type > optparam_maomark
197 %type < param_type > param_maomarksgroup
198 %type < param_type > param_maorenderable2dimage
199 %type < param_type > param_maorenderable2dtext
200 %type < param_type > param_maorenderable3dorj
201 %type < param_type > param_maorenderable3dline
202 %type < param_type > param_maorenderable3dpath
203 %type < param_type > param_maorenderable3dteapot

205 %type < int_type > mlb
206 %type < int_type > def_mlb
207 %type < int_type > mlbactuator
208 %type < int_type > def_mlbactuatoradddynamic
209 %type < int_type > def_mlbactuatorang
```

```
%type < int_type > def_mlbactuatorchangepose
%type < int_type > def_mlbactuatordistance
%type < int_type > def_mlbactuatorpathaddpoint
%type < int_type > def_mlbactuatorpathremovepoints
%type < int_type > def_mlbactuatorproperty
%type < int_type > def_mlbactuatorquitapp
%type < int_type > def_mlbactuatorrandom
%type < int_type > def_mlbactuatorrelativepose
%type < int_type > def_mlbactuatorsound
%type < int_type > def_mlbactuatorvisibility
%type < int_type > def_mlbactuatoranimorej
%type < param_type > param_mlbactuatoradddynamic
%type < param_type > optparam_mlbactuatoradddynamic
%type < param_type > param_mlbactuatorang
%type < param_type > param_mlbactuatorchangepose
%type < param_type > param_mlbactuatordistance
%type < param_type > param_mlbactuatorpathaddpoint
%type < param_type > param_mlbactuatorpathremovepoints
%type < param_type > param_mlbactuatorproperty
%type < param_type > param_mlbactuatorproperty2
%type < param_type > param_mlbactuatorquitapp
%type < param_type > param_mlbactuatorrandom
%type < param_type > param_mlbactuatorrelativepose
%type < param_type > param_mlbactuatorsound
%type < param_type > param_mlbactuatorvisibility
%type < param_type > param_mlbactuatoranimorej

%type < int_type > mlbcontroller
%type < int_type > def_mlbcontrollerand
%type < int_type > list_mlbcontrollerand
%type < int_type > def_mlbcontrollernand
%type < int_type > list_mlbcontrollernand
%type < int_type > def_mlbcontrolleror
%type < int_type > list_mlbcontrolleror
%type < int_type > def_mlbcontrollernor
%type < int_type > list_mlbcontrollernor
%type < int_type > def_mlbcontrollerscript

%type < int_type > mlbsensor
%type < int_type > def_mlbsensoractuator
%type < int_type > def_mlbsensoralways
%type < int_type > def_mlbsensorcollision
%type < int_type > def_mlbsensordelay
%type < int_type > def_mlbsensorkeyboard
%type < int_type > def_mlbsensornear
%type < int_type > def_mlbsensorproperty
%type < int_type > def_mlbsensorrandom
%type < param_type > param_mlbsensoractuator
%type < param_type > param_mlbsensoralways
%type < param_type > param_mlbsensorcollision
%type < param_type > param_mlbsensordelay
%type < param_type > param_mlbsensorkeyboard
%type < param_type > param_mlbsensornear
%type < param_type > param_mlbsensorproperty
%type < param_type > param_mlbsensorproperty2
%type < param_type > param_mlbsensorrandom

//------------------------------------------------------------------------
```

```
269 %start begin

271 //MAO HEADERS AND DEFINITIONS
272 //------------------------------------------------------------------------

274 %%

276 begin: MAOWORLD identifier '{' maos '}' {Logger::getInstance()->out("
       Parsing completed succesfully! Enjoy your app "+*$2);}
277 ;

279 maos: maos mao {}
280     | /* empty */ {}
281 ;

283 mao: def_maomark {}
284     | def_maomarksgroup {}
285     | def_maorenderable2dimage {}
286     | def_maorenderable2dtext {}
287     | def_maorenderable3dorj {}
288     | def_maorenderable3dline {}
289     | def_maorenderable3dpath {}
290     | def_maorenderable3dteapot {}
291 ;

293 mao_properties: mao_properties mao_property {}
294     | /* empty */ {}
295 ;

297 mao_property: T_INTEGER identifier '=' integer {currentMAO->
       addPropertyInt(*$2,$4);}
298             | T_INTEGER identifier{currentMAO->addPropertyInt(*$2);}
299             | T_FLOAT identifier '=' float {currentMAO->addPropertyFloat
                   (*$2,$4);}
300             | T_FLOAT identifier  {currentMAO->addPropertyFloat(*$2);}
301             | T_BOOL identifier '=' bool {currentMAO->addPropertyBoolean
                   (*$2,$4);}
302             | T_BOOL identifier {currentMAO->addPropertyBoolean(*$2);}
303             | T_STRING identifier '=' string {currentMAO->
                   addPropertyString(*$2,*$4);}
304             | T_STRING identifier {currentMAO->addPropertyString(*$2);}
305             | T_POSE identifier '=' pose {currentMAO->addPropertyPose(*$2
                   ,*$4);}
306             | T_POSE identifier {currentMAO->addPropertyPose(*$2);}
307 ;

309 //MAO Definitions
310 def_maomark: MAOMARK identifier '{' param_maomark {currentMAO = &
       MAOFactory::getInstance()->addMAOMark(*$2,*$4->string1,$4->float1); ((
       MAOMark*)currentMAO)->setOffsetMatrix($4->pose1);delete $4;}
       mao_properties mlbs links  ground '}' {}
311 ;
312 param_maomark:  PARAM_PATH '=' string PARAM_SIZE '=' float
       optparam_maomark {$$ = new MSLProperties(*$7); $$->string1 = $3; $$->
       float1 = $6; delete $7;}
313 ;
```

```
314 optparam_maomark: PARAM_OFFSET '=' pose { $$ = new MSLProperties(); $$->
        pose1 = $3;}
315     | /* empty */ {$$ = new MSLProperties();}
316 ;

318 def_maomarksgroup: MAOMARKSGROUP identifier '{' PARAM_MARKS
        param_maomarksgroup {currentMAO = &MAOFactory::getInstance()->
        addMAOMarksGroup(*$2);} mao_properties mlbs links '}' ground {}
319 ;

321 param_maomarksgroup: identifier ',' param_maomarksgroup{MAOMark& mark = (
        MAOMark&)MAOFactory::getInstance()->getMAOPositionator3D(*$1);((
        MAOMarksGroup*)currentMAO)->addMarktoGroup(mark);}
322                     |identifier {MAOMark& mark = (MAOMark&)MAOFactory::
                           getInstance()->getMAOPositionator3D(*$1);((
                           MAOMarksGroup*)currentMAO)->addMarktoGroup(mark);}
323 ;

325 def_maorenderable2dimage: MAORENDERABLE2DIMAGE identifier '{'
        param_maorenderable2dimage {currentMAO = &MAOFactory::getInstance()->
        addMAORenderable2DImage(*$2,*$4->string1,$4->btvector1->x(),$4->
        btvector1->y(),$4->int3,$4->int4);delete $4;} mao_properties mlbs
        links '}' {}
326 ;
327 param_maorenderable2dimage: PARAM_PATH '=' string PARAM_POS '=' vector2di
         PARAM_WIDTH '=' integer PARAM_HEIGHT '=' integer { $$ = new
        MSLProperties(); $$->string1 = $3; $$->btvector1 = $6;  $$->int3 = $9;
         $$->int4 = $12;}
328 ;

330 def_maorenderable2dtext: MAORENDERABLE2DTEXT identifier '{'
        param_maorenderable2dtext {currentMAO = &MAOFactory::getInstance()->
        addMAORenderable2DText(*$2,*$4->string1,$4->int1,*$4->string2,$4->
        btvector1->x(),$4->btvector1->y());delete $4;} mao_properties mlbs
        links '}' {}
331 ;
332 param_maorenderable2dtext: PARAM_PATH '=' string PARAM_SIZE '=' integer
        PARAM_TEXT '=' string PARAM_POS '=' vector2di { $$ = new MSLProperties
        (); $$->string1 = $3; $$->int1 = $6; $$->string2 = $9; $$->btvector1 =
         $12}
333 ;

335 def_maorenderable3dorj: MAORENDERABLE3DORJ identifier '{'
        param_maorenderable3dorj {currentMAO = &MAOFactory::getInstance()->
        addMAORenderable3DOrj(*$2,$4->float1,*$4->string1,*$4->string2,*$4->
        string3); delete $4;} mao_properties mlbs links physicobj'}' {}
336 ;
337 param_maorenderable3dorj: PARAM_SIZE '=' float PARAM_PATH_ORJ '=' string
        PARAM_PATH_TEX '=' string  PARAM_REFERENCE '=' identifier { $$ = new
        MSLProperties(); $$ -> float1 = $3; $$ -> string1 = $6; $$ -> string2
        = $9; $$->string3 = $12;}
338 ;

340 def_maorenderable3dline: MAORENDERABLE3DLINE identifier '{'
        param_maorenderable3dline {currentMAO = &MAOFactory::getInstance()->
        addMAORenderable3DLine(*$2,$4->float1,$4->btvector1->x(), $4->
        btvector1->y(), $4->btvector1->z(),*$4->string1,*$4->string2); delete
        $4;} mao_properties mlbs links physicobj '}' {}
```

```
341 ;
342 param_maorenderable3dline: PARAM_SIZE '=' float PARAM_COLOR '=' vector3di
        PARAM_REFERENCE '=' identifier PARAM_REFERENCE '=' identifier {$$ =
    new MSLProperties(); $$->float1 = $3; $$->btvector1 = $6; $$->string1
    = $9; $$->string2 = $12;}


345 def_maorenderable3dpath: MAORENDERABLE3DPATH identifier '{'
    param_maorenderable3dpath {currentMAO = &MAOFactory::getInstance()->
    addMAORenderable3DPath(*$2,$4->float1,$4->btvector1->x(),$4->btvector1
    ->y(),$4->btvector1->z(), *$4->string1); delete $4;} mao_properties
    mlbs links physicobj'}' {}
346 ;
347 param_maorenderable3dpath: PARAM_SIZE '=' float PARAM_COLOR '=' vector3di
        PARAM_REFERENCE '=' identifier {$$ = new MSLProperties(); $$->float1
    = $3; $$->btvector1 = $6; $$ -> string1 = $9 ;}
348 ;

350 def_maorenderable3dteapot: MAORENDERABLE3DTEAPOT identifier '{'
    param_maorenderable3dteapot {currentMAO = &MAOFactory::getInstance()->
    addMAORenderable3DTeapot(*$2,$4->float1, *$4->string1); delete $4;}
    mao_properties mlbs links physicobj '}' {}
351 ;
352 param_maorenderable3dteapot: PARAM_SIZE '=' float PARAM_REFERENCE '='
    identifier {$$ = new MSLProperties(); $$->float1 = $3; $$->string1 =
    $6;}
353 ;

355 //Bullet stuff!
356 ground: GROUND '{' PARAM_AXIS '=' string PARAM_GRAVITY '=' float
    optparam_ground '}' { PhysicsController::getInstance()->initPhysics();
        PhysicsController::getInstance()->setMAOGround(*((MAOPositionator3D*)
    currentMAO),*$5,$8,$9->bool1,$9->btvector1); delete $9;}
357     | /* empty */ {}
358 ;
359 optparam_ground: PARAM_SHADOWS '=' bool PARAM_SUN '=' vector3df { $$ =
    new MSLProperties(); $$->bool1 = $3; $$->btvector1 = $6;}
360     | /* empty */ {$$ = new MSLProperties();}
361 ;

363 physicobj: DYNAMICOBJECT '{' param_dynamicobj '}' {PhysicsController::
    getInstance()->addDynamicRigidBody(*((MAORenderable3D*)currentMAO),$3
    ->float1,$3->pose1,$3->btvector1,*$3->string1); delete $3;}
364     | STATICOBJECT  '{' PARAM_SHAPE '=' string '}' {PhysicsController::
        getInstance()->addStaticRigidBody(*((MAORenderable3D*)currentMAO)
        ,*$5); delete $5;}
365     | /* empty */ {}
366 ;

368 param_dynamicobj: PARAM_MASS '=' float PARAM_SHAPE '=' string
    optparam_dynamicobj {$$ = new MSLProperties(*$7); $$->float1 = $3; $$
    ->string1 = $6; delete $7;}
369 ;
370 optparam_dynamicobj: optparam_dynamicobj PARAM_OFFSET '=' pose { $$ = new
        MSLProperties(*$1); $$->pose1 = $4; delete $1;}
371                 | optparam_dynamicobj PARAM_IMPULSE '=' vector3df {$$ =
                    new MSLProperties(*$1); $$->btvector1 = $4; delete $1
                    ;}
```

```
372                 | /* empty */ {$$ = new MSLProperties();}
373 ;


376 //MLB HEADERS
377 //-------------------------------------------------------------------
378 mlbs: mlbs mlb {}
379         | /* empty */ {}
380 ;

382 mlb: def_mlb mlbactuator {}
383     | def_mlb mlbcontroller {}
384     | def_mlb mlbsensor {}
385 ;

387 def_mlb: {}
388 ;

390 mlbactuator: def_mlbactuatoradddynamic {}
391     | def_mlbactuatorang {}
392     | def_mlbactuatorchangepose {}
393     | def_mlbactuatordistance {}
394     | def_mlbactuatorpathaddpoint {}
395     | def_mlbactuatorpathremovepoints {}
396     | def_mlbactuatorproperty {}
397     | def_mlbactuatorquitapp {}
398     | def_mlbactuatorrandom {}
399     | def_mlbactuatorrelativepose {}
400     | def_mlbactuatorsound {}
401     | def_mlbactuatorvisibility {}
402     | def_mlbactuatoranimorej {}
403 ;

405 mlbcontroller: def_mlbcontrollerand {}
406     | def_mlbcontrollernand {}
407     | def_mlbcontrolleror {}
408     | def_mlbcontrollernor {}
409     | def_mlbcontrollerscript {}
410 ;

412 mlbsensor: def_mlbsensoractuator {}
413     | def_mlbsensoralways {}
414     | def_mlbsensorcollision {}
415     | def_mlbsensordelay {}
416     | def_mlbsensorkeyboard {}
417     | def_mlbsensornear {}
418     | def_mlbsensorproperty {}
419     | def_mlbsensorrandom {}
420 ;


423 //MLB DEFINITIONS
424 //-------------------------------------------------------------------

426 //MLB Actuators
427 def_mlbactuatoradddynamic: MLBACTUATORADDDYNAMICOBJECT identifier '{'
    param_mlbactuatoradddynamic '}' {MLBFactory::getInstance()->
    addMLBActuatorAddDynamicObject(*$2,currentMAO->getName(),*$4->string1,
```

```
    $4->int1,$4->pose1,$4->btvector1); delete $4;}
428 ;
429 param_mlbactuatoradddynamic: PARAM_REFERENCE '=' identifier
    optparam_mlbactuatoradddynamic {$$ = new MSLProperties(*$4); $$->
    string1 = $3; delete $4;}
430 ;
431 optparam_mlbactuatoradddynamic: optparam_mlbactuatoradddynamic PARAM_TIME
     '=' integer {$$ = new MSLProperties(*$1); $$->int1 = $4; delete $1; }
432     | optparam_mlbactuatoradddynamic PARAM_OFFSET '=' pose {$$ = new
        MSLProperties(*$1); $$->pose1 = $4; delete $1;}
433     | optparam_mlbactuatoradddynamic PARAM_IMPULSE '=' vector3df {$$ =
        new MSLProperties(*$1); $$->btvector1=$4; delete $1;}
434     | /*empty*/ {$$ = new MSLProperties();}
435 ;

437 def_mlbactuatorang: MLBACTUATORANG identifier '{' param_mlbactuatorang
    '}' {MLBFactory::getInstance()->addMLBActuatorAng(*$2,currentMAO->
    getName(),*$4->maoproperty1, *$4->string2); delete $4;}
438 ;
439 param_mlbactuatorang: PARAM_PROPERTY '=' maoproperty PARAM_ANG_AXIS '='
    string {$$ = new MSLProperties(); $$ -> maoproperty1 = $3; $$->string2
     = $6;}
440 ;

442 def_mlbactuatorchangepose: MLBACTUATORCHANGEPOSE identifier '{'
    param_mlbactuatorchangepose '}' {MLBFactory::getInstance()->
    addMLBActuatorChangePose(*$2, currentMAO->getName(),*$4->string1, $4->
    btvector1, *$4->string2, $4->btvector2); delete $4;}
443 ;
444 param_mlbactuatorchangepose: param_mlbactuatorchangepose
    param_mlbactuatorchangepose {$$ = new MSLProperties(*$1); $$->fill(*$2
    ); delete $1; delete $2;}
445 | PARAM_LOC_TYPE '=' string PARAM_LOCATION '=' vector3df {$$ = new
    MSLProperties(); $$->string1 = $3; $$->btvector1 = $6;}
446 | PARAM_ROT_TYPE '=' string PARAM_ROTATION '=' vector3df {$$ = new
    MSLProperties(); $$->string2 = $3; $$->btvector2 = $6;}
447 ;

449 def_mlbactuatordistance: MLBACTUATORDISTANCE identifier '{'
    param_mlbactuatordistance '}' {MLBFactory::getInstance()->
    addMLBActuatorDistance(*$2,currentMAO->getName(),*$4->string1, *$4->
    maoproperty1); delete $4;}
450 ;
451 param_mlbactuatordistance: PARAM_MAO '=' identifier PARAM_PROPERTY '='
    maoproperty {$$ = new MSLProperties(); $$->string1 = $3; $$->
    maoproperty1 = $6;}
452 ;

454 def_mlbactuatorpathaddpoint: MLBACTUATORPATHADDPOINT identifier '{'
    param_mlbactuatorpathaddpoint '}' {MLBFactory::getInstance()->
    addMLBActuatorPathAddPoint(*$2,currentMAO->getName()); delete $4;}
455 ;

457 param_mlbactuatorpathaddpoint: {$$ = new MSLProperties();}
458 ;

460 def_mlbactuatorpathremovepoints: MLBACTUATORPATHREMOVEPOINTS identifier
    '{'  param_mlbactuatorpathremovepoints '}' {MLBFactory::getInstance()
```

```
    ->addMLBActuatorPathRemovePoints(*$2,currentMAO->getName()); delete $4
    ;}
461 ;
462 param_mlbactuatorpathremovepoints: {$$ = new MSLProperties();}
463 ;

465 def_mlbactuatorproperty: MLBACTUATORPROPERTY identifier '{'
    param_mlbactuatorproperty '}' {MLBFactory::getInstance()->
    addMLBActuatorProperty(*$2,currentMAO->getName(),*$4->maoproperty1, *
    $4->maovalue1,*$4->string1); delete $4;}
466 | MLBACTUATORPROPERTY identifier '{'  param_mlbactuatorproperty2 '}' {
    MLBFactory::getInstance()->addMLBActuatorProperty(*$2,currentMAO->
    getName(),*$4->maoproperty1, *$4->maoproperty2,*$4->string1); delete
    $4;}
467 ;
468 param_mlbactuatorproperty: PARAM_TYPE '=' string PARAM_PROPERTY '='
    maoproperty PARAM_VALUE '=' maovalue {$$ = new MSLProperties(); $$ ->
    string1 = $3; $$ ->maoproperty1 = $6; $$ -> maovalue1 = $9;}
469 ;
470 param_mlbactuatorproperty2: PARAM_TYPE '=' string PARAM_PROPERTY '='
    maoproperty PARAM_VALUE '=' maoproperty {$$ = new MSLProperties(); $$
    ->string1 = $3; $$ ->maoproperty1 = $6; $$ -> maoproperty2 = $9;}
471 ;

473 def_mlbactuatorquitapp: MLBACTUATORQUITAPP identifier '{'
    param_mlbactuatorquitapp '}' {MLBFactory::getInstance()->
    addMLBActuatorQuitApp(*$2,currentMAO->getName()); delete $4;}
474 ;
475 param_mlbactuatorquitapp: { $$ = new MSLProperties();}
476 ;

478 def_mlbactuatorrandom: MLBACTUATORRANDOM identifier '{'
    param_mlbactuatorrandom '}' {MLBFactory::getInstance()->
    addMLBActuatorRandom(*$2, currentMAO->getName(),*$4->maoproperty1);
    delete $4;}
479 ;
480 param_mlbactuatorrandom: PARAM_PROPERTY '=' maoproperty {$$ = new
    MSLProperties(); $$ -> maoproperty1 = $3;}
481 ;

483 def_mlbactuatorrelativepose: MLBACTUATORRELATIVEPOSE identifier '{'
    param_mlbactuatorrelativepose '}' {MLBFactory::getInstance()->
    addMLBActuatorRelativePose(*$2,currentMAO->getName(),*$4->string1, *$4
    ->maoproperty1, $4->bool1); delete $4;}
484 ;
485 param_mlbactuatorrelativepose: PARAM_REFERENCE '=' identifier
    PARAM_PROPERTY '=' maoproperty PARAM_INVERSE '=' bool {$$ = new
    MSLProperties(); $$->string1 = $3; $$ -> maoproperty1 = $6; $$->bool1
    = $9;}
486 ;

488 def_mlbactuatorsound: MLBACTUATORSOUND identifier '{'
    param_mlbactuatorsound '}' {MLBFactory::getInstance()->
    addMLBActuatorSound(*$2, currentMAO->getName(),*$4->string1); delete
    $4;}
489 ;
490 param_mlbactuatorsound: PARAM_PATH '=' string {$$ = new MSLProperties();
    $$->string1 = $3; }
```

```
491 ;

493 def_mlbactuatorvisibility: MLBACTUATORVISIBILITY identifier '{'
        param_mlbactuatorvisibility '}' {MLBFactory::getInstance()->
        addMLBActuatorVisibility(*$2, currentMAO->getName(), $4->bool1);
        delete $4;}
494 ;
495 param_mlbactuatorvisibility: PARAM_VALUE '=' bool {$$ = new MSLProperties
        (); $$ -> bool1 = $3;}
496 ;

498 def_mlbactuatoranimorej: MLBACTUATORANIMOREJ identifier '{'
        param_mlbactuatoranimorej '}' { MLBFactory::getInstance()->
        addMLBActuatorAnimOrej(*$2, currentMAO->getName(), *$4->string1, $4->
        string2); delete $4;}
499 ;
500 param_mlbactuatoranimorej: PARAM_TYPE '=' string PARAM_ANIM_TYPE '='
        string {$$ = new MSLProperties(); $$->string1 = $3; $$ -> string2 = $6
        ;}
501     | PARAM_TYPE '=' string { $$ = new MSLProperties(); $$->string1 = $3
            ;}
502 ;

504 //MLB Controllers
505 def_mlbcontrollerand: MLBCONTROLLERAND list_mlbcontrollerand {}
506 ;
507 list_mlbcontrollerand: list_mlbcontrollerand ',' identifier {MLBFactory::
        getInstance()->addMLBControllerAND(*$3, currentMAO->getName());}
508     | identifier {MLBFactory::getInstance()->addMLBControllerAND(*$1,
            currentMAO->getName());}
509     | /*empty*/ {}
510 ;

512 def_mlbcontrollernand: MLBCONTROLLERNAND list_mlbcontrollernand {}
513 ;
514 list_mlbcontrollernand: list_mlbcontrollernand ',' identifier {MLBFactory
        ::getInstance()->addMLBControllerNAND(*$3, currentMAO->getName());}
515     | identifier {MLBFactory::getInstance()->addMLBControllerNAND(*$1,
            currentMAO->getName());}
516     | /*empty*/ {}
517 ;

519 def_mlbcontrolleror: MLBCONTROLLEROR list_mlbcontrolleror {}
520 ;
521 list_mlbcontrolleror: list_mlbcontrolleror ',' identifier {MLBFactory::
        getInstance()->addMLBControllerOR(*$3, currentMAO->getName());}
522     | identifier {MLBFactory::getInstance()->addMLBControllerOR(*$1,
            currentMAO->getName());}
523     | /*empty*/ {}
524 ;

526 def_mlbcontrollernor: MLBCONTROLLERNOR list_mlbcontrollernor {}
527 ;
528 list_mlbcontrollernor: list_mlbcontrollernor ',' identifier {MLBFactory::
        getInstance()->addMLBControllerNOR(*$3, currentMAO->getName());}
529     | identifier {MLBFactory::getInstance()->addMLBControllerNOR(*$1,
            currentMAO->getName());}
530     | /*empty*/ {}
```

```
531 ;

533 def_mlbcontrollerscript: MLBCONTROLLERSCRIPT identifier '{' PARAM_PATH
        '=' string '}' {MLBFactory::getInstance()->addMLBControllerScript(*$2,
        currentMAO->getName(),*$6); delete $6;}
534 ;


537 //MLB Sensors

539 def_mlbsensoractuator: MLBSENSORACTUATOR identifier '{'
        param_mlbsensoractuator '}' {MLBFactory::getInstance()->
        addMLBSensorActuator(*$2, currentMAO->getName(), *$4->string1); delete
         $4;}
540 ;
541 param_mlbsensoractuator: PARAM_ACTUATOR '=' identifier {$$ = new
        MSLProperties(); $$ -> string1 = $3;}
542 ;

544 def_mlbsensoralways: MLBSENSORALWAYS identifier '{'
        param_mlbsensoralways '}' {MLBFactory::getInstance()->
        addMLBSensorAlways(*$2, currentMAO->getName()); delete $4;}
545 ;
546 param_mlbsensoralways: {$$ = new MSLProperties();}
547 ;

549 def_mlbsensorcollision: MLBSENSORCOLLISION identifier '{'
        param_mlbsensorcollision '}' {MLBFactory::getInstance()->
        addMLBSensorCollision(*$2,currentMAO->getName(),*$4->string1); delete
        $4;}
550 ;
551 param_mlbsensorcollision: PARAM_PROPERTY '=' identifier { $$ = new
        MSLProperties(); $$ -> string1 = $3;}
552 ;

554 def_mlbsensordelay: MLBSENSORDELAY identifier '{'  param_mlbsensordelay
        '}' {MLBFactory::getInstance()->addMLBSensorDelay(*$2, currentMAO->
        getName(),$4->int1); delete $4;}
555 ;
556 param_mlbsensordelay: PARAM_TIME '=' integer {$$ = new MSLProperties();
        $$ -> int1 = $3;}
557 ;

559 def_mlbsensorkeyboard: MLBSENSORKEYBOARD identifier '{'
        param_mlbsensorkeyboard '}' {MLBFactory::getInstance()->
        addMLBSensorKeyboard(*$2, currentMAO->getName(),*$4->string1, *$4->
        string2); delete $4;}
560 ;
561 param_mlbsensorkeyboard: PARAM_TYPE '=' string PARAM_KEY '=' string { $$
        = new MSLProperties(); $$ -> string1 = $3; $$ -> string2 = $6;}
562 ;

564 def_mlbsensornear: MLBSENSORNEAR identifier '{' param_mlbsensornear '}' {
        MLBFactory::getInstance()->addMLBSensorNear(*$2, currentMAO->getName()
        ,*$4->string1,$4->float1); delete $4;}
565 ;
566 param_mlbsensornear: PARAM_PROPERTY '=' identifier PARAM_DISTANCE '='
        float {$$ = new MSLProperties(); $$ ->string1 = $3; $$->float1 = $6;}
```

```
567 ;

569 def_mlbsensorproperty: MLBSENSORPROPERTY identifier '{'
        param_mlbsensorproperty '}' {MLBFactory::getInstance()->
        addMLBSensorProperty(*$2, currentMAO->getName(),*$4->string1, *$4->
        maoproperty1, $4->maovalue1, $4->maovalue2); delete $4;}
570 | MLBSENSORPROPERTY identifier '{' param_mlbsensorproperty2 '}' {
        MLBFactory::getInstance()->addMLBSensorProperty(*$2, currentMAO->
        getName(),*$4->string1, *$4->maoproperty1, $4->maoproperty2); delete
        $4;}
571 ;
572 param_mlbsensorproperty: PARAM_TYPE '=' string PARAM_PROPERTY '='
        maoproperty PARAM_VALUE '=' maovalue{ $$ = new MSLProperties(); $$->
        string1 = $3; $$->maoproperty1 = $6; $$->maovalue1 = $9;}
573                          | PARAM_TYPE '=' string PARAM_PROPERTY '='
                               maoproperty PARAM_VALUE1 '=' maovalue
                               PARAM_VALUE2 '=' maovalue{ $$ = new
                               MSLProperties(); $$->string1 = $3; $$->
                               maoproperty1 = $6; $$->maovalue1 = $9; $$->
                               maovalue2 = $12;}
574 ;
575 param_mlbsensorproperty2: PARAM_TYPE '=' string PARAM_PROPERTY '='
        maoproperty PARAM_VALUE '=' maoproperty {$$ = new MSLProperties(); $$
        ->string1 = $3; $$->maoproperty1 = $6; $$->maoproperty2 = $9;}
576 ;

578 def_mlbsensorrandom: MLBSENSORRANDOM identifier '{' param_mlbsensorrandom
         '}' {MLBFactory::getInstance()->addMLBSensorRandom(*$2,currentMAO->
        getName(),$4->float1); delete $4;}
579 ;
580 param_mlbsensorrandom: PARAM_PROBABILITY '=' float {$$ = new
        MSLProperties(); $$->float1 = $3;}
581 ;

583 //LINKS DEFINITIONS
584 //------------------------------------------------------------------------

586 links: links link {}
587         | /* empty */ {}
588 ;
589 link:  list_mlbidentifiers ARROW identifier ARROW list_mlbidentifiers {
590                 for(unsigned int i=0;i<$1->size();i++){
591                     MLBFactory::getInstance()->addMLBLink(currentMAO->
                            getName(),*($1->at(i)),*$3);
592                 }

594                 for(unsigned int i=0;i<$5->size();i++){
595                     MLBFactory::getInstance()->addMLBLink(currentMAO->
                            getName(),*$3,*($5->at(i)));
596                 }
597                 delete $1; delete $5;}
598     | list_mlbidentifiers ARROW list_mlbidentifiers {
599                 for(unsigned int i=0;i<$1->size();i++){
600                     for(unsigned int j=0;j<$3->size();j++){
601                         MLBFactory::getInstance()->addMLBLink(currentMAO
                                ->getName(),*($1->at(i)),*($3->at(j)));
602                     }
603                 }
```

```
604                 delete $1; }

606 ;
607 list_mlbidentifiers: identifier ',' list_mlbidentifiers {
608                     $$ = new std::vector<std::string*>(); $$->push_back(
                            $1);
609                     for(unsigned int i=0;i<$3->size();i++){
610                         $$->push_back($3->at(i));
611                     }
612                     delete $3;
613                     }
614         | identifier {$$ = new std::vector<std::string*>(); $$->push_back
               ($1);}
615 ;


618 //BASIC DATA TYPES DEFINITION
619 //------------------------------------------------------------------------

621 integer : INTEGER { $$ = atoi(lexer.YYText());}
622 ;

624 bool : BOOL {if(strcmp(lexer.YYText(),"False")==0) $$ = false; else $$ =
        true;}
625 ;

627 float : FLOAT {$$ = atof(lexer.YYText());}
628 ;

630 string : STRING {int l = strlen(lexer.YYText()); $$ = new std::string(
        lexer.YYText()+1,l-2);}
631 ;

633 pose :  float float float float float float float float float float float
         float float float float float {float* f = new float[16];  cv::Mat* m
        = new cv::Mat(4,4,CV_32F,(void*) f); $$ = m;}
634 ;

636 maovalue: integer {$$ = new MAOValue(MAOPROPERTY_INT, $1);}
637         | bool {$$ = new MAOValue(MAOPROPERTY_BOOLEAN, $1)}
638         | float {$$ = new MAOValue(MAOPROPERTY_FLOAT, $1)}
639         | string {$$ = new MAOValue(MAOPROPERTY_STRING, $1)}
640         | pose {$$ = new MAOValue(MAOPROPERTY_POSE, $1)}
641 ;

643 maoproperty: identifier DOT identifier { $$ = &MAOFactory::getInstance()
        ->findProperty(*$1,*$3); }
644             |identifier {$$ = &MAOFactory::getInstance()->findProperty(
                    currentMAO->getName(),*$1);}
645 ;

647 vector2di: '(' integer ',' integer ')' { $$ = new btVector3($2,$4,-1);}
648 ;
649 vector2df: '(' float ',' float ')' { $$ = new btVector3($2,$4,-1);}
650 ;

652 vector3di: '(' integer ',' integer ',' integer ')' {$$ = new btVector3($2
        ,$4,$6);}
```

```
653 ;

655 vector3df: '(' float ',' float ',' float ')' {$$ = new btVector3($2,$4,$6
        );}
656 ;

658 identifier : IDENTIFIER { $$ = new std::string(lexer.YYText());}
659 ;

661 %%
```

APÉNDICE K

ALGORITMO PARA LA OBTENCIÓN DEL PLANO GROUND

Algoritmo para la obtención del plano del suelo a partir de la matriz de transformación de la marca de referencia del mundo físico.

```
1    GLfloat planeScale = .1;
2        cv::Mat& mground = ground.getPosMatrix();
3        btVector3 vx(mground.at<float> (0, 0), mground.at<float> (0, 1),
             mground.at<float> (0, 2));
4        btVector3 vy(mground.at<float> (1, 0), mground.at<float> (1, 1),
             mground.at<float> (1, 2));
5        btVector3 vz(mground.at<float> (2, 0), mground.at<float> (2, 1),
             mground.at<float> (2, 2));
6        btVector3 p(mground.at<float> (3, 0), mground.at<float> (3, 1),
             mground.at<float> (3, 2));
7        GLfloat E = -0.001; /* Micro gap between the plane ground and the
              shadows */

9        glDisable(GL_TEXTURE_2D);
10       glColor3f(.9, .9, .9);

12       /* Drawing the ground plane! */
13       glBegin(GL_POLYGON);
14       glVertex3f(p.x() + planeScale * (+vx.x() + vy.x()), p.y() +
             planeScale* (+vx.y() + vy.y()) - E, p.z() + planeScale * (+vx.
             z()+ vy.z()));
15       glVertex3f(p.x() + planeScale * (+vx.x() - vy.x()), p.y() +
             planeScale* (+vx.y() - vy.y()) - E, p.z() + planeScale * (+vx.
             z()- vy.z()));
16       glVertex3f(p.x() + planeScale * (-vx.x() - vy.x()), p.y() +
             planeScale* (-vx.y() - vy.y()) - E, p.z() + planeScale * (-vx.
             z()- vy.z()));
17       glVertex3f(p.x() + planeScale * (-vx.x() + vy.x()), p.y() +
             planeScale* (-vx.y() + vy.y()) - E, p.z() + planeScale * (-vx.
             z()+ vy.z()));
18       glEnd();
```

APÉNDICE L

EXTRACTO DE INFORME DE *profiling* CON *gprof*

```
Each sample counts as 0.01 seconds.
    %   cumulative   self
 time   seconds   seconds     name
 48.11      0.89     0.89     labeling2
 13.51      1.14     0.25     arModifyMatrix
  8.65      1.30     0.16     arParamObserv2Ideal
  5.41      1.40     0.10     arGetPatt
  4.32      1.48     0.08     arGetNewMatrix
  2.16      1.52     0.04     arGetRot
  1.62      1.55     0.03     arMatrixPCA
  1.08      1.61     0.02     arGetCode
  1.08      1.63     0.02     arGetLine2
  1.08      1.65     0.02     arMatrixMul
  1.08      1.57     0.02     btPoolAllocator::btPoolAllocator
  1.08      1.59     0.02     btSequentialImpulseConstraintSol
  0.54      1.66     0.01     btBoxShape::btBoxShape(btVector3
  0.54      1.68     0.01     btQuantizedBvh::sortAndCalcSplit
  0.54      1.71     0.01     btManifoldResult::addContactPoin
  0.54      1.72     0.01     btGjkPairDetector::getClosestPoi
  0.54      1.73     0.01     btCollisionDispatcher::defaultNe
  0.54      1.74     0.01     float& cv::Mat::at<float>(int, int)
  0.54      1.75     0.01     cv::Mat::create(int, int, int)
  0.54      1.76     0.01     cv::Mat::release()
  0.54      1.77     0.01     btSequentialImpulseConstraintSol
  0.54      1.80     0.01     btPersistentManifold::getCacheEn
  0.54      1.81     0.01     btConvexInternalShape::getMargin
  0.54      1.67     0.01     MAOProperty::getName()
  0.54      1.69     0.01     MAORenderable3D::setRelativeMatr
  0.54      1.70     0.01     MAORenderable3D::isVisible()
  0.54      1.78     0.01     MAO::getProperty(std::string)
  0.54      1.79     0.01     __gnu_cxx::new_allocator<Face>::
  0.54      1.82     0.01     std::vector<MAOProperty*, std::a
  0.54      1.83     0.01     std::vector<cv::Mat, std::alloca
  0.54      1.84     0.01     std::_Miter_base<__gnu_cxx::__no
  0.27      1.84     0.01     MAOPositionator3D::isPositioned(
```

APÉNDICE M

CÓDIGO FUENTE

Dada la extensión del código fuente (más de *12.000* líneas), únicamente se incluye el código fuente en su versión electrónica en el CD adjunto a este documento.

Dicho código fuente se estructura en los siguientes directorios:

- **/Controllers:** contiene aquellas clases que *controlan* algún subsistema.
- **/Factories:** contiene aquellas clases que se encargan de crear y gestionar objetos.
- **/Kernel:** contiene diferentes clases para el funcionamiento de Minerva.
- **/MPY:** contiene las clases que implementan la *API* de Minerva para Python.
- **/MAO:** contiene la implementación y jerarquía de los MAO.
- **/MLB:** contiene la implementación y jerarquía de los MLB.
  - **/Sensor:** contiene la implementación de los MLB sensores.
  - **/Controller:** contiene la implementación de los MLB controladores.
  - **/Actuator:** contiene la implementación de los MLB actuadores.

Bibliografía

[A+97] R.T. Azuma et al. A survey of augmented reality. *Presence-Teleoperators and Virtual Environments*, 6(4):355–385, 1997.

[A+06] M. Alfoncesa et al. *Compiladores e interpretes*. Pearson Prentice Hall, 2006.

[AMHH08] T. Akenine-Moller, E. Haines, y N. Hoffman. *Real-time rendering*. AK Peters, 2008.

[emb] Página web sobre embeber Python. `http://docs.python.org/extending/embedding.html`.

[Esc01] A. Escalera. *Vision por computador*. Prentice Hall, 2001.

[FMHW97] S. Feiner, B. MacIntyre, T. Hollerer, y 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.

[Hol92] D.E. Holmgren. Design and Construction of a 30-Degree See-Through Head-Mounted Display. *University of North Carolina Chapel Hill Department of Computer Science, Tech. Rep*, 1992.

[JPV04] S. Junestrand, X. Passaret, y D. Vázquez. *Domótica y hogar digital*. Editorial Paraninfo, 2004.

[jun] Mobile Augmented Reality, por Juniper Research. `http://www.bmw.com/com/en/owners/service/augmented_reality_introduction_1.html`.

[KB99] H. Kato y M. Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. En *iwar*, página 85. Published by the IEEE Computer Society, 1999.

[KD03] G. Klein y T. Drummond. Robust visual tracking for non-instrumented augmented reality. 2003.

[KS03] H. Kaufmann y D. Schmalstieg. Mathematics and geometry education with collaborative augmented reality. *Computers & Graphics*, 27(3):339–345, 2003.

[KT02] N. KAWASAKI y Y. TAKAI. Video Monitoring System for Security Surveillance based on Augmented Reality. En *Proceedings of the 12th International Conference on Artificial Reality and Telexistence*, páginas 4–6, 2002.

[Lip92] Seymour Lipschitz. *Algebra lineal*. McGraw-Hill, 1992.

[LWE05] K.S. Low, W.N.N. Win, y M.J. Er. Wireless sensor networks for industrial environments. 2005.

[MK94] P. Milgram y F. Kishino. A taxonomy of mixed reality visual displays. *IEICE Transactions on Information and Systems E series D*, 77:1321–1321, 1994.

[R+02] M. Rosenthal et al. Augmented reality guidance for needle biopsies: An initial randomized, controlled trial in phantoms+* 1. *Medical Image Analysis*, 6(3):313–320, 2002.

[S+05] D. Shreiner et al. *OpenGL programming guide*. Addison-Wesley, 2005.

[weba] Página web Boost libraries. `http://www.boost.org/doc/libs/1_46_1/libs/libraries.htm`.

[webb] Página web de AGS. `http://www.adventuregamestudio.co.uk/`.

[webc] Página web de Algodoo. `http://www.algodoo.com/wiki/Home`.

[webd] Página web de ARMediaPlugin. `http://www.inglobetechnologies.com/en/new_products/arplugin_su/info.php`.

[webe] Página web de ARToolKit. `http://www.hitl.washington.edu/artoolkit/`.

[webf] Página web de BazAR. `http://cvlab.epfl.ch/software/bazar/`.

[webg] Página web de Blender GameKit. `http://www.blender.org/gamekit/`.

[webh] Página web de Bullet Physics Library. `http://bulletphysics.org/wordpress/`.

[webi] Página web de EyePet. `http://www.eyepet.com`.

[webj] Página web de Google Sketchup. `http://sketchup.google.com/intl/es/`.

[webk] Página web de Invizimals. `http://www.invizimals.com`.

[webl] Página web de la Escuela Superior de Informática de Ciudad Real. `http://www.esi.uclm.es`.

[webm] Página web de Lego Mindstorms. `http://mindstorms.lego.com/en-us/default.aspx?domainredir=www.legomindstorms.com`.

[webn] Página web de Realidad Aumentada de BMW. `http://www.bmw.com/com/en/owners/service/augmented_reality_introduction_1.html`.

[webo] Página web de SmallBasic. http://smallbasic.sourceforge.net/.

[webp] Página web de Tips sobre código portable C++ de Mozilla Fundation. https://developer.mozilla.org/en/C___Portability_Guide.

[webq] Página web del juego YoFrankie! http://www.yofrankie.org/.