Documentación Técnica
Wolfenstein 3D

[7542] Taller de Programación I
Segundo cuatrimestre de 2020

Alumno :	BIANCARDI, Julián
Padrón:	103945
Alumno :	CZOP, Santiago
Padrón:	104057
Alumno:	GIARDINA, Fernando
Padrón:	103732
Alumno :	STENGHELE, Juan Francisco
Padrón:	104000

Índice

1. Requerimientos de software

2. Descripción general

El proyecto consta de dos partes: cliente y servidor. Ambos son programas separados con lógica de funcionamiento distinto mas allá de comunicarse bajo un protocolo común y compartir implementaciones de algunas clases.

Desarrollar cliente...

En cuanto al servidor se puede decir que su función principal es la de contestar los paquetes enviados por los clientes y comunicarlos con el resto de clientes. Para ello contiene un hilo principal llamado `ProcessorThread` cuyo funcionamiento consiste en para cada paquete recibido, modelados con la clase `Packet`, crear a partir de un factory de handlers (siguiendo el patrón de diseño Factory), el handler correspondiente según el tipo de paquete recibido, que se procesará y responderá a los clientes a través de su método `void handle(...)`. El servidor termina su ejecución al recibir una 'q' por la entrada estándar y el hilo principal es el que llama a un stop forzado del hilo de procesamiento. Además se tienen dos entidades muy importantes: `ClientManager` y `MatchManager`. El primero es el encargado de manejar cada cliente modelado en la clase `Client` y provee una interfaz limpia y sencilla para comunicarse con cada uno a través del envió de paquetes. En cambio, para la recepción el mánager de clientes contiene una cola bloqueante donde se encolarán todos los paquetes recibidos y de donde el `ProcessorThread` los desencolará para crear los respectivos handlers. Cada Client, contiene un hilo interno de recepción y uno de envió donde a través del uso del socket (que es movido desde el hilo aceptador) creado para vincularse con el cliente, se transfiere la información. Por otro lado el `MatchManager` se encarga de administar las diferentes partidas que se ejecutan en simultaneo. Cada handler lo que hace es solicitarle la partida al que el paquete pertenece y que indica en uno de sus campos, y llamar al método correspondiente en cada partida dependiendo de que es lo que el paquete represente (movimiento, rotación, disparo, etc) y actualizar de esta manera el modelo de la partida que el servidor contiene.

3. Módulos Cliente

3.1. Armas

Las armas se encuentran implementadas como clases capaces de actualizarse y devolver una instancia de Hit con toda la información relevante sobre la actualización. Durante esta actualización, se dispara de ser lo correcto, se modifica el estado interno del arma y se hacen los chequeos y cálculos que le corresponden a la lógica de cada arma. La lógica del rocket launcher se encuentra en su mayoría en el servidor, dado que el cliente solo dispara los cohetes.

3.1.1. Propiedades

Las armas cuentan en general con las siguientes propiedades:

- GunState: un estado interno del arma.
- Rango Mínimo: por default es 0 para todas.
- Rango Máximo: rango máximo, donde el daño pasa a ser 0.
- Spray: ángulo completo del desvío del arma, permitiendo que la bala se desvíe hasta en Spray / 2 a cada lado de a donde apunta el jugador.
- Desviación Estándar: concentración de las balas en el centro de la mira
- Variables de Control (varias): variables capaces de controlar cuando debe salir una bala (o acuchillarse).

3.1.2. Precisión

La precisión de las armas está diseñada como una desviación sobre el ángulo en el que apunta actualmente el jugador. Este pequeño desvío está modelado como una distribución normal con media 0 y desvío estándar distinto según el arma. El ángulo del spray hará que se conserven solo ciertos valores generados por la distribución, descartando todos los que superen la mitad del ángulo especificado en el spray, hacia cualquiera de los lados. La lógica de este desvío se modeló en la clase Spray.

3.1.3. Modificadores

Los daños realizados por cada bala obtienen un daño base distribuido uniformemente entre 1 y 10. Luego son modificados por la distancia del disparo, decayendo linealmente entre el rango mínimo y máximo. Por último, son multiplicados por un factor dependiente del ángulo. Este factor fue arbitrariamente modelado como un coseno que tiene su primera raíz en 0.75 el spray del arma. Por lo tanto, este modificador como mucho alcanzará un valor de 0.5 cuando el objetivo se encuentra exactamente en el borde de nuestro spray.

3.1.4. Trayectory

La clase madre Gun provee un método llamado Trayectory capaz seguir la trayectoria de una bala y devolver un objeto identificable al que se le pegó. Basándonos en el juego real, lo único a lo que se le puede pegar es a las paredes y a los jugadores, y todos los otros objetos en el mapa son ignorados. La lógica interna general de Trayectory es la siguiente:

- Recibe una bala representada por una semirecta (Ray), un BaseMap (solo nos importan sus paredes) y un vector de jugadores.
- Se obtiene la distancia a la pared más cercana.
- Por cada jugador, se revisa si está en la trayectoria y más cerca que el objeto más cercano y la pared más cercana.
- Si algún jugador estaba en la trayectoria de la bala y no se encontraba detrás de una pared, se lo devuelve. Sino, se devuelve un puntero nulo.

Para conocer si un jugador se encuentra en la trayectoria de la bala se recurrió a un concepto ampliamente utilizado en Astronomía llamado diámetro angular. El diámetro angular nos indica el tamaño aparente en una esfera (o circunferencia para nosotros) de un objeto que se encuentra a cierta distancia y es de cierto tamaño. El diámetro angular se mide en cuántos ángulos ocupa, por lo que conociendo la posición del objeto podemos saber entre que 2 ángulos se encuentra, y si nuestra bala está entre esos 2 ángulos entonces le puede pegar. El tamaño de los jugadores se calibró manualmente testeando, tal que a una distancia de aproximadamente 15 casillas, a veces le pegue y a veces no.

4. Módulos servidor

4.1. Principal

4.1.1. Descripción general

Denominamos módulo principal al corazón del servidor. Siguiendo con lo explicado en la descripción general el módulo principal estaría compuesto del servidor creado en el main del programa y los diferentes hilos y sub clases que salen de él. La clase `Server` en su construcción crea los managers que luego explicaremos y que cuenta con el método principal de `void run_server(std::string& port)`. Este método recibe el puerto al que se conectará el servidor y se encarga de crear el

hilo aceptador (`AcceptClientsThread`) y el hilo procesador (`ProcessorThread`). Mientras el primer hilo se encargará de aceptar conexiones y mover los sockets obtenidos al `ClientManager` que se encargará de su manejo, el segundo como se detalló en la descripción general, se encargará de desencolar paquetes de la cola de recepción, crear sus handlers mediante el uso de una clase fábrica y ejecutarlos. El hilo procesador deja de procesar cuando el hilo principal recibe una 'q' por entrada estándar y llama a un stop forzado y un join del hilo.

4.1.2. Clases

Las clases que forman este módulo son:

- `Server`: Cuenta con un método principal de `void run_server(std::string& port)` cuyo funcionamiento se explicó en la sección anterior.
- `AcceptClientsThread`: Se encarga de aceptar las conexiones entrantes a través del método `void run() override` que implementa de su clase madre `Thread`. Los sockets que nacen del accept son movidos al `ClientManager`.
- `ProcessorThread`: Se encarga de desencolar paquetes de la cola de recepción, crear el handler correspondiente y ejecutarlo. Al igual que la clase anterior utiliza para ello el método `void run() override` que implementa de su clase madre `Thread`.

4.1.3. Diagramas UML

4.2. Partidas

4.2.1. Descripción general

En este módulo se puede encontrar la clase principal `MatchManager`, que se encarga de almacenar y de administrar los diferentes `Match` y, a su vez, se incluye en este módulo todas las diferentes clases que conforman el modelo del juego del servidor.

El `MatchManager` guarda en un unordered_map las diferentes partidas existentes como valor y un id como clave (`std::unordered_map<unsigned char, std::shared_ptr<Match>> matches`). Dicho id es un identificador de cada partida y sirve para determinar principalmente un paquete pertenece a una partida o a otra o si se quiere por ejemplo eliminar una partida (cuando finaliza). Lógicamente, el manager contiene un getter que se utiliza para obtener una partida según su id para luego operar con ella.

Cada partida se representa con una instancia de `Match`. La clase en su constructor recibe un string con el nombre del mapa que se utilizará, además de un id que se lo otorga el `MatchManager` y por último el id del host (jugador) para que pueda comenzar la partida cuando presione Enter.

Cuando la partida se crea también se carga el archivo YAML del mapa y se crea una instancia de la clase `Map`. La instancia del mapa contendrá: los items en un unordered_map con sus respectivos ids (que se utilizan para identificar si se agarró tal o cual item), los objetos que necesiten un id en un unordered_map llamado identifiable_objects (por ejemplo los cohetes), en un vector los objetos que no necesitan un id (como las columnas, mesas, etc), un unordered_map para las puertas y pasadizos secretos y por último otro unordered_map para los jugadores que necesitan un id para identificarlos. Además contendrá los puntos en que aparecerán los jugadores y los perros.

Para leer el YAML el mapa instancia un `MapLoader` que se encargará de iterar el archivo agregando los diferentes objetos. Para las paredes, se utiliza una lógica similar salvo que se hace en la clase madre de `MapLoader`, ya que su comportamiento es similar al del cliente. En dicha clase común, lo que se hace es cargar las paredes a la matriz base del mapa. Finalmente, otro punto a recalcar, es que el mapa a medida que crea los diferentes objetos del mapa, les va otorgando un id. Por como fue pensando es necesario que el mapa del cliente también haga lo mismo en el mismo orden para de esta manera sincronizar paralelamente los ids que utilizarán.

Una vez que la partida se creo, se debe iniciar con el método `bool start(unsigned int player_id, BlockingQueue<Packet>& queue)`. En dicho método se instancian y se hacen correr los hilos de los bots y

el del reloj de la partida. El hilo de cada bot se describirá en una sección posterior y el de la partida lo que hace es contar el tiempo hasta que se acabe y deba terminar la partida. No solamente se encarga de esta importante tarea sino que también se encarga de ejecutar los eventos por tiempo. Entre los eventos por tiempo se puede encontrar el controlador del cohete, que cada instantes pequeños actualiza su posición hasta estrellarse y explotar y también el cierre automático de las puertas. Dichos eventos se encuentran en un unordered_map y se actualizan constantemente. El tiempo que tardan en ejecutarse es medido (para tomarse como tiempo utilizado) y se hace dormir al hilo por un tiempo determinado (menos de un segundo) menos el tiempo ya consumido. De esta forma se utiliza el hilo como un reloj.

Volviendo a la funcionalidad de la clase `Match`, cuenta con una diversidad de metodos diferentes que representan acciones dentro del juego. Como por ejemplo lo más basico:

```
bool Match::move_player(unsigned int player_id, unsigned char direction) {
  if (!started) {
    throw MatchError("Failed to move player. Match hasn't started.");
  }

  if (!player_exists(player_id)) {
    throw MatchError("Failed to move player. Player %u doesn't exist.",
                     player_id);
  }

  Player& player = map.get_player(player_id);
  Point requested_position = player.next_position(direction);

  if (checker.can_move(requested_position, player)) {
    player.set_position(requested_position);

    return true;
  }

  return false;
}
```

Como se puede ver, se obtiene al jugador y la posición a la que solicita moverse (el cliente solicita moverse en una dirección, se explicará más adelante) y luego se le pregunta al checker, instancia de `CollisionChecker`, si se puede mover al jugador a ese punto. Si se puede devuelve true y si no, false.

`CollisionChecker` es la clase que se encarga de verificar las colisiones y todo lo relacionado en el juego. El servidor es quien se encarga de determinar si un movimiento es valido o no, el cliente solamente acata lo que el servidor le diga. Antes de explicar la clase encargada de las colisiones, se comentará que todo objeto que ocupe un área tendrá una máscara de colisión. Dicha máscara tiene un método principal `bool occupies(const Point& where) const`, que lo implementan todos los tipos de máscara (rectangulares, circulares y switcheables) y se encarga de determinar mediante algoritmos muy simples si un punto es ocupado por ellos o no. El `CollisionChecker` presenta varios métodos importantes. Uno de ellos es `bool can_move(const Point& where, const Moveable& who)` y se encarga de determinar si el Moveable recibido (los jugadores heredan de Moveable) se puede mover o no a la posición pedida. Para ello se revisan dos puntos, el centro del Moveable y el punto en el borde de la máscara en la dirección en la que se mueve. Con esos dos puntos se revisa que no haya paredes en el mapa y que ningún objeto o jugador colisione con ellos, llamando al método antes mencionado para cada máscara. Se decidieron utilizar estos dos puntos para obtener un sistema de colisiones optimizado que sea rápido y sencillo (los resultados obtenidos así lo demuestran).

Entre las máscaras se puede encontrar una interesante como es la `SwitchMask`. Básicamente es una máscara que contiene otra máscara salvo que tiene un atributo `bool active`. Si se encuentra activada (`active == true`) entonces el `bool occupies(const Point& where) const`, lo determina la máscara interna, si esta desactivada (`active == false`), entonces no ocupa nada (se vuelve invisible). De esta forma se pueden modelar puertas, pasadizos secretos y fantasmas.

El `Match` cuenta con muchísimos otros métodos fundamentales que no se desarrollarán del todo porque se podría tornar abrumante. Dichos métodos serán descriptos más abajo en la subsección

de Clases.

Por último se encuentran todos los objetos pertenecientes a la partida en sí. Hay dos clases madres principales: la primera es `Object` que es una entidad con una máscara de colisión y una posición con ángulo (`Ray`) y la segunda es `Identifiable` que tiene un `unsigned int id` que es enviado externamente (no se establecen automáticamente).

Objetos como lo son las mesas y columnas por ejemplo se modelan con las entidades `CircularObject` o `RectangularObject` que ambas heredan solamente de `Object`, pues un id es inutil. Luego encontramos los items que son todas clases que heredan de `Item` e implementan los métodos `bool can_be_used_by (const Player& whom)` y `void use(Player& user)`. El primer método se utiliza para determinar si un jugador esta en condiciones de usar el item (por ejemplo, no esta con la vida al máximo para agarrar el kit médico) y el segundo hace que el jugador use el item (por ejemplo, se suma la vida si se agarra el kit médico). También se tienen las puertas con son modeladas con las clases `NormalDoor` y `LockedDoor`, que ambas heredan de `Door` y su funcionamiento es basado en la `SwitchMask` que se explicó anteriormente.

Finalmente y no menos importante, se tiene la clase del jugador `Player` que a su vez hereda de `Moveable`. Esta clase `Moveable`, se utiliza para representar todas las entidades del juego que se puedan mover. Por ello los cohetes son modelados con instancias de la clase y el jugador hereda de ella. La clase `Player` contiene más que nada getters y setters y métodos para consultar estado. Lo más destacable es su posible conversión en fantasma. Básicamente la partida termina cuando queda solamente un jugador con vida o si se queda sin tiempo la partida. Si un jugador muere, vuelve a aparecer en el punto donde arrancó. Sin embargo, si se queda sin vidas, entrará en un estado de espectador fantasma, que aumentará su velocidad y hará que los jugadores no colisionen con él. Por esto último el jugador también utiliza una `SwitchMask`.

4.2.2. Clases

Las clases que forman este módulo son:

- `MatchManager`: Es una de las clases más importantes del módulo, su importancia recae sobre el hecho de que es la encargada de almacenar las partidas y manejar su creación y destrucción. El método más importante es el de `Match& get_match(unsigned char match_id)`, aunque también caben destacar `unsigned char create_match(unsigned int host_id, std::string& map_name)` (devuelve el id de la partida creada) y `void delete_match(unsigned char match_id)`.
- `Match`: Es posiblemente la clase más importante del módulo ya que representa una partida del juego. Su función principal es la de ejecutar las diferentes acciones que ocurren a lo largo de la partida por lo que a continuación se listarán los métodos.
  - `bool start(unsigned int player_id, BlockingQueue<Packet>& queue)`: Comienza una partida y empieza a correr el reloj y a funcionar los bots.
  - `bool add_player(unsigned int player_id)`: Agrega un jugador a la partida.
  - `bool move_player(unsigned int player_id, unsigned char direction)`: Mueve el jugador enviado en la dirección recibida (más adelante se explicarán las direcciones). Devuelve true o false depediendo si el movimiento es realizable (no hay colisión) o si no (hay colisión).
  - `bool rotate_player(unsigned int player_id, unsigned char direction)`: Similar al método anterior salvo que con una rotación y siempre se realiza (no hay chequeos de colisiones).
  - `unsigned int grab_item(unsigned int player_id)`: Devuelve el id del item si el jugador esta en condiciones de agarrar uno, sino devuelve 0 (id invalido). Por que esté en condiciones se refiere a que no solamente este sobre el item sino que además posee el estado necesario para tomarlos (determinada vida o balas por ejemplo).
  - `bool change_gun(unsigned int player_id, unsigned char gun_id)`: Cambia el arma del jugador a la solicitada.

  - `void damage_player(unsigned int player_id, unsigned int damager_id, unsigned char damage)` : Le realiza el daño al jugador. Si su vida es igual o menor a 0, mata al jugador.
  - `void shoot_gun(unsigned int player_id, unsigned int objective_id, unsigned char damage)`: El jugador enviado dispara su arma contra el otro jugador enviado. No realiza la lógica de disparo (se hace en el cliente), solo efectúa sus resultados.
  - `bool is_using_rocket_launcher(unsigned int player_id))`: Devuelve si el jugador esta usando el lanza cohetes.
  - `void shoot_rocket(unsigned int player_id)`: Lanza un cohete desde la posición del jugador y crea su controlador.
  - `bool move_rocket(unsigned int rocket_id)`: Mueve el cohete y devuelve si pudo o no.
  - `std::map<unsigned int, unsigned char> explode_rocket(unsigned int rocket_id, unsigned int owner_id)`: Explota el cohete enviado y devuelve un mapa con los ids como key y los daños como value.
  - `void kill_player(unsigned int player_id)`: Mata al jugador de la partida. Solamente es utilizado cuando un jugador sale, ya que de otra forma se vuelve fantasma.

- `Object`: Clase madre de todos los objetos físicos del juego. Contiene una posición.
- `Moveable`: Representa a los objetos que pueden ser movidos en el juego. Un método interesante es `Point collision_mask_bound(const Point& next_position)`, que calcula el punto de la máscara que esta en la dirección del punto recibido. Sirve para calcular uno de los puntos en las colisiones.
- `Player`: Representa a un jugador en la partida. Sus métodos son más que nada setters, getters, consultores de estado y métodos con poca relevancia para mostrar en el informe.
- `RectangularObject`: Se utiliza para modelar objetos físicos con forma rectangular, no posee id.
- `CircularObject`: Se utiliza para modelar objetos físicos con forma circular, no posee id.
- `Item`: Para los items no se listarán todos debido a que son muchos. Hay una clase por cada tipo de item que se pide implementar. De esta clase heredan todos los items. Posee el método `void use(Player& user)` que hace que el jugador use el item y método `bool can_be_used_by( const Player& whom)` que determina si puede ser utilizado por el jugador recibido. `PointItem`: Representa los items que otorgan puntos. Implementa los dos métodos nombrados en en `Item`.
- `Map`: Modela un mapa. Contiene los items, los objetos (tanto los que tienen id como los que no), las puertas y pasadizos y los jugadores. El método más desatacable es el de `void add_drop (Player& dead_player)` que se encarga de agregar los items, si se debe, que deja el jugador tras morir.
- `MapLoader`: Se encarga de pasar de un YAML la información a una instancia de la clase `Map`.
- `Mask`: Es una máscara de colisión. Contiene una referencia al centro de su dueño (así esta su posición siempre actualizada). Contiene el método `bool occupies(const Point& where)` que determina si la máscara ocupa en su área el punto enviado.
- `BoxMask`: Representa una máscara rectangular. Hereda de Mask.
- `CircleMask`: Representa una máscara circular. Hereda de Mask.
- `SwitchMask`: Una mascara que contiene otra mascara. Si se encuentra activada entonces `bool occupies(const Point& where)` se lo delega a la máscara interna. Si esta desactivada devuelve false (esta invisible). Un método a destacar es `void switch_mask()`, que cambia el estado de la máscara.

- `CollisionChecker`: Se encarga de todo aquello que se relacione con las colisiones. Con su método `bool is_free(const Point& where)` detecta si el punto pedido esta libre, es decir, no hay paredes ni objetos colisionables. Otro método importante es `bool can_move(const Point& where , Moveable& who)` que se encarga de detectar si un Moveable se puede mover al punto enviado. Y por último se destacará `unsigned int grabbed_item(const Player& by_whom)` que devuelve el item agarrado por el jugador en su posición (o 0 si no hay item).

4.2.3. Diagramas UML

A continuación se mostrará un diagrama de clases del modelo del juego y la partida en el servidor. Se decidió mostrar especialmente este diagrama debido a su importancia y a que sigue una lógica fervientemente orientada a objetos.

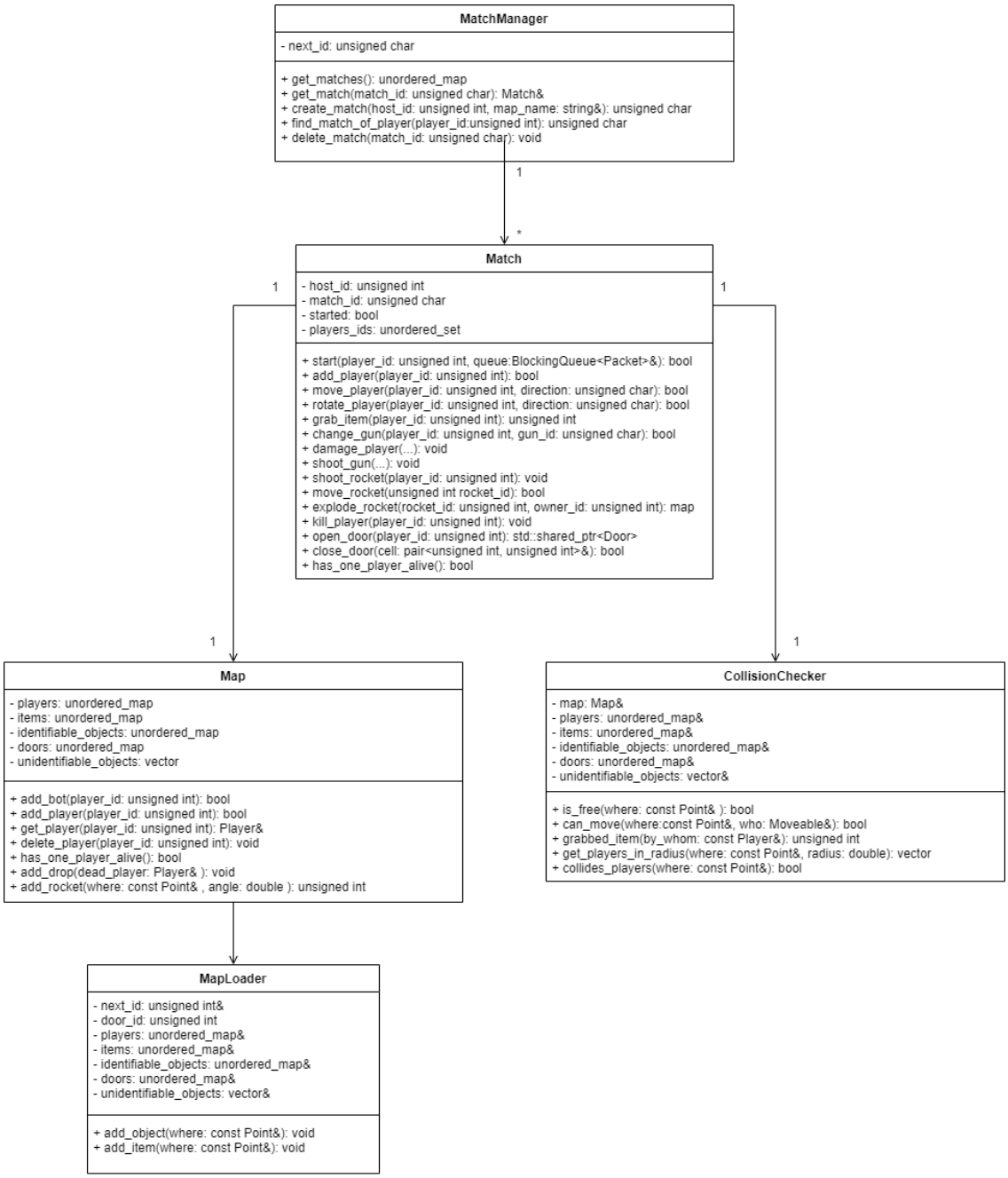


Figura 1: Diagrama de clases de Match.

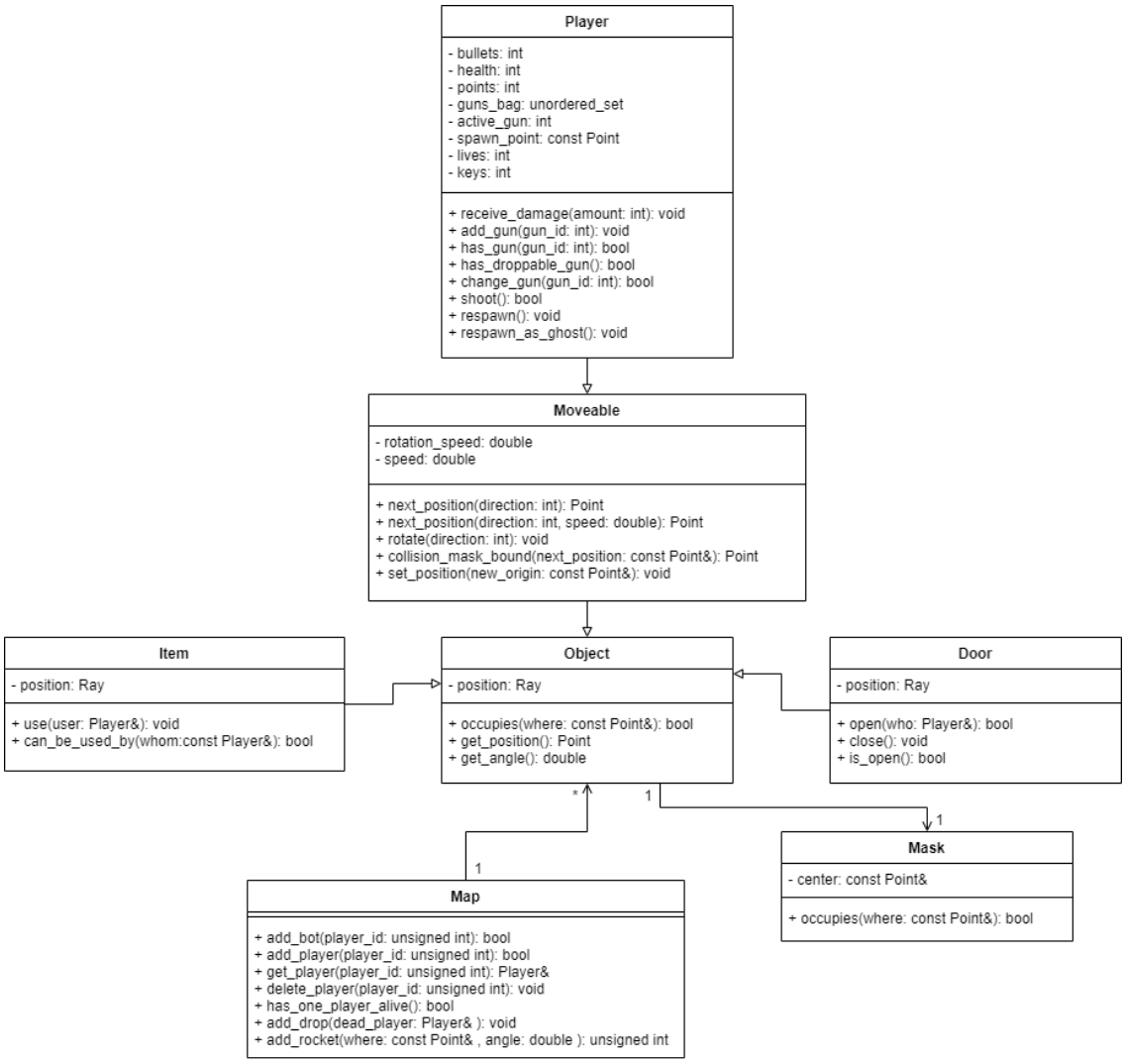


Figura 2: Diagrama de clases de Map.

4.3. Handlers

4.3.1. Descripción general

Se decidió desarrollar por separado un módulo de los handlers. La función de los handlers es simple: se crean dependiendo del paquete recibido (por algún cliente o por los hilos internos del servidor) y poseen un método principal `void handle(Packet& packet, ClientManager& client_manager, MatchManager& match_manager)` que se encarga de ejecutar lo que el paquete solicita. Hay diversos tipos de handlers, en total uno por paquete y se desarrollarán en la siguiente sección. Cabe aclarar que la lógica del funcionamiento de la partida se delega al `Match` como se explicó anteriormente.

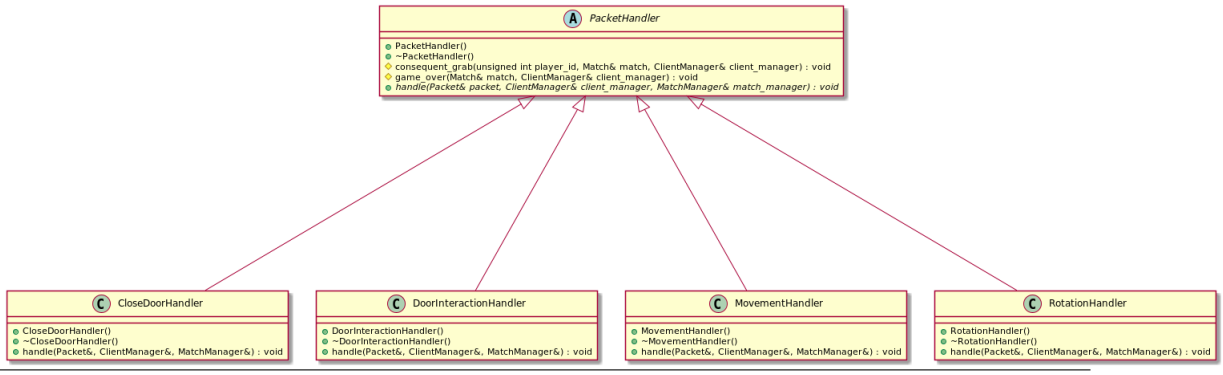
4.3.2. Clases

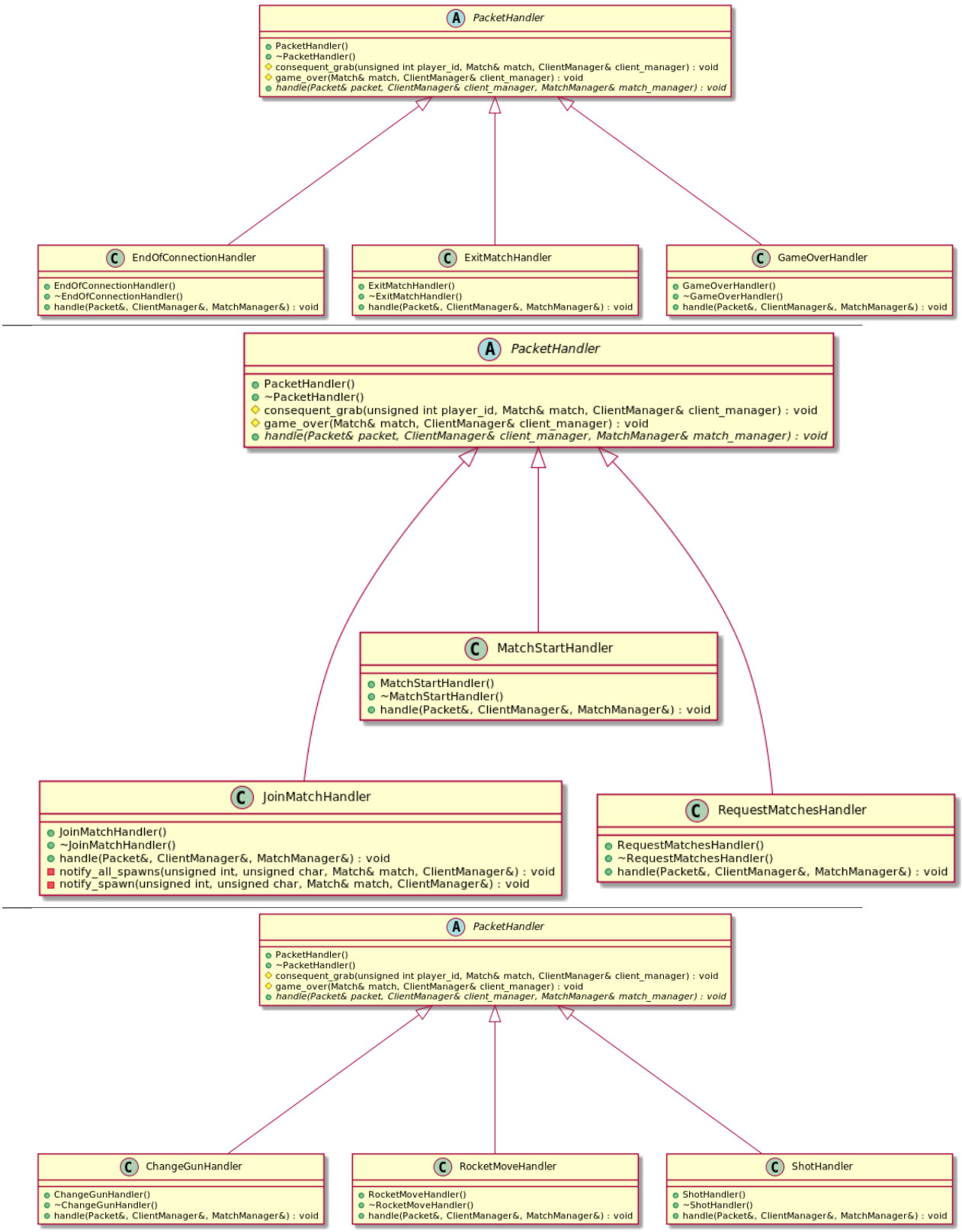
- `PacketHandler`: Clase de la cual heredan todos los handlers. Contiene métodos para situaciones de finalización de partida o de agarrar items, que son utilizados por varias clases hijas.
- `PacketHandlerFactory`: Es una fabrica de handlers. A partir del método `static PacketHandler*`

`build(Packet& packet)` crea un handler correspondiente según el paquete enviado.

- `ChangeGunHandler`: Cambia el arma de un jugador y si pudo reenvia el paquete a los jugadores.
- `CloseDoorHandler`: Cierra una puerta. El paquete lo genera el reloj de la partida.
- `DoorInteractionHandler`: Apertura de una puerta, si es posible. Si pudo se reenvia a los clientes el paquete recibido.
- `EndOfConnectionHandler`: Sirve para notificar el final de una conexión con un cliente.
- `ExitMatchHandler`: Se utiliza para cuando un jugador decide salir en medio de una partida. Se dejan de enviar los paquetes de la partida al jugador que salió.
- `GameOverHandler`: Interpreta lo paquetes de juego terminado, que son enviados internamente por el clock. Se lo envia a los juadores de la partida para notificar su final.
- `MatchStartHandler`: Sirve para comenzar una partida.
- `JoinMatchHandler`: Agrega un cliente como jugador a una partida.
- `MovementHandler`: Se encarga de ejecutar los paquetes de movimiento enviados y, si se pudo mover, reenvía a todos los clientes
- `RequestMatchesHandler`: Sirve cuando un cliente solicita las partidas existentes. Se envían todas ellas al cliente que las solicita.
- `RequestNewMatchHandler`: Crea una partida cuando el jugador envía un paquete para hacerlo.
- `RocketMoveHandler`: Mueve el cohete una posición. Si el cohete no se puede mover lo explota, dañando a los enemigos en un radio y enviado ese daño como paquete.
- `RotationHandler`: Similar al movimiento pero para una rotación.
- `ShotHandler`: Sirve para procesar un disparo. Llama a que la partida produzca un disparo entre los jugadores enviados o crea un cohete si así se necesita. Si el jugador queda sin vida, se encarga de matarlo.

4.3.3. Diagramas UML




PacketHandler
PacketHandler()
~PacketHandler()
consequent_grab(unsigned int player_id, Match& match, ClientManager& client_manager) : void
game_over(Match& match, ClientManager& client_manager) : void
handle(Packet& packet, ClientManager& client_manager, MatchManager& match_manager) : void
EndOfConnectionHandler
EndOfConnectionHandler()
~EndOfConnectionHandler()
handle(Packet&, ClientManager&, MatchManager&) : void
ExitMatchHandler
ExitMatchHandler()
~ExitMatchHandler()
handle(Packet&, ClientManager&, MatchManager&) : void
GameOverHandler
GameOverHandler()
~GameOverHandler()
handle(Packet&, ClientManager&, MatchManager&) : void
PacketHandler
PacketHandler()
~PacketHandler()
consequent_grab(unsigned int player_id, Match& match, ClientManager& client_manager) : void
game_over(Match& match, ClientManager& client_manager) : void
handle(Packet& packet, ClientManager& client_manager, MatchManager& match_manager) : void
MatchStartHandler
MatchStartHandler()
~MatchStartHandler()
handle(Packet&, ClientManager&, MatchManager&) : void
JoinMatchHandler
JoinMatchHandler()
~JoinMatchHandler()
handle(Packet&, ClientManager&, MatchManager&) : void
notify_all_spawns(unsigned int, unsigned char, Match& match, ClientManager&) : void
notify_spawn(unsigned int, unsigned char, Match& match, ClientManager&) : void
RequestMatchesHandler
RequestMatchesHandler()
~RequestMatchesHandler()
handle(Packet&, ClientManager&, MatchManager&) : void
PacketHandler
PacketHandler()
~PacketHandler()
consequent_grab(unsigned int player_id, Match& match, ClientManager& client_manager) : void
game_over(Match& match, ClientManager& client_manager) : void
handle(Packet& packet, ClientManager& client_manager, MatchManager& match_manager) : void
ChangeGunHandler
ChangeGunHandler()
~ChangeGunHandler()
handle(Packet&, ClientManager&, MatchManager&) : void
RocketMoveHandler
RocketMoveHandler()
~RocketMoveHandler()
handle(Packet&, ClientManager&, MatchManager&) : void
ShotHandler
ShotHandler()
~ShotHandler()
handle(Packet&, ClientManager&, MatchManager&) : void

5. Módulos Comunes

5.1. Comunicación de Paquetes

Para permitir un esquema de cliente-servidor, es necesario contar con un protocolo y un módulo que pueda utilizarse para transmitir información desde el cliente hasta el servidor. Para esto se cuenta con 4 partes altamente dependientes entre ellas que llamaremos el módulo de comunicación de paquetes.

5.1.1. Socket

Obviamente se cuenta con un Socket TCP capaz de conectarse, aceptar conexiones, enviar y recibir streams de datos y cerrarse correctamente. Se encuentra implementado en C++ y ante errores graves lanza SocketError, mientras que ante errores menores devolverá -1 permitiendo que el contexto superior decida si es un error o no. El socket es capaz de enviar streams de tamaño fijo y recibir streams de tamaño fijo.

5.1.2. Packet

La clase Packet es tal vez la más importante de todo el módulo de comunicación. Inspirada por el libro *Beej's Guide to Network Programming*, esta clase tiene capacidad de ser identificada en hasta 256 tipos y guardar una cantidad indefinida de datos. El paquete cuenta con 3 partes que son: un tipo (0-255), un tamaño (variable) e información. Se construye a partir de un conjunto de variables empaquetadas en un stream de tipo unsigned char*, a través de una función con firma semejante a un printf(). Ésta es pack(buffer, formato, variables...) y soporta los tipos enteros de hasta 8 bytes y strings. Internamente esta función empaqueta las variables en formato de network, haciendo que no sea necesario usar las funciones host-to-network y su compañera. La función antagonista es unpack(buffer, formato, punteros a variables), que se encarga de hacer el desempaquetamiento de forma inversa. Por convención arbitraria del proyecto, se decidió que el primer byte siempre indique el tipo de paquete.

Se crearon poco más de 20 paquetes que tienen el propósito de enviar información de distinto tipo entre el cliente y servidor, teniendo cada uno un formato específico y un handler específico, como se especificó previamente.

5.1.3. Blocking Queue

La Cola Bloqueante es un elemento importante en la comunicación. Los threads de comunicación (explicados posteriormente) se encargan de encolar paquetes recibidos y desencolar paquetes a enviar. Además de métodos de enqueue() y dequeue() que son bloqueantes, se agregó un poll() inspirados por la Cola Bloqueante de Java que permite desencolar si hay algo para desencolar, sin trabar el hilo.

5.1.4. Threads de Comunicación

Se cuenta con 2 threads con funcionalidades inversas. El primero de ellos es el SendToPeerThread, el cual se encarga de desencolar paquetes de la cola bloqueante de envío cuando hay algo y de enviarlos. Para esto simplemente agarra el paquete, lee su tamaño, codifica su tamaño, lo envía y detrás manda la información del paquete. Esto permite enviar paquetes de tamaño variable, pues el ReceiveFromPeerThread sabe que el primer dato recibido es el tamaño del paquete a recibir. Por lo tanto, como se anticipó, el ReceiveFromPeerThread recibe un tamaño (2 bytes), lo decodifica y recibe exactamente la cantidad de bytes avisada. Luego se encarga de encolar en la cola bloqueante de recepción el paquete recibido, para que ya sea el cliente o el servidor lo procesen.

5.2. Utils

El módulo de útiles provee un conjunto de clases útiles tanto para cliente y servidor que proveen funcionalidades básicas.

- Angle: ángulo normalizado entre 0 y $2 \cdot \pi$.
- BaseMap: mapa con funcionalidades básicas comunes al cliente y servidor.
- Matrix: template de matriz R^{2x2}, usada como base del BaseMap.
- Point: punto en el plano R^2.
- RangeError: error para valores fuera de rango.
- Ray: semirecta modelada como un origen (Point) y un ángulo (Angle).

5.3. Configuration Loader

El Configuration Loader (CL) o ConfigLoader es una clase que carga toda la configuración que puede ser modificada sin necesidad de recompilar. Estos datos se cargan a partir del archivo config.yaml, y refieren a todas las áreas del juego entero. Todos sus atributos son estáticos y la clase es global, permitiendo que todos los módulos puedan acceder libremente a datos que requieran de allí en cualquier momento. Se colocó en el archivo de config.yaml todo aquello que se consideró que sería útil configurar libremente una vez que el juego estuviera completo.

6. Programas intermedios y de prueba

Referencias

[1] Beej's Guide to Network Programming, http://beej.us/guide/bgnet/