

Universidad de Buenos Aires  
Facultad de Ingeniería



96.08- Taller de programación 1

1<sup>er</sup> Cuatrimestre de 2018

Trabajo Práctico Final

Manual de usuario

Grupo 2

Bourbon Navarro, Rodrigo- #96961

Gómez Peter, Federico- #96091

Lidjens, Axel- #95772

## Índice

<b>1. Requerimientos de software</b>	<b>2</b>
<b>2. Descripción general</b>	<b>2</b>
<b>3. Cliente</b>	<b>2</b>
3.1. Clases . . . . .	3
<b>4. Servidor</b>	<b>7</b>
4.1. Desarrollo del juego . . . . .	8
4.2. Clases . . . . .	8
<b>5. Editor</b>	<b>8</b>

## 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 incluidas.

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 iniciarla. 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 iniciarla), 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. 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**: HACER BIEN. 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**: esta clase está preparada para que las clases hijas *ClientSocket* y *ServerSocket* establezcan la conexión y por lo tanto el *file descriptor*. 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.
- **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.

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* inicial 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 partir de una posición, un ancho y un alto. Puede setearse 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.
- **WormState**: 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:
  - *WormWalk*.
  - *WormStill*.
  - *WormStartJump*.
  - *WormJump*.
  - *WormEndJump*.
  - *BackFlip*.
  - *WormBackFlipping*.
  - *WormEndBackFlip*.
  - *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.

## 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 realice 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.

## 4.1. Desarrollo del juego

Durante el juego, que transcurre en la clase *Game*, los distintos clientes son tratados con dos hilos para cada uno, uno que se encarga de recibir datos del cliente (por medio del método *inputWorker*) y otra que se encarga de enviarle el estado del juego (por medio del método *outputWorker*). Estos métodos de la instancia *Game* se encargan de *serializar* y *deserializar* los mensajes que envía y los que recibe del cliente.

## 4.2. 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*.
- ***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.

## 5. Editor