Duck Game

Documentación técnica

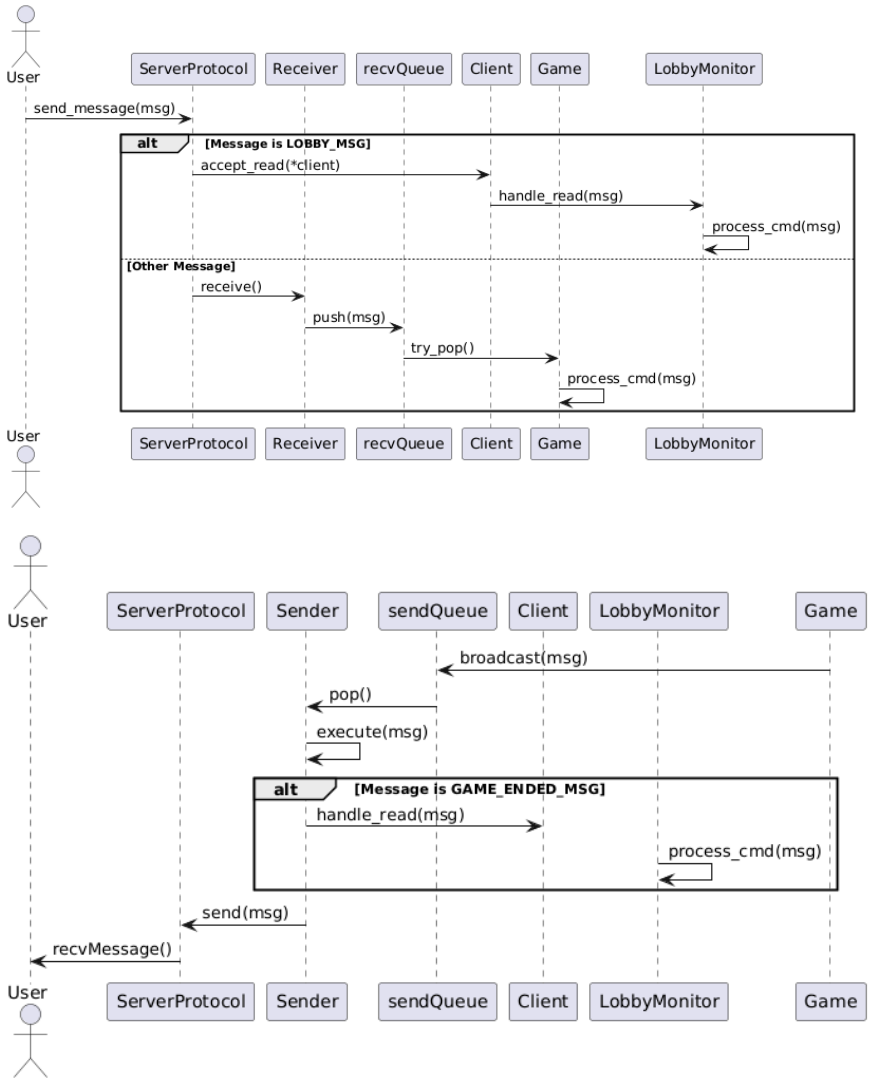
Índice

Servidor

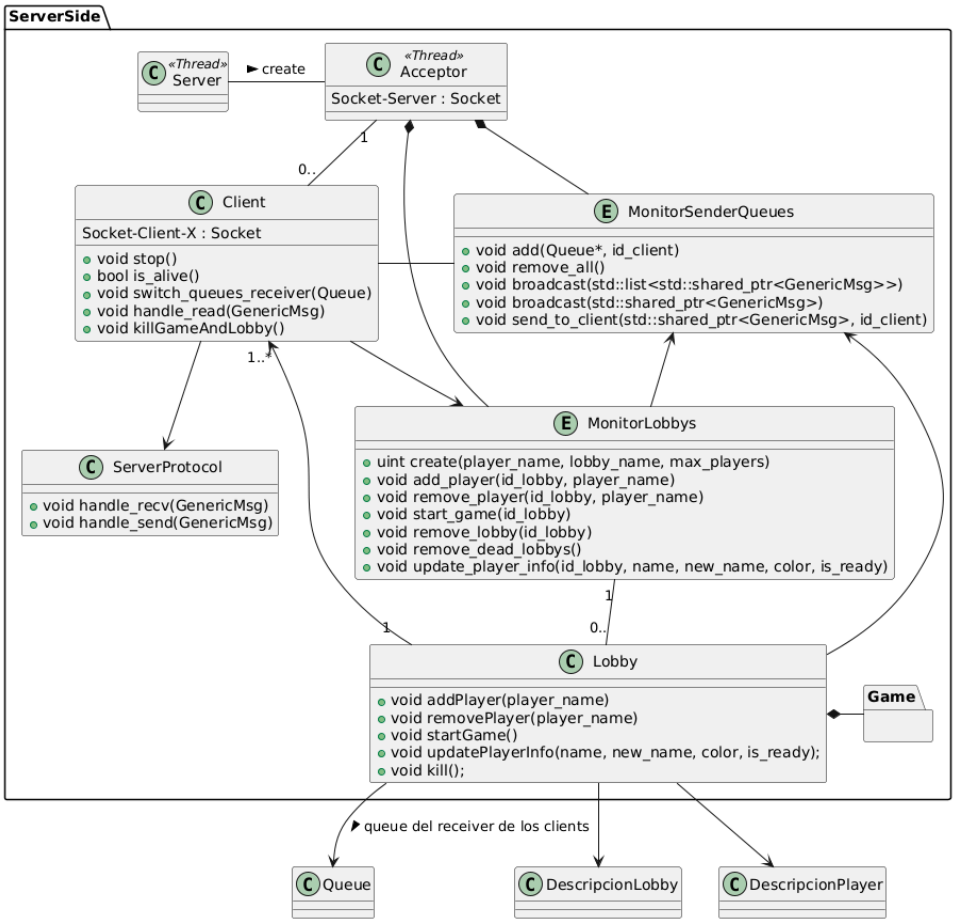
El servidor es responsable de articular la comunicación entre distintas máquinas a través de un sistema de sockets con protocolo TCP, además de gestionar y procesar este intercambio de información. Cuenta con distintas piezas, principalmente son:

- Hilo aceptador de clientes
- Monitor de lobbies
- Lógica del juego: hilo gestor del juego en curso

El hilo `Acceptor` se encarga de recibir conexiones entrantes, generar un `Client` por cada una y eliminar Clients de conexiones que terminaron. Cuando se crea un Client, este contará con dos *protected queues* bloqueantes, que a través de dos hilos `Receiver` y `Sender`, una para recibir mensajes a través del socket y otra para enviarlos, todo utilizando el protocolo del servidor. El servidor gestiona una *send queue* por cada socket de cliente, entonces se cuenta con un monitor de send queues ya que es un recurso compartido entre el aceptador y otras partes del servidor.



El monitor de lobbies `LobbysMonitor` se encarga de administrar los juegos (o lobbies) que se crean por los clientes. Contiene un mapa de lobbies `Lobby` activos identificados por su ID. Permite crear un lobby (agregándolo a la lista de lobbies) y remover lobbies; agregar y remover jugadores de un lobby determinado; remover lobbies que cerraron o que empezaron la partida y también le indica a determinado lobby que debe iniciar su juego. Cuando se inicia un juego se hace un cambia la referencia a la recv_queue del cliente para que ahora sea la del Game.



La lógica del juego empieza cuando se inicializa el hilo `Game` que lanza que lanza la partida. Cada partida cuenta con un `Stage` que modifica el mapa actual (representado en una matriz) en juego. Cuenta con clases que determinan el estado de los jugadores `Player`, de los ítems `Weapons` (que incluye armas, armaduras y cajas misteriosas). Y se encarga de recibir comandos desde el lado del cliente (a través de la recv_queue) y enviar el estado actualizado del juego. En el apartado Lógica del juego se explica con más detalle esta estructura.

Lógica del juego

La clase `Game` se encarga de orquestar todo el flujo de juego: inicializa stages, jugadores y ejecuta la lógica del juego. Por cada ronda, el game instancia un objeto `GameLoop` para manejar esa ronda específica que tendrá tales ítems en tal mapa.

El game le provee al gameloop el conjunto de jugadores (mapa de objetos `Player`) y la cola para recibir comandos de todos los clientes en esa partida. El gameloop interactúa con estos objetos para procesar las acciones de los jugadores, la generación de ítems y el ganador de una ronda.

La clase `Stage` representa al *escenario* donde ocurre el juego. Se ocupa de rastrear el comportamiento de los jugadores; maneja el agregado, remoción o actualización de proyectiles; maneja las tiles del mapa y las interacciones entre ítems y jugadores: determina posiciones válidas, procesa daños, y maneja eventos in-game (por ejemplo explosiones). Los eventos que afecten a un jugador se gestionan a través de la clase mencionada anteriormente `Player`: estado del jugador, equipamiento, interacción con el stage (moverse, disparar, tirar items).

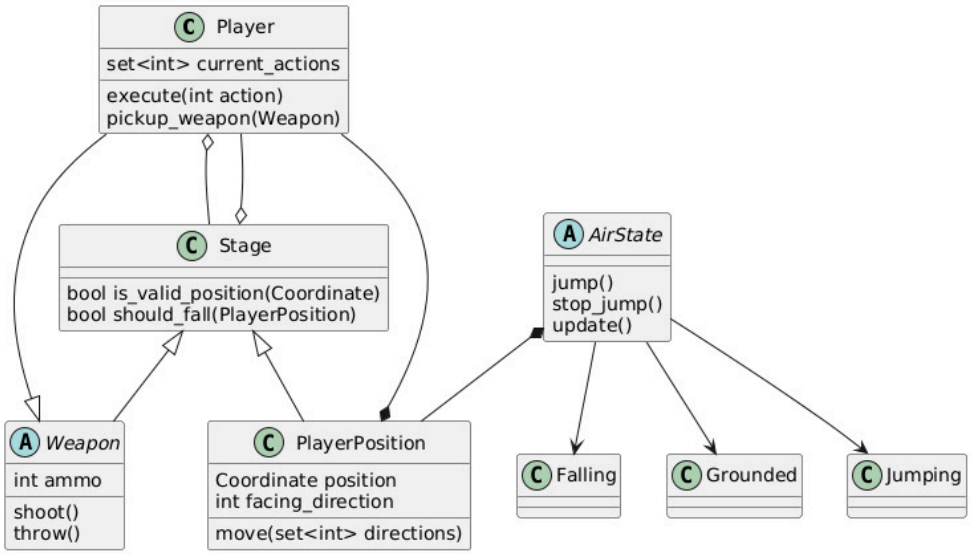
La clase `MapManager` lee de archivos .yaml la información necesaria para generar un `Map` que plasma a un único mapa, almacenando y gestionando la matriz que lo representa, además mantiene información sobre los sitios de spawn de jugadores e items y valida límites. El game flow inicia con el MapManager cargando todos los mapas, Game le pide un mapa random a MapManager, Stage se inicializa con el Map proporcionado por el MapManager y el Stage lee y modifica al Map

Para la creación y configuración de las características particulares de cada arma `Weapon` en el juego se extrae esa información de un archivo de configuración `weapon_config.yaml` y la información relacionada a los ids que representa cada ítems en el mapa, y datos específicos relacionados a eventos del juego se extrae del archivo `config.yaml`

La lógica del movimiento del jugador está encapsulada en `PlayerPosition`, que en un puede tener uno de varios `AirState` implementados. Dichos estados determinan cómo el jugador se comporta en distintos escenarios. Por ejemplo, si el estado es Jumping y llega un mensaje para que salte, no volverá a saltar.

El jugador tiene guardada también una instancia de la clase abstracta `Weapon`, que tiene encapsulada la lógica de disparar. Una vez que dispara, el arma crea una o varias instancias de la clase `Proyectil`, que se encarga de su propio movimiento a lo largo del Stage.

En cuanto al comportamiento de Weapon, ya que hay colisiones entre todas las entidades del juego, o sea que un jugador no puede pasar por el "frente" de un arma, se definió que si un jugador suelta un arma contra una pared, el arma desaparece, no rebota. Sí se puede tirar al piso y al vacío.



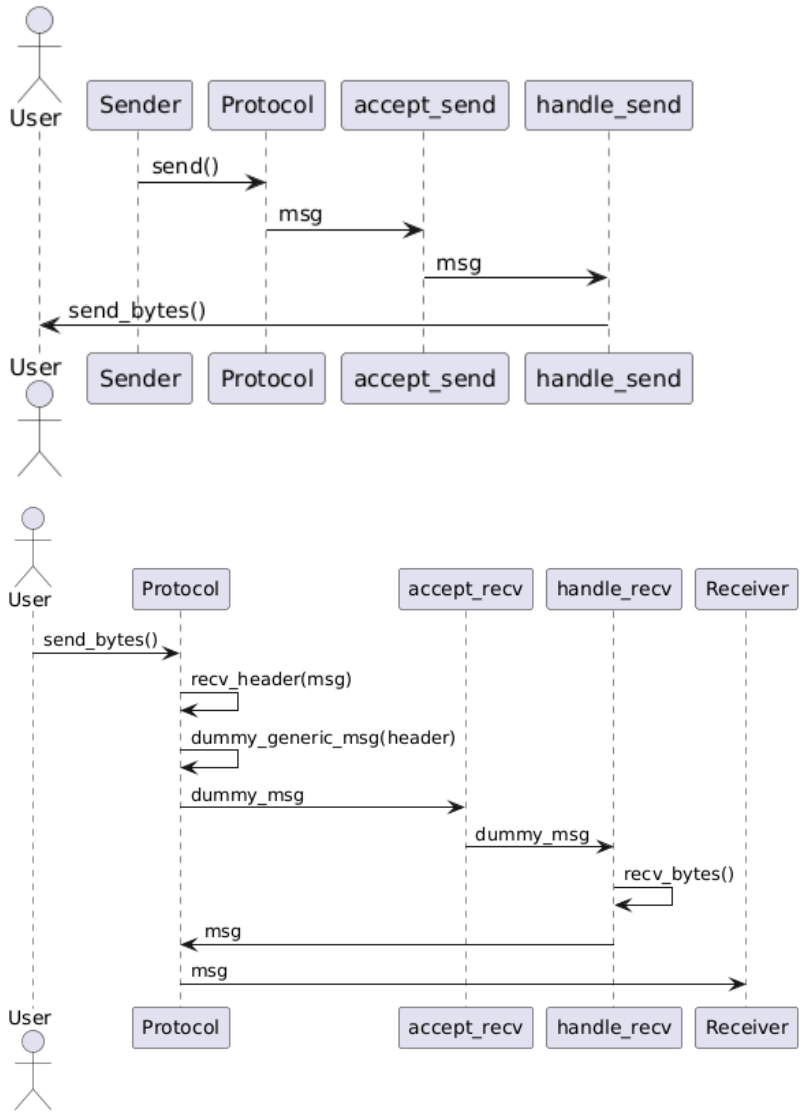
Protocolo

El protocolo está diseñado para gestionar la comunicación entre el server y un cliente serializando, transmitiendo y deserializando mensajes encapsulados en un tipo genérico `GenericMsg`. Consiste de una clase base `ProtocoloCommon` y las implementaciones especializadas según el tipo de mensaje que envía y que recibe cada parte: `ServerProtocol` y `ClientProtocol`.

La clase `GenericMsg` soporta polimorfismo y se basa en el patrón de diseño double dispatch donde cada mensaje que hereda de GenericMsg va a tener un método de serialización y deserialización específicos adaptados a su contenido. Cada mensaje tiene un header `uint8` único que lo hace identificable.
En GenericMsg también se definen los enumerativos, IDs únicos de mensajes, proyectiles, colores y estados varios que se usan a lo largo de todo el código.

Para el envío de un mensaje, el protocolo común delega la serialización del mensaje al tipo de mensaje apropiado, según el tipo que se haya recibido (`msg->accept_send(*this)`) . Y para recibir mensajes, se lee el header del mensaje, se lo identifica utilizando el mapa de handlers, se instancia ese tipo de mensaje con un contenido *dummy* y se delega a la función específica correspondiente al tipo de mensaje (`msg->accept_recv(*this)`).

El protocolo de server y client extienden la lógica del protocolo common para definir el manejo de mensajes especifico que le corresponde a cada lado.
Con `handle_recv` parsean el mensaje entrante, se deserializa la data del generic msg y se almacena en el tipo de mensaje específico, recibiendo exactamente los bytes que se esperan según el tipo de mensaje. Similarmente, para enviar un mensaje se serializa el mensaje con `handle_sent` transmitiendo el header y data específicos del mensaje.



Cliente

El cliente se conecta al servidor a partir de que establece su conexión a través de un socket que busca conectarse con el *hostname* y *port* del servidor levantado.
Cuando logra la conexión se instancia un objeto `Client` que va a recibir una referencia a las *protected queues* a las cuales de van a pushear/popear mensajes;

también lanzará sus hilos `Sender` y `Receiver` que enviarán y recibirán los mensajes entrantes del servidor que pasan a través del `ClientProtocol`.
La conexión se realiza mediante una interfaz gráfica (Menú principal) y mientras que la ventana del menú o del juego existan, el cliente también.

Interfaz gráfica

Menú principal

Main Window

El menú principal está realizado con el framework Qt. Cuando se ejecuta el cliente se instancia una aplicación de Qt `QApplication` y se levanta una `MainWindow` que va a gestionar varios widgets.

La ventana principal es la `MainWindow` que hereda de `QMainWindow`. Se encarga de gestionar las distintas pantallas dentro del menú a través de señales emitidas tras hacer clic en botones que emiten esas señales. Cada pantalla (*screen*) hereda de `QWidget` y cuando se quiere mostrar una ventana u otra se apilan o desapilan widgets en el `QStackedWidget` de la main window, solo una ventana es visible a la vez.

Cada pantalla se conecta a señales que permiten cambiar entre ellas. Se implementan animaciones para mejorar la experiencia visual (como el deslizamiento del fondo cuando se va de una screen a otra). También se gestiona la reproducción de música, permitiendo al usuario silenciarla. Las pantallas permiten funcionalidades como crear un juego o unirse a un lobby. Se permite salir del menú antes de iniciar cualquier partida, cuando se recibe una señal para salir de la aplicación (se clickeo en un botón de *quit*) se ejecutará `QApplication::quit()` que finaliza la aplicación y cierra todas las ventanas (en este caso una sola) de la interfaz gráfica, llamando a los destructores de los objetos gráficos.

Ventanas iniciales

Cuando se empieza a visualizar el menú la primera screen será `LogoScreen` que su único propósito es ser un recurso estético; muestra el logo del juego y para ir a la siguiente screen se debe presionar cualquier tecla. La próxima screen en donde el cliente se conecta al servidor `ConnectionScreen` ingresando el hostname y port. Cuando se hace clic en el botón de *connect* se intenta generar un socket con el input del usuario, si no logra catchea una excepción y muestra un aviso de que no

se pudo establecer una conexión. Si la conexión es exitosa se mostrará la pantalla `MainMenuScreen`.

Estas dos screens iniciales tienen un fondo con efecto *parallax* implementado con la clase `ParallaxBackground` que muestra una imagen en constante movimiento a una velocidad elegible. La fluidez de este efecto está dada por la constancia en la que se actualiza la pantalla (implementado con 30 ms) pero esto mismo implica un uso del CPU elevado (aproximadamente 40%), si la velocidad es 0, el uso del CPU disminuye considerablemente (aprox. 2%).

Main menu

Después de establecerse la conexión se muestra la screen `MainMenuScreen` con un efecto de fade-in/fade-out para una transición más suave.
En esta pantalla están los botones para salir del menú, crear un lobby o elegir un lobby. Al hacer clic en los dos últimos va a mostrar las pantallas correspondientes.

La clase `JoinLobbyScreen` es la pantalla que se va a mostrar cuando se hace click en el botón *join lobby*, el fondo se va a mostrar con un efecto de deslizamiento "hacia adelante". Cuando se emite la señal para ir a esta pantalla se dispara la directiva para pedirle al lobby la lista de lobbies disponibles para poder dibujarla en pantalla, cuando se hace clic en el botón de *refresh* se llama a la misma función. Para poder mostrar la lista de lobbies cuando se hace clic en el botón de join lobby sin tener que hacer clic en refresh se necesita de un cliente ya creado por lo que esa pantalla se instancia recién cuando ya se establece una conexión con el servidor.
Si hay lobbies creados en el servidor se van a mostrar y si no están llenos tendrán un botón *join* para unirse al lobby, si se clickea se va a mandar un mensaje al servidor para unirse al lobby y entrar a la sala. Se va a crear la pantalla `LobbyScreen` y se van a ver los jugadores unidos a la sala.
Si se quiere volver a la pantalla anterior se puede hacer click en el botón *back* y se va mostrar un efecto de deslizamiento "hacia atrás", en sentido contrario al que se mostró para mostrar la pantalla de join lobby screen.

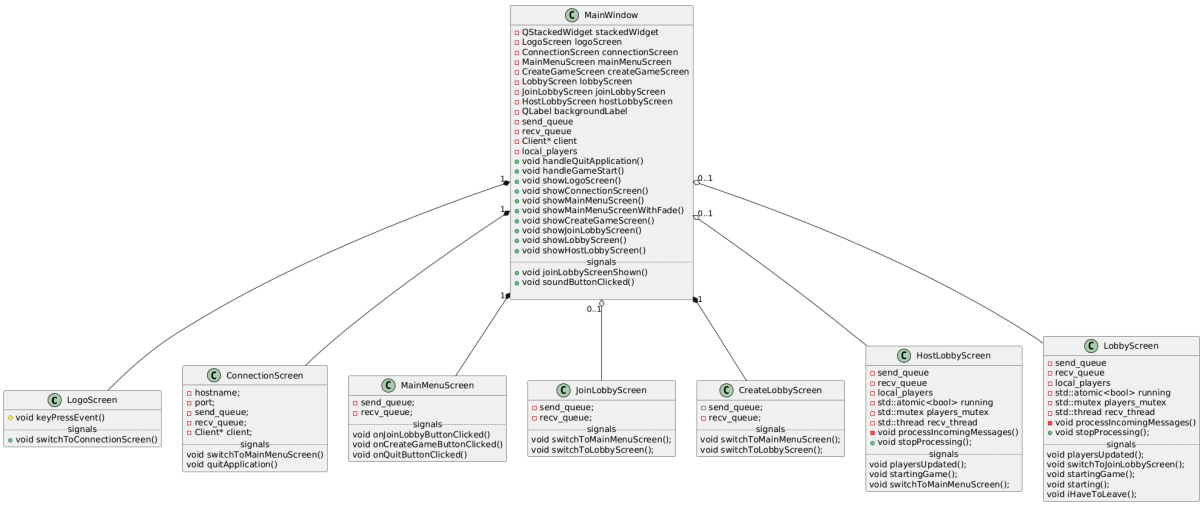
Si se quiere crear un lobby se puede hacer clic en el botón de *create lobby*, se va a elegir el nombre del lobby y la cantidad máxima de jugadores.

Pantallas de lobby

Quien crea una partida va a ver la pantalla `HostLobbyScreen` que es muy similar a LobbyScreen pero con los privilegios que tiene ser el *host* de la sala: añadir un jugador local y empezar la partida.

Ambas pantallas `HostLobbyScreen` y `LobbyScreen` reciben mensajes del servidor en un hilo aparte para poder mostrar recursivamente cuando se une un jugador, cambia su nombre o seteo su estado a *ready* para iniciar la partida. La lista de jugadores con sus características actuales es un recurso compartido por el hilo que recibe la lista (y la modifica) y por la función (que lee la lista) que dibuja en pantalla a los jugadores y a su estado, por lo que está protegida por un mutex.

Cuando hay el mínimo de jugadores (2) para iniciar la partida y todos los jugadores en la sala estan ready, el host puede hacer clic en *start game* va a enviar al servidor la nueva información de que una partida está empezando y cuando los jugadores (incluido el host) reciban esa información se va a enviar una señal a la main window para mostrar una pantalla con efecto fade-in indicando que la partida está empezando, además de que va a cerrar la ventana y como hay más de un jugador local indicando que la ventana se cerró porque existe un juego, se va a levantar la Ventana de Juego.



Ventana de Juego

Fue realizado utilizando librerías gráficas de SDL

Game

La función `Game::play` es el núcleo de la ejecución del juego. Esta se encarga de inicializar los recursos, procesar mensajes entrantes, manejar el renderizado de gráficos y mantener la sincronización de los fotogramas a través de un bucle principal.

Antes de entrar al loop principal se realiza una configuración del entorno.

Se recibe un mensaje con información del mapa (`SEND_MAP_MSG`) que describe el mapa del juego, incluyendo su diseño (matriz de tiles), dimensiones (filas y columnas) y el *theme* del mapa.
A partir de las dimensiones del mapa y la resolución de la pantalla, se calcula el tamaño de cada tile para asegurarse de que todo el mapa encaje correctamente en la ventana. Con esta información, se utiliza la clase `Map` para configurar el mapa visual, establecer el tema y dibujar el entorno inicial.

Para la inicialización de gráficos, audio y texto, se utilizan las siguientes bibliotecas de SDL:

- **SDL_image**: Manejo de imágenes para texturas y gráficos del juego. Se abstrae su uso con la clase `Image`.
- **SDL_mixer**: Reproducción de música y efectos de sonido.
- **SDL_ttf**: Renderizado de texto.

Si ocurre un error en alguna de estas inicializaciones, la función lanza una excepción y se asegura de liberar todos los recursos ya inicializados.

Main event loop

En la clase `Game` está el bucle principal donde ocurre la ejecución continua del juego donde se maneja el control de tiempo, sincronización de frames, procesamiento de mensajes, renderizado y la actualización de estados.

El juego opera a una tasa fija de 30 FPS. Para lograr esto se calcula el tiempo transcurrido desde el último frame y se determina cuantos frames deben procesarse. Si hay tiempo sobrante después de procesar un frame, el juego espera (`SDL_Delay`) para mantener la sincronización.

Durante cada iteración se revisa si hay mensajes para popear de la recv_queue. Los mensajes pueden ser:

- `UPDATED_PLAYER_INFO_MSG`: Actualiza la posición y estado de los jugadores (movimiento, dirección, acciones como *play dead*).
- `PICKUP_DROP_MSG`: Maneja la recolección o el uso de objetos en el mapa.
- `PROJECTILE_INFO_MSG`: Agrega proyectiles o armas al mapa, como balas o láseres.
- `SEND_MAP_MSG`: Actualiza el mapa completo con nuevos datos, como un cambio de nivel. Llega cada vez que empieza una ronda.
- `GAME_ENDED_MSG`: Indica el final del juego, lo que provoca la salida del bucle principal.

El renderizado limpia el contenido previo de la ventana con `win->clear()`, rellena el mapa con toda la información de la escena (`map.fill()`) y se muestra en pantalla. Además se verifican los estados específicos de los jugadores. Se registra el cambio de estado de un `Player` y se refleja en el juego.

Map

La clase `Map` organiza y gestiona todos los objetos y elementos visuales dentro del mapa, representa el entorno gráfico del juego. Administra los tiles del mapa, jugadores, armas, explosiones y otros elementos visibles.

Se configura el mapa actual cargando texturas desde archivos con la clase `Image`, se setea el tema visual (day o night) y establece tamaños para los tiles y objetos del mapa. Se renderiza en un flujo ordenado el fondo, tiles del mapa, ítems del juego, jugadores y explosiones. Dibuja todos los elementos en el orden correcto, comenzando con el fondo y terminando con los jugadores.

Convierte una matriz de datos (`std::vector<uint16_t>`) en tiles visuales.
Asigna tipos de tiles (piso, pared) y verifica si se pueden colocar en posiciones específicas.

El Map gestiona estos objetos: añade, actualiza y elimina elementos del mapa, como armas, cascos y explosiones. Administra los jugadores vivos y sus posiciones. Ajusta las posiciones de los objetos según los datos recibidos. Libera texturas y otros recursos al finalizar.

Además se maneja un zoom dinámico del mapa según la posición de los jugadores.

Zoom

El zoom del juego ajusta automáticamente el nivel de acercamiento en función de las posiciones de los jugadores y el tamaño del área visible. Este mecanismo asegura que los jugadores siempre estén visibles dentro de los límites de la pantalla, mientras se adapta fluidamente a cambios en el escenario. El sistema se divide en dos funciones clave:

- `animationZoom`:suaviza la transición del nivel de zoom actual (`currentZoom`) al objetivo (`targetZoom`), simulando un comportamiento fluido en 30 FPS.
- `adjustMapZoom`: ajusta el rectángulo visible del mapa (`SDL_Rect zoomRect`) según las posiciones de los jugadores y el nivel de zoom calculado. Este rectángulo se utiliza para renderizar el mapa con el zoom dinámico.

Manejo de eventos

La clase `EventHandler` es responsable de manejar las entradas del usuario, procesando eventos generados por el teclado y enviándolos a la cola de salida (`queueSend`). Estos eventos se traducen en mensajes que describen acciones del jugador, como moverse, disparar o saltar, y se comunican con otros componentes del juego.

El constructor de `EventHandler` configura un mapa (`key_accion_map`) que asocia combinaciones de eventos y teclas con funciones lambda. Cada función genera un mensaje de acción correspondiente a la combinación.

- Eventos de tecla presionada (`SDL_KEYDOWN`): Se generan mensajes `StartActionMsg` que indican el inicio de una acción.
- Eventos de tecla soltada (`SDL_KEYUP`): Se generan mensajes `StopActionMsg` que indican el fin de una acción.

Si hay un segundo jugador local, se asignan combinaciones adicionales para sus controles.

Funciona en un hilo separado, permitiendo que el manejo de eventos sea independiente del resto del juego y proporciona métodos para pausar (`block`) o reanudar (`unblock`) el procesamiento de eventos.

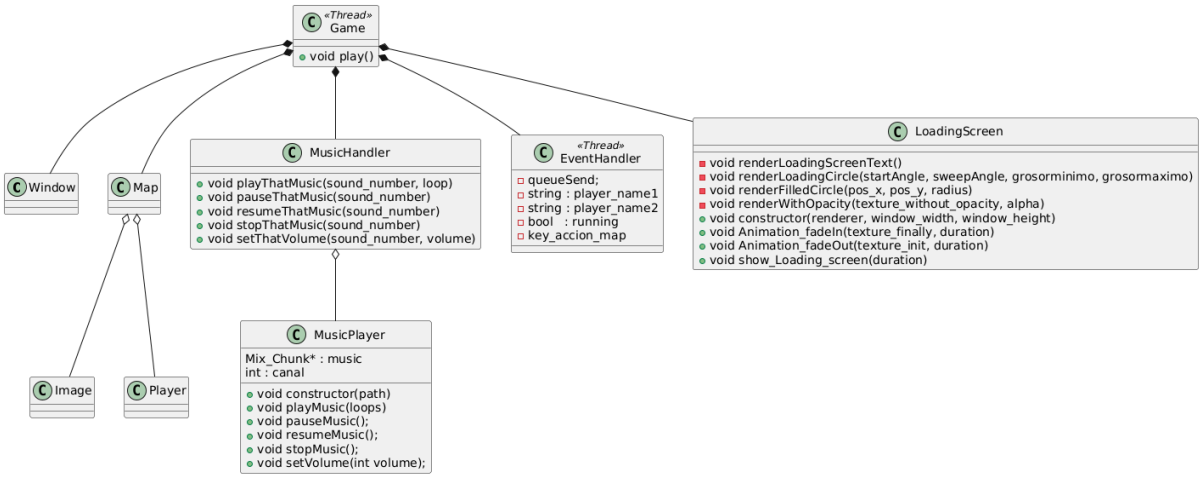
El bucle principal de eventos utiliza `SDL_WaitEvent` para esperar y procesar eventos como `SDL_KEYDOWN`, `SDL_KEYUP` y `SDL_QUIT` e ignora eventos redundantes o no relevantes utilizando el método `corroboraciones`. Busca la combinación de evento y tecla en `key_accion_map`, llama a la función asociada para generar un mensaje de acción, intenta encolar el mensaje generado en `queueSend` y si la cola está llena, el mensaje no se envía.
Si `is_blocked` está activo, el bucle se detiene temporalmente para evitar procesar eventos

Pantalla de carga y pantalla de puntos

La clase `LoadingScreen` implementa una pantalla de carga gráfica que combina texto, imágenes y animaciones para ofrecer una transición visual con efecto *fade in* y *fade out* entre que termina y empieza una ronda (cambio de mapa). Su diseño utiliza las bibliotecas SDL y SDL_ttf para manejar gráficos y texto, respectivamente.
También se implementó una pantalla final donde se muestra por un lado quien ganó el juego tras alcanzar los puntos necesarios antes que el resto de jugadores, y después se muestra la pantalla con los puntajes finales.

Música

La clase `MusicHandler` gestiona la reproducción, pausa, reanudación y control del volumen de la música del juego. Está diseñada como un contenedor que organiza múltiples pistas musicales, delegando las operaciones específicas a objetos de la clase `MusicPlayer`, es un controlador eficiente para gestionar la música en el juego. Utiliza la biblioteca `SDL_mixer` para manejar el audio.



Editor de niveles

El editor de niveles es una aplicación gráfica basada en Qt.
Se trata de una aplicación aparte del cliente y del servidor, independiente de ambos. Es un programa que a través de una interfaz gráfica permite crear un mapa y guardarlo al alcance del servidor, para luego poder jugar esos mapas.

Cuando se corre el ejecutable del editor se inicia una aplicación de Qt `QApplication` y se levanta una ventana `EditorMainWindow` que hereda de `QMainWindow` y que se encarga de gestionar dos pantallas que heredan de `QWidget`, a través del apilado de estas pantallas con un `QStackedWidget`.

Inicialmente se muestra la pantalla `SetInitialValuesScreen` donde se van a captar las elecciones del usuario: tamaño del mapa y *theme*. Cuando se haga clic en *start editing* se va a emitir una señal para cambiar a la pantalla de edición.

La pantalla donde se edita el nivel es `EditorScreen` que cuenta con una matriz del tamaño elegido previamente permite a los usuarios seleccionar y colocar elementos como baldosas (`tiles`), armas, jugadores, y armaduras sobre una cuadrícula, con opciones para pintar, arrastrar y borrar.

Zoom

La función `wheelEvent` permite al usuario ajustar el nivel de zoom utilizando la rueda del mouse. Cada vez que se detecta un movimiento en la rueda, se modifica el factor de escala (`scale`) en un incremento o decremento definido por `zoomFactor`. Si el usuario desplaza la rueda hacia arriba, el zoom se incrementa; si la rueda se desplaza hacia abajo, el zoom se reduce. La función garantiza que el factor de escala no caiga por debajo de 0.1, evitando un tamaño inapropiado. La llamada a `update()` asegura que la interfaz gráfica refleje el cambio de zoom inmediatamente.

Click

La función `mousePressEvent` reacciona cuando el usuario hace clic con el botón izquierdo del mouse. Dependiendo del estado actual del editor, esta función activa diferentes modos de interacción: modo pintura, coloca un nuevo tile en la posición del cursor y desactiva el modo de borrado; modo de borrado, elimina el tile en la posición actual, si ninguno de estos modos está activo, inicia el modo de arrastre, almacenando la posición del cursor para calcular desplazamientos posteriores.

Movimiento del mouse

Cuando se mueve el mouse, `mouseMoveEvent` se activa para manejar dos posibles escenarios. Si el editor está en modo de arrastre, la función calcula el desplazamiento del cursor desde su última posición y actualiza los valores de desplazamiento (`offsetX` y `offsetY`). Por otro lado, si el modo de pintura está activo, la función coloca tiles en las posiciones por las que pasa el cursor, permitiendo al usuario pintar de manera fluida. Después se llama a `update()` para que los cambios se reflejen en la interfaz.

Release del mouse

La función `mouseReleaseEvent` maneja el momento en que el usuario libera el botón izquierdo del mouse. Desactiva el modo de arrastre.

Inicio del Drag

La función `startDrag` se encarga de iniciar el proceso de arrastrar un tile desde un menú. Se activa al seleccionar una acción asociada a un tile, obteniendo la imagen correspondiente desde un mapa de datos y creando un objeto de arrastre (`QDrag`) con esa información. El tile se identifica mediante su nombre, almacenado en un objeto MIME, y se asocia una imagen representativa para mostrarla durante el arrastre.

Drag and Drop

`dragEnterEvent` y `dropEvent` trabajan en conjunto para implementar la funcionalidad de arrastrar y soltar elementos en el editor. La primera función valida que los datos arrastrados sean compatibles, verificando que contienen texto. Cuando el usuario suelta un elemento, `dropEvent` extrae el nombre del tile de los datos MIME, lo establece como el tile actual (`currentTile`) y lo coloca automáticamente en la posición del cursor. Activa el modo de pintura para permitir interacciones continuas con el nuevo tile.

Test mode

Se cuenta con un modo de testeo tanto para el servidor como para el cliente, orientado al testeo en una única máquina, de forma local. Además de un cheat mode manejado por el servidor.

Client

Del lado del cliente se puede ejecutar el cliente agregando el parámetro `test` (./client test), lanzará una partida local u online de máximo tres jugadores. Para ejecutar el cliente con este modo ya se debe haber levantado un servidor local en el puerto 8080 (se puede cambiar en client/main.cpp) porque lo que hace es:

1. Si no hay un lobby creado, crea uno.
2. Si hay uno creado, se une.

La partida siempre va a tener dos jugadores locales y si se une un jugador online, la partida será de tres jugadores.

Para jugar solamente de forma local con dos jugadores se corre en una consola `./client test` y se espera los 3 segundos que da el modo test para que se una un tercer jugador, entonces si no se une ningún jugador online, la partida va a ser solo de dos jugadores. Análogamente, si se quiere jugar con un jugador online, se debe ejecutar el mismo comando en otra consola antes de que pasen los 3 segundos y se podrá unir el cliente a ese lobby.

Este modo permite testear rápidamente la ejecución de una partida local u online sin pasar por toda la interfaz gráfica del menú, ni tener que crear un lobby ni jugadores, todo se inicializa con valores predeterminados.

Server testmode

El modo test del servidor permite visualizar la matriz que representa al juego y que se encarga de contener el estado actual de una partida. Ingresando un formato de comando específico permite simular movimientos tile a tile de cualquier jugador en la partida.
Para ejecutar este modo se debe agregar el parámetro test: `./server <port> test` y para empezar se debe ejecutar el cliente. Cuando se conecte el cliente, en la consola del servidor, se mostrará la matriz del juego. Cada jugador se va a poder controlar con este tipo de comando: input de tres dígitos:

1. Primer dígito:
    - Si el movimiento empieza, es `s`
    - Si el movimiento termina, es `x`
2. Segundo dígito. Para elegir el tipo de movimiento:
    - `a` = izquierda
    - `d` = derecha
    - `w` = apuntar arriba
    - `x` = disparar
    - `t` = tirar arma
    - `j` = saltar
3. Tercer dígito. Para elegir el jugador que va a ejecutar la acción:
    - Número de jugador que va a ejecutar la acción.

Ejemplificación:

`sa1`	Hace que el jugador “1” empiece a ir a la izquierda
`xa1`	Hace que el jugador “1” deje de ir a la izquierda

Con este modo se va a poder visualizar a detalle el comportamiento de la escena con cada movimiento y cambio que realice el comportamiento de los jugadores.

Server cheatmode

El modo cheat del servidor va a estar activado solamente si el servidor fue inicializado con el argumento `cheat`, es decir `./server <port> cheat` y no se puede usar en conjunto con el modo *test* del servidor.

En la ventana de juego del cliente se debe oprimir la tecla `c`, se envia ese input al servidor a través del socket, y cuando lo lee bloquea el turno hasta que se ingrese el comando por entrada estándar, entonces: el cliente va a dejar de poder moverse y entonces se va a permitir escribir por entrada estándar un comando con el formato

```
spawn {player_name} {weapon_id}
```

donde player_name es el nombre de algún jugador en la partida y weapon_id el número que representa a algún arma. Ingresando ese comando se le va a equipar el arma al jugador elegido. Si no se ingresa un comando correcto la partida sigue normalmente.

Este modo está orientado a un testeo local donde se tiene un control y seguimiento de los datos ingresados en el lado del cliente ya que se deben conocer los nombres de los jugadores, además de que es necesario saber los ids con los que están identificadas las armas (ubicados en `generic_msg.cpp`).