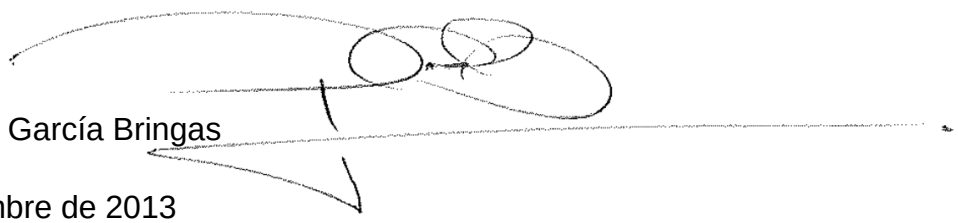
Deusto
Facultad de Ingeniería
Universidad de Deusto
Ingeniaritza Fakultatea
Deustuko Unibertsitatea

Ingeniero en Informática
Informatikako Ingeniaria

Proyecto fin de carrera
Karrera amaierako proiektua

Diseño e implementación de un motor de videojuegos 3D multiplataforma basado en tecnologías open-source

David García Miguel



Director: Pablo García Bringas

Bilbao, septiembre de 2013

Resumen

Este proyecto, de aquí en adelante Caelum-Engine, se compondrá del diseño y desarrollo de un motor de videojuegos 3D multiplataforma para Windows y Linux escrito principalmente en C++.

Todas las tecnologías que se van a utilizar serán open-source y con posibilidad para que pueda disponer de una licencia comercial a ser posible libre de cargos económicos (LGPL, Zlib, MIT, BSD etc.).

- Renderizado 3D: utilizando básicamente Ogre3D (soporte para opengl y directx) y diversos plugins y librerías para añadir funcionalidad y optimizaciones.

- Dispositivos de entrada: soporte para los dispositivos de entrada clásicos teclado + ratón y gamepad genéricos usb.

- Simulación Física: mediante bulletphysics, utilizada en películas y videojuegos a nivel profesional.

- Sonido: sonido ambiental y 3D utilizando OpenAL y códec Ogg vorbis.

- Red: permitiendo el juego multijugador en red.

Descriptores

Ingeniería del software, Tecnologías multimedia, Seguridad en las tecnologías de la información y comunicación, Inteligencia artificial.

Índice de contenido

Índice de figuras

Índice de pseudo-códigos

Índice de tablas

1. INTRODUCCIÓN

En el presente capítulo se va a presentar una introducción al proyecto de fin de carrera **"***Diseño e implementación de un motor de videojuegos 3D multiplataforma basado en tecnologías open-source"*. En éste capítulo se identificará una necesidad, se mostrará el problema identificado y se despejarán los motivos de la realización de éste proyecto. Para ello se ha dividido este capítulo en varias secciones.

- **Historia y tendencia**: En esta sección se comentará la problemática inicial de creación de videojuegos y el "nacimiento" de los primeros motores de videojuegos como respuesta a esa necesidad. Así mismo se comenta el creciente auge de los estudios de desarrollo independiente y el estancamiento progresivo de grandes compañías.

- **Presentación del problema**: Este apartado se menciona cómo la creación de un videojuego involucra gran cantidad de recursos y coste asociado. Se presenta también la necesidad de estudios independientes de mantener la innovación como marca de identidad y conseguir un cierto margen de beneficio.

- **Estado actual del entorno**: Breve análisis de los motores más conocidos hoy día. Se muestra cómo los motores de videojuegos actuales presentan la problemática descrita para los desarrolladores independientes y cómo Caelum-Engine pretende diferenciarse.

- **Solución propuesta**: Descripción de los objetivos y alcance del proyecto, junto con la descripción del producto final tras la realización del proyecto.

- **Presentación del documento**: Esta sección nos muestras una descripción de los capítulos de la presente memoria que servirá de guía para explicar el proceso de diseño e implementación de Caelum-Engine.

1.1 HISTORIA Y TENDENCIA

Modelo de negocio

Hasta hace poco, los motores de videojuegos eran creados por empresas profesionales dedicadas al sector orientados a juegos AAA, es decir de alto presupuesto y "categoría". Sin embargo en los últimos años este tipo de títulos están empezando a ver beneficios mermados.

Expongamos un caso práctico, en el año 2011 Ubisoft obtuvo un beneficio neto de 48 millones de dólares, pero para conseguirlo precisó de unas ventas de 1.4 billones de dólares, eso demuestra un margen de solo un 3% de beneficio. EA Games obtuvo, también en 2011, unos beneficios estimados de unos 76 millones de dólares con una recaudación de 977 millones, es decir, en torno al 7% de margen. Este tipo de juegos y compañías comparten algo en común, un modelo de negocio tradicional.

Sin embargo, aunque los beneficios en los grandes proyectos tienen un margen escaso, los mayores casos de éxito se empiezan a concentrar en juegos de especificaciones más bajas, con un coste menor y con un precio más asequible obteniendo grandes porcentajes de beneficio. Éste hecho ha abierto un panorama esperanzador para la creación de juegos independientes.

Indie gaming

En los últimos años y con la llegada de los nuevos terminales y sistemas operativos móviles como IOS y Android muchas compañías independientes se han lanzado al desarrollo de juegos con tecnologías emergentes como HTML5 y CSS, JavaScript etc. debido al bajo coste que supone desarrollar pequeños juegos móviles y los beneficios que aporta.

Para ilustrarlo con alguno ejemplo, un juego simple como el conocido “Angry Birds” que con un coste de producción realmente bajo (podría rondar las varias decenas de miles de euros), ha aportado a su compañía, Rovio Games, una recaudación de 48 millones de euros en 2011.

Son menos las compañías que se han lanzado al mundo del PC, pero entra las que lo han hecho y empiezan a hacerlo, se encuentran varios casos de éxito como “Minecraft” y “Torchlight” entre otros.

Además la plataforma de juegos *steam,* la más popular hoy día para PC, Mac y Linux, ofrece soporte para los desarrolladores independientes, para vender y distribuir sus juegos a través de la plataforma.

1.2 PRESENTACIÓN DEL PROBLEMA

El desarrollo de videojuegos es un proceso largo y costoso, que en la mayoría de los casos involucra cuantiosos recursos, tanto económicos como humanos. Para hacernos una idea general las grandes compañías de videojuegos llegan a gastar decenas (incluso a veces cientos) de millones de dólares en un solo título.

A través de los años y debido a que el desarrollo de videojuegos es una tarea que involucra tantos recursos, el concepto de “motor de videojuegos” surgió como una herramienta que respondía a la necesidad de englobar los procesos comunes a la creación de cualquier videojuego.

Con el objetivo de facilitar el desarrollo de videojuegos a desarrolladores independientes en PC, este proyecto implementa todo lo necesario para la realización de un videojuego comercial 3D multijugador.

1.3 ESTADO ACTUAL DEL ENTORNO

En esta sección se discutirán otras alternativas similares a Caelum-Engine contrastando las similitudes y diferencias con este proyecto.

1.3.1 Motores Open Source

A continuación se muestran algunos ejemplos de librerías 3D open-source.

- **Crystal Space**: Es un motor de desarrollo para aplicaciones 3D en general, no solo videojuegos [1]. Ofrece unos buenos resultados, pero sin embargo solo hay una escasa comunidad alrededor de éste proyecto. La documentación existente es muy escasa en comparación con otras librerías y carece de ejemplos prácticos, lo que provoca que la curva de aprendizaje inicial sea muy elevada.

Figura 1.1 - Imagen de Crystal Space

- **Cube 2**: Es un motor de videojuegos utilizado en el popular juego disponible para Linux, Sauerbraten. La mayor pega de este motor es que no es genérico, sino que está enfocado principalmente a la creación de FPS. Tampoco dispone de tutoriales o ejemplos aunque el código del propio juego Sauerbraten es abierto.

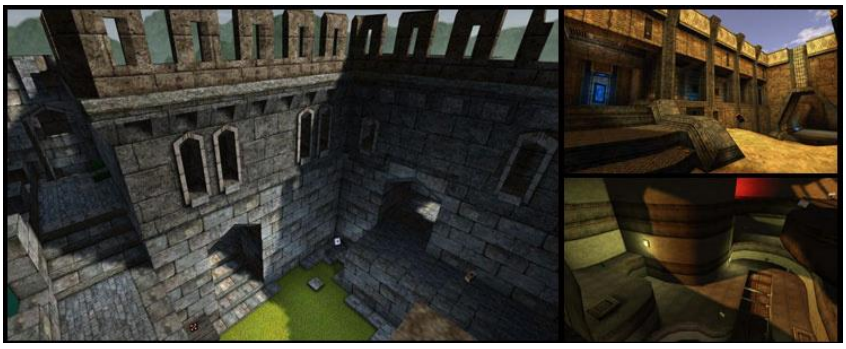


Figura 1.2 - Imágenes de Sauerbraten (realizado con Cube2)

- **Irrlicht Engine**: Es una de las opciones open-source más utilizadas y con una mayor comunidad a su alrededor [2]. Dispone de multitud de ejemplos, foro activo y ofrece una calidad decente, aunque por desgracia puede no ser suficiente para cumplir con los estándares gráficos que estamos acostumbrados a ver.

Figura 1.3 - Imagen de Irrlich Engine

1.3.2 Motores 3D propietarios

- **Unity 3D:** Este es el motor para equipos de desarrollo indie por excelencia hasta el momento [3]. Su diseño orientado a la creación de juegos con un editor, lo hace verdaderamente atrayente a los diseñadores y la simplicidad y calidad que proporciona son muy buenos incluso para juegos AAA. No obstante esto solo es posible con la versión de pago del motor, ya que la versión gratis carece de muchas características casi necesarias hoy día y solo es portable a Windows y Mac (para móviles navegador y consola, se necesita la versión de pago + extras). El coste total del paquete con todas las características que ofrece ronda los 6000$.

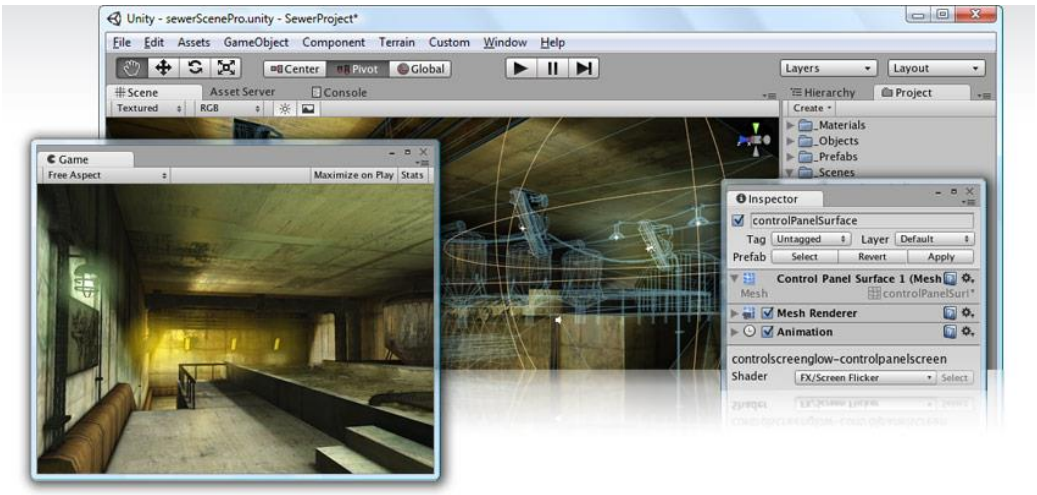


Figura 1.4 - Imagen del editor de Unity 3D

- **CryEngine**: Motor de última generación y con probablemente la mejor calidad existente en el mercado para juegos AAA [4]. Está orientado a la creación de juegos, tanto por programación (ofreciendo acceso al API C++) como por editor de juego, siendo una de las herramientas más flexibles y muy a considerar especialmente en la creación de juegos AAA.

Figura 1.5 - Imagen de CryEngine 3

Las pegas de éste motor vienen de la mano de los costes de licencia, que en su versión para desarrolladores independientes (la más económica) piden un 20% de la recaudación del juego (no del beneficio), dejando al estudio independiente con la única opción de proponer un margen del producto como mínimo de ese 20%, lo cual a veces no es una opción.

Otra de las grandes pegas de éste motor es que la versión para *indies* no contempla el desarrollo en consolas, solamente Windows. Sin contratar asesoramiento o instrucción se disponen de pocos ejemplos y recursos, especialmente para la parte de programación lo que provoca que la curva de aprendizaje sea excepcionalmente alta.

- **Unreal Engine**: Motor 3D de última generación y excelente calidad gráfica (AAA) [5]. Dispone de muchos ejemplos y comunidad alrededor de él, con un potente editor de juego con debugging visual y muchas características que facilitan la creación del juego enormemente.

Figura 1.6 - Imagen del editor de Unreal Development Kit

Una de sus mayores pegas está en que este motor solo es utilizable mediante el lenguaje de scripting creado específicamente para éste motor lo que le otorga una baja flexibilidad en el sentido de extensibilidad e innovación.

1.3.3 Diferenciación

La intención de Caelum-Engine es ofrecer un motor de videojuegos adecuado a los estudios de creación de juegos independientes. Uno de los principales "fallos" en las herramientas de hoy día es la combinación de una buena curva de aprendizaje y una flexibilidad adecuada para la innovación. Por lo tanto los pilares de este proyecto son:

- **Practicidad**: Los ejemplos prácticos y tutoriales son una de las mejores y más valiosas herramientas de las que se dispone al comienzo de un nuevo proyecto. Los ejemplos prácticos ayudan a familiarizarse con la funcionalidad básica y hacen mucho más sencilla la curva de aprendizaje.

- **Innovación**: Es una de las grandes demandas entre los desarrolladores independientes que buscan ganar al público mediante apuestas innovadores o arriesgadas, por encima de una calidad gráfica foto-realista. Es uno de los factores clave del éxito de los juegos independientes a los que Caelum-Engine puede ayudar con su diseño flexible y ampliable.

- **Flexibilidad**: El diseño flexible es uno de los pilares clave de un motor de videojuegos y altamente recomendable hoy día para el mantenimiento del proyecto a los largo de los años. Un diseño flexible permite que un motor pueda evolucionar a la par que las nuevas tecnologías se abren camino en la industria, lo que le permite mantener su competitividad y solidez. Además el hecho de que Caelum-Engine está basado en tecnologías libres con una gran comunidad detrás permite que el proyecto vaya creciendo a medida que estas tecnologías lo hacen.

- **Económico**: Debido a que Caelum-Engine está basado en tecnologías 100% libres y en su gran mayoría libres de cargos para licencia comercial, el coste se ha visto reducido enormemente en comparación con la creación de otros motores.

1.4 SOLUCIÓN PROPUESTA

1.4.1 Objetivos

Este proyecto se compone del diseño y desarrollo de un motor de videojuegos 3D.

El objetivo principal de este proyecto es facilitar el desarrollo de videojuegos 3D multiplataforma multijugador para Windows y Linux. Caelum-Engine no pretende ser un motor de videojuegos AAA, si no ofrecer una alternativa con un coste relativamente bajo, pero con un diseño lo suficientemente flexible para adaptarse a diferentes géneros de juego.

Todas las tecnologías que se van a utilizar serán open-source y con posibilidad para que Caelum-Engine pueda disponer de una licencia comercial a ser posible libre de cargos económicos (LGPL, Zlib, MIT, BSD etc.).

Hay varios aspectos generales a todo tipo de videojuegos que Caelum-Engine tendrá que solventar. La abstracción de gráficos 3d es probablemente la más importante y compleja de todas, ya que deberá ofrecer de una forma sencilla la problemática de dibujado 3D. La simulación de físicas realistas es otro de los factores que se deberán tener en cuenta en la realización del proyecto para que se considere satisfactorio. Además de todo se debe lograr que varios usuarios finales puedan compartir el juego en un mismo universo simulado con estas características, es decir, juego en red.

Todo esto deberá ir compaginado con una gestión eficiente de los recursos del PC para ofrecer un producto de calidad y competitivo.

1.4.2 Alcance del proyecto

Las áreas en las que Caelum-Engine ayudará en la creación de videojuegos serán:

- **Renderizado** 3D utilizando una arquitectura flexible que permite el cambiar entre diferentes sistemas de renderizado y esquemas de optimización.

- Soporte para una amplia gama de **dispositivos de entrada,** desde los dispositivos de entrada clásicos teclado y ratón, hasta gamepads y joysticks.

- **Simulación de físicas realista** que permita que los objetos del juego interaccionen entre ellos de forma lógica. Será necesaria una detección de colisiones entre objetos y simulación de fuerzas.

- **Reproducción de sonido** que permita dotar al juego de un cierto realismo y cinemática, pudiendo añadir música ambiental y efectos de sonido. Además será necesaria una gestión adecuada del almacenamiento de estos recursos.

- **Juego multijugador** que proporcionará una abstracción sobre las conexiones de red y facilitará la inmersión creando la ilusión de un mundo virtual compartido. La consistencia entre los puntos de vista de cada jugador debe garantizar que el mundo virtual está totalmente sincronizado para que los jugadores experimenten la ilusión de universo compartido. No obstante, es primordial que ninguno de ellos pueda alterar las reglas de este universo en su beneficio.

- **Inteligencia Artificial** necesaria para que los actores del juego tengan un mínimo conocimiento del entorno, como rutas hacia sus objetivos y acciones posibles a realizar en cada momento para cumplir sus objetivos.

1.4.3 Producto final

A la finalización del desarrollo de Caelum-Engine, se obtendrán los siguientes productos:

- Motor de videojuegos en forma de una o varias **librerías**
    - Multiplataforma: disponible para Windows y Linux.
    - Renderizado de objetos 3D y efectos de post-procesado.
    - Multijugador entre diferentes plataformas.
    - Simulación realista de físicas.
- **API** y **Manual de usuario**
    - Interfaz para la programación de juegos a través de los ficheros de cabecera que se incluirán en el proyecto.
    - Manual básico de usuario con una guía para la iniciación.
- **Programas de ejemplo**
    - Demostración de algunas funcionalidades principales del motor.
    - Aprendizaje de aspectos comunes a realizar en la creación de juegos con Caelum-Engine.

1.5 MODELO DE NEGOCIO

El modelo de negocio de Caelum-Engine es una parte muy importante de su concepción. En primer lugar se va a utilizar una doble licencia.

1.5.1 Licencia libre

En principio Caelum-Engine utilizará una **licencia libre GPL** (véase anexo B. Licencias), la cual se ha elegido por las siguientes razones:

- **Precio**: Ofrecer el motor gratis podría atraer en público y comunidad inicial.
- **Libertad de uso**: Cualquiera es libre de probarlo y modificarlo siempre y cuando acepte las cláusulas de la licencia.
- **Educación**: La licencia GPL solo permite crear proyectos con la misma licencia, lo que la hace perfecta para proyectos de uso no comercial y educativo.

- **Comunidad:** Ofrecer una versión libre del motor sirve para retribuir a la comunidad de software libre toda la ayuda obtenida para realizar el proyecto a través de las librerías utilizadas.

- **Feedback**: Al ofrecer el código del proyecto, la comunidad es capaz de probarlo y reportar fallos o incluso contribuir en el futuro desarrollo del proyecto.

1.5.2 Licencia Privativa

Existe otra licencia adecuada para quienes no deseen incurrir en las cláusulas de la licencia libre GPL. Esta licencia privativa resulta más apropiada para proyectos de uso comercial y no obliga al software derivado a licenciarse mediante la misma. *Esta licencia es intransferible.*

Las razones de utilización de esta licencia son meramente comerciales, ya que ayudan a retribuir económicamente al proyecto en caso de que se creen proyectos comerciales derivados.

El coste de esta licencia privativa será de 200€ (IVA no incluido) en el cual se incluyen 10 horas de asistencia remota gratuita.

**Nota: El precio y condiciones del contrato de licencia privativa estipuladas aún se encuentran bajo estudio y podrían variar en un futuro.*

1.5.3 Métodos de recaudación

Finalmente se han encontrado las siguientes formas para generar recaudación para el proyecto.

Venta de licencias privativas: En caso de realización de proyectos comerciales, se deberá contratar la licencia privativa del motor.

Planes de asistencia y formación: Próximamente se establecerán planes de formación y asistencia por horas y por plan de formación.

Solicitud de nuevas características: Consiste en que el motor está abierto a solicitudes para pedir nuevas características cuyo precio variará dependiendo de la característica a añadir. Las características solicitadas se incorporarán a la versión libre con un plazo aproximado de 6 meses.

Campañas crowdfunding: Para la solicitud algunas características generales se crearan campañas de crowdfunding para soportarlo. Algunas de estas campañas son por ejemplo la portabilidad del motor a Android o IOS.

1.6 PRESENTACIÓN DEL DOCUMENTO

En el documento que se les presenta a continuación se les mostrará el proceso de diseño y desarrollo del proyecto de fin de carrera titulado **"***Diseño e implementación de un motor de videojuegos 3D multiplataforma basado en tecnologías open-source"*

Este proyecto está realizado por David García Miguel, alumno de 5º de Ingeniería Informática.

El proyecto se ha realizado independientemente por el alumno, sin una colaboración directa con empresa y a partir de una idea propia del alumno presentada a Pablo García Bringas y Javier Nieves Acedo, director y codirector del proyecto respectivamente.

El presente documento consta de los siguientes capítulos:

- **Introducción**: En éste capítulo se introduce el documento de la memoria del proyecto, y la presentación del problema a solventar. Además se contempla la definición del proyecto, los objetivos a conseguir y las condiciones de ejecución para la realización del proyecto.

- **Planificación**: En éste apartado se definen la planificación de trabajo y tareas a realizar. Así como la selección y reparto de cargas de trabajo y estimación de tiempos para cada tarea.

- **Análisis de requisitos**: Documento detallado de funcionalidades que el proyecto debe ofrecer. Se hace distinción entre requisitos funcionales y no funcionales, e identifican los prioritarios. Ésta parte vendrá acompañada de criterios para la validación del cumplimiento de los requisitos.

- **Diseño del sistema**: En este capítulo se ilustra y describe el diseño de software planteado para el motor de videojuego propuesto y se justificarán las decisiones de diseño tomadas.

- **Consideraciones de implementación**: Este capítulo se centra en explicar los recursos técnicos empleados en la implementación del proyecto, el entorno de desarrollo utilizado.

- **Pruebas**: Esta sección describe las pruebas realizadas para la comprobación y correcta validación del funcionamiento del proyecto.

- **Manual de usuario**: Este documento recoge la forma de utilización del motor desde una perspectiva de usuario y se muestran todas las opciones disponibles de uso del motor así como las más usadas.

- **Conclusiones**: Para finalizar con la memoria del proyecto se exponen las conclusiones a la que se ha llegado tras la realización del proyecto y posibles líneas futuras interesantes a abordar.

2. PLANIFICACIÓN

Éste capítulo está dedicado a la organización y planificación del proyecto a lo largo del tiempo así como la gestión de los recursos y el tiempo disponible. Para explicar con detalle el proceso de planificación se ha dividido el capítulo en las siguientes partes.

- **Descripción de la realización**: Donde se describe el método de desarrollo elegido para el proyecto y la definición sobre los productos intermedios que se crearán antes de llegar al resultado final.

- **Tareas principales**: Desglose del proyecto en fases y tareas con una breve descripción de cada una de ellas.

- **Plan de trabajo**: División de las tareas elegidas a lo largo del tiempo creando un reparto de cargas de trabajo, una buena división de la precedencias de tareas y concurrencia posible entre las tareas y un diagrama de Gantt con la planificación de tiempos detallada.

- **Condiciones de ejecución**: Una descripción sobre los recursos necesarios para llevar a cabo el proyecto, tanto hardware, software como de lugar de trabajo. Se incluyen también los planes de gestión de incidencias.

- **Presupuesto**: Tarificación del trabajo y creación de un presupuesto orientativo.

2.1 DESCRIPCIÓN DE LA REALIZACIÓN

2.1.1 Método de desarrollo

En el desarrollo e implementación de este proyecto se han utilizado 2 metodologías distintas.

Fase 1 – Desarrollo del núcleo y sistema de juego

En la primera parte del desarrollo es necesario crear una base sólida para el proyecto, creando un núcleo sobre el que se sustentará el resto del software, por lo tanto para ésta primera iteración se utilizará un desarrollo tradicional o en cascada. En cada una de las iteraciones del desarrollo del núcleo no acabamos con un producto usable, si no que añadimos nuevos set de funcionalidad a un núcleo aún incompleto. Sólo una vez que todas las partes estén terminadas será funcional el núcleo.

El desarrollo del núcleo, proporcionará la base para el motor y es la parte más delicada para la fase de pruebas ya que no se dispone de un producto ejecutable sobre el que probarlo. En su lugar se realizarán pruebas a cada una de las funcionalidades independientemente.

El sistema de juego será el encargado de ofrecernos la definición de las capas o incrementos que se pueden añadir al motor posteriormente.

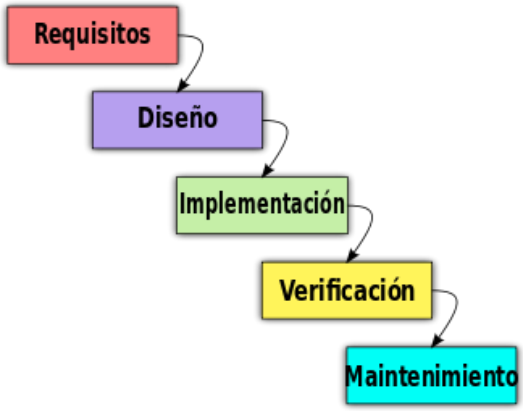


Figura 2.1 – Esquema de desarrollo en cascada

Fase 2 – Desarrollo de capas y componentes

Al terminar la primera fase obtendremos un núcleo con el que seguiremos trabajando y utilizaremos un método de desarrollo incremental donde incluiremos nuevas características a nuestro producto en cada iteración.

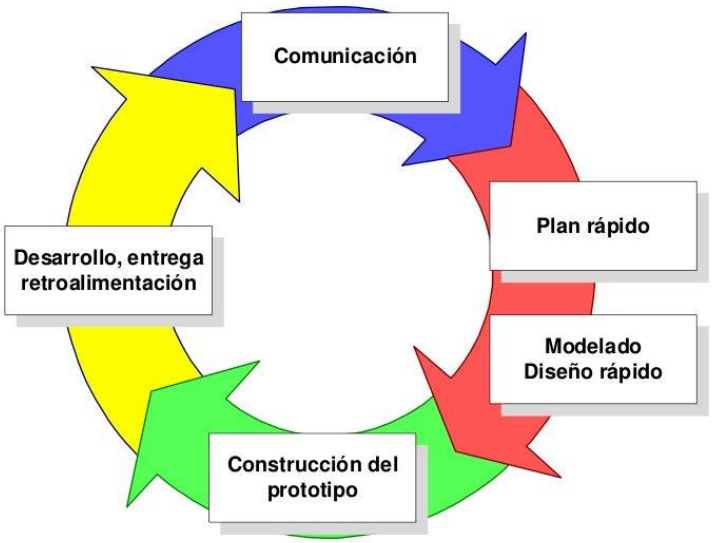


Figura 2.2 – Esquema de desarrollo incremental

La mayor ventaja del método de desarrollo incremental es que permite a los desarrolladores implementar una pequeña parte funcional y realizar las pruebas sobre ella en cada iteración, lo que resulta una forma muy efectiva de controlar la complejidad del proyecto, así como los riesgos.

2.1.2 Productos intermedios

Mediante la fase de diseño de Caelum-Engine se generarán diseños del sistema mediante diagramas UML. Lo que podría considerarse un producto asociado al desarrollo de la librería.

Durante esta fase de diseño se generarán principalmente diagramas de clases de la librería donde se podrá ver la relación entre las distintas partes del motor.

2.2 TAREAS PRINCIPALES

Fase Inicial

- **INI1** Selección de director de proyecto:
  Se buscará y seleccionará al director de proyecto más adecuado para el asesoramiento del proyecto.

- **INI2** Propuesta y reunión inicial:
  Reunión inicial con el director de proyecto para la propuesta del proyecto a desarrollar en el semestre.

- **INI3** Organización inicial:
  Reunión con el director de proyecto para determinar las tareas iniciales del proyecto y primeras consideraciones.

- **INI4** Fijación de plazos:
  Una vez determinadas las tareas realizar una estimación inicial del tiempo requerido por cada una y priorizarlas.

Fase de Investigación

- **INV1** Investigación de proyectos similares:
  Investigación sobre proyectos similares al que se va a realizar y búsqueda de alternativas.

- **INV2** Investigación de tecnologías de desarrollo:
  Investigación sobre tecnologías de software libre que posibiliten realizar el proyecto y a ser posible utilicen licencias no restrictivas (LGPL por encima de GPL)

- **INV3** Investigación de entornos de trabajo multiplataforma:
  Investigación sobre las formas de desarrollo multiplataforma posibles para motores realizados en detalle a bajo nivel de programación.

- **INV4** Investigación de arquitectura de otros *game-engine:*
  Investigación a fondo sobre el desarrollo de otros motores y patrones comunes en la realización de motores de videojuegos ya realizados para comparar las ventajas y desventajas de su arquitectura.

Fase preparación

- **PREP1** Instalación de herramientas de desarrollo:
  Instalación y prueba de funcionamiento de los programas y herramientas a utilizar durante el proyecto: librerías, IDE, SCV…

- **PREP2** Preparación del entorno de trabajo multiplataforma
  Preparación y configuración personalizada del entorno de trabajo o IDE y parametrizar un sistema de trabajo y compilación válido para las plataformas objetivo utilizando el sistema de compilación CMake.

- **PREP3** Preparación de un sistema de control de versiones
  Creación y personalización del sistema de control de versiones a utilizar que nos permita guardar una historia de la evolución del proyecto y volver a anteriores versiones en caso de error.

- **PREP4** Parametrización del IDE
  Parametrización del entorno de programación para adaptarse a la cadena de compilación multiplataforma y reglas de estilo.

Fase de análisis

- **ANA1** Definición de objetivos:
  Definición de objetivos a conseguir con el proyecto.

- **ANA2** Análisis de alternativas:
  Evaluación de alternativas ya disponibles y comparativa asociada.

- **ANA3** Análisis de requisitos
  Análisis de requisitos funcionales y no funcionales del proyecto.

- **ANA4** Validación de requisitos
  Creación de criterios de validación para los requisitos

Fase de diseño

- **DIS1** Diseño del núcleo del motor
  Diseño del sistema de ficheros y recursos del motor, gestión de errores y plugins externos así como del arranque y parada del motor.

- **DIS2** Diseño del modelo matemático
  Diseño de los tipos de datos que permitirán las transformaciones en el espacio 3D de nuestros objetos de juego.

- **DIS3** Diseño del sistema de juego

    - **DIS3.1** Gestión de estados de juego
      Diseño de un sistema para separar los distintos tipos de gestión de eventos basado en estados del juego.

  - **DIS3.2** Sistema de escenas
    Diseño del tipo de escena que representará el espacio 3D y la jerarquía de objetos.

  - **DIS3.3** Sistema de capas y componentes
    Diseño del sistema de extensión de escenas. Las capas representan un nivel de abstracción para otros sistemas como render, sonido, o físicas. Los componentes representan acoples, creados por las capas, que podrán añadirse a los objetos de juego y sincronizarse con ellos.

- **DIS4** Diseño del sistema de entrada
  Diseño del sistema de entrada de eventos de dispositivos externos.

- **DIS5** Diseño del sistema de render
  Diseño del sistema de dibujado. Diseño de la capa para escena y componentes.

- **DIS6** Diseño del sistema de sonido
  Diseño del sistema de reproducción y localización de sonido. Diseño de la capa y componentes de la escena. Diseño del sistema de códec.

- **DIS7** Diseño del sistema de físicas
  Diseño del sistema de simulación de físicas realista para cuerpos rígidos.

- **DIS8** Diseño del sistema de red
  Diseño del sistema de comunicación de red y diseño del protocolo de red a utilizar.

Fase de implementación y pruebas

- **IMP1** Implementación y prueba del núcleo

  - **IMP1.1** Gestor de memoria
    Desarrollo de un gestor de memoria para la eficiencia de utilización de recursos del juego.

  - **IMP1.2** Gestor de ficheros/recursos
    Implementación de un gestor de ficheros basado en las tecnologías utilizadas que reconozca los tipos de extensión de ficheros utilizados y los cargue adecuadamente.

  - **IMP1.3** Gestor de registro de logs
    Gestor de excepciones y errores, que almacenará las incidencias ocurridas para posterior información y respuesta.

  - **IMP1.4** Gestor de plugins
    Gestor de un sistema de carga de plugins tanto del motor como de los sistemas internos.

  - **MP1.5** Fachada del sistema
    Implementación de la fachada global del motor que permita su inicialización, cierre, extensión y acceso.

- **IMP2** Implementación y prueba del modelo matemático
  Implementación del modelo de datos que soporte transformaciones en el espacio 3D, matrices, vectores y cuaterniones.

- **IMP3** Implementación y prueba del sistema de gestión de juego

  - **IMP3.1** Gestor de estados de juego
    Gestión de la máquina de estados de juego y la clase abstracta para los estados de juego.

  - **IMP3.2** Gestor de escenas
    Creación de un gestor de escenas que permita almacenar escenas y crear nuevas.

  - **IMP3.3** Escena y GameObject
    Implementación de la clase de escena que gestione el espacio 3D y la creación de objetos de juego y su jerarquía.

  - **IMP3.4** Capas y gestores de capas
    Creación de la abstracción para nuevas capas que añadir a las escenas.

  - **IMP3.5** Componentes y gestores de componentes
    Creación de la abstracción para los componentes creados por las capas y su sincronización con los objetos de juego.

- **IMP4** Implementación y prueba del sistema de renderizado

  - **IMP4.1** Gestor de capa y capa de renderizado
    Implementación de la fachada del sistema de renderizado general (gestor) y del tipo de capa que albergará la escena.

  - **IMP4.2** Componente: Cámara
    Implementación del componente de cámara que podrá adjuntarse a los objetos de juego.

  - **IMP4.3** Componente: Entidad Gráfica (modelo3d)
    Implementación del tipo de datos para contener modelos 3D que podrán adjuntarse a los objetos de juego.

  - **IMP4.4** Componente: Luz
    Implementación del componente de luz y los distintos tipos de éstas como puntos focos y luces direccionales.

  - **IMP4.5** Componente: Sistema de partículas
    Implementación del componente para añadir sistemas de partículas a la escena.

- **IMP4.6** Gestor de ventanas y eventos de ventana
  Implementación del gestor de ventana del sistema de renderizado y los eventos disponibles.

- **IMP4.7** Sistema de Interfaz de usuario
  Implementación y fachada para la creación de interfaces de usuario.

- **IMP4.8** Gestor de texturas y materiales
  Creación de un gestor para simplificar el uso de recursos 3D y sus materiales.

- **IMP4.9** Gestor de iluminación dinámica y sombras
  Gestor para la iluminación dinámica del juego y sombras producidas por dicha iluminación.

- **IMP4.10** Gestor de "shading"
  Gestor para los efectos de luz sobre el sombreado (no sombras) y comportamiento de la luz sobre objetos.

- **IMP4.11** Sistema de animación
  Sistema de animación de esqueleto utilizando cinemática directa.

- **IMP4.12** Sistema de animación basada en puntos de trayectoria
  Sistema de animación para objetos de juego utilizando puntos clave o keyframes.

- **IMP4.13** Renderizado de entornos abiertos
  Sistema de renderizado de terrenos basados en mapas de altura y texture splatting.

- **IMP4.14** Sistema de post-procesado de imagen
  Sistema de efectos especiales de post-procesado tales como bloom saturación etc.

- **IMP5** Implementación y prueba del sistema de entrada

  - **IMP5.1** Gestor basado en eventos
    Implementación de un gestor de entrada basado en eventos con buffer.

  - **IMP5.2** Soporte teclado + ratón
    Implementación del modelo de datos y captura de eventos de teclado y ratón

  - **IMP5.3** Soporte para gamepads y joysticks
    Implementación del modelo de datos y captura de eventos de gamepads y joysticks genéricos.

- **IMP6** Implementación y prueba del sistema de sonido

  - **IMP6.1** Gestor de capa y capa de sonido

Implementación de la fachada y la capa de gestión de sonido asociada a la escena.

- **IMP6.2** Componente: Emisor
  Implementación del componente que realiza el papel de emisor de sonido.

- **IMP6.3** Componente: Receptor
  Implementación del componente que realiza el papel de receptor.

- **IMP6.4** Implementación de códecs
  Implementación de la extensión de gestión de memoria para cargar codificaciones de audio.

- **IMP7** Implementación y prueba del sistema de simulación física

  - **IMP7.1** Gestor de capa y capa de simulación física
    Implementación de la fachada del sistema de físicas y la capa de simulación de la escena junto con su sincronización.

  - **IMP7.2** Implementación de debugger visual
    Implementación de un visualizador de mallas de colisión para debugging en tiempo real.

  - **IMP7.3** Implementación del conversor de mallas de colisión
    Implementación de un conversor que convierte los modelos 3D a mallas de colisión convexas.

  - **IMP7.4** Componente: Cuerpo rígido
    Implementación del componente que dota a los objetos de juego de simulación realista gestionando fuerzas choques etc.

  - **IMP7.5** Componente: Controlador de personaje
    Implementación de un componente similar al objeto rígido pero con funciones especiales para la gestión de personajes.

  - **IMP7.6** Simulación adecuada a red
    Implementación de métodos para garantizar simulaciones deterministas.

- **IMP8** Implementación y prueba del sistema de red

  - **IMP8.1** Gestor de capas de red
    Implementación de la fachada de comunicación de red.

  - **IMP8.2** Capa de red servidora
    Implementación de la capa de escena que gestiona automáticamente la sincronización de objetos de juego entre los clientes.

  - **IMP8.3** Capa de red cliente

Implementación de la capa de escena que gestiona automáticamente la serialización de objetos y actualización de red.

- **IMP8.4** Serialización de objetos
  Implementación de la abstracción para la serialización de objetos que determina la información clave a sincronizar de cada objeto.

- **IMP8.5** Componente: Objeto de red
  Implementación del componente de objeto de red que será serializado y sincronizado.

- **IMP8.6** Optimizaciones de simulación en red
  Implementación de optimizaciones varias para la simulación en tiempo real y disminución de factores retardo en la red.

- **IMP9** Implementación y prueba de la Inteligencia Artificial

  - **IMP9.1** Máquina de estados finita
    Implementación de una máquina de estados finita genérica.

  - **IMP9.2** Árbol de estados y comportamiento
    Implementación de árboles de comportamiento basados en máquinas de estado.

- **IMP10** Implementación y prueba de las demos
  Implementación de la demo final de presentación.

Fase de finalización y verificación

- **FIN0** Entrega de la documentación
  Verificación de la memoria del proyecto con el director.

- **FIN1** Presentación del proyecto
  Presentación y defensa del proyecto ante el tribunal.

2.3 PLAN DE TRABAJO

A continuación se muestra un plan de trabajo diseñado con la herramienta de software libre **"planner"** para planificación de proyectos software.

El equipo de trabajo con el que se ha realizado la planificación supone X roles, pero únicamente una persona para realizarlos.

Por motivos de conveniencia, las jornadas diarias se han supuesto de 4 horas diarias durante los días laborables (lunes a viernes).

A la vez que se describen las tareas se han ido asignando los presupuestos de cada una de las fases para ofrecer una estimación del presupuesto inicial detallado.

El plan de trabajo con las tareas a realizar es el siguiente:

*Aviso: Todas las duraciones de esfuerzo están acotadas a la jornada laborar especial aplicada para éste caso que es de 4 horas al día de lunes a viernes. Por lo tanto si una tarea tiene un esfuerzo de 2d 3h significa que la tarea dura (2*4)+3=11 horas*

WBS	Name	Start	Finish	Work	Duration	Cost
1	**PFC**	**Sep 21**	**Sep 6**	**264d**	**250d**	**31,915**
1.1	**Fase Inicial**	**Oct 8**	**Nov 7**	**2d 3h**	**22d 3h**	**550**
1.1.1	Selección de director de proyecto	Oct 8	Oct 8	1h	30min	50
1.1.2	Propuesta y reunion inicial	Oct 22	Oct 22	2h	1h	100
1.1.3	Organización inicial	Oct 29	Oct 29	3h	3h	150
1.1.4	Fijación de plazos	Nov 7	Nov 7	1d 1h	3h 20min	250
1.2	**Fase de investigación**	**Sep 21**	**Nov 1**	**30d**	**30d**	**3,600**
1.2.1	Investigación de proyectos similares	Sep 21	Sep 25	3d	3d	360
1.2.2	Investigación de tecnologias de desarrollo	Sep 26	Oct 16	15d	15d	1,800
1.2.3	Investigación de entornos de multiplataforma	Oct 17	Oct 18	2d	2d	240
1.2.4	Investigación de arquitectura de game-engines	Oct 19	Nov 1	10d	10d	1,200

Figura 2.3 - Plan de trabajo de las fases inicial e investigación

WBS	Name	Start	Finish	Work	Duration	Cost
1.3	**Fase de preparación**	**Nov 2**	**Nov 29**	**20d**	**20d**	**2,400**
1.3.1	Instalación de herramientas de desarrollo	Nov 2	Nov 22	15d	15d	1,800
1.3.2	Preparación de entorno de trabajo	Nov 23	Nov 27	3d	3d	360
1.3.3	Sistema de control de versiones	Nov 28	Nov 28	1d	1d	120
1.3.4	Parametrización de IDE	Nov 29	Nov 29	1d	1d	120
1.4	**Fase de análisis**	**Nov 7**	**Nov 19**	**8d**	**8d**	**960**
1.4.1	Definición de objetivos	Nov 7	Nov 9	2d	2d	240
1.4.2	Análisis de alternativas	Nov 9	Nov 13	2d	2d	240
1.4.3	Análisis de requisitos	Nov 13	Nov 16	3d	3d	360
1.4.4	Validación de requisitos	Nov 16	Nov 19	1d	1d	120

Figura 2.4 - Plan de trabajo de las fases de preparación y análisis

WBS	Name	Start	Finish	Work	Duration	Cost
1.5	**Fase de diseño**	**Nov 19**	**Jan 11**	**39d**	**39d**	**4,680**
1.5.1	Diseño del nucleo del motor	Nov 19	Nov 26	5d	5d	600
1.5.2	Diseño del modelo matemático	Nov 26	Nov 30	4d	4d	480
1.5.3	**Diseño de sistema de juego**	**Nov 30**	**Dec 12**	**8d**	**8d**	**960**
1.5.3.1	Gestion de estados de juego	Nov 30	Dec 4	2d	2d	240
1.5.3.2	Sistema de escenas	Dec 4	Dec 6	2d	2d	240
1.5.3.3	Sistema de capas y componentes	Dec 6	Dec 12	4d	4d	480
1.5.4	Diseño del sistema de entrada	Dec 12	Dec 18	4d	4d	480
1.5.5	Diseño del sistema de render	Dec 18	Dec 25	5d	5d	600
1.5.6	Diseño del sistema de sonido	Dec 25	Dec 28	3d	3d	360
1.5.7	Diseño del sistema de físicas	Dec 28	Jan 3	4d	4d	480
1.5.8	Diseño del sistema de red	Jan 3	Jan 11	6d	6d	720

Figura 2.5 - Plan de trabajo de la fase de diseño

WBS	Name	Start	Finish	Work	Duration	Cost
1.6	**Fase de implementación y pruebas**	**Jan 11**	**Aug 29**	**164d**	**164d**	**19,680**
1.6.1	**Imp. y prueba del nucleo del sistema**	**Jan 11**	**Jan 29**	**12d**	**12d**	**1,440**
1.6.1.1	Imp. gestor de memoria	Jan 11	Jan 16	3d	3d	360
1.6.1.2	Imp. gestor de ficheros/recursos	Jan 16	Jan 21	3d	3d	360
1.6.1.3	Imp. gestor de logs	Jan 21	Jan 23	2d	2d	240
1.6.1.4	Imp. gestor de plugins	Jan 23	Jan 25	2d	2d	240
1.6.1.5	Imp. de fachada del sistema	Jan 25	Jan 29	2d	2d	240
1.6.2	Imp. y prueba del modelo matemático	Jan 29	Feb 4	4d	4d	480
1.6.3	**Imp. y prueba del sistema de juego**	**Feb 4**	**Feb 25**	**15d**	**15d**	**1,800**
1.6.3.1	Imp. del gestor de estados de juego	Feb 4	Feb 6	2d	2d	240
1.6.3.2	Imp. del gestor de escenas	Feb 6	Feb 11	3d	3d	360
1.6.3.3	Imp. de escena principal y gameobject	Feb 11	Feb 18	5d	5d	600
1.6.3.4	Imp. de capas y gestores de capas	Feb 18	Feb 20	2d	2d	240
1.6.3.5	Imp. de componentes y gestores de compon	Feb 20	Feb 25	3d	3d	360

Figura 2.6 - Plan de trabajo de la fase de implementación (parte 1)

WBS	Name	Start	Finish	Work	Duration	Cost
1.6.4	**Imp. y prueba del sistema de renderizad**	**Feb 25**	**May 3**	**49d**	**49d**	**5,880**
1.6.4.1	Gestor de capa y capa de renderizado	Feb 25	Feb 27	2d	2d	240
1.6.4.2	**Componentes móviles**	**Feb 27**	**Mar 13**	**10d**	**10d**	**1,200**
1.6.4.2.1	Componente: Cámara	Feb 27	Mar 1	2d	2d	240
1.6.4.2.2	Componente: Entidad Gráfica (modelo3D)	Mar 1	Mar 6	3d	3d	360
1.6.4.2.3	Componente: Luz	Mar 6	Mar 7	1d	1d	120
1.6.4.2.4	Componente: Sistema de partículas	Mar 7	Mar 13	4d	4d	480
1.6.4.3	Gestor de ventanas + eventos de ventana	Mar 13	Mar 15	2d	2d	240
1.6.4.4	Sistema de GUI	Mar 15	Mar 25	6d	6d	720
1.6.4.5	Gestor de texturas y materiales	Mar 25	Mar 27	2d	2d	240
1.6.4.6	Gestor de iluminación dinámica y sombras	Mar 27	Apr 2	4d	4d	480
1.6.4.7	Gestor de "shading"	Apr 2	Apr 5	3d	3d	360
1.6.4.8	**Sistema de animación**	**Apr 5**	**Apr 12**	**5d**	**5d**	**600**
1.6.4.8.1	Basado en esqueleto (Cinemática directa)	Apr 5	Apr 10	3d	3d	360
1.6.4.8.2	Basado en puntos de trayectoria	Apr 10	Apr 12	2d	2d	240
1.6.4.9	Renderizado de entornos abiertos: terreno	Apr 12	Apr 24	8d	8d	960
1.6.4.10	Sistema de post-procesado de imagen	Apr 24	May 3	7d	7d	840
1.6.5	**Imp. y prueba del sistema de entrada**	**May 3**	**May 14**	**7d**	**7d**	**840**
1.6.5.1	Imp. del gestor basado en eventos	May 3	May 8	3d	3d	360
1.6.5.2	Soporte para teclado + ratón	May 8	May 10	2d	2d	240
1.6.5.3	Soporte para gamepads y joysticks	May 10	May 14	2d	2d	240

Figura 2.7 - Plan de trabajo de la fase de implementación (parte 2)

WBS	Name	Start	Finish	Work	Duration	Cost
1.6.6	**Imp. y prueba del sistema de sonido**	**May 14**	**May 31**	**13d**	**13d**	**1,560**
1.6.6.1	Gestor de capa y capa de sonido	May 14	May 17	3d	3d	360
1.6.6.2	Componente: emisor	May 17	May 22	3d	3d	360
1.6.6.3	Componente: receptor	May 22	May 24	2d	2d	240
1.6.6.4	Imp. de codecs (.ogg)	May 24	May 31	5d	5d	600
1.6.7	**Imp. y prueba del motor de físicas**	**May 31**	**Jul 2**	**22d**	**22d**	**2,640**
1.6.7.1	Gestor de capa y capa de simulación física	May 31	Jun 5	3d	3d	360
1.6.7.2	Imp. de debugger visual	Jun 5	Jun 12	5d	5d	600
1.6.7.3	Imp. de conversol de mallas de colisión	Jun 12	Jun 17	3d	3d	360
1.6.7.4	Componente: Cuerpo rígido	Jun 17	Jun 20	3d	3d	360
1.6.7.5	Componente: Controlador de personaje	Jun 20	Jun 26	4d	4d	480
1.6.7.6	Simulación adecuada a red	Jun 26	Jul 2	4d	4d	480

Figura 2.8 - Plan de trabajo de la fase de implementación (parte 3)

WBS	Name	Start	Finish	Work	Duration	Cost
1.6.8	**Imp. y prueba del sistema de red**	**Jul 2**	**Aug 8**	**27d**	**27d**	**3,240**
1.6.8.1	Imp. gestor de capas de red	Jul 2	Jul 5	3d	3d	360
1.6.8.2	Imp. capa de red servidora	Jul 5	Jul 11	4d	4d	480
1.6.8.3	Imp. capa de red cliente	Jul 11	Jul 23	8d	8d	960
1.6.8.4	Serialización de objetos	Jul 23	Jul 26	3d	3d	360
1.6.8.5	Componente: objeto de red	Jul 26	Jul 31	3d	3d	360
1.6.8.6	Optimizaciones de simulación en red	Jul 31	Aug 8	6d	6d	720
1.6.9	**Imp. y prueba de la Inteligencia Artificial**	**Aug 8**	**Aug 16**	**6d**	**6d**	**720**
1.6.9.1	Imp. de maquina de estados finito	Aug 8	Aug 12	2d	2d	240
1.6.9.2	Imp. de arbol de estados y comportamiento	Aug 12	Aug 16	4d	4d	480
1.6.10	Imp. y prueba de las demos	Aug 16	Aug 29	9d	9d	1,080

Figura 2.9 - Plan de trabajo de la fase de implementación (parte 4)

WBS	Name	Start	Finish	Work	Duration	Cost
1.7	**Fase de finalización y verificación**	**Aug 30**	**Sep 6**	**1h 30mir**	**5d**	**45**
1.7.1	Entrega de la memoria	Aug 30	Aug 30	1h	1h	30
1.7.2	Presentación del proyecto	Sep 6	Sep 6	30min	30min	15

Figura 2.10 - Plan de trabajo de la fase de finalización

2.4 DIAGRAMA DE GANTT

Para ilustrar las tareas a realizar a lo largo del tiempo y la distribución de éstas entre los diferentes roles se ha realizado un diagrama de Gantt. Debido a que el tamaño del diagrama es bastante grande se presenta también a continuación una versión simplificada orientativa de cada una de las fases del proyecto.

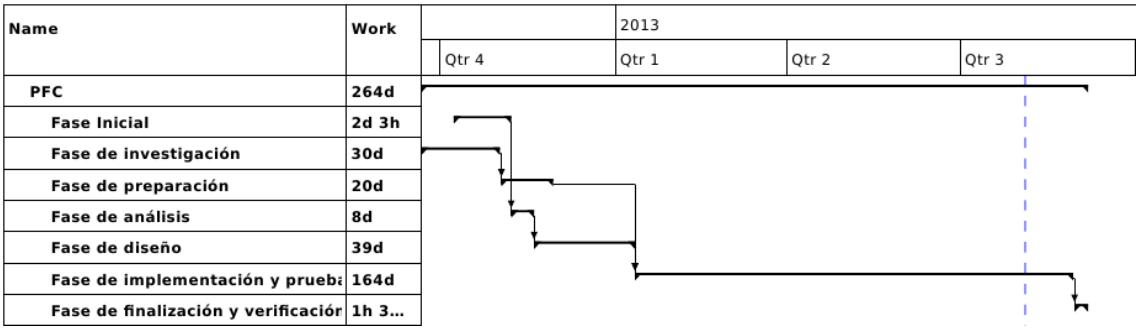


Figura 2.11 - Diagrama de Gantt de las fases del proyecto

Una vez visto el diagrama simplificado de la Figura 2.11 se muestra a continuación la planificación temporal en detalle.

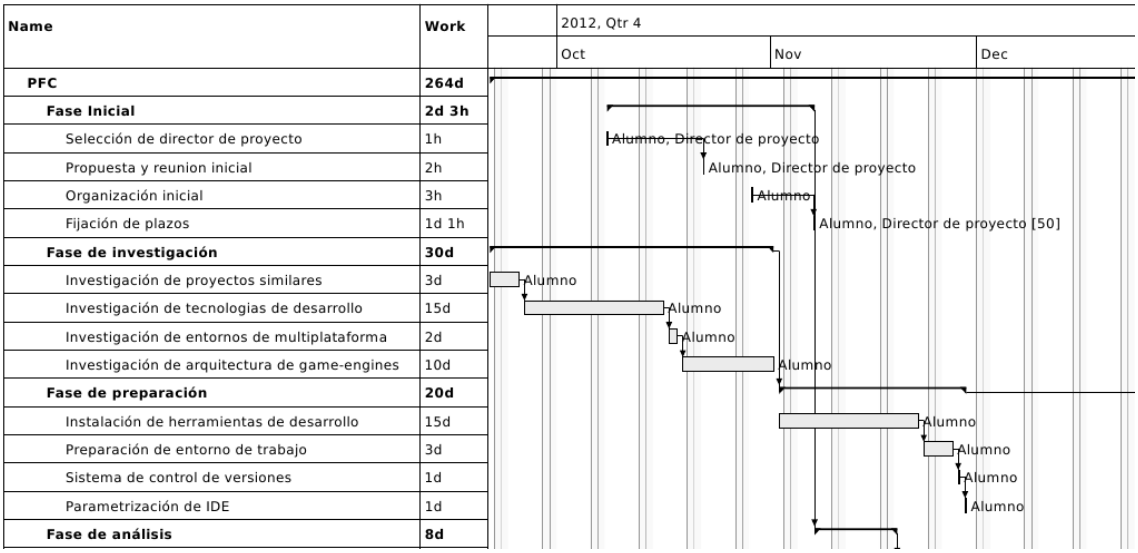


Figura 2.12 - Diagrama de gantt de las fases inicial, investigación y preparación

Name	Work
Fase de análisis	**8d**
Definición de objetivos	2d
Análisis de alternativas	2d
Análisis de requisitos	3d
Validación de requisitos	1d
Fase de diseño	**39d**
Diseño del nucleo del motor	5d
Diseño del modelo matemático	4d
Diseño de sistema de juego	**8d**
Gestion de estados de juego	2d
Sistema de escenas	2d
Sistema de capas y componentes	4d
Diseño del sistema de entrada	4d
Diseño del sistema de render	5d
Diseño del sistema de sonido	3d
Diseño del sistema de físicas	4d
Diseño del sistema de red	6d
Fase de implementación y pruebas	**164d**

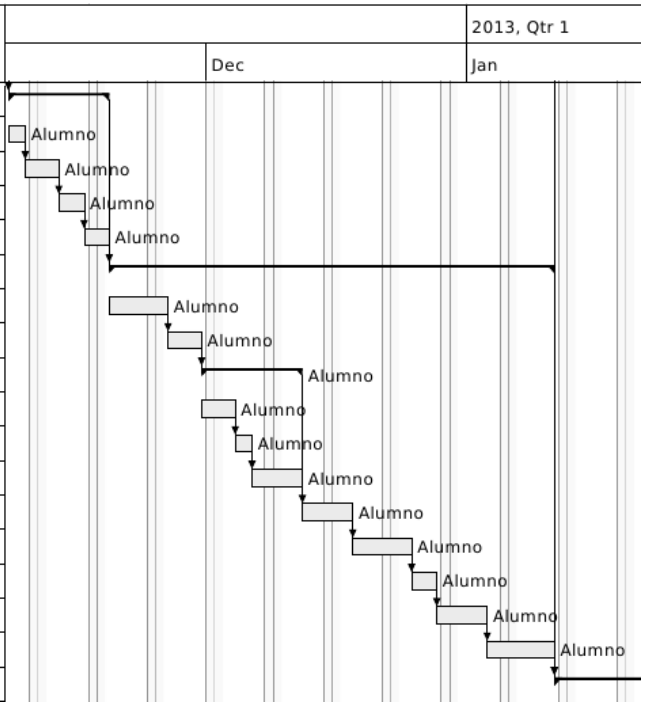


Figura 2.13 - Diagrama de gantt de las fases de análisis y diseño

Name	Work
Fase de implementación y pruebas	**164d**
Imp. y prueba del nucleo del sistema	**12d**
Imp. gestor de memoria	3d
Imp. gestor de ficheros/recursos	3d
Imp. gestor de logs	2d
Imp. gestor de plugins	2d
Imp. de fachada del sistema	2d
Imp. y prueba del modelo matemático	4d
Imp. y prueba del sistema de juego	**15d**
Imp. del gestor de estados de juego	2d
Imp. del gestor de escenas	3d
Imp. de escena principal y gameobject	5d
Imp. de capas y gestores de capas	2d
Imp. de componentes y gestores de componentes	3d
Imp. y prueba del sistema de renderizado	**49d**

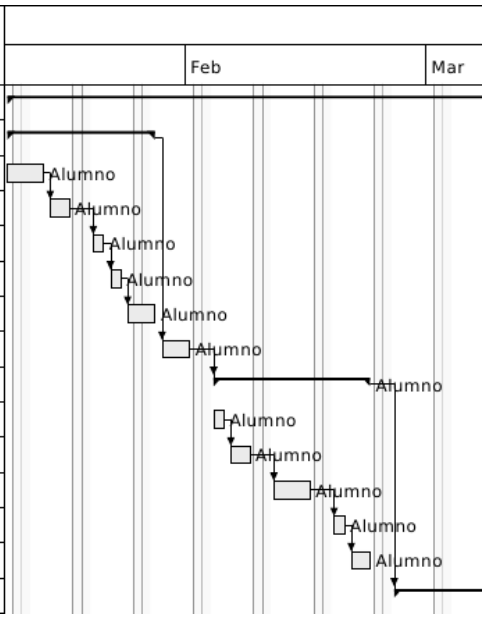


Figura 2.14 - Diagrama de gantt de la implementación y pruebas (núcleo y sistema de juego)

Name	Work
Imp. y prueba del sistema de renderizado	**49d**
Gestor de capa y capa de renderizado	2d
Componentes móviles	**10d**
Componente: Cámara	2d
Componente: Entidad Gráfica (modelo3D)	3d
Componente: Luz	1d
Componente: Sistema de partículas	4d
Gestor de ventanas + eventos de ventana	2d
Sistema de GUI	6d
Gestor de texturas y materiales	2d
Gestor de iluminación dinámica y sombras	4d
Gestor de "shading"	3d
Sistema de animación	**5d**
Basado en esqueleto (Cinemática directa)	3d
Basado en puntos de trayectoria	2d
Renderizado de entornos abiertos: terreno	8d
Sistema de post-procesado de imagen	7d
Imp. y prueba del sistema de entrada	**7d**

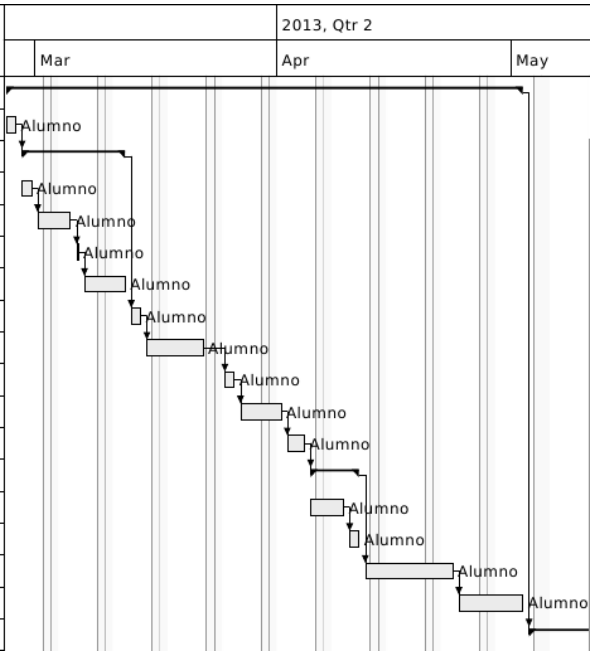


Figura 2.15 - Diagrama de gantt de la implementación y pruebas (sistema de renderizado)

Name	Work
Imp. y prueba del sistema de entrada	**7d**
Imp. del gestor basado en eventos	3d
Soporte para teclado + ratón	2d
Soporte para gamepads y joysticks	2d
Imp. y prueba del sistema de sonido	**13d**
Gestor de capa y capa de sonido	3d
Componente: emisor	3d
Componente: receptor	2d
Imp. de codecs (.ogg)	5d
Imp. y prueba del motor de físicas	**22d**
Gestor de capa y capa de simulación física	3d
Imp. de debugger visual	5d
Imp. de conversol de mallas de colisión	3d
Componente: Cuerpo rígido	3d
Componente: Controlador de personaje	4d
Simulación adecuada a red	4d
Imp. y prueba del sistema de red	**27d**

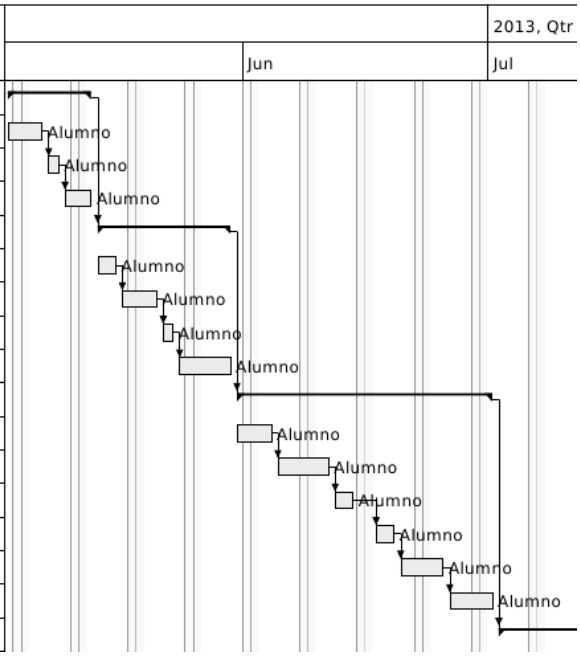


Figura 2.16- Diagrama de gantt de la implementación y pruebas (input, sonido y físicas)

Name	Work
Imp. y prueba del sistema de red	**27d**
Imp. gestor de capas de red	3d
Imp. capa de red servidora	4d
Imp. capa de red cliente	8d
Serialización de objetos	3d
Componente: objeto de red	3d
Optimizaciones de simulación en red	6d
Imp. y prueba de la Inteligencia Artificial	**6d**
Imp. de maquina de estados finito	2d
Imp. de arbol de estados y comportamiento	4d
Imp. y prueba de las demos	9d
Fase de finalización y verificación	**1h 3…**
Entrega de la memoria	1h
Presentación del proyecto	30min

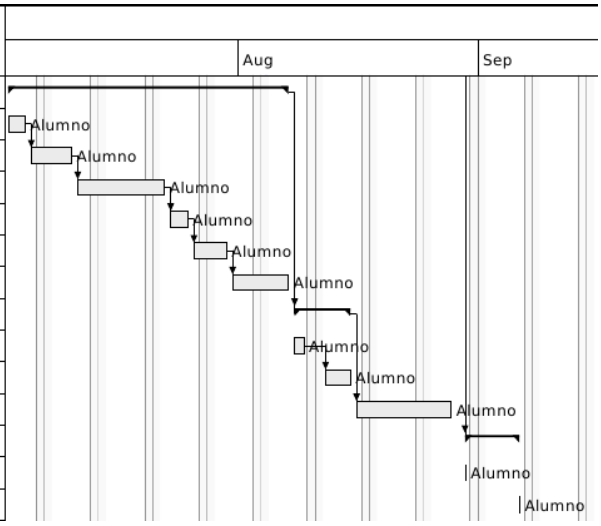


Figura 2.17- Diagrama de gantt de la implementación y pruebas (red e inteligencia artificial)

2.5 CONDICIONES DE EJECUCIÓN

2.5.1 Entorno de trabajo

El trabajo de este proyecto será desarrollará en la Universidad, biblioteca de Deusto y la propia casa del alumno. El trabajo en la universidad y la biblioteca res realizarán durante las horas libres entre las clases de la propia universidad.

Para la implementación de este proyecto se necesitará proveer los siguientes recursos que conseguirá la empresa/alumno que realiza el proyecto. Principalmente debido a que los requisitos presentados aquí son de uso común, ya se poseen de antemano.

HARDWARE

- Ordenador portátil Intel Core i5-3337U 1,8Ghz con gráfica Nvidia 625M
- Ordenador sobremesa Intel Core Quad con gráfica Ati HD6870 con Windows 7 y Debian Testing
- GamePad genérico para PC

SOFTWARE

- Sistemas Operativos, Windows 7/8 y Debian Testing
- Librerías externas
    - Ogre3D, librería encargada de aportar los gráficos 3D y una base para la gestión de recursos.

  - Paged Geometry, plugin para la gestión eficiente de múltiples instancias de objetos estáticos en entornos abiertos.
  - SkyX, plugin para la simulación realista de fenómenos atmosféricos, ciclos día y noche, movimiento de nubes y tormentas.
  - Hydrax, plugin para la simulación de océanos dinámicos y gestión de refracciones bajo el agua.
  - MyGUI, librería de gestión de interfaz de usuario adaptable a varios sistemas de renderizado.
  - OIS, librería dedicada al soporte de dispositivos de entrada teclado ratón y joysticks
  - Bullet, librería de simulación de físicas utilizada comercialmente en películas y videojuegos
  - OpenAL, librería de sonido ambiental 3D
  - Ogg Vorbis, códec de audio necesario para reducir el tamaño de los ficheros de audio

- Entorno de desarrollo
  - Cmake, sistema de compilación y generación de proyectos multiplataforma.
  - GNUMake, sistema de compilación C++.
  - QtCreator, editor de texto y entorno de desarrollo multiplataforma.
  - Git, sistema de control de versiones.
  - Cpplint, programa de comprobación de normas de estilo de google.
  - Cuenta en Bitbucket para crear un repositorio git remoto privado.

2.5.2 Control de cambios

En caso de necesidad de cambio de alguna de las partes se dispondrá a seguir el siguiente protocolo:

1. Comunicación al equipo de trabajo y al director de proyecto del cambio o incidencia producido.

2. Estudio de la incidencia y valoración del impacto sobre el proyecto.

3. En caso de aprobación del cambio, modificación de los pertinentes presupuestos, tareas y plan de trabajo involucrados.

2.5.3 Recepción de productos

Todos los documentos y productos realizados durante el desarrollo del proyecto deberán ser entregados al director de proyecto para la evaluación y aprobación pertinente.

El protocolo a realizar para la aprobación de las diferentes entregas es el siguiente:

1. Después de cada entrega se dispone de un plazo de 1 semana para notificar la aprobación de la entrega.

2. En caso de no recibir notificaciones al respecto se supondrá que la entrega se entiende aprobada automáticamente.

3. Cualquier cambio necesario al respecto fuera del plazo de una semana deberá ser acogido bajo el plan de control de cambios e incidencias.

2.6 PRESUPUESTO

Dependiendo de las fases del proyecto se han utilizado diferentes perfiles profesionales, por lo tanto su coste es variable en función de estos perfiles. A continuación se presenta el recuento de horas trabajadas por cada perfil profesional y su tarificación.

Perfil	Trabajo(h)	Coste(€/h)	Importe(€)
Jefe de proyecto	11	50	550
Ingeniero analista	44	30	1.320
Ingeniero de diseño	256	30	7.680
Programador	745’5	30	22.365
TOTAL			*31.915*

Tabla 2.1 - Presupuesto de RRHH

Además de los costes por recursos humanos se han requerido ciertas herramientas y dispositivos para la realización del proyecto, cuyo coste exponemos a continuación:

Artículo	Unidades	Precio(€)	Amortización	Importe(€)
Portátil	1	750	4	187,5
Ordenador fijo	1	1.000	6	166,7
Gamepad genérico	1	30	5	6
TOTAL				*360,2*

Tabla 2.2- Presupuesto de recursos hardware

Debido a que solo hemos utilizado el hardware durante 1 año para realizar el proyecto, solamente imputaremos los gastos del 1 año.

Tipo de coste	Importe(€)
Recursos Humanos	31.915
Hardware	360,2
TOTAL	*32.275,2*

Tabla 2.3 – Resumen de presupuesto

3. ANÁLISIS DE REQUISITOS

En el presente capítulo se pretende hacer un análisis en detalle de los requisitos que Caelum-Engine debe conseguir para considerarse finalizado con éxito.

Debido a que no existe un único cliente potencial y el proyecto trata de satisfacer una necesidad de mercado no disponemos de una única fuente de información. Puesto que el mercado de los videojuegos y motores de videojuego está en cierta medida bien establecido podemos extraer gran parte de estos requisitos del sistema de nuestros competidores directos.

Otras fuentes de información provienen de la extracción de las necesidades en conjunto de artistas y desarrolladores de videojuegos que permitan a ambos una interacción sencilla y una fácil comunicación.

Se ha dividido este capítulo en varias secciones:

- **Requisitos funcionales**: recoge todas las funcionalidades específicas que el motor debe cumplir, describiendo el comportamiento del motor pero sin entrar en detalle de sus restricciones.

- **Requisitos no funcionales**: requisitos que especifican criterios para juzgar el funcionamiento del sistema y que no describen el funcionamiento del motor en sí mismo.

- **Criterios de validación**: consiste en una serie de pruebas de caja negra cuya aprobación demuestra la conformidad de los requisitos entregados.

3.1 REQUISITOS FUNCIONALES

Se presenta a continuación la lista de requisitos funcionales obtenidos:

- **RF1** Requisitos del núcleo:

    - **RF1.1** El motor debe ser capaz de gestionar y controlar el uso de memoria dinámica reservada.

    - **RF1.2** El motor debe ofrecer un sistema de carga de recursos o ficheros multiplataforma que debe ser extensible a nuevos tipos de ficheros.

    - **RF1.3** El motor debe ofrecer una forma de registrar sus acciones y además debe ser extensible.

    - **RF1.4** El motor debe ser capaz de cargar y descargar plugins en tiempo de ejecución.

- **RF2** Requisitos del modelo matemático:

  - **RF2.1** El motor debe ser capa de definir ángulos, tanto en radianes como en grados y definir conversiones entre ambos y sus operaciones básicas.
  - **RF2.2** El motor debe ser capaz de representar vectores de 2 y 3 dimensiones y definir operaciones básicas entre vectores.
  - **RF2.3** El motor debe ser capaz de representar matrices de 3x3 y definir sus operaciones comunes.
  - **RF2.4** El motor debe ser capaz de representar cuaterniones o cuaternios y realizar operaciones básicas entre ellos.
  - **RF2.5** El motor debe ser capaz de definir transformaciones espaciales.
  - **RF2.6** El motor debe ser capaz de realizar operaciones entre los distintos tipos de datos propuestos, como conversiones, cálculos de tangentes, normales etc.

- **RF3** Requisitos del sistema de entrada:
  - **RF3.1** El motor debe ser capaz de detectar el eventos del ratón del pc
  - **RF3.2** El motor debe ser capaz de detectar eventos del teclado del pc.
  - **RF3.3** El motor debe ser capaz de detectar eventos de mandos (gamepads genéricos) conectados por usb.
- **RF4** Requisitos del sistema de sonido:
  - **RF4.1** El motor debe ser capaz de reproducir sonido ambiental.
  - **RF4.2** El motor debe ser capaz de reproducir sonido posicional en 3D.
  - **RF4.3** El motor debe ser capaz de decodificar al menos un formato de audio.
- **RF5** Requisitos del sistema de renderizado:
  - **RF5.1** El motor debe ser capaz de cargar texturas en diversos formatos comunes (psd, jpeg, png, tga)
  - **RF5.2** El motor debe ser capaz de cargar y ejecutar shaders (tanto en GLSL, HLSL como en CG)
  - **RF5.3** El motor debe ser capaz de cargar y ejecutar scripts para la definición de materiales.
  - **RF5.4** El motor debe ser capaz de cargar modelos 3D y sus respectivos esqueletos para animaciones.

- **RF5.5** El motor debe permitir la animación tanto de transformaciones espaciales (traslación, rotación y escalado) como animación basada en esqueleto con cinemática directa.

- **RF5.6** El motor debe permitir crear y utilizar cámaras con diferentes tipos de proyección y campo de visión.

- **RF5.7** El motor debe permitir de utilizar distintos tipos de luces (puntuales, focales y direccionales)

- **RF5.8** El motor debe ser capaz de calcular las sombras producidas por los modelos 3D tanto estática, como dinámicamente.

- **RF5.9** El motor debe ser capa de utilizar los shaders para calcular la incidencia de la luz en los modelos 3D.

- **RF5.10** El motor debe ser capaz de ofrecer un sistema sencillo para crear e interactuar con interfaces gráficas de usuario.

- **RF6** Requisitos del sistema de simulación física:

  - **RF6.1** El motor debe ser capaz de representar mallas de colisión.

  - **RF6.2** El motor debe ser capaz de crear mallas de colisión a partir de primitivas

  - **RF6.3** El motor debe ser capaz de crear mallas de colisión a partir de modelos 3D.

  - **RF6.4** El motor debe ser capaz de crear mallas de colisión completamente convexas a partir de modelos 3D (no convexos).

  - **RF6.5** El motor debe ser capaz de representar cuerpos rígidos y asociarles mallas de colisión.

  - **RF6.6** El motor debe ser capaz de aplicar fuerzas e impulsos sobre cuerpos rígidos.

  - **RF6.7** El motor debe ser capaz de calcular colisiones entre mallas de colisión asociadas a cuerpos.

  - **RF6.8** El motor debe ser capaz de calcular la recuperación de colisiones entre cuerpos rígidos involucrados en una colisión.

  - **RF6.9** El motor debe ofrecer controladores para el manejo de cuerpos rígidos especiales (p.e. un personaje que sube automáticamente alturas de tamaño de un escalón)

- **RF7** Requisitos del sistema de red:
    - **RF7.1** El motor debe ofrecer un servidor encargado de enviar y sincronizar datos entre varios clientes.
    - **RF7.2** El motor debe ofrecer un cliente capaz de conectar a un servidor y sincronizarse con el servidor.
    - **RF7.3** El motor debe poder responder a los eventos producidos en cada cliente actualizando al resto de clientes.
    - **RF7.4** El motor debe realizar predicciones para mantener la fluidez de la simulación.

3.2 REQUISITOS NO FUNCIONALES

- **RNF1** Generales
    - **RNF1.1** El motor debe ser multiplataforma funcionando al menos bajo Windows 7,8 y Debian (Linux).
    - **RNF1.2** El motor debe ofrecer opciones feedback frente a errores mediante logs y avisos.
- **RNF2** Extensibilidad
    - **RNF2.1** El sistema de recursos debe permitir desarrollador cargar sus propios tipos de ficheros.
    - **RNF2.2** El sistema de capas debe permitir añadir nuevos tipos de capas y componentes al sistema.
    - **RNF2.3** El sistema debe permitir la carga de plugins tanto del propio sistema como de los desarrolladores que utilicen el motor.
- **RNF3** Rendimiento
    - **RNF3.1** El motor debe ofrecer una tasa de actualización aproximada de 30 fotogramas por segundo para la demo de presentación y preferiblemente de 60 fotogramas con el portátil especificado en la sección 2.5.1.
    - **RNF3.2** El motor debe ofrecer una simulación de red con latencia < 80ms para redes locales o conexiones directas.
- **RNF4** Seguridad

- **RNF4.1** El servidor de red debe ser autoritativo, es decir, el servidor es quien tiene la última decisión sobre las acciones de los clientes y quien contiene la única versión completamente válida de la simulación red.

- **RNF4.2** Todas las entradas de usuario a través del sistema de GUI y de ficheros deben estar saneadas.

3.3 CRITERIOS DE VALIDACIÓN

Los criterios de validación se realizan mediante pruebas o la superación de unos hitos. Una vez aceptados éstos, suponen la aprobación de los requisitos inicialmente presentados.

En el caso de Caelum-Engine y debido a la naturaleza del proyecto las pruebas suponen una parte fundamental en el proceso y deben realizarse de manera continua para garantizar la estabilidad del sistema.

Por lo tanto los criterios de validación de Caelum-Engine se entienden como la superación de las pruebas presentadas en el capítulo 6.

4. DISEÑO DEL SISTEMA

En este capítulo se expone el diseño propuesto para el sistema, así como las decisiones más importantes tomadas a lo largo del proceso y el porqué de estas haciendo un breve análisis de los pros y contras de cada decisión.

Para esbozar y entender el diseño de cada sección dividiremos el contenido de cada sección en dos fases:

1. **Especificación de las funciones** y objetivos generales que la sección ha de cumplir.

2. **Presentación del diseño**: Se presentan las decisiones de diseño tomadas y los motivos de la elección. Se muestra el diseño propuesto en Caelum-Engine y se explica su funcionamiento.

En ocasiones se ha decidido realizar un pequeño análisis previo del diseño de otros motores ya existentes para contrastar los beneficios e imposiciones de las soluciones ya existentes. De esta manera podemos analizar las debilidades y fortalezas en el diseño de nuestros competidores y tratar de superarlos.

**Nota: La mayoría de diagramas de clases están enormemente simplificados con el objetivo de centrar la atención en el diseño y ante todo hacer que sean legibles y quepan en una página.*

4.1 VISIÓN DEL DISEÑO GLOBAL

Para comenzar a explicar el diseño global del sistema parece interesante dividir en varias secciones el diseño, en este caso dependiendo de su funcionalidad. Esta división del diseño en secciones ayuda a mantener el desarrollo y mantenimiento del motor bien definido y sostenible. A continuación se explica el proceso de decisión llevado a cabo para la división de los diferentes sistemas del motor:

- **Diseño del núcleo del motor**: En primer lugar necesitaremos una sección del proyecto donde centralizar la forma de arranque y parada de todos los demás subsistemas y utilidades de uso general para el motor.

- **Diseño del modelo matemático**: Diseño del modelo matemático que soportará el sistema de juego 3D y con el que se podrán definir posteriormente las transformaciones espaciales de nuestros objetos de juego.

- **Diseño del sistema de juego**: Se trata de la parte más importante en términos de diseño del motor. Se propone el diseño de las escenas de juego y el soporte de “capas” de abstracción para cada uno de los subsistemas como dibujado, físicas, red etc. Éste diseño es la base de integración de todos los demás subsistemas.

- **Diseño del sistema de entrada**: Encargado de gestionar la entrada de los dispositivos externos para interactuar con el motor.

- **Diseño del sistema de *render***: Este subsistema es el encargado del dibujado de los modelos 2d y 3d. Presenta una extensión del motor como capa de juego y componentes asociados.

- **Diseño del sistema de sonido**: Dedicado a decodificar y reproducir los sonidos del juego. Extiende el sistema de recursos y el sistema de capas y componentes.

- **Diseño del sistema de físicas**: Se encargará de las colisiones y reacciones dinámicas entre los cuerpos rígidos.

- **Diseño del sistema de red**: Sección para la serialización de los elementos del juego y comunicación de estos a través de la red.

A continuación se presenta un diagrama de clases que presenta la estructura de las diferentes clases del motor sin entrar en detalle. En las siguientes secciones iremos seccionando y explicando las diferentes partes:

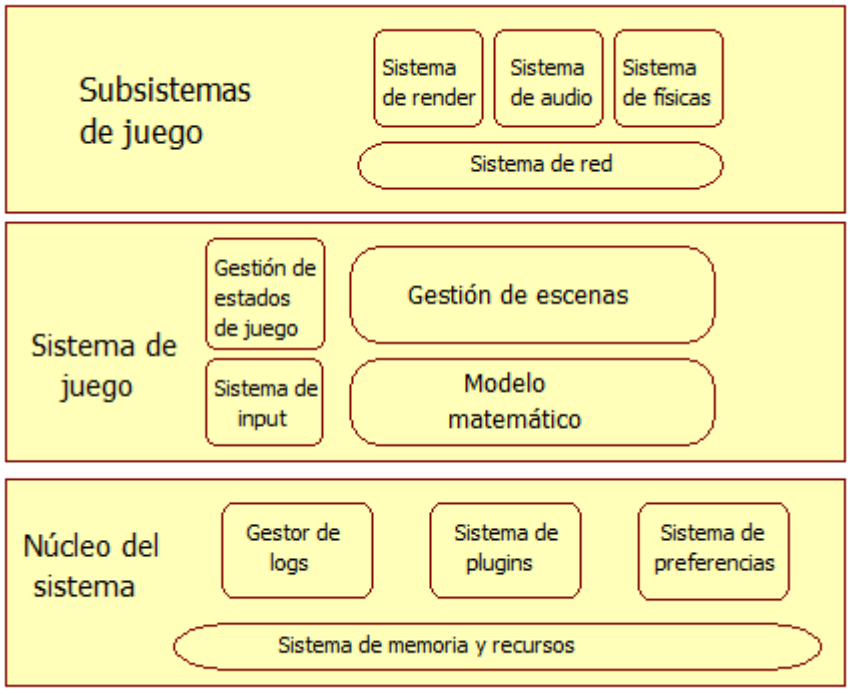


Figura 4.1 - Diagrama de jerarquía de sistemas

4.2 DISEÑO DEL NÚCLEO DEL MOTOR

Función

El núcleo principal del motor debe centralizar una serie de funciones y utilidades ampliamente utilizadas en todo el sistema. Algunos ejemplos de esta funcionalidad son:

- Crear una capa de abstracción multiplataforma:

  En principio para Windows y Linux. A pesar de que el alcance del proyecto está planteado para estas 2 plataformas todas las librerías son portables también a Mac y Android. Por lo tanto el sistema es actualmente portable a Windows y Linux pero queda abierta una posibilidad de portabilidad a Mac y Android con relativamente pocos cambios.

- Carga en tiempo real de *plug-ins*:

  Debido a que cada uno de los subsitemas del motor, como por ejemplo el de físicas o dibujado tienen opción para cargar plugins necesitamos proveer una forma única de abstraer la carga de todos los plugins disponibles.

- Crear un entorno de gestión de errores y fallos:

  Para lograr esto se creará un sistema de logs que registre las acciones del motor y a la vez sirve para que los usuarios del motor puedan registrar su propios cambios en tiempo real.

- Proporcionar un sistema de carga de recursos extensible

  Se debe ofrecer una forma de cargar los recursos externos de juego que funcione a su vez en todas las plataformas. Es probablemente la función más importante del núcleo del motor y es muy importante que sea extensible ya que el programador de juegos que utiliza el motor debe poder implementar fácilmente la forma de cargar tipos de recursos no nativos para el motor.

Diseño

La decisión de las funciones y diseño general del núcleo han sido inspirada por el estudio de varios ejemplos prácticos y análisis de las soluciones existentes [6]. A continuación se presenta un diagrama de clases que presenta un esquema de diseño general y muy simplificado del núcleo del sistema:

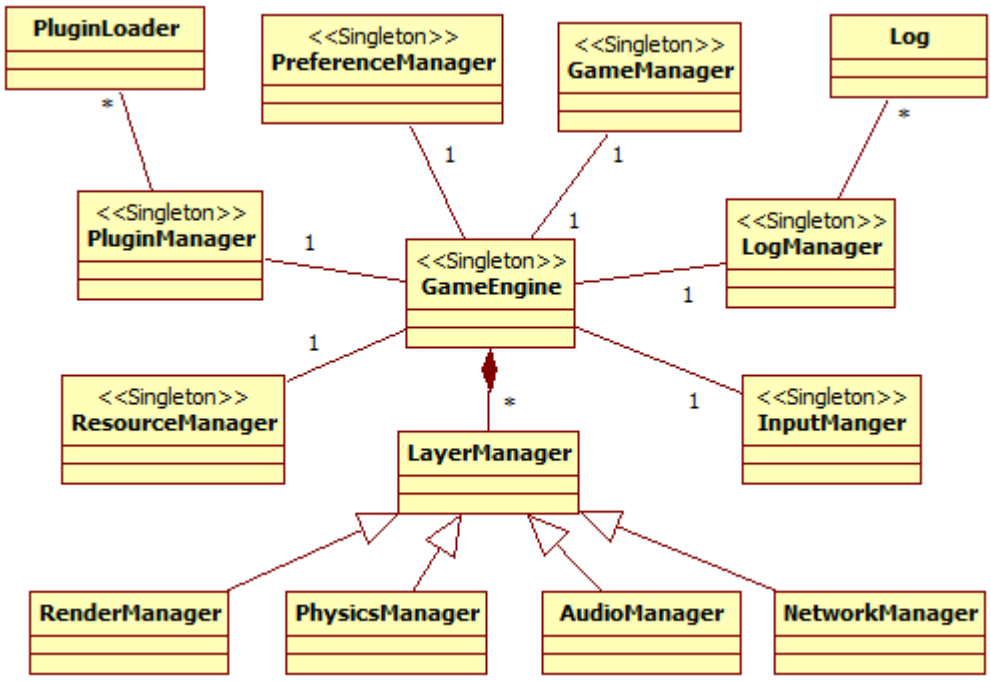


Figura 4.2 - Diagrama de clases: Diseño global del sistema

En este diseño la clase principal es "GameEngine". Ésta clase representa el motor en su totalidad y a través de ella podremos inicializar, apagar el motor y obtener acceso a las distintas partes del motor.

Gestores principales

- LogManager: El sistema de registro de errores es simple pero permite al motor registrar tanto en fichero de texto como en la consola los eventos que suceden el motor. Además permite al usuario definir sus propios registros de errores.

- PluginManager: Esta clase gestiona los cargadores de plugins que se pueden registrar utilizando la función *"bool registerPluginLoader(const String& groupName, PluginLoader* loader);"* siempre y cuando implementen la interfaz *"PluginLoader"*. Por defecto tratará de cargar los plugins que se especifiquen en el fichero *"config/plugins.cfg"* cada grupo de plugins irá precedido de una cabecera [groupname] donde groupname asociará los siguientes plugins a su cargardor.

- ResourceManager: Se trata de una clase que identifica el gestor de recursos, a través de ella podemos añadir nuevas carpetas a la lista de carpetas recursos del proyecto y cargar o descargar recursos individuales o grupos de recursos. Por defecto carga los grupos de recursos del fichero *"config/resources.cfg"*. Hablaremos del gestor de memoria más en profundidad en la sección 4.2.2.

- PreferenceManager: Es la clase encargada de cargar las preferencias del motor desde fichero. Por defecto tomará las preferencias de *"config/caelum.cfg"*.

- GameManager: Este gestor es el encargado de gestionar el sistema de juego y todo lo relativo a él. Hablaremos de él en profundidad en la sección 4.4.

- InputManager: Esta clase es la encargada de realizar la conexión con los distintos dispositivos externos de entrada. Comentaremos todo el sistema de entrada en profundidad en la sección 4.5.

- LayerManager: Clase abstracta que define el comportamiento de un gestor de capas, todos los subsistemas que el usuario decida añadir al motor por su cuenta deberán heredar de esta clase e implementar su interfaz. Hablaremos más adelante sobre qué es una capa en el motor y como se utilizan los gestores de capas en la sección 4.4.2.

Clases útiles

- ConfigFile: Ésta clase permite cargar y parsear un fichero .cfg tanto directamente (permitiendo cargar configuraciones incluso antes de inicializar el sistema de memoria) como automáticamente a través del sistema de memoria del motor.

- Singleton: Esta clase es una forma simple de aplicar a cualquier clase el patrón de diseño "singleton" tan solo heredando de ésta base, es decir, hace que su clases hijas sólo se puedan instanciar una vez. Sin embargo, y a diferencia de otras implementaciones, no supone la carga obligada de al inicio del programa si no que se pueden crear y destruir bajo demanda.

4.2.1 Arranque y parada

Función

El núcleo del sistema es el encargado de realizar la carga y descarga de todos los demás subsistemas, así como de sus plugins.

Se ha tomado especial importancia a este proceso debido a que hay tener cuidado al elegir el orden de secuencia de arranque y cierre. Es importante para el correcto funcionamiento tomar en cuenta las interdependencias entre los diferentes subsistemas.

Diseño

A continuación un diagrama de secuencia del proceso de inicialización del sistema:

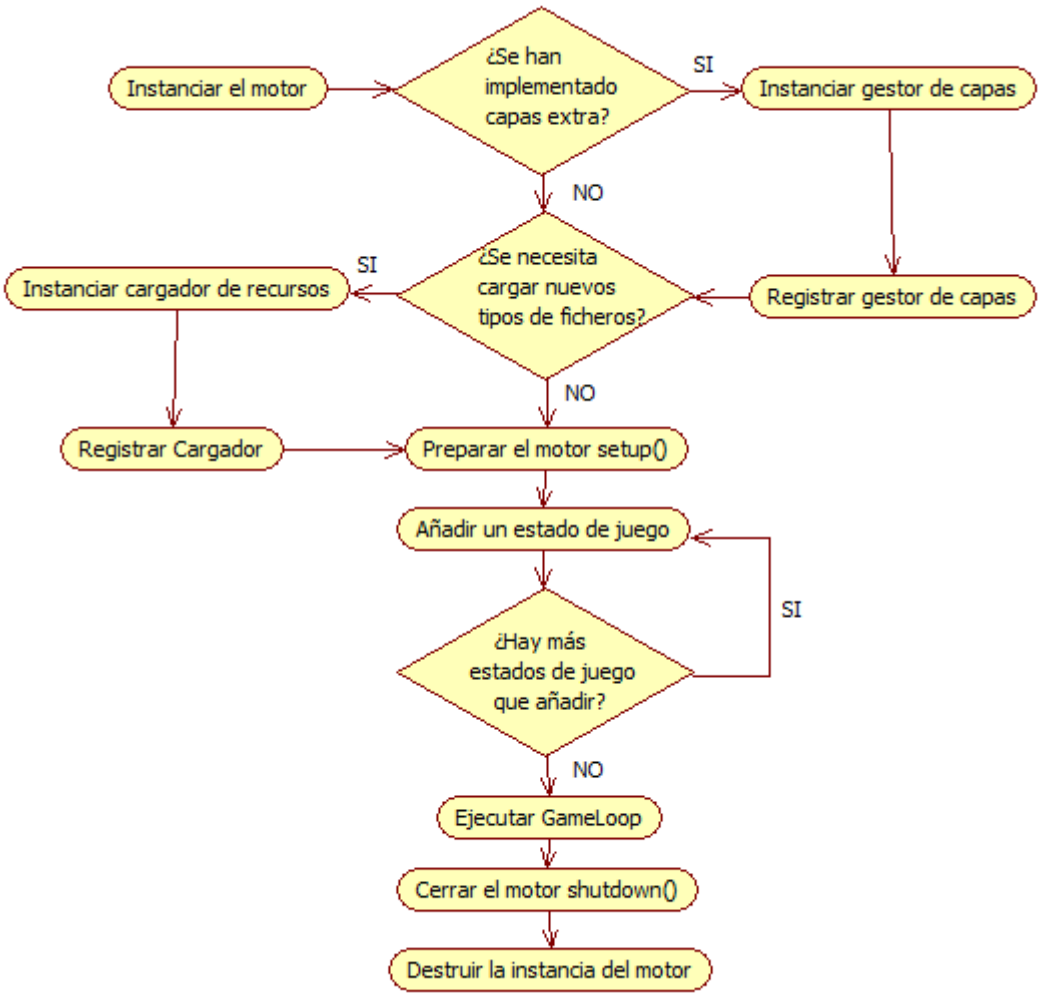


Figura 4.3 - Diagrama de actividad: Arranque y parada

La secuencia de arranque sería la siguiente:

1. Se instancia el motor y éste crea la clases más básicas

2. [Opcional] El usuario tiene la opción de registrar en el motor sus propios gestores de capas (más información en la sección 4.4.2) para el juego así como sus cargadores de recursos asociados (más información en la sección 4.2.2)

3. Se llama a la función *"setup()"* del motor. Esto que prepara el sistema de recursos y de juego utilizando los cargadores de recursos internos y los registrados por el usuario en el punto anterior para cargar una instancia vacía de los ficheros encontrados y conocidos.

4. En este punto se definen los estados de juego y se utiliza el motor.

5. Se termina el ciclo de juego y se llama a la función *"shutdown()"* que descarga todos los recursos plugins etc. Y libera la memoria asociada del motor. En este punto solo quedan disponible el sistema de registro de logs del motor.

6. Se borra la instancia del motor.

4.2.2 Sistema de memoria y recursos

Función

El sistema de memoria es la parte más importante del núcleo además del arranque y parada. El sistema propuesto presenta una abstracción de carga de recursos independiente de la plataforma en la que se ejecuta. Uno de los objetivos más importantes del proyecto es la flexibilidad, por lo tanto el motor ofrece posibilidad de añadir nuevos sub-sistemas los cuales podrían necesitar cargar sus propios ficheros y recursos. Dada esta situación parece lógico ofrecer una forma unificada para definir un nuevo tipo de recurso y su gestor de recursos asociado.

Diseño

Como podemos ver en la Figura 4.4 el gestor de recursos nos sirve para registrar el todas las localizaciones donde encontrar los ficheros que el motor pueda usar y organizarlos recursos en diferentes grupos. El gestor de recursos puede registrar diferentes cargadores de recursos *"ResourceLoader"* que serán los encargados de cargar el tipo de recursos *"Resource"* asociado.

Esta parte es completamente flexible ya que permite al usuario extender las clases abstractas *"ResourceLoader" "Resource"* y *"ResourcePtr"*, registrar su cargador personalizado y de esta forma integrar en el sistema de memoria la carga de su propio tipo de ficheros.

Este sistema es realmente potente, ya que permite al programador abstraerse completamente de los métodos de carga de ficheros dependientes del SO y a la vez seguir manteniendo un control exhaustivo del uso de memoria. Además puesto que integra la carga de recursos en el motor, será el propio sistema el encargado de la gestión de memoria asociada a éstos ficheros lo que resulta en una utilización de la memoria mucho más eficiente.

A continuación se muestra un diagrama de clases que enseña el diseño del sistema de memoria planteado:

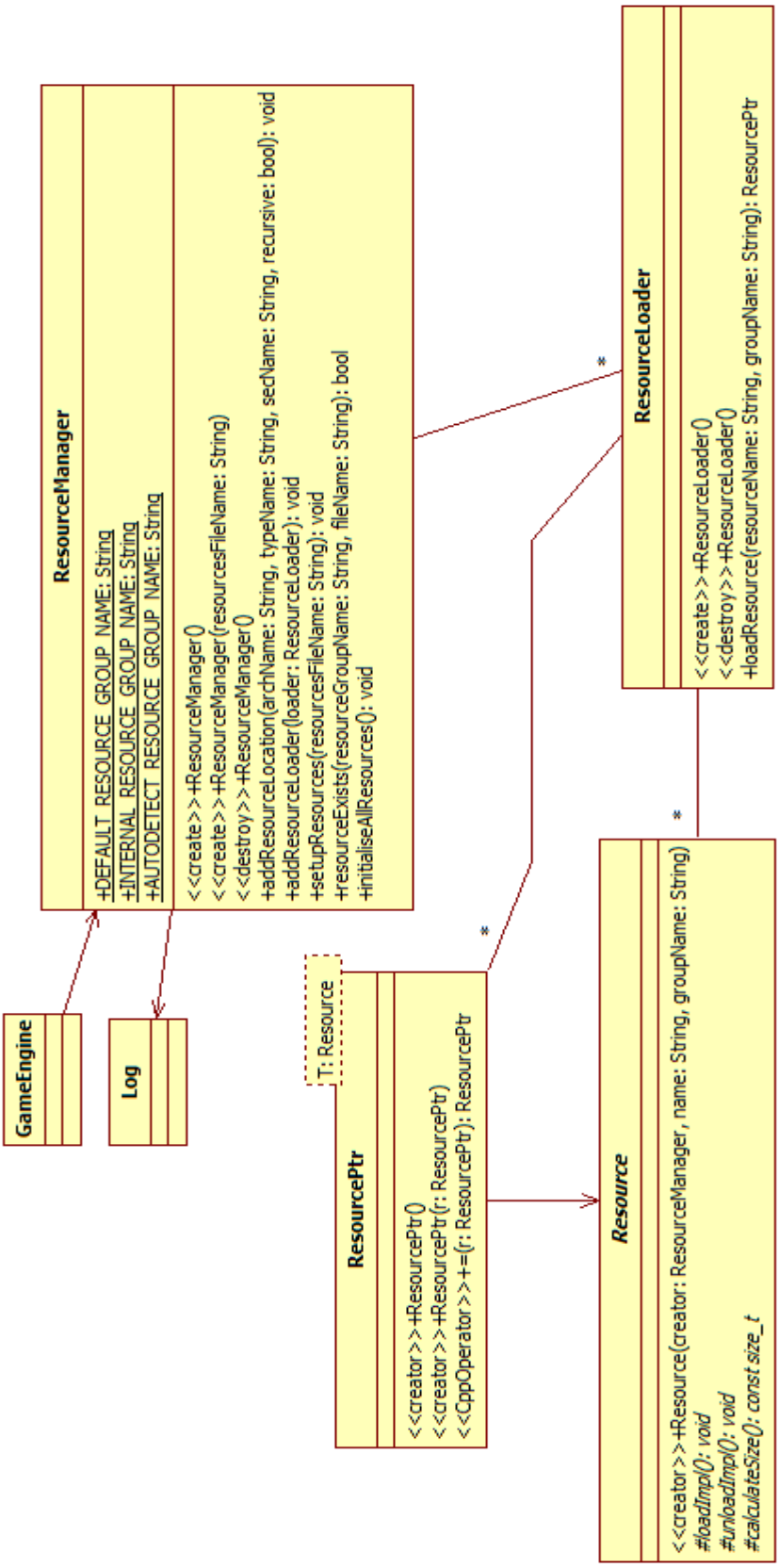


Figura 4.4 - Diagrama de clases: Gestión de memoria y recursos

4.3 DISEÑO DEL MODELO MATEMÁTICO

Función

Antes de comenzar a diseñar las escenas de juego o los objetos 3D gráficos, etc. necesitamos definir un modelo de datos matemático para representar el espacio y sus transformaciones.

Todo el diseño de objetos de juego y escenarios se apoyará en este modelo pero a la vez debe servir como utilidad de propósito general para simplificar las operaciones matemáticas del usuario.

Diseño

El diseño utilizado para la representación del modelo matemático está mayormente basado en el utilizado en la librería Ogre3D que a su vez está basado en la implementación sugerida en el libro Geometric Tools for Computer Graphics [7].

Como se puede observar en la Figura 4.5 existen las siguientes clases:

- Math: Clase de utilidad con funciones de conversión de datos y operaciones generales.

- Angle, Radian, Degree: Estas tres clases corresponden a la definición de un ángulo. Por defecto se deberá utilizar la clase “Angle” para la recepción de parámetros ya que ofrece una conversión automática desde cualquiera de las otras definiciones de ángulos.

- Matrix3: Define una matrix de 3x3 que es capaz de representar rotaciones alrededor de ejes. Implementa las opciones más comunes como la suma, resta, multiplicación, trasposición, inversión y también otras menos comunes como transformación a un vector y ángulo asociado.

- Vector3: Representa una dirección y su distancia en un espacio 3D a lo largo de los 3 ejes ortogonales. Además dependiendo de la interpretación que se le dé se puede utilizar para representar posiciones, direcciones y factores de escala. Implementa las operaciones básicas de los vectores, suma, resta, producto, producto escalar, producto cruzado, división, longitud, distancia etc…

- Quaternion: Representa una rotación u orientación en el espacio. Dispone de las operaciones básicas pero puede realizar otras más complejas a su vez como calcular la rotación óptima entre dos orientaciones.

A continuación el diagrama de clases del modelo matemático:

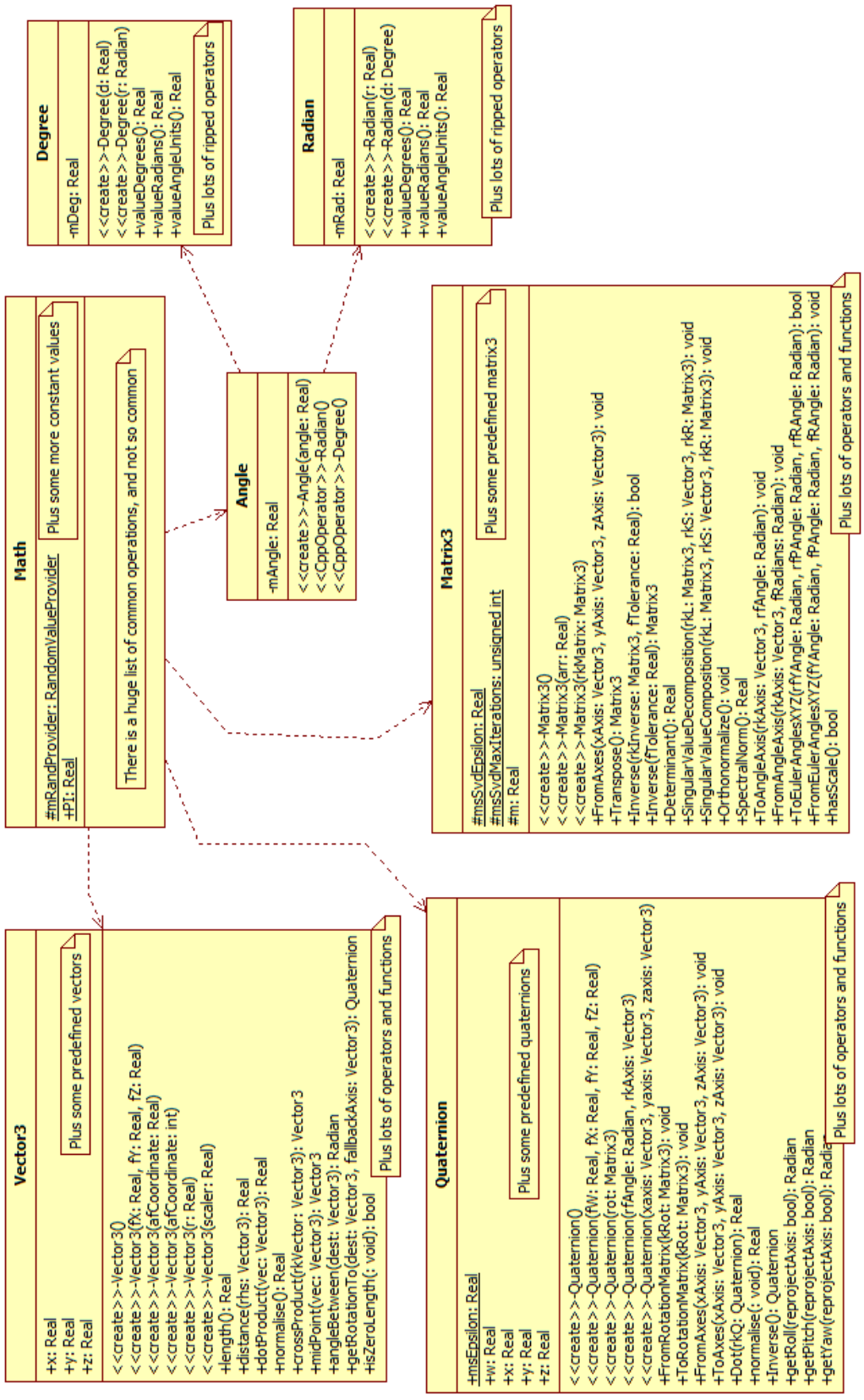


Figura 4.5 - Diagrama de clases: Modelo matemático

4.4 DISEÑO DEL SISTEMA DE JUEGO

Función

El sistema de juego es la sección encargada de proporcionar la capa del motor que utilizará el usuario para crear las escenas de juego. Esta capa supone además el punto de integración entre todos los siguientes subsistemas por lo que hay varias decisiones de diseño importantes en esta parte.

El sistema de juego involucra que el usuario pueda:

- Crear diferentes **estados de juego** que utilicen diferente lógica para gestionar de los eventos de entrada y establecer un orden lógico de sucesión de estos.

- Crear nuevas **escenas de juego** y gestionar las que ya que existen así como los objetos de juego que lo componen.

- Crear **objetos de juego** y añadir cualquier componente móvil (gráfico, físico, etc.) a éstos.

Diseño

Una vez visto el diseño general de la Figura 4.6 puede parecer un diagrama bastante amplio, sin embargo, desglosaremos el diagrama más adelante y lo explicaremos en varias partes. Se ha decidido utilizar únicamente los nombres de clase para aportar una visión global del diseño.

La mayor parte de las clases de la Figura 4.6 lo forman:

- Especificaciones de estados de juego (GameManager).
- Especificaciones de gestores de capas (LayerManager) y sus correspondientes capas (GameLayer).
- Especificaciones de los componentes móviles (MovableComponent).

Estas clases derivadas no forman en sí mismas parte del sistema de juego, si no que ayudan a extenderlo y han sido añadidas en el diagrama con el motivo de mostrar visualmente los puntos de extensión del sistema. Estos puntos son los gestores y capas de juego, así como los componentes.

El sistema de juego en sí mismo consta de 2 partes.

- La primera parte mostrará para qué sirve la gestión de estados que podemos ver en la parte izquierda de diagrama (sección 4.4.1).

- La segunda parte nos enseñará la estructura de clases concerniente a las escenas, los objetos, capas y componentes (sección 4.4.2).

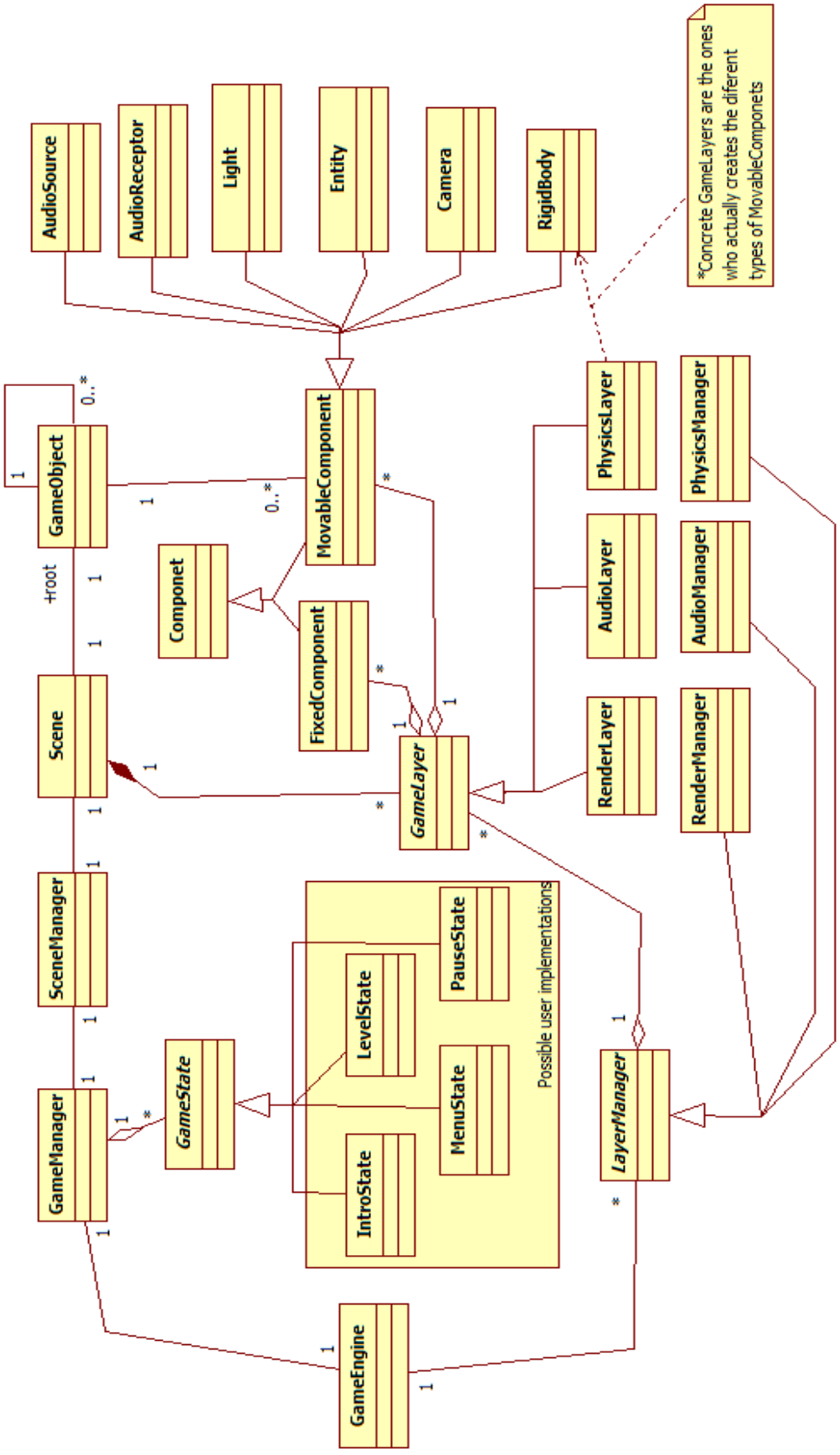


Figura 4.6 - Diagrama de clases: Diseño global del sistema de juego

4.4.1 Gestión de estados de juego

Función

Una de las partes claves del diseño anterior es el sistema de gestión de estados.

Los estados de juego son las diferentes fases de un juego donde la lógica de éste varía, es decir, donde la gestión de los eventos se realiza de diferente forma. Un ejemplo práctico de esto es la diferente de control entre utilizar un menú o controlar un personaje en una escena de juego real.

Un cambio de estado de juego no implica necesariamente cambiar la escena de juego. Supongamos que en nuestro juego controlamos a un personaje (estadoA), pero en un momento dado se necesita acceder a un menú de pausa (estadoB). Podemos utilizar la gestión de estados para separar la lógica que controla el juego, de la lógica de selección del menú. Aun así ambos estados pueden estar utilizando la misma escena ya que tras el menú de pausa se puede seguir viendo el juego en funcionamiento.

Diseño

El sistema de gestión de estados es simple pero muy útil, está basado en el diseño de varios framework casos prácticos [6]

Se trata de una implementar una cola de estados que se suceden pudiendo en cualquier caso avanzar a un nuevo estado o recuperar el anterior.

1. La primera parte del proceso de gestión de estados de juego implica que el usuario cree sus propios estados de juego derivados de “GameState” y los registre mediante la función:
```
void GameManager::addState(const String& name, GameState * state,
bool initialState = false);
```
Los diferentes estados implementan tanto las funciones básicas de estados (enter, exit, pause, resume) como la lógica de control de eventos.

2. El siguiente paso consiste en arrancar el loop del sistema de juego mediante:
```
void GameManager::start();
```

3. En algún punto los estados de juego llamarán a:
```
void GameManager::transitionTo(const String& stateName);
void GameManager::transitionBack();
```
provocando una transición hacia un nuevo estado o recuperando uno antiguo.

4.4.2 Sistema de escena y objetos de juego

Ésta parte es sin duda alguna la parte del diseño más importante de todo el motor, y la que delimitará la mayor parte de interacción con el usuario. Por lo tanto trataremos de analizar esta sección en profundidad.

Función

La función principal del sistema de escenas es proporcionar un espacio donde añadir contenido, es decir proporcionar una zona donde añadir objetos, personajes, sonidos y todos los recursos y contenidos que un juego pueda necesitar. A su vez el diseño de las escenas debe ser suficientemente flexible como para permitir al usuario añadir sus propios tipos de contenido (no solo de los tipos ya definidos como objetos visibles, simulaciones físicas, sonidos etc.).

Diseño

Como ya hemos comentado antes en ciertos puntos del diseño se ha fijado el punto de mira en los diseños ya existentes de otros motores de juego. Éste es uno de esos puntos.

Las claves del diseño de la escena son dos. Por un lado la escena contiene una jerarquía de objetos la cual explicaremos más en profundidad más adelante.

Al tratar acerca del diseño de los objetos de juego se han identificado 2 diseños predominantes en la industria:

1. Los objetos de juego son una jerarquía de clases donde poder especializar nuestros propios objetos de juego.

    - Pros: Al especificar la jerarquía es posible añadir funciones apropiadas a cada clase que hacen más cómodo el uso de las diferentes tipos de entidades.

    - Contras: No es un enfoque demasiado flexible. Además los objetos de juego serán dependiente de los tipos de datos internos del motor.

2. Los objetos de juego son un punto de referencia en el espacio y agregan componentes móviles con funcionalidades concretas.

    - Pros: Es una solución realmente flexible, permite añadir a un objeto de juego componentes con propiedades predefinidas por el motor o incluso permite al usuario del motor crear sus propios tipos de componentes que se integran muy fácilmente.

    - Contras: Es más complicado de implementar en el motor y se requieren más líneas de código desde el punto de vista del usuario para preparar cada objeto de juego.

Para Caelum-Engine se ha elegido optar por utilizar un diseño mixto pero basado ampliamente en la segunda opción (la cual utiliza Unity3D [3]) debido a la enorme flexibilidad que ofrece, pero que también permite la definición de herencias de objetos.

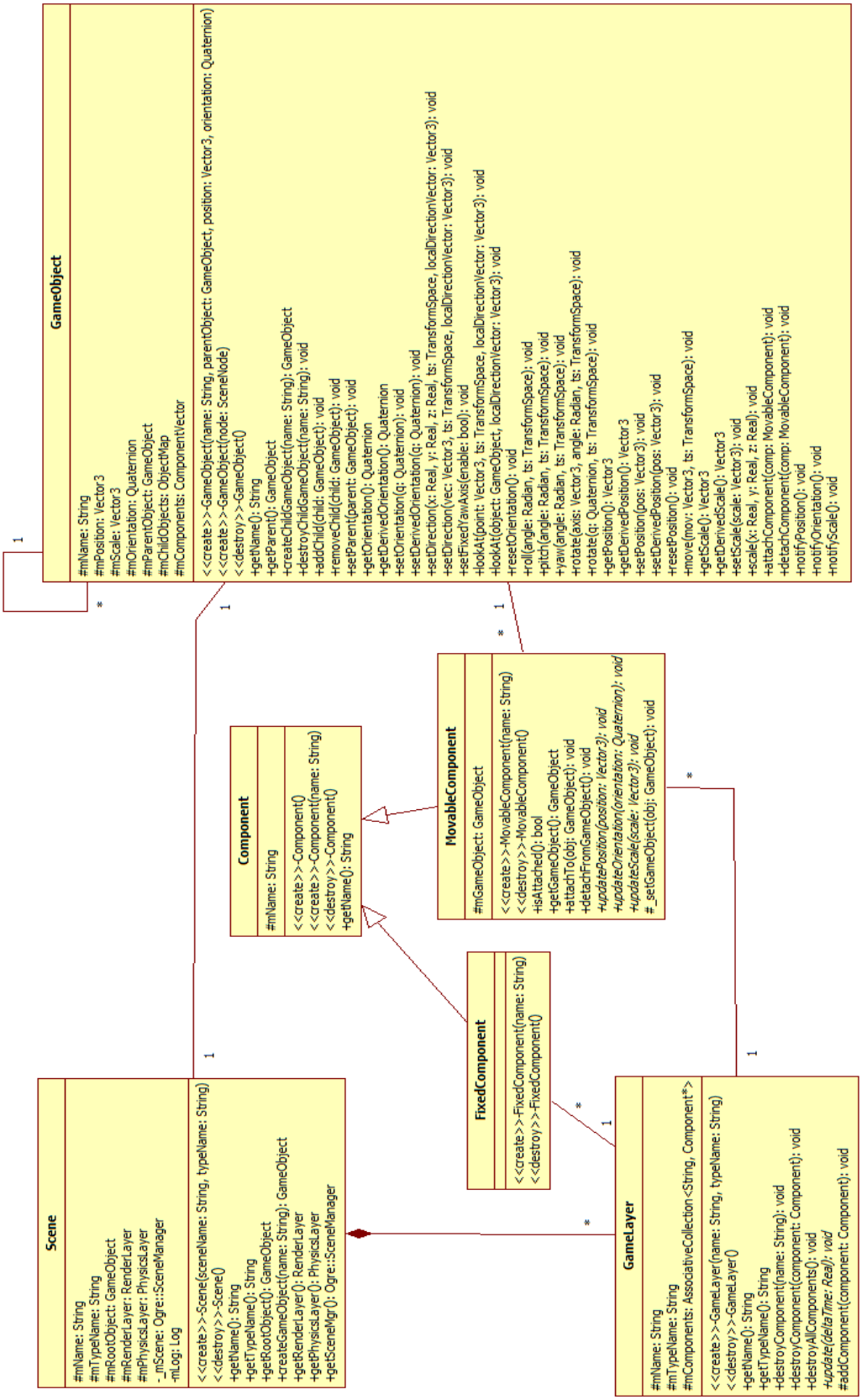


Figura 4.7 - Diagrama de clases: Diseño de la escena, capas, objetos y componentes

Escenas y objetos de juego

El objetivo de las **escenas** de juego o "Scene" es proporcionar el espacio para colocar todos los contenidos del motor y situar en él los objetos de juego, definiendo un espacio tridimensional donde el usuario puede interactuar. La escena posee a su vez el objeto de juego raíz.

Los **objetos de juego** o "GameObject" son los elementos más básicos de una escena y sirven para definir un punto y su transformación asociada en el espacio ofreciendo una posición, orientación y escala. Este diseño ha sido inspirado por los árboles de grafo de escena utilizados en los motores de renderizado [8].

Los objetos de juego representan nodos en un grafo articulado como se puede ver en la parte derecha de la Figura 4.8, es decir, un nodo contiene información sobre la transformación que se aplicará a él y a todos sus hijos. Los objetos hijos pueden tener transformaciones propias las cuales serán combinadas con las de sus padres. El grafo de escena a su vez sirve para indicar la organización de los objetos en la escena. En el ejemplo de la Figura 4.8 la rotación tierra implica mover toda la excavadora y la rotación de la base implica mover la base, codo y pala de la excavadora pero no así la tierra.

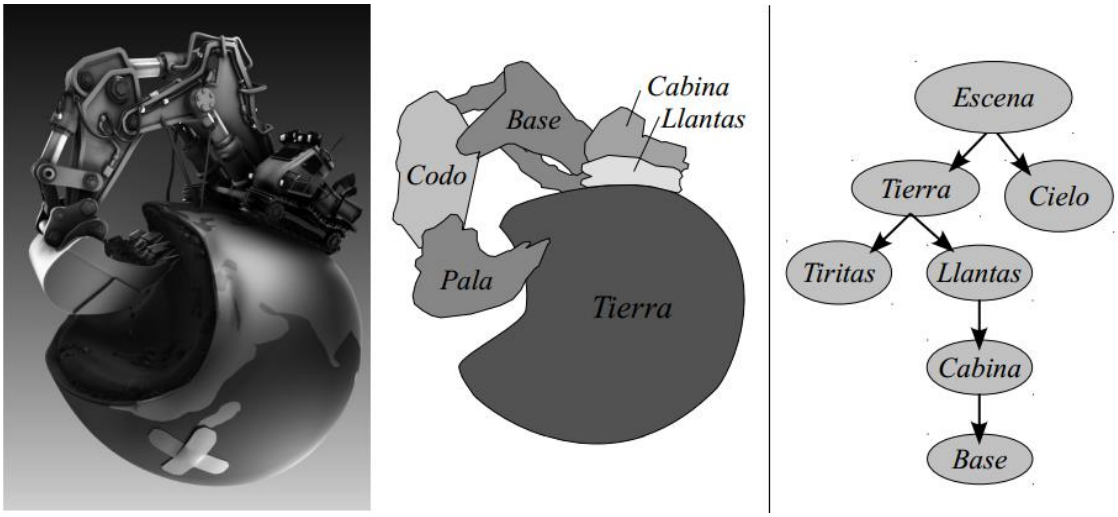


Figura 4.8 - Jerarquía visual de objetos de juego

Los objetos de escena también poseen la habilidad de adjuntar componentes móviles permitiendo la sincronización entre todos los componentes pertenecientes al mismo objeto.

Resumiendo, hasta este punto sabemos que la escena nos debe proporcionar un espacio, y los objetos de juego definen puntos para colocar el contenido que queramos. Ahora nos queda definir cómo debe funcionar el proceso de crear esos contenidos en el motor para adjuntar a los objetos de juego.

Capas y componentes

Para definir un sistema de escenas extensible se ha optado por realizar una división entre los ya conocidos escena y objetos de juego y las diferentes capas de contenido que se pueden aplicar.

Se denomina una **capa de juego** o *"GameLayer"* a la estructura capaz de crear y gestionar componentes que ofrecerán contenido relativo a esa capa. Una escena puede contener múltiples capas de juego, pero sólo una de cada tipo.

Se podría decir que las capas de contenido son "sub-escenas de componentes" relativas únicamente a uno o varios tipos de recursos relacionados. Por ejemplo en una escena por defecto de Caelum-Engine podemos encontrarnos las siguientes capas: "RenderLayer", "PhysicsLayer", "SoundLayer" y "NetLayer"; gestionando cada una de ellas el dibujado, la simulación física, el sonido, y la sincronización de red respectivamente.

Los **componentes móviles** o "MovableComponents" son tipos de recursos que ofrecen realmente contenido para añadir al sistema. Sólo se hacen realmente efectivos tras adjuntarlos a un objeto de juego. Todos los componentes móviles son susceptibles de tener una transformación en el espacio, la cual toman del objeto de juego al que van adjuntos y que ayuda a la sincronización entre todos los componentes.

También existen **componentes fijos** o "FixedComponents" que proveen una interfaz muy básica para crear componentes no asociables a un objeto del espacio 3D de la escena. Ejemplos de este tipo de componentes pueden ser por ejemplo la simulación atmosférica general, o el componente de niebla (ambos automáticamente relativos a la cámara) o la música ambiental (no asociada a un punto en el espacio).

4.5 DISEÑO DEL SISTEMA DE ENTRADA

Función

El sistema de entrada es el encargado de proporcionar la interfaz humano-máquina y recoger el "input" o entrada de los diferentes dispositivos externos.

Actualmente se necesita soporte para ratón, teclado, joysticks y gamepads genéricos.

Diseño

El diseño de esta aplicación depende del tipo interacción que queramos con el sistema, las posibilidades más frecuentes son:

1. Iterativa: Se pregunta directamente al dispositivo por su estado. No se utiliza un buffer para guardar el estado anterior de los dispositivos.

2. Basada en eventos: Se registra un "listener" y cuando se produce un cambio en el dispositivo se provoca el evento. Se utiliza un buffer que almacena el estado del dispositivo anteriormente para comprobar si se han realizado cambios.

Caelum-Engine utiliza una interacción basada en eventos, con lo cual la propuesta de diseño es la siguiente:

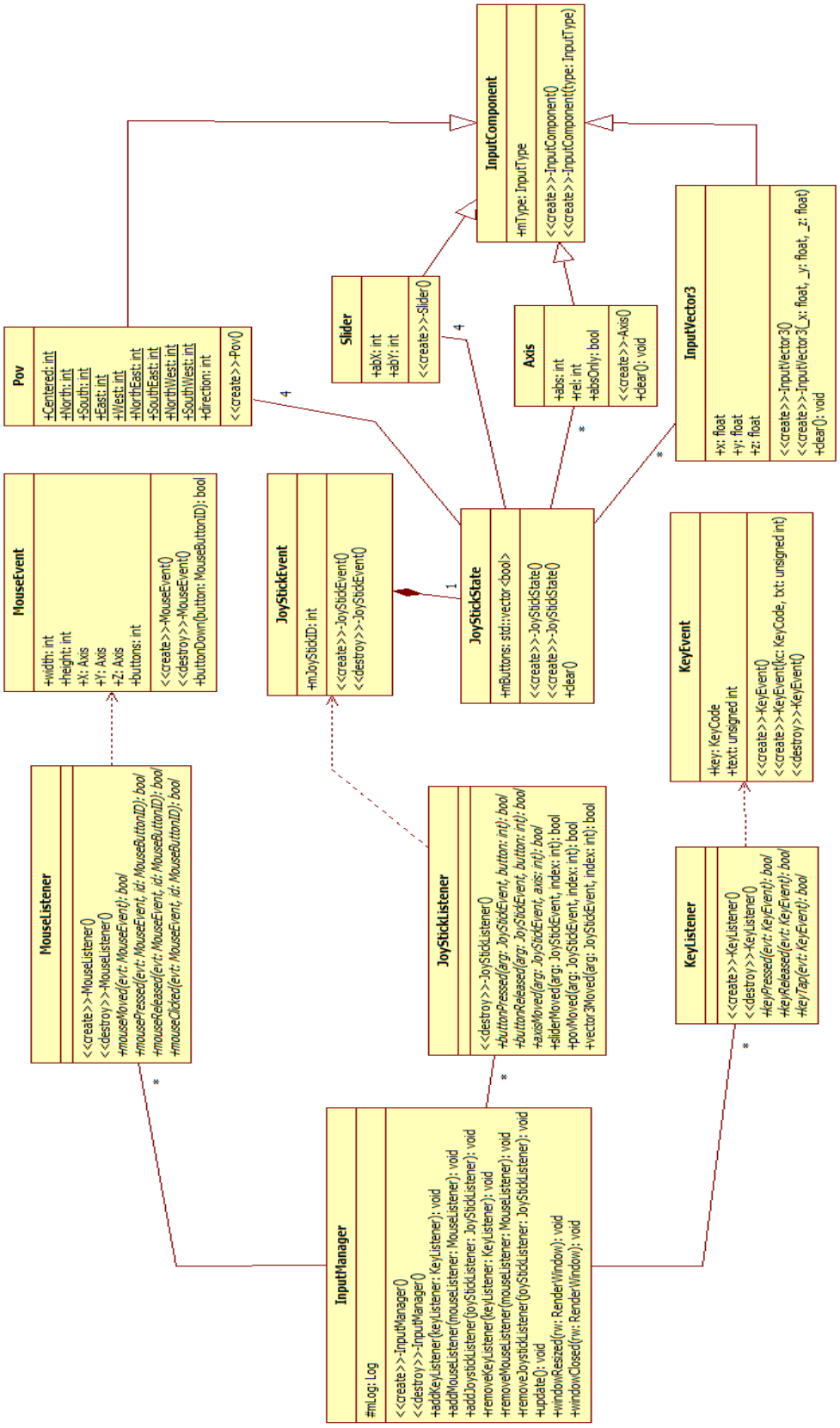


Figura 4.9 - Diagrama de clases: Sistema de input

4.6 DISEÑO DEL SISTEMA DE RENDER

Función

El sistema de renderizado se ocupa de realizar el dibujado en pantalla de los diferentes modelos 3D y efectos visuales. Las funciones principales del sistema son:

- Gestionar las ventanas del sistema: El sistema puede crear una ventana para el programa donde ejecutar el renderizado y ofrece además una forma para que el sistema pueda responder a los eventos de ésta.

- Gestión optimizada del proceso de dibujo: Dado que el proceso de dibujado involucra la utilización de la GPU y puesto que ésta es asíncrona, se debe ofrecer un modo de optimización de estos tiempos para el resto del sistema.

- Gestión de cámaras: Creación de cámaras y configuración de parámetros como FOV (field of view), el tipo de perspectiva etc.

- Gestión de modelos 3D móviles:

    - Mallas 3D: El sistema debe permitir la carga de las malas 3D de los modelos y cargar las animaciones asociadas para su posterior utilización.

    - Sistema de animación: Gestión de las transformaciones aplicadas a los huesos de un modelo 3D incluyendo posibilidad para mezclar animaciones y asignar pesos a cada una.

    - Materiales: El sistema se ocupa de la carga de los ficheros de definición de materiales y los asocia a sus correspondientes mallas.

    - Texturas: A menudo la carga de materiales implica a una o varias texturas para lograr el efecto deseado.

- Gestión de la iluminación

    - Luces: Se ofrece la creación de distintos tipos de luces, puntos, focos luces direccionales etc.

    - Iluminación y sombreado: El sombreado implica la forma en que los objetos del juego reciben la luz y cómo afecta eso a su dibujado, existen varios modelos de iluminación común disponible, pero a su vez ampliables también mediante el uso de shaders.

    - Sombras: Las creación de sombras implica la proyección de sombra que se genera después de que la luz incide sobre un objeto. Se deben proponen diferentes modelos de creación de sombras tanto estáticas como dinámicas

basadas en cálculo o en uso de texturas, para responder a diferentes necesidades.

- Gestión de geometría paginada: Se ofrecen métodos de optimización para el dibujado y la carga y descarga de geometría.
    - LOD (Level of detail): Establece dinámicamente una reducción en el número de vértices de un modelo 3D dependiendo de la distancia relativa a la cámara.
    - MipMapping: Es un efecto similar a LOD pero en lugar de reducir el número de vértices reduce la calidad de las texturas a medida que el modelo se aleja.
    - Paginación: Se trata de proporcionar la carga y descarga rápida de ciertos elementos en la escena, es muy comúnmente utilizada para determinar las distancias hasta donde deben cargarse cierto tipo de objetos.
- Gestión de efectos especiales
    - Efectos de post-procesado de imagen: Como pueden ser la profundidad de campo, bloom, SSAO etc…
    - Sistemas de partículas: Para la simulación de explosiones, fuego, humo y otros.
- Gestión de simulación de entornos abiertos: Se trata de ofrecer ciertas ayudas a la simulación de entornos abiertos con:
    - Simulación atmosférica: mediante técnicas simples y conocidas como Skyboxes Skydomes y técnicas más avanzadas de simulación de día y-noche.
    - Creación de terreno: Basados en mapas de alturas y técnicas de pintado como Splatting.
- Gestión del GUI: Un sistema específico de gestión de la interfaz de usuario que ayude a la creación de menús e información de pantalla.
- Integración con el sistema de juego: Como ya hemos comentado, el sistema de juego tiene una posibilidad de extender la funcionalidad de las escenas mediante capas, por lo tanto este sistema de renderizado debe adaptarse a éstas necesidades

Diseño

El diseño propuesto para el sistema de renderizado está altamente basado en la extensión de capas de la escena comentado en la sección 4.4.2. Además se ofrece una variada gama de gestión de eventos asociados al renderizado.

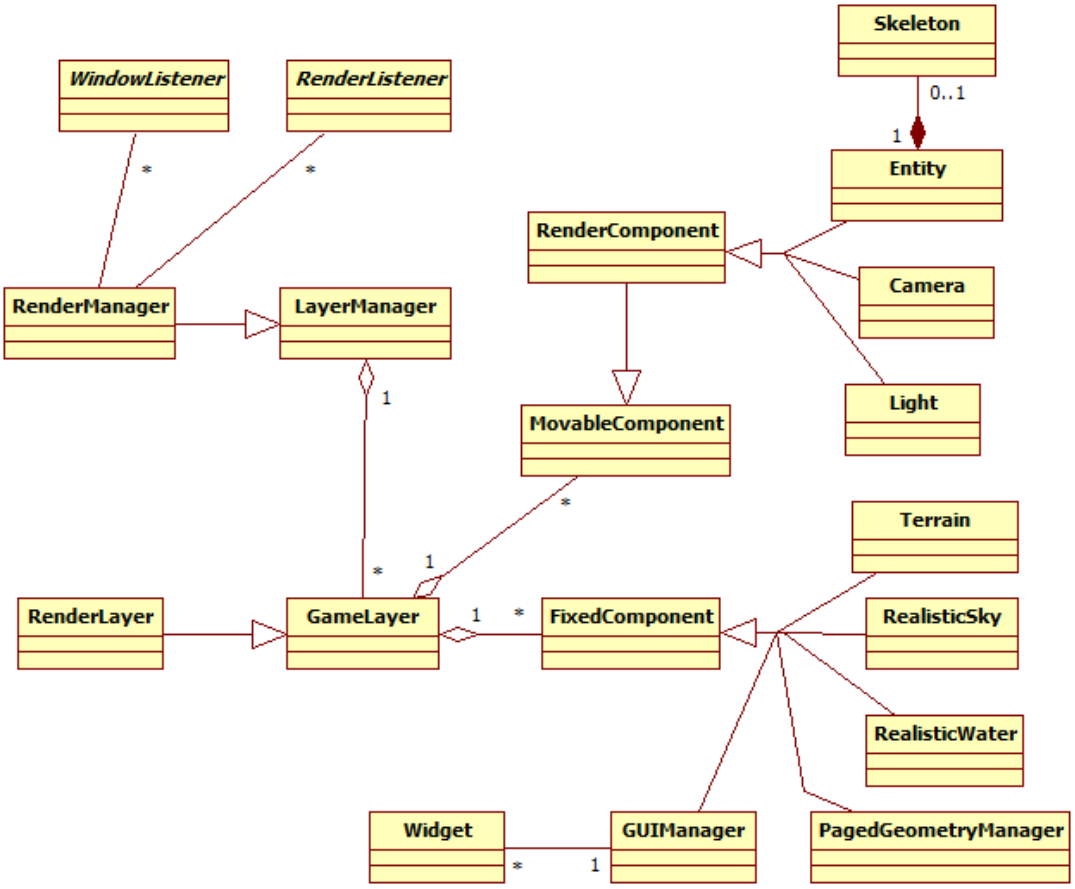


Figura 4.10 - Diagrama de clase: Diseño global del sistema de renderizado

4.7 DISEÑO DEL SISTEMA DE AUDIO

Función

El sistema de sonido es encargado de cargar los ficheros de sonido desde el sistema de ficheros, decodificarlos (en caso de y reproducirlos.

Debido a las exigencias de muchos juegos, el sonido debe notarse en 3D, de modo que se pueda distinguir cuando un sonido proviene desde tu derecha o desde detrás por ejemplo.

Diseño

El sistema propuesto en la Figura 4.11 está ampliamente basado en la extensión del sistema de capas de escena comentado en la sección 4.4.2 y muestra un ejemplo muy claro de la extensión de recursos del motor mencionada en el punto 4.2.2.

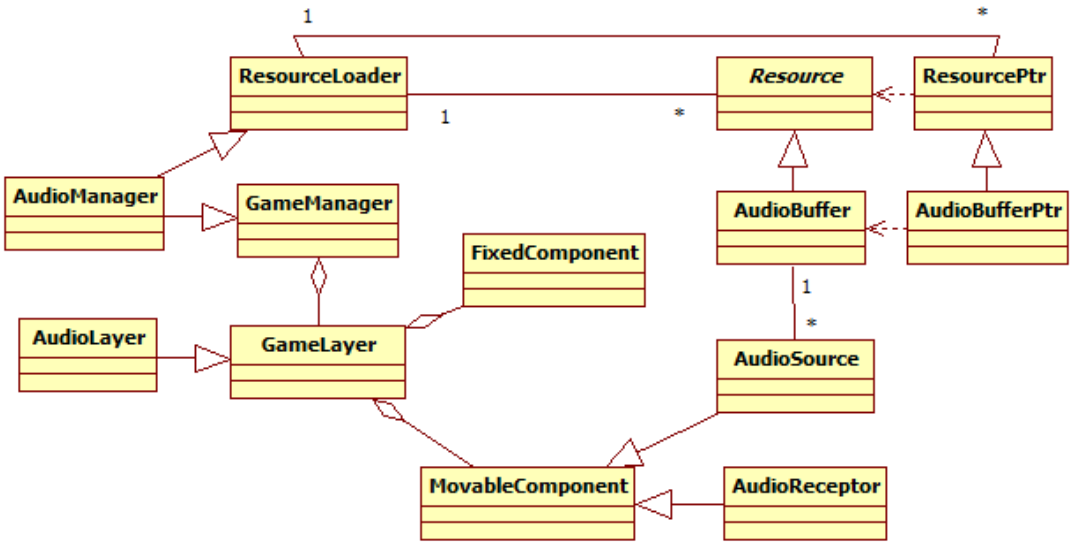


Figura 4.11 - Diagrama de clases: Diseño global del sistema de sonido

4.8 DISEÑO DEL SISTEMA DE FÍSICAS

Función

El sistema de físicas es el encargado de proporcional un entorno de simulación de dinámica y colisiones donde poder integrar con los objetos de juego para que tomen dinámicas realistas.

Nuestro sistema de físicas debe ofrecer:

- Sistema de colisiones: Sistema de detección colisiones entre las mallas de colisión creadas. Las mallas de colisión pueden formarse a partir de la malla de renderizado o tratarse de una malla primaria de colisión, como cajas, esferas, cilindros, o una combinación de éstas.

- Simulación dinámica: Simulación donde los objetos con físicas reaccionen a las fuerzas aplicadas y en caso de choque ejecuten una recuperación de la colisión reaccionando con otros objetos.

- Debugging: Se necesita a su vez un apartado que se encargue de dibujar las mallas de colisión en el sistema de renderizado, para poder comprobar fallos en la simulación y para que el desarrollador de juegos ubique realmente las mallas de colisión.

Diseño

Este diseño de la Figura 4.12 también está ampliamente centrado en la extensión del sistema de capas de escena comentado en el punto 4.4.2.

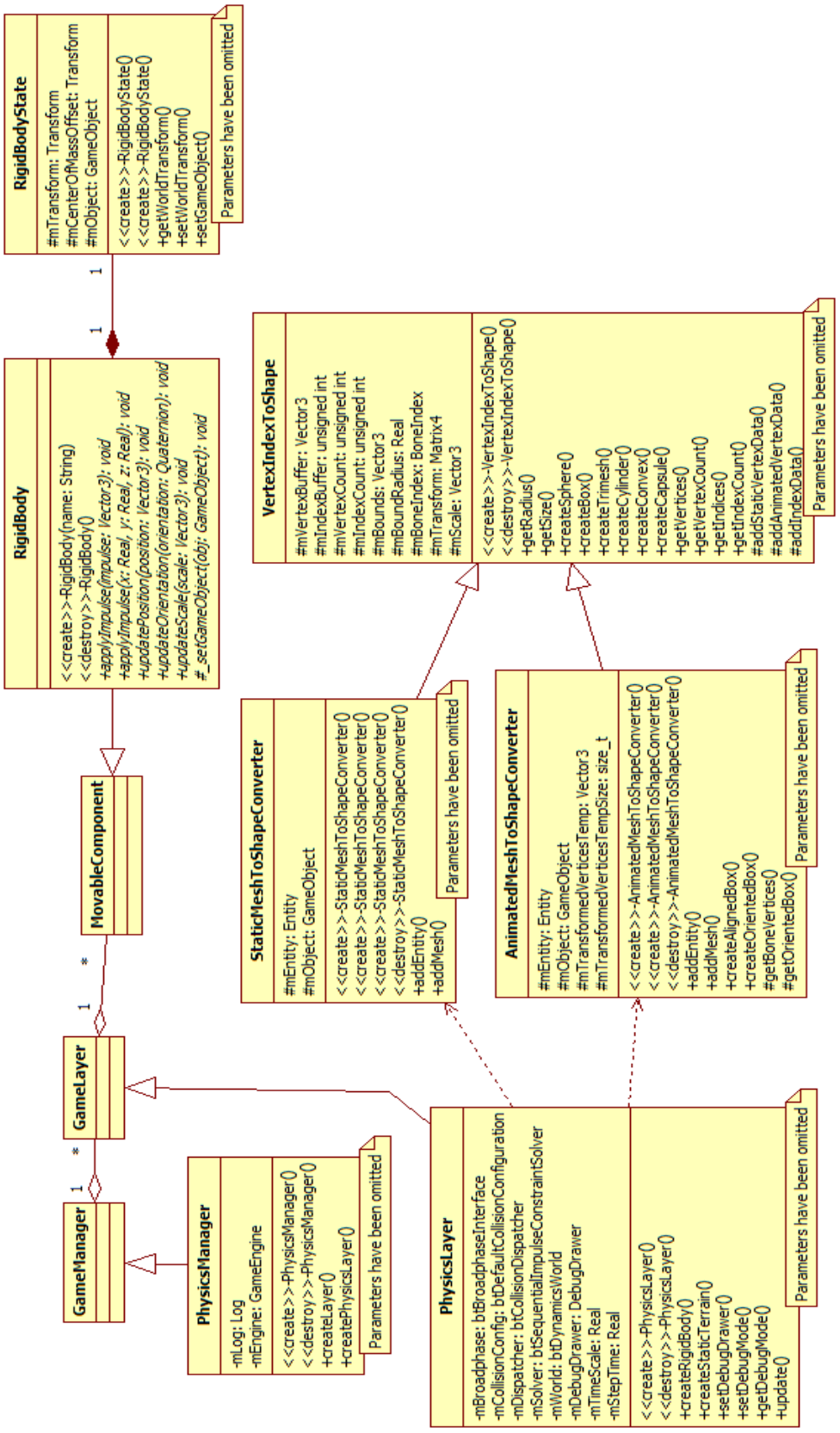


Figura 4.12 - Diagrama de clases: Sistema de físicas

4.9 DISEÑO DEL SISTEMA DE RED

Función

La función del sistema de red es crear la experiencia de un entorno compartido entre varios sistemas. Si la capacidad de transmisión de datos de la red fuese ilimitada, la sincronización de red no sería más complicada que empaquetar el estado actual de juego y enviarlo periódicamente.

Por desgracia, las capacidades de red son bastante limitadas por lo cual el objetivo principal del sistema de red es sincronizar el estado de juego sin enviarlo completamente.

Debido al diseño de objetos y componentes resulta fácil localizar los elementos a sincronizar en un objeto. Este proceso de sincronización de varios objetos de red se denomina replicación.

Las funciones específicas del sistema de red son:

- Creación de cliente-servidor: Se necesita crear un servidor que aloje una escena de juego inicial y un cliente capaz de conectar al servidor y arrancar una escena sincronizando los elementos del servidor.

- Replicación de objetos: Provee la serialización e interpretación de las variables que definen los objetos y componentes del juego.

- Optimizaciones de ancho de banda: Se ofrecen optimizaciones para la sincronización únicamente de los objetos de juego relevantes para cada cliente.

- Optimizaciones de sincronización: Derivados del retardo de comunicación en la conexión y del carácter impredecible de ciertos elementos, es necesario realizar optimizaciones para reducir el impacto del “lag” entre jugadores [9].

Diseño

El diseño propuesto para Caelum-Engine está basado en una versión simplificada del sistema de red utilizado en el motor UnrealEngine [10]. Desgraciadamente, aunque el sistema de red también está basado en la extensión de la escena, es un sistema especial, ya que necesita asentarse una capa por debajo de los demás subsistemas (véase Figura 4.1). Esto se debe a la necesidad de replicación posible de cualquier componente. Esta replicación viene dada por la serialización de objetos. La serialización de objetos la puede realizar:

1. Serialización en el sistema de red: El sistema de red elige los componentes que podrán sincronizarse y crea un método de serialización para cada uno. Esto tiene la desventaja de que los nuevos componentes no estarán incluidos en la simulación de red.

2. Serialización en los objetos y componentes: Son los propios objetos y componentes de juego quienes implementan una interfaz de conexión de red o “Serializable”. Esto

implica que la interfaz serializable en el caso de los componentes debe ser heredada desde la clase abstracta “Component”.

Se ha elegido la segunda opción para permitir la extensibilidad de red del sistema a los nuevos componentes que el usuario añada. A continuación un diagrama simplificado del diseño de red:

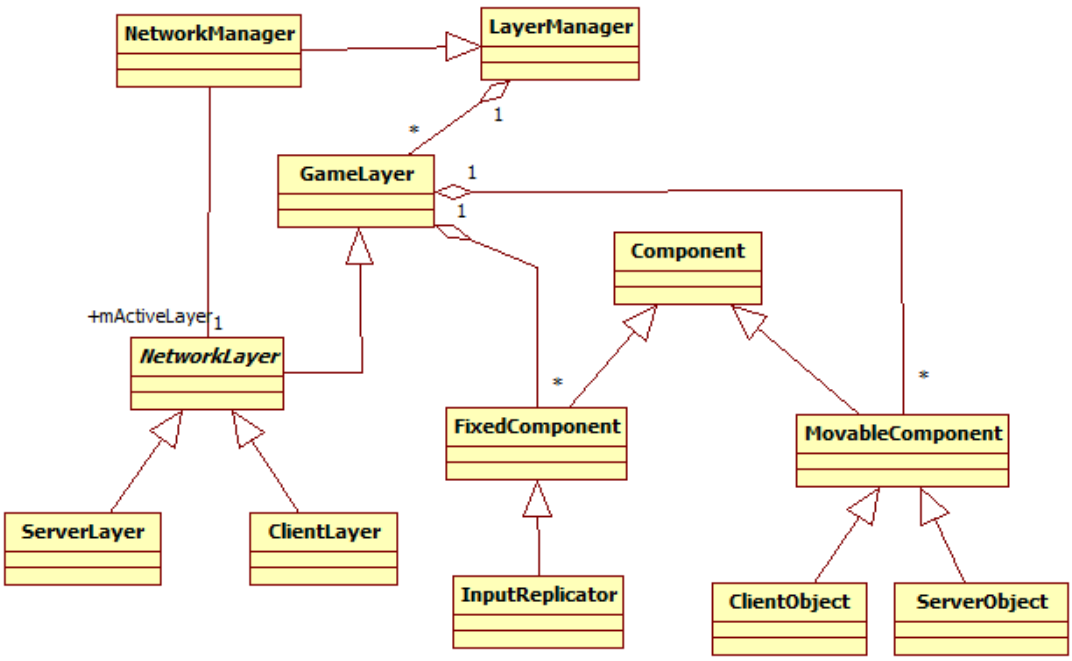


Figura 4.13 - Diagrama de clases: Diseño global del sistema de red

5. CONSIDERACIONES DE IMPLEMENTACIÓN

En el siguiente capítulo se describen las modificaciones que se han necesitado aplicar al proyecto debido a las características específicas de la implementación, como son el lenguaje, la creación específica de librerías o las tecnologías utilizadas.

El presente capítulo se divide en los siguientes apartados:

- **Estructura del proyecto:** En esta sección se comenta la división y estructuración de carpetas del proyecto para localizar apropiadamente cada uno de los recursos del motor.

- **Implementación de librerías:** En este apartado se discuten las especificaciones de creación de librerías en C++, sus problemas y peculiaridades.

- **Implementación del núcleo:** Sección en la que se comenta la implementación de gestión memoria y otros aspectos delicados del núcleo.

- **Implementación de subsistemas:** Apartado para comentar la implementación de subsistemas de juego extensibles.

- **Normas de estilo:** Breve repaso a las normas de estilo utilizadas en el proyecto, tanto para la codificación como para la organización de los ficheros de código.

- **Entorno de desarrollo:** Descripción de los medios y herramientas utilizadas en el desarrollo del proyecto, como el entorno de programación y compilación multiplataforma.

- **Tecnologías utilizadas:** Presentación de todas las dependencias internas utilizadas en el proyecto y decisiones de la selección de librerías adecuadas.

5.1 ESTRUCTURA DEL PROYECTO

La estructura de carpetas de Caelum-Engine es la siguiente:

- Bin: En esta carpeta quedarán los programas ejecutables y librerías tras su compilación.

- Build: Se utiliza como carpeta intermedia para compilar el código fuente de la librería.

- Cmake: En esta carpeta se alojan varios módulos para ayudar al sistema de compilación a encontrar las dependencias del proyecto.

- Config: Aquí se situarán los ficheros de configuración del motor, como las opciones generales del motor, plugins y selección de recursos.

- External: Esta carpeta sirve para almacenar las dependencias del proyecto y posteriormente cargarlas en su compilación.
- Include: Esta carpeta contiene todas las cabeceras necesarias para utilizar Caelum-Engine.
- Media: Carpeta por defecto donde se localizan todos los recursos 3D sonidos texturas etc. para cargar por el motor.
- Plugins: Carpeta donde se sitúan los plugins utilizados por el motor como la conexión con los sistemas de renderizado como directx y opengl, plugins de gestión de escena y la extensión para la carga de shaders CG.
- QtProject: Esta carpeta es la utilizada actualmente para guardar los ficheros de configuración de proyecto en el entorno de desarrollo.
- Src: Carpeta donde podemos encontrar el código fuente de Caelum-Engine.
- Tools: Carpeta donde se encuentran varias herramientas y utilidades para el proyecto. Estas herramientas van desde un comprobador de normas de estilo a editores compatibles con el motor como editor de GUI o el editor de mapas.

5.2 IMPLEMENTACIÓN DE LIBRERÍAS

Debido a la necesidad utilizar el motor de videojuegos como una librería necesitamos afrontar las limitaciones impuestas al crear una librería dinámica en C++.

Formas de exportar desde C++

Para crear una librería lo básico que necesitamos hacer es exponer la funcionalidad exportando las clases que deseemos. Para ello, todo lo que se necesita es utilizar el especificador "__declspec(dllexport)" antes del nombre de la clase o antes de la función exportar [11]. Se ha utilizado una macro como ésta para simplificar el trabajo.

```
// ------------- STATIC LIBRARY (COMPILE & LINK) ---------
#if defined(STATIC_LIB)
#define _ADDExport
// ------------- DYNAMIC LIBRARY (COMPILE) ---------------
#elif defined(DYNAMIC_LIB)
#define _ADDExport __declspec(dllexport)
#else
// ------------- DYNAMIC LIBRARY (LINK) ------------------
#define _ADDExport __declspec(dllimport)
#endif
```

Código 5.1 – Definición de macro general para exportar clases en C++

Con esta macro podemos utilizar la librería de múltiples formas:

- Para **compilar** y **utilizar** una **librería estática** (.lib o .a) debemos utilizar #define ESTATIC_LIB
- Para **compilar** como **librería dinámica** debemos utilizar #define DYNAMIC_LIB
- Para **utilizar** la **librería dinámica** basta con que borremos el define anterior al distribuir las cabeceras del proyecto

A la hora de querer exportar la funcionalidad de una clase solo deberemos definirla de la siguiente manera

```
class _ADDExport FooClass {
      public:  int foo(int x);
}
```

Código 5.2 – Ejemplo de utilización de macro de exportación

Limitaciones al exportar

En resumen estas son las limitaciones básicas impuestas en las clases a exportar

1. Ausencia de un estándar ABI (Application Binary Interface) en C++:

   A diferencia de los compiladores de C, en C++ no existe un estándar que defina la forma en que los binarios de ejecutables y librerías deben estructurarse. La falta del ABI provoca que cada compilador e incluso a veces cada versión del mismo compilador cree la librería de manera distinta, lo que da lugar a incompatibilidades y restringe el compilador a utilizar por el usuario.

   En el caso de las aplicaciones no suele suponer un mayor problema, sin embargo en el caso de las librerías sí. Esto implica que tanto el usuario de la librería como la propia librería dinámica deben utilizar una misma versión del compilador a fin de evitar incompatibilidades, lo que supone una restricción para el usuario. Si ambos utilizan diferentes versiones del compilador uno de ellos puede corromper el estado interno del otro produciendo probablemente un cuelgue en la aplicación.

2. Transitividad al exportar clases:

   Es decir, exportar una clase desde C++ requiere exportar todos los tipos de datos relacionados a dicha clase: todas sus clases base y todas las clases utilizadas en la definición de sus miembros etc.

   Esta característica es lógica y garantiza que todas las clases creadas en una librería puedan ser especializadas por el usuario, por lo tanto todas las clases implicadas directamente (como atributos miembro), o indirectamente (como parámetros o retorno

de funciones) deberán ser exportadas también con el objetivo de que la definición de la clase padre no quede incompleta.

A priori esta característica no parece un problema, sin embargo esto significaría que en caso de herencia o composición utilizando otras librerías dentro de Caelum-Engine éstas deberían ser también exportadas, obligándonos además a exponer distribuir las cabeceras de nuestras dependencias.

3. Manejo de excepciones:

   Debido a la forma en la que las librerías dinámicas se crean tanto el usuario como la librería deben utilizar el mismo planteamiento para el manejo y propagación de excepciones.

De estas limitaciones, es la segunda la más importante y la que nos supone un mayor problema. No solo a la hora de implementar si no, que incluso puede modificar ligeramente nuestro diseño si no lo hubiésemos tenido en cuenta hasta ahora.

Aproximaciones y soluciones

Como ya hemos comentado exportar nuestras clases no es tan sencillo ya que se deben exportar todos sus tipos de datos asociados. En el caso de utilizar tipos de datos de otras librerías, queremos abstenernos de exportarlos. Hemos detectado varios casos conflictivos:

1. Utilizar una clase externa como variable miembro:

   Éste caso es probablemente el más simple de solucionar. Para evitar exponer una clase externa que utilizamos como variable miembro podemos utilizar lo que se conoce como "puntero opaco".

   Un puntero opaco consiste en crear la variable miembro como un puntero a un tipo de datos de un tipo no especificado. En concreto en Caelum-Engine se han utilizado d-pointer (dereferenced pointer). Un ejemplo práctico de su utilización es el siguiente:

```
// Forward declaration of ogrescenenode
namespace Ogre { class SceneNode;}

namespace Caelum {
  class GameObject {
    public:
      GameObject(/*...*/);
      virtual ~GameObject();
      /*...*/
    private:
      Ogre::SceneNode *_mNode;
  };
} // namespace Caelum
```

Código 5.3 - Utilización de punteros opacos en C++

En la sección de código mostrada arriba nuestro propio objeto GameObject necesita un objeto de la librería de renderizado Ogre3D de la que hablaremos más adelante. En este caso con realizar una declaración previa sin especificar el tipo de dato de la clase externa será suficiente para poder declarar un puntero sin problemas. Posteriormente solo deberemos incluir la cabecera con la definición completa de la clase externa en nuestro fichero de implementación donde la utilizaremos.

Esta implementación supone además no incluir el archivo de la clase externa en la cabecera del nuestro, lo que nos evita tener que distribuir nuestras dependencias para utilizar el motor.

2. Herencia de una clase externa:

Éste caso resulta un poco más problemático y su solución modifica ligeramente el diseño de una clase que necesite especializar de otra externa [12].

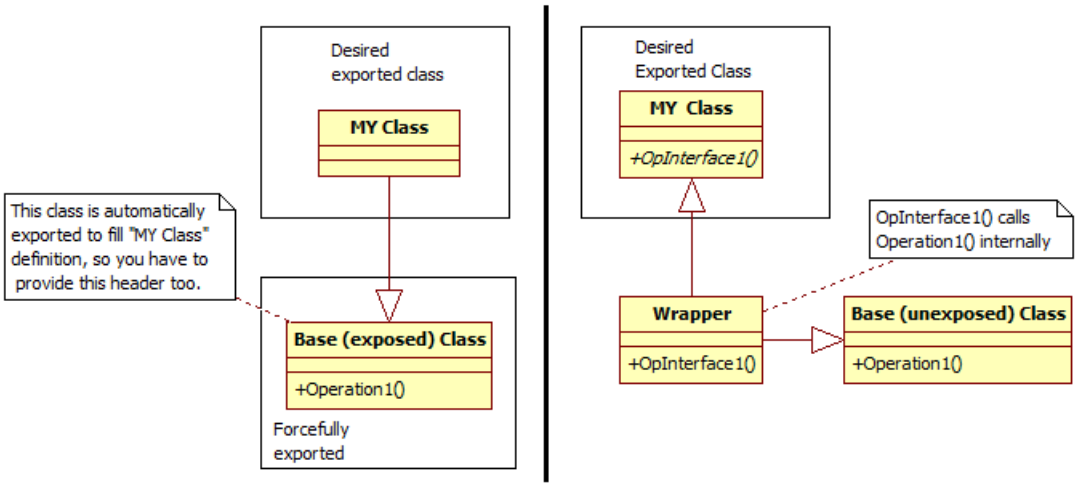


Figura 5.1 - Refinamiento de la herencia (para evitar exponer la clase Base/Padre)

Como se puede ver en la Figura 5.1 la solución más clásica consiste en declarar las funciones de nuestra clase “My Class” como abstractas. Posteriormente crear una clase “Wrapper” que hereda tanto de “My Class” como de la clase “Base” y realizar un mapeo de las funciones necesarias. De esta manera nuestra clase “Base” que pertenece a otra librería por ejemplo no necesita ser expuesta.

Con esta solución la clase expuesta solo conoce la funcionalidad del programa y no la implementación de ésta. La única limitación pendiente en esta solución es que las clases abstractas expuestas no podrán ser instanciadas directamente por el usuario si no que se deberá solicitar sus implementaciones a la librería a través del patrón factory o similar.

5.3 NORMAS DE ESTILO

La utilización de normas de estilo no es estrictamente necesaria para que un programa funcione, sin embargo, en la práctica resulta algo casi esencial para facilitar la rápida comprensión y manipulación del código fuente.

No existe un único estilo de formato para el código pero sí existen ciertas normas mayormente aceptadas por la comunidad de desarrolladores. En Caelum-Engine se utilizan como base las normas de estilo propuestas por google[1].

Algunas de las normas más importantes utilizadas son las siguientes:

1. Para la nomenclatura de variables se utiliza el formato *Camel Case*.

2. Para las extensiones de ficheros se utilizará el estándar de Visual C++ usando ".h" para las cabeceras y ".cpp" para la implementación.

3. Los nombres de ficheros utilizarán el nombre de la clase que implementan sin espacios y completamente en minúsculas.

4. Para mayor difusión del código se codificará completamente en inglés, incluidos los comentarios.

5. Las funciones de uso interno del motor o cuya utilización necesite de requisitos previos irán precedidas del carácter especial "_".

6. Los ficheros estarán codificados en UTF8, ya que al contrario de ISO-8859-1 funciona de igual manera en los editores de texto de Windows y Linux.

5.4 ENTORNO DE DESARROLLO

Para realizar el proyecto Caelum-Engine y especialmente debido a la naturaleza multiplataforma de éste, la preparación de un entorno de desarrollo adecuado y único para todas las plataformas es esencial.

Para poder realizar el proceso de programación, compilación y prueba del motor se han utilizado herramientas especializadas en cada uno de estos procesos en lugar de un entorno de desarrollo único.

5.4.1 Sistema de control de versiones

Como método de control de la evolución del proyecto se utiliza un sistema de control de versiones. En el desarrollo de Caelum-Engine se ha utilizado Git.

[1] http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

Git es un sistema de control de versiones distribuido pensado principalmente para ser eficiente y confiable. Se ha elegido Git como sistema de control de versiones sobre otras alternativas como Subversion, debido a que Git se ha convertido últimamente en un estándar de facto por su rapidez y comodidad y ha sido adaptado ya por grandes proyectos como el desarrollo del núcleo de Linux.

Otra de las razones por las que se ha utilizado Git es por su carácter descentralizado, que al contrario que la mayoría de alternativas permite crear "commits" sin necesidad de conexión.

A pesar de ser un servicio distribuido, lo cual implica que cada repositorio contiene el historial y proyecto completo, se ha utilizado otro repositorio remoto privado en BitBucket. Este repositorio extra aporta una copia de seguridad en caso de pérdida de datos.

5.4.2 IDE

El IDE (Integrated Development Environment) supone el entorno de programación y desarrollo de un proyecto software. Por norma general los IDE se componen de varias herramientas como el editor de textos y cadena de compilación integrada, sin embargo, para el desarrollo de Caelum-Engine solo se ha tomado en cuenta en los IDE la edición de código fuente y ayudas de programación como el autocompletado de código plantillas etc.

Figura 5.2 - Logo de QtCreator

Se ha optado por elegir QtCreator como IDE por múltiples razones:

- Es multiplataforma y los archivos de configuración son intercambiables entre sistemas operativos.
- Es un IDE que permite integrar sistemas de compilación alternativos al suyo propio.
- Tiene un buen soporte de herramientas de ayuda para programación en C++.
- Es un IDE rápido y eficaz.
- El equipo de trabajo conoce el entorno y lo ha utilizado previamente.
- Es un producto gratuito y de software-libre[2]

[2] http://qt.gitorious.org/qt-creator

5.4.3 Compilación multiplataforma

Cuando se desarrolla un juego o motor de videojuegos utilizando un lenguaje compilado, implica que se necesita realizar la compilación de este en cada plataforma objetivo independientemente.

La mayor problemática en esta situación está en que cada sistema operativo es diferente y las llamadas al sistema de la librería serán diferentes para cada plataforma. Esto no resulta una tarea sencilla, existen varios compiladores para cada plataforma pero con el objetivo de unificar el modo de compilación en las diferentes plataformas la opción más sensata es encontrar un sistema de compilación multiplataforma.

La compilación multiplataforma es posible utilizando los compiladores de GNU (gcc y g++ en Linux y MinGW en Windows) y la herramienta Make para establecer las reglas de compilación al compilador.

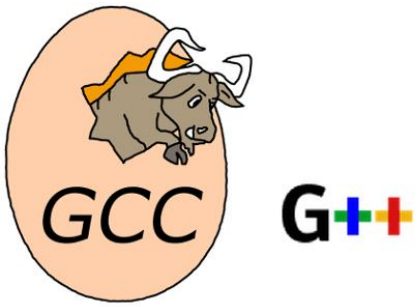


Figura 5.3 - Logo de los compiladores GCC y G++

Aun así, a pesar de las facilidades de estas herramientas, la realización de ficheros de configuración Makefile, hay ciertos aspectos de la configuración dependientes del sistema operativo. Para subsanar estas carencias se ha utilizado CMake, una herramienta multiplataforma de generación y automatización de scripts de compilación.

Figura 5.4 - Logo de CMake

5.5 TECNOLOGÍAS UTILIZADAS

A la hora de elegir este proyecto, se ha tomado una decisión de diseño estratégica muy importante con respecto a la elección de las librerías a utilizar.

Caelum-Engine se concibió en un inicio como una manera de comprender el funcionamiento interno de un motor de videojuegos, pero a su vez realizar un proyecto práctico y útil. Debido a

la enorme envergadura que supone este tipo de proyecto, era obvio que se necesitaba partir de una base afianzada.

La elección de optar por utilizar software libre, vino de una decisión estratégica que ayudará a mantener el proyecto vivo durante mucho tiempo:

- **Mejora continua**: La cualidad más importante de los proyectos elegidos es la mejora continua. El motor de renderizado, el de físicas etc. se encuentran en un proceso de mejora continua hacia las nuevas generaciones de gráficos y simulación, lo cual resultaría muy difícil de realizar con los recursos disponibles para este proyecto.

- **Comunidad**: La mayoría de los proyectos de software libre incorporados en Caelum-Engine cuentan con una gran comunidad detrás, que ayuda a resolver dudas y están abiertos a proposiciones y cambios.

- **Código abierto**: Ante cualquier necesidad de cambio las librerías utilizadas son de código abierto y permiten la modificación del código fuente para adaptarse a as necesidades propias.

5.5.1 Renderizado

La parte más importante de un motor de videojuegos, es sin duda su capacidad para representación de gráficos en pantalla. Para ello y debido a los motivos presentados en el punto 5.5 que han llevado a Caelum-Engine a basarse en proyectos de software libre, se han analizado y contrastado diferentes propuestas para motores de renderizado.

Ogre3D
Licencia: MIT.

Ogre3D (Object-oriented Graphics Rendering Engine) es un motor de renderizado 3D escrito en C++ y lanzado bajo licencia MIT y que supone una capa de abstracción sobre OpenGL y DirectX para utilizar gráficos acelerados por hardware. Este proyecto fue creado por Steve Streeting en el año 2000, actualmente ya retirado del proyecto, pero que sin embargo continúa su desarrollo con un equipo de 9 desarrolladores.

Figura 5.5 - Logotipo de Ogre3D

Finalmente el motor de renderizado elegido ha sido Ogre3D por los siguientes motivos:

- Motor de renderizado completo y bien establecido.
- Soporta un amplio rango de hardware.
- Ofrece soporte para OpenGL y GLSL y permite utilizar el motor de igual manera en diferentes plataformas.
- Tiene una amplia comunidad detrás donde resolver dudas ante cualquier problema.
- Contiene gran cantidad de documentación y ejemplos prácticos.
- Existe un gran número de plugins disponibles que amplían su funcionalidad.
- Ofrece compatibilidad con un gran número de suites de diseño 3D como 3DMax, Blender, Maya, etc.

Como ya hemos comentado la comunidad de Ogre3D ha desarrollado varios plugins que añaden funcionalidad al motor y que Caelum-Engine ha querido aprovechar.

MyGUI

Licencia: MIT.

MyGUI no es exactamente un plugin, pero se trata de una librería que utiliza el motor para crear y manejar interfaces de usuario interactivas. Se ha elegido MyGUI debido a que dispone de un editor para crear las interfaces fácilmente integrable. Además posee localización de idioma y dispone de soporte para creación de temas visuales.

Figura 5.6 - Imagen demo de MyGUI

SkyX

Licencia: LGPL.

El propio motor de renderizado de Ogre ofrece varias opciones comunes para la simulación de cielo como son skyboxes o skydomes, pero ambas técnicas ofrecen una simulación estática y poco elaborada.

Figura 5.7 - Imagen demostración de SkyX

Ante esta problemática surgió el plugin SkyX que realiza una simulación dinámica implementando incluso la variación de ciclos de día y noche. Además de la simulación de día y noche, SkyX ofrece un sistema de renderizado de nubes volumétricas para complementar el sistema.

Hydrax

Licencia: LGPL.

Hydrax es un plugin del mismo creador de SkyX, Xavier Verguín Gonzalez, y provee una librería fácil de usar para renderizar simulación de mares bastante realista.

Figura 5.8 - Imagen demo de Hydrax

Es altamente configurable y permite utilizar multitud de técnicas diferentes en la simulación. Viene acompañado de un editor para modificar y probar sus valores de configuración en tiempo real.

PagedGeometry
Licencia: zlib/libpng

Este plugin es uno de los más importantes en cuanto a optimizaciones de rendimiento. El plugin de geometría paginada ofrece una gestión automática de ciertos elementos estáticos y repetitivos en la escena como por ejemplo árboles rocas hierba etc. Este plugin es comúnmente utilizado para representar la vegetación de una escena.

Figura 5.9 - Imagen demo de PagedGeometry

Los beneficios de utilizar este plugin son múltiples:

- Gestión de la distancia de renderizado de cada elemento. Esto permite por ejemplo dejar de renderizar elementos que se encuentran a mucha distancia y no son apreciables, ahorrando mucho tiempo de GPU.

- Gestión automática de las transiciones de LOD (Nivel de detalle de las mallas) y MipMaps (Nivel de detalle de texturas) dependiendo de la distancia de renderizado.

- Gestión automática de transición entre mallas estáticas y planos impostores en largas distancias. A grandes distancias donde el detalle de las mallas 3D no es apreciable, se intercambian las mallas por planos.

- Disposición de elementos por densidad. Mediante un mapa de densidad donde negro indica la ausencia de elementos y blanco su máxima densidad, se puede distribuir cierta cantidad de elementos estáticos por cada zona en una escena.

5.5.2 Entrada

Otro de los puntos esenciales de un motor de videojuegos es la gestión de eventos y el procesamiento de las señales de entrada.

OIS
Licencia: zlib/libpng.

La librería OIS (Object-Oriented Input System) se utiliza para la recepción multiplataforma de los eventos de entrada, tanto de teclado y ratón como cualquier tipo de joystick. En teoría esta librería dispone de soporte para el mando WiiMote, al cual se trata como un joystick más. Sin embargo, debido a que no se dispone de este dispositivo no ha sido posible probarlo.

La librería tiene un diseño sencillo e intuitivo que resulta bastante práctica, pero a veces requiere de extensión para añadir otras funcionalidades necesarias. Si diseño minimalista solo permite registrar un *listener* por cada dispositivo, y carece de eventos muy conocidos como "click" (presionar y soltar un botón en un periodo de tiempo breve) o "doble click".

5.5.3 Simulación física

BulletPhysics
Licencia: zlib.

Es una librería de simulación de físicas para videojuegos muy famosa y utilizada comercialmente incluso por grandes estudios en juegos para consolas de última generación y PC. Para Caelum-Engine nos hemos decantado por BulletPhysics debido a que ofrece los mejores resultados de rendimiento en tiempo real con respecto a otras librerías de software libre como ODE.

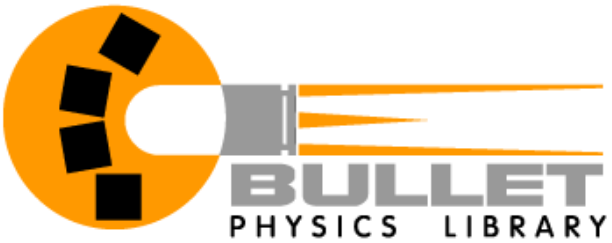


Figura 5.10 - Logo de Bullet Physics

Es un motor de físicas bien establecido en el mercado y que aporta unos rendimientos muy buenos, sin embargo, carece de tutoriales para ciertos aspectos más complicados y avanzados como la simulación de *ragdolls* y la disección de cuerpos rígidos. A pesar de su relativa dificultad de uso, ofrece multitud de funcionalidades:

- Detección de colisiones.
- Simulación dinámica de cuerpos rígidos.
- Simulación de ragdolls.
- Simulación de cuerpos blandos.
- Selección por *raycasting.*

5.5.4 Sonido

El sonido en los videojuegos es probablemente una de las partes más infravaloradas de la experiencia. Siempre parece un componente superfluo y un añadido, pero la realidad es que el sonido y la música juegan un rol muy importante de cara a la inmersión en el juego.

OpenAL
Licencia: LGPL.

OpenAL (Open Audio Library), es una librería de renderizado de audio posicional, esto es, es capaz de reproducir sonido simulando las características de su posición en el espacio como la atenuación, localización etc.

Figura 5.11 - Logo de OpenAL

Se ha utilizado OpenAL por su simplicidad y por la capacidad de éste para permitir usar códec propios. Es una librería eficiente con multitud de capacidades para simular eco, reverberación efecto doppler e incluso la reproducción del sonido en diferentes entornos como cuevas, espacios abiertos etc.

OggVorbis
Licencia: BSD.

Para recudir el uso de espacio de disco se utilizan códec de compresión para reducir el tamaño de los ficheros de audio. Esta librería ofrece un códec de compresión de audio muy eficiente y con un formato libre. El ratio de compresión y la calidad son muy similares a lo ofrecido por el conocido códec mp3.

5.5.5 Conexión de red

Enet
Licencia: MIT.

Enet es una librería de bajo nivel para facilitar la comunicación de red mediante una capa de comunicación basada en UDP (User Datagram Protocol). Enet permite opcionalmente establecer comunicación fiable y ordenada (como TCP) o con posibilidad de pérdida de paquetes.

El diseño de esta librería se basa en la simplicidad y la robustez por lo que no provee otras funcionalidades básicas como cifrado, autenticación y búsqueda de servidores. Esta librería fue desarrollada para el motor de videojuegos Cube2, utilizado en el juego Sauerbraten.

Es una librería muy sencilla pero efectiva y una de sus principales puntos fuertes es que no solo funciona en diferentes plataformas si no que permite la comunicación entre diferentes los diferentes sistemas operativos.

6. PRUEBAS

En este capítulo se describen la metodología y pruebas realizadas para validar el desarrollo de Caelum-Engine. En el capítulo XXX ya se ha comentado que se ha realizado este proyecto mayormente utilizando un desarrollo iterativo en el cual vamos realizando pruebas en cada iteración.

Para la comprobación del correcto funcionamiento del motor se deben dividir las pruebas en varias secciones:

- **Pruebas de unidad**: Pruebas realizadas sobre pequeños módulos o secciones de los subsistemas, es decir, se realizan pequeñas pruebas que validan cada una de las funcionalidades de los subsistemas (p.e. La decodificación de sonido del sistema audio)

- **Pruebas de Integración**: Pruebas realizadas para comprobar el funcionamiento conjunto de todos los módulos y características de un subsistema en conjunción.

- **Pruebas de sistema**: Pruebas finales que comprueban el funcionamiento del sistema en su conjunto y validan que la interacción entre los diferentes subsistemas se realiza de forma correcta.

A continuación se presentará una serie de tablas que describen las pruebas llevadas a cabo para testear la validez del proyecto.

Las tablas están divididas por secciones, diferenciando cada uno de los subsistemas del motor para describir las pruebas de unidad de cada clase y la integración de estas en el subsistema.

Posteriormente se realizan las últimas pruebas, las de sistema, correspondientes a las pruebas realizadas para la validación del motor de videojuegos en su conjunto.

6.1 PRUEBAS DEL NÚCLEO

Subsistema: Núcleo y memoria	Resultado test
Clase: ConfigFile	OK
Debe poder realizar una carga directa de ficheros.	OK
Debe poder cargar los pares clave valor de un fichero a una estructura multimap.	OK
Debe devolver el valor o valores correctos asociados a una clave.	OK
Clase: Log	OK
Debe mostrar por línea de comandos los mensajes introducidos.	OK
Debe poder volcar a fichero cada cierto tiempo los mensajes introducidos.	OK
Debe poder volcar sus propios mensajes al log designado como principal.	OK
Clase: LogManager	OK
Debe poder crear y almacenar la lista de logs creados.	OK
Debe ser capaz de cerrar cada log independientemente.	OK

Clase: PluginManager	OK
Debe ser capaz de cargar el fichero de configuración de plugins.	OK
Debe ser capaz de almacenar la localización de los plugins a cargar.	OK
Debe permitir registrar cargadores de plugins (PluginLoader).	OK
Debe ser capaz de cargar un plugin redirigiendo su carga automáticamente a su cargador (PluginLoader).	OK
Debe ser capaz de encolar la carga de plugins hasta que su PluginLoader asociado se registre.	OK
Clase: PreferenceManager	OK
Debe ser capaz de cargar el fichero de preferencias del motor.	OK
Debe ser capaz de realizar la conversión de formato de preferencias de string a int, bool, y float.	OK
Clase: ResourceManager	OK
Debe ser capaz de almacenar las rutas donde se encuentran los recursos.	OK
Debe ser capaz de localizar los ficheros que se encuentran en las rutas de recursos.	OK
Debe ser capaz de localizar el cargador de recursos asociado (ResourceLoader) para cada recurso dependiendo de la extensión de su extensión.	OK

Tabla 6.1 - Lista de pruebas del núcleo

6.2 PRUEBAS DEL SISTEMA DE JUEGO

Subsistema: Sistema de juego	Resultado test
Clase: GameLayer (abs)	OK
Debe permitir almacenar la lista de componentes creados.	OK
Debe permitir destruir un componente creado.	OK
Clase: GameManager	OK
Debe ser capaz de almacenar los estados de juego a ejecutar.	OK
Debe ser capaz de almacenar una cola de los estados de juego sucedidos.	OK
Debe ser capaz de ejecutar un ciclo de juego basado en el renderizado.	OK
Debe ser capaz de ejecutar un ciclo de juego independiente del renderizado.	OK
Debe ser capaz de recibir todos los eventos de entrada para transmitirlos al estado en curso.	OK
Clase: GameObject	OK
Debe ser capaz de crear y almacenar nuevos objetos de juego hijos de sí mismo.	OK
Debe ser capaz de almacenar componentes móviles (MovableComponents) añadidos a él.	OK
Debe ser capaz de desligar otros objetos de juego de su jerarquía y añadirlos a la suya propia.	OK
Debe ser capaz de almacenar y modificar su posición en el espacio tanto local, respecto a su padre y respecto a su posición global.	OK
Debe ser capaz de almacenar y modificar su orientación en el espacio tanto local, respecto a su padre y respecto a su posición global.	OK
Debe ser capaz de almacenar y modificar su escala en el espacio tanto local, respecto a su padre y respecto a su posición global.	OK

Debe ser capaz de notificar las modificaciones de su transformación espacial (posición, orientación y escala) a todos sus componentes.	OK
Clase: LayerManager	OK
Debe ser capaz de almacenar capas de juego (GameLayer)	OK
Clase: SceneManager	OK
Debe ser capaz de crear y almacenar y destruir escenas de juego.	OK
Debe ser capaz de establecer la escena de juego actual.	OK
Debe ser capaz de obtener escenas de juego en base a su nombre.	OK
Clase: Scene	OK
Debe crear y almacenar las capas de juego disponibles.	OK

Tabla 6.2 - Lista de pruebas de unidad del subsistema de juego

6.3 PRUEBAS DEL MODELO MATEMÁTICO

Subsistema: Modelo matemático	Resultado test
Clase: Radian	OK
Debe permitir crear copias profundas de un objeto Radian.	OK
Debe permitir comparaciones con otros radianes (<, >, >=, <=, ==).	OK
Debe permitir operaciones sencillas con otros Radian (+, -, *, /).	OK
Debe ofrecer su conversión de Radian a Degree.	OK
Clase: Degree	OK
Debe permitir crear copias profundas de un objeto Degree.	OK
Debe permitir comparaciones con otros Degree (<, >, >=, <=, ==).	OK
Debe permitir operaciones sencillas con otros Degree (+, -, *, /).	OK
Debe ofrecer su conversión de Degree a Radian	OK
Clase: Matrix3	OK
Debe ser capaz de crear copias profundas de un objeto Matrix3.	OK
Debe poder realizar comparaciones con otras matrices 3x3 (<, >, >=, <=, ==).	OK
Debe poder realizar operaciones sencillas con otras matrices 3x3 (+, -, *, /).	OK
Debe permitir el acceso a los valores de la matriz independientemente.	OK
Debe ser capaz de realizar la operación de trasposición.	OK
Debe ser capaz de realizar la operación de inversión.	OK
Debe ser capaz de calcular el determinante.	OK
Clase: Quaternion	OK
Debe permitir crear copias profundas de un objeto Quaternion.	OK
Debe permitir comparaciones con otros Quaternion (<, >, >=, <=, ==).	OK
Debe permitir operaciones sencillas con otros Quaternion (+, -, *, /).	OK
Debe permitir el acceso a los valores del Quaternion independientemente.	OK
Debe ser capaz de crearse o convertirse a partir de una matriz 3x3.	OK
Debe ser capaz de rotar un Vector3 a partir de un Quaternion.	OK
Clase: Vector3	OK
Debe permitir crear copias profundas de un objeto Vector3.	OK
Debe permitir comparaciones con otros Vector3 (<, >, >=, <=, ==).	OK
Debe permitir operaciones sencillas con otros Vector3 (+, -, *, /).	OK
Debe permitir el acceso a los valores del vector independientemente.	OK
Debe ser capaz de calcular la longitud de un Vector3.	OK

Debe ser capaz de calcular la distancia entre dos Vector3.	OK
Debe ser capaz de calcular el producto cruzado o producto vectorial entre dos Vector3.	OK
Debe ser capaz de normalizar sus valores.	OK
Debe ser capaz de calcular el ángulo entre dos Vector3.	OK
Clase: Math	OK
Debe proveer generación de números aleatorios.	OK
Debe ser capaz de realizar operaciones matemáticas básicas (seno, coseno, tangente, arcoseno, arcocoseno y arcotangente).	OK
Debe poder realizar redondeos (a la alza y baja, así como truncar)	OK

Tabla 6.3 - Lista de pruebas del modelo matemático

6.4 PRUEBAS DEL SUBSISTEMA DE INPUT

Subsistema: Input	Resultado test
Clase: InputManager	OK
Debe ser capaz de detectar el número de dispositivos conectados	OK
Debe ser capaz de almacenar la lista de dispositivos creados.	OK
Debe permitir registrar y almacenar “listeners” por cada tipo de dispositivo.	OK
Debe capturar los eventos de los dispositivos y transmitirlos por los listeners registrados.	OK
Debe ser capaz de recibir eventos de pantalla y actualizar los parámetros de ésta para el control del ratón.	OK

Tabla 6.4 - Lista de pruebas del subsistema de entrada

6.5 PRUEBAS DEL SUBSISTEMA DE RENDER

Subsistema: Render	Resultado test
Clase: RenderManager	OK
Debe ser capaz de crear una ventana donde realizar el renderizado.	OK
Debe permitir elegir el sistema de renderizado a utilizar (OpenGL o Directx)	OK
Debe permitir almacenar y registrar listeners de la ventana y retransmitirles los eventos.	OK
Debe ser capaz de crear y almacenar capas de renderizado o “RenderLayer” que añadir a las escenas.	OK
Debe ser capaz de proveer un ciclo de juego “loop” basado en eventos de renderizado.	OK
Clase: RenderLayer	OK
Debe ser capaz de crear y almacenar componentes de cámara “Camera”.	OK
Debe ser capaz de crear y almacenar componentes de luces “Light”	OK
Debe ser capaz de crear cargar y almacenar componentes de modelos 3D “Entity”.	OK
Debe ser capaz de crear un componente de renderizado de terreno “Terrain”.	OK

Debe ser capaz de crear un componente de simulación atmosférica "RealisticSky"	OK
Debe ser capaz de fijar la luz ambiental de la escena.	OK
Debe ser capaz de fijar y ejecutar las técnica de renderizado de sombras por plantilla y basada en texturas	OK
Debe ser capaz de fijar el color de las sombras.	OK
Debe ser capa de fijar y ejecutar las técnicas de proyección de sombras simple, enfocada, LiSPSM y PSSM.	OK
Clase: Camera	OK
Debe ser capaz de renderizar en proyección en perspectiva y ortogonal.	OK
Debe ser capaz de fijar el campo de visión vertical.	OK
Debe ser capaz de fijar las distancias de visión (cercana y lejana)	OK
Debe ser capaz de modificar el tipo de renderizado a completo o *wireframe*	OK
Clase: Entity	OK
Debe ser capaz de cargar un modelo 3D de un fichero con extensión .mesh y detectar si tiene un esqueleto asociado.	OK
Debe ser capaz de almacenar un esqueleto asociado	OK
Debe ser capaz de fijar el material de renderizado.	OK
Clase: Skeleton	OK
Debe ser capaz de cargar el esqueleto y animaciones disponibles desde un fichero con extensión .skeleton	OK
Debe ser capaz de almacenar el estado de las diferentes animaciones cargadas.	OK
Debe ser capaz de modificar la malla de una entidad 3D.	OK
Debe ser capaz de reproducir las animaciones almacenadas en la entidad asociada.	OK
Debe ser capaz de aplicar pesos a cada animación.	OK
Debe ser capaz de reproducir la mezcla de animaciones con distintos pesos.	OK
Clase: AnimationState	OK
Debe ser capaz de almacenar el estado de la animación en un momento dado.	OK
Debe ser capaz de fijar la propiedad de loop (ciclo sin fin) y la velocidad de animación)	OK
Clase: Light	OK
Debe ser capaz de representar puntos de luz.	OK
Debe ser capaz de representar focos de luz.	OK
Debe ser capaz de representar luces direccionales.	OK
Debe permitir fijar el color difuso de la luz.	OK
Debe permitir fijar el color especular de la luz.	OK
Clase: RealisticSky	OK
Debe ser capaz de fijar las propiedades definidas en la clase SkySettings en tiempo real.	OK
Debe permitir modificar la velocidad de simulación en tiempo real.	OK
Clase: RenderComponent	OK
Debe ser capaz de conectar los componentes de renderizado a los objetos de juego.	OK
Debe ser capaz de conectar directamente los nodos de Ogre3D con los componentes de renderizado.	OK

Clase: RenderWindow	OK
Debe ser capaz de presentar la ventana de renderizado.	OK
Debe permitir registrar y almacenar listeners de los eventos de ventana.	OK
Debe ser capaz de informar de los eventos de ventana a los listeners.	OK
Clase: Terrain	OK
Debe ser capaz de cargar e interpretar la textura del mapa de altura.	OK
Debe ser capaz de cargar e interpretar un fichero con extensión .dat de información de terreno.	OK
Debe ser capaz de generar un mapa de luz estática dada una luz direccional.	OK
Debe ser capaz de ofrecer la información de la altura del terreno en un punto dado.	OK
Clase: GuiManager	OK
Debe ser capaz de cargar ficheros con extensión .layout y generar una interfaz.	OK
Debe ser capaz de recibir eventos de dispositivos de entrada.	OK
Debe ser capaz de registrar y almacenar funciones callback para botones.	OK

Tabla 6.5 - Lista de pruebas del subsistema de render

6.6 PRUEBAS DEL SUBSISTEMA DE SONIDO

Subsistema: Sonido	Resultado test
Clase: AudioManager	OK
Debe ser capaz de crear y almacenar capas de sonido “AudioLayer”	OK
Debe ser capaz de detectar los dispositivos de audio e inicializarlo.	OK
Debe ser capaz de modificar el volumen general del sistema.	OK
Clase: AudioLayer	OK
Debe ser capaz de crear, almacenar y gestionar un receptor de audio.	OK
Debe ser capaz de crear y almacenar fuentes de sonido.	OK
Clase: AudioSource	OK
Debe ser capaz de pedir una instancia del sonido a reproducir.	OK
Debe ser capaz de reproducir el buffer de sonido elegido.	OK
Clase: AudioReceptor	OK
Debe ser capaz de representar al receptor de audio.	OK
Debe ser capaz de sincronizar su transformación espacial con un objeto de juego.	OK
Clase: AudioBuffer	OK
Debe ser capaz de almacenar la información de audio en formato *raw*.	OK
Clase: AudioLoader	OK
Debe ser capaz de cargar ficheros con la extensión .ogg.	OK
Debe ser capaz de decodificar el audio ogg vorbis a raw.	OK

Tabla 6.6 - Lista de pruebas del subsistema de sonido

6.7 PRUEBAS DEL SUBSISTEMA DE FÍSICAS

Subsistema: Físicas	Resultado test
Clase: PhysicsManager	OK
Debe ser capaz de crear y almacenar las capas de físicas “PhysicsLayer”.	OK
Clase: PhysicsLayer	OK
Debe ser capaz de crear y almacenar los cuerpos rígidos.	OK
Debe ser capaz de crear una malla de colisión del terreno dados los puntos de altura.	OK
Debe ser capaz de comunicar al sistema de renderizado las aristas de las mallas de colisión asociadas a cuerpos rígidos.	OK
Debe ser capaz de crear objetos estáticos o dinámicos.	OK
Clase: RigidBody (bulletRigidBody)	OK
Debe ser capaz de responder a fuerzas aplicadas (p.e. gravedad)	OK
Debe ser capaz de responder a impulsos aplicados (p.e. explosión)	OK
Debe ser capaz de actualizar el objeto de juego al que esté asociado.	OK
Debe ser capaz de responder a colisiones con otros cuerpos rígidos.	OK
Clase: MeshToShapeConverter	OK
Debe ser capaz de crear una malla de colisión de una esfera circunscrita a una entidad3D.	OK
Debe ser capaz de crear una malla de colisión de una caja que rodee una entidad3D.	OK
Debe ser capaz de crear una malla de colisión de un cilindro que rodee una entidad3D.	OK
Debe ser capaz de crear una malla de colisión convexa creada a partir de una entidad3D	OK

Tabla 6.7 - Lista de pruebas del subsistema de físicas

6.8 PRUEBAS DEL SUBSISTEMA DE RED

Subsistema: Red	Resultado test
Clase: NetWorkManager	OK
Debe ser capaz de crear y almacenar capas de servidor de juego GameServer.	OK
Debe ser capaz de crear y almacenar capas de cliente de juego GameClient.	OK
Debe ser capaz de aceptar y almacenar conexiones remotas.	OK
Clase: ServerLayer	OK
Debe ser capaz de crear y almacenar componentes de juego servidor ServerObject.	OK
Debe ser capaz de enviar información de sincronización de la escena a los clientes.	OK
Clase: ClientLayer	OK
Debe ser capaz de crear y almacenar componentes de juego cliente GameClient.	OK
Debe ser capaz de recibir la información de sincronización del servidor.	OK
Debe ser capaz de actualizar la escena según la sincronización del servidor.	OK

Debe ser capaz de mantener una simulación predictiva independiente al servidor.	OK
Clase: ServerObject	OK
Debe ser capaz de almacenar y mantener la información de servidor.	OK
Debe ser capaz de interpretar la lógica del cliente para replicarla en el objeto servidor.	OK
Clase: ClientObject	OK
Debe ser capaz de actualizar el objeto de juego al que está asociado.	OK
Debe ser capaz de utilizar las predicciones de la capa cliente para actualizarse.	OK
Debe ser capaz de enviar al servidor la transformación espacial del objeto de juego para ser sincronizado.	OK

Tabla 6.8 - Lista de pruebas del subsistema de red

7. MANUAL DE USUARIO

El presente capítulo se utilizará para tratar de introducir los primeros pasos de uso del motor Caelum-Engine y las posibilidades disponibles para añadir contenido y sacar el máximo partido de la experiencia de éste motor.

Los pasos a seguir a la hora de utilizar Caelum-Engine son los siguientes:

- **Empezando con Caelum-Engine**: Esta sección consta de varios tutoriales de uso del motor de videojuegos, desde el arranque del motor a la preparación de una escena con modelos 3D, sonidos/música y una simulación física completa.

- **Renderizado avanzado**: En este apartado se entrará más en detalle en la descripción de los diferentes elementos de renderizado especiales, como sistemas de partículas, efectos de post procesado y la codificación de materiales.

- **Simulación de red**: En éste episodio se utilizará un tutorial para instruir en la utilización del sistema de comunicación de red. Se presentará la creación de un servidor y un cliente básico así como la sincronización de los objetos de juego y los distintos tipos de comunicación.

- **Extendiendo Caelum-Engine**: Sección que ofrece una explicación práctica para extender el sistema de memoria y recursos del motor, así como el sistema de capas de juego que permite añadir nuevos componentes y lógica al juego.

- **Herramientas externas**: En este apartado se comentan varias suites y programas externos para crear contenido compatible con Caelum-Engine.

7.1 EMPEZANDO CON CAELUM-ENGINE

En este tutorial se van a introducir los conceptos básicos para utilizar Caelum-Engine: Estados de juego, escenas, capas, objetos y componentes.

Según el tutorial progresa iremos añadiendo código lentamente al proyecto y viendo los resultados a medida que avanzamos. Es altamente recomendable seguir paso a paso el tutorial para evitar dejarnos algo en el camino.

Prerrequisitos

- Este tutorial asume que usted tiene conocimientos de programación en C++ a nivel medio y conoce el funcionamiento básico de la librería estándar STL.

- Este tutorial asume que usted ha sido capaz de preparar y compilar una aplicación añadiendo la librería Caelum-Engine como dependencia.

7.1.1 Comenzando con Caelum-Engine

En primer lugar comenzaremos la experiencia con Caelum-Engine aprendiendo a arrancar y apagar el motor. Para realizar esto utilizaremos el siguiente código.

```
#include "core/gameengine.h"

int main() {
    // 1. Creación de la instancia del motor
    Caelum::GameEngine *ce = new Caelum::GameEngine();
    // 2. Arranque del motor
    ce->setup();
    // 3. Parada del motor
    ce->shutdown();
    // 4. Destrucción de la instancia del motor
    delete ce;
    return 0;
}
```

Código 7.1 - main.cpp (v1) - Arranque y parada de Caelum-Engine

En este trozo de código simplemente arrancamos y apagamos el motor sin utilizarlo. Para comenzar a utilizar el motor debemos añadir un estado de juego a la secuencia de juego.

Un estado de juego en Caelum-Engine representa cada una de las distintas etapas en un juego diferenciadas por el tipo de tratamiento necesario para la lógica de la escena.

Dos ejemplos de estados de juego son el estado de juego de menú principal, el de un nivel de juego, o el estado de juego de uso de inventario en un juego. Cada uno de estos estados de juego utilizan y procesan de diferente forma los eventos, en el menú es posible seleccionar secciones, en el nivel de juego sería posible controlar un personaje y en el inventario mover y utilizar objetos.

Añadamos los siguientes ficheros

```
#include "game/gamestate.h"

class TestState : public Caelum::GameState {
  public:
    // ***************************
    // Sección 1 - Estado de juego
    // ***************************
    void enter();
    void exit();
    void pause();
    void resume();

    // ****************************
    // Sección 2 - Ciclo de juego
    // ****************************
    bool preRenderUpdate(const Caelum::RenderEvent& evt);
    bool renderingUpdate(const Caelum::RenderEvent& evt);
    bool postRenderUpdate(const Caelum::RenderEvent& evt);
```

```
    // ***************************
    // Sección 3 - Eventos de ratón
    // ***************************
    bool mouseMoved    (const Caelum::MouseEvent& evt);
    bool mousePressed  (const Caelum::MouseEvent& evt,
                        Caelum::MouseButtonID id);
    bool mouseReleased (const Caelum::MouseEvent& evt,
                        Caelum::MouseButtonID id);
    bool mouseClicked  (const Caelum::MouseEvent& evt,
                        Caelum::MouseButtonID id);

    // ****************************
    // Sección 4 - Eventos de teclado
    // ****************************
    bool keyPressed  (const Caelum::KeyEvent &evt);
    bool keyReleased (const Caelum::KeyEvent &evt);
    bool keyTap      (const Caelum::KeyEvent &evt);

    // ******************************
    // Sección 5 - Eventos de joysticks
    // ******************************
    bool buttonPressed (const Caelum::JoyStickEvent &arg, int button);
    bool buttonReleased(const Caelum::JoyStickEvent &arg, int button);
    bool axisMoved     (const Caelum::JoyStickEvent &arg, int axis);
    bool sliderMoved   (const Caelum::JoyStickEvent &arg, int index);
    bool povMoved      (const Caelum::JoyStickEvent &arg, int index);
    bool vector3Moved  (const Caelum::JoyStickEvent &arg, int index);

    // ********************************
    // Sección 6 - Funciones del tutorial
    // ********************************
    void createCamera();
    void setupShadows();
    void createCharacter();
    void createEnvironment();

  private:
    // Escena de juego y capas de gestión de contenido
    Caelum::Scene mScene;
    Caelum::RenderLayer *mRenderLayer;
    Caelum::AudioLayer *mAudioLayer;
    Caelum::PhysicsLayer *mPhysicsLayer;
    // Cámara
    Caelum::GameObject *mCamObj;
    Caelum::Camera *mCam;
    // Personaje
    Caelum::GameObject *mCharacterObj;
    Caelum::Entity *mCharacterEnt;
    // Entorno
    Caelum::Light *mSunLight;
    Caelum::Terrain *mTerrain;
    Caelum::RealisticSky *mSky;
    // Audio
    Caelum::AudioReceptor *mReceptor
    Caelum::AudioSource *mSource;
}
```

Código 7.2 - teststate.h - Cabecera del estado de juego Test

En principio la cabecera de nuestro estado de prueba puede parecer abrumadora, sin embargo es bastante sencillo una vez lo analizamos por secciones.

1. **Estado de juego**: Las funciones enter() y exit() son llamadas cuando se crea y se destruye un estado respectivamente. Las funciones pause() y resume() se llaman cuando se para o se recupera un determinado estado de juego.

2. **Ciclo de juego**: Las funciones de control del ciclo de juego son 3 y todas ellas son llamadas una vez cada fotograma.

    a. **preRenderUpdate()**: Este callback es llamado antes de que los elementos gráficos se vayan a actualizar y la geometría se envíe a la GPU.

    b. **renderingUpdate()**: Esta función se llama después de que las órdenes de renderizado han sido enviadas a la gráfica.

    Debido al carácter asíncrono de la GPU, ésta función aprovecha el tiempo muerto hasta que la GPU termina el renderizado. Si la GPU termina antes que este callback, se interrumpe momentáneamente el procesamiento de esta función hasta que se termina de atender a la GPU.

    c. **postRenderUpdate()**: Este callback es llamado justo después de que todos los elementos han sido renderizados y se ha cargado el buffer de pantalla.

3. **Eventos de ratón**: Estas funciones se ejecutan cada vez que un evento de ratón se produce, es decir, se mueve el ratón o se presiona o suelta un botón.

4. **Eventos de teclado:** Similar a los eventos de ratón estas funciones se ejecutan cada vez que un evento de teclado se produce.

5. **Eventos de joystick:** Callback que se ejecutan cada vez que se produce alguno de los eventos indicados en un joystick. La variedad de funciones para joysticks es mayor porque da soporte a multitud de tipos de joysticks, incluidos dispositivos de movimiento.

6. **Funciones del tutorial**: Estas funciones darán cabida a cada uno de los pasos a seguir en este tutorial en los cuales las iremos rellenando y explicando progresivamente.

En este punto debemos modificar nuestro arranque del motor para incluir nuestro estado.

```
#include "core/gameengine.h"
#include "teststate.h"

int main() {
    // 1. Creación de la instancia del motor
    Caelum::GameEngine *ce = new Caelum::GameEngine();
    // 2. Creación de la instancia del estado de juego
    TestState *state = new TestState();
    // 3. Arranque del motor
```

```
    ce->setup();
    // 4. Añadir el estado de juego al motor
    ce->getGameManager()->addState("TestState", state, true);
    // 5. Comienzo del game loop o ciclo de juego
    ce->getGameManager()->start();
    // 6. Parada del motor
    ce->shutdown();
    // 7. Destrucción del estado de juego
    delete state;
    // 8. Destrucción de la instancia del motor
    delete ce;
    return 0;
}
```

Código 7.3 - main.cpp (v2) - Arranque y adición de un estado de juego en Caelum-Engine

Llegados aquí hemos creado la cabecera de nuestro estado de juego y la hemos incluido al motor, si realizásemos una implementación vacía de los métodos de nuestro estado de juego veríamos como al ejecutar el juego se inicializaría una ventana en negro. Ahora bien, nosotros queremos añadir contenido a nuestro juego así que en la próxima parte del tutorial vamos a explicar cómo deberíamos implementar nuestro estado de juego para crear una escena.

7.1.2 Creando la primera escena

En esta sección del tutorial vamos a explicar en qué consisten las escenas de juego y cómo podemos gestionarlas.

Una escena la representación del espacio tridimensional de nuestro juego, en ella podremos añadir objetos y contenidos y visualizarlo todo a través de las cámaras. Existe la posibilidad de crear diferentes escenas de juego para optimizar cada una de ellas al tipo de mapa o nivel que queramos, para espacios cerrados o mundos abiertos por ejemplo.

Para crear nuestra escena añadiremos el siguiente código.

```
#include "teststate.h"

#include "game/scenemanager.h"

using namespace Caelum;

void TestState::enter() {
    // 1. Creamos una escena optimizada para exteriores.
    mScene = mSceneManager->createScene("TestScene","EXTERIOR CLOSE");
    // 2. Obtenemos las capas de gestión de contenido
    mRenderLayer = mScene->getRenderLayer();
    mAudioLayer = mScene->getAudioLayer();
    mPhysicsLayer = mScene->getPhysicsLayer();

    // 3. Creación de cámara
    createCamera();
    // 4. Preparación del modelo de sombras
    setupShadows();
    // 5. Carga de un modelo 3D
    void createCharacter();
```

```
    // 6. Carga de terreno y simulación atmosférica y agua
    void createEnvironment();
    // 7. Carga y reproducción de audio
    void loadMusic();
    // 8 Preparación de físicas
    void setupPhysics();
}
```

Código 7.4 - teststate.cpp – Método TestState::enter

1. En primer paso pedimos al gestor de escenas que cree la escena que utilizaremos para añadir contenido. La clase gamestate incorpora un puntero al gestor de escenas(mSceneManager) y otro puntero preparado para almacenar una escena (mScene). En este caso elegimos una escena optimizada para exteriores, lo que supone que el motor de renderizado prepara una capa con soporte para creación de terreno abierto y que el motor de físicas prepara una capa optimizada para las dimensiones de un mundo relativamente grande.

2. En segundo lugar obtenemos de nuestra escena los punteros a los diferentes gestores de contenido o capas, los cuales nos permitirán crear componentes que añadir a la escena.

Iremos rellenando y comentando la utilidad de las siguientes funciones a lo largo del tutorial.

Añadir contenido a la escena es muy sencillo en Caelum-Engine, una vez hemos creado nuestra escena y tenemos los diferentes gestores de la escena, solo nos queda crear un objeto de juego y añadirle contenido.

Cámara

Antes de nada y para poder visualizar todo lo que hagamos, necesitamos crear una cámara. Para ello implementaremos el método TestState::createCamera().

```
// 3. Creación de un objeto y un componente cámara
void TestState::createCamera() {
    // Creación
    mCamObj = mScene->createGameObject("CameraObject"); // a
    mCam = renderLayer->createCamera("MainCamera"); // b
    mCamObj->attachComponent(mCam); // c
    // Configuración
    mCamObj->setPosition(0, 7, 12); // d
    mCamObj->setFixedYawAxis(true); // e
    mCam->setAsActiveCamera(); // f
}
```

Código 7.5 – teststate.cpp - Método TestState::createCamera()

En el paso 3 hemos realizado lo siguiente:

3. Creación de la cámara

a. Hemos pedido a la escena que cree un objeto de juego, que por defecto se crea en el origen de coordenadas.
b. Hemos pedido a la capa de renderizado que cree una cámara.
c. Hemos insertado la cámara en juego a través del objeto de juego.
d. Fijamos la posición del objeto de juego (y por tanto de la cámara añadida).
e. Establecemos que el eje Y no pueda moverse, lo que evitará que la cámara se incline lateralmente.
f. Establecemos que la cámara creada sea la que se utilice para dibujar actualmente en pantalla.

Configuración de sombras

```
// 4. Configuración de sombreado
void TestState::setupShadows() {
    mScene->getRenderLayer()->setAmbientLight(
                              Caelum::ColourValue(0.2,0.2,0.2)); // a
    mScene->getRenderLayer()->setShadowTechnique(
                        Caelum::SHADOWTYPE_TEXTURE_MODULATIVE); // b
    mScene->getRenderLayer()->setShadowProjectionType(
                             Caelum::SHADOW_PROJECTION_LISPSM); // c
    mScene->getRenderLayer()->setShadowFarDistance(400); // d
    mScene->getRenderLayer()->setShadowTextureSettings(512, 1); // e
}
```

Código 7.6 - teststate.cpp – Método TestState::setupShadows()

4. Configuración de sombras
    g. Establecemos una luz ambiental mínima que ilumine tenuemente la escena incluso sin luz.
    h. Determinamos el modelo utilizado para la generación de sombras. En este caso vamos a utilizar “SHADOWTYPE_TEXTURE_MODULATIVE” con el cual se crearán y dibujarán las sombras basadas en la proyección de los modelos 3D a un buffer de textura en lugar de realizar el cálculo completo como en el modelo “STENCIL”. Este modelo de sombras resultan mucho más eficientes pero requiere mayor parametrización. (consultar más modelos de sombras en render/renderlayer.h)
    i. Fijamos el tipo de técnica de proyección de sombras desde la textura al mundo virtual. Utilizaremos la proyección LiSPSM la cual ofrece unos resultados visualmente similares al modelo de sombreado STENCIL, pero ofrece un rendimiento muy superior. (consultar más modelos de proyección en render/renderlayer.h)
    j. Establecemos el tamaño de los buffers de sombreado y el número de buffers de sombras por cada modelo 3D en la escena.
    Nota*: el número de buffers disponibles indica el número máximo de sombras posibles para un mismo objeto en un momento dado.

Entidades3D

Para comenzar, vamos a añadir a nuestra escena un modelo 3D representado por la clase “Entity” en Caelum-Engine. Cargar y añadir un modelo es algo tan sencillo como lo siguiente.

```
// 5. Carga de un modelo 3D
void TestState::createCharacter() {
    mCharacterObj = mScene->createGameObject("CharObject"); // a
    mCharacterEnt = mRenderLayer->createEntity(
                                "CharEnt", "Sinbad.mesh"); // b
    mCharacterObj->attachComponent(mCharacterEnt); // c

    AnimationState* anim = charEnt->getAnimation("Dance"); // d
    anim->setLoop(true);
    anim->start(2);
    ent = charEnt;

    mCamObj->lookAt(charObj); // e
}
```

Código 7.7 - teststate.cpp - Método TestState::createCharacter()

5. Carga de un modelo 3D
    a. Creamos un objeto de juego donde añadir el modelo.
    b. Cargamos el modelo 3D pasando el nombre que tendrá la entidad y el nombre del fichero de donde debemos cargar el modelo.
    c. Añadimos la entidad 3D al objeto de juego para que sea visible en nuestra escena.
    d. Obtenemos una animación, fijamos la propiedad loop para que se ejecute en ciclo constante y comenzamos a ejecutarla pero con un fundido de 2 segundos, es decir la animación irá ganando peso hasta verse completa al de 2 segundos.
    e. Hacemos que nuestra cámara mire directamente a nuestro nuevo modelo.

Elementos ambientales

Como hemos comentado al principio del tutorial nuestra escena va a estar optimizada para exteriores. Es el momento de que comencemos a añadir un entorno visual a nuestra escena. En esta parte vamos a cargar un terreno y una simulación atmosférica.

En cuanto a la creación del terreno existen varias formas de cargarlo.

- La forma más fácil de cargar un terreno consiste en cargar un fichero de terreno con formato binario .dat el cual contiene ya toda la información sobre el relieve del terreno material texturas y dimensiones.

- Otra forma de cargar un terreno, consiste en generarlo directamente en base a un *HeightMap* para definir el relieve del terreno y después definir sus texturas de cualquiera de las siguientes formas:

    - Y utilizar una serie de normas para definir las texturas del mapa dependiendo de la altitud del terreno.

    - O utilizar una técnica conocida como *texture-splatting*, en el cual a través de una textura extra y dependiendo del color utilizado en cada zona indica la textura a utilizar en ese lugar del mapa.

Para simplificar este tutorial vamos a utilizar la primera técnica de carga de terreno.

```
// 6. Carga de terreno y simulación atmosférica y agua
void TestState::createEnvironment() {
    // a) Luz
    mSunLight = mRenderLayer->createLight(
                                "SunLight", Light::LT_DIRECTIONAL);
    mSunLight->setDirection(0.55, -1.5, -0.75);
    mScene->getRootObject()->attachComponent(mSunLight);

    mSunLight->setDiffuseColour(ColourValue::White);
    mSunLight->setSpecularColour(ColourValue(0.4, 0.4, 0.4));

    // b) Terreno
    mTerrain = mRenderLayer->createTerrain("terrain", 513, 4000);
    mTerrain->configureImport(500, 8, 8, 3000, 20);
    mTerrain->configureLight(mSunLight);
    mTerrain->setTile(0, 0, "");

    // c) Simulación atmosférica
    mSky = mRenderLayer->createRealisticSky("Sky");
    mSky->setPreset(Caelum::RealisticSky::SKY_CLEAR);
}
```

Código 7.8 - teststate.cpp - Método TestState::createEnvironment()

6. Carga de terreno y simulación atmosférica y agua

    a. Como viene siendo habitual creamos un componente, esta vez de luz direccional y lo añadimos a un objeto de la escena. Como no necesitamos mover la luz, ya que es direccionar utilizaremos el objeto de juego raíz para añadir la luz. Posteriormente fijamos el color de nuestra iluminación.

    b. Creamos una instancia de terreno que se compondrá de 1 a N terrenos más pequeños los cuales estarán definidos por 513x513 puntos de altura y supondrán 4000x400 unidades de distancia cada uno.

    Establecemos una escala de altura de 500 unidades y la posibilidad de uso de hasta 8 texturas diferentes y junto con otras propiedades extras de optimización.

    Establecemos la generación de sombras pre-calculadas para la iluminación creada, técnica concida comúnmente como Light-mapping [13].

    Por último cargamos el tile de terreno 0 x 0. Al utilizar el primer método de carga se utilizará el nombre provisto para el terreno y las coordenadas dadas para la búsqueda del fichero .dat, en este caso “terrain0x0.dat”.

    c. La simulación de cielo es bastante sencilla de crear simplemente pedimos la capa de render que nos cree la simulación atmosférica y posteriormente establecemos los parámetros de la simulación. Dado que existen multitud de

parámetros en este caso utilizamos una de las varias preselecciones que Caelum-Engine ofrece.

Sonido

La carga y reproducción de sonidos utiliza un esquema similar a la carga de entidades 3D o cualquier otro contenido, se necesita añadir un componente a un objeto de juego.

En el caso del sonido existen 2 componentes, “audiosource” y “audioreceptor”. Pueden existir N emisores, pero sólo 1 receptor.

```
// 7. Carga y reproducción de audio
void TestState::loadMusic() {
    // a) Creamos el receptor de audio
    mReceptor = mAudioLayer->createReceptor("AudioReceptor");
    mCamObj->attachComponent(mReceptor);
    // b) Creamos un sonido
    mSource = mAudioLayer->createSource("AudioSource", "music.ogg");
    mCharacterObj->attachComponent(mSource);
    mSource->play();
}
```

Código 7.9 - teststate.cpp - Método TestState::loadMusic()

7. Carga y reproducción de audio
    a. Creamos el componente receptor y lo añadimos al objeto de la cámara
    b. Creamos un componente de fuente de audio que utilice el sonido “music.ogg” y lo unimos al objeto del personaje que habíamos creado.

Simulación física y eventos

La simulación de cuerpos físicos en Caelum-Engine es un proceso sencillo e intuitivo. Al igual que el funcionamiento carga de sonido y modelos 3D, todo funciona con componentes que añadir a los objetos de juego.

```
// 8 Preparación de físicas
void TestState::setupPhysics() {
    // a) Creación de malla de colisión de terreno
    mPhysicsManager->createStaticTerrain(mTerrain);
}

// b) Evento de click de ratón
bool TestState::mousePressed(
            const Caelum::MouseEvent& evt, Caelum::MouseButtonID id)
{
    static int barrelcount = 0; // barrel number
    static String count;        // barrel number as string

    std::ostringstream convert;   // stream used for the conversion
    convert << barrelcount++;
    count = convert.str();
    // c) Creación del objeto, entidad y cuerpo rígido
    GameObject *barrelObj = mScene->createGameObject(
```

```
                                                String("barrelobj")+count);
    Entity *barrelEnt =    mRenderLayer->createEntity(
                               String("barrel")+count, "Barrel.mesh");
    RigidBody *barrelPhy = mPhysicsLayer->createRigidBody(
                                    String("barrelphy")+count,
                                    barrelEnt,
                                    Vector3::UNIT_SCALE, 1,
                                    PhysicsLayer::PHY_SHAPE_CONVEX);
    barrelObj->attachComponent(barrelEnt);
    barrelObj->attachComponent(barrelPhy);
    // d) Lanzamiento del barril
    barrelObj->setPosition(camobj->getPosition());
    barrelObj->notifyPosition();
    barrelPhy->applyImpulse(camobj->getOrientation().zAxis() * -50);
    return true;
}
```

Código 7.10 - teststate.cpp - Método TestState::setupPhysics() y TestState::mousePressed()

8. Preparación de físicas
    a. Creamos una malla de colisión del terreno
    b. Utilizamos la función de evento de pulsación del ratón
    c. Se creará un objeto de juego, una entidad3D de un barril y por último un objeto rígido para la simulación física que obtiene la malla de colisión a través del modelo 3D convertido a una malla convexa.
    d. Se establece la posición del objeto barril en el mismo lugar que la cámara y se notifica a todos los componentes (entidad de renderizado y cuerpo rígido de las físicas), y se le aplica un impulso al barril que lo lanza alejando el barril de la cámara.

7.2 CARACTERÍSTICAS AVANZADAS

En el anterior tutorial hemos preparado una escena bastante completa y hemos enseñado las características funcionales más importantes del motor, pero esto no es lo único de lo que Caelum-Engine es capaz.

Probablemente la característica más potente de un motor de videojuegos es precisamente el proceso de renderizado y la personalización de los gráficos. Hablaremos por un lado de las características de personalización de gráficos.

Como ya hemos comentado, el motor de renderizado de Caelum-Engine ofrece soporte para multitud de funciones de las cuales hemos utilizado únicamente unas pocas. Una de las funcionalidades más potentes es el sistema de materiales.

Caelum-Engine cargará automáticamente cualquier material incluido en la declaración de recursos cuando sea necesario. Para definir un material se necesita crear un fichero de texto con extensión ".material", en un mismo fichero se pueden definir muchos materiales diferentes. Una vez cargado un fichero de materiales se identificará cada material únicamente por el nombre indicado en su declaración y no por el fichero donde se declara.

Un material permite configurar algunos parámetros utilizados para renderizar entidades como el color, texturas o shaders a usar.

- La definición de un material se compone de 1 o más técnicas. Un mismo material puede utilizar cada técnica en un momento necesario, por ejemplo cuando necesita diferentes niveles de calidad visual.

- Una técnica se compone de 1 o más pases. Cada pase supone una operación de renderizado. Una técnica con más de un pase implica que el objeto al que se aplica se dibujará tantas veces como pases se ejecuten. Los pases tienen multitud de opciones a configurar, como la iluminación, uso del buffer de profundidad (zbuffer) etc.

- Un pase puede contener 0 o más unidades de textura. Una unidad de textura referencia una textura, además de algunos parámetros a aplicar a la textura como el tipo de coordenadas UV (Blender/3dsmax etc.) El tipo de filtro y el nivel de detalle a utilizar y otros muchos parámetros.

Un ejemplo muy básico de script de material es el siguiente:

```
material M_Lighting+OneTexture
{
    technique
    {
        pass
        {
            lighting on
            ambient 0.3 0.3 0.3
            diffuse 0.8 0.8 0.8 1.0
            specular 1.0 1.0 1.0 1.0 64.0
            emissive 0.1 0.1 0.1 1.0
            texture_unit
            {
                texture SimpleTexture.bmp 2d
                tex_coord_set 0
            }
        }
    }
}
```

Código 7.11 - Script de material básico, iluminación y uso de textura.

En el material propuesto en Código 7.11 es un ejemplo de un material básico que permite iluminación en el material y la aplicación de una textura.

En el pase fijamos el color ambiente, difuso, especular y la iluminación propia o emisiva mediante el formato de color RGB y RGBA. En el caso del color especular el último parámetro corresponde a la fuerza o dureza del brillo. Por último se carga y se aplica una textura 2D al material. Esto resulta solo una introducción a la creación de materiales pero para más información puede consultar [14] y [15].

Utilización de shaders

La creación de materiales ofrece multitud de posibilidades y parámetros configurables. En cualquier caso los scripts de materiales permiten utilizar shaders personalizados, tanto en ensamblador como lenguajes de alto nivel HSLSL, GLSL y CG los cuales resultan mucho más recomendables y portables.

Actualmente los shaders funcionan de 3 formas:

- Vertex Shaders: este tipo de shaders se ejecutan por cada vértice donde se puede modificar su geometría, color etc.
- Geometry Shaders: estos shaders aparecieron a partir de DirectX10 y OpenGL 3.0, en ellos es posible definir la adición de vértices extra en cada cara, lo que se conoce popularmente como teselado.
- Fragment Shaders: después del proceso de rasterización, proceso en el cual se determinan los pixeles de las caras a dibujar en pantalla, se determinará el color final de éstos pixeles.

Antes de utilizar los shaders en los materiales, necesitamos realizar una declaración de éstos en un fichero ".program".

```
vertex_program Ogre/BasicVertexPrograms/AmbientOneTextureCg cg
{
    source Example_Basic.cg
    entry_point ambientOneTexture_vp
    profiles vs_1_1 arbvp1

    default_params
    {
        param_named_auto worldViewProj worldviewproj_matrix
        param_named_auto ambient ambient_light_colour
    }
}
fragment_program Ogre/BasicFragmentPrograms/DiffuseOneTextureCg cg
{
    source Example_Basic.cg
    entry_point diffuseOneTexture_fp
    profiles ps_2_0 arbfp1
}
```

Código 7.12 – Declaración de vertex y fragment shader de CG

En la declaración necesitamos especificar

- Source: el fichero en el que se encuentra el shader.
- Entry_point: la función inicial a ejecutar o punto de entrada.
- Profiles: por último se especifican los perfiles mínimos requeridos para ejecutar el vertex y fragment shader, es decir, especificar las capacidades mínimas de las que la GPU debe disponer para ejecutar los shaders.
- Params: parámetros que se enviarán desde el motor de renderizado a los shaders.

No entraremos en detalle sobre la programación de shaders, ya que supone un amplio proceso de aprendizaje en sí mismo, sin embargo se pueden encontrar multitud de guías y tutoriales en la red, e incluso libros oficiales de aprendizaje gratuitos [16], [17].

7.3 EXTENDIENDO CAELUM-ENGINE

Uno de los puntos fuertes de Caelum-Engine es la flexibilidad que ofrece. Incluso si después de revisar el tutorial de iniciación, las características avanzadas y el API del motor hay alguna funcionalidad que no está aún disponible no hay problema. Caelum-Engine provee varios puntos para integrar nuevas características en el motor.

7.3.1 Capas y componentes de escena

La forma más importante de extensión de Caelum-Engine y la cual ha supuesto la base principal de su diseño es el sistema de capas y componentes.

Como ya se ha comentado las escenas de juego están compuestas de diferentes capas que se sincronizan a través de la escena y aportan diferentes tipos de contenido.

Los pasos para extender el sistema de capas de escena son los siguientes

1. El sistema principal de extensión debe ser capaz de crear capas “GameLayer” por lo tanto el sistema principal debe heredar de la clase “LayerManager” he implementar las funciones de gestión de capas.

2. El segundo paso es extender la interfaz “GameLayer” e implementar la capa de gestión específica del nuevo tipo de recursos.

3. El tercer paso consiste en crear los componentes específicos de la nueva capa de juego. Los componentes fijos “FixedComponent” no tienen mayor complicación, sin embargo, los componentes móviles “MovableComponent” deben implementar las funciones de actualización.

```
virtual void updatePosition(const Vector3& position);
virtual void updateOrientation(const Quaternion& orientation);
virtual void updateScale(const Vector3& scale);
```

   Estas funciones garantizan la sincronización entre los diferentes componentes añadidos al mismo objeto de juego.

7.3.2 Sistema de memoria

En ocasiones al implementar e integrar un nuevo sistema y capas para Caelum-Engine nos podemos encontrar en la necesidad de cargar nuevos tipos de recursos.

Las ventajas de utilizar el sistema de recursos integrado de Caelum-Engine son bastantes.

- La localización de los recursos en el sistema es responsabilidad de Caelum-Engine, el cual ya está preparado para localizar los ficheros a cargar en los diferentes sistemas operativos soportados.

- Se realiza una gestión eficiente de la memoria donde sólo se carga una vez en memoria cada recurso del cual se pueden generar múltiples instancias.

- Al utilizar el sistema de memoria del motor, la carga de recursos está completamente integrada y gestionada a través del propio fichero de configuración de recursos.

- Tolerancia a fallos, es decir, utilizar el sistema integrado de memoria implica que aunque los recursos no se liberen de memoria explícitamente, el propio sistema los liberará en su cierre, lo que ayuda a reducir los memory leaks.

Para utilizar el sistema de recursos de Caelum Engine se deben seguir los siguientes pasos:

1. En primer lugar debemos crear una clase que represente al recurso en cuestión. Esta clase debe implementar la interfaz Caelum::Resource localizado en "core/resource.h".

2. Para la gestión de diferentes instancias se utilizan punteros inteligentes, por lo tanto la definición del recurso debe ir acompaña de la extensión de un puntero inteligente para el recurso en la plantilla Caelum::SharedPtr<T> localizado en "core/resource.h".

3. Se debe crear un cargador del recurso en cuestión extendiendo la interfaz Caelum::ResourceLoader localizada en "core/resourceloader.h" y registrando posteriormente el cargador creado a través del gestor de recursos principal utilizando la función:

```
bool ResourceManager::registerResourceLoader(
                                const String& resLoaderName,
                                const String& resExtension,
                                ResourceLoader* resLoader);
```

7.3.3 Carga de plugins

El sistema de carga de plugins permite que los sistemas acoplados al motor carguen sus plugins a través del sistema integrado del motor.

Los pasos para que el nuevo sistema integre la carga de plugins en el motor es el siguiente.

1. El sistema en cuestión debe extender la interfaz PluginLoader, localizada en "core/pluginloader.h"

2. El sistema debe registrarse en el motor a través de la función

```
bool PluginManager::registerPluginLoader(
                                        const String& groupName,
                                          PluginLoader* loader);
```

3. Se debe modificar el archivo "config/plugins.cfg" introduciendo la linea [groupName] y a continuación la lista de plugins a cargar por el sistema. El nombre groupName debe coincidir con el nombre groupName en el registro del sistema.

7.4 HERRAMIENTAS EXTERNAS

Editores de modelos3D

Caelum-Engine propulsado por el motor de renderizado Ogre3D permite cargar mallas o modelos, esqueletos y animaciones realizados en de multitud de diferentes suites de diseño. Para ello existen exportadores de formato para los siguientes programas.

- 3Ds Max
- Maya
- SoftImage/XSI
- Blender
- Wings3D
- Cinema4D
- Lightwave
- AC3D
- Google Sketchup

Figura 7.1 - Imágen de etapas de diseño de un modelo 3D

Editor de terreno

El sistema de terreno de Caelum-Engine se basa en la utilización de mapas de alturas, imágenes en escala de grises que indican la altura de cada sección del terreno. Actualmente existen multitud de programas que permiten crear estos mapas de alturas. Sin embargo y aunque los editores de mapas de altura son suficientes para la generación de la malla del terreno son insuficientes para la distribución de texturas.

Para una edición completa de terreno está disponible “Ogitor”, el cual genera ficheros binarios sobre el terreno compatibles con Caelum-Engine.

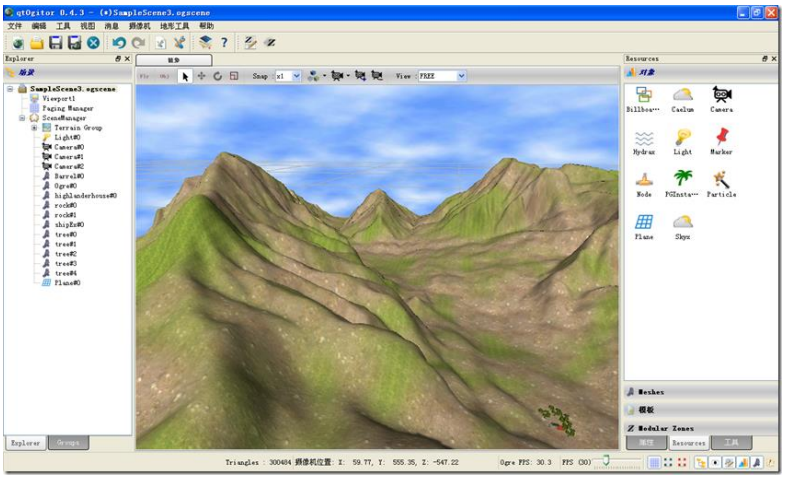


Figura 7.2 - Imagen la de edición de terreno con Ogitor

GUI Layout Editor

Para facilitar la creación de interfaces de usuario, viene incluido en la distribución un editor de interfaces de usuario provisto por la librería MyGUI y compatible con Caelum-Engine.

GUI Skin Editor

Para facilitar la creación y modificación de temas para interfaces de usuario, viene incluido en la distribución un editor de temas de interfaces de usuario provisto por la librería MyGUI y compatible con Caelum-Engine.

Hydrax Editor

El plugin de simulación de agua Hydrax viene provisto de un editor para visualizar las propiedades de configuración en tiempo real. Los ficheros de configuración de agua del editor son totalmente compatibles con Caelum-Engine.

8. CONCLUSIONES

En este último capítulo se pretende presentar un resumen de las experiencias y conclusiones recabadas tras realización del motor de videojuegos Caelum-Engine.

Para revisar y sintetizar todo lo expuesto en esta memoria se dividirá este capítulo en las siguientes secciones.

- **Resumen del proyecto**: Breve descripción del problema y las soluciones existentes así como la descripción del proyecto y la diferenciación que destaca a Caelum-Engine.
- **Incidencias**: Resumen de las dificultades e imprevistos encontrados en el proceso de desarrollo del proyecto y solución de las mismas.
- **Limitaciones del sistema**: Limitaciones actuales del proyecto con respecto a otras soluciones del mercado.
- **Líneas futuras**: Breve estudio de posibles vías de investigación y desarrollo futuro.
- **Consideraciones finales**: Conclusiones finales y análisis de la experiencia obtenida tras la investigación y realización del proyecto.

8.1 RESUMEN DEL PROYECTO

La industria del videojuego es un sector en alza y las grandes compañías ya no son las únicas beneficiadas. Al contrario, esta revolución está protagonizada por multitud de pequeños estudios independientes. Sin embargo, excepto alguna excepción como Unity3D muy popular entre este tipo de estudios, el mercado de los motores de videojuegos no ha sabido adaptarse realmente a las necesidades de los pequeños estudios. Se imponen licencias restrictivas y precios poco asequibles. Incluso en sus modelos de negocio adaptados a pequeños estudios se piden porcentajes de la recaudación del juego como pago, lo que obliga a establecer unos márgenes de beneficio en ocasiones demasiado grandes.

Esta situación nos lleva a preguntarnos hasta qué punto sale rentable utilizar estos motores actuales cuando el más barato ronda el desembolso de unos 5000$ dependiendo de las características que necesitemos.

Caelum-Engine

En este proyecto se propone un motor de videojuegos:

- **Económico**: Empezar a utilizarlo y a crear tu propio proyecto es completamente gratis. Con un modelo de negocio realmente adaptado a las necesidades de pequeños estudios (véase sección 1.5).

- **Práctico**: Familiarizarse con Caelum-Engine es realmente sencillo y empezar a utilizarlo es cuestión de minutos. Además de los tutoriales de iniciación y el manual de usuario (véase capítulo 7), el API es muy claro y sencillo.

- **Flexible**: Es fácilmente extensible, tanto mediante la integración de los elementos de memoria (sección 4.2.2), la gestión de plugins o el sistema de extensión de escenas por capas (sección 4.4.2).

8.2 INCIDENCIAS

Durante el desarrollo de Caelum-Engine han sucedido varios problemas e incidencias que han obligado a modificar en varias ocasiones el plan de trabajo con el objetivo de buscar soluciones y adaptación a los nuevos planteamientos.

Algunas de las incidencias más destacadas han sido las siguientes:

Retirada de drivers oficiales de tarjetas gráficas en Linux

Al comienzo del desarrollo del proyecto se empezó utilizando un portátil Toshiba (Dual Core 1,6Ghz y una gráfica de gama baja Ati HD 2100) comprado en 2007.

Por desgracia aunque las primeras pruebas del proyecto en Linux funcionaban correctamente, el soporte del driver oficial privativo para la gráfica HD 2100 en Linux fue retirado, lo que supuso una bajada de rendimiento 3D abismal. A pesar de que el motor pudo seguir ejecutando las pruebas, algunas bajaron su rendimiento de 60fps hasta 1-2fps, debido al precario soporte para shaders de los drivers libres disponibles.

Este nuevo escenario, planteó varios problemas para el testeo de las demos, por lo que el desarrollo de Caelum-Engine continuó esta vez en Windows en un ordenador fijo Asus (QuadCore 2,4Ghz con gráfica Ati HD 6870 gama media) hasta la llegada de un nuevo equipo portátil Lenovo (i5 DualCore 1,8Ghz con gráfica GT 625M gama baja).

Restructuraciones y cambios de versión

Cada cierto tiempo, las librerías suelen hacer una retrospectiva del proyecto y ciertas partes de API público son modificadas o añadidas para dar paso a una nueva versión mejorada o refactorizada del proyecto. Este es el caso de Ogre3D el cual, durante el transcurso de realización este proyecto, realizó su cambio de versión estable de la versión 1.7 a 1.8. En este cambio de versión, uno de los grandes cambios fue la completa reestructuración del sistema de paginación y el sistema de terreno, el cual mejoró enormemente pero aumentó su complejidad de la misma manera.

En primer lugar el cambio de API supuso que las pruebas y demos implementadas con el antiguo sistema dejasen de funcionar en la nueva versión. En segundo lugar, todo el conocimiento aprendido para realizar la misma tarea en la antigua versión supuso una gran pérdida de tiempo.

Finalmente a pesar de retrasar el calendario la adaptación al nuevo sistema resultó satisfactoria.

A parte del caso de Ogre3D, surgió este mismo problema con el plugin MyGUI, para el cual se conocía el funcionamiento de una antigua versión de antemano. En este caso resultó ser una incidencia menor, ya que no se llegaron a implementar pruebas ni demos utilizando la antigua versión de la librería, pero sí supuso la actualización del conocimiento implicado al uso de la nueva versión de la librería.

8.3 LIMITACIONES DEL SISTEMA

A pesar de los excelentes resultados obtenidos, un motor de videojuegos es un proyecto de una enorme envergadura. Lamentablemente y debido principalmente a las limitaciones temporales, de recursos humanos y quizás a una falta de experiencia en este sector, no se han podido alcanzar los estándares de calidad a los que la industria de los videojuegos AAA nos tiene acostumbrados.

Si bien es cierto que la utilización de tecnologías libres y bien establecidas ha permitido a Caelum-Engine una calidad más que aceptable para desarrollo de pequeños juegos, aún existen bastantes carencias en este proyecto de cara a una salida comercial exitosa.

Una de sus mayores limitaciones actuales es el uso de formatos específicos para la aplicación. El motor solo es capaz de utilizar modelos 3D en formato .mesh y .skeleton siendo necesaria la conversión previa desde las suites de diseño al propio formato del motor.

Otra de las limitaciones actuales del motor se encuentra en las mallas de la simulación física las cuales actualizan correctamente la posición del objeto de acuerdo a las normas físicas establecidas, pero sin embargo no están preparadas para deformarse a la par que las animaciones de los modelos 3D lo hacen. No se trata de una característica imprescindible, pero aporta mayor realismo y un mayor nivel de pulidez.

Una limitación más, derivada de la anterior, es la falta de un método de *raycasting* a nivel de modelo visual 3D. Esto es, no es posible localizar con precisión real los puntos de corte de un rayo simulado con las entidades 3D, únicamente con las mallas de colisión asociadas, las cuales no son animadas. Esta característica es enormemente usada en los juegos de acción en primera persona, los llamados FPS.

Sin embargo, ninguna de estas limitaciones es crítica y todas son susceptibles de ser subsanadas en futuras líneas de desarrollo.

8.4 LINEAS FUTURAS

Escaneado del terreno - Inteligencia artificial

Una funcionalidad principal apenas desarrollada en Caelum-Engine debido a las limitaciones temporales, es el desarrollo de un sistema de inteligencia artificial más elaborado. Las

funcionalidades actuales son realmente acotadas y dejan casi completamente de lado del programador de juegos la programación de la inteligencia artificial.

Se propone trabajar la generación automática de waypoints, es decir, cálculo de puntos del mapa y sus transiciones posibles que permita recrear rutas óptimas de movimiento. La implementación del cálculo de la ruta óptima no supone un mayor problema e incluso el usuario podría implementar su propia estrategia. Sin embargo, el escaneado y generación de puntos de ruta con los que trabajar es una tarea más complicada, ya que involucra comprobar la posibilidad de realizar ciertas transiciones para la inteligencia artificial.

Figura 8.1 – Generación de waypoints y comprobación de transiciones posibles

Físicas avanzadas

Como ya se ha comentado en las incidencias, las mallas de colisión se crean en tiempo de carga y no se modifican con las animaciones. Por ello la creación de mallas seccionadas por huesos es el siguiente paso a tomar, esto permitirá la animación de las mallas de colisión y también permitirá actuar a cada hueso como un cuerpo rígido independiente.

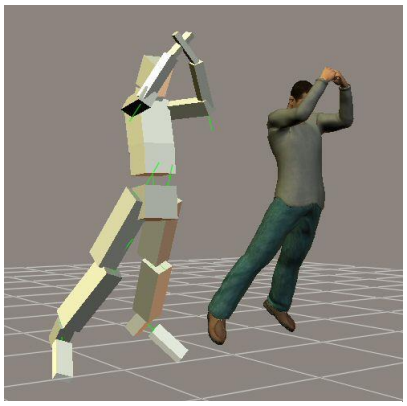


Figura 8.2 - Modelo 3D y malla de colisión del esqueleto

En Caelum-Engine nos hemos centrado por el momento en utilizar cuerpos rígidos, los más comunes en simulación. Sin embargo, nos parece una idea acertada añadir soporte para cuerpos blandos en próximas versiones. La utilización de cuerpos blandos permite ejecutar la simulación de ropa y otros materiales flexibles como la amortiguación de vehículos.

Scripting

Uno de los mayores puntos débiles y que en un principio formaban parte de la idea inicial de Caelum-Engine es la falta de integración de un lenguaje de scripting en el motor. La integración de un lenguaje de scripting y los bindings adecuados permite crear multitud de contenido para

un juego sin la necesidad de recompilar el juego completo. Además es uno de los elementos fundamentales para probar los modelos y los diseños del juego y acortar la distancia entre programadores y diseñadores.

Editor integrado en tiempo real

Otra de las grandes carencias de Caelum-Engine especialmente en comparación a las soluciones existentes, es la falta de un editor de juego que permita visualizar la experiencia de juego en tiempo real, los materiales, texturas y la escena de juego en general.

8.5 CONSIDERACIONES FINALES

La realización de un proyecto de fin de carrera supone no sólo un trabajo más, si no el mayor propósito de temática libre en el que nos embarcamos en la carrera. Por ello, se ha entendido este proyecto de fin de carrera como el ejemplo de lo que podría ser una carta de presentación.

Todo comenzó como un esbozo de investigación y aprendizaje personal sobre el mundo de los videojuegos. Pero debido al propio interés, lo que comenzó como curiosidad y una pequeña investigación, evolucionó en el proyecto que hoy es Caelum-Engine.

Conseguir esta meta no ha sido fácil en absoluto, y su realización ha requerido de cantidad de estudio, preparación y pruebas. Además la embarcación en una iniciativa de tal envergadura ha supuesto la dedicación de gran parte del tiempo libre personal.

Se espera que el propio producto obtenido sea provechoso y de ayuda a la comunidad de software libre y educativa. Llegados a este punto creemos que Caelum-Engine ha cumplido correctamente con los requisitos necesarios y objetivos propuestos. Además, la experiencia obtenida sobre el funcionamiento de los videojuegos ha sido realmente gratificante y esperemos sea de gran utilidad para el futuro profesional.

A. GLOSARIO

AAA (juegos): Literalmente implica juegos con un alto presupuesto, sin embargo se utiliza popularmente para referirse a juegos que son grandes superproducciones de éxito.

Android: Sistema Operativo concebido para smartphones y tablets basado en Linux y comprado por Google en 2005. Actualmente es uno de los SO para smartphones más extendidos del mundo.

API (Application Programming Interface): Conjunto de funciones y procedimientos que ofrece una librería como capa de abstracción para la creación de software.

Bloom: Efecto visual de post-procesado que realza el color de las zonas iluminadas dotándolas de cierto "halo" difuminado de luz para añadir realismo al objeto.

Buffer: Zona de memoria utilizada para almacenar datos temporalmente.

CG (Shading): No confundir con CG (Computer Graphics), CG es también un lenguaje de programación de shaders desarrollado por la compañía de tarjetas gráficas Nvidia y que funciona tanto en DirectX como en OpenGL.

Códec: Abreviatura de codificador-decodificador. Se trata de una especificación capaz de convertir y comprimir o descomprimir un flujo de datos para trabajar con audio y video por ejemplo.

Debugging: Proceso en el cual se testea el funcionamiento y la calidad del software durante la ejecución del sistema.

DirectX: Interfaz de programación de bajo nivel de gráficos 3D desarrollada por Microsoft y disponible para Windows y Xbox.

FOV (field o view): Campo de visión. Referente al ángulo de visión, generalmente vertical, de una cámara virtual.

FPS (juegos, First Person Shooter): Relativo a juegos de acción en primera persona, es decir visto desde los ojos del personaje. Comúnmente utilizado para nombrar el género de videojuegos de disparos.

FPS (rendimiento, Frames Per Second): Relativo al número de cuadros o imágenes por segundo que se reproducen.

Gamepad: Dispositivo de entrada. Controlador de juego utilizado con 2 manos, donde los dedos, especialmente los pulgares se utilizan para interactuar con un videojuego.

GLSL (OpenGL Shading Language): Lenguaje de programación de shaders de OpenGL.

GPU: Graphics procesing unit. Utilizado para designar a la unidad de procesado de las tarjetas gráficas siendo capaz de recibir y procesar shaders.

HLSL (*High Level Shader Language)*: Lenguaje de programación de shaders creado por Microsoft para DirectX.

Indie gaming: Término utilizado para referirse a juegos creados por estudios independientes y normalmente relativamente pequeños.

IOS: Sistema Operativo concebido para móviles creado por Apple y originalmente desarrollado para IPhone.

IDE (Integrated Development Environment): Entorno de programación consistente en editor de código, ayudas de programación y que en la mayoría de ocasiones integra una cadena de compilación, y debugger.

Joystick: Dispositivo de control de control de dos o tres ejes que se usa desde un ordenador o consola.

Keyframe: Es la información que describe, la situación clave de una animación en ciertos momentos dados, entre los cuales se interpola esta información para conseguir transiciones suaves.

LOD (Level Of Detail): Literalmente nivel de detalle. Se utiliza para referirse a la técnica utilizada para renderizar.

Logs: Registro de datos e información de estado un sistema.

Loop: Referido a un juego, el ciclo de acciones que se ejecutan en sucesión constantemente.

Makefile: Fichero de especificación y configuración para compilación.

Memory leak: Fenómeno conocido como fuga de memoria. Error por el cual un proceso, se olvida de liberar un bloque de memoria reservada, quedando ésta indefinidamente inusable.

Mipmap: Referente a texturas, son el conjunto de imágenes de la misma textura y diferentes resoluciones utilizados para mostrar en diferentes niveles de detalle y distancia, acelerando el renderizado

Open-source: Filosofía que promueve el libre acceso, uso, copia, estudio, modificación y distribución del software.

OpenGL: Interfaz de programación de gráficos 3D de bajo nivel desarrollada por Silicon Graphics. Se trata de una librería de código abierto, disponible para Window, Linux, Mac, Unix y PlayStation 3.

Plug-ins: Complemento que se acopla a una librería y que se comunica con ésta a través de un API común entre las dos, añadiendo funcionalidad.

Ragdoll: Se trata de una colección de cuerpos rígidos que representan los huesos de una entidad 3D y que sirven para realizar simulaciones en sistemas articulados.

Raycasting: Utilización de la intersección de una superficie en forma de rayo/recta con otros elementos a modo de test para resolver varios problemas.

Render / Renderizado: Es el proceso de generación de una imagen a partir de uno o varios modelos o mallas.

Script: Programa simple utilizado como un archivo de procesamiento por lotes que se almacena generalmente en texto plano y se interpreta en tiempo de ejecución.

SCV / CVS: Sistema de control de versiones de software.

Shader: Programa de sombreado ejecutado en la GPU utilizado para reflejar los niveles de luz y colores adecuados o para producir efectos especiales o de post-procesado.

Shading: Técnica utilizada para representar la sensación de profundidad en un modelo 3D o ilustración utilizando diferentes niveles de oscuridad y sombreado.

SSAO (Scree Space Ambient Occlusion): Es una técnica de renderizado desarrollada por Crytek en 2007 utilizada para aproximar eficientemente la oclusión ambiental por ordenador.

TCP (Transmission Control Protocol): Uno de los protocolos fundamentales de internet. Provee una transmisión de paquetes ordenada, confiable y con cierta tolerancia de fallos.

Texture Splatting: Es un método utilizado para combinar diferentes texturas a través de uno o varios mapas de transparencias, donde las trasparencia indican que la visibilidad de una textura de una capa inferior.

UDP (User Datagram Protocol): Uno de los protocolos fundamentales de internet. Tiene un mecanismo de protocolo mínimo, pero ofrece transmisión no confiable, donde no hay garantía de ordenación, duplicados o recepción.

B. LICENCIAS

Licencia libre de Caelum-Engine (GPL)

```
Caelum-Engine
Copyright (C) 2012-1013  David García Miguel <noxwings@gmail.com>

This program is free software: you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation, either version 3 of the License, or (at your
option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
Public License for more details.

You should have received a copy of the GNU General Public License along
with this program.
If not, see http://www. gnu.org/licenses/gpl.txt.
```

Licencias utilizadas por las dependencias de Caelum-Engine

Además de la propia licencia de Caelum-Engine en este anexo se muestra la versión resumida de las diferentes licencias de las dependencias mencionadas a lo largo de la memoria del proyecto.

- Licencia MIT

```
Copyright (C) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to permit
persons to whom the Software is furnished to do so, subject to the
following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

- Licencia LGPL

```
Copyright (C) <year> <copyright holders>

This program is free software; you can redistribute it and/or modify it
under the terms of the GNU Lesser General Public License as published
by the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser
General Public License for more details.

You should have received a copy of the GNU Lesser General Public License
along with this program; if not, write to the Free Software Foundation,
Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, or go to
http://www.gnu.org/copyleft/lesser.txt
```

- Licencia BSD (3-clause “New BSD License”)

```
Copyright (c) <year>, <copyright holder>
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:
    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.
    * Neither the name of the <organization> nor the names of its
      contributors may be used to endorse or promote products derived
      from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER>
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
THE POSSIBILITY OF SUCH DAMAGE.
```

- Licencia Zlib/libpng

```
Copyright (c) <year> <copyright holders>

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
claim that you wrote the original software. If you use this software in
a product, an acknowledgment in the product documentation would be
appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not
be misrepresented as being the original software.

3. This notice may not be removed or altered from any source
distribution.
```

- Licencia Boost

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or
organization obtaining a copy of the software and accompanying
documentation covered by this license (the "Software") to use, reproduce,
display, distribute, execute, and transmit the Software, and to prepare
derivative works of the Software, and to permit third-parties to whom
the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement,
including the above license grant, this restriction and the following
disclaimer, must be included in all copies of the Software, in whole or
in part, and all derivative works of the Software, unless such copies
or derivative works are solely in the form of machine-executable object
code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-
INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE
DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY,
WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

BIBLIOGRAFÍA

[1] J. Tyberghein, «Crystal Space 3D,» 2011. [En línea]. Available: http://www.crystalspace3d.org/. [Último acceso: 21 Agosto 2013].

[2] N. Gebhardt, «Irrlicht Engine - A free Open Source 3D Engine,» 2010. [En línea]. Available: http://irrlicht.sourceforge.net/. [Último acceso: 21 Agosto 2013].

[3] T. Higgins, «Unity-3D Game Engine,» 2010. [En línea]. Available: http://www.unity3D.com. [Último acceso: 21 Agosto 2013].

[4] Crytek, «Cryengine,» 2012. [En línea]. Available: http://mycryengine.com/. [Último acceso: 21 Agosto 2013].

[5] Epic Games, «Unreal Engine,» 2012. [En línea]. Available: http://www.unrealengine.com/. [Último acceso: 21 Agosto 2013].

[6] D. Vallejo Fernández y C. Martín Angelina, Desarrollo de Videojuegos: Arquitectura del Motor, Triangle, 2012.

[7] . P. Schneider y D. H. Eberly, Geometric Tools for Computer Graphics, Morgan Kaufmann, 2002.

[8] C. González Morcillo, J. A. Albusac Jiménez, S. Pérez Camacho, J. López Gonzalez y C. Mora Castro, Desarrollo de videojuegos: Programación gráfica, Triangle, 2012.

[9] Y. W. Bernier, «Latency compensating methods in client/server in-game protocol design and optimization,» de *Game Developers Conference*, 2001.

[10] T. Sweeney, «Unreal networking architecture,» 2001. [En línea]. Available: http://udn.epicgames.com/Three/NetworkingOverview.html. [Último acceso: 21 Agosto 2013].

[11] Microsoft, «Using dllimport and dllexport in C++ Classes,» [En línea]. Available: http://msdn.microsoft.com/en-us/library/81h27t8c(v=vs.80).aspx. [Último acceso: 21 Agosto 2013].

[12] A. Blekhman, «HowTo: Export C++ classes from a DLL,» [En línea]. Available: http://www.codeproject.com/Articles/28969/HowTo-Export-C-classes-from-a-DLL. [Último acceso: 21 Agosto 2013].

[13] K. Channa, «Light mapping-theory and implementation,» 2003. [En línea]. Available: http://www.flipcode.com/archives/Light_Mapping_Theory_and_Implementation.shtml. [Último acceso: 21 Agosto 2013].

[14] G. Junker, Pro OGRE 3D programming, Apress, 2006.

[15] S. Streeting, «OGRE Manual v1.8 - Material Scripts,» 20 Agosto 2012. [En línea]. Available: http://www.ogre3d.org/docs/manual/manual_14.html#Material-Scripts. [Último acceso: 21 Agosto 2013].

[16] R. Fernando y M. J. Kilgard, The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[17] R. Fernando y M. J. Kilgard, «The CG Tutorial (Free Web Book),» [En línea]. Available: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html. [Último acceso: 21 Agosto 2013].