FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

Taller de Programación I (TA 045)
Cátedra Veiga

Duck Game

Trabajo Práctico Grupal - Documentación Técnica

Grupo 14

Integrantes:
- Nicolás Ezequiel Grüner - 110835
- Victoria Fernandez Delgado - 110545
- Bautista Boeri - 110898
- Franco Daniel Capra - 99642

Índice

Arquitectura

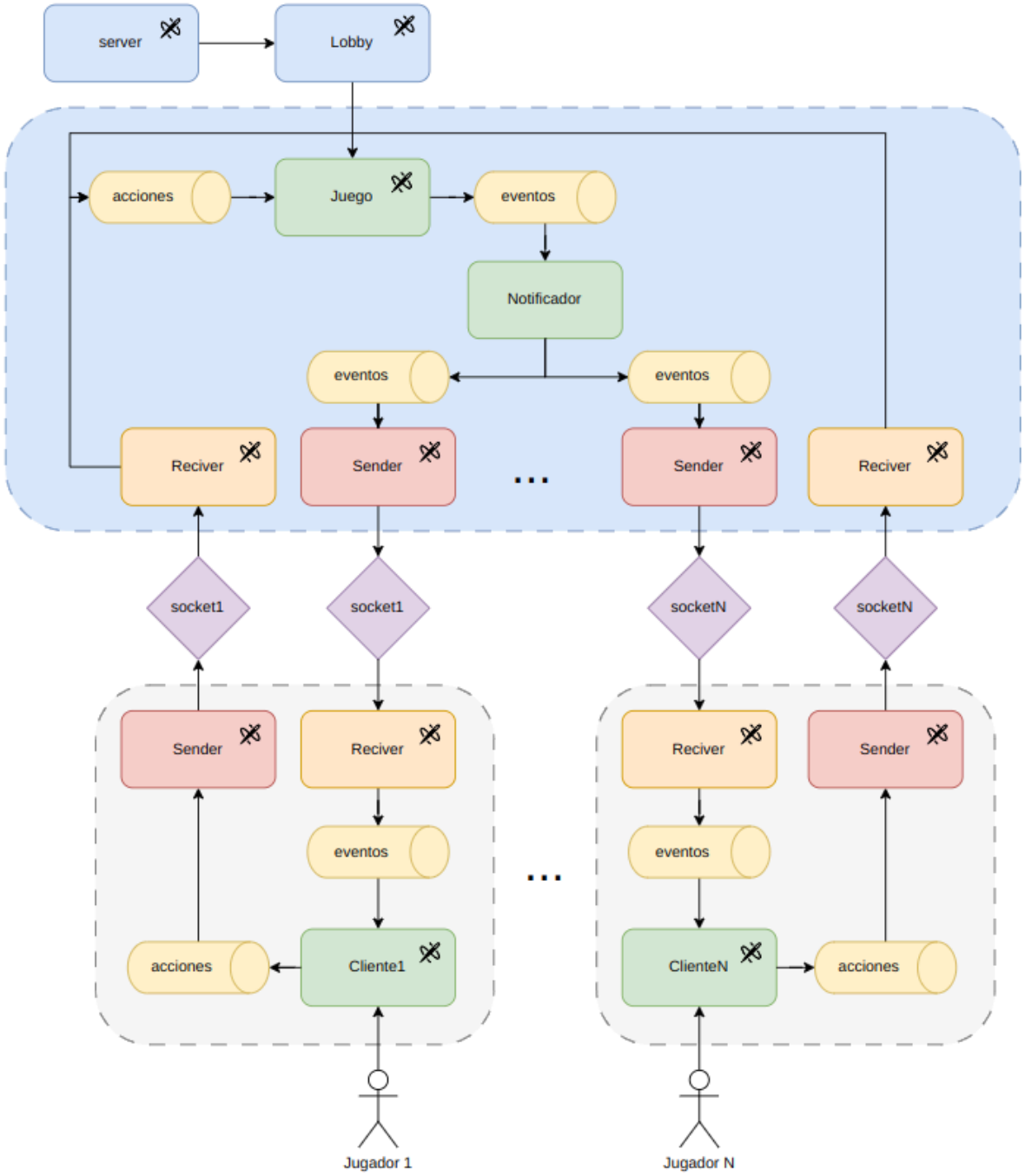

Figura 1: Arquitectura de Cliente - Servidor.

Protocolo

Se establece el siguiente protocolo de comunicación entre cliente y servidor:
[Especificación protocolo]

Servidor

Diagrama de clases

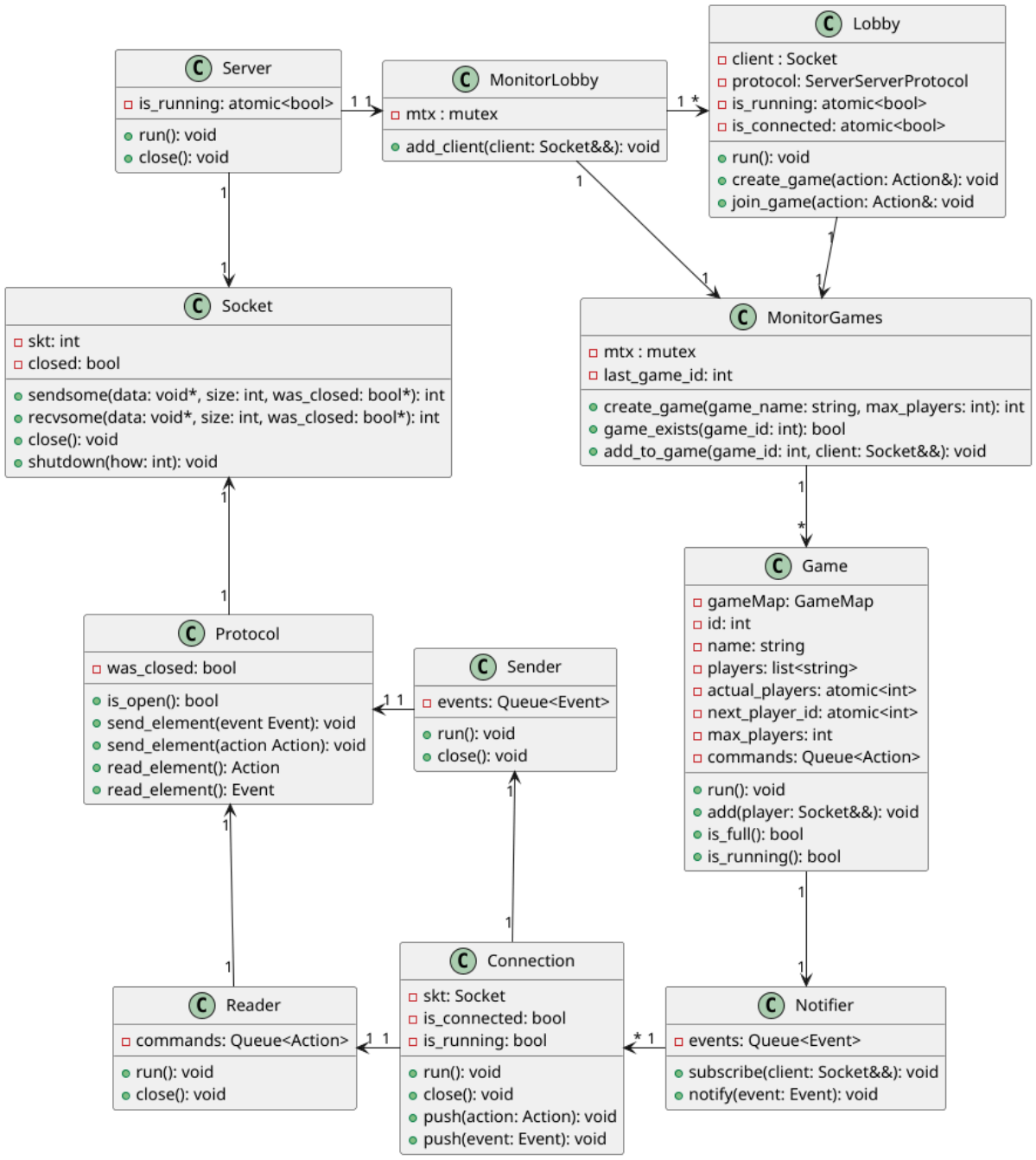

Figura 2: Diagrama de clases del servidor.

Main loop

El main loop del servidor se dedica a aceptar nuevas conexiones de sockets.

Lobby

El servidor delega al lobby la responsabilidad de administrar las conexiones de los clientes a un juego. Este juego puede ser uno ya existente, en cuyo caso valida que el cliente pueda unirse, o uno a crear en el cual crea y añade a los jugadores.
Para añadir a los jugadores y poder ser accedido desde múltiples threads (uno por cliente), el lobby tiene un monitor que se encarga de asegurar que todas las acciones a realizar no tengan race conditions.

Game

El juego se encarga de recibir acciones y enviar eventos.
Las acciones son producidas por los usuarios y llegan mediante la cola de acciones.
Los eventos son generados por el juego y enviados a la cola de eventos para ser enviados por el protocolo a los usuarios.
En cada loop, el juego procesa acciones, actualiza el juego e informa los eventos que se hayan producido (Broadcast, Score o GameOver).
El juego hace una pausa de 20 ms por loop para descansa

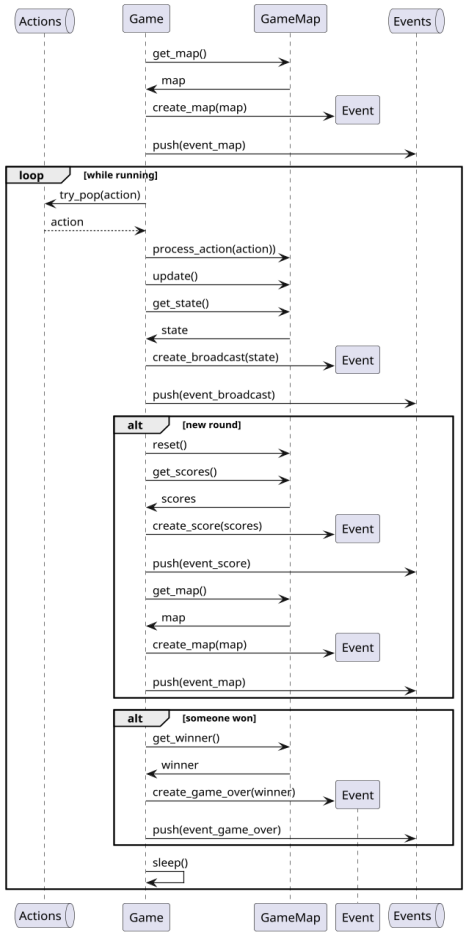

Figura 3: Diagrama de secuencia del juego.

Lógica del juego

El diseño del juego sigue una lógica que se centraliza en la clase GameMap, que se encarga de administrar el entorno, las entidades y la interacción entre ellas. Todas las clases, además de definir comportamientos para el juego, crean un DTO específico para poder enviar la información de cómo se actualiza el juego en cada frame. A continuación se describe cómo se organiza la lógica del juego con las distintas clases principales. Se presenta también, el siguiente diagrama que representa las clases principales de la lógica del juego. Por simplicidad, la clase weapon no se detalla en este diagrama ya que se hace en la sección de Weapon.

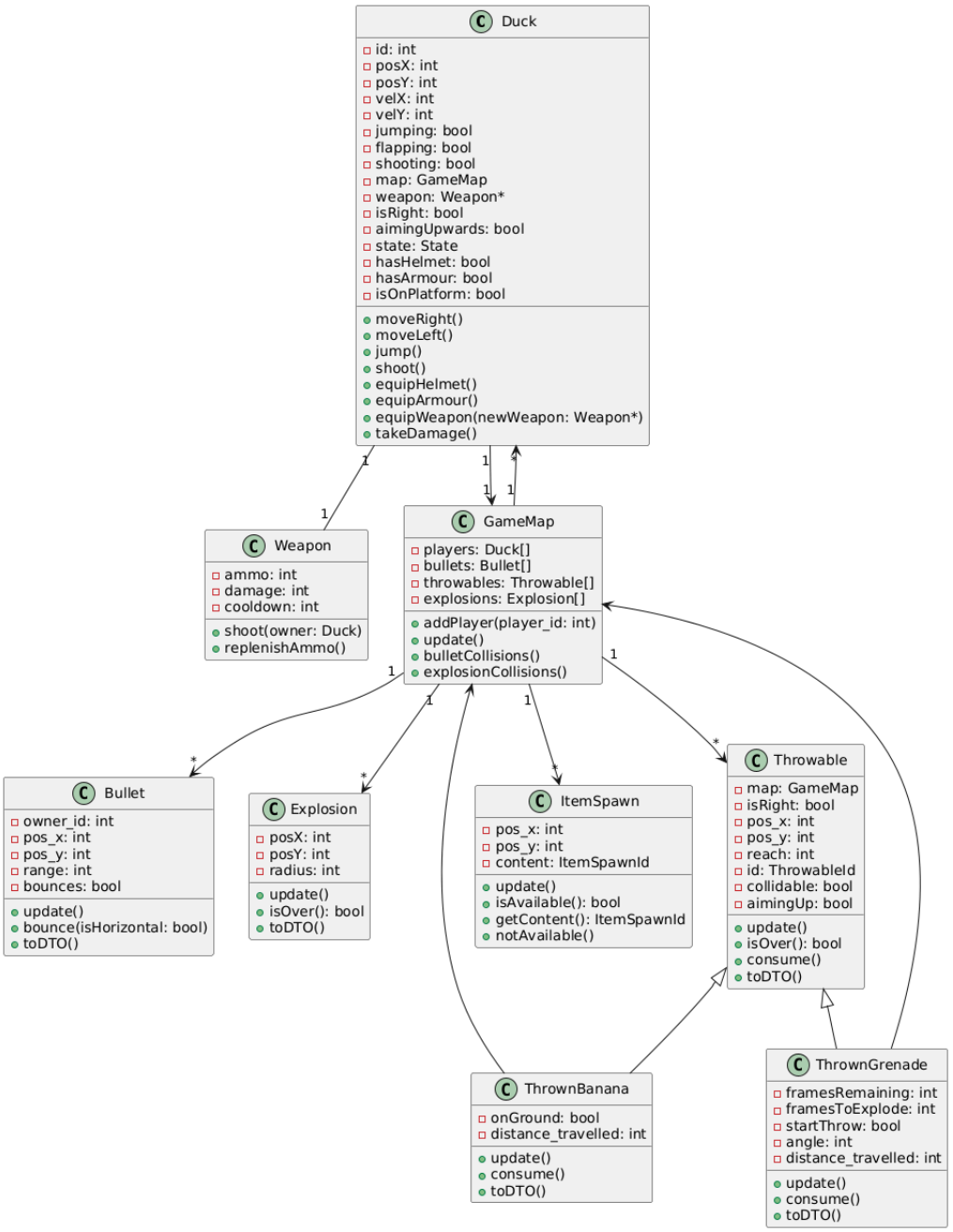

Figura 4: Diagrama de clases de lógica.

Game map

Se utiliza la clase Game map para organizar todas las clases principales del juego y generar un entorno en el cual se desarrolla el juego e interactúan las distintas entidades. Esta mantiene un vector de jugadores, balas, itemspawns, cajas y explosiones. También se administra la creación y destrucción de dichas entidades .

Las colisiones son gestionadas utilizando un sistema conocido como AABB (Axis-Aligned Bounding Box), implementado en la clase HitBox. Cada entidad cuenta con una "caja" delimitadora que define su espacio en el entorno del juego. El método isColliding detecta si hay una superposición entre las cajas de dos entidades en cualquier eje. Este sistema permite identificar de forma eficiente interacciones como disparos acertados, recolección de ítems, y choques entre jugadores.

También se simula el entorno mediante el mapa para poder saber las posiciones de las plataformas, los spawns de los patos, las cajas y los spawn items.

Duck

La clase Duck es la principal encargada de los jugadores y simular los movimientos y la física del pato. Se encarga de calcular y actualizar las posiciones del pato basándose en el input del jugador y las leyes de la física que son simuladas (como la gravedad y rebotes). Es también responsable de detectar colisiones entre el pato y las plataformas del mapa, usando información proporcionada por GameMap. Esto permite determinar con precisión cuándo debe pararse en una plataforma y cuál es la posición de esta o caerse al vacío.

Weapon

La clase Weapon es la clase abstracta base que define el comportamiento de las armas. Cada arma hereda de weapon e implementa shoot de la manera en la que debe para así lograr que cada una tenga su propio comportamiento. Esta se encarga de crear Bullets y Throwables que son directamente agregados al GameMap para poder ser tenidos en cuenta en el entorno general del juego. Se presenta un diagrama de como funciona la herencia de las armas.

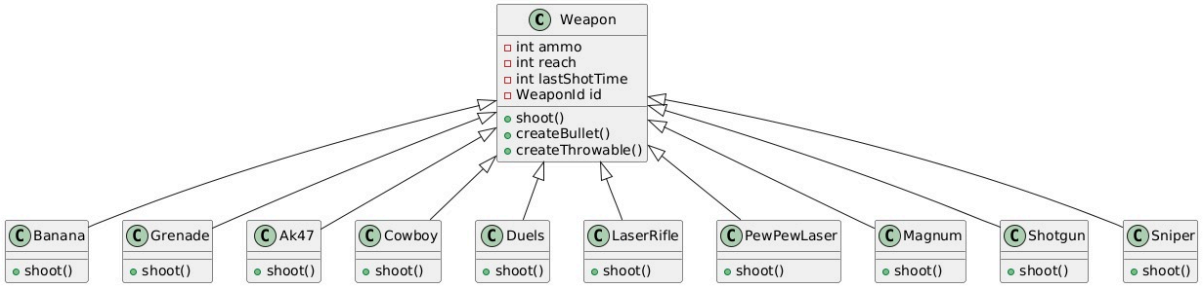

Figura 5: Diagrama de clases de Weapon.

Throwable

La clase Throwable es una clase base abstracta que sirve para definir el comportamiento de las Granadas y las Bananas (los objetos que pueden ser lanzados por el jugador). Es responsable de la posición en el mapa del objeto. Se utilizan referencias a GameMap para poder detectar las colisiones con estructuras y los elementos del entorno.

Las clases ThrownBanana y ThrownGrenade heredan de Throawble, encargándose de definir un comportamiento específico para cada una de ellas.

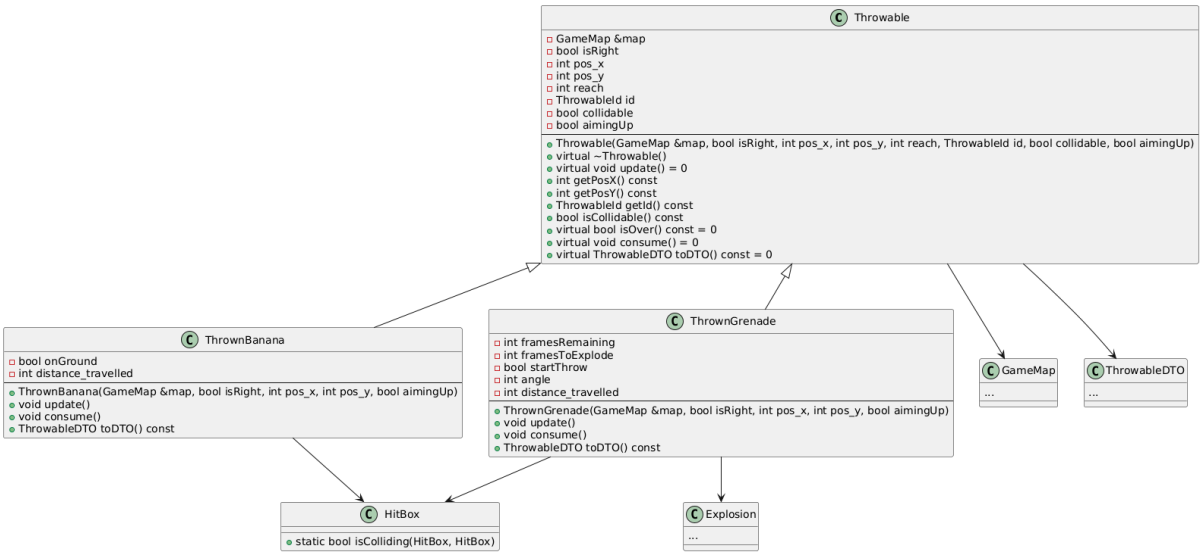

Figura 6: Diagrama de clase de Throwable..

Bullet

La clase bullet se encarga de actualizar la posición de cada bala basándose en el ángulo de disparo, la posición del pato y la distancia recorrida por la bala. También es la encargada de manejar las colisiones de las balas con las plataformas en el caso de que estas puedan rebotar. Esto se hace actualizando el ángulo según la inclinación que debería tener tras el rebote.

Box

La clase Box contiene información sobre la posición de cada caja y genera el contenido aleatorio de ellas. Además tiene una “vida” que define cuántas veces hay que dispararle para que se rompa. Esta clase se encarga de actualizar esta vida y romperse en caso de que sea 0.

ItemSpawns

La clase ItemSpawns contiene las posiciones de los spawns que puede haber en el mapa y se encarga de generar un tiempo de reaparición aleatorio para que una vez después de ser agarrados por un pato estos puedan volver a aparecer en el juego.

Sistema de rondas

El juego verificará en cada iteración si la partida finalizó o si es necesario mostrar los puntajes de cada jugador. Cada jugador en la partida contendrá el número de partidas ganadas. Al final de cada actualización en cada iteración, el juego verificará si hay un solo pato vivo. En caso de que sea así, se le otorgará una victoria a dicho jugador y se pasará a una nueva ronda.

Cada 5 rondas, el juego obtendrá el número de victorias de cada jugador y las enviará al cliente para que pueda mostrar los puntajes en pantalla.
Si algún jugador tiene más victorias que el resto y este supera las 10 victorias, significa que ganó. Una vez que ocurra esto, se le enviará un evento a todos los clientes, los cuales mostrarán una pantalla del ganador, finalizando la partida para todos.

Cheats

Dentro del juego, se designaron teclas especiales que permiten realizar trampas para probar el funcionamiento del juego. Dichas acciones se encuentran detalladas en el manual de usuario.

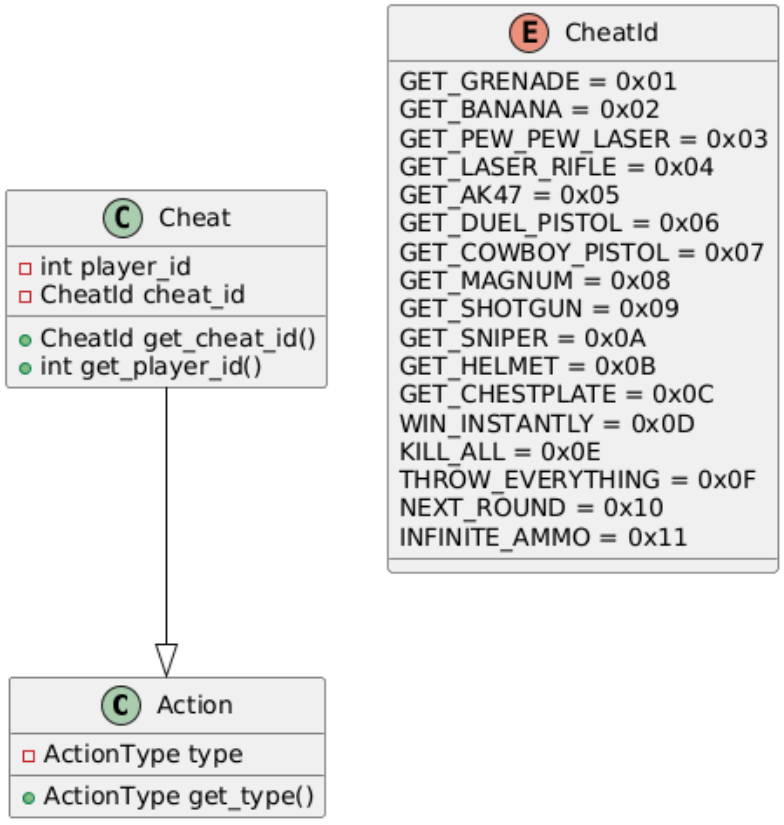

Figura 7: Diagrama de los cheats

La clase Cheat es tratada como una acción, por lo cual son tratados al igual que cualquier otra acción del pato. Mediante el id del cheat definido por el enum, el servidor puede determinar que cheat quiso realizar el jugador y llama la función correspondiente que lleva a cabo el procesamiento del cheat en el juego.

Cliente

Manejo de acciones

Dentro del cliente, se procesan y manejan las acciones realizadas mediante input de teclado de parte de los jugadores (en caso de estar jugando de a dos) o de un solo jugador. El encargado de hacer esto es la clase ActionHandler, la cual poppea de una cola de SDL eventos como presionar o soltar una tecla.

Al desencolar dicho evento, se puede determinar que tecla fue pulsada y, a partir de esto, qué acción se quiso realizar. Luego, a través del cliente, se enviará la acción correspondiente que contiene toda la información necesaria para que el servidor actualice el estado del juego correctamente.

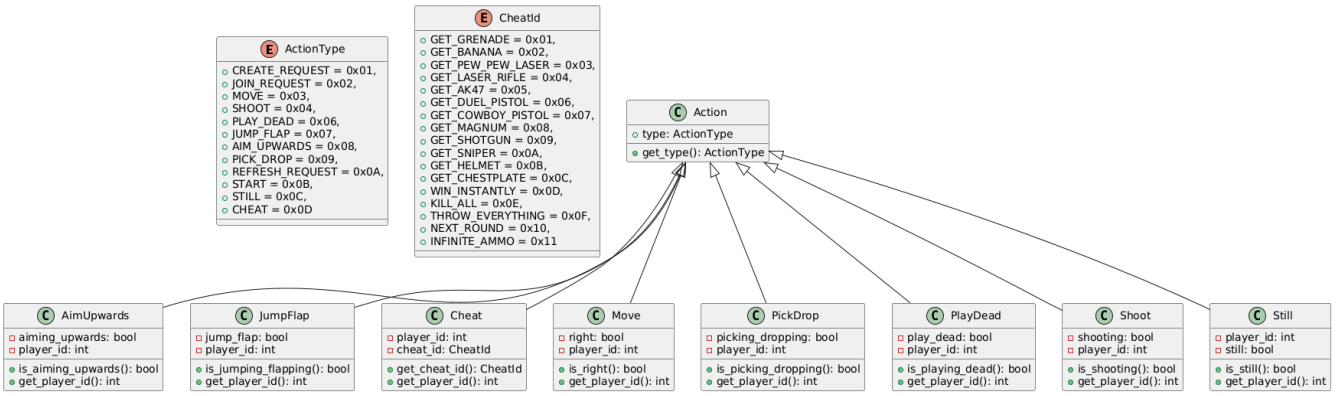

Figura 8: Diagrama de clases de las acciones.

La clase acción es base para el resto de las acciones posibles. Cada una de estas acciones tendrá un ActionType, el cual está definido en un enum. De esta manera, cuando el servidor reciba una acción, podrá obtener el tipo de acción y llamar a las primitivas correspondientes del tipo de acción en específico para acceder a los datos.

Para la acción Cheat, fue necesario crear otro enum que represente qué tipo de cheat se quiso utilizar.

Interfaz gráfica

Estructura general

Figura 9: Estructura de Game.

La clase game es la encargada de ejecutar el mainloop y orquestar el renderizado del programa. Existe una instancia de esta clase por cada cliente conectado al juego.
Esta tiene:

- Una referencia a un cliente, para poder brindarselo al ActionHandler, el cual se encargará de las acciones realizadas por los jugadores y llamará al cliente para poder enviarlas al servidor.
- Clases necesarias para el renderizado y mostrar por pantalla a través de la biblioteca de SDL, como el renderizador, la ventana en sí, SDL y la parte de SDL que se encarga del renderizado de las fuentes.

- Una instancia de la cámara, la cual luego se explicará su funcionamiento en mayor detalle.
- El mixer, encargado de reproducir la música de fondo y la música de victoria.
- Un vector de patos, el cual nos permitirá tener acceso a los distintos patos de cada jugador para actualizarlos y renderizarlos en cada iteración.
- Listas de balas, explosiones, cajas, ítems que spawnean y arrojables. Todos estos serán actualizados en cada iteración con la información que se recibe de parte del server y la lógica del juego.
- Las clases “Draw”, las cuales tendrán las texturas correspondientes a lo que se desea dibujar y serán las responsables de renderizar los distintos elementos en la pantalla.
- Dos booleanos; stop y pause que indican si terminó el juego o si se hizo una pausa.

Main loop

Buscaremos mantener un ritmo constante, por lo que estableceremos un rate el cual deseamos mantener. En este caso, será de 60 FPS. En cada iteración, actualizaremos el estado del juego a través de los eventos recibidos por parte del server. Dentro de los eventos posibles, tenemos:

- Broadcast, contiene toda la información necesaria para actualizar los patos, sus armas, cascos, armaduras, cajas en el mapa, balas, explosiones, arrojables e items que aparecen. Este es el evento que se recibirá la mayor parte del tiempo
- MapLoad, se recibe cada vez que se inicia una nueva ronda y es necesario cargar otro mapa.
- Scores, se recibe cada vez que es necesario mostrar los puntajes. Durante el renderizado de estos, el juego no se muestra en pantalla.
- GameOver, se recibe al haber un ganador del juego. Se muestra una pantalla de victoria con el nombre del jugador ganador y finaliza el programa para todos los clientes.

Luego de actualizar el estado del juego, la clase ActionHandler se encargará de procesar el input de los jugadores y asegurar su envío al servidor. Finalmente, con el estado de juego actualizado, se llama a renderizar todo lo presente en dicho estado. Una vez hecho esto, tomamos el tiempo que hemos tardado en realizar todo esto en la iteración actual y determinamos si estamos atrasados de acuerdo a nuestro rate o adelantados. Si estamos atrasados, seguiremos trabajando para ponernos al día con el rate establecido. En caso contrario, descansaremos lo que nos sobró de tiempo hasta iniciar la próxima iteración.

Duck

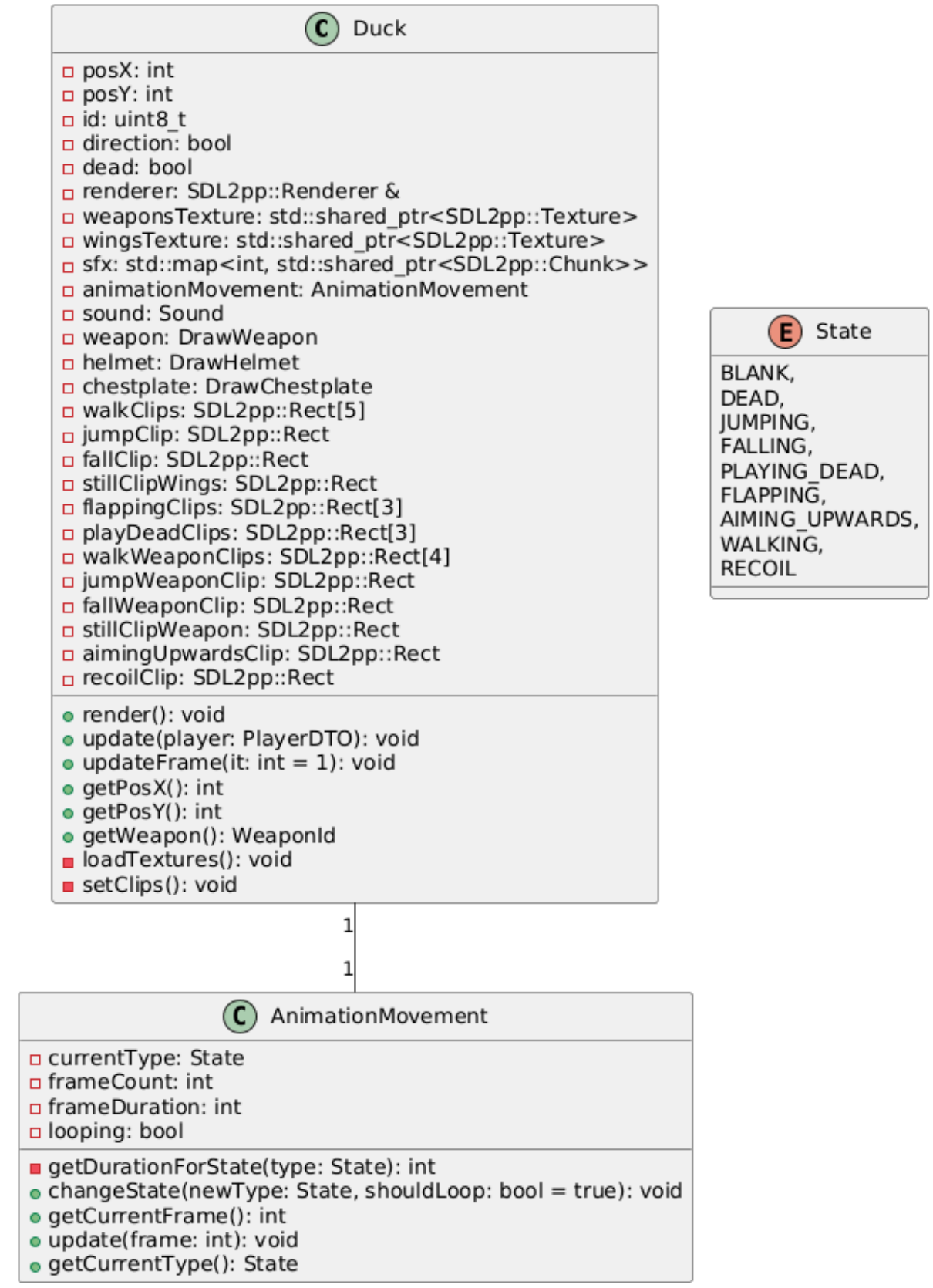

Figura 10: Diagrama de clases de Duck.

La clase pato contendrá la información necesaria para actualizar a cada jugador en cada iteración y además renderizar. Tendrá las texturas necesarias para dibujar el pato realizando cada acción posible, así como también una clase AnimationMovement, que le permitirá animar los movimientos del pato para que no sean tan rígidos. Esta le indicará el tipo de movimiento que el pato está realizando y también que frame de la animación del pato debe renderizar, para lograr la animación. Es importante notar que esta clase se encargará de cada animación del pato, las cuales cuentan con una cantidad distinta de frames e incluso algunas animaciones se repiten en loop.

Además contiene tres clases Draw, las cuales se encargan de dibujar el arma actual, el casco y la armadura del pato.
También cuenta con una clase Sound, la cual es la encargada de reproducir los efectos de sonido de los saltos del pato. Más adelante se explicará en mayor detalle dicha clase.

Las funciones más importantes de esta clase son update y render. En cada iteración donde se actualice el estado del juego, cada pato recibirá el último estado enviado desde el servidor, con el cual se actualizará para luego ser renderizado correctamente, mostrando su estado más reciente en el juego.

El DTO Player contiene toda la información necesaria para actualizar el pato con su estado más reciente en cada iteración. A continuación, un diagrama del mismo:

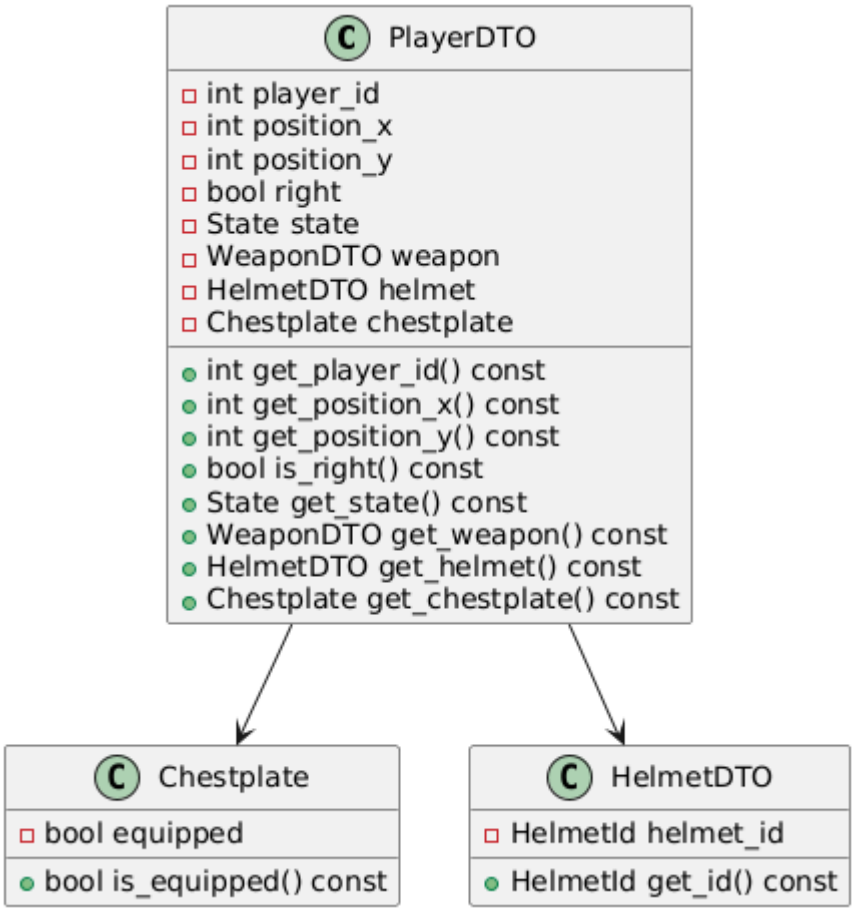

Figura 11: Diagrama de PlayerDTO.

Clases draw

Por una cuestión de simplicidad, a continuación se mostrará a modo de ejemplo, el diagrama de la clase DrawWeapon. Todas las clases draw presentan una lógica similar con algunas diferencias.

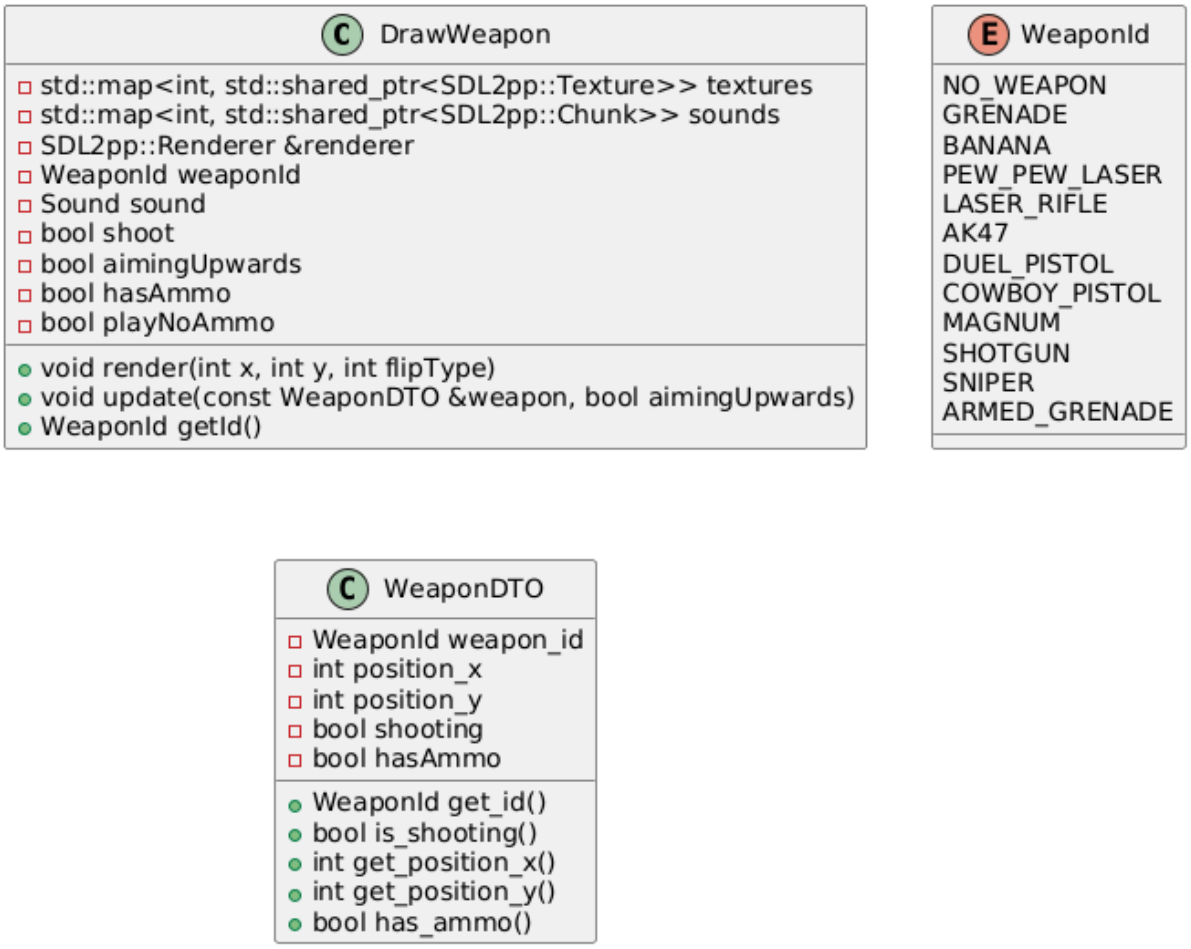

Figura 12: Diagrama de ejemplo de clases “Draw”.

Como se mencionó anteriormente, las clases Draw operan con una lógica muy similar. Reciben un DTO que contiene toda la información necesaria para actualizar o renderizar el objeto deseado y luego se realiza la renderización correspondiente. Todas estas clases, al ser instanciadas, cargan las texturas y sonidos necesarios para dicho objeto.

La principal diferencia entre los Draw, es que DrawWeapon, DrawHelmet y DrawChestplate son parte de la clase Duck. Esto significa que tendremos tantas clases Draw de las previamente mencionadas como jugadores en el juego. En cambio, solo tendremos una clase Draw de las otras no mencionadas. Estas no tendrán un registro de su estado anterior, simplemente recibirán la información necesaria para renderizar el objeto. Tendremos un DrawMap, un DrawCrate, etc por cada cliente gráfico.

Sonido

Múltiples clases de tipo Draw y la clase Duck tienen una clase Sound, la cual se encarga de reproducir los sonidos:

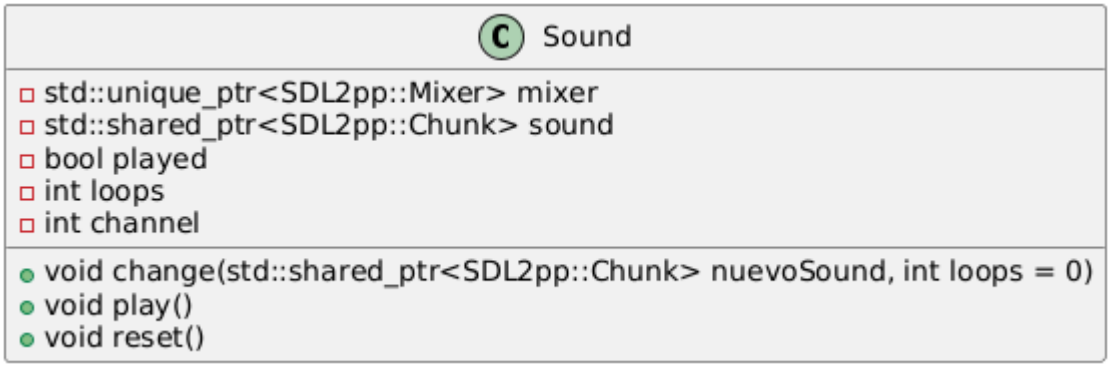

Figura 13: Clase Sound.

Dicha clase contiene un mixer, el cual será el encargado de reproducir el sonido, un sonido el cual será el archivo cargado en memoria que será reproducido, un booleano que indica si el sonido ha sido reproducido o no, una cantidad de loops y un channel. Este último es un canal por el cual se reproducirá el sonido.

La primitiva change nos permite cambiar de un sonido a otro, restableciendo los parámetros que indicaban la reproducción del anterior sonido. Al llamar a play, se buscará el siguiente canal disponible para reproducir el sonido, y este se reproducirá con un volumen establecido. Si este no está pensado para reproducir en loop y se llama otra vez a play, el sonido no se reproducirá.

Por último, la primitiva reset nos permite restablecer los parámetros de reproducción del sonido actual para volver a reproducirlo nuevamente.

Cámara

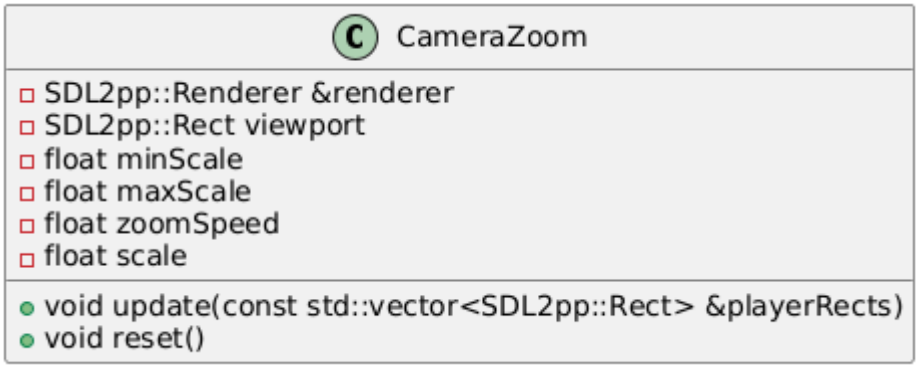

Figura 14: Clase CameraZoom.

La función de esta cámara es hacer zoom cuando los patos vivos se encuentren cercanos entre sí, pero siempre asegurándose que todos sean mostrados en pantalla en todo momento.

Para eso, la cámara se actualizará con la posición de todos los jugadores vivos. Tomará la posición de cada uno y buscará un punto medio entre todos. Luego, analizará qué jugador está más lejos, de manera de adaptar el zoom para que incluso ese jugador se muestre en pantalla. Finalmente, actualiza la escala con el valor de zoom correspondiente y el viewport, el cual encuadra donde se dibujará en la pantalla. Siempre se verificará que el viewport no salga de los límites establecidos de la resolución de la ventana.

Además, la cámara cuenta con parámetros que permiten adaptar la funcionalidad de la misma. minScale establece una escala mínima en la cual se mostrará el juego. En este caso, se busca un minScale de 1, lo que quiere decir que mostrará todo el mapa como mucho. Análogamente, tenemos maxScale, que establecerá una escala máxima, simbolizando el zoom máximo que podemos aplicar. En este caso, será 3. Por último, el zoomSpeed representa qué tan rápido hace zoom / se aleja la cámara.

Lobby Gráfico

Estructura del Lobby

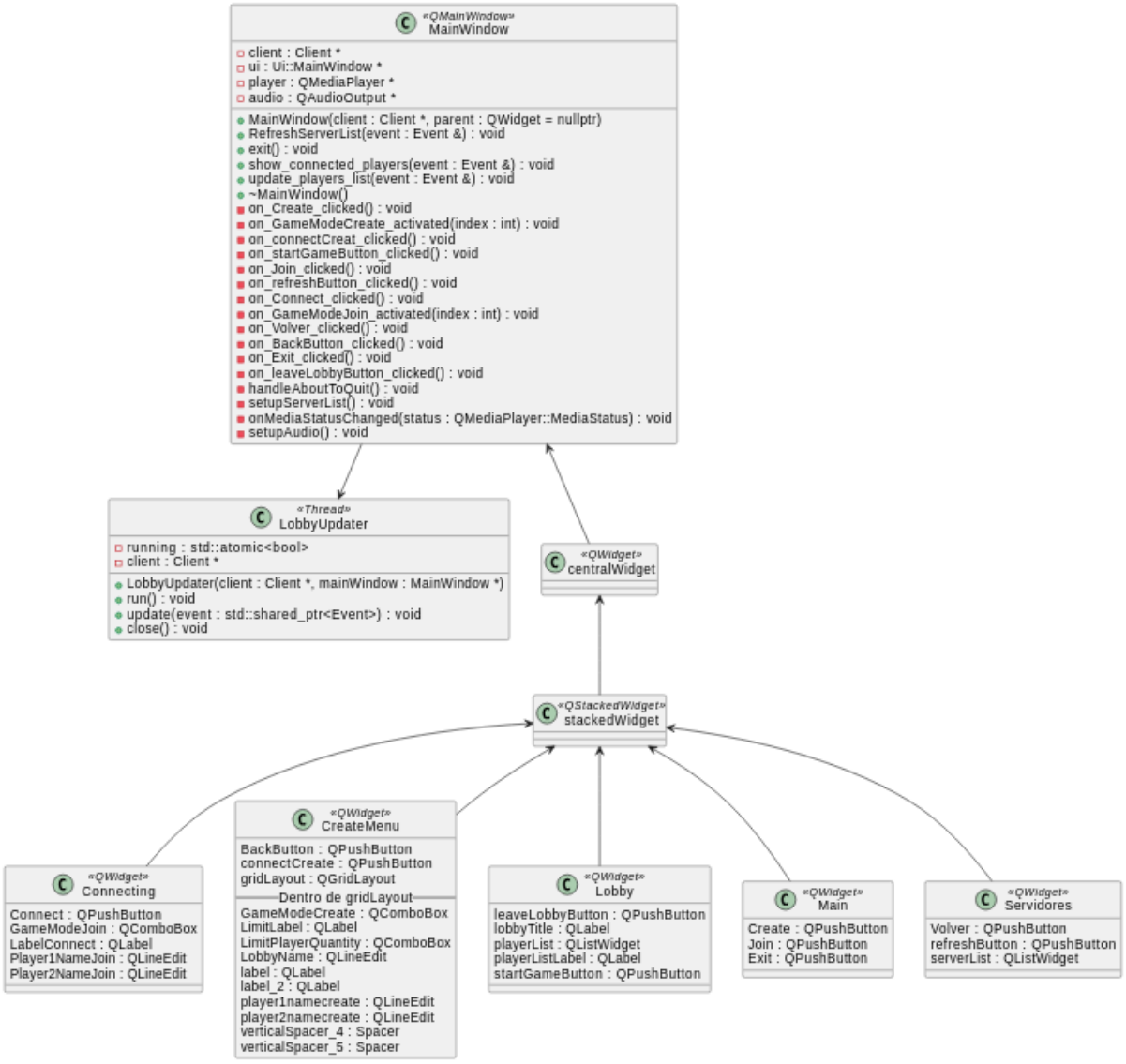

Figura 15: Diagrama de clases de la estructura del lobby.

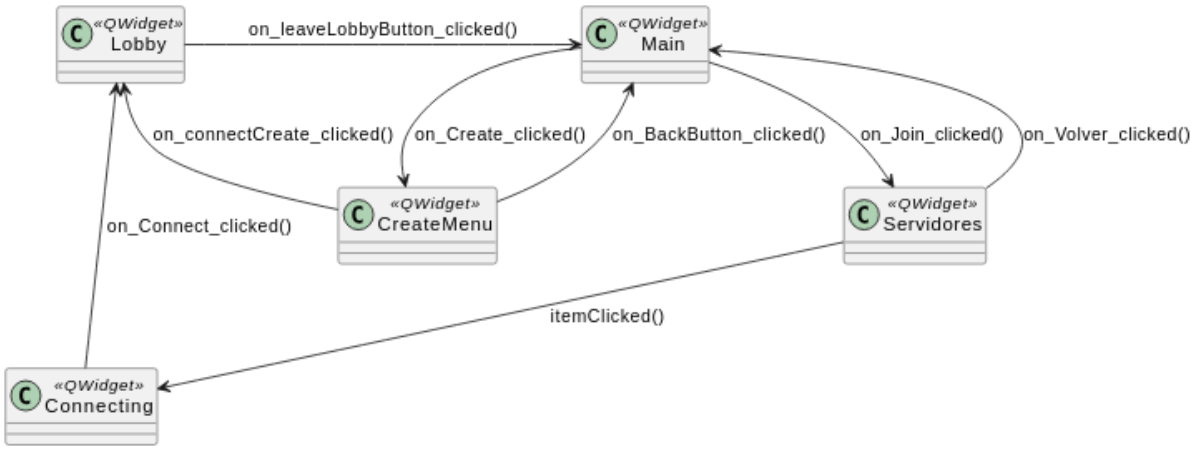

Figura 16: Diagrama de pasajes de widgets.

La estructura del lobby está basada en un QStackedWidget, esta es una pila de distintos widget. Cada una de estas widget representa otra parte del menú. Se pasa de un widget a otro, modificando el index actual del QStackedWidget.

Qué hacer al iniciar al lobby

Una vez iniciado el programa vamos a tener un conjunto de botones, en este caso por utilizar QT, habrán QPushButton. Uno te llevará a otro widget para crear una partida y otro botón te llevará a la lista de partidas.

Crear una partida

Dentro del creador de partidas habrán varios QLabel indicando que se debe completar en el QLineEdit al costado del label. También va a haber 2 QComboBox, que te dejaran elegir el número de jugadores locales como el límite de jugadores totales. Una vez se complete toda esta información, te llevará a un lobby. Al hacer click se le enviará al servidor la solicitud para crear una partida a través de la clase LobbyUpdater que utiliza el protocolo.
Una vez hecho todo esto, estaremos en el Widget Lobby donde , como administradores de la partida, podremos iniciar la partida. Para esto se debe tener al menos 2 usuarios conectados o que el administrador tenga 2 jugadores locales. El LobbyUpdater estará esperando que el servidor le notifique a cada cliente si alguien más se unió al lobby actual. En este widget, habrá algo parecido a la lista de servidores pues hay un QListWidget que contiene QListItem pero que esta vez no hacen nada si se le hace click.

Unirse a una partida

En el menú de lista de partidas, habrá un QListWidget. Este contiene QListItem, los cuales indican el nombre de la partida como sus jugadores. En este widget se encuentra un boton

que permite hacer “Refresh” a la listas de jugadores actuales, para hacer esto se le envía una señal al servidor para obtener las partidas disponibles y se las enviará al cliente. Si se le hace click cambiará el widget actual al WIdget Connecting, donde se deben completar más datos, como la cantidad de jugadores locales y sus respectivos nombres. Una vez se hizo esto y se apretó el botón correspondiente, se pasará al Lobby donde indicará el nombre de cada jugador como su color. En este caso, como se está uniendo a la partida y no se está creando, no se podrá iniciar el servidor sino que se esperará la señal enviada por el servidor al cliente de que la partida comenzó o se unen más jugadores.

Editor

Estructura del editor

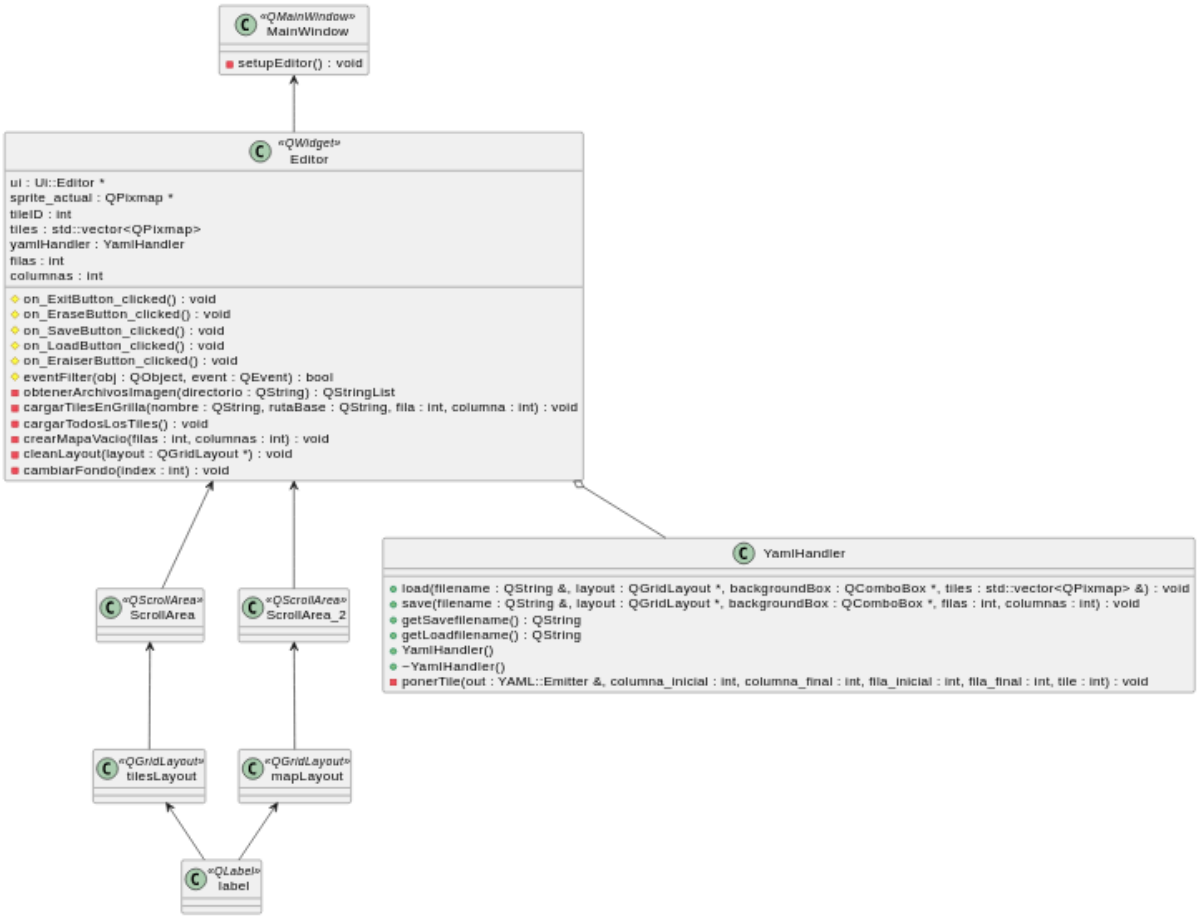

Figura 17: Estructura del editor.

La clase Editor es un QWidget la cual es contenida por una QMainWIndow. A su vez el editor contendrá unos cuantos botones para hacer distintas cosas. También contiene 2 QScrollArea distintos, donde cada uno contendrá una QGridLayout de muchos QLabel.

Preparación de la grilla de tiles disponibles

Al iniciar el editor se llama a un método del editor llamado cargarTilesEnGrilla, el cual separa los sprites de las tiles por cada tile. Se utilizará QPixMap, una clase de QT que contiene el contenido de la imagen en si, estos QPixMap pueden colocarse en las QLabel correspondientes. Se colocaran todos los tiles en un vector de QPixMap y también se colocara en la grilla. Una vez que se procesaron todos los tiles, se colocaron los ítems interactuables del juego así como el spawn de los patos.

Colocación de un tile/interactuable en el mapa

Para poder identificar que se le hizo click a una grilla en particular, se le puso a las labels de cada grilla 2 propiedades. Una contiene un id de la grilla para así poder identificar la grilla del mapa o el de los tiles disponibles. También se les colocó una propiedad de tile ID, esta es utilizada para poder identificar al guardar el mapa para saber qué tile fue colocada. Notemos que también habrá un ID para un tile vacío para así ignorarlo en el futuro.

Se sobrescribe el método de QT EventFilter donde podremos tener acceso si se hizo un evento de click, si se hizo uno, se fija si fue un label. Como las únicas labels del widget son las de las grillas, ya podemos acceder a la label. Notemos que los labels en qt no pueden ser clickeables y tener un evento. Pero si se puede saber si se le hizo click. Una vez que se hace el click, se fija en qué grilla fue. Si fue en la grilla de tiles, se queda con el id y una copia del pixmap. Si fue en la grilla del mapa, se cambiará el pixmap actual de esa grilla por el pixmap guardado, si no hay uno guardado, nada mas se borra el de la grilla y quedará el label sin contenido, es decir, esa posicion estara vacia.

Operaciones disponibles

Erase

Se puede borrar todo el mapa. Para esto se borraran todos los pixmap de los labels de la grilla.

Eraser

Se puede utilizar una “goma de borrar”, es decir, se puede borrar algo puesto en algún lado en particular. Para eso, se borra el pixmap seleccionado y se cambia el id actual indicando que no hay un pixmap seleccionado o está vacío.

Save

Se guardarán los tiles colocados en la grilla del mapa en un archivo .yaml. Para esto se utilizó la biblioteca yaml-cpp. En este proceso se juntaran los tiles que sean consecutivos de forma horizontal como vertical. Después coloca los tiles que hayan quedado de forma

individual. Notemos que este proceso estará diferenciado sólo por las estructuras, es decir, las cajas, spawner como armaduras serán colocadas de una forma más simple pues no es necesario juntarlas. En el yaml se separa en varios sectores, primero se indicará el fondo como el largo y alto. Después se guardarán todos las estructuras del juego como pisos y paredes. Cada estructura guardará su posición inicial como final en eje x e y, también se indicará el id para la parte del cliente y que se muestre la tile correspondiente. Una vez guardado esto, se guardarán los objetos interactuables del juego como spawn de armas, cajas y armaduras. Cada una tendrá una posición como un ID para que después se pueda diferenciar una de otra. Finalmente están los spawn de los patos que tendrán su posición en x e y.

Load

Se puede cargar el mapa a partir de un .yaml. Este separa los distintos datos necesarios para un mapa. Para poder colocar los tiles correspondientes, se utiliza un método de QGridLayout al cual se le pasan la posición en x como y te devuelve la label colocada en esa posición. Por lo que por cada tile guardado en el yaml, se utilizó este método para acceder a esa posición y colocar el tile correspondiente con el ID. Para esto es que se hizo un vector de pixmap, pues ahora podemos acceder a los tiles correspondientes para colocarlos en la grilla nuevamente. Una vez se colocaron los tiles, se pondrán interactuables en la grilla de forma correspondiente al ID indicando en el yaml. Finalmente, se colocarán los Spawners de patos según indica en el yaml.

Cambiar el fondo

Para poder cambiar el fondo se deberá seleccionar cuál fondo utilizar dentro de los disponibles en el QComboBox. Este será mostrado en el fondo utilizado en la grilla del mapa.