UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Informática

PROYECTO FIN DE CARRERA

"YAOMEV: Plataforma de Generación de Videojuegos Educativos Multidispositivo"

Autor: Jorge López González
Director: Carlos González Morcillo

Septiembre, 2011.

TRIBUNAL:

Presidente:

Vocal 1:

Vocal 2:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

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

Este documento fue compuesto con LaTeX. Imágenes generadas con Google Docs y BOUML.

A todos los gigantes,
porque sin su base
no lo habría conseguido.

Resumen

El desarrollo de videojuegos es el proceso mediante el cual se diseña e implementan los elementos que son necesarios en un videojuego. La creación de videojuegos se realiza en un ámbito multidisciplinar que involucra habitualmente a profesionales de la informática, del diseño, grafistas, músicos y especialistas en diversos aspectos de la ingeniería (inteligencia artificial, simulación física y un largo etcétera).

Existe hoy en día una amplia gama de dispositivos (móviles, tablets, etc...) que han popularizado enórmemente el desarrollo de videojuegos y ha permitido el nacimiento de una enorme comunidad de desarrolladores *"indies"* ajenos a la industria tradicional. Al hilo de este fenómeno, han surgido en España numerosos cursos, másters e incluso carreras que pretenden formar nuevos profesionales especializados en la creación de videojuegos.

El presente proyecto fin de carrera surge del deseo de proporcionar una herramienta de aprendizaje e introducción para programadores nóveles en este ámbito. *YAOMEV* está basado en herramientas y estándares libres con el fin de facilitar su uso por parte de todo aquel que desee dar sus primeros pasos en este apasionante mundo.

YAOMEV ofrece herramientas que facilitan la importación de gráficos y animaciones creadas en la principal suite de gráficos 3D libre (*Blender*), la reproducción de vídeo y audio, la interacción del jugador con el juego, el uso de estructuras que permiten el uso de árboles de comportamiento para definir la inteligencia artificial de los personajes y una completa biblioteca matemática que completan la ayuda ofrecida al desarrollador.

Gracias a la construcción de *YAOMEV* siguiendo las guías del patrón de arquitectura del software *Modelo-Vista-Controlador*, se introduce al desarrollador en conceptos avanzados de ingeniería del software y se posibilita que éste pueda ocuparse sólamente del aspecto que realmente le interese. De este modo, si el desarrollo requiere un esfuerzo adicional en la representación, se puede mejorar la capa de vista sin perjudicar el resto del entorno, si lo que necesita es enfocarse en la entrada y procesamiento de la interacción del usuario, se puede concentrar el esfuerzo en la capa de controladores y, si el principal objetivo es la creación de un juego, se pueden obviar las capas anteriores y dedicar toda la atención a la construcción del modelo interno del mismo (sus reglas) mediante el uso de árboles de comportamiento.

Por último, se ha desarrollado un juego demostrador llamado *Bit Them All!!!*, un videojuego educativo, fácilmente expandible, que pretende ser un ejemplo de lo que se puede lograr mediante el uso de la plataforma *YAOMEV*.

Abstract

The development of video games is the process whereby the necessary elements of a video game are designed and implemented. Video game creation is carried out in a multidisciplinary field, which usually involves computer science professionals, designers, graphic artists, musicians and specialists in various aspects of computer engineering (artificial intelligence, physic simulation, etc.).

Nowadays, there is a wide range of devices (mobile phones, tablets, etc.) that have become video game development really popular and, due to this fact, has allowed the birth of a huge community of *"indie"* developers outside the traditional industry. As a consecuence of this phenomena, a considerable amount of courses, masters, and even degrees, have arisen expecting to train new specialists in videogame creation.

The current project comes up with the hope of providing a learning and introductory tool for novice programmers in this field. *YAOMEV* is based on free standards and tools in order to ease its use and help everyone who is interested in being initiated in this fascinating world.

YAOMEV offers tools to ease the importation of graphics and animations created in the main free software 3D graphics suite (*Blender*), video and audio reproduction, the player-videogame interaction, the use of structures that allow the use of behavior trees to define the artificial intelligence of the characters, and a complete mathematic library that round up the help offered to the developer.

By building *YAOMEV* following the guidelines of the *Model-View-Controller* software architecture pattern, it introduces the developer into software engineering advanced concepts, allowing him to takes care just of the aspects he is really interested in. This way, if the development requires an additional effort in the representation part, the view layer can be improved without harming the rest of the environment, if the need is focused on the input and the processing of the user interaction, the efforts can be concentrated onto the controllers layer, and if the primary aim is the creation of a game, it is possible to skip the previous layers and pay all the attention to the construction of the video game intern model (its rules) by the use of behavior trees.

Finally, a game demo called *Bit Them All!!!* has ben developed, an educational video game, easily expandable that pretends to be an example of what can be achieved by the use of *YAOMEV*.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. El Mercado del Videojuego

El mundo de los videojuegos ha cambiado espectacularmente en los últimos 20 años, espoleado por los avances tecnológicos y la cada vez mayor intercomunicación existente. Su penetración en todas las capas de la sociedad ha crecido tanto que esta industria se ha convertido en una de las principales del mundo en ingresos.

Quizás uno de los principales cambios no ha sido a nivel tecnológico, se ha producido en la percepción que de los videojuegos se tiene en la sociedad. Reflejo de esto ha sido el reconocimiento conseguido en España como industria cultural en marzo de 2009 [3], en un intento por potenciar la industria española. Quedando así al mismo nivel que el cine, la música o las artes plásticas.

Y es que esta industria se ha consolidado en la última década como una de las más potentes que existen. En 2009 generó mundialmente ingresos por valor de 57600 millones de euros, de los cuales los videojuegos generaron la mayor parte con 41200 millones de euros (de acuerdo a los datos ofrecidos por el estudio de Ibis Capital [1]). Sirvan como comparación los datos

relativos a la industria del cine que, según europapress [5], generó en el mismo periodo 63.600 millones de euros o la industria de la música con 65000 millones, según Paul Verna en sus estudio *"Global Music: Tuning Into New Opportunities"*[16].

En España los datos también son alentadores. Según el informe *"Balance económico de la industria del videojuego 2010"* de aDeSe [4], el videojuego es la primera industria de ocio audiovisual e interactivo, con una cuota de mercado del 50 %. Y aunque los ingresos totales se han visto rebajados en un 5,2 % en comparación con 2009, la venta de software (los videojuegos) se ha mantenido en las mismas cifras, a pesar del contexto internacional, convirtiendonos así en el 4º mayor mercado mundial.

1.2. Aprendizaje mediante Videojuegos

Videojuegos y enseñanza. No hace mucho tiempo estos dos términos podrían haber parecido incompatibles, incluso antagónicos. Sin embargo como se apunta en el estudio *"Videojuegos y Educación."* del Ministerio de Educación y Ciencia [17], ya desde 1978 G. H. Ball, pionero en el campo de los videojuegos aplicados a la educación, mostraba en su estudio *"Telegames Teach More Than You Think"* dos líneas de investigación al respecto de esta relación [7], que posteriores trabajos han reforzado. En concreto subrayan el potencial de los videojuegos para:

- Desarrollar las capacidades espaciales.
- Favorecer destrezas intelectuales como la asimilación de conceptos numéricos, la comprensión lectora e incluso el estímulo de la lectura.

Aprovechando esto, el videojuego planteado para demostrar las características de la plataforma a crear, tiene como objetivos prioritatios la enseñanza y la diversión. Y aunque estos dos términos también pudieran parecer opuestos, son muchas las sagas que han surgido a lo largo de los años con el principal objetivo de mejorar alguna destreza. Podemos encontrar desde juegos que enseñan a cocinar, facilitar el aprendizaje de idiomas, pintura y un largo etcétera.

Según el estudio de la Universidad Autónoma de Madrid *"Aprendiendo con los videojuegos comerciales"* [18], no solo con videojuegos especialmente orientados a educación se logra facilitar el aprendizaje. Algunos juegos comerciales pueden convertirse en instrumentos de aprendizaje en el colegio y en un aliado para la comunicación y la transmisión de valores en casa.

A la hora de elegir la temática del juego se ha tenido en cuenta diversos factores como por ejemplo: que los juegos de agilidad mental son los preferidos por los españoles (de entre los videojuegos de desarrollo intelectual) [4] o las posibilidades para mostrar los diferentes aspectos de nuestra plataforma.

Así se plantea como uno de los objetivos del proyecto fin de carrera la construcción de un videojuego demostrador concreto de las posibilidades de **YAOMEV** inspirado en el título de PlayStation 2: *"Buzz: El Gran Reto"*[10], popular juego de preguntas y respuestas.

1.3. Motivación y Justificación

El desarrollo de videojuegos no es una tarea fácil. Implica una gran variedad de disciplinas y habitualmente equipos con decenas de personas. Según el estudio de Ibis Capital, el desarrollo de un videojuego comercial puede costar entre 7,4 y 9,7 millones de euros [1]. Cifras que pueden dar una idea de la enorme complejidad que puede conllevar un desarrollo de este tipo.

Sin embargo la evolución tecnológica ha permitido el crecimiento de una enorme comunidad de desarrolladores independientes que desde cero y con pocos recursos desarrollan videojuegos que incluso pueden llegar a convertirse en éxitos.

Hoy en día mediante el uso de herramientas libres es posible desarrollar un juego con una inversión mínima. Sin embargo usarlas puede llegar a implicar un excesivo consumo de tiempo aplicado al aprendizaje de su funcionamiento en lugar de al desarrollo del juego. Y es que este área de la informática, unido a la hetereogeneidad de las plataformas existentes, requieren de ámplios conocimientos en diversas áreas como son la geometría euclídea, simulación, informática gráfica, animación por computador, inteligencia artificial, diseño de interfaces persona-computador, optimización, amén de un largo etcétera.

Es por ello que nace la idea de **YAOMEV**, cuya principal motivación consiste en construir un entorno multidispositivo que facilite la introducción a este mundo a cualquiera que lo desee. Se hace por ello necesario basarlo en software y estándares libres, que permitan poner al alcance de cualquier diseñador las herramientas indispensables para comenzar la construcción de un videojuego, aislándole de los detalles de la implementación que no tengan que ver con el desarrollo estricto de los requisitos del juego en si. De esta manera se pretende favorecer el aprendizaje de este campo de la forma más sencilla posible.

1.4. Estructura del Documento

El presente documento se ha redactado en base a la normativa para proyectos fin de carrera de Escuela Superior de Informática de la Universidad de Castilla-La Mancha, estructurándose de la siguiente forma:

- **Capítulo 2: Objetivos del proyecto.** En este capítulo se exponen brevemente los requisitos del presente proyecto, clasificándolos en objetivos principales y específicos.
- **Capítulo 3: Antecedentes, estado de la cuestión** En este capítulo se presenta un estudio del estado del arte de las plataformas existentes para ayudar al desarrollo de juegos y de las tecnologías implicadas.
- **Capítulo 4: Método de trabajo.** En este capítulo se explica y justifica la metodología de desarrollo elegida, a la vez que se introducen brevemente las herramientas elegidas para la realización de este proyecto
- **Capítulo 5: Arquitectura.** En este capítulo se analizan los componente que forman la plataforma construida, así como la arquitectura de su juego demostrador, siendo ambas resultado tanto de la metodología elegida como de los requisitos establecidos En el estudio de cada componente funcional se detalla el diseño e implementación de cada uno de los sistemas y subsistemas creados, entrando en detalle sobre los problemas surgidos y las soluciones aportadas.

- **Capítulo 6: Evolución y costes.** En este capítulo se estudia el ciclo del proceso de desarrollo, mediante el análisis de las diferentes fases en las que se descompuso el trabajo que ha dado lugar a este proyecto y, posteriormente, se discuten los costes estimados asociados al mismo.

- **Capítulo 7: Resultados.** En este capítulo se comentan los resultados obtenidos al final del desarrollo, así como se realiza un análisis de rendimiento a partir de ejemplos realizados para experimentar con las posibilidades de **YAOMEV**.

- **Capítulo 8: Conclusiones y propuestas.** En este capítulo se hace un resumen de lo conseguido con la realización de este proyecto, se presentan las conclusiones obtenidas y se establecen unas posibles líneas de trabajo futuro.

Capítulo 2

Objetivos

2.1. Objetivos principales

Este proyecto cuenta con dos objetivos principales, uno consecuencia del otro. Como primer paso se pretende construir una plataforma genérica de creación de videojuegos multidispositivo. Una vez concluido ese paso se pretende demostrar las funciones de la misma mediante la creación de un videojuego propio de carácter educativo.

La consecución de estos objetivos queda supeditada a un esfuerzo muy serio por aplicar técnicas de diseño de ingeniería del software en su desarrollo, logrando con ello una plataforma que sea fácilmente comprendida por un programador y que permita comenzar a desarrollar juegos en el menor tiempo posible. Por ello la arquitectura seguida en la plataforma **YAOMEV** se ajustará al patrón de arquitectura del software conocido como **Modelo-Vista-Controlador** (desde ahora **MVC**).

Este patrón persigue la separación de los datos, la interfaz de usuario y la lógica de control y está especialmente orientado hacia sistemas de representación gráfica de datos.

2.2. Objetivos específicos

Tomando como base los dos objetivos principales descritos anteriormente, se plantea la siguiente relación de objetivos específicos.

2.2.1. Construcción de plataforma genérica de creación de videojuegos multidispositivo

- **Uso del patrón de arquitectura del software Modelo-Vista-Controlador.** La estructura en capas de **YAOMEV** estará guiada por este patrón, ofreciendo al usuario unos principios de diseño que respeten la filosofía de éste. La idea consiste en ofrecer al desarrollador un esqueleto, que ajustándose al patrón, le permita, sin mucho esfuerzo o conocimientos previos, crear un juego de manera rápida y sencilla. El usuario, de este modo, sólo tendrá que implementar aquellos elementos de la capa de control y de modelo específicos de su juego, quedando ocultos para el todo lo relativo a la capa de vista y de gestión de los diferentes sistemas que componen esta plataforma.

- **Creación de un motor gráfico.** Una de las cuestiones más importantes en un juego se corresponde con el aspecto visual. Agrupando diversas bibliotecas estándar multidispositivo, se construirá un pequeño motor gráfico independiente de la plataforma, que ofrezca herramientas de alto nivel y que, de forma eficiente, faciliten tanto la gestión como la representación de: gráficos 3D, gráficos 2D, cámaras, luces, etc... De esta forma, se aislará al usuario de la forma en que se representan los modelos usados.

- **Biblioteca para la creación y gestión de interfaces gráficas de usuario.** El control de la entrada suministrada por el usuario es otro de los aspectos a considerar. En consecuencia la plataforma proveerá un conjunto de widgets específico junto con un sistema de gestión asíncrona de eventos, empleando en su desarrollo diversos paradigmas de interacción Persona - Ordenador, así como se prestará especial atención a su extensibilidad futura y sencillez de uso.

- **Soporte para la reproducción y despliegue de elementos multimedia.** Se tendrá en cuenta la importancia y capacidad inmersiva que se consigue con efectos de sonido,

vídeo o síntesis de voz. Se contará por lo tanto con la creación de herramientas software que faciliten el despliegue de vídeo y audio en tiempo real en un videojuego, a la vez que se incorporarán utilidades para el uso de sistemas de síntesis de voz.

- **Compatibilidad con formatos estándar para el despliegue de modelos 3D con animación basada en vértices.** Tan importante como representar gráficos 3D es su almacenamiento y gestión. Se contempla otorgar soporte a algunos de los formatos estándar de descripción de geometría y animación basada en vértice.

- **Biblioteca matemática.** La representación de gráficos 3D o las transformaciones en un espacio tridimensional hacen un uso intensivo de matemática vectorial y matricial. En este proyecto se creará una pequeña biblioteca que simplifique el uso de: matrices de transformación, cuaternios, curvas y vectores de 2, 3 y 4 dimensiones.

- **Herramientas de carga de recursos.** Otro de los aspectos más importante en un juego es el de la carga de los recursos necesarios para la simulación. Estos pueden ser visuales o multimedia. **YAOMEV** ofrecerá herramientas genéricas que permitirán lidiar con la carga de recursos de forma sencilla, y transparente al programador, mediante el uso de ficheros de configuración XML.

2.2.2. Creación de videojuego educativo propio.

- **Diseño del videojuego.** Para hallar el perfecto equilibrio entre jugabilidad y diversión es fundamental estudiar y afinar las mecánicas del juego. Mediante el análisis de diversos concursos televisivos se intentará extraer la información necesaria para que, una vez procesada y aplicada al juego, sumerja al jugador en una situación similar a la que viven los concursantes televisivos.

- **Creación de entornos y modelos 3D.** La meta de introducir al jugador en la acción hace necesario dotar al juego de elementos gráficos originales especialmente pensados para otorgar al juego de personalidad propia. Para ello, mediante el uso de herramientas libres, se creará un conjunto de *assets* gráficos que representen el entorno de un concurso televisivo, junto con los personajes que lo pueblan.

- **Herramientas de soporte, scripts, xml, etc...** Como ayuda en el proceso de creación de los *assets* gráficos del juego se implementarán una serie de herramientas que faciliten su exportación e importación, es decir, simplificarán el pipeline de diseño. Entendiendo el pipeline como el proceso desde que se planea un nuevo *asset*, se crea (en un programa como Blender), se exporta a un formato intermedio y termina con su uso en el juego.

- **Uso de herramientas avanzadas de modelado de comportamientos.** En lugar de crear una enorme jerarquía de entidades que ofrezca todos los comportamientos necesarios para implementar las reglas del juego, se optará por la creación de una jerarquía simple. De esta forma y mediante el uso de diversos patrones de diseño, junto con la estructura de datos *Behavior Tree*, se creará un sistema que permita modelar los comportamientos de cualquier aspecto del flujo del juego.

Capítulo 3

Estado del arte

La realización de los objetivos expuestos en el capítulo anterior requieren de unos conocimientos que abarcan diversas áreas de la ingeniería informática, como son la programación gráfica, matemáticas, simulación o inteligencia artificial.

En este capítulo se ofrecerá una reseña de algunos de los campos estudiados para la realización de **YAOMEV** y **Bit Them All!!!**.

3.1. Bibliotecas gráficas

El proceso de renderizado de una escena 3D requiere una cantidad de cálculos de tal magnitud que es muy sencillo que estos sobrepasen la capacidad de la CPU de un ordenador.

Hoy en día ese problema se encuentra solucionado en gran medida puesto que casi todos los ordenadores cuentan con hardware específico para esta tarea (las tarjetas gráficas). Hoy en día existen APIs muy poderosas que actúan de interfaz entre este dispositivo y el desarrollador, aislándole además de los distintos tipos de hardware gráfico.

Existen hoy en día varios sistemas que proporcionan un API para el despliegue de gráficos 3D en un ordenador, sin embargo no todos ellos son adecuados para su uso en aplicaciones que requieren ejecutar simulaciones en tiempo real.

3.1.1. OpenGL

OpenGL es una especificación estándar para definir gráficos de alto rendimiento, que actúa como interfaz software del hardware gráfico. Esta interfaz consiste en alrededor de 700 comandos que pueden usarse para especificar los objetos y operaciones necesarias para producir aplicaciones interactivas tridimensionales.

Fue desarrollada originalmente por *Silicon Graphics* en 1992 y se usa ampliamente en aplicaciones CAD, de realidad virtual, simulación o videojuegos. Actualmente su especificación depende del *Khronos Group*, que al mantenerla abierta ha permitido que surjan distintas implementaciones libres.

Una de sus ventajas es que existen implementaciones para múltiples sistemas, lo que facilita enormemente la portabilidad de las aplicaciones.

3.1.2. DirectX

DirectX es una colección de APIs desarrolladas por Microsoft para ayudar al desarrollo de aplicaciones multimedia, con especial énfasis en la programación de videojuegos y en la reproducción de vídeo.

Es ampliamente usado en la industria del videojuego para el desarrollo de los motores gráficos.

Esta opción presenta grandes inconvenientes, pues es exclusiva de plataformas Windows, aunque hoy en día podemos encontrar una implementación de código abierto de su API para sistemas *Unix* y *X Window System*, aunque tienen el handicap de no ser 100 % compatibles.

3.2. Informática gráfica

3.2.1. Metaformatos para modelos 3D

Los metaformatos de definición de modelos 3D, son ficheros que cumplen con el propósito de codificar la información de los modelos creados en las aplicaciones de diseño gráfico para su almacenamiento y uso en aplicaciones gráficas.

Habitualmente un modelo tridimensional se define como un conjunto de vértices (un vértice incluye información sobre un punto del espacio 3D, su normal y su coordenada de textura) agrupados formando polígonos (en el mundo de la programación gráfica lo más habitual es que se usen triángulos). También puede estar definido como un conjunto de curvas o una mezcla entre ambas formas. Dependiendo del tipo formato del modelo la información adicional puede ser muy variada, pueden incluir información sobre materiales y texturas, animaciones definidas de formas distintas (bien sea animación de sólido - rígido, animación basada en vértice o animación basada en esqueletos) o propiedades físicas.

Se han evaluado diferentes tipos de formatos a la hora de elegir cual sería el más apto para los requisitos de este proyecto.

- **OreJ** Formato creado por el grupo *Oreto* de la Escuela Superior de Informática de Ciudad Real. Se basa en el formato *Obj*, añadiendo información sobre animación de sólido-rígido. Este formato tiene a favor su simplicidad y la ventaja de contar con un exportador en *Blender*, sin embargo sólo es útil para definir objetos rígidos cuya maya permanece inmóvil.

- **MD2** Este formato fue creado por *Id Software* para definir los modelos que usaba el motor gráfico *id Tech 2*. Este es el formato en videojuego *Quake II*. El uso de este formato es muy común en aplicaciones libres puesto que *Id Software* liberó el código del motor gráfico y del juego en 2001 bajo una licencia *GNU General Public License*. Las animaciones se consiguen mediante la interpolación de los fotogramas clave, a nivel de vértice, almacenados en el archivo.

  Este formato cubre las necesidades básicas de representación y animación de personajes, sin embargo su especificación limita mucho el número de polígonos y la velocidad

a la que transcurre la animación (15 frames por segundo).

- **MD3** Este formato es el sucesor del anterior. Fue desarrollado para dar soporte al motor gráfico *id Tech 3* creado para desarrollar *Quake III Arena*. Su especificación también fue liberada, junto con el código del motor y el juego en 2005, lo cual ha permitido que también sea muy popular su uso en motores y juegos libres.

  Este formato almacena información sobre la maya 3D, permitiendo que un modelo esté conectado con más modelos, independizando de este modo las diferentes partes de un personaje y con ello que se puedan mezclar diferente animaciones. Las animaciones se almacenan de la misma forma que en el formato *MD2*, en el archivo se encuentran los frames clave de vértice del modelo a lo largo de la animación.

  Contiene también información sobre el material y las texturas del modelo, junto con los datos de la *Bounding Box* y *Bounding Sphere* que rodea cada modelo.

3.3. Motores Gráficos

Tras estudiar las bibliotecas gráficas se procede a hacer un repaso de dos alternativas libres en cuanto a motores gráficos se refiere, con el requisito de ser compatibles con *OpenGL*.

3.3.1. Ogre

Ogre, acrónimo de *Object-Oriented Graphics Rendering Engine* es una biblioteca de desarrollo 3D con soporte para sistemas de escritorio y plataformas móviles.

Está escrito en C++ y su objetivo principal es la creación de videojuegos, ofrece un motor gráfico orientado a escenas que aislan al programador de las bibliotecas gráficas usadas. Debido a su versatilidad ha sido incluso usado en el desarrollo de videojuegos comerciales.

Como características se pueden contar la compatibilidad que ofrece con múltiples sistemas, el alto nivel de abstracción que ofrece al usar el motor gráfico y sus cuidados diseño y documentación.

Este motor es software libre bajo licencia MIT, lo cual ha generado a su alrededor una comunidad muy activa, que ha permitido que cuente con numerosa y abundante documentación

sobre cómo empezar a usarlo.

3.3.2. CrystalSpace

CrystalSpace es un kit de desarrollo software (SDK) que ofrece soporte para el uso de gráficos 3D en tiempo real en aplicaciones de realidad virtual o videojuegos.

Fue desarrollado por Jorrit Tyberghein en C++ usando un diseño orientado a objetos.

Entre sus principales características se encuentran su escalabilidad (gracias a su sistema de plugins), su portabilidad (se encuentra disponible en multitud de plataformas) y su integración con *Blender*.

Ofrece dos formas de trabajar con el, mediante el uso de la biblioteca con C++ o mediante scripting con *Python* o *XML* mediante el uso de *CEL* (Crystal Entity Layer), un conjunto de plugins y aplicaciones que ofrecen un conjunto de abstracciones que facilitan el proceso de creación de juegos.

3.4. Inteligencia Artificial

Existen diferentes aproximaciones a la hora de modelar la toma de decisiones de un agente. Dos de ellas son usar máquinas de estados o árboles de comportamiento.

3.4.1. Máquinas de estados

Una máquina de estados finita (Finite State Machine o FSM) es una abstracción matemática muy usada en el mundo de la computación para modelar comportamientos. Básicamente se compone de tres cosas: un conjunto finito de estados, un conjunto de condiciones de entrada y una función de transición que conecta unos estados con otros a partir de las condiciones de entrada [14].

El principal inconveniente de las máquinas de estados es que se vuelven muy complicadas de manejar o modificar cuando crecen demasiado.

3.4.2. Árboles de comportamiento

Los árboles de comportamiento son estructuras de datos que organizan los comportamientos de nuestro sistema y permiten tratar con ellos de forma muy sencilla.

Sus principios son muy simples y recogen ideas de las máquinas de estados jerárquicas, planificación y scripts. Su funcionamiento es muy sencillo, los árboles tienen dos tipos de nodos: compuestos y atómicos. Al ejecutarse pueden tener éxito o fallar.

Los nodos compuestos son nodos intermedios en el árbol, están formados por uno o más nodos hijos (un nodo hijo puede ser árbol de comportamiento) y ofrecen una manera de recorrer sus hijos siguiendo diferentes reglas. Estos nodos tienen éxito o fallan dependiendo de la ejecución de sus hijos.

Los nodos atómicos por otro lado pueden ser acciones a realizar en la simulación (cambiar algún atributo del juego, reproducir un sonido, usar un algoritmo, etc...) o condiciones (comprobar el estado de una variable).

Las aristas entre nodos indican a qué nodo se puede pasar.

Las principales ventajas de los árboles de comportamiento que permiten definir comportamientos de forma muy simple a la vez que favorecen mucho la reutilización de los mismos.

Algunos títulos comerciales que los usan para definir los comportamientos de sus personajes son *Grand Theft Auto*, *Halo* o *Spore*.

Capítulo 4

Método de trabajo

En este capítulo se expone el marco de trabajo, la metodología de desarrollo software elegida, así como las herramientas utilizadas en la implementación de **YAOMEV** y **Bit Them All!!!**.

4.1. Metodología de desarrollo

Para la creación de **YAOMEV** y **Bit Them All!!!** se ha optado por usar una adaptación de la metodología **SCRUM**, perteneciente al conjunto de metodologías de desarrollo software conocidas como ágiles [2].

La filosofía de las metodologías ágiles consiste en otorgar mayor valor al individuo; en el sentido de favorecer una colaboración estrecha entre el cliente y el equipo de desarrollo, y dentro del equipo mismo; a la vez que propone un desarrollo incremental del software con iteraciones muy cortas que permitan disponer de entregables rápidamente [2].

Por otro lado, la elección de **SCRUM** se ha debido a la experiencia previa con esta metodología y al excesivo coste relacionado con el uso de otras más estrictas. En el desarrollo de un motor de videojuegos, un intento de controlar todos los aspectos de la gestión del proceso de desarrollo es muy complicado y **SCRUM** propone una aproximación muy

interesante. Ken Schaber habla de este problema en [15], donde según sus propias palabras: *"Hasta hace no mucho, era aceptado que el proceso de desarrollo de software era un asunto bien conocido que podía ser totalmente planeado, estimado y acabado con éxito. Esto se ha probado incorrecto en la práctica.* **SCRUM** *asume que el proceso de desarrollo de sistemas es impredecible, complicado y sólo puede ser más o menos descrito como una progresión general".*

SCRUM por lo tanto, proporciona un marco de trabajo basado en un proceso iterativo e incremental, idóneo para proyectos flexibles, con requisitos inestables, que requieran capacidad de adaptación y rapidez [15].

SCRUM intenta sobre todo mejorar la forma en que se realiza el trabajo en equipo, intentando sacar orden del caos apostando por la autoorganización. Y aunque este proyecto ha sido realizado sólo por una persona, **SCRUM** aporta una serie de buenas prácticas que se han considerado muy útiles para abordar este desarrollo. Resultando que para el caso concreto de este proyecto, esta forma de organizar el trabajo era la deseable, pues se partía de un conocimiento moderado de la materia y de las herramientas de trabajo, todo lo cual hacía preveer un desarrollo en el que había altas probabilidades de que se incluyeran cambios.

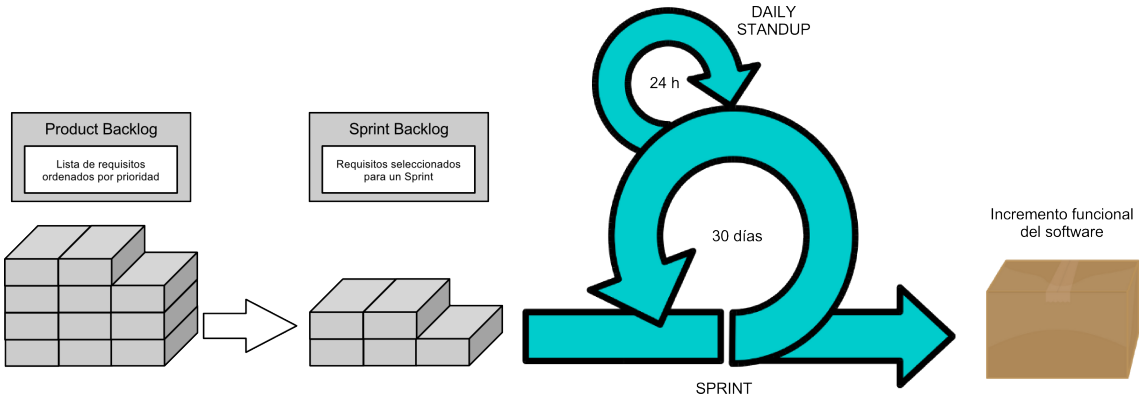

Figura 4.1: El proceso SCRUM

En la figura 4.1 se presenta el proceso de desarrollo de una aplicación usando **SCRUM** y a continuación se explica el mismo presentando la adaptación que se hizo de el para el desarrollo de este proyecto:

- El director de proyecto o cliente aportó en origen una propuesta para iniciar un desa-

rrollo software.

- En sucesivas reuniones se establecieron unos requisitos básicos, lo cuales fueron priorizados y añadidos a lo que se conoce en esta metodología como *Product Backlog*. Éste es un documento en el que se almacenan todos los requisitos ordenados por prioridad. El *Product Backlog* no es cerrado y puede cambiar en cualquier momento.

- Una vez se tenía especificada una primera versión del *Product Backlog*, en una nueva reunión se decidió un reparto de estas funcionalidades en las distintas iteraciones (conocidas en **SCRUM** como *Sprints*) que se querían realizar. El conjunto de requisitos a desarrollar en un *Sprint* es lo que se denomina como *Sprint Backlog*. Se estableció que cada uno de los *Sprints* debía tener una duración máxima de un mes, tras el cual debía haber un entregable que el cliente pudiera ejecutar y usar.

- Cada día de un *Sprint* se establece que debe haber una reunión del equipo de desarrollo en el que se comente qué tal fue el trabajo del día anterior y qué se tiene planificado para el actual. Es obvio que en un proyecto realizado por una sola persona esta parte del proceso no era necesaria.

- Al término de un *Sprint* se establecía una nueva reunión en la que se comprobaba, a parte del estado del desarrollo, qué requisitos se habían podido cumplir, cuales no y si era necesario añadir alguno más que no se hubiera planteado antes.

4.2. Herramientas

A continuación se realiza una breve descripción de todas las herramientas usadas en el desarrollo de este proyecto. **Lenguajes**

- **C++** - El pilar sobre el que se sostiene el desarrollo de este proyecto es C++, debido a que con este lenguaje es con el que se ha escrito la gran mayoría del código. Entre las características a destacar cabe recordar que es un lenguaje de programación multiparadigma, que debido a su versatilidad y potencia es, actualmente, ampliamente usado

en el desarrollo de videojuegos. Y es que este lenguaje es de los pocos que permitiendo programación de alto nivel, sigue permitiendo un acceso de bajo nivel al ordenador cuando es necesario, otorgando con ello a las aplicaciones escritas con este lenguaje un plus de rendimiento [9]. Todas estas características lo hacían indispensable para su uso en este proyecto.

- **Python** - Python ha sido usado para crear los exportadores de cámaras, escenarios y personajes desde Blender. Se usó la versión 2.5.2.

- **BASH** - El interprete de comandos BASH ha sido usado para la creación de algunos scripts de soporte para automatizar tareas tediosas como la conversión de las preguntas de **Bit Them All!!!** a audio empleando un módulo de síntesis de voz automática.

- **XML** - Ha sido usado para la creación de los diferentes archivos de configuración de **YAOMEV** y **Bit Them All!!!**, por su sencillez a la hora de definir lenguajes según las necesidades y su amplio uso hoy en día en el intercambio de información estructurada.

Compilación y desarrollo

- **Eclipse CDT** - Entorno integrado de desarrollo con soporte de C++. Se utilizó la versión Eclipse Helios 3.6.2, junto con el plugin Eclipse CDT 7.0.2.

- **GCC** - Se ha utilizado el compilador *g++* perteneciente a GCC, versión 4.3.1.

Edición 3D

- **Blender** - Poderosa suite de creación de contenido 3D de código abierto. Se usó la versión 2.49b junto con Python 2.5.2.

Control de Versiones

- **SVN** - Avanzado sistema de control de versiones. Se usó la versión 1.5.1.

- **Subclipse** - Plugin para Eclipse que facilita el control del versiones en un proyecto. Se usó la versión 1.6.18.

Documentación

- **Latex** - Sistema de composición de textos orientado a la creación de libros, documentos científicos y técnicos, usado para escribir esta documentación. Se usó el paquete TeX Live versión 2007.dfsg.1-2.
- **Doxygen** - Sistema de documentación de código fuente, compatible con una gran variedad de lenguajes, entre los que está C++.
- **Inkscape** - Programa de edición de imágenes vectoriales. Se usó la versión 0.46.
- **Gimp** - Programa de manipulación de imágenes. Se usó la versión 2.6.1.

Bibliotecas

- **OpenGL** - Especificación estándar de un API multiplataforma que tiene como propósito servir para escribir aplicaciones que produzcan gráficos 2D o 3D. Se usó el paquete freeglut3 versión 2.4.0.
- **SDL** - Biblioteca multimedia, multiplataforma, diseñada para proveer de acceso de bajo nivel a audio, teclado, ratón, hardware 3D y al framebuffer de vídeo 2D. Se usó la versión 1.2.
- **SDL_mixer** - Biblioteca que envuelve a mixer para SDL. Se usó la versión 1.2.8.
- **SDL_gfx** - Extensión de SDL para el dibujado y la generación de efectos gráficos. Se usó la versión 2.0.13.
- **FTGL** - Biblioteca para renderizar texto con OpenGL usando FreeType. Se usó la versión 2.1.3.
- **FreeType 2** - Engine de fuentes de texto. Se usó la versión 2.3.7.
- **TinyXml** - Parser pequeño y simple de XML para C++. Se usó la versión 2.6.2.
- **Festival** - Sistema general multilingüe para la síntesis de voz. Se usó la versión 1.96.
- **smpeg** - Biblioteca de reproducción de MPEG para SDL. Se usó la versión 0.4.5.

- **libbehavior** - Biblioteca en C++ para implementar Inteligencia Artificial Reactiva. Se usó la versión 1.0.

Capítulo 5

Arquitectura

La descripción de la arquitectura usada en la plataforma **YAOMEV**, junto con la de su juego demostrador **Bit Them All!!!**, seguirá una aproximación TOP - DOWN. Comenzaremos por enumerar los diferentes sistemas en que se componen, junto con sus relaciones en el nivel más alto de abstracción, para después analizar cada uno de ellos, junto a sus subsistemas, en profundidad en las distintas secciones de este capítulo.

En la figura 5.1 se encuentran esquematizados los sistemas componentes de **YAOMEV**, junto con los correspondientes a **Bit Them All!!!**.

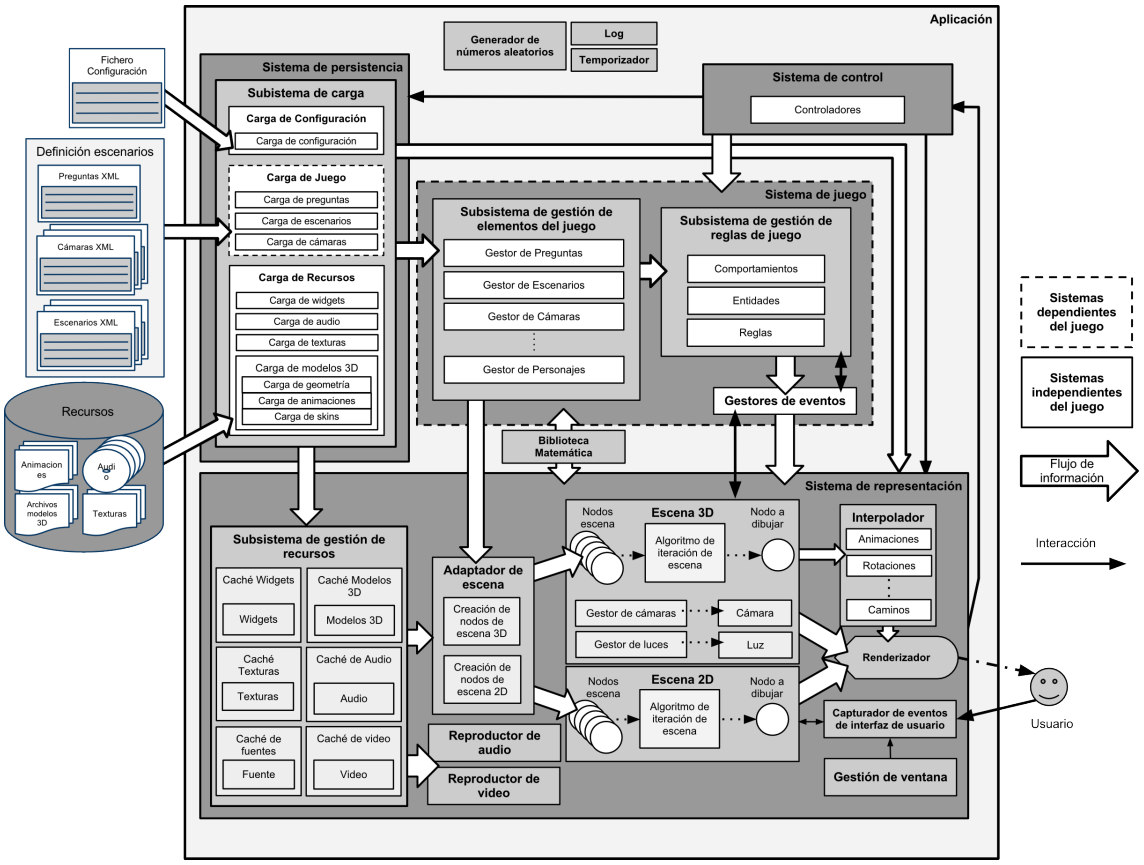

Figura 5.1: Esquema de los módulos de **YAOMEV**

La plataforma **YAOMEV** suministra numerosas herramientas necesarias para crear videojuegos de una forma sencilla: un esqueleto para crear los juegos, una biblioteca matemática, temporizadores, un generador de números aleatorios, un sistema de log y unos completos sistemas de representación gráfica y de persistencia.

A continuación se describen brevemente estos sistemas:

- **Aplicación (sección 5.2)** Se responsabiliza de la gestión de la aplicación en el nivel más alto de abstracción, actúa como contenedor de todos los sistemas del juego, se encarga de inicializarlos y finalizarlos y además proporciona acceso a ellos desde cualquier lugar de la aplicación.

- **Sistema de control (sección 5.3):** Este sistema es el siguiente nivel de abstracción en la aplicación. Decide el comportamiento que tendrá la aplicación según la forma en que haga uso del resto de sistemas y transmite la información capturada por el interfaz de usuario al sistema de juego.

- **Sistema de persistencia (sección 5.4):** Este sistema agrupa los componentes de carga responsables de leer los archivos de configuración y los recursos necesarios para el juego. Además proporciona acceso a los datos obtenidos a los subsistemas de gestión de recursos y de elementos de juego.

- **Sistema de representación ((sección 5.5)):** Este sistema es el más complejo de todos y su principal cometido es crear las escenas gráficas a partir de la información suministrada por el sistema de juego y el subsistema de gestión de recursos. Para la representación organiza la información de forma eficiente, separando los elementos en dos tipos: gráficos 3D y gráficos 2D. Tratándose cada uno de manera diferente. También tiene como responsabilidad capturar los eventos provocados por el usuario y comunicarlos al sistema de control. El susbsistema de gestión de recursos pone a disposición del sistema de representación toda la información visual que necesita para la representación de los entornos 3D e interfaces gráficas, de tal manera que se evite la redundancia de información en memoria. Debido a estos cometidos hace un uso intensivo de la biblioteca matemática.

- **Sistema de juego (sección 5.6):** Este sistema no pertenece a **YAOMEV** en si, sino que forma parte de **Bit Them All!!!**. En se encuentran implementadas las normas del juego, para lo cual mantiene un conjunto de subsistemas que gestionan desde los entidades participantes del juego (como son los elementos del escenario de juego, jugadores, público, presentador, etc...) a los comportamientos de los mismos (la forma en que actúan las entidades en función de lo que ocurre en el juego). Este subsistema ofrece de esta información a los controladores y se comunica con el sistema de representación mediante eventos para transmitirle los cambios de estado ocurridos en el transcurso de una partida.

YAOMEV ha sido construido para no tener que preocuparse por aspectos de bajo nivel como el uso y gestión de audio, vídeo, texturas o modelos 3D entre otros. Para lograr esto se ha adoptado un enfoque fuertemente modular que permite independizar el sistema de representación del sistema encargado de la capa de modelo o lógica interna del juego (sus reglas). Para conseguir esto se ha usado el patrón de arquitectura del software conocido como **Modelo-Vista-Controlador** (**MVC**).

Este patrón fue descrito originalmente por Trygve Reenskaug [12], siendo explicada su utilidad en sus propias palabras: *"MVC es útil si el usuario necesita ver el mismo elemento del dominio desde diferentes contextos o representaciones."*[12]. Es decir, con **MVC**, es posible utilizar diferentes vistas para la representación de los modelos de la capa de dominio sin necesidad de modificar sustancialmente la aplicación.

En general, cuando se aplica **MVC**, el objetivo es: aislar la capa de lógica de la aplicación (conocidas también como capa de dominio o capa de modelo), en nuestro caso las reglas del juego; de la interfaz de usuario, en nuestro caso la representación 3D/2D de los elementos del juego. Permitiendo con ello un desarrollo, testing y mantenimiento independientes.

Como el propio nombre indica, la arquitectura se separa en tres partes diferentes:

- **Modelo:** Gestiona el comportamiento y los datos en el dominio de la aplicación, recibe información del controlador sobre las acciones del usuario y notifica los cambios ocurridos en su ámbito a la vista.

- **Vista:** Representación de la información de la capa de modelo. Debe proporcionar una

forma de interacción entre el usuario y lo mostrado en pantalla.

- **Controlador:** Dictamina el comportamiento de la aplicación ante la interacción del usuario, recibe la entrada del usuario desde la capa de vista y la traduce en información para la capa de modelo. En caso de contar con múltiples vistas también se encarga de gestionarlas.

En el siguiente capítulo se ofrece una descripción detallada de la implementación de esta arquitectura hecha en **YAOMEV**.

5.1. Arquitectura Modelo Vista Controlador en YAOMEV

El desarrollo de la arquitectura **YAOMEV** se ha ajustado a las normas impuestas por el patrón de arquitectura de software **Modelo-Vista-Controlador** (**MVC**).

Esta decisión se ha tomado partiendo de la idea de que si se quiere dar soporte para la creación de otros juegos, es necesario aislar completamente las capas del modelo (lógica del juego) (sección 5.6), de la vista (sistema encargado de presentar la información del modelo al usuario) (sección 5.5). De esta forma para la creación de nuevos juegos sólo es necesario implementar la lógica del juego, junto con los controladores necesarios (sección 5.3). Además **MVC** facilita, en caso de que sea necesario, el posterior desarrollo de cada una de las capas de **YAOMEV** por separado.

En la figura 5.2 se presenta una vista general de las capas MVC en **YAOMEV**, junto con sus relaciones, en el mayor nivel de abstracción. Las responsabilidades de cada capa en **YAOMEV**ño se diferencian del resumen global del **MVC** hecho en el apartado anterior 5.

La forma específica en que se comportan y se relacionan estos módulos dentro de **YAOMEV** es descrita ahora de forma general mediante el seguimiento del flujo de control, no se entrará en detalle sobre las clases nombradas, para información detallada sobre ellas sería conveniente dirigirse a las referencias incluidas en el texto:

1. El usuario interactúa con la interfaz de usuario (pulsando un botón, una tecla, etc...). La clase de la vista `gui::CInputManager` procesa la entrada y genera el evento correspondiente. Existen tres tipos de eventos que gestiona la vista:

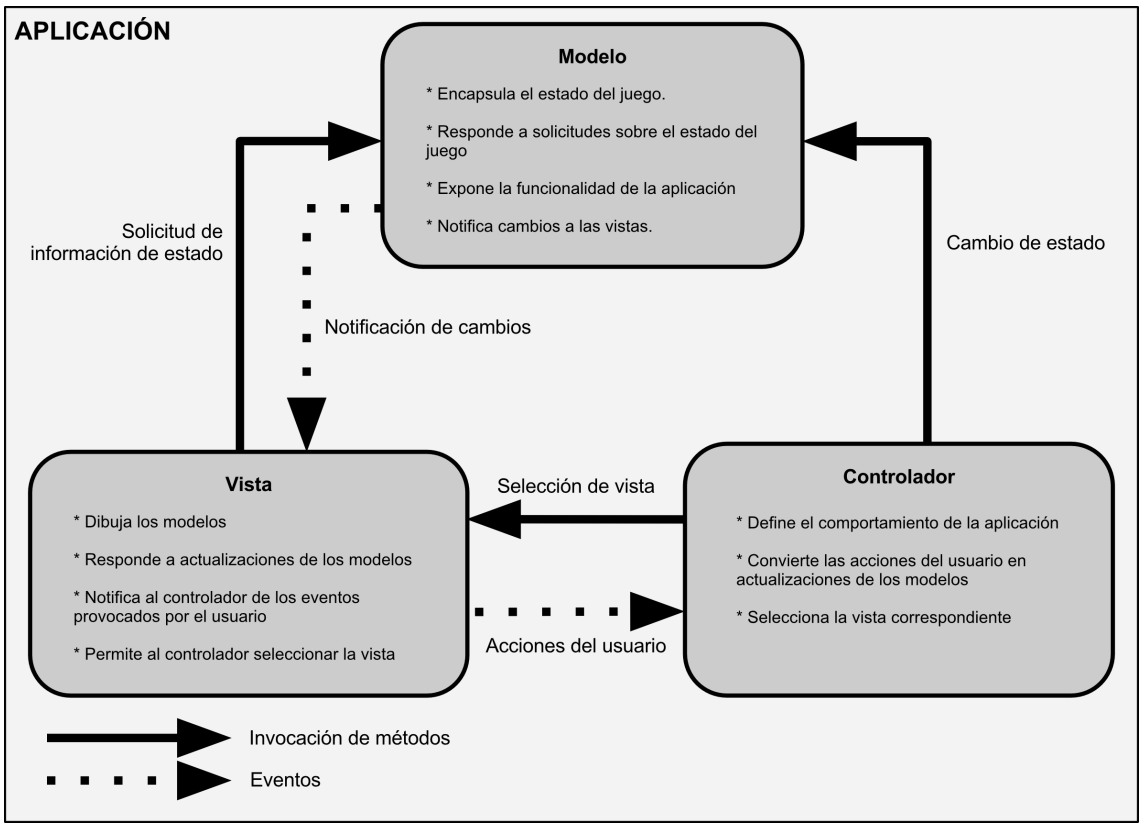

Figura 5.2: Implementación del patrón MVC en YAOMEV

- `event::WindowListener`. Acciones ocurridas en la ventana de la aplicación: redimensión, cierre, etc...
- `event::MouseInputListener`. Acciones realizadas con el ratón: click, movimiento, coordenadas, etc...
- `event::KeyListener`. Pulsaciones de teclas: tecla presionada, tecla soltada, etc...

2. El controlador recibe cualquiera de los eventos anteriores por parte de la vista y accede al modelo actualizándolo a través de `logic::ILogicFacade`.

3. La vista tiene la responsabilidad de desplegar la interfaz de usuario y el mundo del juego. Entendemos por mundo de juego a la representación gráfica (o lógica) de los elementos del juego correspondientes al entorno en que transcurrirá la acción. Para ofrecer las representaciones gráficas, obtiene los datos necesarios del modelo y con

ellos genera la vista correspondiente donde se reflejarán los cambios ocurridos.

Como en la generación de los elementos visuales se desea mantener la independencia entre la vista y el modelo. Mediante la implementación de patrón *Observer* se consigue la separación entre los dos. Se puede encontrar una definición genérica del patrón en (subsección 6.8) En la figura 5.3 se muestra el diagrama de clases[1] de la implementación de este patrón cuando se trata de notificar a la vista cambios en los atributos de posición, rotación o visibilidad de una entidad del juego. Su definición en profundidad se puede leer en el apartado correspondiente al juego (sección 5.6).

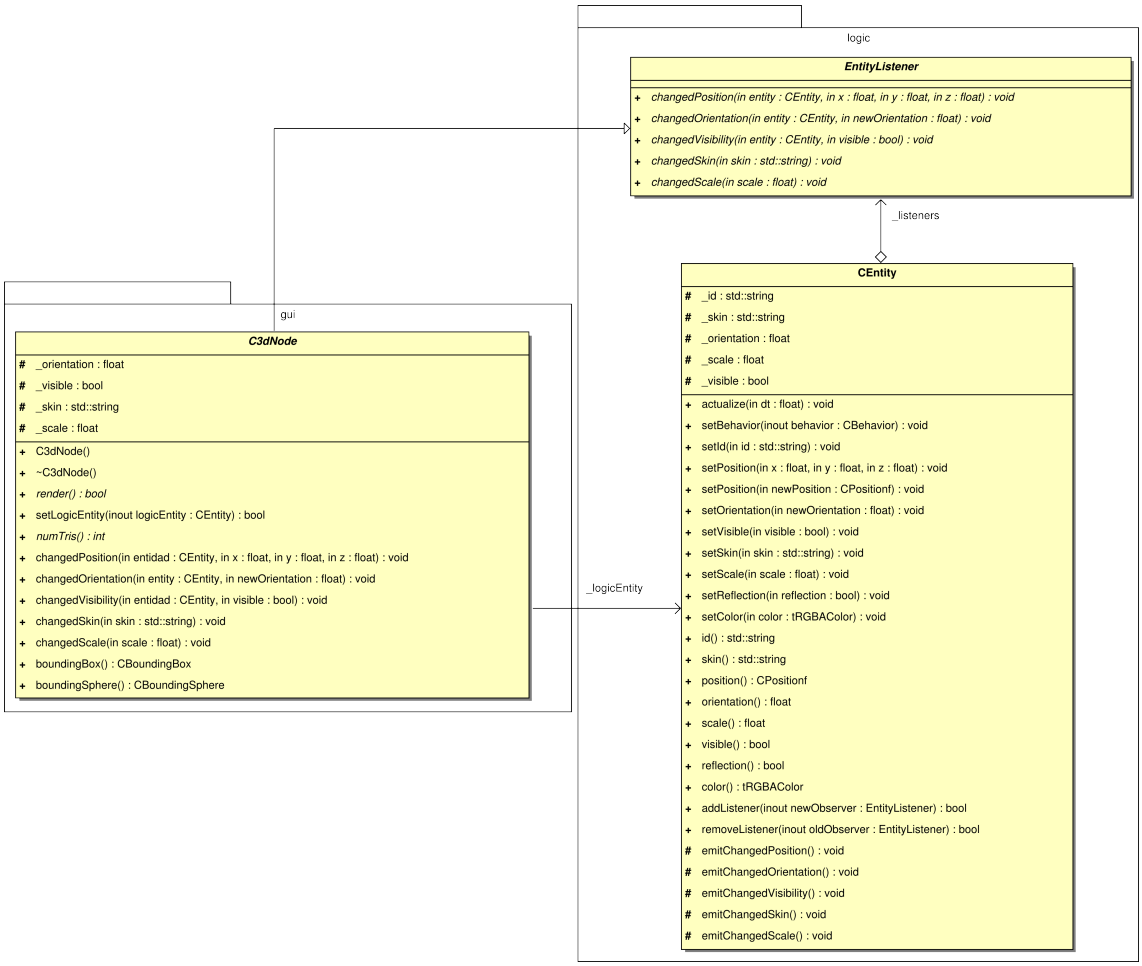

Figura 5.3: Ejemplo de implementación del patrón Observer para la representación de entidades

El controlador es el encargado de usar el gestor de escena, clase `gui::CScene-`

[1]Sólo se muestran los métodos y atributos relativos al traspaso de información entre la vista y el modelo.

`Manager`, componente que se encarga de crear, almacenar y representar los componentes gráficos del juego. Si hubiera más de uno se encargaría de seleccionar el idoneo.

La creación de las vistas usa de nuevo el patrón *Observer* (descripción en (subsección 6.8)). Cada vez que se usa un modelo y se requiere su vista, `logic::ILogicFacade` emite el evento correspondiente a todos los `Listeners` que tiene asociados. Estos `Listeners` son clases que implementan la interfaz del "escuchador" adecuado, en concreto para la creación de las vistas se usa los "escuchadores" que implementan la interfaz `logic::CreationDestroyListener`. El único requisito para añadir una nueva vista a la aplicación es crear un gestor de escena que implemente este `Listener`.

4. Como hemos apuntado antes, el modelo no tiene conocimiento directo de la vista, pero para transmitir la sensación de que se interactúa con el juego es necesario que la vista se actualice tan pronto se modifique el estado de algún aspecto del juego. Para la transmisión de información y con objetivo de proveer de nuevo cierta indirección entre las dos capas se implementa también el patrón *Observer* (descripción en en (subsección 6.8)), figura 5.3.

De esta forma, la actualización de las vistas sucede solo cuando ha ocurrido algún cambio en el estado de las entidades del juego mediante la emisión de los eventos oportunos. En la figura 5.4 se muestra un ejemplo de acción del juego mediante su diagrama de secuencia. En el diagrama se puede apreciar el intercambio de información entre el modelo, la vista y el controlador del juego **Bit Them All!!!**cuando se responde a una pregunta y el personaje jugador acierta, lanzando la animación correspondiente del personaje jugador.

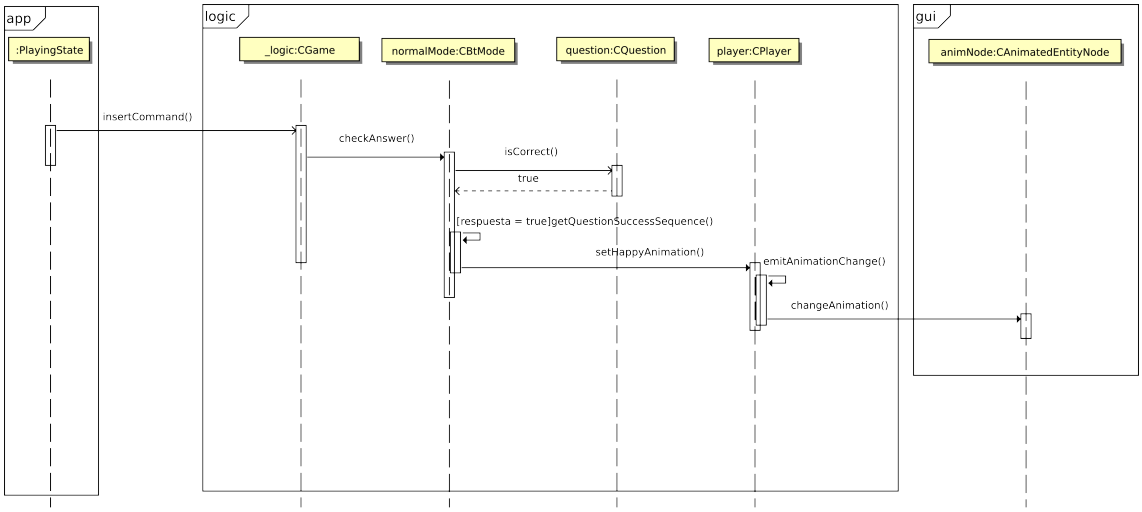

Figura 5.4: Diagrama de secuencia de la transmisión de un evento desde el usuario hasta la vista

5.2. Módulo de aplicación

Este módulo se encuentra en el nivel más alto de abstracción de la aplicación. Se encarga de la inicialización y terminación de todos los componentes que forman parte de **YAOMEV** y de los que participan en el juego, además de proveer acceso a ellos. También estructura la ejecución de los componentes involucrados en la representación y desarrollo del juego y por encima de todo esto gestiona el comportamiento que ofrecerá al usuario la aplicación.

Listing 5.1: Clase `CApplication` usada en el main.

```
int main(){
        CApplication* myGame = new CBitThemAllApp();
        if(!myGame->init()) return 0;
        myGame->run();
        myGame->end();
        delete myGame;
        return 0;
}
```

En vista de que se pretende construir una plataforma para el desarrollo de juegos, es necesario ofrecer un punto de entrada a la aplicación que ofrezca libertad suficiente para adaptarlo sin problemas a cualquier tipo de proyecto. Además debe ofrecer acceso a todos los componentes necesarios para la programación de un juego; encargarse de su arranque y parada; y ofrecer una solución a la discretización de la ejecución de los diferentes sistemas

que componen el juego, sin necesidad de que el programador tenga que tener en cuenta esos pasos.

En **YAOMEV** se ha creado una pequeña jerarquía de clases que ofrecen distintos grados de solución a los problemas planteados antes (figura 5.5). Estas clases actúan como contenedor de los diversos sistemas que ofrece **YAOMEV** y de los controladores [2] (secciones 5.1 y 5.3) que vayan a usarse, también se encargan de ejecutar el controlador actual, inicializar todos los sistemas, proporcionar acceso a ellos al resto de módulos y se responsabiliza de finalizarlos cuando se termine la ejecución.

Para evitar problemas en la gestión de la aplicación, la clase base `app::CApplication` (figura 5.5), se ha creado haciendo uso del patrón *Singleton* (sección 6.3), puesto que en todo momento sólo es necesaria la existencia de una instancia de la aplicación, y de esta forma se asegura que todos los elementos esten bien agrupados en un sólo lugar.

La gestión del tiempo (sección 5.2.1) y la inicialización/destrucción de los módulos, independientes del juego a programar, se realizan en la especialización `app::CYaoApp` (figura 5.5). Esta especialización es la interfaz que se debe implementar para construir un juego con **YAOMEV**.

Aparte de la gestión del tiempo `app::CYaoApp` contiene las instancias a los diferentes módulos de la aplicación. En concreto:

- **Gestor de ventana.** Se encarga de la gestión de la ventana en que se dibuja el juego.
- **Servidor gráfico.** Centraliza el acceso a los elementos gráficos que usará el gestor de escena.
- **Servidor de video.** Encargado de almacenar y suministrar los vídeos del juego.
- **Servidor de audio.** Encargado de almacenar y suministrar los componentes de audio del juego.
- **Servidor de fuentes.** Encargado de almacenar y suministrar las fuentes que se usarán en la representación de texto.

[2] A lo largo de esta sección se usarán el término estado y controlador indistintamente, pues cada controlador es un estado de una máquina de estados (sección 5.3).

- **Servidor de DAOs.** Encargado de suministrar acceso a los objetos de acceso a datos. 5.4
- **Log.** Encargado de crear logs de texto para almacenar las trazas del juego.
- **Temporizador.** Encargado de llevar la cuenta del tiempo de la aplicación.
- **Manejador de controladores (estados de la aplicación).** Almacena y gestiona los controladores del juego.

En la especialización desarrollada para el demostrador **Bit Them All!!!** `app::CBit-ThemAllApp` se incluyen los siguientes atributos propios del juego (sección 5.6):

- **Fachada lógica.** Intermediario entre la capa de lógica (donde se gestionan las reglas del juego y sus entidades participantes) y el resto de capas.
- **Gestor de escenarios disponibles.** Encargado de almacenar y suministrar acceso a los escenarios que se pueden usar en el juego.
- **Gestor de personajes disponibles.** Encargado de almacenar y suministrar acceso a los personajes que aparecen en el juego.
- **Mapa de animaciones.** Una tabla de las animaciones disponibles para cada personajes del juego.

Una vez han quedado descritos los diferentes componentes de la jerarquía de clases de aplicación y los módulos de los que se encargan, queda hacer un repaso por los diferentes pasos en los que se descompone el ciclo de vida de una aplicación escrita con **YAOMEV**.

Este módulo es el punto de entrada a la aplicación por lo que debe de evitarse que su uso sea complejo o innecesariamente complicado en su comprensión. En el código del `main` de la aplicación (ver listado de código 5.1) se pueden apreciar cláramente las tres etapas del juego (inicio, ejecución y parada).

A continuación se describen en detalle estas tres etapas:

- **Inicialización: `void app::CYaoApp::init().`** Todos los módulos de la aplicación realizan aquí su inicialización. Es importante que todos aquellos módulos que implementan el patrón *Singleton* tengan cláramente especificado su lugar de arranque (en caso de que lo necesite) y su lugar de destrucción. Para una descripción genérica del patrón *Singleton* (sección 6.3).

  Y es que uno de los problemas que ofrecen los *Singleton* es el acceso global que ofrecen a su instancia. Esto, sin una adecuada planificación en su uso, puede provocar situaciones no deseadas a causa de que queden dispersas por el código llamadas no controladas a su instancia.

  Por ejemplo: con una mala planificación de los mecanismos y momentos en que se realiza la liberación de memoria, si tras la liberación de la memoria de un *Singleton* queda todavía alguna función importante por llamar y resulta que en ese código se solicita su instancia, esta se volverá a crear. Por lo que al final esa memoria no se liberará. Esto también puede ocurrir en el proceso de reserva de memoria (o construcción) si el *Singleton* usa algún tipo de *lazy inicialization* (mecanismo por el cual se pospone la inicialización de los atributos de un objeto) dependiente de algún atributo. Si se empieza a usar el *Singleton* antes de su inicialización provocará con toda seguridad algún error de ejecución.

  En la inicialización de la aplicación debe ser también donde se creen los controladores (descripción en 5) que se quieran usar en el juego. En nuestro caso concreto para el juego **Bit Them All!!!** se usan tres controladores diferentes (cada uno se corresponde con un estado diferente del juego):

  - Load. Este estado ejecuta las rutinas de lectura de los archivos de configuración y de carga de recursos.
  - Menu. En este estado se despliegan los menús del juego y dan acceso a las opciones implementadas en ellos.
  - Game. Ejecuta el juego completo.

Se puede encontrar una descripción detallada de los controladores del juego en el capítulo dedicado a los controladores (sección 5.3).

- **Ejecución: `void app::CApplication::run()`.** El método `run` sólo esta implementado en `app::CApplication` debido a que no queremos que se modifique en la implementación específica de un juego. Este método simplemente ejecuta el bucle principal del juego mientras no se solicite su terminación.

  El aspecto del bucle principal en **YAOMEV** se puede ver en (ver listado de código 5.3). Se pueden apreciar cláramente las acciones que se realizan en el. Primero se inicia el temporizador (comienza a contar el tiempo a partir de ese instante), acto seguido el gestor de los controladores `_stateHandler` comprueba si ha ocurrido una transición que provoque un cambio de estado y si la hay se realizará, después se ejecuta el estado que `_stateHandler` mantenga como el actual pasandole como parámetro la delta de tiempo calculada en la iteración anterior y por último se almacena la delta de tiempo de esta iteración para ser usada en la siguiente vuelta del bucle.

  Se ha perseguido que la implementación sea lo suficientemente genérica como para que pueda ser usada en cualquier juego.

- **Finalización: `void app::CYaoApp::finish()`.** Este método simplemente se encarga de asegurarse de eliminar todos elementos inicializados anteriormente.

Como queda patente en la figura 5.5 en la construcción de la aplicación se ha creado una pequeña jerarquía de clases de aplicación.

- `app::CApplication` define la interfaz a seguir por una aplicación genérica que necesite un bucle principal.

- `app::CYaoApp` especializa la clase `app::CApplication` añadiendo los módulos propios de la plataforma **YAOMEV** para uso de gráficos 3D, widgets, audio, video, los temporizadores, el log, etc... Esta clase sirve de interfaz para crear un juego con **YAOMEV**.

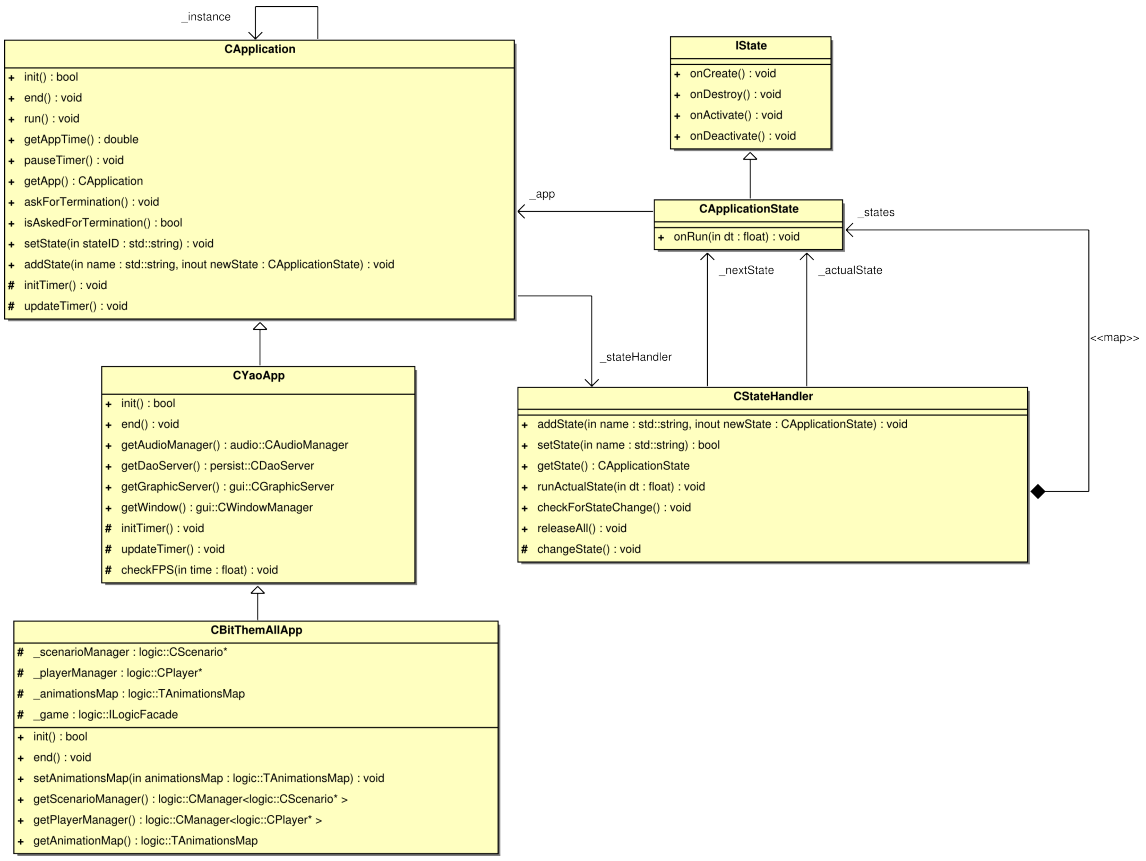

Figura 5.5: Diagrama de clases del módulo aplicación

- `app::CBitThemAllApp` es la clase concreta que incluye los atributos correspondientes a la lógica (capa de modelo) del juego demostrador **Bit Them All!!!**. Esta clase es específica del juego y no forma parte de **YAOMEV**.

El motivo por el que se han hecho dos tipos de clases de aplicación ha sido ofrecer al desarrollador dos clases genéricas que permitan la creación de nuevos juegos pero dándole libertad para elegir qué partes implementar.

Para conseguir esta libertad en el desarrollo, `app::CApplication` ha sido diseñada como una máquina de estados finitos debido a la necesidad de variar su comportamiento en función del estado del juego. En **Bit Them All!!!** se han distinguido tres estados: Carga, Menu y Juego que serán explicados en las subsecciones 5.3.2, 5.3.3 y 5.3.4.

Para el diseño de la FSM se ha usado el patrón *State* (sección 6.9). En el caso concreto de `app::CApplication`, la clase que representa sus diferentes estados es `app::-`

`CApplicationState`. Para evitar la redundancia de código se creó la clase `app::CStateHandler` que se usa para añadir o quitar estados, y para ejecutar el estado que se le indique. Esta clase es usada tanto en`app::CApplication` como en `app::CYaoCompositeController`.

Para la implementación de este módulo se ha hecho uso de los patrones *Strategy* (sección 6.10) y *Template Method* (sección 6.11). Todos los controladores de la aplicación deben implementar el interfaz `app::CApplicationState` y cada controlador modificará el comportamiento de la aplicación en función de la forma en que haya sido programado.

La implementación concreta de *Template Method* se ha realizado creando el esqueleto del método `CApplication::run()` en la clase base `app::CApplication` y dejar sin implementar algunos métodos, con lo que queda en mano de sus especializaciones la definición de los mismos. En concreto se encuentran sin especificar `CApplication::initTimer()` y `CApplication::updateTimer()`. El esqueleto del algoritmo, correspondiente al bucle principal del juego, se puede ver en el listado 5.3.

Uno de los objetivos en la gestión de los controladores es que se comporten como máquinas de estados jerárquicas (HFSM) tal y como se indica en la (sección 5.3), por ello, para el diseño de los mismos se ha usado el patrón de diseño *State* (sección 6.9) junto con el patrón estructural *Composite* (sección 6.4).

Dado que la clase `app::CApplication` ofrece una interfaz genérica, esta puede ser usada en el desarrollo de cualquier tipo de aplicación que necesite de la discretización de los pasos de ejecución del programa. Se ha diseñado con el objetivo de ofrecer una gestión uniforme para cualquier tipo de videojuego, que basan su ejecución en pasos discretos establecidos en lo que se conoce como bucle principal (ver listado de código 5.3), con independencia del género o tipo de juego a desarrollar. Además, gracias a su diseño como máquina de estados finitos (FSM) (capítulo 3), define su comportamiento en tiempo de ejecución en función de los controladores creados y de las transiciones establecidas (sección 5.3).

5.2.1. Gestión del tiempo en el bucle principal

En **YAOMEV**la ejecución del controlador actual (sección 5.3) es el nivel más alto en que se ejecuta lo que se conoce en el mundo de los videojuegos como bucle principal. El bucle principal es el corazón de un juego. Básicamente es un bucle que se ejecuta continuamente mientras el juego está en marcha, es responsable de coordinar las acciones del juego, en el se especifican las tareas que se deben realizar en cada vuelta y en qué orden, para luego ejecutarse en pasos discretos. Simplificando mucho, y a modo de ejemplo, estas tareas suelen ser ser tres: captura de la entrada de usuario, ejecución de la lógica y dibujado de los gráficos.

En este nivel todavía no nos preocupamos de qué acciones se realizan en el bucle principal, eso es responsabilidad de los controladores, sin embargo si nos encargamos de la gestión del tiempo entre dos vueltas del bucle. Esta gestión persigue que el juego actúe de forma uniforme sin tener en cuenta en qué máquina se está ejecutando.

Listing 5.2: Bucle principal en clase `CApplication`.

```
1  void CApplication::run(){
2          // Se ha solicitado en la vuelta anterior la terminacion?
3          while (!isAskedForTermination()) {
4                  // Se inicia el temporizador
5                  initTimer();
6
7                  // Comprobamos si hay cambio de estado. Si lo hay se realizara
8                  _stateHandler->checkForStateChange();
9
10                 // Ejecutamos el estado que se encuentre en marcha en este momento
11                 _stateHandler->runCurrentState(_dt);
12
13                 // Guardamos la delta de tiempo para usarla en la siguiente iteracion
14                 updateTimer();
15         }
16 }
```

Existen tres formas básicas de controlar el tiempo durante la ejecución del bucle:

- **Sin control de tiempo:** Implica que el bucle se ejecuta siempre lo más rápido posible y con ello todas las acciones habrán tenido que fijarse a unos valores concretos de velocidad que con toda seguridad sólo valdrán para la máquina en la que se ha desarrollado. Si se usara en máquinas más lentas o más rápidas se notaría una ralentización o aceleración de las acciones del juego.

- **Bucle controlado por intervalo fijo de tiempo (Fixed time-step loop):**. Este modo permite controlar a qué velocidad se ejecuta el bucle principal obligándola a permanecer fija.

  Dos velocidades típicas serían 30 fps o 60 fps[3]. Habitualmente la forma de hacer esto es controlando cuanto tiempo debe tardar como mucho el bucle en ejecutarse. En el caso de 60 fps la ejecución de una vuelta del bucle debe tardar como mucho 1/60 segundos o 16 milésimas de segundo aproximadamente. Este valor es lo que se conoce como *delta de tiempo*.

  Con esta solución el juego se convierte en un sistema determinista, logrando a su vez que mantenga una velocidad constante (sin picos de velocidad). Esto ofrece ventajas como su sencillez en la implementación, permite la repetición de los sucesos ocurridos en el juego y con ello se facilita la depuración de los problemas que surjan durante la ejecución.

  Como inconvenientes tiene el problema de que una delta demasiado pequeña puede afectar al rendimiento del juego si una o varias iteraciones del bucle tardan en su ejecución más tiempo del fijado.

- **Bucle controlado por intervalo variable de tiempo (Variable time-step loop):** Esta manera de abordar el problema busca independizar la ejecución del bucle del tiempo que pueda tardar en terminar una iteración, sin afectar con ello al rendimiento del juego.

  Para ello se accede al final de cada iteración del bucle a la delta de tiempo (tiempo transcurrido en la ejecución de una iteración), después ese valor de delta se difunde en la siguiente iteración a los elementos lógicos y gráficos (entre otros) del juego. Este valor es usado por estos elementos para interpolar acciones como pueden ser: el movimiento de un personaje, la rotación de un modelo, acumulación en contadores de tiempo, etc...

  Una forma habitual de fijar el movimiento es usando una ecuación de interpolación lineal como puede ser:

[3] Velocidades en frames por segundo o vueltas del bucle por segundo.

$$Velocidad_{actual} = Aceleracion_{actual} * delta_{tiempo} \quad (5.1)$$

$$Posicion_{actual} = Posicion_{actual} + (velocidad_{actual} * delta_{tiempo}) \quad (5.2)$$

Usando ecuaciones de este tipo, con esta aproximación al cálculo de la delta, se logra la sensación de movimiento fluido independientemente del ordenador en que se ejecute el juego. Además si en algún momento la capacidad de proceso se ve excedida por el juego, los acontecimientos seguirán transcurriendo en el orden correcto.

En **YAOMEV** se usa la tercera aproximación, para ello se suministra una clase temporizador que se encarga de calcular la delta de tiempo entre dos llamadas consecutivas del método `void app::CApplication::run()`, para después propagarla al resto de módulos.

5.2.2. Log

Es habitual en el desarrollo de videojuegos tener varias configuraciones de compilación para los proyectos. Una de ellas suele ser la versión `DEBUG` o versión de desarrollo. Otra práctica habitual en estas versiones suele ser crear logs de lo que ocurre en el juego. Por este motivo en **YAOMEV** se ha creado una sencilla clase para la creación de logs. Está implementada como `Singleton` para que se pueda usar en cualquier lado del código y simplemente proporciona: un método para indicar la ruta y el nombre del log, un método para escribir en el log y un método para liberar cualquier tipo de memoria usada.

La única peculiaridad de la clase `utils::CLog` es el proceso de escritura en el log. Como está pensado para ser usado en una versión en desarrollo, no sería raro que su ejecución pudiera ser abortada en cualquier momento sin que se cierre el archivo en el que se está escribiendo. Por ello el método de escritura abre y cierra el archivo de log cada vez que se escribe en el. Esto a priori no es una práctica eficiente, pero se supone que en la versión de compilación `RELEASE` (versión final ya terminada) no se hará uso del log en ningún momento.

5.2.3. Timers

Hay dos tipos de temporizadores accesibles, uno que usa **SDL** y otro la biblioteca **<sys/-time.h>**. Existe un buen motivo por el cual se han construido dos temporizadores en **YAOMEV**, y es que **SDL** ofrece una resolución de tiempo en la escala de los milisegundos, sin embargo esto puede ser considerado hoy en día como una resolución de tiempo demasiado baja, por eso se experimentó con otras bibliotecas para comprobar si se podía acceder a mayores resoluciones. La resolución con la que un temporizador te puede devolver el tiempo transcurrido es muy importante para conseguir simulaciones más fluidas y realistas.

Al final se comprobó que este aspecto es muy dependiente de la máquina y de las implementaciones concretas que se han hecho de las bibliotecas que controlan los temporizadores. Sin embargo con la biblioteca **<sys/time.h>** si se podían alcanzar resoluciones de microsegundos en la máquina de desarrollo.

Habiendo trabajado con dos tipos de temporizadores y sin poder asegurar que la resolución sea la misma en la máquina de desarrollo que en el resto, se decidió implementar un sistema de temporizadores que permitiera añadir nuevos temporizadores si fuera necesario. Para ello se creo una factoría de temporizadores mediante la implementación del patrón *Simple Factory* (sección 6.2). Mediante este patrón se centraliza el acceso a los mismos haciendo transparente al usuario el tipo concreto de temporizador es el usado y permitiendo el cambio entre ellos sin realizar modificaciones más allá de incluir un método de acceso al temporizador en la factoría.

Como se ha dicho antes, se accede a ellos a través de la factoría de temporizadores `time::CTimerFactory` que devuelve objetos `time::Timer`. Cualquier temporizador que se quiera añadir debe implentar esta interfaz, que permite iniciarlo, pararlo y la devolución del tiempo transcurrido con resoluciones de segundos, milisegundos y microsegundos.

El diagrama de clases de la familia de clases encargada de los temporizadores se encuentra en la figura 5.6.

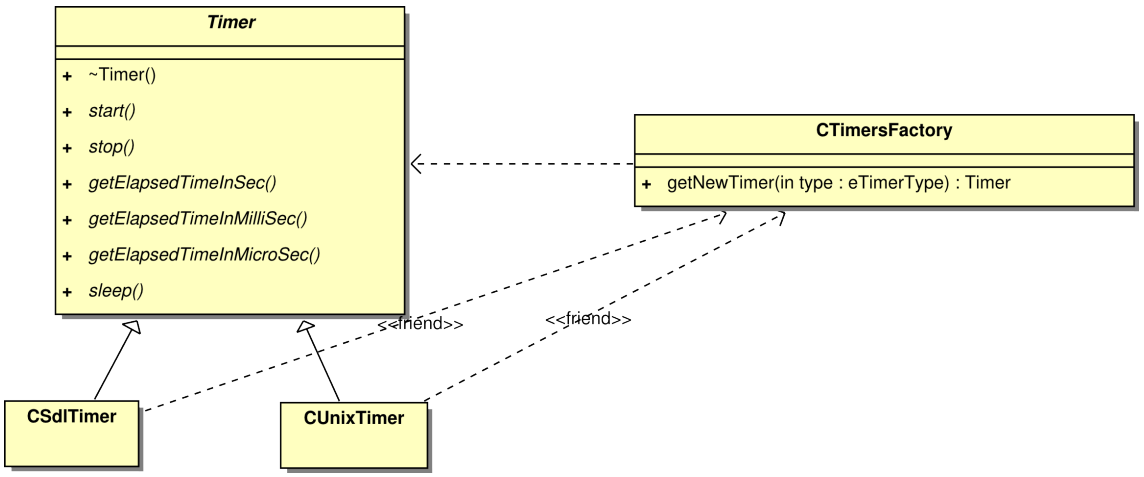

Figura 5.6: Diagrama de clases de la implementación de una factoría simple de temporizadores

5.2.4. Generador de números aleatorios

En una simulación es muy importante la apariencia de aleatoriedad para permitir un desarrollo más realista de la acción. Por eso en **YAOMEV** se ha creado una clase que se encarga de la tarea de generar números pseudoaleatorios (al estar generados mediante una fórmula no son aleatorios, pero lo aparentan) de diferentes formas.

La clase creada es `utils::CRandom` y en ella se han implementado dos tipos de métodos: los que devuelven un número pseudoaleatorio o los que devuelven listas números pseudoaleatorios.

Esta clase se ha implementado como un *Singleton* (sección 6.3) para permitir su acceso desde cualquier lugar de la aplicación.

5.3. Módulo de controladores

Este módulo define el comportamiento de la aplicación en el siguiente nivel de abstracción. El objetivo es que actúe como interfaz entre el usuario y el juego desarrollado, transmitiendo la información suministrada por el jugador a través de la vista (sección 5.5) (mediante la interfaz gráfica de usuario) al modelo (sección 5.6). Además se encarga también de especificar el comportamiento del juego (entendiendo comportamiento como las clases que actúan

en un instante dado en la aplicación, que en función de su implementación variarán la forma en que se comporta el juego). Como su propio nombre indica, en el patrón **MVC** se corresponde a la capa del **Controlador**.

Es muy complicado crear un controlador genérico que gestione cualquier clase de juego, sin embargo la plataforma **YAOMEV** ofrece las herramientas necesarias para la creación de los mismos, de la manera en que se describe a continuación.

Los controladores gestionan la entrada del usuario, la capa de vista y la capa de modelo. Por lo tanto esta clase es el corazón de los juegos, tiene que tener acceso a casi todos los elementos participantes y, por lo tanto, hacer que sean fáciles de usar para el programador. En el caso de **Bit Them All!!!** , todos los controladores implementan la interfaz `app::CYaoController`, que proporciona acceso a las instancias de las clases:

- `gui::CSceneManager`. Clase encargada de la gestión de la escena gráfica, se responsabiliza de crear y dibujar las vistas a partir de los elementos de la lógica del juego.
- `gui::CWindowManager`. Ofrece acceso al sistema de ventanas sobre el que se dibuja el juego.
- `gui::CInputManager`. Procesa la entrada del usuario y lanza los eventos necesarios.
- `gui::CAudioManager`. Gestiona el sistema de audio del juego.
- `app::CBitThemAllApp`. Instancia de la aplicación.

Las especializaciones correspondientes a un juego concreto deberán aportar las clases que gestionen la capa de modelo. De los controladores creados para **Bit Them All!!!** se hablará más adelante en este capítulo.

En vista de que los controladores deben dictar el comportamiento de la aplicación, se ha optado por la solución de tratar cada controlador como un estado dentro de una máquina de estados jerárquica (Hierarchical Finite State Machine o HFSM). De esta forma el modelado de los comportamientos se logran de forma sencilla e intuitiva.

La única diferencia entre una HFSM y una FSM es que en la HFSM cada estado puede contener otro conjunto de estados. Así cada estado es realmente como una máquina de estados finita.

Como ejemplo de lo sencillo que resulta modelar con una HFSM los comportamientos de un juego, a continuación se puede ver el diagrama de la máquina de estados de **Bit Them All!!!** 5.7.

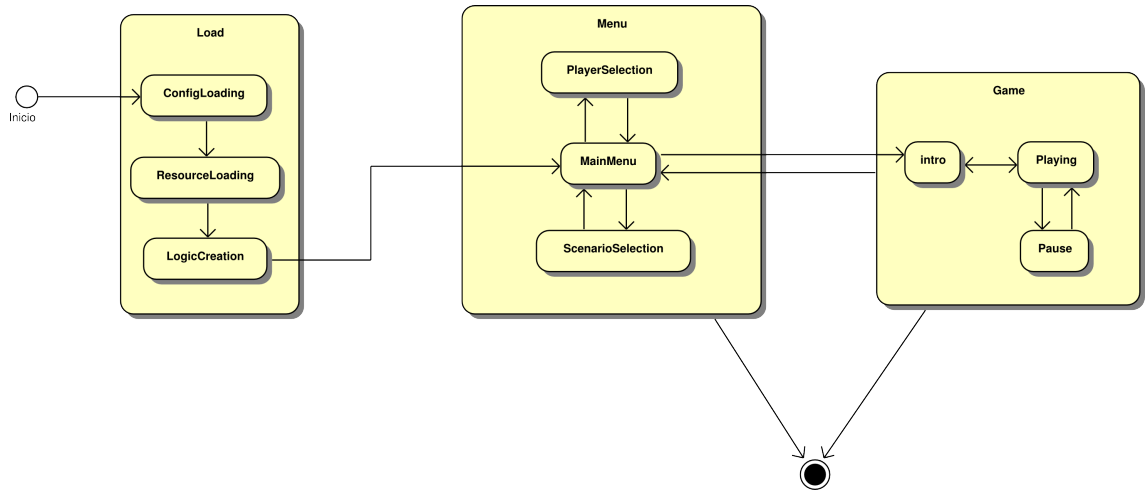

Figura 5.7: Máquina de estados de Bit Them All

Existen tres tipos de controladores en el juego:

- **Carga:** Establece tres pasos en la carga de la información para la aplicación.

  Carga de configuración. Lee el fichero de configuración e inicializa el gestor de fuentes.

  Carga de recursos. Carga los recursos gráficos, sonoros y de vídeo que el juego vaya a necesitar.

  Creación de lógica. Aquí se leen y cargan los ficheros de configuración del juego, en concreto: las preguntas, los jugadores, los escenarios que haya disponibles. Una vez cargados, estos elementos están listos para poder ser usados en el juego.

- **Menú:**

  Menú principal. Muestra el menú principal y permite moverse al resto de menús.

Selección de escenario. Muestra el menú de selección de escenario y permite elegir entre los que haya disponibles.

Selección de personaje. Muestra el menú de selección de personaje y permite elegir entre los que haya disponibles.

- **Juego:**

  Intro. Subestado activo cuando haya alguna situación del juego que no necesita de la intervención del jugador.

  Pausa. Subestado que deja el juego en estado de pausa y permite volver al menú principal o de nuevo al juego.

  Jugando. Subestado activo en los momentos en que el jugador debe participar en el juego.

Para el modelado de la máquina de estados jerárquica (HFSM) se ha optado por el uso del patrón de comportamiento *State* (sección 6.9) junto con el *Composite* (sección 6.4) puesto que reunen las propiedades necesarias que permiten abordar la creación de una HFSM. La decisión de usar estos dos patrones obedece a que resultaba conveniente que no todos los estados fueran compuestos. En la figura 5.8 se muestra el diagráma de clases de la implementación concreta de la capa de controladores de **Bit Them All!!!**, donde se puede apreciar que los subestados de los diferentes estados padres son estados simples.

YAOMEVpermite la creación de nuevos controladores aprovechando la jerarquía creada a tal efecto. Para la implementación de los patrones comentados anteriormente se han usado tres clases.

- Primero se tiene una interfaz genérica que define cómo será cada estado, en la figura sería la clase `CYaoController`. Todo estado simple (que no contenga subestados) o compuesto (que si los contenga) debe implementar esta interfaz.

- Después tenemos la clase `CYaoCompositeController` que implementa esa interfaz y además puede contener subestados. Esta clase es la que se comporta realmente como una HFSM.

- En lugar de tener el código de gestión de los estados que componen una clase en `CYaoCompositeState` se ha optado por usar una clase externa llamada `CState-Handler` que es la que en última instancia mantiene las instancias tanto del estado actual en el que se encuentra la aplicación como del resto de estados de los que se compone. Esto se ha hecho así para poder reutilizar el código creado para la FSM de la clase `app::CApplication` (sección 5.2).

Esta aproximación no implementa en ningún momento la función de transición, eso se deja a cargo del cliente (sistema o subsistema de la aplicación) que use este componente. En los estados creados específicamente para **Bit Them All!!!**son los eventos que capturan las especializaciones de estas clases los que desencadenan las transiciones.

La decisión de construir los controladores como FSM se ha debido a que son fáciles de diseñar, de implementar, de depurar y además bastante eficientes [14]. Y en particular se han usado HFSM por tres motivos:

- Modularidad: Permiten agrupar comportamientos facilitando su reutilización.
- Abstracción: Al ofrecer un alto nivel de abstracción permiten la comprensión del comportamiento general de un estado sin perderse en los detalles.
- Reusabilidad: Las características anteriores facilitan la tarea de ampliar un estado concreto o añadir nuevos estados a los ya existentes.

Como se puede apreciar en la figura 5.8 todos los estados del juego implementan la interfaz proporcianada por `CYaoController` o la de `CYaoCompositeController`.

Cuando se planifica la creación de una máquina de estados para gestionar los comportamientos de una aplicación, el objetivo es poder realizar las transiciones entre estados en tiempo de ejecución. Por lo tanto se plantea un problema a la hora de establecer cómo se realizará esa transición y cómo se tratarán los atributos de los estados al comenzar a ejecutarse o al dejar de hacerlo. ¿Debemos destruir el estado al dejar de ejecutarse? Y si no se destruye los atributos permanecerán tal y como quedaran la última vez que se ejecutó ¿Cómo reutilizarlo entonces? Además surge un problema adicional: la gestión de memoria. Si el estado a implementar necesita hacer uso de *news* y *deletes* para sus atributos y estos se inicializan en

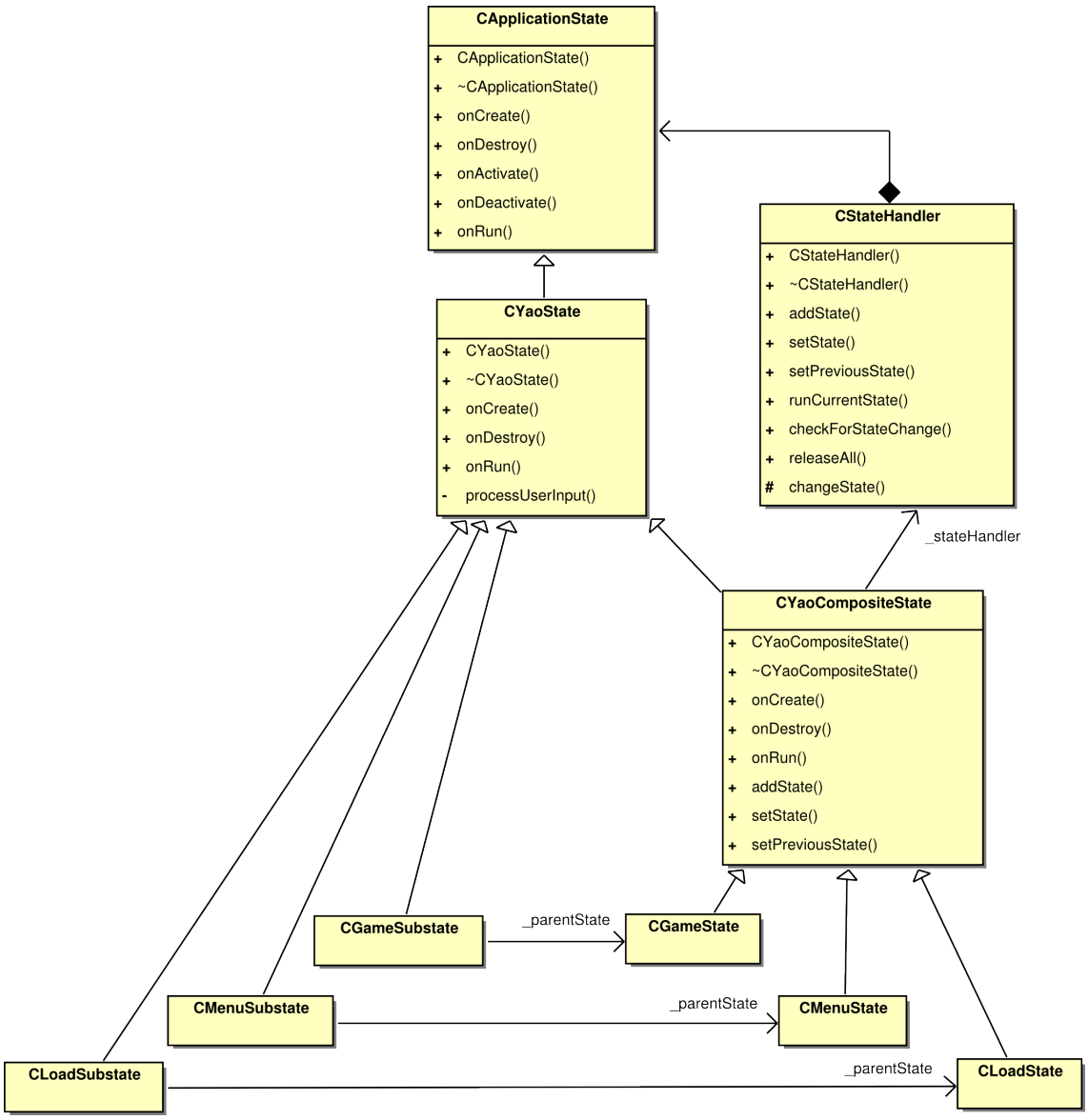

Figura 5.8: Diagrama de clases de los controladores de **Bit Them All!!!**

el constructor y se destruyen en el destructor, entonces es posible que nos encontremos con una sobrecarga en tiempo ejecución a causa de la gestión de memoria.

Al final la decisión de separar la creación/destrucción del estado de su activación/desactivación atiende sobretodo a la necesidad de gestionar la memoria de forma eficiente. Con la aproximación tomada de distinguir entre creación/destrucción de activación/desactivación, al crear los estados invocamos su método de creación `onCreate()` para reservar la memoria que se necesite una sola vez. La inicialización de esos atributos corre a cargo de

`onActivate()` y el reseteo de los mismos a cargo `onDeactivate()`, de este modo se permite el cambio entre estados sin incurrir en la penalización por la gestión de memoria. Por último, cuando el estado ya no es necesario simplemente se usa `onDestroy()` para liberar la memoria.

La gestión de los estados recae sobre la clase `CStateHandler` de la cual `CYaoCompositeController` tiene una instancia como atributo. `CStateHandler` implementa los métodos necesarios para establecer el estado actual, ejecutarlo mediante la invocación de `void app::CYaoController::onRun(float dt)` y por último de eliminarlos todos. `CYaoCompositeController` envuelve a esta clase y es la encargada de usar sus métodos.

5.3.1. Controlador genérico CYaoController

Como ya se ha dicho antes, esta clase contiene el comportamiento genérico que todo controlador de una aplicación construida con **YAOMEV** debe usar. En concreto tiene definidas las siguientes funciones como *virtuales* con el objetivo de que sean redefinidas como sea necesario por sus especializaciones:

- **onCreate:** Crea una instancia de `CInputManager` y se añade a si mismo como escuchador de los eventos de teclado y de ventana, después solicita una instancia del gestor de ventana y del gestor de audio para ofrecer acceso a ellos, por último si el flag `gfxActive` esta a `true` entonces se crea una instancia del gestor de escena y se le solicita a este la instancia del gestor de cámaras que le corresponde. Si el flag no estuviera activo entonces no se hace nada.
- **onDestroy:** Deshace lo hecho en `onCreate()` y si el flag `gfxActive` esta a `true` se elimina el gestor de escena, sino no hace nada.
- **onActivate:** Se deja a cada controlador específico su implementación.
- **onDeactivate:** Se deja a cada controlador específico su implementación.
- **onRun:** Este método decide las etapas del bucle principal del juego. Se ha establecido de forma genérica en tres pasos a los cuales se suministra la delta de tiempo que

`CBitThemAllApp` transmite a todos los controladores (el código correspondiente a este método puede verse en el listado 5.3).

- **processUserInput:** Comprueba si han ocurrido eventos en la interfaz de usuario y se transmiten al modelo.
- **executeLogic:** El objetivo es que procese las actualizaciones en el estado del modelo (lógica en la aplicación **YAOMEV**), se deja sin implementar pues es específico del juego a crear.
- **render:** Su propósito es ordenar al gestor de escena que dibuje la escena si está activo, también se deja sin implementar para que cada controlador específico decida si se debe dibujar o no la escena.

Listing 5.3: Pasos del bucle principal establecidos en `CYaoController`

```
1 void CYaoController::onRun(float dt){
2         CApplicationState::onRun(dt);
3
4         processUserInput();
5         executeLogic(dt);
6         render(dt);
7 }
```

A continuación se explican los diferentes controladores creados para **Bit Them All!!!**.

5.3.2. Controlador específico CLoadState

Este estado activa la vista de carga y establece el comportamiento de la aplicación durante la carga de los recursos necesitados para la ejecución del juego. En este controlador se hace uso tanto del gestor de DAOs como de las cachés de recursos (sección 5.4), con ellos realiza la carga de los archivos de configuración del juego y de todos los recursos necesarios, para posteriormente almacenarlos en memoria.

El proceso de carga e inicialización del juego **Bit Them All!!!**se ha establecido en tres pasos secuenciales para hacer más comprensible y mantenible esta tarea.

Está compuesto por una clase base llamada `CLoadState` que contiene tres subestados simples (no contienen a su vez más estados): `CConfigLoading`, `CResourceLoading`

y CLogicCreation. Los tres se ejecutan en el orden en que se han nombrado y las transiciones se desencadenan cuando uno de ellos ha terminado, hasta que CLogicCreation lanza una transición que activa el estado CMenuState.

La clase CLoadState crea los estados hijos y establece como estado inicial CConfigLoading que se encarga de cargar el fichero de configuración inicial en el que se establecen los parámetros de inicio del juego tales como son la resolución de la ventana y las rutas donde se almacenan los ficheros de configuración del juego y los recursos gráficos, de video, de audio y fuentes. De esta manera la aplicación sabe donde encontrar los diferentes elementos que cargará en los pasos sucesivos.

Cuando CConfigLoading termina, provoca la transición para que se comience a ejecutar CResourceLoading, que cargará todos los recursos del juego y los almacenará en las cachés de recursos: texturas, botones, objetos 3D sin animación, objetos 3D con animaciones, sonidos, efectos sonoros y vídeos.

El último en ejecutarse es CLogicCreation, que se encarga de leer los archivos de configuración del juego y crear las clases correspondientes de la lógica del juego. Estos elementos son: los escenarios, los personajes y sus animaciones. Cuando acaba, este subestado lanza la transición que inicia CMenuState.

Un aspecto a destacar de la implementación de este estado es que debido a la poca lógica interna que se necesitaba se ha preferido no crear una capa de lógica para el mismo. Por lo tanto, toda la programación se ha realizado directamente en los controladores.

5.3.3. Controlador específico CMenuState

Este controlador activa el estado de menú, en el que se presenta un interfaz de usuario para moverse a lo largo de las diferentes opciones que ofrece **Bit Them All!!!**. Esta interfaz se ha creado mediante la aplicación de los widgets creados para **YAOMEV**.

Al igual que en el estado de carga, la parte de código correspondiente al modelo no tiene la suficiente entidad como para que se cree una capa de lógica específica para el menú, por lo que su implementación se ha hecho en el propio controlador.

Al jugador se le ofrecen cuatro opciones:

- **Jugar:** Comprueba si se han elegido jugadores y escenarios, si no se ha hecho se eligen aleatoriamente y se lanza la transición que activa el estado `CGameState`.
- **Selección de jugadores:** Activa un menú diferente en el que se permite elegir los cuatro jugadores que participarán en el juego.
- **Selección de escenario:** Activa un menú en el que se da la opción de elegir el escenario en el que se quiere jugar.
- **Salir:** Lanza la señal de terminación y provoca la finalización de la aplicación.

Para la selección de jugadores y la selección de escenario se han creado los controladores específicos: `CPlayerSelectionSubstate` y `CScenarioSelectionSubstate`. Cada uno de los cuales se encarga de la representación visual y de manejar las opciones del menú correspondiente.

La gestión de transiciones en este estado se realiza mediante la captura de los eventos provocados por la interfaz de usuario. Los controladores sobreescriben los métodos correspondientes y se encargan de provocar la transición cuando se presiona el botón adecuado.

5.3.4. Controlador específico CGameState

Este controlador se encarga de gestionar la entrada durante el transcurso del juego **Bit Them All!!!** , su comportamiento en un nivel intermedio y la fachada lógica que da acceso a los subsitemas del juego (sección 5.6)

Hay tres tipos de subestados (o controladores hijos):

- **CIntroSubstate:** Este controlador recibe cualquier pulsación de teclado y se comunica con la lógica para mandarle un comando. Se activa cuando la fachada lógica le comunica que se va a desarrollar una secuencia no interactiva.
- **CPlayingSubstate:** Este controlador activa las teclas que permiten responder en el concurso. Se activa cuando la fachada lógica le comunica que se va a desarrollar una secuencia en la que el jugador participa.

- **CPauseSubstate:** Este controlador habilita el ratón para poder interactuar con el menú de pausa. Este controlador pausa la partida y posibilita terminar el juego o continuarlo.

Es la lógica la que desencadena las transiciones mediante solicitudes, a sus *Listeners* `GameEventListener`, de cambio de estado (sección 5.6.1). Por otro lado, los controladores de juego se comunican con la fachada mediante la transformación de los eventos de entrada de usuario en comandos específicos del juego. En la figura 5.37, de la sección 5.6.2, se puede ver un esquema de como funciona este proceso.

5.4. Subsistema de carga y gestión de recursos

Este subsistema se encarga de tratar con los elementos persistentes que necesitará el juego. Se entiende que un elemento es persistente cuando la información del mismo permanece en el ordenador aunque la aplicación se haya cerrado, pudiendo ser recuperada en cualquier otro momento. Para ello **YAOMEV** proporciona las clases necesarias para realizar la carga de los elementos básicos de un juego.

En la plataforma **YAOMEV** se ha buscado externalizar en la medida de lo posible los datos de los que harán uso los juegos. De esta manera se facilita la modificación y testeo del juego sin necesidad de cambiar el código y recompilarlo. Para ello se distinguen dos tipos de elementos a cargar, que además se procesan de forma diferente.

- **Datos de configuración.** Estos archivos definen aspectos de configuración del juego como puede ser la posición de los elementos del escenario, el lugar desde el que comienzan los personajes, las cámaras a usar, las preguntas, qué texturas componen cada widget, etc....

  Como estos archivos comparten un lenguaje de definición de datos común (XML) se usa para su lectura un conjunto de *parsers* (lectores de archivos), que transforman los datos de esos documentos en entidades del juego.

  En concreto, para el juego demostrador existe un parser por cada archivo de configuración.

- **Recursos.** En **YAOMEV**se ha definido como recurso todo elemento persistente con un alto costo de carga. Dentro de esta definición se incluyen la música, texturas, fuentes, modelos 3D y vídeos.

  Para su carga y almacenamiento se han usado cachés de recursos (sección 5.5), sistemas que realizan la carga de los elementos a petición de algún cliente externo y que a partir de ahí permanecen almacenados en memoria por si son solicitados de nuevo, ahorrando con ello el costo de una nueva carga y que exista en el juego información duplicada.

En la implementación de los componentes de carga se han usado dos aproximaciones diferentes que son explicadas con detalle a continuación.

Para el almacenamiento de los datos de configuración se han usado archivos XML que actúan como bases de datos. Se ha usado este sistema porque la cantidad de datos a almacenar no es lo suficientemente grande como para justificar el uso de un sistema de gestión de bases de datos, como podría ser MySQL, ya que este tipo de software ve resentida su eficiencia y ventajas cuando no se disponen de muchos datos o no se van a hacer consultas complejas sobre los mismos.

Aún así en previsión de futuros cambios, o desarrollos de juegos más complejos, se ha usado una versión simplificada del patrón de diseño *DAO (Data Access Object)* con el objetivo de conseguir un alto nivel de independencia con respecto a la base de datos concreta que se planee usar.

El patrón *DAO* es un componente software que suministra una interfaz abstracta común entre la aplicación y el dispositivo de persistencia utilizado. Esta interfaz proporciona algunas operaciones básicas sin exponer los detalles de la base de datos concreta. El objeto *DAO* se responsabiliza de gestionar la conexión con la fuente de datos para obtener y almacenar en memoria esa información.

Para unificar el acceso a los *DAOs* se usa el patrón *Factory*. Este es un patrón de diseño creacional que tiene como objetivo centralizar la creación de los *DAOs*, devolviendo instancias de la interfaz común, haciendo transparente de este modo, al cliente, el tipo de *DAO* usado.

Por último, con objetivo de facilitar la gestión de memoria usada en la factoría de *DAOs*

y a su vez globalizar su acceso, se ha usado el patrón *Singleton* tanto en la *factoría* como en los *DAOs*. Este es otro patrón de diseño creacional, garantiza la existencia en memoria de una sola instancia tanto de la factoría como de los DAOs y además, mediante métodos estáticos de acceso a esa única instancia la hace disponible globalmente.

Los *DAOs* se crean como *Singleton* también porque uno sólo puede acceder al dispositivo de persistencia las veces que hagan falta. Tener más de uno sólo complicaría su gestión.

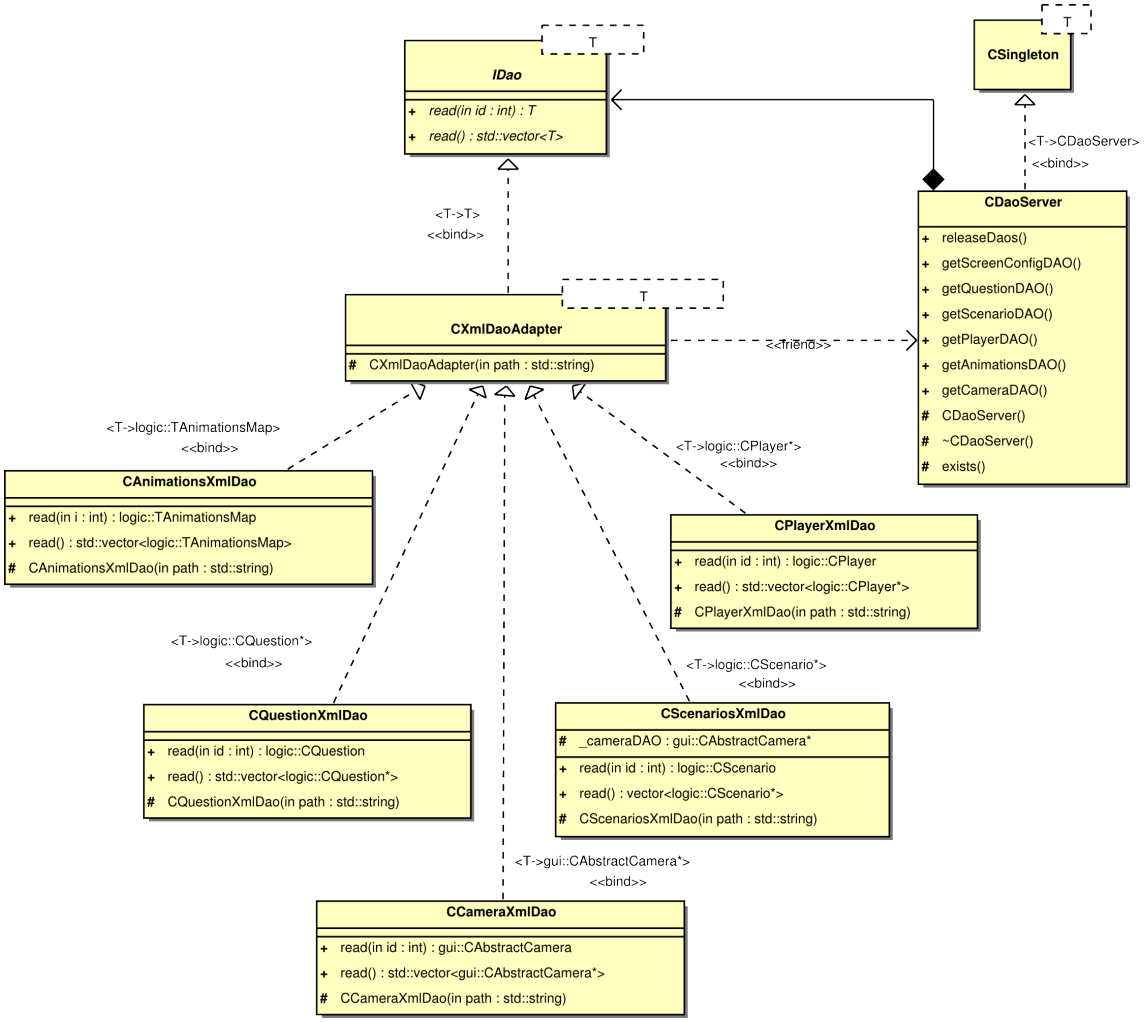

Figura 5.9: Diagrama de clases de la implementación de la factoría de DAOs

En la figura 5.9 se muestra el diagrama de clases de la implementación realizada para el juego construido con **YAOMEV**.

Para poder generalizar los DAOs se ha creado una interfaz genérica en forma de plantilla. Esto se ha hecho así puesto que todo DAO debe devolver lo que se conoce como *DTO* (Data

Transfer Object), este tipo de objeto se corresponde con los objetos del modelo (lógica del juego) y contienen la información leída por el DAO. Esta interfaz define dos operaciones:

- `T persist::IDao::read(int id)`: Lee y devuelve un *DTO* referenciado en la base de datos por un número entero que actúa como identificador.
- `std::vector<T>persist::IDao::read()` Lee todos los *DTOs* de la base de datos y devuelve una lista de ellos.

Los DAOs implementan la interfaz CXmlDaoAdapter y los dos métodos anteriores.

CXmlDaoAdapter es una clase que adapta la interfaz IDao para la lectura de documentos XML. En ella se usa TinyXml, pequeña biblioteca para parseo o creación de documentos XML mediante el uso de DOMs (Document Model Objects), una convención independiente del lenguaje usada para la representación y manipulación de elementos en XML, HTML o XHTML.

Para definir el juego **Bit Them All!!!**se usan varios archivos diferentes, cada uno de los cuales cuenta con sus correspondientes *DAOs* para interpretarlos:

- **players.xml**: Especifica la información necesaria para poder cargar los personajes del juego, es leído por `CPlayerXmlDao` y por `CAnimationsXmlDao`.
- **questions.xml**: Almacena las preguntas que se usarán en el juego. Es leído por `CQuestionsXmlDao`.
- **config.xml**: Contiene la información para la creación de la ventana de juego. Es leído por `CScreenConfigXmlDao`.
- **cameras.xml**: Todos los escenarios usan este archivo para definir las cámaras disponibles en el mismo. Es leído por `CCamerasXmlDao`.
- **scenario.xml**: Este archivo indica la disposición de los elementos de un escenario, tanto los estáticos como los dinámicos. Es leído por `CScenarioXml`

La clase `CDaoServer` es la factoría de *DAOs* explicada anteriormente. Es conveniente resaltar que para evitar la creación de los *DAOs* en otros lugares del código, se ha establecido

la visibilidad de sus constructores como *protegida* de tal forma que sólo `CDaoServer`, que es una clase amiga de los `DAOs`, puede crearlos.

Si en algún momento se decidiera cambiar la persistencia a, por ejemplo, un sistema de gestión de bases de datos relacional solo habría que crear los *DAOs* necesarios y adaptar la factoría para que los devolviera.

Esta parte es muy dependiente del tipo de juego que se esté haciendo, por lo tanto estos *DAOs* en cuestión son específicos del juego demostrador **Bit Them All!!!**. Sin embargo con esta arquitectura se permite la fácil adición de nuevos *DAOs* sin afectar con su desarrollo (o cambios) al resto del juego.

Sin embargo **YAOMEV**si ofrece una serie de herramientas genéricas para la gestión de los recursos del juego, las definidas anteriormente como cachés de recursos. La gestión de los recursos es más fácil de generalizar pues son comunes a cualquier juego que se quiera desarrollar y no son específicos de los mismos.

Cada caché de recursos es específica del tipo de recurso a cargar. Sin embargo todas comparten una estructura común. Su funcionamiento es el siguiente: un cliente externo solicita a la caché un recurso por medio de un identificador, la caché comprueba si ese recurso está ya en una tabla interna, si ya existe se devuelve el recurso previamente cargado y si no existe usa el cargador necesario para leerlo de disco.

Con esta aproximación se resuelve un problema como es el de la memoria que puede llegar a ocupar el uso de múltiples recursos que tienen altas probabilidades de repetirse en un juego como pueden ser: modelos 3D, texturas, sonidos y vídeos.

5.5. Sistema de representación

En el patrón **MVC** el sistema de representación se corresponde con la capa de vista. De forma general, esta capa se encarga de mostrar la información correspondiente a la capa de modelo (lógica del juego) (sección 5.6) mediante gráficos en 3 y 2 dimensiones, de tal manera, que el usuario de un juego desarrollado con **YAOMEV** pueda interactuar con el mismo.

Las escenas gráficas que forman parte de un videojuego tridimensional de cierta entidad son bastante complejas, por lo que la cantidad de información que deben gestionar es muy

grande. En estos casos, bibliotecas de bajo nivel como **OpenGL** son insuficientes por sí mismas, pues su función es facilitar el control del hardware gráfico a bajo nivel. Por lo tanto es necesario abstraer su funcionalidad de tal manera que sea posible crear y manejar escenas gráficas complejas de forma sencilla y controlada. Esta misión es la que debe acometer la capa de vista.

La capa de vista se encarga por tanto de envolver las librerías de bajo nivel y abstraernos de ellas, para conseguirlo se ha adoptado un enfoque modular, quedando descompuesta en los siguientes subsistemas:

- **Gestión de Recursos:** Encargado de mantener en memoria de forma eficiente los recursos del juego, estos pueden ser widgets, modelos 3D, texturas, canciones, efectos sonoros, fuentes y vídeos.
- **Gestor de Escena:** Este componente tiene como propósito dibujar las escenas gráficas tanto 2D, como 3D. Para ello usa nodos de escena específicos de cada tipo de elemento a dibujar. Estos nodos se construyen a partir de recursos obtenidos del sistema de gestión de recursos y se actualizan a partir de la información que la capa de modelo le suministra (sección 5.6).
- **Escena:** Implementación del grafo de escena, ordena los nodos y se los pasa el renderizador para que los muestre en pantalla.
- **Sistema de renderizado:** Se encarga de envolver el API gráfico y de presentar funciones especializadas para dibujar elementos 3D y 2D. Recibe la información de los nodos del grafo de escena y los dibuja.
- **Reproductor de audio:** Envuelve la biblioteca **SDL_mixer** y ofrece un conjunto de funciones para reproducir tanto canciones, como efectos sonoros.
- **Reproductor de video:** Envuelve la biblioteca **smpeg** y ofrece la funcionalidad necesaria para reproducir vídeo como si fuera una textura de un modelo 3D o 2D.
- **Gestión de ventana:** Envuelve **SDL** y ofrece una interfaz para manejar el sistema de ventanas.

- **Gestión de eventos de usuario:** Este sistema se encarga de manejar la entrada de usuario tanto por teclado, como por ratón, para posteriormente comunicar esa información mediante eventos a la capa de modelo (sección 5.6).

La forma en que se ha construido esta capa garantiza su independencia del resto de capas, consiguiendo con ello una serie de ventajas muy convenientes en el desarrollo de videojuegos como son: reutilización (múltiples juegos pueden compartirla), abstracción con respecto al API gráfica (en este caso **OpenGL** y **SDL**) y una mayor facilidad en su mantenimiento (si hay que cambiar la capa de vista, el resto no tienen por qué modificarse).

5.5.1. Subsistema de gestión de recursos

Para conseguir una simulación realista en un videojuego son necesarios diferentes tipos de elementos audiovisuales. Estos elementos, que tienen como misión presentar la información del juego al usuario, son lo que hemos llamado recursos: modelos 3D, texturas, audio, vídeo, etc...

En el capítulo dedicado al sistema de persistencia (sección 5.4) se hablaba de la carga de estos recursos a partir de los ficheros que los definen. Este subsistema sin embargo se encarga de almacenar en memoria esos recursos y de hacerlos accesibles al gestor de escena.

El desarrollo de un gestor de recursos genérico plantea una serie de problemas a resolver con el objetivo de hacerlo lo más útil posible en cualquier situación de desarrollo real de un videojuego.

El primer problema a tener en cuenta es que la información de los recursos almacenados es muy útil para todo tipo de tareas relacionadas con la representación de los mismos, y como consecuencia puede ser necesaria en varios módulos. Por lo tanto es deseable que estos sean fácilmente accesibles por cualquier módulo que necesite los recursos para desempeñar su tarea.

A su vez, los recursos son elementos cuya carga habitualmente es muy costosa, por lo tanto no es una buena práctica cargarlos a medida que se necesiten durante el transcurso de una partida, pues el rendimiento, y con ello la simulación, se verían afectados. Es necesario establecer en el juego una etapa de carga (sección 5.4) donde sólo se carguen aquellos recur-

sos que van a ser utilizados, quedando almacenados en memoria para su uso posterior. Los gestores de recursos deben dar soporte a su almacenaje y un acceso inmediato a los mismos.

Es también común reutilizar la información de esos recursos para definir múltiples entidades de juego. Un caso típico es aprovechar una maya 3D para representar diferentes personajes o elementos del escenario, en cuyo caso, mantener un recurso en memoria por entidad provocaría que el consumo de memoria se disparase. Es por lo tanto indispensable que el gestor mantenga una sola instancia en memoria por recurso.

La última cuestión a resolver se refiere a cómo enfrentar el problema de la gestión de los gestores de recursos. Permitir que cada módulo tenga su propio gestor de recursos puede ser útil en algunos casos, sin embargo eso dispersa la información y la hace más difícil de rastrear y controlar. En pos de la sencillez, por tanto, es necesario que exista un sólo gestor de recursos que se encargue de manejarlos.

Como consecuencia de estos problemas, se han establecido una serie de requisitos que los gestores de recursos deben cumplir y que son comunes a todos:

- Reusabilidad.
- Centralizar el acceso y uso de los gestores de recursos.
- Proveer acceso global a todos los módulos que necesiten los recursos.
- Asegurarnos de que por cada recurso sólo debe existir una instancia en memoria.
- Los recursos deben ser accesibles por una clave única.

En **YAOMEV**se ofrece una serie de herramientas genéricas para la gestión de los recursos del juego que cumplen con casi todos los requisitos anteriores, las definidas como cachés de recursos.

Estas herramientas son fácilmente reutilizables puesto que la gestión de recursos, y los propios recursos, son comunes a cualquier juego que se quiera desarrollar y no son específicos de los mismos.

Cada una es específica del tipo de recurso a almacenar, aunque todas comparten una estructura común. Su funcionamiento es el siguiente: un cliente externo solicita a la caché un

recurso por medio de un identificador, la caché comprueba si ese recurso se encuentra en una tabla interna, si ya existe se devuelve el recurso previamente cargado y si no existe usa el cargador necesario para leerlo de disco. De esta forma quedan asegurados tanto su facilidad de acceso, como que sólo haya un recurso en memoria aunque sea usado por diferentes clases.

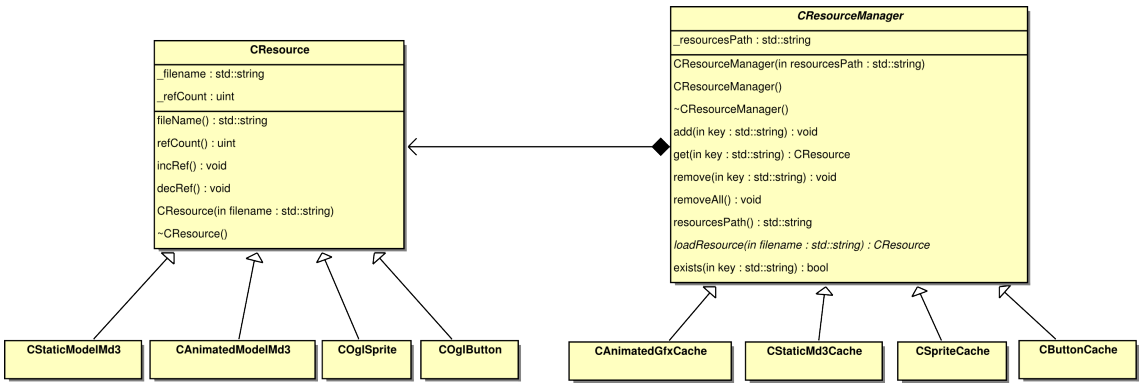

Figura 5.10: Diagrama de clases de las cachés de recursos

En el diseño de las cachés de recursos se ha usado tanto el patrón *Proxy* (sección 6.7), como el patrón *Template Method* 6.11. Con ellos dos se consiguen cumplir algunos de los requisitos mencionados anteriormente (reusabilidad, una instancia en memoria por recurso y acceso por clave única). A partir de ellos se ha creado la clase `gui::CCache` que implementa la caché de recursos y `gui::CResource` que representa el recurso a cargar.

La clase `gui::CCache` es una clase abstracta que presenta el interfaz que toda caché de recursos deberá implementar. Su cometido es almacenar instancias de `gui::CResource` (clase de la que se hablará inmediatamente) y proveer acceso a ellos. Para controlar el acceso a los recursos usa la técnica conocida como "conteo de referencias". Esta técnica es una sencilla forma de controlar cuantas referencias existen a un mismo recurso mediante el uso de un simple contador. Ofrece las funciones:

- **void add(const string& key):** Añade un recurso a la caché controlando que no exista previamente. Si existe no hace nada, si no existe usa la clave para cargarlo en memoria. En cualquiera de los casos, la clave usada como parámetro será la que identifique unívocamente al recurso.

- **CResource* get(const string& key):** Accede al recurso por medio de la clave, si ya

existe incrementa un contador de referencias y devuelve el recurso, en caso contrario lo carga en memoria y lo devuelve.

- **void remove(const string& key):** Elimina el recurso por medio de la clave. Si ya existe decrementa el contador de referencias al recurso y cuando llegue a cero lo elimina.
- **void load(const string& key):** Este método privado se deja sin implementar para que cada gestor de recursos específico lo implemente como sea necesario.

`gui::CCResource` también es una clase abstracta y es el interfaz a implementar por todos los recursos de **YAOMEV**. Ofrece simplemente las funciones necesarias para llevar la cuenta de las referencias hechas al recurso.

Para conseguir cumplir con las restricciones impuestas antes se debe conseguir un acceso global a las cachés y que sólo exista una única instancia en memoria. Para conseguirlo, todas ellas se han organizado dentro de una clase que centraliza su uso y acceso. Esta clase es lo que en **YAOMEV**se ha denomidado como "servidor gráfico" `gui:CGraphicServer`. El servidor aisla al programador de las diferentes cachés consiguiendo con ello una utilización más sencilla.

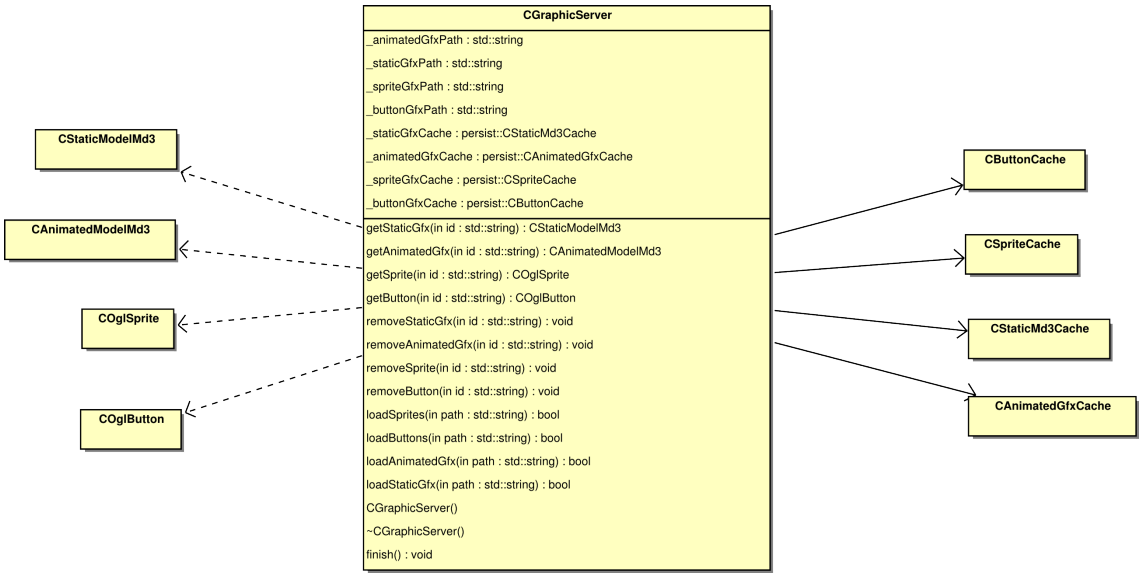

Figura 5.11: Diagrama de clases de servidor gráfico

La clase `gui::CGraphicServer` está implementada siguiendo las guías de los pa-

trones *Abstract Factory* (sección 6.1) y *Singleton* (sección 6.3) y ofrece tres tipos de funciones: de acceso a los recursos, de destrucción de los recursos y de carga de los recursos.

- **Métodos get:** Solicita un recurso a la caché correspondiente por medio de una clave única. Si existe lo devuelve, sino intenta cargarlo y si esto no es posible devuelve una referencia nula.
- **Métodos remove:** Elimina una referencia en la caché de recursos correspondiente, si esta referencia es la única borra el recurso de la caché.
- **Métodos load:** Estos métodos se usan para precargar elementos en las cachés a partir de un fichero .xml.

5.5.2. Gestor de escena

En el desarrollo de la arquitectura **YAOMEV** se ha comentado anteriormente (sección 5.1) la importancia de mantener separadas las diferentes capas de la aplicación. Por lo tanto, en la creación de la capa de vista inmediatamente surge el problema de la creación y uso de los gráficos desde la capa de lógica. El objetivo es mantener al creador de juegos alejado de la representación interna de los gráficos, por lo tanto hay que procurar otorgarle un uso sencillo de la capa gráfica que no requiera conocimientos específicos de esa parte.

Para lograr este objetivo es necesario que haya un elemento encargado de la creación de los nodos que componen el grafo de escena (sección 5.5.3) y que controle este grafo mandándolo dibujar pero que, a su vez, no sea usado directamente por el programador. El gestor de escena `gui::CSceneManager` se encarga de aportar esa separación en la creación de los nodos del grafo de escena, de la utilización del grafo de escena y de otros elementos gráficos como son las luces y las cámaras. En la figura 5.12 se puede ver el diagrama de esta clase.

Esta clase no tiene porque ser usada directamente por el programador, más allá de agregarla como observador de la lógica del juego, como se puede apreciar en el siguiente fragmento de código correspondiente al controlador `app::CGameState`, el cual es el controlador principal del juego **Bit Them All!!!**.

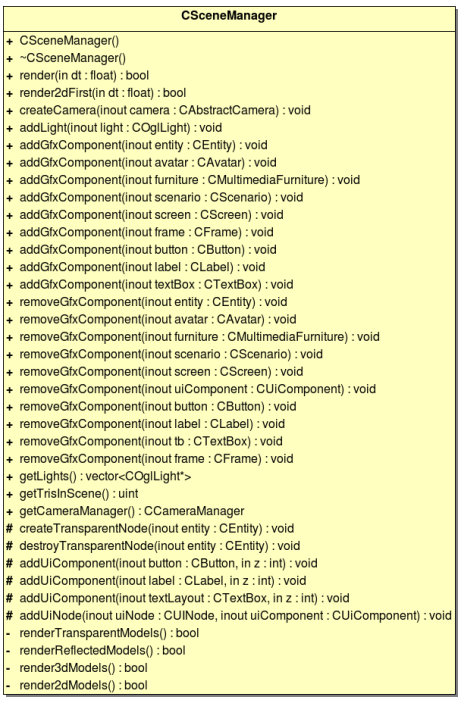

Figura 5.12: Clase CSceneManager con todos sus métodos y atributos

Listing 5.4: Inclusión del gestor de escena como observador en el método onCreate de la clase CGameState

```
1  void CGameState::onCreate(){
2          CYaoCompositeController::onCreate();
3
4          // Creamos el gestor de escena y el gestor de camaras
5          activateGfxManagers();
6
7          // Adicion de los listeners necesarios a la logica
8          _game->addListener((logic::ILogicFacade::CreationDestroyListener*)_sceneManager);
9          ...
10 }
```

El gestor de escena es un atributo de la clase `app::CYaoController` (figura 5.13), de esta forma cualquier controlador que se cree para el juego tendrá acceso a esta clase. Se ha dejado en manos del programador su uso concreto dentro de cada controlador específico de cada juego, para evitar de esta forma que los controladores tengan que tener asociada una escena gráfica.

La creación de nodos se realiza, como se puede apreciar en la figura 5.14, a partir de los elementos de la lógica de juego mediante el uso del patrón *Observer* (sección 6.8). En la figura

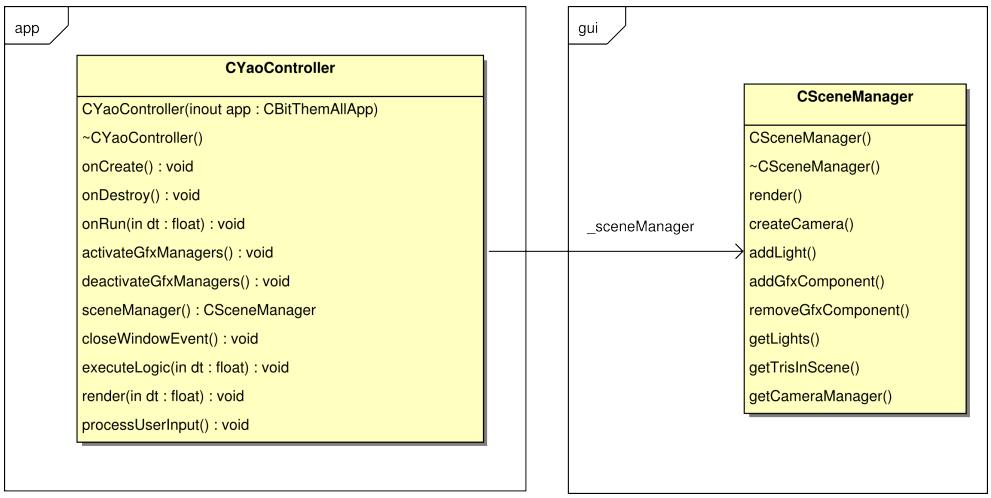

Figura 5.13: Clase CSceneManager como atributo de CYaoController

se puede ver como gracias al uso de la interfaz `logic::CreationDestroyListener` se desacopla la capa de lógica del juego de la capa de vista. Cualquier gestor de escena que implemente esta interfaz puede ser usado por la fachada lógica `logic::ILogicFacade` para crear nodos de escena para el juego. Se hablará más sobre este proceso en el capítulo dedicado a la capa de lógica del juego (o capa de modelo) (sección 5.6).

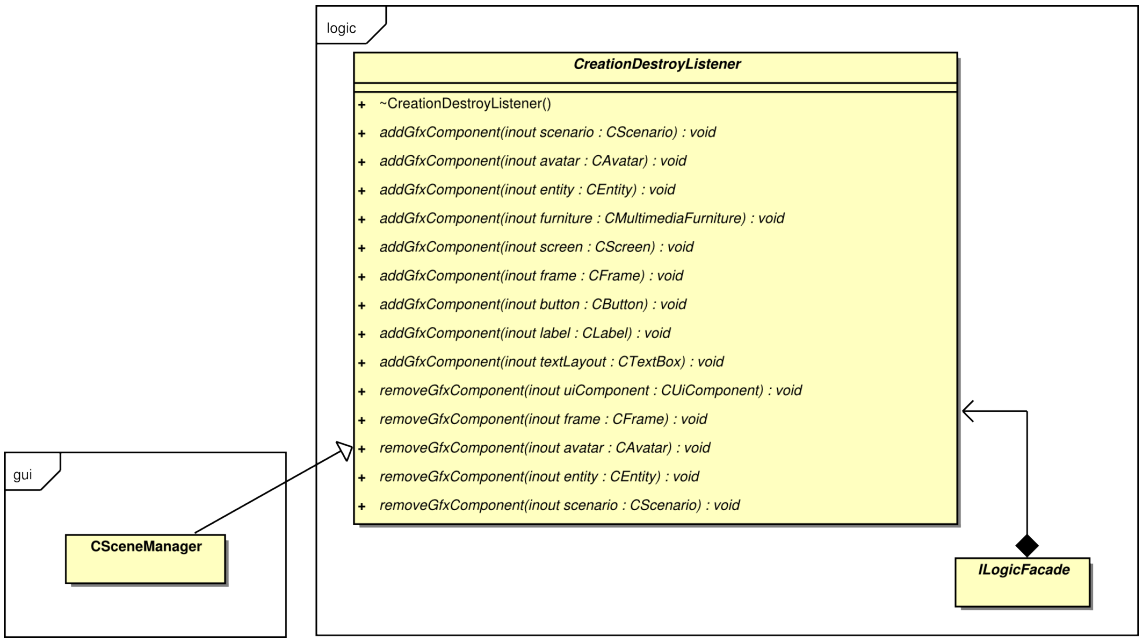

Figura 5.14: Diagrama de clases de la implementación del patrón Observer en la creación y destrucción de nodos de escena

El gestor de escena, queda claro entonces, actúa de adaptador entre los elementos de la capa de modelo y los de la capa de vista. De esta forma, desde la fachada lógica (sección 5.6) se pueden construir los nodos del grafo de escena a partir de las entidades lógicas, sin necesidad de preocuparnos por cómo se van a crear o cómo se van a dibujar. Estos nodos quedan almacenados en el grafo de escena y este es usado, por el gestor de escena, para dibujar el mundo gráfico tal y cómo se haya especificado en el controlador que maneje la escena.

La clase `app::CGameState` es controlador creado para el juego **Bit Them All!!!** , en el, a parte de ser añadido el gestor de escena como escuchador de los eventos de creación y destrucción de entidades gráficas, se usa también para renderizar la escena, como se puede ver en el siguiente ejemplo de código.

Listing 5.5: Uso del gestor de escena para renderizar en la clase CGameState

```
1  void CGameState::render(float dt){
2          if(_pause) dt = 0;
3
4          // Limpiamos la pantalla
5          _window->clearScreen();
6
7          _sceneManager->render(dt);
8
9          // Intercambiamos los buffers de video
10         _window->swapBuffers();
11 }
```

La tarea de renderizado se ha organizado en el gestor de escena mediante el uso de grafos de escena. En concreto se usan dos grafos de escena: uno para el dibujado de elementos gráficos tridimensionales y otro para la representación de elementos bidimensionales (figura 5.15). En el siguiente capítulo se hablará en profundidad de ellos.

5.5.3. Grafo de escena

Cuando un videojuego lidia con escenas gráficas complicadas que involucran elementos como pueden ser: entidades de geometría compleja, texturas, materiales, transformaciones, animaciones, cámaras, luces, etc... se hace necesaria una estructura de datos de alto nivel que permita gestionar toda esta información de forma eficiente.

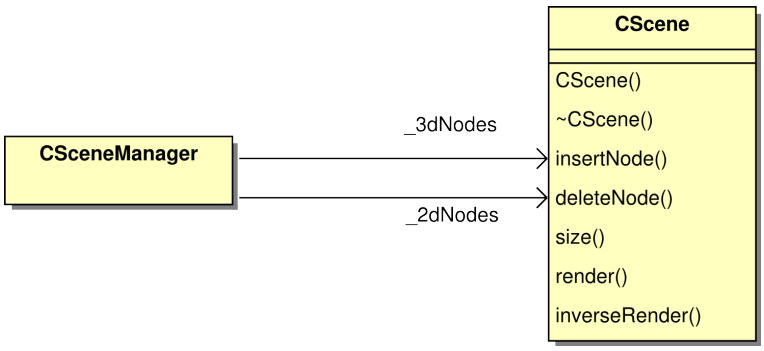

Figura 5.15: Grafos de escena usados en el gestor de escena

Esta estructura se conoce como “grafo de escena”, la cual hoy por hoy es la forma más común de organizar toda esta información para su representación visual. El grafo de escena agrupa todo el entorno virtual y lo controla dependiendo de la forma en que se haya implementado. Los grafos de escena al ser grafos acíclicos y dirigidos, realmente se comportan como un árbol n-ario, cuyo objetivo es organizar la geometría de la escena en una jerarquía que simplifique su utilización al usarse en una simulación en tiempo real. De esta forma se facilita enormemente el tratamiento de las escenas gráficas, permitiendo diferentes aplicaciones sobre ella como son particionamiento espacial, aceleración de la visualización, detección de colisiones, etc...

En **YAOMEV** el grafo de escena se ha construido intencionadamente muy sencillo para evitar una excesiva complejidad en su uso o comprensión (figura 5.16). Los nodos se organizan en una lista para su dibujado en función de su prioridad. Esta prioridad es variable y en cada vuelta del bucle la lista se organiza en función de la prioridad de los nodos, logrando con esto que se dibujen antes los elementos más importantes de la escena gráfica.

Y aunque no se ha usado ningún tipo de particionamiento espacial (ni ninguna regla específica para ahorrar nodos en el dibujado más allá de un atributo que impida que se dibujen los nodos marcados como no visibles). Si que se ha creado de tal manera que sea fácil añadir nuevas implementaciones del grafo de escena. Para ello se han dejado los métodos de inserción/borrado de nodos y el de renderizado como virtuales, de esta forma se puede plantear la opción de crear una jerarquía de grafos de escena para luego poder elegir entre los que haya disponibles. En la figura 5.17 se muestra la jerarquía de los nodos de escena usados en

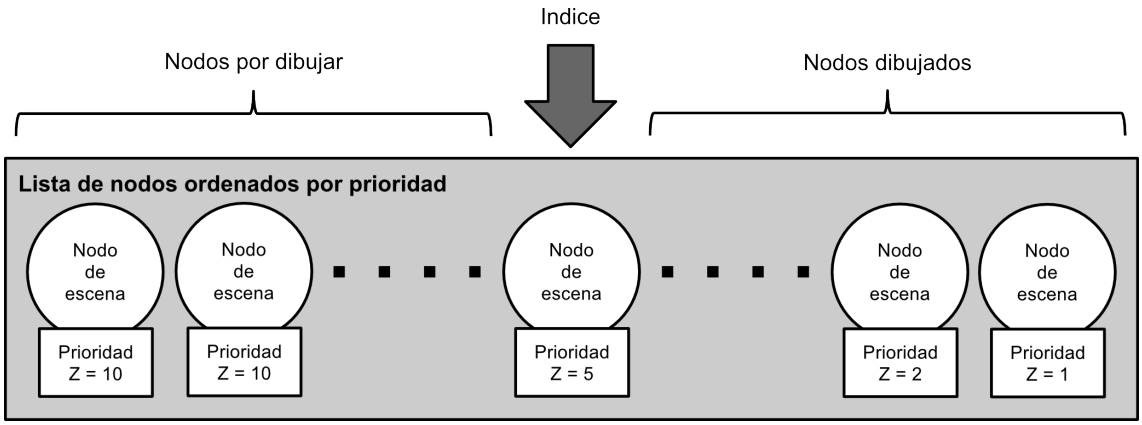

Figura 5.16: Organización de los nodos en el grafo de escena simple incluido en **YAOMEV**

YAOMEV.

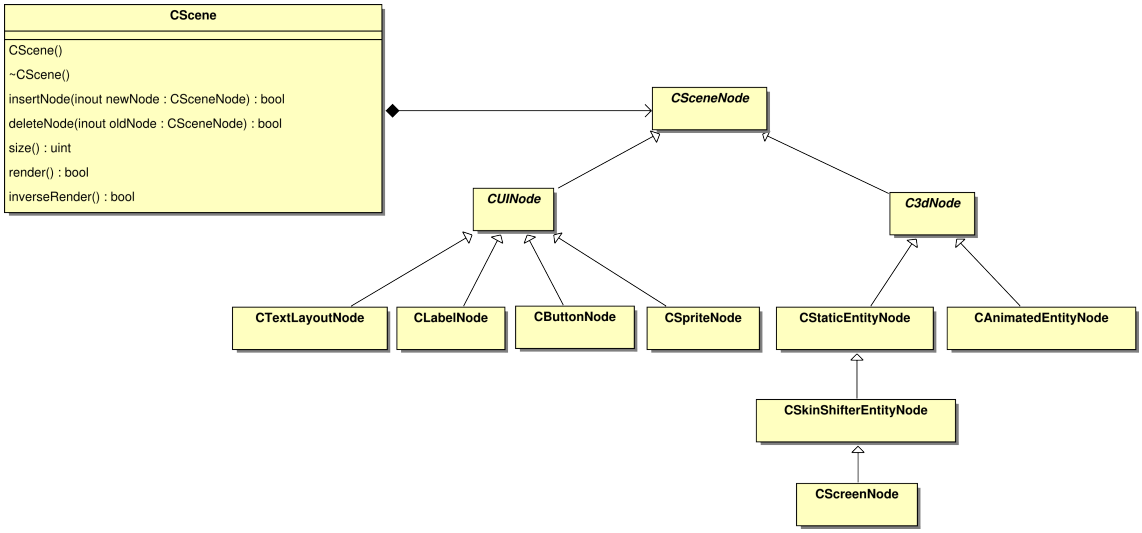

Figura 5.17: Diagrama de clases y jerarquía del grafo de escena

Hay dos tipos de nodos en la jerarquía, los nodos que representan elementos de la interfaz de usuario (elementos bidimensionales) y los nodos que representan entidades tridimensionales. Los nodos de escena en **YAOMEV** encapsulan la información de las entidades participantes en el juego, a la vez que la información de los recursos gráficos necesarios, para de esta manera lograr una representación eficiente de las mismas.

Todos los nodos se han implementado siguiendo el patrón de diseño *Flyweight* (sección 6.6), con la meta de impedir que cuando haya un recurso compartido por más de un nodo

existan varias instancias del mismo recurso en memoria.

Para conseguir esto fue necesaria la identificación de los estados tanto extrínseco, como intrínseco, de cada nodo de escena. El estado extrínseco se refiere a aquellos atributos que son particulares de cada nodo que usa el recurso, mientras que el intrínseco se refiere a aquellos atributos que se pretenden compartir. Por ejemplo: En **Bit Them All!!!** dos cajas iguales, situadas en las coordenadas (1, 0, 0) y (-1,0, 0), tendrían como estado intrínseco la propia maya de la caja con su textura, y como estado extrínseco la posición. Esta ha sido una de las razones por la que existen las cachés de recursos gráficos (sección 5.5.1).

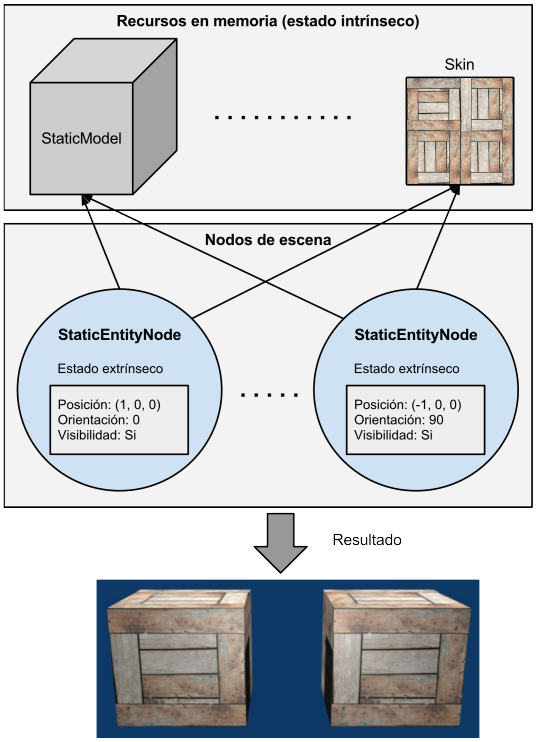

Figura 5.18: Ejemplo de la compartición de recursos en la representación de dos cajas en diferentes posiciones.

Cada nodo del grafo de escena representa a una entidad lógica del juego, por lo tanto en **YAOMEV**, el estado extrínseco se corresponde con aquellos atributos propios de los elementos de la jerarquía de entidades de la capa de lógica (sección 5.6.3). El estado intrínseco mientras tanto se corresponde a la información gráfica que representa a estas entidades lógicas y que es muy dada a su reutilización. En la figura 5.18 se muestran las estructuras de datos usadas para la representación del ejemplo anterior de las cajas y se puede apreciar qué

atributos corresponden a cada estado.

Al igual que se hizo con la creación de los nodos en el gestor de escena, como se puede ver en la figura 5.14, para separar la representación gráfica de una entidad del juego (es decir, su nodo gráfico) de la entidad lógica misma, se usa el patrón *Observer* (sección 6.8) para desacoplar la transmisión de información entre ellas, y es en la creación del nodo, donde estos se añaden como *Listener* de las entidades lógicas, como bien podemos apreciar en el listado de código 5.6.

Se hablará más sobre esto en la parte correspondiente a la jerarquía de entidades de juego (sección 5.6.3) y en la siguiente sección, en la que se tratarán los nodos de escena en profundidad.

Listing 5.6: Creacion de un nodo a partir de una entidad lógica

```
void CSceneManager::addGfxComponent(logic::CEntity* entity){
        [...]
        // Recuperamos el recurso
        CStaticModelMd3* model = CGraphicServer::instance().getStaticGfx(entity->id());
        if(!model) return;

        // Creamos el nodo correspondiente
        CStaticEntityNode* node = new CStaticEntityNode(model);

        // Al incluir la entidad logica en el nodo hacemos
        // que este se incluya como Listener de la entidad
        node->setLogicEntity(entity);

        // Incluimos el nodo a la escena
        _3dNodes->insertNode(node);

        // Guardamos en una tabla al nodo, indexado por la
        // entidad logica a partir de la cual se ha creado
        _staticGfxTable[entity] = node;
        [...]
}
```

5.5.4. Nodos de escena

En esta subsección se hará un repaso a los diferentes nodos de escena empleados en **YAOMEV** y a su implementación. Uno de los aspectos que más quebraderos de cabeza pueden dar a un programador de videojuegos es la correspondencia entre las entidades que participan en su juego y los gráficos que los representan, puesto que un mal planteamiento de este problema

puede dar lugar a un acoplamiento muy grande (e indeseable), entre la capa de vista y la capa de modelo, haciendo con ello imposible su portabilidad o modificación. En **YAOMEV** se ha abordado este problema buscando aislar al programador de la capa gráfica en la medida de lo posible.

Para lograr la separación entre las capas se ha usado una jerarquía de nodos de escena que se corresponde en cierta medida con la jerarquía de entidades lógicas (en los nodos 3D) y con la jerarquía de los componentes de interfaz de usuario (en los nodos 2D) (sección 5.6.3). Cada uno de los nodos implementa los interfaces de los *Listeners* necesarios para desacoplar la representación visual, tanto de las entidades del juego, como de los componentes de la interfaz gráfica (sección 6.8).

Primero se analizarán los nodos correspondientes a la representación de elementos tridimensionales. En la figura 5.19 se puede ver la definición completa de estas clases.

En la jerarquía se aprecia enseguida que existe un nodo raíz `gui::C3dNode`. Este nodo no dibuja nada, es una clase abstracta que deja el método de renderizado sin implementar, simplemente sirve como interfaz para el resto de nodos que vayan a representar un objeto 3D. Implementa el interfaz del *Listener* `logic::CEntityListener` que permite que la entidad lógica raíz `logic::CEntity` (sección 5.6) le transmita información sobre: la posición, la orientación (rotación de la entidad en el eje Y), la visibilidad, la textura y la escala. Todos estos atributos pertenecen al estado extrínseco de todos los modelos 3D que se usen en **YAOMEV** (sección 5.5.3).

Partiendo del nodo raiz hay dos tipos de nodos 3D: estáticos y animados. El primero permite dibujar elementos gráficos que no incluyen animaciones, mientras que el otro se usa para la representación de personajes del juego. Dentro de los estáticos hay un caso especial, que es el nodo de pantalla, pensado para que elementos estáticos puedan usar como textura un vídeo. Como se ha dicho antes, cada uno de estos nodos implementa un *Listener* diferente en función de los atributos que va a recibir de las entidades lógicas a las que representan, por lo que todos ellos incorporan un método para añadirse como *Listeners* de las entidades lógicas llamado `setLogicEntity`, que es usado en la creación de los nodos en `CSceneManager` (ver listado de código 5.6).

Tanto el nodo que representa entidades animadas, como el que representa entidades es-

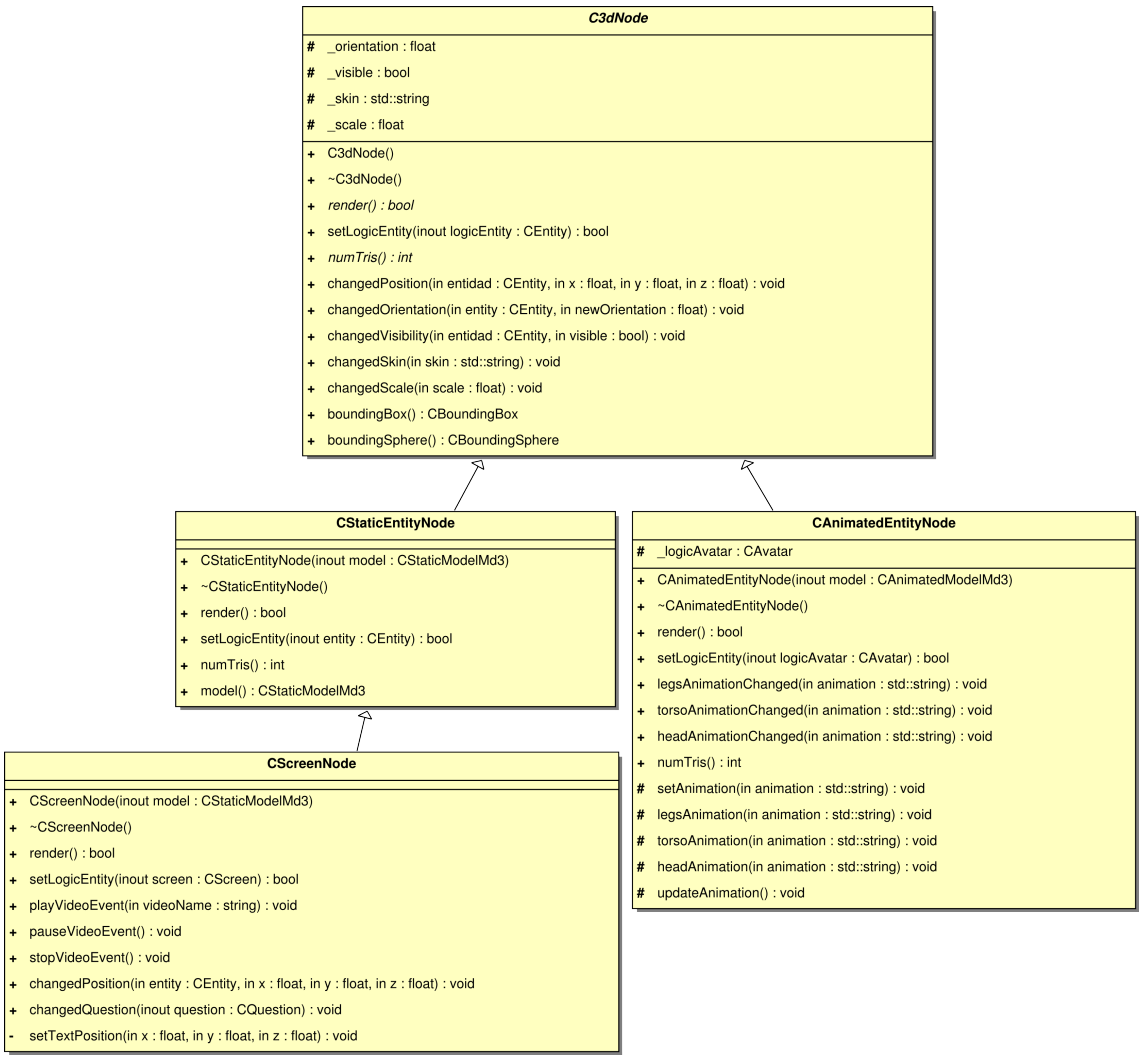

Figura 5.19: Jerarquía detallada de los nodos correspondientes a elementos 3D.

táticas, hace uso de un recurso cada uno. Estos recursos usados por estos nodos son los que contienen toda la información que define un modelo 3D. En la figura 5.20 se muestra el diagrama de clases con estas relaciones.

Las entidades animadas están envueltas en una clase que actúa de *wrapper* de la información del modelo animado. Esta clase, especialización de `CResource`, es `CAnimatedModel`, que está dividida en tres partes: lower, upper y head. Cada uno de estos atributos es una estructura de datos `tModel` (sección 5.5.5), que conjuntamente representan a un personaje del juego. Las entidades estáticas, por otra parte, están construidas de igual manera, exceptuando que sólo contienen una estructura `tModel` para formarlo.

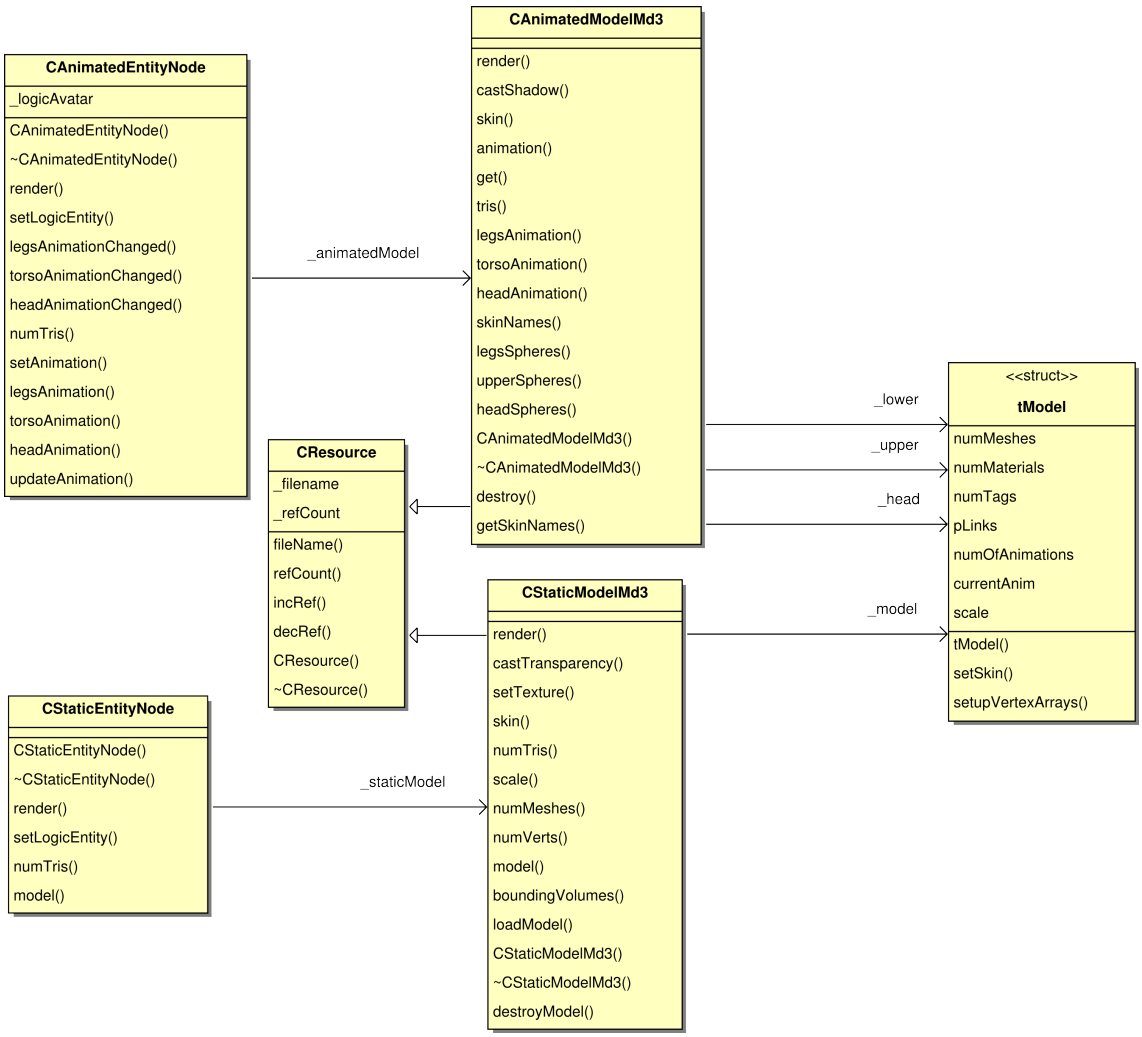

Figura 5.20: Relaciones entras las clases de los nodos de escena y los recursos gráficos.

Los nodos de escena 2D sirven para lo mismo que los nodos 3D, con la excepción de que su tarea es representar los widgets 2D que permiten crear interfaces de usuario. Existen como nodos separados para permitir su uso ajeno al de los nodos 3D. En la imagen 5.21 podemos ver la jerarquía creada.

Su implementación ha seguido los mismos pasos que con los nodos 3D. Cada nodo contiene un recurso que es solicitado a una caché en su construcción (dependiendo del nodo puede ser la caché de botones, la de sprites o la de fuentes), exceptuando el nodo `CTextLayoutNode`, puesto que se ha considerado que no tiene sentido almacenar un texto largo por si se usa más de una vez en la misma escena del juego. Esa caché, junto con la externalización

de parte de los atributos de estos nodos, componen una implementación del patrón *Flyweight* (sección 6.6) del que ya se habló antes.

Figura 5.21: Jerarquía de nodos de escena 2D.

Estos nodos, mediante el uso del patrón de diseño *Observer* (sección 6.8), sirven para desacoplar la representación de las entidades lógicas del juego de su representación gráfica. Esta implementación se ha hecho así para conseguir que el programador no tenga que ocuparse de la representación gráfica de las entidades del juego, pudiendose concentrar sólo en la

programación de las mecánicas del juego en la capa de modelo(o lógica) (sección 5.6). A su vez, otra de las principales ventajas que otorga esta aproximación es la de poder crear varias vistas distintas por cada entidad de la lógica, creando símplemente un nodo de escena nuevo que implemente la interfaz del *Listener* correspondiente (en **YAOMEV** se ha llamado a los observadores `Listeners`).

Por último, la implementación del patrón *Flyweight* (sección 6.6) permite un ahorro sustancial de memoria y tiempo en la representación de los gráficos en **YAOMEV**.

5.5.5. Modelos 3D

En la realización de un sistema de visualización 3D, es necesario contar con un sistema que permita describir la geometría de los elementos que estarán involucrados en una escena cualquiera. Además también es importante que cuente con información sobre la animación de esos elementos (en caso de que contaran con ella), así como información sobre su aspecto (texturas).

En **YAOMEV** se ha usado una representación de modelos 3D con animación basada en vértice inspirado en el formato de los archivos de definición de modelos MD3 (capítulo 3).

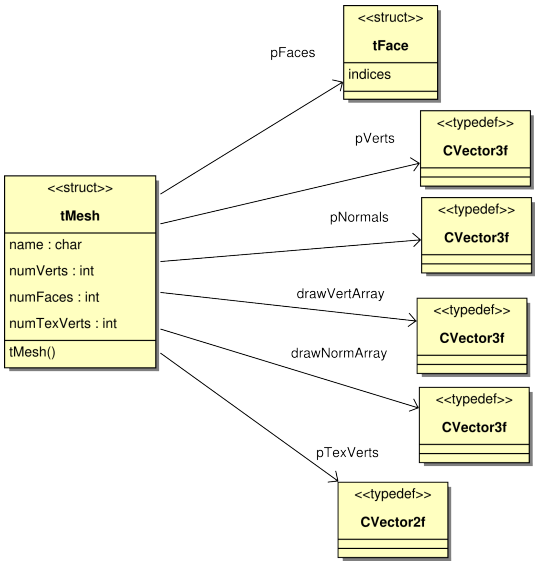

Figura 5.22: Estructura que define la geometría de una malla 3D.

En primer lugar, tal y como se aprecia en la figura 5.22 los modelos se dividen en mallas

3D, que son un conjunto de geometría que tiene asignada una sóla textura. Estas mallas se representan mediante una estructura llamada `tMesh` que contiene la información de la geometría de esa malla: vértices, normales, caras y coordenadas de textura. En esta estructura se definen los siguientes atributos:

- **Nombre:** Nombre que tenga asignado la malla.
- **numVerts, numFaces, numTexVerts:** Número de vértices, caras y coordenadas de textura.
- **pVerts:** Array con todas las coordenadas de los vértices de la malla en cada frame de animación.
- **pNormals:** Array con todas las coordenadas de las normales de cada vértice de la malla en cada frame de animación.
- **pFaces:** Array a elementos de la estructura `tFace`, los cuales contienen un indice a cada vértice y normal, definiendo con ello una cara de la malla (o triángulo).
- **drawVertArray:** Como `pVerts` contiene la posición de cada vértice en cada frame de animación, se usa este array auxiliar para mandar a dibujar en un frame dado un conjunto de los vértices almacenados en `pVerts`.
- **drawNormArray:** Este array cumple el mismo propósito que `drawVertArray`, pero con las nórmales almacenadas en `pNormals`.

Después, estas mallas se agrupan en una estructura mayor, llamada `tModel`, que representa un modelo 3D que agrupa un conjunto mayor de información para definir la geometría de un modelo complejo.

Una estructura `tModel` contiene la información de un modelo 3D con animación basada en vértice, contiene todas las mallas que componen el modelo, información sobre las animaciones y los frames en que se descomponen, contiene todas las texturas que se pueden aplicar a las mallas, sus volúmenes que contienen al modelo y unas estructuras especiales denominadas `tQuatTags` que sirven para establecer puntos especiales en el modelo (ya sean puntos de anclaje para otras mallas, o para indicar el punto que señala la posición del modelo, etc...)

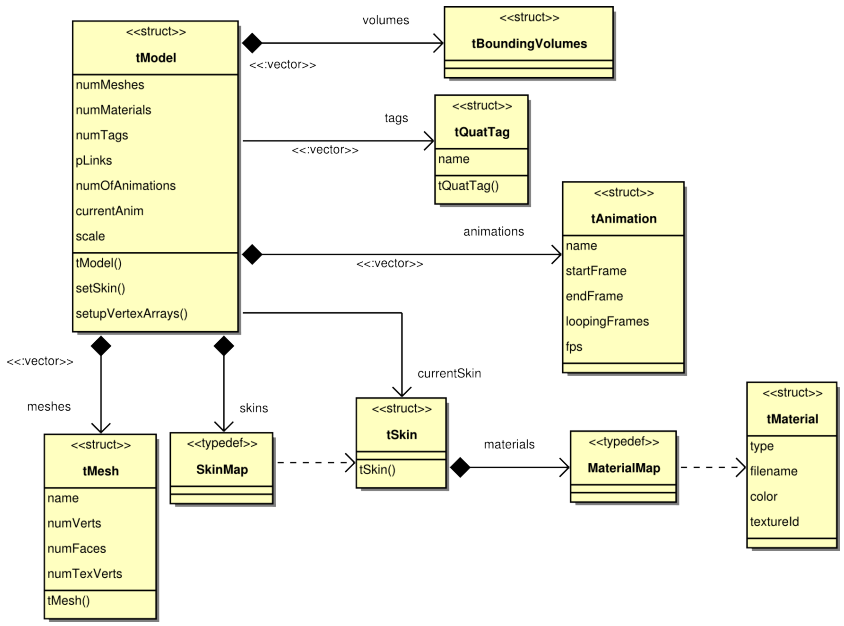

Figura 5.23: Estructura que define un modelo 3D.

Los atributos más importantes de un modelo son:

- **meshes:** Conjunto de todas las mallas que componen el modelo.
- **skins:** Cada modelo puede tener un conjunto de "pieles" (texturas), que se guardan en una tabla hash. Cada skin asigna un material (estructura `tMaterial`, que almacena la información de la textura) a una malla.
- **currentSkin:** Skin actual que se aplica al modelo.
- **animations:** Lista con las animaciones correspondientes al modelo. Al consistir en una animación basada en vértice, la estructura `tAnimation` se limita a almacenar los índices entre los que se encuentra la animación.
- **tags:** Los tags son una estructura (estructura `tQuatTag`) que permite almacenar información sobre la posición y rotación de un punto a lo largo de los diferentes frames de animación. Estos tags pueden ser usados para conectar diferentes modelos entre si o para establecer un punto especial en el modelo.

- **volumes:** La estructura `tBoundingVolumes` almacena tanto el cubo que envuelve el modelo en cada frame, como la esfera. Estas estructuras pueden ser usadas para la detección de colisiones entre modelos.

5.5.6. Representación de texto

La transmisión de información a un jugador quedaría seriamente comprometida sin la capacidad para representar texto. Es por ello que en **YAOMEV** se experimentó con diferentes aproximaciones a la representación gráfica de texto mediante **OpenGL**. En un principio se implementó usando las capacidades ofrecidas por **SDL_ttf**, sin embargo, esa opción no permitía la posibilidad de usar el texto en un entorno 3D.

Al final, se optó por una solución mucho más completa, el uso de una biblioteca externa que ya hubiera resuelto el problema. Es por ello que la representación de texto corre a cargo de la biblioteca **FTGL**, la cual permite de forma muy sencilla el uso de fuentes *TrueType* y su representación 3D mediante **OpenGL**, siendo además multiplataforma y con una licencia libre.

Para simplificar su uso en **YAOMEV**, se han creado dos *wrappers* que abstraen y ocultan la implementación propia de **FTGL**, dejando sólo aquellas funcionalidades necesarias para este proyecto. En la figura 5.24 se pueden ver las clases involucradas en la representación de texto.

Existen dos clases creadas para envolver **FTGL**: `COglText` y `COglTextLayout`. Cada una de las cuales ofrece sólo la funcionalidad requerida para representar etiquetas de texto y texto multilinea respectivamente.

Es obligatorio para las dos usar la clase `FTFont`, propia de **FTGL**. Esta clase representa una fuente y ofrece operaciones para manipularla. Como las fuentes son un recurso muy usado y costoso en memoria, se ha creado también una caché de fuentes `CFontsCache` usando el patrón *Proxy* (sección 6.7), que hace transparente al programador tanto la creación de las fuentes, como el acceso a las mismas evitando que existan duplicados de la misma fuente en memoria.

La clase `CFontsCache` funciona de la siguiente forma: Se inicia indicándole la ruta en la

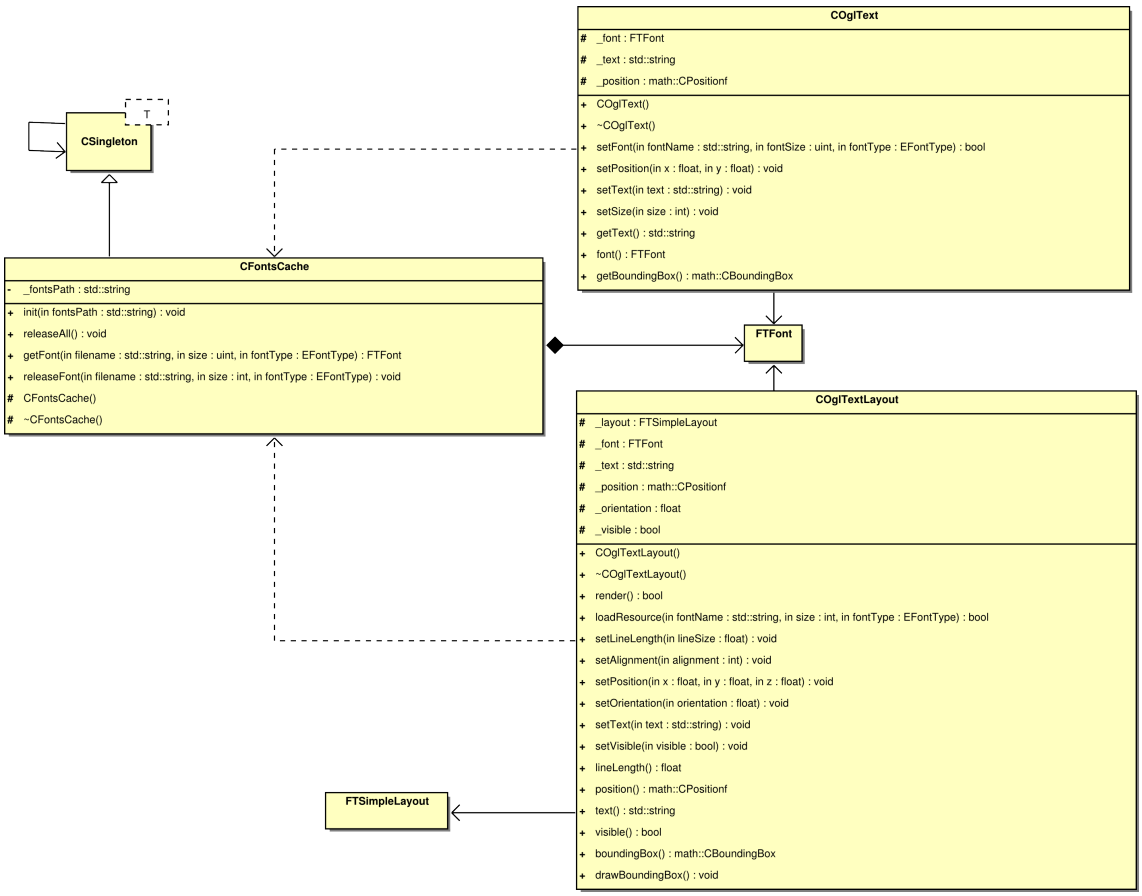

Figura 5.24: Clases involucradas en el uso de la biblioteca **FTGL** para representar texto.

que puede encontrar los archivos .ttf con las fuentes, a partir de ahí mediante el uso del método `getFont(...)`, pasándole como atributos el nombre del archivo con la fuente, el tamaño y el tipo de fuente; creará un objeto `FTFont` nuevo si es que no existe uno previamente creado con los mismos atributos; si este existe devolverá la instancia ya creada, y si no existe, creará el objeto y lo devolverá.

Por último, la clase `COglTextLayout` usa la clase `FTSimpleLayout`, propia también de **FTGL** para adaptar su funcionalidad a las necesidades de **YAOMEV**.

5.5.7. Sistema de renderizado

En **YAOMEV** se ha delegado la tarea de dibujar los modelos gráficos a las clases `CModelRenderer` y `COglInmediateRender`.

La primera de ellas se encarga del renderizado tanto de modelo estáticos, como de los modelos animados. Para el dibujado, utiliza vertex arrays, mucho más eficientes para dibujar modelos complejos que el modo inmediato de **OpenGL**. Es también en este punto en el que se realiza una interpolación en función del tiempo de los vértices y normales, entre su posición en el frame actual y su posición en el frame siguiente (ver listado de código 5.7).

Listing 5.7: Interpolación y renderizado mediante vertex arrays

```
for(int i = 0; i < pModel->numMeshes; i++) {
        const tMesh* pMesh = &pModel->meshes[i];

        // Recupero el frame actual y el siguiente para interpolarlos
        int currentIndex = currentFrame * pMesh->numVerts;
        int nextIndex = nextFrame * pMesh->numVerts;

        // Interpolo los vertices y normales para encontrar
        // el punto intermedio en el que se encuentran entre frames
        for(int v=0; v<pMesh->numVerts; ++v) {
                coord1 = pMesh->pVerts[currentIndex + v];
                coord2 = pMesh->pVerts[nextIndex + v];

                normal1 = pMesh->pNormals[currentIndex + v];
                normal2 = pMesh->pNormals[nextIndex + v];

                interpCoord1 = math::Lerp(coord1, coord2, interp);
                interpCoord1 *= scale;

                interpNormal1 = math::Lerp(normal1, normal2, interp);

                pMesh->drawVertArray[v] = interpCoord1;
                pMesh->drawNormArray[v] = interpNormal1;
        }

        // Enviamos los arrays a OpenGL
        glVertexPointer         (3, GL_FLOAT, 0, pMesh->drawVertArray);
        glNormalPointer         (GL_FLOAT, 0, pMesh->drawNormArray);
        glTexCoordPointer       (2, GL_FLOAT, 0, pMesh->pTexVerts);

        // Enlazamos la textura a la malla a dibujar
        if(pModel->currentSkin)
                glBindTexture(  GL_TEXTURE_2D,
                                pModel->currentSkin->materials[pMesh->name].textureId
                        );

        // Manda dibujar los triangulos
        glDrawElements( GL_TRIANGLES,
                        pMesh->numFaces * 3,
                        GL_UNSIGNED_INT,
                        pMesh->pFaces);
}
```

La clase `COglInmediateRender` se usa para aquellos objetos demasiado simples como para justificar las llamadas a `glDrawElements`. En **YAOMEV** se utiliza para renderizar básicamente rectángulos con texturas que representan los sprites usados para crear las interfaces de usuario (ver listado de código 5.8).

Listing 5.8: Renderizado mediante método inmediato de sprites

```
bool render(COglSprite* gfxSprite,  const math::CVector3f& position, float width, float
    height){
        TRGBAColor color = gfxSprite->getColor();
        uint textureID = gfxSprite->getTextureID();

        // Si no es transparente del todo dibujamos
        if(color.a>0){
                glPushMatrix();
                        // Habilitamos transparencia y el modo para texturas 2D
                        glEnable(GL_BLEND);
                        // Aplicamos transparencia
                        glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

                        // Movemos
                        glTranslatef(position.x,position.y,position.z);

                        // Establecemos color con canal alpha
                        glColor4f(color.r, color.g, color.b, color.a);

                        // Asociamos textura
                        glBindTexture(GL_TEXTURE_2D, textureID);

                        // Dibujamos el rectangulo
                        glBegin(GL_TRIANGLE_STRIP);
                                glTexCoord2f(0,0);
                                glVertex3f(0,0,0);              // V0

                                glTexCoord2f(1,0);
                                glVertex3f(width,0,0);          // V2

                                glTexCoord2f(0,1);
                                glVertex3f(0,height,0);         // V1

                                glTexCoord2f(1,1);
                                glVertex3f(width,height,0);     // V3
                        glEnd();

                        glColor4f(1.0f,1.0f,1.0f,1.0f);

                        // Deshabilitamos la transparencia y el modo de texturas 2D
                        glDisable(GL_BLEND);

                glPopMatrix();
        }

        return true;
```

```
46 }
```

5.5.8. Sistema de vídeo

Las capacidades multimedia en un videojuego son importantes puesto que aumentan la capacidad de inmersión en el mismo y proporcionan una mayor versatilidad a la hora de plantear mecanismos de juego. En **YAOMEV** se ha tomado cuidado para poder usar vídeo en los juegos creados de forma sencilla e intuitiva.

Para conseguir utilizar vídeo en un juego se ha procedido de la siguiente manera. Para poder tratar los vídeos se necesita de una biblioteca que permita tratar con ellos, por lo tanto se eligió una sencilla (en concreto para este proyecto se ha usado la biblioteca **smpeg**) con la que se pueden leer vídeos y cargarlos en memoria para su reproducción. Se crea entonces un *wrapper* que envuelva su funcionalidad adaptándola a nuestras necesidades. En particular, en este proyecto se ha encarga simplemente de proporcionar métodos para reproducir, parar o pausar el vídeo, manejar el volumen y acceder a la información de cada frame del mismo. Esta última funcionalidad es la más importante, puesto que el *wrapper* se encarga de mantener la información de cada frame en memoria y además convertir esto en una textura de **OpenGL** que se pueda usar en el juego. Para ello se reserva un espacio fijo de memoria, que queda reservado como espacio para una textura **OpenGL**. En cada frame, se rellena este espacio con la información del siguiente frame, de esta manera a partir del identificador típico de una textura en **OpenGL** se puede usar un vídeo como textura.

En la figura 5.25 se muestra la estructura del sistema de vídeo creado en **YAOMEV**, que permite hacer lo anteriormente descrito. La clase `gui::CSmpegMovie` cumple las funciones de *wrapper* anteriormente descritas. Cómo la biblioteca **smpeg** está limitada al uso de archivos de vídeo en formato **MPEG-1**, y no se quiere dejar cerrada la plataforma a otros formato de vídeo, se ha usado el patrón *Proxy* (sección 6.7), junto con el patrón *Singleton* (sección 6.3), para crear una caché de vídeos similar a las cachés de recursos anteriormente descritas (sección 5.5.1), accesible desde cualquier lugar, que aisla al programador del tipo de **wrapper** que se use para crear la textura de vídeo. De esta manera se pueden crear diferentes clases, que implementen la interfaz `gui::IVideo`, que usen diferentes bibliotecas

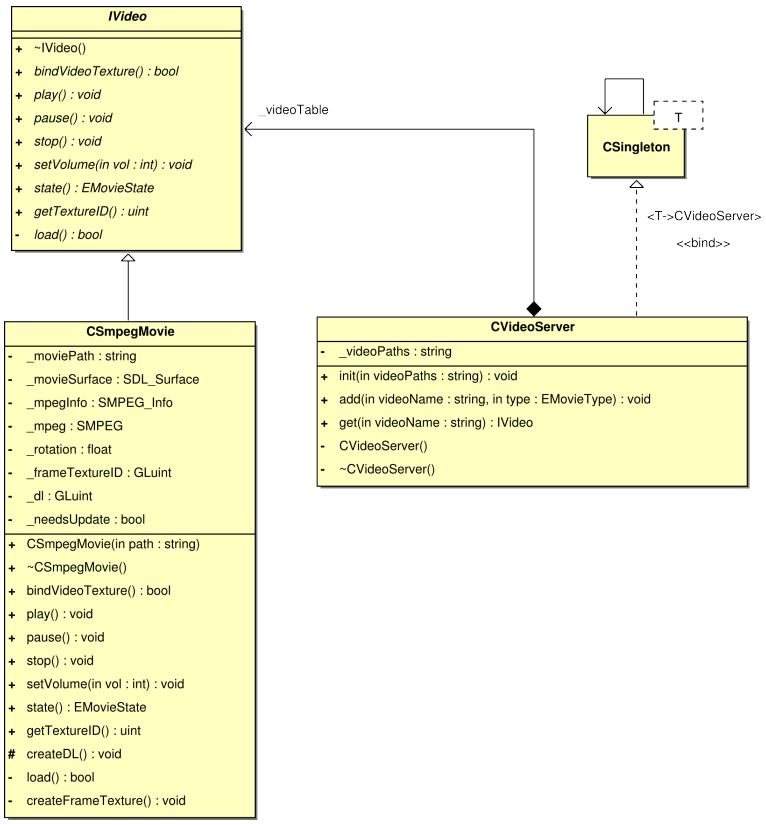

Figura 5.25: Clases que componen el sistema de vídeo.

de reproducción de vídeo. Esta caché de vídeo se ha denominado `gui::CVideoServer`

En su uso en el juego, hay un nodo de escena especial, llamado `gui::CScreenNode` como se aprecia en las figuras 5.16 y 5.26. Este nodo usa dos recursos, un modelo tridimensional estático representado por la clase `gui::CStaticModelMd3` (sección 5.5.5) y otro representado por la clase `gui::IVideo`. Su uso conjunto da lugar a la posibilidad de usar un vídeo como textura como se puede ver en el código mostrado en 5.9.

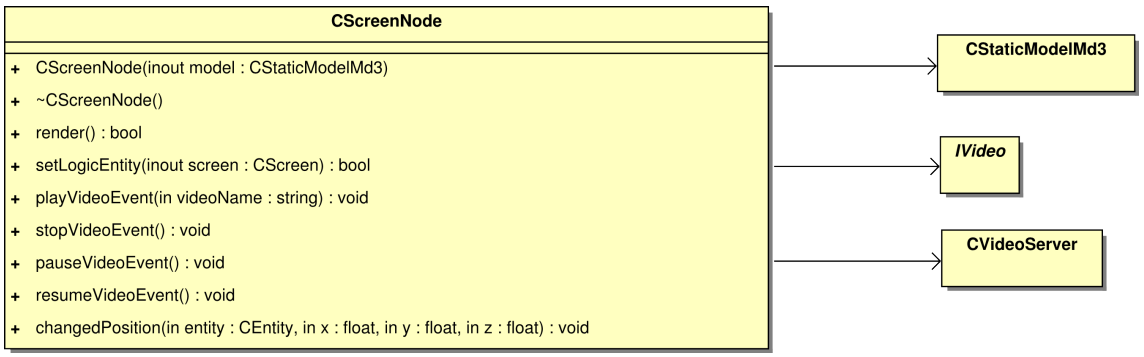

Figura 5.26: Atributos principales de la clase `CScreenNode`.

Listing 5.9: Uso de vídeo como textura de un modelo 3D

```
1  bool CScreenNode::render(){
2          [...]
3          if((_video->state() == IVideo::MOVIE_PLAYING || _video->state() == IVideo::
               MOVIE_STOPPED))
4          {
5                  _video->bindVideoTexture();
6          }
7          [...]
8          CStaticEntityNode::render();
9          [...]
10         return true;
11 }
```

Esta aproximación favorece un uso sencillo y transparente al programador de los vídeos en un juego 3D o 2D. Ofreciendo además la posibilidad de implementar nuevos *wrappers* sin excesivas modificaciones de código.

La decisión final de usar la biblioteca **smpeg** se debe a que al pertenecer a **SDL** es multiplataforma y orientado a la programación de videojuegos.

5.5.9. Sistema de audio

El sistema de audio presenta problemas parecidos a los que se encuentran en la creación del sistema de vídeo (sección 5.5.8). Por lo tanto las decisiones de diseño han sido parecidas, excluyendo la creación de *wrappers* para el manejo de los distintos tipos de audio.

Sus funcionalidades básicas vienen dadas por la necesidad de almacenar y recuperar música; aceptar el uso de efectos sonoros y canciones; permitir su manipulación de forma sencilla

e intuitiva.

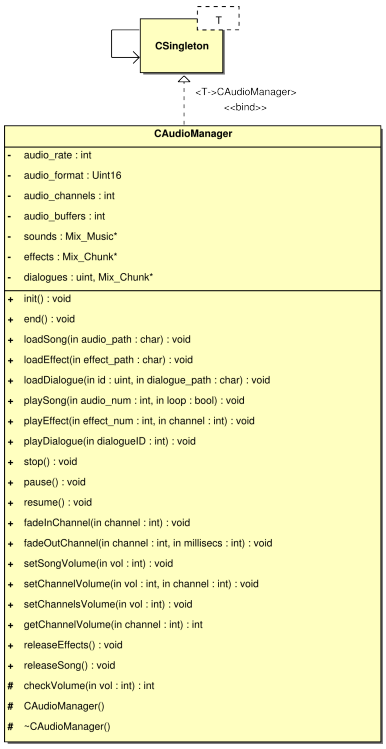

Figura 5.27: Diagrama de la clase `CAudioManager`.

Para poder usar efectos sonoros y canciones, se ha creado un gestor de audio `CAudioManager` (figura 5.27) que actúa de caché de audio y de reproductor. Cómo en los gestores de recursos y en el sistema de vídeo (secciones 5.5.8 y 5.5.1) se han usado los patrones de diseño *Proxy* (sección 6.7) y *Singleton* (sección 6.3). Este gestor utiliza la biblioteca **SDL_mixer** para la gestión del audio y de forma global envuelve su funcionalidad adaptándola a las necesidades de la plataforma **YAOMEV**.

La elección de **SDL_mixer** se debe a sus características como biblioteca multiplataforma orientada además a su uso en la programación de videojuegos. La biblioteca ofrece una separación entre efectos sonoros (o samples de audio) y la música, facilitando con ello su gestión diferenciada. En el caso de efectos sonoros admite formatos como: WAVE, AIFF, RIFF, OGG y VOC. Mientras que para música acepta: WAVE, MOD, MIDI y MP3 mediante *streaming* de forma transparente para el programador.

Como ventajas en su implementación, al permitir acceso global como *Singleton* su uso

está permitido desde cualquier lugar del código y al estar construido como un *Proxy*, mediante las funciones de carga, se almacena la música a usar en memoria y ofrece un acceso a ellas mediante un identificador de forma muy sencilla. Finalmente se ofrecen una serie de funciones que permiten su uso como si de un reproductor de audio normal se tratara.

5.5.10. Gestión de ventana

Uno de los primeros pasos que se da en la programación gráfica es la creación y gestión del entorno en el que se mostrarán los elementos a dibujar. **OpenGL** no está pensado para manejar un sistema de ventanas, por lo que es necesario usar una biblioteca que nos otorgue esa funcionalidad y que además sea multiplataforma.

Este ha sido el principal motivo por el que se usa **SDL** en este proyecto. Esta biblioteca cumple con todos los requisitos necesarios como pueden ser: ser multiplataforma, proporcionar un gestor de ventanas que nos permite ejecutar **OpenGL** en ellas y un sistema de eventos para detectar la interacción entre usuario y ventana.

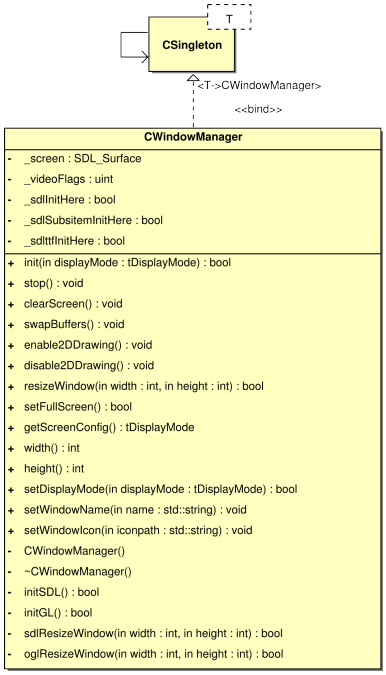

Figura 5.28: Diagrama de la clase `CWindowManager`.

La clase encargada de agrupar la funcionalidad del gestor de ventanas es `CWindowMa-`

`nager`. Esta clase ofrece la funcionalidad necesaria para gestionar la ventana en que transcurre el juego, aunque no se limita únicamente al manejo de la ventana, si no que además se encarga de iniciar tanto **SDL**, como **OpenGL**. Además suministra funciones para la limpieza del buffer de vídeo y para el intercambio de buffers, al usarse doble buffer. Incluye también funciones para un modo de perspectiva ortográfico (o 2D) y por supuesto las funciones necesarias para lidiar con la redimensión de la ventana y la pantalla completa.

Esta clase se ha implementado como *Singleton* (sección 6.3) para permitir un acceso global a ella en diferentes lugares del código.

5.5.11. Gestión de eventos de usuario

En **YAOMEV** se ha tenido en cuenta, por supuesto, la importancia de comunicar la entrada del usuario de una forma sencilla a los objetos que participan en el juego. Según el esquema del patrón **MVC** que se puede ver en la figura 5.2, en la sección 5.1, la vista debe comunicar la entrada de usuario mediante eventos, los cuales actualizan el estado de los elementos de la capa de modelo (sección 5.6).

La gestión de eventos se realiza envolviendo la gestión de los mismos, propia de **SDL** y de estilo imperativo, para convertirla en una gestión de eventos propia de **YAOMEV** orientada a objetos.

Para cumplir esta misión se usa la clase `CInputManager`, que convierte los eventos detectados por **SDL** en eventos de **YAOMEV** (figura 5.32) y que, además, transmite mediante paso de mensajes a los *Listeners* (sección 6.8) que tenga registrados, es decir, a las clases que hayan implementado estos interfaces (figura 5.31).

En la figura 5.29 podemos ver el diagrama de clases correspondiente a `CInputManager`, donde se ve además la relación de agregación que tiene con los diferentes *Listener* de **YAOMEV**.

De esta tarea de conversión se encarga el método `void CInputManager::processUserInput()`. Este método procesa los eventos del clásico bucle de recogida de eventos de **SDL** y en función del tipo de evento, crea un evento de **YAOMEV** y se lo transmite a los listeners que haya (figura 5.30).

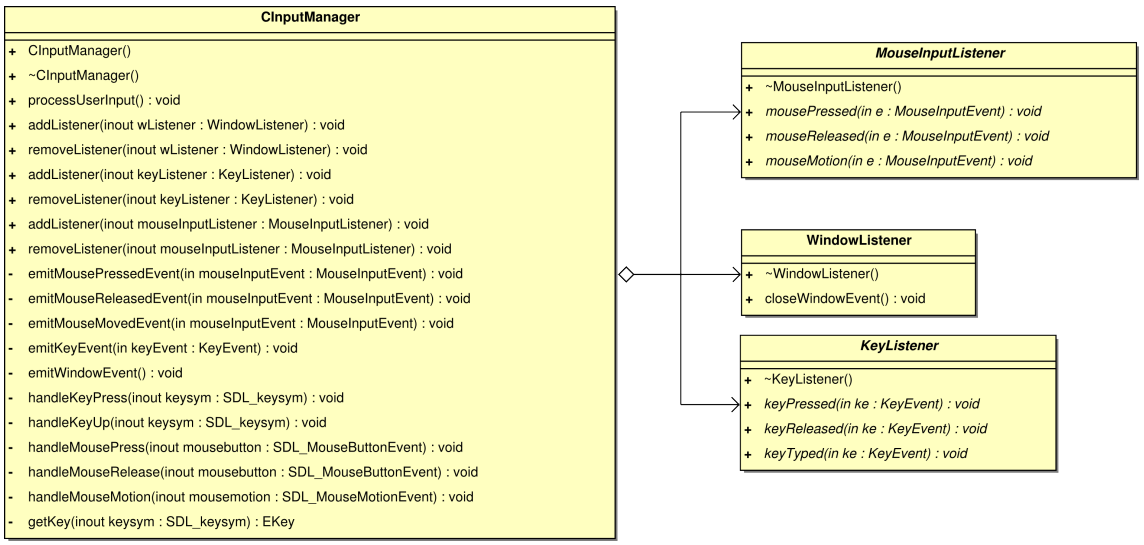

Figura 5.29: Diagrama de clases de `CInputManager` con sus atributos más representativos.

Listing 5.10: Método `processUserInput`.

```
1  void CInputManager::processUserInput(){
2          SDL_Event event;
3          while ( SDL_PollEvent(&event) ){
4                  switch(event.type){
5                  case SDL_KEYUP:                 // Soltamos una tecla
6                          handleKeyUp(&event.key.keysym);
7                          break;
8
9                  case SDL_KEYDOWN:               // Soltamos una tecla
10                         handleKeyPress(&event.key.keysym);
11                         break;
12
13                 case SDL_MOUSEMOTION:           // Movimiento del raton
14                         handleMouseMotion(&event.motion);
15                         break;
16
17                 case SDL_MOUSEBUTTONUP:         // Levantamos el boton del raton
18                         handleMouseRelease(&event.button);
19                         break;
20
21                 case SDL_MOUSEBUTTONDOWN:       // Presionamos el boton del raton
22                         handleMousePress(&event.button);
23                         break;
24
25                 case SDL_QUIT:                  // Cerramos la aplicacion
26                         emitWindowEvent();
27                         break;
28
29                 default:
30                         break;
31                 }
32         }
```

```
33 }
```

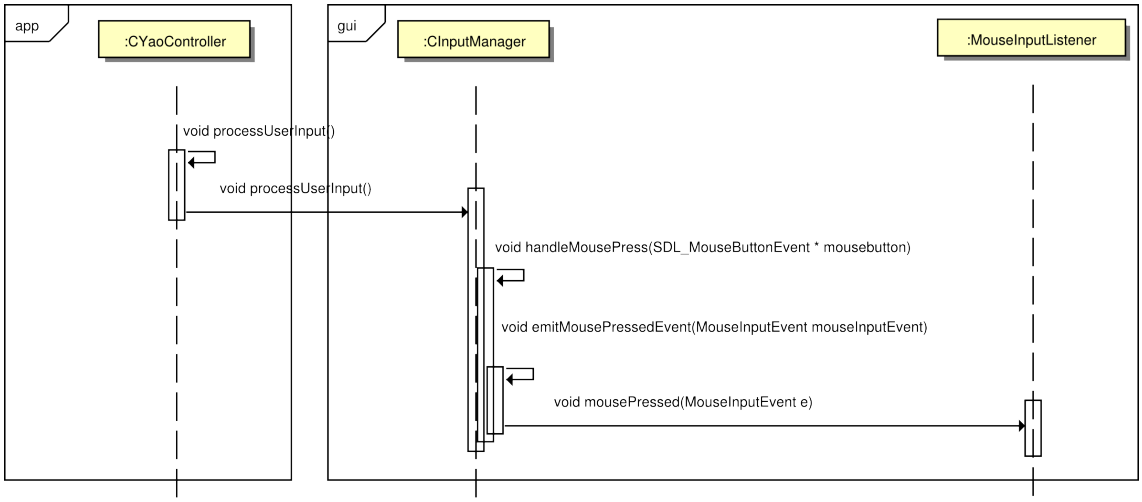

Figura 5.30: Diagrama de secuencia que muestra cómo es procesada la entrada de usuario hasta que se le envía un mensaje al *Listener* adecuado.

Esta clase se usa en la capa de controladores (sección 5.3), en la etapa del bucle principal que se encarga de recopilar la información introducida por el usuario. Es un atributo de la clase `CYaoController` (sección 5.3.1), por lo que todos los controladores que implementen `CYaoController` podrán implementar los *Listeners* necesarios y recibir los eventos generados por el usuario. Para ello, previamente, en la creación del controlador se debe incluir el mismo como escuchador de aquellos eventos en los que esté interesado.

En resumen, `CInputManager` traduce los eventos a partir de **SDL** y envía los eventos equivalentes de **YAOMEV** a los *Listeners* que se hayan registrado. En **YAOMEV** hay cuatro tipos de *Listeners* (figura 5.31).

- **MouseInputListener:** Es un escuchador que esta atento a la entrada de usuario mediante ratón. Este interfaz es implementado por la clase CFrame para averiguar si los componentes del interfaz son afectados por el puntero del ratón o no.

- **MouseListener:** Este escuchador sin embargo esta atento a los eventos ocurridos por la interacción del puntero del ratón con un componente de la interfaz gráfica y se utiliza para transmitir esos eventos a las clases interesadas.

- **KeyListener:** Sirven para transmitir eventos relativos a la entrada de usuario por medio del teclado.
- **WindowListener:** Están atentos a aquellos eventos que tengan que ver con la ventana en que se ejecuta el juego (sección 5.5.10).

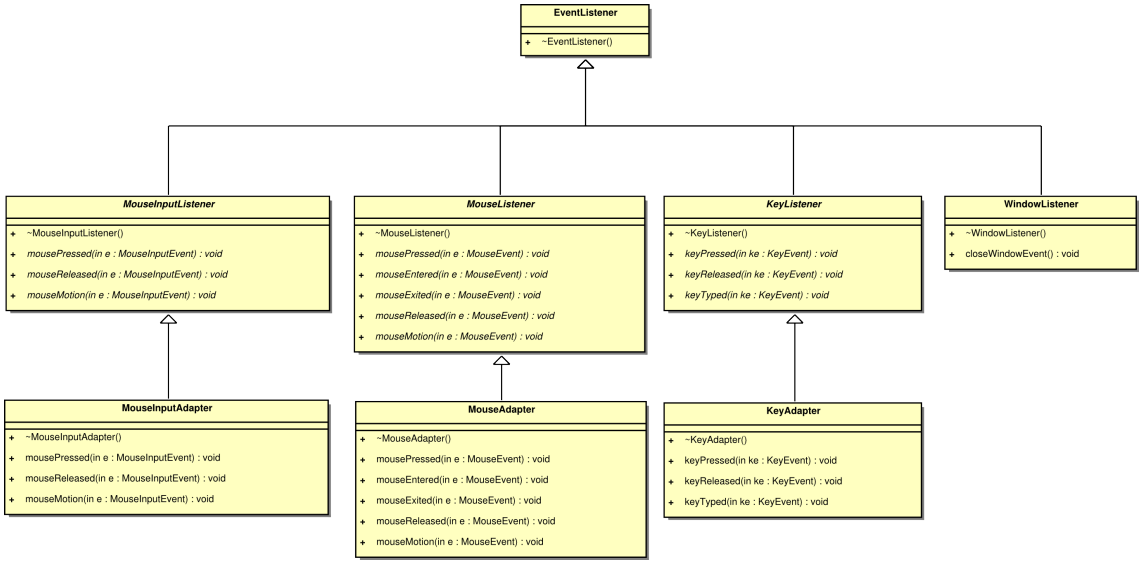

Figura 5.31: Jerarquía de *Listeners*.

Una vez tratados los *Listeners*, encargados de recibir los eventos generados por la interacción del usuario con la interfaz gráfica, es hora de comentar brevemente los eventos de **YAOMEV**. Los eventos son mensajes mandados entre las clases para comunicar información sobre la entrada de usuario. Para una mejor comprensión y utilización de los eventos, se ha creado una pequeña jerarquía (figura 5.32). Existen tres tipos de eventos:

- **ComponentEvent:** Con su especialización `MouseEvent` se encarga de transmitir información sobre un evento ocurrido en un componente de la interfaz de usuario.
- **MouseInputEvent:** Se encarga de transmitir información sobre un evento de ratón ocurrido en la pantalla del juego.
- **KeyEventEvent:** Almacena la información sobre un evento de teclado: tecla que genera el evento, tipo de evento, etc...

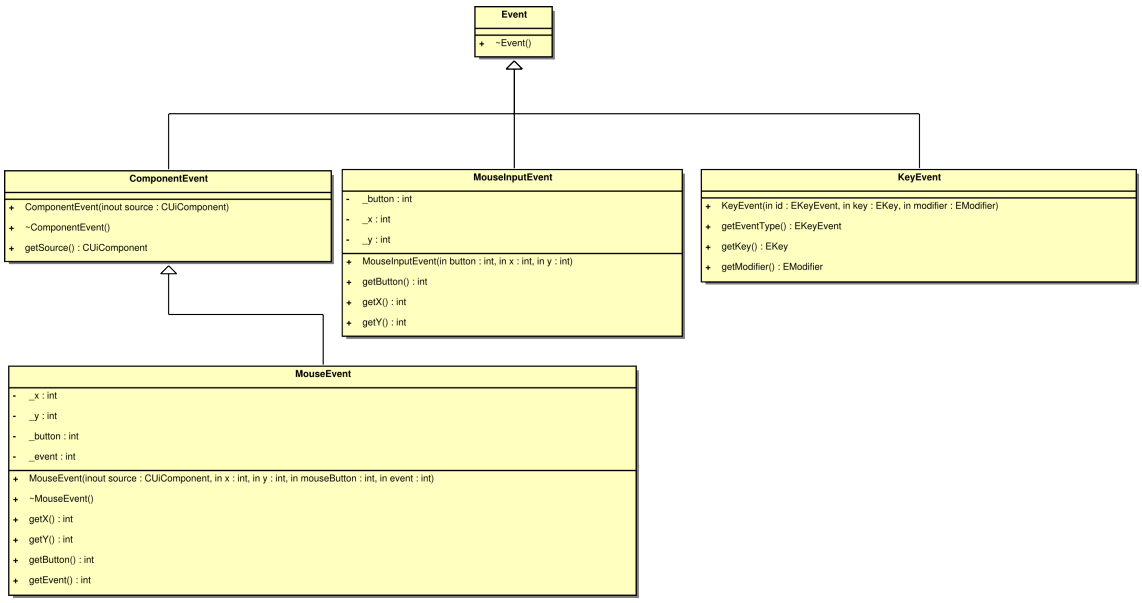

Figura 5.32: Jerarquía de eventos.

Cómo se puede ver en la figura 5.30, la clase `MouseInputListener` es la que recibe los eventos de ratón provenientes de la interacción del usuario, con la pantalla donde se desarrolla el juego, mediante el uso del ratón. Este *Listener* es implementado por la clase `logic::CFrame` (la cual será explicada en la sección 5.6), que es el componente de interfaz de usuario más importante. Por lo tanto `logic::CFrame` debe encargarse de procesar la información del evento codigoMouseInputEvent para comprobar si afecta de algún modo a los componentes que contiene (figura 5.33). En caso de que alguno sea afectado, creará el evento correspondiente y se lo enviará a los *Listeners* que tenga registrados.

Esta adaptación del sistema de eventos de **SDL** a uno propio en **YAOMEV** se debe a que de esta forma su uso es mucho más sencillo, ocupa menos código y es más transparente al programador. Su uso es tan simple como implementar en el controlador deseado los métodos correspondientes a los escuchadores que nos permiten manejar la entrada de usuario, registrar el controlador como *Listener* y a partir de ahí usar los eventos recibidos para actuar en consecuencia.

En el siguiente fragmento de código podemos comprobar como se detecta un evento de ratón sobre un botón. Tan sencillo como implementar el método `mouseReleased` del *Listener* `MouseAdapter` y usar el evento recibido, de tipo `MouseEvent`, para averiguar sobre

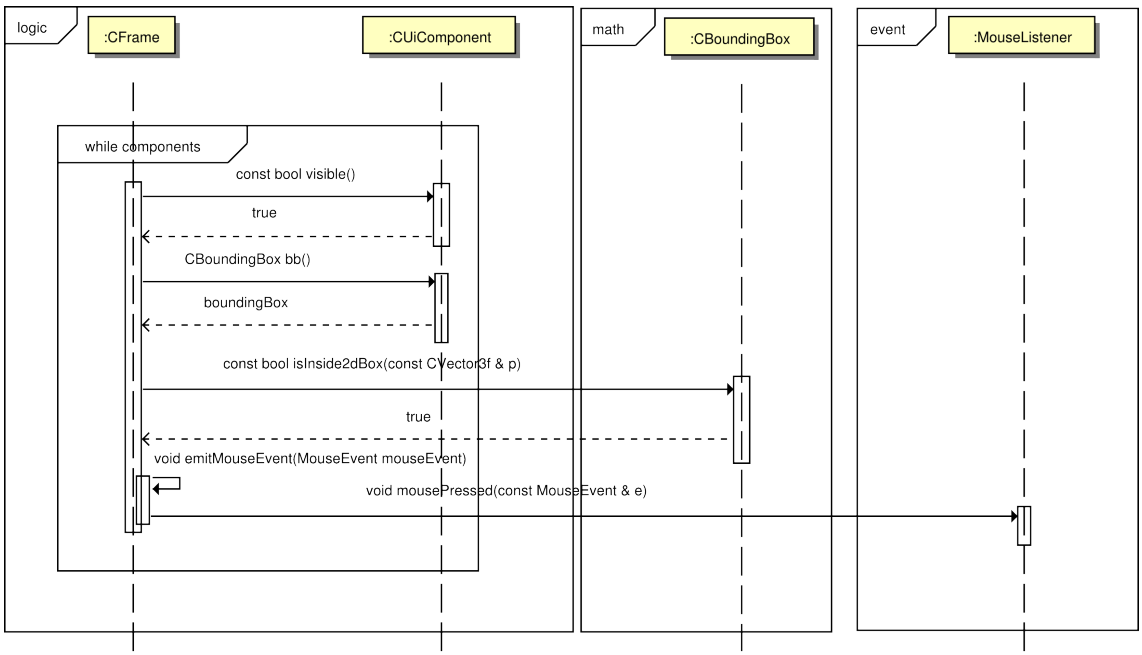

Figura 5.33: Diagrama de secuencia que muestra un ejemplo de transmisión de un evento de ratón hasta el `Listener` adecuado.

qué componente se ha realizado la acción y a partir de ahí, actuar en consecuencia. En este ejemplo se ve qué ocurre al soltar un botón del ratón sobre los botones `_endAppBtn` y `_initGameBtn`.

Listing 5.11: Ejemplo de tratamiento de eventos producidos por el ratón en la clase `CPrincipalMenuSubstate`

```
1  void CPrincipalMenuSubstate::mouseReleased(const event::MouseEvent& me){
2          // Recuperamos el componente de la interfaz sobre el que ha ocurrido el evento
3          const logic::CUiComponent* source = me.getSource();
4
5          // Se comprueba cual es el componente de la interfaz que ha recibo el evento
6          if(source == _endAppBtn){
7                  // Se solicita terminacion de la aplicacion
8                  _app->askForTermination();
9          }
10         else if(source == _initGameBtn){
11                 // Se establece el escenario para jugar
12                 _logic->setScenario(_backgroundScenario);
13
14                 // Si no se han elegido jugadores, se seleccionan al azar
15                 if(!_parentState->arePlayersSelected())
16                         setRandomPlayers();
17
18                 // Damos paso al estado "juego"
19                 _app->setState("juego");
```

```
20              }
21              [...]
22      }
```

5.6. Sistema de juego

Este sistema se corresponde con la capa de modelo en el patrón de arquitectura del software **MVC**, la cual se encarga de definir las reglas del juego y de mantener el estado interno de la partida. En ella es donde se intenta plasmar el diseño del juego definido en el documento de concepto de **Bit Them All!!!** (ver anexo A y artículo [13]) y que, gracias a **YAOMEV**, es la única capa que debe ser implementada por el desarrollador.

De forma resumida, la implementación de esta capa en **Bit Them All!!!** se comporta de la siguiente manera: como se explicó en la sección 5.1 y se ve en la figura 5.2, la capa de vista (sección 5.5) capta la información sobre la interacción del usuario con el juego, esta es transmitida a los controladores en forma de eventos (sección 5.5.11), para posteriormente comunicar esa información a la capa de modelo. La capa de modelo actualizará su estado en función de las reglas que definan su comportamiento y comunicará esos cambios de nuevo a la capa de vista (sección 5.6.3) o de controladores (sección 5.6.2).

Por lo tanto, el corazón del juego se programa en esta capa. La arquitectura **YAOMEV** facilita enórmemente el desarrollo, permitiéndonos olvidar aquellos aspectos no pertenecientes al dominio del problema a resolver (codificación de un juego) o a la entrada de usuario, puesto que es necesario construir controladores específicos (estos se analizaron en la sección 5.3.4).

La construcción de este sistema se ha enfocado también de forma modular. Está dividido en los siguientes componentes:

- **Fachada lógica:** Agrupa todos los subsistemas bajo un sólo sistema que actúa de interfaz con el resto de capas, ofreciendo su funcionalidad de forma controlada.
- **Subsistema de procesamiento de eventos y comandos:** Sistema que permite transmitir información entre esta capa y la capa de controladores.
- **Subsistema de gestión de entidades de juego:** Agrupa y controla todas las entidades

que participan en el juego divididas en dos jerarquías, las entidades y los componentes de interfaz.

- **Subsistema de gestión de elementos de juego:** Sirve para poder acceder a la lógica desde un único punto. Con ello se separa la lógica del juego del resto de capas. Controla todos los elementos participantes en el juego y proporciona tanto métodos de acceso a ellos, como métodos para modificar sus atributos.

- **Subsistema de gestión de comportamientos:** Agrupa y controla todos los comportamientos en estructuras que definen las reglas del juego.

Este sistema es el menos genérico de todos. La mayor parte de el se ha construido específicamente para el juego **Bit Them All!!!**, aunque algunos elementos son totalmente reutilizables como puede ser la jerarquía de entidades o los comportamientos creados para los árboles de comportamiento. A continuación se describen en detalle estos subsistemas junto con información sobre su implementacion.

5.6.1. Fachada lógica:

La capa de modelo involucra muchos subsistemas interrelacionados entre si que además deben comunicarse con otras capas (en el marco de **MVC**). Es por tanto necesaria una aproximación que limite en la medida de lo posible la dispersión de los sistemas de esta capa entre los sistemas del resto de capas. La mejor manera de conseguir esto agrupar todos estos subsistemas de la capa de modelo en un sólo sistema y restringir a nuestro gusto la forma en que el resto se comunican con ella. De esta forma se ofrecerá un interfaz unificado que será fácil de gestionar y de rastrear en el resto de capas.

Por lo tanto, a la hora de afrontar este problema, se ha optado por seguir el patrón de diseño *Façade* (sección 6.5). Consistente en construir una clase que agrupe todos los subsistemas de un sistema dado, con el objetivo de ofrecer una interfaz común, dando la apariencia de formar todos parte de un solo sistema más grande. Para entender este punto mejor, se presenta el diagrama 5.34.

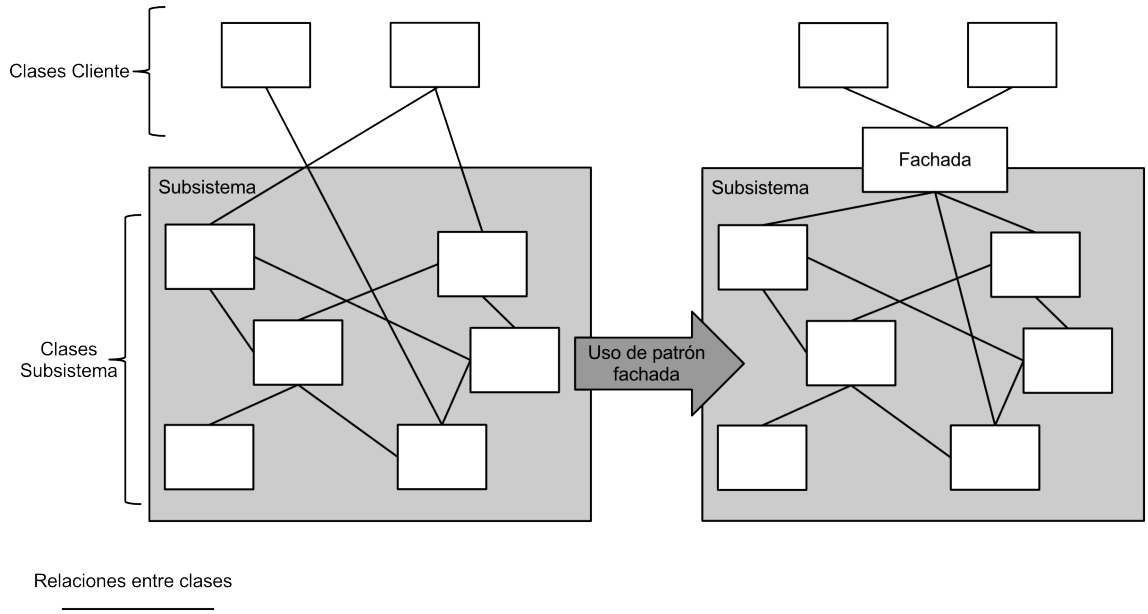

Figura 5.34: Transformación de un subsistema accedido por diferentes clases a un subsistema accedido a partir de una clase fachada.

En **Bit Them All!!!**, se ha optado por crear una interfaz llamada `ILogicFacade` y una clase especializada `CGame` que cumplen el rol de fachada lógica. En la figura 5.35 se muestran las dos clases con algunas de sus funciones más representativas.

En resumen, esta clase agrupa todas las entidades participantes del juego, ofreciendo al resto de capas un sólo punto desde el que acceder, o modificar, aquellos elementos de la lógica del juego que necesitan.

La forma en que se comunica esta capa con el resto ha quedado dividida en dos tipos: por un lado recibe información de la capa de controladores mediante un pequeño sistema de comandos (sección 5.6.2) y, por el otro, mediante un conjunto de *Listeners* (sección 6.8) que son descritos a continuación.

La clase `CGame` mantiene dos tipos de *Listeners*: uno que desacopla la creación y destrucción de entidades gráficas a partir de las lógicas y otro que desacopla la comunicación de eventos entre la capa de modelo y la de controladores. Estos son `CreationDestroyListener` que implementa la clase `CSceneManager` y otro `GameEventListener` que implementa el controlador específico del juego `CGameState` (figura 5.36). El primero de los dos se ha creado como interfaz para la creación de una lógica desacoplada de la capa de vista, mientras

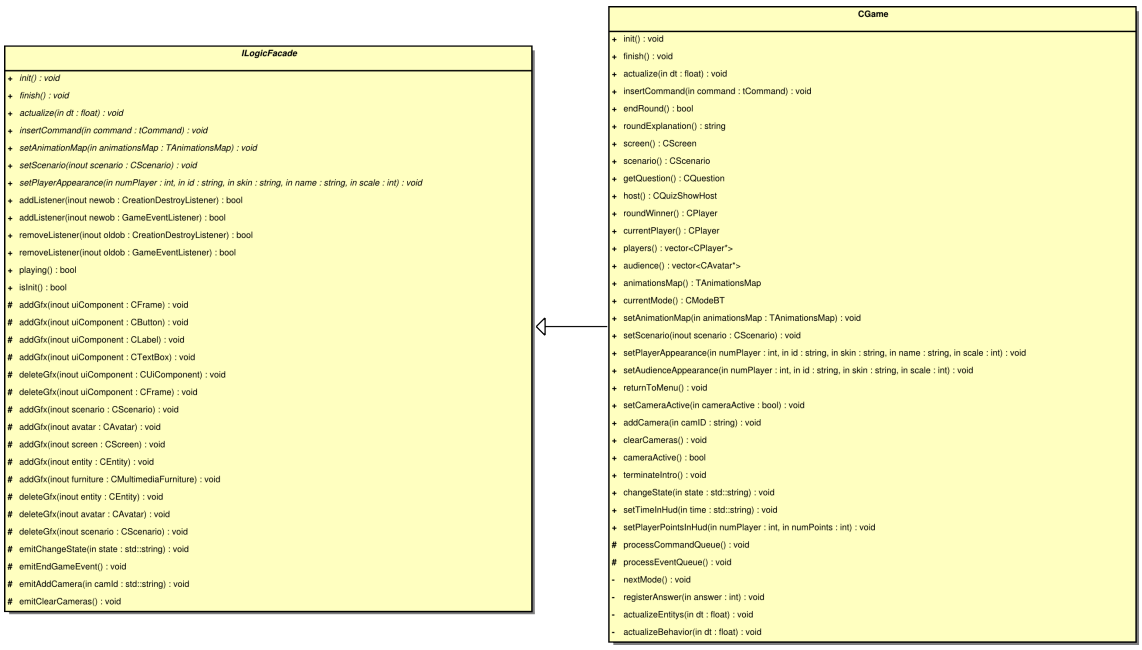

Figura 5.35: Diagrama de clases, parcial, de la interfaz `ILogicFacade` y de su especializacion `CGame`

que el segundo es específico del juego **Bit Them All!!!**.

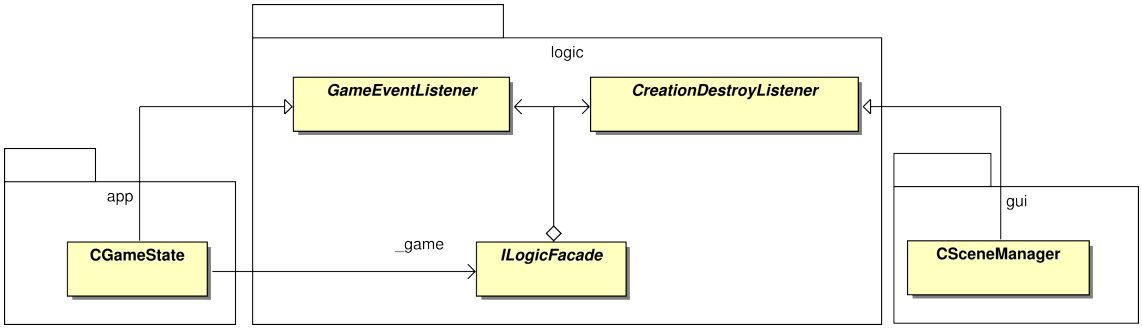

Figura 5.36: Diagrama de clases mostrando la relación entre la fachada y sus *Listeners*

Por otro lado, es necesario ajustar correctamente los métodos públicos de la fachada para permitir su uso desde otras capas. Ajustándonos al patrón **MVC**, los controladores son los encargados de la transmisión de información a esta capa, tal y como queda indicado en la figura 5.2 de la sección 5.1.

Las principales funcionalidades que queremos que ofrezca este sistema son aquellas que permitan el arranque, parada y ejecución del juego, al igual que se hizo con la clase encargada

de la ejecución de la aplicación (sección 5.2). De esta forma se tendrá control sobre el ciclo de vida de una forma sencilla y comprensible. Estos métodos se resumen a continuación:

- **void init():** Inicia todos los atributos de los elementos que participan en el juego dejandolos preparados para comenzar la partida y comunica al *Listener* de creación/destrucción de entidades gráficas que comience su representación.
- **void finish():** Destruye todos los objetos iniciados en el método anterior y los elimina de la capa de vista.
- **void actualize(float dt):** Este método difunde la delta de tiempo transmitida por el bucle principal a todas las entidades del juego para que puedan actualizar su estado en consecuencia (ver listado de código 5.13).

La fachada ofrece también una serie métodos para modificar que permiten modificar la estética de las partidas como son: `setAnimationMap`, `setPlayersAppearance` o `setScenario`, que permiten agregar un mapa de animaciones para cada tipo de personaje que haya y modificar tanto la apariencia de los jugadores como el escenario en el que se desarrolla la partida.

Como se aprecia en la figura 5.35, la clase `CGame` ofrece muchos más métodos para gestionar el transcurso del juego, esto es porque es donde realmente se agrupan todos los subsistemas descritos a lo largo de toda esta sección. Hacer una mención detallada de cada uno de los métodos sería largo y tedioso, por lo que es recomendable echar un vistazo al código suministrado. Sin embargo el funcionamientos y relación de `CGame` con cada uno de los subsistemas de esta capa es explicado en detalle a lo largo de las siguientes subsecciones.

Esta clase es usada por los controladores específicos del juego **Bit Them All!!!**, de la forma en que se describe en la sección 5.3.4.

5.6.2. Subsistema de procesamiento de eventos y comandos

Atendiendo al principio de separación estricta entre las diferentes capas en **MVC** (sección 5.1), se hace necesaria encontrar una forma sencilla de comunicar la información que el usuario transmite a los controladores, mediante eventos (sección 5.5.11), hacia esta capa.

Esos eventos transmiten la interacción del usuario con el juego, por lo que la capa de modelo necesita actualizarse en función de esa interacción. Por supuesto, esa interacción provocará un cambio en el estado del juego y, previsiblemente, en algún momento se de alguna situación que necesite ser comunicada al resto del sistema (por ejemplo, si una acción del juego necesita un cambio de controlador o si el juego termina y es necesario apagar la aplicación).

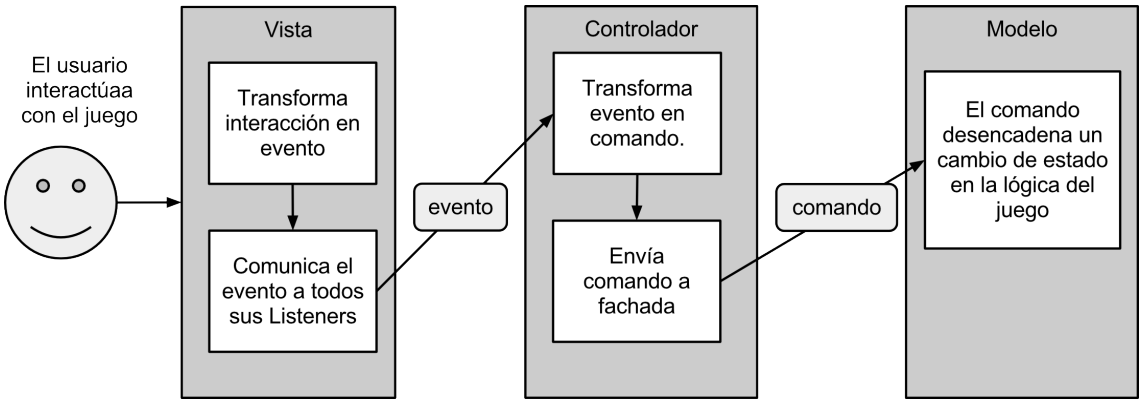

Figura 5.37: Representación de los pasos ocurridos entre una interacción del usuario y su comunicación a la fachada.

En **Bit Them All!!!** los eventos atendidos por los controladores son transformados en comandos. Estos son representados mediante una pequeña estructura llamada `tCommand`, que es procesada en la especialización de la fachada `CGame`. En la figura 5.37 se aprecia de forma simplificada el proceso de transformación de los eventos en comandos. De forma resumida lo que ocurre es lo siguiente: los controladores captan los eventos lanzados a causa de la interacción del usuario con el juego (sección 5.5.11), a partir de estos eventos los controladores crean unos comandos que transmiten a la lógica a través de la fachada. La clase especializada `CGame` procesa estos comandos en el método `processCommandQueue`, como se puede ver en el ejemplo de código del listado 5.12, y modifica la lógica en consecuencia.

La estructura `tCommand`, ha sido creada específicamente para el juego **Bit Them All!!!**. Por lo tanto sólo contienen dos atributos: uno que identifica el tipo de comando en base al enumerado `EIdCommand` y otro que indica cual de las intros del juego ha de activarse. De esta forma el procesado es sencillo, simplemente se recorre la lista de comandos y mediante un `switch` que discrimine aquellas ID que no se correspondan con elementos del enumerado `EIdCommand` se filtran los comandos.

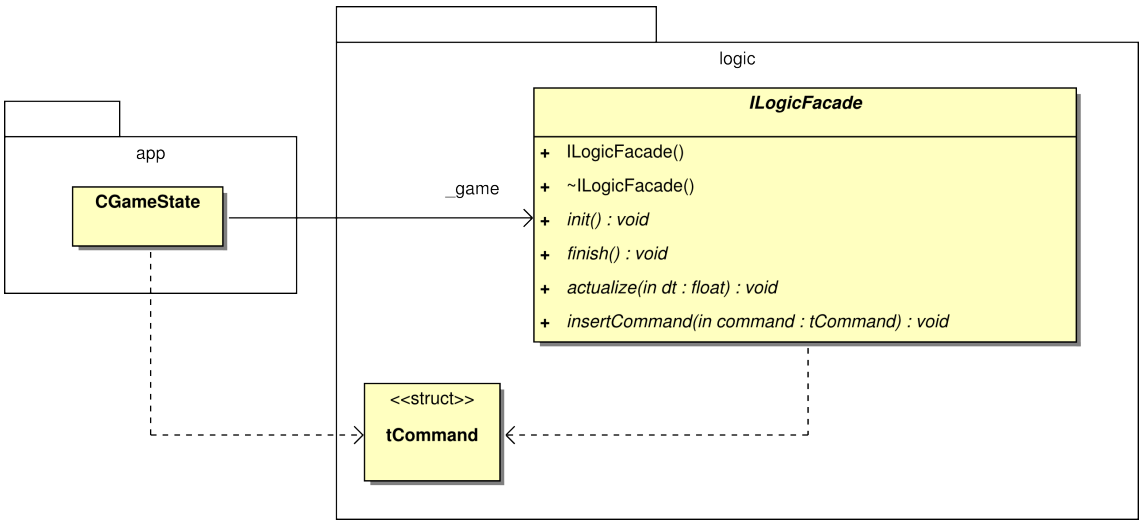

Figura 5.38: El controlador convierte los eventos en comandos y los envía a la fachada lógica.

Listing 5.12: Método de procesado de comandos (sólo se muestra el procesado de las respuestas a una pregunta)

```
void CGame::processCommandQueue(){
        TCommandIterator
                it(_commandQueue.begin()),
                end(_commandQueue.end());

        for(;it!=end; ++it){
                tCommand c = (*it);

                switch(c.id){
                [...]
                // Comunica la respuesta elegida
                case P1_A:
                        registerAnswer(3);
                        break;
                [...]
                }
        }

        _commandQueue.clear();
}
```

Al igual que con los comandos, se ha creado una estructura llamada `tEvent`, que representa los eventos ocurridos durante la actualización de la lógica y que han de ser comunicados al resto de capas. Su codificación ha sido prácticamente igual que la realizada para los comandos, aunque su procesado se realiza como última etapa del método de actualización (ver listado de código 5.13). Su único atributo es un enumerado que simboliza el tipo de evento

ocurrido, en el método `processEventQueue` se comprueba que tipo de evento es y se comunica el evento a los *Listener* `GameEventListener` que la clase `CGame` mantenga (sección 6.8).

En resumen, la clase `CGame` mantiene dos listas, una de comandos y otra de eventos (figura 5.39). Cada una de estas listas se procesa en momentos diferentes en el método de actualización. Primero se procesa la cola de comandos para conocer qué modificaciones han de ser realizadas sobre los elementos del juego, acto seguido se actualizan en base a esa información tanto las entidades como sus comportamientos y, por último, se procesan los eventos que puedan haber ocurrido durante la ejecución de este método y que necesitan ser comunicados a otras capas. El listado de código 5.13 muestra los pasos dados en la actualización de la capa de modelo.

Listing 5.13: Metodo de actualizacion de la lógica del juego

```
1  void CGame::actualize(float dt){
2          if(!playing() || !isInit()) return;
3
4          // 1 - Procesamos cola de comandos del jugador con turno
5          processCommandQueue();
6
7          // 2 - Actualizamos entidades
8          actualizeEntitys(dt);
9
10         // 3 - Actualizamos el comportamiento del juego
11         actualizeBehavior(dt);
12
13         // 4 - Procesamos cola de eventos
14         processEventQueue();
15 }
```

Esta aproximación de comandos y eventos ayuda a la independencia la lógica con el resto de capas, convirtiendo el uso de esta capa en algo sencillo y fácil de utilizar. Permite además un control del juego más adaptado a los requisitos del mismo, a la vez que proporciona una sencilla forma de agregar o quitar funcionalidad a **Bit Them All!!!**.

5.6.3. Gestión de entidades de juego

El juego **Bit Them All!!!** pretende simular un clásico concurso de televisión de preguntas y respuestas en base al documento de concepto presentado en el anexo (anexo A). Para lograr

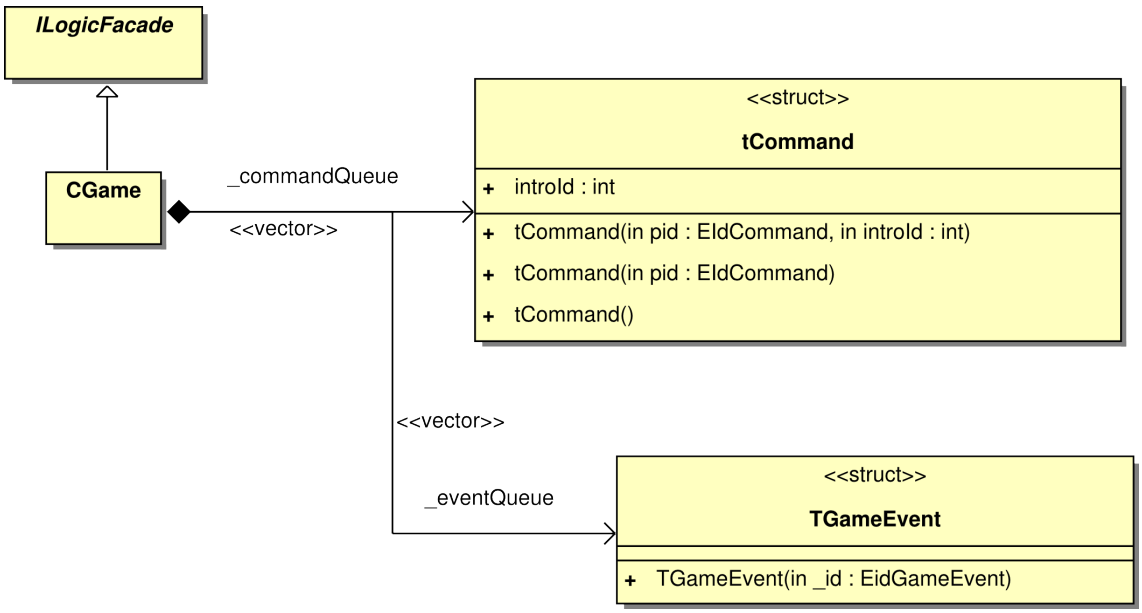

Figura 5.39: La especialización de la fachada mantiene dos listas, una de comandos y otra de eventos.

que esta simulación sea entretenida es necesario simular todo el entorno de un plató televisivo, incluyendo a los concursantes, público, presentador y decoración. Por lo tanto se ha de crear una representación interna de todos estos elementos que participan en el juego, permitiendo que respondan a la interacción con el usuario, reflejando sus cambios de estado en la capa de vista (sección 5.5) y además aportar una solución al problema de gestionar todos estos componentes de forma racional y simple.

Por tanto, todos estos elementos han quedado categorizados como entidades de juego. Las entidades de juego quedan definidas en **Bit Them All!!!** como aquellos elementos que pertenecen a la simulación del entorno virtual en el que transcurre la acción, es decir, aquellos elementos que forman parte de un concurso televisivo.

Para gestionar correctamente estos elementos, primero se clasificaron en una jerarquía de entidades (figura 5.40) y despues se creo un gestor de las mismas, llamado `logic::CScenario` (figura 5.43).

Como se comentó al final de la sección (sección 5.5.3), las entidades lógicas se corresponden con el estado extrínseco de los nodos gráficos. Para poder comunicar este estado a los nodos gráficos, cada uno de los componentes de la jerarquía implementa el patrón

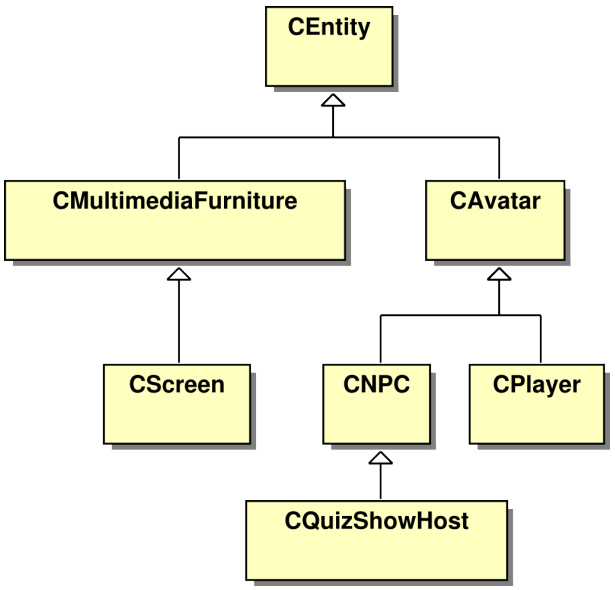

Figura 5.40: Jerarquía de entidades de la capa de modelo

Observer (sección 6.8) y debe gestionar sus propios *Listeners* para transmitir a los nodos gráficos sus atributos cuando haya algún cambio. En la figura 5.41 podemos ver el diagrama de clases de la clase `CEntity`, en el que se aprecia como tiene como atributo una lista de `EntityListener`, que no es otra cosa que una interfaz a implementar por todas las clases que quieran ser notificadas de los cambios de estado ocurridos en `CEntity`. Una vez una clase implementa esta interfaz, debe registrarse como *Listener* de `CEntity` y así podrá recibir las notificaciones (en la sección 5.5.3 se puede ver el listado de código 5.6 en el que se muestra la creación de un nodo y como este se añade como *Listener*).

En la figura 5.42 se puede apreciar como una modificación de la animación de un jugador, representado por la clase `CPlayer` conlleva una actualización del estado de su *Listener*, que es el nodo gráfico `CAnimatedEntityNode`.

A continuación se describen brevemente cada una de las entidades creadas:

- **CEntity:** Clase base de la que debe heredar cualquier entidad del juego. Todas las entidades se distinguen unívocamente por un atributo que las identifica, a su vez se caracterizan por tener una posición en el mundo 3D, una orientación sobre el eje Y (que corresponde al eje de altura), un factor de escala, un atributo booleano que indica si es una entidad visible o invisible y un último atributo que indica el *skin* (aspecto) de

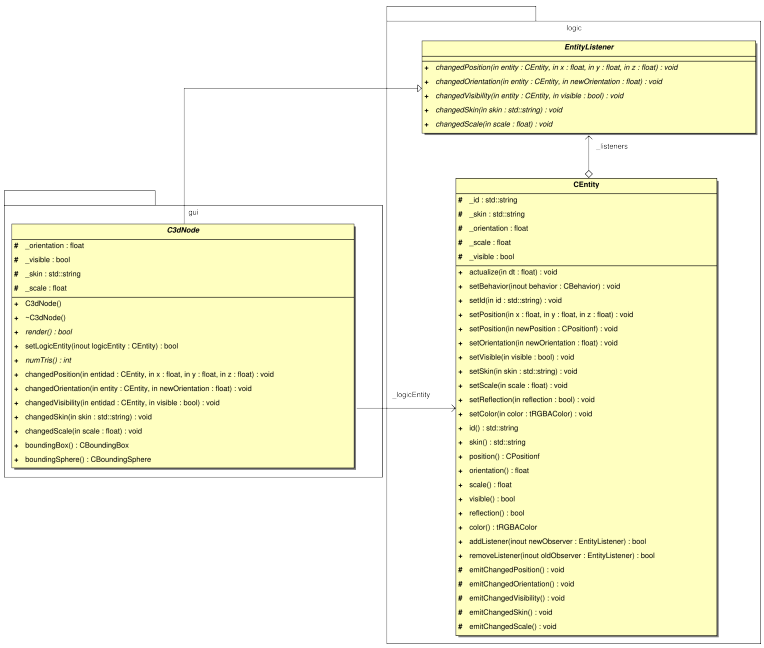

Figura 5.41: Separación entre nodo gráfico y entidad lógica mediante el patrón *Observer*

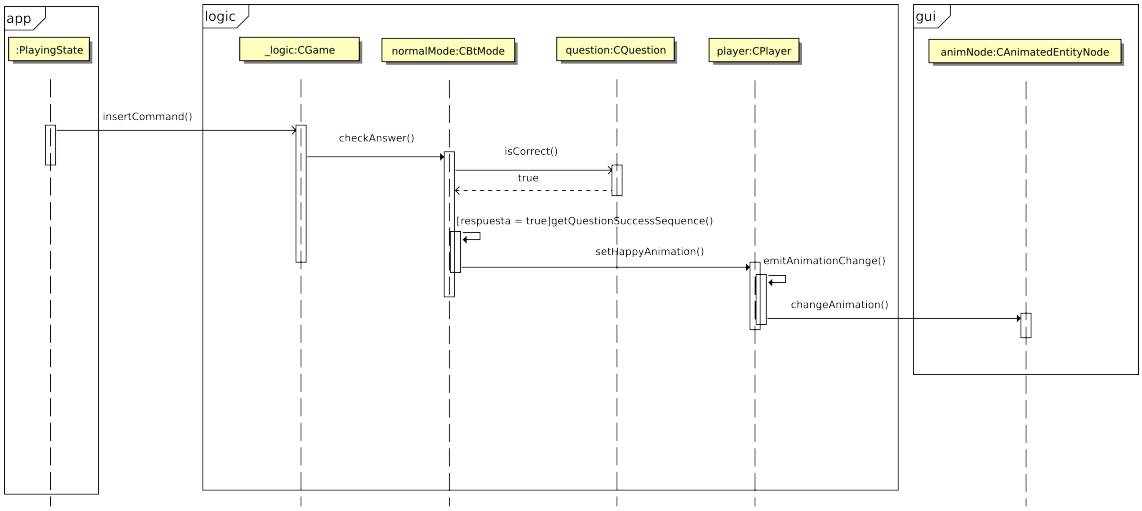

Figura 5.42: Secuencia de transmisión de información que conlleva una actualización de una entidad gráfica.

la entidad. Se usa para representar todos aquellos elementos inertes que participan en la simulación (elementos del escenario como atriles, muros, etc...).

- **CAvatar:** Esta clase es una interfaz de la que deben heredar todas aquellas clases que intenten representar un personaje animado del juego. Sus atributos almacenan la in-

formación sobre las animaciónes del personaje (en piernas, torso y cabeza), así como almacena también el nombre del personaje.

- **CMultimediaFurniture:** Esta clase se corresponde con elementos estáticos del escenario pero que incluyen efectos sobre ellos, como cambiar de textura cada cierto tiempo.
- **CScreen:** Esta especialización de `CMultimediaFurniture` representa una pantalla en la que se puede representar vídeo.
- **CNPC:** Representa a todos los personajes no controlados por el jugador.
- **CPlayer:** Representa a aquellos personajes manejados por el jugador.
- **CQuizShowHost:** Representa al presentador del concurso.

El gestor de escenario, por otra parte, se corresponde con un escenario de juego y se encarga símplemente de agrupar en un sólo sitio todas las entidades de ese escenario y de mandarlas actualizarse. Para poder realizar esto se ha usado el patrón *Strategy* (sección 6.10). Todas las entidades heredan de la clase base abstracta `CEntity` que obliga a implementar el método `actualize` a todas sus especializaciones. Este método es el que se encarga de actualizar el estado de las entidades en función de la entrada de usuario y de los eventos ocurridos en cada vuelta del bucle principal (sección 5.3). La forma en que estas entidades se comportan depende de la forma en que haya sido implementado este método en cada una de las clases de la jerarquía . Como se puede apreciar en el código correspondiente al método de actualización de la clase `CScenario` en el listado 5.14, este proceso se realiza de una forma sencilla y rápida.

Listing 5.14: Actualizacion del estado de todas las entidades de `CScenario`

```
1 void CScenario::actualizeEntitys(float dt){
2         TEntityIterator it(_entitys.begin()), end(_entitys.end());
3         for(;it!=end; ++it)
4                 (*it)->actualize(dt);
5 }
```

De la misma forma que se ha creado una jerarquía de entidades para representar los distintos elementos que participan en la simulación del juego, se ha creado una jerarquía aparte,

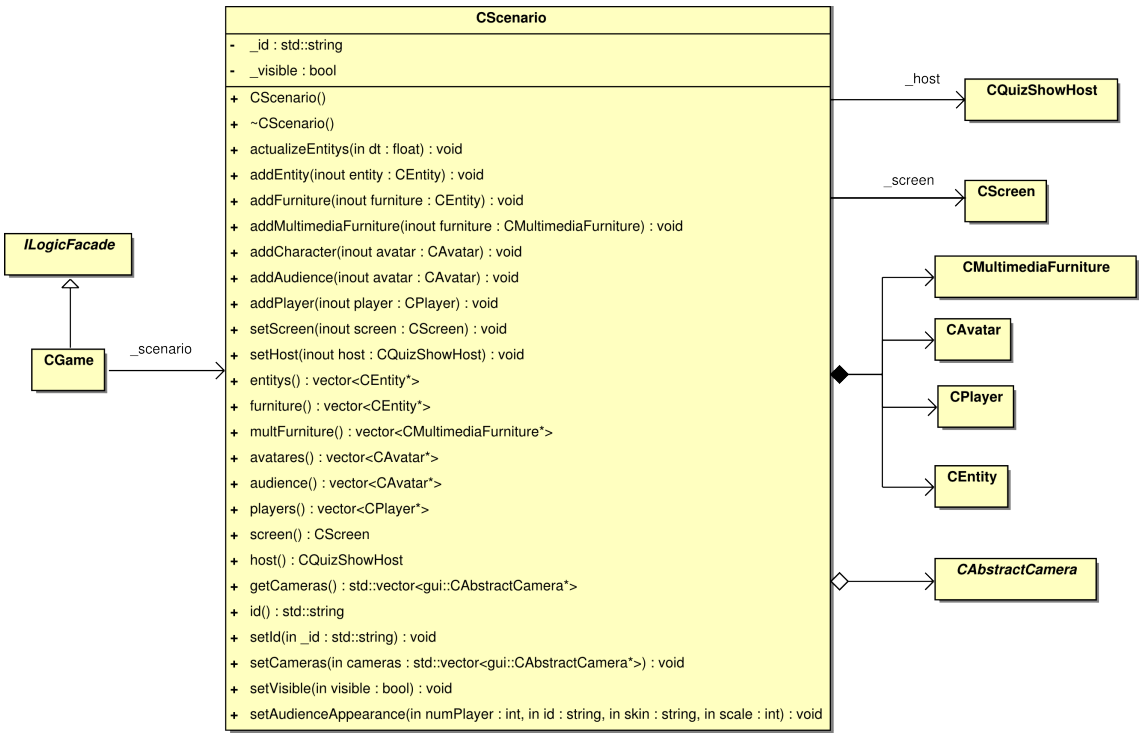

Figura 5.43: Clase `CScenario`.

para clasificar los componentes de la interfaz de usuario y que representan el conjunto de *widgets* disponibles en **YAOMEV**. En la figura 5.44 se muestra el diagrama de clases de la susodicha jerarquía.

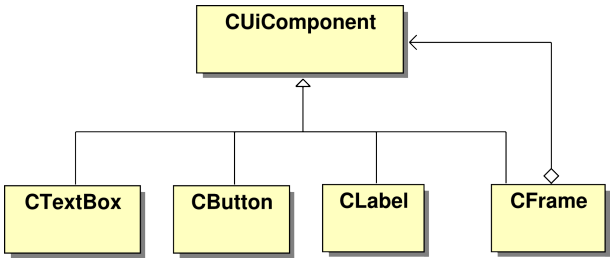

Figura 5.44: Jerarquía de componentes para la creación de la interfaz de usuario.

Al contrario de las entidades anteriormente descritas, estos elementos no necesitan de un método de actualización, pues su funcionamiento está estrictamente dirigido por eventos y reaccionan a los mismos de la forma descrita en la sección 5.5.11. Sin embargo, su diseño e implementación han sido muy similares al de las entidades y al igual que ellas, se ha usado el

patrón *Observer* (sección 6.8) para desacoplar su representación gráfica de su representación lógica. Cada uno de los componentes de la interfaz de usuario almacenan una lista a los *Listener* que deseen estar al tanto de los cambios en su estado.

Por otro lado, la construcción de la jerarquía se ha realizado de manera distinta, usando el patrón de diseño *Composite* (sección 6.4), para permitir la creación de interfaces más ricas y complejas mediante la agregación de componentes.

En **YAOMEV**, la creación de una sencilla interfaz se haría de la siguiente manera:

Listing 5.15: Ejemplo de creación de un frame con dos botones

```
1  _menuFrame = new logic::CFrame(0, 0, scrwidth, scrheight);
2
3  _initGameBtn = new logic::CButton("Inicio Juego", 400, 600, 210, 70);
4  _menuTestBtn = new logic::CButton("Test", 400, 500, 210, 70);
5
6  _initGameBtn->setStyle("webstyle_aqua");
7  _menuTestBtn->setStyle("webstyle_aqua");
8
9  _menuFrame->add(_initGameBtn);
10 _menuFrame->add(_menuTestBtn);
```

A continuación se describen los componentes de la jeraquía:

- **CUiComponent:** Esta es una clase abstracta que implementa los atributos comunes a todos los componentes de la interfaz de usuario: una cadena para identificarlo, la posición en pantalla, sus dimensiones (ancho y alto) y su transparencia. Todo nuevo componente que se quiera agregar a la jeraquía debe heredar de esta clase.

- **CFrame:** La clase `CFrame` (figura 5.45) representa una imágen bidimensional, sin embargo también actúa como un contenedor del resto de componentes de la interfaz y puede ser usado para agrupar componentes dentro de el.

- **CTextBox:** Representa un cuadro con texto, puede ser usado para escribir textos multilinea, permitiendo cambiar la fuente, el tamaño de la misma, el tamaño en caracteres de la línea, el fondo a usar y además permite cuatro tipos de alineación para el mismo: centrado, derecha, izquierda y alineado.

- **CLabel:** Representa una etiqueta de texto, es decir un texto simple mostrado en una sola línea, permitiendo cambiar tanto la fuente, como el tamaño de la misma.

- **CButton:** Representa un clásico botón con tres estados: pulsado, suelto y resaltado. Permite cambiar la fuente, su tamaño y el estilo del mismo (aspecto del botón).

La principal ventaja de esta implementación es la facilidad con la que se pueden añadir nuevos componentes para ser usados en la interfaz de usuario, simplemente habría que crear una nueva clase que heredara de `CUiComponent`, añadir los métodos necesarios en `CFrame` para su gestión y crear su correspondiente método de creación en `CSceneManager`.

Figura 5.45: Jerarquía de widgets de la interfaz de usuario.

Gracias a la implementación realizada, el desarrollador tiene a su disposición no solo un conjunto de entidades que le permite modelar un gran número de situaciones que ocurren en un videojuego, sino que también le es posible crear interfaces de usuario complejas para facilitar la interacción entre el usuario y el juego. Además, este no debe pensar en su representación gráfica, símplemente conocer los mecanismos mediante los que se comunica la lógica con el sistema gráfico.

Por otro lado, el gestor de escenario proporciona una forma sencilla de agrupar y acceder a todas las entidades participantes en el juego **Bit Them All!!!**, a la vez que clasifica todos

los elementos por escenario logrando con ello una mejor gestión de los mismos.

5.6.4. Gestión de comportamientos de juego

Queda por último hablar de uno de los temas más importantes en un videojuego: la implementación las reglas que guiarán la simulación. Es necesario insuflar de vida a todos los elementos participantes en el juego para conseguir que este sea realista. Este aspecto puede ser resuelto de múltiples formas, usando diferentes aproximaciones típicas de la inteligencia artificial. En el caso de **YAOMEV** y **Bit Them All!!!**, se ha optado por usar una herramienta muy poderosa llamada "Arbol de Comportamiento". Los árboles de comportamiento son unas estructuras de datos muy usadas en videojuegos desde hace tiempo y permiten organizar el comportamiento de nuestro sistema de forma sencilla.

Al igual que ocurrió con la representación de texto (sección 5.5.6), para conseguir esto se ha usado una biblioteca externa. En este caso, se ha usado la implementación de árboles de comportamiento que proporciona la biblioteca libre **libbehavior** y se han creado una serie de estructuras propias para contener los árboles que definan los comportamientos (figura 5.46).

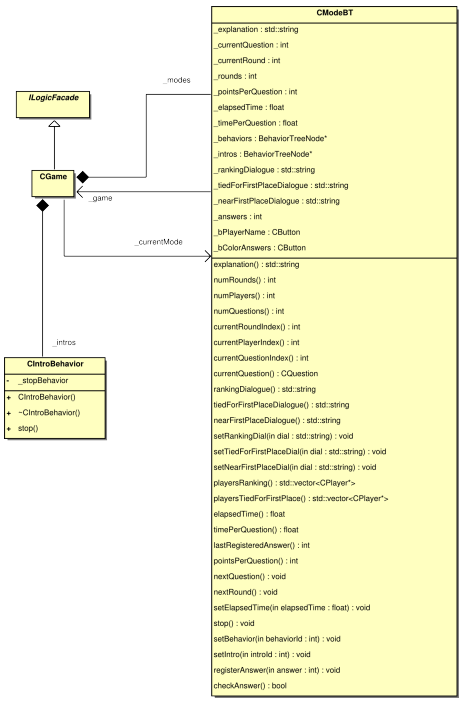

Figura 5.46: Clases encargadas de gestionar los comportamientos de los modos de juego.

Hay dos tipos de comportamientos, los de intro `CIntroBehavior`, en los que se muestra una secuencia de acciones (donde no hay interacción del jugador más allá de la posibilidad de interrumpirla) y los de modo `CModeBT`, que contienen todas las acciones a realizar durante la ejecución de un modo de juego y que además varían su comportamiento en función de la interacción con el usuario.

Para conseguir una amplia variedad de comportamientos ha sido necesario definir una jerarquía de nodos que puedan ser usados por el diseñador del juego para construir los árboles de la forma más completa posible. Los nodos creados desencadenan acciones simples que modifican atributos de la partida, comprueban una condición, activan la reproducción de una canción, cambian la animación de un personaje y un largo etcéterea.

La jerarquía de nodos se ha creado heredando de los nodos básicos que proporciona **libbehavior**. De esta forma se han creado tres tipos de nodos propios de **Bit Them All!!!**:

- `CGameNode`. Nodo atómico que ejecuta una acción sobre algún elemento del juego.
- `CInterruptibleByConditionNode`. Nodo compuesto que ejecuta su único hijo (un árbol) mientras no se de una condición, en el momento en que cumpla la ejecución termina.
- `CConditionNode`. Al igual que el nodo anterior es compuesto, aunque este ejecuta su único hijo sólo si la condición establecida se cumple. Su especialización tiene el efecto inverso, ejecuta su hijo cuando la condición no se cumple.

A partir del nodo `CGameNode` se han creado las siguientes jerarquías, que se pueden ver en las figuras 5.47, 5.48, 5.49, 5.50 y 5.51.

Por las características de los árboles de comportamiento, son muy útiles para los diseñadores pues permiten construir comportamientos complejos rápidamente. Se puede ver un ejemplo en listado de código 5.16 y en la figura 5.52, que muestra la vista de árbol de lo enseñado en el listado de código.

En el código se crea un árbol que funcionaría de la siguiente manera (siguiendo primero la primera rama del árbol): el primer nodo ejecutaría en paralelo sus dos hijos hasta que falle uno o los dos terminen con éxito su ejecución; el primero de ellos ejecutaría también en paralelo

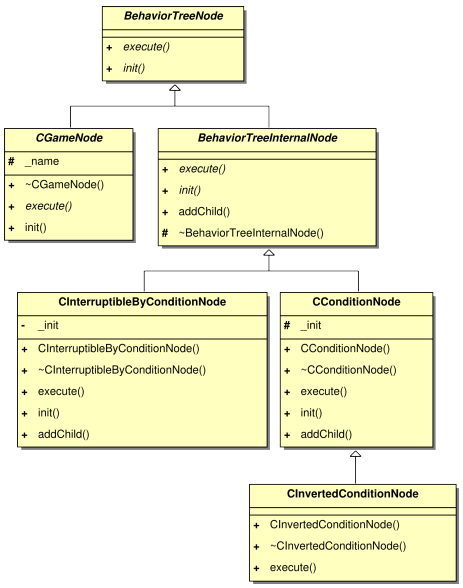

Figura 5.47: Clasificación de la jerarquía de nodos.

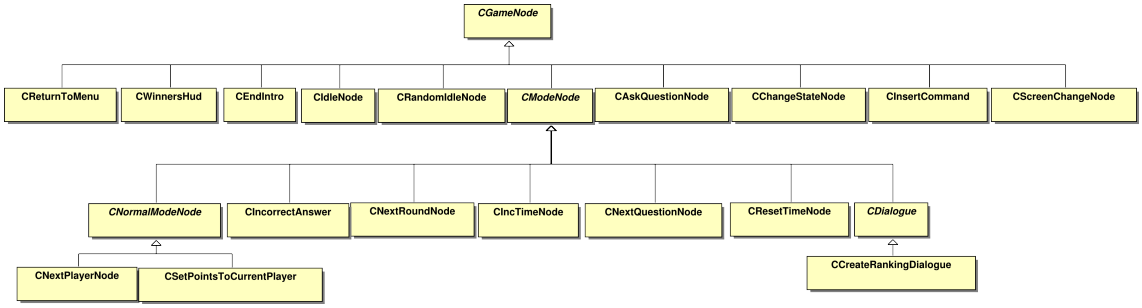

Figura 5.48: Jerarquía de nodos para árboles que modifican atributos del juego.

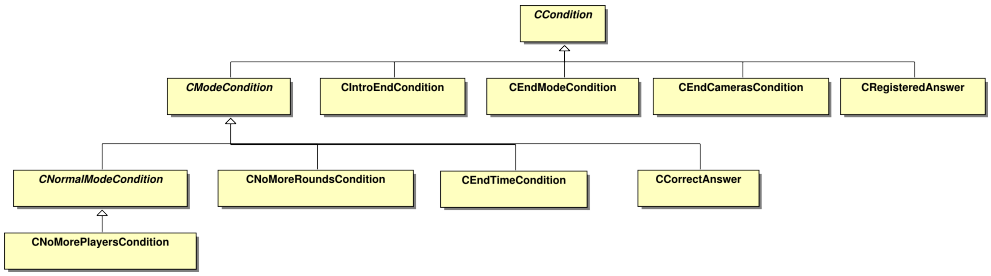

Figura 5.49: Jerarquía de nodos para árboles que comprueban condiciones del juego.

sus dos hijos de la misma forma que su padre, hasta que uno falle o los dos terminen con éxito; como sus dos hijos se ejecutan en paralelo, el resultado sería que el presentador activa una animación y se mantiene en ella mientras rota, una vez la rotación ha terminado se acaba

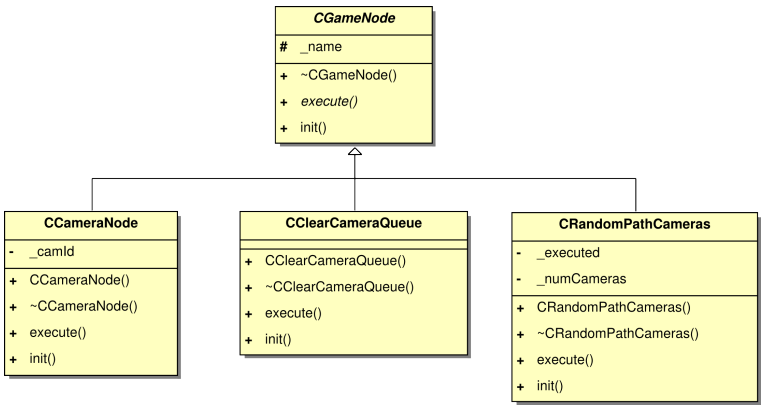

Figura 5.50: Jerarquía de nodos para árboles que ejecutan acciones sobre cámaras.

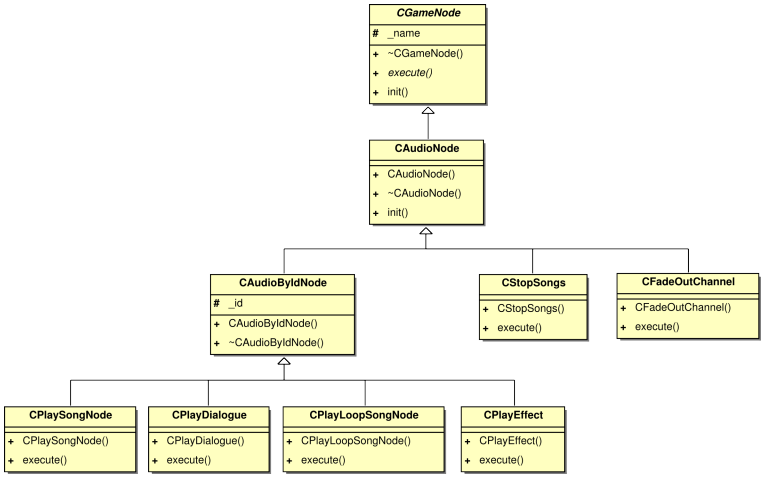

Figura 5.51: Jerarquía de nodos para árboles que ejecutan acciones sobre el sistema de sonido.

esta rama.

La otra rama es más convencional, ejecuta los tres hijos que tiene de forma secuencial, es decir, hasta que no acabe el primero no se ejecutará el segundo y así sucesivamente. El resultado es que se activa la cámara “cam3”, que es una cámara que sigue un camino, hasta que esa cámara no termine el camino no podrá ejecutarse el último nodo que es un cambio de animación.

Listing 5.16: Ejemplo de creación de un arbol de comportamiento

```
1 (modePresentation = new ParallelNode(FAIL_ON_ONE, SUCCEED_ON_ALL))
2                 ->addChild((new ParallelNode(FAIL_ON_ONE, SUCCEED_ON_ALL))
3                                 ->addChild(new CAnimateAvatarNode(_scenario->host(), "
                                      LEGS_IDLE", "TORSO_IDLE", "HEAD_DOUBLE_BLINKING"))
```

```
4                                   ->addChild(new COrientFromGivenOrientation(_scenario->host()
                                        , 0.0f, 180.0f, 0.07))
5                                   )
6                       ->addChild((new SequentialNode())
7                                   ->addChild(new CCameraNode("cam3"))
8                                   ->addChild(getDoNothingWhileCondition(new
                                        CEndCamerasCondition))
9                                   ->addChild(new CAnimateAvatarNode(_scenario->host(), "
                                        LEGS_IDLE", "TORSO_EXPLAINING", "HEAD_SPEAKING2"))
10                                  );
```

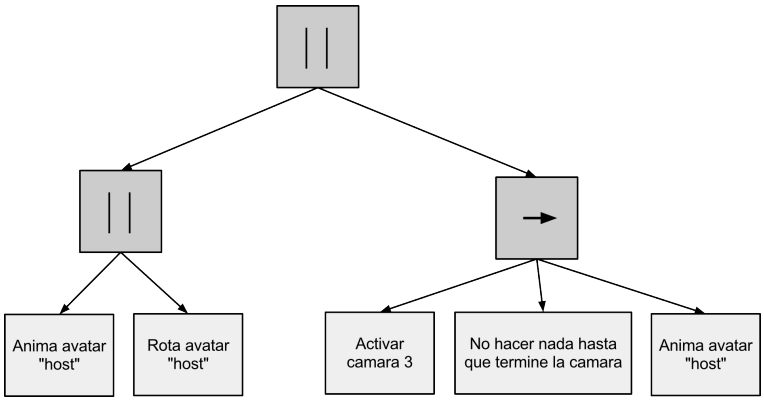

Figura 5.52: Diagrama del árbol formado con el listado de código 5.16

Capítulo 6

Patrones de diseño

A continuación se presenta una breve reseña de los patrones de diseño usados en **YAOMEV**. Sin embargo si se quiere profundizar en el tema es recomendable acudir al libro *"Design Patterns: Elements of Reusable Object-Oriented Software"* de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides [8].

6.1. Abstract Factory

Este patrón provee un interfaz común para la creación de familias de componentes, evitando de esta forma que se mezclen entre si y haciendo transparente al usuario el tipo de

familia concreta que está usando. En este proyecto se puede ver un ejemplo de este patrón en la implementación del servidor gráfico (sección 5.5.1).

6.2. Simple Factory

El patrón factoría simple es un patrón creacional. Se encarga de ofrecer una interfaz común a un conjunto de clases que realizan la misma tarea aunque hayan sido implementadas de forma distinta. La factoría centraliza el proceso de creación de esas clases haciendo invisible a la clase que use la factoría qué implementación concreta se está usando del objeto que devuelve.

Para lograr que la creación de estas clases esté centralizada exclusivamente en la factoría, las clases que devuelve tienen sus métodos de construcción con visibilidad privada. Estos objetos son clases`friend` de la factoría, de esta manera sólo ella los pueda crear.

En este proyecto se puede encontrar un ejemplo de este patrón en la implementación de la factoría de temporizadores (sección 5.2.3).

6.3. Singleton

Este patrón intenta evitar que exista más de una instancia de la clase que lo aplica. Además proporciona acceso global a esa instancia.

Funciona de la siguiente manera: el *Singleton* ofrece un método estático y público para acceder a su instancia, mientras que su constructor permanece privado para impedir que se creen instancias de forma no controlada. El método de acceso a la instancia comprueba si ya se ha inicializado, sino es así la inicializa y la devuelve, si por el contrario ya está inicializada simplemente devuelve la instancia existente.

En el caso de C++ es necesario ofrecer también un mecanismo de liberación de memoria por lo que su destructor también debe permanecer privado.

En este proyecto se pueden encontrar ejemplos de este patrón en las implementaciónes de el sistema de aplicación (sección 5.2), el sistema de log (sección 5.2.2) o la factoría de temporizadores (sección 5.2.3).

6.4. Composite

El propósito de este patrón es estructurar un conjunto de clases relacionadas de tal manera que sus objetos puedan ser tratados uniformemente [8].

Estos objetos se agrupan en una estructura de árbol para representar jerarquías todo-parte. Con la aplicación de este patrón las clases clientes de objetos composite pueden tratar objetos individuales o compuestos de la misma manera.

En este proyecto se pueden encontrar ejemplos de este patrón en las implementaciónes de los controladores genéricos (sección 5.3) o en la jerarquía de componentes de interfaz de usuario (sección 5.6.3).

6.5. Façade

Este patrón tiene como propósito proveer de una interfaz unificada a un subsistema compuesto por un conjunto de interfaces distintas. La fachada define un interfaz en un mayor nivel de abstracción que permite que dicho subsistema sea más fácil de usar. La figura 5.34 muestra un ejemplo claro sobre como facilita y agrupa las relaciones de un subsistema con un conjunto de clases cliente.

En este proyecto se puede encontrar un ejemplo de este patrón en la implementación de la fachada lógica (sección 5.6.1).

6.6. Flyweight

Este patrón estructural tiene como propósito reducir la redundancia cuando un gran número de clases maneja idéntica información. Por ello describe los mecanismos para permitir el uso compartido de objetos.

Un *flyweight* es un objeto compartido que puede ser usado en una gran variedad de contextos simultáneamente, actuando este siempre como un objeto independiente del contexto y que es indistinguible de una instancia del objeto no compartido.

La clave se encuentra en la distinción entre estado intrínseco y estado extrínseco. El

primero es el estado almacenado en el *flyweight* que es independiente del contexto, mientras que el segundo se corresponde con el contexto en el que es usado.

En este proyecto se puede encontrar un ejemplo de este patrón en la implementación de los nodos de escena, que se corresponden con las clases que proporcionan el estado extrínseco al *flyweight* (sección 5.5.3), y en la implementación de los recursos, que actúan como los *flyweight* (sección 5.5.1).

6.7. Proxy

Este patrón tiene como propósito actuar de intermediario de un objeto, controlando el acceso al mismo. En concreto en este proyecto se han usado los *Proxy* de referencia inteligente, que sirven para tareas como contar las referencias a un objeto (permitiendo con ello que múltiples objetos puedan usar una misma instancia del objeto, ver seccion 6.6), cargar un objeto persistente bajo demanda, control de la concurrencia en un objeto compartido, etc...

En este proyecto se puede encontrar un ejemplo de este patrón en la implementación de las cachés de recursos (sección 5.5.1).

6.8. Observer

Como se puede leer en [8]: *"Un efecto secundario típico del particionamiento de un sistema en una colección de clases cooperativas es la necesidad de mantener la consistencia entre los objetos relacionados. No se debe pretender conseguir esa consistencia haciendo las clases estrechamente dependientes (o acopladas) porque eso reduce su usabilidad"*. El patrón de comportamiento *Observer* ayuda en esta tarea de desacoplamiento entre clases relacionadas.

Una clase A posee cierta información que otra clase B necesita conocer, sin embargo estas dos clases no pueden conocerse entre si. El ejemplo típico es una fuente de datos y su representación visual. La forma de conseguir este desacoplamiento es incluyendo una tercera clase en la relación, esta nueva clase es lo que se conoce como *Observer* o *Listener* y no es más que una interfaz que define unos métodos que permiten transmitir información.

La clase B debe implementar esta interfaz, convirtiendose con ello en *Observer* y la clase A, conocida genéricamente como *Subject*, debe mantener una lista de instancias de clases que implementan esta interfaz. Cuando un objeto de la clase B (o cualquiera de una clase que implemente el interfaz) quiere conocer el estado de la información que mantiene A, debe registrarse en A como *Observer*. Así si hay algun cambio en la información que contiene la clase A o *Subject* esta recorre su lista de *Observers* invocando el método correspondiente de intercambio de información, notificando de esta manera a la clase B (o cualquiera que implemente el interfaz) el cambio ocurrido.

En este proyecto se tomó la decisión de llamar a las clases que actúan de *Observer* como *Listener*, lo cual no tiene ninguna consecuencia en la aplicación de este patrón. Se pueden encontrar ejemplos de este patrón en la implementación de los *Listener* de creación/destrucción (sección 5.6.1) o en los de las entidades lógicas (sección 5.6.3).

6.9. State

Este patrón de diseño de comportamiento, propone una solución al problema de variar el comportamiento de un objeto en función de su estado interno. Según el libro *"Design Patterns: Elements of Reusable Object-Oriented Software"* esta una forma elegante para, parcialmente, cambiar el tipo de un objeto en tiempo de ejecución.

La clave en este patrón es crear una clase abstracta Ej: `AbstractState` que represente los estados de la clase a la que queremos permitir cambios en su comportamiento, en este ejemplo llamaremos a esta clase `Context`. La clase `AbstractState` declara un interfaz común a todas las clases que representan los diferentes estados de la clase `Context`. Por cada estado que se quiera tener se deberá implementar una especialización de `AbstractClass`.

La clase `Context` mantiene una lista con todos sus estados posibles (instancias a objetos que implementan la interfaz `AbstractClass`) y también una instancia escogida entre esos estados como su estado actual. La clase `Context` delega todas las peticiones y la ejecución de operaciones dependientes de su estado a este objeto estado.

En este proyecto se puede encontrar un ejemplo de este patrón en la implementación de la máquina de estados usada para gestionar los controladores (sección 5.3).

6.10. Strategy

La motivación de este patrón de comportamiento es resolver el problema de cómo cambiar, en tiempo de ejecución, entre diferentes formas de resolver un mismo problema. Dicho de otra manera: en la resolución de un problema cualquiera es posible que existan diferentes algoritmos que permitan alcanzar una solución, si se pretende ofrecer la posibilidad de elegir en tiempo de ejecución cuál de esos algoritmos usar, este patrón resuelve la forma de hacerlo.

Su aplicación es la siguiente: Creamos una familia de algoritmos que resuelven el problema en cuestión, los encapsulamos en clases y las hacemos intercambiables. La forma de hacerlas intercambiables es que todas implementen el mismo interfaz.

En este proyecto se puede encontrar un ejemplo de este patrón para lograr la actualización de las entidades del juego (ver listado de código 5.14).

6.11. Template Method

El `Template Method` es un patrón de comportamiento que consiste en definir una clase base en la que se implementa el esqueleto de un algoritmo, sin embargo algunos de los métodos que intervienen en el se dejan sin implementar. Las clases hijas serán las encargadas de otorgar funcionalidad a estos métodos como les convenga. Con este diseño, se pueden conseguir variaciones en el comportamiento del algoritmo, pero impidiendo la modificación de su estructura.

En este proyecto se puede encontrar un ejemplo de este patrón en la creación de las clases encargadas de gestionar el funcionamiento de la aplicación (sección 5.2).

Capítulo 7

Evolución y costes

En este capítulo se estudiará el proceso de desarrollo dentro del marco de trabajo **SCRUM** (capítulo 4) mediante el análisis de las diferentes fases en las que se descompuso el trabajo que ha dado lugar a este proyecto. Acto seguido se discutirán los costes estimados asociados al mismo.

7.1. Evolución del proyecto

En esta sección se analizarán los principales hitos alcanzados en la realización de este proyecto, resultado directo de la aplicación de la metodología elegida (capítulo 4.1). El proyecto **YAOMEV** y el juego demostrador **Bit Them All!!!** han tomado un tiempo de trabajo que ronda el año y medio, de los cuales puede estimarse una dedicación a tiempo completo aproximada de un año. El curso 2009 - 2010 se dedicó al estudio exhaustivo de la programación

de videojuegos a nivel general mediante el estudio del Máster de Desarrollo de Videojuego impartido por la Universidad Complutense de Madrid. Hacia el final del Máster se empezó a aplicar lo aprendido en el desarrollo del proyecto actual. Sin embargo no fue hasta la finalización del Máster cuando se comenzó el desarrollo con una dedicación completa, pudiéndose establecer la fecha de inicio en Septiembre de 2010.

Los primeros pasos se centraron en la creación de la infraestructura de desarrollo (elección de las bibliotecas a usar, del lenguaje de programación, del IDE, del sistema de control de versiones, etc...) y al estudio de los requisitos que el proyecto debía cumplir. Inmediatamente después se comenzó a aplicar la metodología elegida (capítulo 4) hasta que, mediante sucesivas iteraciones, se alcanzó el estado final de la implementación.

7.1.1. Análisis de requisitos

La primera parte del proyecto, transcurrida en los seis meses comprendidos entre Marzo y Agosto de 2010, se dedicó al estudio del estado del arte del desarrollo de juegos, de plataformas que dan soporte a la creación de los mismos (capítulo 3) y del estilo de los juegos que se querían usar como base para el demostrador. De esta etapa se obtuvo la información necesaria para comenzar a definir los requisitos a cumplir. En el desarrollo de una plataforma como **YAOMEV** y un juego demostrador como **Bit Them All!!!**, una definición de requisitos exhaustiva puede convertir el desarrollo en inacabable debido a la multitud de opciones que pueden llegar a ser incluidas, por lo tanto en esta etapa se establecieron unos requisitos mínimos a cumplir y otros opcionales, sobre todo en lo relativo al demostrador y al motor 3D implementados.

Se puede separar el análisis de requisitos en dos partes bien diferenciadas: el análisis de los requisitos de **YAOMEV** que debe dar soporte al desarrollo de juegos y el análisis de requisitos para **Bit Them All!!!** que implica el desarrollo de un juego usando la plataforma creada.

Relativo al estudio y selección de requisitos para **YAOMEV** se procedió al análisis de los motores gráficos 3D que podrían haberse usado para simplificar el desarrollo, de lo cual se concluyó que ninguno satisfacía completamente las necesidades de este proyecto, bien por ser

excesivamente complejos en su uso, poco portables o exigir una alta potencia de cálculo. De este estudio surgieron una ingente cantidad de requisitos que podrían usarse para el módulo de representación gráfica. La selección exacta de aquellos a cumplir exigió una cuidadosa criva, dejando una lista abierta con un conjunto mínimo a cumplir por este módulo: renderizado mediante grafo de escena, modelos con animación basada en vértices, luces dinámicas, cámaras, selección de widgets 2D, manejo de texturas, etc...

Una vez concluido que no se usarían motores 3D externos, se estudiaron las matemáticas involucradas tanto en la representación de gráficos 3D, como a la usada en el movimiento de entidades por un espacio euclídeo. Al acabar esta etapa se obtuvieron los requisitos indispensables que la biblioteca matemática debía satisfacer.

Tras esto se estudió el tipo de arquitectura que permitiría el grado de modularidad requerido por la plataforma **YAOMEV** y que facilitará la creación de juegos sin preocuparse por ningún otro aspecto. Se decidió que el patrón de arquitectura del software **Modelo-Vista-Controlador** (sección 5.1) cumplía con estos requisitos y se decidió que **YAOMEV** debía adaptarse a este modo de desarrollo.

Por último se analizaron el tipo de herramientas que se deberían incluir y que facilitarían la programación de juegos. Se planificaron los scripts que ayudarían a importar escenarios y personajes desde *Blender*, el sistema de persistencia (sección 5.4) que facilitara su carga y uso en el juego, y un sistema que capturara la entrada de usuario y la comunique mediante eventos a los componentes interesados.

Tras esta primera etapa relativa al estudio de los requisitos de la plataforma de creación de juegos, se pasó a estudiar los que se aplicarían al juego demostrador. Para ello se analizó el funcionamiento del juego inspirador (*Buzz: ¡El gran reto!*) y diversos concursos televisivos, con el objetivo de transmitir los más fielmente posible el espíritu que acompaña este tipo de juegos a **Bit Them All!!!**.

Esta etapa se dedicó al estudio del flujo de juego (cómo transcurre un concurso), del sistema de cámaras específico a usar (con un estilo televisivo), de las diferentes formas de interacción entre jugador y juego, y por último, una selección de preguntas de cultura general que acompañen el juego.

Aunque la parte que más tiempo consumió en la planificación del juego fue sin duda la

relativa al análisis de la estrategia de inteligencia artificial que se aplicaría. Se optó por el uso de árboles de comportamiento (sección 5.6.4), sobre todo por la magnífica versatilidad que aportan a la hora de crear cualquier tipo de comportamiento imaginable para las entidades que participan en un juego. Tanto es así que se decidió que todos los elementos se controlarían mediante esta estructura de datos.

7.1.2. Infraestructura

Una vez decididos los requisitos generales a cumplir se pasó a la selección de las bibliotecas y herramientas a utilizar y a su preparación para su uso. Este paso apenas consumió tiempo pues se tenía muy claro en los objetivos específicos (sección 2.2) la necesidad de que las bibliotecas fueran multidispositivo, software libre y se adaptaran a estándares reconocidos. Además la infraestructura debía facilitar que cualquier persona pueda participar en un futuro en la ampliación del proyecto y, por supuesto, dar soporte a la metodología elegida (sección 4.1).

Por lo tanto se creó el repositorio para el sistema de control de versiones (gestionado mediante SVN) en `https://code.google.com/p/yaomev/` y se instalaron y probaron las bibliotecas y utilidades necesarias para el desarrollo (sección 4.2).

7.1.3. Diseño

La arquitectura de **YAOMEV** se diseñó siguiendo el patrón de arquitectura del software **MVC** (sección 5.1), tal y como quedó establecido en los requisitos (sección 7.1.1). En esta etapa se procedió a planificar su implementación orientándolo a la creación de videojuegos. Para ello se descompuso en cinco módulos completamente independientes (capítulo 5): aplicación, control, persistencia, juego y representación. Esta división de realizó de tal forma que evite a cualquier desarrollador dificultades a la hora de ampliar, mejorar o solucionar problemas en cualquiera de los módulos, permitiéndole enfocarse única y exclusivamente al problema concreto en el que trabaje.

Además, cada uno de los módulos se planificó para aislar al programador de las bibliotecas concretas que se usen, para ser sencillo de usar, modificar y de comprender. Por eso a lo

largo de esta etapa se aplicaron gran cantidad de patrones de diseño (capítulo 6), herramienta indispensable hoy en día si se quieren cumplir las pretensiones anteriormente mencionadas.

Para el desarrollo del juego **Bit Them All!!!** también se planificó un diseño (sección 5.6) que facilitará su modificación, ampliación o pruebas. Este diseño se basó en crear una pequeña jerarquía de entidades pertenecientes al juego controlables mediante la construcción de nodos que les asignan pequeños comportamientos o tareas. A partir de estos nodos los árboles de comportamiento se pueden construir de forma tan simple como ir añadiendo los mismos de forma jerárquica mediante el uso de nodos compuestos (nodos especiales que siguen diferentes reglas a la hora de recorrer sus nodos hijos). De esta forma la modificación de los comportamientos es tan simple como la variación del orden de los nodos o de las reglas mediante las que se ejecutan estas estructuras de datos. De este modo, una vez se tiene un árbol de comportamiento, éste puede ser reutilizado en cualquier situación del juego o incluso de un juego a otro.

7.1.4. Implementación

En esta sección se presenta la descomposición en iteraciones del proceso de implementación realizado para crear **YAOMEV** y **Bit Them All!!!**. En cada *Sprint* se explicarán los hitos conseguidos, los problemas encontrados y el esfuerzo realizado.

7.1.4.1. Sprint 1

El objetivo del primer *sprint* era tener un esqueleto básico de la arquitectura **MVC** que permitiera capturar la entrada del usuario, cambiar el comportamiento mediante el cambio dinámico de controladores, cargar un modelo 3D con texturas y mostrarlo por pantalla.

Durante esta primera iteración se creó el proyecto de *Eclipse* y el árbol de directorios que contendría la aplicación. También se creó el repositorio en *Google Code*.

En esta primera versión de la aplicación se le dio sobre todo importancia a la capa de aplicación pues parte de su responsabilidad consiste en gestionar el resto de módulos. Al finalizar el *sprint* estaba creada la arquitectura de los controladores junto con un controlador específico para un menú y otro para un estado que simbolizaba un juego, además de la jerarquía de

clases de aplicación.

También se trabajó bastante en la capa de persistencia, creando la primera versión del servidor de *DAOs* y los dos primeros cargadores de recursos, uno de los cuales leía archivos *OreJ* (un formato simple que permite representar modelos 3D con animación de sólido-rígido) y el otro texturas.

Para dar soporte a la visualización de los modelos 3D se crearon las primeras clases de la biblioteca matemática, en concreto las responsables de permitir el uso de vectores en un espacio 3D y otra para las matrices de transformación.

Por último, también se creó la versión preliminar de la capa de presentación, en la que ya se incluyó el gestor de escena con capacidad para crear, a partir de un objeto de la capa de lógica, un nodo de escena con funcionalidad básica que permitía representar un objeto 3D en el formato *OreJ*. También se creó el servidor gráfico que era el que suministraba el modelo 3D al gestor de escena para crear el nodo correspondiente.

7.1.4.2. Sprint 2

Para este segundo *sprint*, se contaba con los objetivos de ampliar lo realizado en el hito anterior y comenzar el desarrollo del juego en paralelo con la arquitectura. De este modo se pretendía comprender los requerimientos de las herramientas genéricas que **YAOMEV** debía contener. Al final se pretendía tener una primera versión de las estructuras que representan los escenarios, de los personajes animados y de un menú funcional.

Se crearon los componentes genéricos de la capa de lógica, con unas implementaciones simples que permitieran tener en el mundo lógico entidades estáticas, entidades animadas, sprites y etiquetas de texto.

En la capa de persistencia se crearon los cargadores de modelos en formato *OreJ* y de sprites 2D.

En la capa de aplicación se crearon los primeros controladores compuestos, que se usaron para crear una versión algo más compleja del menu del *sprint* anterior. Con su ejecución se mostraban una imagen y la etiqueta con el nombre **YAOMEV**.

En la capa de representación se sustituyó el nodo de representación de modelos 3D que usaba el formato *OreJ* por otro que usaba el formato *MD3* y se crearon también los nodo

correspondientes a los sprites 2D y a etiquetas de texto.

En la biblioteca matemática se empezó a trabajar en las estructuras que permitían rotar las entidades e interpolar tanto vectores, como rotaciones, añadiendose para ello nuevas clases que permitían el uso de cuaternios y vectores en 2 y 4 dimensiones.

Se creó la primera versión del exportador de escenarios de *Blender*. Esta herramienta permitía exportar todos los modelos graficos del escenario al metaformato *MD3*, y además, mediante la creación de un fichero *XML* permitía importarlos en **YAOMEV**. Unido a la creación del exportador se creó la primera versión del plató de televisión para el juego **Bit Them All!!!**.

7.1.4.3. Sprint 3

Para este *sprint* se quería aprovechar la implementación de la biblioteca matemática para comenzar a desarrollar las estructuras que dieran control al sistema de cámaras y de luces. A su vez se pretendía avanzar en el desarrollo del motor 3D y de la lógica básica que permitía crear pruebas. Al final se pretendía contar una primera versión de las cachés de recursos, que permitieran una gestión más eficiente de la memoria usada en el proceso de dibujado. también botones funcionales y una primera versión del gestor de la entrada del usuario.

Relativo al motor 3D, se unificaron las cachés de recursos de botones, imágenes y modelos 3D, dentro del servidor gráfico, con el fin de presentar un único punto desde el que gestionarlos.

En la primera aproximación al sistema de cámaras, primero se crearon las clases que envolvía la funcionalidad de **OpenGL** para situar el punto de vista, después se implementaron dos tipos de cámaras: dinámicas y estáticas. Las primeras ofrecían la posibilidad de moverse, en línea recta, de un lado a otro del mundo 3D indicándole el punto de origen y de destino. Las segundas, eran cámaras que permanecía fijas en un punto. Por último, para gestionarlas se implementó una factoría que daba acceso a las distintas cámaras disponibles y que permitía el añadido de nuevos tipos de forma transparente. De esta forma, el uso de cámaras se podía hacer exclusivamente desde el código.

En este *sprint* se planteó también la necesidad de incorporar luces a las escenas. Para ello se creó una clase que se encargaba de representar tres tipos de luces distintas (posicional, foco y ambiente). Para simplificar su uso, se dió la responsabilidad de manejarlas al gestor de

escena.

Se sustituye el uso de etiquetas de texto mediante clases propias escritas usando *SDL_ttf*, por otras nuevas que envuelven la funcionalidad de la biblioteca *FTGL*, mucho más eficientes y poderosa. Con ellas, se crean los nodos de escena correspondientes a etiquetas de texto y cajas de texto.

Para permitir el uso de los widgets creados para las interfaces, se creó un gestor de entrada de usuario de prueba que, usando las estructuras de **SDL**, se encargaba de detectar las pulsaciones del ratón sobre los botones y las pulsaciones de teclas.

Como parte de **Bit Them All!!!**, se creó el menú principal y dos submenús, el de selección de escenario, de selección de personaje.

7.1.4.4. Sprint 4

Hasta este momento se usaban las estructuras de **SDL** para recibir la entrada del usuario y gestionar la respuesta de la aplicación. Durante este *sprint* se planto el cambio de este sistema a otro propio basado en eventos y en el patrón de diseño *Observer*. Por lo tanto se creó una clase que envolvía la funcionalidad de **SDL** y usaba un conjunto de estructuras propias para transmitir esta información.

El añadido anterior hacía necesaria una reimplementación de la forma en que los controladores usaban y manejaban la información proporcionada el usuario. Se hizo necesearia una refactorización de estas clases para soportar el nuevo sistema de procesamiento de la entrada de usuario. Una vez este paso se hubo completado, se contaba ya con una clase genérica que permitía gestionar el comportamiento que ofrece la aplicación a un usuario.

La forma en que se habían implementado las cámaras en el *sprint* anterior hacía complicado el uso de las mismas en un escenario creado en *Blender* y exportado al formato usado por **YAOMEV**. Por lo tanto, tomando como base el gestor de cámaras creado, se decidió rehacer las clase que representaba las cámaras dinámicas, para añadir la posibilidad de que estas recorrieran caminos prefijados. Con el objetivo de facilitar el desarrollo del seguimiento de caminos para las cámaras y el diseño de los escenarios, se creó un exportador que, desde *Blender*, permitía exportar en un fichero *XML* las rutas creadas por un diseñador. De este modo se facilitaba la ejecución de pruebas usando el nuevo tipo de cámaras y además permitía

al diseñador gráfico la creación de cámaras específicas para cada escenario. Con esto se daba por finalizado el sistema de control de cámaras.

Como producto de la creación de las cámaras, se incorporó a la biblioteca matemática una clase que permitía el uso de splines cúbicas naturales para representar internamente caminos a partir de un conjunto de puntos significativos del mismo.

Por último, unido a la creación del exportador de cámaras, se terminó también el desarrollo del exportador para *Blender* de escenarios.

7.1.4.5. Sprint 5

Durante el *sprint* anterior se hizo evidente que era necesario cambiar la forma en que se representaban los gráficos 3D, pues las necesidades de **Bit Them All!!!** en cuanto al número de entidades a representar no se veían satisfechas. Por lo tanto durante esta etapa el desarrollo se centró en la optimización del renderizado, incluyendo también la adición de las capacidades multimedia necesarias para la reproducción de vídeo y audio.

Para conseguir estas metas se decidió seguir el patrón *Flyweight* para reducir el tamaño en memoria de una aplicación escrita con **YAOMEV**. Esto hizo necesaria la refactorización de los nodos de escena para extraer tanto su estado extrínseco, como el intrínseco. Hasta este momento, cada nodo de escena tenía una instancia propia del recurso a dibujar.

Durante la implementación de los nodos anteriores, se detectó que era significatívamente más eficiente usar *vertex arrays* para el dibujado que el método inmediato que ofrece **OpenGL**. Esto llevó a dos mejoras más durante el *sprint*, para implementar el nuevo dibujado, se crearon clases específicas para renderizar los recursos gráficos, lo cual permitía desacoplar la forma de dibujado de los nodos.

En este *sprint* se diseñaron e implementaron los sistemas de representación de vídeo y de audio, y se crearon las estructuras necesarias para manejarlos.

Con la finalización de los elementos anteriores se daba por finalizada la capa gráfica de **YAOMEV**.

7.1.4.6. Sprint 6

Con la plataforma **YAOMEV** terminada, se inicia en este *sprint* el desarrollo real de **Bit Them All!!!**. Para ello se crea una primera aproximación a la lógica del juego, basada en una máquina de estados, en la que se implementan las clases encargadas de la representación interna del concurso. Para ayudar a la separación entre capas, se diseña la lógica oculta tras una fachada lógica.

En este *sprint* se cuenta por primera vez con gráficos originales gracias a la incoporación al desarrollo de David Prieto, grafista, que realiza un trabajo académicamente dirigido [11] ayudando en este proyecto.

Gracias a los exportadores e importadores creados en este proyecto, se incorporan con rapidez todos los *assets* facilitados por el grafista a **Bit Them All!!!**, mientras se solucionan diversos fallos encontrados en las herramientas de exportación.

Por último se desarrolló un script que permitía el uso de *Festival* para crear las locuciones de las preguntas a partir de los textos de las preguntas del juego.

7.1.4.7. Sprint 7

El *sprint* anterior demostró que el uso de una máquina de estados, para gestionar el comportamiento de la simulación del concurso televisivo, hacía muy dificil las modificaciones durante las pruebas. Por lo tanto se decidió cambiar la máquina de estados que gestionaba los comportamientos por otro tipo de estructura que permitía una mayor flexibilidad a la hora de crear comportamientos, los árboles de comportamiento.

Para el uso de árboles de comportamiento, se usó la implementación de los mismos que proporciona la biblioteca *libbehavior*. Por lo tanto, se crearon de nuevo las clases que representan los modos de juego del concurso adaptadas a esta nueva estructura.

El uso de árboles de comportamiento implica la creación de un conjunto de nodos especializados que simbolizan las acciones que los árboles pueden ejecutar. En este *sprint*, la mayor parte del esfuerzo se dedicó a construir un conjunto lo suficientemente amplio de nodos como para poder diseñar las situaciones de concurso definidas en el documento de concepto.

Con la creación de las jerarquías de nodos para los árboles de comportamiento, se dió por

terminada la capa de lógica de juego.

Por último, se crearon botones, marcos e iconos para construir las interfaces usadas en **Bit Them All!!!**.

7.1.4.8. Sprint 8

Llegados a este punto **YAOMEV** ya cumple con los requisitos para la creación de un juego. Los últimos pasos del desarrollo se centraron exclusivamente en el desarrollo de **Bit Them All!!!**, su testeado y en la resolución de los problemas que surgieron.

Con respecto al desarrollo **Bit Them All!!!**, casi todo el foco de atención estuvo puesto en el flujo de juego, es decir, en el diseño de los árboles de comportamiento necesarios para dar vida a todas las entidades participantes en la simulación. Este paso hizo necesario un proceso contínuo de ensallo y error con el fin de localizar fallos en el transcurrir de las pruebas del concurso o en las secuencias animadas. Por lo tanto fue necesaria la creación y modificación de múltiples árboles de comportamiento para dar con el flujo de juego idóneo.

Durante este *sprint* se acabó también el diseño de todo el conjunto de gráficos a usar en el juego, realizados con *Blender* por David Prieto [11].

Con la creación de un ejecutable con la demo del juego, se dió por concluido el último *sprint* y con él, el proceso de desarrollo.

7.1.5. Recursos y costes

Esta sección se ha dedicado a esbozar una estimación sobre los recursos y costes asociados a la realización de este proyecto, a la vez que se realiza un análisis de las estadísticas de desarrollo obtenidas gracias al repositorio en el que se encuentra almacenado el proyecto (`https://code.google.com/p/yaomev/`). Quedando estos aspectos desglosados en las siguientes secciones.

7.1.6. Coste económico

La plataforma **YAOMEV** y el juego demostrador **Bit Them All!!!** se han creado entre Marzo de 2010 y Julio de 2011. Cabe remarcar, como se ha hecho en la introducción de este

capítulo, que realmente el desarrollo efectivo se realizó entre Septiembre de 2010 y Julio de 2011, contando de este modo con 10 meses de trabajo en el código de este proyecto. Por lo tanto se ha hecho una aproximación al trabajo realizado por los dos participantes (el programador y el grafista), quedando estimado (para el programador) un desarrollo de 40 semanas, en jornadas de 8 horas, durante 5 días a la semana. Lo cual da como resultado 1600 horas de trabajo, por lo que basándonos en el coste por hora de un desarrollador *freelance* típico (*30€/hora* según `http://www.infolancer.net`) arrojaría un coste apróximado de 48000 €. Mientras que para el grafista se ha estimado un coste por hora de 15 €/hora (también en base a las tarifas ofrecidas en *infolancer*) durante 4 meses, cinco días a la semana, en jornadas de 5 horas, dando como resultado 400 horas de trabajo, quedando en un coste aproximado de 6000 €/hora.

Cuadro 7.1: Tabla de costes por recursos

Recurso	Cantidad	Coste
Recursos humanos	2	54000,00
Equipo	2	2600,00
Total: 56600,00 €		

7.1.7. Estadísticas del repositorio

En el desarrollo del proyecto se ha usado el sistema de control de versiones SVN, sobre un repositorio de Google Code (`https://code.google.com/p/yaomev/`). Gracias a la herramienta *StatSVN* se ha podido analizar la evolución del repositorio usado diferentes métricas y mediante la herramienta *cloc* se ha podido establecer el número total de líneas de código del proyecto. Gracias a ellas se puede hacer un seguimiento del estado del proyecto a lo largo de los 10 meses de trabajo.

Primero se muestra la evolución en líneas de código, aunque antes es conveniente aclarar que el primer gráfico correponde al proyecto inicial, del que en Ábril se hizo una desviación cuando comenzó el desarrollo del juego **Bit Them All!!!**, y el segundo pertenece a la ramificación.

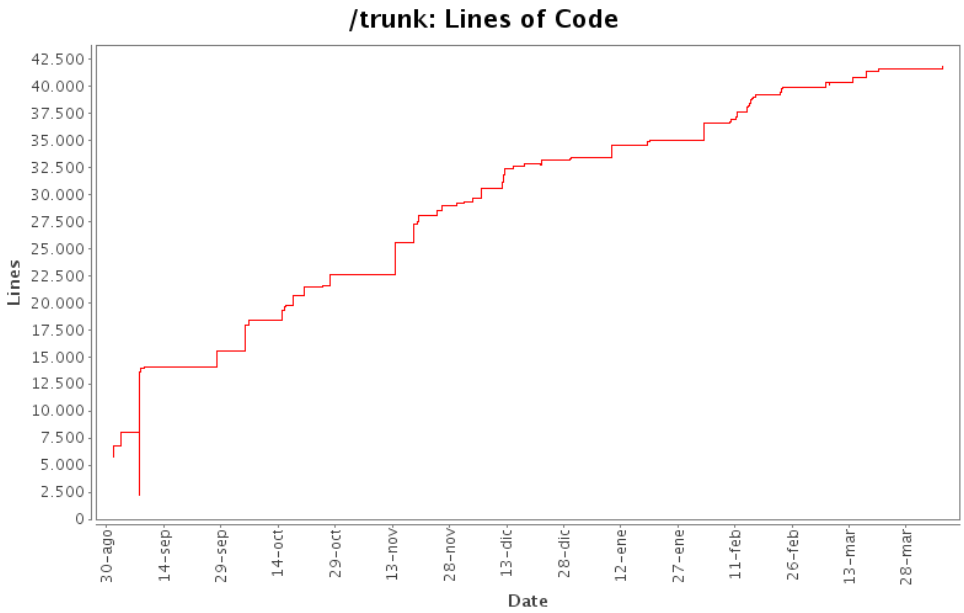

Figura 7.1: Evolución de las líneas de código de Agosto de 2010 a Marzo de 2011

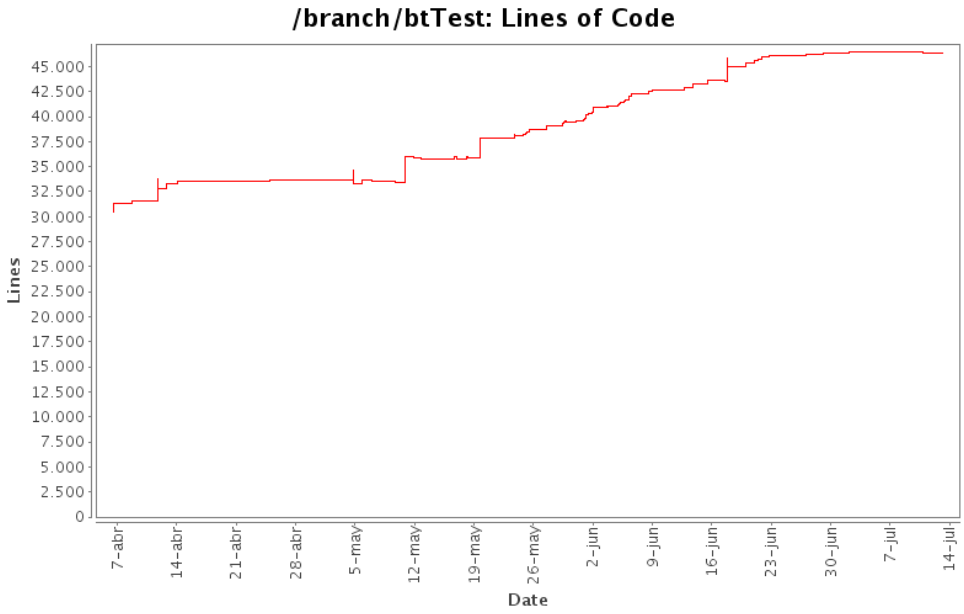

Figura 7.2: Evolución de las líneas de código de Abril a Julio de 2011

En las dos figuras (7.1 y 7.2) se aprecia el caracter incremental que ha aportado a este proyecto la metodología usada. También se puede percibir cómo en cada uno de ellos aparece una evolución en el desarrollo distinta. El primero, correspondiente casi enteramente al desar-

rollo de la plataforma **YAOMEV**, presenta un crecimiento mucho más rápido. Esto es debido a que fue durante ese tiempo cuando se implementaron casi todos los requisitos del proyecto. El segundo, sin embargo, muestra el tiempo transcurrido mientras se desarrollaba el juego y se corregían los errores que se iban detectando, por lo que presenta un crecimiento mucho más suave.

Por último, se muestra el resultado final en líneas de código de la plataforma **YAOMEV** junto con su juego demostrador **Bit Them All!!!**.

Cuadro 7.2: Total de líneas de código del proyecto

Lenguaje	Núm. de Archivos	Lín. en blanco	Comentarios	Lín. de código
Cabeceras C/C++	221	6328	19353	12743
C++	111	3127	1865	9128
Python	7	451	323	1222
XML	13 +	41	27	1817
make	1	11	8	70
Total:	353	9958	21573	24980

Capítulo 8

Resultados

En este capítulo se comentarán brevemente los resultados obtenidos durante el desarrollo, tanto de **Bit Them All!!!**, como de algunos ejemplos realizados para experimentar con las posibilidades de **YAOMEV**. Para ello se han realizado unas pequeñas pruebas de estrés con el objetivo de ser analizadas y poder comparar cómo varía el rendimiento.

8.1. Análisis del rendimiento

Esta sección se dedica al análisis de rendimiento (*profiling*) de la plataforma **YAOMEV**. En ella se explican las diferentes pruebas de rendimiento realizadas y los resultados obtenidos mediante el uso de aplicaciones específicas para su medición.

Estas pruebas se realizan sometiendo a diferentes niveles de estrés a la aplicación, para determinar la velocidad de ejecución bajo ciertas condiciones específicas de carga computacional. El uso de herramientas de *profiling* permite realizar un análisis dinámico de la aplicación, es decir, analiza el comportamiento interno del programa durante su ejecución, con lo que es posible averiguar qué partes actúan como *"cuellos de botella"* y son susceptibles de ser optimizadas.

Para realizar este análisis en **YAOMEV**, se ha utilizado la herramienta libre *gprof*. Esta

herramienta recoge estadísticas sobre el uso de las funciones empleadas durante la ejecución del programa analizado.

Es necesario para ello compilar el proyecto usando el flag `-pg`. De esta forma se insertan en código funciones que sirven para recopilar información sobre el tiempo que tarda cada función en ejecutarse y permite establecer una jerarquía de llamadas entre las mismas.

Una vez compilado, es necesario hacer uso del programa de forma normal, para que se reunan las estadísticas necesarias, las cuales, son almacenadas en un fichero llamado `gmon.out`. Una vez se tiene este archivo, se ha usado un programa libre llamado *Gprof2Dot* [6] (ver listado de código 8.1), que permite usar la biblioteca **graphviz** para obtener una representación visual en forma de grafo de la información suministrada por *gprof* (figura 8.1[1]).

El grafo muestra una jerarquía de llamadas entre funciones. Cada nodo representa una función y muestra información sobre su ejecución, contiene la siguiente información (diagrama esquemático en el listado 8.1):

- *tiempo total %.* Porcentaje de tiempo de ejecución dentro de la función, junto con el todos sus hijos.
- *tiempo interno %.* Porcentaje de tiempo de ejecución dedicado sólo a ejecutar instrucciones de la función, no de sus hijos.
- *llamadas totales.* Número total de llamadas que ha recibido esta función durante la ejecución (incluyendo llamadas recursivas).

Cada arista representa las llamadas a una función ocurridas desde el nodo padre, contiene la siguiente información:

- *tiempo total %.* Porcentaje de tiempo de ejecución transferido desde el nodo hijo al nodo padre.
- *llamadas.* Número total de llamadas que ha recibido esta función desde el nodo padre.

La figura 8.1 muestra el análisis de uso de uno de los ejemplos llevados a cabo (representación de más de 300 entidades mediante modelos simples). Con este grafo se vuelve

[1] El grafo mostrado ha sido recortado, ante la imposibilidad de mostrarlo entero.

mucho más sencilla la tarea de estudiar qué partes de la aplicación son las más costosas en tiempo de ejecución. En la figura se aprecia el reparto del tiempo entre las diferentes funciones de las clases. En seguida salta a la vista observando el grafo que la mayor parte del tiempo se consume en el renderizado de las escenas (89,91 % del tiempo, nodo naranja correspondiente a `gui::CScene::render` al final de la figura). Por otro lado, el resto del tiempo se dedica a la ejecución de la lógica (no aparece en la figura), que consume un 1,1 % del tiempo y a la carga de recursos con un 8,94 % (en la figura sólo se muestra la parte relativa a la inicialización del juego, en el nodo azul que representa a la función `app::CBitThemAllApp::startGame` con un 4,64 %).

```
+-------------------------------------+
|        nombre funci\a'{o}n          |           tiempo total %
| tiempo total % ( tiempo interno % ) |              llamadas
|       llamadas totales              |   padre --------------------> hijo
+-------------------------------------+
```

Listing 8.1: Comando para crear el grafo mediante *Gprof2Dot* y obtener una imagen en formato *svg*

```
$ gprof Debug/BitThemAll | ./gprof2dot.py -s | dot -Tsvg -o output_strip.svg
```

De esta forma, se han tomado como objeto de análisis dos variables: por un lado el número medio de frames por segundo que mantiene la aplicación y, por otro, el porcentaje de tiempo que consume cada método a lo largo de la ejecución de cada uno de los tests.

- **Uso de modelos complejos para mostrar una escena**. Para este test se plantea una escena muy simple, con un suelo y poblada por un gran número de personajes representados por un modelo complejo (3596 triángulos). De todos los modelos se crea uno que puede ser usado por el jugador, con el objetivo de proporcionar algo de carga de lógica al test. Usando esta escena se realizan pruebas con diferente número de personajes, mientras se recopilan datos de *profiling* y del *framerate*. Con modelos complejos se ha llegado a alcanzar una tasa de 25 frames por segundo al representar 64 entidades, dando un total de 230322 triángulos en pantalla (figura 8.3).

- **Uso de modelos simples para mostrar una escena.** Este test se ha creado de la misma

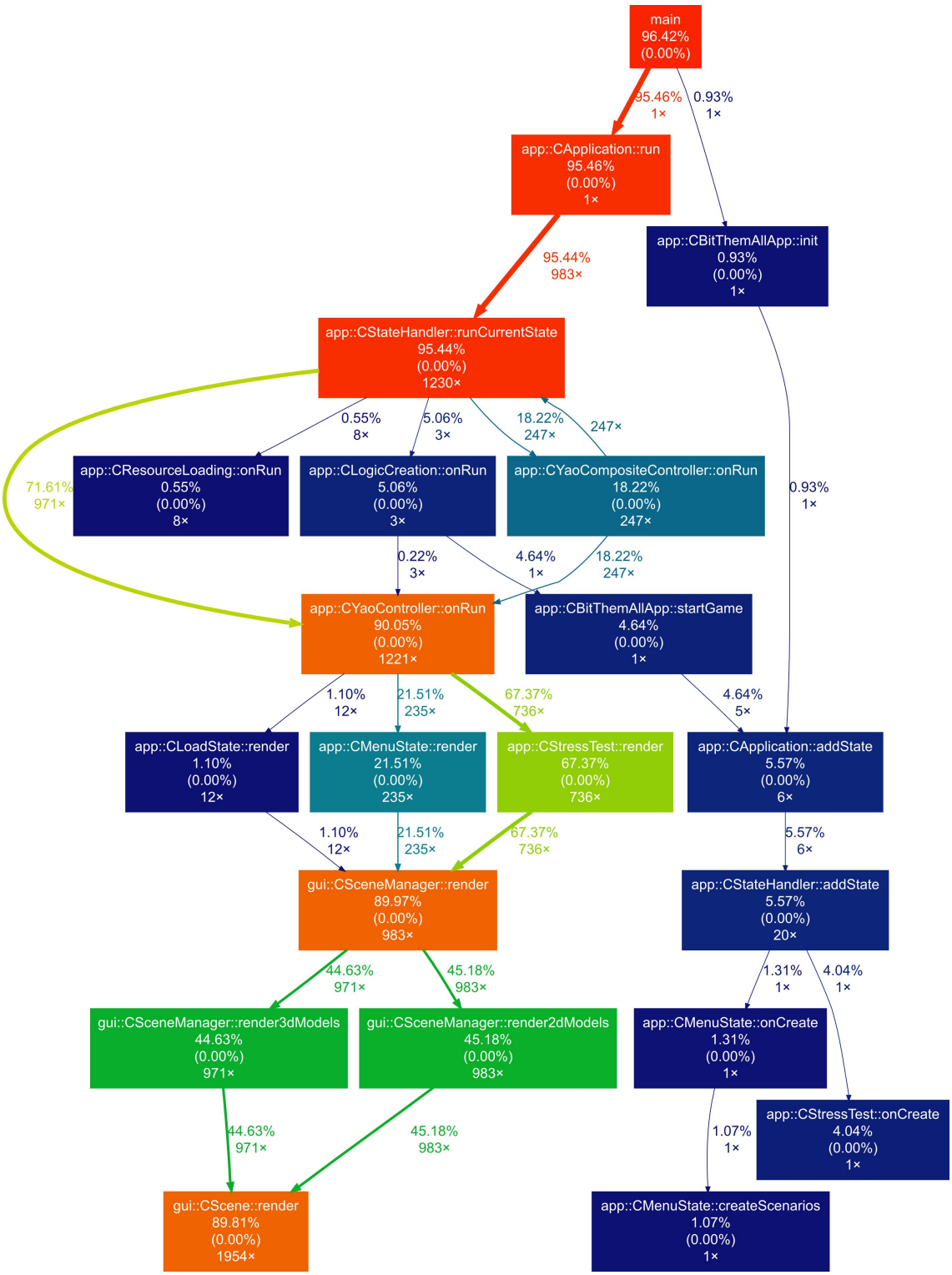

Figura 8.1: Ejemplo de grafo generado con *Gprof2Dot*

forma que el anterior, cambiando simplemente la complejidad del modelo usado para representar las entidades lógicas, que para este caso cuentan con 88 triángulos cada uno. Con modelos simples se han conseguido representar 901 entidades a 25 frames por segundo, suponiendo un total de 79288 triángulos en pantalla (figura 8.4).

- **Juego demostrador Bit Them All!!!.** El juego demostrador cumple el papel de una prueba mixta y con una alta carga de procesamiento de lógica. Durante su ejecución se alcanzan tasas sostenidas de 120 frames por segundo, con hasta 71 entidades en pantalla (contando personajes y moviliario), con la música y animaciones activas. En total, esta prueba supone 17376 triángulos en pantalla.

Los resultados se presentan de forma resumida en la tabla 8.1 y en la figura 8.2.

Cuadro 8.1: Tabla con un resumen de los resultados obtenidos mediante diferentes pruebas usando **YAOMEV**

Lenguaje	Entidades	FPS	Triángulos	Representación	Lógica	Persistencia
Modelo complejo	64	25	230322	95,12 %	1,1 %	3,15 %
	16	60	19353	91,36 %	1,05 %	7,59 %
Modelo simple	901	25	79288	94,48 %	1,68 %	3,84 %
	325	60	28600	89,97 %	1,1 %	8,93 %
Juego	71	120	17376	93,88 %	1,47 %	4,65 %

En la tabla se aprecia un dato curioso, el número de triángulos soportados por **YAOMEV** aumenta cuanto más complejo es el modelo a representar, aunque a costa de verse reducidas las entidades disponibles. Esto tiene su explicación en la forma en que se dibujan los modelos. Al no usar lo que se conoce como método inmediato de dibujado de **OpenGL** (altamente ineficiente), en **YAOMEV** se usan *vertex arrays* (sección 5.5.7). Esta técnica supone que en cada vuelta del bucle principal se mandan a la tarjeta gráfica, para cadad entidad, todos los vértices, normales, coordenadas de textura, etc... en una sola llamada a la función de dibujado. Si el modelo es complejo, el número de entidades disminuye, al disminuir el número de entidades disminuye el número de llamadas a la función que envía toda la información a dibujar y por lo tanto se permiten más triángulos. Esta forma de dibujado penaliza un número alto de llamadas con pocos triángulos y optimiza pocas llamadas aunque se envíen más triángulos.

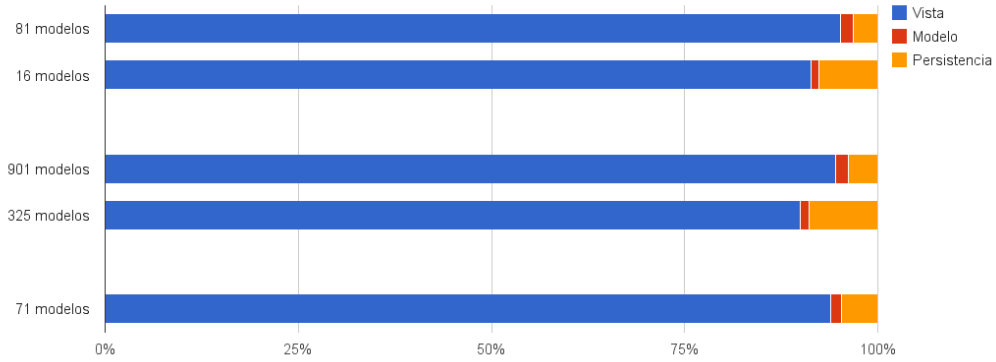

Figura 8.2: Gráfico comparativo de porcentaje de tiempo invertido en cada capa

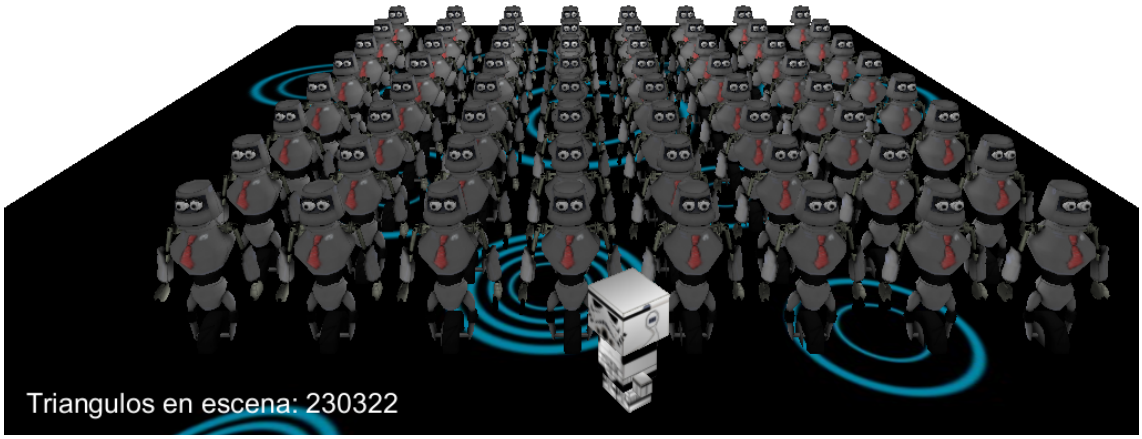

Figura 8.3: Ejemplo de prueba realizada durante los tests con modelos complejos

Vistos los porcentajes de tiempo que se dedica a cada función, se puede concluir el tiempo de proceso en cada sistema (columnas Representación, Lógica y Persistencia de la tabla 8.1). En ella salta a la vista enseguida cuales son las tareas más complejas y que más tiempo consumen, que se corresponden con los sistemas de dibujado y carga de recursos. Por lo tanto, se puede concluir que **YAOMEV** evita al usuario, que se está iniciando el mundo de la programación de videojuegos, las partes más costosas del desarrollo.

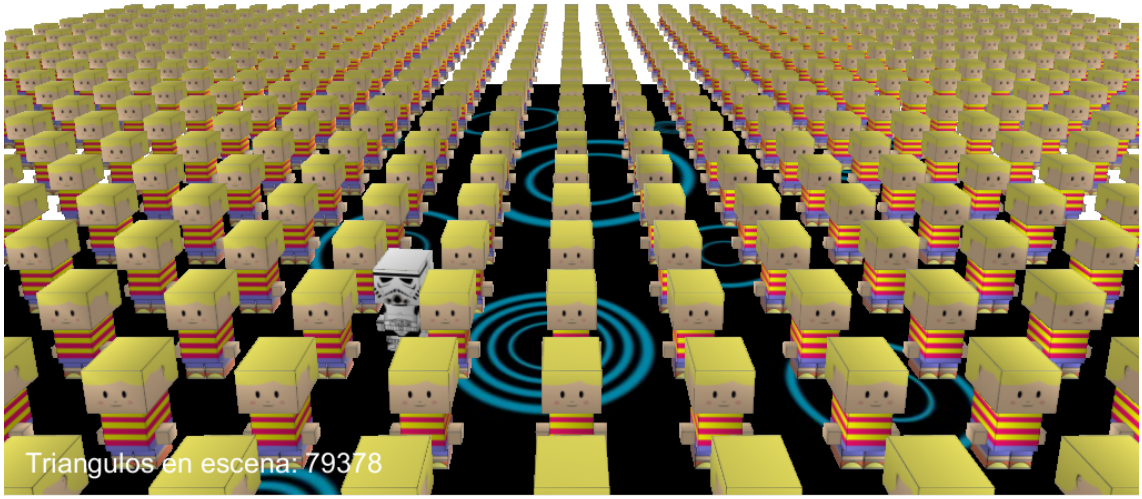

Figura 8.4: Ejemplo de prueba realizada durante los tests con modelos simples

8.2. Resultados

El desarrollo completo de **Bit Them All!!!** y **YAOMEV** ha ido mostrando diferentes caras a lo largo del tiempo. Para finalizar este capítulo, se muestra la evolución que ha sufrido este proyecto y para ello se presentan dos mosaicos (figuras 8.5 y 8.6). El primero sirve para reflejar el progreso que ha sufrido el juego demostrador **Bit Them All!!!** (desde las primeras pruebas para representar objetos, a el uso de personajes de Quake 3 para actuar como jugadores, a por fin contar con un conjunto propio de gráficos). Mientras, el segundo, se ofrece a revelar el aspecto final del juego, con sus interfaces de usuario, personajes, efectos, etc...

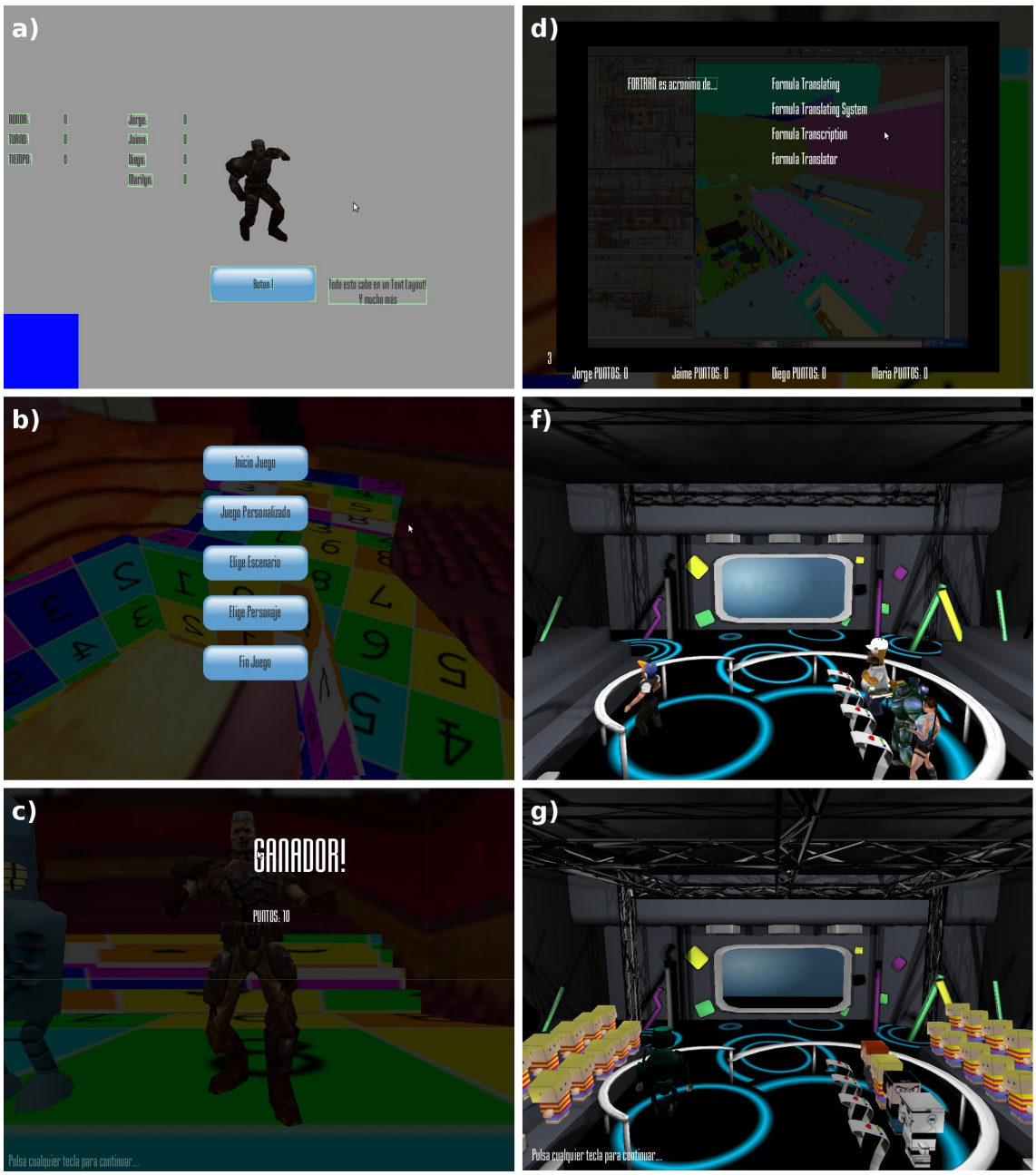

Figura 8.5: Aspecto del juego demostrador **Bit Them All!!!** durante su evolución. a) Versión inicial, con soporte para texto, botones, frames y modelos 3D animados (Noviembre 2010). b) Primera versión con escenario y menú funcional. Cargadores de recursos y widgets completo. (Diciembre 2010). c) Pruebas con sistema de comportamiento primitivo. Sistema gráfico y de persistencia casi completo. (Enero 2011). d) Primeras versiones de la pantalla del escenario con vídeo. (Enero 2011). e) Escenario de juego en su versión casi definitiva. Soporte para múltiples jugadores y árboles de comportamiento. (Marzo 2011). f) Versión de test, con gráficos propios. (Mayo 2011)

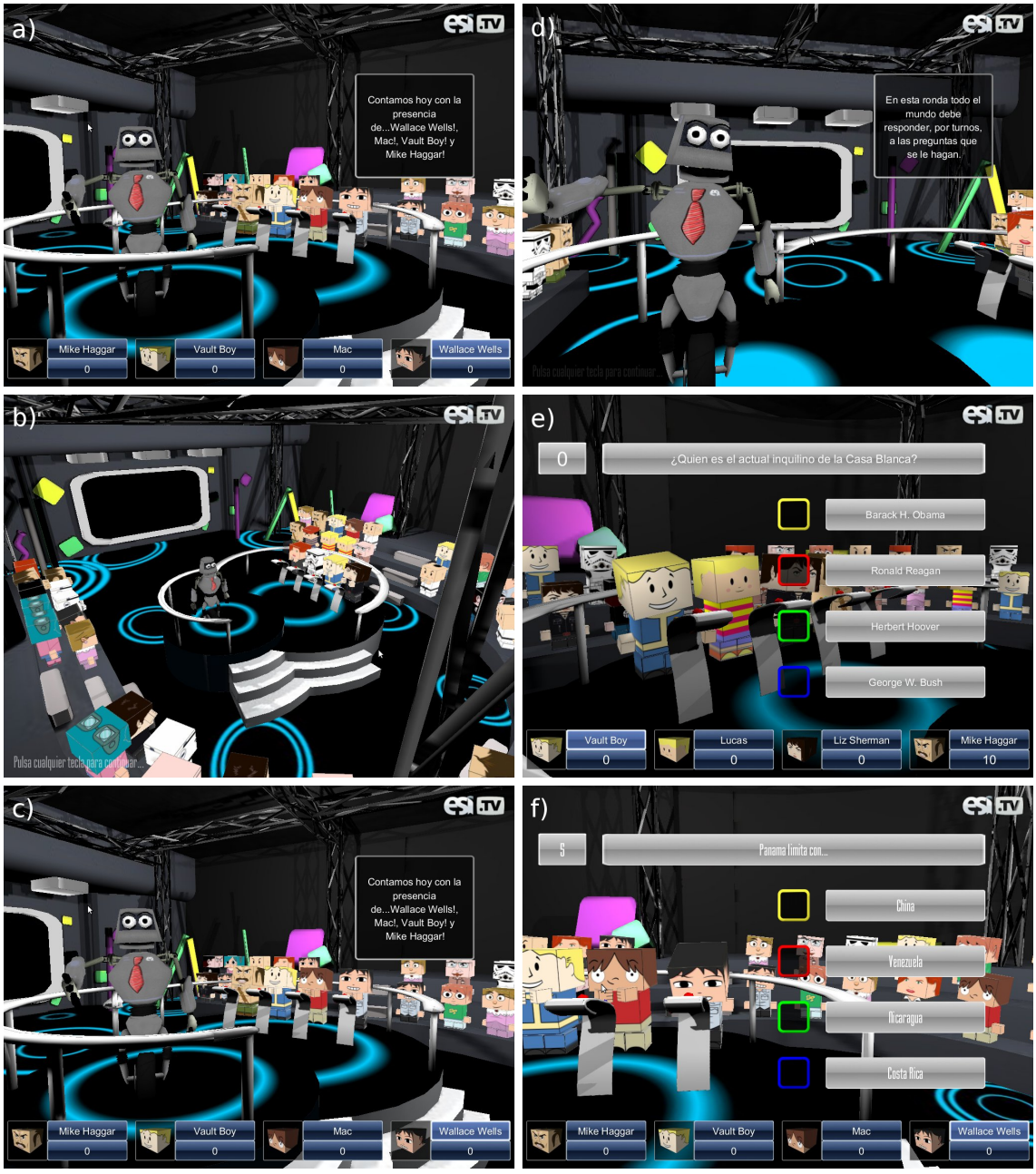

Figura 8.6: Aspecto final del juego demostrador **Bit Them All!!!**. a) El juego soporta widgets personalizados para la creación sencilla de interfaces complejos. b) Los caminos seguidos por las cámaras se pueden exportar desde Blender de forma automática. c) Soporte para modelos con texturas dinámicas. d) La plataforma permite la exportación de personajes animados desde Blender. f) El juego permite el uso de sistemas de síntesis de voz para la lectura de preguntas.

Capítulo 9

Conclusiones y propuestas

Este capítulo se orienta a hacer una pequeña reflexión sobre los objetivos alcanzados durante la realización del presente proyecto fin de carrera, comparándolos con los objetivos propuestos en el capítulo 2. Por otro lado se exponen las conclusiones personales producto del enfrentamiento a un desarrollo con una envergadura y complejidad equivalentes a las que uno puede encontrar en el mundo laboral, y por último se deja constancia de aquellos posibles caminos a seguir en un ulterior desarrollo de **YAOMEV**.

9.1. Objetivos alcanzados

El objetivo prioritario de este proyecto era construir una plataforma para el desarrollo de videojuegos multidispositivo, que facilitara la vida a programadores con escaso bagage en el campo de la programación de videojuegos. Para demostrar que este punto ha sido llevado a cabo con éxito, se propuso también como segundo objetivo la creación de un juego demostrador que exhibiera las capacidades de **YAOMEV**.

Una vez concluido el presente proyecto fin de carrera, se puede decir que los objetivos impuestos en un principio, se han cumplido (como queda reflejado en la explicación de la

arquitectura en el capítulo (capítulo 5)). **YAOMEV** es una plataforma multidispositivo, con un diseño modular que, gracias a su implementación siguiendo el patrón de arquitectura del software **MVC** (sección 5.1), y un buen conjunto de patrones de diseño (sección 6), suministra las herramientas necesarias que permiten abordar el desarrollo de un videjuego de una forma rápida y sencilla. **Bit Them All!!!** es un ejemplo perfecto de lo que se puede conseguir usando esta plataforma.

De esta forma, gracias a la estructuración por capas conseguida siguiendo el patrón **MVC** (sección 5.1), junto con el conjunto de herramientas creadas para la capa de modelo (sección 5.6), se ha logrado aislar al desarrollador de videojuegos de gran parte de la complejidad intrínseca existente al programar un juego desde cero. Gracias a esto se puede presuponer que aunque el usuario de **YAOMEV** cuente con un conocimiento escaso de materias como informática gráfica, sonido, interfaces de usuario o inteligencia artificial, dispondrá de una herramienta muy poderosa para conseguir llevar a cabo el desarrollo de su juego.

Como ya se ha comentado, la base en la que se sustenta la filosofía de **YAOMEV** reside en la consideración de que el patrón **MVC** facilita enormemente la programación de una aplicación. Por lo tanto se han diseñado con éxito los sistemas necesarios para ahorrar tiempo y trabajo en la aplicación de este patrón a la hora de crear un videojuego.

La capa más completa es la de vista. Encargada de la representación visual de lo que ocurre en la simulación mediante el uso de bibliotecas multidispositivo. Para ello se han creado diferentes subsistemas que en conjunto permiten el despliegue de gráficos complejos tanto en 3D, como en 2D, proporcionando además herramientas de reproducción de elementos multimedia. Cabe destacar que están soportados formatos de representación de modelos complejos 3D con animación basada en vértice; suministra un completo conjunto de widgets para la creación de interfaces de usuario; permite la representación de texto tanto en 2D, como en 3D; permite el uso de vídeo en formato MPEG-1 como textura de modelos 3D o 2D; es capaz de reproducir audio en los formatos WAVE, OGG, MP3, AIFF, RIFF, VOC, MOD y MIDI; así como el soporte de un gran conjunto de formatos de imagen para ser usados como textura.

El primer subsistema sobre el que yace la representación gráfica es el de gestión de recursos. En él, mediante un conjunto de cachés (sección 5.5.1), se almacenan en memoria de

forma eficiente los recursos gráficos (o *assets*) y multimedia que se usarán (texturas, modelos 3D, botones, audio, fuentes, vídeo). Este subsistema cumple también con la misión de de hacerlos accesibles al resto de subsistemas de la capa gráfica.

Las cachés de gestión de recursos dan soporte al subsistema de gestión de escena (sección 5.5.2), que agrupa dentro de si tanto al grafo de escena (sección 5.5.3) como al adaptador de nodos de escena (sección 5.5.2) y el sistema de renderizado (sección 5.5.7). El gestor de escena, se encarga simplemente de ordenar todos los recursos gráficos como paso previo a que se manden dibujar y de permitir el uso de un mismo recurso gráfico para la representación de múltiples entidades lógicas (sección 5.5.3). Es también este subsistema el que agrupa tanto los elementos 3D, como los 2D, como las cámaras y las luces, separándolos para una mejor gestión que además provoca un despliegue más eficiente (sección 5.5.7).

Aparte de estos subsistemas, la capa gráfica ofrece igualmente un sistema de reproducción de audio y otro de vídeo, que simplifican en gran medida las tareas de usar canciones, efectos sonoros o vídeo en un juego.

Por otro lado, se ha creado un completo sistema de procesamiento de eventos de usuario asíncrono (sección 5.5.11) que simplifica en gran medida la tarea de hacer que nuestra aplicación responda de forma correcta a la interacción del jugador.

La capa de control se encarga de gestionar todos los sistemas de la plataforma (persistencia, juego y representación, ver figura 5.1) y es la encargada de transmitir la entrada de usuario a la lógica de juego (sección 5.3). Al ser éste el punto de comienzo a la hora de desarrollar un juego con **YAOMEV**, se suministran los medios para poder crear controladores nuevos de forma asequible y rápida, gracias sobre todo a su implementación como máquina de estados, que convierte el diseño del comportamiento de la aplicación en una tarea muy sencilla.

Como complemento, se ha implementado una completa biblioteca matemática capaz de dar soporte a todo tipo de operaciones, desde la representación de gráficos 3D a las transformaciones en un espacio tridimensional (rotación, escalado, etc...). La biblioteca simplifica el uso de matrices de transformación, cuaternios, curvas splines cúbicas naturales y vectores de 2, 3 y 4 dimensiones.

También han sido desarrolladas otras utilidades como son temporizadores (sección 5.2.3), un creador de logs para la depuración (sección 5.2.2) y un generador de números aleatorios.

Estas herramientas son muy necesarias a la hora de desarrollar un juego.

El videojuego demostrador **Bit Them All!!!**, ha sido creado usando la plataforma **YAOMEV**. Es preciso apuntar, que este, es sólo una demo de un juego que continúa en desarrollo. Sin embargo, en su creación se han desarrollado todo tipo de herramientas que facilitan enormemente la tarea de ampliarlo.

Las primeras herramientas que deben ser mencionadas, son los scripts creados en Python para la exportación de personajes, modelos, escenarios y cámaras desde Blender a los formatos usados por **Bit Them All!!!**. Estos scripts permiten con un par de clicks de ratón usar estos *assets* gráficos en el juego. La creación de todos los *assets* gráficos del juego corrió a cargo de David Prieto Murillo [11], que ayudó con su trabajo enormemente a la finalización de este juego, y que vió simplificada, en gran parte, su tarea gracias a los scripts creados.

En la creación del juego, se constuyeron los controladores necesarios que permiten establecer una serie de pasos en el inicio y ejecución del juego, dividiendo su comportamiento en tres estados: estado de carga, cuando arranca el juego y debe leer los ficheros de configuración, cargar los *assets*; estado de menu, desde el que el jugador podrá acceder a diversas opciones de configuración; y por último el estado de juego que es donde ocurre toda la simulación del concurso televisivo.

Uno de los aspectos a resaltar en **Bit Them All!!!**, es la forma en que se han modelado los comportamientos. Mediante unas estructuras propias, se ha envuelto el uso de árboles de comportamiento, para facilitar el modelado de las reglas que guian el juego. Estas reglas son fácilmente modificables y extensibles gracias a la forma en que se crean los árboles de comportamiento.

En la implementación del presente proyecto se han tenido siempre en cuenta las buenas prácticas de la programación orientada objetos, así como principios como el de: “bajo acoplamiento y alta cohesión”, modularizando los componentes para lograr una mejor comprensión del código y usando patrones de diseño allí donde ha sido necesario. Ha sido necesario también necesario el uso de algunas técnicas de programación eficiente en C++ para el desarrollo.

9.2. Propuestas de trabajo futuro

Este proyecto puede ser la base para la creación de un entorno de desarrollo completamente libre que abarque todas las áreas que cubre el desarrollo de videojuegos, siendo a la vez posible, su uso para investigación y aprendizaje en la creación de videojuegos. De la misma forma, **Bit Them All!!!**, puede convertirse en un divertido y educativo entretenimiento, al que es posible añadir todavía muchas más opciones.

Como posibles líneas de desarrollo futuro, se realizan las siguientes propuestas:

- **Inclusión de técnicas avanzadas de programación gráfica:** Uno de los puntos fuertes de **YAOMEV** es aislar al programador de juegos de la dificultad que entraña programar la capa gráfica de un juego. Por lo tanto sería deseable incluir técnicas avanzadas de representación gráfica, como shaders, buffer objects, frustum culling, octrees o sombras. Como propuesta de trabajo se podría proponer el incluir estas técnicas envueltas en una capa de abstracción que facilite y haga sencillo su uso por parte del programador novel. El frustum culling, los octrees y la generación de sombras quedarían ocultos por la capa gráfica y el programador no tendría que saber cómo funcionan, símplemente que están activos o inactivos. Sin embargo, el uso de shaders implica aprender el lenguaje en que se programan y conocimientos específicos sobre el pipeline gráfico, por lo que se podría proponer una pequeña base de datos de shaders entre los que el programador podría elegir cuales usar.

  La inclusión de técnicas como frustum culling u octrees podría realizarse en unos 10 días. Sería necesario cambiar la clase encargada del grafo de escena e incluir en ella los algoritmos de particionamiento espacial necesarios. Se podría usar el patrón *Strategy* para tener una familia de grafos de escena que implementen una interfaz única, luego mediante una factoría de grafos de escena se podría elegir cual se quiere usar.

  El añadido de sombras dinámicas requeriría la modificación de la representación interna de los modelos 3D para incluir las estructuras necesarias para implementar el algoritmo correspondiente. Esta tarea podría requerir de unos 10 días.

  La inclusión de shaders requeriría montar la infraestructura de los mismos desde cero

y luego crear la base de datos. Esta tarea se podría realizar en unos 20 días. Por último el añadido del dibujado de las entidades gráficas mediante buffer objects requeriría la creación de un nuevo renderizador que se adaptara a las particularidades de esta técnica. En unos 15 días de desarrollo se podría tener implementado.

- **Creación de una comunidad de desarrollo: YAOMEV** pretende ser un punto de entrada hacia el mundo de la programación de videojuegos, por lo que sería interesante la creación de una comunidad de desarrolladores que permitiera continuar el desarrollo, ampliándo las características de la plataforma y solucionando errores. Esto además permitiría darlo a conocer para comprobar qué es capaz la gente de hacer con **YAOMEV**. La creación de una comunidad implicaría la creación de un blog desde el que se transmitirían las últimas noticias, de un foro donde la gente pudiera resolver sus dudas, de tutoriales para facilitar el comienzo de su uso y la creación de más ejemplos que demuestren lo que se puede conseguir. Por último se facilitaría la descarga del código y la posibilidad de contribuir a el.

- **Soporte para más formatos de representación de modelos 3D:** Como no se quieren limitar las posibilidades de **YAOMEV**, sería una interesante propuesta de trabajo incluir soporte para más metaformatos de definición de gráficos 3D. Para conseguir esto sería necesario crear una nueva forma de representar los modelos internamente que pudiera usar animación mediante huesos. Tambien habría que construir importadores específicos y sus cachés de recursos. Ejemplos de estos metaformatos podrían ser archivos MD5 o collada. Esta propuesta no conllevaría mucho tiempo de desarrollo, unos 15 días podrían ser suficientes para incluir el soporte para collada y MD5. Para conseguirlo habría que refactorizar las clases que representan los modelos 3D, creando un interfaz que haga transparente al nodo de escena correspondiente qué tipo de representación está usando exactamente. Después se podría crear una factoría que permitiera elegir qué tipo de representación se quiere usar.

- **Inclusión de soporte para el uso de físicas:** La inclusión de una capa de física en **YAOMEV**, otorgaría un extra a los juegos diseñados con el. Mediante el uso de la biblioteca libre *bullet*, y de la creación de los wrappers necesarios, se podría crear una capa

incluida en la lógica del juego, que facilitara su uso. En **YAOMEV** la representación interna de los modelos gráficos ya incluye la información sobre sus bounding boxes y bounding volumes, por lo que habría que crear las clases que envolvieran la representación física que use *bullet* y añadir un nuevo paso más en el bucle principal que ejecute la simulación física. Debido a que este sistema se crearía prácticamente desde cero, la dificultad para implementar esta propuesta sería intermedia, por lo que se estima que podría llevar unos 30 días incluirlo.

- **Inclusión de soporte para el uso de inteligencia artificial:** Sería muy interesante generalizar y extender el uso de estructuras clásicas de la IA en **YAOMEV**. Existen diversos sistemas que se podrían implementar para facilitar la creación de agentes inteligentes. En concreto, como propuesta, se podría añadir un sistema que facilitara la inclusión de mecanismos de percepción en las entidades lógicas, para que de esta manera las entidades pudieran usarlo como entrada para la toma de decisiones en base a los árboles de comportamiento creados.

- **Finalización del desarrollo de Bit Them All!!!:** Como última propuesta, se incluye la terminación del juego **Bit Them All!!!**. Ahora mismo sólo es una demo de las capacidades de **YAOMEV**, sin embargo cuenta con todas las estructuras necesarias para simplificar el añadido de más modos de juego. Por otro lado se podría incluir un mayor número de escenarios, personajes, etc...

  Gracias al uso de los árboles de comportamiento y a los nodos creados para representar las acciones, esta tarea se limitaría a la creación de los árboles específicos y de las clases que representan los diferentes modos. Dependiendo del número de modos de juego a implementar, esta tarea podría llevar unos 20 días.

9.3. Conclusión personal

Los Proyectos Fin de Carrera son la carta de presentación de un ingeniero en su camino hacia el mundo laboral y es por eso que desde un primer momento tuve muy claro sobre qué versaría el mío. Debido a que supone el primer enfrentamiento serio con problemas de

gran envergadura, que no solo requiere de prácticamente todas las competencias adquiridas durante el transcurso de la carrera, sino que supone también la oportunidad de acercarse a aquellos conocimientos que por los motivos que sean no se han podido conseguir.

Este proyecto, por lo tanto, ha supuesto para mí la oportunidad para acercarme a dos de las cosas que más me gustan de la informática: hacer la vida más fácil a los demás y aprender todo lo necesario sobre el desarrollo de videojuegos.

Y es que es un sueño habitual en muchos de los estudiantes que deciden matricularse en carreras como Ingeniería Informática el dedicarse al terminar al desarrollo de videojuegos. Es esta un área tremendamente interesante donde tienen cabida casi todos los campos que durante la carrera se estudian. Esto ha provocado, en mi caso particular, que la realización de este proyecto haya servido para relacionar y entender desde una perspectiva mucho más amplia el motivo por el cual se estudia lo que se estudia a lo largo de los diferentes cursos.

Este proyecto ha supuesto todo un desafío, al mismo tiempo que se convertía en toda una oportunidad. El ámbito multidisciplinar por el que se mueve ha logrado que descubriera lo mucho que me apasiona la informática. Quedándome en concreto con dos áreas: la inteligencia artificial y la informática gráfica. Campos a los que espero poder dedicar mucho tiempo en el futuro cercano. Y es que la sensación que uno experimenta cuando ve convertido su documento de concepto en imágenes interactivas es una de las más satisfactorias que creo que puede haber.

En definitiva, durante el desarrollo de **YAOMEV** y **Bit Them All!!!**, he tenido que enfrentarme a multitud de problemas que no están alejados de los que deberé afrontar en el mundo laboral y uno no puede sino darse cuenta de lo poco que sabe cuando empieza, y de lo mucho que ha aprendido cuando acaba. De todos los errores que he cometido durante el transcurso de la carrera, quizás el más grande fue no darle la importancia que se merece a materias como la Ingeniería del Software, pero sí de algo puedo sentirme orgulloso al terminar este proyecto es de haber puesto remedio a semejante imprudencia.

Apéndices

Apéndice A

Documento de Concepto

1. **Ficha Juego**

   Título: Bit Them All!!!

   Género: Concurso

   Plataforma: PC

   Modos: cuatro jugadores

   Edad: 16+

2. **Descripción**

   Sencillo juego de preguntas y respuestas al estilo de los concursos de la tele. La temática estará centrada en el mundo de la informática y su historia, aunque se incluirán también preguntas del ámbito de la cultura general.

   Cada jugador podrá controlar a uno de los cuatro concursantes a lo largo de las distintas pruebas existentes.

   En cada prueba, un presentador explicará las reglas y dará paso a las distintas preguntas, pudiendo estas ser de todo tipo (audio, vídeo, texto).

   El objetivo será conseguir llegar al final del concurso con más puntos que los demás.

3. **Ambientación**

   Te has presentado al famoso concurso **Bit Them All!!!**. Sientes la mirada del público sobre tu espalda, las luces del escenario te ciegan, las cámaras de TV te ponen algo nervioso, tus rivales parecen más tranquilos y seguros que tú, y el presentador está a punto de dar comienzo al concurso. ¿Realmente vas a permitir que los demás demuestren que saben más?

4. **Mecánicas centrales**

   - **Escenario:** Plató de TV. Se podrá elegir entre varios antes de comenzar el juego o dejar que se seleccione uno aleatoriamente.

- **Niveles:** Cada partida del concurso estará dividida en 5 pruebas diferentes, elegidas aleatoriamente, de entre todas las posibles (todas ellas permiten algunas variaciones en sus reglas). Las pruebas se irán sucediendo secuencialmente hasta llegar al final del concurso.
- **Configuración:** Antes de empezar el juego, se podrá elegir el nivel de dificultad de las preguntas y, en caso de no tener adversarios humanos, de los personajes no jugadores.
- **Turnos:** Cada prueba tiene su propia mecánica, a veces tocará responder por turnos y sólo responderá el más rápido.
- **Pruebas:**
  - **Normal:** Hay una batería de preguntas que pueden ser de uno o de diferentes tipos y cada jugador debe responder por turnos. Por cada pregunta se tiene un tiempo fijo para responder.
  - **Overclocking (Time Attack):** Se deben responder el mayor número de preguntas posibles en un determinado tiempo.
  - **CPU Caliente (Patata Caliente):** Aleatóriamente se elige al jugador que empezará con la CPU, entonces deberá responder a una pregunta, si acierta podrá elegir a quién le pasa la CPU y entonces será ese el que tenga que responder. No se podrá pasar la CPU hasta que se acierte la pregunta y pasado un tiempo la CPU estallará y el que la tenga en su poder perderá todos sus puntos.
  - **Buffer:** Cada jugador tendrá un buffer de preguntas a responder en un tiempo determinado. Si el jugador logra llevar un buen ritmo vaciando el buffer podrá elegir a otro jugador para llenarle su buffer con más preguntas. Ganará el que antes vacíe su buffer o el que más vacio lo tenga al final del tiempo.
  - **Ladrón de puntos:** Cualquiera de los modos anteriores pero con la posibilidad de robar puntos a los contrincantes.
  - **Acumulador de tiempo:** Se juega en modo normal o buffer pero sumando tiempo para las rondas que impliquen tiempo para responder.

- **Más rápido:** Preguntas con múltiples respuestas donde prima la velocidad por encima de todo.
- **Aún más rápido:** Hay un tiempo determinado en el que se lanzan preguntas a toda velocidad, sólo los más rápidos tendrán tiempo de contestar. Se acaba cuando un jugador haya llegado a una puntuación determinada o se acabe el tiempo.

6. Referencias

Saga Buzz! : Queda claro que es el principal inspirador del juego. La intención es tener un juego similar a pequeña escala y centrado en el mundo de la informática.

Apéndice B

Código fuente

Debido al tamaño del proyecto (más de 24.000 líneas de código, sin contar comentarios), no es viable incluirlo impreso como anexo. Se incluye un CD adjunto a este documento con el código fuente de **YAOMEV** y **Bit Them All!!!** , scripts, arte y ejecutable de **Bit Them All!!!**.

Estructura del proyecto **YAOMEV**:

- **/include:** Contiene las cabeceras de las bibliotecas estáticas usadas como apoyo: freetype2, FTGL, time y tinyxml.
- **/lib:** Contiene las bibliotecas estáticas descritas anteriormente.
- **/src:**
  - **/app:** Contiene la implementación genérica tanto de los controladores, como de la clase aplicación.
  - **/audio:** Contiene la implementación del sistema de audio.
  - **/event:** Contiene la jerarquía de eventos y listeners usados para capturar la interacción del usuario.
  - **/gui:** Contiene la implementación de los sistemas y componentes que se encargan de la representación gráfica: gestor de escena, nodos de escena, sistema de vídeo, renderers, etc...
  - **/logic:** Contiene las jerarquías de entidades, de componentes de interfaz y de aquellos nodos de los árboles de comportamiento genéricos. Incluye también las interfaces de los *Listeners* que deben implementar los componentes que deseen recibir actualizaciones del estado de la lógica.
  - **/math:** Contiene la biblioteca matemática.
  - **/persist:** Contiene la implementación de las cachés de recursos.
  - **/utilitys:** Contiene diferentes utilidades para ayudar a desarrollo: una implementación del patrón Singleton mediante templates, el log, diferentes métodos genéricos, etc...

Estructura del proyecto **Bit Them All!!!**:

- **/data:** Contiene los archivos de recursos (texturas, modelos en formato MD3, audio, vídeo), junto con los scripts, arte, fuentes y demás elementos utilizados como *assets* del juego.
  - **/audio:** Contiene los efectos, canciones y diálogos del juego.
  - **/export:** Contiene las texturas y modelos en formato MD3 usados en el juego.
  - **/graficos:** Contiene los ficheros .blend y los scripts usados para crear el arte del juego.
  - **/resources:** Contiene los ficheros de configuración XML que referencian los recursos para el juego.
  - **/scenarios:** Contiene los ficheros de configuración XML que indican cómo están construidos los escenarios.
- **/src:**
  - **/app:** Contiene la implementación de los controladores del juego.
  - **/logic:** Contiene la implementación de toda la lógica del juego, incluyendo los modos de juego controlados por árboles de comportamiento.
  - **/persist:** Contiene la implementación de los DAOs necesarios para acceder a la configuración del juego.

Bibliografía

[1] Ibis Capital. Global video games investment review. http://cmpmedia.vo.llnwd.net/o1/vault/gdceurope2010/slides/T_Merel_Business%20&%20Management_Global%20Video%20Games%20Investment.pdf, 2009. [Online; accedido Julio 2011].

[2] A. Cockburn. *Agile software development: the cooperative game*. The Agile software development series. Addison-Wesley, 2007.

[3] Tereixa Constenla. Un respeto para el videojuego. http://www.elpais.com/articulo/cultura/respeto/videojuego/elpepucul/20090326elpepicul_2/Tes, March 2009. [Online; accedido Julio 2011].

[4] Asociación Española de Distribuidores y Editores de Software de Entretenimiento (aDeSe). Balance económico de la industria del videojuego 2010. http://www.adese.es/pdf/informe_UAH.pdf, 2009. [Online; accedido Julio 2011].

[5] Europapress. Los videojuegos rivalizan con hollywood con una caja de 57600 millones de euros. http://www.hoytecnologia.com/noticias/videojuegos-rivalizan-Hollywood-caja/165569, 2009. [Online; accedido Julio 2011].

[6] J. Fonseca. Gprof2dot. http://code.google.com/p/jrfonseca/wiki/Gprof2Dot. [Online; accedido Septiembre 2011].

[7] Howard G. Ball. Telegames teach more than you think. *Computer-Aided Design*, pages 24–26, May 1978.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[9] N. Llopis. *C++ for game programmers*. Game development series. Charles River Media, 2003.

[10] Meristation. Análisis de Buzz! El Gran Reto. http://www.meristation.com/v3/todosobre.php?pic=PS2&idj=cw43d3a65c44a0f, 2006. [Online; accedido Julio 2011].

[11] D. P. Murillo-Rico. Generación de modelos, texturas y animaciones 3d para videojuegos multiplataforma. *UCLM*, June 2011.

[12] T. Reenskaug. Models-views-controllers. *Technical note, Xerox PARC, December*, 1979.

[13] T. Ryan. The anatomy of a design document, part 1: Documentation guidelines for the game concept and proposal, October 1999. [Online; accedido Julio 2011].

[14] B. Schwab. *AI game engine programming*. Course Technology, 2009.

[15] Ken Schwaber. Scrum development process. In *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA*, pages 117–134, 1995.

[16] Paul Verna. Global music: Tuning into new opportunities. `http://www.emarketer.com/Report.aspx?code=emarketer_2000428`, 2009. [Online; accedido Julio 2011].

[17] A. Méndiz y J. Pindado y J. Ruiz y J. M. Pulido. Videojuegos y educación. 3 - una revisión crítica de la investigación y la reflexión sobre la materia. `http://ares.cnice.mec.es/informes/02/documentos/iv04_0301a.htm`. [Online; accedido Julio 2011].

[18] L. Méndez y M. Alonso y H. del Castillo y S. Cortés y M. Checa y R. Hernández y A. Varela. Aprendiendo con los videojuegos comerciales: Un puente entre ocio y educación. `http://www.adese.es/pdf/informe_UAH.pdf`. [Online; accedido Julio 2011].