



Taller de Programación I (75.42 / 95.08)

Jazz Jackrabbit 2

Trabajo Práctico Grupal - Documentación Técnica
Curso Veiga - Grupo 7

Integrantes

Avalos, Victoria	108434
Castro Martinez, José Ignacio	106957
Diem, Walter Gabriel	105618

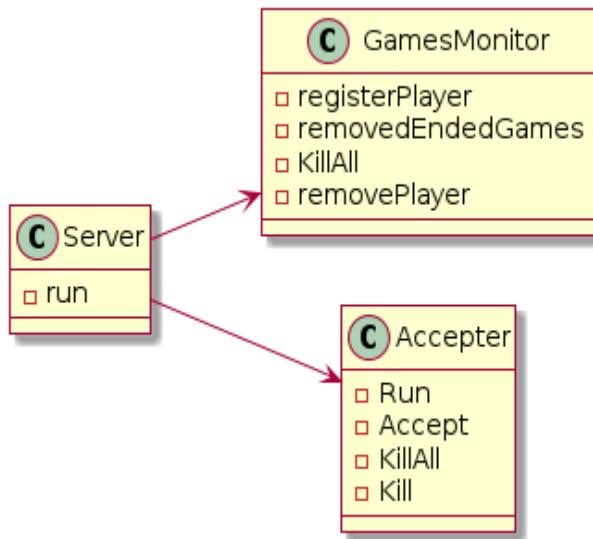
Índice

Integrantes.....	1
Índice.....	2
Servidor.....	3
Cliente.....	7
Engine.....	9
Interfaz gráfica.....	11
Lobby.....	11
Game.....	14
GraphicEngine.....	14
AudioEngine.....	15
Map.....	16
HUD.....	18
Text.....	19
Renderables.....	19
Renderer.....	20
GameOver Screens.....	21

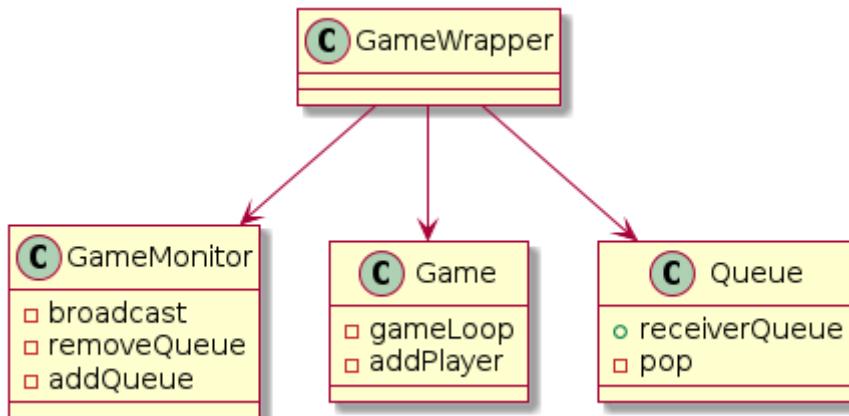
Servidor

El servidor corresponde a una pieza fundamental de la aplicación el cual habilita a las distintas computadoras a comunicarse mediante sockets TCP. Para la construcción del servidor se realizó un diseño considerando las necesidades en la agilidad para la comunicación entre el cliente y el servidor. El servidor está compuesto principalmente de dos piezas:

- Un monitor de partidas
- Un hilo aceptador



El monitor de partidas está encargado de funcionar como wrapper y administrador del recurso compartido más importante: los juegos, estos están contenidos en forma de una clase llamada game Wrapper que almacena todos los datos vitales de la partida y necesarios para poder identificarla, así como también del hilo del game loop y las colas necesarias para realizar la comunicación entre los distintos hilos.



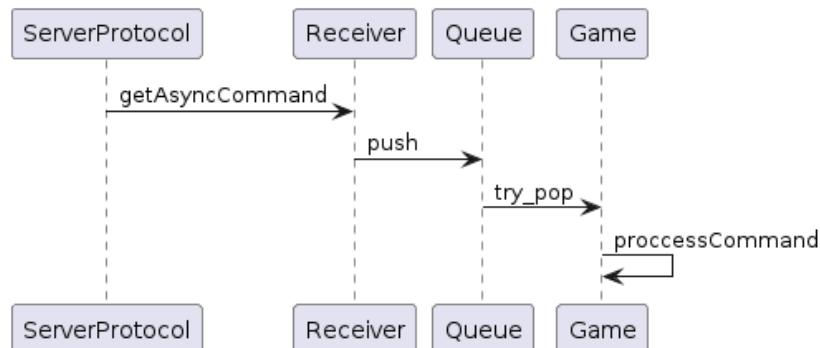
El hilo aceptador por otro lado es el manejador de los clientes que se conectan al servidor, contiene a los hilos Sender y Receiver de la partida y gestiona las comunicaciones y los estados en que se encuentran los mismos.



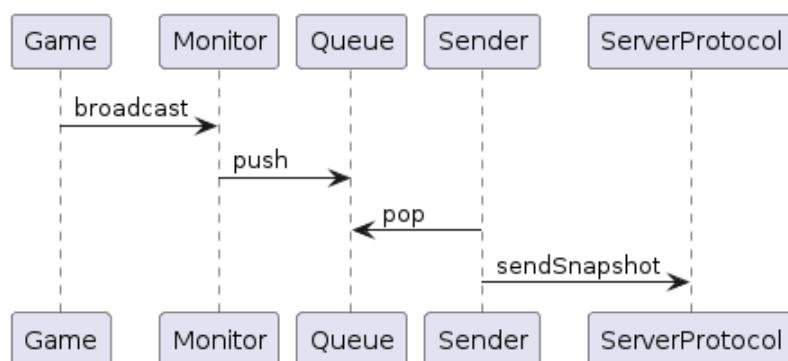
Los hilos de comunicación sender y receiver son hilos que utilizan colas bloqueantes para recibir y enviar mensaje entre cliente y servidor, ambos corren independientes del resto de la lógica y utilizan el protocolo del servidor para poder recibir la información mediante socket.

Las colas utilizadas en el servidor son estructuras de datos “thread save” y utilizan lógica bloqueante dentro de los hilos para reducir el uso del cpu y aprovechar mejor los recursos

La información viaja de manera lineal a través de las capas del server para poder realizar la comunicación efectiva y para que el server pueda actuar de manera correspondiente comunicando el estado del juego a sus clientes y actualizando según los comandos que han sido realizados. El siguiente diagrama muestra cómo le llega la info al server a través del protocolo



Por otro lado, de la siguiente manera se envia la informacion desde el juego hasta el correspondiente hilo sender



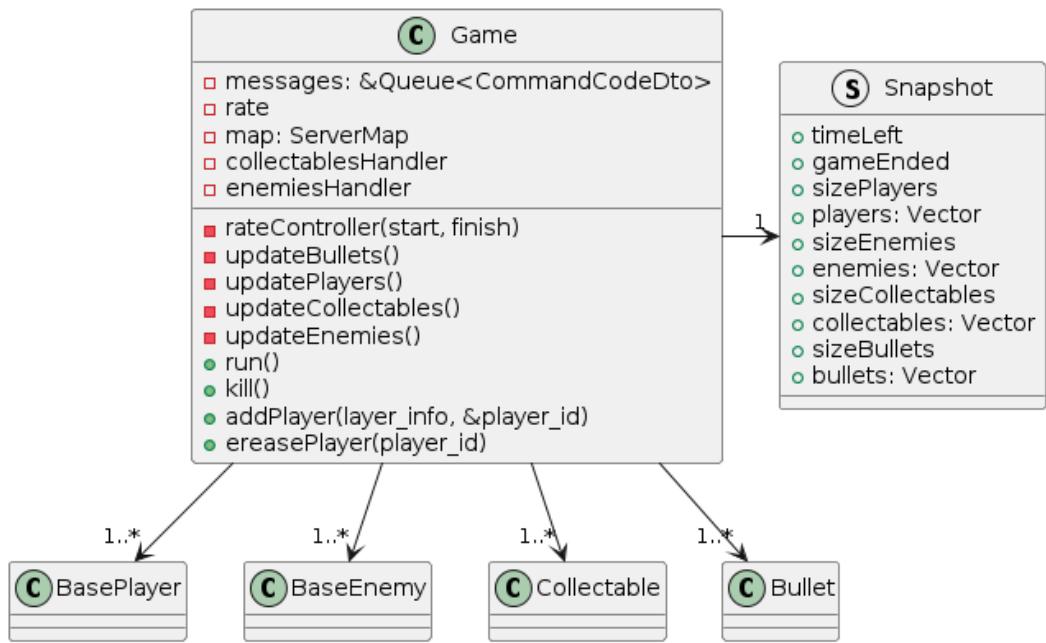
Respecto a la integración con el engine, se cuenta con la clase Game que posee un mapa con los distintos jugadores. Esta clase es la que lleva a cabo el loop principal del juego con un rate constante. Por cada iteración se procesan un máximo configurable de instrucciones, establecido a priori como 100. En base a la instrucción, se invoca al método de BasePlayer correspondiente, excepto para los cheats que realizan otras funciones en el juego no relacionadas al jugador.

Para las instrucciones, el Game posee una referencia a la cola de comandos. El gameloop no se puede detener, así que se utiliza una cola no bloqueante para extraer las instrucciones, de forma repetida hasta cumplir con la cantidad máxima mencionada anteriormente. Si antes del máximo la cola se encuentra vacía, se deja de intentar por esa iteración.

El Game posee el mapa, un vector de balas, uno de enemigos y uno de coleccionables. Cuenta con un método de actualización para cada vector, en los cuales se verifican las intersecciones: si una bala intersecta a un jugador o a un enemigo para hacerle daño, si un jugador intersecta con un ítem para recolectarlo, si un jugador está realizando una acción especial e intersecta con otro jugador o con un enemigo para realizar daño, y si un jugador está en el rango de un enemigo para hacerle daño. Luego, se invoca al método correspondiente.

Se posee una única instancia del struct Snapshot que se modifica con cada acción, y la misma instancia se envía al cliente constantemente. El mismo posee vectores de otros structs, un struct por cada tipo de entidad: jugador, enemigo, coleccionable y bala. Todas las entidades conocen al snapshot y modifican los flags correspondientes cuando sucede una acción. Por ejemplo, cuando un jugador se mueve, modifica su posición en la snapshot.

A continuación se muestra un diagrama de clases con los métodos y atributos más importantes. En el apartado de Engine, se entra más en detalle respecto de las entidades y su funcionamiento.



Cliente

En éste apartado se hablará únicamente del back-end del cliente, el cual posee dos componentes principales: el lobby y el cliente.

El Lobby posee una estructura secuencial sin la implementación de hilos. Consiste de una clase Lobby que posee como atributo a la clase LobbyProtocol. La comunicación a través del socket se delega al protocolo.

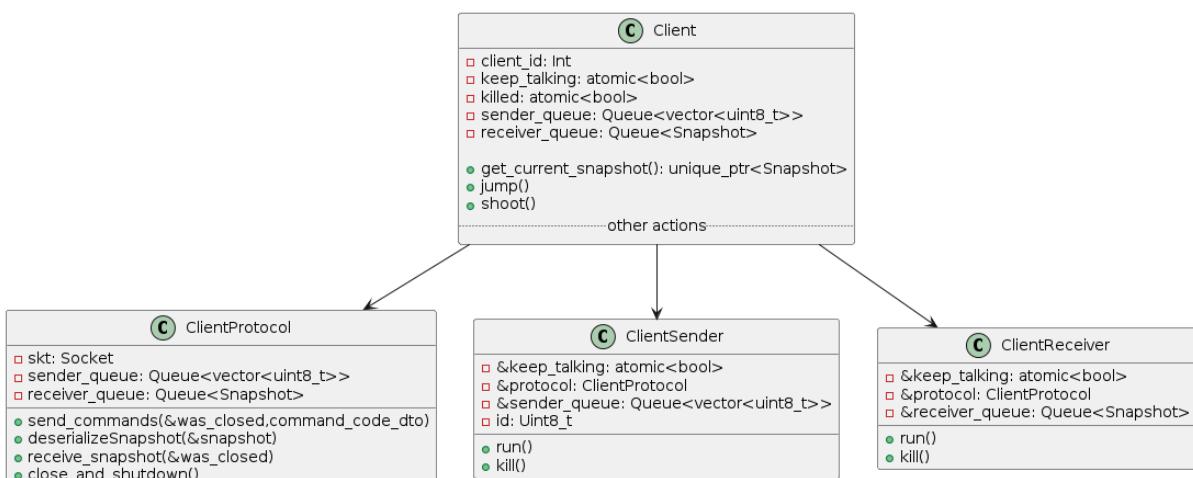
El mismo realiza las siguientes acciones:

1. Realizar la conexión con el servidor mediante un socket.
2. Recibir la información de los juegos disponibles.
3. Refrescar (avisar al servidor y volver a recibir) dicha información si el usuario presionó el botón de refresh.
4. Enviar el juego seleccionado. Si el jugador creó una partida en lugar de unirse a una, la información del juego seleccionado se envía de la misma forma, y es el servidor el que se encarga de procesarlo como una partida nueva.
5. Recibir el id que el servidor le asignó al jugador.

El Lobby libera los recursos del socket si el usuario cierra el juego antes de que muera el Lobby.

Luego, el ownership del socket utilizado para la conexión con el servidor es transferido al Client y la instancia de Lobby muere.

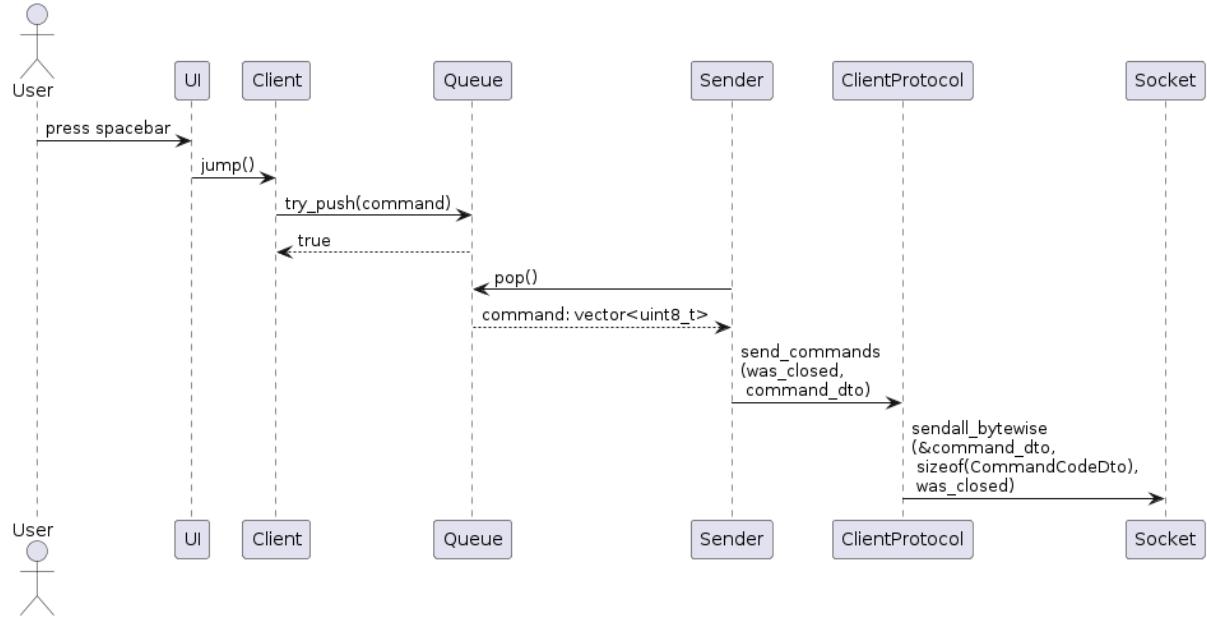
El Client cuenta con una arquitectura basada en hilos: posee un hilo receiver y uno sender. Ambos ejecutan un ciclo que dura hasta que el usuario cierre el juego o el mismo finaliza. La clase Client es quien lanza ambos hilos y los une al final, y también quien posee el ownership de las colas tanto del sender (la que posee los comandos del usuario como vectores de uint_t) y del receiver (la que posee las snapshots).



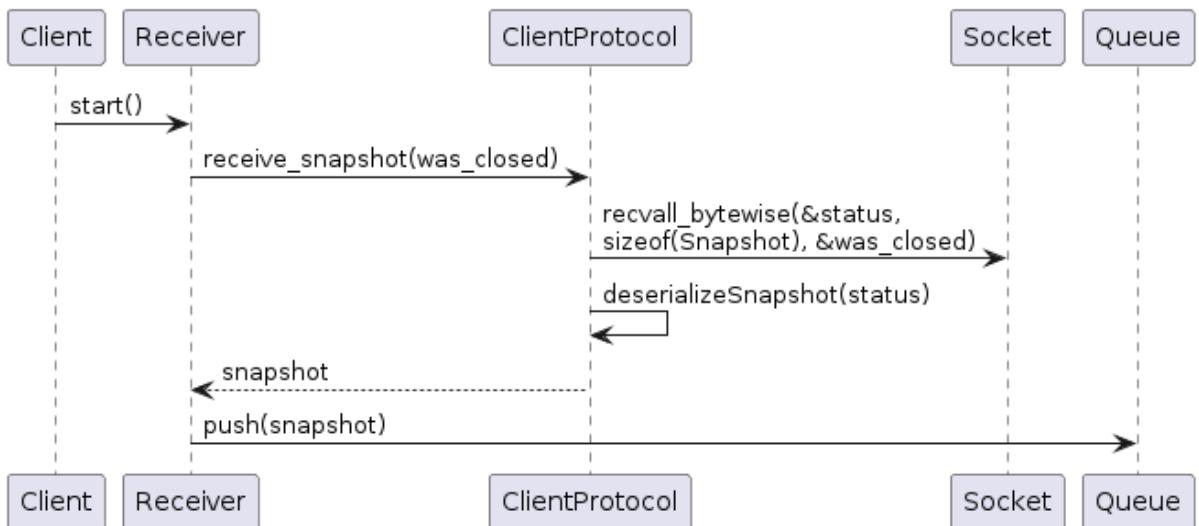
Cabe aclarar que las clases ClientSender y ClientReceiver heredan de la clase Thread ofrecida por la cátedra: <https://github.com/eldipa/hands-on-threads>

Así mismo, la clase Socket utilizada también está basada en la ofrecida por la cátedra:
<https://github.com/eldipa/sockets-en-cpp>

A continuación, muestro un caso de uso del cliente donde un usuario presiona la tecla de espacio:



Por parte del receiver, el flujo de trabajo siempre se ve de la misma manera ya que siempre recibe snapshots y las maneja de la misma forma.

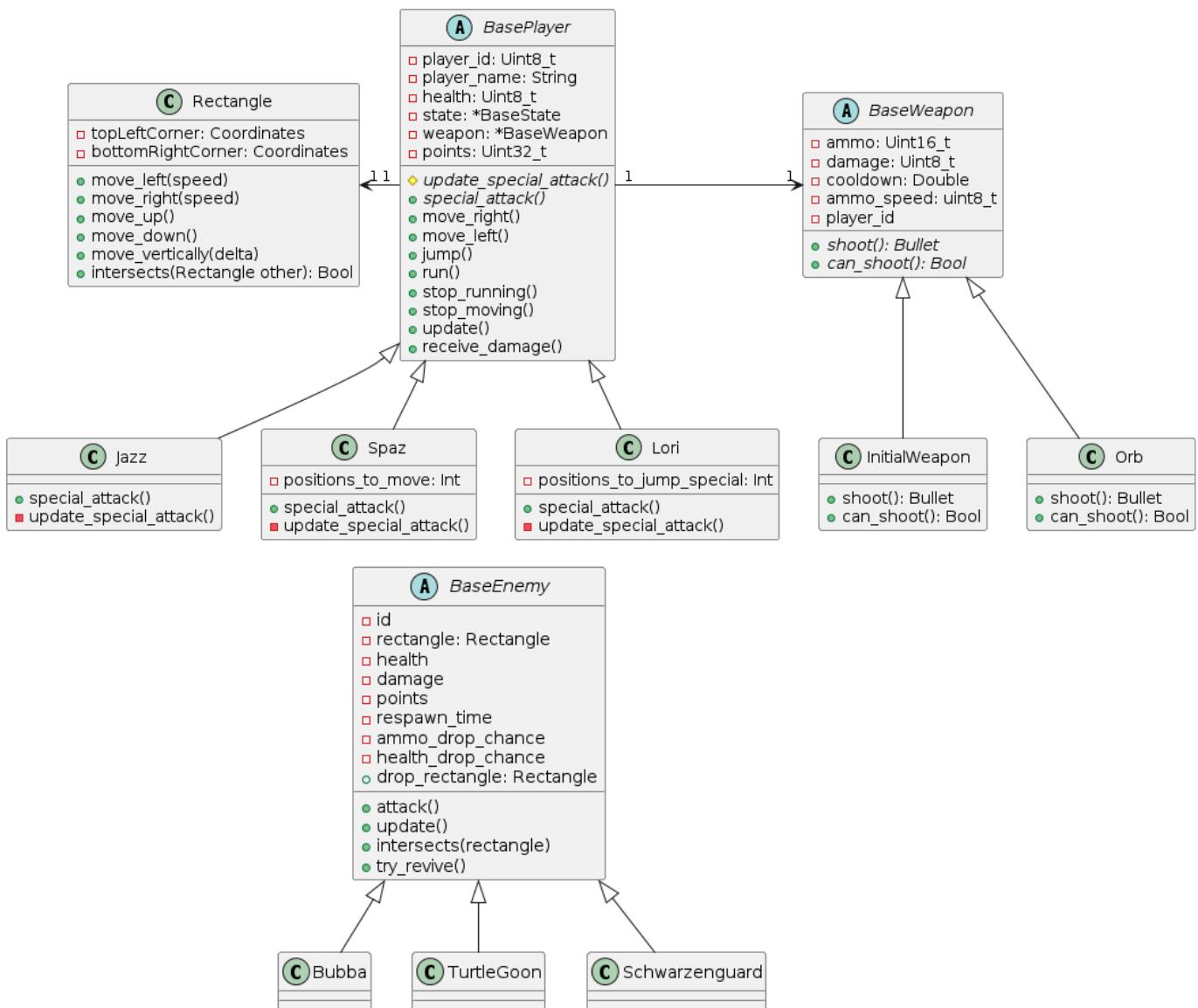


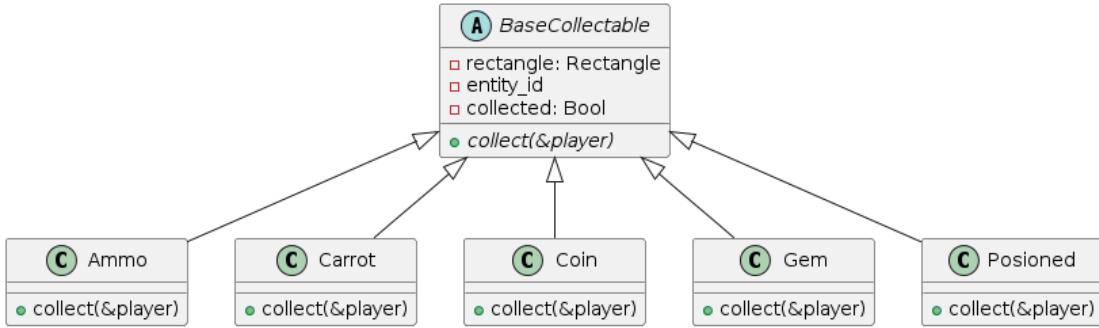
Finalmente, el loop principal del cliente tiene lugar en el Renderer, donde se integra el cliente con la UI. Dicho loop tiene un rate constante y es donde se invocan los métodos del Client en base a la acción realizada por el usuario.

Engine

El engine del juego cuenta con los siguientes elementos: players, weapons, states, enemies, collectables y bullets. Para todos ellos, excepto bullets, se utiliza polimorfismo para manejar los diferentes comportamientos y cuentan con una clase base abstracta. Se utilizan punteros inteligentes a las clases base. Un jugador tiene un arma y un estado, que cambian dinámicamente en base a lo que suceda en el juego. El arma crea una bullet que se agrega al vector de balas en el Game, a la cual le inyecta el daño y velocidad correspondientes, junto con el id del jugador que la disparó.

A continuación se muestran diagramas de clase con algunos de los métodos y atributos más importantes:





En los enemigos, las clases hijas setean los atributos correspondientes. De forma similar, con los estados hay una clase padre abstracta y las clases hijas determinan si se pueden realizar ciertas acciones, como disparar o moverse.

Se utiliza un contador global con el patrón singleton para crear ids únicos para todas las entidades.

Respecto del manejo de colisiones, se utiliza el sistema conocido como AABB. Cada entidad tiene un rectángulo, el cual posee una esquina superior izquierda y una esquina inferior derecha. Para identificar una intersección, la clase Rectangle posee un método intersects que se compara a sí mismo con otro rectángulo en base a dichas esquinas para determinar si están colisionando o no.

El jugador posee un método de update que, entre muchas otras cosas, intenta mover hacia abajo su rectángulo, y se fija si hay una intersección con el mapa. El mapa del juego es un vector de rectángulos que representan el piso, las paredes, el techo y las plataformas. De ésta forma se simula una “gravedad”.

Interfaz gráfica

El juego consta de 2 ventanas gráficas, el Lobby y el Game.

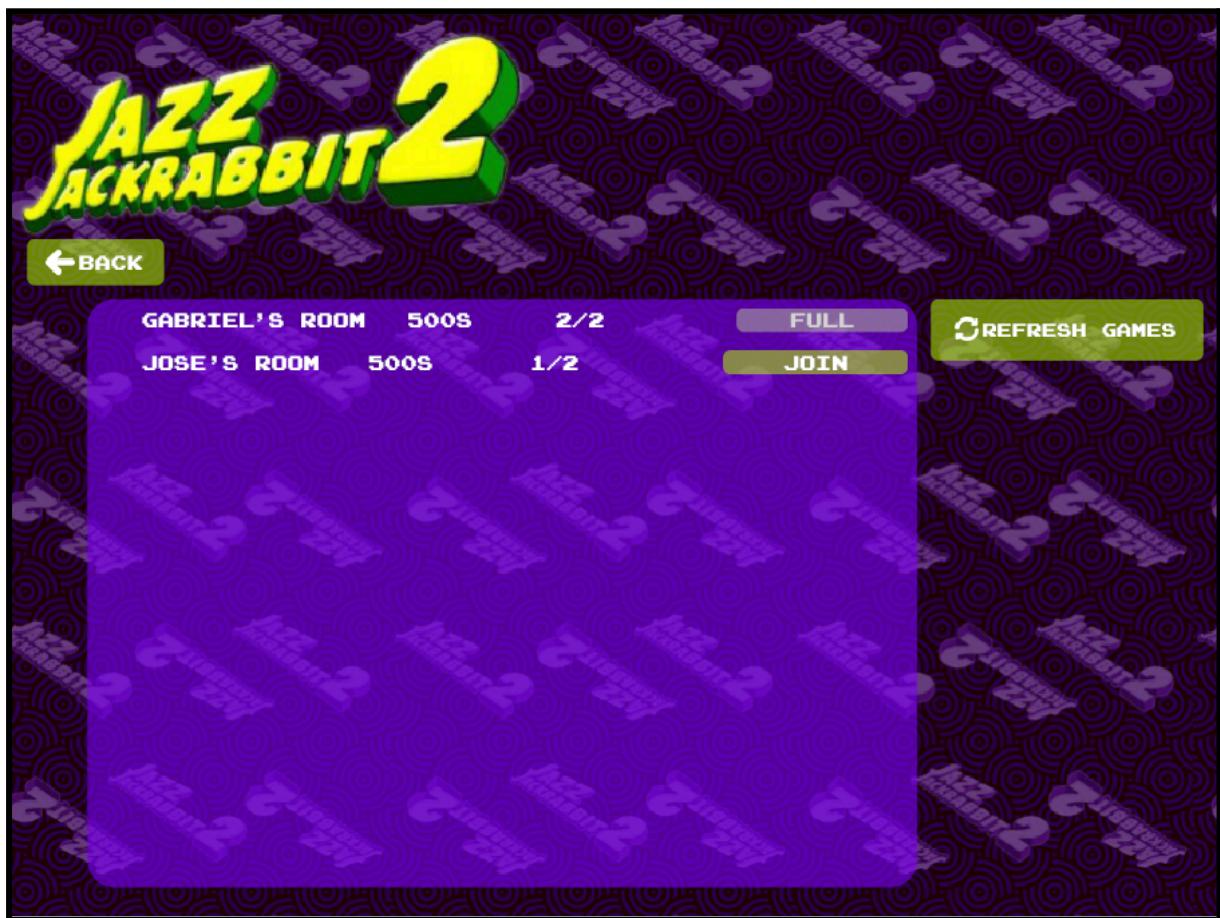
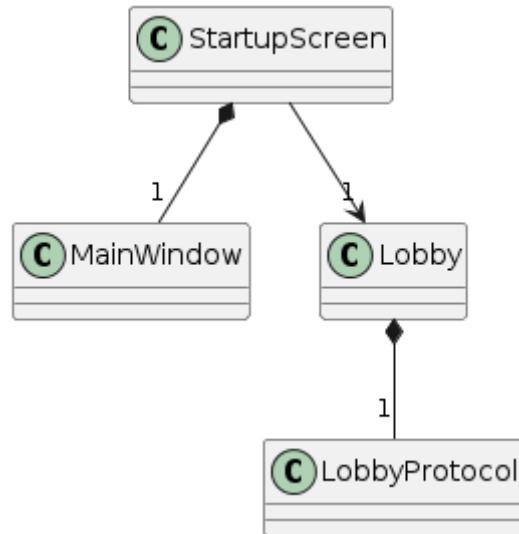
Lobby

Es la primera ventana que aparece al iniciar el cliente. Construida íntegramente en Qt5 mediante la fusión del XML generado con *Qt Creator* y código custom para generar elementos on-the-fly de manera dinámica.

La ventana contiene un *StackedWidget*, donde cada “pantalla” distinta en un *Widget* del *StackedWidget* que se va cambiando.



Se encuentra en el código como la clase *StartupScreen*, esta se conecta con la clase *Lobby*, que posee el *LobbyProtocol* encargado de hacer las comunicaciones con el servidor mediante comandos.



El Widget de la sala de juegos se genera dinámicamente en base a la información recibida por el server al inicio de la conexión o al presionar el botón de *Refresh Games*.

Cada input, ya sea de determinados botones como el botón de *Join*, como los *LineEdits* primero son validados para ver si se puede proceder correctamente. Como ejemplo, en el caso del *LineEdit* de *username* se valida su longitud. En el caso particular del botón de *Join*

se hace un refresh de los juegos apenas se lo presiona para ver si el juego sigue con espacio disponible, sino se le avisa al usuario que está lleno.



En la ventana de espera, se le delega la responsabilidad de cerrar la *StartupScreen* a un thread específico que se bloquea hasta recibir la primera snapshot del server, en términos del protocolo esto significa que todos los jugadores se unieron y comienza la partida. De esta manera no se bloquea el thread de la UI de Qt.



Al seleccionar un personaje se reproduce el gif del mismo, realizando un movimiento.

Cada botón del Lobby tiene un sonido de click asociado. También está implementada la música de fondo que es controlada (encendido, apagado y volumen) por el archivo de configuración YAML.

Todos los recursos que *StartupScreen* usa mediante Qt están declarados en un *QResource File* para que sea accesible.

Game

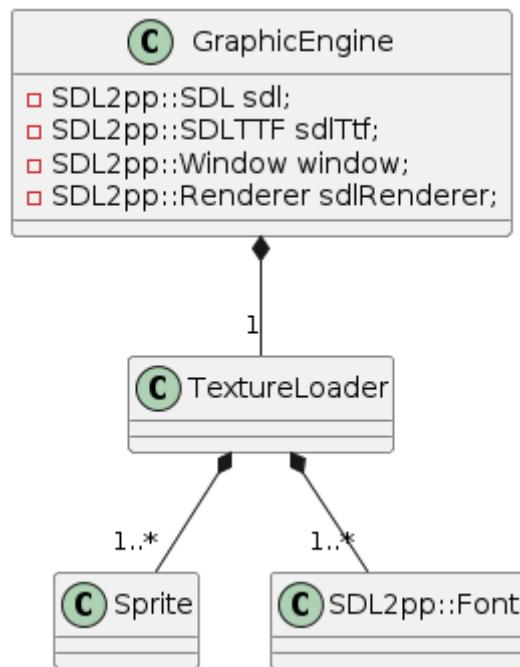
Una vez se cierra el *Lobby* porque se recibió la primera snapshot, se precarga el *GraphicEngine*, el *AudioEngine*, el mapa y el HUD para comenzar el renderizado por el *Renderer*.

GraphicEngine

Primero, un *Sprite* consta de la textura de SDL almacenada en memoria (SDL Texture) más los datos de cómo usar el *Sprite* que se cargaron de un archivo YAML de metadata asociado. Dicha metadata cuenta con la cantidad de frames y la posición de los mismos dentro del spritesheet. El proceso de creación de estos sprites fue manual, usando un software especializado para partir spritesheets generales (o sea con múltiples animaciones

y tamaños irregulares) en spritesheets específicos con el YAML asociado. El nombre del software es *TexturePacker*.

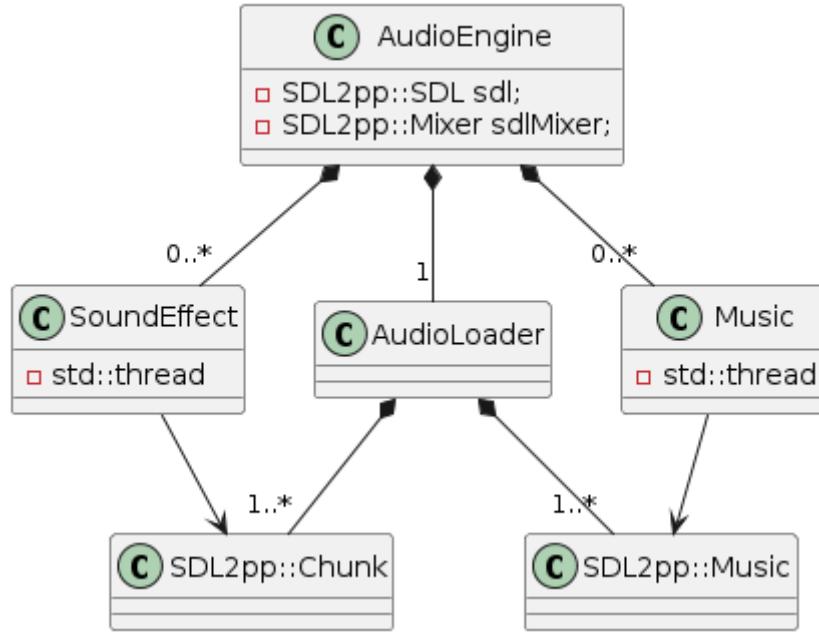
El *GraphicEngine* propiamente dicho, inicializa la ventana de SDL con los parámetros adecuados y el renderer propio de SDL. Mediante el *TextureLoader* precarga los *Sprites* en memoria y mantiene su ownership, de esta manera cada clase que necesite renderizar un *Sprite* utilizará una referencia en lugar de cargar de nuevo un sprite desde el archivo, para luego hacer el rendering.



AudioEngine

El *AudioEngine* funciona de manera similar al *GraphicEngine* pero con los audios, haciendo diferencia entre *SoundEffects* (sonidos cortos) y *Music* (piezas de música largas). En este caso el *AudioEngine* hace la precarga mediante el *AudioLoader*.

Cuando se solicita al engine que reproduzca un sonido, lanzará un thread individual para reproducirlo, y lo joineará una vez terminada la reproducción, mediante una revisión periódica de los tracks finalizados.



Map

El Map (mapa) realiza la carga de las coordenadas desde un archivo YAML de coordenadas, donde se diferencia entre bloques de tierra (full dirt), bloques con pasto (top grass) y pendientes (slopes). Esto se usa para luego renderizar los sprites correspondientes al tipo de bloque. Dicho YAML contiene las definiciones de los segmentos del mapa, estos son los costados, la base, el piso y las plataformas. Las coordenadas para las pendientes están definidas con una posición en X, una posición en Y y la orientación hacia donde mira la pendiente. Las coordenadas del resto de los tiles están definidas con un inicio y fin de una tira de tiles, cada coordenada compuesta por una posición en X y una en Y.

La cámara se mueve de acuerdo a la posición de jugador (clase Player) y a su ubicación relativa a la esquina superior izquierda. Esto permite luego no renderizar partes del mapa, renderables ni reproducir sonidos que estén fuera del foco de la cámara.

Hay dos escenarios posibles para el mismo mapa, Carrotus y Beach World, los cuales son seleccionados en la ventana del Lobby.

Carrotus:



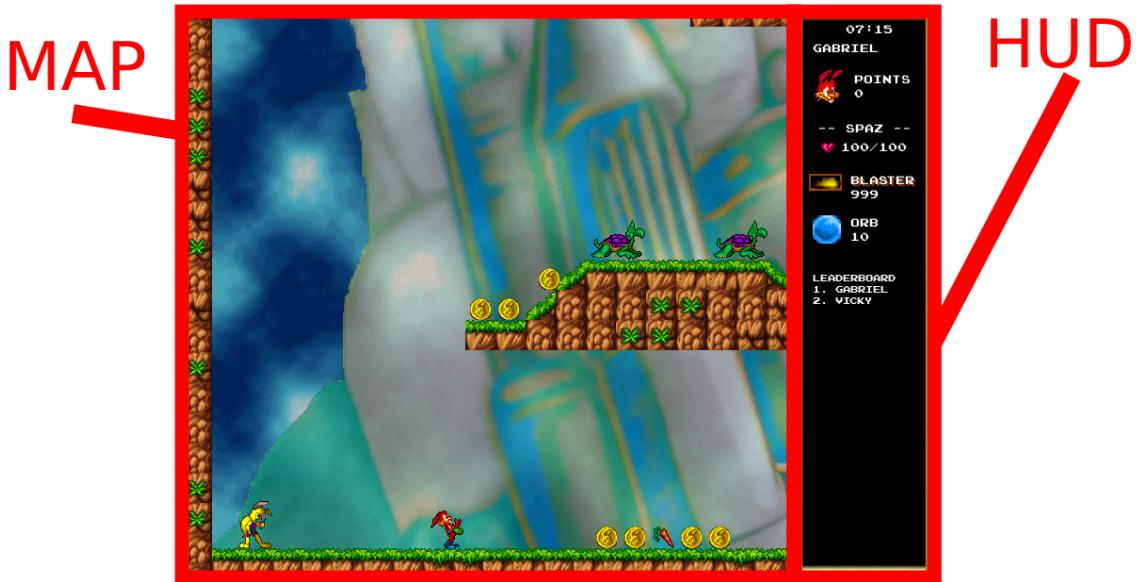
Beach World:



HUD

Junto con el Mapa conforman toda la vista de la pantalla de SDL. Contiene información relevante del estado del juego junto con los atributos clave del jugador. En orden desde arriba hacia abajo se observa:

- El tiempo restante de la partida. Cuando llegue a cero la partida finalizará y se verá el leaderboard final.
- El username del jugador.
- Los puntos de jugador al lado del ícono del personaje que eligió.
- El nombre del personaje que eligió para jugar.
- Los puntos de vida restantes sobre los puntos de vida totales.
- El arma principal (Blaster), que muestra una munición de 999 porque tiene munición ilimitada. Tendrá un recuadro naranja sobre el ícono del arma si el arma está seleccionada.
- El arma secundaria (Orb), junto con la munición restante. Tendrá un recuadro naranja sobre el ícono del arma si el arma está seleccionada.
- Un resumen del top 3 de cómo va el leaderboard.



La información mostrada se actualiza frame a frame acorde a las snapshots recibidas desde el server.

Text

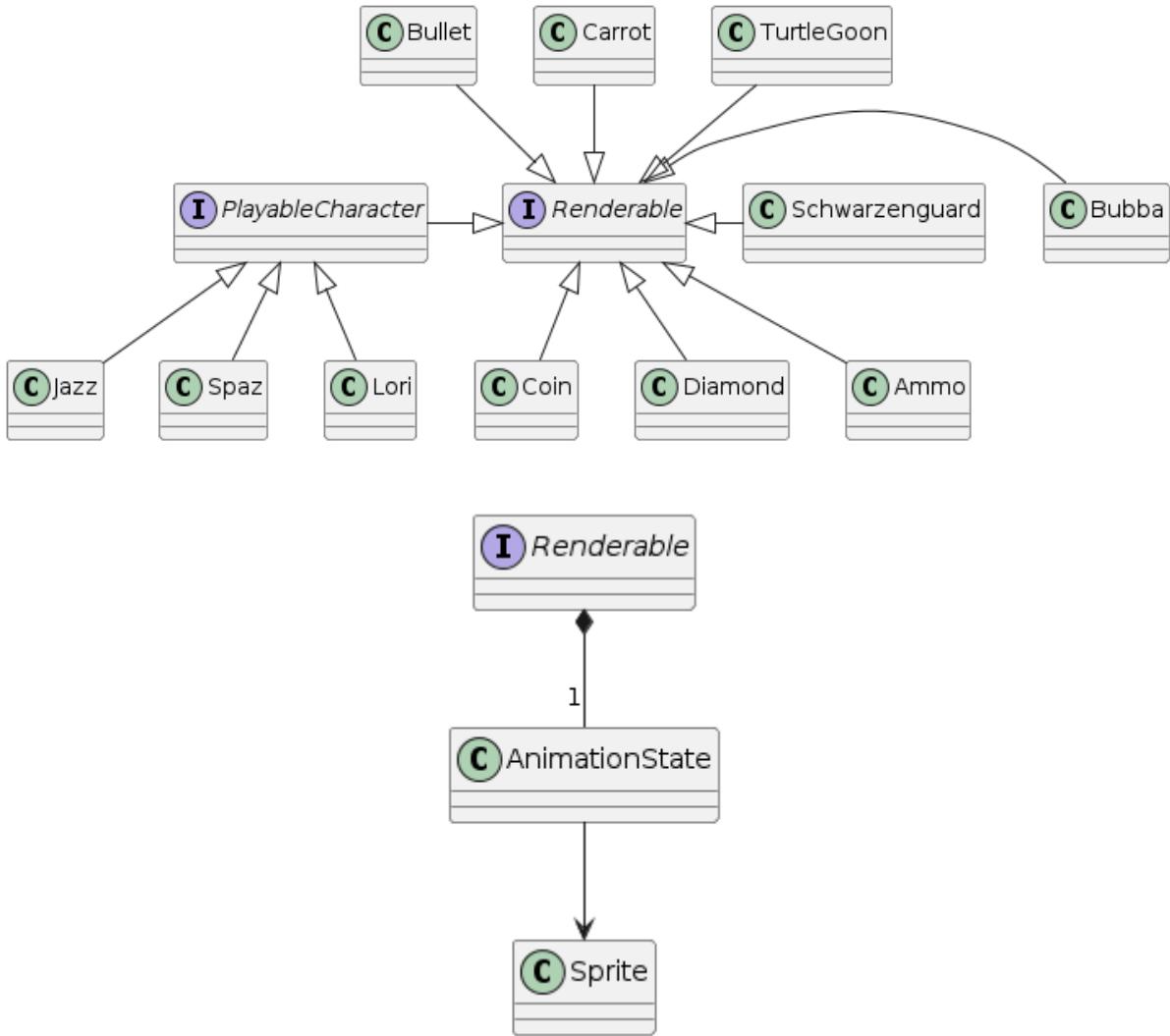
Esta clase asegura que un texto que será renderizado esté contenido y no se desperdicie memoria y/o procesamiento. Esto es debido a que el texto interno puede cambiar o no, y si no cambia no se quiere crear una nueva Texture de SDL para almacenar ese texto, sino preservar la que ya está, lo cual es la responsabilidad de esta clase.

Renderables

Los *Renderables* son aquellos que comparten una interfaz y se pueden renderizar sobre el renderer de SDL. *Renderables* son:

- *Enemies*: los 3 enemigos implementados, Bubba, Turtle Goon y Schwarzenegger.
- *Collectables*: las zanahorias, las monedas, los diamantes y los packs de municiones.
- *Bullets*: las balas de las dos armas, Blaster y Orb.
- *Playable Characters*: los personajes jugables, Jazz, Spaz y Lori.

Estos *Renderables* usan un *AnimationState* que es lo que efectivamente renderiza el frame del *Sprite* correspondiente. Este estado cambia dependiendo del estado de la entidad acorde a los datos recibidos en la snapshot y es el encargado de determinar comportamientos como si la animación puede ser interrumpida (reemplazada) por otra animación (llamado si la animación “es rompible”), si debe ciclar o debe mantenerse en el último frame una vez terminados los frames del sprite, entre otros.



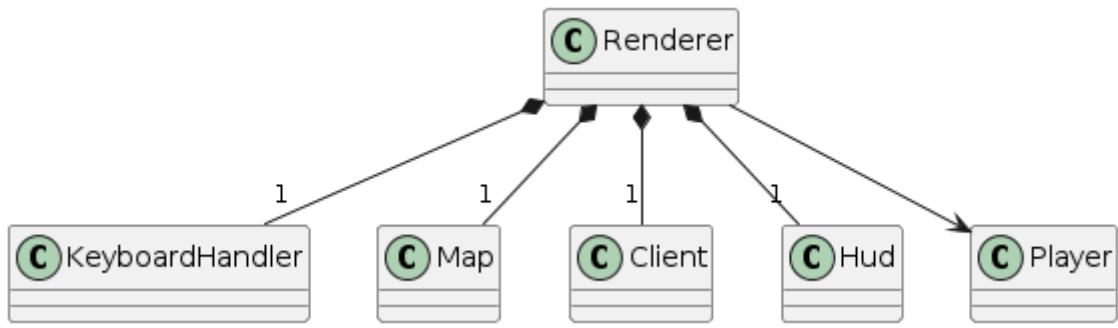
Renderer

El *Renderer* maneja el *constant rate loop* utilizado para el renderizado, renderizando cada uno de los *Renderables* existentes y creando nuevos si se debiera.

Se detecta que un nuevo *Renderable* debe ser creado cuando no se encuentra la entity id del mismo en los *Renderables* que están instanciados en ese momento. Y se detecta que uno debe ser eliminado cuando el mismo *renderable* responde con true al método *shouldDelete*.

Este además maneja el control de la ejecución del *Game* mismo, además del manejo de entidades para el renderizado, actualiza la snapshot y las entidades, y recibe y envía mensajes con el servidor utilizando el *Client*, y procesa el input del usuario con el *KeyboardHandler*.

El *KeyboardHandler* ignora los inputs marcados como repeats, esto se da cuando la tecla se mantiene presionada.



GameOver Screens

Hay dos pantallas que se muestran cuando el juego termina, y cada una corresponde al motivo de finalización del juego.

Cuando el juego finaliza por la desconexión del servidor o de un jugador, se muestra la pantalla de desconexión:



Y para el caso cuando se finaliza el juego normalmente, esto es porque el tiempo restante llegó a cero, se muestra la pantalla de finalización de juego con el Leaderboard final.



Ambas pantallas deshabilitan todo input excepto el utilizado para cerrar la ventana y desconectan al cliente del servidor.