Estudio de los estándares gráficos actuales: desarrollo de un video juego de plataformas utilizando las tecnologías OpenGL y DirectX

Proyecto fin de carrera

Michael Harry O'Gay García
Tutor: Agustín Trujillo Pino

Las Palmas de Gran Canaria, Junio de 2015

Agradecimientos

Agradezco a mi hermano, Alexander, por ayudarme en el diseño del juego y por dibujar todas las texturas visibles en el escenario de juego. Y a mi tutor, Agustín Trujillo Pino por tutorizar el proyecto.

Índice

1. Introducción

1.1 Motivación y objetivo del proyecto

En verano del año 2012, Nintendo anunció un nuevo juego de su famosa serie, Super Smash Bros. Esta es una serie de juegos de pelea y plataformas, protagonizada por personajes de varios distintos juegos de Nintendo, donde el objetivo es lanzar a los rivales fuera de un escenario. El escenario se compone de plataformas en dos dimensiones, de vista lateral.

Se me ocurrió la pregunta ¿Añadirán esta vez una tercera dimensión a los escenarios?¿Cómo sería un juego de Super Smash Bros. en tres dimensiones? La respuesta a la primera pregunta fue negativa. La respuesta de la segunda se convirtió en este proyecto. Comencé el desarrollo de un prototipo al principio del verano de 2012. El proyecto se realizaría en Visual Studio 2008, utilizando DirectX para los gráficos y la entrada de los mandos.

1.2 Continuación del proyecto como proyecto fin de carrera

Debido al inicio un nuevo curso, tuve que dejar de trabajar en el proyecto en septiembre. En enero de 2013, decidí aprovechar el proyecto fin de carrera para poder seguir trabajando en el proyecto. Siendo un trabajo personal personal, no estaba bien organizado. Decidí que comenzaría de nuevo el proyecto, utilizando OpenGL para los gráficos.

Propuse el proyecto al tutor Agustín Trujillo Pino. Él propuso aprovechar el trabajo realizado en 2012 para realizar un estudio sobre las diferencias entre Direct3D y OpenGL. Con esto comenzó el proyecto tal como está definido en este documento.

1.3 Objetivos del proyecto

Los objetivos son los siguientes

- Crear un juego como el que se describe en la introducción y que se detallará en apartados posteriores.
- Hacer un estudio sobre las diferencias básicas entre las tecnologías OpenGL y DirectX.

2. Estado del arte

2.1 Estándares gráficos

Hoy en día existen dos estándares principales de APIs gráficos de bajo nivel: DirectX y OpenGL.

2.1.1 DirectX

La historia de DirectX está fundamentalmente ligada a la industria de los video juegos. La creación de DirectX comenzó en el año 1994. Con el futuro lanzamiento de Windows 95, Microsoft tenía que convencer a los desarrolladores de juegos que Windows 95 era una mejor plataforma para juegos que su sistema operativo anterior, MS-DOS. DOS daba a los programadores acceso directo a todos los dispositivos de entrada y salida, como las tarjetas gráficas, teclados, ratones y altavoces mientras que Windows limitaba el acceso a estos dispositivos.

Tres empleados de Microsoft - Craig Eisler, Eric Engstrom y Alex St. John - implementaron una solución a este problema. La solución era el conjunto de APIs llamado DirectX. Se lanzó la primera versión de DirectX en septiembre de 1995 como Windows Games SDK. Con el lanzamiento de Windows 95, DirectX 2.0 se convirtió en una parte de Windows a partir de 1996. Direct3D se introdujo inicialmente como una alternativa a OpenGL, más ligera y diseñada específicamente para el desarrollo de juegos. Pero más tarde se convirtió en un competidor de OpenGL.

Fuente: *http://en.wikipedia.org/wiki/DirectX*

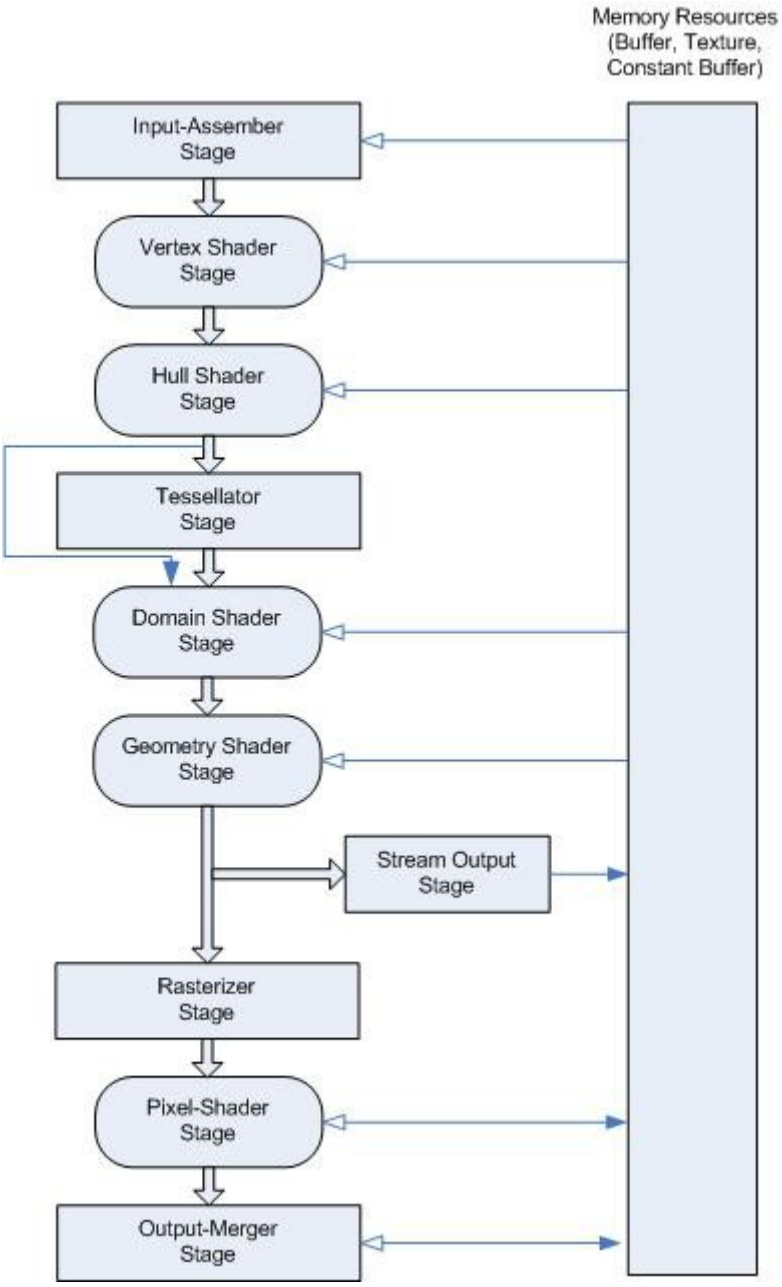
DirectX es un conjunto de APIs, desarrollado por Microsoft, para la creación de video juegos y otras aplicaciones de multimedia de alto rendimiento. Se compone de los siguientes APIs

- **Direct2D:** Un API para la renderización inmediata de geometría 2D, mapas de bits y texto. Es de alto rendimiento y utiliza aceleración hardware. Se incorporó en DirectX 10.1
- **Direct3D:** Un API para la renderización de geometría tridimensional.
- **DirectWrite:** Un API para la renderización de texto de alta calidad, independiente de la resolución y compatible con Unicode.
- **DirectXMath:** Una interfaz para aritmética y álgebra lineal con vectores de dos, tres y cuatro dimensiones y matrices de tamaño 3x3 y 4x4.
- **DirectSound:** Un API para el procesamiento de señales de audio. Fue reemplazada por XAudio2
- **DirectInput:** Un API para el manejo de entradas de datos, como el ratón, teclado y mandos de juego. Fue reemplazada por XInput, que sirve para manejar la entradas de mandos de XBox 360.

Fuente: *https://msdn.microsoft.com/en-us/library/ee663274%28v=vs.85%29.aspx*

En este proyecto, se centra en la comparación de **Direct3D** con OpenGL. El objetivo principal de Direct3D es renderizar una multitud de triángulos, líneas y puntos en cada instante. Para ello, se hace uso de una GPU. El GPU es una unidad de procesamiento diseñado para renderizar gráficos aprovechando la independencia entre vértices y entre píxeles para realizar procesamiento paralelo. Direct3D proporciona una abstracción del GPU de manera que el programador no necesita crear una versión del programa para cada tarjeta gráfica.

Fuente: *https://msdn.microsoft.com/en-us/library/hh769064%28v=vs.85%29.aspx*



Secuencia de pasos en la renderización de gráficos

Fuente: *https://msdn.microsoft.com/en-us/library/ff476882%28v=vs.85%29.aspx*

2.1.2 OpenGL

En los años 80, los desarrolladores implementaban interfaces especializadas para cada tipo de tarjeta gráfica. Esto multiplicada de manera innecesario el esfuerzo y el coste de sus proyectos. A principios de los años 90, Silicon Graphics Inc. dominaba el mercado de los gráficos 3D con su API gráfico, IRIS GL. IRIS GL eclipsó a su competidor, PHIGS, por que era más fácil de usar y permitía la renderización inmediata de imágenes. Los competidores empezaron a traer gráficos 3D al mercado realizando extensiones a PHIGS, reduciendo a influencia que tenía SGI. En un intento de influencia de nuevo el mercado, SGI convirtió IRIS GL en un estándar abierto llamado OpenGL.

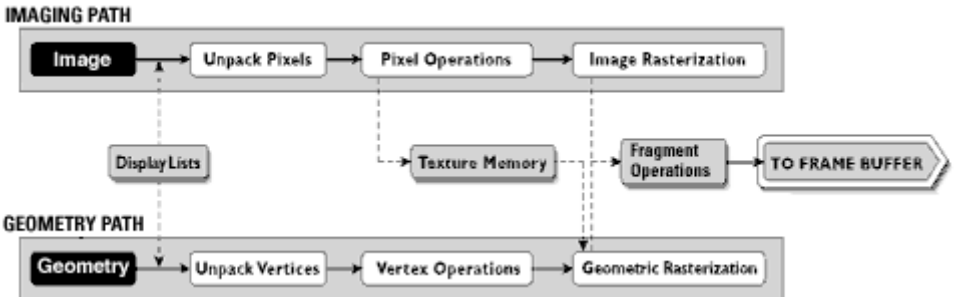
La conversión no fue inmediata. Además de librerías gráficas, IRIS GL contenía librerías para el manejo de teclados, ratones y ventanas. Por motivos de licenciación y patentes, estas librerías no podían convertirse en estándares abiertos.

Una limitación de IRIS GL fue que solo daba acceso a funcionalidades soportadas por el hardware. OpenGL consiguió resolver el problema implementando versiones software para las funcionalidades no disponibles en el hardware. OpenGL estandarizó el acceso al hardware, movió la responsabilidad de crear drivers a los desarrolladores del hardware y la de crear ventanas a los desarrolladores de sistemas operativos. En 1992, SGI creó el OpenGL Architecture Review Board (OpenGL ARB), un grupo de empresas que mantendrían y expandirían la especificación de OpenGL.

En 1995, Microsoft lanzó su API gráfico contenido en DirectX, que se convirtió en el competidor principal de OpenGL. En 1997, Microsoft y SGI se reunieron para intentar unificar las dos librerías, pero debido a problemas económicas y estratégicas, nunca se llegó a un acuerdo. Desde julio de 2006, el OpenGL ARB transfirió en control de OpenGL a Khronos Group.

Fuente: *http://en.wikipedia.org/wiki/OpenGL*

Hoy en día, OpenGL es el estándar gráfico más utilizado en el mundo. Sirve para crear aplicaciones con renderización de gráficos 2D y 3D. Incluye un conjunto de funciones para la renderización, mapeado de texturas, efectos especiales y otras efectos visuales. Es portable, permitiendo la renderización de gráficos independiente del lenguaje, del hardware y del sistema operativo. Utiliza tipos de datos primitivos, para que cualquier otra librería pueda manipular fácilmente las texturas y geometrías.



Secuencia de pasos para la renderización de gráfico en OpenGL

Fuente: *https://www.opengl.org/about/*

2.1.3 Comparación previa

Ambas librerías están diseñadas para renderizar geometrías tridimensionales, abstrayendo al programador del hardware gráfico y del sistema operativo. Direct3D se enfoca más en gráficos 3D, delegando la renderización de gráficos 2D y texto a Direct2D y DirectWrite. En cambio, OpenGL está diseñado para la rederización de todos los elementos gráficos, tanto 2D como 3D.

Aunque existen implementaciones en versiones de Mac y Linux, DirectX fue diseñado específicamente para Windows. OpenGL es independiente del sistema operativo.

OpenGL es una estándar abierto. Consiste en una serie de funciones para manejar los gráficos, pero todos los datos utilizados son primitivos. DirectX contiene un conjunto de objetos y funciones propietarios. El SDK de DirectX se puede descargar de manera gratuita, pero es un estándar cerrado y se distribuye en forma de ficheros binarios.

El proceso de renderización de OpenGL es más sencillo que el de DirectX. Marca una separación clara entre la renderización de gráficos 2D y 3D.

2.2 Video juegos

La industria de los video juegos es grande y variado. Podemos clasificarlo según el dispositivo destinatario del juego: Ordenadores personales, consolas o dispositivos móviles.

En el mercado de las consolas, las mayores empresas se han dedicado a producir juegos cinemáticos, enfocando principalmente en recrear la experiencia de las películas. Esta concentración en los detalles gráficos ha hecho que los presupuestos lleguen a decenas de millones de dólares. Para amortizar el coste, se suelen vender los juegos por un precio base de aproximadamente 60€ y luego vender contenido adicional descargable (DLC) por precios más bajos, entre 5€ y 30€.

En el mercado de los juegos móviles, la mayoría de las empresas crean juegos cortos y adictivos. Los juegos se ofrecen en plataformas virtuales, como iOS y Google Play. El modelo económico más popular se llama Free To Play, conocido más comúnmente como F2P. Consiste en permitir la descarga gratuita del juego base y poner un pequeño coste adicional a elementos del juego. Estos elementos pueden ser contenido descargable o bien elementos consumibles, los cuales desaparecen una vez que se usan en el juego. A estos costes adicionales se les llama micro-transacciones.

Debido a la facilidad con la que se puede realizar y descargar copias ilegales de software, la piratería supone un grave problema para la industria de los juegos de ordenador personal. Para solucionar el problema, han surgido las plataformas virtuales, donde los juegos están ligados a cuentas de usuario en línea. La venta y distribución de juegos, actualizaciones y contenido descargable se ha vuelto más sencillo. Un ejemplo es Steam, de Valve Corporation. Steam ofrece servicios tanto para desarrolladores como para los usuarios. Gracias a este nuevo sistema, los desarrolladores independientes tienen acceso al mismo mercado que las grandes empresas.

3. Análisis

3.1 Requisitos

3.1.1 Requisitos del proyecto

- Un ordenador con Windows.
- El entorno de programación Visual Studio 2012.
- Las librerías gráficas OpenGL y DirectX
- Al menos un mando de juego y un teclado

3.1.2 Requisitos del juego

- Un ordenador con sistema operativo que sea compatible con OpenGL 3, DirectX 9.0.
- El sistema operativo Windows XP, Windows Vista o Windows 7.
- Al menos un mando de juego compatible con DirectX y un teclado.

3.2 Herramientas de desarrollo

3.2.1 Lenguaje

Se ha elegido el lenguaje C++ para realizar el proyecto. Es un lenguaje de nivel relativamente bajo y un poco complicado de dominar, pero da mucho poder a programadores familiarizados con el lenguaje y es compatible con muchas librerías, incluyendo DirectX y OpenGL.

3.2.2 Entorno de desarrollo

El entorno de desarrollo utilizado fue Visual Studio Express. Facilita bastante el manejo de proyectos. Permite mantener organizados los ficheros de código y tiene una función que permite auto-completar el código mientras se escribe. También tiene un depurador avanzado que permite ver e incluso manipular valores de las variables durante la ejecución.

Logotipo de Microsoft Visual Studio

Se utilizó inicialmente el entorno Visual Studio Express 2008 para realizar el prototipo de DirectX. Para hacer el prototipo en OpenGL se cambió a Visual Studio 2010. Finalmente se pasó a Visual Studio 2012 más tarde. Los dos prototipos siguen siendo compatibles con las versiones de Visual Studio en los cuales fueron creados.

Logotipo de Microsoft Visual Studio Express 2010

Logotipo de Microsoft Visual Studio Express 2012

3.2.3 Otras herramientas

Para descomprimir imágenes en formato PNG, se ha utilizado la pequeña librería, LodePNG,
http://lodev.org/lodepng/

3.3 Motores gráficos

3.3.1 Direct3D

Para el primer prototipo, se utilizó el motor gráfico Direct3D de la librería DirectX 9.0. Se eligió esta versión de DirectX para poder tener la versión más nueva de la librería que todavía fuera compatible con Windows XP - un sistema operativo que todavía se usaba en el año 2012.

Logotipo de Microsoft DirectX

3.3.2 OpenGL

Para el segundo prototipo, se utilizó el motor gráfico OpenGL 3. Se utiliza la técnica de VBOs para referenciar a objetos en la memoria gráfica y VAOs para almacenar información sobre objetos renderizables.

Logotipo de OpenGL

3.4 Entradas de usuario

3.4.1 DirectInput

DirectInput es la librería de DirectX para manejar entradas de datos del usuario, tales como mandos de juego, ratones y teclados, aunque para el ratón y el teclado se recomienda utilizar librerías más básicas. Se utilizó esta librería para manejar la entrada de mandos.

3.4.2 XInput

XInput se incluyó en DirectX 9.0. Su función es simplificar el acceso a mandos de la XBox 360 y otros mandos que tengan un modo compatible con los mandos de XBox 360, como el modo X de los mandos de Logitech. En este proyecto, se utilizó XInput para manejar aquellos mandos que estén en el modo X.

4. Diseño

El proyecto es un juego de lucha y plataformas. El juego toma lugar sobre un grupo de plataformas flotantes, donde entre 2 y 4 personajes intentarán lanzar a sus oponentes fuera de las plataformas. Cada personaje tiene un número de vidas y un porcentaje de daño que determina a qué velocidad serán lanzados al ser golpeados. Cuando un personaje llegue a los límites del escenario, perderán una vida. Si un personaje pierde todas sus vidas, no podrá seguir jugando. El ganador es el último personaje en pie cuando todos los demás personajes hayan sido derrotados.

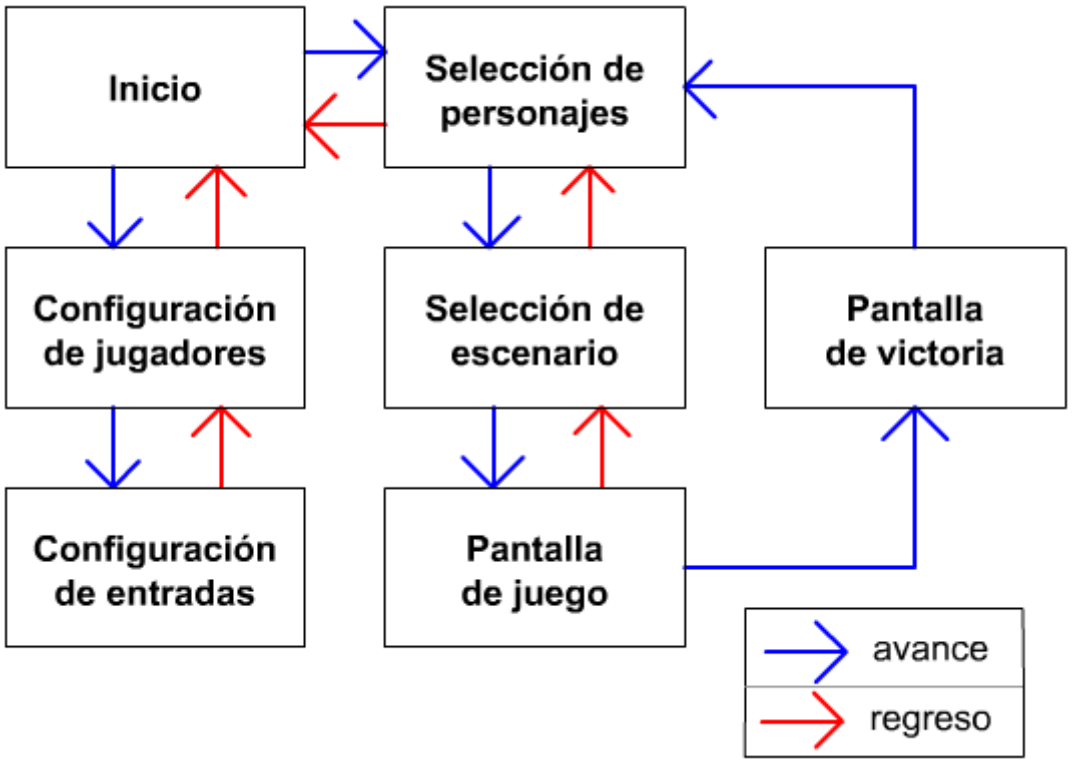
4.1 Diseño del menú

El juego hará uso únicamente de las entradas de mando o del teclado. No se usará el ratón. Exceptuando en las pantallas de selección de personajes, configuración de entradas y en la pantalla del juego, solamente el jugador 1 podrá controlar el juego.

La descripción de los dispositivos de entradas y sus funciones se explicará en el apartado **4.6**. A menos que se diga lo contrario, esta es la funcionalidad principal del menú:

En cada pantalla se hará uso de los sticks analógicos o teclas de dirección para moverse entre elementos del menú. Para activar la función de un elemento seleccionado, se pulsará el botón **A**. Para volver atrás en un menú, se pulsará el botón **B** en el mando o se activará el botón "Atrás" que habrá en cada pantalla. En algunas pantallas, el botón **START** servirá para avanzar, sea cual sea el elemento seleccionado en la pantalla. En pantallas donde esto no se aplica, el botón **START** actúa del mismo modo que el botón **A**.

En la siguiente imagen se representa el diagrama de pantallas de menú del juego.



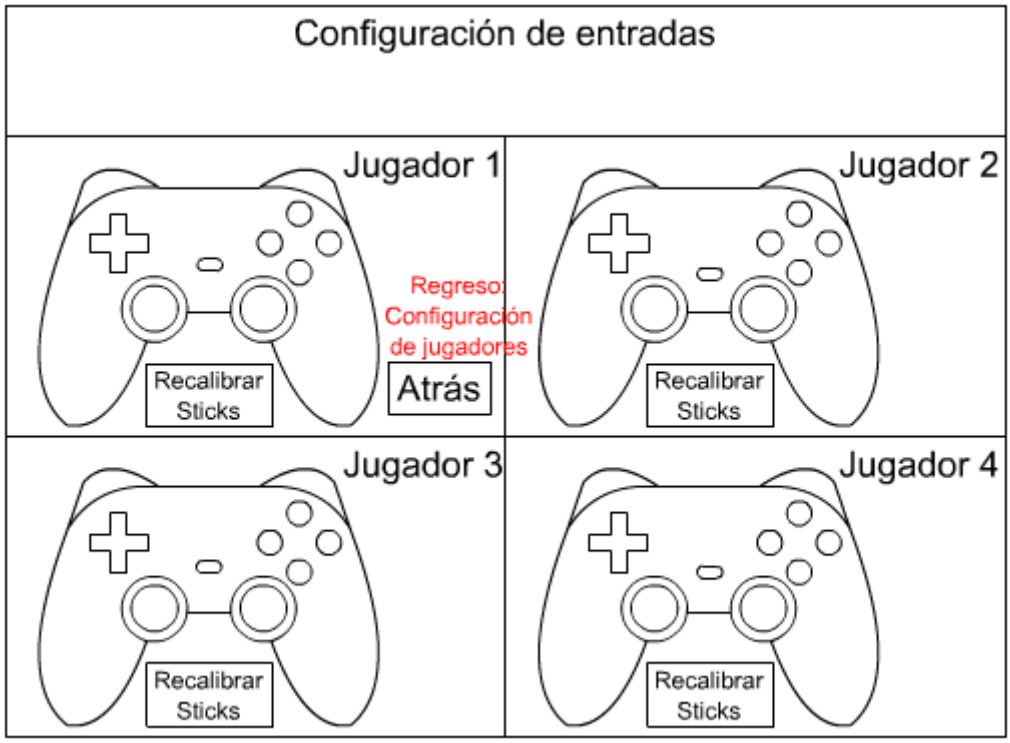
4.1.1 pantalla de inicio

La pantalla de inicio es la primera pantalla que aparece cuando comienza el programa del juego. Tiene tres botones que llevan a otros elementos del menú.

4.1.2 Pantalla de configuración de jugadores

La pantalla de configuración de jugadores servirá para activar o desactivar jugadores y para intercambiar entre los jugadores los dispositivos de entrada. El jugador 1 se encargará de hacer los cambios. Puede seleccionar un mando con el botón **A** e intercambiar su hardware con otro jugador pulsando **A** sobre el jugador deseado. Los cambios son inmediatos. Por lo tanto, si el jugador 1 cambia su propio dispositivo hardware con otro, pierde su habilidad de seguir haciendo cambios con ese mando.

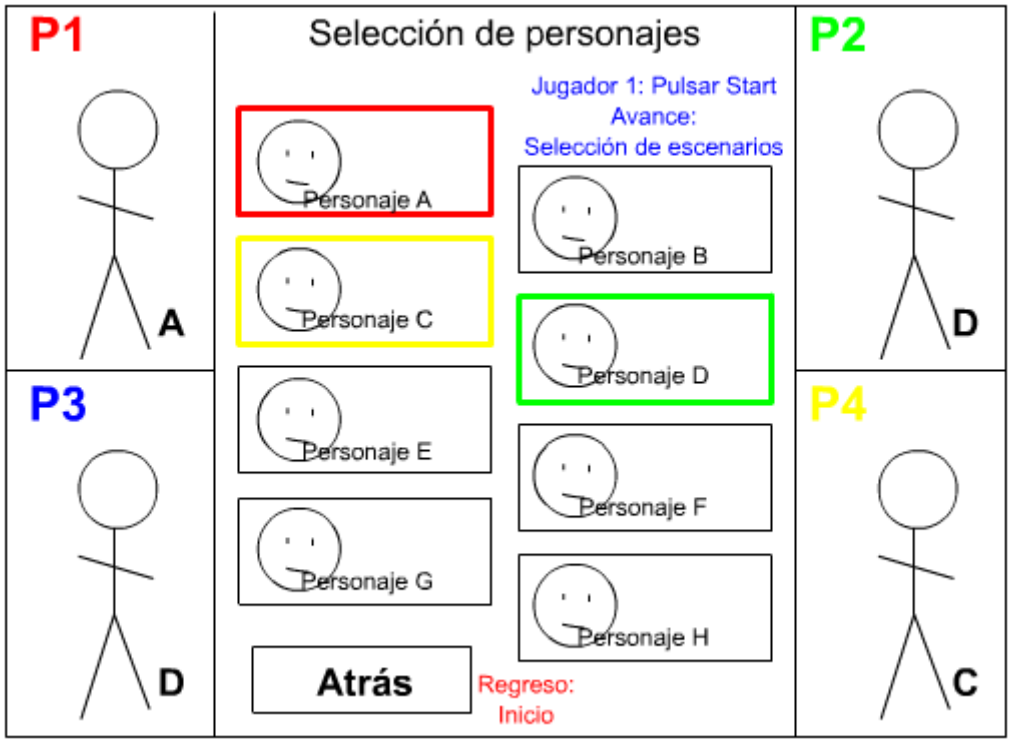
4.1.3 Pantalla de configuración de entradas



En esta pantalla todos los jugadores pueden controlar el menú, pero cada jugador está limitado a su zona. El jugador 1 tiene un botón más que le permite regresar al menú anterior, puesto que el botón B no implementará esta acción. En cada zona, un jugador puede hacer tres cosas:

- **Comprobar el funcionamiento de los botones:** Pulsando cualquier botón en el mando o teclado real iluminará el botón correspondiente en la imagen del mando.
- **Cambiar la asignación de los botones:** El jugador podrá cambiar el mapeado de botones del mando real (o teclas del teclado real) sobre el mando virtual.
- **Recalibrar los sticks analógicos:** Sirve para mandos que están un poco descentrados. El jugador mueve el cursor sobre el botón “Recalibrar mandos” y pulsa **A** sin mover los sticks analógicos para considerar su posición actual como a la posición 0.

4.1.4 Pantalla de selección de personajes.



En esta pantalla, cada jugador activo y con un dispositivo de entrada asignado tendrá un marcador. Los colores de los marcador son fijos: Rojo, verde, azul y amarillo para los jugadores de 1 a 4 respectivamente. Puede haber de 1 a 4 marcadores en la pantalla a la vez, dependiendo del número de jugadores activos.

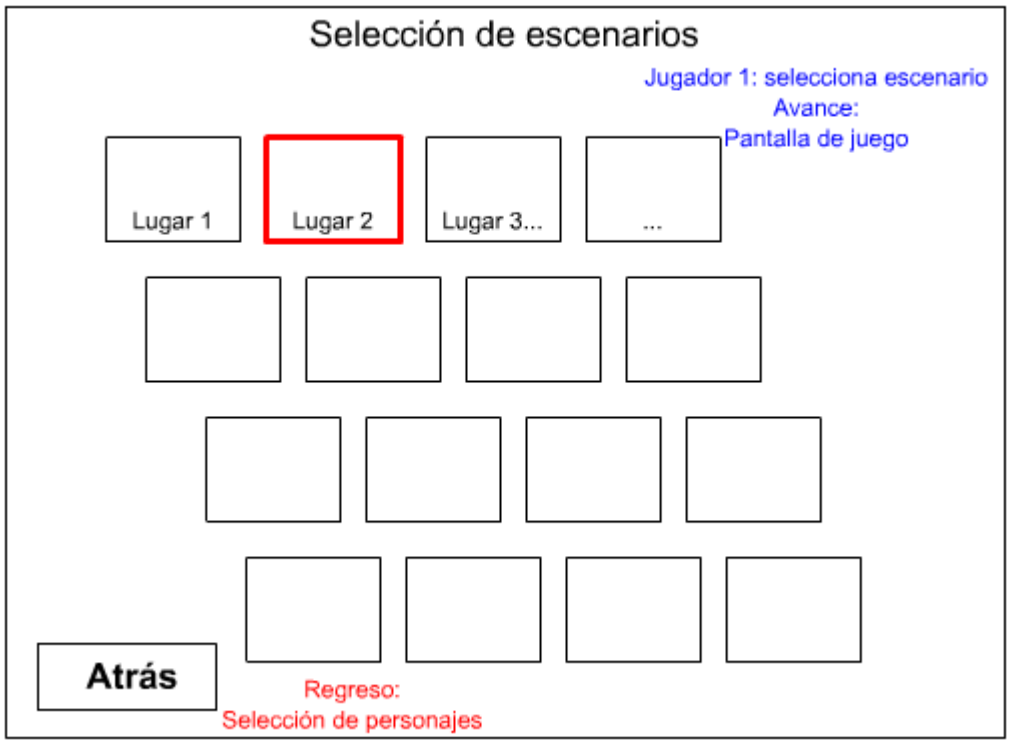
Cada jugador tiene control sobre su marcador. Y puede desplazarlo a través de la lista de personajes para seleccionar el que desea. Si uno o más marcadores se encuentran sobre el mismo icono de un personaje, la visibilidad de cada marcador se va alternando periódicamente. Todos los marcadores comienzan, inicialmente, sobre el personaje 1.

El jugador 1 tiene, además, la opción de mover su marcador sobre el botón “Atrás” para salir del menú. Alternativamente, puede pulsar **B** en cualquier momento para conseguir el mismo efecto.

Cuando un jugador cualquiera pulse **A** con su marcador sobre un personaje, una imagen más detallada de ese personaje aparecerá dentro de la caja perteneciente al jugador que lo ha seleccionado. Ahora se considera que el jugador ha elegido un personaje. Aun así, el jugador puede seguir moviendo su marcador y seleccionar otro personaje.

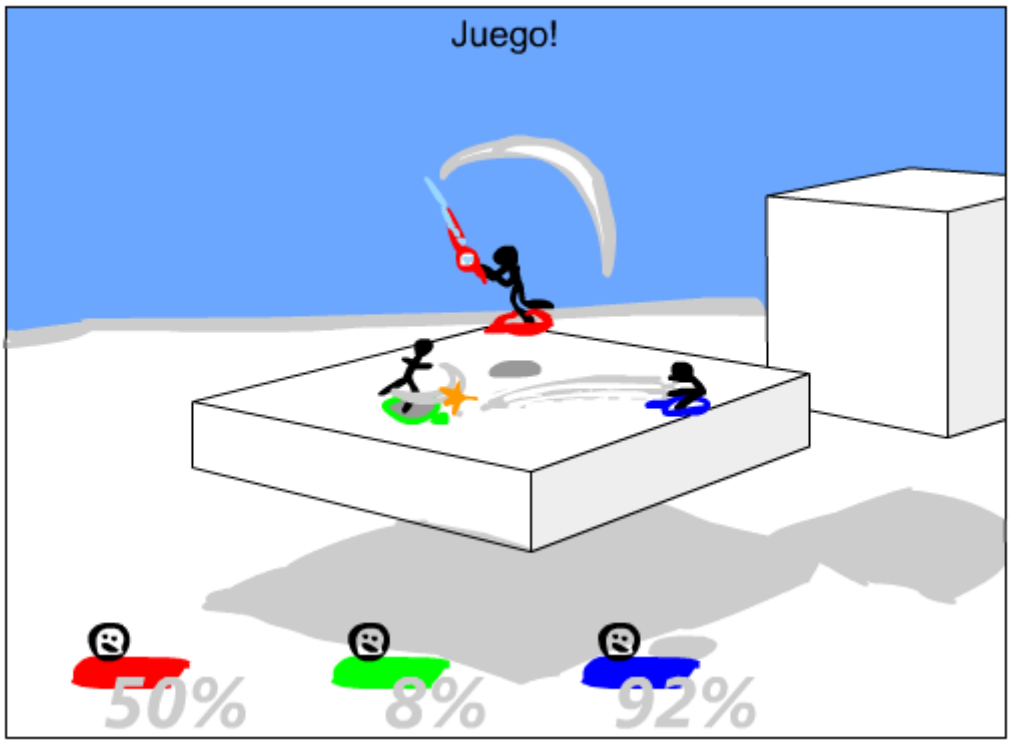
Esta es la última pantalla a la que se puede acceder con un solo jugador. Para poder avanzar más allá de esta pantalla el jugador 1 debe pulsar el botón **START** y al menos dos jugadores tienen que haber seleccionado un personaje.

4.1.5 Pantalla de selección de escenarios



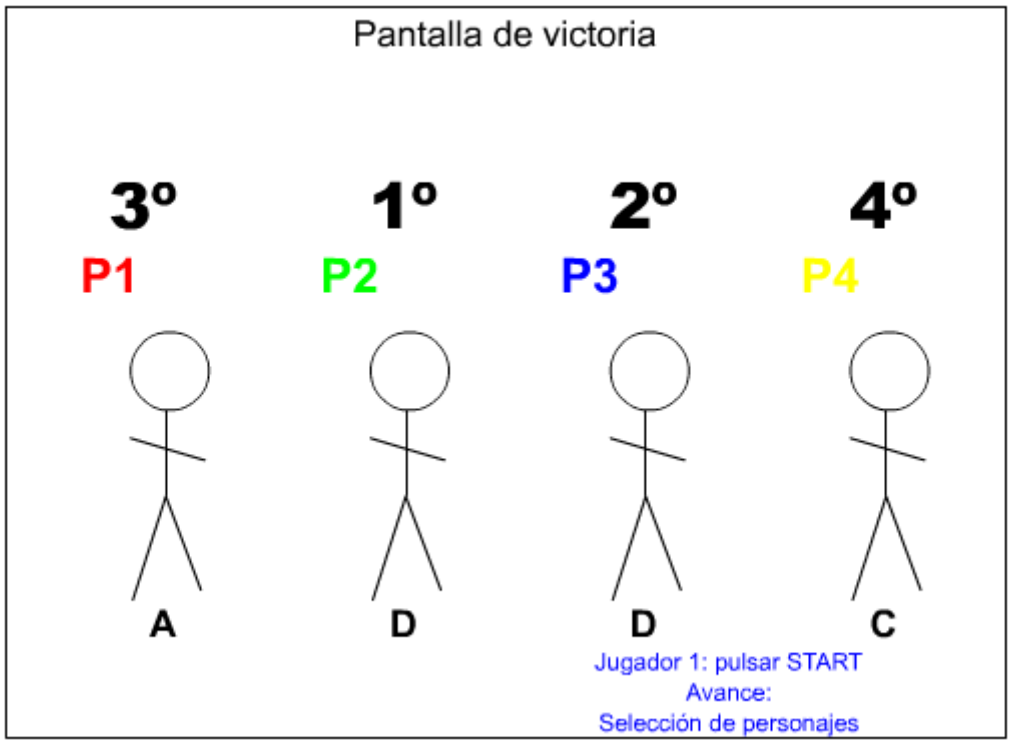
En esta pantalla, el jugador 1 podrá mover su marcador sobre los iconos de los distintos escenarios. Pulsando **A** o **START**, elegirá ese escenario y saltará directamente a la pantalla de juego.

4.1.6 Pantalla del juego



Esta pantalla contiene el juego. Hasta 4 jugadores pueden jugar. Al finalizar el juego, se cambia a la pantalla de victoria. Se considera la posibilidad de poder abandonar el combate cuando el juego está pausado.

4.1.7 Pantalla de victoria



En esta pantalla se muestra los resultados del combate, mostrando las posiciones de los jugadores en orden de mejor a peor. También podrían aparecer otros datos recopilados durante el juego.

4.2 Diseño de personajes

4.2.1 Gráficos

Durante el combate, los personajes se representan como un rectángulo con una textura animada, cuyo centro está localizado en el punto inferior central del rectángulo. El rectángulo siempre está mirando hacia la cámara, y existen texturas animadas para 4 diferentes ángulos: Delante, detrás, izquierda y derecha.

4 ángulos de vista para el personaje

4.2.2 Acciones

Los personajes tienen esta serie de acciones, cada una con su animación correspondiente. Las acciones básicas son: Quedarse de pie, caminar, correr, saltar, caer, colgarse de un borde y ponerse un escudo.

Además de estas acciones, cada personaje tiene 11 ataques normales, 4 ataques especiales y un ataque de agarre. Estos ataques se describirán más adelante, en el apartado **4.5**.

Finalmente, existen 4 estados más, cada uno con su animación: Agarrado por otro jugador, Aturdido por un golpe, volando a causa de un golpe fuerte y derribado en el suelo después de recibir un golpe.

4.2.3 Atributos de cada personaje

A continuación hay una lista de todos los atributos importantes para cada personaje. Muchos de estos atributos se explicarán en más detalle en el apartado **4.5**.

- **Nombre:** Una ristra literal con el nombre propio del personaje.
- **ID:** Un identificador entero, positivo y único entre todos los personajes para el personaje.
- **Velocidad máxima:** La máximo velocidad que puede correr el personaje controlado por un jugador.
- **Velocidad de salto:** La velocidad vertical inicial que tienen el personaje cuando empieza a saltar.
- **Anchura y altura:** Un número indicando la forma del personaje para aplicar la física del juego. La anchura se aplica por igual en las dos dimensiones horizontales, X y Z.
- **Peso:** Un multiplicador para la velocidad al ser golpeado. Cuando mayor sea este número, más rápidamente se irá el personaje al ser golpeado.
- **Distancia de agarre:** La distancia máxima que debe estar otro jugador para poder agarrarle.
- **Comienzo de agarre:** Identificador del primer marco de animación en el cual el personaje intenta agarrar a un enemigo.
- **Fin de agarre:** El marco de animación en donde se termina de intentar agarrar a otro personaje.
- **Velocidad mínima de correr:** La velocidad mínima que debe tener el personaje en el suelo para que se vea la animación de correr.

4.3 Diseño de escenarios

Un escenario es un conjunto de plataformas en donde lucharán los personajes. Se compone de varias partes, que se explican a continuación.

4.3.1 Atributos

El escenario tiene varios datos además de los elementos que lo componen. Aquí está la lista de los atributos:

- **Imagen de fondo:** El identificador de una imagen que se utilizará como fondo.
- **Puntos de comienzo:** Una lista de 4 puntos con coordenadas (X, Y, Z). Cada punto servirá para posicionar a cada personaje al comienzo de una batalla.
- **Límites del escenario:** Una lista de 6 números que indican las 6 paredes de la caja que contiene el escenario. Si un personaje se sale de la caja en cualquier dirección (Incluso arriba) perderá.

4.3.2 El fondo

Se llama “fondo” a un cilindro vertical gigante que envuelve el escenario por completo. Mapeado sobre el cilindro hay una imagen. El cilindro es puramente gráfico y no afecta a la física del juego. No se espera que la cámara pueda ver por encima o por debajo de los límites del cilindro. El funcionamiento de la cámara se explica en el apartado **4.5**.

4.3.3 Plataformas

Cada plataforma es un prisma rectangular y debe estar obligatoriamente alineado con los ejes de coordenadas (X, Y, Z). Sobre el prisma está mapeada un mapa de bits, dividido en 12 partes de igual tamaño, como se muestra aquí.

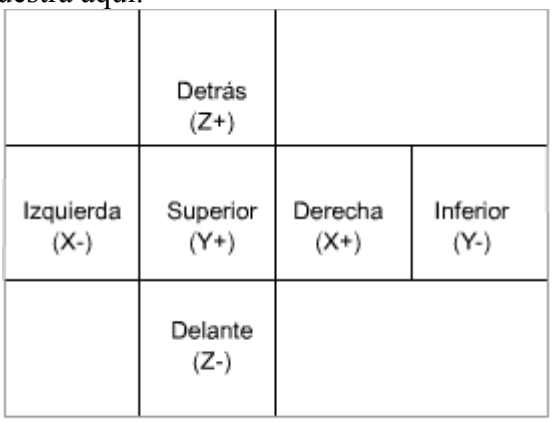


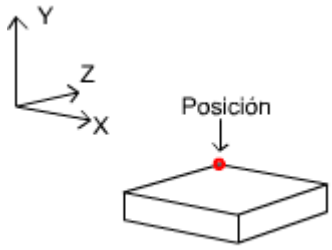
Imagen de la textura de una plataforma

Las plataformas tienen tres atributos principales:

- **Dimensiones:** Anchura (X), Altura (Y) y Longitud (Z), de la plataforma.
- **Textura:** El identificador de una imagen para la textura del prisma rectangular
- **Color:** Un valor entero de 32 bits en formato 0xAARRGGBB que representa el color de la plataforma si no tiene textura.

Las plataformas se dividen en tres tipos según su movimiento. Y cada tipo tiene sus propios atributos:

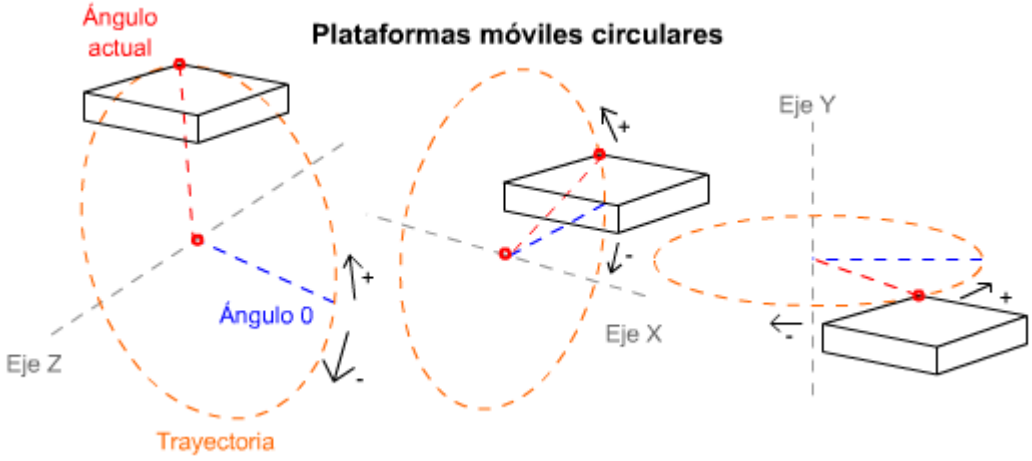
- **Tipo normal:** tiene un punto (X, Y, Z) indicando la posición de la plataforma. El punto marca la esquina superior (Y positivo), izquierda(X negativo) y trasera (Z positivo) de la plataforma.



Estructura de una plataforma

- **Tipo lineal:** Se desplaza un punto (X, Y, Z) de origen hasta otro punto destino y vuelve de nuevo al punto de origen. Tiene estos atributos:
  - **Punto de origen:** Un punto (X, Y, Z)
  - **Punto de destino:** Un punto (X, Y, Z)
  - **Periodo:** Un valor entero midiendo el tiempo, que tarda la plataforma en ir y volver al punto de origen.
  - **Fase:** Un número que representa un ángulo entre 0 y 2*PI indicando el desfase en el movimiento de la plataforma en comparación con una plataforma que comienza en el punto origen en el momento 0.
  - **Pausa:** Un valor entero indicando el número de segundos que debe esperar la plataforma al llegar al punto de destino antes de comenzar su vuelta al punto de inicio.

- **Tipo circular:** Esta plataforma gira alrededor de un punto en el espacio sobre una de tres ejes: X , Y o Z. La plataforma en sí no gira, sino que gira solamente su posición. Tiene estos atributos:
  - **Punto central:** Un punto X,Y,Z alrededor del cual gira la plataforma.
  - **Radio:** El tamaño del círculo de giro de la plataforma.
  - **Eje:** Un valor enumerado X, Y o Z indicando el eje sobre el que debe girar la plataforma.
  - **Periodo:** Un valor entero midiendo el tiempo, que tarda la plataforma en ir y volver al punto de origen.
  - **Fase:** Un número que representa un ángulo entre 0 y 2*PI indicando el desfase en el movimiento de la plataforma en comparación con una plataforma que comienza en el punto origen en el momento 0.
  - **Pausa:** Un valor entero indicando el número de segundos que debe esperar la plataforma a cada extremo del círculo.



A continuación se presentan una serie de datos sobre el movimiento periódico de las plataformas móviles, sean lineales o circulares:

- Una revolución tarda esta cantidad de tiempo:
  `revolución = 2*pausa+periodo.`
- La velocidad angular se calcula de este modo:

```
velocidad_angular = revolución/marcos_por_segundo
momento_actual = (TIEMPO_GLOBAL + fase/velocidad_angular)
                      MOD (2*pause+period)
SEGÚN (momento_actual):
   CASO: [0  ...  periodo/2] ◊
      ángulo = momento_actual*velocidad_angular
   CASO: [periodo/2  ...  period/2+pause] ◊
      ángulo = PI/2
   CASO: [periodo/2+pausa ...periodo+pausa] ◊
      angulo = (momento_actual-pausa)*velocidad angular
   OTRO CASO ◊
      ángulo = 0;
```

Finalmente, la posición de la plataforma se calcula utilizando el seno del **ángulo** obtenido a partir de este algoritmo para saber en qué parte del movimiento periódico se encuentra.

4.3.4 Elementos de Decorado (Sprites)

Un elemento de decorado es un objeto plano y rectangular con una textura animada, como los personajes. El funcionamiento de sus animaciones sigue las mismas reglas que las animaciones de los personajes. Los elementos no tienen ninguna influencia sobre el funcionamiento del juego, pero un escenario sin estos elementos quedará bastante vacío.

Tienen estos atributos:

- **Posición:** Un punto (X, Y, Z) indicando la posición de la zona inferior central del elemento de decorado.
- **Textura:** Un identificador de la imagen que se utiliza como textura.
- **Orientación:** Un valor entero entre 0 y 360 que indica, en grados, en qué dirección debería estar girado el elemento sobre el eje Y. El valor 0 hace que el elemento esté mirando hacia la dirección Z negativa. Cualquier valor negativo hará que elemento de decorado esté siempre mirando hacia la cámara. Habrá más detalles sobre orientación y texturas animadas en el apartado **4.7**.
- **Dimensiones:** Altura y anchura del proyectil. La anchura es igual para las dos coordenadas horizontales, X y Z.

4.4 Diseño de objetos proyectiles

Un objeto proyectil es un objeto creado por un personaje cuando utiliza algunos tipos de ataque especial. Vuela en una dirección y daña a otros personajes. Una vez haya dañado a alguien o haya pasada suficiente tiempo, el proyectil desaparece. Si dos proyectiles colisionan, se cancelan entre sí y desaparecen.

Gráficamente, se representa como un rectángulo con una textura animada que siempre está orientada hacia la cámara y sigue las mismas normas que los personajes y elementos de decorado.

Los proyectiles tienen estos atributos:

- **Daño:** La cantidad de daño que causa a un personaje cuando lo golpea. El daño se mide en porcentaje (%). Se explicará más sobre el daño en el apartado **4.5**.
- **Tipo de daño:** Es un tipo enumerado con los valores Eléctrico, Impacto y Ninguno. El proyectil de tipo eléctrico aturde al objetivo durante un corto tiempo después de golpearle. El proyectil de tipo impacto golpea al personaje fuertemente, haciendo que pueda salir volando si tienen suficiente daño. El tipo ninguno causa daño sin otros efectos secundarios.
- **Gravedad:** Un valor booleano que indica si este proyectil debe cae con gravedad o si se mueve en línea recta.
- **Anchura**
- **Altura**

4.5 Descripción de la física

Este apartado describe en detalle cómo funciona el sistema de combate del juego. Incluyendo las acciones de los jugadores y la interacción entre todos los objetos del escenario.

4.5.1 Acciones y física de los personajes

Los personajes se mueven en un espacio tridimensional. Si no están sobre alguna plataforma, caerán con la gravedad. Los personajes se mueven sobre el plano horizontal (X, Z) utilizando el stick analógico o los botones de dirección. Pueden moverse igualmente en el aire y en el suelo.

Para moverse verticalmente, los personajes pueden saltar pulsando el botón **Y** en el mando. Cada personaje puede saltar una vez desde el suelo y tiene la opción de saltar una vez más en el aire después de haber saltado la primera vez.

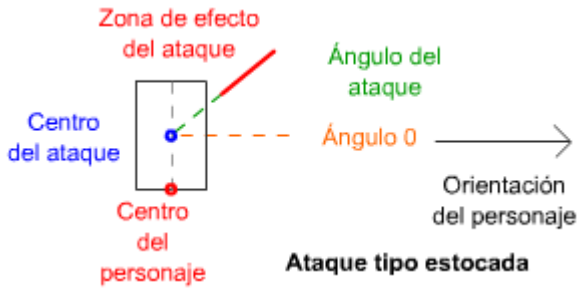
En este juego existe un mecanismo llamado “Apuntar” que fija a otro personaje como un objetivo. Cuando un personaje está apuntando a otro, su movimiento y dirección siguen siendo igual que antes, pero al realizar un ataque, se orienta siempre e inmediatamente hacia su objetivo. Si no se está apuntando hacia algún objetivo en concreto, se considera que el personaje siempre está mirando a su objetivo. Este mecanismo del juego tiene su mayor utilidad en la realización de ataques.

Hay dos tipos de ataques: Ataques normales y ataques especiales.

- Los ataques normales son ataques que causan daño y pueden hacer volar al enemigo. Se hacen pulsando el botón **A** durante el combate. Se caracterizan por realizar daño - a veces lanzando al oponente - sin causar efectos secundarios. La clasificación de ataques se explicará en el apartado **4.6**. Un ataque normal está formado por una lista de objetos "Golpe", que representan un solo movimiento. Durante el ataque, los golpes se harán de uno en uno consecutivamente. Cada golpe tiene estos atributos:
    - **Momento de inicio:** En qué marco de la animación de ataque comienza el ataque.
    - **Duración:** Número de marcos de la animación en el cual se está haciendo el ataque.
    - **Multiplicador de velocidad:** Indica la velocidad a la que iría un personaje golpeado si tuviera 100% de daño. No se aplica al ataque neutral o neutral aéreo.
    - **Velocidad mínima:** La velocidad mínima a la debe ir el personaje golpeado si tiene poco daño. No se aplica al ataque neutral o neutral aéreo.
    - **Umbral de porcentaje:** Solo para ataques en movimiento. Es el porcentaje mínimo que debe tener el personaje golpeado para salir volando.
    - **Porcentaje de daño:** La cantidad de daño que causa el ataque al enemigo.
    - **Tipo del ataque:** Si es de tipo estocada o rotación.

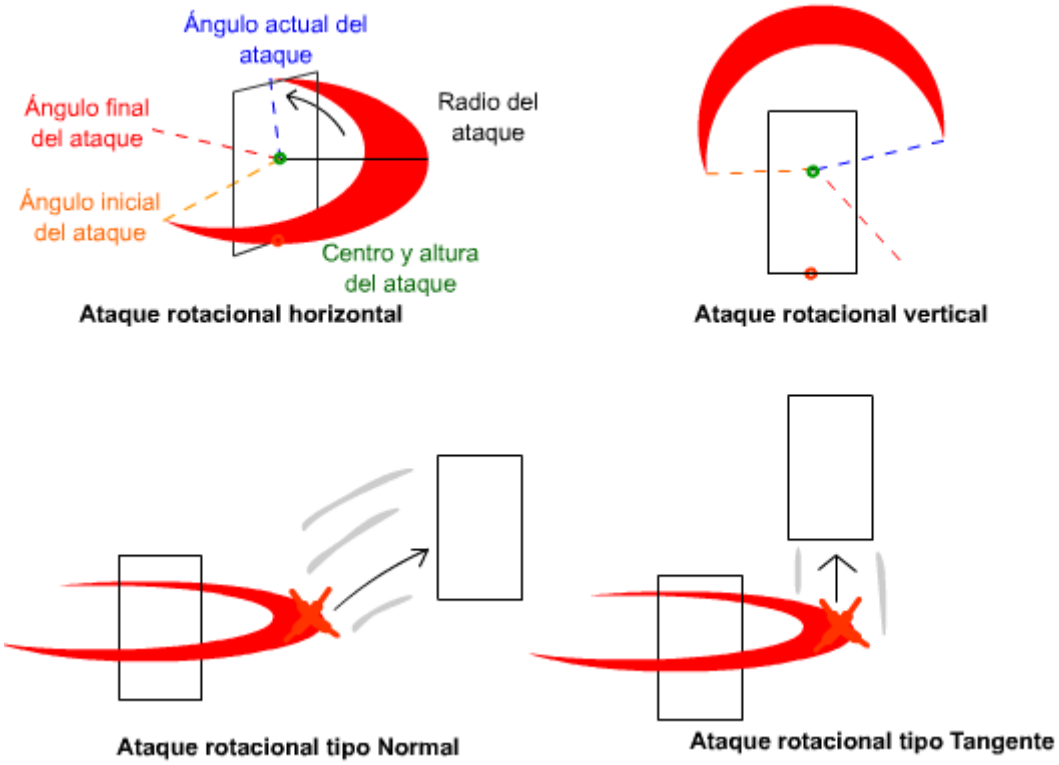
Existen dos tipos de golpes: Estocada y rotación.

- El ataque estocada se mueve desde el centro del personaje hasta su longitud máxima en línea recta. Tiene atributos propios:
    - **Altura de comienzo:** Altura en donde se comienza el ataque.
    - **Longitud de comienzo:** Longitud inicial del ataque
    - **Longitud final:** Longitud final del ataque
    - **Ángulo vertical:** La rotación del ataque alrededor del centro del personaje. El valor 0 es horizontal hacia delante.



Ataque tipo estocada

- El ataque rotación gira alrededor del personaje de manera horizontal o vertical. Puede subdividirse en dos tipos: Normal o tangente. El tipo normal hará que el personaje golpeado vuele en dirección de la normal del círculo. El tipo tangente hará que el personaje golpeado vuele en dirección de la tangente del círculo. Los ataques de tipo rotación tienen estos atributos:
  - **Altura del centro del círculo**
  - **Radio del círculo**
  - **Ángulo de comienzo del ataque.**
  - **Ángulo de final del ataque.**
  - **Eje:** horizontal o vertical.
  - **Tipo:** Normal o tangente.



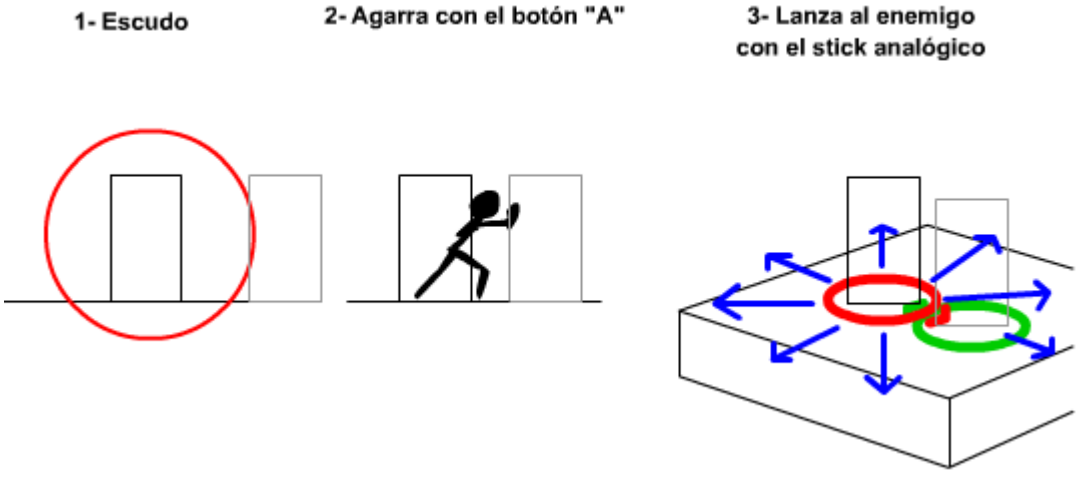
Nota: *La distancia y el sentido del giro dependen de la diferencia total entre el ángulo inicial y el ángulo final. Por lo tanto, el personaje puede dar dos vueltas enteras si los ángulos son [0º, 720º] y puede dar vueltas en sentido contrario si el ángulo final es menor que el ángulo inicial.*

- Los ataques especiales se hacen pulsando el botón “B” y pueden tener varios efectos. Se dividen en 6 tipos, cada uno con sus atributos:
    - **Tipo ataque normal:** Actúa como un ataque normal, pero puede añadirse el efecto de aturdir al enemigo durante un corto tiempo. Sus atributos son:
        - Una **lista de objetos golpe**, con los atributos descritos en el apartado de ataques normales.
        - **Carga:** Valor booleano que indica que la efectuación del ataque puede retrasarse manteniendo pulsado el botón **B**.
        - **Eléctrico:** Valor booleano que indica si el ataque puede aturdir al enemigo.
    - **Tipo explosión:** Crea una esfera creciente alrededor del personaje que daña a los personajes de alrededor en todas direcciones. Los personajes golpeados saldrán volando. Sus atributos son:
        - **Momento de inicio**
        - **Duración**
        - **Radio final de la explosión**
        - **Multiplicador de velocidad**
        - **Velocidad mínima**
        - **Color de la esfera**
        - **Altura del centro de la esfera**
    - **Proyectil:** El personaje lanza uno o varios proyectiles. Tiene estos atributos:
        - **Altura:** La altura donde comienzan los proyectiles.
        - **Identificador del** objeto **proyectil** a crear.
        - **Número de proyectiles**
        - Lista de **ángulos horizontales** de los proyectiles
        - Lista de **ángulos verticales** de los proyectiles.
        - **Velocidad inicial** de los proyectiles
        - **Momento de inicio:** Marco de la animación en el cual los proyectiles se lanzan.
    - **Movimiento:** El personaje se mueve en una dirección. Puede realizar un ataque mientras se está moviendo. Sus atributos son:
        - Todos los **atributos del ataque normal**.
        - **Velocidad:** La velocidad del personaje al comenzar el ataque.
        - **Ángulo:** Un ángulo en el plano vertical hacia donde se mueve el personaje.
        - **Momento de inicio:** El marco de animación en donde comienza a moverse el personaje.
        - **Duración:** La cantidad de marcos que dura el movimiento.
        - **Momento:** Valor booleano. Si es verdadero, el personaje sigue moviéndose después de terminar el ataque. Si es falso, el personaje se mueve a velocidad constante durante el ataque y deja de moverse al finalizarlo.
        - **Eléctrico:** Valor booleano indicando si personajes golpeados por el ataque son aturdidos.
    - **Escudo:** El personaje crea un escudo esférico a su alrededor que refleja proyectiles. El escudo comienza con tamaño y crece hasta su tamaño final. Sus atributos son:
        - **Color:** El color del escudo
        - **Radio:** El tamaño final del escudo
        - **Altura del centro** del escudo.
        - **Momento de inicio:** Marco de animación en el cual el escudo comienza a crecer.
        - **Momento final:** Marco de animación en el cual el escudo llega a su tamaño máximo.

        - **Duración:** Tiempo, en marcos, que tarda el escudo en desaparecer después del momento final. El valor -1 indica que el escudo termina cuando se deja de pulsar el botón "B".
        - **Absorber:** Un valor booleano. Si es cierto, repara daño cuando es golpeado por un proyectil. Si es falso, refleja el proyectil.
    - **Contraataque:** Un ataque que espera un corto tiempo y ataca si es golpeado. Además, durante ese tiempo no recibe daño. Sus atributos son:
        - Todos los **atributos del ataque normal**.
        - **Momento de inicio:** Marco de animación en el cual se prepara para recibir golpes.
        - **Momento final:** Marco de animación en el cual termina la animación de prepararse.
        - **Duración:** Tiempo que está el personaje preparado para recibir golpes.
        - **Multiplicador:** Se multiplica por el daño recibido y sirve para calcular el daño causado por el contrataque.
    - **Fuente de proyectiles:** El personaje se queda quieto y comienza a lanzar muchos proyectiles a intervalos regulares mientras esté pulsado el botón "B". Sus atributos son:
        - **ID del proyectil** a lanzar
        - **Marco** de animación en **donde** se **comienza a lanzar proyectiles**.
        - **Duración:** Duración de la animación que se repite constantemente
        - **Ángulo vertical:** La dirección en el cual se mueven los proyectiles.
        - **Velocidad:** La velocidad inicial del proyectil.
        - **Intervalo:** La cantidad de marcos que pasan entre lanzar un proyectil y otro.

Para defenderse de los golpes, los personajes tienen un escudo esférico. Pueden ponerse el escudo pulsando **R**. Mientras el escudo está puesto va perdiendo poder. Si recibe golpes, lo pierde más rápidamente. Cuando haya perdido todo su poder, se rompe, aturdiendo al personaje durante un tiempo. Cuando no está puesto el escudo, va recuperando su poder lentamente.

Los personajes pueden agarrar a otros personajes pulsando **A** mientras tienen puesto el escudo. El agarre es un tipo de ataque con características especiales. Si consigue golpear a un enemigo, lo mantendrá agarrado durante un corto tiempo. Durante ese tiempo, el personaje puede elegir lanzar al enemigo en cualquier dirección. El personaje lanzado volará en dirección del stick analógico con un ángulo de 45 grados hacia arriba. Si se pasa el tiempo sin lanzar al personaje, el personaje agarrado se suelta.

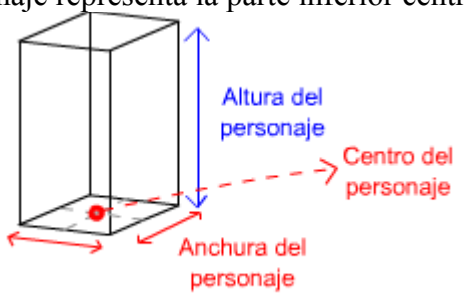


4.5.2 Interacciones

Formas de los objetos

A continuación se describe la forma geométrica que tienen todos los objetos con relación a la física del juego.

Personajes: Los personajes están formados por su posición, su anchura y su altura. Con estas propiedades, se forma una caja alrededor del personaje de base cuadrada y altura igual a la del personaje. La posición del personaje representa la parte inferior central de la caja.



Dimensiones de un personaje

Ataques: Los ataques son objetos ligados a los personajes. Si un personaje está atacando, el ataque en cualquier momento dado tiene la forma de una línea unida por dos puntos.

Plataformas: Las plataformas son cajas que están definidas por sus dimensiones y posición. La posición de la plataforma representa la posición de la esquina izquierda, trasera y superior de la caja.

Proyectiles: Los proyectiles se representan como los personajes: con su posición, anchura y altura. Pero la posición del proyectil representa el punto central, en todas las dimensiones, de la caja que lo rodea.

Interacción entre objetos

Dos objetos interactúan cuando la forma que lo representa intersecta con la forma que representa al otro objeto.

Interacción entre personajes

Hay tres tipos de interacción entre personajes:

- **Personaje a personaje:** Cuando dos personajes se tocan, se separan lentamente.
- **Personaje a ataque:** Cuando un personaje esté tocando un ataque, recibe daño. Puede salir volando o quedarse aturdido. El daño y la dirección de vuelo dependen de los atributos del ataque.
- **Ataque a ataque:** Si se conectan dos ataques, se cancela el más débil o ambos si son de igual potencia. La potencia del ataque se divide en tres tipos: Neutral, En movimiento y fuerte. Estas potencias corresponden a los tipos de ataque del apartado **4.6**: neutral, en movimiento y fuertes.

Interacción entre personajes y plataformas

Si un personaje está tocando una plataforma por encima, la plataforma actuará como un suelo. El personaje guarda la plataforma que está tocando, para poder actualizar su posición en plataformas móviles.

Si un personaje está tocando alguna de las paredes de una plataforma y alguna parte del personaje está por encima de la plataforma, el personaje se quedará colgando de la plataforma. Puede moverse hacia la plataforma para subirse, moverse en dirección contraria al borde para caerse y pulsar **Y** para saltar.

Interacción de proyectiles

Si un proyectil colisiona con un personaje, otro proyectil o una plataforma, desaparece. Si lo que está tocando es una persona, el personaje recibe el daño y los efectos secundarios dados por los atributos del proyectil.

4.6 Descripción de los controles.

El mando para este juego debería tener una forma como la mostrada en la siguiente imagen. Si el jugador está usando el teclado, se mapearán las teclas del teclado sobre los botones de este mando virtual:

A	Ataque normal. Agarrar.
	Activar un botón en el menú.
B	Ataque especial.
	Regresar a la pantalla de menú anterior
START	Pausar el juego.
	Activar un botón en el menú. Avanzar en el menú.
X	Cancelar objetivo.
Y	Saltar. Doble salto.
L	Fijar objetivo. (Target) Cambiar objetivo.
R	Activar escudo.

Direcciones

En este mando se pueden utilizar cualquiera de los sticks analógicos o el botón de direcciones para mover el cursor en el menú o los personajes en el escenario. Si se utiliza un stick analógico durante el juego, los personajes correrán más rápidamente cuanto más alejado está el stick del centro. El botón de direcciones siempre utiliza la velocidad máxima.

Botón L

En el juego, el botón **L** sirve para apuntar a un personaje. Cada vez que se pulsa el botón, se va rotando el personaje fijado. Si llega al final de la lista, se cancela el mecanismo "Apuntar"

Botón R

En el juego, pulsar **R** activa el escudo, y soltar **R** lo desactiva.

Botón A

El botón **A** sirve para activar elementos en el menú y para atacar durante el juego. Según el estado actual del juego, los ataques pueden variar. En la siguiente lista la palabra "Objetivo" indica otro personaje fijado con el mecanismo "apuntar". Hay 11 ataques en total:

- **Ataque neutral:** Cuando el personaje está parado en el suelo.
- **Ataques fuertes:** Cuando el personaje está en el suelo y moviéndose. Puede hacer volar al enemigo en está lo bastante dañado. Hay tres tipos:
  - **Hacia delante:** Cuando el personaje está acercándose a su objetivo.
  - **Hacia atrás:** Cuando el personaje está alejándose de su objetivo.
  - **Central:** Cuando el personaje está moviéndose lateralmente comparado con su objetivo.
- **Ataques fuertes:** Cuando el jugador inclina el stick analógico rápidamente justo en el momento de pulsar **A**. Si se mantiene pulsado el botón **"A"**, el golpe se retrasa un poco. El personaje tiene que estar en el suelo. Siempre hace volar al enemigo. Hay tres tipos:
  - **Hacia delante.**
  - **Hacia atrás.**
  - **Central.**
- **Ataques aéreos:** Cuando el personaje realiza un ataque en el aire. Hay cuatro tipos:
  - **Neutral:** Cuando el stick analógico está centrado.
  - **Hacia delante:** Cuando el personaje está acercándose a su objetivo.
  - **Hacia atrás:** Cuando el personaje está alejándose de su objetivo.
  - **Central:** Cuando el personaje está moviéndose lateralmente comparado con su objetivo.

Botón B

En el menú, el botón **B** sirve para regresar a la pantalla anterior. Dentro del juego, activa los ataques especiales. Hay 4 tipos de ataques según la inclinación del stick analógico y la orientación del personaje comparado con el objetivo:

- **Neutral:** Cuando el stick analógico está centrado.
- **Hacia delante:** Cuando el personaje está acercándose a su objetivo.
- **Hacia atrás:** Cuando el personaje está alejándose de su objetivo.
- **Central:** Cuando el personaje está moviéndose lateralmente comparado con su objetivo.

Botón X

El botón **X** cancela el mecanismo apuntar.

Botón Y

El botón **Y** sirve para saltar. Se puede saltar desde el suelo o mientras el personaje está colgándose del borde de una plataforma. También se puede saltar una vez en el aire después de haber saltado la primera vez.

Botón START

En algunas pantallas del menú sirve para avanzar a otra pantalla. En el juego sirve para pausar el juego. Si un jugador ha pausado el juego, solo ese jugador podrá reanudarlo, pulsando de nuevo el botón **START**.

4.7 Diseño de los gráficos

4.7.1 Iluminación

La iluminación del juego consistirá en una luz blanca direccional. Esta luz es fija y no depende del escenario. Solo las plataformas se verán afectados por la iluminación.

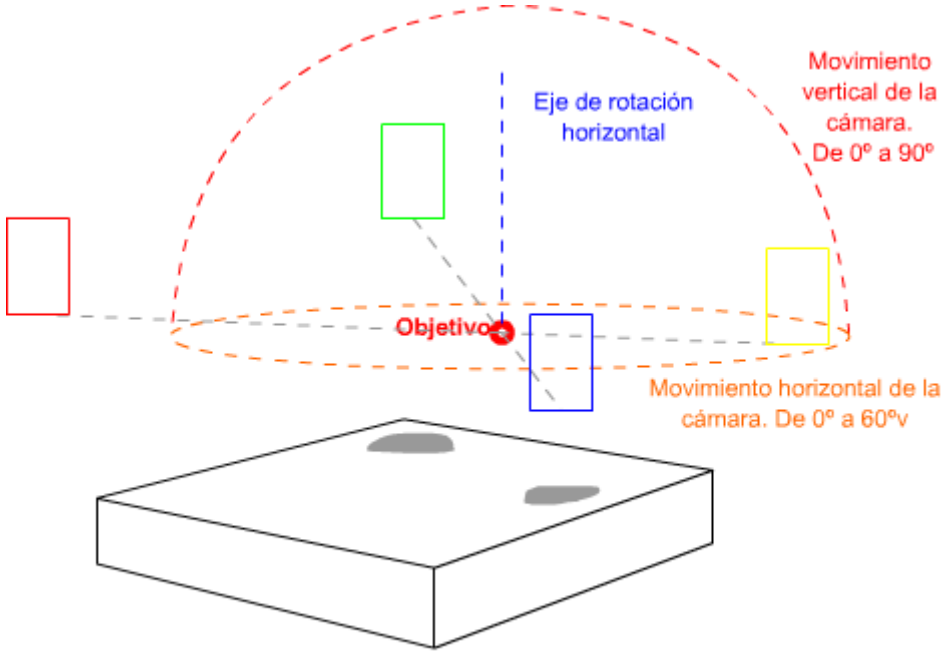
4.7.2 La cámara

La cámara es un objeto que sirve para visualizar el juego. Tiene dos componentes importantes: un punto (X, Y, Z) de posición y un punto (X, Y, Z) indicando el objetivo de la cámara. La cámara siempre se fijará en su objetivo.

La posición del objetivo de la cámara será la media aritmética de la posición de todos los personajes. La posición de la cámara se calculará en función de su objetivo. Tendrá un ángulo vertical, un ángulo horizontal y un valor indicando la distancia al objetivo. Hay algunas normas que debe seguir la cámara:

- La distancia de la cámara a su objetivo tiene que garantizar que todos los personajes estén dentro del ángulo de visión.
- La rotación horizontal de la cámara tiene que intentar ser perpendicular al ángulo de la línea entre los dos personajes más alejados.
- La rotación vertical debe estar a la altura o por encima del objetivo.

En el futuro se intentará implementar una cámara que tenga en cuenta la posición de las plataformas, para que ningún personaje quede fuera de vista.



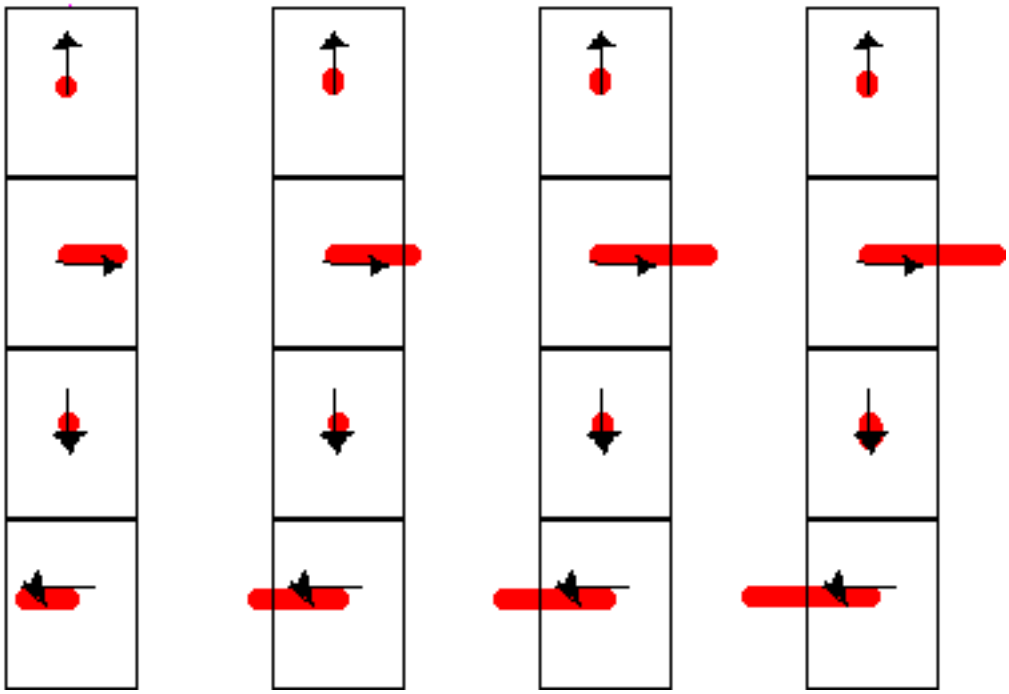
Funcionamiento simple de la cámara

4.7.3 Objetos animados

Hay tres objetos animados: Personajes, proyectiles y elementos de decorado. Todos se representan como un rectángulo con una textura animada. En el caso de los personajes y proyectiles, el rectángulo siempre tiene que tener su cara mirando a la cámara. En el caso de los elementos del decorado, la orientación depende de su atributo.

Las texturas animadas son imágenes en formato PNG. Cada imagen representa una animación y está dividida en una matriz de rectángulos del mismo tamaño. Hay 5 filas y un número no predefinido de columnas. La primera fila está reservada para datos y las otras cuatro representan una de las cuatro vistas de la cámara: Delante, detrás, izquierda o derecha. Cada columna representa un marco de la animación del objeto. La sección de la imagen utilizada en cada momento se elige basada en el ángulo de la cámara, la orientación del objeto y su marco actual.

La imagen en la primera columna y fila representan el punto cero del objeto. Hay un píxel "mágico" que indica en qué parte de la imagen debe ir el punto "Posición" del objeto. A continuación se muestra una imagen de una animación de 4 marcos para un personaje. Se puede observar que la primera fila está vacía y tienen un píxel de color magenta (0xFF00FF en hexadecimal de 24 bits).



4.7.4 Objetos adicionales

Existen algunos objetos gráficos más que aparecerán en el juego.

Gráficos tridimensionales

Sombras: Una sombra es un plano octagonal con transparencia media que siempre está en la plataforma más alta por debajo de un personaje, o en el límite inferior del escenario si no hay plataformas por debajo.

Marcadores de jugador: Debajo de cada jugador aparecerá un aro de su color junto con una flecha indicando la dirección a la que está apuntando (Útil para saber si está utilizando el mecanismo "Apuntar").

Formas de ataque: Los ataques de tipo rotación generarán un disco del color del jugador que sigue la trayectoria del ataque. Los ataques de tipo estocada generarán un objeto en forma de flecha. Que se posicionará en la dirección y distancia del ataque.

Esferas: Son siempre semitransparentes y se utilizan para representar escudos y explosiones. Los escudos de los jugadores tienen el color del jugador. Las explosiones y los escudos de ataques especiales tienen un color asignado por el atributo del ataque.

Gráficos bidimensionales

Información de personajes: Incluye una imagen del personaje, el porcentaje de daño que lleva y un contador del número de vidas que le quedan.

Mensaje de victoria/derrota: Cuando un personaje ha salido de los límites del escenario, aparecen dos imágenes con texto. En la esquina superior, el personaje que ha perdido y en la esquina inferior el último personaje que le ha golpeado. Cada imagen viene junto con un texto representando el comentario del personaje vencedor o perdedor.

5. Implementación del primer prototipo

El primer prototipo se implementó antes del comienzo formal de proyecto, no se utilizó una metodología formal. Por lo tanto, el programa no está muy bien organizada y la clases no encapsulan muy bien su funcionamiento. A continuación, se muestra una lista de las clases y estructuras principales ordenadas según su función.

5.1 Ficheros de datos

Antes de comenzar con las listas de estructuras, es importante explicar de qué manera se almacenaron los ficheros de imágenes y datos para el programa. No se utilizaron ficheros normales, sino que se encapsularon los datos en ficheros DLL. Las imágenes en *samoyede.dll*, los datos en *basset.dll* y el texto en *spinon.dll*. Cada fichero se generó con un proyecto distinto: *Battle3DImages*, *Battle3DEnglish* y *Battle3DData*.

Los motivos de esta implementación fueron proteger los datos almacenados de y reducir el número de ficheros que deben acompañar el programa para que funcione.

Para hacer uso de un fichero DLL, se debe crear un manejador de módulos, de tipo *HMODULE*, llamando a la función ***LoadLibraryEx()***. Cada fichero se almacenó en el módulo como un recurso con un número de identidad. Para extraer ficheros del módulo, se llama a la función ***FindResource(),*** que devuelve un identificador del recurso, de tipo *HRSRC*. Con este valor, se llama a la función ***LoadResource()***, que devuelve un manejador de recurso de tipo *HGLOBAL*. Pasando este valor a la función ***LockResource()***, se devuelve un puntero a la cadena de bytes que contiene los datos.

Se ha concluido que existen maneras mucho mejores de conseguir los objetivos que se intentaron alcanzar con los ficheros DLL. La solución final se explica en la descripción de la clase *DataBase* del prototipo basado en OpenGL.

5.2 Programa principal

El programa principal se ejecuta en la clase *Game*. Esta clase contiene todos los objetos relacionados con los gráficos, el menú y el juego.

Contiene información de los jugadores en un vector *players* de tipo *Player*. El objeto *abstract3D*, de tipo *Abstract3D* maneja los gráficos. Se almacenan las pantallas del menú en el vector *menus* de tipo *MenuScreen*. El valor de *currentMenu* indica qué pantalla se está mostrando actualmente. La propiedad *stage* es un puntero a un objeto de tipo *Stage*, que representa un escenario del juego.

Las funciones principales son:

- ***initialize(HWND hwnd):*** Inicializa los jugadores, el menú y los gráficos.
- ***setupMenus():*** Crea cada uno de las pantallas del menú con sus botones
- ***switchToMenu(int menu, int advance=0):*** Cambia de la pantalla de menú actual a una pantalla nueva.
- ***run():*** Se ejecuta repetidamente durante toda la ejecución del programa. Actualiza las entradas de datos, lee las entradas de datos, ejecuta la función ***frame()*** del objeto *stage* si existe y ejecuta la función ***run()*** del objeto *MenuScreen* activo, si existe.
- ***close():*** Elimina todos los datos de la memoria.
- ***handleMenu(EventObject *object):*** Maneja eventos relacionadas con el menú.
- ***handlePress(EventObject *object):*** Cuando un jugador pulsa un botón, realiza una acción según la pantalla de menú y el botón que está activo.

5.3 Gráficos

Los gráficos se implementan principalmente en el fichero Abstract3D.h.

5.3.1 Vertex3D

El struct *Vertex3D* se utiliza para almacenar vértices de modelos en Direct3D. En las propiedades *x*, *y*, *z* se indica la posición del vértice, en la propiedad *normal*, de tipo D3DVECTOR, se almacena la normal del vértice. En la propiedad *color* se almacena el color del vértice en formato hexadecimal y las propiedades *tu* y *tv* representan el mapeado UV del vértice sobre la textura.

5.3.2 Texture

El struct *Texture* representa una textura de un modelo. La propiedad *image* contiene datos de la *imagen* que representa la textura. Es de tipo *ImageData*, que se explicará más adelante. La propiedad *texture* representa la textura en memoria gráfica. Es del tipo *IDirect3DTexture9*. Las propiedades *x*, *y*, *width* y *height* representan un rectángulo que define la porción de la imagen a utilizar como textura. El valor de *changed* indica si la textura ha cambiado desde la última vez que se almacenó en la memoria gráfica.

5.3.3 Model

El struct Model representa in modelo tridimensional. Los vectores *vertices* e *indices*, junto con sus respectivas longitudes *vertLen* e *indLen*, representan los vértices indizados del modelo en Direct3D. Las propiedades *pos*, *rotation*, *angle* y *scale* representan transformaciónes del modelo en el espacio tridimensional: Posición, rotación horizontal, rotación vertical y escalado. La propiedad *visible* indica si el modelo se renderizará o no. El valor de *type* indica el tipo de modelo, que sirve para decidir el orden de representación. Finalmente, la propiedad *texture* representa la textura del modelo.

5.3.4 Sprite

El struct Sprite representa un modelo bidimensional en la pantalla. Sirve para mostrar objetos del menú. Su forma se representa con la propiedad *texture*, que contiene la textura del modelo y, consecuentemente, sus dimensiones. Los valores de *x* e *y* determinan su posición en la pantalla y *visible* determina si se renderiza el modelo. La propiedad *text* representa un texto que puede contener el modelo y *letters* representa los modelos de cada carácter del text. No se implementan en este prototipo.

5.3.5 Abstract3D

La clase Abstract3D realiza la función de almacenar datos sobre el escenario 3D en donde se renderizan los modelos. Contiene cuatro listas enlazadas de objetos relacionados con el escenario: *models* es una lista de modelos 3D, *sprites* una lista de modelos 2D, *textures* una lista de texturas e images una lista de imágenes de tipo *ImageData*.

Se utilizan también algunas variables de tipos pertenecientes a Direct3D para la renderización del escenario. *Direct3DInterface* es un valor de tipo *LPDIRECT3D9*, que representa la interfaz de Direct3D. *Direct3DDevice*, de tipo *LPDIRECT3DDEVICE* es una abstracción de la pantalla a través de Direct3D. VertexBuffer e IndexBuffer son objetos de tipo IDirect3DVertexBuffer9 e IDirect3DIndexBuffer9 que representan los vértices de modelos en la memoria gráfica.

Como se puede observar, esto resulta en un solo buffer de vértices para todos los modelos del escenario. Es decir, que para la renderización de cada modelo, se cargaron los vértices en el buffer, se renderizaron y seguidamente se eliminaron de memoria. Esta implementación ineficaz fue debido al escaso conocimiento del proceso gráfico cuando se creó el programa.

Las propiedades *camera* y *objective* representan la posición de la cámara y la posición del punto hacia donde mira. Con esta información se puede generar la matriz de vista de la cámara.

Finalmente, la propiedad *dllModule* es el manejador de módulos DLL responsable de cargar imágenes en las texturas.

Las funciones principales son:

- ***createScreen(HWND window, int fullscreen=false):*** Esta función inicializa los objetos de Direct3D para mostrar datos en la pantalla.
- ***addModel(Model &model):*** Añade un modelo 3D a la lista de modelos.
- ***removeModel(Model *model):*** Elimina un modelo 3D de la lista de modelos.
- ***addSprite(Sprite &sprite):*** Añade un modelo 2D a la lista de modelos.
- ***removeSprite(Sprite *sprite):*** Elimina un modelo 2D de la lista de modelos.
- ***createImage(int imageID):*** Crea un objeto de tipo *ImageData* basado en la imagen de identidad *imageID* en el fichero DLL de imágenes. Este objeto representa una imagen.
- ***createTexture(int textureID, bool unique=true):*** Crea una texture a partir de la ID de una imagen. La imagen se genera pasando el valor textureID a la función createImage().
- ***loadTexture(Texture *texture):*** Si la textura está marcada como cambiada, carga la textura en memoria gráfica.
- ***updateTexture(Texture *texture, int x, int y, int width, int height):*** Cambia las dimensiones del rectángulo de la textura y la marca como cambiada.
- ***render():*** Borra la pantalla. Después renderiza todos los modelos 3D. Luego renderiza todos los modelos 2D. Finalmente muestra el resultado en la pantalla.
- ***drawModel(Mode &model):*** dibuja un modelo 3D en la pantalla.
- ***drawSprite(Sprite &sprite, ID3DXSprite *directSprite):*** dibuja un modelo 2D en la pantalla. Esta función hace uso de un objeto *ID3DXSprite* para simplificar el proceso de renderización.
- ***make*():*** Las funciones ***make*()*** generan una serie de objetos de tipo *Model* o *Sprite* de diferentes tipos. Por ejemplo: cajas, cilindros, esferas y objeto rectangulares 2D.

5.3.6 ImageData

Esta clase se encarga de cargar una imagen de un fichero DLL y almacenarlo en memoria principal o memoria gráfica. La imagen cargada se almacena en *image*, un objeto de tipo *Bitmap* de GDIplus. Almacena en *textureID* el número de identificación del recurso en el fichero DLL.

Sus funciones principales son:

- ***loadResource(LPCSTR pName, HMODULE hInst):*** Abre el fichero DLL dado por *hInst*, crea un objeto de tipo *Bitmap* y carga en ese objeto la imagen contenida en el recurso dado por el número *pName*.
- ***img2tex(IDirect3DDevice9 *device, Bitmap *src, Rect &sroi):*** Devuelve una textura, de tipo IDirect3DTexture9, creadas a partir de la imagen *src*. Se copian solamente lo píxeles contenidos en el rectángulo dado por *sroi*.

5.4 Eventos

El módulo de eventos se implementa en el fichero *EventDispatcher.h*. Implementa un sistema donde un objeto puede difundir un mensaje. Los objetos que estén escuchando al difusor podrán recibir el mensaje emitido. La utilidad de un módulo de eventos era muy limitada. En el prototipo de OpenGL se omitió completamente.

5.4.1 EventObject

El mensaje enviado por el difusor tiene la forma de un struct, llamado *EventObject*. Este struct tiene un valor type, de tipo texto, que indica el nombre del mensaje enviado. También tiene vectores de propiedades genéricas de tipo texto, entero y de coma flotante para pasar datos en el mensaje.

5.4.2 EventHandler

La clase *EventDispatcher* utiliza este struct para almacenar una lista de escuchadores para un evento en concreto. El struct *EventHander* contiene el tipo del mensaje en la propiedad *type* y un puntero al objeto receptor en la propiedad *handler*. La propiedad *next* sirve para implementar la lista enlazada.

5.4.2 Event

El struct *Event* sirve para enviar un evento a un escuchador en concreto. El mensaje se almacena en la propiedad *object* y el escuchador en la propiedad *handler*. La propiedad *next* se utiliza para implementar una lista enlazada.

5.4.4 EventDispatcher

La clase *EventDispatcher* implementa tanto el emisor de mensajes como el receptor. Para recibir mensajes de evento, un objeto debe derivar de esta clase.

Cada objeto de la clase tiene una lista de escuchadores en la propiedad *handlers*, que almacena una lista de objetos de tipo *EventHandler*. Cada objeto representa un escuchador para un mensaje. La clase tiene también dos propiedades globales *firstEvent* y *lastEvent* para almacenar la lista de eventos para enviar a todos los objetos del escenario.

Las funciones principales son:

- ***addEventListener(const char *type, EventDispatcher *handler):*** Añade un escuchador para el evento *type*.
- ***removeEventListener(const char *type, EventDispatcher *handler):*** Elimina un escuchador para el evento *type*.
- ***dispatchEvent(EventObject *object):*** Difunde el mensaje dado en el objeto *object*.
- ***handleEvent(EventObject *object):*** Una función que se llama cuando un escuchador recibe un evento. La clase del escuchador debe sobrecargar esta función.

La clase también tiene dos funciones estáticas para el manejo global de eventos:

- ***pushEvent(EventDispatcher *handler, EventObject *object):*** Añade un mensaje a la lista de mensajes a difundir.
- ***globalDispatch():*** Difunde todos los mensajes. Es necesario llamar a esta función para que la difusión funcione.

5.5 Entradas de datos

Las entradas de datos se implementan en la clase Player. Esta clase representa un jugador con un mando. El mando puede ser un mando de juego real o un teclado.

La clase almacena en *inputDevice* la interfaz de DirectInput, de tipo *IDirectInputDevice8* y en *controller* la interfaz del mando, de tipo *IDirectInput8*. El valor *controllerID* indica el número del dispositivo de entrada asignado el jugador. El valor *id* es el número de jugador.

El dispositivo de entrada se representa como un mando de 11 botones. El vector *buttonStates* almacena el estado de los botones en el momento actual. Los botones digitales pueden tener el valor 0 o 1, mientras que los botones direccionales pueden tener valores que varían entre 0 y 1. La propiedad *prevStates* almacena el estado de los botones en el momento previo. El vector *buttonBuffer* mantiene el estado del mando entre un momento y el siguiente. Los vectores *keys* y *buttons* guardan el mapeado de las teclas del teclado o los botones del mando real sobre los botones del mando ficticio.

Las funciones principales son:

- ***setController(int num=-1):*** Asigna el dispositivo de entrada número *num* al jugador. Si *num* es -1, se asigna el controlador con el mismo número que el *id* del jugador.
- ***setKey(int keyCode, ButtonID button):*** Mapea la tecla dada por *keyCode* al botón dado por *button*.
- ***down(ButtonID button):*** Devuelve verdadero si el estado actual del botón es mayor o igual a 0.5.
- ***tapped(ButtonID button):*** Devuelve verdadero si el estado actual del botón es mayor o igual a 0.5 y su estado anterior era menor que 0.5.
- ***state(ButtonID button):*** Devuelve el estado del botón button en el momento actual
- ***lastState(ButtonID button):*** Devuelve el estado del botón button en el momento previo.
- ***updateController():*** Actualiza el estado del mando almacenado en *buttonBuffer*.
- ***readController():*** Almacena *buttonStates* en *prevStates* y escribe *buttonBuffer* en *buttonStates*.

5.6 Juego

El juego se implementa como un escenario compuesto por objetos. El escenario se implementa principalmente en la clase *Stage*. Los objetos que componen el escenario implementan una interfaz básica, *Animated*. Los objetos puede ser plataformas, proyectiles, objetos decorativos o personajes.

5.6.1 GameData

La clase *GameData* lee ficheros de texto plano contenidos en un fichero DLL. Además, permite leer datos de texto almacenados como recursos en un fichero DLL. No se utiliza texto en este protitipo. Se ignorarán las propiedades y funciones relacionadas con el lenguaje.

El propósito principal de esta clase es leer ficheros conteniendo los datos de escenarios y personajes. Cuando se diseñó el programa, estos datos se guardaron en formato XML, pero los datos contenidos son prácticamente iguales a los datos contenidos en los ficheros anexos *formato_personaje.txt* y *formato_escenario.txt*. Los datos se encuentran en un DLL nombrado *"basset.dll"*.

La propiedad *hModule* es un manejador para el fichero DLL. La propiedad *pStream* sirve para copiar los datos en memoria principal y *xmlDoc* es una variable de tipo *IXmlReader* que sirve para leer datos en formato XML.

Las funciones principales son:

- ***loadCharacter(int charID, Character &character):*** Carga el fichero XML asociado a un personaje del fichero DLL y rellena el objeto *character* con los datos del personaje. El fichero XML es el recurso identificado por el valor de *charID*.
- ***loadStage(int stageID, Stage &stage):*** Carga el fichero XML asociado a un escenario del fichero DLL y rellena el objeto *stage* con los datos del escenario. Estos datos incluyen plataformas, objetos de decorado, la imagen de fondo y las posiciones iniciales de cada personaje. El fichero XML es el recurso identificado por el valor de *stageID*.

5.6.2 Animated

La interfaz *Animated* es una clase abstracta pura, implementada polimórficamente por otras clases. Representa un objeto animado en el escenario. El objeto puede ser un personaje, un proyectil, una plataforma o un objeto de decoración.

Las funciones principales son

- ***frame(int frameNum, Point3D cameraPos, float cameraRotation):*** Esta función se ejecuta repetidamente para actualizar el estado del objeto en cada momento. El parámetro *frameNum* es el valor del momento actual. Los parámetros *cameraPos* y *cameraRotation* son la posición y la rotación de la cámara. Se utilizan para calcular qué textura mostrar.
- ***getType():*** Devuelve el tipo del objeto.
- ***addtoScreen(Abstract3D *screen):*** Algunos objetos pueden tener varios elementos gráficos. Esta función permite a cada objeto añadir sus elementos gráficos a la pantalla, dada en el parámetro *screen*.

La clase contiene, además, algunas constantes globales, como el número de ángulos de la cámara y el número de marcos por segundo de las animaciones.

5.6.3 GameSprite

Esta clase implementa la interfaz *Animated*. representa un objeto rectangular estático con una textura animada. La textura se divide en una matriz de rectángulos, horizontalmente en marcos y verticalmente en ángulos de la cámara. La primera fila está vacía exceptuando en el primer marco, donde existe un solo píxel de color hexadecimal 0xFFFF00FF. Este píxel indica dónde se encuentra el centro del personaje para esta animación.

El valor *frames* indica el número de marcos que tiene la animación del modelo y *currentFrame* indica el marco actual. La propiedad *position* indica la posición del objeto en el escenario y *orientation* indica su rotación horizontal.

La propiedad *textureID* almacena el identificado del recurso de la textura en el fichero DLL. Se carga la textura en la propiedad *texture*, de tipo *Texture*. La propiedad *model*, de tipo *Model*, contiene el modelo 3D que representa el objeto de manera gráfica. La propiedad *size* indica las dimensiones del rectángulo que representa el modelo. Los valores se calculan según el tamaño de la textura y el número de marcos que tiene.

Las funciones principales son:

- ***frame(int frameNum, Point3D cameraPos, float cameraRotation):*** En cada momento, se cambia el marco de animación aumentando en 1 el valor de *currentFrame*. Si *currentFrame* llega al valor de frames, vuelve al marco 0. Si el valor de *orientation* es negativo, el modelo 3D se gira para mirar a la cámara y se actualiza la textura según el ángulo de la cámara y el valor de la orientación.
- ***addToScreen(Abstract3D *screen):*** Se utiliza un objeto de tipo ImageData para cargar la textura del objeto. Después se calcula las dimensiones del modelo y se calcula el centro del modelo buscando el píxel de color 0xFFFF00FF en la textura. Finalmente, crea el modelo 3D con la función ***Abstract3D::makeAnimated()*** y añade el modelo al escenario.

5.6.4 Movable

La clase *Movable* implementa la interfaz *Animated3D* y deriva de la clase *EventDispatcher*. Representa un rectángulo vertical com una textura animada, una posición y una velocidad. Se utiliza como la clase base para las clases *Platform*, *Projectile* y *Character*.

Contiene dos propiedades, *position* y *velocity*, de tipo Point3D, que representan la posición y la velocidad del objeto. El valor *currentFrame* indica el marco actual de animación.

5.6.5 Platform

La clase Platform deriva de la clase Movable. Representa una plataforma rectangular de tres dimensiones que puede moverse de varias maneras. Puede moverse linealmente entre dos puntos, de modo circular alrededor de un eje o puede simplemente quedarse en un sitio. Su comportamiento se define en la propiedad *platformType*.

Las dimensiones del rectángulo se dan en los valores *x*, *y* y *z* de la propiedad *size*. Si la plataforma se mueve linealmente, la propiedad *start* indica la posición de comienzo y *end* indica la posición final. Si se mueve de forma lineal, el centro de rotación se almacena en la propiedad *center*. El radio y el eje de rotación se dan en las propiedades *radius* y *axis*.

El tiempo que tarda en completar un ciclo de movimiento se da en la propiedad *period*. El desfase de este movimiento cíclico se da en la propiedad *phase*. El valor de pause determina cuánto tiempo se queda parada la plataforma en cada extremo de su movimiento lineal o en extremos opuestos de su movimiento circular.

La textura de la plataforma se extrae de un fichero DLL. El valor de *textureID* indica el número del recurso en donde se encuentra esta textura. El modelo gráfico se almacena en la propiedad *model*.

Sus funciones principales son:

- ***nextPosition(int frameNum):*** Esta función devuelve la posición en donde se encontrará la plataforma en el siguiente momento. El valor de *frameNum* indica el momento de tiempo global del escenario. Sirve para coordinar todas las plataformas.
- ***frame(int frameNum, Point3D cameraPos, float cameraRotation):*** Se actualiza la posición de la plataforma y de su modelo 3D.
- ***addToScreen(Absract3D *screen):*** Crea un objeto de tipo *Model*, con la función ***Abstract3D::makeBox()***, para representar la plataforma y los añade a la pantalla, dada en el parámetro *screen*.

5.6.6 Projectile

La clase *Projectile* deriva de la clase *Movable*. Representa un proyectil lanzado por un personaje. El proyectil se representa como un modelo rectangular con una textura que cambia según el ángulo de la cámara. Un proyectil comienza con una velocidad inicial y un tiempo de vida. El proyectil se mueve a velocidad constante. Cuando se agota el tiempo de vida, el proyectil desaparece.

Las funciones principales son:

- ***frame(int frameNum, Point3D cameraPos, float cameraRotation):*** Actualiza la posición del proyectil y reduce en 1 su tiempo de vida. Se actualiza la posición y rotación del modelo, y se actualiza la textura según el ángulo de la cámara.
- ***addToScreen(Abstract3D *screen):*** Crea un objeto de tipo *Model* con la función ***Abstract3D::makeAnimated()***. Carga la textura asociada al proyectil desde el fichero DLL. Finalmente añade el modelo a la pantalla. Se guarda una referencia a la pantalla para poder borrar el modelo cuando se elimine el objeto.
- ***getTimer():*** Devuelve el tiempo restante de la vida del proyectil.

5.6.7 Ataques

En el fichero Character.h se definen tres structs relacionados con los ataques del personaje.

Impact

El struct *Impact* guarda información de un un impacto de un ataque. Un ataque se puede componer de varios impactos seguidos. El ataque está ligado a la animación del personaje que está realizando el ataque. Por lo tanto, se utilizan los marcos de la animación para definir el estado del impacto.

La propiedad *startFrame* indica el marco donde comienza el impacto. *duration* indica el número de marcos que dura el impacto. El valor de *damage* indica el daño que hace a un oponente si lo toca. *height* indica la altura donde comienza el ataque. La propiedad *next* es un puntero al struct que define el siguiente impacto. Si el ataque no tiene otro impacto, el valor es NULL.

Si el ataque es de tipo fuerte o "Smash", el valor velocityMultiplier se multiplica con el porcentaje del personaje gopeado para calcular si velocidad de lanzamiento. Si el ataque es de tipo "Smash", el valor *minVelocity* indica la velocidad mínima a la que se lanza el oponente. Si es de tipo fuerte, el valor de *percentageThreshold* indica el daño mínimo que debe tener el enemigo para que pueda ser lanzado.

La propiedad *type* indica si el impacto es de tipo rotación o estocada. El impacto de tipo rotación utiliza la propiedad *radius* para indicar el radio de efecto del ataque. El valor de *axis* indica si la rotación es horizontal o vertical. *start* indica el ángulo de comienzo del ataque y *end* indica el ángulo final del ataque. El valor de *rotationType* indica si el el personaje se lanza en dirección de la normal del círculo o en direción de su tangente.

Si el impacto es de tipo estocada, la propiedad angle indica el ángulo vertical de la estocada. Las propiedades *start* y *end* indican las distancias inicial y final de la estocada.

Attack

El struct Attack contiene una lista enlazada de objetos *Impact*. Define un ataque completo del personaje.

SpecialAttack

Existen siete ataques especiales en total: normal, explosión, proyectil, movimiento, escudo, cotraataque y fuente de proyectiles. Este struct contiene todas las propiedades necesarias para ejecutar cada una de ellas. La propiedad *type* indica cuál de los seis ataques se está realizando.

El ataque normal utiliza la propiedad *impact* para definir la lista de impactos del ataque. Utiliza también los valores booleanos *charge* y *electric*, para definir si el ataque se puede retardar como los ataques "Smash" y para definir si el ataque paraliza al oponente.

El ataque explosión usa los valores *startFrame* y *duration* para describir el marco inicial de la explosión y el número de marcos que dura. El valor *damage* indica el daño que realiza la explosión y *radius* indica el radio de la explosión en el momento final. *velocityMultiplier* es un multiplicador que se usa para calcular la velocidad del oponente tras ser golpeado por la explosión.

El ataque proyectil puede lanzar varios proyectiles. Se usa el valor *height* para indicar la altura de comienzo del ataque. El valor de *projectileID* es el identificador del proyectil a lanzar. numProjectiles indica el número de proyectiles a lanzar. *hAngles* y *vAngles* son vectores del ángulo horizontal y vertical de cada proyectil que se lanza. El valor de *speed* indica la velocidad inicial de los proyectiles lanzados.

El ataque de movimiento desplaza al personaje en una dirección mientras realiza un ataque normal. Utiliza los valores *impact* y *electric* igual que el ataque normal. Además, usa la propiedad duration para indicar cuánto tiempo se queda moviendo el personaje. Las propiedades *hAngles*, *vAngles* y *speed* se utilizan de manera similar al ataque proyectil para definir la velocidad del movimiento. La propiedad *momentum* determina si el personaje mantendrá su inercia después de finalizar el ataque.

El ataque escudo genera un escudo que protege de ataques proyectiles y crece hasta llegar a su tamaño máximo. La propiedad *startFrame* indica el marco en donde comienza el escudo y *endFrame* indica el marco donde el escudo llega a su tamaño final, definido por el valor de *radius*. Además, la animación del personaje se repite entre estos dos valores hasta que se termina el ataque. La propiedad *duration* indica la duración del escudo. Un valor de -1 significa que no termina hasta que se haya soltado el botón **B**. Si *absorb* es verdadero, el ataque recuperará daño cuando es golpeado por un proyectil. Si es falso, reflejará el proyectil. La propiedad *color* define el color del escudo.

El contraataque comienza en el marco *startFrame* y espera un tiempo definido por la propiedad *duration*. Mientras está esperando, la animación se repite en el intervalo dado por *startFrame* y *endFrame*. Si es golpeado por un oponente, lanza un ataque dado por la propiedad *impact*. El daño del ataque será el daño recibido multiplicado por el valor de *multiplier*.

El ataque de tipo fuente de proyectiles mantiene al personaje en un sitio, lanzando proyectiles de una vida muy corta en una dirección. Comienza a lanzar proyectiles en el marco dado por *startFrame* y termina después de pasar el tiempo dado por *duration*. Si *duration* tiene el valor -1, no se termina hasta soltar el botón **B**. El proyectil lanzado se da con el valor de *projectileID* y en la dirección vertical dado por *vAngles[0]*. La propiedad *interval* indica el número de marcos a esperar entre cada proyectil.

5.6.8 Character

La clase *Character* deriva de la clase *Movable*. Es la clase más compleja e importante de programa. Representa un personaje controlado por un jugador.

Aspectos básicos del personaje

Los aspectos básicos del personaje se definen con los siguientes valores. *jumpHeight* define la velocidad inicial cuando el personaje salta. *speed* define la velocidad máxima a la que puede correr el personaje. *weight* es un multiplicador que se utiliza para calcular la velocidad cuando el personaje es lanzado. w*idth* y *height* definen la anchura en las dimensiones X y Z y la altura en la dimensión Y.

Los ataques del personaje se almacenan en el vector *normalAttacks*, y sus ataques especiales se almacenan en el vector *specialAttacks*.

Todos los valores del personaje se cargan desde un fichero llamando a la función ***GameData::loadCharacter()***.

Propiedades gráficas

El personaje se compone de 27 animaciones diferentes, cada una actúa como las animaciones definidas en el objeto *GameSprite*. El vector *frames* indica el número de marcos que hay en cada animación. El vector *sizes* indica para cada animación el tamaño [X,Y] de un marco. El vector *magicPixels* indica la posición del píxel de color 0xFFFF00FF para cada animación. Estos píxeles mágicos se utilizan para determinar la posición del rectángulo gráfico en comparación con la posición del personaje. La propiedad *currentAnimation* indica la animación actual.

El personaje tiene varios modelos. La propiedad *model* contiene el rectángulo 2D que representa el personaje de forma gráfica. La propiedad *attackModel* contiene el modelo que se utiliza para representar uno de sus ataques. *ring* contiene el marcador circular que aparece por debajo del personaje, que se utiliza para mostrar su posición y orientación. *shadow* contiene el modelo de la sombra del personaje, que aparece siempre sobre la plataforma más alta por debajo del personaje.

Propiedades dinámicas

El personaje tiene una propiedad *state*, de tipo entero positivo, que almacena una secuencia de bits para indicar diferentes propiedades booleanas del personaje. Se acceden mediente las funciones ***bit()*** para leer un bit, ***bitSet()*** para poner un bit a 1 y ***bitReset()*** para poner un bit a 0.

El valor *percent* indica el daño acumulados del personaje, y *lives* indica el número de vidas que le quedan. Se definen una serie de temporizadores, que tienen el nombre "*Timer". Tienen en cuenta el tiempo de varias acciones del personaje. Por ejemplo, *shieldTimer* indica cuánto tiempo le queda al escudo del personaje. *attackTimer* cuenta cuánto tiemppo ha pasado desde que se comenzó un ataque, *disabledTimer* cuenta el tiempo que está el personaje sin poder realizar más acciones.

Existen algunas propiedades que almacenan objetos activos del personaje en el momento. La propiedad *currentImpact* indica qué golpe está realizando ahora el personaje. *currentSpecialAttack* es un puntero al ataque especial que está haciendo el personaje. currentTarget guarda un puntero al personaje apuntado. *currentPlatform* guarda un puntero a la plataforma sobre la cual está el personaje.

Funciones principales

Las funciones principales son:

- ***animate(int animationID):*** Carga la animación dada por el parámetro animationID. Actualiza la textura en el modelo. Cambia las dimensiones del modelo según el valor de *sizes[currentAnimation]* para adaptarse a las dimensiones de cada marco en la textura actual.
- ***measureLimits(Point3D startLimits, Point3D endLimits):*** Compara el personaje con los límites del escenario. Si no está dentro de los límites, el personaje pierde una vida y se marca como derrotado.
- ***readController(int cameraRotation):*** Lee datos del mando y actualiza el estado del personaje según estos datos. En esta función se comienzan ataques, se pone el o quita el escudo, se establece la velocidad del personaje si es necesario y se hacen otros cambios al personaje relacionados con la entrada del mando.
- ***updateGraphics(int frameNum, Point3D cameraPos, float cameraRotation):*** Actualiza la posición y rotación del modelo del personaje, su sombra y su marcador. Actualiza los modelos de escudo y de ataque si existen. Finalmente, actualiza el estado de la animación y la textura del personaje según el marco actual y la dirección de la cámara.
- ***addToScreen(Abstract3D *screen, Stage *stage):*** Carga todas las imágenes del personaje, buscando en cada una los "píxeles mágicos". Se genera el modelo del personaje, su sombra y su marcador.
- ***frame(int frameNum, Point3D cameraPos, float cameraRotation):*** Comproueba si el personaje está en el suelo, actualiza los temporizadores y llama a ***readController()*** para realizar acciones relacionadas con el mando. Después actualiza el estado del ataque si el personaje está atacando. Luego mueve la posición del personaje, añade gravedad si el personaje no está en el suelo. Finalmente comprueba otros datos del estado del personaje y actualiza la animación si es necesario.

5.6.8 Stage

La clase *Stage* deriva de la clase *EventDispatcher*. Representa el escenario del juego, que contiene a todos los personajes, plataformas, proyectiles y objetos de decorado. Almacena en *playerPositions* las posiciones iniciales de todos los personajes en el escenario. En *startLimits* y *endLimits* almacena los límites del escenario. Si un personaje se sale de estos límites, perderá una vida.

Guarda varias listas de objetos del escenario en objetos de tipo *LinkedList*: Plataformas en *platforms*, personajes en *characters*, proyectiles en *projectiles* y objetos de decorado y *sprites*. Además, tiene un puntero a la lista de jugadores, de tipo *Player*, en la propiedad *players*. Esta lista sirve para acceder al mando del jugador.

Tiene el identificador de la textura de fondo en la propiedad *backgroundID* y un modelo en forma de cilindro en donde mapea esta textura en la propiedad *cylinder*. Este modelo se genera con la función ***Abstract3D::makeCylinder()***.

El escenario guarda un contador del tiempo global en *currentFrame*. Este contador se utiliza principalmente para coordinar plataformas móviles. El valor de *objective* indica el centro de atención del la cámara. La posición de la cámara se almacena en la propiedad *camera* y se calcula utilizando los valores del *camRadius*, *camAngle* y *camVAngle*, que representan la distancia, la rotación horizontal y la rotación vertical de la cámara con respecto a su objetivo.

Las funciones principales son:

- ***updateCamera():*** Primera calcula la posición del objetivo de la cámara, que es la media aritmética de las posiciones de todos los personajes que aún estén presentes en el escenario. Después se calcula el valor de *camAngle* según el ángulo de los dos personajes más distantes entre sí. Finalmente se calcula la propiedad *camRadius* según la distancia entre estos dos personajes. Se actualiza el valor de camera y, finalmente, se establece la posición de la cámara en la pantalla según *camera* y *objective*.
- ***getTarget(Character *lastTarget, Characters *caller):*** Dado el personaje actual *caller* y su último objetivo *lastTarget*, se devuelve un puntero al nuevo objetivo de *caller*.
- ***addCharacter(Character *character):*** Se añade el personaje al final de la lista de personajes.
- ***addProjectile(Projectile *projectile):*** Se añade el proyectil al final de la lista de proyectiles y se añade el modelo del proyectil a la pantalla llamando a ***Projectile::addToScreen()***.
- ***removeProjectile(Projectile *projectile):*** Se elimina el proyectil de la lista y se destruye. En la función del destructor, el proyectil se encarga de eliminar su modelo de la pantalla.
- ***setupStage(Abstract3D *abstract3D, Player **playerInput):*** Se guardan punteros a los parámetros de entrada: La pantalla y la lista de jugadores. Se crea el cilindro que representa el fondo. Se añaden todas las plataformas, personajes y objetos de decorado a la pantalla. Asigna a cada personaje un jugador y un color, y pone al personaje en su posición inicial.
- ***frame():*** Actualiza las plataformas, los proyectiles, los elementos de decorado y los personajes llamando a ***frame()*** para cada una. Elimina aquellos proyectiles que se han salido de los límites del escenario y llama a ***Character::measureLimits()*** para cada personaje. Después compara la intersección entre los personajes y las plataformas. También compara la intersección entre pares de personajes. Debido a que es un prototipo, no se hacen más comparaciones. Finalmente, se actualizan los gráficos de los personajes llamando a ***Character::updateGraphics()*** y se actualiza la cámara.

5.7 Otras clases

Se preparó un sistema para almacenar datos del juego. Para ello, se implementó una clase *SaveData*. Esta clase no realiza ninguna función vital para el programa. Los datos guardados corresponden a personajes y escenarios desbloqueados, y se almacenan en un fichero nombrado *"futzu.sola"*. Si el fichero no existe, el juego creará uno nuevo.

Para implementar algunas de las listas encadenadas, se implementó una clase plantilla, *LinkedList*. Algunos de los datos del juego utilizan una clase *Point3D*, para representar puntos en el espacio 3D. Tiene únicamente tres propiedades (x, y, z) y ninguna función.

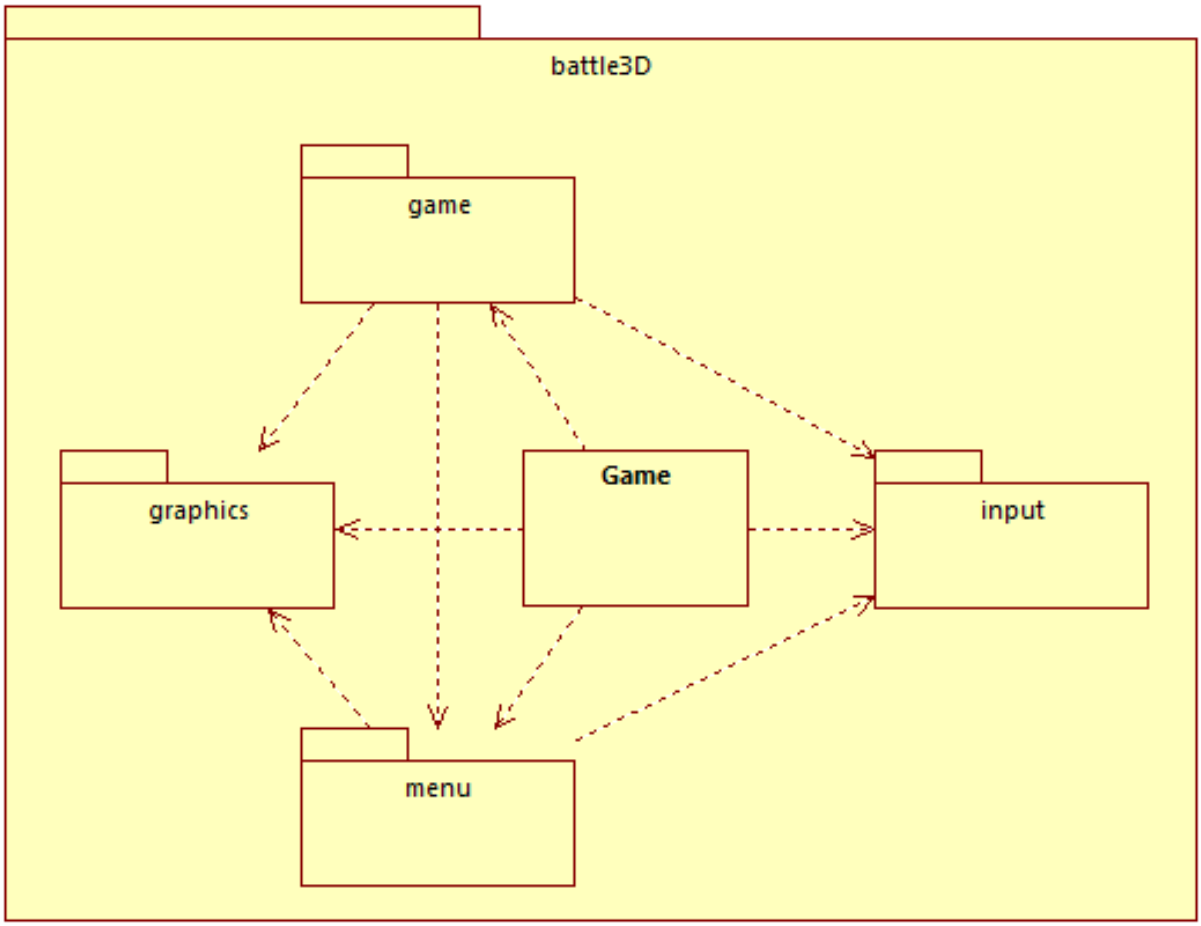
Para algunas funciones no implementadas, se crearon algunas clases adicionales. La clase *GameFont* implementaría la fuente de un texto para mostrar en la pantalla. Las clases *Network*, *NetworkClient*, *NetworkServer* y *MultiCaster* estaban diseñados para implementar varios aspectos de juego en línea. Era complicado implementar un conexión de red en tiempo real y se abandonó la idea. La clase *GameSound* implementaría sonido utilizando la librería DirectSound.

6 Implementación final del juego

La versión final del juego se creo a partir del prototipo de OpenGL. A continuación se describen las clases del proyecto, su funcionamiento y sus componentes más importantes.

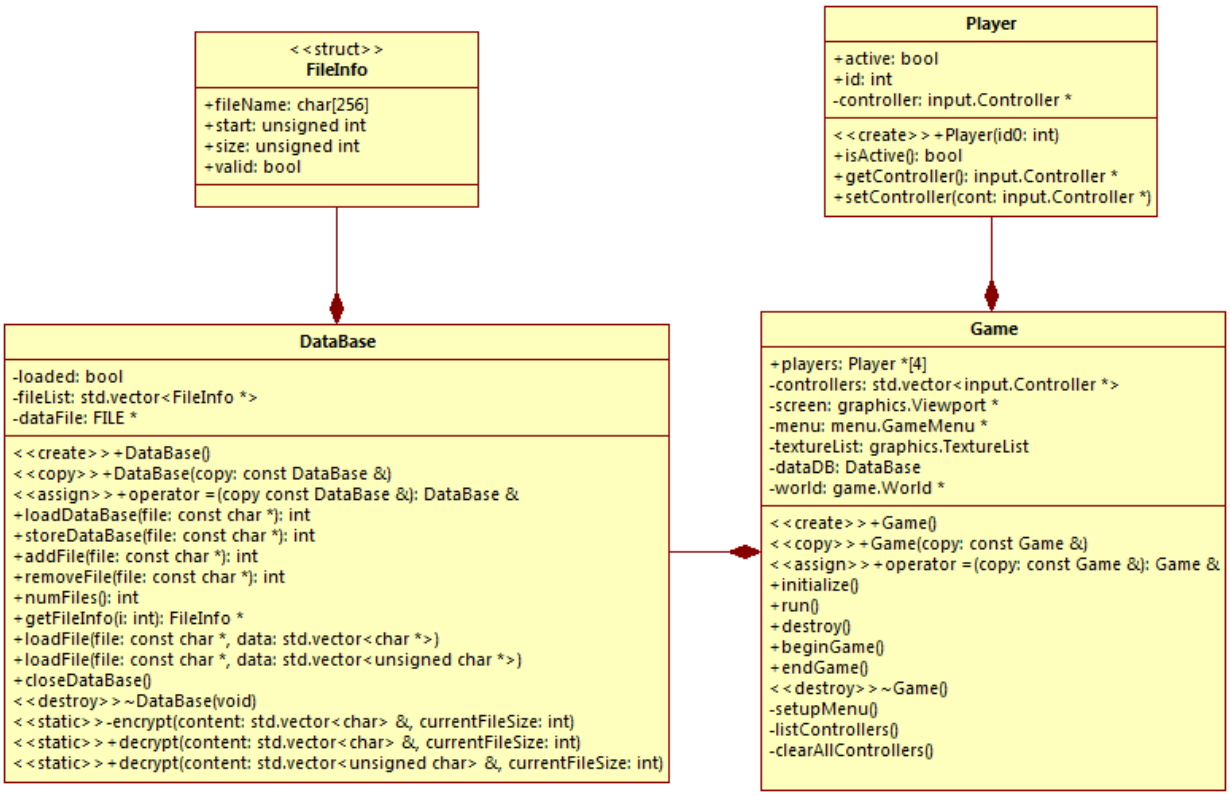
6.1 División en módulos

Para darle al programa una estructura sencilla y robusta, se ha dividido en cuatro módulos separados. Cada módulo tiene una función particular: Gráficos, entradas de datos, menú y juego. Estos módulos se encapsulan en un módulo principal, que contiene el programa entero.



6.2 Módulo principal: battle3D

El módulo principal contiene clases relevantes a la ejecución global del juego.



6.2.1 battle3D::Game

La clase *Game* juega el papel de programa principal. En él se almacenan objetos relacionados con el menú, los gráficos, los jugadores, las entradas de datos, el acceso a ficheros y el juego en ejecución. La propiedad *screen* contiene un objeto Viewport para mostrar gráficos en la pantalla. La propiedad *controllers* contiene un vector de todos los disposivitos de entrada conectados que sean compatibles con el juego. La propiedad *menu* contiene el menú del juego y la propiedad *world* contiene el escenario si se está ejecutando un juego.

La clase *Game* ejecuta tres funciones principales:

- **initialize():** Esta función se ejecuta al abrir el programa. Carga datos de ficheros. Luego inicializa los gráficos, las entradas de datos y el menú.
- **run():** Esta función se ejecuta repetidamente durante toda la ejecución del programa. Está diseñado para ejecutarse 60 veces por segundo. Lee las entradas de datos, luego actualiza el menú y si hay un juego en ejecución, se actualiza el juego. Finalmente actualiza los gráficos.
- **destroy():** Si alguna acción dentro o fuera del programa causa la terminación del programa, se ejecuta esta función. Esta función libera la memoria de todo el programa: Los gráficos, el menú y los objetos relacionados con la entrada de datos.

La clase también contiene dos funciones adicionales relacionados con la ejecución de un juego:

- ***beginGame():*** Crea un nuevo objeto de tipo *game::World*, que representa un escenario y crea un objeto de tipo *game::Character* para cada jugador que ha seleccionado un personaje en el menú. Los objetos *World* y *Character* cargan datos del escenario y los personajes desde fichero de datos e imágenes desde una base de datos de imágenes. Luego llama a la función ***begin()*** del objeto *world* para comenzar la ejecución del juego.
- ***endGame():*** Muestra la pantalla de victoria en el menú, recoge datos del juego finalizado y finalmente elimina de la memoria el escenario, los personajes y todas las texturas relacionadas con ellos.

Además de estas funciones, la clase contiene algunas otras funciones importantes

- ***setupMenu():*** Esta función genera el menú, las pantallas del menú y todos los objetos contenidos en cada pantalla. Se hace uso de algunos macros para generar estos elementos. Los macros y las funciones de los botones se implementan en los ficheros *setupmenu.h* y *setupmenu.cpp*.
- ***listControllers():*** Esta función busca a todos los dispositivos de entrada que hay conectados al ordenador y genera objetos derivados de la clase Controller para manejar cada uno. Estos objetos se guardan en la propiedad *controllers*.

6.2.2 battle3D::DataBase

La clase *DataBase* se utiliza para leer y escribir conjuntos de ficheros en un solo fichero de datos. Tiene dos tareas principales: Reducir el número el ficheros necesarios para ejecutar el programa correctamente y evitar, al menos en cierta medida, que se hagan copias de los ficheros sin el permiso del autor. En el juego se utiliza únicamente para leer datos e imágenes de los ficheros. Las imágenes se almacenan en un fichero llamado *"images.dat"* y los datos se almacenan en *"data.dat"*. Estos ficheros son esenciales para el funcionamiento del programa.

Las funciones importantes para el juego son:

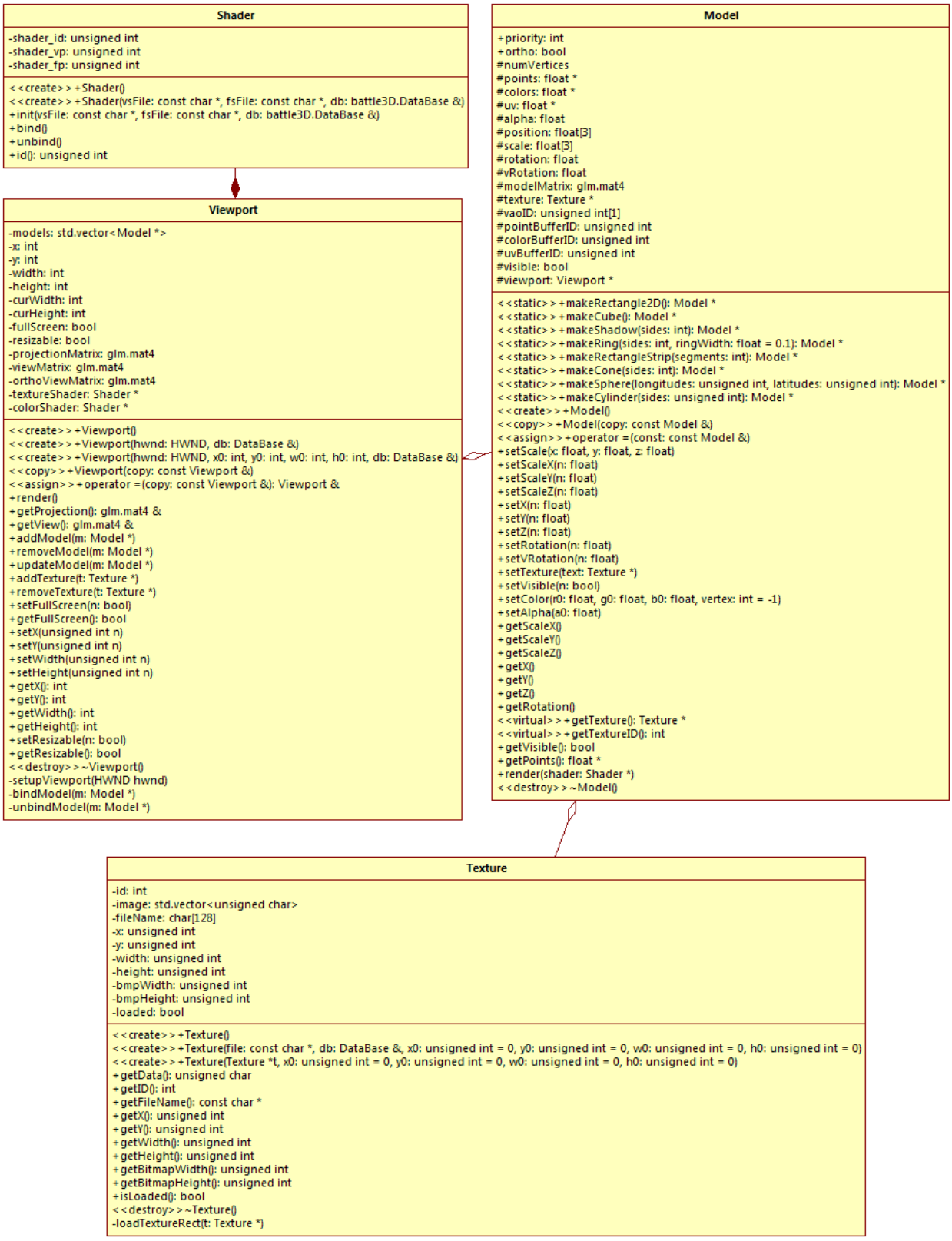
- **loadDataBase(const char *fileName):** Abre un fichero conteniendo la base de datos.
- **numFiles():** Devuelve el número de ficheros guardados en la base de datos.
- **getFileInfo(int i):** Devuelve un objeto con el nombre del fichero en la posición *i*.
- **loadFile(const char *file, std::vector<char> *data):** Escribe en el vector *data* los datos contenidos en el fichero de nombre *file*.
- **closeDataBase():** Cierra el fichero que representa la base de datos.

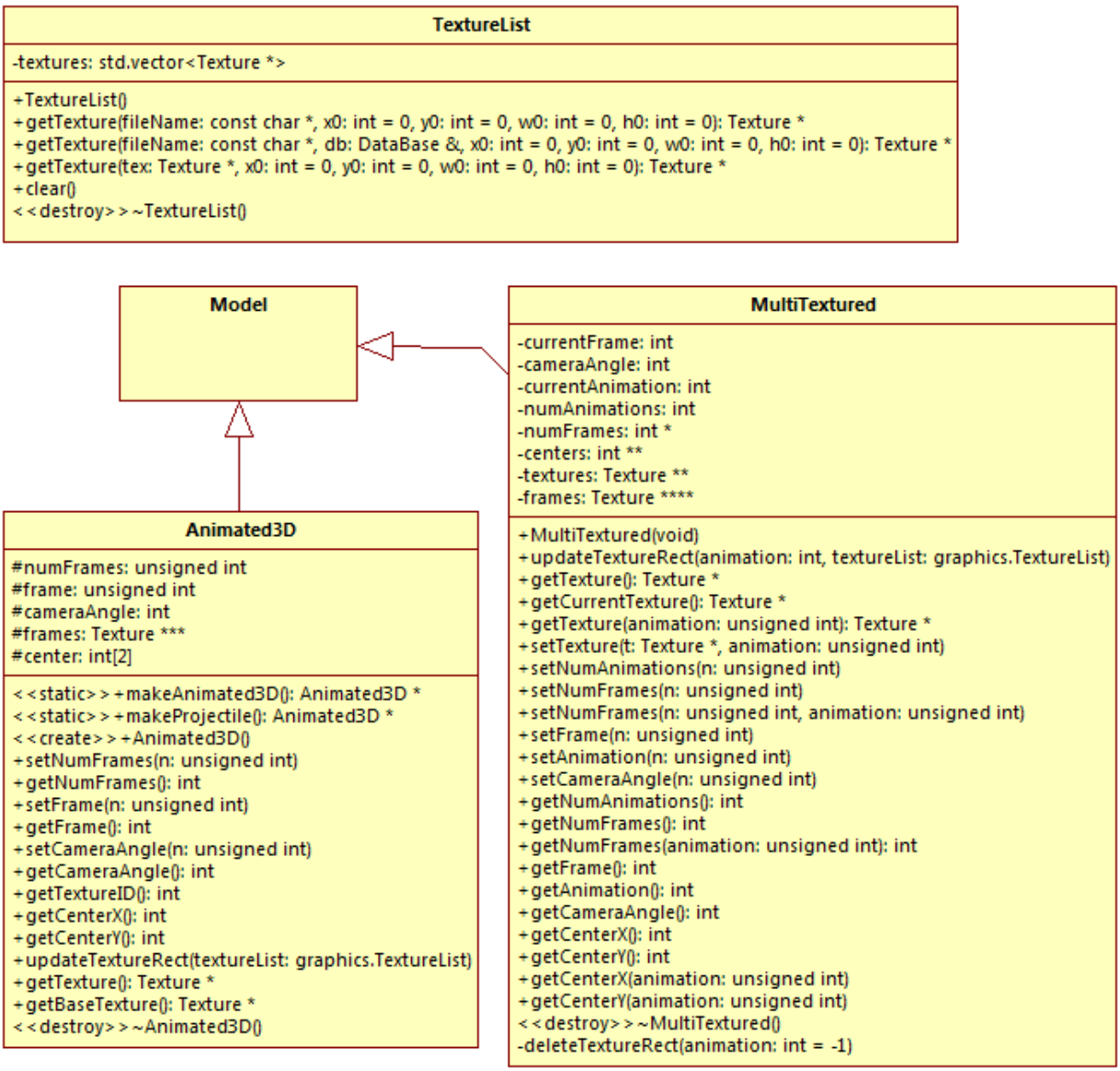
6.2.3 battle3D::Player

La clase *Player* representa un jugador humano. Sirve principalmente para poder intercambiar las entradas de datos en caso de que uno de los jugadores desee utilizarlo. También sirve para activar o desactivar jugadores no activos sin tener que desenchufar los mandos conectados. La función ***isActive()*** devuelve verdadero si el jugador está activo y tiene conectado una entrada de datos. Se pueden asignar diferentes entradas al jugador mediante la función ***setController()***.

6.3 Módulo de gráficos

El módulo de gráficos controla la muestra de datos en la pantalla y las estructuras de datos que almacenan información gráfica sobre objetos 2D y 3D.





6.3.1 battle3D::graphics::Viewport

La clase *Viewport* representa la pantalla. Almacena una lista de modelos y texturas para mostrar. También sirve como una capa de abstracción sobre OpenGL. Cuando se crea un objeto Viewport, se genera un contexto de OpenGL. Al crear un objeto *Viewport*, se debe pasar una referencia a una ventana, de tipo *HWND*. Se permite añadir valores de posición y escala a la ventana. La función ***setupViewport()*** se encarga de crear un contexto OpenGL.

Las funciones más importantes son las siguientes:

- ***Viewport(HWND hwnd, DataBase &db[, ...]):*** Los constructores generan un contexto OpenGL con una escala y posición específicos. Además, carga shaders programables de la base de datos dada por *db*.
- ***addModel(Model *m):*** Añade un modelo a la lista de modelos a renderizar y guarda en memoria gráfica los vértices del modelo.
- ***removeModel(Model *m):*** Elimina un modelo de la lista, si existe. Además, borra sus datos de la memoria gráfica.

- ***updateModel(Model *m):*** Actualiza los datos de un modelo en memoria gráfica.
- ***addTexture(Texture *t):*** Almacena la textura en memoria gráfica.
- ***removeTexture(Texture *t):*** Elimina la textura de la memoria gráfica.
- ***render():*** Se renderizan los modelos de la lista según su orden de prioridad. Cuanto mayor sea el valor de prioridad, más tarde se renderiza el modelo. Si dos modelos tienen la misma prioridad, se renderiza antes el primero que se insertó en la lista.

6.3.2 battle3D::graphics::Texture

Representa una textura en formato RGBA, representando cada color con 8 bits. Las funciones principales son:

- ***loadBitmap(const char *file, DataBase &db,[, ...]):*** Carga un mapa de bits desde un fichero. Si se especifican los parámetros *x0*, *y0*, *w0* y *h0*, se extrae una porción rectangular de la imagen.
- ***Texture(Texture *t[, ...]):*** Realiza una copia de una textura. Si se especifican los parámetros *x0*, *y0*, *w0* y *h0*, se extrae una porción rectangular de la imagen.
- ***isLoaded():*** Devuelve verdadero si la textura contiene un mapa de bits válido.

6.3.3 battle3D::graphics::TextureList

Representa un conjunto de texturas. Su función principal es poder manejar fácilmente los grupos de texturas de manera que no ocurran pérdidas de memoria. Se aconseja hacer todas las creaciones y destrucciones de objetos Texture mediante esta clase. Sus funciones principales son:

- ***getTexture(const char *fileName, int x0=0, int y0=0, int w0=0, int h0=0):*** Devuelve una textura con el nombre y las dimensiones especificadas. Si no lo encuentra, devuelve nulo.
- ***getTexture(const char *fileName, DataBase &db, int x0=0, int y0=0, int w0=0, int h0=0):*** Devuelve una textura con el nombre y las dimensiones especificadas. Si no existe, lo creará con el fichero especificado en la base de datos. Si el fichero no existe, devuelve nulo.
- ***getTexture(Texture *tex, int x0=0, int y0=0, int w0=0, int h0=0):*** Busca una textura con el mismo nombre que la textura especificada. Si no lo encuentra, devuelve nulo.
- ***clear():*** Borra la lista y elimina todas las texturas de la memoria.

6.3.4 battle3D::graphics::Model

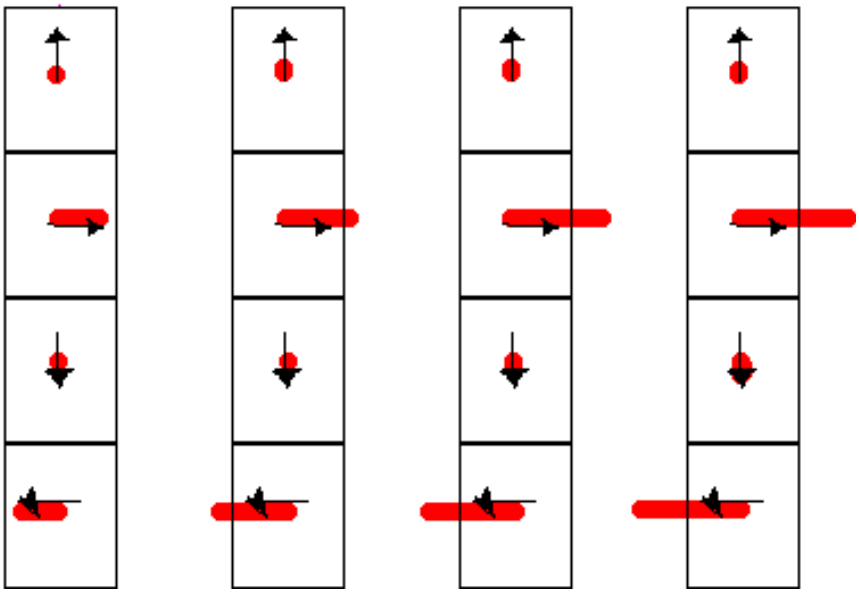
Representa un modelo tridimensional. Contiene tres listas de datos para los vértices: *points* para las posiciones [X,Y,Z], *uv* para las coordenadas [U,V] de la textura y *colors* para los valores [R,G,B] del color. Además tiene una propiedad *texture*, un puntero a una textura asociada al modelo. También tiene las propiedades *position*, *scale*, *rotation* y *vRotation* para representar transformaciones del modelo.

Las funciones más importantes son:

- ***make*():*** Las funciones estáticas ***make*()*** sirven para generar modelos con valores específicos. Por ejemplo: cubos, cilindros, conos, círculos y rectángulos. En este proyecto no se puede crear un modelo útil sin utilizar alguna función de creación de este tipo.
- ***render(Shader *shader):*** Se renderiza el modelo en OpenGL utilizando los shaders programables referenciados por *shader*.

6.3.5 battle3D::graphics::Animated3D

Esta clase hereda propiedades de la clase *Model*. Representa un objeto rectangular vertical con una textura animada. La textura se divide en cinco filas de igual tamaño, cada una representando un ángulo de la cámara. Cada fila se divide en marcos, representando un instante en el tiempo. El resultado es una matriz como se muestra en la siguiente imagen:



La primera fila se utiliza para marcar el centro del rectángulo, con un píxel de color magenta, de valor hexadecimal FF00FF, y no se muestra durante el juego. El centro del rectángulo sirve para posicionar los vértices del rectángulo con respecto a la posición [X,Y,Z] del modelo.Las filas resultantes muestran el personaje desde detrás, la izquierda, delante y la derecha respectivamente. Basado en una textura base y el número de marcos que contiene la animación, se generan 5x*numFrames* punteros a texturas, que se almacenan en la matriz *frames* en el formato [Marco, Ángulo de cámara].

Las funciones más importantes son:

- ***setNumFrames(unsigned int n):*** Define el número de marcos que contiene la animación.
- ***setFrame(unsigned int n):*** Especifica el marco específico a mostrar.
- ***setCameraAngle(unsigned int n):*** Especifica un ángulo de cámara a mostrar. Debe ser un número entre 1 y 4.
- ***updateTextureRect(graphics::TextureList &textureList):*** Después de actualizar la textura o el número de marcos, es necesario llamar a esta función manualmente para poder actualizar la matriz de texturas. Se utiliza el parámetro *textureList* para un mejor manejo de la memoria. Cuando se actualizan las texturas, se hace una búsqueda del píxel magenta en el marco [0,0]. Si encuentra el píxel, guarda sus coordenadas [X,Y] en la propiedad *center*.

6.3.6 battle3D::graphics::MultiTextured

Esta clase hereda propiedades de la clase *Model*. De modo similar a la clase Animated3D, representa un rectángulo vertical con una textura animada. Esta clase permite guardar múltiples animaciones a la vez.

Se almacenan todos los marcos en la matriz *frames* en el formato [Animación, Marco, Ángulo de cámara]. Además, se guarda en *textures* las texturas base de cada animación, en *centers* los centros de cada animación y en *numFrames* el número de marcos en cada animación.

Las funciones más importantes son:

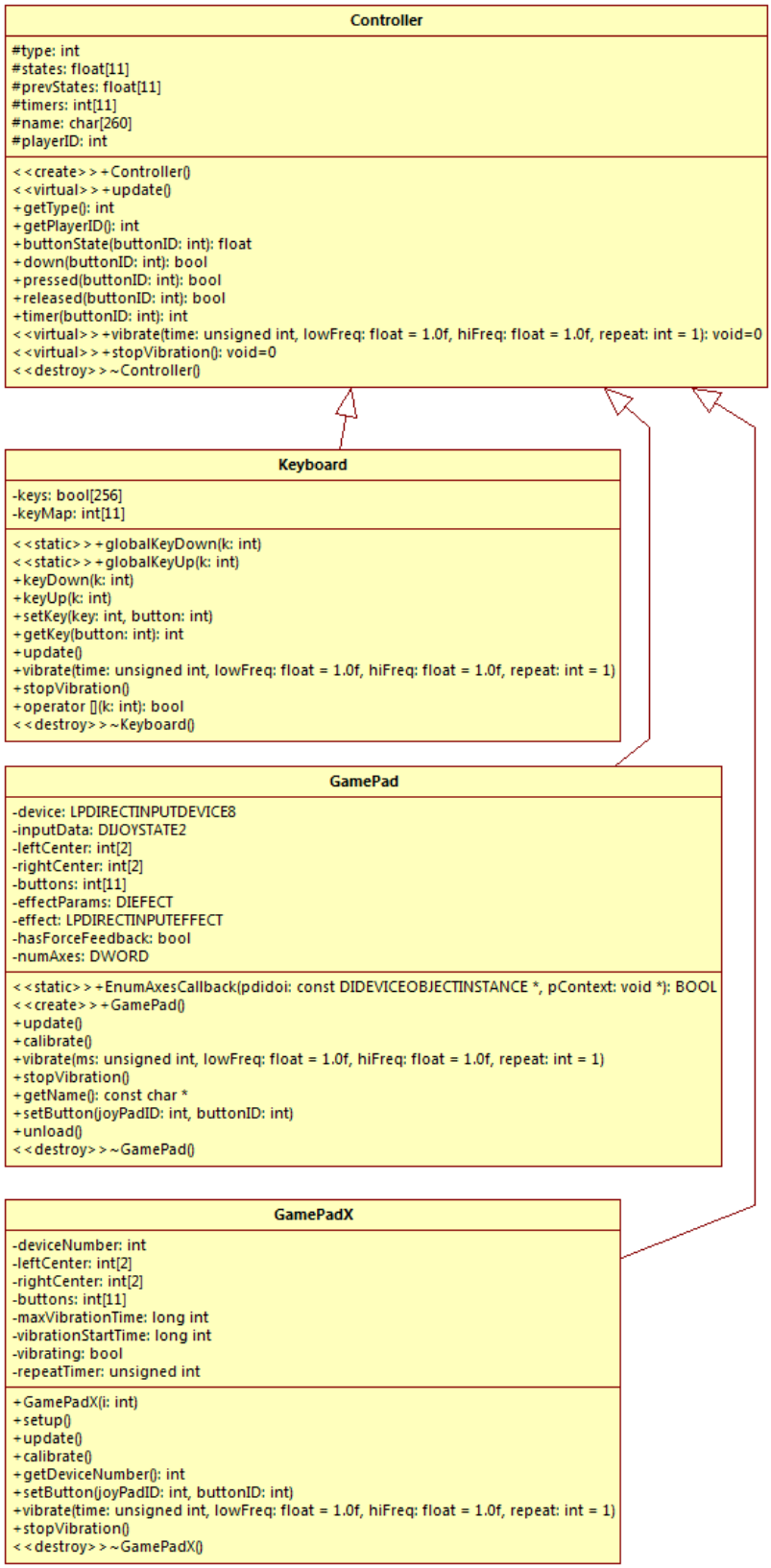
- ***setNumAnimations(unsigned int n):*** Define el número de animaciones que tiene el objeto.
- ***setNumFrames(unsigned int n, unsigned int animation):*** Define el número de marcos que tiene la animación dada por *animation*.
- ***setTexture(Texture *t, unsigned int animation):*** Asigna una textura a la animación dada por *animation*.
- ***setFrame(unsigned int n):*** Define el marco a mostrar en la animación actual.
- ***setAnimation(unsigned int n):*** Define la animación a mostrar. El marco actual se establece a 0.
- ***setCameraAngle(unsigned int n):*** Define el ángulo de cámara a mostrar. Debe ser un valor entre 1 y 4.
- ***updateTextureRect(int animation, graphics::TexgtureList &textureList):*** Actualiza la matriz *frames* según el número de animaciones y el número de marcos en cada animación. Además, se actualiza el vector *centers* con los centros de cada animación.

6.3.7 battle3D::graphics::Shader

Esta función encapsula el funcionamiento de los shaders programables en OpenGL. Con la función ***init()***, se cargan dos ficheros contienendo el shader de vértices y el shader de fragmentos respectivamente. Los programas se compilan y se almacenan en la memoria gráfica. Una vez creados los programas, se utiliza la función ***bind()*** para aplicar el shader y ***unbind()*** para dejar de aplicarlo.

6.4 Módulo de entrada

El módulo de entrada sirve como una capa de abstracción entre el juego y los dispositivos de entrada. Los dispositivos de entrada pueden ser de tres tipos: teclado, mando compatible con DirectInput y mando de XBox.



6.4.1 battle3D::input::Controller

La clase *Controller* es la clase base de la cual derivan clases para cada tipo de entrada. Para hacer uso de esta clase, de deberá crear una instancia de alguna de sus clases derivadas. Esta clase representa un mando ficticio como el que se muestra en el apartado **4**. El mando consta de 7 botones binarios - **A**, **B**, **X**, **Y**, **L**, **R** y **START** - y 4 valores continuos, entre 0 y 1 - **Izquierda**, **Derecha**, **Arriba** y **Abajo**.

Para representar el estado de cada botón, se utiliza el vector *states*. Además, para detectar cambios de estado, se guarda en *prevStates* los valores de estado en el momento anterior. Se almacena en un vector *timers* el número de instantes que han pasado desde el último cambio de estado del botón.

Las funciones más importantes son:

- ***buttonState(int ButtonID):*** Devuelve el valor numérico del estado del botón dado por *buttonID*.
- ***down(int buttonID):*** Devuelve verdadero si el estado del botón es mayor que 0.5.
- ***pressed(int buttonID):*** Devuelve verdadero si el estado del botón era menor o igual a 0.5 en el instante anterior y es mayor que 0.5 en el estado actual.
- ***released(int buttonID):*** Devuelve verdadero si el estado del botón era mayor que 0.5 en el instante anterior y es menor o igual a 0.5 en el estado actual.
- ***timer(int buttonID):*** Devuelve el número de instantes que han pasado desde el último cambio de estado del botón dado por *buttonID*.
- ***vibrate()*** y ***stopVibration()***: Estas funciones sirven para aplicar vibración al mando si tiene capacidad para ello. De este modo

6.4.2 battle3D::input::Keyboard

La clase *Keyboard* representa el teclado. Este programa solo reconoce la existencia de un teclado. Por lo tanto, todas las entradas de teclado se envían a la misma instancia del teclado mediante las funciones estáticas ***globalKeyDown()*** y ***globalKeyUp()***.

Las dos propiedades principales de la clase *Keyboard* son *keys*, que almacena el estado booleano de cada tecla y *keyMap*, que almacena la asignación de cada botón a una tecla.

Las funciones principales son:

- ***keyDown(int k):*** Cambia el estado de la tecla *k* a verdadero.
- ***keyUp(int k):*** Cambia el estado de la tecla *k* a falso.
- ***setKey(int key, int button):*** Asigna la tecla *key* al botón *button* del mando ficticio.
- ***update():*** Almacena en *prevStates* los valores de estado actual y guarda en *states* el estado de las teclas asignadas a los botones del mando.

6.4.3 battle3D::input::GamePad

La clase *GamePad* representa un mando de juego compatible con DirectInput. Almacena en el vector *buttons* el número de cada botón del mando real asignado con cada botón del mando ficticio.

Si el mando tiene un stick analógico, se utiliza el vector *leftCenter* para marcar la posición neutra del stick en los ejes X e Y. Los valores izquierda, derecha, arriba y abajo se calculan según la diferencia entre el valor actual del stick analógico y el centro dado por *leftCenter*.

Si el mando es compatible con vibración, el valor *hasForceFeedback* será verdadero. Se utilizan los valores *effectParams* y *effect* se utilizan para controlar las vibraciones del mando mediante la función ***vibrate()*** y ***stopVibration()***. Es importante reconocer que - si el mando realmente tiene tecnología **Force Feedback -** estas funciones van a efectuar fuerzas reales sobre el stick analógico o su equivalente, haciendo más difícil la habilidad del jugador de jugar al juego.

Las funciones más importantes son:

- ***update():*** Almacena en *prevState* el valor del estado actual. Después actualiza el vector *state* con los valores de los botones binarios. Luego lee los valores de entrada direccionales del stick analógico y de los botones direccionales (si existen) y actualiza el valor de *state* según estos valores. Si hay un conflicto de valores direccionales - por ejemplo, el stick analógico está a la izquierda y el botón direccional a la derecha - se elige el valor con la mayor magnitud.
- ***calibrate():*** Asigna la posición actual del stick analógico como su posición neutra. Almacena los valores en el vector *leftCenter*.
- ***setButton(int joyPadID, int buttonID):*** Almacena en la posición buttonID del vector *buttons* el valor del botón dado por *joyPadID*.
- ***vibrate()*** y ***stopVibrate():*** Se sobrecargan las funciones de la clase Controller para generar efectos de vibración en el mando.

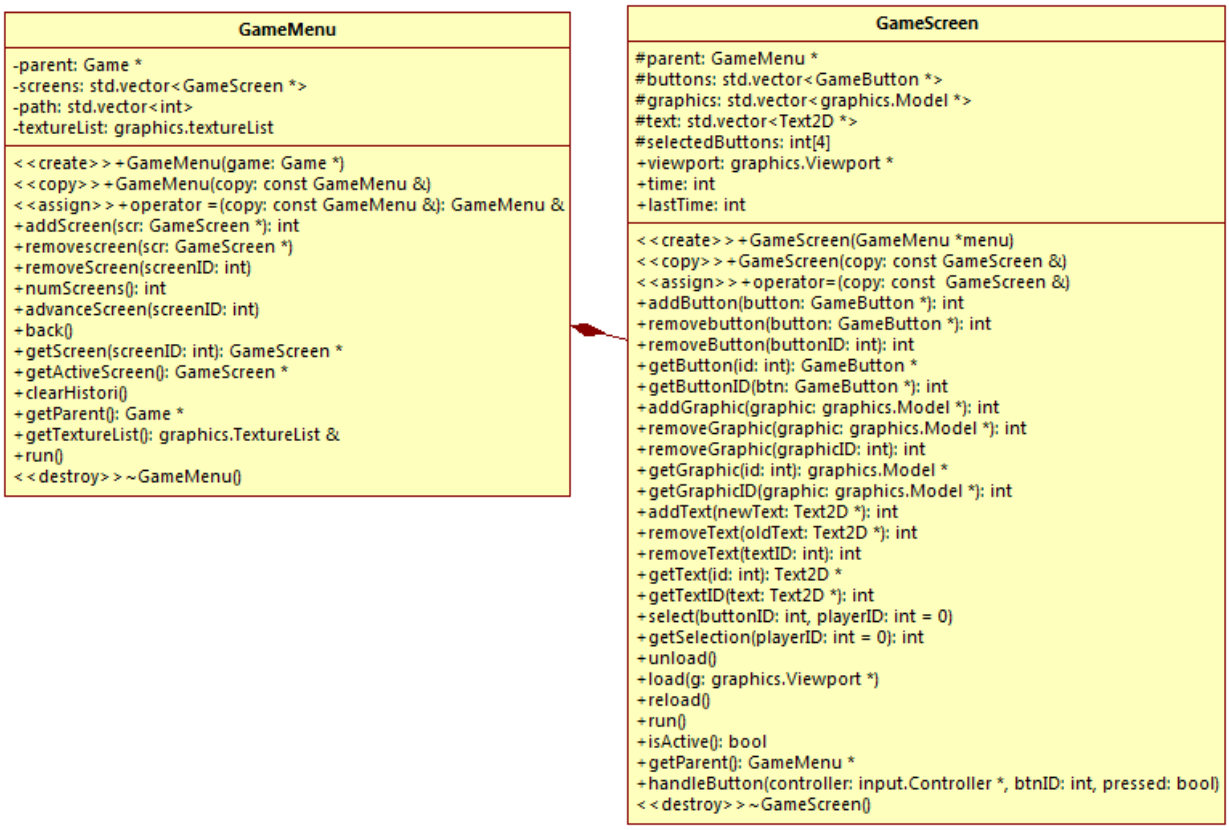
6.4.4 battle3D::input::GamePadX

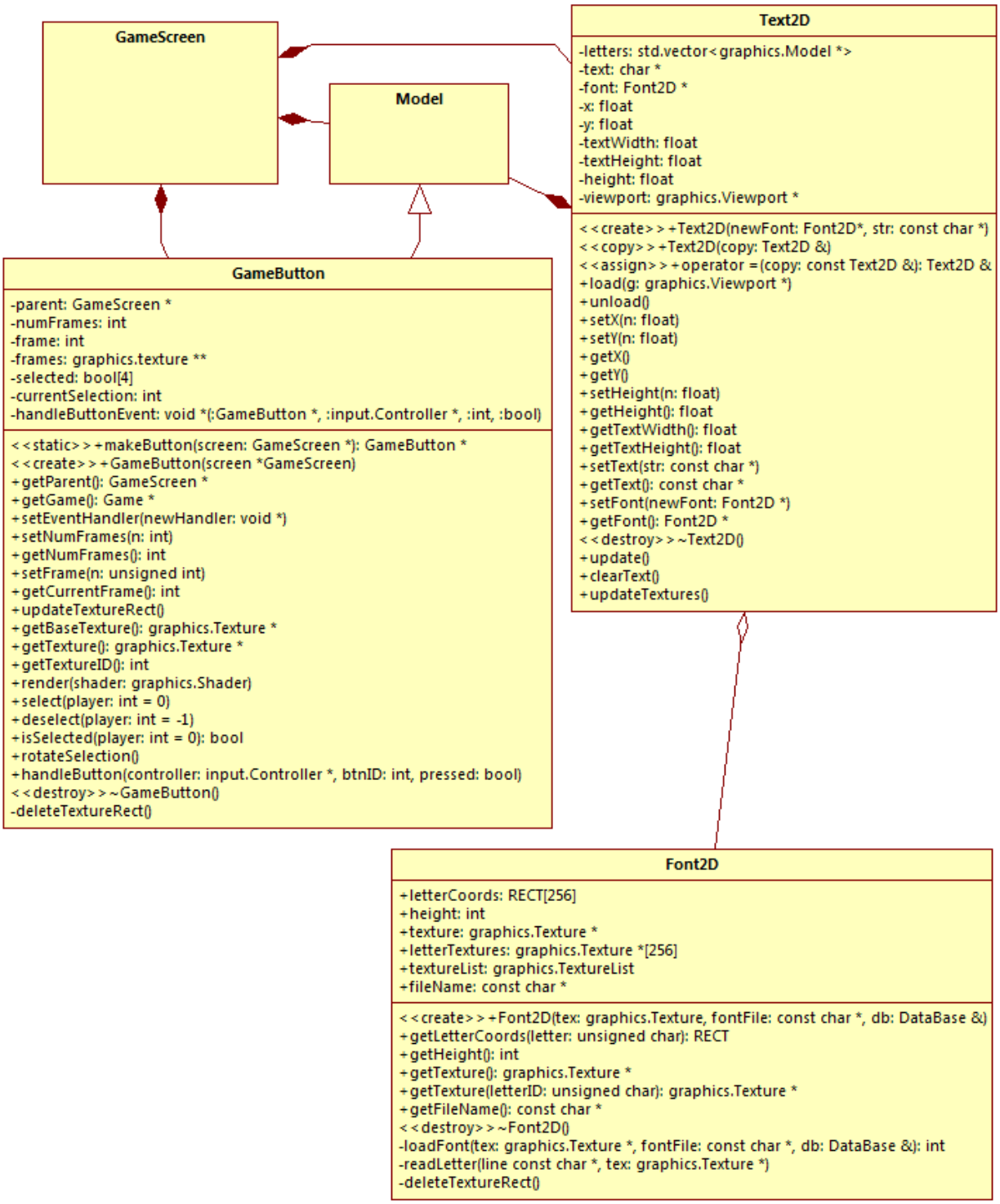
Representa un mando de XBox o un mando compatible con XInput. Tiene una funcionalidad similar a la clase *GamePad*, pero utiliza la librería XInput para implementar esta funcionalidad.

Además de la implementación interna de las funciones, una de las deferencias con respecto a la clase *GamePad* es que no se guarda una instancia de DirectInput sino el número del mando XBox a utilizar. Además, se utilizan valores de vibración distintos a los de la clase *GamePad*.

6.5 Módulo de menú

El módulo de menú hace uso del módulo de entrada y el módulo de gráficos para generar un menú. El menú se implementa con la clase *GameMenu* y se compone de un conjunto de pantallas, cada una representada por un objeto de la clase *GameScreen*. Cada pantalla se compone de botones, texto y elementos gráficos, implementados mediante las clases *GameButton*, *Text2D* y *Model* respectivamente. Los textos implementados con la clase *Text2D* hacen uso de la clase *Font2D* para especificar la forma y la textura de cada uno de los caracteres.




GameScreen
Model
Text2D
-letters: std.vector< graphics.Model *>
-text: char *
-font: Font2D *
-x: float
-y: float
-textWidth: float
-textHeight: float
-height: float
-viewport: graphics.Viewport *
<<create>> +Text2D(newFont: Font2D*, str: const char *)
<<copy>> +Text2D(copy: Text2D &)
<<assign>> +operator =(copy: const Text2D &): Text2D &
+load(g: graphics.Viewport *)
+unload()
+setX(n: float)
+setY(n: float)
+getX()
+getY()
+setHeight(n: float)
+getHeight(): float
+getTextWidth(): float
+getTextHeight(): float
+setText(str: const char *)
+getText(): const char *
+setFont(newFont: Font2D *)
+getFont(): Font2D *
<<destroy>> ~Text2D()
+update()
+clearText()
+updateTextures()
GameButton
-parent: GameScreen *
-numFrames: int
-frame: int
-frames: graphics.texture **
-selected: bool[4]
-currentSelection: int
-handleButtonEvent: void *(:GameButton *, :input.Controller *, :int, :bool)
<<static>> +makeButton(screen: GameScreen *): GameButton *
<<create>> +GameButton(screen *GameScreen)
+getParent(): GameScreen *
+getGame(): Game *
+setEventHandler(newHandler: void *)
+setNumFrames(n: int)
+getNumFrames(): int
+setFrame(n: unsigned int)
+getCurrentFrame(): int
+updateTextureRect()
+getBaseTexture(): graphics.Texture *
+getTexture(): graphics.Texture *
+getTextureID(): int
+render(shader: graphics.Shader)
+select(player: int = 0)
+deselect(player: int = -1)
+isSelected(player: int = 0): bool
+rotateSelection()
+handleButton(controller: input.Controller *, btnID: int, pressed: bool)
<<destroy>> ~GameButton()
-deleteTextureRect()
Font2D
+letterCoords: RECT[256]
+height: int
+texture: graphics.Texture *
+letterTextures: graphics.Texture *[256]
+textureList: graphics.TextureList
+fileName: const char *
<<create>> +Font2D(tex: graphics.Texture, fontFile: const char *, db: DataBase &)
+getLetterCoords(letter: unsigned char): RECT
+getHeight(): int
+getTexture(): graphics.Texture *
+getTexture(letterID: unsigned char): graphics.Texture *
+getFileName(): const char *
<<destroy>> ~Font2D()
-loadFont(tex: graphics.Texture *, fontFile: const char *, db: DataBase &): int
-readLetter(line const char *, tex: graphics.Texture *)
-deleteTextureRect()

6.5.1 battle3D::menu::GameMenu

La clase *GameMenu* representa el menú completo del juego. Se compone de varias pantallas, que se almacenan el vector *screens*, que almacena punteros a objetos de tipo *GameScreen*. Cada pantalla se puede acceder mediante su posición en el vector. Se deberá tener en cuenta esto si se eliminan pantallas de la lista. La propiedad *path* guarda el camino que ha seguido el usuario mientras navegaba el menú. De este modo puede volver atrás. Cada objeto GameMenu contiene un objeto de tipo TextureList. En él se almacenarán todas las texturas generadas por los objetos contenidos en el menú.

Las funciones más importantes son

- ***addScreen(GameScreen *scr):*** Añade una nueva pantalla al final de la lista y devuelve su posición.
- ***numScreens():*** Devuelve el número de pantallas en la lista.
- ***advanceScreen(int screenID):*** Almacena el valor *screenID* al final del vector *path* y carga la pantalla en esa posición del vector *screens*. Si la pantalla ya se encuentra en el camino, se busca la posición de *screenID* en el vector *path* y se eliminan todos los valores posteriores a esa posición.
- ***back():*** Elimina la pantalla actual del vector *path* y carga la pantalla representada por el último valor de este vector.
- ***getScreen(int screenID):*** Devuelve la pantalla en la posición *screenID* del vector *screens*.
- ***clearHistory():*** Elimina la pantalla actual y vacía el vector *path*.
- ***getParent():*** Devuelve una referencia al objeto *Game*, que representa el juego entero.
- ***run():*** Sirve para actualizar el menú. Ejecuta la función ***run()*** de la pantalla actual, si existe.

6.5.2 battle3D::menu::GameScreen

Representa una pantalla en el menú del juego. La clase contiene un puntero al objeto *GameMenu* que lo contiene.

Los objetos de la clase *GameScreen* contienen vectores de objetos del menú. El vector *buttons* almacena objetos de la clase *GameButton*. El vector *text* almacena objetos de la clase *Text2D*. El vector *graphics* almacena objetos de la clase *Model*. Aunque funciona bien con cualquier objeto de tipo *Model*, está diseñado para representar aquellos objetos creadas con la función estática ***Model::makeRectangle2D()***, que representa un gráfico de dos dimensiones.

En una pantalla del menú, hasta 4 jugadores pueden tener marcados diferentes botones. El vector *selectedButtons* guarda el identificador del botón señalado por cada uno de los jugadores. Si un jugador no está señalando a ningún botón, su elemento en el vector tendrá el valor -1. Las propiedades *time* y *lastTime* tienen la función específica de alternar periódicamente los botones que estén seleccionados por múltiples jugadores.

Las funciones más importantes son:

- ***addButton(), addGraphic(), addText():*** Añaden botones, gráficos y texto al menú. Devuelve un identificador del objeto añadido. El texto siempre aparece por encima de los botones, y los botones aparecen por encima de los gráficos.
- ***removeButton(), removeGraphic(), removeText():*** Dado un puntero a un objeto o su identificador, se elimina el objeto de la pantalla del menú.
- ***getButton(int id), getGraphic(int id), getText(id):*** Devuelve el botón, texto o gráfico con el identificador dado por *id*.

- ***getButtonID(), getGraphicID(), getTextID():*** Devuelve el identificador del objeto si existe en la pantalla.
- ***select(int buttonID, int playerID=0):*** El jugador dado por *playerID* señala al botón de identificado *buttonID*.
- ***load(graphics::Viewport *g):*** Añade todos los botones, modelos y gráficos a la memoria gráfica mediante el objeto *Viewport*. Se guarda una referencia a este objeto. Como resultado, se mostrará el contenido del objeto *GameScreen* en la pantalla.
- ***unload():*** Elimina a los objetos de la memoria gráfica y elimina la referencia al objeto *Viewport*.
- ***reload():*** Ejecuta la función ***unload()*** seguido de la función ***load()*** para actualizar la pantalla con cualquier cambio que se haya hecho al objeto *GameScreen*.
- ***run():*** De manera periódica, actualiza el estado de los botones que estén señalados por múltiples jugadores.
- ***isActive():*** Devuelve verdadero si se está mostrando el contenido de *GameScreen* en la pantalla en este momento.
- ***getParent():*** Devuelve un puntero al objeto *GameMenu* al que pertenece el objeto *GameScreen*.
- ***handleButton(input::Controller *controller, int btnID, bool pressed):*** Se busca el jugador asociado al objeto *controller*. Después se busca en *selectedButtons* el identificador del botón seleccionado por este jugador. Si existe, se llama a la función ***handleButton()*** del objeto *GameButton* seleccionado por el jugador.

6.5.3 battle3D::menu::GameButton

Representa un botón en una pantalla del menú. La clase deriva de la clase *Model*. Se crea una instancia del botón utilizando la función estática ***GameButton::makeButton()***. La clase GameButton realiza la mayoría de la actividad del menú, convirtiéndola en la clase más importante del módulo.

El gráfico del botón se compone de un rectángulo de vértices y una textura dividida horizontalmente en una fila de marcos. Los marcos son objetos de tipo *Texture* que se almacenan en el vector *frames*. Cada marco representa la apariencia del botón cuando está seleccionado por uno de los jugadores. El primer marco representa el botón cuando no está seleccionado. El valor *numFrames* indica cuántos marcos hay en total y *frame* indica cuál de los marcos se está mostrando en cada momento.

El botón contiene un vector de valores booleanos, *selected*, que indica si el botón está seleccionado por el jugador de cada posición. La propiedad *currentSelection* indica cuál de los jugadores se está representando en el momento actual.

Cada botón contiene una variable de tipo puntero a una función, *handleButtonEvent*. Se hace una llamada a esta función cada vez que cambie el estado del mando del jugador que ha seleccionado el botón. Se permite al programador elegir qué hará cada botón cuando ocurra esta llamada. La función tiene cuatro parámetros:

- *btn* es un puntero al botón mismo. Esto es necesario porque la función está fuera de la clase Gamebutton y no existe una referencia implícita al objeto.
- *controller* es una referencia al mando que ha enviado un comando.
- *btnID* es el número del botón en el mando ficticio que ha cambiado.
- *pressed* es un valor booleano indicando si el jugador acaba de pulsar el botón o si acaba de soltarlo.

Las funciones más importantes son:

- ***makeButton(GameScreen *screen):*** Esta función crea y devuelve una nueva instancia de la clase GameButton. El botón guarda una referencia a la pantalla *screen*, a la cual pertenece.
- ***getParent():*** Devuelve una referencia a la pantalla que le pertenece.
- ***getGame():*** Devuelve una referencia al objeto Game, representando el programa principal. Devuelve el mismo valor que hacer la llamada getParent()->getParent()->getParent(). A partir de esta referencia, el botón tiene acceso a propiedades del juego en general.
- ***setEventHandler(newHandler):*** Asigna al botón una función a ejecutar cuando recibe un comando de un objeto de tipo *Controller*.
- ***setNumFrames(int n):*** Cambia el número de marcos que contiene el botón.
- ***setFrame(unsigned int n):*** Cambia el marco que se muestra actualmente.
- ***updateTextureRect():*** Actualiza los marcos almacenados en la propiedad *frames*. Se debe llamar a esta función después de llamar a ***setTexture()*** o ***setNumFrames()***.
- ***select(int player=0):*** Marca el botón como seleccionado para el jugador identificado por *player*. El valor de *player* debe ser entre 0 y 3.
- ***deselect(int player=-1):*** Marca el botón como no seleccionado por el jugador identificado por *player*. Si se utiliza el valor -1, se marca como no seleccionado para todos los jugadores.
- ***isSelected(int player=0):*** Devuelve verdadero si el jugador dado por *player* ha seleccionado el botón.
- ***rotateSelection():*** Actualiza el marco actual del botón. Si está seleccionado por múltiples jugadores, se elige el marco asociado al siguiente jugador. Si no está seleccionado, muestra el marco 0.

6.5.4 battle3D::menu::Text2D

La clase *Text2D* se utiliza para mostrar texto en la pantalla. El texto se almacena en la propiedad *text*. Los caracteres se extraen de un objeto *Font2D* a través del puntero *font*. Los caracteres se implementan con objetos de tipo *Model*, que se almacenan en el vector *letters*.

Los valores *x*, *y*, *textWidth* y *textHeight* definen la posición y el tamaño del rectángulo que envuelve el texto en su totalidad. La propiedad *height* define la distancia entre una línea y la siguiente. Además, controla la altura base del texto. Todos los caracteres se escalan según este valor.

Las funciones principales son:

- ***load(graphics::Viewport *g):*** Almacena los caracteres en memoria gráfica a través del objeto *Viewport* y guarda un puntero al objeto en la propiedad *viewport*.
- ***unload():*** Elimina los caracteres de la memoria gráfica y elimina el puntero al objeto de tipo *Viewport*.
- ***setX(float n)***, ***setY(float n):*** Actualiza las posiciónes X e Y del rectángulo que envuelve al texto. La posición se mide en píxeles.
- ***setHeight(float n):*** Actualiza la altura base del texto.
- ***getTextWidth()***, ***getTextHeight():*** Devuelve la anchura y la altura del rectángulo que envuelve el texto
- ***setText(const char *str):*** Cambia el texto a mostrar en la pantalla.
- ***getText():*** Devuelve el texto que se muestra en la pantalla.
- ***setFont(Font2D *newFont):*** Cambia la fuente del texto que se muestra en la pantalla.
- ***update():*** Actualiza la posición y la escala de los caracteres. Esta función es llamada por las funciones ***setX()***, ***setY()*** y ***setHeight()***.
- ***clearText():*** Elimina los caracteres del texto y vacía el vector *letters*.

- ***updateTextures():*** Actualiza las texturas de cada carácter. Esta función se llama en la función ***setFont()***.

6.5.5 battle3D::menu::Font2D

Representa una fuente para el texto almacenado en un objeto Text2D. Cada fuente se genera a partir de dos ficheros: Una imagen conteniendo todos los caracteres de la fuente y un fichero de texto que almacena la posición y el tamaño en píxeles de cada letra dentro de la imagen. El fichero se compone de texto plano dividido en filas. Cada fila tiene el formato:

```
<Caracter>,X,Y,Altura,Anchura
```

El carácter espacio " " es obligatorio. La altura de este carácter define la altura base de la fuente. Esta altura se utiliza para definir el espaciado entre las líneas de un texto y se almacena en la propiedad *height*. Las posiciones y los tamaños de cada letra se almacenan en el vector *letterCoords*.

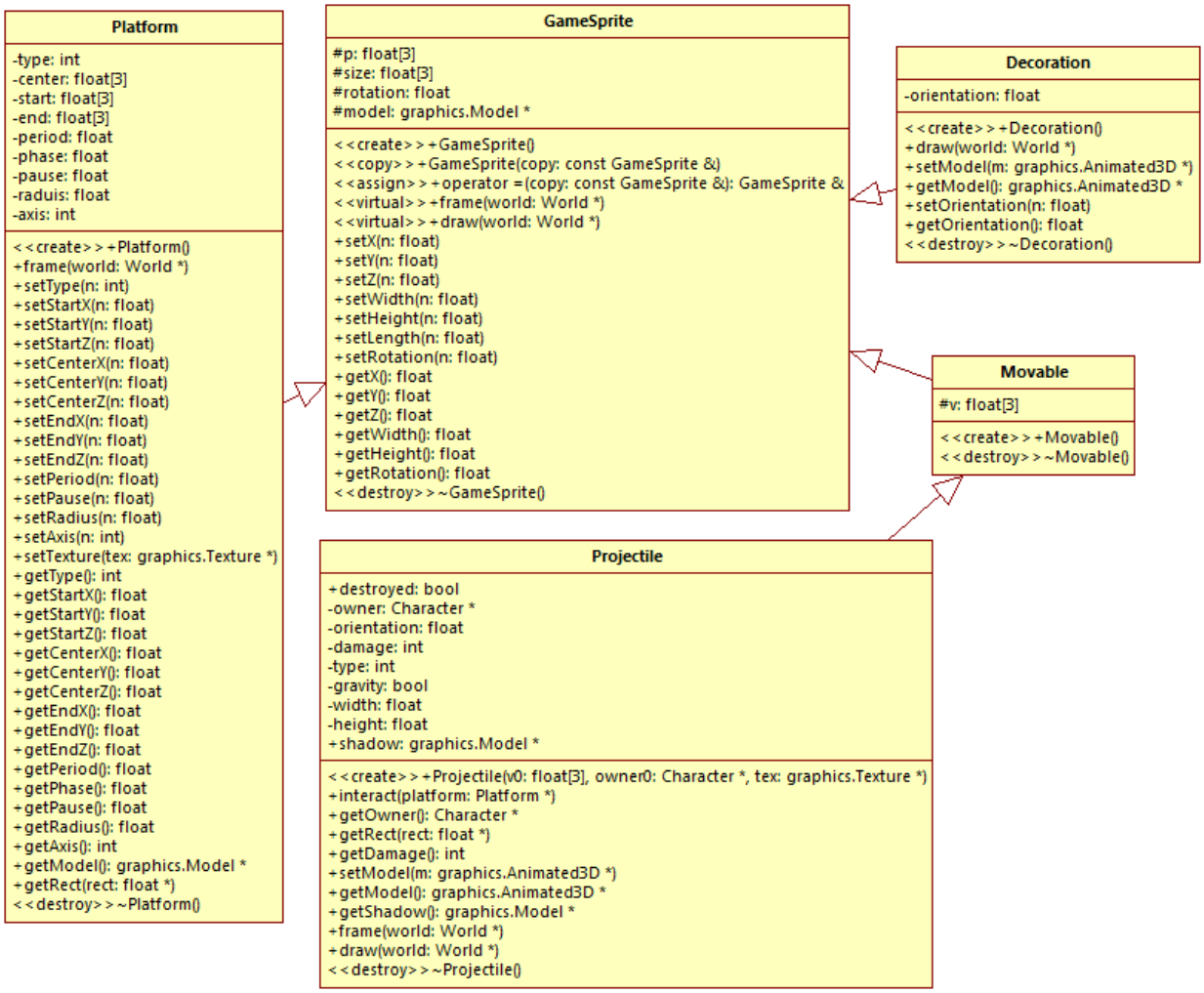
La clase Font2D contiene una textura base en la propiedad *texture*, que representa la imagen de los caracteres y un vector, *letterTextures*, de texturas para cada carácter, que se extraen de la textura base. Para el manejo de texturas en memoria, cada objeto Font2D contiene su propio objeto *TextureList* para las texturas de cada carácter.

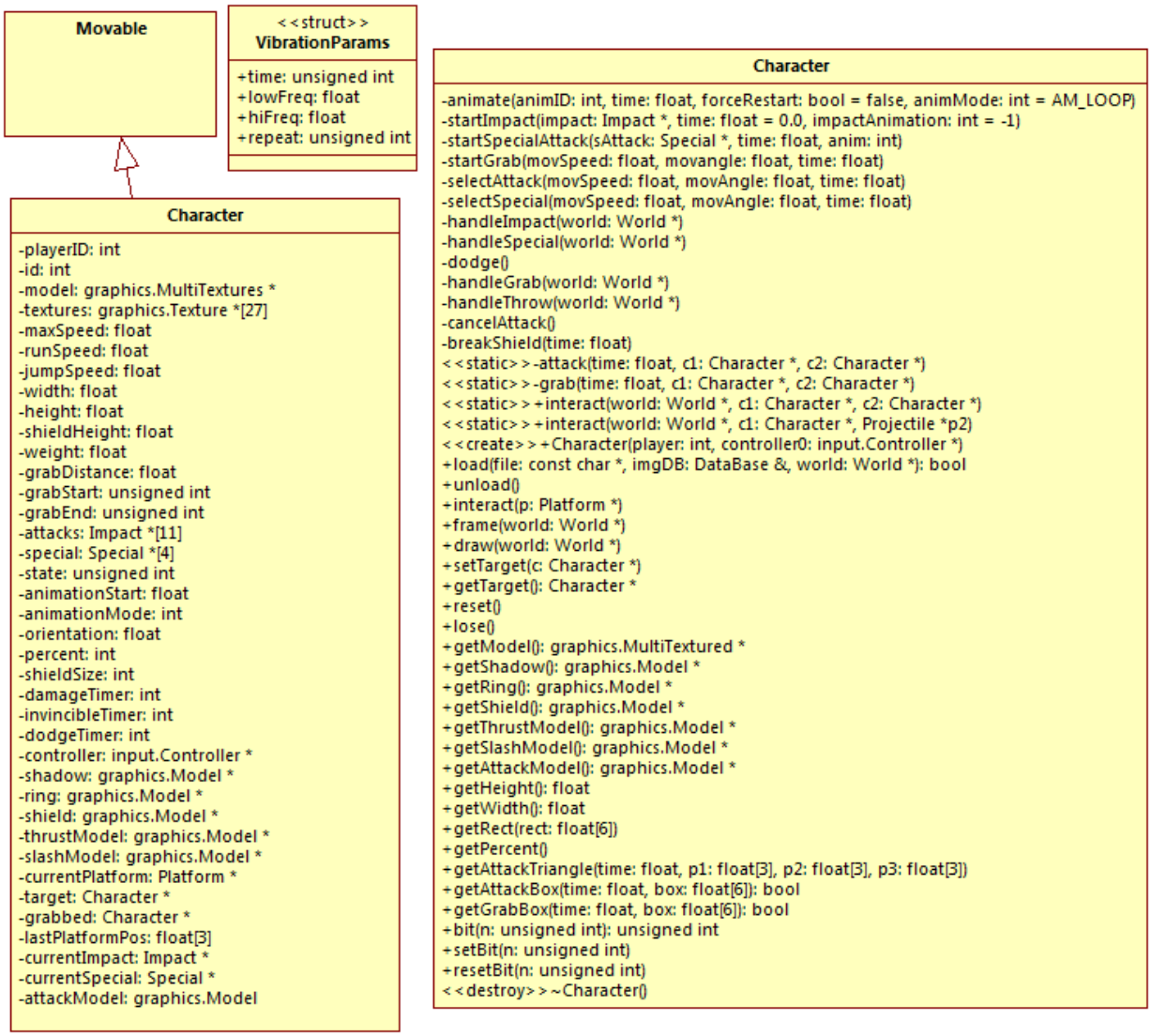
Las funciones más importantes son:

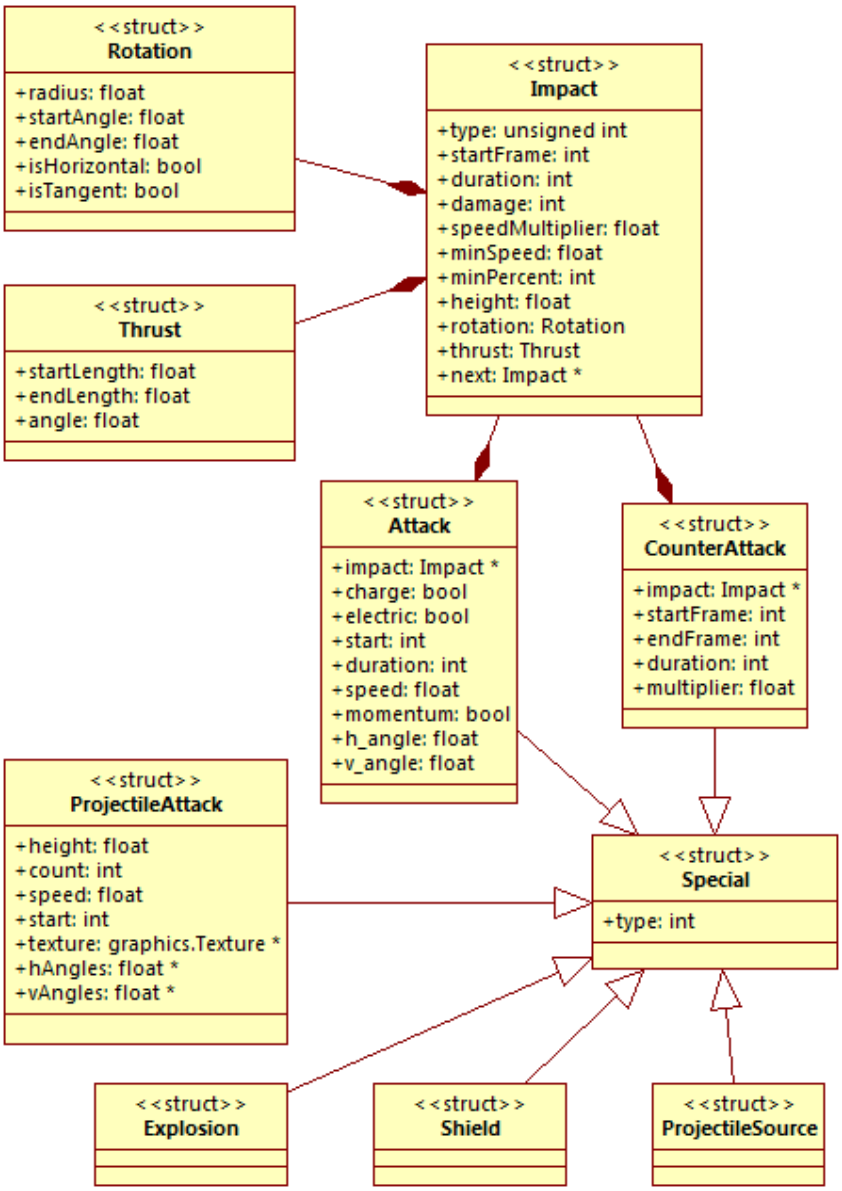
- ***Font2D(graphics::Texture *tex, const char *fontFile, DataBase &db):*** Crea el objeto *Font2D*, lee los datos del fichero de nombre *fontFile* contenido en la base de datos *db*. Se llama a la función ***loadFont()*** con la textura y la información del fichero.
- ***loadFont(graphics::Texture *tex, std::vector<char> data):*** El parámetro *tex* contiene la imagen de los caracteres y el parámetro data contiene la información de los caracteres extraido del fichero. Esta función lee el fichero fila a fila y llama a la función ***readLetter()*** para cada fila.
- ***readLetter(const char *line, Texture *tex):*** Lee información de un carácter de una fila del fichero de datos y rellena uno de los elementos del vector *letterCoords*. Además, crea una textura y guarda un puntero a la textura en el vector *letterTextures*.
- ***getTexture(unsigned char letterID):*** Dado un carácter, se devuelve la textura correspondiente al carácter.
- ***getLetterCoords(unsigned char letter):*** Dado un carácter, se devuelve el rectángulo que define su posición y tamaño en la imagen.
- ***getHeight():*** Devuelve la altura base de la fuente.

6.6 Módulo de juego

El módulo de juego contiene todas las clases relacionadas con el juego en sí: El escenario, las plataformas, los personajes y clases de otros objetos más abstractos. Es el módulo más importante y complejo del programa.




Movable
<<struct>>
VibrationParams
+time: unsigned int
+lowFreq: float
+hiFreq: float
+repeat: unsigned int
Character
-playerID: int
-id: int
-model: graphics.MultiTextures *
-textures: graphics.Texture *[27]
-maxSpeed: float
-runSpeed: float
-jumpSpeed: float
-width: float
-height: float
-shieldHeight: float
-weight: float
-grabDistance: float
-grabStart: unsigned int
-grabEnd: unsigned int
-attacks: Impact *[11]
-special: Special *[4]
-state: unsigned int
-animationStart: float
-animationMode: int
-orientation: float
-percent: int
-shieldSize: int
-damageTimer: int
-invincibleTimer: int
-dodgeTimer: int
-controller: input.Controller *
-shadow: graphics.Model *
-ring: graphics.Model *
-shield: graphics.Model *
-thrustModel: graphics.Model *
-slashModel: graphics.Model *
-currentPlatform: Platform *
-target: Character *
-grabbed: Character *
-lastPlatformPos: float[3]
-currentImpact: Impact *
-currentSpecial: Special *
-attackModel: graphics.Model
Character
-animate(animID: int, time: float, forceRestart: bool = false, animMode: int = AM_LOOP)
-startImpact(impact: Impact *, time: float = 0.0, impactAnimation: int = -1)
-startSpecialAttack(sAttack: Special *, time: float, anim: int)
-startGrab(movSpeed: float, movangle: float, time: float)
-selectAttack(movSpeed: float, movAngle: float, time: float)
-selectSpecial(movSpeed: float, movAngle: float, time: float)
-handleImpact(world: World *)
-handleSpecial(world: World *)
-dodge()
-handleGrab(world: World *)
-handleThrow(world: World *)
-cancelAttack()
-breakShield(time: float)
<<static>>-attack(time: float, c1: Character *, c2: Character *)
<<static>>-grab(time: float, c1: Character *, c2: Character *)
<<static>>+interact(world: World *, c1: Character *, c2: Character *)
<<static>>+interact(world: World *, c1: Character *, Projectile *p2)
<<create>>+Character(player: int, controller0: input.Controller *)
+load(file: const char *, imgDB: DataBase &, world: World *): bool
+unload()
+interact(p: Platform *)
+frame(world: World *)
+draw(world: World *)
+setTarget(c: Character *)
+getTarget(): Character *
+reset()
+lose()
+getModel(): graphics.MultiTextured *
+getShadow(): graphics.Model *
+getRing(): graphics.Model *
+getShield(): graphics.Model *
+getThrustModel(): graphics.Model *
+getSlashModel(): graphics.Model *
+getAttackModel(): graphics.Model *
+getHeight(): float
+getWidth(): float
+getRect(rect: float[6])
+getPercent()
+getAttackTriangle(time: float, p1: float[3], p2: float[3], p3: float[3])
+getAttackBox(time: float, box: float[6]): bool
+getGrabBox(time: float, box: float[6]): bool
+bit(n: unsigned int): unsigned int
+setBit(n: unsigned int)
+resetBit(n: unsigned int)
<<destroy>>~Character()


<<struct>>
Rotation
+radius: float
+startAngle: float
+endAngle: float
+isHorizontal: bool
+isTangent: bool
<<struct>>
Thrust
+startLength: float
+endLength: float
+angle: float
<<struct>>
Impact
+type: unsigned int
+startFrame: int
+duration: int
+damage: int
+speedMultiplier: float
+minSpeed: float
+minPercent: int
+height: float
+rotation: Rotation
+thrust: Thrust
+next: Impact *
<<struct>>
Attack
+impact: Impact *
+charge: bool
+electric: bool
+start: int
+duration: int
+speed: float
+momentum: bool
+h_angle: float
+v_angle: float
<<struct>>
CounterAttack
+impact: Impact *
+startFrame: int
+endFrame: int
+duration: int
+multiplier: float
<<struct>>
ProjectileAttack
+height: float
+count: int
+speed: float
+start: int
+texture: graphics.Texture *
+hAngles: float *
+vAngles: float *
<<struct>>
Special
+type: int
<<struct>>
Explosion
<<struct>>
Shield
<<struct>>
ProjectileSource


GameScreen
World
+paused: bool
+pauseOwner: int
+ready: bool
+timeSpeed: float
+characters: Character *[4]
+lives: int[4]
+victories: int[4]
+losses: int[4]
-time: float
-lastTime: int
-start: float *[4]
-limits: float[6]
-startTime: int
-font: menu.Font2D *
-frameRate: int
-background: graphics.Model *
-charactersRemaining: int
-characterDamage: int[4]
-percent: menu.Text2D *[4]
-platforms: std.vector<Platform *>
-decorations: std.vector<Decoration *>
-projectiles: std.vector<Projectile *>
-textureList: graphics.TextureList
-camDist: float
-camVAngle: float
-camHAngle: float
-camDegrees: float
-camSpeed: float
-camPos: glm.vec3
-camCenter: glm.vec3
-camUp: glm.vec3
-loadFile(fileName: const char *, imgDB: DataBase &)
-defeat(i: int)
<<create>>+World(: menu.GameMenu *, : const char *, : graphics.Viewport *, : menu.Font2D *, : int, : DataBase &)
+addCharacter(playeRID: unsigned int, c: Character *)
+begin()
+end()
+load()
+unload()
+frame()
+getTimeMultiplier(): float
+isGameOver(): bool
<<destroy>>~World()

6.6.1 battle3D::game::GameSprite

La clase *GameSprite* es una versión abstracta de todos los objetos contenidos en el escenario. Almacena información sobre la posición, escala y rotación de un objeto en el espacio tridimensional. Estos valores se almacenan en las propiedades *p*, *size* y *rotation*. La propiedad *rotation* representa la rotación del objeto con respecto al eje vertical, Y. La propiedad *model* contiene un objeto de clase *Model* para representar los gráficos del objeto.

La clase declara dos funciones ***frame()*** y ***draw()*** que sirven para actualizar datos y actualizar los elementos gráficos del objeto. Estas funciones se pueden sobrecargar en las clases que derivan de la clase *GameSprite*.

6.6.2 battle3D::game::Platform

La clase *Platform* deriva de la clase *GameSprite*. Representa una plataforma en el escenario. La plataforma puede ser de tres tipos: fija, lineal o circular. El tipo viene dado por la propiedad *type*.

Las plataformas fijas se mantienen en un sitio. Las plataformas lineales se mueven periódicamente entre dos puntos, dados por los vectores *start* y *end*, de manera periódica. El período viene dado por la propiedad *period*. Cuando la plataforma llega a cualquiera de los extremos, se queda en esa posición durante el tiempo dado por la propiedad *pause*. Las plataformas circulares se mueven de manera circular alrededor de un punto central, dado por la propiedad *center*. El radio del círculo viene dado por la propiedad radius. La propiedad *axis* determina el eje de rotación de la plataforma.

El elemento gráfico de la plataforma es un objeto de tipo *Model* se crea con la función ***Model::makeCube()***. La textura que envuelve el cubo se divide en una matriz de 4x3 rectángulos de igual tamaño. La siguiente imagen muestra la distribución de la textura sobre las caras del cubo.

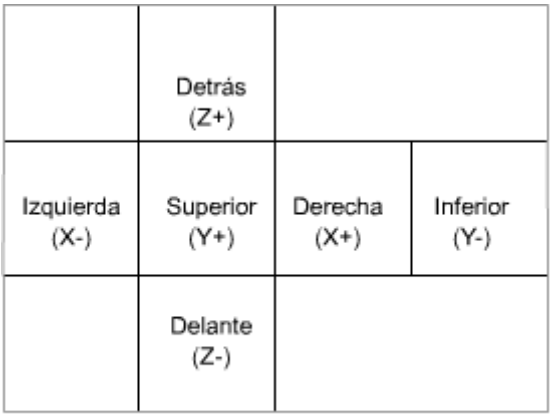


Imagen de la textura de una plataforma

La plataforma tiene una función ***getRect(float *rect)***. Esta función escribe en el vector *rect* la posición y las dimensiones de la plataforma en el formato [X,Y,Z,Anchura, Altura, Profundidad].

6.6.3 battle3D::game::Decoration

La clase *Decoration* deriva de la clase *GameSprite*. Representa un objeto plano rectangular con una textura animada. Utiliza un objeto de la clase *Animated3D* como elemento gráfico.

La propiedad *orientation* que indica la rotación del objeto en grados. Un valor de 0 significa que el rectángulo se ve más claramente desde el eje Z. Si se utiliza un valor negativo, el objeto siempre se orientará hacia la posición de la cámara.

6.6.4 battle3D::game::Movable

La clase *Movable* deriva de la clase *GameSprite*. Es una clase abstracta que representa un objeto con velocidad. Contiene un vector *v*, que representa la velocidad del objeto en cada dimensión.

6.6.5 battle3D::game::Projectile

La clase *Projectile* deriva de la clase *Movable*. Representa un proyectil lanzado por uno de los personajes. La propiedad *owner* es un puntero al personaje que lanzó el proyectil.

La propiedad *damage* indica el daño que hará al siguiente personaje que toque. Si el valor de *gravity* es verdadero, el proyectil caerá bajo el efecto de la gravedad. En caso contrario, se mueve de forma lineal según su vector de velocidad. El proyectil seguirá moviéndose hasta que colisiones con un personaje, una plataforma o uno de los límites del escenario. La propiedad *destroyed* indica que se ha marcado el proyectil para eliminarlo.

El proyectil tiene una caja de colisión dada por los valores *width* y *height*. La propiedad *width* indica la dimensiones horizontales de la caja y *height* indica la dimensión vertical de la caja.

El proyectil utiliza un objeto de tipo *Animated3D* como elemento gráfico. Además, almacena un objeto de tipo *Model* en la propiedad *shadow*. Este objeto se crea con la función ***Model::makeShadow()*** y representa la sombra del proyectil. El objeto siempre se encuentra en la superficie de la plataforma más alta por debajo del proyectil.

Las funciones más importantes son:

- ***Projectile(float v0[3], Character *owner0, graphics::Texture *tex):*** Crea un nuevo proyectil lanzado por el personaje *owner0*. El proyecto usará *tex* como textura del elemento gráfico.
- ***interact(Platform *platform):*** Comprueba si el proyectil colisiona con una plataforma.
- ***getRect(float rect[6]):*** Escribe en el vector rect las dimensiones de la caja de colosión del proyectil en el formato [X,Y,Z,Anchura,Altura,Profundidad]

6.6.6 Estructuras del fichero attacks.h

El fichero *attacks.h* contiene las definiciones de los structs necesarios para los ataques de los personajes. A continuación se muestran los structs y sus funciones básicas. Los ataques se definen con más detalle en el apartado **4**.

battle3D::game::Impact

El struct *Impact* representa un impacto de un ataque. La temporización de los ataques depend del marco de animación actual del personaje. Se inicia un impacto cuando llegue el marco *startFrame*. El impacto termina cuando se llega al marco *startFrame+duration*. Un ataque se puede componer de varios impactos. Cuando termina un impacto, la propiedad *next* indica cuál es el impacto que debe seguir. Si es nulo, no se ejecutan más impactos.

Cada impacto tiene información sobre los efectos que tendrá el ataque cuando haga contacto con otro personaje. El daño se almacena en la propiedad *damage*. La propiedad *speedMultiplier* es factor en decidir a qué distancia se lanza el oponente. La velocidad del oponente se calcula multiplicando este factor por su porcentaje y dividiendo el resultado por 100.

Si el impacto es de un ataque fuerte, la propiedad *minPercent* indica cuánto daño debe tener el oponente para que se pueda lanzar. Un ataque “Smash” siempre lanzará el oponente, el valor *minVelocity* indica la velocidad mínima que tendrá el oponente después del impacto.

La propiedad *height* indica la altura del centro del impact. La propiedad type indica si el ataque es de tipo rotación o estocada. La información de cada tipo de impacto se almacena en las propiedades *rotation* y *thrust*. Solo se hace uso de una de estas dos propiedades.

battle3D::game::Rotation y battle3D::game::Thrust

El struct *Rotation* representa un ataque rotacional a un radio fijo del personaje, dado por *radius*. Este ataque comienza en el ángulo *startAngle* y termina en el ángulo *endAngle*. Los ángulos se miden en grados. Si está activado el valor *isHorizontal*, el ataque se ejecuta horizontalmente. En otro caso, el ataque se hace verticalmente alineado con la orientación del personaje. Si está activado el valor *isTangent*, el oponente se lanzará en dirección de la tangente del círculo. En caso contrario, el oponente se lanzará en dirección de la normal del círculo.

El struct *Thrust* representa un ataque de tipo estocada. La propiedad *startLength* indica la longitud al inicio del impacto y *endLength* indica la longitud al final del impacto. La estocada se hace en dirección de la orientación del personaje. La propiedad *angle* indica el ángulo en dirección vertical del ataque.

battle3D::game::Special y derivados

El struct Special es la representación abstracta de un ataque especial. El struct solo tiene una propiedad, *type*, que indica el tipo de ataque especial. Los ataques especiales tiene una gran variedad de efectos. Se han propuesto 6 ataques especiales para el proyecto, pero debido a la dificultad de dibujar animaciones de los personajes, solo se implementaron los tres ataques que utiliza el personaje de prueba.

ProjectileAttack

Este struct representa el lanzamiento de varios proyectiles. Los proyectiles se lanzan cuando la animación del personaje llegue al marco indicado por *start*. Se generan los proyectiles en el centro del personaje a la altura *height* con la textura *texture*. La propiedad *count* indica cuántos proyectiles generar. La magnitud de la velocidad inicial viene dada por la propiedad *speed* y su dirección se determina a partir de a orientación del personaje, el valor en *hAngles* y en *vAngles*.

Attack

Este struct se utiliza para dar el formato de un ataque normal al ataque especial. Tiene la funcionalidad adicional de poder desplazar al personaje mientras ejecute el ataque. El desplazamiento comienza cuando el marco del personaje llegue al valor indicado por *start*.

La propiedad *speed* indica la magnitud de su velocidad, mientras que las propiedades *h_angle* y *v_angle* indican los ángulos horizontal y vertical del desplazamiento con respecto a la orientación del personaje. Si el valor de *momentum* es verdadero, la velocidad se aplica solamente en el momento inicial. En caso contrario, el personaje mantiene una velocidad lineal y constante durante el número de marcos dado por *duration*.

CounterAttack

Cuando se inicia este ataque especial, el personaje se detiene esperando recibir un impacto de un oponente. Si recibe el impacto en un tiempo limitado, anula los efectos del impact y realiza un contraataque.

startFrame indican el marco donde comienza el periodo de contraataque. El valor de *duration* indica cuántos marcos tiene disponible el personaje para contraatacar. *endFrame* indica el último marco de la animación antes del ataque. Si un contraataque tiene éxito, se activa el ataque contenido en la propiedad *impact*. En caso contrario, la animación termina cuando llegue el marco *endFrame*. La propiedad *multiplier* multiplica el daño que recibe del oponente y lo envía de vuelta al oponente cuando contraataca.

6.6.7 battle3D::game::Character

La clase Character deriva de la clase Movable. Representa un personaje controlable en el escenario del juego. Esta es la clase más importante y compleja del programa.

Creación del personaje

El personaje se crea leyendo sus propiedades desde un fichero de datos. El fichero contiene información en texto plano del personaje, contenido en bloques. Los bloques tienen el formato:
`[Nombre parámetro parámetro parámetro ]`
Cada parámetro puede ser una cadena de texto, un número o incluso otro bloque.

El nombre de cada bloque indica cuales deben ser los parámetros que siguen. El formato cada tipo de bloque se describe brevemente en el fichero anexo *formato_personaje.txt*. Este documento está escrito en inglés.

La creación del personaje se realiza con la función ***load(const char *fileName, DataBase &imgDB, World *world)***. Esta función carga el fichero dado por el nombre *fileName* y lee el texto del fichero palabra por palabra. Las palabras se leen con la función ***Utils::readWord()***. Se considera una palabra cualquier conjunto de símbolos seguidos de un espacio, salto de línea u otro texto blanco. Los símbolos `'['` y `']'` se consideran palabras. La función omite comentarios, que pueden ser de bloque `/**/` o fin de línea `//`.

Para cada parámetro de cada bloque, se hacen llamadas a las funciones privadas ***parse*()*** para rellenar las estructuras del objeto *Character* con datos leídos del fichero. Si tiene que leer texturas del fichero, se leen desde la base de datos dada por *imgDB*. Las texturas se manejan con la propiedad *textureList* del objeto *world*.

Estructura del personaje

La caja de colisión del personaje se representa mediante una prisma rectangular. Esta caja se utiliza para detectar colisiones con plataformas, proyectiles y otros personajes. Su anchura en las dimensiones X y Z se da por el valor *width* y su altura por el valor *height*. El centro del personaje está en la parte central de la base del rectángulo.

Hay una variedad de propiedades que determinan las capacidades del personaje para realizar diferentes acciones. Algunas de las cuales son *maxSpeed*, *runSpeed*, *jumpSpeed*, *weight* y *grabDistance.* Los ataques normales del personaje se almacenan en el vector *attacks*. Los ataques especiales se almacenan en el vector *special*. Los ataques se explican con más detalle en apartados anteriores.

Hay propiedades para representar diferentes aspectos del estado del personaje. La propiedad *state* es un entero que almacena en sus bits algunas propiedades binarias del personaje. Por ejemplo, si está atacando, se tiene el escudo puesto o si ya ha realizado un salto en el aire. *percent* almacena el daño que ha recibido el personaje. *target* guarda un puntero al oponente a la que está apuntando y *grabbed* almacena un puntero a un oponente que ha agarrado. *currentImpact* y *currentSpecial* guardan punteros al ataque que está realizando el personaje en el momento.

Estructuras gráficas

El personaje se representa como una imagen plana animada. La imagen siempre está mirando hacia la cámara. Se utiliza un objeto de tipo MultiTextured para implementar este modelo. El personaje también contiene 5 objetos de tipo *Model*.

shadow es un modelo circular, plano y horizontal que representa la sombra del personaje. Se genera con la función ***Model::makeShadow()***. La sombra mantiene la posición X y Z del personaje y la posición Y de la superficie de la plataforma más alta por debajo del personaje.

ring es un modelo en formal de un anillo plano horizontal con una flecha. El anillo tiene un color que representa el número del jugador que lo controla. Este anillo se mantiene siempre en la base del modelo principal del personaje y apunta en la dirección de la orientación del personaje o hacia un oponente si el jugador está apuntando a ese oponente. Se genera este modelo mediante la función ***Model::makeRing()***.

shield es un modelo esférico semitransparente que rodea al personaje. El modelo se hace visible cuando el personaje se pone el escudo. Se genera el modelo con la función ***Model::makeSphere()***.

Las propiedades *thrustModel* y *slashModel* son modelos que representan el movimiento de los impactos de tipo estocada y rotación. cuando el jugador está atacando, el puntero *attackModel* apuntará al modelo que representa el impacto actual.

La función frame()

Esta función realiza todas las acciones del personaje en cada instante del juego. Se comienza leyendo información del mando.

Dependiendo de las entradas del mando y el estado actual del personaje, puede moverse, saltar, comenzar un ataque normal o especial, ponerse el escudo, apuntar a un enemigo o agarrar a un enemigo.

Si el personaje comienza un ataque normal o especial, se utilizan las funciones ***selectAttack()*** y ***selectSpecial()*** para elegir uno de los ataques normales o especiales de las listas *attack* y *special* respectivamente. El ataque se elige según la dirección del mando en comparación con el objetivo del personaje, según la rapidez con la que se pulsó el botón de dirección y según si el personaje está en el suelo o en el aire.

Algunas acciones que realiza el personaje se comienzan con funciones específicas: ***startImpact()*** para comenzar un impacto de un ataque, ***startSpecialAttack()*** para comenzar un ataque especial y ***startGrab()*** para comenzar un movimiento de agarre.

Si un personaje está realizando alguna acción y no puede recibir comandos del mando, la función llama a una de las funciones encargadas de actualizar la acción actual: ***handleImpact()*** para ataques normales, ***handleSpecial()*** para ataques especiales, ***dodge()*** para esquivar, ***handleGrab()*** para agarrar a oponentes y ***handleThrow()*** para el lanzamiento de oponentes.

Después, se actualiza la posición del personaje si está sobre una plataforma móvil o se añade el efecto de la gravedad si no está sobre una plataforma. Finalmente se actualizan contadores y otros valores de estado.

La función draw()

Esta función actualiza la posición del personaje según su velocidad, luego actualiza la posición de los modelos pertenecientes al personaje: *shadow*, *ring*, *shield* y *attackModel*. La actualización de *attackModel* se realiza con las funciones ***updateSlashModel()***, ***updateThrustModel()*** y ***updateGrabModel()*** según si el personaje está atacando o agarrando y qué tipo de ataque está realizando. Finalmente, se actualiza la textura a mostrar en el modelo según el marco de la animación, la orientación del personaje y la posición de la cámara.

Interacción con otros objetos

La clase Character tiene tres funciones para manejar la interacción entre un personaje y otros objetos:

- ***interact(Platform *p):*** Esta función detecta la colisión del personaje con una plataforma y actualiza el estado y la posición del personaje.
- ***Character::interact(World *world, Character *c1, Projectile *p2):*** Esta función detecta la colisión de un personaje con un proyectil. Si el proyectil colisiona, el personaje recibirá daño y se cancelará cualquier acción que esté realizando el personaje con la función ***cancelAttack()***. Si el personaje tiene un escudo puesto, no recibirá daño, pero el escudo perderá energía.
- ***Character::interact(World *world, Character *c1, Character *c2):*** Es una función estática que detecta y resuelve colisiones de varios tipos entre dos personajes.

La naturaleza de la colisión entre dos personajes puede ser de tres tipos:

- **Cuerpo contra cuerpo:** Si dos personajes colisionan, se moverán en direcciones opuestas a velocidades constantes hasta que la colisión deje de ocurrir.
- **Ataque contra ataque:** Si dos ataques colisionan, los ataques de ambos personajes se cancelan con la función ***cancelAttack()***. Si uno de los ataques es de tipo agarre, se ignora esta colisión y se procede a comprobar la colisión de ataque contra cuerpo.
- **Ataque contra cuerpo:** Si el ataque de un personaje colisiona con el cuerpo de otro, se resuelve la colisión con la función estática ***Character::attack()***. Si el ataque es de tipo agarre, se resuelve con la función estática ***Character::grab()***.

Para la colisión de ataque contra cuerpo existen dos funciones:

- ***Character::attack(World *world, Character *c1, Character *c2):*** Esta función ejecuta el ataque del *c1* sobre *c2*. Si *c2* tiene el escudo puesto, se devuelve falso. Si el escudo pierde toda su energía, se rompe con la función ***breakShield()***. Si *c2* está esperando para contraatacar, se cancela el ataque de *c1* y se activa el contraataque de *c2*. También devuelve falso. Si el ataque de *c1* tiene éxito, se aplica el porcentaje de daño a *c2* y se calcula su nueva velocidad según las características del ataque. Finalmente se devuelve verdadero.
- ***Character::grab(World *world, Character *c1, Character *c2):*** Esta función cancela todas las acciones de *c2* y lo inmobiliza. *c1* lo tiene agarrado.

6.6.8 battle3D::game::World

La clase *World* deriva de la clase *menu::GameScreen*. De este modo, puede representar imágenes en la pantalla delante del juego. Esta clase representa el escenario en donde se encuentran los personajes, las plataformas, los proyectiles y los otros objetos del juego. Su función es ejecutar todo el funcionamiento del juego, haciendo llamadas a las funciones de cada objeto contenido en el escenario.

Se compone principalmente de listas de objetos que lo componen. *platforms* contiene una lista de plataformas, de tipo *Platform*. *decorations* contiene una lista de objetos decorativos, de tipo Decoration. *projectiles* contiene la lista de proyectiles, de tipo *Projectile*. *characters* contiene una lista de personajes, de tipo *Character*.

La clase *World* también tiene varias propiedades para controlar la cámara. La cámara se centra en el punto promedio de las posiciones de todos los personajes y se sitúa a una distancia suficiente para que se vean todos los personajes. La distancia se da en la propiedad *camDist* y cambia según cómo se mueven los personajes. La cámara tiene un ángulo vertical fijo - dado por *camVAngle* - y el ángulo horizontal - dado por *camHAngle* - se mueve para ponerse perpendicular a la línea que une los dos personajes más separados entre sí.

Cuando se crea el escenario, se cargan los datos desde un fichero de datos mediante la función ***loadFile()***. El fichero tiene una estructura similar al fichero de datos de un personaje y la función tiene una funcionalidad muy similar a la función ***load()*** de la clase *Character,* incluyendo llamadas a funciones ***parse*()*** para cada tipo de bloque. El fichero contiene el nombre de la imagen de fondo, los límites del escenario, una lista de las posiciones iniciales de cada personaje en el escenario, una lista de plataformas y una lista de objetos de decorado.

La clase también contiene algunas variables para el manejo de gráficos. *viewport* mantiene un puntero al objeto *Viewport* que representa la pantalla. La propiedad *textureList* contiene las texturas de todos los objetos que existen en el escenario. La propiedad *background* contiene el modelo de un cilindro gigante, creado con la función ***Model::makeCylinder()***, que rodea el escenario. El cilindro representa la imagen de fondo.

El escenario guarda datos sobre el progreso. En el vector *lives* se almacena el número de vidas de cada personaje. El vector *victories* almacena el número de oponentes que ha derrotado cada personaje. El vector *losses* indica cuántas veces ha perdido un personaje. El juego termina cuando solo queda un personaje en el escenario. El número de personajes se almacena en la propiedad *charactersRemaining*.

Las funciones más importantes son:

- ***begin():*** Esta función añade los elementos gráficos de todos sus objetos a la memoria gráfica del objeto *Viewport*. Pone a los personajes en sus posiciones iniciales a inicializa algunos valores. Finalmente pone la variable ready a verdadero para indicar que está listo para ejecutar la función ***frame()*** repetidamente.
- ***end():*** Elimina de la memoria gráfica los modelos de todos los objetos del escenario y borra la lista de texturas.

- ***frame():*** Esta función realiza todo el trabajo. Primero actualiza los objetos del escenario ejecutando la función ***frame()*** de las plataformas, los personajes y los proyectiles. Si un personaje no existe o se ha quedado sin vidas, no se ejecuta esta función. Después computa la interacción de los personajes con otros personajes, proyectiles y plataformas. Comprueba si el personaje se ha salido del escenario, en cuyo caso, llama a la función ***defeat()*** para ese personaje. Luego computa la interacción de proyectiles con plataformas y con otros proyectiles. Actualiza los gráficos llamando a la función ***draw()*** de cada objeto, moviendo las sombras de los personajes y proyectiles. Actualiza las imágenes y el texto en la pantalla con la nueva información. Finalmente actualiza la posición de la cámara.
- ***defeat(int i):*** Quita una vida al personaje del jugador *i*, lo pone en su posición inicial y restablece sus propiedades con la función ***Character::reset()***. Si el personaje se ha quedado sin vidas, se marcan los modelos del personaje como invisibles.

7. Comparativa de tecnologías gráficas

Esta sección comparará la implementación de gráficos en Direct3D con la implementación de gráficos en OpenGL. No se entrará en detalle sobre el proceso de renderización de gráficos sino que se compararán las estructuras de datos y las funciones de cada librería y el resultado final de la renderización.

7.1 Representando gráficos en DirectX 9.0

DirectX está diseñado para ser utilizado por el lenguaje C. Por lo tanto, se utilizan structs y funciones globales en vez de clases con propiedades y métodos. A continuación se explicará la implementación de gráficos en Direct3D. Se utilizará la palabra "objeto" para referirse a la instancia de un struct.

7.1.1 Inicialización del dispositivo

Un objeto *LPDIRECT3D9* es un puntero a la interfaz de Direct3D. Un objeto *LPDIRECT3DDEVICE9* es un puntero a un dispositivo específico de Direct3D.

Antes de comenzar a usar Direct3D es necesario crear una instancia de *LPDIRECT3D9* mediante la función ***Direct3DCreate9()***

```
LPDIRECT3D9 Direct3DInterface = Direct3DCreate9(D3D_SDK_VERSION));
```

Se crea un objeto de tipo *D3DPRESENT_PARAMETERS*, que almacena la configuración del dispositivo a utilizar. Se define la ventana, el formato del buffer, el tamaño de la pantalla y otros valores. Luego se llama a la función ***CreateDevice()*** para crear el dispositivo:

```
D3DPRESENT_PARAMETERS direct3DParams;
LPDIRECT3DDEVICE9 Direct3DDevice;
hr = Direct3DInterface->CreateDevice(D3DADAPTER_DEFAULT,
      D3DEVTYPE_HAL, window, D3DCREATE_SOFTWARE_VERTEXPROCESSING,
      &direct3DParams, &Direct3DDevice)
```

7.1.2 Cargando texturas

Las texturas se almacenan en un objeto de tipo *IDirect3DTexture9*. Para poder almacenar una imagen en la textura, se hace uso de la clase *Bitmap* de *GDIPlus*.

Primero se crea una textura con la función ***CreateTexture()***. Después, se lee el fichero PNG en un objeto de tipo *Bitmap*. Luego se cargan los datos del Bitmap en un objeto de tipo *BitmapData*. Finalmente, se copian los bytes de este objeto a la textura.

```
IDirect3DTexture9 *dst;
Gdiplus::Bitmap src;
Gdiplus::Rect sroi;
hr=Direct3DDevice->CreateTexture(sroi.Width, sroi.Height, 1,
      D3D_USAGE_DYNAMIC, D3DFMT_A8R8G8B8, D3DPOOL_DEFAULT, &dst, NULL);
// [ ... Leer el fichero PNG en un objeto Bitmap ... ]

Gdiplus::BitmapData data;
Gdiplus::Status gst=src->LockBits(&sroi, Gdiplus::ImageLockModeRead,
      PixelFormat32bppARGB, &data);
D3DLOCKED_RECT droi = {0};
dst->LockRect(0, &droi, NULL, D3DLOCK_NOSYS_LOCK|D3DLOCK_DISCARD);
//[ ... Copiar los bytes ... ]
dst->UnlockRect(0);
src->UnlockBits(&data);
```

7.1.3 Renderizando modelos

DirectX utiliza los structs *IDirect3DVertexBuffer9* y *IDirect3DIndexBuffer9* para almacenar información de los vértices en memoria gráfica. Para renderizar los modelos, se tiene que definir una estructura que representa los vértices de los modelos. Esta fue la estructura que se definió en el proyecto:

```
struct Vertex3D{
      float x,y,z;
      D3DVECTOR normal;
      DWORD color;
      float tu, tv;
};
```

Se almacenan los vértices de cada modelo en memoria gráfica mediante la función ***CreateVertexBuffer()***. Se almacenan los índices de los vértices mediante la función ***CreateIndexBuffer()***. Los índices indican cuáles de los vértices forman las caras del modelo.

Antes de renderizar los modelos, se borra la pantalla y el Z-Buffer con la función ***Clear()*** y se comienza la representación del escenario con ***BeginScene()***. Se establece el valor FVF mediante la función ***SetFVF()***. El FVF indica cuáles de los posibles datos del struct se usarán para renderizar el modelo. En nuestro caso usaremos un punto [X,Y,Z], un colo difuso, la normal del vértice y el valor de una textura. Se establecen los valores de la matriz de vista y proyección mediante la función ***SetTransform()***.

Se establece la textura, el material, y la transformación de cada modelo con las funciones ***SetTexture()***, ***SetMaterial()*** y ***SetTransform()***. Se eligen los vértices del modelo a renderizar mediante la función ***SetStreamSource()*** y los índices mediante ***SetIndices()***. Finalmente, se dibuja el modelo con ***DrawIndexedPrimitive()***.

Para finalizar la renderización del escenario, se llama a la función ***EndScene()***. Se llama a la función ***Present()*** para mostrar el contenido de la imagen resultante en la pantalla.

```
Direct3DDevice->Clear(0,NULL,D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
      bgColor,1.0f,0);
Direct3DDevice->BeginScene();
Direct3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE |
            D3DFVF_NORMAL |  D3DFVF_TEX1 );
D3DXMATRIX matView, matProjection;
Direct3DDevice->SetTransform(D3DTS_VIEW, matView);
Direct3DDevice->SetTransform(D3DTS_PROJECTION, matProjection);

// Creando cada modelo:
IDirect3DVertexBuffer *VertexBuffer;
IDirect3DIndexBuffer *IndexBuffer;
Direct3DDevice->CreateVertexBuffer(vertLen*sizeof(Vertex3D), 0, FVF3D,
      D3DPOOL_MANAGED, &VertexBuffer, NULL);
Direct3DDevice->CreateIndexBuffer(indLen*sizeOf(short), 0, D3DFMT_INDEX16,
      D3DPOOL_MANAGED, &IndexBuffer, NULL);
// [... Copiar los vértices a índices en los buffers correspondientes ...]

//Renderizando cada modelo
D3DXMATRIX  worldMatrix;
D3DMATERIAL9 material;
IDirect3DTexture9 *texture;
material.color = D3DXCOLOR(1.0f, 1.0f, 1.0f, 1.0f);
Direct3DDevice->SetMaterial(material);
Direct3DDevice->SetTexture(0, texture);
Direct3DDevice->SetTransform(D3DTS_WORLD, &worldMatrix);
Direct3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
      vertLen, 0, (int)(indLen/3));
```

```
//Finalizando la escena:
Direct3DDevice->EndScene();
Direct3DDevice->Present(NULL,NULL,NULL,NULL)
```

7.2 Representando gráficos en OpenGL 3

Al igual que DirectX, OpenGL está diseñado para ser compatible con el lenguaje C. Por lo tanto, en lugar de clases y objeto se utilizan structs. De nuevo, se utilizará la palabra "objeto" para referirse a una instancia de un struct.

7.2.1 Inicialización del dispositivo

Una variable de tipo *HDC* tiene una referencia al contexto de la ventana de Windows. Se consigue este valor con la función ***GetDC()***, pasando como parámetro el valor *HWND* de la ventana en donde se renderizará el escenario.

Antes de inicializar los valores de OpenGL, es necesario configurar el contexto de la ventana de Windows. El struct *PIXELFORMATDESCRIPTOR* es un descriptor de formato de píxeles. Almacena información sobre el uso del dispositivo. Algunos datos que se dan son el número de frame buffers, el método de renderización y el formato de los píxeles. Se pasa el descriptor a la función ***ChoosePixelFormat()***, que devuelve un valor entero indicando el formato de píxeles más cercano a los parámetros definidos. Se aplica esta configuración pasando el valor entero y el descriptor a la función ***SetPixelFormat()***.

Una vez configurado el contexto de Windows, se crea un contexto de OpenGL con la función ***wglCreateContext()***, en el cual se dibujará el escenario. El contexto se almacena en una variable de tipo *HGLRC*. La función ***wglMakeCurrent()*** establece el contexto creado como el contexto actual. Una vez creado el contexto, está listo para usarse.

Se puede cargar una librería llamada GLEW, que ayuda a determinar cuáles de las funcionalidades de OpenGL están disponibles en el sistema operativo. Se carga la librería con la función ***glewInit()***. La librería contiene una alternativa a ***wglCreateContext()***, ***wglCreateContextAttribsARB()***, que genera un contexto teniendo en cuenta la versión de OpenGL que se está utilizando.

```
HWND hwnd;
HDC hdc = GetDC(hwnd);
PIXELFORMATDESCRIPTOR pfd;
//[... Definir valores de pfd ... ]
int nPixelFormat = ChoosePixelFormat(hdc, &pfd);
int bResult = SetPixelFormat(hdc, nPixelFormat, &pfd);

GLenum error = glewInit();
int attributes[] = {
WGL_CONTEXT_MAJOR_VERSION_ARB, 3,
WGL_CONTEXT_MINOR_VERSION_ARB, 0,
WGL_CONTEXT_FLAGS_ARB, WGL_CONTEXT_FORWARD_COMPATIBLE_BIT_ARB,
      0};
HGLRC context;
if(error == GLEW_OK && wgleIsSupported("WGL_ARB_create_context")==1){
      context = wglCreateContextAttribsARB(hdc,NULL,attributes);
} else {
      context=wglCreateContext(hdc);
}
wglMakeCurrent(hdc, context);
```

7.2.2 Cargando shaders programables

Los shaders programables son programas cortos en un lenguaje similar a C, llamado GLSL. Se almacenan en ficheros de texto plano. Para hacer uso de ellos, se deben leer, compilar y guardar en la memoria gráfica.

Se guarda el programa en memoria gráfica pasando el texto como parámetro a la función ***glShaderSource()***. Si la función tiene éxito, se escribirá en el primer parámetro de la función un valor entero que identifica el programa.

Se compila el programa con la función ***glCompileShader()***, pasando el identificador del programa como parámetro. Después de compilar el programa, se llama a la función ***glGetShaderiv()***. Esta función escribe en el primer parámetro un valor entero identificando el programa compilado y en el tercer parámetro el estado de compilación.

Se crea un programa completo con la función ***glCreateProgram()***, que devuelve un identificador del programa. Se llama a la función ***glAttachShader()*** para cada shader programable para asignar los shaders al programa. Finalmente, se llama a ***glLinkProgram()*** para enlazar el programa y ***glValidateProgram()*** para verificar que el programa funciona correctamente.

Se pueden asignar identificadores numéricos a variables del programa mediante la función ***glBindAttribLocation()***. Se utlizarán estos identificadores más tarde para pasar valores desde el el programa de la CPU al programa en memoria gráfica.

```
const char *shader_vertices;
const char *shader_fragmentos;
int shader_vp, shader_fp, shader_id;
GLint compiled
glShaderSource(shader_vp, 1, &shader_vertices, 0);
glShaderSource(shader_fp, 1, &shader_fragmentos, 0);

glCompileShader(shader_vp);
glCompileShader(shader_fp);

glGetShaderiv(shader_vp, GL_COMPILE_STATUS, &compiled);
glGetShaderiv(shader_fp, GL_COMPILE_STATUS, &compiled);

shader_id = glCreateProgram();
glAttachShader(shader_id, shader_fp);
glAttachShader(shader_id, shader_vp);

glBindAttribLocation(shader_id, 0, “in_Position”);
glBindAttribLocation(shader_id, 1, “in_Color”);
glBindAttribLocation(shader_id, 2, “in_UV”);

glLinkProgram(shader_id);
glValidateProgram(shader_id);
```

Para utilizar el programa durante el proceso de renderización, se llama a la función ***glUseProgram()***. Si se pasa como parámetro el valor 0, se deja de utilizar el programa.

```
glUseProgram(shader_id);
```

Para separar a los shaders del programa, se llama a la función ***glDetachShader()***. Luego se llama a ***glDeleteShader()*** para eliminarlos de memoria y finalemente ***glDeleteProgram()*** para borrar al programa de la memoria.

```
glDetachShader(shader_id, shader_fp);
glDetachShader(shader_id, shader_vp);

glDeleteShader(shader_fp);
glDeleteShader(shader_vp);
glDeleteProgram(shader_id);
```

7.2.3 Cargando texturas

Para cargar una textura en memoria gráfica, se llama primero a la función ***glGenTextures()*** para generar un vector de identificadores de texturas. En el primer parámetro se pasa el número de texturas a generar y el segundo parámetro es un puntero a un vector de enteros, en donde se escriben los identificadores.

Después se llama a la función ***glBindTexture()***, pasando en el primer parámetro el tipo de textura y en el segundo parámetro el identificador de la textura. Esta función establece la textura como la textura activa, de manera que todas las operaciones relacionadas con texturas afectarán a esta textura en concreto. Una vez llamada a ***glBindTexture()***, la textura se mantiene en memoria hasta que se lame a ***glDeleteTextures()*** pasando el identificador de la textura como parámetro.

Para configurar los parámetros de la textura, se llama a la función ***glTexParameteri()***. Los parámetros *GL_TEXTURE_MAG_FILTER* y *GL_TEXTURE_MIN_FILTER* indican el método de elección del color según la posición de cada fragmento en la textura. En este programa se ha utilizado *GL_NEAREST* para indicar que se selecciona el color del píxel más cercano.

Para almacena la imagen que representa la textura en memoria, se llama a la función ***glTexImage2D()***. En esta función se pasan como parámetros el formato de la imagen, sus dimensiones y los datos que representan la imagen.

Para hacer uso de una textura, se llama a la función ***glBindTexture()*** con el identificado de la textura a usar y se llama a ***glActiveTexture()*** para indicar que es la textura activa. Pasando el valor 0 como segundo parámetro de ***glBindTexture()*** indica que no hay ninguna textura activa del tipo dado por el primer parámetro.

```
unsigned int textureID;
unsigned char *imgData;
unsigned int width, height;
glGenTextures(1,&textureID);
glBindTexture(GL_TEXTURE_2D,textureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8,
      width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
      imgData);
glBindTexture(GL_TEXTURE_2D,0);
```

7.2.4 Renderizando modelos

Primero se deben almacenar los modelos a renderizar en la memoria gráfica. Se utilizan VBOs y VAOs. Un VBO es un buffer que almacena un vector de datos en memoria. Este vector puede contiene una serie de datos sobre los vértices. Por ejemplo, la posición, la normal o el color. Un VAO es un identificador que reúne varios VBOs para formar una descripción completa del modelo.

Se genera un VAO con la función ***glGenVertexArrays()***. Esta función escribe en el segundo parámetro los identificadores de los VAOs generados. El número de VAOs se da en el primer parámetro. Se llama a la función glBindVertexArray() para utilizar el VAO.

Se genera un VBO con la función ***glGenBuffers()***, pasando como primer parámetro el número de VBOs y como segundo parámetro el vector donde se almacenan los identificadores de los VBOs. Se llama a la función ***glBindBuffer()*** para utilizar el VBO dado por el primer parámetro. Para escribir datos en el VBO, se llama a ***glBufferData()***, pasando el vector de datos a escribir y el formato de los datos. Para asignar un VBO al parámetro de un shader, se llama a la función ***glVertexAttribPointer()***. Se pasa el identificador del parámetro e información sobre los datos del VBO. Finalmente, se llama a ***glEnableVertexAttribArray()*** pasando el identificador del parámetro para habilitar el uso de ese parámetro.

```
unsigned int vaoID, vboXYZ, vboColor;
float *puntos, *colores;
int numPuntos;
glGenVertexArray(1, &vaoID);
glBindVertexArray(vaoID);

glGenBuffers(1,&vboXYZ);
glBindBuffer(GL_ARRAY_BUFFER, vboXYZ);
glBufferData(GL_ARRAY_BUFFER, 3*numPuntos*sizeof(GLfloat),
      puntos, GL_STATIC_DRAW);
glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0,0);
elEnableVertexAttribArray(0);

glGenBuffers(1,&vboColor);
glBindBuffer(GL_ARRAY_BUFFER, vboColor);
glBufferData(GL_ARRAY_BUFFER, 3*numPuntos*sizeof(GLfloat),
      colores, GL_STATIC_DRAW);
glVertexAttribPointer((GLuint)1, 3, GL_FLOAT, GL_FALSE, 0,0);
glEnableVertexAttribArray(1);

glBindVertexArray(0);
```

Antes de comenzar a renderizar un escenario, se llama a la función ***glViewport()***. Esta función utiliza los cuatro parámetros para establecer la posición y las dimensiones del frame buffer. Después se llama a la función ***glClear()*** para borrar la información de los colores y el Z-Buffer.

Como se describió en la sección “cargando shaders”, se pueden definir identificadores fijos para algunos parámetros de los shaders. Se pueden pasar parámetros no predefinidos llamando a la función ***glGetUniformLocation()***. Pasando como parámetro el identificador del shader y el nombre del parámetro, esta función devuelve un identificador de ese parámetro. Existe una variedad de funciones ***glUniform*()*** para escribir valores en los parámetros de los shaders. En este programa se utilizaron tres: La matriz de proyección, la matriz de vista y la opacidad del modelo a renderizar.

Para cada modelo, se envía también la matriz de transformación y el identificador de la textura asociada. Se llama a ***glBindVertexArray()*** para usar el VAO del modelo actual, luego se llama a ***glDrawArrays()*** para dibujar el modelo.

Finalmente, después de renderizar todos los modelos, se llama a la función ***SwapBuffers()*** para actualizar la imagen en la pantalla.

```
glViewport(0,0,windowWidth,windowHeight);
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCL_BUFFER_BIT);

int shader_id;
glm::mat4 projMat, viewMat;
int projMatrixID, viewMatrixID, alphaID;

projMatrixID = glGetUniformLocation(shader_id, “projectionMatrix”);
viewMatrixID = glGetUniformLocation(shader_id, “viewMatrix”);

glUniformMatrix4fv(projMatrixID, 1, GL_FALSE, &projMat[0][0]);
glUniformMatrix4fv(viewMatrixID, 1, GL_FALSE, &viewMat[0][0]);

// Para cada modelo
float alpha;
int texture_id, numVertices;
glm::mat4 modelMat;
int modelMatrixID, textureID, vaoID;

alphaID = glGetUniformLocation(shader_id, “alpha”);
modelMatrixID = glGetUniformLocation(shader_id, “modelMatrix”);
textureID = glGetUniformLocation(shader_id, “textureSampler”);

glUniform1fv(alphaID, 1, &alpha);
glUniform1i(textureID,0);
glUniformMatrix4fv(modelMatID, 1, GL_FALSE, &modelMat[0][0])

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_3D, texture_id);

glBindVertexArray(vaoID);
glDrawArrays(GL_TRIANGLES, 0, numVertices);
glBindVertexArray(0);

//Después de renderizar todos los modelos
SwapBuffers(hdc);
```

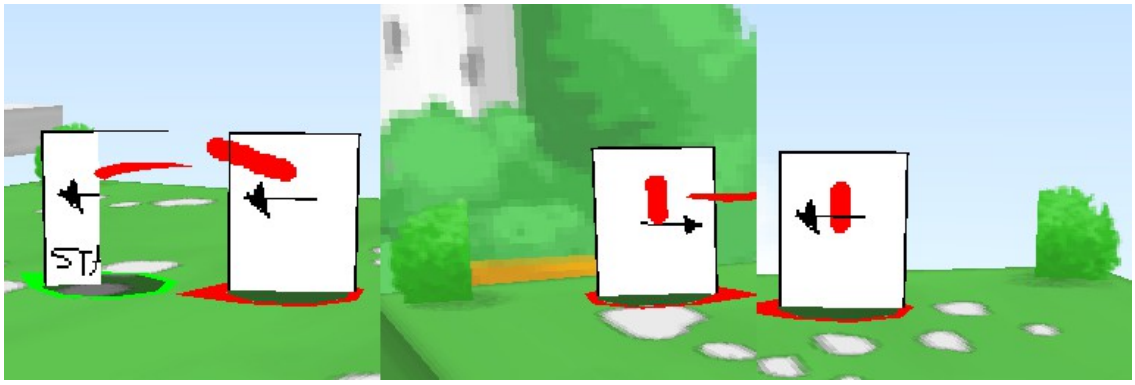
7.3 Conclusiones

Las librerías OpenGL y Direct3D tienen funcionalidades muy similares. Los VAOs y VBOs de OpenGL tienen paralelo con las funciones CreateVertexArray() y CreateIndexArray() en Direct3D. La función ***glDrawArrays()*** es similar a la función ***DrawIndexedPrimitive()***. La función ***SwapBuffers()***, aunque es una función de Windows y no OpenGL, realiza la misma función que la función ***Present()*** de Direct3D.

Una de las mayores diferencias entre Direct3D y OpenGL es que Direct3D hace mucho uso de estructuras de datos propietarios de Microsoft, mientras que la los datos de OpenGL se almacenan en vectores de distintos tipos básicos, como *unsigned int*, *float* y *double*. Esto simplifica el proceso de analizar el funcionamiento de OpenGL.

Otra gran diferencia es el uso de shaders programables. Direct3D permite al programador dibujar gráficos en la pantalla sin necesidad de programar los shaders. En cambio, OpenGL 3 requiere que el programador escriba el shader de vértices y el shader de fragmentos.

Esta diferencia fue crucial para el éxito del proyecto. Durante la programación del prototipo de DirectX, ocurrió un problema en la representación de los gráficos. Muchos de los objetos del escenario utilizan imágenes con píxeles transparentes como textura. El Z-Buffer de DirectX estaba omitiendo la renderización de objetos que aparecían detrás de estos píxeles, resultando en la desaparición de porciones de los objetos del escenario.



Objetos del escenario que aparecen detrás de píxeles transparentes no se renderizan. Esto ocurre incluso con personajes.

Este problema se resolvió utilizando el comando ***discard*** del shader de fragmentos sobre píxeles que tuvieran una opacidad menor del 80%. Este comando omite el procesamiento del fragmento, evitando poner una marca en el Z-Buffer. Aunque la librería DirectX *sí* dispone de compatibilidad con shaders programables, no da a los programadores una necesidad de aprender a usarlos.

Debido a esto, **la librería gráfica elegida fue OpenGL**. Debido a su uso de estructuras de datos más sencillos y comunes, es más fácil de entender y usar. Además, incentiva el aprendizaje de shaders programables, que son muy útiles para tener mejor control sobre la renderización de los gráficos.

A pesar de esta elección, es importante destacar que no se encontraron diferencias importantes en el funcionamiento de las librerías sino simplemente en la interfaz con el programador. Los gráficos resultantes de la implementación con cada librería con prácticamente idénticos.

La implementación del módulo *battle3D::graphics* tardó bastante tiempo en realizar y su utilidad es limitada. Se ha llegado a una conclusión más importante: La elección de la librería gráfica de bajo nivel no es muy importante. Para simplificar la implementación de un proyecto, **es preferible utilizar una librería gráfica de más alto nivel**.

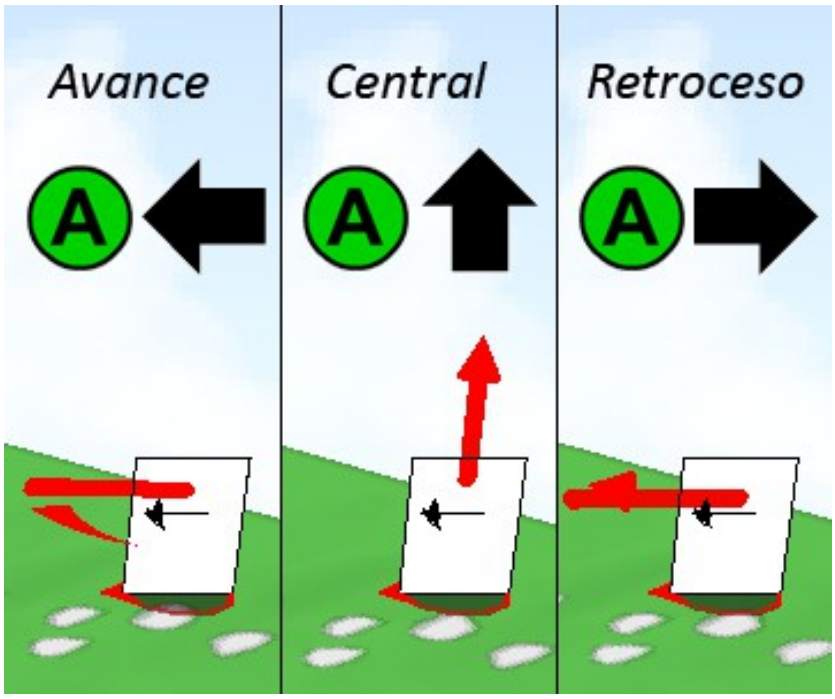
8. Conclusiones y mejoras futuras

8.1 Resultados del juego

El juego se ha implementado con éxito. Se implementaron todas las pantallas del menú exceptuando la de configuración de botones. Pueden jugar hasta cuatro personas simultáneamente. Se ha implementado un personaje y un escenario. Todos los elementos del escenario funcionan y se han implementado todas las acciones del personaje.

Aunque se ha implementado un juego completamente funcional, el juego aún está incompleto. Solo existe un personaje y un escenario. Todos los gráficos, exceptuando los del escenario, son provisionales y necesitan ser cambiados para una versión final del juego. Algunos de los ataques especiales no se han implementado, pues el personaje no hace uso de ellos. No se han hecho pruebas extensivas sobre el juego y aún falta calibrar algunos valores, como la velocidad de los personajes, la altura del salto y la fuerza de los ataques.

El **tiempo total de implementación** fue de aproximadamente **300 horas**, aunque este resultado no tiene en cuenta la implementación del prototipo de DirectX. que se implementó antes de comenzar formalmente el proyecto.



Se observa los diferentes tipos de ataque Smash según la dirección del stick analógico en comparación con la dirección del oponente, que se encuentra a la izquierda.

El jugador 2 (verde) ataca al jugador 1 (rojo), que se pone el escudo y evita el daño

El jugador 1 ha lanzado un proyectil triple. El proyectil central ha dañado al jugador 2

El juego implementado es completamente funcional, pero aún queda mucho para que sea un producto viable en el mercado. A continuación se proponen posibles arreglos y cambios que se podrían realizar al programa.

8.2 Errores y fallos

Aunque el programa carece de errores fatales conocidos, existen algunos problemas que hacen menos agradable el programa. En primer lugar, se necesitan realizar pruebas extensivas para buscar posibles fallos en el comportamiento de los personajes.

Los atributos de ataque del personaje aún son provisionales. Cuando ataca a oponentes con poco daño, no salen volando a mucha velocidad, pero con un nivel de daño medio, se mueve demasiado rápido. Es posible que una función lineal no sea el más adecuado para modelar el efecto de los ataques.

8.3 Características incompletas

Existen algunas características que no se han terminado de implementar. En primer lugar, la mayoría de las texturas son provisionales y necesitan ser susituidos.

En el menú falta una pantalla en donde los jugadores puedan configurar el mapeado de las entradas reales sobre el mando ficticio. Por ejemplo, las teclas del teclado, o los botones de mandos reales. Además de cambiar los gráficos, también se necesita cambiar la posición de algunos de los elementos existentes en las pantallas del menú.

Aún falta por implementar tres de los 6 ataques especiales que se definieron en el documento de diseño. El primero es un ataque de tipo explosión, donde el personaje genera una esfera dentro de la cual todos los oponentes reciben mucho daño y salen volando. El segundo es el ataque de tipo escudo que refleja proyectiles pero no defiende contra ataques físicos. El tercer ataque es una fuente de proyectiles, donde el personaje se queda en un sitio y lanza una serie continua de proyectiles que viajan una distancia corta.

8.4 Posibles nuevas características

El juego en su estado actual aún es demasiado sencillo para poder competir en el mercado de los video juegos. A continuación se muestra una lista de posibles características que se podrían añadir para mejorarlo:

- Más personajes y escenarios.
- Añadir elementos de decorado adicionales para representar efectos de ataques y movimientos.
- Opción de desbloquear personajes y escenarios completando objetivos.
- Inteligencia artificial pare que un jugador pueda jugar solo.
- Sonido y música.
- Juego en red.

9. Anexo: Manual de usuario

9.1 Introducción

Battle 3D es un juego de lucha y plataformas donde podrán jugar entre 2 y 4 personas de manera competitiva. El objetivo del juego es lanzar a todos los oponentes fuera del escenario y ser el único personaje que quede en el escenario.

9.2 Requisitos del juego

Para poder jugar, se necesitará un ordenador con Windows XP o superior, compatible con DirectX 9.0 y OpenGL 3. También se necesitará al menos un teclado y un mando de juego compatible con DirectInput, o bien dos mandos.

9.3 Controles

Si tienes un mando, los botones estarán en estas posiciones.

Si utilizas un teclado, las teclas están configuradas de la siguiente manera:

Botón	**A**	**B**	**X**	**Y**	**L**	**R**	**START**	**Arriba**	**Abajo**	**Izquierda**	**Derecha**
Tecla	J	K	M	I	L	Espacio	Intro	W	S	A	D

Los controles se muestran en la siguiente tabla:

Botón	**En el menú**	**En el juego**
A	Seleccionar / Avanzar	Ataque normal / Agarrar
B	Volver atrás	Ataque especial
X		Dejar de apuntar a oponentes
Y		Saltar / Saltar en el aire
L		Apuntar a oponentes
R		Poner el escudo
START	Seleccionar / Avanzar	Pausar el juego
Direcciones	Mover el cursor	Mover el personaje

9.4 Navegando el menú

Para navegar el menú, normalmente el jugador moverá su cursor sobre los botones de la pantalla, pulsando **A** para accionar el botón y **B** para volver atrás.

9.4.1 Pantalla de inicio

En la pantalla de inicio aparecen tres botones

- **Start:** Abre la pantalla de selección de personajes.
- **Options:** Abre la pantalla de opciones.
- **Quit:** Cierra el juego.

9.4.2 Opciones

La pantalla de opciones permite configurar los dispositivos de entrada y cambiar entre modo de pantalla normal y pantalla completa.

En la parte de arriba hay una fila de cuatro símbolos. Cada uno representa el dispositivo de entrada de un jugador. Un teclado indica que el jugador está usando un teclado. Un mando negro indica que el jugador está usando un mando compatible con DirectInput. Un mando con una X verde indica que el jugador está un mando de XBox 360 o uno similar. Una X roja indica que no existe un dispositivo de entrada para el jugador.

El jugador 1 podrá intercambiar el dispositivo de entrada de un jugador con el dispositivo de otro jugador. Para hacerlo, mueve el cursor sobre un mando, pulsa el botón A para seleccionar el mando, mueve el cursor sobre otro mando y pula el botón A de nuevo para intercambiar los dispositivos de entrada.

Aviso! *Nunca desenchufes los dispositivos de entrada mientras está el juego abierto!*

Para desactivar un jugador, mueve el cursor sobre el rectángulo verde debajo de su mando y pulsa el botón **A**. Para volver a activarlo, pulsa de nuevo el botón **A** sobre el rectángulo, que ahora tendrá el color rojo.

Para cambiar a modo de pantalla completa, mueve el cursor sobre el botón “Full screen” - en la parte baja de la pantalla - y pulsa el botón A,. Para desactivar el modo de pantalla completa, vuelve a pulsa A sobre el botón. *Podrás desactivar el modo de pantalla completa en cualquier momento pulsando la tecla ESC.*

El jugador 1 podrá volver atrás seleccionando el botón “Back” y pulsando **A**.

9.4.3 Selección de personajes

En esta pantalla, cada jugador podrá utilizar los botones direccionales para mover su cursor sobre los botones. Los colores de cada jugador van en este orden: Rojo, Verde, Azul, Amarillo. Para seleccionar el personaje, pulsa el botón **A** sobre el jugador. Pulsa el botón **B** para dejar de seleccionar el personaje.

El jugador 1 podrá avanzar a la pantalla de selección de escenario pulsando **START** si al menos dos personas han seleccionado un personaje. Podrá volver atrás moviendo el cursor sobre el botón "Back" y pulsando **A**.

9.4.4 Selección de escenario

El jugador 1 podrá mover el cursor sobre uno de los escenarios y seleccionarlo con el botón **START**. Esto comenzará el juego.

9.4.5 Pantalla de juego

En esta pantalla se controla a los personajes. En la parte de abajo de la pantalla se muestran entre dos y cuatro números. Estos números muestran el daño acumulado de cada jugador. cuando se termina el juego, se abre la pantalla de victoria.

9.4.6 Pantalla de victoria

En la pantalla de victoria se muestran los personajes y su puesto en la competición. El jugador 1 podrá pulsar el botón **A** o **START** para volver al menú de selección de personajes.

9.5 El juego

9.5.1 Descripción del juego

El juego comienza con varios personajes en un escenario. Cada jugador controla un personaje. El objetivo es atacar a los oponentes para acumular daño y darles un golpe fuerte para echarles del escenario. Si lanzan a tu personaje, deberás intentar volver a una plataforma antes de que te caigas al vacío y pierdas una vida. Si pierdes todas tus vidas, no podrás seguir jugando. El ganador es el último personaje que queda en pie cuando todos los demás hayan perdido.

9.5.2 Controles básicos

Podrás mover a tu personaje por el escenario con los botones direccionales. Para saltar, pulsa el botón **Y**. Podrás saltar una vez más pulsando de nuevo el botón **Y**. Si estás en el aire, cercano al borde de una plataforma, podrás colgarte de él.

Si estás en el suelo y te atacan, podrás ponerte el escudo con el botón **R**. Cuando tienes puesto el escudo, los ataques no te harán daño. Si te desplazas con los botones direccionales, podrás moverte y esquivar ataques. Con el escudo puesto, podrás intentar agarrar a un oponente cercano pulsando el botón **A**. Si consigues agarrarlo, podrás lanzarlo en cualquier dirección con los botones direccionales.

Con el botón **L** puedes apuntar a un oponente, que se convierte en tu objetivo. Mientras estés apuntando a un objetivo, todos tus ataques se harán en su dirección. Además, podrás desbloquear varios ataques direccionales. Para dejar de apuntar, pulsa el botón **X**.

Puedes pausar el juego con el botón **START**. Para reanudar el juego, pulsa el botón de nuevo.

9.5.3 Ataques

Los ataques normales se hacen con el botón **A**.

- Si pulsas **A** en el suelo, harás un ataque neutro. Este ataque realiza daño, pero no lo lanza muy lejos.
- Si te estás moviendo, realizarás un ataque fuerte. Este tipo de ataque puede lanzar al oponente.
- Si pulsas un botón direccional en el mismo momento que realizas el ataque, podrás hacer un ataque “Smash”. Un ataque “Smash” tiene una mayor probabilidad de lanzar al oponente.
- Si pulsas **A** en el aire, realizarás un ataque aéreo neutro.
- Si pulsas **A** en el aire mientras te mueves, realizarás un ataque aéreo direccional.

Los ataques fuertes, “Smash” y aéreos se pueden hacer de tres formas según qué direcciones has pulsado en comparación con tu objetivo: Hacia el enemigo, en dirección opuesta a tu enemigo o en dirección paralela a tu enemigo.

9.5.4 Ataques especiales

Los ataques especiales pueden tener varios efectos. Al igual que con los ataques normales, existen cuatro según tu dirección de movimiento en comparación con tu objetivo: Hacia el enemigo, en dirección opuesta a tu enemigo, en dirección paralela a tu enemigo o un ataque neutro. Los ataques especiales de cada personaje son diferentes. Hay tres tipos:

- **Proyectil:** Lanza proyectiles en una dirección.
- **Ataque:** Realiza un ataque que te desplaza. Si el ataque se desplaza hacia arriba, podrás usarlo para volver a una plataforma si te han lanzado.
- **Contraataque:** Utiliza este ataque justo antes de recibir un golpe para evitar el daño y realizar un contraataque.

10. Anexo: Formato del fichero de un escenario: formato_escenario.txt

```
/* COMMENTS ARE WRITTEN LIKE THIS */
//END OF LINE COMMENTS ARE WRITTEN LIKE THIS
//Line breaks are considered white space
// '_' indicates an empty parameter
[BACKGROUND bg.png]

Anything written outside of square brackets is considered a comment.

[LIMITS left right up down front back]

[PLAYER number x y z] //Player number from 0 to 3 (or more, maybe...)

[PLATFORM NORMAL color texture.png width height length x y z]

[PLATFORM LINEAR color texture.png width height length
   startX startY startZ
   endX endY endZ
   period phase pause ]

[PLATFORM CIRCULAR color texture.png width height length
   centerX centerY centerZ
   Axis(X,Y,Z)
   period phase pause ]

[SPRITE texture.png orientation x y z frames ]
//Orientation can be -1 for a sprite that always faces the camera
```

11. Anexo: Formato del fichero de un personaje: formato_personaje.txt

```
[NAME Character Name]
[ID id]
[ATTRIBUTES speed jump width height weight grab_distance grabstart grabend
runspeed]
[FRAMES stand walk run jump fall hang shield
   neutral_attack
   strong_forward strong_back strong_center
   smash_forward smash_back smash_center
   neutral_aerial
   aerial_forward aerial_back aerial_center
   special_neutral special_forward special_back special_center
   grab grabbed flinch knock_back down ]
[TEXTURES stand walk run jump fall hang shield
   neutral_attack
   strong_forward strong_back strong_center
   smash_forward smash_back smash_center
   neutral_aerial
   aerial_forward aerial_back aerial_center
   special_neutral special_forward special_back special_center
   grab grabbed flinch knock_back down ]

/* ATTACKS */

[NEUTRAL [IMPACT ... ] [IMPACT ... ] ... ]

//Direction: [FORWARD,BACK,CENTER]
[STRONG direction [IMPACT ... ] [IMPACT ... ] ... ]
[SMASH direction [IMPACT ... ] [IMPACT ... ] ...

//Direction: [NEUTRAL,FORWARD,BACK,CENTER]
[AERIAL direction [IMPACT ... ] [IMPACT ... ] ... ]
[SPECIAL direction [SPECIAL_ATTACK ...]]



/* IMPACTS */
//Velocity: Not for neutral attacks.
//Min velocity: The minimum velocity to knock the character back.
//                      Not for neutral attacks.
//                    Set to -1 for other types.
//Min Percent: Minimum percent needed to knock back.
//                   Strong attacks only.
//                   Set to -1 for other types.
[IMPACT start duration velocity
   min_velocity min_percent damage height [HIT] ]

/* HITS */
//ROTATION HIT
// start_angle, end_angle: Seen from above: Clockwise is positive.
//                         Seen from the side: Down is positive.
// axis: [H,V] //Horizontal or vertical
// type: [N,T] //Normal or Tangent (Not for neutral attacks)
// radius
[ROTATION radius start_angle end_angle axis type]

// THRUST HIT
// angle: Always vertical. Down is positive
// length: in pixels.
[THRUST start_length end_length angle]
```

```
/* SPECIAL ATTACKS */
//charge: [0,1] Hold the button to charge, release to attack
//electric: [0,1] Paralyzes the enemy
//Set motion_speed to 0 to ignore motion
[ATTACK charge? electric?
     motion_start  motion_duration  motion_speed  motion_momentum  motion_h_angle
motion_v_angle
   [IMPACT...] [IMPACT ... ] ...
]

//Velocity multiplier: Speed at which the character flies.
//Min velocity: Lowest speed at which the character flies.
[EXPLOSION start_frame duration radius damage velocity_multiplier min_velocity]

//projectileID: A projectile in the database.
//Last two parameters: Horizontal and vertical angles in degrees.
[PROJECTILE height numProjectiles speed start_frame bitmap_name
   [ANGLES ...] [ANGLES ...]]


//Start frame: Begin shield increase
//End frame: End shield increase
//Duration: Time the shield can exist. -1 for endless.
//Absorb energy: Absorbs projectiles
[SHIELD color radius start_frame end_frame duration absorb_energy?]


//start_frame, end_frame: Animation before the attack.
//duration: Time when the counterattack is available.
//          The animation loops between start_frame
//          and end_frame during this time
//Multiplier: Multiplied by the damage received
[COUNTER_ATTACK start_frame end_frame duration multiplier
 [IMPACT ...] [IMPACT  ...] ... ]


//interval: Time between shooting one projectile and another.
//start_frame: Time to start shooting. The animation loops back
//             to this frame when it repeats.
//
[PROJECTILE_SOURCE projectileID start_frame duration angle speed interval]
```