Taller de Programación I (75.42)
Facultad de Ingeniería
Universidad de Buenos Aires

Worms Remake

Trabajo Práctico Final

2º Cuatrimestre 2023

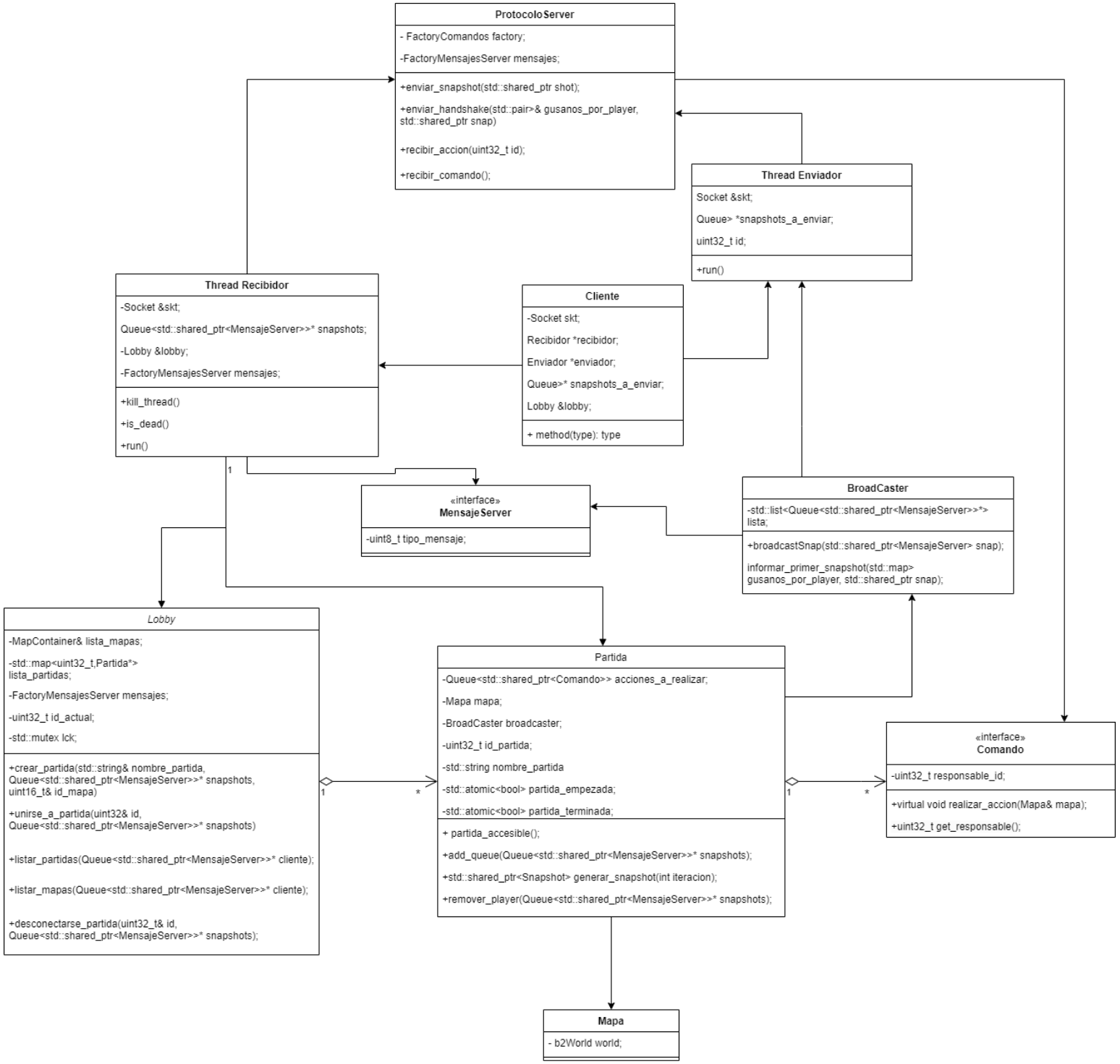
Documentación Técnica

Integrantes	Padrón
Ignacio Ezequiel Vetrano	106129
Ramiro Recchia	102614
Juan Hoszowski	104557
Manuel Rivera Villatte	106041

Índice

Documentación Técnica del servidor

Diagrama de clase:

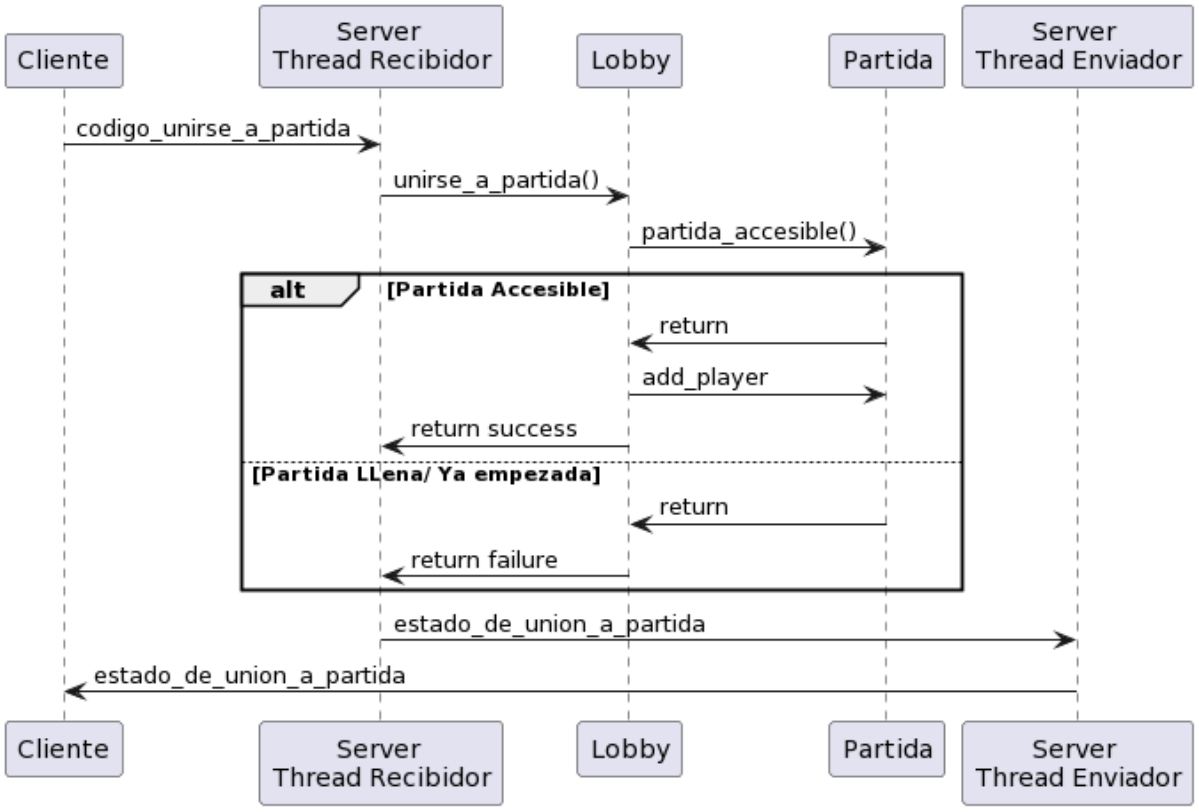


Lobby

La clase principal donde se encuentra el Game loop es la clase Partida pero una clase igual de importante es el Lobby, que se encarga de manejar las partidas existentes, de crearlas y de permitir que los jugadores puedan unirse a partidas creadas. El Lobby también se encarga de eliminar las partidas que terminaron.

Al intentar conectarse al servidor, la clase Thread Aceptador creará un clase Cliente que contiene dentro de él los 2 threads de comunicación, el thread enviador y el thread recibidor. Una vez aceptado el jugador puede pedirle al servidor (se interactúa con el lobby directamente) de: listar las partidas disponibles, listar los mapas cargados en el servidor, pedirle de unirse/crear una partida.

Listar partidas y mapas es muy simple, el thread recibidor al recibir el código para listar mapas/partidas le pedirá a la clase Lobby que le devuelva la lista para luego pushearla a la queue del thread enviador para que le envíe al cliente la información pedida.
Un ejemplo de cómo se maneja el jugador con el servidor antes de unirse a la partida podría ser justo cuando intenta pedir de unirse a una partida:



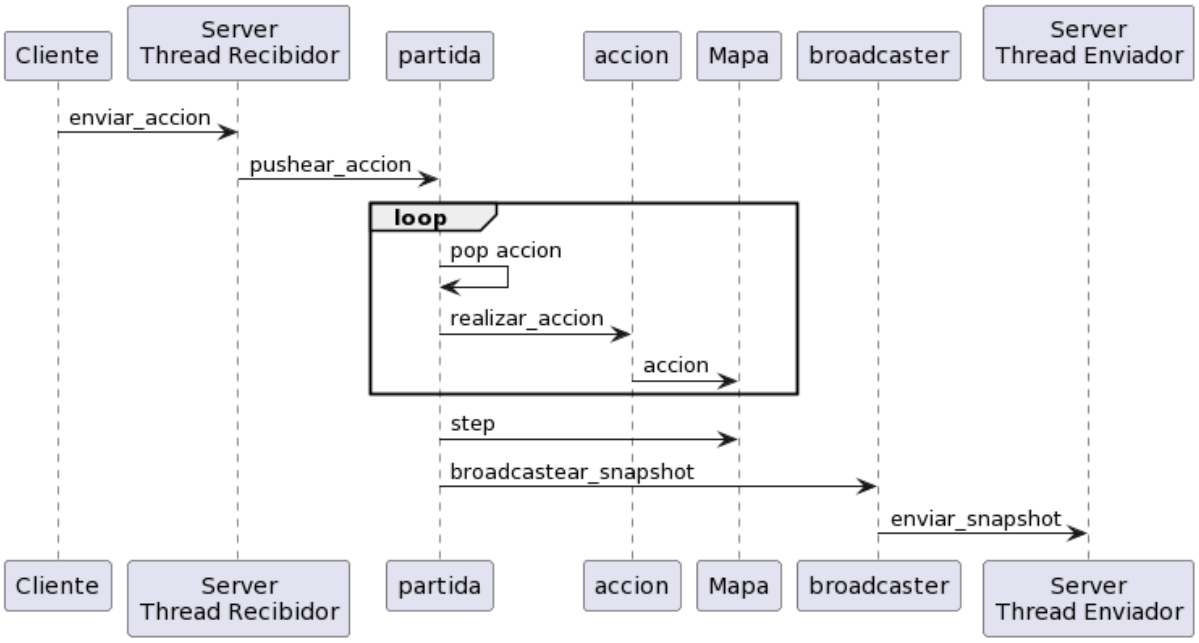
Interfaz de mensajes

La interfaz de mensajes nos será útil para que cuando el protocolo del servidor reciba un mensaje, el thread recibidor/enviador sepa que tipo de mensaje es y , si el código que se recibe envía más información adicional, poder almacenarla en el mensaje. Por ejemplo el mensaje MensajeUnirsePartida, tendrá un identificador que el recibidor reconocerá, y sabrá que dentro de este mensaje debería de haber un ID de la partida a la cual se quiere unir.

Esta interfaz le permitirá al enviador también saber si lo que debe enviar es una Snapshot de la partida, un handshake al cliente o de informarle al cliente que la partida terminó.

Interfaz de comandos/acciones

La interfaz de comandos nos permitirá, cada vez que recibamos una acción del cliente. poder rápidamente afectar al juego. Cada hijo de la clase Comando deberá implementar la función void realizar_accion(Mapa& mapa) override. Cuando el protocolo recibe una acción de, dependiendo del tipo llamará al factory de comandos y creará la acción dependiendo del código recibido. El recibidor la pusheara a la cola de acciones de la partida para que pueda ejecutarse su acción en el mapa. Un flujo típico sería por ejemplo:



Partida

En la clase partida es donde encontraremos el game loop principal, donde para conseguir un rate loop constante utilizamos *If behind, drop & rest.*
La partida no comienza inmediatamente una vez que es creada sino que espera a que se unan más jugadores o que el creador de la partida de la orden de comenzar la partida. Una vez recibido el comando de empezar la partida, se le envía a todos los jugadores conectados a esa partida un “Snapshot Handshake”, donde se envía la posición de todos los gusanos y de las vigas, como también el fondo de pantalla para que el cliente pueda graficar todo. Una vez terminado de broadcaster el “handshake” empieza el game loop como se muestra en el diagrama de flujo visto arriba.

Snapshot Partida/Handshake

Ambas clases son contenedores de información que se enviaran a la queue de los threads enviadores de todos los jugadores conectados por un MensajePartida y un

MensajeHandshake respectivamente. Se decidió separar ambos porque en el handshake se envía solo la posición de los gusanos y de las vigas mientras que en el Snapshot Partida se envia todo tipo de información: la posición de los gusanos, a qué entidad debe la cámara del cliente seguir, los proyectiles,los sonidos, las explosiones y la munición del Worm actual.

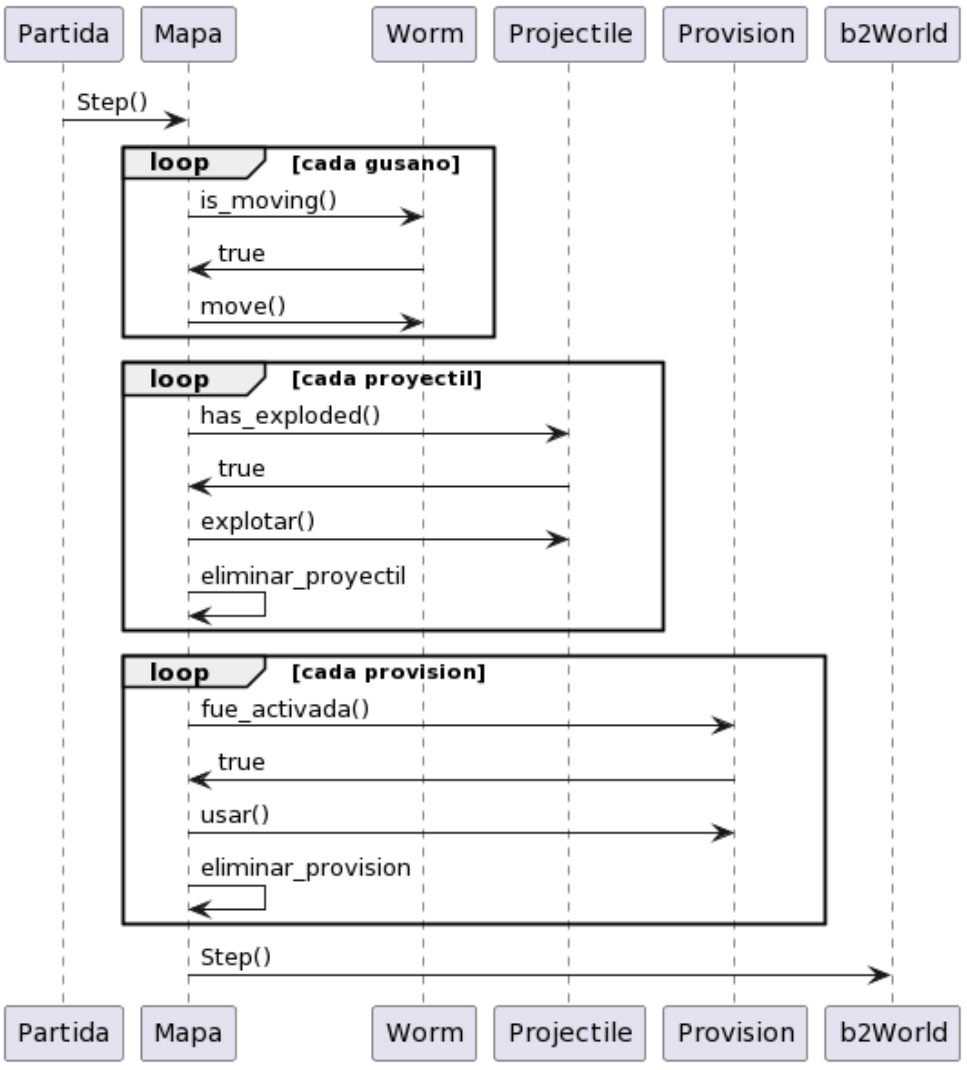
Broadcaster

El broadcaster es una clase que contiene a todas las Queue (estas queues son las que el thread enviador está esperando hacer pop) y se encarga de enviar los snapshots a los jugadores. Se encarga también de que no haya problema de concurrencia de por ejemplo, Si un jugador se desconecta que se remueva la queue antes de enviar un mensaje a un “cliente muerto”.

Juego

Mapa

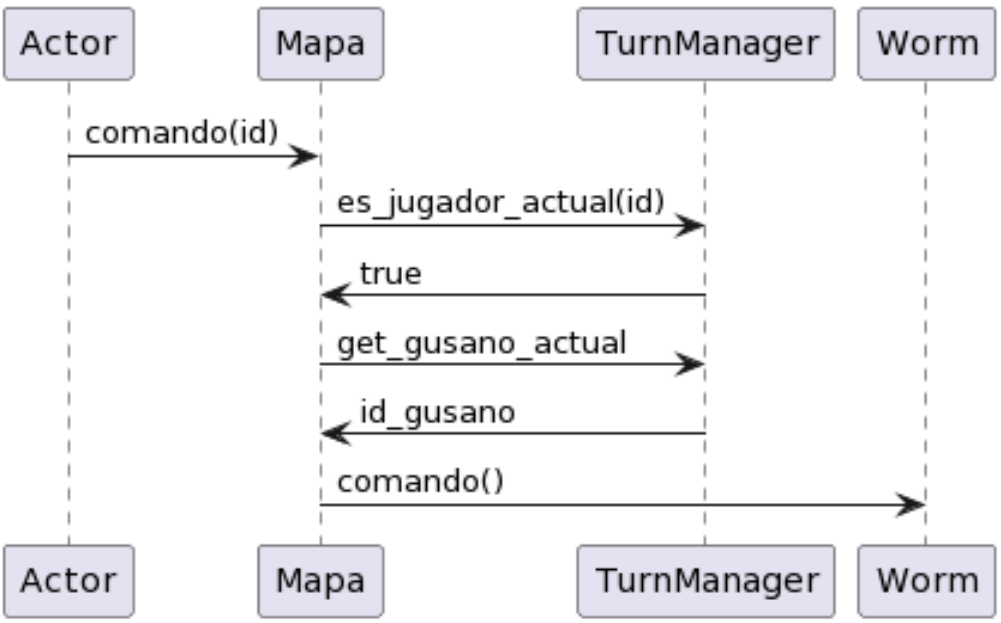
La clase Mapa se encarga de encapsular el mundo físico de Box2d. Almacena todo lo que posee un cuerpo físico: vigas, gusanos y proyectiles. Actúa como la interfaz que permite que comandos de usuario se apliquen sobre gusanos. Al inicializarse, levanta de un yaml pasado por parametro toda la información del mapa (posiciones de vigas, posiciones de gusanos, ángulos de vigas, etc.) y lo guarda en vectores, para que fácilmente se pueda pedir informacion sobre cuerpos dinámicos para cada Snapshot.



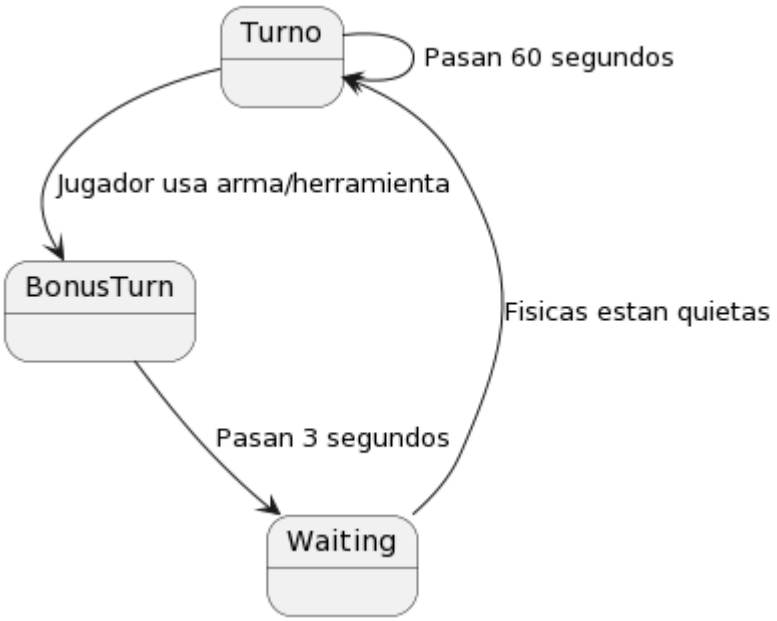
Como tal, Mapa es quien se encarga en cada Step, de actualizar los estados de los gusanos y proyectiles, y realizar verificaciones necesarias en cada game loop. Para esto, en cada Step, se iteran todos los gusanos, los proyectiles y las provisiones. Por ejemplo, hay que chequear si durante el step de físicas, hubo algún proyectil que contactó algo, entonces debe ser explotado, o bien activar una provisión que fue interactuada, etc.
En el diagrama de secuencia de arriba, vemos cómo en cada Step, Mapa realiza estas iteraciones. Dentro de cada loop, se muestra solo una verificación a fines de que se entienda, pero en realidad por cada gusano se verifican más cosas, como los sonidos que realizo, si tomo daño, si está apuntando, si está cargando su arma, etc. Para proyectiles también hay más acciones a realizar, por ejemplo, se debe aplicar el empuje del viento a los proyectiles susceptibles a este, en cada step.

TurnManager

Cuando le llega a Mapa un comando sobre un gusano (x. ej.: Mover, Saltar, Apuntar, Disparar…), solamente recibe el ID del usuario que realizó esa acción. Para determinar si ese comando debe ser aplicado al gusano, Mapa delega a TurnManager para que verifique si se trata del turno del jugador cuyo ID se recibió, y en caso positivo, se le solicita a TurnManager el id del gusano actual de ese jugador, sobre el cual se debe aplicar el comando. TurnManager también mantiene la cuenta de en qué estado se encuentra la partida actualmente (turno de un gusano, o bien esperando a que terminen las físicas, etc.). Es quien decide si un comando puede pasar o no. Es, efectivamente, quien evita que los jugadores puedan jugar fuera de su turno.



Para representar los estados de la partida y los eventos que provocan que cambie, tenemos el siguiente diagrama de estados:



Colisionables

Para garantizar un mundo físico correcto, donde las colisiones son manejadas acordemente, se utiliza una interfaz “Colisionable”, implementada por cada clase con un cuerpo físico.

Worm

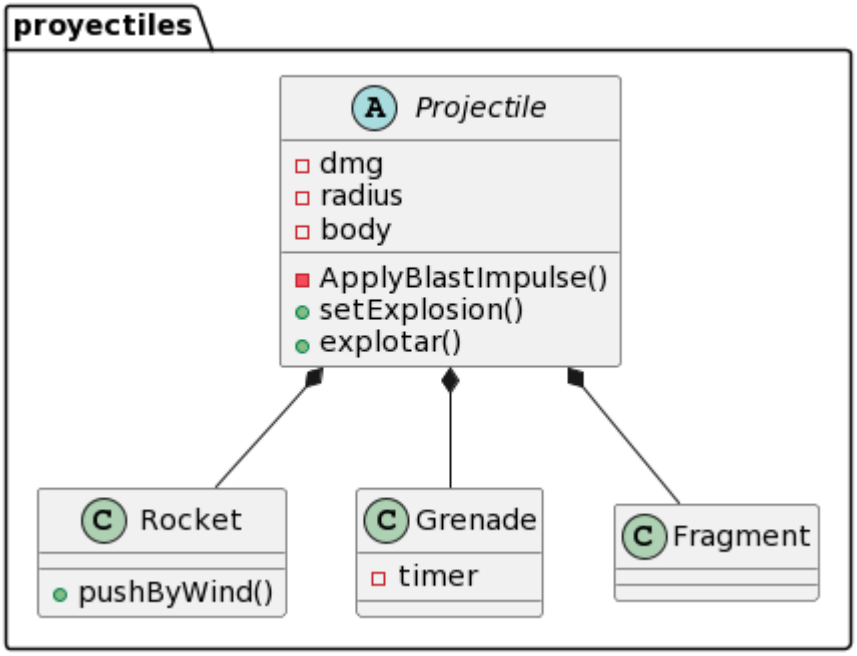
Es la clase que representa un gusano en la partida. Posee el cuerpo físico de Box2d, y un set de características relevantes al juego, como la vida, la dirección, si está saltando, etc.

Beam

Representa una viga del escenario. Es un cuerpo completamente estático. Pueden ser largas o cortas.

Proyectil

Son creados cuando un arma es disparada, y se lanzan con un impulso. Pueden ser Rocket (bazooka y mortero), que explotan al contactar un gusano o una viga; Granadas, que explotan luego de un cierto tiempo; o Fragmentos, creados luego de la explosion de un cohete de Mortero o una granada de Granada Roja.

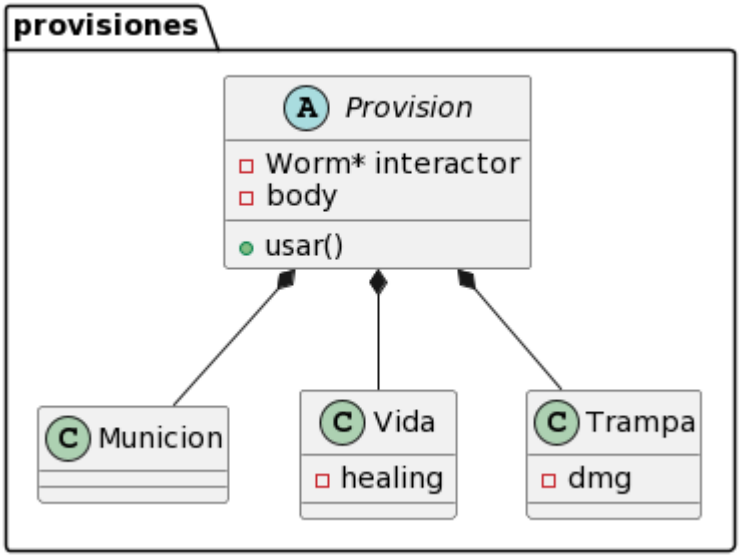


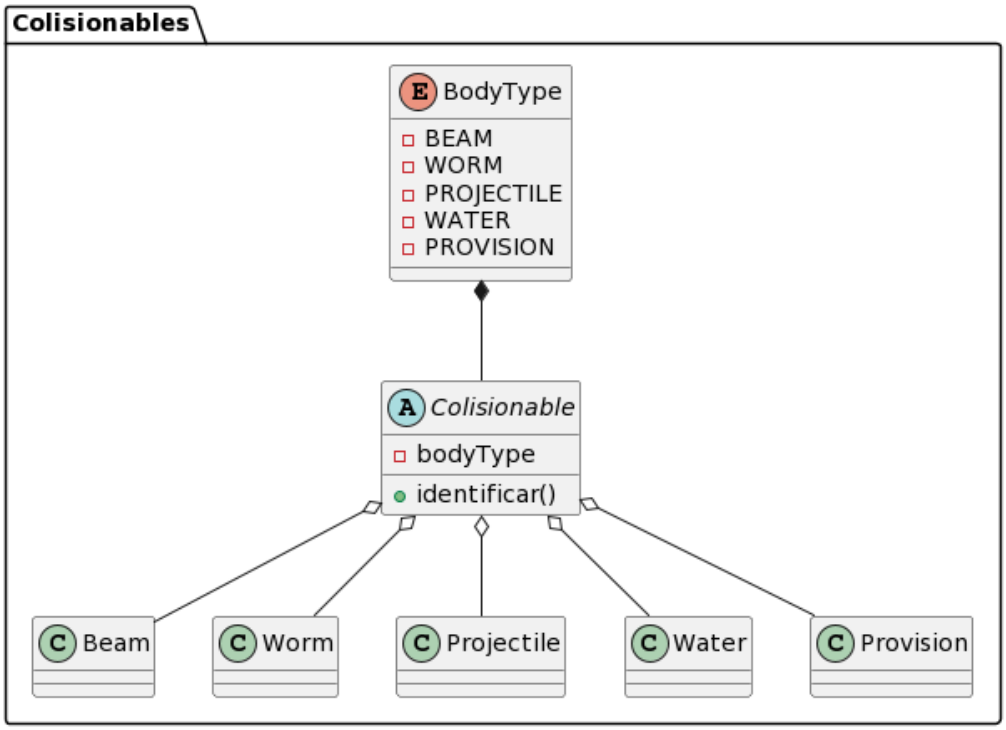
Water

Zona que, al colisionar con gusanos, los mata.

Provisión

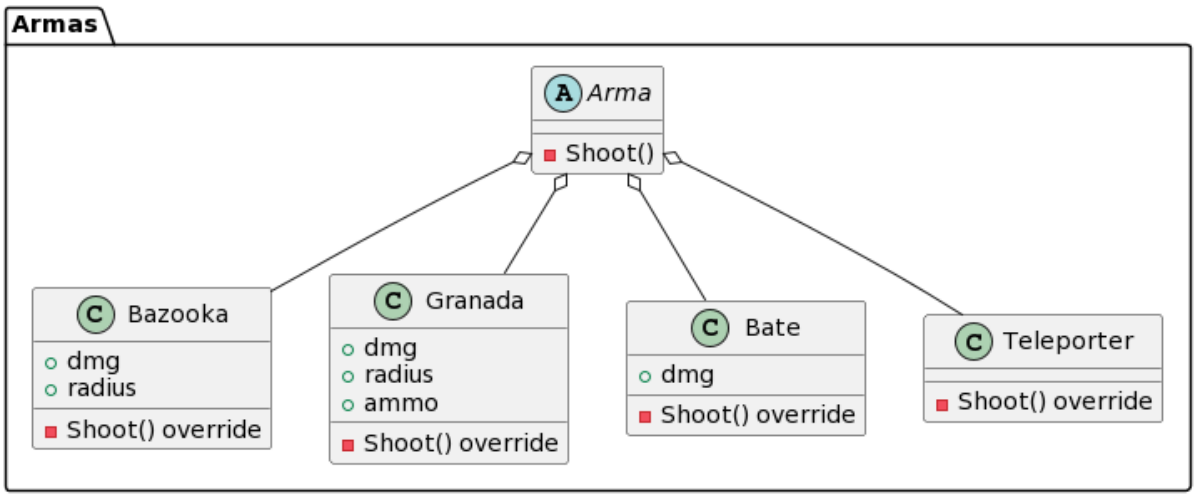
Son cajas que al ser contactadas por gusanos, desaparecen, y realizan una acción sobre el gusano que las activa (curarlo, recargar sus armas, o dañarlo).





Armas

Cada gusano posee una colección de sus armas mediante, valga la redundancia, la clase ColeccionArmas. Además, posee un puntero a su arma actual. Cada arma implementa esa interfaz, más notablemente el método Shoot() que según el arma hace algo distinto. Para las armas estilo granadas o lanzacohetes, el Shoot crea un Proyectil, y le aplica un impulso dependiendo de cuanta potencia tuvo el disparo.



Documentación técnica del cliente

Documentación técnica del menú del cliente

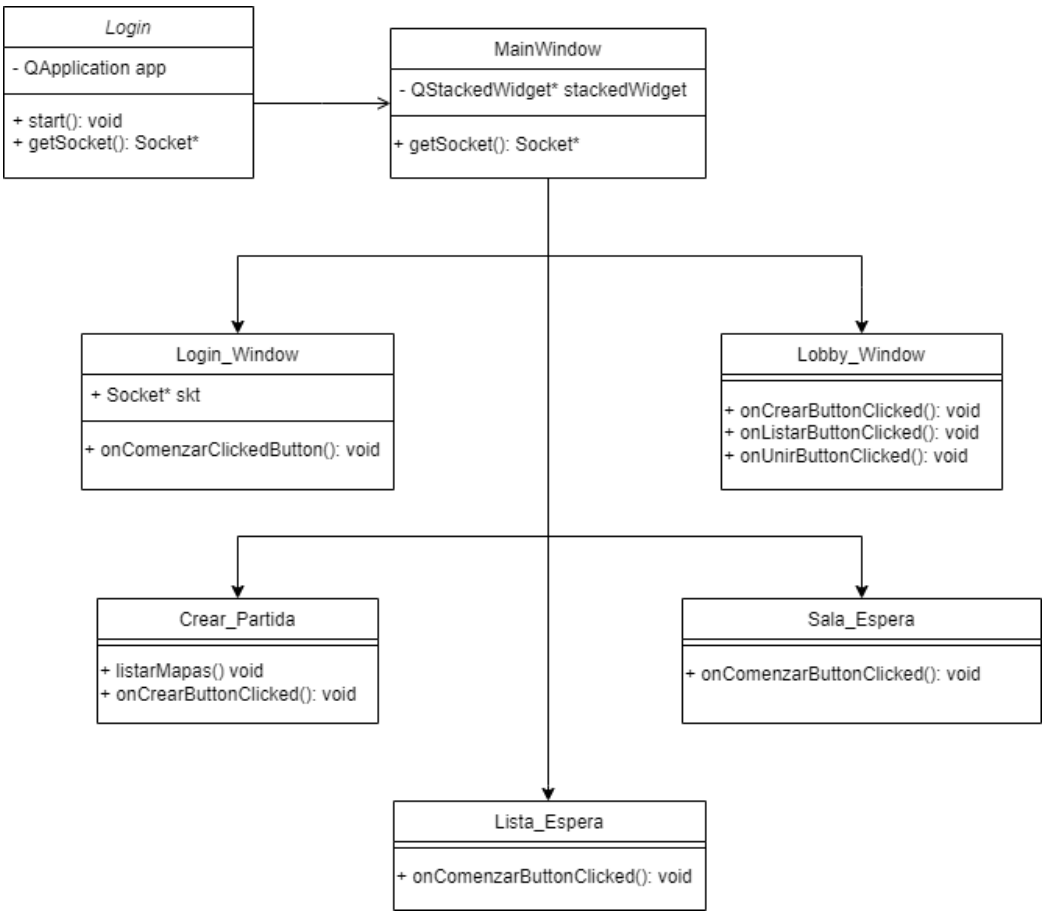
Luego de ejecutar el cliente entra en juego la lógica de QT. Esta llama a una clase Login encargada de encapsular la aplicación de QT ejecutándose. Esta clase tiene un MainWindow que hereda de la clase QMainWindow de propia de QT. Esta clase es la encargada de administrar las distintas vistas que presenta cada pantalla antes de iniciar el juego. Tiene un stack de widgets los cuales representan las distintas vistas que se irán presentando dependiendo de los botones que se presionen. Cada una de las distintas pantallas hereda de QWidget, lo cual es fundamental para poder darle formato a cada pantalla con los archivos '.ui'. Estos archivos son utilizados para toda la parte del diseño visual.

La primera pantalla es Login_Window, la cual el usuario debe colocar el servidor y el puerto. Estos datos serán utilizados para enviar información sobre la lista de las partidas y los mapas actuales ya que se los pide al servidor.

Luego viene una pantalla en donde el usuario puede listar partidas, crear mapas y unirse a las partidas. Esto lo gestiona la clase Lobby_Window. Si no hay partidas creadas al listar partidas no se verán en pantalla, si no se selecciona ninguna partida el usuario no podrá unirse, por lo tanto, es necesario crear una partida antes.

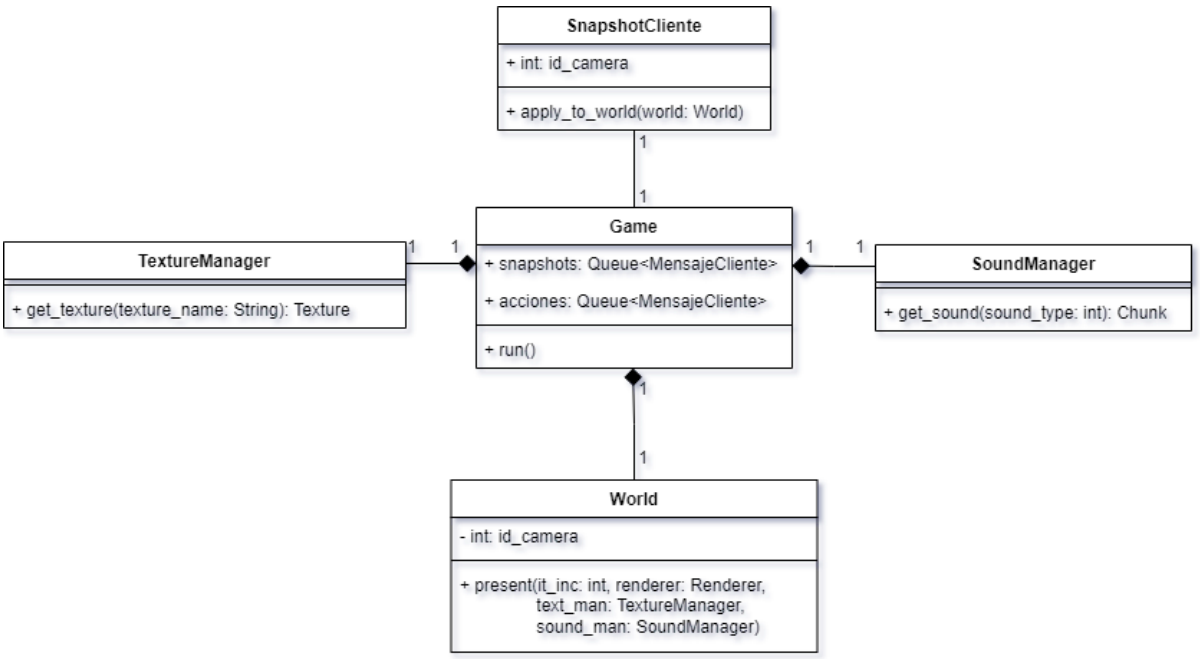
Luego de apretar el botón para crear una partida aparece la pantalla para crear la partida. El usuario puede colocar un nombre a dicha partida y seleccionar el mapa. Luego puede unirse a la partida directamente y saltar a una lista de espera. La clase Lista_Espera es la encargada de mandar una señal al servidor para comenzar la partida cuando el usuario que la creó lo indique.

Si un usuario quiere unirse a una partida puede listarlas, seleccionar una y apretar el botón para unirse y automáticamente pasará a la ventana de Sala_Espera. Esta tiene un botón para unirse definitivamente a la partida y esta no comenzará hasta que el usuario que la creó lo indique.



Documentación técnica del cliente dentro de una partida

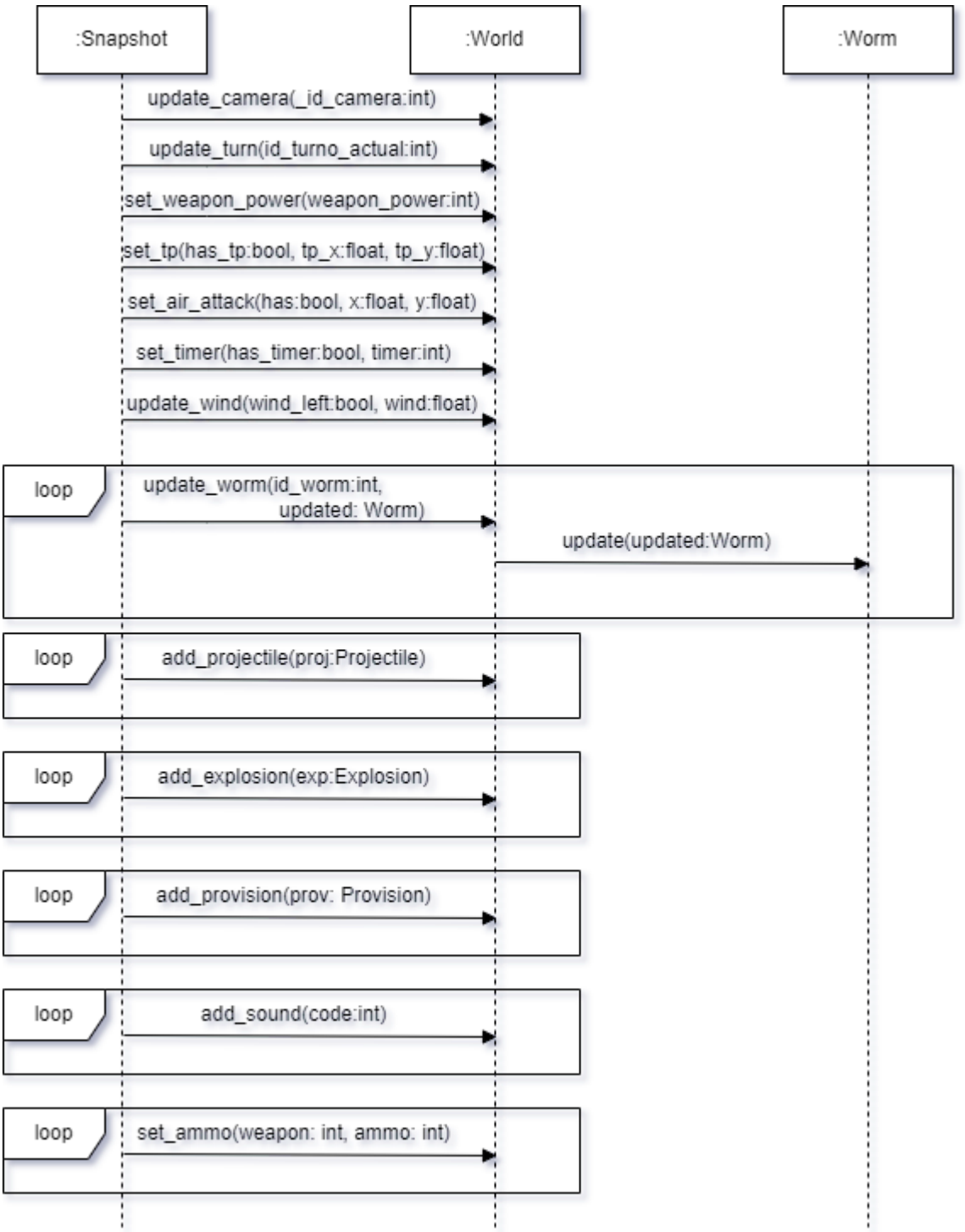
Dentro de una partida la clase principal es **Game**, la cual gestiona los eventos del mouse y teclado, gráfica la partida, mantiene el game-loop constante, y actualiza el mundo que se grafica (encapsulado en la clase **World**) con las **Snapshots** (datos de los elementos de la partida) que le llegan del servidor, como puede verse en el siguiente diagrama de clases:



Aquí puede verse que **Game** posee dos Queues, una para Snapshots recibidas, y otra para Acciones de eventos que detecta y que le envía al servidor.
A continuación se realizará una explicación más a fondo de cada una de las clases vistas en este diagrama.

Snapshot

Encapsula los datos de los elementos de la partida que recibe el cliente del servidor, estos pueden ser las posiciones de los distintos elementos, sus estados, de quien es el turno, a quien debe seguir la cámara, etc.
La clase **Snapshot** posee un método “apply_to_world” que recibe una instancia de la clase **World** y actualiza todos sus elementos con los datos recibidos del servidor, y lo hace como se ve en el siguiente diagrama de secuencia:



TextureManager y SoundManager

Son los encargados de cargar y gestionar las texturas y los sonidos respectivamente. Se utilizan para no tener que estar cargando durante la partida ningún recurso, y para no cargar varias veces recursos que podrían ser compartidos. Ambos cargan al comienzo de la partida todos los recursos y comparten sus referencias con las clases que se los pidan. Para la carga de los recursos ambos utilizan archivos .yaml ubicados en:

- Texturas: client\game\Texturas\superficies.yaml
- Sonidos: client\game\Sonidos\sounds.yaml

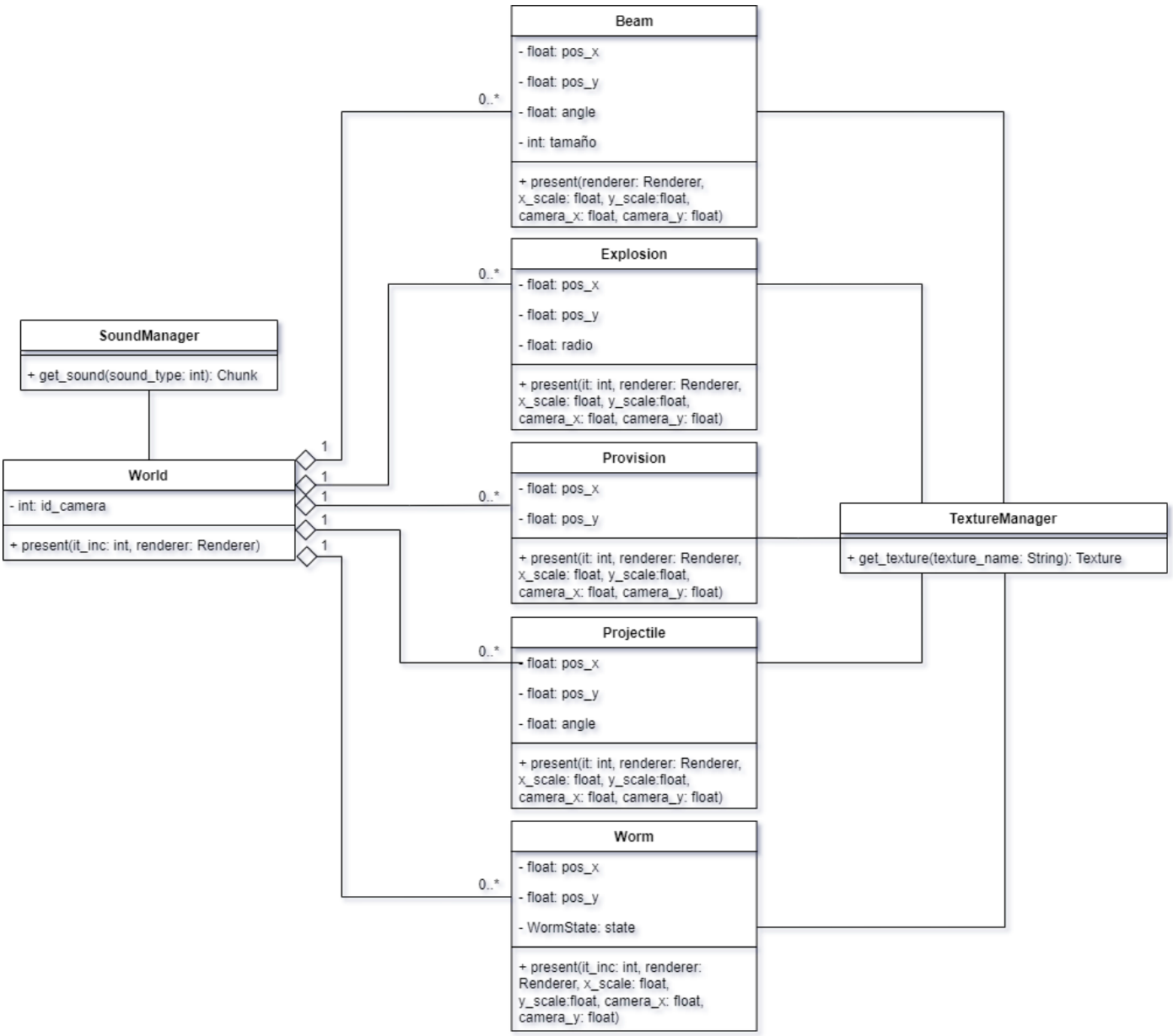
Ambos con el formato:
- *nombre: <nombre1>*
  ruta: <ruta1>
- *nombre: <nombre2>*
  ruta: <ruta2>
- *nombre: <nombre3>*
  ruta: <ruta3>

.
.
.

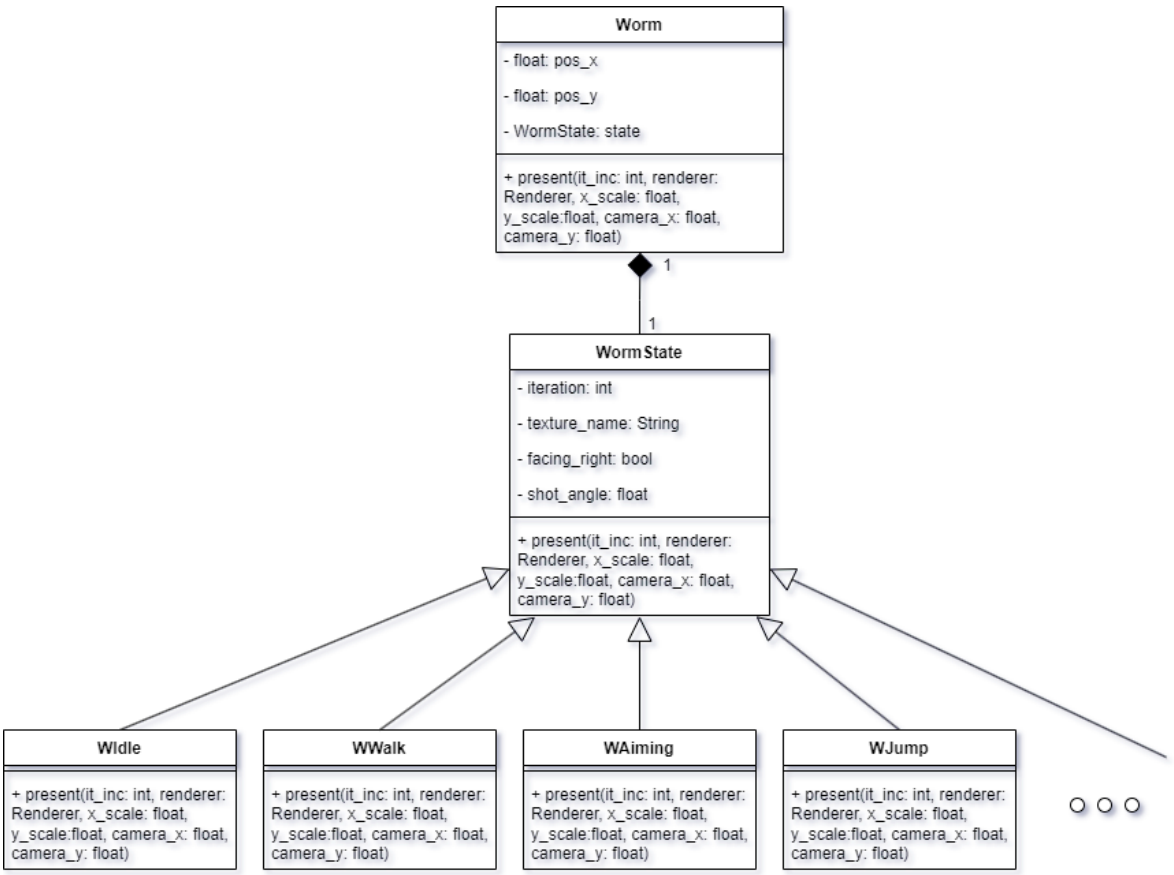
Donde el campo **nombre** es el nombre con el cual se va a poder pedir el recurso al gestor, y el campo **ruta** es la ruta del archivo con respecto a **“/var/worms/resources/sprites”** en el caso del TextureManager, y con respecto a **"/var/worms/resources/sonidos"** en el caso del SoundManager (Que son las rutas donde deben estar guardados los recursos).

World

Es la clase que encapsula los elementos de juego que están en el cliente actualmente, y es la encargada de graficar todo lo que se ve en pantalla y reproducir los sonidos. Vamos a profundizar más en esto, este es un diagrama de clases que muestra qué elementos contiene **World** y hace que se grafiquen:



Como se puede observar, cada elemento de la partida (vigas, gusanos, proyectiles, etc.) tiene un método para graficarse llamado “present”, **World** delega la graficación a cada uno de los elementos que componen la partida, de esa manera cada clase implementa su manera de graficarse en la ventana. Un caso destacable es el de la clase **Worm**, esta clase no implementa su método de graficación, sino que lo delega a su estado, representado por la clase **WormState**, que es una clase abstracta de la cual descienden todos los estados que puede tener el gusano, como se puede ver en el siguiente diagrama de clases:



De esta manera, agregar un nuevo estado del gusano al cliente es simplemente crear una nueva clase que herede de **WormState**, agregarlo al enum WormStates (dentro de **common\worm_states.h**) y agregar su código al **WormStateGenerator** (una clase encargada de recibir el código del estado y devolver el estado correspondiente a ese código).

Documentación técnica del protocolo del cliente

En el cliente, el protocolo se comporta de forma similar a lo que se explicó en la documentación del servidor con la interfaz de mensajes y comandos/acciones del servidor. Por cada tecla válida que apriete el cliente, se enviará al servidor por protocolo una acción/comando con un mensaje de AccionJugador.
El mensaje Handshake Enviar se utiliza para hacerle saber al thread recibidor de parte del servidor que se leyó correctamente el handshake. El recibidor del cliente puede también pushear a la Queue de Game un mensaje de que la partida terminó o un snapshot dependiendo de lo recibido por el socket.

Protocolo

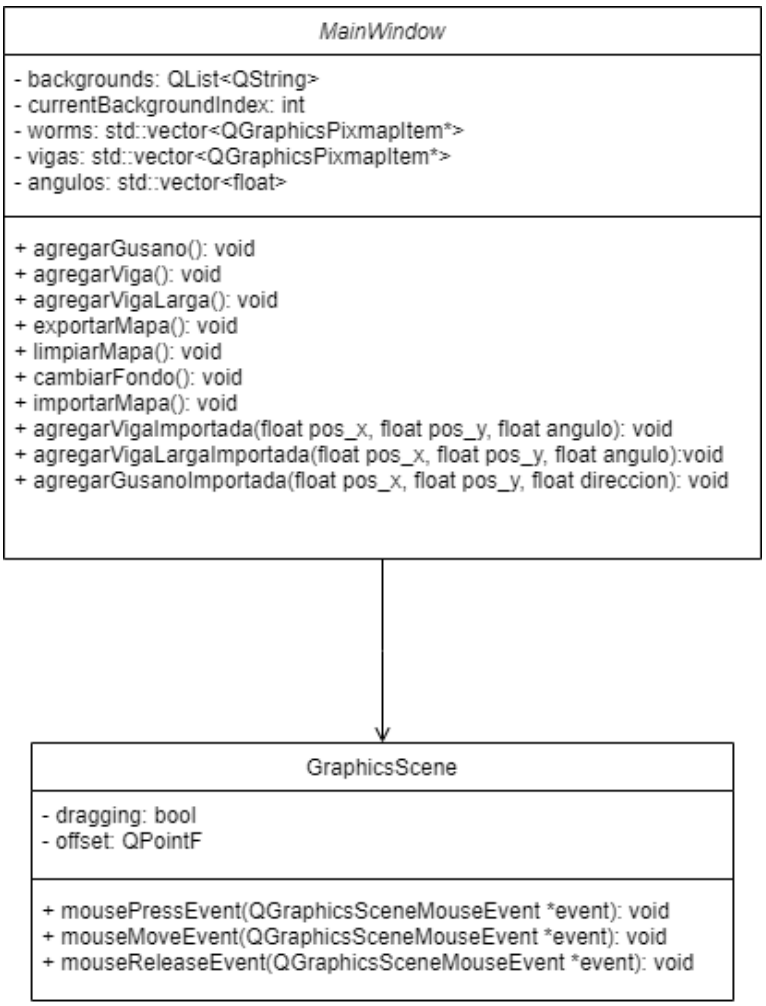
Para simplificar el protocolo, se hizo que Protocolo sea una clase padre y que ProtocoloServer y ProtocoloCliente sean sus hijos. Esto permite poner en común las funciones más típicas de comunicación. Por ejemplo cuando se quiere enviar un string, en vez de tener que poner líneas de código transformando el endianness del len del string a network y etc.. Se pueden crear funciones como enviar_string que facilitan la comprensión de lo que se quiere enviar en el protocolo cliente y del servidor. Se crearon también enviar_4_bytes() por ejemplo o recibir_4_bytes().
Permite también poner en común los distintos códigos que pueden recibir el servidor y el cliente. Esto facilita también el tirar errores, se creó un error ClosedSocket en caso de que algún socket se cierre, y luego se catchea en los WHILE de los recibidores del enviador y del recibidor del cliente y del servidor por igual.

Documentación técnica del editor

El editor de mapas consta de una clase MainWindow que hereda de QMainWindow. El editor sólo tiene una pantalla y MainWindow es la encargada de manejar los botones que se presionan, los cambios del fondo y de la importación y exportación de los mapas. En el constructor se generan los vínculos entre botones y acciones. También se establece el formato de la ventana con el archivo '.ui'.

Para mover los objetos incluidos en el mapa es necesario implementar una clase extra GraphicsScene que hereda de la clase QGraphicsScene de QT. Para mover los items el usuario puede hacer drag&drop y sobreescribir los métodos de la clase padre es necesario para customizar dicho movimiento de las imágenes.

Las imágenes se almacenan en un vector de punteros a QGraphicsPixedItem, uno para vigas y otro para los gusanos. que luego serán exportados o importados.


MainWindow
- backgrounds: QList<QString>
- currentBackgroundIndex: int
- worms: std::vector<QGraphicsPixmapItem*>
- vigas: std::vector<QGraphicsPixmapItem*>
- angulos: std::vector<float>
+ agregarGusano(): void
+ agregarViga(): void
+ agregarVigaLarga(): void
+ exportarMapa(): void
+ limpiarMapa(): void
+ cambiarFondo(): void
+ importarMapa(): void
+ agregarVigaImportada(float pos_x, float pos_y, float angulo): void
+ agregarVigaLargaImportada(float pos_x, float pos_y, float angulo):void
+ agregarGusanoImportada(float pos_x, float pos_y, float direccion): void
GraphicsScene
- dragging: bool
- offset: QPointF
+ mousePressEvent(QGraphicsSceneMouseEvent *event): void
+ mouseMoveEvent(QGraphicsSceneMouseEvent *event): void
+ mouseReleaseEvent(QGraphicsSceneMouseEvent *event): void