ESCUELA SUPERIOR DE INGENIERÍA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

SIMULADOR DE JUEGO DE LUCHA 1vs1

Antonio Jaime Rodríguez Medina

8 de junio de 2011

ESCUELA SUPERIOR DE INGENIERÍA

INGENIERO TÉCNICO EN INFORMÁTICA DE GESTIÓN

SIMULADOR DE JUEGO DE LUCHA 1vs1

- Departamento: Lenguajes y Sistemas informáticos
- Directores del proyecto: Manuel Palomo Duarte y Antonio García Domínguez
- Autor del proyecto: Antonio Jaime Rodríguez Medina

Cádiz, 8 de junio de 2011

Fdo: Antonio Jaime Rodríguez Medina

Licencia

Notación y formato

Los **comandos** siguen este formato:: `ls -lh /etc`

Las **rutas de directorios o ficheros** se escriben en cursiva: */home/jaime/xfreefighter/readme.txt*

Los **nombres de las clases** van en mayúsculas y en negrita por ejemplo: La clase **Fighter**

Índice general

Índice de figuras

Índice de listados

Capítulo 1

Introducción

A modo de breve introducción el principal objetivo de este proyecto es crear un simulador de juego de lucha 1vs1. Esto consistirá en un videojuego en donde el usuario puede ampliar y personalizar el mismo a su gusto siguiendo una serie de reglas.

Aunque el principal objetivo es crear un combate entre dos personajes, los cuales al menos uno es manejado por un humano, también lo es que el simulador sea extensible y reutilizable incluso por usuarios con escasos conocimientos de programación. Por estos motivos se han utilizado una serie de tecnologías, algunas usadas ampliamente en la industria del videojuego, que han hecho posible que el sistema fuera ampliable y reutilizable.

1.1. Sobre este documento

A continuación se explica cómo se ha organizado este documento:

- En el capítulo 1 se explica el principal objetivo del proyecto y la organización de este documento.
- En el capítulo 2 se explica una visión general sobre el proyecto. El concepto de juego de lucha 1vs1, la historia de los videojuegos de lucha, los acrónimos y definiciones más usuales que se utilizaran, los objetivos a alcanzar, así como una descripción de los requisitos de este proyecto.
- En el capítulo 3 se explica la organización temporal del sistema para realizar este proyecto, mostrando los cambios generales y los resultados obtenidos de cada fase del proyecto.
- En el capítulo 4 se explica el análisis para este proyecto, Donde se modelan los requisitos funcionales del sistema y se da una primera aproximación al resultado final, explicando los casos de uso y el desarrollo del análisis.
- En el capítulo 5 se explica el diseño del sistema y su arquitectura definiendo un modelo estructurado en tres capas: Presentación, dominio y datos.
- En el capítulo 6 se explica la implementación del sistema, donde se pone de manifiesto todas las dificultades de desarrollar un simulador de estas características.

- En el capítulo 7 se explica la fase de pruebas, en donde se describen las pruebas y su utilidad en el simulador, para por ejemplo cargar personajes nuevos.
- En el capítulo 8 se muestran las conclusiones del proyecto, sus posibles ampliaciones en versiones futuras y las mejoras que deberían implementarse.
- Se incluyen apéndices adicionales: Manual de la máquina de estados (apéndice A), manual de usuario (apéndice B), manual para la maquetación de animaciones (apéndice C).

Capítulo 2

Visión general

2.1. Videojuegos de lucha

A continuación se explica el concepto y las características de un juego de lucha. Para explicar este concepto se ha utilizado como fuente principal Wikipedia, en el siguiente enlace: `http://en.wikipedia.org/wiki/Fighting_game`, se puede ver ampliada la información que se proporciona sobre este concepto, la clasificación del género y la historia del mismo.

Definición

> «Un juego o videojuego de lucha es un género de videojuegos donde el jugador controla un personaje en la pantalla y se involucra en combate cuerpo a cuerpo con un oponente. Estos personajes suelen ser de igual potencia (o al menos deberían de estar equilibrados) y la lucha esta compuesta por varios asaltos, que tienen lugar en un escenario. Los jugadores deberán dominar las técnicas como el bloqueo, contraataque, y encadenar secuencias de ataques que se conocen con el nombre de combo.»

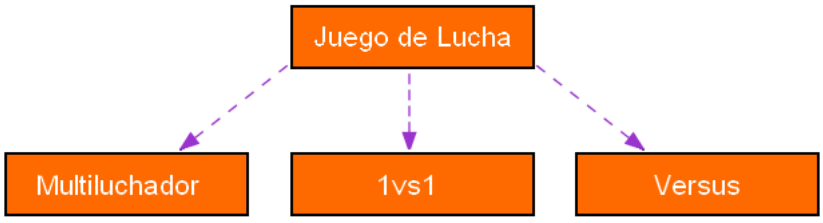


Figura 2.1: Subgéneros de juegos de lucha

Clasificación de los juegos de lucha

El género de los juegos o videojuegos de lucha, dependiendo de la fuente consultada se puede clasificar de varias maneras. Se ha seguido la clasificación basándose en el número de luchadores que forman un combate, dividendo este género en 3 subgéneros:

1. **1vs1:** Un combate lo componen dos luchadores, controlado uno al menos por un humano.
2. **Multiluchador:** Un combate lo componen más de dos luchadores, controlado uno al menos por un humano.
3. **Versus:** Un combate lo componen dos luchadores, controlado uno al menos por un humano, siendo capaz este de cambiar el luchador dentro del mismo combate, de manera que puede manejar varios luchadores.

El simulador se enmarca dentro de los videojuegos de lucha en el subgénero 1vs1 y en un entorno 2 dimensiones.

Requisitos de un juego de lucha

A continuación se explica los requisitos para que un juego o videojuego sea de lucha, diferenciando entre los juegos de deportes como el boxeo, lucha libre, etc.

> «Los videojuegos de lucha varían mucho de un juego a otro, pero por lo general consiste en pelear contra otro luchador, manejado por la máquina o por otro videojugador, cuya finalidad es derrotarlo o evitar que derroten al nuestro, dándole golpes o cualquier otro tipo de ataque para debilitarlo y vencerlo, así como esquivar y contraatacar cualquier ataque recibido.»

Los requisitos de los juego de lucha 1vs1 consiste son los siguientes:

- Los luchadores deben estar equilibrados, es decir, que ningún luchador en el simulador sea demasiado superior a los demás.
- Los movimientos de artes marciales son muy exagerados.
- Limitaciones en el espacio (la pantalla).
- Los ataques del luchador son: puñetazos, patadas, llaves y lanzamiento de proyectiles.
- Ataques y defensas muy relacionados existiendo contraataques para ciertos golpes pero estos son aprendidos por el jugador, mediante ensayo y error.
- Inclusión de ataques especiales o movimientos secretos que se consiguen mediante la pulsación de combinaciones de teclas terminadas en un puñetazo o patada.
- El objetivo del juego es desarrollar un combate, estos están formados de varios asaltos, el jugador que gane la mayoría gana el combate.

Figura 2.2: Videojuego Karate Champ de 1984

- Durante el combate el luchador tiene una barra de vida, que se agota cuando recibe un ataque, si esto ocurre pierde el asalto.
- Algunos juegos suelen incluir una campaña para un solo jugador o un torneo, donde el jugador debe derrotar a una serie de oponentes.

Historia

La historia de los juegos de lucha comienza en la década de los 80 en los que se podían confundir con los de deportes o lucha libre. Durante los años 90 donde el género se convierte en un fenómeno de masas dentro de las máquinas recreativas, afianzando el género, hasta tal punto que hoy en día se siguen produciendo títulos de este tipo de juegos.

Años 70/80

Empezó con juegos de boxeo (considerados como de deportes) hasta evolucionar a juegos de lucha 1vs1, durante todo este periodo los videojuegos eran en entornos 2D, Otro género como los juegos lucha libre no es considerado juego de lucha, sino también de deportes.

- 1984 **Karate Champ** de Technos Japan, nace el primer juego considerado del género.
- 1985 **Yie Ar Kung Fu** de la compañia Konami, influenciado por **Karate Champ**, introduce jugadores con aspecto y estilos de lucha únicos.

- 1985 **The Way of the Exploding Fist** de Bean Software, muy influenciado por **Karate Champ**.
- 1987 **Street Fighter** de Capcom, nace basado en **Yie Ar Kung Fu** y **Karate Champ**. Aunque no tuvo mucho éxito, fue el primer juego que introdujo movimientos secretos mediante la pulsación de combinaciones de teclas.
- 1989 **Budokan: The Martial Spirit** de Electronic Arts y fue el primer juego de consola 1vs1 para Sega Génesis.

Principios de los 90

Nace **Street Fighter II** considerado como la revolución del género por el equipo de Yoshiki Okamoto. Se desarrolló el mando de control más exacto, esto permitía a los jugadores ejecutar de una forma más fiable los movimientos especiales con varios botones, que había requerido previamente un elemento de suerte. Hicieron saltar al género a la consola y se convirtió en una plantilla para todo juego de lucha.

En este tiempo se consolida el género y aparecen las grandes sagas de los juegos de lucha: **Fatal Fury**, **Art of Fighting**, **Samurai Shodown**, **Mortal Kombat**. Así como la consolidación en el género de las compañías que los desarrollaron, que siguen publicando videojuegos de lucha en la actualidad para videoconsolas y ordenador: SNK, Midway Games, Capcom y SEGA.

Hay que destacar también que los personajes de **Street Fighter** han sido protagonistas de series y películas de animación, dos producciones cinematográficas. Unos de sus personajes principales llamado Ryu es unos de los personajes animados más famosos del mundo. Toda esta información se puede ver ampliada en la siguiente url: `http://en.wikipedia.org/wiki/Street_Fighter_%28series%29`.

También hay que destacar la serie de videojuegos **Mortal Kombat**, la cual también ha sacado series de televisión y varias películas cinematográficas y de animación. También hay que destacar que la banda sonora de los videojuegos de **Street Fighter** es muy famosa entre los aficionados de género llegándose a sacar incluso discos con las pistas del juego.

- 1991 **Fatal Fury** de la compañía SNK.
- 1991 **Street Fighter II** de Capcom.
- 1992 **Art of Fighting** de SNK para Neo Geo.
- 1992 **World Heroes** de SNK. Introduce elementos hostiles en el escenario.
- 1993 **Dark Edge** de Sega, el primer juego de lucha en 3D desechado por el público y nunca se exportó fuera de Japón.
- 1993 **Samurai Shodown** de SNK para Neo Geo.
- 1993 **Eternal Champions** de Sega.
- 1993 **Mortal Kombat** de Midway. Se caracteriza por su violencia explícita, dejando la puerta abierta para la crítica a los juegos de este tipo.

Figura 2.3: Imagen del videojuego Street Fighter II de Capcom

- 1993 **Virtua Fighter** de Sega. Es un juego de lucha 3D con gráficos poligonales.
- 1994 **The King of Fighters '94** de SNK.
- 1995 **Tekken** de Namco.

Finales de los 90

Se caracteriza por una serie de videojuegos que normalmente son continuaciones de otros y publicados mayoritariamente por la compañía SNK. Además del nacimiento de los juegos de lucha conocidos como **versus**. Los títulos que se publicaron y más fama tuvieron fueron juegos que incluían a los luchadores de **Street Fighter** contra otros de diversa índole como Marvel, Nintendo, etc.

- 1996 **Street Fighter EX** de Capcom. Revisión del Street Fighter II con gráficos 3D con un estilo de lucha igual que un entorno 2D.
- 1997 **The Last Blade** de SNK.
- 1997 **Street Fighter III** de Capcom.
- 1998 **The King of Fighters '98** de SNK.
- 1998 Nacimientos los versus de Capcom para Street Fighter: **Xmen vs Street Fighter**, **Maverl vs Capcom** que sorprendieron y gustaron al gran público.

Figura 2.4: Portada del videojuego Marvel Super Heroes vs Street Fighter

Figura 2.5: Videojuego Street Fighter IV

Año 2000 hasta actualidad

La primera parte de la década vio el surgimiento de los principales torneos internacionales de este género entre jugadores profesionales. En esta época los juegos se publican para las videoconsolas y el resurgir de los juegos de este género.

En especial nace **Street Fighter IV** considerado como unos de los mejores juegos de lucha de toda la historia, mezclando el estilo de lucha de los videojuegos de los 90 y un entorno de 2D con texturas en 3D, marcando una nueva era en los juegos de lucha.

Siguiendo el mismo estilo que **Street Fighter IV**, han aparecido dos títulos en el año 2011: **Marvel vs Capcom 3: Fate of Two Worlds**, juego tipo versus de la mítica serie de videojuegos que enfrenta a los súper héroes de Marvel contra los luchadores de **Street Fighter**, y **Mortal Kombat** que reaparece distribuido por la compañía Time Warner.

- 2001 **Capcom vs SNK: Millennium Fight** de Capcom.
- 2002 **Melty Blood** de TYPE-MOON.

- 2005 **Super Smash Bros Brawl** de Nintendo, personajes clásicos de esta compañía luchando entre ellos.
- 2008 **Mortal Kombat vs DC Universe** de Midway.
- 2008 **Street Fighter IV** por Capcom. Gran revolución de este género y considerado como el mejor de los juegos junto con Street Fighter II, mezclando el 3D y siendo de estilo 2D.
- Febrero de 2011, **Marvel vs Capcom 3: Fate of Two Worlds** de Capcom.
- Abril de 2011, **Mortal Kombat 2011** de Time Warner.

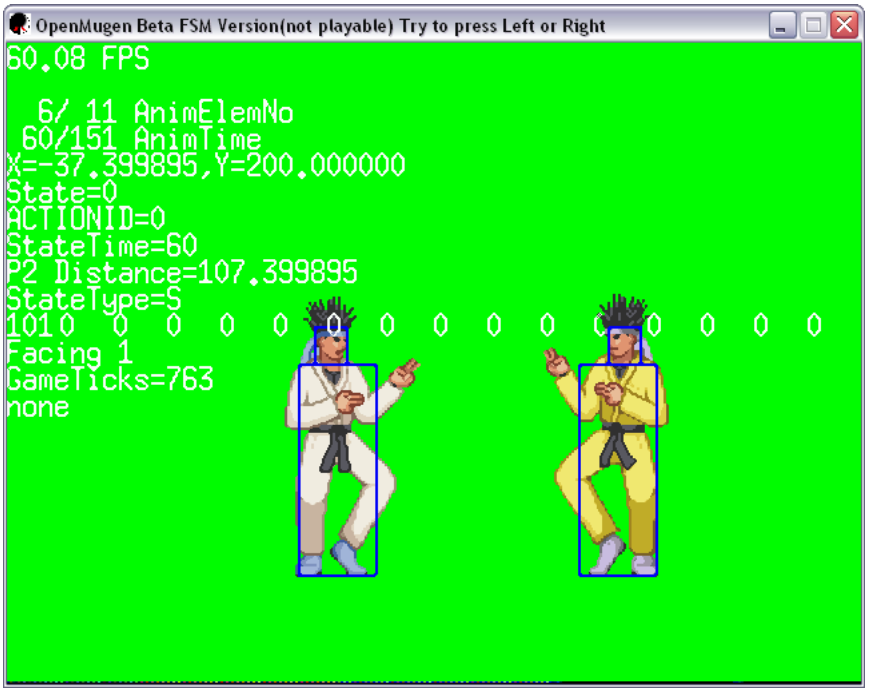


Figura 2.6: OpenMugen en ejecución

Estado del arte

Este proyecto no ha ignorado el principal motor de juegos de lucha llamado Mugen creado por la empresa Electbyte y nacido para realizar juegos de lucha al estilo Marvel vs Capcom, orientado para la plataforma MS Windows. Aunque el uso personal es gratis, no está permitido desarrollar versiones comerciales. Usa estándares propios y no está disponible su código fuente.

OpenMugen es una revisión Open Source y solo operativa para sistemas operativos MS Windows y su desarrollo esta abandonado desde 2004. Escrito en C/C++, Este proyecto usa el mismo formato de ficheros binarios que Mugen posiblemente obtenidos por ingeniera inversa. Electbyte no ha denunciado ni ha puesto impedimentos en contra de este proyecto. Sus autores no especifican nada sobre su licencia, con lo cual impide a la comunidad de software libre retomar el proyecto.

Mugen ha servido como un marco de referencia a este proyecto, recogiendo ideas como el tratamiento de las animaciones y colisiones, rechazando seguir sus formatos por su alta obsolescencia en el tratamiento de las animaciones, al querer mantener compatibilidad con versiones anteriores desarrolladas bajo el sistema operativo MS-DOS.

Nombre del producto

El nombre que se le ha puesto al proyecto es: Simulador de lucha 1vs1. Tendrá un acrónimo que será XFF (X de Unix, F de Free y F de Fighter).

Aplicaciones del software

El principal uso en el que se aplica este simulador es el de diseñar y ejecutar un combate entre dos luchadores siguiendo el estilo de los juegos de lucha de los años 90, principalmente inspirado en **Street Fighter II**. Los usuarios podrán diseñar luchadores y escenarios siguiendo así el mismo estilo que Mugen.

Características del usuario

Este simulador puede ser usado por una tres tipos de usuarios. A continuación se describe el papel u objetivo de cada uno de ellos:

- **Jugador:** Su objetivo es jugar al simulador.
- **Desarrollador:** Su objetivo es ampliar el juego, para ello existen manuales para añadir nuevos luchadores y escenarios.
- **Programador:** Su objetivo es modificar el código fuente de la aplicación para añadir o mejorar cualquier aspecto relacionado con el funcionamiento interno del simulador. Para ello a este usuario se le proporciona las siguientes ayudas:
  - Código fuente en C++ siguiendo una arquitectura concreta y orientado a objetos.
  - Documentación del código fuente. Esto hace que sea rápido y clarificador cuando se quiera buscar que hace una función o para que sirve determinada clase. Además de proveer de diagramas de colaboración para ver las relaciones entre los objetos o diagramas de herencia para ver sus generalizaciones y especializaciones.
  - Licencia GPL v3. Esto permite total libertad para estudiar, ampliar, modificar y distribuir el código fuente, haciendo este proyecto público y de amplia difusión.
  - Sistema de pruebas. El simulador tiene muchas partes y aspectos, que pueden ir desde cargar una imagen, una pista de audio, una animación o comprobar que un luchador lance bien una magia. Para ello se han creado pruebas en forma de comandos, para que se puedan testear independientemente propiedades de un luchador, imagen, pista de audio, etc.

2.2. El sistema de código embebido

Lo más complicado de desarrollar un juego de lucha y hacerlo ampliable, es el tener que desarrollar un sistema en el que todo luchador tiene características diferentes. Si sumamos que un usuario sea el que desarrolle estas características hace que este proyecto se salga del ámbito de desarrollar un videojuego de lucha. El simulador debe proporcionar un entorno donde se desarrolla una serie de personajes diferentes, siguiendo una serie de reglas comunes para la correcta integración con el sistema.

Mugen, para solventar este problema desarrolló una serie de ficheros, encargados de manipular el estado de un luchador y su física, además de añadir un sistema de expresiones lógicas y operadores matemáticos.

En este simulador no se ha pretendido volver a inventar la rueda, o lo que es peor volver a inventar la rueda cuadrada. Para ello se ha usado un sistema de código embebido, el cual hace posible que un lenguaje de programación externo se comunique con el sistema desarrollado en C++, modificando el estado de objetos que se encuentran en ejecución o creando objetos nuevos desde otro lenguaje de programación.

El lenguaje que se ha utilizado para el sistema de código embebido es Lua, ampliamente usado en la industria del videojuego. El intérprete estará embebido dentro de la aplicación y conectado con ésta, de manera que se pueden crear y modificar objetos C++ pasados como argumentos en funciones, crear objetos o devolver objetos. Todo mediante un script Lua, escrito por un usuario que no tiene por qué saber cómo está programado internamente el simulador pero debe de poseer una noción muy básica de programación en Lua. Mugen indirectamente también obliga a sus usuarios conocimientos de programación al incluir expresiones lógicas en sus formatos propietarios.

Ventajas de utilizar un sistema de código embebido

La fuente que he utilizado para enumerar las ventajas ha sido el libro "Programming Game AI by Example" de Mat Buckland, que distingue las siguientes:

1. Un rápido y fácil camino para leer variables y datos del juego sin necesidad de ficheros de inicialización.

2. Ahorrar tiempo e incrementar la productividad. Un videojuego profesional tiene un código fuente a veces muy grande, esto hace que tarde mucho tiempo en compilarse, usando un lenguaje embebido podemos cambiar características o incluso ampliar el juego sin necesidad de volver a compilar.

3. Aumenta la creatividad. Un lenguaje de alto nivel es mucho más operativo que C/C++ y su sintaxis es mucho más intuitiva para las personas que no se dedican a la programación. Esto permite que otras personas involucradas en el desarrollo (diseñadores, maquetadores y productores) puedan modificar aspectos del juego sin necesidad de conocimientos avanzados.

4. Un sistema extensible. Hace posible que la interfaz del juego, así como los modos de juegos, sean modificables o sean cambiados fácilmente por los desarrolladores. Esta ventaja o característica es muy popular en el mundo de los videojuegos, utilizando el término de 'mod' para referirse a la palabra modificación. Un mod se utiliza para que los usuarios avanzados del juego sean capaces de modificar parcial o totalmente un videojuego.

5. Inteligencia artificial en sistemas no críticos. En el mundo de los videojuegos es común que se use un lenguaje embebido para la inteligencia artificial, permitiendo usar lenguajes de alto nivel en lugar de C/C++.

El lenguaje de programación Lua

Lua es un lenguaje de programación que soporta muchos paradigmas diferentes: Imperativo, funcional, orientado a objetos y basado en prototipos. A continuación expongo una serie de

Figura 2.7: Intefaz de World of Warcraft, modificable mediante Lua

aplicaciones que usa este lenguaje como sistema embebido y cuya fuente es: `http://en.wikipedia.org/wiki/Lua_(programming_language)`.

Aplicaciones en videojuegos:

- **World of Warcraft:** Donde el usuario tiene la posibilidad de personalizar casi completamente la interfaz.
- **Supreme Commander:** El cual es modificable casi totalmente por el usuario.
- **Tibia:** Modificable casi totalmente (poderes, mapas, etc) junto con XML.
- **S.T.A.L.K.E.R. - Shadow Of Chernobyl:** Permitiendo al jugador modificar armas, armaduras y aspectos varios del juego.
- **Grim Fandango y La Fuga de Monkey Island:** Cuarta entrega de la saga Monkey Island, utiliza internamente scripts en Lua para definir la historia y los eventos que ocurren durante la partida.
- **Worms 4:** Mayhem utiliza Lua y XML para definir las misiones y desafíos.
- **Videoconsola portátil PSP:** Mediante un programa permite ejecutar archivos Lua en la conocida consola portátil de Sony.
- **Ragnarok Online:** Usa Lua para programar la inteligencia artificial.

Listado 2.1: Ejemplo del uso de Lua por el software WireShark

```
-- register http to handle ports 4888-4891
do
        local tcp_port_table = DissectorTable.get("tcp.port")
        local http_dissector = tcp_port_table:get_dissector(80)
        for i,port in ipairs{4888,4889,4890,4891} do
                tcp_port_table:add(port,http_dissector)
        end
end
```

- **Regnum Online:** Usa Lua para la mayoría de scripts del juego como interfaz, modo de juego, acciones, etc.
- **TASpring:** un juego de estrategia en tiempo real, usa Lua para la mayoría de scripts del juego como interfaz, modo de juego, acciones, etc.
- **Multi Theft Auto San Andreas:** Usa Lua para diseñar modos de juego y mapas.
- **Blitzkrieg:** Se usa el lenguaje Lua en los editores de mapas.
- **Counter-Strike 2D:** Permite utilizar scripts Lua para crear, por ejemplo, modos de juego completamente nuevos.
- **StepMania:** Se usa el lenguaje Lua para desarrollar la implementación de animaciones del entorno gráfico, y asimismo la ejecución de comandos internos relacionados con la jugabilidad.
- **Wolfenstein: Enemy Territory:** Algunos modos de este juego usa el lenguaje Lua para ejecutar mini-modos, scripts de administración, modificación de mapas, etc.

Otras aplicaciones que usan Lua son: **VLC Media Player**, **Apache HTTP Server**, **Adobe Photoshop Lightroom WireShark** entre otros.

2.3. Acrónimos

CD Compact Disc

2D 2 dimensiones

3D 3 dimensiones

FPS Frames Per Second

FSM Finite State Machine

GNU GNU is Not Unix

GPL General Public License

GUI Graphical User Interface

SDL Simple DirectMedia Layer

SF2 Street Fighter II

1vs1 Uno contra uno

TTF True Type Font

STL Standar Template Library

PNG Portable Network Graphics

URL Uniform Resource Locator

2.4. Definiciones

- **Alpha:** Cuarta capa de una imagen que hace referencia al grado de transparencia.
- **Biblioteca de código.** Es un conjunto de subrutinas, clases y datos que están compilados de manera que, de forma estática o dinámica, pueden ser añadidos en tiempo de compilación o ejecución a un ejecutable.
- **Binding:** Es una adaptación de una biblioteca para ser usada en un lenguaje de programación distinto de aquél en el que ha sido escrita.
- **Búfer:** Es una zona de memoria dedicada para la transferencia de información.
- **Callback:** Es una función que recibe como argumento la dirección o puntero de otra función. Cuando es llamado recurre al puntero de la función y la ejecuta.
- **CLISP:** Herramienta para implementar sistemas expertos.
- **Combo:** Combinación de golpes que un luchador propina a otro sin que éste se pueda defienda.
- **Diccionario:** Es una estructura de datos que asocia llaves o claves con valores, llamado también hash o tabla hash.
- **Diseñador:** Se refiere al oficio de diseñador gráfico.
- **Engine:** Es una biblioteca de programación que permite a los programadores de un videojuego desarrollar una solución contando con un nivel superior de abstracción.
- **Framework:** Es una estructura conceptual y tecnológica de soporte definida, normalmente con artefactos o módulos de software concretos, con base en la cual otro proyecto de software puede ser organizado y desarrollado.

- **Frame:** Es una imagen particular dentro de una sucesión de imágenes que componen una animación.
- **GNU/Linux:** Familia de sistemas operativos, basados en la combinación del núcleo del sistema Linux (núcleo) y en herramientas distribuidas por el proyecto GNU.
- **Hardware:** Partes tangibles de un ordenador, como la tarjeta de sonido, la lectora de CD, etc.
- **Joystick:** Es un dispositivo que cuenta con 2 ejes y varios botones, usado generalmente para jugar a juegos de ordenador o videojuegos. Palabra cuyo origen es el inglés, y el equivalente en español es mando.
- **libQT:** Biblioteca multiplaforma cuyo objetivo principal es implementar interfaces gráficas de usuario.
- **libSDL:** Conjuntos de bibliotecas multiplaforma usadas para visualizar imágenes, escuchar sonidos, etc.
- **Log:** Es un registro de eventos durante un rango de tiempo en particular.
- **MS Windows:** Se refiere a la familia de sistemas operativos Microsoft Windows.
- **Maquetador:** Se refiere a la persona que prepara imágenes para adaptarlas a un sistema en concreto. A diferencia del diseñador gráfico, el maquetador no dibuja ni diseña nada.
- **MS-DOS:** Familia de sistemas operativos pertenecientes a Microsoft y cuyas siglas significan en español sistema operativo de disco de Microsoft.
- **Mugen:** Es un motor de videojuegos de lucha 2 dimensiones (2D), 1vs1 y gratuito.
- **Multiplataforma:** En informática se refiere a una aplicación o biblioteca que puede usarse en distintos sistemas operativos.
- **OpenMugen:** Versión independiente y de código abierto del motor de videojuegos de lucha Mugen.
- **Open Source:** Con este término es conocido el software distribuido y desarrollado libremente.
- **Player:** Término por el que se conoce a un jugador de videojuegos.
- **Plugin:** Aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy específica.
- **Scripting:** Es la acción de programar pequeños programas en un fichero de texto usando un lenguaje de programación interpretado. Normalmente a estos lenguajes se les llama lenguajes de scripting.
- **Renderizar:** Es una jerga informática referida al dibujado de una imagen en un ordenador.

- **Round:** Cada uno de los asaltos de un combate.
- **Singleton:** Patrón de diseño aplicado a una clase y que consiste en mantener una solo instancia de un objeto en toda la aplicación.
- **Sistema embebido:** Son sistemas programados generalmente en C/C++, en el por medio de otro lenguaje de programación de alto nivel, cuyo intérprete esta embebido, se puede personalizar, controlar y extender la aplicación.
- **Sistema experto:** Es una aplicación que posee información en un área específica a partir de una base de conocimientos.
- **Software:** Partes no tangibles de un ordenador, como un programa.
- **Socket:** Concepto abstracto por el cual dos programas pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada.
- **Videoconsola:** Es un ordenador de uso específico cuyo objetivo, a través de un medio de entrada ejecuta un videojuego.
- **Videojuego:** Es un software orientado al ocio en el que una persona interactúa con la máquina a fin de alcanzar un objetivo.

Capítulo 3

Organización temporal

La realización del proyecto ha estado muy ligado a las pruebas y se ha dividido en iteraciones, unas nueve en total. Cada vez que se termina una iteración una versión del sistema con una funcionalidad concreta es liberada, y unos casos de prueba también. A continuación se listan y se detalla el desarrollo del proyecto:

1. **Primera iteración:** En esta primera fase del proyecto se implementa la capa de dominio de bajo nivel, en concreto las clases que representan una imagen y la clase que representa una fuente de texto.

2. **Segunda iteración:** En esta fase se implementa las clases para reproducir animaciones y para la gestión de las colisiones.

3. **Tercera iteración:** En esta fase se implementa las clases para reproducir música y efectos de sonido, además de funciones para gestionar eventos en distintos dispositivos al más bajo nivel.

4. **Cuarta iteración:** En esta fase se implementa la clase **FiniteStateMachine** que dota a una clase de comportamiento mediante una máquina de estados, relacionándola con el sistema de eventos y comunicándose con el sistema de código embebido. Se diseña y desarrolla la clase para el luchador, llamada **Fighter**.

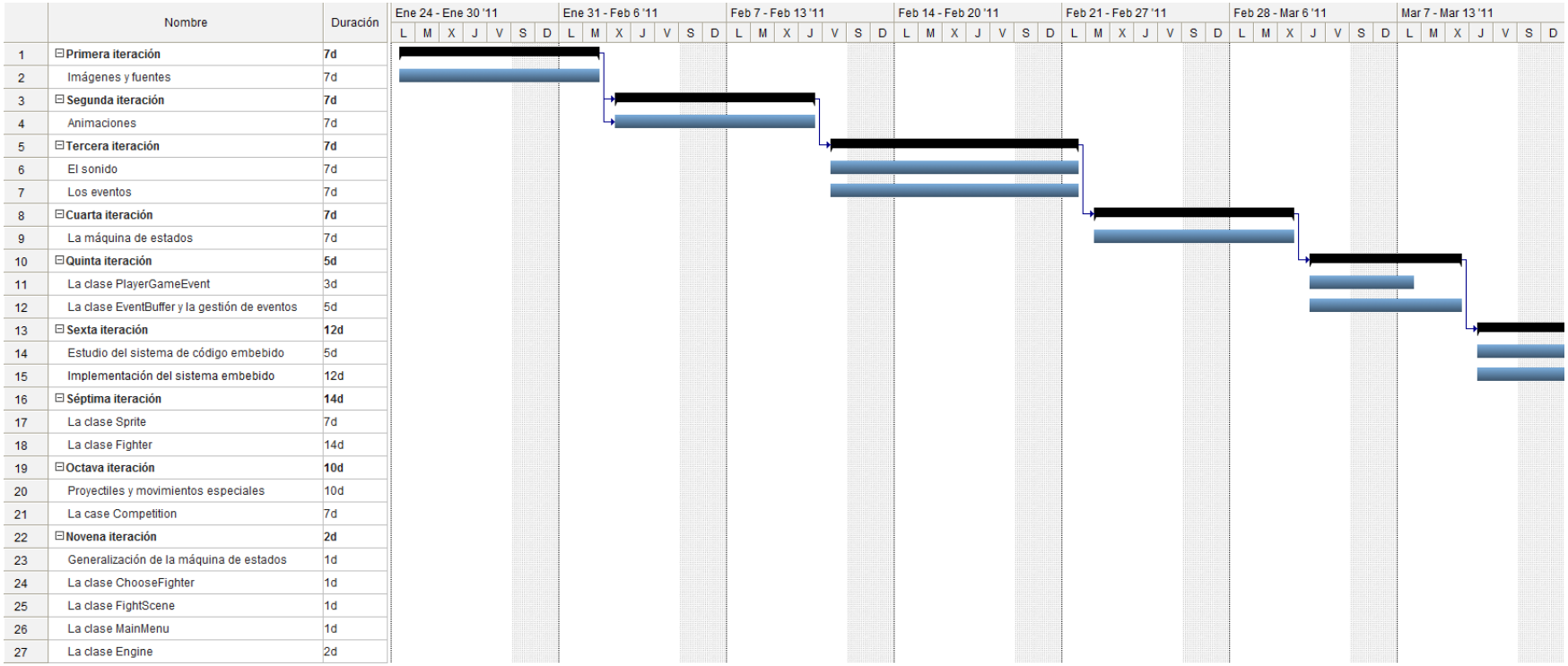
5. **Quinta iteración:** En esta fase se implementa la clase **Sprite** que sirve como base a toda clase que soporte animaciones en el simulador, así como la física del simulador.

6. **Sexta iteración:** En esta fase se diseña e implementa la clase **Projectile**, que representa un proyectil en el simulador y los movimientos especiales que puede lanzar un luchador, además de crear una clase común FightEntity que hereda toda clase del simulador que participa en un combate. También se implementa la clase **FightScene** para representar el escenario, añadiendo componentes visuales propios de un juego de lucha (barras de vida, cronómetro para el tiempo de un asalto, etc).

7. **Séptima iteración:** En esta fase se implementa en el simulador una estructura de directorios y se diseña e implementa la clase **Game**, que administra las propiedades globales del simulador, sirviendo al resto de clases como capa de datos.

8. **Octava iteración:** En esta fase se implementan los distintos tipos de competiciones, mediante el diseño y la implementación de la clase **Competition** y sus clases derivadas, una por cada tipo de competición o de juego.

9. **Novena iteración:** Creación de los distintos escenarios como la clase **FightScene** (para hacer luchar a dos luchadores), la clase **ChooseScene** (que hace elegir a un luchador dependiendo de la competición) y la clase **MainMenu** (que permite elegir el tipo de competición y modificar el archivo de configuración del simulador).



	Nombre	Duración
1	**Primera iteración**	**7d**
2	Imágenes y fuentes	7d
3	**Segunda iteración**	**7d**
4	Animaciones	7d
5	**Tercera iteración**	**7d**
6	El sonido	7d
7	Los eventos	7d
8	**Cuarta iteración**	**7d**
9	La máquina de estados	7d
10	**Quinta iteración**	**5d**
11	La clase PlayerGameEvent	3d
12	La clase EventBuffer y la gestión de eventos	5d
13	**Sexta iteración**	**12d**
14	Estudio del sistema de código embebido	5d
15	Implementación del sistema embebido	12d
16	**Séptima iteración**	**14d**
17	La clase Sprite	7d
18	La clase Fighter	14d
19	**Octava iteración**	**10d**
20	Proyectiles y movimientos especiales	10d
21	La case Competition	7d
22	**Novena iteración**	**2d**
23	Generalización de la máquina de estados	1d
24	La clase ChooseFighter	1d
25	La clase FightScene	1d
26	La clase MainMenu	1d
27	La clase Engine	2d

Figura 3.1: Diagrama de Gantt 1

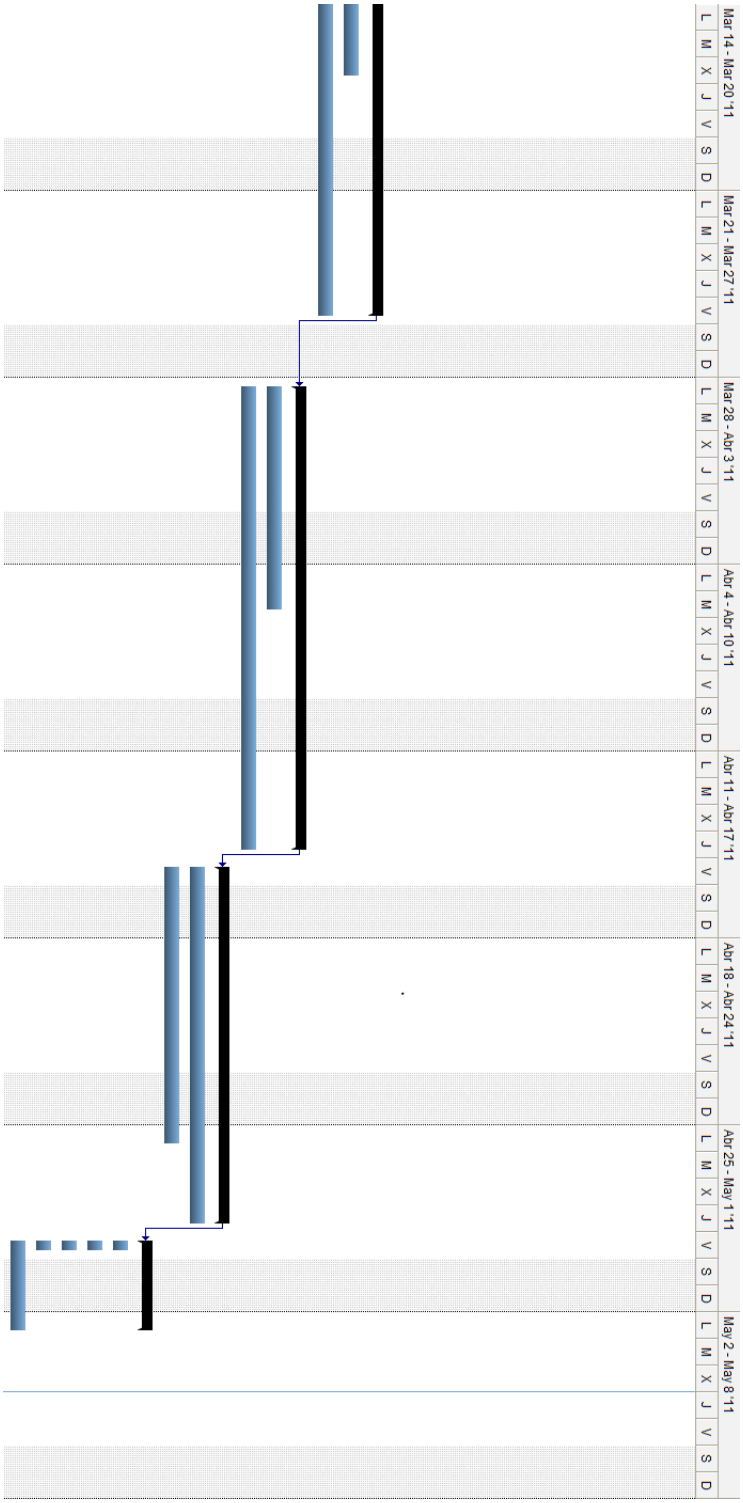


Figura 3.2: Diagrama de Gantt 2

Capítulo 4

Análisis del sistema

En el capítulo 2 se explica en que consiste un juego de lucha. Siguiendo esta descripción se han especificado los requisitos y se ha documentado el desarrollo del análisis.

4.1. Especificación de los requisitos del sistema

En esta sección se explican los requisitos funcionales y no funcionales, los cuales han servido de base para modelar los casos de uso 4.2, el modelo conceptual 4.3 y los diagramas de secuencia 4.4.

4.1.1. Requisitos de interfaz externa

La forma de interactuar del usuario con el sistema será mediante un joystick o el teclado del ordenador. El videojuego muestra primero un menú en el que el jugador puede elegir el modo de juego o modificar las opciones de éste. Una vez elija el modo de juego se accede a otra pantalla donde se escoge un luchador y luego con ese luchador se enfrenta a un combate Uno contra uno (1vs1).

4.1.2. Requisitos funcionales

- Existirán varios personajes que el jugador podrá elegir. Cada uno de estos personajes tendrá características distintas como tamaños, magias y movimientos especiales.
- Se permitirá que se introduzcan jugadores nuevos de acuerdo a una serie de condiciones (esto hará el juego extensible). No será necesario para ello el conocimiento de aspectos técnicos internos del videojuego.
- El juego tendrá un modo torneo donde dos jugadores elegirán a varios personajes y lucharán entre sí. Se contemplarán varios tipos de torneos.
- Igualmente se dispondrá de una opción de entrenamiento donde practicar con un personaje sus habilidades, magias, etc.

- También existirá un modo para jugar contra el ordenador en distintos niveles de dificultad.

4.1.3. Otros requisitos

En un juego de acción los movimientos y las combinaciones de teclas deben responder en tiempo real. Referente a los atributos del sistema software, la interfaz de usuario debe de ser intuitiva. El videojuego debe de ser manejable tanto usando un joystick como un teclado de ordenador.

4.2. Modelo de casos de uso

Caso de Uso: Arcade

- Descripción: Un juego en modo Arcade consiste en una serie de combates de un luchador controlado por el jugador contra luchadores controlados por la máquina.
- Actor principal: Jugador.
- Precondiciones: Ninguna.
- Postcondiciones: Ninguna.
- Escenario principal:
    1. El Jugador escoge luchador y el sistema los gestiona. Incluye Escoger luchador.
    2. El Sistema calcula el número de luchadores disponibles, creando una lista de oponentes.
    3. El Sistema escoge al azar a un luchador no derrotado de la lista. Incluye Luchar.
    4. Repetir 3 hasta que todos los luchadores de la lista hayan sido derrotados.
    5. El sistema muestra un mensaje que ha terminado el juego satisfactoriamente.
- Escenarios alternativos:

    3 El luchador es vencido por el ordenador.

    a) El sistema muestra una pantalla indicando si se quiere continuar Jugando.

    1) El Jugador escoge seguir jugando.

    a' El jugador escoge luchador y el sistema lo gestiona. Incluye Escoger luchador.

    2) El Jugador escoge no seguir jugando.

    a' Se muestra un mensaje de juego terminado y se sale del caso de uso.

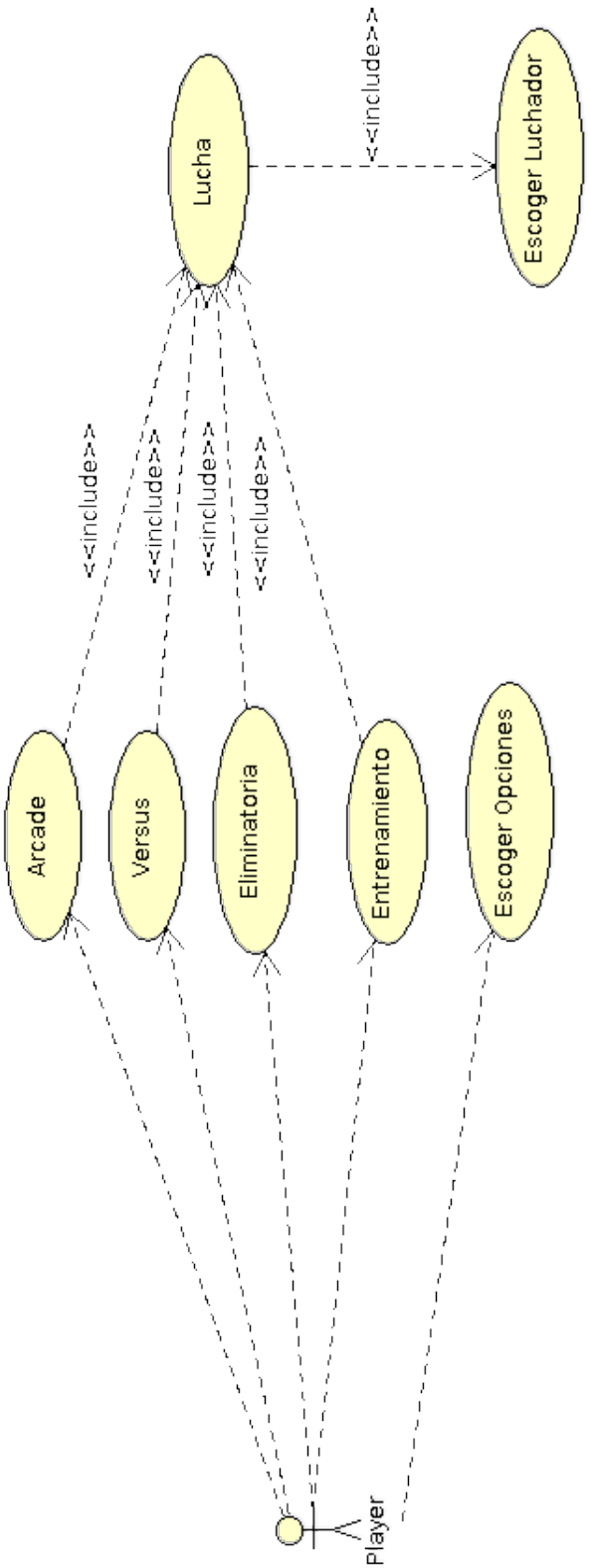


Figura 4.1: Diagrama de casos de uso

Caso de Uso: Versus

- Descripción: Dos luchadores manejados por el jugador luchan en un combate.
- Actor principal: Jugador.
- Precondiciones: Ninguna.
- Postcondiciones: Ninguna.
- Escenario principal:
    1. El jugador escoge a un luchador. Incluye Escoger luchador.
    2. El jugador escoge a otro luchador. Incluye Escoger luchador.
    3. El sistema genera una lista de escenarios y la muestra por pantalla.
    4. El jugador escoge un escenario de esa lista y empieza el combate. Incluye Luchar.

Caso de Uso: Eliminatoria

- Descripción: Se escoge una serie de luchadores que combatirán en duelo uno contra otro hasta quedar uno vencedor.
- Actor principal: Jugador.
- Precondiciones: Ninguna.
- Postcondiciones: Ninguna.
- Escenario principal:
    1. El sistema, dependiendo del número de luchadores disponibles, genera un número par de luchadores, que será el número máximo de luchadores que puede escoger el jugador. Éste se muestra por pantalla.
    2. El jugador escoge un luchador. Incluye Escoger luchador.
    3. Repetir 3 hasta que se escoge un número de luchadores igual al número de luchadores máximos.
    4. Por el orden en el que se hayan escogido, el sistema crea una lista de combates.
    5. El sistema ejecuta combates de la lista. Incluye Luchar.
    6. Repite 6 hasta que quede sólo un luchador invicto.
    7. El sistema muestra un mensaje de juego ganado.
- Escenarios alternativos:
    1. Error: Sólo existe un luchador disponible, el sistema muestra el error.

Caso de Uso: Training

- Descripción: Un jugador escoge a un luchador y se enfrenta contra otro que no se mueva, con el fin de probar las características de éste.
- Actor principal: Jugador.
- Precondiciones: Ninguna.
- Postcondiciones: Ninguna.
- Escenario principal:
    1. El jugador escoge a un luchador para probar. Incluye Escoger luchador.
    2. El sistema escoge a un luchador automáticamente.
- El sistema carga un escenario especial. Incluye Luchar.
- Escenarios alternativos:
    1. En cualquier momento el jugador puede salir del modo Training.

Caso de Uso: Escoger luchador

- Descripción: De una lista de luchadores disponibles el jugador escoge uno.
- Nivel subfunción.
- Actor principal: Jugador.
- Precondiciones: Ninguna.
- Postcondiciones: Ninguna.
- Escenario principal:
    1. El Sistema muestra una lista con los luchadores disponibles.
    2. El jugador escoge uno.
    3. El sistema lo registra.
- Escenarios alternativos:
    1. Error: No existen luchadores disponibles y el sistema muestra el error.

Caso de Uso: Escoger Opciones

- Descripción: El jugador puede cambiar aspectos del juego globales, como el tiempo del round o asalto, la dificultad del ordenador, etc.
- Actor principal: Jugador.
- Precondiciones: Ninguna.
- Postcondiciones: Ninguna.
- Escenario principal:
    1. El sistema muestra las opciones por pantalla.
    2. El jugador modifica el tiempo de un asalto.
    3. El sistema registra el nuevo tiempo.
    4. Repetir 1-3 hasta que el jugador indique.
- Escenarios alternativos:
    1. El jugador cambia la dificultad de la máquina.
        a) El sistema registra el nuevo tiempo.
    2. El Jugador cambia las opciones de los controles.
        a) El sistema cambia las opciones de los controles.
    3. El jugador cambia el número de asaltos.
        a) El sistema guarda el número de asaltos.

Caso de Uso: Luchar

- Descripción: Dos luchadores escogidos por el jugador o no, luchan en un combate 1vs1.
- Nivel subfunción.
- Actor principal: Jugador.
- Precondiciones: Ninguna.
- Postcondiciones: Ninguna.
- Escenario principal:
    1. El sistema indica el comienzo de un asalto.
    2. El sistema muestra por pantalla a los jugadores y el escenario en el que combaten.
    3. El jugador interactúa con el sistema mediante el dispositivo.
    4. El sistema registra los eventos de los dispositivos de entrada, comprobando las colisiones y los efectos de esos eventos.

5. Repite 2-4 hasta que alguna barra de vida llegue a cero o el tiempo del round o asalto se acabe.
6. El sistema registra al ganador de un asalto.
7. Repite 1-6 hasta que sistema indique un ganador del combate.

- Escenarios alternativos:

3.a El sistema se apropia del personaje. Esto es debido a que éste ha activado alguna acción relevante como: Golpear, Saltar, etc. El sistema no escuchará determinados eventos, y si es el caso resta la barra de vida de algún luchador respondiendo a determinados eventos.

3.b El sistema comprueba que en un búfer global que registra eventos del jugador haya una magia o movimiento especial, entonces lo crea. Encargándose el sistema de gestionarlo.

6.a Si los dos jugadores pierden su barra de vida a la vez no se registra el ganador de un asalto.

4.3. Modelo conceptual

4.3.1. Modelo de clases

Las clases que vamos a necesitar son:

- **Game:** Se encarga de gestionar el juego y representa un ambiente global dentro de la aplicación y contiene variables que determinan características de los combates.
- **Combat:** Clase encargada de administrar una lucha 1vs1.
- **Competition:** Clase encargada de almacenar todos los aspectos de una competición compuestas de varios combates y un ganador.
- **TCompetition:** Enumerado que representa un tipo de competición, cuyos valores son *ARCADE, VERSUS, TOURNAMENT, TRAINING*, correspondientes a los tipos de competiciones de modo arcade, versus, eliminatoria y training.
- **Player:** Identifica y mantiene todos los datos necesarios para administrar a un jugador.
- **VirtualKey:** Enumerado que consta de los siguientes significados: *UP, DOWN, RIGHT, LEFT, A, B, C, X, Y, Z, START*. Estos valores son correspondientes a los de un joystick de seis botones y una tecla START.
- **PlayerGameEvent:** Se encarga de almacenar una serie de VirtualKeys que identifican los posibles eventos que puede lanzar un jugador.
- **TypePlayer:** Enumerado que representa un tipo de jugador, los valores son HUMAN, NPC, NPC _TRAINING.
- **GameFighter:** Un luchador del juego.
- **GameScene:** Un escenario del juego.

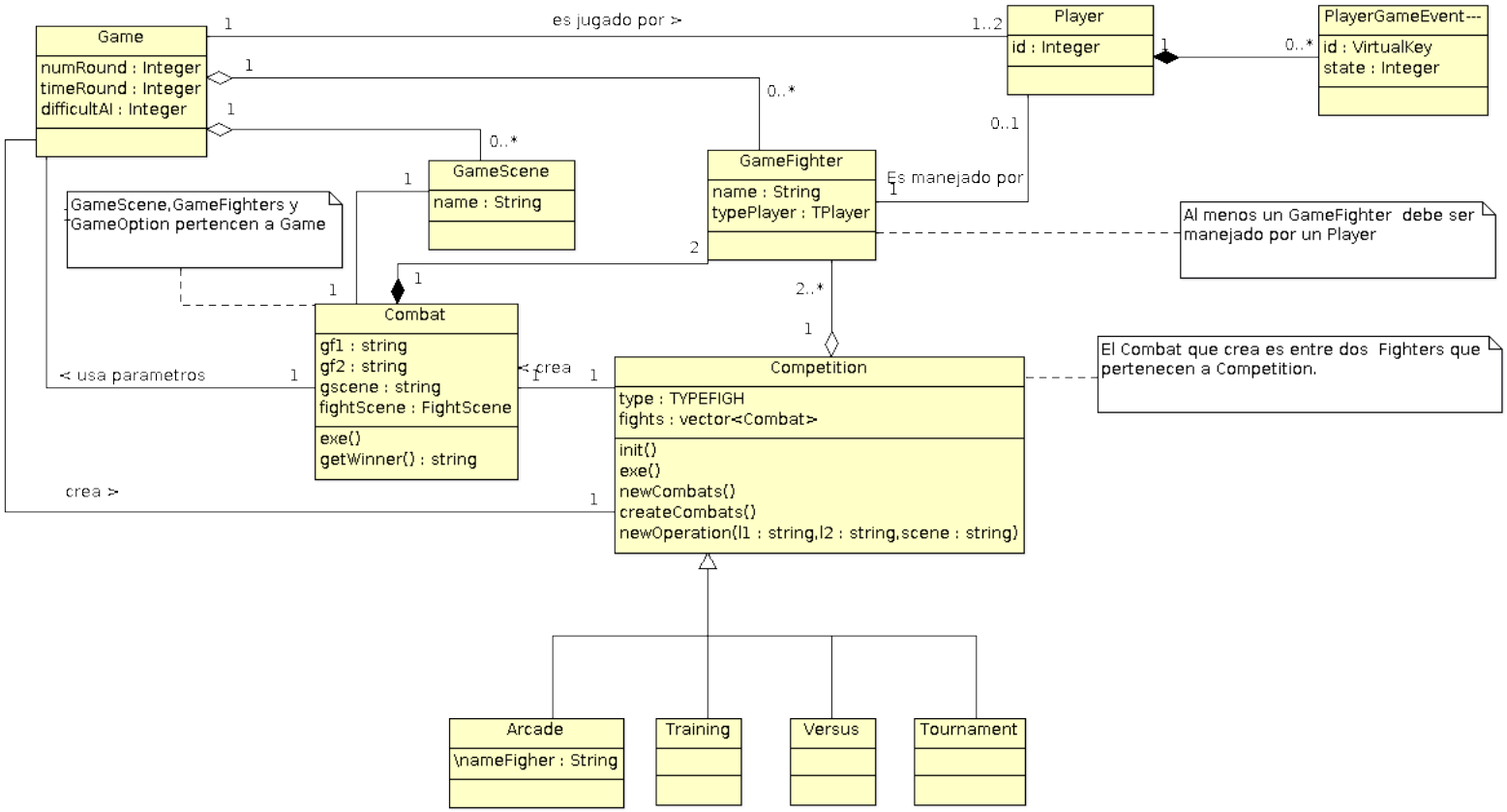


Figura 4.2: Diagrama de clases

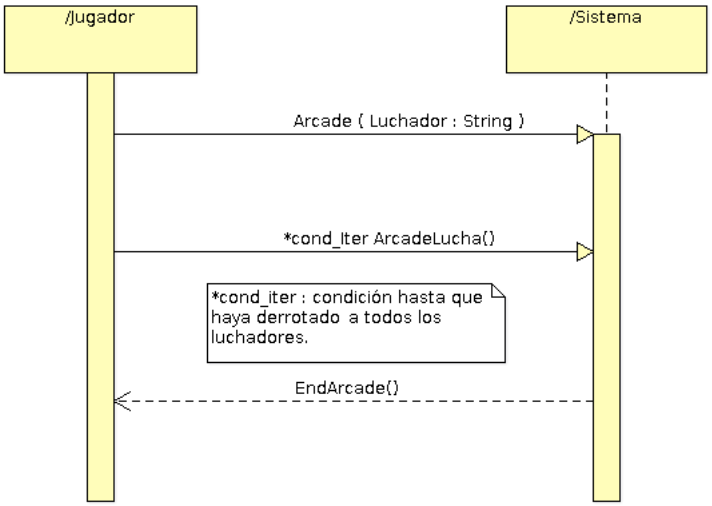


Figura 4.3: Diagrama secuencia de arcade

4.4. Diagramas de secuencia

Caso de Uso: Arcade

Contrato de las operaciones[7.3pt] Operación: **Arcade**(luchador : String)

- Responsabilidades: Crea un modo de juego Arcade.
- Precondiciones: Luchador es el identificador de un objeto **GameFighter** existente y su TypePlayer = HUMAN.
- Postcondiciones: Se crea un objeto de la clase **Arcade** y se relaciona con una lista de oponentes a los que tiene que derrotar.
- Detalles:
  - Se crea una instancia de **Competition**: competition.
  - Se especializa competition en **Arcade**.
  - Se asocia un objeto **GameFighter** L con L.name = luchador y se asocia con competition.
  - Se modifica competition.nameFighter = L.name.
  - Calcula el número de luchadores disponibles en el sistema, creando un objeto **GameFighter** por cada uno y asociándolo a competition.

Operación: **ArcadeLucha()**

- Responsabilidades: Se encarga de gestionar los combates ganados por el jugador para saber si el juego ha terminado.
- Precondiciones: Existe una lista de instancias de **GameFighter**: gameFighter, existe dos instancias de **GameFighter**: f1, f2.
- Postcondiciones: Crea un combate entre dos luchadores en donde: f1.type = PLAYER y f2.type = NPC.

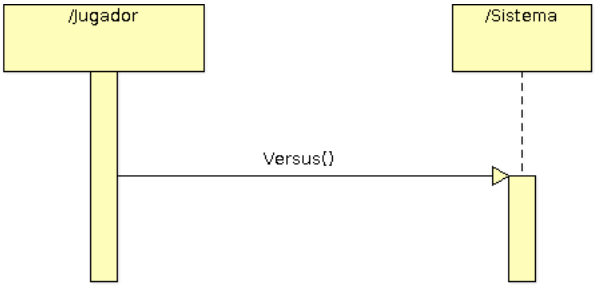


Figura 4.4: Diagrama secuencia de versus

Caso de Uso: Versus

Contrato de las operaciones[7.3pt] Operación: **Versus**()

- Responsabilidades: Se encarga de hacer luchar a dos **GameFighter** manejados por el jugador.
- Precondiciones: Existe una instancia de **Game**: game.
- Postcondiciones: Ninguna.
  - Se crea una instancia de **Competition**: competition.
  - Se asigna competition.type = VERSUS.
  - Se especializa competition en la clase Versus.

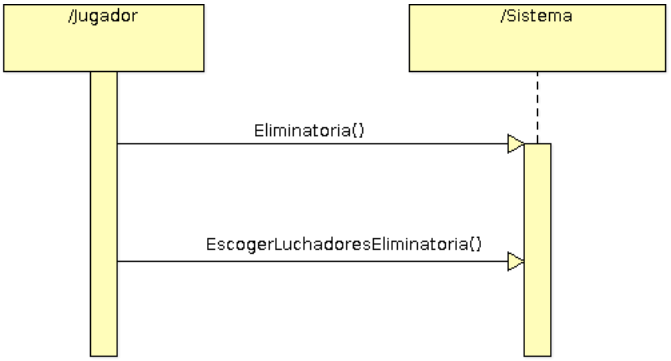


Figura 4.5: Diagrama de secuencia de eliminatoria

Caso de Uso: Eliminatoria

Contrato de las operaciones[7.3pt] Operación: **Eliminatoria**()

- Responsabilidades: Se encarga de hacer luchar a varios **GameFighter** manejados por el jugador, en combates a modo de eliminatoria.
- Precondiciones: Existe una instancia de **Game**: game.
- Postcondiciones: Ninguna.
  - Se crea una instancia de **Competition**: competition.
  - Se asigna competition.type = TOURNAMENT.
  - Se especializa competition en la clase **Tournament**.
  - Se asocia game con competition.

Operación: **escogerLuchadoresEliminatoria**()

- Responsabilidades: Se encarga de hacer escoger un luchadores al jugador.
- Precondiciones: Existe una instancia de **Game**: game, **Competition**: competition, una lista de luchadores.
- Postcondiciones: Se crea una lista de luchadores para la competición y una lista de combates.

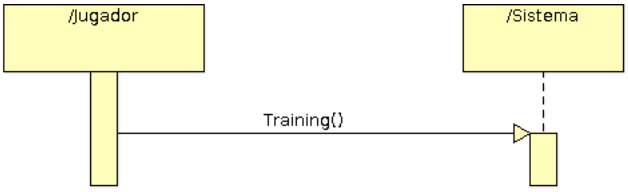


Figura 4.6: Diagrama de secuencia de training

Caso de Uso: Training

Contrato de las operaciones.[7.3pt] Operación: **Training**()

- Responsabilidades: Se encarga de hacer luchar a un **GameFighter** contra otro manejado por la maquina inmortal y que no se mueve.
- Precondiciones: Existe una instancia de **Game**: game, existe una instancia de **GameFighter**: Fighter.
- Postcondiciones: Ninguna.
  - Se crea una instancia de **Competition**: competition.
  - Se asigna competition.type = TRAINING.
  - Se especializa competition en la clase Training.
  - Se asocia game con competition.
  - Se crea un **Fighter** fighter escogido al azar.
  - Se asigna fighter.type = TRAINING.

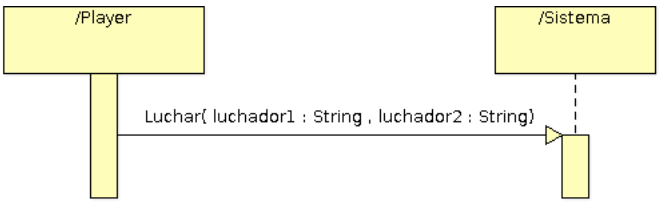


Figura 4.7: Diagrama de secuencia de luchar

Caso de Uso: Luchar

Contrato de las operaciones [7.3pt] Operación: **Luchar**(luchador1 : String, luchador2 : String)

- Responsabilidades: Se encarga de crear un combate entre dos luchadores, indistintamente manejado por la máquina o por el jugador.
- Precondiciones: Existe una lista de **GameFighter**, existen dos instancias de la clase **Fighter**, existe una lista de GameScene, existe una instancia de **Competition**: competition.
- Postcondiciones: Crea un combate entre dos luchadores y carga un escenario al azar del sistema.
- Detalles:
  - Se crea una instancia de **GameScene**: gameScene.
  - Se calcula el jugador Siguiente para enfrentarse: fighterForward.
  - Se crea una instancia de **Combat**: combat.
  - Se asocia combat con **gameScene**, fighterForwar y un objeto **GameFighter** L con L.name = competition.nameFighter.

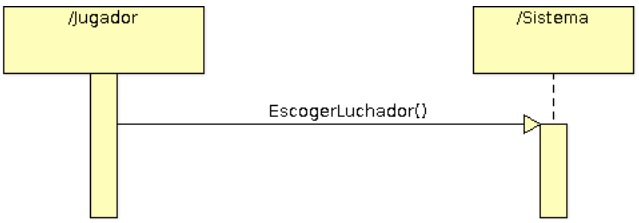


Figura 4.8: Diagrama de secuencia de escoger luchador

Caso de Uso: Escoger luchador

Contrato de las operaciones[7.3pt] Operación: **EscogerLuchador**()

- Responsabilidades: Se encarga de cargar un luchador en el sistema.
- Precondiciones: Existe una instancia de **Player**: player y otra de **Competition**: competition.
- Postcondiciones: Muestra un menú para escoger a un máximo de dos jugadores.
- Detalles:
  - Se crea una instancia de **GameFighter**: gameFighter.
  - Se asocia **gameFighter** con Player: player.
  - Se asocia **gameFighter** con competition.

Capítulo 5

Diseño del sistema

En el capítulo anterior 4 se ha descrito el análisis del sistema y tomando como punto de partida éste, ahora se describe el diseño del sistema.

5.1. Arquitectura del sistema

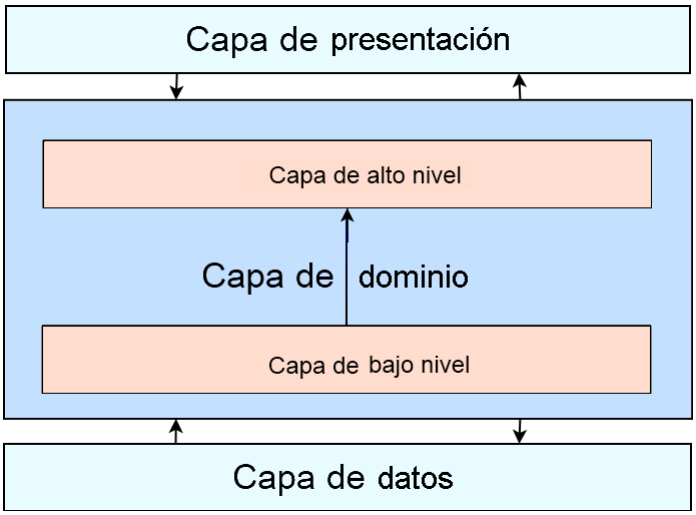


Figura 5.1: Arquitectura del simulador

Seguiremos una arquitectura en tres capas:

1. **La capa de datos:** Se encarga de gestionar los datos globales y accesibles del juego, se comunica con la capa de dominio.

2. **La capa de negocio o dominio**: Se encarga de dar funcionalidad al sistema respondiendo a los eventos del usuario. En este capa se ha implementado soporte de bajo nivel que se encarga de gestionar los aspectos relacionados con el hardware y el sistema operativo (eventos, gráficos, sonidos, etc.) dando soporte a la capa de dominio. A su vez podemos, para más claridad, dividir en 2 esta capa:

a) **La capa de bajo nivel:** Abstrae los aspectos de más bajo nivel de la programación dando soporte a una de más alto nivel. Esta capa envuelve los elementos de programación de las bibliotecas utilizadas para relacionarse con el hardware, creando funciones, enumerados y clases.

b) **La capa de alto nivel:** Utiliza elementos de la capa de alto nivel y proporciona las funciones típicas de la capa de dominio, además de usar las clases de la capa de bajo nivel.

3. **La capa de presentación:** Se encarga de la interfaz del juego y se comunica con la capa de dominio.

5.2. La capa de dominio de bajo nivel

La capa de dominio de bajo nivel da soporte a la capa de dominio de alto nivel, envolviendo los aspectos de más bajo nivel. La biblioteca que se encarga de la relación con al hardware es libSDL. Como está escrita en C, carece de clases y la programación esta orientada a bajo nivel y aspectos relacionados con el hardware.

El objetivo principal de esta subcapa es dotar de clases básicas a la subcapa de alto nivel. Estas clases proporcionan la base para el simulador, como lo son mostrar imágenes (clase **Image**), escuchar música (clase **MusicPlayer**), escuchar efectos de sonido (clase **SFXMusicPlayer**), visualizar fuentes de texto (clase **Font**), manejar eventos del usuario relacionados con los dispositivos (clase **PlayerGameEvent**), contabilizar el tiempo (clase **Time**) o tener un búfer para guardar eventos (clase **EventBuffer**).

5.2.1. Tipos

La biblioteca de bajo nivel usada para interactuar con el hardware es libSDL. Esta biblioteca está escrita en el lenguaje C y está pensada para manipular gráficos, sonidos y eventos a bajo nivel. Se han nombrado los siguientes tipos:

- typedef SDL_Surface* Surface; Representa una imagen.
- typedef SDL_Rect Rect; Representa un rectángulo.
- typedef Mix _Chunk* Sound; Representa un efecto de sonido.
- typedef Mix_Music* Music; Representa un archivo de audio.
- typedef SDL_Color Color; Representa un color.
- typedef TTF_Font* TFont; Representa un archivo de fuente True Type Font (TTF).

Por claridad se ha envuelto tipo estándar std::map de la Standar Template Library (STL) en un tipo THash<T,K>envolviendo operaciones comunes con este tipo.

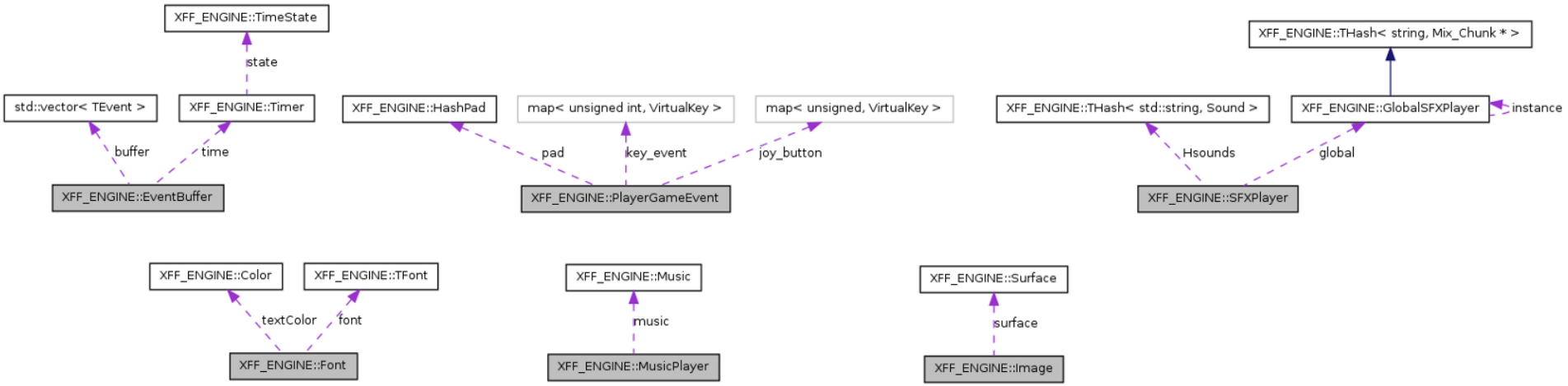


Figura 5.2: La capa de dominio de bajo nivel

5.2.2. Gráficos

La clase Image

Representa una imagen en el sistema. Hay que tener en cuenta el comportamiento de la imagen ya que estará preparada no solo para mostrarse por pantalla, si no para tareas de manipulación de imágenes como rotaciones horizontales y verticales.

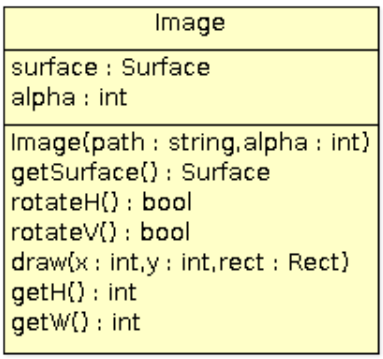


Figura 5.3: Diagrama de clase de Image

- **Métodos públicos:**
  - Image(std::string path, int &alpha) Constructor que recibe la ruta de la imagen y el color alpha. La imagen debe esta en el formato Portable Network Graphics (PNG).
  - Surface getSurface () Devuelve una instancia de Surface.
  - bool flipH () Rota horizontalmente la imagen.
  - bool flipV () Rota verticalmente la imagen.
  - int getW () Devuelve el ancho de la imagen.
  - int getH () Devuelve el alto de la imagen.
  - void draw (int x, int y, Rect *rect=0) Dibuja por pantalla la imagen.

La clase Font

Representa una forma de dibujar texto por pantalla proporcionando una cadena de caracteres. Para conseguir esta funcionalidad internamente, carga un archivo de fuentes TTF usando la biblioteca libSDL_ttf. Esta clase está limitada a un tamaño y a un color, es decir, cuando creamos una instancia de una clase **Font** debemos proporcionar su tamaño y su color. Esto es debido al diseño de la biblioteca utilizada para cargar fuentes.

- **Métodos públicos:**

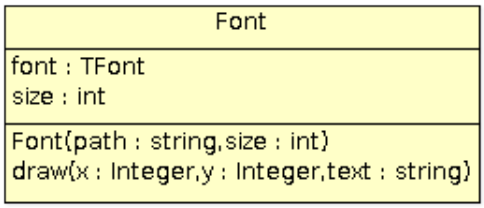


Figura 5.4: Diagrama de clase de Font

- Font(std::string& path_font, int size, int color) Constructor que recibe la ruta de un fichero de fuente TTF, el tamaño de la fuente y el color de la misma expresado en hexadecimal.
- void draw (int x, int y, std::string& text, Rect *=0, bool inverse) Dibuja la fuente por pantalla, en una posición concreta definida por los enteros x e y, una determinada cadena de caracteres definida por text, en un determinado rectángulo definido por Rect y en una determinada orientación inverse, que si tiene el valor verdadero se escribe en el sentido contrario.

- **Atributos privados:**
  - TFont font Estructura para la fuente.
  - Color textColor Estructura para el color.
  - unsigned size Tamaño de la fuente.

5.2.3. Sonido

Para manipular el sonido se ha utilizado la biblioteca libSDL_mixer, la cual permite reproducir pistas de audio y efectos de sonido. En el simulador se han diseñado tres clases que están relacionadas con la reproducción de medios: GlobalSFXPlayer, SFXPlayer y MusicPlayer.

La clase GlobalSFXPlayer

La clase **GlobalSFXPlayer** usa el patrón de diseño singleton, lo que le hace que sea global y accesible desde cualquier lugar de la aplicación, esto se hace debido a la cantidad de clases que pueden llegar a reproducir sonidos y porque existen sonidos que pueden ser reproducidos en cualquier lugar de la aplicación.

Utiliza para este efecto un diccionario de sonidos, en donde las claves son cadenas de caracteres que actúan como identificador y los valores que son las estructuras de datos de la biblioteca para almacenar en memoria un sonido.

- **Métodos públicos:**
  - GlobalSFXPlayer& getInstance() Método público y estático que devuelve la única instancia de la clase **GlobalSFXPlayer** que existe en el sistema.

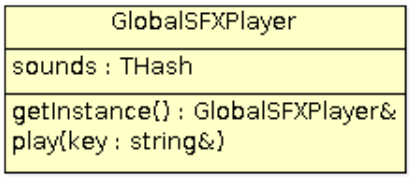


Figura 5.5: Diagrama de clase de GlobalSFXPlayer

- void play(string& key) Reproduce un sonido mediante su clave en el diccionario de sonidos.

- **Atributos privados:**
  - THash<string,Sound>sounds Diccionario cuya clave es un string y valor un sonido.

La clase SFXPlayer

La clase **SFXPlayer** tiene como objeto en el sistema servir de reproductor de sonidos complementario al reproductor de sonidos global **GlobalSFXPlayer**.

Esta clase reescribe las claves del reproductor de sonidos global **GlobalSFXPlayer**, de manera que si por ejemplo si existe en el sistema un sonido con clave sound_1 y un objeto de la clase **SFXPlayer** tiene un sonido con clave sound_1 se reproduce este sonido cuando una instancia de **SFXPlayer** llama a su método play(), si por el contrario intentara reproducir un sonido que no existe en su diccionario , intentará reproducirlo del reproductor de sonidos global.

El motivo por el que se ha diseñado esto así es principalmente porque un luchador tiene muchos sonidos que están cargados en el reproductor global: Puñetazos, patadas, caídas, etc. Pudiendo éstos ser reescritos en el diccionario que posee esta clase.

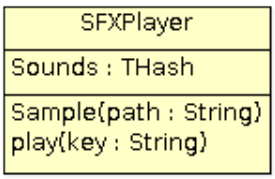


Figura 5.6: Diagrama de clase de SFXPlayer

- **Métodos públicos:**
  - void play(string& key) Reproduce un sonido mediante su clave en el diccionario de sonidos.

- **Atributos privados:**
  - THash<string,Sound>sounds Diccionario cuya clave es un string y valor un sonido.

La clase MusicPlayer

Clase capaz de reproducir una pista de audio en el sistema, cuya funcionalidad es servir como reproductor de música. Reproduce diferentes tipos de medios diferentes MP3, OGG, WAV, etc. Aunque en el simulador se recomienda que se utilice el formato OGG para evitar las patentes de otros formatos al distribuir el audio.

Esta clase también sigue el patrón de diseño singleton, con lo cual cualquier clase puede cargar música y hacer que se reproduzca ésta. La justificación de esta manera de trabajar es que hay multitud de clases que pueden reproducir música, para ahorrar el estar creando objetos y destruyendo objetos en todo momento con el simple propósito de reproducir pistas de audio, se ha diseñado esta clase de manera que haya una sola instancia en todo momento en la aplicación, de manera que sean las clases que lo necesiten las que se encarguen de llamar a la instancia global de **MusicPlayer**.

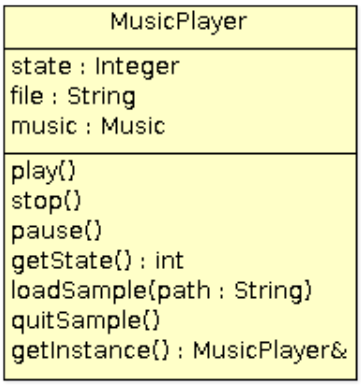


Figura 5.7: Diagrama de clase de MusicPlayer

- **Métodos públicos:**
  - void play() Reproduce la pista de audio cargada.
  - void stop() Para la reproducción una pista.
  - void pause() Pausa o reanuda la reproducción de una pista.
  - int getState() Devuelve el estado del reproductor: -1 no cargado, 0 parado 1 reproduciéndose, 2 pausa.
  - void loadSample(string& path) Carga una pista de audio mediante su ruta.
  - void quitSample(string& path) Libera el espacio de memoria ocupado por una pista de audio.
  - MusicPlayer& getInstance() Método estático que devuelve la instancia global de MusicPlayer.
- **Atributos privados:**
  - Music music Estructura de datos para la pista de audio.

- string file Cadena que representa la ruta del fichero, el motivo es que si alguna clase intenta cargar una pista con la misma ruta ésta no se cargue de nuevo.
- int state Guarda el estado del reproductor.

5.2.4. Control del tiempo

Ésta clase proporciona al simulador la funcionalidad de contar el tiempo. Esto es muy importante, por ejemplo, contar el tiempo de un asalto o para vaciar el búfer de teclas pasado un cierto tiempo si el jugador no pulsa ninguna tecla. Para implementar esta funcionalidad se hace uso de la biblioteca libSDL, en concreto de la función SDL_delay, que nos devuelve una marca de tiempo. La clase encargada de la gestión de tiempo es la clase **Timer**, la cual se explica a continuación.

La clase Timer

La clase Timer se usa en el sistema con funcionalidad parecida a un cronómetro de bolsillo. Tiene un método start, que la activa, un método pause que la pausa. La escala a seguir es el milisegundo.

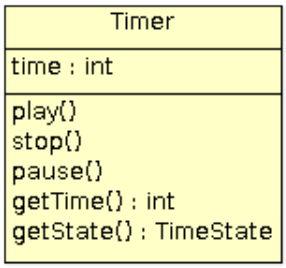


Figura 5.8: Diagrama de clase de Timer

- **metodos públicos:**
  - void start () Comienza el tiempo.
  - void stop () Para el tiempo.
  - void pause () Pausa el tiempo.
  - int getTime () Obtienes en milisegundos el tiempo transcurrido desde la llamada a Timer::start.
  - TimeState getState () Devuelve el estado del cronometro.
- **Atributos privados:**
  - enum TimeState IDLE, PLAY, STOP, PAUSE Enumerado que describe el estado del tiempo, IDLE no iniciado, PLAY está activo, STOP está parado, PAUSE está pausado.

5.2.5. Gestión de eventos

La gestión de eventos está pensada para dar soporte al teclado y al joystick. Es capaz de diferenciar entre los dos jugadores, guardando todos los eventos reconocibles de los dispositivos en un búfer global accesible en todo el juego, escuchado y manipulado por los objetos que hereden de las clases **FiniteStateMachine** y **Scene**.

La estructura TEvent

Representa un evento recogido de un dispositivo. Esta estructura proporciona al simulador una forma común para recoger eventos de un dispositivo arbitrario, con lo que si añadiésemos un ratón, por ejemplo, y éste generase un evento, generaría un **TEvent**. En el simulador los dispositivos que son reconocidos son el joystick y el teclado, siendo definidos estos dispositivos en el enumerado Device.

- **Atributos públicos:**
  - VirtualKey tecla Significado del botón mirar documentación de los eventos.
  - int estado Significa int estado: 0 no pulsado, 1 pulsado.
  - EventPlayer id Identificador del jugador.
  - int time Marca de tiempo.
  - Device device Dispositivo del cual es recogido el evento.

La clase HashPad

Representa un mando virtual para saber qué botones o teclas virtuales están pulsadas en todo momento. En esencia es un vector de booleanos de posiciones igual al número de teclas que puede pulsar un jugador.

- **Métodos públicos**
  - bool& at (VirtualKey key) Método modificador que modifica un valor dado su identificador.
  - bool read (VirtualKey key) Consulta un valor de un botón dado su identificador.
- **Atributos privados:**
  - vector<bool>pad Vector de booleanos cuya longitud es igual al número de teclas virtuales del sistema.

La clase PlayerGameEvents

Es una clase que se encarga de administrar los aspectos de bajo nivel de la escucha de eventos de los distintos dispositivos, de manera que contiene las teclas ,de éstos, así como el identificador del jugador que lo ha lanzado.. De esta clase solo existirá dos instancias en el sistema, una por cada jugador.

- **Atributos Públicos**
  - map<unsigned int, VirtualKey >key_event Diccionario que contiene SDL_key y un VirtualKey.
  - unsigned joystick_id Identificador del joystick de SDL.
  - map<unsigned, VirtualKey >joy_button Diccionario que contiene un identificador de un botón del joystick y un VirtualKey.
  - EventPlayer player Identificador global para el jugador.
  - HashPad pad Mando virtual.

El búfer global EventBuffer

Es un vector de **TEvent** que se encarga de recoger todos los eventos producidos por los dispositivos, para ello se ayuda de dos funciones:

- void ListenGlobalEvents() Escucha todos los eventos de los dispositivos.
- void ClearGlobalEvents() Limpia el búfer de eventos globales.

La función ListenGlobalEvents es muy importante, es la encargada de recoger los eventos a partir de la lectura de los manejadores de eventos de los usuarios que son las instancias existentes en el sistema de **PlayerGameEvent**, rellenando el búfer de instancias de **TEvent**. En esta función se engloba todos los aspectos de bajo nivel del tratamiento de eventos de libSDL, y si se añadiese un nuevo dispositivo, es aquí donde habría que añadirse para que el búfer global almacene un evento de ese dispositivo.

En la figura 5.9 se explica el tratamiento de los eventos, el gráfico representa la comunicación entre la pulsación de un botón o eje de un joystick y el sistema de gestión de eventos. XFF.out representa la aplicación en memoria y libSDL la biblioteca de bajo nivel que se encarga de comunicarse con el dispositivo. Se ha dividido en 6 pasos:

1. **Recogida de eventos en segundo plano:** La aplicación está en memoria y usa libSDL para gestionarlos. Esta biblioteca internamente guarda su propio búfer interno con los eventos escuchados que hay que procesar.
2. **La aplicación:** XFF.out representa la aplicación y en este paso libSDL guarda en su espacio de memoria reservado los eventos escuchados.
3. **Llamada a ListenEventBuffer:** En algún punto de la aplicación se llama a esta función.
4. **Recogida del manejador de eventos para el jugador1:** La función se encarga de leer los manejadores para el jugador1 (instancia de PlayerGameEvent), de manera que es capaz de discriminar en el búfer interno de libSDL, global en el sistema, entre eventos válidos y eventos que debe rechazar. De esta forma, si el manejador sólo lee la tecla A y se ha pulsado C y esta tecla no pertenece a ningún jugador (es decir, no esta registrado en ninguno de sus manejadores) este evento no se registra en el búfer global.

5. **Recogida del manejador de eventos para el jugador2:** Lo mismo que el paso 4 pero con el jugador2.

6. **Escritura en el búfer:** Se escribe en el búfer global un evento válido una vez que se ha comprobado que existe un jugador que lo tiene registrado como válido, entonces se crea un instancia de **TEvent** y se introduce en el vector global.

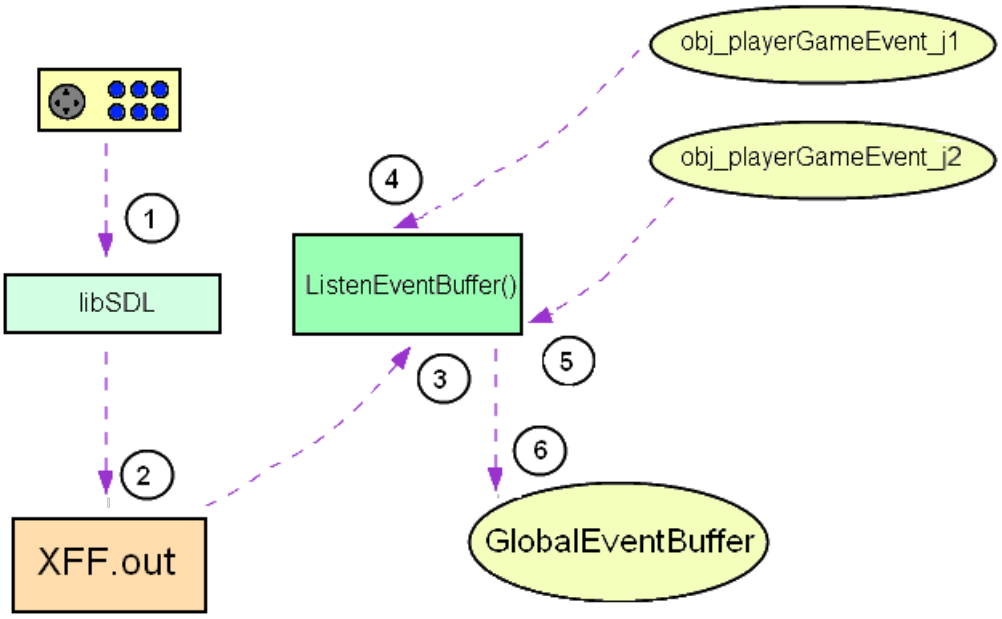


Figura 5.9: El tratamiento del búfer global de eventos

5.3. La capa de dominio de alto nivel

5.3.1. Las colisiones

Para las colisiones se usará el rectángulo como figura, para ello se usa el tipo **Rect**. El algoritmo, expresado en forma de función de C++ que recibe dos rectángulos, comprueba si los cuatro puntos de cada esquina de un rectángulo está contenido dentro de otro, si esto se cumple entonces devuelve verdadero.

Listado 5.1: Código que implementa una colisión entre rectángulos

```
bool RectCollideRect(const Rect& a, const Rect& b)
{
    return ( (b.x + b.w < a.x)  ||
             (b.x > a.x + a.w)  ||
             (b.y + b.h < a.y)  ||
             (b.y > a.y + a.h)
           ) ? false : true;
}
```

5.3.2. Animaciones

Una animación en un entorno de dos dimensiones, es mostrar consecutivamente una serie de imágenes que dan la sensación de movimiento al ojo humano. Para realizar esta acción se ha creado la clase **Animation**, la cual guarda una secuencia de diapositivas que representa una animación, en consecuencia esta clase representa un movimiento. Una diapositiva es representada en el sistema por la clase Frame. El simulador se ha diseñado de forma que guarde solamente los rectángulos de una imagen, de manera que dada una imagen, una instancia de Frame, guardaría el trozo de rectángulo dentro de esa imagen que representa una diapositiva.

Animations, como se muestra en la figura 5.10 representaría el conjunto de animaciones para una imagen dada, aunque no guarda la imagen en sí, solamente sus conjunto de diapositivas, para ello tendría un diccionario de **Animation**, siendo las claves cadenas de caracteres que sirven como identificador para una animación o movimiento. El objetivo de esta clase es que se puedan seleccionar animaciones. Para saber qué animación está cargada tiene un atributo llamado current, que guarda el identificador de una animación y una referencia a una secuencia de diapositivas o instancias de la clase **Frame** correspondiente a current.

Particularmente hay que destacar que siguiendo el mismo modelo que Mugen, una diapositiva guardará no sólo el rectángulo dentro de la imagen que le corresponde, si no también información sobre su dibujado y las colisiones. Por este motivo utiliza un punto de referencia para su dibujado (clase Point) además de dos vectores de rectángulos (clase **TCollisionRect**), uno para guardar la colisión de su cuerpo y otra la de sus golpes, sumado a la información del rectángulo de su animación.

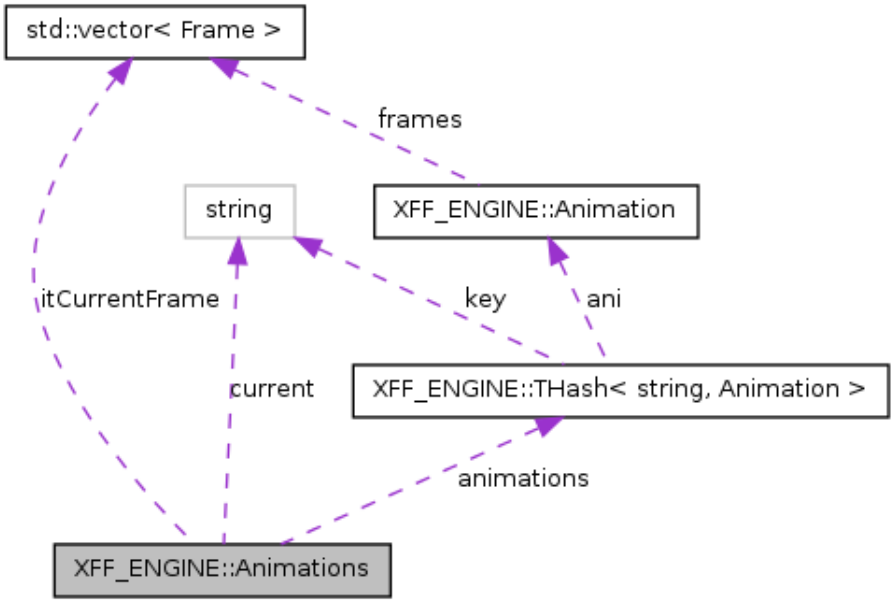


Figura 5.10: Diagrama de colaboración de Animations

La clase TCollisionRect

Estructura que almacena los rectángulos de colisión. Hay que destacar que se ha decidido fusionar las colisiones con las animaciones, debido a la naturaleza de un juego de lucha. En estos tipos de juegos las colisiones son algo fundamental y cada diapositiva, por ello guardar la información de sus rectángulos tantos de sus golpes como de su cuerpo, de manera que sea lo más exacto posible.

- **Atributos públicos:**
  - vector<Rect>bodyBoxes Vector de colisiones del cuerpo.
  - vector<Rect>hitBoxes Vector de colisiones de ataque, representan un golpe.

La clase Point

Representa un punto en un eje cartesiano. Esta clase, aunque aparece aquí por primera vez, es ampliamente usada en todo el simulador. Esta clase provee una forma única de representar un punto en todo el simulador, de manera que se obtiene una abstracción lógica y necesaria para un punto.

- **Atributos. públicos:**
  - int x Eje de coordenadas del eje cartesiano.
  - int y Eje de abscisas del eje cartesiano.

La clase Frame

Representa las propiedades que tiene una diapositiva dentro de una animación. Como se ha explicado anteriormente, no sólo guarda las colisiones, también guarda información para el dibujado. El punto de referencia fundamental para situar en su correcta posición a la diapositiva será explicado con más detalle en el capítulo 6. Cambien guarda la información de giro de una imagen, porque puede ser que una diapositiva exija girar una imagen en un determinado número de grados. Para ello esta clase provee de un atributo flip, cuyo tipo es un enumerado que representa los tipos de giros permitidos.

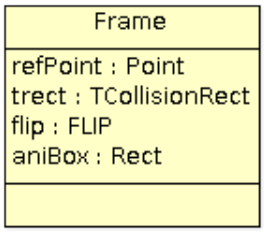


Figura 5.11: Diagrama de clase de Frame

- **Tipos públicos:**
  - FLIP Representa el ángulo de un Frame.
    - N: Normal No gira ningún grado.
    - V: vertical Gira 180 grados.
    - H: horizontal normal Gira 90 grados.
    - VH: vertical y horizontal Gira 180 grados y luego 90 grados.
- **Atributos públicos:**
  - Rect aniBox Rectángulo de la diapositiva dentro de la imagen.
  - TCollisionRect tc_rect Representa las colisiones.
  - FLIP flip Grados que se gira el Frame
  - Point refPoint Punto de referencia.

La clase Animation

Representa un conjunto de diapositivas, es decir, un movimiento. Ha sido diseñada para almacenar instancias de **Frame** en una secuencia, administrando la misma de manera que sirva a una clase que la tenga como agregada, como por ejemplo un proveedor de diapositivas.

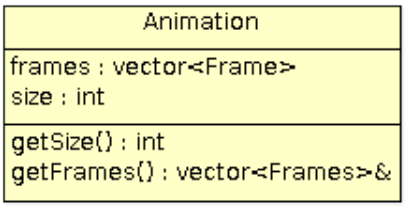


Figura 5.12: Diagrama de clase de Animation

- **Métodos públicos:**
  - vector<Frame>& getFrames () Devuelve una referencia constante al vector de instancias de la clase Frame
  - const int & size () Tamaño de la secuencia de instancias de la clase Frame.
- **Atributos privados:**
  - vector<Frame>frames Secuencia de Frames dentro de una animación.
  - int size Tamaño de la secuencia de instancias de la clase Frame.

La clase Animations

Almacena y maneja un conjunto de animaciones identificándolas por su nombre y las cuales se obtienen desde un fichero. El formato de este fichero, será explicado en apéndice C. Esto se ha diseñado desde el primer momento debido a que es una de las partes más importantes del sistema, porque independientemente de la calidad de los dibujos (que dependen del artista), el sistema garantiza que una imagen pensada para ser animada y con un formato correcto, deberá ser visualizada perfectamente.

Esta clase está pensada para almacenar el conjunto de animaciones o movimientos de una entidad en concreto, con lo cual tiene todos lo métodos necesarios para seleccionar animaciones por su clave, reproducirlas, pararlas o para seleccionar una diapositiva en concreto dentro una secuencia de diapositivas.

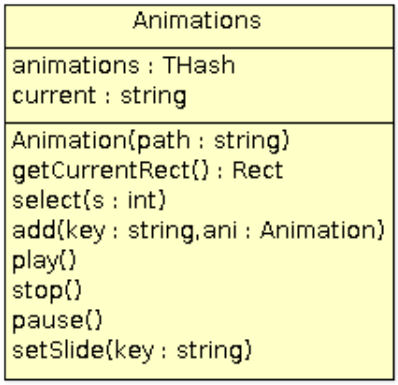


Figura 5.13: Diagrama de clase de Animations

- **Métodos públicos:**
  - Animations(string& path) Constructor que recibe la ruta de un fichero.
  - const Frame & getCurrentFrame () Se obtiene la instancia de la clase Frame seleccionada actualmente.
  - bool setSlide (std::string key) Selecciona una diapositiva por su clave.
  - bool add (std::string key, Animation &) Añade un elemento a la animación.
  - bool play () Se reproduce la animación seleccionada.
  - bool pause () Se pausa la animación seleccionada.
  - bool stop () Se para la animación seleccionada.
  - bool select (size_t i) Se selecciona por el índice un frame dentro de la animación actual.
- **Atributos privados:**
  - THash<string,Animation>animations Diccionario de animaciones donde las claves son cadena de caracteres.

- std::string current Nombre de la actual de la animación seleccionada.

5.3.3. La física

La física del juego, desde el punto de vista de diseño, proporcionan y almacenan información sobre magnitudes y atributos, que luego serán implementadas. Hay que destacar que estas clases son estructuras que almacenan magnitudes para que luego sean manipuladas por algún algoritmo, que cambien el estado de estos objetos. La clase **Positionable** proporciona la propiedad de posicionar un objeto dentro de un eje cartesiano, y convirtiéndose en una clase base para todas aquellas que pertenezcan a un combate o tengan que ser dibujada por pantalla.

La clase **FightPhysicParam** representa una serie de magnitudes físicas que serán muy importantes para proporcionar movimiento a una entidad, estando relacionada con la clase que representa a un luchador (**Fighter**) y que será explicada más adelante.

Estas dos clases forman la base para las clases principales del simulado como muestra la figura 5.14.

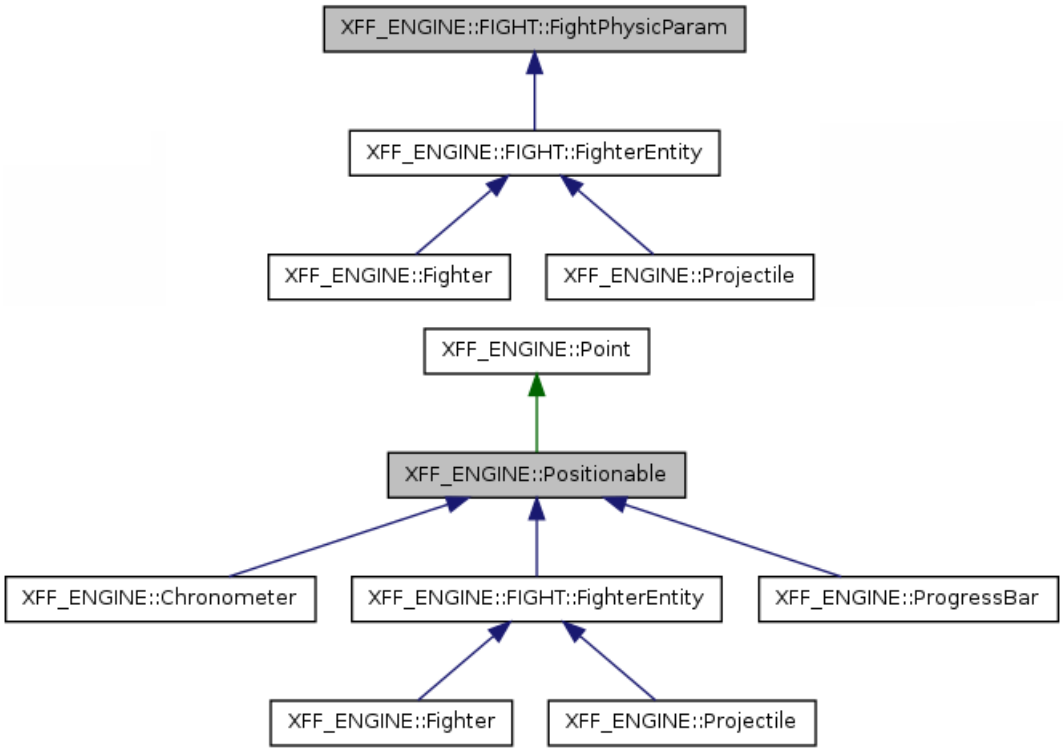


Figura 5.14: Diagrama de herencia de la física

La clase Positionable

Esta clase ha sido diseñada para abstraer el concepto de posicionamiento en un eje cartesiano. Hay que destacar también que es abstracta, debido a que sus especializaciones son las que implementan en particular dos métodos, los cuales definen su altura y su anchura. Éstos han sido

pensados para no sólo posicionarse en la pantalla, sino para tener información del rectángulo en su conjunto.

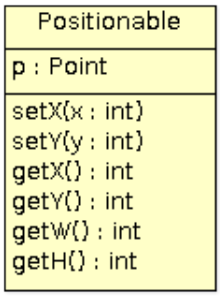


Figura 5.15: Diagrama de clase de Positionable

- **Métodos públicos:**
  - void setX (int px) Asigna un eje de coordenadas.
  - void setY (int py) Asigna un eje de abscisas.
  - const int & getX () Devuelve el eje de coordenadas.
  - const int & getY () Devuelve el eje de abscisas
  - virtual int getW ()=0 Método virtual puro que implementará una clase especializada y que devuelve la anchura.
  - virtual int getH ()=0 Método virtual puro que implementará una clase especializada y que devuelve la altura.
- **Atributos privados:**
  - Point p Representa un punto cartesiano.

La clase FightPhysicParam

Esta clase se encarga de almacenar las magnitudes necesarias para el movimiento en un entorno de dos dimensiones. Para ello tiene métodos para modificar y consultar las velocidades de los ejes y las fuerzas aplicadas a estos ejes. Hay destacar que esta clase sólo se encarga de guardar variables, con lo cual son las clases que se relacionan con ella, las que deben de implementar la física de estas magnitudes.

Esta clase es heredada como se observa en la figura 5.16 por las clases que conformarán un combate, siendo éstas las que aplican sus propias reglas para manejar estas magnitudes.

- **Métodos públicos:**
  - const int getVelocityY () Devuelve la velocidad del eje y.

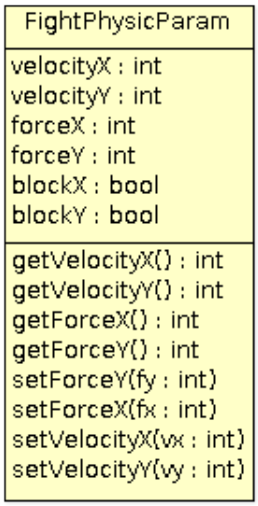


Figura 5.16: Diagrama de clase de FightPhysicParam

- void setVelocityY (int velo) Asigna una velocidad al eje y.
- const int getVelocityX () Devuelve la velocidad del eje x.
- void setVelocityX (int velo) Asigna una velocidad al eje x.
- const int getForceY () Devuelve la la fuerza del eje y.
- void setForceY (int strong) Asigna fuerza al eje y.
- const int getForceX () Devuelve la fuerza del eje x.
- void setForceX (int strong) Asigna una velocidad al eje x.

- **Atributos públicos**
  - bool blockX Bloquea el eje x, no se puede aplicar magnitudes.
  - bool blockY Bloquea el eje y, no puede aplicarse magnitudes.
- **Atributos privados**
  - int velocityX Velocidad del eje x.
  - int velocityY Velocidad del eje y.
  - int forceX Fuerza del eje x.
  - int forceY Fuerza del eje y.

5.3.4. Componentes gráficos

A continuación se van a describir una serie de clases, como se observa en la figura 5.17, cuya finalidad principal en el simulador es la de mostrar imágenes animadas, como es el caso de la clase **Sprite**, **Chronometer** y **ProgressBar**.

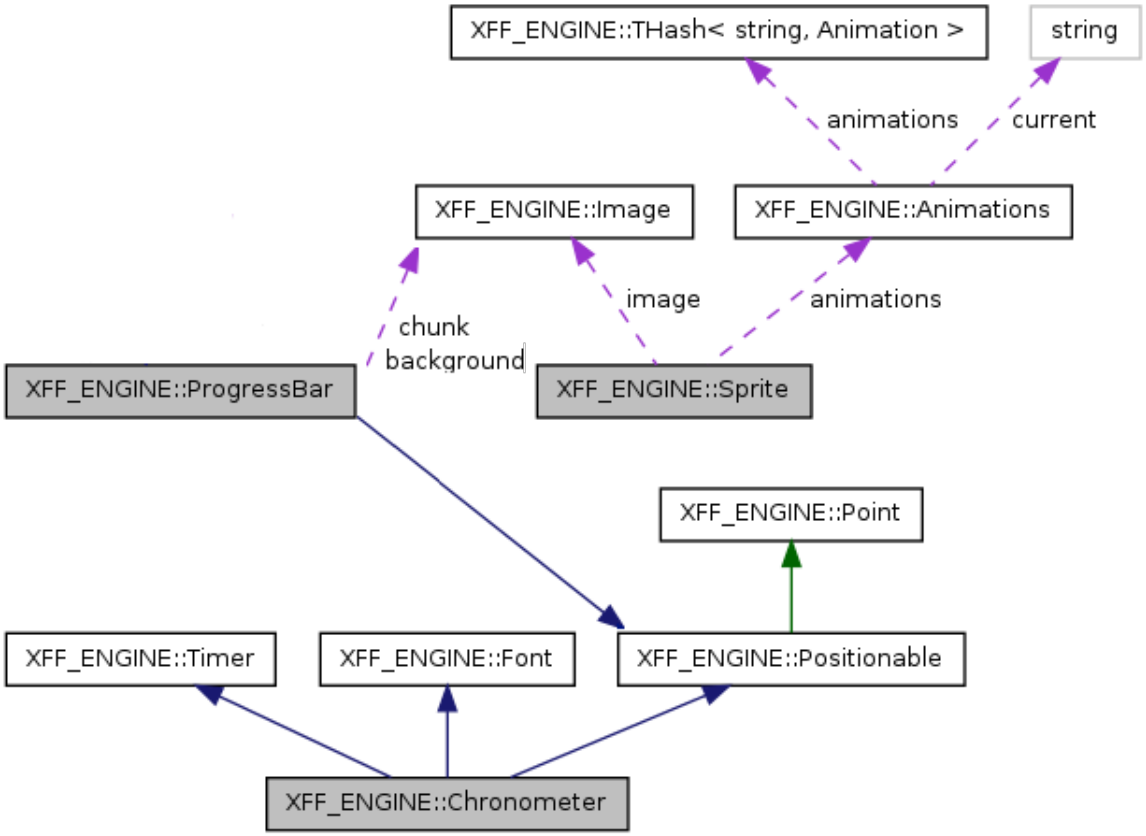


Figura 5.17: Diagrama de colaboración de los gráficos

La clase Sprite

Es utilizada para mostrar animaciones, por ello tiene agregado una animación (clase **Animations**) y una imagen (clase **Image**) como muestra la figura 5.17. Esta clase une en una clase la imagen y sus animaciones, vistas como una clase que guarda los rectángulos de animación y sus colisiones.

Además es utilizada como base para las clases que se dedican a librar un combate. El diseño ha tenido en cuenta la orientación de los ejes, estando muy relacionado con el combate 1vs1, de manera que si la orientación de los ejes cambia la imagen se rotará.

Habrá situaciones, como es el caso de las clases que forman un combate, que deban manipular animaciones directamente, para que esto se pueda realizar correctamente se ha diseñado un método para devuelve una referencia de **Animations**.

- **Métodos públicos:**
  - void setOrientationX (bool ori) Recoge la orientación del eje x.
  - void setOrientationY (bool ori) Recoge la orientación del eje y.
  - bool getOrientationX () Devuelve la orientación del eje x.
  - bool getOrientationY () Devuelve la orientación del eje y.

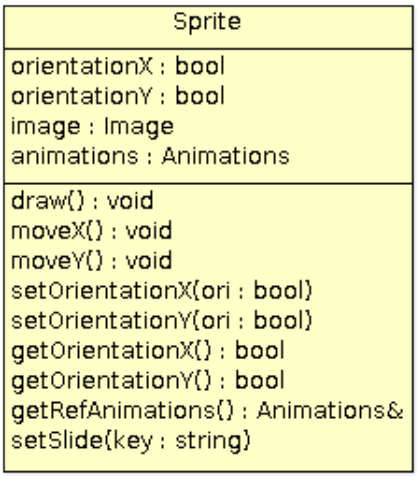


Figura 5.18: Diagrama de clase de Sprite

- Animations & getAnimations () Devuelve las animaciones de una instancia de Sprite.
- void setSlide (std::string &key) Selecciona una diapositiva y automáticamente empieza su reproducción.
- void draw (int x, int y) Dibuja por pantalla al luchador.

- **Atributos privados:**
  - Animations animations Animaciones.
  - Image image Imagen del sprite.
  - bool orientationX Orientación del eje x, si es verdadero es a la derecha y si es falso es a la izquierda.
  - bool orientationY Orientación del eje y, si es verdadero es a la derecha y si es falso es a la izquierda.

La clase Chronometer

Representa un cronómetro dibujado en la pantalla y se utilizará para visualizar una cuenta de tiempo.

Una clase **Chronometer** es un cronometro que es dibujado y representado por pantalla, por ello hereda, como se muestra en la figura 5.17, de las siguientes clases: **Font** (para dibujar el tiempo en forma de números), **Timer** (utilizada para llevar la cuenta de tiempo) y **Positionable** (para dibujarse en pantalla).

Su principal funcionalidad es la de llevar un control de tiempo, ya sea hasta una determinada cantidad o hasta 0 en una cuenta atrás.

- **Tipos públicos:**

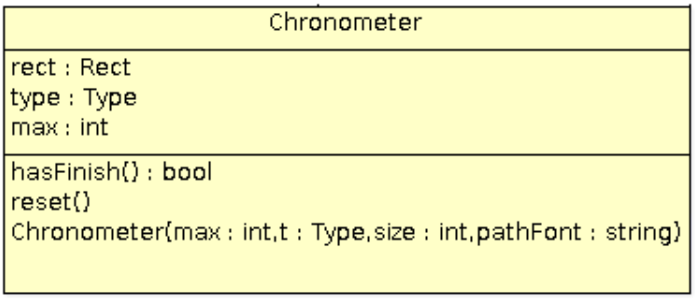


Figura 5.19: Diagrama de clase de Chronometer

- **Type**: Enumerado que representa un tipo de cronometro y cuyos valores son:
  - NORMAL Es una cuenta hacia un valor máximo.
  - INVERSE Es una cuenta atrás desde un valor máximo hasta cero.

- **Métodos públicos:**
  - bool hasFinish () Devuelve si ha terminado el cronómetro.
  - int getW () Devuelve la anchura.
  - int getH () Devuelve la altura.
  - void draw () Dibuja por pantalla el cronómetro.
- **Atributos privados:**
  - unsigned max Máximo valor a contar.
  - Type type Tipo del cronómetro.
  - Rect rect Rectángulo del cronometro en la pantalla.

La clase ProgressBar

Representa una barra de progreso y es usada en el juego para dibujar la barra de vida de los luchadores. Como muestra la figura 5.17, hereda de **Positionable** para obtener la propiedad de situarse en pantalla y además tiene dos imágenes: background (que será la imagen que representará el fondo de la barra) y chunk (que representa el trozo de barra que se va rellenando hasta llegar al límite).

Hay que tener en cuenta que, desde el punto de vista del simulador, esta clase se utiliza para representar la barra de vida de un luchador, cuyos valores están comprendidos entre 0 y 100.

- **Métodos públicos:**
  - void draw () Dibuja por pantalla la barra.
  - void setValue (unsigned v) Recoge un valor para la barra.
  - bool hasFinish () Devuelve si ha finalizado la barra.

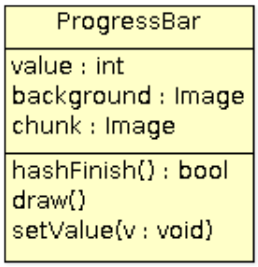


Figura 5.20: Diagrama de clase de ProgressBar

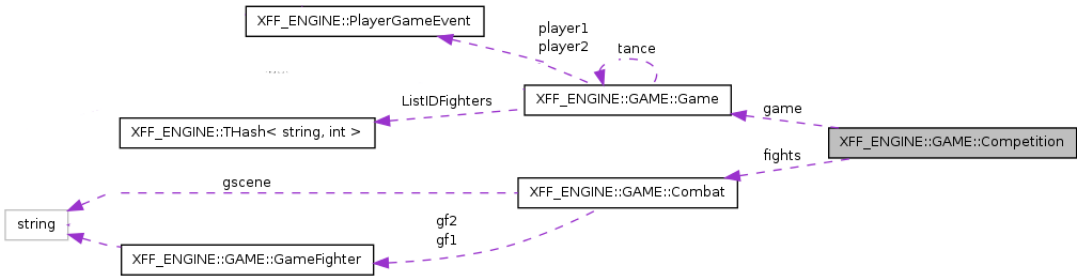


Figura 5.21: Diagrama de colaboración de Competition

- int getH () Devuelve su alchura.
- int getW () Devuelve su anchura.

- **Atributos privados:**
  - Image background Imagen de fondo de la barra.
  - Image chunk Imagen de la barra.
  - unsigned value Valor actual de la barra.
  - unsigned max_value Valor máximo que puede tener la barra.

5.3.5. Las clases que administran los combates

Las clases que administran los combates como muestra la figura 5.21, son aquellas que se encargar de escoger los identificadores de los luchadores y un escenario y con ello formar un combate, como es el caso de la clase **Combat**. La clase **Competition** sin embargo, es la encargada de organizar una secuencia de combates haciendo de clase general para todo tipo de competiciones.

Para conocer los luchadores y los escenarios que existen en el simulador, la clase **Competition** debe de consultar a la **capa de datos**, la cual tiene almacenada esa información en la clase **Game**, como muestra la figura 5.21. Los identificadores de los luchadores y de los escenarios son una cadena de caracteres.

La clase Combat

Esta clase representa un combate entre dos luchadores en un escenario concreto. Es la encargada de registrar el ganador del combate y de una vez que es creada lanzar un combate en su método Combat::exe.

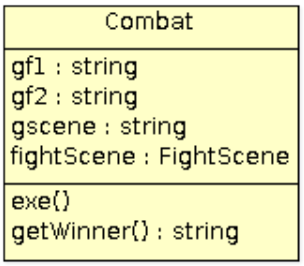


Figura 5.22: Diagrama de clase de Combat

- **Métodos públicos:**
  - bool exe () Ejecuta el combate, devuelve falso si ha terminado.
  - string getWinner() Devuelve el nombre del luchador que ha ganado.
- **Atributos privados:**
  - string gscene Identificador del escenario.
  - GameFighter gf1 Representa al luchador 1.
  - GameFighter gf2 Representa al luchador 2.
  - FightScene* scene Representa la vista del escenario.

La clase Competition

Es definida como una secuencia de uno o más combates, relacionados entre sí, de manera que existe al final de todos ellos un luchador ganador.

Competition es una clase abstracta que pretende modelar el comportamiento de todos sus heredados que son los mismos que la figura 4.2. De manera que todas sus especializaciones implementan los métodos newCombats y createCombats.

Así, toda competición crea unos combates definidos por createCombats y luego se ejecutan. Una vez finalizan se llama a newCombats, que decide qué hacer con los ganadores de esos combates, de forma que toda competición es diferente en esencia.

- **Métodos públicos:**
  - void add (string l1, string l2, string scene) Añade un combate a la secuencia de combates de una competición.

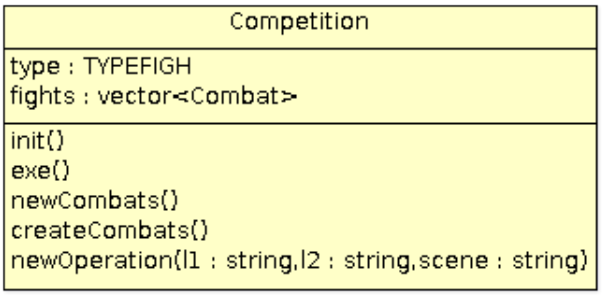


Figura 5.23: Diagrama de clase de Competition

- virtual void newCombats ()=0 Método abstracto que implementa nuevos combates una vez terminados.
- virtual void createCombats ()=0 Método abstracto que representa la creación de combate.
- void init () Inicializa los datos necesarios para una competición.
- void exe () Ejecuta una competición.

- **Atributos privados:**
  - vector<Combat >fights Vector de combate que tiene una competición.
  - TYPEFIGHT tf Tipo de competición.

5.3.6. La máquina de estados

La máquina finita de estados es la encargada de implementar y responsable de llevar a cabo eventos, sean motivados por el usuario o no. La clase **FiniteStateMachine** es una clase paramétrica, la cual proporciona un comportamiento e interacción con el sistema de eventos a la clase que recibe como parámetro. Con esto, se consigue no sólo usar esta clase como puente entre los eventos del luchador y el jugador, sino también modelar el comportamiento de toda clase que maneje eventos o tenga un comportamiento definido, como son los proyectiles, el escenario, el menú principal, etc.

La clase **FiniteStateMachine** como muestra la figura 5.24, es una estructura de datos compleja. En esencia es una enorme grafo compuesto por estados y una cadena de caracteres para identificarlos, esto se consigue mediante su atributo fsm de tipo diccionario. Estos estados son instancias de **FSMState**, el cual a su vez es una secuencia de nodos, representados por la clase **FSMNode**. Hay que destacar que no es una secuencia, si no más bien un grafo con los nodos estáticos, de manera que tiene transiciones entre los nodos mediante condiciones, que en el capítulo 6 se verá que son llamadas al sistema embebido de código.

Cuando se selecciona un estado de la máquina finita de estados, ésta empieza a ejecutar transiciones y tareas entre sus nodos, hasta que ese estado termina llegando a un nodo nulo. Una vez pasa esto, el estado debe ejecutar una serie de condiciones para saber qué estado le sigue. Para

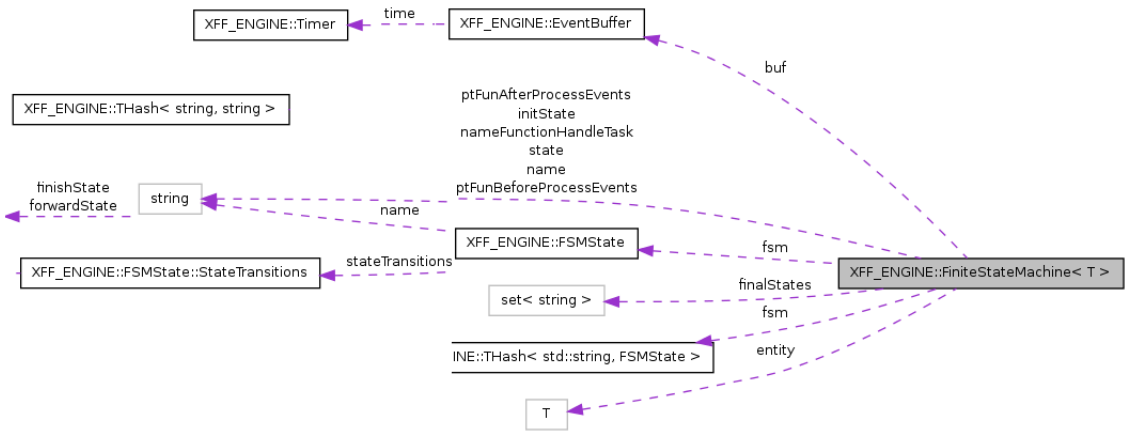


Figura 5.24: Diagrama de colaboración de FiniteStateMachine

ello se utiliza la clase **StateTransitions**, que contiene un diccionario de estados y condiciones de esos estados.

La máquina finita de estados comenzará en un estado concreto y terminará en uno o varios, estos estados determinarán el fin de la ejecución de la máquina finita de estados.

La clase FSMNode

Representa un nodo dentro de **FSMState**, que a vez representa un estado dentro de **FiniteStateMachine**. Un nodo contiene una tarea a ejecutar que más adelante se verá que es una llamada a una función del sistema embebido de código con el parámetro de **FiniteStateMachine** como argumento, y un diccionario de condiciones e identificadores de nodos de **FSMState**.

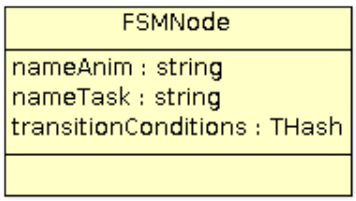


Figura 5.25: Diagrama de clase de FSMNode

- **Atributos públicos:**
  - std::string nameTask Nombre de la tarea a ejecutar.
  - std::string nameAnim Nombre de una diapositiva.
  - THash<Nodo,string>transitionConditions Diccionario con los nodos y el nombre de un estado al que lleva el cumplimiento de una condición.

La clase FSMTransitions

Se encarga de la transición entre distintos estados, hay dos diccionarios, uno que debe comprobarse en cada ejecución de una tarea y si se cumplen, saltar a ese estado, ese diccionario es fowardState; y otro llamado endState, que debe comprobarse cada vez que un estado concluye su ejecución.

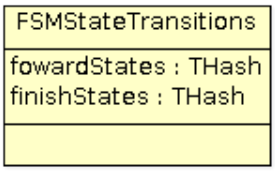


Figura 5.26: Diagrama de clase de FSMStateTransitions

- **Atributos públicos:**
  - THash<string,string>forwardState Diccionario de condiciones.
  - THash<string,string>finishState Diccionario de condiciones.

La clase FSMState

Representa un estado dentro de un Finite State Machine (FSM). De esta clase es importante destacar que es una máquina de estados por sí misma, y que las transiciones de estados las definen las condiciones dentro de sus nodos (que contienen a su vez un diccionario) que son instancias de **FSMNode** contenidos en su secuencia de nodos. Las transiciones entre estados también son gestionadas desde el mismo estado conteniendo identificadores de estado y condiciones a evaluar.

- **Métodos públicos:**
  - FSMState (unsigned tam, std::string &_name) Constructor que recibe el tamaño y el nombre del estado que lo identificará como único.
  - bool isNull () Si el estado ha llegado a su fin.
  - const Nodo & getNode () Nos devuelve el identificador del nodo.
  - void reset () Resetea el Estado volviendo al inicio.
  - void setNode (Nodo n) Seleccionamos un nuevo nodo.
  - const FSMNode & getActualFSMNode () Nos devuelve el nodo actual.
  - FSMNode & at (Nodo n) Acceso directo que devuelve un nodo.
  - const unsigned & getNumStates () Devuelve el número de estados que existen.
  - const string & getName () Devuelve el nombre del estado.

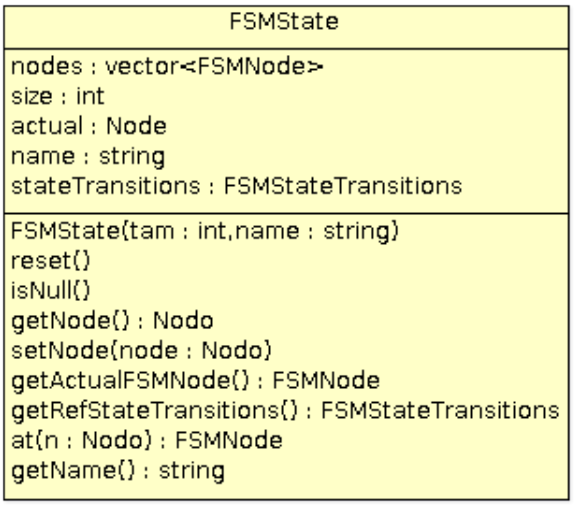


Figura 5.27: Diagrama de clase de FSMState

- StateTransitions & getRefStateTransitions () Devuelve la tranciones entre estados.

- **Atributos privados:**
  - vector<FSMNode>nodes Todos los nodos del estado.
  - std::string name Identificador del estado.
  - unsigned size Numero de nodos, siempre es n+1 siendo 0 un nodo especial considerado como nulo o terminal.
  - Nodo node Nodo actual, si es 0 es que el estado ha terminado.
  - StateTransitions stateTransitions Transiciones entre otras instancias de **FSMState**.

La clase FiniteStateMachine

Es la clase que representa la máquina finita de estados.

- **Métodos públicos:**
  - FiniteStateMachine (T &_entity, string funTask, bool actTran, EventPlayer _player) Constructor que recibe una referencia a una entidad de tipo arbitrario, el nombre de la función Lua que se llamará para modificar a entity, si están activas las transiciones entre estados con actTran, y el manejador de eventos _player.
  - void blockFSM () Bloquea la máquina de estados.
  - bool quitBlockFSM (string st) Desbloquea la máquina de estados.
  - void listenEvents () Escucha los eventos de los dispositivos.
  - const string & getState () Devuelve el estado del autómata actual.

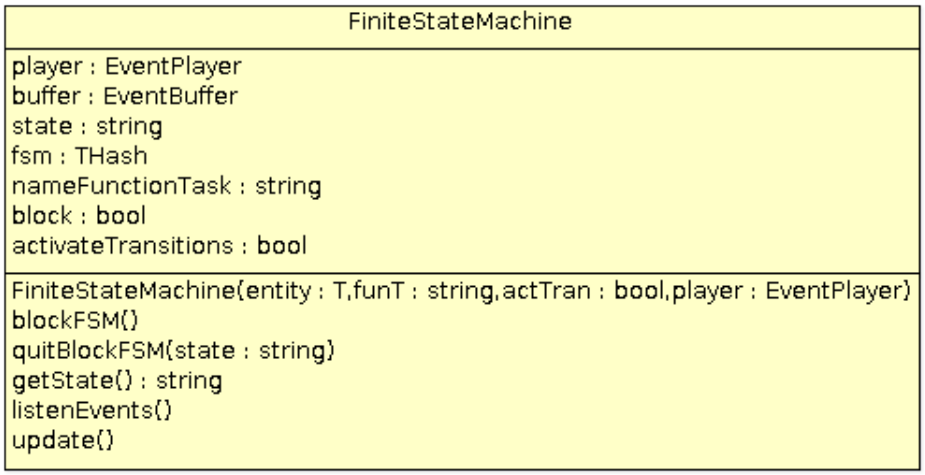


Figura 5.28: Diagrama de clase de FiniteStateMachine

- void update () Se actualiza el estado: escuchan eventos, analiza el búfer se dan respuestas a esos eventos y se cambia el estado de entity cuando un estado así lo requiera.

- **Atributos protegidos:**

  - EventPlayer player Identificador del manejador de eventos, usado en listenEvents para escuchar eventos con el mismo identificado en el búfer global de eventos.
  - EventBuffer buf Búfer de teclas.
  - std::string state Estado actual.
  - THash<std::string, FSMState>fsm Diccionario de estados.
  - T & entity Entidad la cual sera manejada.
  - int recovery Ciclos que ejecuta la misma tarea sin evaluar condiciones.
  - bool activateTransitions Si es verdadero se comprueban las condiciones de transición entre estados.
  - std::string nameFunctionHandleTask Tarea maestra que se llama al ejecutar cada tarea.
  - bool block Si es verdadero la máquina de estados no evaluará ninguna condición.

5.3.7. La lucha

El objetivo del juego es hacer luchar a dos luchadores. Estos luchadores pueden lanzar proyectiles, que daña a ambos luchadores si colisiona con ellos. Esto nos hace que tanto la física como el motor de colisiones quede entre estas dos clases. Sólo existen en el simulador dos instancias de luchador.

La clase que representa a un luchador es **Fighter**, y la clase que representa una magia es **Projectile**. Como ambas clases tienen elementos comunes (magnitudes físicas, animaciones, etc.) ambas son especializaciones de una clase más general que es **FightEntity**. Esta clase engloba las propiedades que tiene que tener un elemento del universo del simulador para poder entablar un combate.

La clase FightEntity

Toda clase del universo del juego que sea capaz de moverse, dañar y ejecutar comportamiento en un combate debe de heredar y cumplir la interfaz de **FightEntity**. Esta incluye dos vectores de rectángulos con las colisiones de los golpes y el cuerpo actualizadas en todo momento respecto de su posición en la pantalla.

Esta clase se ha diseñado como abstracta, teniendo un método que deben ser implementado en sus especializaciones, y representa la actualización del estado del objeto en un combate.

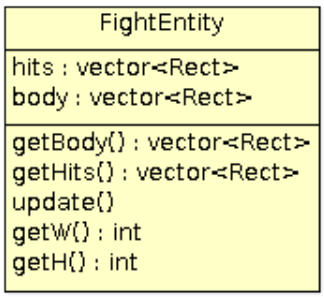


Figura 5.29: Diagrama de clase de FightEntity

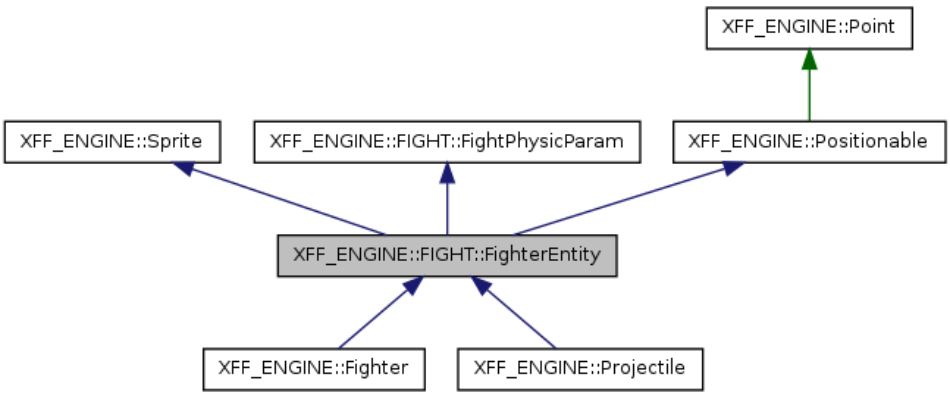


Figura 5.30: Diagrama de herencia de FightEntity

- **Métodos públicos:**
  - virtual void update ()=0 Método abstracto que representa la actualización del estado de la entidad.

- const vector<Rect >& getBody () Devuelve la referencia a la secuencia de rectángulos que representa el cuerpo de la entidad.
- const vector<Rect >& getHits () Devuelve la referencia a la secuencia de rectángulos que representa los golpes que esta ejecutando la entidad.
- int getW () Devuelve la anchura de la entidad.
- int getH () Devuelve la altura de la entidad.

- **Atributos privados:**
  - vector<Rect >Hits Vector de colisiones de los golpes, actualizado a la posición actual.
  - vector<Rect >Body Vector de colisiones del cuerpo, actualizado a la posición actual.

La clase Fighter

Representa a un luchador. Se ha diseñado de manera que contenga los datos necesarios para librar un combate y es la encargada de administrar la física y las colisiones entre los elementos del universo del simulador, así no solo se controlan las colisiones contra su oponente (el cual se guardará mediante su dirección de memoria) sino también las colisiones con los proyectiles existentes.

También implementa la física de un luchador, de manera que es capaz de moverse por la pantalla si detecta un cambio en sus magnitudes físicas; no obstante, esos cambios son motivados por su máquina finita de estados y estos movimientos o cambios de estados, cuyo origen son eventos producidos por el usuario o consecuencia de algún estado.

El luchador tendrá su propio reproductor de sonidos, esto es debido a que un luchador tiene algunos sonidos que son propios, de manera que se diferencia aún más un luchador de otro. Además de poseer la propiedad de lanzar proyectiles al universo del simulador, esto debe ser motivado por una combinación de teclas y será la máquina de estados la que compruebe mediante el diccionario que contiene esta clase si ha lanzado un movimiento especial, si es así, entonces se llama al método throwProjectile de **Fighter**.

Contiene además una manera de mandar y recibir mensajes entre su máquina de estados y la de su oponente. Esto es debido a la relación de apropiación que tienen los luchadores. Cuando un luchador le hace una llave a otro, éste queda bloqueado, el luchador pasa a manejar a su oponente hasta que éste es tirado al suelo y se recupera.

- **Métodos públicos:**
  - void setOpponent (Fighter &opo) Se asigna un oponente al luchador.
  - Fighter & getRefOpponent () Devuelve una referencia a su oponente.
  - const string & getCurrentFsmState () Devuelve el nombre de su estado actual.
  - bool hasDrawPriority () Devuelve verdadero si la condición de ser dibujado antes pertenece al jugador.

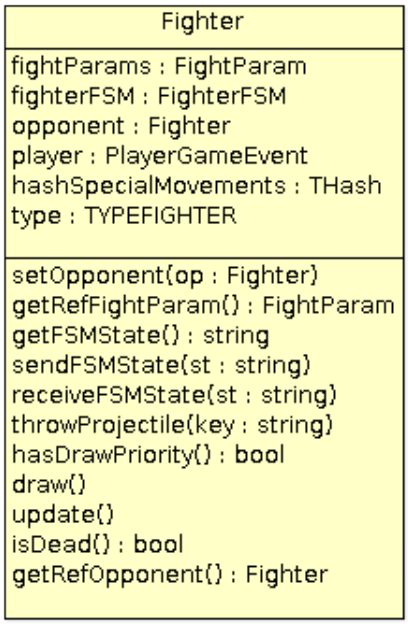


Figura 5.31: Diagrama de clase de Fighter

- void throwProjectile (std::string name) Lanza un proyectil.
- void sendFSMState (string) Envía un mensaje a su oponente.
- void receiveFSMState (string) Lee un mensaje de su oponente.
- bool isDead () Devuelve si el luchador esta muerto.
- void update () Actualiza el estado del luchador.
- void draw () Dibuja al luchador por pantalla.
- FightParams & getRefFightParams () Devuelve los parámetros de la lucha.

- **Atributos privados:**

  - Fighter * opponent Puntero al oponente.
  - SFXPlayer mySample Sonidos del luchador.
  - auto_ptr<FighterFSM >fsm Máquina de estados del luchador.
  - PlayerGameEvent & player Referencia al manejador de eventos.
  - FightParams fightParams Parámetros de la lucha.
  - THash<string,string>hashSpecialMovements Diccionario cuya clave son combinaciones de teclas y clave el identificador del proyectil.
  - TYPEFIGHTER type Tipo de luchador.

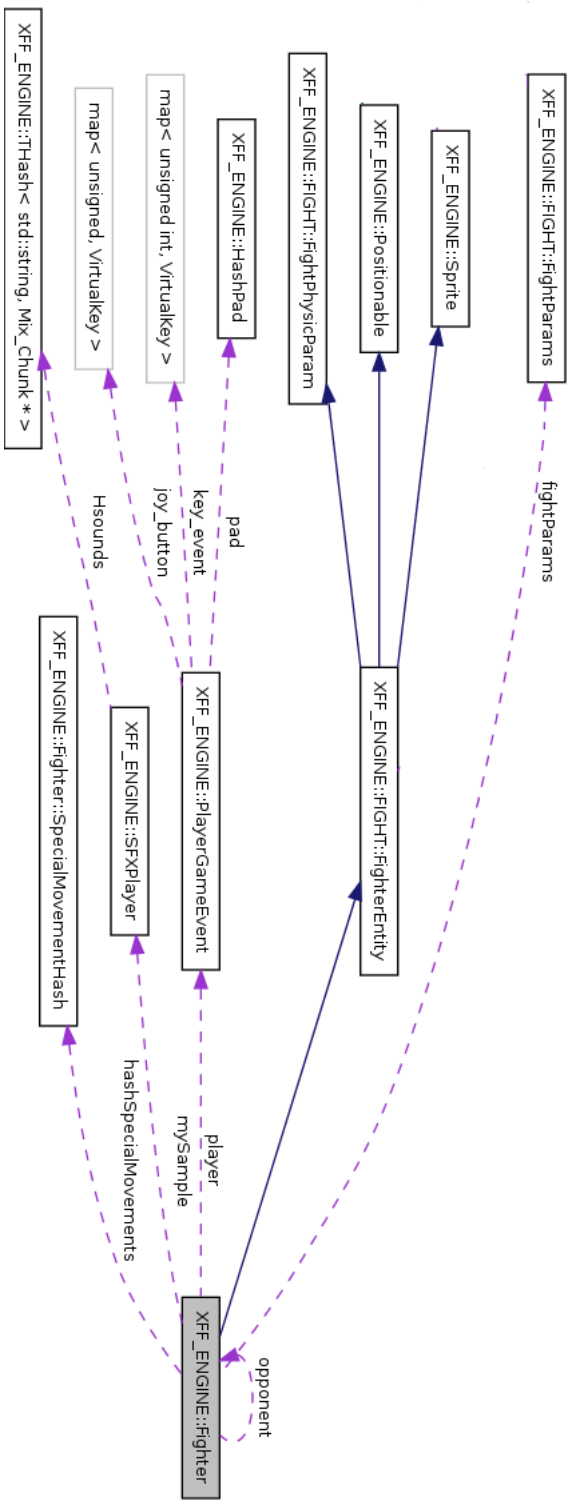


Figura 5.32: Diagrama de colaboración de Fighter

La clase Projectile

Un proyectil o magia es un elemento del universo del simulador lanzado por un luchador, éste a su vez es manejado por un usuario. El proyectil será lanzado cuando este jugador realice una combinación de teclas. Una vez un proyectil es lanzado puede dañar tanto al luchador que lo lanzó como a su oponente y no detiene su curso por la pantalla hasta chocar con un luchador, contra otro proyectil o salir de la pantalla.

En el simulador existe un vector global llamado Projectiles que mantiene todas las instancias de **Projectile** que hay en el universo del juego. La colaboración entre la clase **Projectile** se puede observar en la figura 5.34.

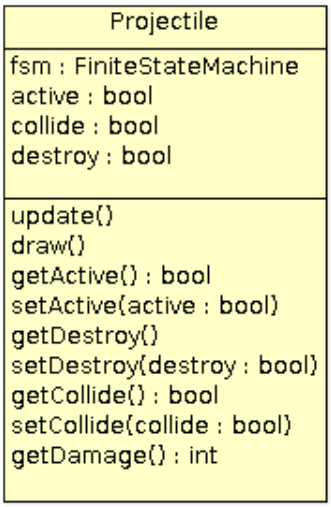


Figura 5.33: Diagrama de clase de Projectile

- **Métodos públicos:**
  - void update () Actualiza la máquina de estados.
  - void draw () Dibuja por pantalla el proyectil.
  - bool getActive () Devuelve si está activado.
  - void setActive (bool b) Asigna una estado al proyectil.

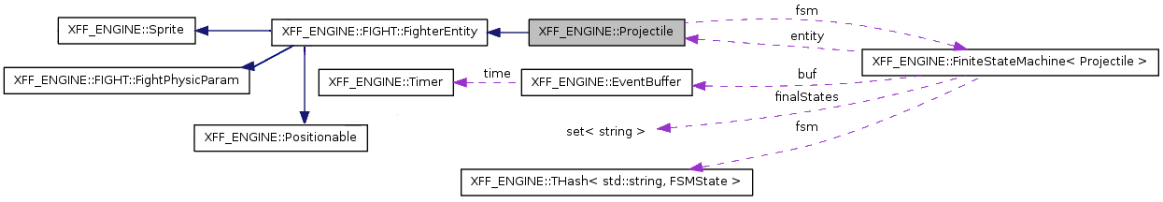


Figura 5.34: Diagrama de colaboración de Projectile

- void setDestroy (bool b) Si se asigna verdadero se destruye el proyectil.
- bool getDestroy () Devuelve si está destruido el proyectil.
- bool getCollision () Devuelve si puede colisionar el proyectil.
- void setCollision (bool b) Si el parámetro puede ser destruido si b es verdadero.
- int getDamage () Devuelve el daño que hace el proyectil.

- **Atributos privados:**

  - bool active Indica si está activo el proyectil.
  - bool destroy Indica si se puede destruir el proyectil.
  - bool collision Indica si puede colisionar el proyectil.

La clase FightParam

Es una clase que solamente utiliza la clase **Fighter**. Esta clase se encarga de guardar todos los atributos de un luchador relacionados con un combate.

Además, es la encargada de administrar los daños con su método setDamage, y es la que se encarga de almacenar en todo momento el daño de los golpes tanto propinados como recibidos; así como el daño acumulado.

También se encarga de almacenar propiedades como los puntos de vida, la defensa y el ataque. Por ello, tanto los puntos de vida que resten a su oponente como los golpes que reciban y dependiendo del valor de esta variables, hacen una cantidad determinada de daño.

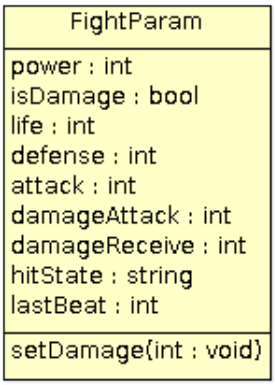


Figura 5.35: Diagrama de clase de FightParam

- **Métodos públicos:**

  - void setDamage (int) Contabiliza un daño recibido.

- **Atributos públicos:**

- unsigned life Puntos de vida de un luchador.
- unsigned attack Puntos de ataque de un luchador.
- unsigned defense Puntos de defensa de un luchador.
- int power Vida o puntos de vida.
- string hitState Nombre del estado que ha dañado.
- bool isDamage Indica si está dañado.
- int lastBeat Daño del último golpe.
- int damageAcumulate Daño acumulado.
- int damageAttack Daño del golpe actual.
- int damageReceive Daño recibido.

5.4. La capa de datos

La capa de datos proporciona la información necesaria a las clases de dominio sobre los luchadores, escenarios, configuraciones de los controles, etc. También se encarga de almacenar parámetros que serán de utilidad en un combate, como el tiempo límite de un asalto o el número de asaltos que debe ganar un luchador.

Esta capa la forman la clase **Game** y la clase **GameFighter** como se muestra en la figura 5.36, utilizando la segunda a la primera para guardar los datos de los luchadores pertenecientes a los distintas esquinas de la pantallas. De esta manera se añade funcionalidad de almacén, de manera que una competición cualquiera consulta las secuencias gameFighter1 y gameFighter2, para saber que combates tiene que formar.

El tratamiento de errores se ha realizado mediante una cadena de caracteres que almacena el error. Siendo clases de la capa de dominio las que lo gestionen.

La razón por la que en el diagrama de colaboración no aparecen las relaciones con las demás capas es debido a que la clase principal de la capa de datos es singleton y, por lo tanto global. Una relación que se observa en la figura 5.36 es la de **Game** y **PlayerGameEvent**, en donde la primera inicializa la segunda, a partir de una serie de ficheros.

5.4.1. La clase GameFighter

La clase **GameFighte**r representa a un luchador en sentido abstracto, es decir, guarda un identificador de éste y su tipo (si es el jugador1, el jugador2 o lo maneja la máquina), de manara que con esta información la clase **Combat** de la capa de dominio puede crear un objeto **Fighter**.

También hay que destacar que se almacena una referencia del manejador de eventos que se utiliza para comunicar la máquina de estados de Fighter cuando Combat cree un objeto está.

- **Atributos públicos:**
  - string keyFighter Identificador del luchador.
  - TYPEFIGHTER type Tipo de luchador.
  - EventPlayer player Identificador del manejador de eventos del luchador.

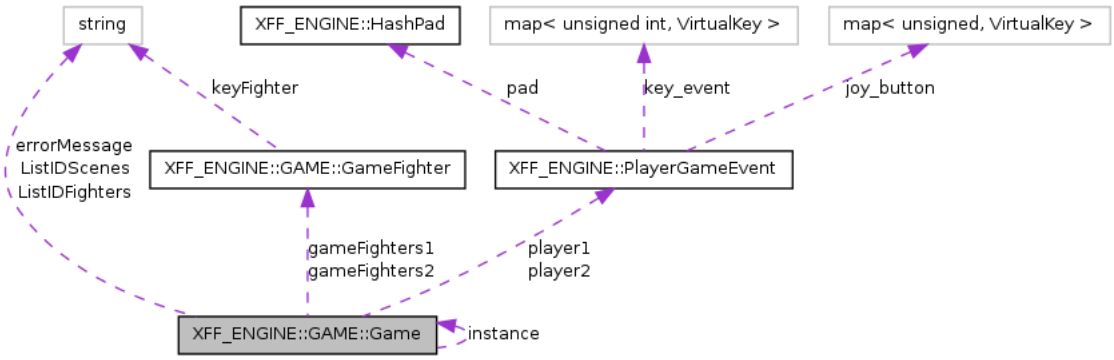


Figura 5.36: Diagrama de colaboración de la capa de datos
rellenados o deberían

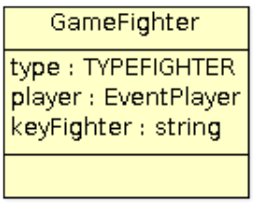


Figura 5.37: Diagrama de clase de GameFighter

5.4.2. La clase Game

Guarda todos los datos comunes del juego: Los luchadores, escenarios cargados en el juego, los manejadores de eventos con los datos de los dispositivos. Es una clase diseñada como singleton debido al continuo uso por las clases de la capa de dominio.

Hay que destacar que existen dos secuencias de **GameFighter**: Una representa el número de luchadores que pertenecen al jugador1 (sea humano o no) y al número de luchadores que pertenecen al jugador2.

En el caso de tener que escuchar todos los eventos de los dispositivos (para tener que configurar un mando o un teclado) se ha diseñado un booleano que interpretado por la función ListenGlobalEvents de la capa de dominio, añadirá al búfer global cualquier evento del teclado o joystick conectado.

- **funciones miembros públicos estáticos:**
  - static Game * getInstance () Devuelve la única instancia de Game.
- **Atributos públicos:**
  - int numRound Número de asaltos que tiene que ganar un luchador para ganar un combate.
  - int timeLimit Tiempo máximo de un asalto.

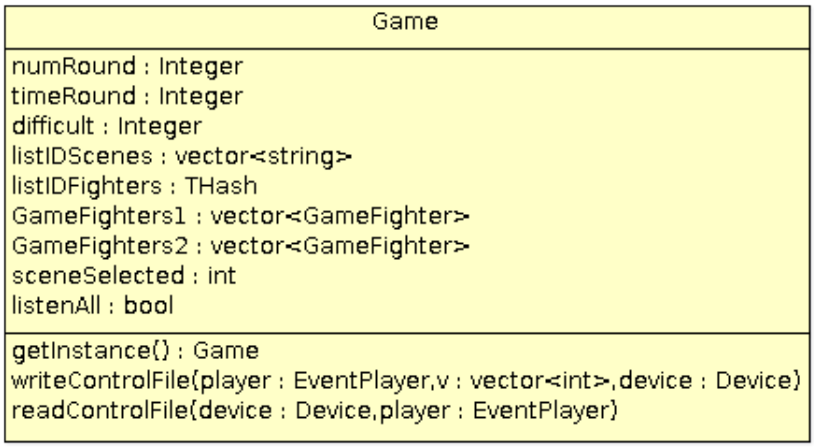


Figura 5.38: Diagrama de clase de Game

- int difficult Dificultad de la máquina.
- std::string errorMessage Detectado en el sistema provoca que se aborte la aplicación y se muestre el error.
- THash<string, int >ListIDFighters Diccionario de luchadores cuya clave son sus nombres y valores el color alpha de su imagen.
- std::vector<string >ListIDScenes Secuencia de identificadores de escenarios.
- std::vector<GameFighter >gameFighters1 Vector de luchadores cargados pertenecientes al jugador1.
- std::vector<GameFighter >gameFighters2 Vector de luchadores cargados pertenecientes al jugador2.
- int sceneSelected Identificador de secuencia del escenario seleccionado.
- PlayerGameEvent player1 Manejador de eventos del jugador1.
- PlayerGameEvent player2 Manejador de eventos del jugador2.
- bool listenAll Si es verdadero se escuchan todos lo eventos de los dispositivos.

- **Métodos públicos:**
  - void writeControlFile (Device device, vector<unsigned int >&v, EventPlayer player) Escribe el fichero de configuración de un dispositivo.
  - void readControlFile (Device device, EventPlayer player) Lee el fichero de configuración de un dispositivo..

5.5. Capa de presentación

La capa de presentación se encarga de visualizar por pantalla los elementos gráficos del sistema y recoger los eventos externos para que puedan ser tratados por la capa de dominio. Toda clase

perteneciente a esta capa debe heredar de **Scene** como se muestra en la figura 5.39.

En el sistema hay tres pantallas bien diferenciadas y que deben ser modeladas por la capa de presentación:

1. **El menú principal:** Donde el usuario elije el tipo de juego que quiere ejecutar o bien entra en un nuevo menú de opciones donde puede cambiar, entre otras cosas, la configuración del juego o los controles del usuario. El menú principal lo implementa la clase **MainMenu**.
2. **La lucha:** Donde se dibujan a los dos luchadores, sus barras de vida, el cronómetro del tiempo y el escenario. La clase que la implementa es **FightScene**.
3. **La elección del luchador:** Donde se dibuja un entorno donde un jugador puede escoger a un luchador partiendo de una lista de luchadores, La clase que se encarga de la elección es **ChooseFighter**.

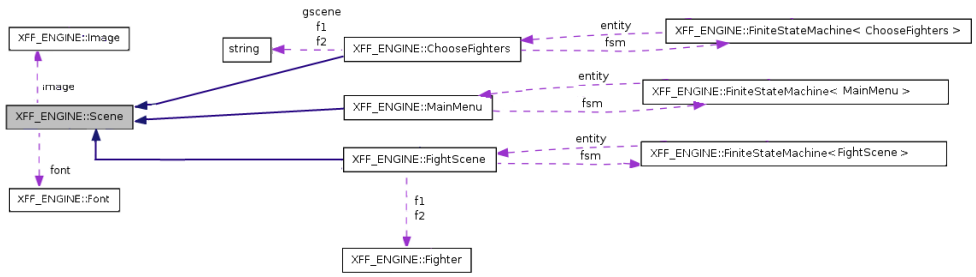


Figura 5.39: Diagrama de colaboración de la capa de presentación

5.5.1. La clase Scene

Su objetivo es hacer de clase base para todas las clases de la clase de presentación. El método abstracto draw debe encargarse de todas las operaciones de dibujado, y el método update debe encargarse de todas las operaciones de lógica como actualizar las máquinas de estados de la capa de dominio. El método hasFinish determinará si la ejecución ha terminado.

- **Métodos públicos:**
  - virtual bool hasFinish ()=0 Indica si ha terminado la ejecución.
  - void exe () Dibuja y actualiza el escenario, llamando a los métodos protegidos y abstractos draw y update.
  - const int & getFPS () Devuelve el ratio de actualización de la pantalla o Frames Per Second (FPS).
  - void setFPS (int f) Asigna un nuevo valor al ratio de actualización de la pantalla o FPS.
- **Funciones miembros protegidas:**

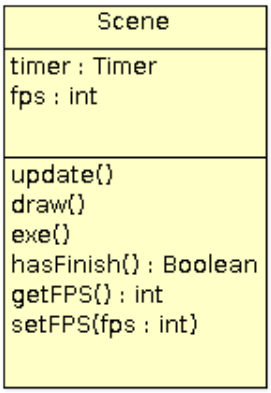


Figura 5.40: Diagrama de clase de Scene

- virtual void update ()=0 Se encarga de actualizar la lógica del escenario.
- virtual void draw ()=0 Se encarga de actualizar el escenario.

- **Atributos privados:**
  - Timer time Temporizador para controlar el ratio de actualización de la pantalla.
  - int time_fps Ratio de actualización de la pantalla, expresados en milisegundos.

5.5.2. La clase MainScene

Esta clase es la encargada de visualizar el menú principal en el que se puede seleccionar el tipo de lucha, además de un menú de opciones.

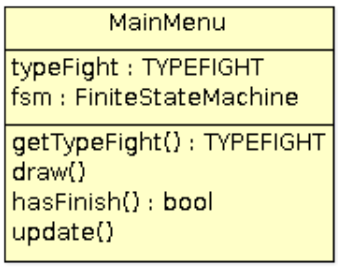


Figura 5.41: Diagrama de clase de MainMenu

- **Métodos públicos:**
  - bool hasFinish () Devuelve verdadero si ha terminado la ejecución.
  - void update () Se encarga de actualizar la lógica del escenario.

- void draw () Se encarga de mostrar por pantalla el menú.
- TYPEFIGHT getTypeFight () Devuelve el tipo de lucha seleccionada.

- **Atributos privados:**
  - TYPEFIGHT tFight Tipo de lucha seleccionada.
  - FiniteStateMachine fsm Máquina de estados.

5.5.3. La clase ChooseFighter

Se encarga de construir un menú en el que un jugador puede seleccionar un luchador.

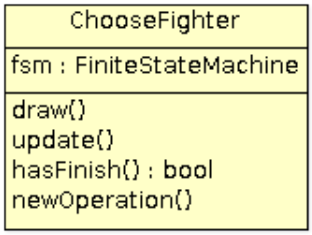


Figura 5.42: Diagrama de clase de ChooseFighter

- **Métodos públicos:**
  - ChooseFighter(TYPEFIGHT tf) Actualiza la lógica.
  - void update () Actualiza la lógica.
  - void draw () Visualiza por pantalla.
  - bool hasFinish () Indica si ha terminado la ejecución.
- **Atributos privados:**
  - FiniteStateMachine fsm Máquina de estados.

5.5.4. La clase FighterScene

Contiene los componentes para visualizar una lucha. Es una de las clases más importantes del juego y está relacionada totalmente con un combate 1vs1. Su método update se encarga de actualizar su máquina de estados y la de los luchadores, además de, cada proyectil que exista en el simulador.

- **Métodos públicos:**
  - void update () Actualiza la lógica.

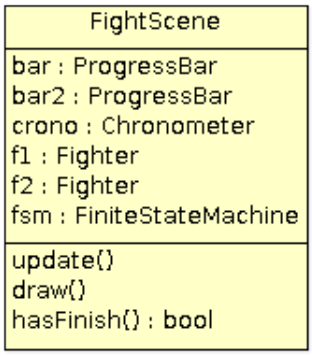


Figura 5.43: Diagrama de clases de FighterScene

- void draw () Visualiza por pantalla.
- bool hasFinish () Indica si ha terminado.

- **Atributos privados:**
  - FiniteStateMachine fsm Máquina de estados.
  - Fighter f1 Luchador que se sitia a la izquierda.
  - Fighter f2 Luchador que se sitúa a la derecha.
  - ProgressBar bar1 Barra de progreso que se sitúa la izquierda.
  - ProgressBar bar2 Barra de progreso que se sitúa la derecha.
  - Chronometer cronos Cronómetro que se situará en medio de las barras de vida.

Capítulo 6

Implementación del sistema

En el capítulo 5 se ha explicado la fase de diseño, dotando de una arquitectura al sistema. En este capítulo, se explican las decisiones que se han tomado frente a ciertos aspectos complejos del desarrollo. A lo largo de las secciones que aparecen a continuación se detallan estas complejidades y la forma de solucionarlas.

6.1. Organización del simulador

El simulador esta pensado para ser ampliable y mantenible. Esto se consigue desde la organización del código fuente hasta la organización de los directorios, de manera que cada elemento del simulador tenga un sitio, una clase o una familia de clases, un fichero fuente, los scripts Lua, etc.

6.1.1. Estructura de directorios

Listado 6.1: Árbol de directorios del simulador

```
.
|-- engine
|-- etc
|-- resource
|-- script
`-- test
```

- **engine:** Contiene los archivos de cabecera y los fuente de C++ que forman el simulador.
- **etc:** Contiene el archivo de configuración del simulador y los controles que configuran los distintos dispositivos de entrada.
- **resource:** Contiene los ficheros multimedia (música, imágenes y sonidos), los action (animaciones), los eff (clave de movimientos especiales de un luchador) y los sfx (configuración de sonidos).

- **script:** Contiene parejas de ficheros fsm (declaración de los estados) y lua (implementación de los estados).
- **test:** Contiene todas la pruebas realizadas en el simulador.

6.1.2. La organización de los ficheros

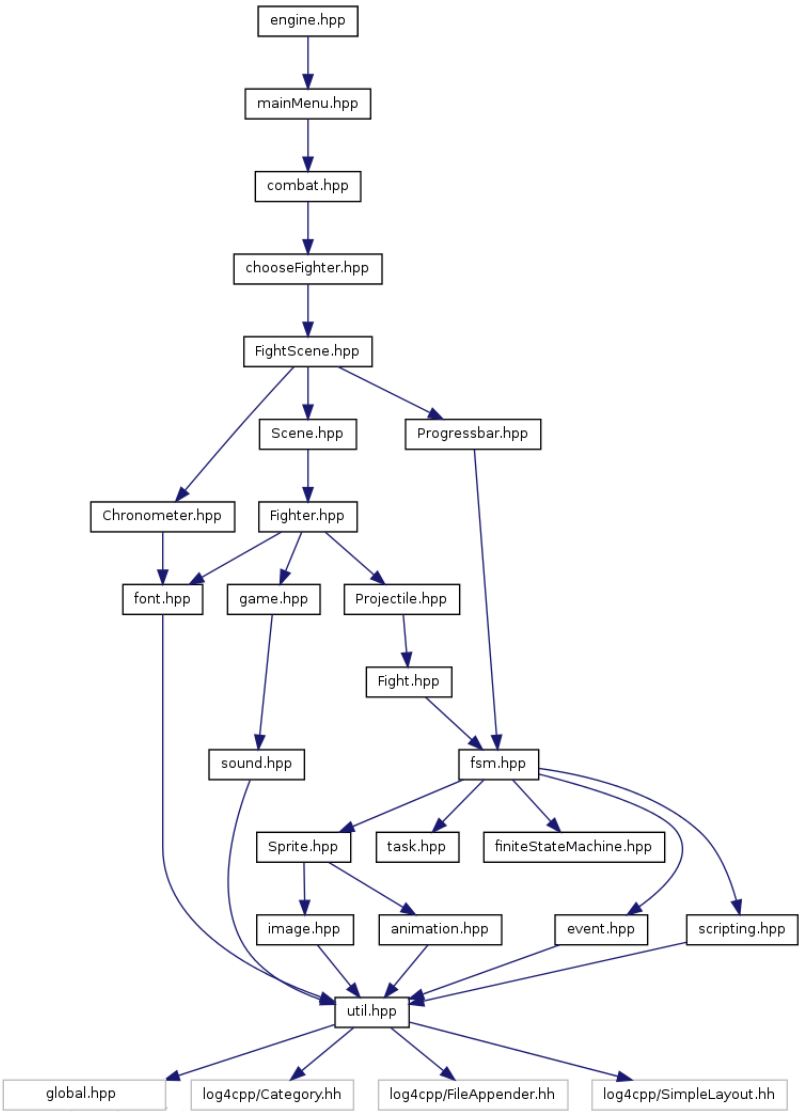


Figura 6.1: Organización de los ficheros del simulador

En *global.hpp* se encuentran los tipos de bajo nivel, clases y enumeraciones más básicas del juego. **GamePlayerEvent** se encuentra en este fichero. En *util.hpp* se encuentran las clases que se utilizarán en el simulador, como la clase paramétrica THash que envuelve un std::map, la clase **Timer** que hace como cronómetro, la clase **Point**, la clase abstracta **Positionable** y la interfaz con el sistema de log.

En *event.hpp* se declaran todas las clases para manejar eventos: **TEvent** y **EventBuffer**, declara la interfaz para manejar el búfer global. En *scripting.hpp* se declaran las funciones para inicializar y llamar a funciones Lua desde código C++. En *sound.hpp* se declaran las clases que manejan el sonido en el juego. En *fsm.hpp* se declara la máquina finita de estados, sus especializaciones, así como todas las estructuras de datos y clases que la hacen funcionar y están declaradas en *FiniteStateMachine.hpp* y *task.hpp*.

En *animation.hpp* se declaran las clases que hacen funcionar las animaciones. En *image.hpp* está declarada la clase Image. En *sprite.hpp* está declarada la clase **Sprite**. En *fight.hpp* se declaran las clases y funciones relacionadas con la lucha y la física del combate como la clase abstracta **FightEntity**, **FightPhysicParam**, **TCollision**; así como las clases encargadas de los efectos de imágenes. En *font.hpp* se declara la clase **Font**, encargada de visualizar texo por pantalla. En *projectile.hpp* se declara la clase **Projectile**.

En *Fighter.hpp* se declara la clase **Fighter**. En los ficheros *Chronometer.hpp*, *progressBar.hpp*, *scene.hpp*, *fightScene.hpp*, *game.hpp*, *mainMenu.hpp*, *chooseFighter* y *combat.hpp*, las clases **Chronometer**, **ProgressBar**, **Scene**, **FightScene**, **Game**, **MainMenu**, **ChooseFighter** y **Combat**. En *competition.hpp* se encuentran la clase abstracta **Competition** así como sus especializaciones. En *engine.hpp* se encuentra declarada la clase que inicia y ejecuta el simulador.

6.1.3. La organización del código

El código de C++ se ha organizado mediante espacio de nombres, agrupando las clases y funciones. Todo el simulador se encuentra bajo el mismo espacio de nombres XFF_ENGINE, existiendo nombre de espacios dentro de éste, que modelan tareas muy concretas. A continuación se detallan:

- **EXCEPTION:** Contiene clases para el manejo de excepciones.
- **FIGHT:** Contiene las clases y funciones relacionadas con la física y los efectos de imagen.
- **GAME:** Contiene las estructuras y tipos básicos para controlar el juego a un nivel funcional.
- **LOGGER:** Contiene un sistema para llevar una traza de código mediante la escritura en un fichero.
- **SCRIPTING:** Organiza el sistema de código embebido.

6.1.4. La carga del sistema

Cuando se inicia el simulador, se carga todas las variables de la clase **Game**, asignándose un valor por defecto. A continuación se carga el fichero de configuración xff.conf y para cada jugador se leerán, si existen, los ficheros que configuran los controles del teclado: player1.key.ctrl y player2.key.ctrl y los del joystick: player1.joy.ctrl y player2.joy.ctrl.

El fichero de configuración

Listado 6.2: Ejemplo de fichero de configuración del simulador

```
[[ GAME ]]
        numRound= 2
        timeLimit= 99
        difficult= 1
        path= ../multimedia/
[[ CONFIG ]]
        printConsole= 1
        debugMode= 1
        printMode= 0
[[ FIGHTERS ]]
        ryu 0xFFFFFFF
        chunli 0x0
        zangief 0xFFF24FF
[[ SCENES ]]
        scene1
        scene2
        scene3
```

En el fichero *xff.conf* se pueden configurar los aspectos globales del juego. El fichero está formado de manera que tiene cuatro secciones divididas por las palabras clave GAME, CONFIG, FIGHTERS y SCENES.

En la primera podemos configurar: El número de asaltos modificando numRounds, la dificultad modificando **difficult** o el tiempo máximo de un asalto con **timeLimit**.

En la segunda se puede configurar los modos de depuración, activándolos con 1 o desactivándolos con 0, con **printConsole** el sistema imprime por pantalla mensajes de depuración, con debugMode se activa un modo gráfico donde la clase Sprite, Fighter y Projectile muestran sus colisiones por pantalla.

En la tercera enumeramos todos los luchadores que existen en nuestro simulador. Por cada luchador deben existir los siguientes ficheros:

- **<nombre_luchador>.png:** Imagen del luchador.
- **<nombre_luchador>.action:** Fichero de configuración de animaciones.
- **<nombre_luchador>.lua:** Fichero con funciones llamadas desde **FiniteStateMachine** que manipulan al luchador en concreto y declara variables concretas para ese luchador.
- **<nombre_luchador>.eff:** Fichero que configura los movimientos especiales.
- **<nombre_luchador>.sfx:** Fichero que indican la ruta de los sonidos del luchador.
- **<nombre_luchador>.fsm:** Fichero con la declaración de los estados y las transiciones de la máquina de estados concretas a ese luchador.

Nota: Los ficheros lua y fsm estarán ubicados en la carpeta script y el resto estarán ubicados en la ruta indica por la variable **path**.

En la cuarta y última se indica el número de escenarios que existen, estos necesitarán de los siguientes ficheros:

- **<nombre_escenario>.png:** Imágenes del escenario.
- **<nombre_escenario>.conf:** Configuración del escenario (posición de sus componentes y animaciones).

Bibliotecas y herramientas utilizadas para la implementación

Todas las herramientas utilizadas para este proyecto son compatibles con otros sistemas operativos. Se ha utilizado el sistema operativo Ubuntu 10.10 y las versiones de todas las bibliotecas y herramientas que se utilizan están ubicadas en los repositorios oficiales de esta distribución de la familia de sistemas operativos GNU/Linux basados en Debian.

Para la implementación del sistema se han utilizado e integrado distintas utilidades para diversos aspectos como el diseño, dibujado, reproducción de sonidos, documentación, etc.

- Compiladores, lenguajes de programación y bibliotecas utilizadas:
  - **Compilador gcc 4.4.5:** Compilador de C/C++ y otro lenguajes producido por el proyecto GNU.
  - **lua 5.1:** Versión del intérprete del lenguaje de programación Lua.
  - **luabind 0.9:** Biblioteca para enlazar Lua y C++.
  - **libSDL 1.2:** Conjunto de bibliotecas para realizar operaciones de dibujado 2D, gestión de efectos de sonido y música, y carga y gestión de imágenes.
  - **libSDLttf 2.0:** Biblioteca para visualizar fuentes tipo TTF.
  - **libSDLmixer 1.2:** Biblioteca para reproducir sonido y música.
  - **libSDLimage 1.2:** Biblioteca para cargar imágenes en formato PNG.
  - **libSDLnet 1.2:** Biblioteca para comunicaciones.
  - **libcppunit:** Biblioteca para las pruebas unitarias.
  - **liblog4cpp5:** Biblioteca para la función de log al sistema.
  - **pdfTeX 3.1415926-1.40.10-2.2:** Utilizado para realizar esta documentación.
- Herramientas para el diseño, documentación y edición de código:
  - **Netbeans 6.9 y su plugin para C/C++:** Entorno de desarrollo utilizado para el simulador.
  - **argoUML 0.3.2:** Herramienta de modelado UML.
  - **Geany 0.19:** Editor de texto utilizado para escribir código en Lua.
  - **TexMaker 2.0:** Editor de Latex.
  - **doxygen 1.7.1:** Herramienta para documentar código en C++.
  - **luadoc 3.0.1:** Herramienta para documentar código en Lua.

6.2. Los fotogramas por segundo

Los fotogramas o FPS es un problema común a todo videojuego y que afecta a la velocidad de ejecución global del programa en su conjunto. La clase encargada de controlar este aspecto es **Scene**.

Para ello usamos el atributo **Scene::time** que se usa para conocer el tiempo transcurrido desde que se inició el método hasta que se terminan las llamadas a Scene::draw y Scene::update. Así se controla si este tiempo es superior a Scene::fps (valor que almacena los fotogramas por segundo), la pausa no se produzca, restándose el tiempo transcurrido del tiempo de actualización de la pantalla. De esta manera controlamos que los fotogramas por segundos sean siempre los mismos independientemente de la máquina.

Listado 6.3: Código fuente del método Scene::exe

```
bool Scene::exe(){
                // ...
        time.stop();//se para el reloj y lo volvemos a reanudar
        time.start();
        fillScreen();//la pantalla se rellena con del color por
           defecto
        listenGlobalEvents(); //escuchamos los eventos
        update();//actualiza la  la escena
        draw();//dibuja la escena
        int time_elapsed = time.getTime();
        if( time_elapsed < time_fps )
          delay( time_fps - time_elapsed   );
        flipScreen();// actualiza la pantalla
        clearBufferGlobalEvents();//limpiamos el buffer global de
           eventos
        // ...
}
```

6.3. El dibujado del luchador

Cuando la vista **FightScene** dibuja un escenario, luego dibuja ordenadamente a los luchadores. El problema se produce cuando un luchador golpea a otro, entonces éste debe dibujarse primero, con lo que se rompe el dibujado ordenado de los luchadores. La prioridad la controla la clase **Fighter** cuando pega un golpe mediante el método FightScene::draw.

Cuando un luchador recoge la prioridad de dibujarse modifica **drawPriority** definido en *fighter.cpp* y de tipo **EventPlayer**, el valor que le asigna cuando la modifica es el valor de su **EventPlayer** que será Player_1 si es el luchador1 o Player_2 si es el luchador2. El luchador modificará esta variable cuando propine un golpe o salte, de esta manera se consigue que nunca un luchador que golpee a otro no se vea a través del ojo humano atravesando a su oponente.

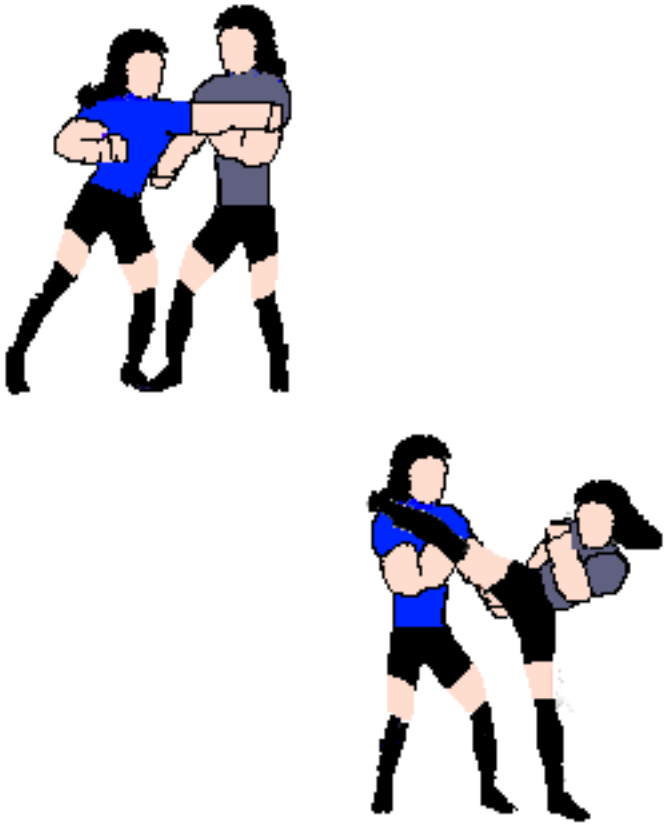


Figura 6.2: Un luchador golpeando a su oponente

6.4. El punto de referencia

La biblioteca que se utiliza para visualizar imágenes usa como punto de referencia, es decir, si tenemos una imagen cualquiera y dibujamos en el punto (0,0) se dibuja en el punto (0,0) del eje de la pantalla, en este caso libSDL dibuja este punto en la esquina superior izquierda. Si se sigue este patrón nos surgen problemas, que se detallan a continuación.

Un problema es la orientación. Si el luchador está orientado hacia la derecha no hay problema, pero si está orientando a la izquierda se dibuja hacia atrás usando la forma de dibujar que tiene la biblioteca por defecto. Una solución sería adelantar al luchador. Con esta solución se consigue golpear a nuestro adversario cuando éste está orientado hacia la izquierda. Pero se debe guardar el valor del suelo respecto a la anchura de cada diapositiva en todo momento y el valor de x que debemos adelantar cuando se golpea y éste está orientado hacia la izquierda.

Pero seguimos teniendo el problema de cuando nuestro luchador este orientado hacia la derecha y su movimiento requiera moverse hacía atrás. Esto se puede solucionar guardando un punto de referencia con el valor de su posición cuando éste orientado hacia la derecha y posteriormente aplicar ese valor para que se vaya hacia atrás.

En conclusión, como las soluciones anteriores requieren guardar valores para casos particula-

res, finalmente se ha implementado la solución solamente guardando un punto de referencia que será guardado en una instancia de **Frame** y será definido por el maquetador en el fichero action de un luchador. Al final, cada diapositiva que representa un movimiento dentro de una animación tiene, por obligación, un punto de referencia como se muestra en la figura 6.3.

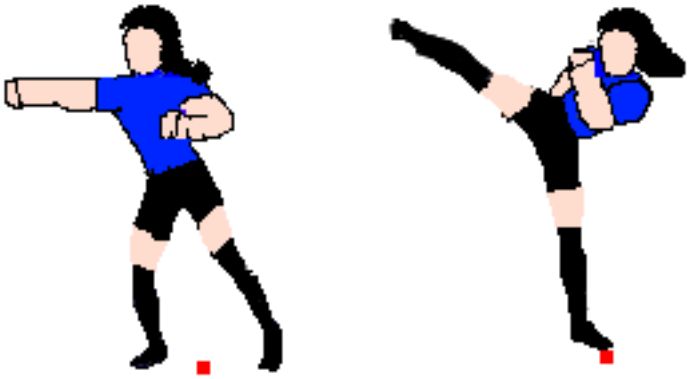


Figura 6.3: Los puntos de referencias de un luchador

La clase **Fighter** es un ejemplo del uso del punto de referencia cuando se dibuja una diapositiva de un luchador, como se muestra en el código 6.4. El método Fighter::draw se encarga de dibujar la diapositiva con su punto de referencia real.

Listado 6.4: Cálculo del punto de dibujado a partir del punto de referencia

```
Point Fighter::getRealPoint() {
        const Frame& frame = animations.getCurrentFrame();
        int move_x =
                (orientationX)
                ? frame.refPoint.x
                : frame.refPoint.x - frame.aniBox.w;
        int move_y = abs(frame.refPoint.y - frame.aniBox.h);
        int realY = abs(getPoint().y - move_y);
        int realX =
                (orientationX)
                ? getPoint().x - abs(move_x)
                : getPoint().x + move_x;
        return Point(realX, realY);
}
```

6.5. Las colisiones y la clase FightEntity

En este simulador cada diapositiva guarda dos vectores de rectángulos, uno con el cuerpo y otro con los golpes, siendo estos utilizados para comprobar las colisiones. Para simplificar este proceso se utiliza la clase **FightEntity**, de manera que toda clase que intervenga en un combate la tome como base.

Las especializaciones de la clase **FightEntity** son responsables de actualizar en cada nueva diapositiva que carguen estos vectores aplicando el punto real. Como se muestra en la figura 6.4, los rectángulos que forman el cuerpo son de color violeta y los rectángulos que forman el golpe son de color rojo.

El punto de referencia y los ejes del luchador deben de ser convertidos, de manera que los vectores de colisiones actualizados contengan los puntos en donde realmente se dibuja la entidad. Hay que tener en cuenta que en el caso del vector de golpes la colisión cuenta a partir del la anchura si está orientado a la izquierda.

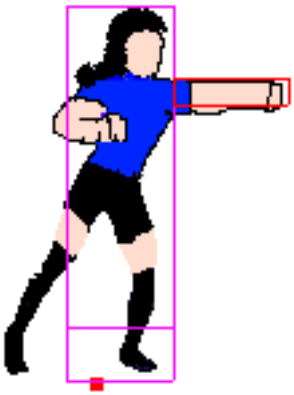


Figura 6.4: Rectángulos de colisión de un luchador

6.6. La colisión entre especializaciones de FightEntity

Las especializaciones de **FightEntity** son **Projectile** y **Fighter**, ambos tienen dos vectores de colisiones como se ha explicado en la sección anterior. Como una entidad está compuesta de una serie de rectángulos organizados en dos tipos de vectores, se ha diseñado un algoritmo para determinar la colisión entre dos entidades. Se recorren ambos conjuntos de rectángulos y se detecta cuando alguno del primer conjunto colisiona con alguno del segundo.

Este algoritmo devuelve una clase **TCollision** que guarda la posición de los vectores rectángulos concretos que han colisionado y un método que devuelve un booleano informando si han colisionado estos vectores y dos enteros que representan las posiciones de los vectores de rectángulos que han colisionado entre sí.

Para comprobar la colisión entre un luchador y su oponente, su posición debe estar actualizada con respecto al dibujado, y después llamar a su método Fighter::hasCollideWithOpponent. Para comprobar las colisiones con los proyectiles existe el método Fighter::hasCollidedWithProjectiles, que se encarga de comprobar colisiones con algún elemento del vector Projectiles.

Listado 6.5: Algoritmo para determinar la colisión entre dos entidades

```
TCollision FIGHT::FightEntityCollisionFightEntity( const vector<Rect
   >& vr1, const vector<Rect>& vr2)
{
    for (int i = 0; i < vr1.size(); i++)
        for (int j = 0; j < vr2.size(); j++)
            if(SDL_CollideBoundingBox(vr1[i], vr2[j]))
                return TCollision(i,j,true);
    return TCollision();//TCollision ilegal
}
```

6.7. La física

La unidad física que se utiliza para separar luchadores y comprobar si se han colisionado con la pantalla es el rectángulo. También se utilizan valores definidos en las clases **FightPhysicParam**, **Positionable** e **Image**. Todas son accesibles desde las instancias de **Fighter**, bien por herencia o por composición. Los proyectiles del simulador también implementan su propia física al heredar de estas clases.

6.7.1. La relación con la pantalla

Existen una serie de reglas que controlan que un luchador no salga nunca del área visible de la pantalla. Estas normas tienen por objeto consultar la posición de los ejes y compararlas con ciertos valores predefinidos.

Se definen dos variables: **Ground**, utilizada para saber dónde se encuentra el suelo, nunca siendo sobrepasado por ningún luchador; y **SCREEN_WIDTH**, definida para saber en todo momento el ancho de la pantalla. Ambas son muy importantes para el correcto funcionamiento de la física del simulador.

La función **lawCornerXFighter(Fighter&)** y **lawCornerWFighter(Fighter&)** definidas en XFF_ENGINE::FIGHT se encarga de comprobar si el luchador ha sobrepasado la pantalla por algunos de sus bordes, la primera comprueba el borde izquierdo y la segunda el borde derecho. De esta manera se controla que ningún luchador sobrepase nunca los bordes de la pantalla.

La función **updateCornerScreen(Fighter& , int)** definida en XFF_ENGINE::FIGHT se encarga de modificar la posición de un luchador, modificando su posición del eje x si ha sobrepasado algún borde de la pantalla, invocando internamente **lawCornerXFighter(Fighter&)** y **lawCornerWFighter(Fighter&)** explicadas anteriormente.

6.7.2. Separación de los luchadores en movimiento

La separación de luchadores ocurre cuando han colisionado sus cuerpos, como se ve en la figura 6.5, almacenados en el vector de rectángulos **FightEntity::Body**. Además necesita como argumento un objeto **TCollision** para determinar qué rectángulo ha colisionado con el luchador y separarlo.

Se observa como en una primera fase uno de los luchadores choca contra su oponente, y posteriormente éste es separado de su oponente de manera que no se pisen las imágenes. El algoritmo que calcula la distancia de esta separación se observa en 6.6.

Hay que destacar que para la separación hay que tener en cuenta los bordes de la pantalla, de esta manera habrá situaciones que la separación conlleve que un luchador empuje a su oponente hacia delante cuando este se mueva y colisione controlándose a su vez los bordes de la pantalla.

Listado 6.6: Código para separar a dos luchadores si sus fotogramas se pisan

```
int FIGHT::LawSeparateIfCollitionMovewithOpponent(
                Point p,
                const vector<Rect>& vr,
                Point op,
                const vector<Rect>& vr2,
                TCollision& tcol,
                bool oriX
) {
    if (oriX)
        p.x += vr.at(tcol.pos).w;
    else
        op.x += vr2.at(tcol.opPos).w;
    return CalculateDistance(p, op);
}
```

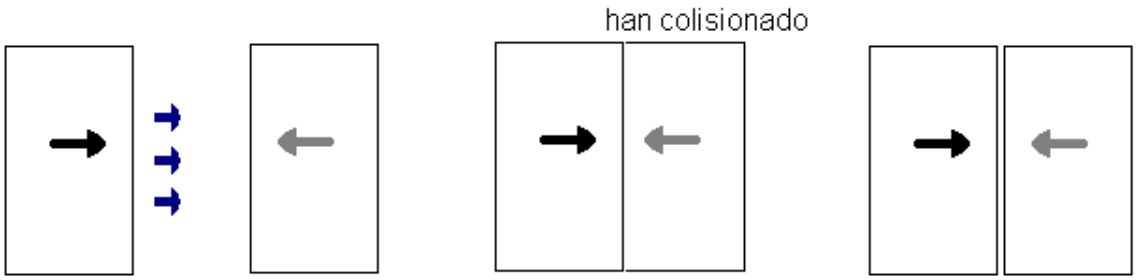


Figura 6.5: Ilustración de la separación de un luchador

6.7.3. Separación de los luchadores en el aire

Cuando un luchador salta, su rectángulo y su eje se decrementa. Esto ocurre mientras Fighter::getVelocityY() >0 y Fighter::getForceY() >0. Fighter::update se encarga de que se incremente su eje y hasta que sea igual a la constante Ground.

Como se muestra en la figura 6.6, la imagen está divida en dos partes: En la primera, el luchador representado por el rectángulo de la derecha cae pero no supera el umbral, con lo cual modifica la posición del eje del oponente; y en la segunda parte, si supera ese umbral, como se muestra la figura en su parte inferior se modifica la posición del eje x del luchador.

La implementación de estos umbrales, determinan si un luchador debe de ser separado de su oponente para que no pise al dibujo de su oponente, en el caso de superar ese umbral. Si no

lo supera, entonces saltar y cae normalmente siendo esta vez su oponente empujado de manera que, al igual que en el caso anterior, no se pisen tampoco sus dibujos.

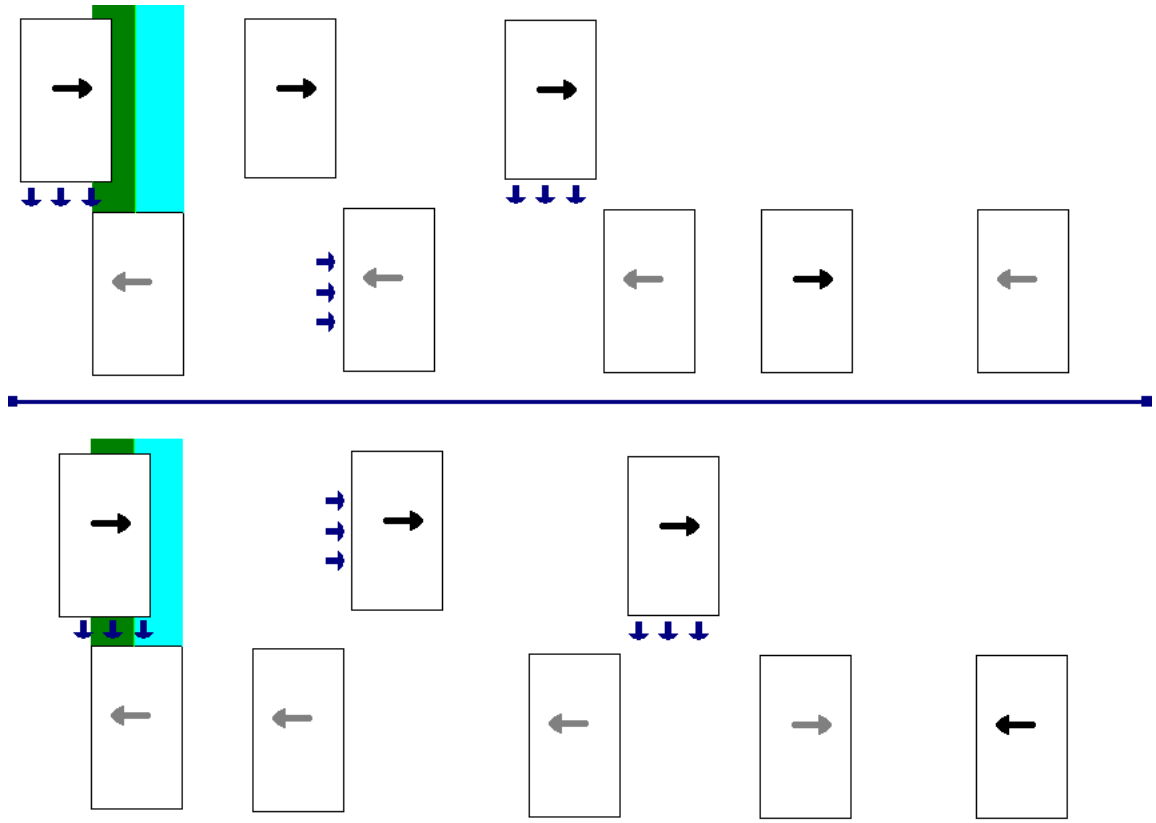


Figura 6.6: Ilustración de la caída de un luchador

6.8. El sistema embebido

A medida que se fue desarrollando el sistema se tomó la decisión de utilizar un sistema de código embebido para proporcionar funcionalidad a la máquina finita de estados, de manera que los estados estén declarados en un fichero de texto de formato fsm, que relacionan los estados y las transiciones entre estados con funciones escritas en Lua llamadas por la clase **FiniteStateMachine** como se observa en la figura 6.7.

Esta solución fue implementada incrementalmente: Primero se pensó implementar el sistema embebido para que sólo diera soporte a la clase **Fighter**, luego en siguientes iteraciones se añadió el proyectil, luego el escenario y por último se generalizó la máquina finita de estados para una clase arbitraria. Siempre y cuando esta clase cumpla una serie de requisitos, entre ellos y el más importante, que pueda ser llamada desde el sistema de código embebido. Para ello solamente hay seguir una serie de pasos que son explicados con más detalle en el apéndice A.

Se ha generado código fuente en C++ y Lua. Un script Lua es un código externo al simulador, de esta manera se amplia el juego incluyendo nuevos luchadores. El simulador ha portado ciertas

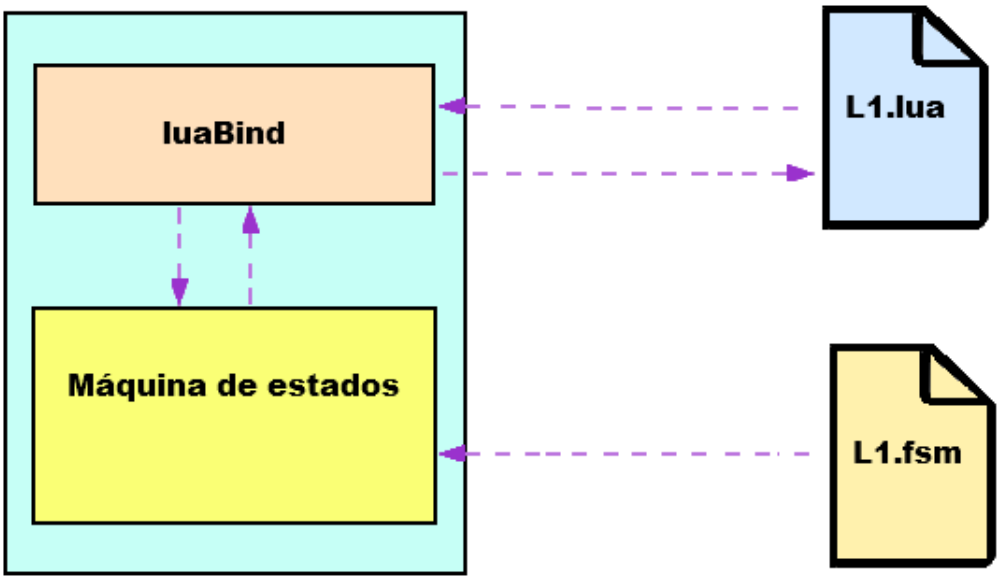


Figura 6.7: Arquitectura del sistema embebido

clases a Lua, de manera que se pueden crear/eliminar/modificar objetos desde este lenguaje y pasarlos a C++ y viceversa. El diseño de la máquina de estados permite por lo tanto, ejecutar código embebido de las clases del simulador que han sido portadas a Lua y que pueden ser accedidas desde cualquier script que se interprete. Las clases que cumplen esta portabilidad se encuentran en *engine/scripting.cpp*.

Como se observa en la figura 6.7, el sistema embebido está representado por luaBind que está implementado en el simulador en *engine/scripting.hpp* y *engine/scripting.cpp*, soporta tanto las llamadas a la biblioteca libluabind como las clases de C++ preparadas para ser llamadas desde Lua. Las clases que han sido portadas a C++, pueden ser creadas, modificadas y devueltas desde cualquier script Lua que se ejecute en el simulador.

También se observa la máquina de estados que es representada dentro del simulador por la clase **FiniteStateMachine**, y se encarga de cargar el fichero fsm con el que extrae información acerca de sus estados. Esta clase se comunica con luaBind para llamar a funciones Lua que modifiquen objetos y devuelvan objetos a petición de la máquina de estados.

La apropiación de un luchador

El diseño de la máquina de estados no está, en principio, diseñada para un paso de mensajes. Cuando un luchador realiza una llave a su oponente, hay una apropiación de comportamiento hacia éste, bloqueando la respuesta hacia eventos externos y congelando la ejecución de su estado, pasando éste a ser controlado por la máquina de estados de su oponente.

Este comportamiento de la máquina de estados se ha conseguido añadiendo un booleano FiniteStateMachine<T>::blockFSM que indica si está bloqueada. En consecuencia, si está a verdadero, cuando se llame a FiniteStateMachine<T>::update no se actualiza. Para quitar el bloqueo

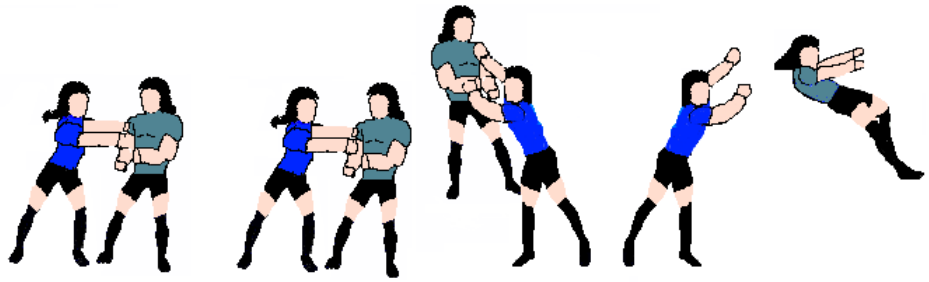


Figura 6.8: Un luchador realiza una llave a su oponente

se llama a FiniteStateMachine<T>::quitBlock(string) que carga el estado indicado en el parámetro del método.

La clase **Fighter**, puede enviar un estado a su oponente mediante Fighter::sendFSMState. Cuando su oponente reciba este estado, representado por una cadena de caracteres, bloqueará su máquina de estados si no es un estado vacío y en caso contrario le envía un estado (el identificador de un estado es siempre una cadena de caracteres) y su máquina de estados se desbloquea y comienza la ejecución.

La implementación de este sistema utiliza características de Lua como lenguaje. Cuando FiniteStateMachine<T>::update es llamada, esta a su vez, llama a otra función que selecciona un nodo de vector de instancias de **FSMNode**. Una vez pasa esto, ese estado seleccionado es pasado como argumento a una función Lua junto con la instancia del objeto que se quiera manipular, en este caso un objeto de la clase **Fighter**.

En Lua todas las funciones pueden ser nombradas mediante una tabla global llamada **_G**. Toda función que necesite apropiarse de un luchador deberá comenzar por "op" seguido del nombre de la función. Las funciones de este estado se ejecutan paralelas a otras llamadas con el mismo nombre pero que empiezan por "Modify". De esta manera se bloquea al oponente y la máquina de estados del luchador es la que se encarga de ejecutar estados, no sólo del luchador, también del oponente.

Si por ejemplo una instancia de FSMNode tiene como tarea asociada opllave2, la función que gestiona al luchador (**FighterParamTask**) ejecuta Modifyopllave2 que modifica el estado del oponente, de manera automática.

6.9. El combate

El objetivo principal del simulador es implementar un combate, para ello se tiene en cuenta que éste está compuesto por una serie de asaltos. El luchador que lo gane, es aquel que vence a su oponente un número de asaltos igual al definido en la variable Game::numRounds.

Un luchador gana un asalto si acaba con la barra de vida de su oponente definida en Fighter::fightParams::power, o si una vez se acaba el tiempo del asalto definido en Game::timeLimit, su barra de vida es mayor a la de su oponente, en otro caso el asalto se queda en tablas.

La clase que se encarga de implementar un combate es **Combat** y la vista que se encarga de gestionar el dibujado y la recogida de eventos es **FightScene**, donde una máquina de estados

es la encargada de gestionar el combate, implementando toda la lógica.

6.9.1. Los estados globales

Los estados globales definen las reglas del combate. Existen cuatro clasificaciones: Los estados de movimiento, los de golpes, los de ser golpeados y los especiales. Los estados están declarados en el fichero **engine/global.fsm**, a excepción de los especiales que son únicos de cada luchador.

Los movimientos:

Los estados de movimiento son aquellos que son activados por las teclas de movimiento (arriba, abajo, izquierda y derecha) y como se observa en la figura 6.9 hacen referencia a saltos, agacharse, moverse, etc.

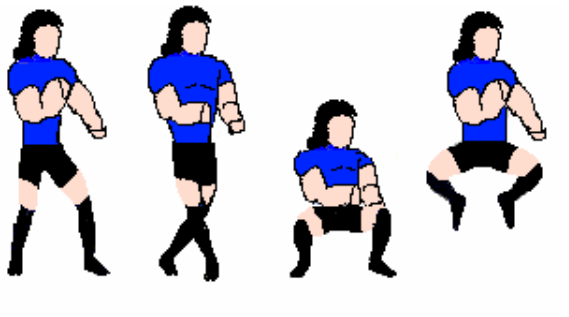


Figura 6.9: Luchador en movimiento

- **QUIT:** Es el estado que representa estar quieto.
- **MOVEFORW:** Es el estado que representa moverse hacia la derecha.
- **MOVEBACK:** Es el estado que representa moverse hacia la izquierda.
- **CROUCH:** Es el estado que representa agacharse.
- **JUMP:** Es el estado que representa saltar.
- **JUMPMOVEFORW:** Es el estado que representa saltar y moverse hacia la derecha.
- **JUMPMOVEBACK:** Es el estado que representa saltar y moverse hacia la izquierda.

Los golpes:

Son estados que representan los golpes que pueden dañar al oponente si colisiona con éstos, como se observa en la figura 6.10. Si el luchador está avanzando o quieto, saltando, agachado o saltando y avanzando, en total forman cuatro formas de golpear. Como hay tres tipos de golpes, ordenados en velocidad y fuerza (débil, medio y fuerte) y dos tipos de golpe (puñetazo y patada), en total forman veinticuatro estados.

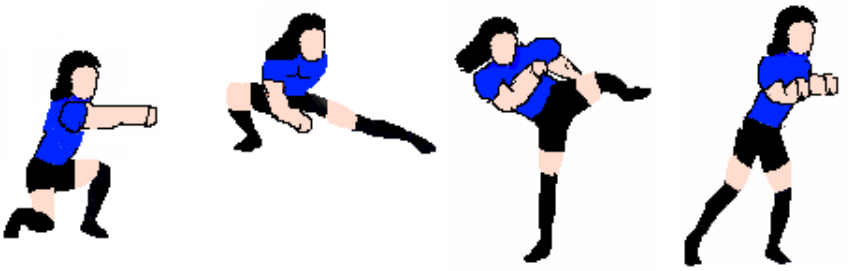


Figura 6.10: Luchador propinando golpes

Recibir golpes:

Estos estados son el reflejo de golpes que colisionan con un luchador y provocan diferentes efectos:

- **BEATEN_QUIT:** Es golpeado mientras está moviéndose o quieto.
- **BEATEN_JUMP:** Es golpeado mientras está saltando.
- **BEATEN_CROUCH:** Es golpeado mientras está agachado.
- **KO:** Cuando hay muchas acumulaciones de golpes recibidos seguidas en un intervalo de tiempo, el luchador entra en un estado que no se puede mover durante varios segundos, pudiendo ser golpeado de nuevo.
- **FALLEN:** El golpe ha sido muy fuerte y el luchador cae al suelo.
- **DEAD:** El golpe ha acabado con la barra de vida del luchador.
- **PROTECT_QUIT:** El luchador se ha protegido de un golpe mientas estaba quieto o moviéndose y no recibe el golpe.
- **PROTECT_CROUCH:** El luchador se ha protegido de un golpe mientras estaba agachado y no recibe el golpe.

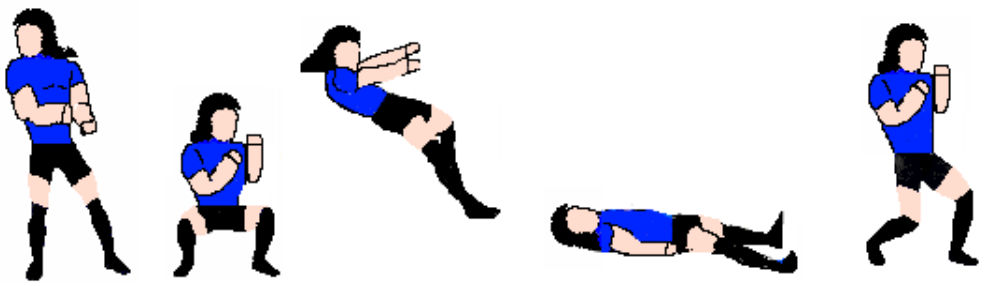


Figura 6.11: Luchador siendo golpeado.

Cuando un golpe es contabilizado, quita parte de la barra o puntos de vida, se llama a la función FightParams::setDamage. Para saber si un luchador ha entrado en el estado de KO, se los intervalos de tiempo en el que ha recibido golpes, usando una instancia de **Timer** global definido en *engine/event.cpp*.

Estados especiales

Son aquellos estados únicos de cada luchador y normalmente están divididos en magias (proyectiles) y movimientos especiales (movimientos de artes marciales muy exagerados). Ambos están determinados por la intensidad del golpe que los activen junto con la combinación de teclas; de manera que una magia lanzada con un puñetazo flojo lanza una magia lenta, y una lanzada con un puñetazo fuerte lanza una magia más rápida. Lo mismo pasa con los movimientos especiales.

Figura 6.12: Luchador que se encuentra en un estado especial, lanzando una proyectil

6.9.2. Las variables especiales del luchador

El luchador tiene una serie de variables especiales para que sus estados de movimientos o golpes sean característicos. Éstas deben estar definidas en un script Lua con el mismo nombre del luchador todo en minúsculas y siguen la siguiente plantilla:

- **<nombre_luchador>Life:** Define el nivel de vida que tiene el luchador que se ponderará sobre cien.
- **<nombre_luchador>Attack:** Nivel de ataque, el daño que hace los golpes del luchador ponderados al nivel de puntos que quita un golpe.
- **<nombre_luchador>Deffense:** Nivel de defensa, el daño que recibe los golpes del luchador, una vez se haya protegido de un golpe o hay sido dañado.
- **<nombre_luchador>WalkForw:** Instancia de **Movement** que define el movimiento cuando el luchador camina hacia adelante.

- **<nombre_luchador>WalkForw:** Instancia de **Movement** que define el movimiento cuando el luchador camina hacia detrás.

- **<nombre_luchador>Jump:** Instancia de **Movement** que define el movimiento cuando el luchador salta.

- **<nombre_luchador>JumpForw:** Instancia de **Movement** que define el movimiento cuando el luchador salta hacia adelante.

- **<nombre_luchador>JumpForw:** Instancia de **Movement** que define el movimiento cuando el luchador salta hacia atrás.

Listado 6.7: Ejemplo de script Lua para un luchador llamado Ryu

```
require "script.global"
--[[
        definicion de las variables:
--]]
--nombre del luchador
Ryuname = "Ryu";
--parametros fisicos
RyuLife = 1000
RyuAttack = 100
RyuDefense = 100
--movimientos
RyuWalkForw = Movement(20,20,0,0)
RyuWalkBack = Movement(20,-20,0,0)
RyuJump = Movement(0,0,180,-40)
RyuJumpForw = Movement(180,20,220,-50)
RyuJumpBack = Movement(120,-20,220,-50)
```

La clase **Movement** representa el movimiento de un luchador y se utiliza para la variables especiales de un luchador. Además define un movimiento del eje x e y, definiendo para cada uno de estos una fuerza y una velocidad.

Listado 6.8: La clase Movement

```
struct Movement {
        int forceX;
        int forceY;
        int velX;
        int velY;
        Movement(int fx = 0, int vx=0, int fy=0, int vy=0):
                forceX(fx), forceY(fy), velX(vx), velY(vy) { }
};
```

6.9.3. Los estados especiales y el lanzamiento de proyectiles

Un proyectil es lanzado por un luchador, esto significa que un objeto de la clase **Projectile** es añadido a un vector global llamado Projectiles y que **FightScene** se encarga de gestionar.

Tanto un proyectil como un movimiento especial son lanzados mediante combinaciones de teclas, esto es algo muy común en un juego de lucha 1vs1. Para conseguir esto se usa la clase **TSpecialMovement** definida en *projectile.hpp*. Esta clase almacena el tipo de movimiento especial en TSpecialMovement::Magic (si es un proyectil o no) y la condición que lo activa en TSpecialMovement::condition (una función Lua que evalúa alguna condición de estado de una instancia de **Fighter**). Almacena también dos enteros (TSpecialMovement::x, TSpecialMovement::y) que representan la posición en la que se lanza el proyectil con respecto al luchador, así como la velocidad (TSpecialMovement::velX, TSpecialMovement::velY). En el caso que no sea un proyectil (un movimiento especial), se omiten estos parámetros.

Listado 6.9: La clase TSpecialMovement

```
struct TSpecialMovement {
        bool Magic;
        int x, y,velX, velY;
        string name;
        string condition;
};
```

La clase **Fighter** está compuesta de una instancia de **SpecialMovementHash** que se encarga de almacenar la combinación de teclas y el nombre del movimiento especial o del proyectil y coinciden con estados definidos en <nombre_luchador>.lua y cuyo nombre es <nombre_magia_o_movimiento_especial>, todo en mayúsculas.

Para comprobar que se ha lanzado una combinación de teclas se llama a la función Lua hasThrowSpecialMovements, definida en *script/damageFighterConditionsAndTask.lua*, que evalúa el búfer de la máquina de estados del luchador para determinar si se ha lanzado un movimiento especial o magia.

Las magias y movimiento especiales de un luchador están definidas en un fichero <nombre_luchador>.eff como se muestra en 6.10 con la siguiente sintaxis: <nombre>(M <combinación de teclas >pos: <x><y>move: <velX><velY>| SM <combinación de teclas >)[condición].0

Listado 6.10: Fichero ryu.eff

```
hadoken drp  M Tauto pos: 40 40  move: 20 0 hadokenCondition
shordoken  rdrp SM shordokenCondition
```

6.10. Efectos de imagen

Aunque no tengan ningún sentido funcional, el juego tiene una serie de efectos cuyo objetivo es el de crear un mundo animado donde el jugador se sienta cómodo y le traslade a una realidad ficticia. Estas animaciones son instancias de la clase **Sprite**. La clase encargada de almacenar

estas animaciones globales es **FightEffectsManager** definida en *engine/fight.hpp* mediante un vector de objetos **FightEffect**.

Listado 6.11: La clase FightEffect

```
struct FightEffect{
        string nameSprite;
        int x,y;
        unsigned current_i;
        auto_ptr<Sprite> sprite;
};
```

FightEffect::nameSprite representa el nombre de una animación. FightEffect::x y FightEffect::y son puntos del plano en que el efecto es dibujado. FightEffect::current_i corresponde a la actual diapositiva de la animación y FightEffect::sprite es un puntero inteligente que representa la animación cargada. Todo estos aspectos, así como la utilización de éstos en el escenario, se explican con más detalle en el apéndice C.

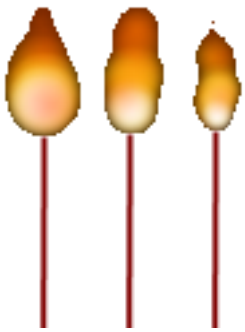


Figura 6.13: Imagen de un efecto de escenario

6.11. El sonido

El sonido de los golpes, las magias y la música del escenario forman una parte importante del simulador. Para los sonidos ha sido diseñada una clase, como se ha explicado en el capítulo 5, para que esté disponible y global en todo el simulador.

En el caso del luchado, en el sistema existen sonidos que son generales para todo luchador, algunos encargados de los golpes, cuando un luchador es golpeado etc. Un luchador puede tener sus propios sonidos para estos eventos genéricos del sistema, de manera que puede reescribirlos cuando los reproduzca, de manera que se consigue que un luchador se puede personalizar en todos estos aspectos.

Existe un fichero en el simulador, *global.sfx*, donde se encuentran todos los sonidos que se cargan en **GlobalSFXPlayer**. La clase **SFXPlayer** es utilizada para que otra clase pueda reproducir sus propios sonidos, primero buscando un sonido en su propio diccionario y si no lo encuentra buscándolo en **GlobalSFXPlayer**.

Listado 6.12: Código para reproducir un efecto de sonido

```
void SFXPlayer::play(std::string key) const {
    // se busca en el diccionario privado de la clase.
    if(  Hsounds.exists(key)  )
        sound = Hsounds.at(key);
        // si no se encuentra se busca en el diccionario global
    else if( GlobalSFXPlayer::getInstance()->exists(key) )
        sound = GlobalSFXPlayer::getInstance()->at(key);
        // se reproduce el sonido
    if(Mix_PlayChannel(-1, sound ,0) == -1){
        cerr << "Error no se ha reproducido" << endl;
    }
}
```

6.12. Los eventos

El tratamiento de eventos se ha implementado mediante un búfer global que almacena instancias de **TEvent** que a su vez representan eventos tanto del teclado como del joystick. Para más detalle del formato de estos ficheros, consultar el apéndice B.

La función listenGlobalEvents definida en *event.hpp* se encarga de rellenar el búfer de eventos, y la función clearBufferGlobalEvents se encarga de vaciarlo. En la clase **Scene** se llama a estas dos funciones en su método Scene::exe, rellenando el búfer para que las máquinas de estados del simulador puedan tratarlo.

Listado 6.13: Código para procesar el búfer global

```
void listenEvents(){
        if( player == Player_any)//no escucha eventos
                return;
        vector<TEvent>::iterator i = global_event_buffer.begin();
        // recorremos el vector de eventos globales, guardando los
           eventos
        // que correspondan  al luchador
        for (;i < global_event_buffer.end(); i++) {
                if ( i->id != player  && player != Player_all)
                        continue;
                if( i->estado == 1)
                        buf.add(*i);
                if(player != Player_all)
                        global_event_buffer.erase(i);
        }
}
```

6.12.1. El joystick

El joystick tiene un problema frente al teclado: Mientras el teclado tiene dos eventos bien diferenciados (tecla presionada y tecla dejada de presionar), un joystick no.

Este dispositivo es tratado al más bajo nivel por libSDL que identifica los joystick conectados mediante un número. En el simulador se usa un diccionario global llamado **Joysticks** que está definido en *engine/global.cpp*. Por cada movimiento del joystick escuchado, se almacenan dos eventos en vez de uno. Mientras que en el teclado un evento representa pulsar o dejar de pulsar una tecla; en el joystick representan un movimientos de ejes que generan dos instancias de **TEvent** (una por cada eje) al búfer.

El movimiento del eje x es recogido por la función **getMoveJoystickX**, definida en *event.cpp*, que se encarga de registrar un evento si se ha pulsado la tecla de dirección izquierda o la derecha.

Listado 6.14: Movimiento del eje x del joystick

```
static TEvent getMoveJoystickX(SDL_Joystick& joy) {
    TEvent e;
    int x_ = e.axisMovement = SDL_JoystickGetAxis(&joy, AXIS_X);
    if (x_ != 0){
        e.tecla = (x_ < 0) ? LEFT : RIGHT;
    }
    e.estado = ( x_!= 0)?1:0;
    e.axis = AXIS_X;
    return e;
}
```

El movimiento del eje y es recogido por la función **getMoveJoystickY**, definida en *event.cpp*, que se encarga de registrar un evento si se ha pulsado la tecla de dirección arriba o abajo.

Listado 6.15: Movimiento del eje y del joystick

```
static TEvent getMoveJoystickY(SDL_Joystick& joy) {
    TEvent e;
    int y_  = e.axisMovement = SDL_JoystickGetAxis(&joy, AXIS_Y);
    if (y_ != 0)
        e.tecla = (y_ < 0) ? UP : DOWN;
    e.estado = (y_!=0)?1:0;
    e.axis = AXIS_Y;
    return e;
}
```

La función **updateJoystickPad**, definida en *event.cpp*, actualiza el mando virtual. Éste nos ayuda a determinar qué eje había estado pulsado antes de añadir un evento para desactivarlo, estoy es muy importante ya que frente a un teclado que las teclas se pulsan y se dejan de pulsar, este tipo de dispositivo genera dos eventos uno por caja eje y el mando virtual que se mantiene en el manejador de eventos es actualizado cuando se llama a ésta función.

6.12.2. La configuración de los controles

La configuración de los controles, permite que el usuario sea capaz de personalizar el significado de las teclas de su dispositivo, adaptándolo a su gusto, cambiando las teclas por defecto.

Esto que parece muy trivial, en el simulador no lo es ya que desde que se carga la aplicación se asigna unas teclas por defecto y si existe algún fichero de formato joy.ctrl (define el joystick) o key.ctrl (define el teclado) en el directorio *etc*, lo leerá y definirá los controles del jugador1 (*player1.key.ctrl* o *player2.joy.ctrl*) y del jugador2 (*player2.key.ctrl* o *player2.joy.ctrl*). Esta lectura inicializa los manejadores de eventos que luego la función listenGlobalEvents, definida en *event.cpp*, se encarga de escuchar para registrar esos eventos en el búfer.

La solución para modificar las teclas del jugador fue añadir una bandera en la clase **Game** llamada Game::listenAll. Si esta bandera está activa, entonces al búfer global se añade todo evento escuchado de los dispositivos.

Para el teclado es muy fácil cambiar las teclas, pero con el joystick no es tan sencillo, ya que a diferencia del teclado, un joystick puede tener muchas teclas (depende del dispositivo del usuario). Por suerte libSDL nos proporciona funciones para conocer cuantos botones tiene en concreto un joystick y numerarlos de [0,n-1], donde *n* es el número máximo de botones.

Cuando esta activo Game::listenAll, tanto para el teclado como para el joystick, se utiliza un atributo de la clase **TEvent**, TEvent::customTecla que almacena todo evento escuchado de un dispositivo. Cuando se genere una nueva configuración se escribe en un fichero de tipo ctrl en la carpeta *etc*.

6.13. Seguimiento del código

En esta sección se explica el mecanismo que se ha utilizado para el seguimiento de los defectos software. En el simulador hay un mínimo tres máquinas de estado funcionando, multitud de animaciones y subsistemas como el intérprete de Lua que ejecuta código interpretado y cuyas excepciones son especialmente delicadas, además del visualizado de imágenes, la reproducción de sonidos, etc. Las herramientas que explico a lo largo de este capítulo ayudan a los programadores a detectar errores y hacer trazas del código de manera ordenada.

6.13.1. El sistema de log

El espacio de nombres LOGGER, definido en *util.hpp*, tiene una serie de funciones que se encargan de escribir en un fichero líneas de texto relativas a un seguimiento o traza de código. Para implementar este sistema de log, se ha utilizado la biblioteca liblog4cpp5.

Por medio de una serie de funciones diseñadas para envolver los aspectos de más bajo nivel de una biblioteca de log, se proporciona una interfaz concreta y una manera de trabajar específica para la comunicación de errores y trazas de código, de manera que un desarrollador puede luego consultar esos eventos o trazas que hay registrado, leyendo un fichero de texto.

En el siguiente ejemplo se muestra un ejemplo el uso de este fichero de registro para comprobar si se ha inicializado correctamente los subsistemas de bajo nivel al inicializar la aplicación y conocer si se han liberado correctamente.

Listado 6.16: Fichero de log del simulador

```
1304979248 INFO main_cat :
                         ---Inicializando los subsistemas de bajo
                            nivel---
                               inicializando la SDL
                               inicializando los controles por
                                  defecto
                               lanzando ventana principal
                               inicializando el subsistema de audio
                               inicializando el subsistema de
                                  fuentes
                               inicializando subsistema Lua
                               leyendo los archivos de config de
                                  controles del usuario
                         --Fin--
1304979250 INFO main_cat :
                         ---LIBERANDO MEMORIA---
                               cerrando subsistema Lua
                               cerrando joystick abiertos
                               cerrando el subsistema de fuentes
                               cerrando el subsistema de audio
                               cerrando sdl
                         --Fin--
```

6.13.2. La pantalla de información

Para que un programador pueda obtener información en todo momento sobre los parámetros de ambos jugadores durante el combate, se ha implementado una ventana que informará sobre los estados de éstos. Esto que parece muy trivial, en el simulador no lo es, debido a la biblioteca utilizada para manejar ventanas libSDL que no permite trabajar con varias ventanas a la vez.

Para solucionar este problema se ha diseñado otro programa escrito en C++ usando el mismo simulador como motor. Por medio de **sockets**, el simulador manda una cadena con los datos de los luchadores, con el fin que se muestre por la pantalla de la aplicación servidor.

Una vez el servidor se reciba las cadenas (que representan al luchador1 y al luchador2) las corta, procesa y muestra por pantalla. Para el tratamiento de la interfaz de comunicaciones se ha utilizado SDL_NET, biblioteca que implementa estas características en varias plataformas.

La clase **Server** está diseñada para llamar a dos funciones cuando se reciba un socket cliente: la función print (para dibujar los datos por pantalla) y listen (para escuchar del búfer y modificar la variable XFF_ENGINE::QUIT usada para cerrar la aplicación).

El cliente que manda cadenas al servidor se controla en *fight.hpp* en el espacio de nombres FIGHT, mediante una cadena global llamada PrintFightBuffer que serán los datos que se envíen al servidor.

La función cuyo nombre es produce manda los datos al servidor en un hilo, para no bajar el rendimiento. Para el tratamiento de hilos se ha hecho uso de libSDL.

En el tratamiento de FightPublicBuffer se controla que no se envíe un búfer vacío, de manera

que al llamar a produce, ésta carga un hilo que toma como argumento una función que, a modo de retrollamada o callback, se encarga de mandar datos al servidor y vaciar el búfer. Una vez es recibida esta cadena, el servidor la trata siguiendo una serie de patrones y la muestra por pantalla.

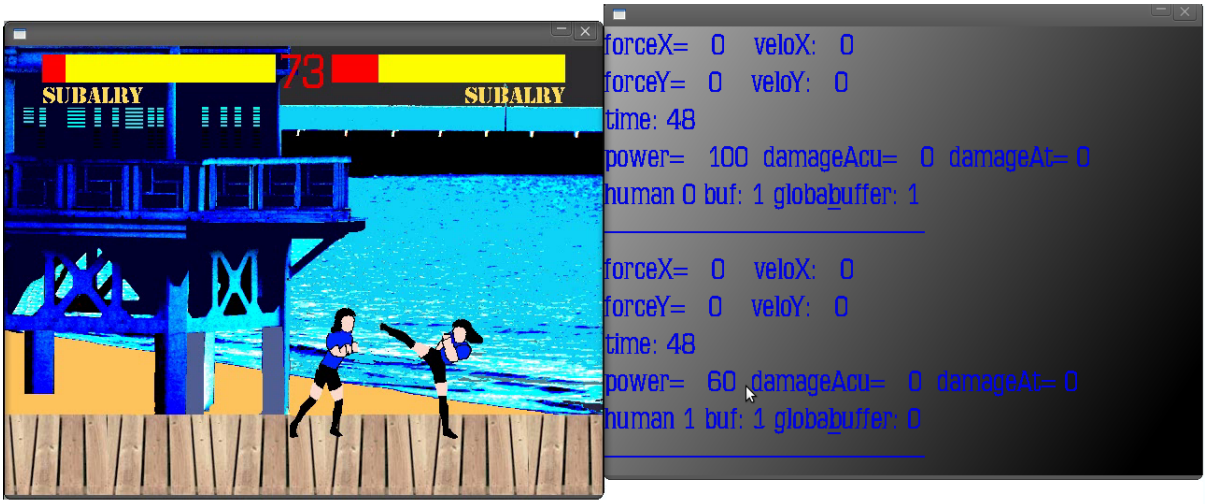


Figura 6.14: La pantalla de información

Listado 6.17: Función para enviar un mensaje a la pantalla de información

```
int send(void* unused) {
        // ...
        sock=SDLNet_TCP_Open(&ip);
        if(!sock){
                string msj ="SDLNet_TCP_Open: "  + string(
                   SDLNet_GetError()) ;
                throw EngineError(msj);
        }
        strcpy(message,FightPrintBuffer.c_str());
        int len =strlen(message);
        SDLNet_TCP_Send(sock,message,len);
        SDLNet_TCP_Close(sock);
        // ...
}
```

6.13.3. El modo depuración

El modo depuración o debug está implementado para visualizar los rectángulos de los que están formados las diapositivas de una animación. En el caso de un luchador, se visualiza el rectángulo que forma la diapositiva, el conjunto de rectángulos que conforman su cuerpo, el conjunto de rectángulo que conforman los golpes que esta propinando en ese momento, el punto de referencia y el punto real de su posición en el plano cartesiano de la pantalla.

El modo de depuración está implementado en Sprite::draw y en Fighter::draw mostrando **Sprite** la diapositiva y el punto de referencia, y **Fighter** los rectángulos de colisiones y el punto real. De esta manera un desarrollador puede observar de una manera visual si las colisiones de un luchador son correctas o no, así como opciones de dibujado como el punto de referencia.

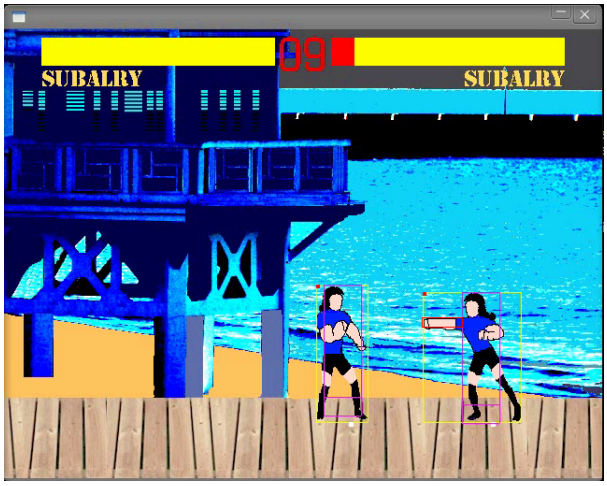


Figura 6.15: Simulador en modo de depuración

Capítulo 7

Pruebas de software

La implementación del sistema, explicada en el capítulo anterior 6, sin un plan de pruebas hubiese resultado muy difícil de organizar. Las pruebas se han clasificado en:

1. **Pruebas unitarias** Comprueban el correcto funcionamiento de una clase. En el simulador se utilizan para probar algoritmos o comportamientos de clases concretas.

   a) **Automáticas:** Los casos de prueba son definidos por el desarrollador.

   b) **Manuales:** Los casos de prueba son definidos por el usuario.

2. **Pruebas de integración:** En el simulador se utilizan para comprobar una funcionalidad concreta del sistema, como elegir personaje o que se lleve a cabo correctamente un combate.

7.1. Pruebas unitarias automáticas

Para la implementación de las pruebas unitarias se ha utilizado la biblioteca libcppunit escrita en C++, construida para servir de marco de trabajo o framework para proporcionar soporte automático a las pruebas de unidad, construyéndolas a través de una serie de pautas y métodos indicados.

Para crear una prueba unitaria se debe crear una clase que herede de otras específicas de la biblioteca como se muestra en el listado 7.1. Cada función o método forman los distintos casos de pruebas implementados, que ordenadamente se comprobarán en la ejecución de ésta.

Todas las pruebas unitarias del simulador están diseñadas siguiendo el modelo de la clase que se observa en el código 7.1, cada método representa un caso de prueba, internamente invocan a macros y a funciones de libcppunit, principalmente a **CPPUNIT_ASSERT**. Esta función acepta un solo argumento de tipo booleano, si éste es falso provocará que en la ejecución de la prueba unitaria se muestre un mensaje de error.

Las pruebas unitarias se han utilizado en el simulador para probar una clase o un algoritmo en concreto (por ejemplo el caso de las colisiones entre rectángulos), de manera que lo que se comprueba tenga independencia, al menos en ese contexto, del resto del sistema.

Listado 7.1: Pruebas unitarias para las colisiones entre dos rectángulos

```
#include <cppunit/TestFixture.h>
#include <cppunit/extensions/HelperMacros.h>
#include "../../engine/engine.hpp"
using namespace XFF_ENGINE;
class TestCollision : public CPPUNIT_NS :: TestFixture
{
        CPPUNIT_TEST_SUITE (TestEngine );
                CPPUNIT_TEST    (       init_pruebas    );
                CPPUNIT_TEST    (       collisionUp_Down );
                CPPUNIT_TEST    (       CollisionCorners );
                CPPUNIT_TEST    (       notCollision    );
                CPPUNIT_TEST    (       hasCollision   );
        CPPUNIT_TEST_SUITE_END ();
    public:
                void init_pruebas(void);
                void collisionUp_Down (void);
                void CollisionCorners(void);
                void notCollision(void);
                void hasCollision(void);
        private:
                XFF_ENGINE::Rect r1,r2;
};
```

La clase Engine

Ubicada en la carpeta *test/engine*, el objetivo es comprobar el comportamiento de la clase **Engine**, encargada de inicializar, ejecutar y destruir el simulador. Las pruebas consisten en crear y destruir un objeto y crear varias instancias de esta clase a la vez.

Todas estas pruebas tienen como objetivo controlar si un objeto se crea y destruye con normalidad, además de comprobar si existen efectos anómalos en la creación de varias instancias, este último se considera como un mal uso del simulador, pero se registra en estas pruebas que ocurre en este caso.

Listado 7.2: Prueba unitaria de la clase Engine

```
jaime@jaime-Aspire-5740:~$ ./engine
TestEngine::setUp : OK
TestEngine::tearDown : OK
TestEngine::initDosVeces : OK
OK (3)
```

La clase Animations

Ubicada en la carpeta *test/animation*, comprueba varios aspectos relacionados con el correcto funcionamiento de una animación, que se detallan a continuación:

- **Fichero action:** Se comprueba si un fichero action está bien construido y no contiene errores.
- **Reproducción:** Se comprueban aspectos relacionados con la reproducción de las diapositivas de una animación.

Pruebas para colisiones:

Ubicada en la carpeta *test/collision*, el objetivo es conocer el comportamiento del algoritmo utilizado para las colisiones, en casos muy concretos, comprobando cómo responde el sistema a determinados comportamientos.

En la figura 7.1 se observan algunos de los comportamientos para el algoritmo, algunos de ellos son: si dos rectángulos que han colisionado colisionan, si dos rectángulos que no han colisionado no colisionan, si dos rectángulos cuyos puntos extremos estén pegados colisionan, si la colisión se produce si sus puntos están pisados, pero no son atravesados, por el otro rectángulo, etc.

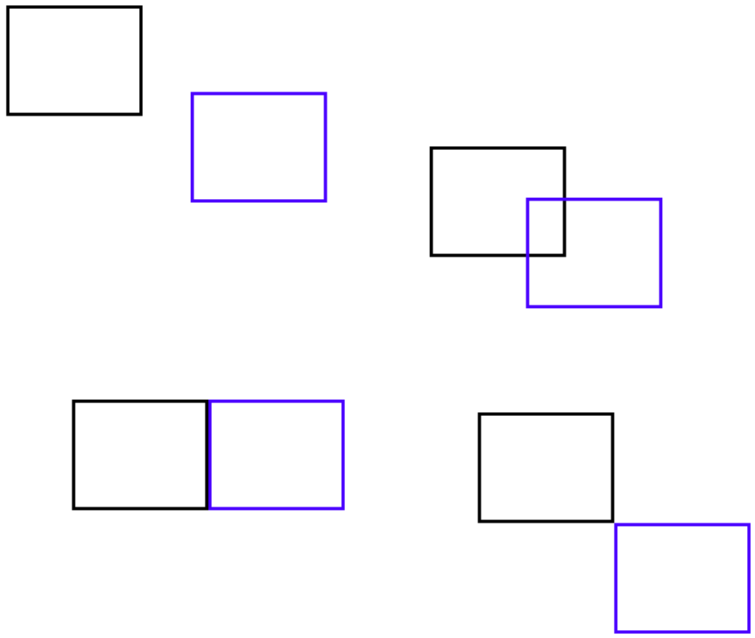


Figura 7.1: Representación de los casos de prueba para las colisiones

7.2. Pruebas unitarias manuales

Estas pruebas se han programado como órdenes, de manera que fuesen pruebas genéricas. Así un usuario puede crear sus propios casos de prueba para comprobar el buen funcionamiento de una imagen, reproducción de sonido o de una animación cargada en el sistema. Las pruebas aceptan el argumento *-h* a modo de ayuda para explicar su funcionamiento.

Pruebas para las imágenes:

Ubicada en la carpeta *test/image* y su sintaxis es `./testImage ruta_imagen`. El objetivo es que el usuario pueda observar el comportamiento correcto de una imagen cargada en el sistema.

Pruebas para el sonido:

Ubicada en la carpeta *test/sound* y acepta como argumento la ruta de un fichero de audio o la ruta de un fichero sfx. En esta prueba comprueba la correcta reproducción de sonidos a partir del nombre de una pista o fichero sfx. El usuario debe comprobar si se escuchan los sonidos.

Pruebas para las fuentes de texto

Ubicada en la carpeta *test/font* y su sintaxis es `./testFont texto ruta_fuente tamaño color`, el tipo de fichero usado para las fuentes de texto es TTF. El objetivo es observar el comportamiento correcto de una fuente cargada en el simulador, en donde se muestran distintos mensajes escritos en texto que el usuario debe comprobar si se visualizan correctamente.

Pruebas para un Sprite o Animación

Ubicada en la carpeta *test/sprite* y su sintaxis es `./testSprite ruta nombre`, a partir del nombre del sprite, en la ruta indicada, busca y carga una imagen y un fichero action. El objetivo es observar el comportamiento correcto de una animación cargada en el simulador, donde se muestra, uno a uno, cada visualización de todas y cada una de las diapositivas de un animación.

Pruebas para el luchador:

Ubicada en la carpeta *test/fighter* y su sintaxis es `./testFighter ruta nombre`, a partir del nombre del luchador, en la ruta indicada, busca y carga un luchador en el sistema. El objetivo es observar el comportamiento correcto de una luchador cargado en el simulador, donde se muestra, uno a uno, cada visualización de todos y cada una de los movimientos de un luchador.

7.3. Pruebas de integración

El combate:

Ubicada en la carpeta *test/combat* y su sintaxis es `./combat luchador1 tipoLuchador luchador2 tipoLuchador escenario`, toma como argumento el nombre de los dos luchadores, además del tipo de luchador, y el nombre de un escenario.

El objetivo es hacer luchar a dos luchadores y ver si se cumplen lo siguiente:

- Un luchador es capaz de golpear todos los tipos de golpes (patada, puñetazo) mediante todos los estados posibles de movimiento.
- Un luchador avanza y colisiona contra su contrario, ambos luchadores se separan.
- Un luchador salta y cae encima de su oponente, empujándolo o modificando su posición.
- Un luchador es capaz de lanzar una magia mediante combinaciones de teclas.
- Un luchador golpea a otro. Hay que comprobar si se contabiliza ese daño en la barra de vida.
- Un luchador es capaz de agacharse y propinar una patada fuerte, haciendo que se caiga el oponente cuando es golpeado.
- Un luchador es capaz de cubrirse y no recibir daño de su oponente.
- Un luchador es capaz de realizar una llave a su oponente.
- Si se ganan los asaltos mínimos termina el combate.

Elegir personaje:

Ubicada en la carpeta *test/chooseFighter*, toma como argumento un tipo de juego. En esta prueba se comprueba lo siguiente:

- Todos los luchadores integrados en el simulador han sido cargados correctamente.
- Se puede escoger un luchador y un escenario.
- En el modo de selección Training se visualizan correctamente las magias de los luchadores.
- En el modo de selección Tournamentent se escogen hasta 8 luchadores.

El menú:

Ubicada en la carpeta *test/mainMenu* comprueba que el menú principal y el menú de opciones funcionan correctamente, comprobando lo siguiente:

- Se puede navegar por el menú principal y el de opciones correctamente.
- En el menú de opciones, las opciones cambiadas, afectan realmente al sistema.
- Si se ha configurado correctamente un dispositivo.

7.4. La ejecución de las pruebas

Para ejecutar las pruebas correctamente en la carpeta *test* existe un script llamado **exe_test.pl** escrito en el lenguaje de programación **Perl** y cuyo objetivo es compilar un caso de prueba concreto. El uso del comando es:

Listado 7.3: Comando para compilar las pruebas en el simulador

```
exe_test <nombre_test> | -l | -a
       -a      Compila todas las pruebas unitarias
       -l  Lista el nombre de las pruebas que se pueden ejecutar (unitarias y no unitarias)
       <nombre_test> Nombre de una prueba, para ver las pruebas ver -l
```

Listado 7.4: Compilación de una prueba

```
$> ./exe_test.pl chooseFighter
ejecutando pruebas: chooseFighter
-- The CXX compiler identification is GNU
-- The C compiler identification is GNU
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Looking for include files CMAKE_HAVE_PTHREAD_H
-- Looking for include files CMAKE_HAVE_PTHREAD_H - found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /xfreefighter/test/chooseFighter
Scanning dependencies of target chooseFighter
[100%] Built target ../bin/chooseFighter
```

Capítulo 8

Conclusiones

La realización de este proyecto me ha supuesto un logro personal. El hecho de haber leído libros sobre programación orientada a los videojuegos y haber aplicado soluciones que se aplican a un nivel profesional ha significado por una parte, una gran responsabilidad y esfuerzo, y por otra ha sido fantástico dedicar tanta seriedad a un tema que me encanta.

La categoría de los juegos de lucha no ha cambiado mucho desde los años 90 y todavía hoy en día se siguen sacando títulos, que en esencia, son iguales en lo que respecta a la forma de jugar pero con un lavado de cara (gráficos y texturas en 3 dimensiones - 3D). En conclusión se puede afirmar que con este tipo de software, aún hoy en día, se gana dinero y no es una aplicación antigua ni obsoleta.

Sobre la interfaz de usuario tengo que admitir que ha sido personal diseñarla y adaptarla como la de una videoconsola de los años 90. No hay ratón y para elegir opciones se mueve arriba y abajo, seleccionando con el botón A de tu joystick o teclado. Además los tipos de lucha son rescatados de los videojuegos clásicos de lucha. Para un usuario acostumbrado a este tipo de software, no será problema navegar por éste, aunque las opciones estén escritas en inglés.

Otro aspecto que me gustaría destacar, es por qué se ha desarrollado un simulador y no un videojuego. El principal motivo es debido a que este tipo de software atraerá a mucha más gente y mucha más comunidad. Aunque la dificultad de tener que hacer ampliable y modificable por el usuario este producto añade una dificultad añadida y en consecuencia que el proyecto tenga otro enfoque distinto al de programar un juego.

Al principio del proyecto, el aspecto de generalidad y modificación por el usuario no se tuvo en cuenta, simplemente se diseñó en las primeras iteraciones una forma de cargar imágenes, fuentes de texto, animaciones, sonidos y música, tal que, se crease una capa de bajo nivel que ya nunca fuese modificada y formase en consecuencia la capa más baja del simulador, y así fue.

Una vez con una buena base, empezó a construirse al luchador, se empezó con un luchador que no era modificable externamente y el proyecto se centró en crear una física fiable entre dos luchadores, con lo que se poco a poco se fue introduciendo el sistema embebido de código mediante la biblioteca luabind, que permite portar código de C++ a Lua y viceversa.

Una vez se diseñó una física y una arquitectura para el sistema embebido, se empezó a implantar la arquitectura para los eventos, añadiendo el joystick y las combinaciones de teclas. Una vez esto se diseñó el primer combate, añadiendo a los luchadores una serie de clases para administrar sus parámetros de la lucha, no sólo frente a la física, sino también para las funciones

Lua que modifican el estado de éste.

Cuando se añadió la gestión de proyectiles, se rediseñó la arquitectura de la máquina de estados y se generalizó al luchador y al proyectil a una misma clase base (**FightEntity**). Con esto la clase **FiniteStateMachine** también se generalizó y pasó a tener dos especializaciones: una para un luchador y otra para un proyectil.

Una vez que se tiene un luchador, un sistema para que lancen proyectiles, y estos automáticamente se muevan por medio de una máquina de estados, llegó el momento de dar respuesta a los requisitos funcionales del proyecto. Así se pensó en generalizar aún más la máquina de estados convirtiéndola en una clase paramétrica cuyo parámetro sea una clase portada a Lua. Este aspecto a desarrollar ocupó bastante tiempo, pero se consiguió no sólo modificar el comportamiento del luchador y los proyectiles mediante código embebido, sino también de los modos de juego. En consecuencia, se consiguió el principal objetivo de implementar un simulador: el modificar y personalizar por parte del usuario la aplicación.

Desarrollar este proyecto me ha costado mucho esfuerzo y dedicación, no me gustó el hecho de tener que programar desde cero pero no me quedó más remedio. Mis investigaciones iniciales estuvieron centradas en Mugen, cuya licencia te permite usarlo gratuitamente y con una amplia comunidad, pero de código y formatos cerrados. OpenMugen, proyecto abandonado en 2004 y que se basa en un intento de copia de Mugen pero opensource, fue otra alternativa pero al estar abandonado y tener más de 10.000 líneas de código en C, sin comentarios, repleto de errores y llamadas a ensamblador. Decidí entonces aprovechar algunas funcionalidades, pero desechando retomar su código.

En mi proyecto se han dedicado muchas horas a las pruebas e integración de distintas bibliotecas, el hecho de integrar un intérprete de Lua ha sido especialmente duro. Hacer funcionar un sistema embebido de código ha dotado al simulador que los demás productos parecidos no tienen y que han sustituido desde mi punto de vista por miles de formatos propios y analizadores hechos a mano, que dificultan el aprendizaje para un usuario principiante.

Este simulador llamado XFREE FIGHTER acaba de nacer, como en todo software deben de hacerse mejoras en muchos aspectos, algunas de ellas hubiesen significado en algunos casos otro proyecto fin de carrera.

Mejoras futuras

Lenguaje formal para la máquina de estados

El formato de la máquina de estados es muy estricto y estático, lo ideal hubiese sido crear una serie de palabras reservadas para la descripción de los estados, usando para ello analizadores léxicos y sintácticos como flex o bison. Debido al poco tiempo que he tenido para hacer funcionar y generalizar la máquina de estados a cualquier objeto (no solo al luchador), me ha hecho pasar este detalle por alto, aunque eso sí dejarlo como la primera mejora en la siguiente versión del simulador.

Entorno para la maquetación

Mugen tiene programas que manipulan los formatos de animación, de manera que una persona sin necesidad de manejar formatos de ficheros (muchas veces muy estrictos) y de manipular imágenes a mano, es capaz de crear un luchador. Para ello tiene varios programas, dotados de interfaz gráfica de usuario, para definir cada diapositiva de una animación, y además de cada una de éstas su punto de referencia y sus colisiones. De manera que definir un personaje se hace mucho más rápido y mucho más intuitivo. Para esta mejora se plantean dos alternativas:

1. **Aprovechar los formatos de Mugen:** Dado un luchador que sigue los formatos de Mugen, convertirlo en un luchador compatible aplicando una conversión de formatos. Con eso se puede aprovechar todos los luchadores que hay por internet y además reutilizar toda la comunidad que trae consigo Mugen y los programas que manipulan los formatos de animación.

2. **Desarrollar un entorno propio:** Desarrollar un programa que cargue una imagen de un luchador y sea capaz a través de una interfaz de usuario, generar los formatos necesarios para hacerlo funcionar en el simulador.

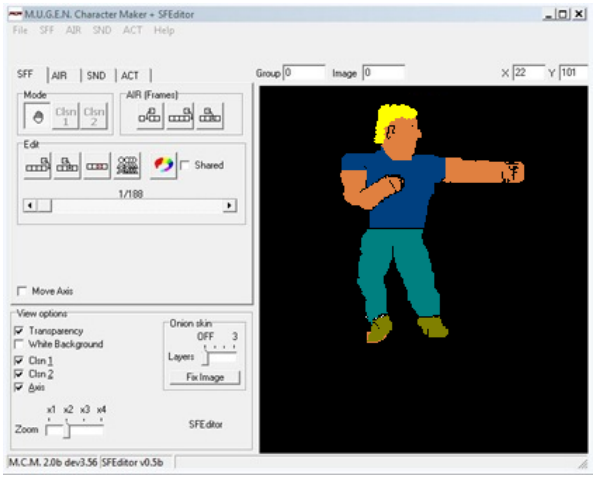


Figura 8.1: MUGEN Character Maker utilizado para integrar animaciones en MUGEN

El modo Online

Permitiría que dos jugadores puedan luchar entre sí a través de una red. Para conseguir este modo de juego habría que adaptar la arquitectura del simulador para aceptar una interfaz de comunicaciones de red, es decir, diseñar e implementar una nueva capa.

Además habría que adaptar un servidor que almacene puntuaciones, administre torneos y haga de interfaz entre los distintos usuarios del simulador, añadiendo una herramienta de comunicación instantánea, que permita a los jugadores, dialogar para ejecutar un combate entre ellos.

Portabilidad a otras plataformas

Las herramientas que se han utilizado para desarrollar el simulador son multiplataformas. Un usuario puede descargarse el código fuente del simulador y compilarlo donde quiera. Sin embargo, las distintas plataformas que se han seleccionado para portar la aplicación en futuras versiones son:

- Otros sistemas operativos:
  - **MS Windows:** de Microsoft.
  - **Mac OS X** de Apple.
- Videoconsolas portátiles:
  - **PSP** de Sony.
  - **GP32X** de Game Park.

Apéndice A

Máquina de estados

A lo largo de este manual se explica cómo se ha codificado, integrado y adaptado la máquina finita de estados a una clase en concreto dentro del simulador.

Vamos organizar esta explicación en tres secciones: La primera A.1, describe la clase **FiniteStateMachine**, la segunda A.2 muestra un ejemplo de máquina de estados para una clase y la tercera A.3 el uso en un código C++.

A.1. Primera parte: La clase FiniteStateMachine

Para saber cómo funciona la máquina de estados, primero hay que comprender la clase paramétrica **FiniteStateMachine** definida en *engine/fsm.hpp* que recibe como parámetro cualquier clase que haya sido portada a Lua. Estas clases portadas se encuentran en *engine/scripting.cpp*.

Hay que destacar que la máquina de estados (**FiniteStateMachine**) se encarga de gestionar los eventos y está provista de un búfer que según el identificador de eventos, escucha al jugador1, al jugador2, ambos o ninguno. Los identificadores están definidos como un enumerado llamado **EventPlayer** y están definidos en *engine/global.hpp*.

Listado A.1: La clase FiniteStateMachine

```
template< typename T > class FiniteStateMachine {
        //...
        public:
                FiniteStateMachine(T&,string,bool,EventPlayerr );
                void blockFSM();
                bool quitBlockFSM(string st);
                void listenEvents();
                const string& getState() {return state; }
                void update();
                const string& getState() {return state; }
                std::string toString();
};
```

Del constructor hay que destacar **funTask** que es el nombre de una "función maestra" que

es llamada cada vez que se ejecuta una tarea, aspecto se ve con más detalle en la siguiente sección A.2.

Listado A.2: Ejemplo de función maestra de una máquina de estados

```
function <nombre de funTask >( entity, fsmNode,nombre_estado_actual)
        -- resto del procedimiento
end
```

El método más importante es FiniteStateMachine::update que es llamado por todo objeto cuando quiere actualizar su estado y escuchar los eventos externos.

Todas las llamadas al sistema embebido se realizan usando este método. Según qué valores tenga el estado actual (**FSMState**) se evalúan sus condiciones de transición llamando a funciones Lua, si alguna es verdadera se pasa a un nuevo estado. En el caso que no exista transición se ejecuta el nodo actual del estado (**FSMNode**) evaluando condiciones y ejecutando tareas que modifican el estado de ese objeto a través de la función maestra.

A.2. Segunda parte: Adaptar la máquina de estados a una clase

Para ilustrar el uso de la máquina de estados se muestra el siguiente ejemplo de una calculadora de enteros que suma y resta. Mediante este ejemplo se pretende dar una visión general sobre cómo funciona y cómo se implementa en el sistema. Conectar código de C++ y Lua requiere una serie de pautas ordenadas.

El punto de partida es una clase **Calculadora** cuyo comportamiento se quiere portar a una máquina de estados, la declaración de la clase es la siguiente:

Listado A.3: La clase Calculadora

```
class IntCalculadora{
private:
        int operacion;
public:
        IntCalculadora():op1(-1),op2(-1),operacion(-1)
        int op1;
        int op2;
        void setTipoOperacion(int to );
        int void setTipoOperacion();
        int getResultado();
}
```

Portar código a Lua

En primer lugar se debe editar el fichero de código *engine/scripting.cpp* y añadir en la función SCRIPTING::initLua la clase **Calculadora**, de esta forma se puede llamar desde cualquier

script ejecutado desde el simulador.

Listado A.4: La clase Calculadora añadida a scripting.cpp

```
module(luaHandle)[
        class_<IntCalculadora > ("IntCalculadora")
        .def(luabind::constructor<>())
        .def_readwrite("op1", &IntCalculadora::op1)
        .def_readwrite("op1", &IntCalculadora::op1)
        .def("setTipoOperacion", &IntCalculadora::setTipoOperacion)
        .def("getTipoOperacion", &IntCalculadora::getTipoOperacion)
        .def("getResultado", &IntCalculadora::getResultado)
];
```

En el siguiente código se puede ver cómo se usa esta clase una vez portada a Lua:

Listado A.5: Script Lua en donde se utiliza la clase Calculadora

```
-- pruebaCalculadora calc.lua
        calc = IntCalculadora()
        calc.op1 = 200
        calc.op2 = 200
        calc:setTipoOperacion(2) -- selecciono resta
        -- realiza la operacion  y muestra el resultado
        rtdo = calc:getResult()
        -- nos muestra 0
        print( "Resultado: ",rtdo )
-- fin script
```

El fichero fsm

El fichero en donde se declara el comportamiento de la máquina de estados es un fichero con formato fsm. Éste se encarga de gestionar la relación de los estados entre sí, cuando termina la ejecución y donde se guarda la información para que el simulador se pueda comunicar con el sistema de código embebido.

Listado A.6: El fichero calculadora.fsm

```
[ INIT
        anim= null
        numState= 1
        1, initCalculadora, 1
                0, tautologia
]
[ RECOGER_OPERANDOS
        anim= null
        numState= 1
        1, recogerOperandos, 1
                0, tautologia
```

```
]
[ FINISH
        anim= null
    numState= 3
        1, ejecutarResultados, 2
                2, resultadoIlegal
                3, resultadoOk
        2, showResultado, 1
                0, tautologia
        3, showError, 1
                0, tautologia
]
[[STATES]]
[ INIT
        [ FORW 0
        ]
        [ END 1
                RECOGER_OPERANDOS ftautologia
        ]
]
[ RECOGEROPERANDOS
        [  FORW 1
                FINISH lecturaIlegal
        ]
        [  END 1
                FINISH finishTransition
        ]
]
[ FINISH
        [  FORW 0
        ]
        [  END 1
                INIT ftautologia
        ]
]
[[INIT]]
init= QUIT
ends= FINISH
```

Un fichero fsm está dividido en tres partes. En la primera se describe el comportamiento de cada unos de los estados, la segunda que comienza con [[STATES]], describe las relaciones de cada uno de los estados y la última delimitada por [[INIT]] se encarga de definir el estado inicial (estado por el que empieza cuando es construida una instancia de **FiniteStateMachine** y cargado el fichero fsm) y los estados terminales (estados que una vez ejecutados la máquina de estados se bloquea).

En el fichero fsm de **Calculadora** el estado finish, en la primera parte del fichero, se describe su número de estados (con numState). Donde para cada estado hay definida una tarea y una serie de condiciones que derivan a otros. En el estado 1 es ejecutarResultados, esta tarea (función

Lua) modifica el estado del objeto que recibe como parámetro **FiniteStateMachine**.

Listado A.7: El estado FINISH

```
[ FINISH
        anim= null
        numState= 3
        1, ejecutarResultados, 2
                2, resultadoOk
                3, resultadoIlegal
        2, showResultado, 1
                0, tautologia
        3, showError, 1
                0, tautologia
]
```

Una vez se ejecute la tarea se evalúan las condiciones. En este caso hay dos, resultadoIlegal y resultadoOk (funciones Lua ambas que devuelve un booleano y que son excluyentes entre sí). Por ejemplo, si resultadoIlegal es verdadero se pasa al estado 3 y se ejecuta showError, esto se repite hasta que se llegue al estado 0, un estado especial que indica que el estado ha terminado.

Una vez se sale de un estado entran las transiciones. Por ejemplo, si la máquina de estados se encuentra en el estado RECOGEROPERANDOS o en cualquier otro siempre se ejecutarían las condiciones FORW correspondientes al estado y definidas en la segunda parte del fichero fsm. Estas condiciones de transición, si son verdaderas, un nuevo estado es seleccionado. En este caso se pasa de RECOGEROPERANDOS a FINISH si operandoIlegal es verdadero.

Las condiciones definidas como END dentro de un estado en la segunda parte del fichero fsm, son aquellas que cuando termina el estado (se encuentra en el nodo especial cero) se evalúan y si es verdadera alguna de ellas se pasa al otro estado, en este caso si se cumple finishTransition se pasa de RECOGEROPERANDOS a FINISH, si no se cumple otra vez se pasa a RECOGEROPERANDOS.

El formato para un estado de la sección segunda de un fichero fsm es el siguiente:

```
<ID ESTADO>, <NOMBRE TAREA>, <NÚMERO DE CONDICIONES>
<ESTADO>, <NOMBRE DE LA CONDICIÓN>
.. tantas como <NUMERO DE CONDICIONES>
```

Listado A.8: Transiciones del estado RECOGEOPERANDOS

```
[ RECOGEROPERANDOS
        [  FORW 1
                FINISH lecturaIlegal
        ]
        [  END 1
                FINISH finishTransition
        ]
]
```

Crear un script Lua que proporciona comportamiento al fichero fsm

Lo primero es crear la función maestra, que recibe un objeto de la clase IntCalculadora, un objeto de la clase **FSMNode** que representa el estado dentro de un estado de la máquina de estados y un string con el nombre del estado actual.

Esta función se encarga de ejecutar las tareas (el nombre de la función así como la estructura de un estado están recogidas en **FSMNode**, de manera que un **FSMState** esta compuestos de muchos FSMNode) y siempre devuelve una estructura **FSMTask** que se usa para uso interno.

Listado A.9: El fichero calculadora.lua

```
--variable global
resultado = -1 -- valor ilegal
--fin variable global
function IntCalculadoraParamTask( calc, fsmNode, actualState )
        fsmTask = FSMTask()
        --tarea a ejecutar ( nombre de la funcion )
        nameFunction = fsmNode.nameTask
        _G[ nameFunction] (calc)
        return fsmTask
end
function initCalculadora(calc)
        calc.op1 = 0
        calc.op2 = 0
        calc:setTipoOperacion(0)
end
function recogerOperandos (calc)
        op1 = 0
        op2 = 0
        to = 0
        print ( "introduce el operando 1 " )
        op1 = io.read( "*number" )
        print ( "introduce el operando 2 " )
        op2 = io.read( "*number" )
        print ( "introduce el tipo de operacion" )
        to = io.read( "*number" )
        calc.op1 = op1
        calc.op2 = op2
        calc:setTipoOperacion( to )
end
function ejecutarResultados(calc)
        rtdo =  calc:getResult()
end
function showResultado(calc)
        print( "El resultado es ", rtdo )
end
function showError (calc){
        to = calc:getTipoOperacion()
        if to <= 0 or to > 2 then
```

```
39                  print( "error el operando no es correcto")
40          end
41          if calc.op1 < 0  or calc.op2 < 0 then
42                  print ( "los operandos no son correctos ")
43          end
44          if rtdo < 0 then
45                  print ( "El resultado no es correcto" )
46          end
47  }
```

De esta manera se ejecuta un comportamiento común para las condiciones entre los estados que serán llamadas desde la plantilla de la clase **FiniteStateMachine** en la función evaluateConditions, a continuación la definición de las condiciones de IntCalculadora:

Listado A.10: Condiciones de estado

```
1  function tautologia( calc )
2          return true
3  end
4  function resultadoIlegal( calc )
5          return  rtdo < 0
6  end
7  function resultadoOk ( calc )
8          return not resultadoIlegal( calc )
9  end
```

Las transiciones de estados se evalúan en la clase FiniteStateMachine::update, llamando a evaluateStateTransitions, siempre y cuando esté a verdadero el booleano activateTransitions, si éste no esta activo las transiciones se hacen a mano (o deberán hacerse) mediante el paso de funciones a beforeProcessEvent y afterProcessEvent.

Listado A.11: Condiciones de transición

```
1  function lecturaIlegal( calc,actualState)
2          to = calc:getTipoOperando()
3          return
4          ( to <= 0 or to < 2 )
5          or
6          ( calc:op1 < 0 )
7          or
8          ( calc:op2 < 0 )
9  end
10 function finishTransition ( calc, actualState )
11         return not lecturaIlegal( calc,actualState)
12 end
13 function ftautologia ( calc, actualState )
14         return true
15 end
```

A.3. Tercera parte: Ejemplo de uso

En el siguiente código se observa cómo desde la aplicación C++, se crea una máquina de estados que se encarga de dar comportamiento a la clase **Calculadora** automáticamente solamente creando una instancia de la máquina de estados, **FiniteStateMachine** y ejecutándola.

Listado A.12: Condiciones para la clase Calculadora

```
1 IntCalculadora calc;
2 IntCalculadoraFSM fsm(calc);
3 while( !fsm.finish() )
4         fsm.update();
```

Apéndice B

Manual de usuario

B.1. Instalación del simulador

El proyecto se encuentra alojado en sourceforge.net un sitio web que actúa como repositorio de código fuente de diversos proyectos libres. El sitio del proyecto se encuentra en esta dirección web: `http://sourceforge.net/projects/xfreefighter/`.

Para descargarse el código fuente y poder compilar, es necesario tener la aplicación **cmake**, la cual se puede descargar `http://www.cmake.org/` y tener instalada la aplicación subversion `http://subversion.tigris.org/`

El sistema operativo utilizado para desarrollar el simulador ha sido **Ubuntu 10.10 - Maverick Meerkat**, tanto subversion como cmake son fáciles de instalar:

Listado B.1: Orden para instalar CMake y subversion

```
sudo apt-get install  cmake subversion
```

Una vez instaladas estas aplicaciones, se descarga el código fuente desde Sourceforge introduciendo en un terminal del sistema operativo la siguiente orden:

Listado B.2: Descargar el simulador desde la forja

```
svn co https://xfreefighter.svn.sourceforge.net/svnroot/xfreefighter xff
```

Una vez se haya descargado correctamente, se compila y se ejecuta escribiendo la siguiente orden:

Listado B.3: Compilar y ejecutar el simulador

```
cd xff/trunk && cmake . && make && ./gxff
```

Cmake va informando de las dependencias no instaladas, si alguna falta, en el sistema operativo Ubuntu 10.10 se instalan usando orden:

Listado B.4: Orden para instalar las dependencias de la aplicación en Ubuntu

```
sudo apt-get install  libluabind0.9.0 libsdl-ttf2.0-dev libsdl-net1.2-dev
libsdl-mixer1.2-dev libsdl-image1.2-dev libsdl1.2-all libsdl1.2-dev
liblog4cpp5 libcppunit-dev
```

B.2. Pantalla principal

En la pantalla principal, que se muestra en la figura B.4, podemos acceder a los distintos modos de lucha que hay en el simulador, salir de la aplicación, o ir a un menú de opciones. Tanto en el menú principal como en el de opciones, las teclas arriba y abajo son usadas para navegar entre las secciones y el botón A para seleccionar una sección en concreto.

En la sección de opciones se puede guardar una configuración personalizada de aspectos generales del juego, como se ve en en la figura B.2. Un ejemplo de ello es que se pueden cambiar los controles por defecto para el teclado y para los mandos de ambos jugadores.

Figura B.1: Menú principal del simulador

Listado B.5: Controles de teclado por defecto del jugador 1

```
cursor flecha arriba:           Saltar.
cursor flecha abajo:            Agacharse.
cursor flecha derecha:          Avanzar hacia la derecha.
cursor flecha izquierda:        Avanzar hacia la izquierda.
tecla 1:                        Golpe lento.
tecla 2:                        Golpe medio.
```

```
tecla 3:                        Golpe fuerte.
tecla 4:                        Patada lenta.
tecla 5:                        Patada media.
tecla 6:                        Patada fuerte.
tecla escape:                   Pause.
```

Listado B.6: Controles de teclado por defecto del jugador 2

```
tecla w:                Saltar.
tecla s:                Agacharse.
tecla d:                Avanzar hacia la derecha.
tecla a:                Avanzar hacia la izquierda.
tecla u:                Golpe lento.
tecla i:                Golpe medio.
tecla o:                Golpe fuerte.
tecla j:                Patada lenta.
tecla k:                Patada media.
tecla l:                Patada fuerte.
tecla m:                Pause.
```

Los controles por defecto del joystick para los jugadores dependerá de la numeración interna que lleven sus dispositivos como se expresa a continuación:

- **botón 1:** Golpe lento.
- **botón 2:** Golpe medio.
- **botón 3:** Golpe fuerte.
- **botón 4:** Patada lenta.
- **botón 5:** Patada media.
- **botón 6:** Patada fuerte.
- **botón 7:** Pause.

Desde las opciones como se muestra en la figura B.2 se puede modificar los asaltos modificando Rounds, la dificultad del juego en Difficult y el tiempo de un asalto en Time limit. Para configurar los controles se selecciona la sección *Config Controls* y a continuación, se selecciona dispositivo, entonces el sistema va pidiendo teclas y el usuario pulsando, escribiendo el correspondiente fichero de configuración de dispositivo.

B.3. Escoger un luchador

Cuando se selecciona un tipo de juego, como se muestra en la figura B.3, aparece el título de éste y junto con las teclas correspondientes a moverse hacia la derecha o la izquierda se van mostrando en orden a todos los luchadores que tiene cargado el sistema. Para elegir un luchador en concreto se pulsa la tecla A.

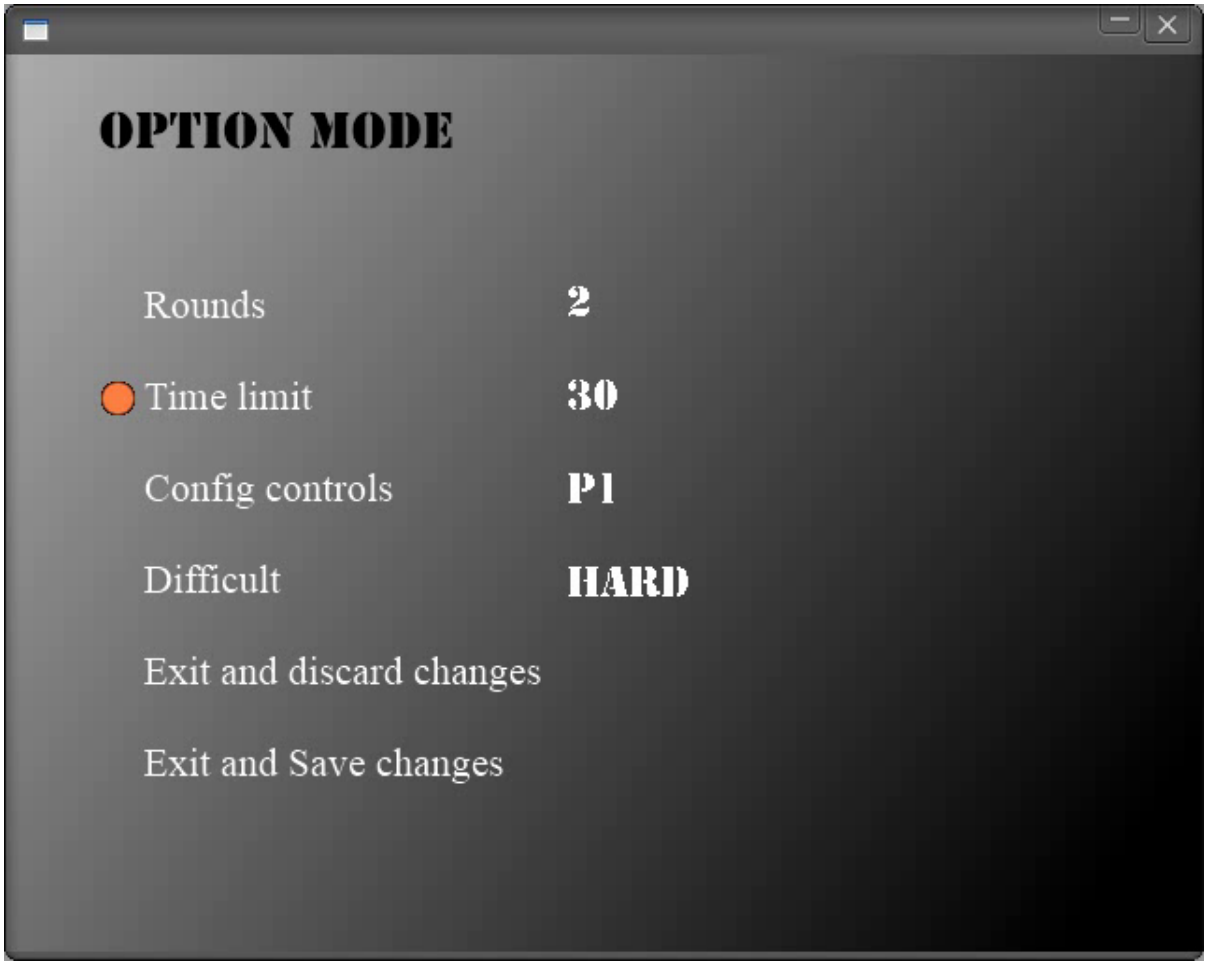


Figura B.2: Menú de opciones

B.4. Controlar el luchador

En esta sección se exponen una serie de consejos y buenas prácticas para conocer las claves para ganar un combate 1vs1. Estas claves deben ser practicadas por el jugador hasta familiarizase con ellas.

Protegerse

Si se avanza hacia atrás cuando el oponente pega un golpe al luchador, éste se protege quedando bloqueado unos segundos y no contabilizándose el daño en el caso de colisionar con éste. Lo mismo ocurre si un proyectil está muy cerca.

Contraataques

Siempre que un luchador haga una magia o propine un golpe, hay que intentar atacar, pero hay que recordar que los golpes fuertes son duros en daño pero son lentos, con lo cual puede

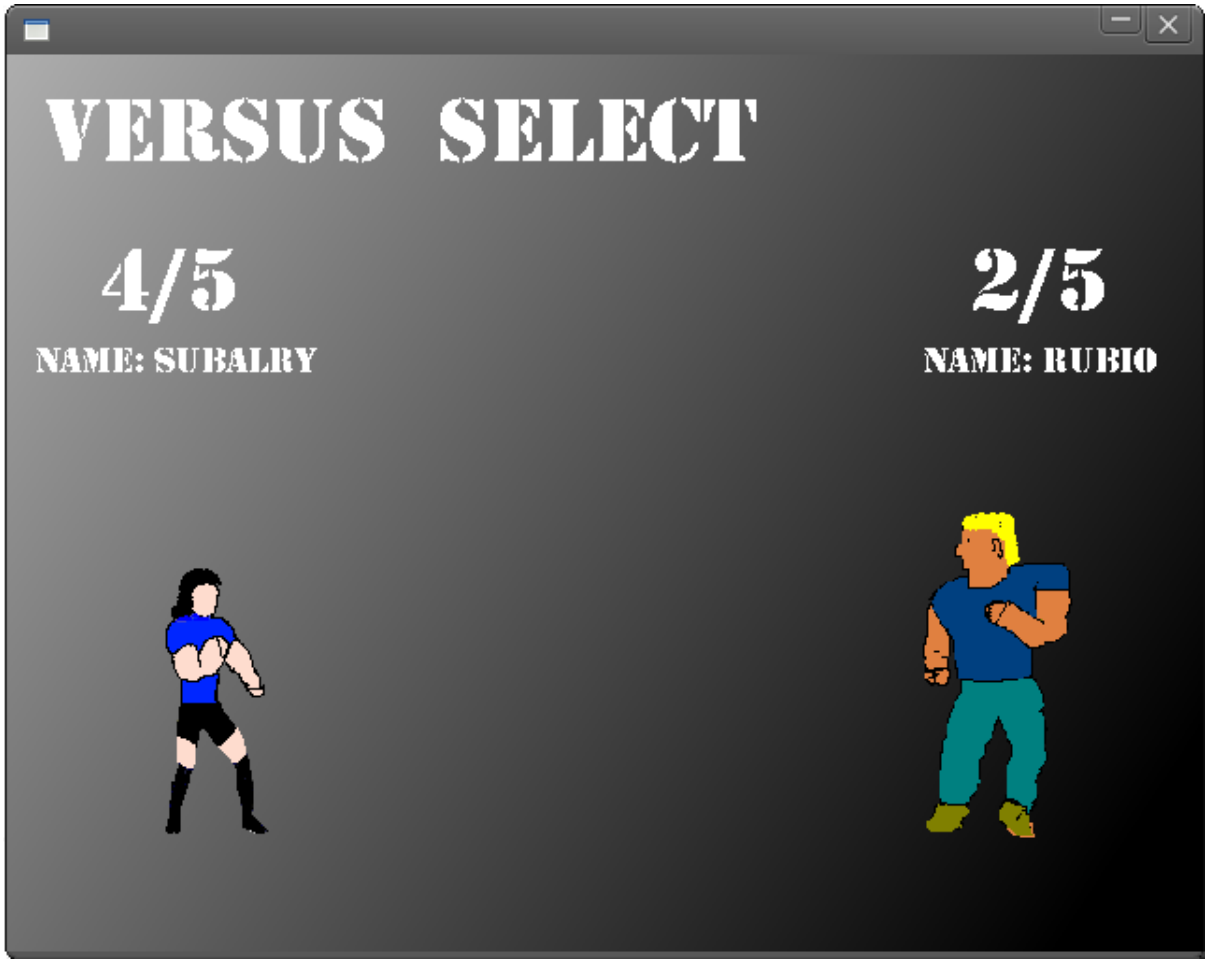


Figura B.3: Selección de luchadores en el tipo de juego versus

contraatacar tu oponente también.

Lo ideal es intentar dejar al oponente en estado de KO, esto se produce cuando en poco tiempo se reciben muchos golpes. Dejando en este estado a un luchador, se puede derrotar en muy poco tiempo.

Las magias y los movimientos especiales son muy poderosos, pero normalmente dejan desprotegido al que los lanza, siendo vulnerable a golpes o a otros movimientos especiales.

Practicar

Cuando no se conoce bien a un luchador lo mejor es escoger el modo de entrenamiento y con mucha tranquilad, practicar las combinaciones de teclas y ver los puntos débiles y fuertes de cada personaje.

Ataques que hacen caer al oponente

Toda patada fuerte propinada a un luchador, si el que la propina esta agachado, provoca que el oponente se caiga, esto hace que los contraataques sean fáciles de ejecutar, incluso lanzar un proyectil y que al oponente no le tiempo a defenderse.

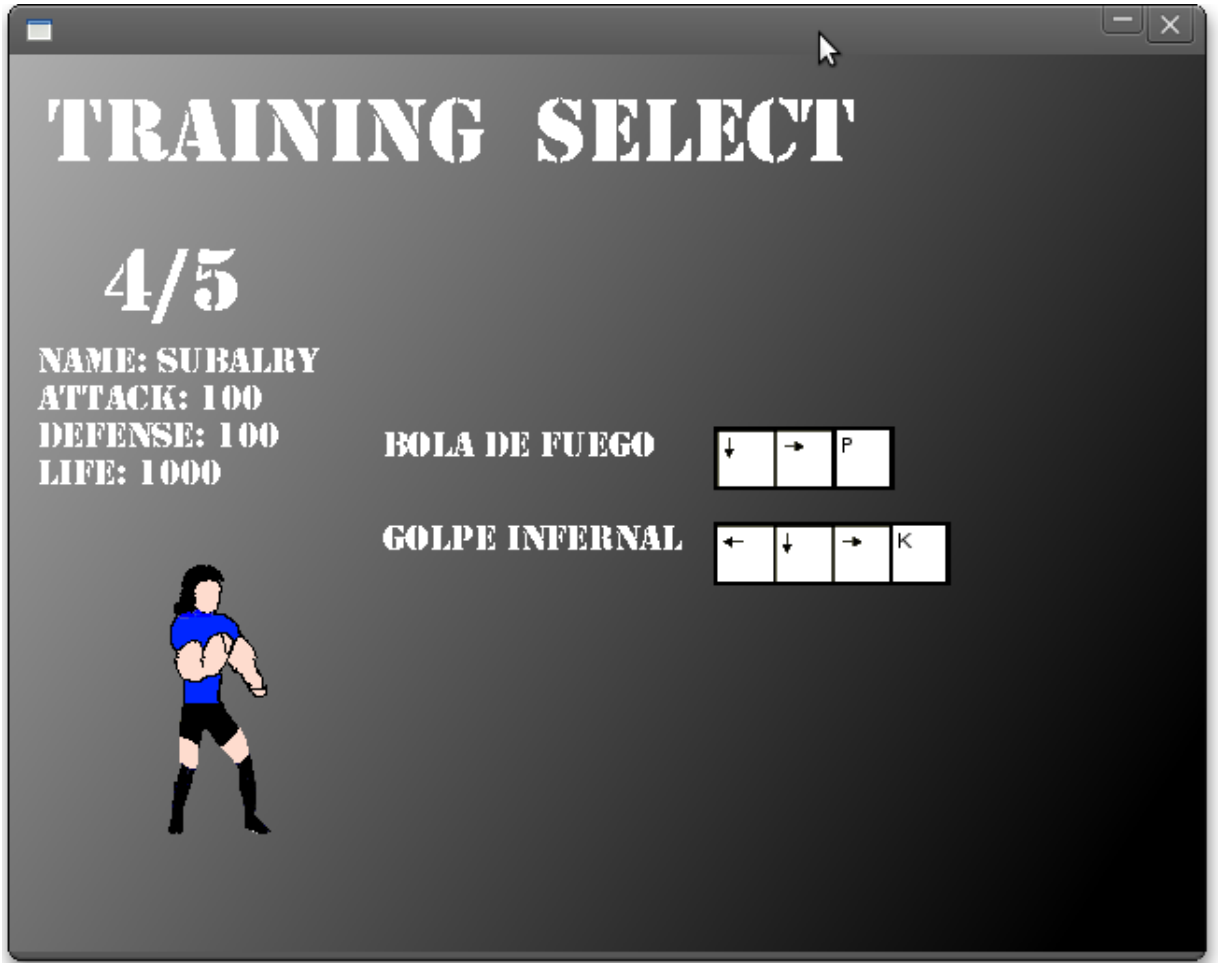


Figura B.4: Selección de luchador en modo de entrenamiento

B.5. Sistemas de juegos

Los sistemas de juego son cuatro, con las siguientes características:

- **Entrenamiento o Training:** Se puede probar a un personaje contra sí mismo, este personaje no muere nunca.
- **Arcade:** Se puede probar a un personaje contra todos los demás manejados por la máquina.

- **Tournament o Torneo:** Los jugadores eligen cuatro personajes cada uno, los cuales se enfrentan en sucesivos combates.
- **Versus:** Los jugadores eligen un personaje cada uno y estos se enfrentan en un combate.

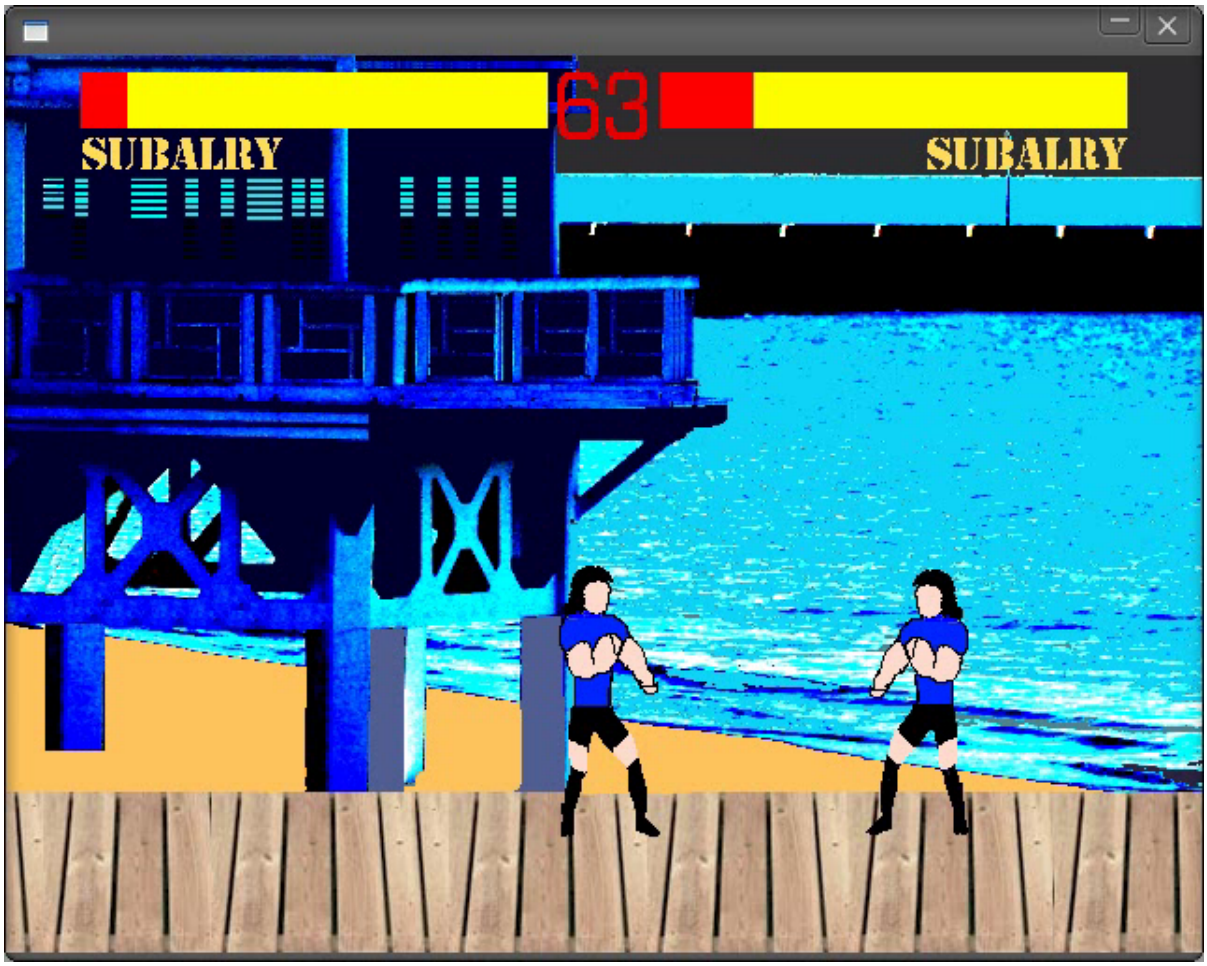


Figura B.5: La lucha

Apéndice C

Manual para la maquetación de animaciones

La maquetación se refiere al proceso por el cual se adapta una imagen a una forma determinada y se diseña la forma en que las animaciones van ha ser cargadas en el sistema. En el simulador hay cuatro elementos a maquetar: luchadores, escenarios, proyectiles y sprites o animaciones.

En los sucesivas secciones, se describen los formatos a seguir C.1, cómo se maqueta un luchador C.2, un proyectil C.3 y un escenario C.4. La maquetación de las animaciones o sprites se obvia al poderse maquetar de la misma manera que un proyectil.

C.1. La imagen y el formato de la animación

La maquetación se centra a partir de una imagen en formato PNG dibujada en una determinada forma, escribir un fichero action que definen las animaciones aplicadas a esta imagen, sus colisiones y la forma en que se dibuja al luchador en una determinada diapositiva.

La biblioteca utilizada para cargar imágenes es libSDLImage y puede soportar multitud de formatos PCX, Gif, JPG, XPM, TIF, BMP, PNG etc. Pero se elige el formato PNG debido a que es un formato estándar, libre y no sujeto a patentes.

El objetivo de esta sección es mostrar cómo se carga la animación de un luchador mediante su imagen y un fichero de texto que se tendrá que crear y editar a mano siguiendo una serie de reglas.

La imagen no puede estar dibujada arbitrariamente, el simulador es capaz de rotar una imagen inteligentemente dependiendo de la orientación. La imagen debe ser de la forma de la forma que muestra la figura C.3 y el luchador, como se observa, debe estar mirando hacia la derecha. Los **proyectiles**, y en consecuencia cualquier clase que herede de **Sprite**, deberán seguir la misma regla.

Un fichero action enumera las animaciones por un nombre y a su vez están dividas por varios frames, cada frame tiene un punto de referencia, un rectángulo que delimita a su vez un trozo de rectángulo dentro de la imagen que se quiere mostrar, y dos vectores de rectángulos dentro del cuadro de la animación que determinará el cuerpo (lo que puede colisionar) y los golpes (lo que puede dañar).

Un fichero action esta compuesto por una o más animaciones, el formato de una animación es:

```
[ <identificador de la animación>
begin
 x,y < N | V | H | HV > { x , w , y , w  }
 { x , w , y , w } < A | B >
 ...
end
...
]
...
```

Como se muesta en el listado C.1, cada animación esta delimitada dentro de unos corchetes, dentro de estos corchetes hay un identificador de la animación que es único y una región que define todas sus diapositivas.

Cada diapositiva, delimitadas en el espacio que comprenden las palabras reservadas begin y end, guarda una serie de propiedades de dibujado y las colisiones del luchador cuando esta reproduciendo esa diapositiva en concreto.

El formato para una diapositiva es el siguiente:

1. Datos generales de una diapositiva:
   - <x,y>, define el punto de referencia dentro del rectángulo seleccionado dentro de la imagen.
   - La rotación de la imagen:
     - N, no se rota.
     - V, se rota verticalmente la imagen.
     - H, se rota horizontalmente la imagen.
     - HV, se rota horizontal y verticalmente la imagen.
   - { x , w , y , h }, Rectángulo seleccionado dentro de la imagen y es lo que se muestra al usuario por pantalla, donde x e y son los puntos del eje cartesiano dentro de la imagen, w su ancho y h su altura.
2. Los distintos rectángulos de colisiones, estos toman como punto de partida el rectángulo definido anteriormente que sirve para mostrar la diapositiva, se define además una propiedad y se repite esta estructura hasta que se encuentre con la palabra clave end. El formato de una colisión es:
   - { x , w , y , h }, rectángulo que define una colisión dentro del rectángulo de la animación definido anteriormente.
   - <A | B >, tipo de colisión, si es A es que es de ataque con lo que daña al oponente, si es B representa la parte del luchador que puede colisionar cuando este ejecuta esta diapositiva.

Listado C.1: Fichero de animaciones con un solo estado

```
[ quit
        begin
                35,0 N { 8 , 55 , 23 , 148 }
                col:
                        { 8,40,0,120  } B
                        { 8,40,120,20 } B
        end
        begin
                35,0 N { 83 , 55 , 25 , 147 }
                col:
                        { 8,40,0,120  } B
                        { 8 , 40 , 120 , 20 } B
        end
]
```

Como se muestra en la figura C.1, se esta seleccionando el ancho de la diapositiva, este ancho seleccionado será el trozo de la imagen que se mostrará para una diapositiva dada dentro de una animación.

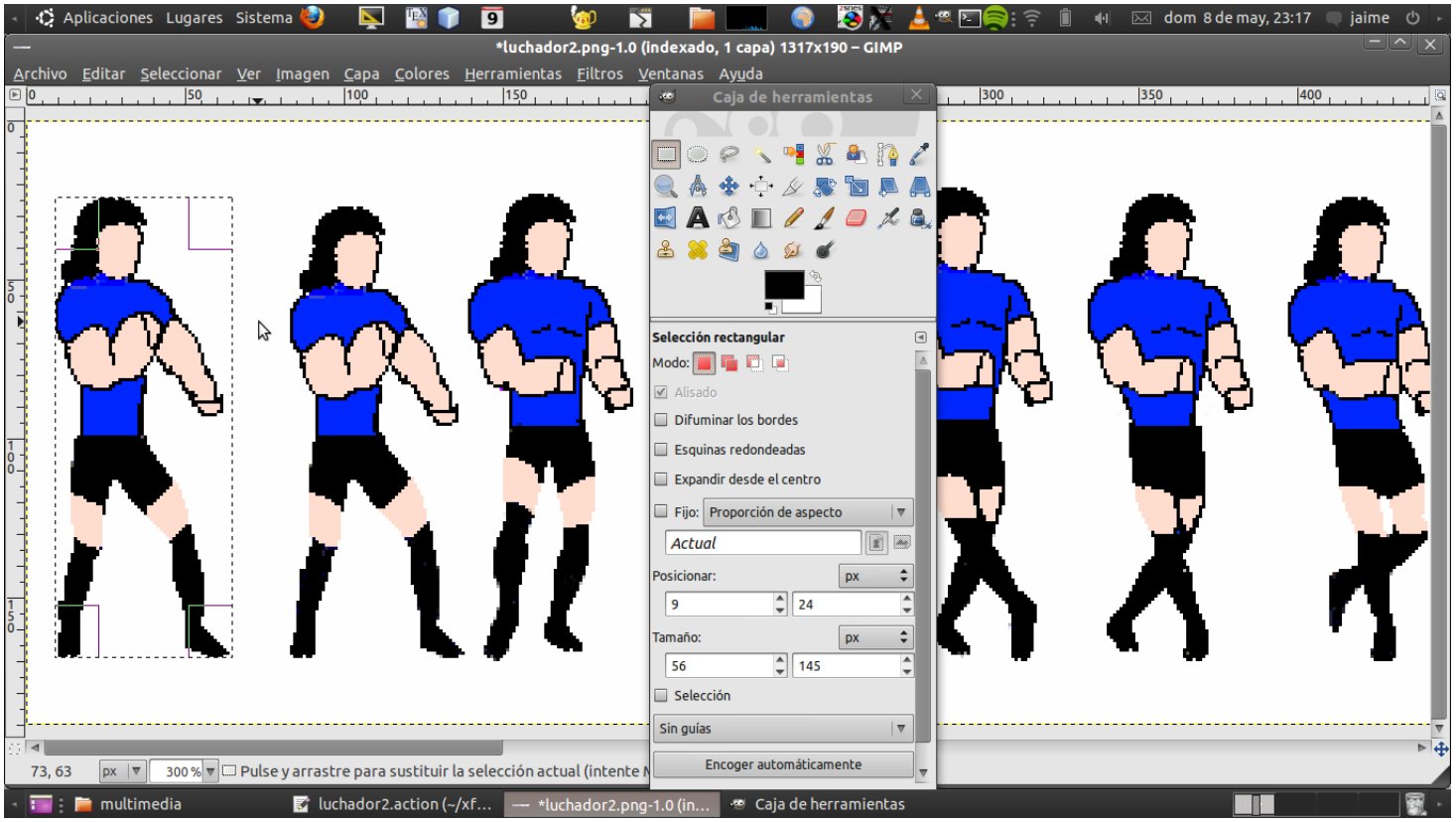


Figura C.1: Maquetación de un luchador

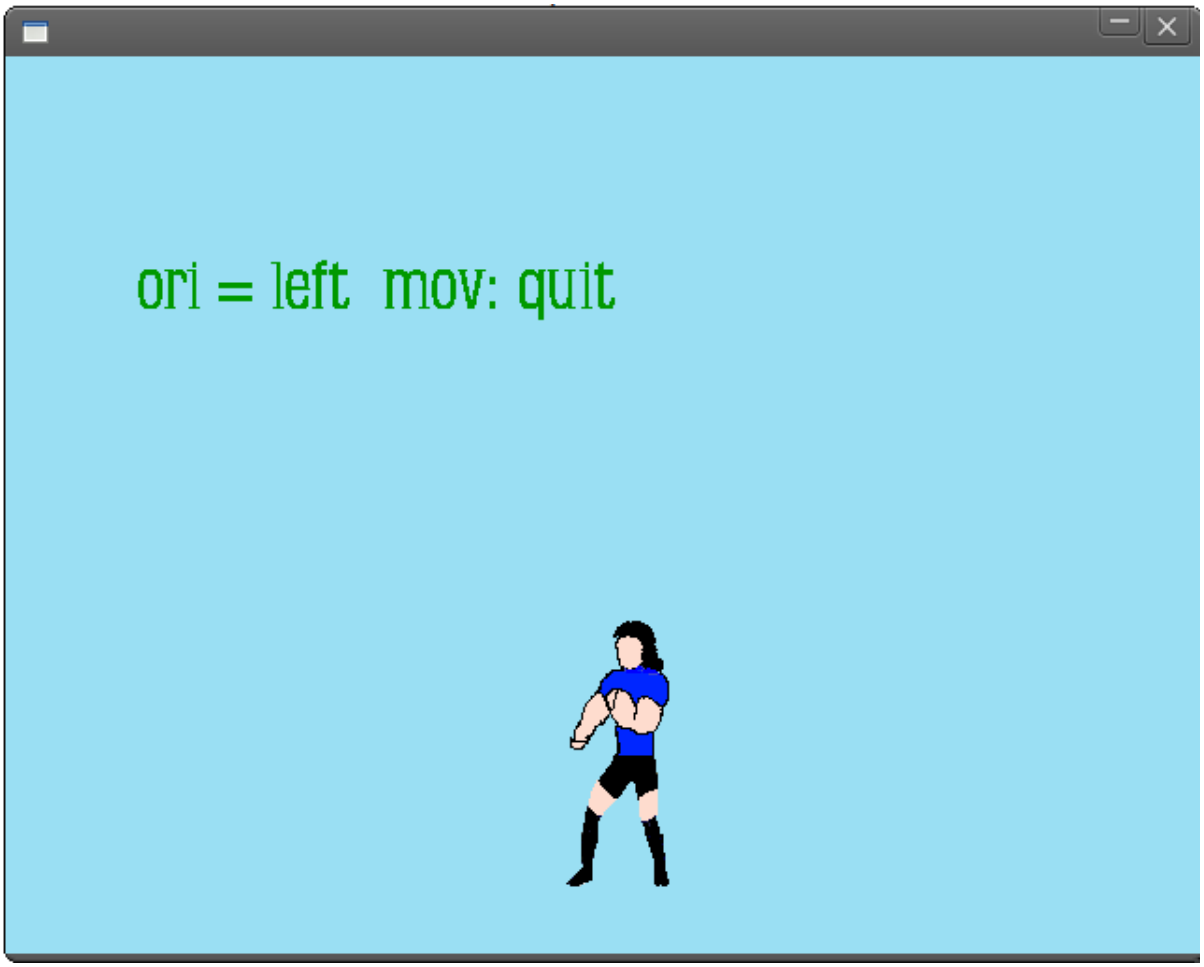


Figura C.2: Comprobación de una animación

Comprobación de la carga de una animación

Para comprobar un fichero action se ejecuta `test/exe_test.pl Sprite fichero.action -pr`. Esta prueba muestra por pantalla todos los estados de una animación, de manera que podemos comprobar el correcto visualizado de las animaciones, como muestra la figura C.2.

C.2. El luchador

Para un luchador llamado Ryu, este debe de tener los siguientes ficheros:

- **ryu.png:** Imagen del luchador.
- **ryu.fsm:** Fichero que declara los estados de la máquina finita de estados del luchador (hereda de global.fsm).
- **ryu.lua:** Script Lua que implementa el comportamiento del luchador (hereda de global.lua) .

- **ryu.action:** Fichero que contiene las animaciones y las colisiones.
- **ryu.eff:** Combinaciones de teclas y definición de las condiciones para su ejecución.
- **ryu.sfx:** Sonidos personalizados del luchador.

Para su maquetación hay que tener muy en cuenta que las siguientes animaciones deben de existir, en el caso que no existan no se verá correctamente la animación del luchador.

Animaciones que deben de existir en el fichero action:

- **transition:** Es la animación que representa cuando pasa de cualquier animación a quit.
- **move:** Es la animación que representa cuando se esta moviendo y esta en el suelo.
- **quit:** Es la animación que representa cuando esta quieto el luchador.
- **punch_hard:** Es la animación que representa cuando el luchador lanza un puñetazo fuerte.
- **punch_medium:** Es la animación que representa cuando el luchador lanza un puñetazo medio.
- **punch_slow:** Es la animación que representa cuando el luchador lanza un puñetazo lento.
- **kick_hard:** Es la animación que representa cuando el luchador lanza una patada fuerte.
- **kick_medium:** Es la animación que representa cuando el luchador lanza una patada medio.
- **kick_slow:** Es la animación que representa cuando el luchador lanza una patada lento.
- **couch:** Es la animación que representa cuando el luchador esta agachado.
- **couch_punch_hard:** Es la animación que representa cuando el luchador esta agachado y propina un puñetazo fuerte.
- **couch_punch_medium:** Es la animación que representa cuando el luchador esta agachado y propina un puñetazo medio.
- **couch_punch_slow:** Es la animación que representa cuando el luchador esta agachado y propina un puñetazo lento.
- **couch_kick_hard:** Es la animación que representa cuando el luchador esta agachado y propina una patada fuerte.
- **couch_kick_medium:** Es la animación que representa cuando el luchador esta agachado y propina una patada media.
- **couch_kick_slow:** Es la animación que representa cuando el luchador esta agachado y propina una patada lenta.
- **jump:** Es la animación que representa cuando el luchador salta.

- **jump_punch_hard:** Es la animación que representa cuando el luchador esta saltando y propina un puñetazo fuerte.
- **jump_punch_medium:** Es la animación que representa cuando el luchador esta saltando y propina un puñetazo medio.
- **jump_punch_slow:** Es la animación que representa cuando el luchador esta saltando y propina un puñetazo lento.
- **jump_kick_hard:** Es la animación que representa cuando el luchador esta saltando y propina una patada fuerte.
- **jump_kick_medium:** Es la animación que representa cuando el luchador esta saltando y propina una patada media.
- **jump_kick_slow:** Es la animación que representa cuando el luchador esta saltando y propina una patada lenta.
- **jump_move:** Es la animación que representa cuando el luchador salta y se mueve.
- **jump_move_punch_hard:** Es la animación que representa cuando el luchador esta saltando, avanzando y propina un puñetazo fuerte.
- **jump_move_punch_medium:** Es la animación que representa cuando el luchador esta saltando, avanzando y propina un puñetazo medio.
- **jump_move_punch_slow:** Es la animación que representa cuando el luchador esta saltando, avanzando y propina un puñetazo lento.
- **jump_move_kick_hard:** Es la animación que representa cuando el luchador esta saltando, avanzando y propina una patada fuerte.
- **jump_move_kick_medium:** Es la animación que representa cuando el luchador esta saltando, avanzando y propina una patada media.
- **jump_move_kick_slow:** Es la animación que representa cuando el luchador esta saltando, avanzando y propina una patada lenta.
- **beaten_couch:** Es la animación que representa cuando el luchador es golpeado y esta agachado.
- **beaten_quit:** Es la animación que representa cuando el luchador es golpeado.
- **beaten_jump:** Es la animación que representa cuando el luchador es golpeado y esta en el aire.
- **ko:** Es la animación que representa cuando el luchador esta en un estado de ko.
- **fallen_transition:** Es la animación que representa el paso de estar en el suelo y levantarse.

- **fallen:** Es la animación que representa el paso de estar en el aire, agachado o normal y caerte al suelo.
- **protect_quit:** Es la animación que representa protegerte de un golpe.
- **protect_couch:** Es la animación que representa protegerte de un golpe y estar agachado.
- **win** Es la animación que representa cuando un luchador ha ganado un combate.
- **lose** Es la animación que representa cuando un luchador ha perdido un combate.

C.3. Los proyectiles

Para un proyectil llamado hadoken, deben de existir estos ficheros:

- **hadoken.png:** Imagen del luchador.
- **hadoken.action:** Fichero que contiene las animaciones y las colisiones.

Además deben de existir los siguientes estados:

- **init:** Representa cuando el proyectil es lanzado.
- **exe:** Representa cuando el proyectil se esta moviendo por la pantalla.
- **destroy:** Representa cuando el proyectil colisiona y se destruye.

Listado C.2: Ejemplo de fichero action para un proyectil

```
[ begin 1
        begin
                20,0 N { 0 ,74,0 , 50}
                col:
                        { 0 , 74 , 0 , 50  } A
                        { 0 , 74 , 0 , 50  } B
        end
        begin
                20,0 N { 75 ,74,0 , 50}
                col:
                        { 0 , 74 , 0 , 50  } A
                        { 0 , 74 , 0 , 50  } B
        end
]
[ exe
        begin
                20,0 N { 125 ,74,0 , 50}
                col:
                        { 0 , 74 , 0 , 50  } A
                        { 0 , 74 , 0 , 50  } B
        end
        begin
                20,0 N { 200 ,74,0 , 50}
                col:
                        { 0 , 74 , 0 , 50  } A
                        { 0 , 74 , 0 , 50  } B
        end
]
[ destroy
        begin
                20,0 N { 270 ,74,0 , 50}
```

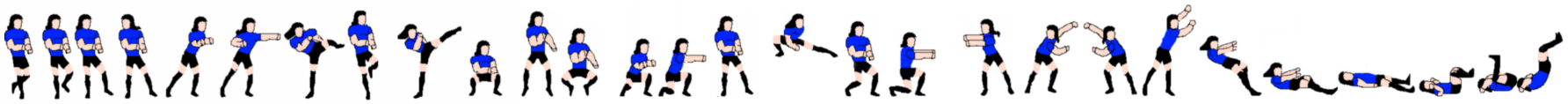


Figura C.3: Imagen de un luchador

```
                col:
                        { 0 , 74 , 0 , 50  } A
                        { 0 , 74 , 0 , 50  } B
        end
        begin
                20,0 N { 350 ,74,0 , 50}
                col:
                        { 0 , 74 , 0 , 50  } A
                        { 0 , 74 , 0 , 50  } B
        end
]
```

El archivo de configuración del escenario se encargará de personalizar los elementos que aparecen en un escenario, desde la propia imagen del escenario, hasta las posición de la barras de cargas o las animaciones. En esta sección se explicará el contenido de un fichero de configuración.

C.4. El escenario

El escenario es una parte muy importante del simulador, es el lugar donde se desarrolla un combate que estará compuesto de un archivo de configuración, una imagen y una serie de animaciones.

El fichero de configuración se compone de dos partes diferenciadas delimitadas por: [[GENERAL]] y [[SPRITES]], en la primera se configuran los elementos del escenario (las barras de vida, el cronómetro, la posición del escenario, etc.) y en la segunda las animaciones del escenario.

Hay que tener en cuenta que las animaciones están situadas en una posición relativa al escenario, de manera que una animación en el punto (800,0) no lo está de la pantalla sino del punto (x,y) donde se encuentra situada la imagen del escenario.

Listado C.3: Fichero de configuración de un escenario

```
[[GENERAL]]
picture= scene.png
position= -640 0
track= scene1.ogg
progressbar1= 40 10
progressbar2= 350 10
chronometer= 1 50
[[SPRITES]]
fuego1 400 300
fuego1 740 300
```

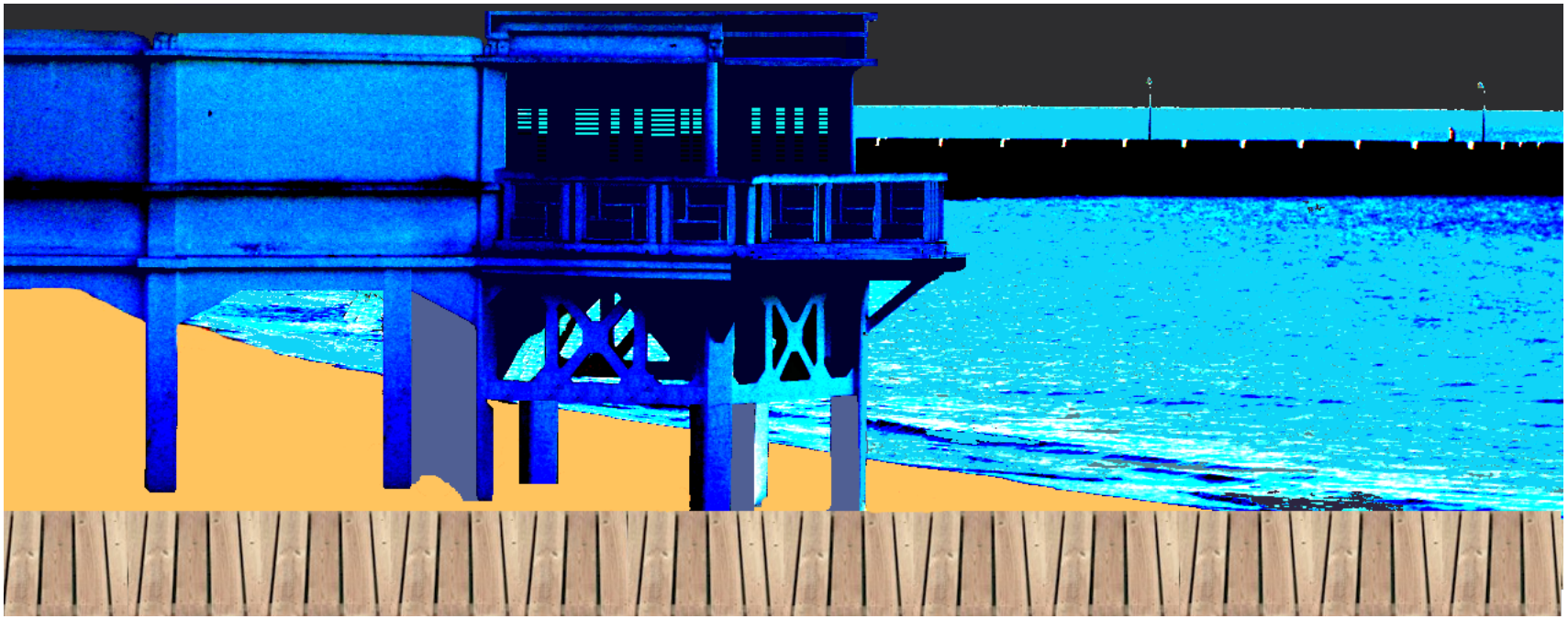


Figura C.4: Imagen de un escenario

Bibliografía

[1] Definición de juego de lucha según Wikipedia, http://en.wikipedia.org/wiki/Fighting_game

[2] Lenguaje Lua según Wikipedia, http://en.wikipedia.org/wiki/Lua_(programming_language)

[3] Stroustrup, Bjarne,The C++ programming lenguage, Addison-Wesley, 1997.

[4] Blog especializado en C++, http://www.lonecpluspluscoder.com/

[5] Programming Game AI by Example, Mat BuckLand, WorWare game developers's library, 2003.

[6] The C programming lenguage,Brian W. Kernighan & Dennis M. Ritchie, Prentice-Hall, 1988.

[7] Sitio oficial de luabind, Daniel Wallin, Arvid Norberg, http://www.rasterbar.com/products/luabind/docs.html

[8] manual de referencia, R. Ierusalimschy, L. H. de Figueiredo, W. Celes, http://www.lua.org/docs.html#manual

[9] Sitio oficial de libSDL, http://www.libsdl.org/

[10] Sitio sobre el lenguaje de programación C++, http://www.cplusplus.com/

[11] Sitio oficial de cppunit, http://www.cs.nmsu.edu/~jeffery/courses/371/cppunit/index.html

[12] Tutorial online para iniciarse en la programación usando libSDL, http://lazyfoo.net/SDL_tutorials/index.php

[13] Sitio oficial de liblogcpp, http://log4cpp.sourceforge.net/

[14] Wiki sobre cppunit, http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=Main_Page

[15] Wiki sobre libSDL, http://softwarelibre.uca.es/wikijuegos/

[16] Thinking in C++ 2nd Edition, Bruce Eckel, http://www.mindview.net/Books/DownloadSites/

[17] Sitio con tutoriales y ejemplos de Lua, http://lua-users.org/

[18] Manual sobre LaTeX, http://www.fceia.unr.edu.ar/lcc/cdrom/Instalaciones/LaTex/latex.html

[19] Blog especializado en C++, autor: Sumant Tambe, http://cpptruths.blogspot.com/

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently

incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.