Universidad de Buenos Aires
Facultad de Ingeniería

# 96.08- Taller de programación 1

$1^{er}$ Cuatrimestre de 2018

## Trabajo Práctico Final

## Manual de usuario

Grupo 2
Bourbon Navarro, Rodrigo- #96961
Gómez Peter, Federico- #96091
Lidjens, Axel- #95772

# Índice

# 1. Requerimientos de software

El juego fue diseñado para correr bajo un sistema operativo *Debian*, particularmente fue desarrollado en Ubuntu 16.04. Como requisitos para poder compilar el cliente, servidor y el editor se encuentran:

- SDL2 versión 2.0.4.
- SDL2 image versión 2.0.1.
- SDL2 ttf versión 2.0.14.
- SDL2 mixer versión 2.0.1.
- Qt-5 versión 5.6.0.

Cabe mencionar que se utilizan las bibliotecas Box2D como motor físico del juego y jbeder-yaml-cpp para las comunicaciones y la configuración del mismo, que se encuentran ya incluídas.

La compilación e instalación de los programas se hace mediante *CMake* en su versión *3.1.0*, utilizando el compilador *g++* versión *5.4.0*. Como herramienta de *debug* se utilizó *GDB* versión *8.1*.

# 2. Descripción general

El proyecto se encuentra dividido en tres módulos, correspondientes a las aplicaciones de cliente, servidor y editor.

El cliente es una aplicación gráfica que permite conectarse al servidor indicando la ip del mismo y el puerto. Una vez conectado existen dos opciones, crear una partida, o unirse a una existente. En caso de crear una partida, el servidor envía la información de todos los niveles existentes al cliente (nombre y cantidad de jugadores), y una vez que se selecciona uno, envía el archivo correspondiente al mismo, que está en formato *YAML*, y sus fondos (archivos *png*), crea la partida y queda a la espera de que se conecten los jugadores faltantes para inciarla. Si se selecciona la opción de unirse a una partida, el servidor envía al cliente todas las partidas que están disponibles (cantidad de jugadores actual en la partida y cantidad de jugadores necesarios para inciarla), es decir aquellas que no comenzaron aún. Cuando el cliente se une a una partida, el servidor envía el archivo correspondiente al nivel y los fondos del mismo. Cuando se unió el último jugador necesario para iniciar la partida, esta comienza.

El juego transcurre y cuando solo queda un equipo o ninguno (todos perdieron), se muestra en los clientes una pantalla haciendo referencia a si ganaron o perdieron, y cerrando dicha ventana la aplicación termina. Si un cliente se desconecta de la partida, se muestra automáticamente la pantalla indicando que perdió.

Cada vez que una partida finaliza el servidor la remueve.

# 3. Cliente

Como lo explicado en la sección precedente, la aplicación del cliente inicia mostrando una sucesión de pantallas para conectarse al servidor y crear o unirse a una partida. Cada *input* de teclado o mouse que se detecta por parte de un jugador es procesado en caso de que sea el turno del mismo, no se le haya acabado el tiempo, la partida no haya terminado, y el jugador no haya perdido. La acción que debe realizarse en base al *input* del jugador se decide de acuerdo al estado en el que se encuentre el gusano, el cual responde y dicha respuesta es enviada al servidor. Se utilizó el patrón de diseño *State* para modelar todos los posibles estados del gusano. El servidor es el que tiene la lógica del juego, entonces siempre se envía al cliente el estado en el que se encuentran los gusanos. Cada estado tiene asociada una animación y en ciertos casos también un sonido.

Las texturas utilizadas en las animaciones y los sonidos (tanto los efectos de sonido como la música de fondo), se cargan en memoria una única vez al comienzo de la partida a fin de no comprometer la *performance* de la aplicación, ya que el proceso de animar se realiza permanentemente, y cargar las diferentes texturas una y otra vez no sería eficiente, al igual que en lo que respecta a los sonidos.

Para las diferentes armas que ofrece el juego se realizó otro patrón *State*, de modo que cada vez que hay un disparo, cada arma sabe cómo responder.

## 3.1. Desarrollo del juego

El cliente tiene 3 threads:

- El principal (renderer).
- El input worker.
- El output worker.

### 3.1.1. Renderer

Este thread es el encargado de dibujar el juego en la pantalla. Si bien maneja algunas animaciones *client side*, mayormente dibuja el último *snapshot* recibido del modelo. Es importante que este thread no se bloquee en ninguna operación ya que daría la sensación de que el juego no responde. Es por esto que de no haber recibido un *snapshot* nuevo, de todas formas continua ejecutándose y dibujando el último recibido (además de continuar actualizando las animaciones estéticas).

Otra tarea es la de obtener los eventos de *SDL* para procesarlos y actualizar animaciones. Esto no se realiza en un *thread* independiente porque no es necesario ya que un ser humano no tiene la velocidad de generar demasiados eventos en una iteración de forma que se pudiera demorar procesándolos. Esto resulta una ventaja porque de esta forma no es necesaria la utilización de *threads* (y *mutex*) en el *render loop*, minimizando los posibles errores y la complejidad del mismo.

### 3.1.2. Input worker

Este *thread* obtiene de la conexión con el servidor los nuevos *snapshots*. De forma similar a como trabaja el servidor, este *thread* almacena los snapshots en un `DoubleBuffer`, con la diferencia de que el `Renderer` no se bloquea esperando un `swap`, sino que siempre obtiene una copia (aunque sea una ya leída previamente).

### 3.1.3. Output worker

El cliente debe enviar las acciones del mismo al servidor mediante el *socket*. Para evitar bloquearse en un `send`, esta tarea es realizada por un *thread*. La comunicación entre este *thread* y el *renderer* se realiza mediante un `Stream`. El *renderer* hace `push` y este *thread* hace un `pop` bloqueante para evitar un *busy wait*. Cada acción es luego serializada y enviada al servidor.

En la figura 1 muestra la comunicación entre un cliente y el servidor. Cada flecha vertical representa un *thread* en el proceso en el cual se origina, mientras que las flechas horizontales indican comunicación de alguna forma:

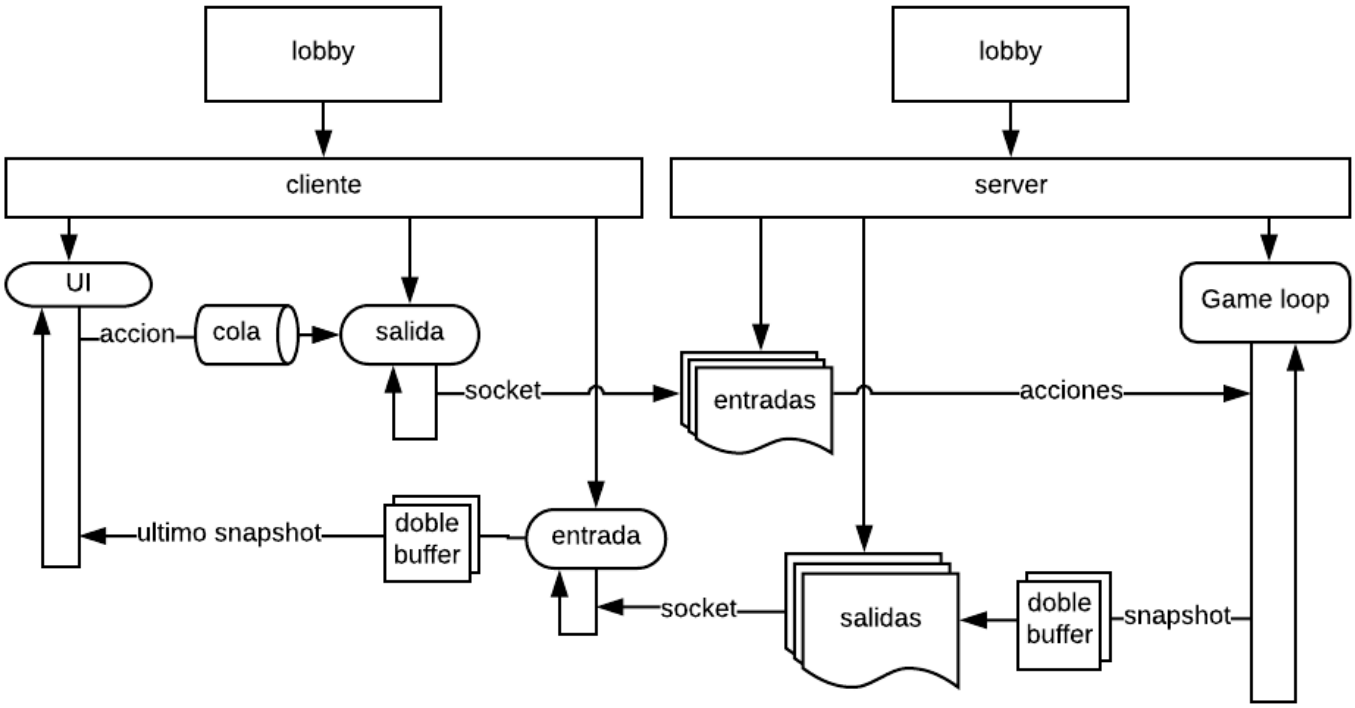


**Figura 1:** Comunicación entre cliente y servidor

## 3.2. Estructura general

El renderer repite unas etapas básicas:

- Actualiza el snapshot del juego.
- Maneja el input del usuario.
- Actualiza su estado.
- Renderiza.

Las primeras 2 etapas fueron descriptas previamente. La actualización del estado consiste en propagar un evento de `update`. Esto simplemente es notificar a cada objeto que se realiza una nueva iteración, proveyendo además el tiempo transcurrido desde la última llamada a update. De esta forma se logra animar la interfaz gráfica de forma tal que la velocidad de las animaciones no varíe con la del procesador. Cada objeto es responsable de calcular el tiempo transcurrido y actuar en base a él (o tomar cualquier acción necesaria). También se actualiza el estado de los elementos basado en el snapshotdel modelo, ya sea actualizando las posiciones de los worms, las balas o el turno del jugadr, etc.

Luego, la fase de render dinuja en pantalla el modelo ya actualizado. Para esto se propaga un `render` en el cual un objeto `Camera` actúa como visitor. Cada objeto visitado provee una textura para dibujar, así como la posición donde hacerlo. La cámara luego se encarga de realizar la transformación de coordenadas correspondiente y llamar a SDL para renderizar la imagen final.

El juego maneja 2 tipos de coordenadas:

- Globales (o de juego).
- Pantalla (o screen coordinates).

Las coordenadas de juego son las que maneja el modelo, en metros. Por otro lado, las de pantalla se manejan en pixels. La cámara tiene un factor de conversión pixels/metro y una posición que utiliza para convertir las coordenadas de juego en coordenadas de pantalla. De esta forma se logra abstraer este tipo de lógica a la hora de dibujar el modelo.

## 3.3. Clases

Primero se describen las clases que son utilizadas tanto por el cliente como por el servidor.

- ***Direction***: clase *enum* que indica la dirección del gusano (derecha o izquierda).
- ***DoubleBuffer***: se utiliza para enviar (en el servidor), y recibir (en el cliente), los *snapshots* con la información del estado del juego.
- ***EnumClassHash***: es una estructura que define el operador *()* para tipos enumerativos.
- ***Exception***: se adaptó la clase *OSError*, usando funciones estándar de C++11, para hacer una clase genérica de excepción, la cual recibe el formato de la cadena de texto y los argumentos para completar el formato.
- ***Stream***: es una clase *template* que encapsula una cola que puede ser utilizada como bloqueante. El parámetro del *template* es el mensaje que se va a enviar por la misma. Sus métodos son *push*, *pop* y *close*, que impide el uso de la cola ya que al hacer *pop* lanza una excepción. Tiene sobrecargados los operadores << y >> para hacer *push* y *pop* de manera bloqueante respectivamente.
- ***StateID***: clase *enum* que posee todos los estados del gusano.
- ***WeaponID***: clase *enum* que posee todos los estados de las armas.
- ***PlayerInput***: clase *enum* que posee todos los *inputs* que puede generar el usuario.
- ***PlayerMsg***: es una estructura que posee un *PlayerInput* y una posición (cuando se produce un evento de click con el mouse en la utilización de un arma que lo requiera) y que envía el cliente al servidor.
- ***GameStateMsg***: es una estructura que posee todos los elementos que conforman el estado del juego en un momento dado, el cual el servidor envía a los clientes. Sus métodos son *serialize*, y *deserialize*, que se encargan de procesar los datos para enviarlos y recibirlos adecuadamente de una forma portable.
- ***LevelInfo***: es una estructura que posee la información correspondiente a un nivel (*id*, nombre y cantidad de jugadores). La envía el servidor al cliente.
- ***GameInfo***: es una estructura que posee la información correspondiente a una partida ya creada (*id* de la partida, *id* y nombre del nivel asociado a la misma, cantidad actual de jugadores y cantidad total de jugadores necesaria para comenzar la partida). La envía el servidor al cliente.
- ***Event***: es una clase *enum* que posee todos los eventos que pueden suceder en el juego y fuera del mismo, tanto en el cliente como en el servidor.
- ***Subject***: es una clase que posee un *set* de punteros a *Observer*. Sus métodos son *addObserver* y *removeObserver* para agregar o quitar observadores al sujeto, y *notify*, el cual recibe a dicho sujeto como referencia y el evento que este quiere notificar.
- ***Observer***: es una interfaz cuyo único método es *onNotify*, que recibe un *Subject* como referencia y un *Event* que este notifica.
- ***Point***: es una clase *template* que define un punto de coordenadas *(x, y)* del tipo numérico indicado en el *template* y define las operaciones de suma, resta, multiplicación, división y los operadores == y ! =, además de la distancia entre dos puntos.
- ***Socket***: implementación según el paradigma de objetos socket. Es una implementación para que las clases hijas (ServerSocket, CommunicationSocket y ClientSocket) implementen y se utilicen de forma RAII la comunicación entre equipos. En la figura 2 puede verse un diagrama de clase de estas clases mencionadas. Sabe como destruirse y también como construirse por movimiento (también asignación por movimiento). Sus métodos son *close*, que cierra el *file descriptor* y *shutdown*, que cierra la comunicación bidireccionalmente. El método close es protegido y solo puede usarse internamente en alguna de las hijas. También se usa internamente en el destructor.
- ***Protocol***: es una clase *template* donde el parámetro del *template* es el tipo de *socket* que utilizará. Establece un protocolo de comunicación entre cliente y servidor. Tiene sobrecargados los operadores >> y << para diversos tipos de datos, y los métodos *getSocket*, que devuelve por movimiento el *socket* dejando inutilizado el protocolo, y *stopCommunication*, que para la comunicación del *socket* mediante un *shutdown*.

- ***Thread***: es una clase abstracta que encapsula un hilo. Sus métodos son *start* para lanzar el hilo y *join* para terminarlo. Todas las clases que hereden de esta deben implementar los métodos *run*, donde se define lo que se desea que el hilo haga, y *stop*, para terminar su ejecución de manera ordenada si es necesario.
- ***GirderData***: es una estructura utilizada en *Stage* para almacenar la información correspondiente a las vigas (largo, alto, ángulo y posición).
- ***WormData***: es una estructura utilizada en *Stage* para almacenar la información correspondiente a los gusanos (vida y posición).
- ***Color***: es una estructura utilizada en *Stage* para almacenar la información correspondiente a un color (sus componentes *r*, *g* y *b*).
- ***Stage***: carga la información de un nivel desde un archivo de configuración en formato *YAML* con el método estático *fromFile* utilizando las funciones estáticas *_parsePoint*, *_parseWorm* y *_parseGirder*, y luego devuelve un *Stage* con los datos ya cargados. Se construye creando un mapa *unordered map* para asociar los nombres de las armas con sus *ids*. El método *getAmmoCounter* devuelve una referencia constante a un mapa donde se almacena la cantidad de municiones que cada jugador dispone de cada arma en el nivel.

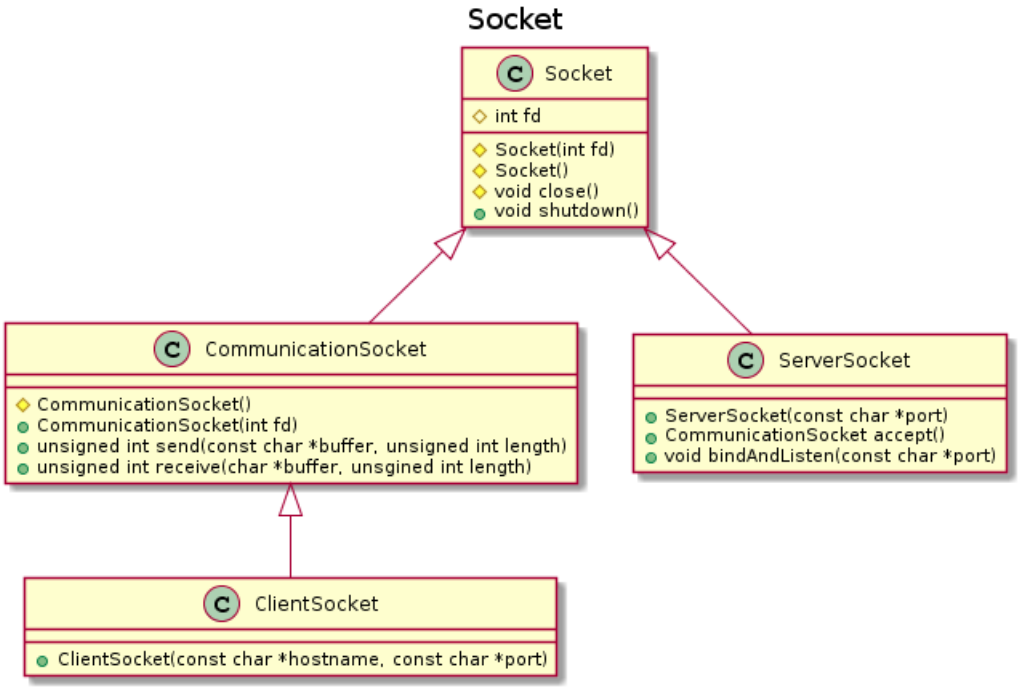


**Figura 2:** Diagrama de clase de Socket

A continuación se describen las clases propias del cliente:

- ***Window***: se construye a partir de un ancho y alto fijos o personalizados, e inicia todos los recursos de *SDL* que se utilizarán. Al destruirse llama al método *close*, el cual se encarga de liberar todos los recursos adquiridos. Se destacan los métodos *clear*, que pone la ventana en un color determinado por parámetro o en blanco por defecto, *containsMouse*, que indica si el mouse está contenido en la ventana, *maximize*, que maximiza la ventana, *getRenderer*, que devuelve el renderizador de la ventana, y *render*, que renderiza el contenido de la ventana.
- ***Animation***: se construye a partir de una textura y se encarga de renderizar la misma de acuerdo a un *framrate* de 25 *frames* or segundo en la posición que se le indica. Puede renderizarse en *loop* desde el *frame* incial hasta el último y a continuación el primero nuevamente, en *loop* llegando al final y volviendo al comienzo (indicado con el *flag playReversed*), animarse una única vez quedando en el último *frame* (indicado con el *flag playOnce*), animarse en sentido inverso (indicado con el *flag playInverse* y utilizado para la teletransportación) o puede setearse un *frame* manualmente (para el caso en el que se esté apuntando un arma por ejemplo). La actualización del *frame*

que debe renderizarse se hace mediante el método *update*, el cual recibe el tiempo que ha pasado desde el último cambio. Cuando este tiempo acumulado supera el *framrate* se realiza el cambio de *frame*.
Los métodos principales son el ya mencionado *update*, el método *render*, que recibe la posición en donde debe renderizarse, el modo de *flip*, y la cámara del juego que es la que calcula las coordenadas y la muestra en pantalla. Finalmente el método *advanceFrame* es el que se encarga, en base a los *flags* que se encuentren seteados, de establecer el siguiente *frame* que debe ser animado.

- ***Camera***: se construye en base a la ventana donde se renderizará, la relación pixels/metro deseada y el ancho y alto del área a donde la cámara puede ir. Se destacan los métodos *isMoving* para saber si la cámara está en movimiento a la hora de decidir qué renderizar, *update* que se encarga de actualizar la posición de la cámara, y luego los métodos *draw* y *drawLocal* que dibujan en pantalla una textura o un rectángulo que reciben por parámetro.

- ***Texture***: encapsula la creación y liberación de recursos correspondiente a una textura. Únicamente devuelve el puntero correspondiente a la textura, su alto o su ancho.

- ***Font***: encapsula la creación y liberación de recursos correspondiente a una fuente de texto. Solo devuelve un puntero correspondiente a la fuente.

- ***Text***: se construye a partir de una fuente. Se le setea el texto deseado e internamente crea una textura con el mismo, la cual es renderizada con o sin fondo. El método *render* es el encargado de dibujar el texto en pantalla de acuerdo a una posición y la cámara recibidos por parámetro.

- ***WrapTexture***: se crea a partir de una textura, un ancho y un alto. Solapa la misma textura o la recorta en base a las dimensiones de la misma y al ancho y al alto especificados. Se destaca el método *render*, que dibuja la textura en pantalla en la posición deseada y el método *render* que hace lo propio pero especificando también un ángulo.

- ***Button***: representa un botón y se crea a partid de una posición, un ancho y un alto. Puede seteársele un mensaje y el color de texto y de fondo. Se destaca el método *inside* que establece a partir de la posición en la que el usuario hizo click recibida por parámetro si esta se encuentra dentro del botón, y *render*, que a partir de una *cámara* recibida por parámetro lo dibuja en pantalla.

- ***GameWindow***: hereda de *Subject*. Es una interfaz para todas las ventanas que posee la aplicación y comunica las acciones realizadas en estas a un observador que será *LobbyAssistant*, que se describirá luego. Se construye a partir de la ventana del juego, una fuente de texto y una cámara. Define una estructura *TextField* que procesa los *inputs* de texto del usuario, y tiene un vector de botones *Button*. Posee los métodos *handleKeyDown* (responde a *inputs* del usuario), *appendCharacter* (responde a *inputs* de texto del usuario), *buttonPressed* (responde en el caso que un usuario presione un botón), y *render* (dibuja todo lo que deba dibujarse en la ventana). Las clases que implementan esta interfaz son:

  - ***ConnectionWindow***: tiene dos *TextField* donde permite al usuario ingresar la *ip* y el puerto del servidor al que desea conectarse y un botón que al presionarlo crea la conexión.
  - ***SelecActionWindow***: tiene dos botones con los cuales permite al usuario elegir entre crear una partida o unirse a una existente.
  - ***CreateGameWindow***: permite crear una partida. Muestra en pantalla un nivel con su nombre y cantidad de jugadores y tiene tres botones, dos para alternar entre los niveles disponibles (anterior y siguiente), y otro para seleccionar el nivel.
  - ***JoinGameWindow***: permite unirse a una partida. Muestra en pantalla una partida con la cantidad de jugadores que hay actualmente en dicha partida y la cantidad de jugadores que debe haber para comenzar. Tiene tres botones, dos para alternar entre las partidas disponibles (anterior y siguiente), y otro para seleccionar la partida.
  - ***WaitingPlayersWindow***: una vez que se seleccionó el nivel a crear o se eligió la partida a unirse, se muestra una pantalla con la cantidad actual de jugadores conectados y la cantidad necesaria para que comience el juego. Cuando esta cantidad es alcanzada, el juego comienza automáticamente.
  - ***GameEndWindow***: al finalizar el juego o el jugador desconectarse, se muestra en pantalla un mensaje haciendo alusión a si ganó o perdió.

- ***ClientGUIInput***: clase *enum* que posee las acciones que el cliente puede realizar.
- ***ServerResponseAction***: clase *enum* que posee las acciones que puede indicarle el servidor al cliente.
- ***ClientGUIMsg***: es una estructura que posee un *ClientGUIInput* y que utiliza *LobbyAssistant* para comunicar acciones a *CommunicationProtocol*, que serán descriptas luego.
- ***ServerResponse***: es una estructura que posee un *ServerResponseAction* y que utiliza *CommunicationProtocol* para comunicar acciones a *LobbyAssistant*, que serán descriptas luego.
- ***ClientSocket***: es un socket que hereda de *CommunicationSocket* y tiene la capacidad de realizar una conexión con el servidor, partiendo del dato del *host* y el puerto a donde conectarse. No posee métodos propios, solo su constructor que es donde se realiza la conexión.
- ***CommunicationProtocol***: es un hilo, hereda de la clase *Thread* y se construye a partir de un *ClientSocket* que se utiliza para inicializar un atributo correspondiente a un protocolo *Protocol* y dos *Stream*, uno para recibir mensajes de la interfaz gráfica (*ClientGUIMsg*) y otro para enviar mensajes a la interfaz (*ServerResponse*). El *stream* de mensajes del cliente se utiliza como cola bloqueante ya que el hilo no realiza ninguna acción a menos que el cliente lo requiera. Hace de interfaz entre el servidor y el cliente. Posee atributos públicos que son modificados por los datos provenientes del servidor para luego ser leídos por el cliente, o bien para ser modificados por el cliente y posteriormente enviados al servidor. Posee los métodos *run*, donde espera un mensaje del cliente, *handleClientInput*, donde toma la decisión de qué realizar en base al requerimiento del cliente, y luego métodos correspondientes a las acciones que puede realizar dicho cliente. Estos son:
  - *startCreateGame*: envía el comando correspondiente al servidor y recibe la información de los niveles disponibles.
  - *createGame*: envía el comando correspondiente al servidor y el nivel elegido. Recibe el archivo de configuración del nivel y sus fondos y luego espera el comienzo del juego en *waitGameStart*.
  - *startJoinGame*: envía el comando correspondiente al servidor y recibe las partidas disponibles.
  - *joinGame*: envía el comando correspondiente al servidor, la partida elegida y el nivel que asociado a la partida elegida para luego recibir su archivo de configuración y fondos y luego espera el comienzo del juego en *waitGameStart*.

  También posee el método *waitGameStart*, que espera hasta que la partida alcance el número de jugadores necesario para empezar y avisa al cliente cuando esto sucede. El método *getLevelFiles* se encarga de recibir y guardar en el cliente el archivo de configuración del nivel y sus fondos. Finalmente, el método *getSocket* remueve el *socket* del protocolo mediante *move semantics* (se utiliza para dárselo al juego una vez que este comienza), y le método *stop* para la ejecución del hilo y la comunicación del protocolo para un cierre ordenado.
- ***LobbyAssistant***: hereda de *Observer*, se construye con una ventana *Window* e internamente crea una cámara, una fuente de texto y una ventana de tipo *GameWindow* que cambiará de acuerdo a las acciones del usuario. Se encarga de manejar la lógica de las ventanas iniciales y crea un *CommunicationProtocol* cuando el usuario se conecta al servidor. Posee *streams* de tipo *ClientGUIMsg* para enviar mensajes al hilo del protocolo de comunicación y *ServerResponse* para recibir su respuesta (utilizado como cola no bloqueante), que se procesa en el método *handleServerResponse*. Se destaca el método *run*, donde se reciben *inputs* del usuario, se renderiza la ventana, se realiza un cambio de ventana si es necesario y se procesan respuestas del servidor en caso de haberlas. También se destaca el método *onNotify*, que corresponde a notificaciones de las ventanas y se toma la decisión de qué hacer en base a la interacción del usuario con las mismas. Finalmente, el método *getSocket* devuelve el *socket* obtenido del protocolo de comunicación mediante *move semantics*.
- ***Worm***: esta clase representa al gusano y se construye con un *id* que lo identifica, un *GameTextureManager* y un *SoundEffectManager* ya que de acuerdo a su estado se renderizará con distintas texturas y reproducirá sonidos. Se destacan los métodos *handleKeyDown*, *handleKeyUp* y *mouseButtonDown* para procesar *inputs* del usuario, y *update* que actualiza el estado, la animación, el arma, la explosión asociada a esta si existe y el efecto de sonido si este correspondiera. El método *setState*, que setea su estado con la información proveniente del servidor, y *getAnimation* y *playSoundEffect*, que establecen la textura a renderizar y el sonido a reproducir en base al estado. El método *setWeapon* establece el arma a utilizar y *playWeaponSoundEffect* su efecto de sonido asociado. Finalmente, *startShot* y *endShot* realizan la lógica del disparo de acuerdo a cómo responde el arma.

- ***State***: representa el estado del gusano. Todos los *inputs* del jugador están representados en métodos (*moveLeft*, *moveRight*, *jump*, *bazooka*, *startShot*, etc.), y cada estado sabrá responder en consecuencia. Se destaca el método que devuelve el *id* del estado. Las clases que implementan esta interfaz son:
  - ***Walk***.
  - ***Still***.
  - ***StartJump***.
  - ***Jump***.
  - ***EndJump***.
  - ***BackFlip***.
  - ***BackFlipping***.
  - ***EndBackFlip***.
  - ***Hit***.
  - ***Die***.
  - ***Dead***.
  - ***Drowning***.
  - ***Falling***.
  - ***Land***.
  - ***Sliding***.
  - ***Teleporting***.
  - ***Teleported***.
  - ***Batting***.
- ***SoundEffect***: carga un efecto de sonido para ser reproducido luego. Sus métodos son *getChunk*, que devuelve el puntero al efecto de sonido cargado, y *play*, que lo reproduce una vez o en *loop* de acuerdo al *booleano* que recibe por parámetro.
- ***BackgroundMusic***: carga un archivo de música para utilizarlo como fondo en el juego. Sus métodos son *getMusic*, que devuelve el puntero al archivo de música cargado, y *play*, que lo reproduce en *loop*.
- ***TextureManager***: es un template que permite guardar texturas en un *unordered map* con un *hash* redefinido.
- ***SoundEffectManager***: idéntico funcionamiento que el *TextureManager* salvo que ahora se almacenan efectos de sonido en vez de texturas, y ya no es necesario el renderizador.
- ***BackgroundMusicManager***: idéntico funcionamiento que el *SoundEffectManager* salvo que ahora se almacenan archivos de música de fondo en vez de efectos de sonido.
- ***SoundEffectPlayer***: se construye con un *SoundEffect* obtenido del *SoundEffectManager* y opcionalmente con la duración que se desea del efecto de sonido o si debe actualizarse automáticamente. Sirve de interfaz para la reproducción de efectos de sonido. Se puede establecer mediante un atributo si se desea reproducir el efecto de sonido en *loop*. Sus métodos son *play*, que reproduce el efecto de sonido de acuerdo al valor del atributo *loop*, y *update*, que recibe el tiempo transcurrido desde la última actualización y si no se eligió la actualización automática acumula dicho tiempo. Si este acumulado supera la duración establecida del efecto lo reproduce nuevamente y vuelve el acumulado a cero.
- ***BackgroundMusicPlayer***: se construye a partir de un *BackgroundMusic* obtenido del *BackgroundMusicManager* y reproduce el archivo de música mediante el método *play*.
- ***Armory***: se construye a partir de un *GameTextureManager*, que es un *TextureManager* cuyo *id* es de tipo *GameTextures* (clase *enum* con los *ids* de cada textura utilizada), y también a partir de la cámara del juego y su fuente de texto (ambas referencias). Se encarga de renderizar los íconos de las armas del juego indicando la cantidad de municiones de cada una que le quedan por utilizar al jugador. Posee los métodos *loadWeapons*, donde se cargan en un vector las texturas correspondientes a los íconos de las armas del juego, *update*, que actualiza las municiones de las armas que le quedan al jugador, y *render*, que dibuja en pantalla los íconos con la cantidad de municiones y las teclas para utilizar cada arma (*F1 - F10*).

- ***Wind***: se construye a partir de un *GameTextureManager* y una cámara, y mediante el método *render* dibuja en pantalla la dirección del viento con un tamaño de acuerdo a la intensidad del mismo.
- ***Water***: se construye a partir de un *GameTextureManager*. Su método *render* dibuja en pantalla la textura del agua, y el método *update* realiza el efecto de animación de la misma.
- ***Explosion***: se construye a partir de un *GameTextureManager* de donde se obtiene la animación necesaria. En el método *render* se renderiza la animación y en el método *update* se actualiza, seteando un *flag* de acuerdo a si esta terminó. Dicho *flag* es devuelto en el método *finished*.
- ***Bullet***: se construye a partir de un *GameTextureManager* de donde se obtiene la animación necesaria de acuerdo al *id* del arma que la disparó, y de un *GameSoundEffectManager* para reproducir el sonido de la explosión. El método *setAngle* setea el ángulo de la bala para actualizar luego la animación en base a este. Con el método *madeImpact* el *Game* le indica a la bala que hizo impacto, por lo que setea un *flag* indicándolo y reproduce el sonido de la explosión. En *render* y *update* se renderiza y actualiza respectivamente la animación de la bala si esta no explotó, o la explosión (de clase *Explosion*, que posee internamente la bala) en caso contrario.
- ***Scope***: se construye a partir de un *GameTextureManager* de donde se obtiene la animación necesaria. El método *setAngle* setea el ángulo en el que apunta el gusano, que luego se utiliza en el método *render* para dibujar la mira en la posición correcta. El método *update* actualiza la animación.
- ***PowerBar***: se construye a partir de un *GameTextureManager* de donde se obtienen las animaciones necesarias. El método *setAngle* setea el ángulo en el que apunta el gusano, que luego se utiliza en el método *render* para dibujar la barra en la posición correcta. El método *update* agrega animaciones para simular el cargado de la barra conforme pasa el tiempo. Los métodos *startShot* y *endShot* determinan cuándo debe renderizarse.
- ***Weapon***: es una clase abstracta que encapsula el comportamiento de las armas. Las clases que heredan de esta se construyen en base a un *GameTextureManager*, el *id* de una textura y el *frame* en que debe setearse (según el ángulo en el que está apuntando el gusano, que luego será alterado por el método *setAngle*). Según el arma, puede poseer mira (*Scope*) y disparar con potencia variable (*PowerBar*). El método *positionSelected* se utiliza para animar el ataque aéreo, y los métodos *startShot* y *endShot* comienzan y terminan la animación de la barra de potencia respectivamente. En el método *update* se actualizan y en el método *render* se dibujan: la mira, la barra de potencia (si estas existen), y la animación del arma. Las clases de armas que heredan de esta son:
  - ***AerialAttack***.
  - ***Banana***.
  - ***BaseballBat***.
  - ***Bazooka***.
  - ***Cluster***.
  - ***Dynamite***.
  - ***Grenade***.
  - ***Holy***.
  - ***Mortar***.
  - ***Teleport***.
  - ***WeaponNone***.
- ***Game***: es la clase donde se desarrolla el juego. Uno de los parámetros que recibe al construirse es el número de equipo asociado al jugador, que se utilizará para decidir si se aceptan *inputs* del mismo. Su método *start* tiene el ciclo que se repite hasta que la partida termina o el jugador se desconecta, y en el cual se actualiza el manejo de la cámara mediante *handleCamera*, se actualizan los gusanos, la cámara, el agua, y la/s bala/s si existen mediante *update*, y se renderiza mediante *render*. Se destacan los métodos *loadTextureManager*, *loadSoundManager* y *loadBackgroundManager*, donde se cargan las texturas, efectos de sonido y la música de fondo respectivamente.

# 4. Servidor

Continuando con la descripción de los módulos, se verá ahora en detalle el servidor. Este comienza creando *GameLobby*, que se encargará de aceptar conexiones que se realicen con el servidor, creando un *GameLobbyAssistant* para cada uno. Esta clase recibirá los comandos que envíe el cliente luego de haber tenido una conexión exitosa. Las opciones que puede realizar son crear una partida, obtener los niveles disponibles, ingresar a una partida creada y obtener una lista de partidas creadas. Todo esto sucede en *threads* separados. Tanto el *GameLobby* como cada *GameLobbyAssistant* realizan sus tareas en hilos separados, el primero para poder aceptar clientes y dejar el hilo principal para recibir el comando por *stdin* necesario para comenzar el proceso de cerrado ordenado del servidor, y los segundos para que el primero pueda aceptar sin rechazar conexiones durante el lapso que el cliente tarda en comenzar una partida.

Cuando un cliente decide crear una partida, se creará una nueva instancia de la clase *Lobby*, por medio del uso de la clase *Lobbies*. La clase *Lobbies* es una clase de importancia ya que es el recurso compartido que relaciona todas las conexiones que se realicen al servidor. En esta se guardan todas las partidas creadas. Dado que varios clientes distintos podrían querer conectarse a la misma sala de juego, esta también posee una *race condition* que debe ser tenida en cuenta. *Lobbies* opera como un monitor, que realiza las operaciones de crear partida, unirse a una partida y obtener los juegos creados de forma atómica. Para esto, dispone de un *mutex* de protección. *Lobbies* posee internamente un arreglo de *Lobby*, que tiene un registro de los clientes. Cuando la sala se completa, notifica al *GameLobby* que la partida puede comenzar. Este inmediatamente dota al *Lobby* de los *sockets* de cada cliente, para que este pueda iniciar en un hilo propio la partida. Es en este momento también que sucede la finalización de los *GameLobbyAssistant* involucrados. La liberación de los recursos de estas instancias (su destrucción), la realiza el *GameLobby*, quien revisa luego de aceptar una conexión todos los hilos que terminaron, aplicando su correspondiente *join* y su destrucción.

La partida transcurre en la clase *Game*. Esta fue pensada en un principio como una clase que iba a heredar de *Thread*, sin embargo, se delegó esa herencia en el *Lobby* que lo contiene. En esta clase se creará el mundo físico y se recibirán las interacciones que tenga el usuario con el cliente, para modificar este mundo en la medida de lo posible.

Una vez que el juego termina, ya sea porque terminó normalmente, o porque quedó un solo jugador conectado, se debe proceder a realizar un *join* del hilo. De esto se encarga el *LobbyJoiner*. Este proceso, que opera en un hilo aparte, se encarga de iterar sobre los *Lobby* terminados, para realizar un *join* y eliminarlo del arreglo. A primera vista, pareciera que este ciclo ocurre indefinidamente, pudiendo consumir una cantidad de recursos considerable de la computadora. Sin embargo, el *GameLobby* se comunica con esta clase mediante una cola bloqueante. Esta le manda mensajes al *LobbyJoiner* cuando una partida termina, para que este se active y libere el recurso. Un diagrama de secuencia correspondiente al proceso de aceptación de una conexión por parte del *GameLobby* y de la creación de una partida se muestra en la figura 3, mientras que en la figura 4 se aprecia un diagrama de secuencia del proceso de unirse a una partida y el comienzo de la misma.

## 4.1. Desarrollo del juego

Las partidas las maneja el objeto `Game`. Al crear el juego, debe recibir los jugadores (sockets) y el `Stage` que debe crear. Este objeto consta de varios threads:

- El gameloop.
- Un thread que lee de la entrada de cada cliente.
- Un thread que envía los datos para cada cliente.

## 4.2. Gameloop

El game loop obtiene y ejecuta las acciones del jugador al cual le corresponde el turno, actualiza el motor de física, serializa el nuevo estado del juego en un snapshot y duerme el tiempo necesario. El servidor es totalmente autoritario, es decir, no acepta órdenes por parte del cliente, sino que recibe acciones y es él el que decide si son aceptadas o no. El gameloop se actualiza en un framerate determinado (60 veces por segundo), con lo cual es importante que pueda trabajar sin bloquearse en ninguna operacion, ya que afectaría a todos los jugadores y arruinaría la experiencia de juego.

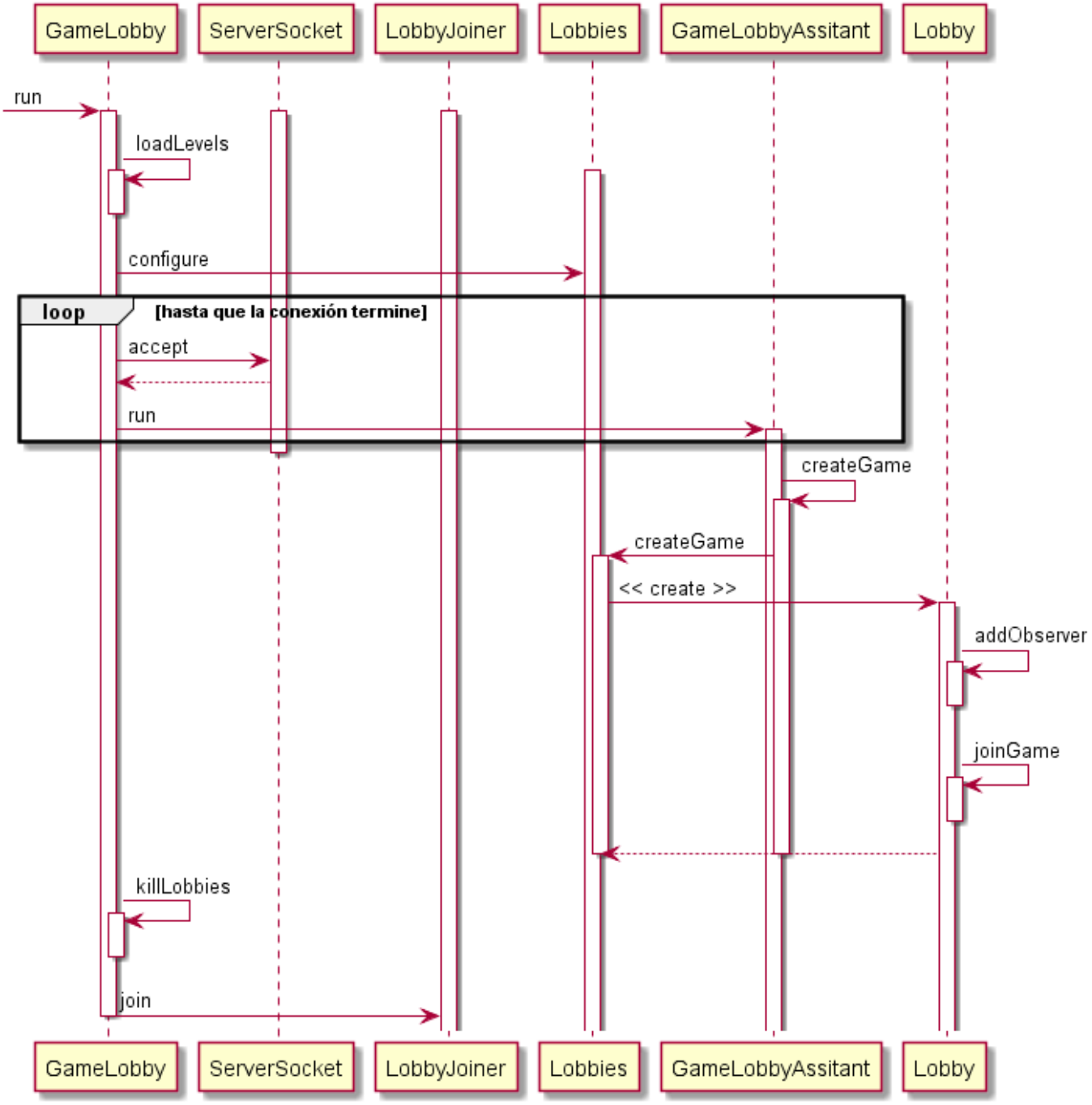


**Figura 3:** Diagrama de secuencia de la aceptación de una conexión y la creación de una partida.

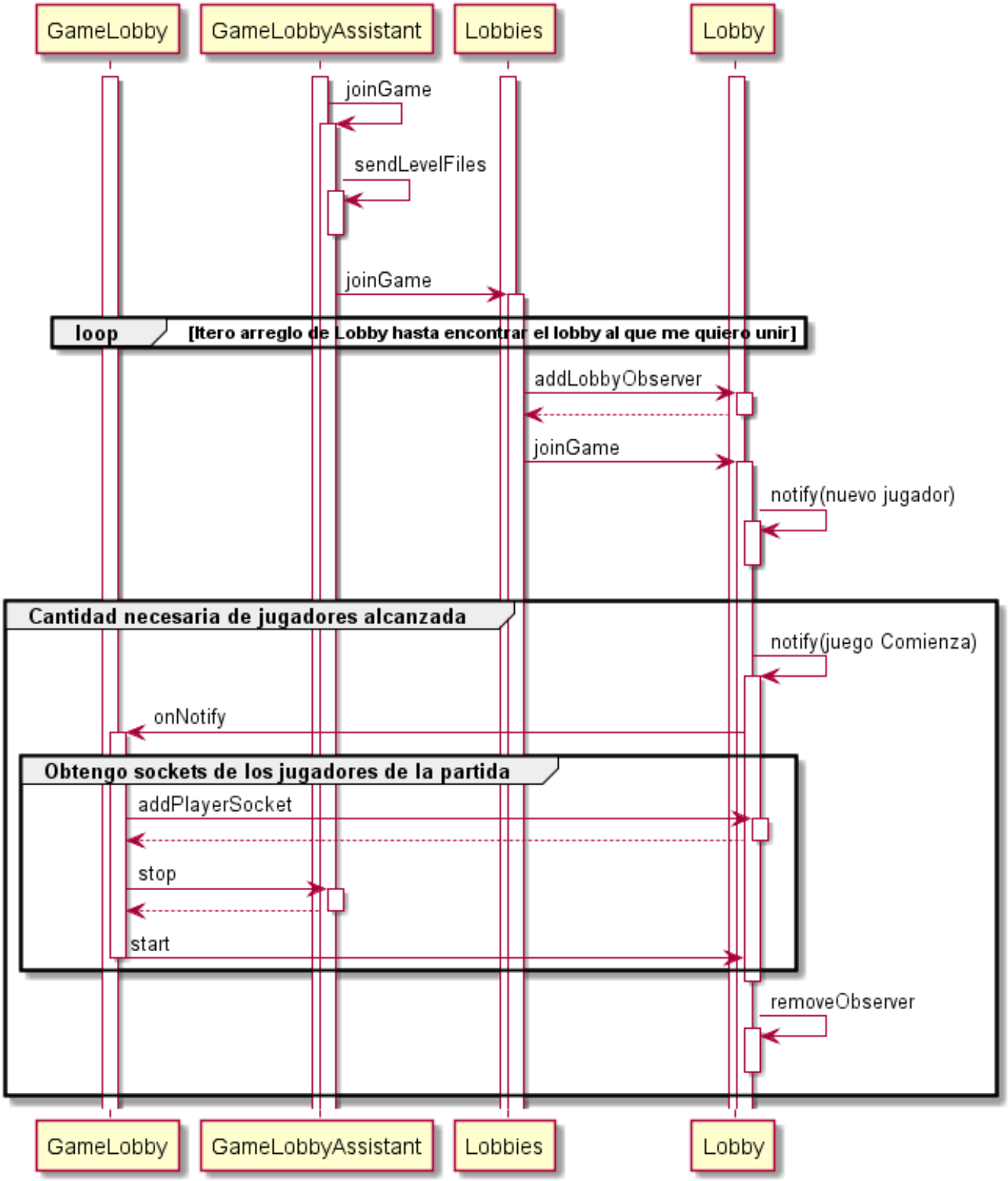


**Figura 4:** Diagrama de secuencia del proceso de unirse a una partida y el comienzo de la misma.

### 4.2.1. Input workers

Para evitar bloquear el gameloop, toda comunicación con los clientes se realiza mediante threads independientes. La acciones de los jugadores son insertadas por los threads de entrada (`inputWorkers`) en una cola. Las colas utilizadas son una objetos de tipo `Stream`, el cuál funciona como una cola protegida por un mutex que permite hacer `pop` tanto en forma bloqueante o no bloqueante. El `Game` luego obtiene en forma no bloqueante una acción del jugador (cada uno tiene una cola asignada) y la procesa. Leer de forma no bloqueante significa que, de no haber ninguna acción del cliente encolada, se retorna `false` y el gameloop continua con la ejecución. En caso que un cliente se desconecte, este thread realiza un `push` de un evento `disconnect` en la cola, el cuál es leido por el gameloop cuando llega su turno, y maneja el evento de la forma que corresponda.

### 4.2.2. Output workers

Al terminar una iteración, el estado del juego se almacena en un snapshot el cual es leido por threads asignados a cada cliente (los `outputWorkers`), los cuales lo envían serializado por el socket correspondiente. Como cada snapshot tiene toda la información sobre el estado juego en un determinado momento, los `outputWorkers` solo necesitan enviar el mas reciente, por lo que no es necesario utilizar una cola, sino que se usa un `DoubleBuffer`. Este objeto reserva memoria para 2 copias de un objeto de un tipo determinado. En una de ellas (background copy) es en la que se escribe, y la otra (current copy) puede ser leida al mismo tiempo. El escritor puede realizar un `swap` de las copias, con lo que background se convierte en current. Para evitar problemas de concurrencia el `DoubleBuffer` contiene un mutex que bloquea las operaciones `swap` mientras otro thread esté copiando al current. Un thread puede bloquearse esperando un `swap` en un `DoubleBuffer`, lo cual permite que los `outputWorkers` sólo envien datos cuando hay un nuevo snapshot disponible.

## 4.3. Clases

Se describen ahora las clases utilizadas en el servidor.

- ***CommunicationSocket***: clase que se usa para comunicarse con el cliente. Tiene la posibilidad de enviar y recibir mensajes, y es devuelta por movimiento cuando se acepta una conexión. En el cliente se usa indirectamente, ya que es padre de la clase *ClientSocket*, la cual tiene la capacidad de realizar un *connect* al servidor. Sus métodos son *send* y *receive*, utilizados para enviar y recibir información por el *socket* respectivamente.
- ***ServerSocket***: acepta una conexión y devuelve un *CommunicationSocket* por movimiento. Sus métodos son *bindAndListen*, donde se *bindea* a un puerto y escucha conexiones, y *accept*, que hace lo explicado anteriormente.
- ***ServerInternalAction***: clase *enum* que posee las acciones que internamente el servidor envía en *GameLobby* a *LobbyJoiner*.
- ***ServerInternalMsg***: es una estructura que posee un *ServerInternalAction*. Es un mensaje que envía *GameLobby* a *LobbyJoiner*.
- ***GameLobby***: hereda de *Thread* y *Observer*. La explicación de sus tareas se realizó en la descripción del módulo. Se puede agregar que posee los métodos *loadLevels*, que carga recorre el directorio de los niveles cargando su información, y *loadLevel*, que carga la información de un nivel, es decir su archivo de configuración y los fondos ubicados en las direcciones indicadas en el mismo.
- ***GameLobbyAssistant***: hereda de *Thread* y *Observer*. La explicación de sus tareas se realizó en la descripción del módulo. Se pueden destacar los métodos que realizan las acciones indicadas por el cliente:
  - *getLevels*.
  - *createGame*.
  - *getGames*.
  - *joinGame*.

  El método *sendLevelFiles* envía al cliente el archivo de configuración y los fondos del nivel correspondiente.
- ***GamesGetter***: es un *functor* que mediante el operador *()*, el cual recibe una referencia a *Lobbies*, extrae de este la información que debe enviarse en una estructura *GameInfo* al usuario que quiere unirse a una partida.

- ***Lobbies***: posee todos los archivos de configuración y fondos de los niveles. La explicación de sus tareas se realizó en la descripción del módulo. Los métodos que se destacan son *createGame*, que crea una partida, *joinGame*, que se une a una partida, *getGames*, que mediante el *functor GamesGetter* obtiene la información de las partidas, *getLevelData*, que devuelve la información de cada nivel en una estructura *LevelData*, y *configure*, que extrae de los archivos de configuración de cada nivel la información para enviar a los clientes cuando estos quieren crear una partida.

- ***Lobby***: hereda de *Thread* y *Subject*. La explicación de sus tareas se realizó en la descripción del módulo. se destacan los métodos *joinGame*, mediante el cual un jugador se une a una partida, y *addPlayerSocket*, que recibe el *socket* de un jugador y lo almacena para luego dárselo al juego. Antes de dar comienzo a este último envía a cada jugador su número de equipo.

- ***LobbyJoiner***: hereda de *Thread*. La explicación de sus tareas se realizó en la descripción del módulo. Se destacan los métodos *handleServerInput*, en el cual se procesa el mensaje recibido en la cola bloqueante por parte del servidor, y *killLobbies*, que frena la ejecución de todos los *lobbies* y los *joinea*.

- ***Physics***: maneja la lógica correspondiente a la física del juego. Internamente crea un mundo *b2World* de la biblioteca *Box2D* que es donde estará contenido todo lo que suceda en el juego. Sus métodos son *update*, que actualiza el estado del mundo con el tiempo transcurrido desde la última actualización, y *createBody*, que crea un cuerpo *b2Body* a partir de una definición para el mismo de tipo *b2BodyDef*.

- ***PhysicsEntity***: hereda de *Subject*, se construye a partir de un *id* de tipo enumerativo que indica el tipo de entidad que es y sus métodos son *startContact*, *endContact* y *contactWith*, que serán redefinidos por las clases que hereden de esta.

- ***ContactEventListener***: hereda de la clase *b2ContactListener* de la biblioteca *Box2D* y redefine los métodos *PreSolve*, *BeginContact* y *EndContact*, los cuales delegan la acción en las entidades físicas.

- ***TouchSensor***: hereda de *PhysicsEntity*, y se construye a partir de un cuerpo y una forma que asociará al cuerpo como sensor. Se destacan los métodos *startContact*, que se llama cuando el sensor entra en contacto con otra entidad física, *endContact*, que se llama cuando el sensor deja de hacer contacto con otra entidad física, *isActive*, que indica si el sensor está en contacto con otro cuerpo, e *ignore*, que agrega una entidad que debe ser ignorada por el sensor.

- ***Chronometer***: es una clase que encapsula el cálculo del tiempo transcurrido entre distintas llamadas a su método *elapsed*, que devuelve el valor de dicho tiempo.

- ***GameClock***: cuenta el tiempo de los turnos del juego y responde a distintos eventos del mismo, que modifican su valor. Hereda de *Subject*, y notificará eventos al *Game*. En su construcción toma de la configuración los valores correspondientes a la duración de un turno, el tiempo extra que recibe un jugador al disparar, y el tiempo que se deja pasar entre turno y turno para mejorar la dinámica del juego. El método *playerShot* pone el tiempo transcurrido en cero y el tiempo actual del turno en el tiempo extra que corresponde al jugador luego de disparar. Cuando el juego establece que el turno ha terminado (ya sea por el final del tiempo u otro evento como que el gusano activo sufre daño), llama al método *waitForNextTurn*, que pone el tiempo transcurrido en cero, el tiempo que debe transcurrir igual al correspondiente a la espera entre turno y turno, y setea un *flag*. Hace uso de este último en su método *update*, donde recibe el tiempo transcurrido desde su última llamada y lo acumula en un atributo, y si su valor supera al tiempo que posee el jugador actualmente, se notifica o bien que el turno terminó en lo que respecta al tiempo, o bien que el turno siguiente debe comenzar si el *flag* está seteado. El método *endTurn* fuerza el reloj a terminar y notificar un evento de fin de turno, por ejemplo cuando el gusano activo sufre daño al caerse de una altura determinada. El método *restart* vuelve el tiempo acumulado a cero, quita el *flag* de espera del siguiente turno y establece el tiempo de turno en su valor original.

- ***Team***: se construye con los *ids* de los gusanos que son parte del equipo. Los métodos que se destacan son *endTurn*, que define qué gusano utilizará el equipo en un nuevo turno, *weaponUsed*, que decrementa en uno la cantidad de municiones disponible de un arma, *serialize*, que incluye en el mensaje destinado al jugador la cantidad de municiones de que dispone, *checkAlive*, que determina si hay algún gusano con vida en el equipo y de lo contrario setea el *flag alive* en falso, y *kill*, que mata a todos los gusanos del equipo y setea el *flag alive* en falso.

- ***GameTeams***: esta clase posee toda la información relacionada a los equipos del juego. El método *makeTeams* crea los equipos (*Team*) correspondientes asignando de manera aleatoria los gusanos del nivel y definiendo su vida de acuerdo a la cantidad de gusanos que haya. Se destacan también los métodos *checkAlive*, que realiza para todos los equipos el chequeo de si tienen algún gusano vivo, *endTurn*, que define cuál es el siguiente equipo a jugar, *weaponUsed*, que le indica al equipo actual qué arma se utilizó y de la cual dispone de una munición menos, *serialize*, que hace que todos los equipos serialicen su estado, y *kill*, que elimina un equipo del juego cuando un jugador se desconecta. Finalmente, el método *getWinner* devuelve el *id* del equipo ganador si es que existe, o un *id* que no corresponde a ningún equipo si no hay un ganador.
- ***GameTurnState***: es una clase abstracta que hereda de *Subject*, ya que notificará al *Game* de los eventos que sucedan durante el turno. Los métodos a destacar son *endTurn*, que determina si el turno terminó y lo notifica, *explosion*, que indica al estado que una explosión ha ocurrido, *update*, que actualiza la información del turno utilizada para determinar su fin, y *getWormToFollow*, que devuelve el *id* del gusano que debe ser seguido por la cámara de acuerdo a los eventos acontecidos durante el turno. El resto de los métodos indican todos el comienzo y el fin de un estado del gusano, del estilo *wormHit*, *wormEndHit*, *wormDrowning*, *wormDrowned*, etc. Las clases que heredan de esta son:
  - ***StartTurn***: cuando el tiempo termina el turno termina.
  - ***PlayerShot***: es el estado cunado un jugador dispara.
  - ***ImpactOnCourse***: es el estado cuando una explosión sucede. Una vez que el tiempo terminó, todas las balas impactaron y los gusanos están quietos el turno puede terminar.
- ***GameTurn***: hereda de *Subject*. Internamente tiene un *GameTurnState*, hace de interfaz entre este y el *Game*. Se construye con el estado inicial *StartTurn* y en el método *restart*, que se llama cuando el turno ha terminado, vuelve a setearse en ese estado. En el método *update* actualiza el estado de ser necesario, lo cual se indica con un *flag*.
- ***Girder***: hereda de *PhysicsEntity*. Se construye creando un cuerpo con las dimensiones provistas en la información de la viga.
- ***State***: representa el estado del gusano. Todos las acciones provenientes del cliente están representadas en métodos (*moveLeft*, *moveRight*, *jump*, *bazooka*, *startShot*, etc.), y cada estado sabrá responder en consecuencia. Se destaca el método que devuelve el *id* del estado. Las clases que implementan esta interfaz son:
  - ***Walk***.
  - ***Still***.
  - ***StartJump***.
  - ***Jump***.
  - ~~***EndJump***~~.
  - ***BackFlip***.
  - ***BackFlipping***.
  - ~~***EndBackFlip***~~.
  - ***Hit***.
  - ***Die***.
  - ***Dead***.
  - ***Drowning***.
  - ***Falling***.
  - ***Land***.
  - ***Sliding***.
  - ***Teleporting***.
  - ***Teleported***.
  - ***Batting***.

En la figura 5 puede verse un diagrama de estado que muestra como pueden ir cambiando estos. Las dos clases tachadas se hicieron previamente pero no cuentan con uso en el programa final.

- ***Bullet***: hereda de *PhysicsEntity* y cuando se construye crea el cuerpo de la bala, que es un círculo de un radio determinado. En el método *startContact* se setea un *flag* que se utiliza para determinar si explotó en el método *hasExploded* (que también chequea si impactó en el agua o si se terminó la cuenta atrás en caso de que la bala posea la propiedad de *timeout*). En el método *update*, si se ejecuta por primera vez se aplica el impulso a la bala proporcionalmente a la potencia de disparo y en la dirección en la que apuntaba el gusano. Luego se aplica la fuerza del viento hasta que explota, que es cuando se notifica al *Game* que explotó, y se destruye el cuerpo.

- ***Weapon***: es una clase abstracta que encapsula el comportamiento de las armas. Las clases que heredan de esta se construyen en base a una estructura de configuración, el *id* del arma correspondiente y el ángulo en el que está apuntando el gusano (que luego será alterado por los métodos *increaseAngle* y *decreaseAngle*). El método *onExplode* hace lo que necesario cuando la bala disparada hizo impacto, es decir genera fragmentos a partir de esta si el arma así lo requiere y el método *checkBoundaryAngles* evita que los ángulos en que el jugador apunta superen los máximos y mínimos establecidos. El método *positionSelected* se utiliza para lanzar el ataque aéreo, y el método *startShot* setea un *flag* para indicar que debe acumularse potencia de tiro si el arma posee esta habilidad. En el método *update* se acumula dicha potencia en función del tiempo transcurrido, la máxima potencia establecida y el tiempo estipulado para alcanzarla, y cuando es alcanzada, si el jugador aún no disparó, el disparo se realiza automáticamente.
  Esta clase se encuentra contenida en el *Player* mediante *smart pointers*. El player delega en el estado la respuesta que debe tener ante algún evento de las armas. Solo el estado *Still* aplica las acciones del arma. Esto si bien pareció correcto en un inicio, generó que la clase *State* sea una especie de "bolsa de gatos". Si bien es cierto que las acciones del arma dependen del estado (no puedo disparar cuando estoy saltando, o caminando, o cayendo), no debería haberse realizado de esta manera. En un *refactor* posterior podría ser que el estado tenga una función *handleWeaponAction*, y que la clase *Still* llame a las funciones necesarias en el *Player*. En la figura 6 puede verse un diagrama de clase que detalla el uso de las armas en el servidor. Las armas implementadas son:

  - ***AerialAttack***.
  - ***Banana***.
  - ***BaseballBat***.
  - ***Bazooka***.
  - ***Cluster***.
  - ***Dynamite***.
  - ***Grenade***.
  - ***Holy***.
  - ***Mortar***.
  - ***Teleport***.
  - ***WeaponNone***.

- ***Player***: hereda de *PhysicsEntity*. Se construye creando un cuerpo con las dimensiones de un gusano, añadiéndole un *TouchSensor* (el gusano está compuesto por un rectángulo en su parte superior y un círculo en su parte inferior). Los métodos *contactWith*, *isOnGround* y *getGroundNormal* manejan la lógica necesaria para que los gusanos no se empujen entre sí, no puedan desplazarse por vigas con inclinación superior a 45° (se deslicen por ellas si están cayendo), y puedan hacerlo por aquellas con pendiente menor o igual a 45°. Tiene sobrecargados los operadores == y ! = para compararlo como entidad física. Posee un *PlayerState*, que maneja la lógica de cómo responder ante los *inputs* que llegan del cliente, los cuales se procesan mediante *handleState*, y posee un *Weapon*, el cual se utiliza para delegar los métodos *increaseAngle*, *decreaseAngle* y *startShot*. Al momento de realizar el disparo, se ejecuta el método *endShot*, el cual crea la bala y la dispara, avisándole al equipo del gusano que dispone de un proyectil menos del arma usada. El método *acknoledgeDamage* calcula el daño sufrido por el gusano y el impulso que debe aplicársele de acuerdo a su posición y la de la explosión, y *landDamage*, determina si el gusano sufrió algún daño al caer, en base a la altura de la caída. El método *die* mata al gusano (se da cunado un jugador se desconecta), *reset* elimina las balas que haya y reincia el arma. Finalmente, *onExplode* devuelve los fragmentos de bala si corresponde, y *update* actualiza el estado, el arma, y realiza chequeos sobre la pendiente sobre la cual está el gusano y si está por debajo del nivel de agua (se está ahogando).

- ***Wind***: es una estructura que posee la información del viento (intensidad, mínima y máxima intensidad, y dirección en el eje x).
- ***Config::Weapon***: es una estructura en el *namespace Config* (aclarado ya que en otro *namespace* hay otra clase de igual nombre) que posee toda la información de configuración que necesitan las armas.
- ***P2PWeapon***: es una estructura que posee la información de las armas cuerpo a cuerpo (daño que ocasiona, dirección y ángulo en que se la utilizó, y posición al momento de utilizarla).
- ***Config***: clase que carga todos los parámetros configurables del juego de su archivo de configuración.
- ***Game***: es la clase donde se desarrolla el juego. Maneja la comunicación con todos los clientes en el transcurso del mismo. En su método *start* se reciben *inputs* de los usuarios, y se realiza una actualización de los gusanos, de la/s bala/s si existen, y de la física del juego. También actualiza el estado del juego, lo serializa y lo envía a los clientes. Se destacan los métodos *onNotify*, donde se controla toda la lógica del juego (qué sucede cuando hay un disparo, una explosión, un gusano glpeado, ahogándose, muriendo, si el tiempo se terminó, etc.), *playerDisconnected*, que mata a los gusanos del equipo desconectado y termina el juego si la cantidad de equipos restante es igual a uno, y *endTurn*, donde se setean los parámetros necesarios para comenzar un nuevo turno.

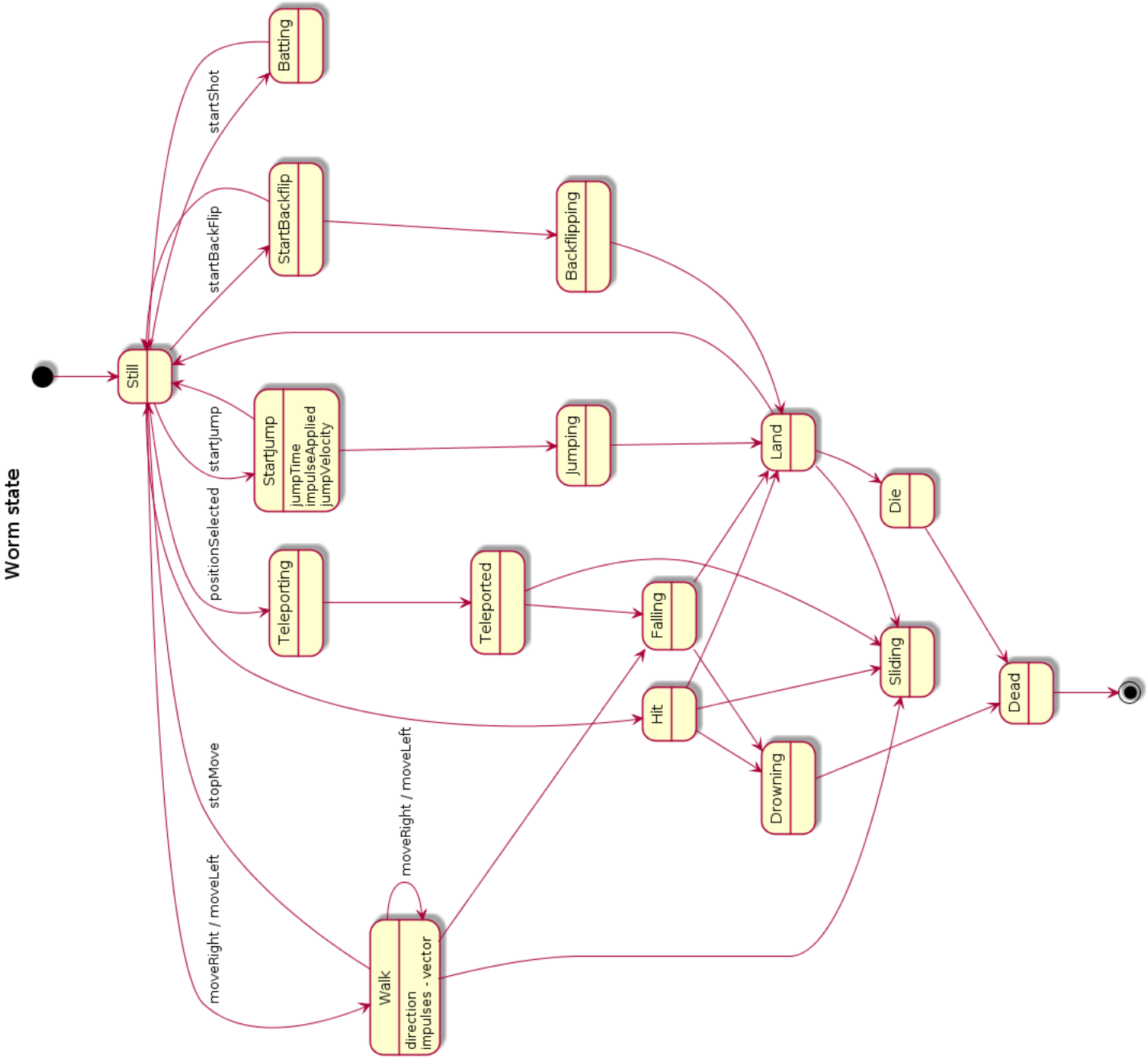


**Figura 5:** Diagrama de estados de los estados posibles del jugador.

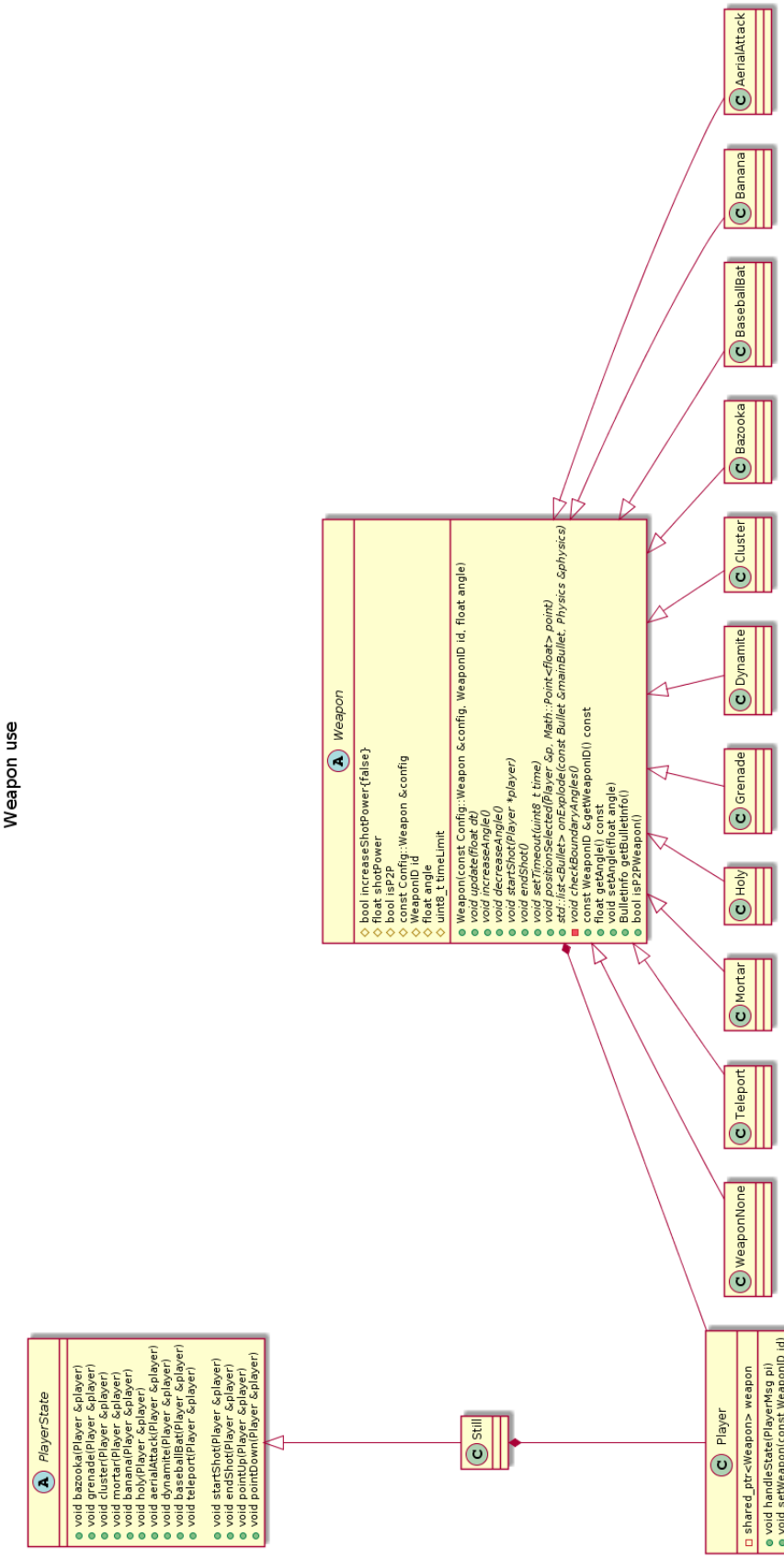


**Figura 6:** Diagrama de clase que muestra el uso de las armas.

# 5. Editor

El editor fue desarrollado en Qt5 debido a que, al contrario de SDL, posee ciertas facilidades como entradas de texto, scroll bars, etc.

Utilizando QtCreator se creó una ventana con los controles básicos y una vista dónde se dibuja el nivel. La vista es un objeto `EditorView` con su `EditorScene`, que heredan de `QGraphicsView` y `QGraphicsScene` respectivamente. Estas clases se encargan de manejar la lógica del editor, como dibujar el "ghost.ª modo de cursor, llevar la cuenta de los objectos creados, etc.

`EditorView` se encarga además de manejar los eventos del usuario (teclado, mouse) de forma tal que permita el uso adecuado del `EditorScene`. Este, por su parte, almacena internamente en un `map` las vigas y worms para que puedan ser obtenidas mas tarde. También dibuja los fondos para reflejar las elecciones del usuario, y evita que se inserten objetos que no corresponden, como ser una viga superpuesta a un worm o un elemento fuera del área definida para el nivel.

Cada elemento del juego (vigas, o worms) son subclases de `StageElement` (subclase de `QGraphicsItem`). Esta interfaz permite que la vista tenga un puntero a un `StageElement` genérico (el cuál se elige mediante los controles) que puede utilizar para instanciar elementos en el nivel de forma similar a un prototype. Estos objetos además implementan su propia serialización, que luego utiliza un objeto `StageData` como visitor para recolectar los datos finales. De esta forma crear un nuevo elemento no resulta muy complejo ya que sólo de debe implementar una nueva subclase de `StageElement`.

El objeto `StageData` es el encargado de transformar los datos obtenidos en un YAML que luego la ventana principal guarda en un archivo seleccionado por el usuario.

## 5.1. Conclusiones

Si bien Qt ofrece bastante funcionalidad .ºut of the box", no siempre es tan inmediato el uso, ya que la variedad de parámetros a configurar es muy grande y suelen afectarse entre si. Además, al ser un framework tan grande, tiene una curva de aprendizaje mas empinada que la de SDL.