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:

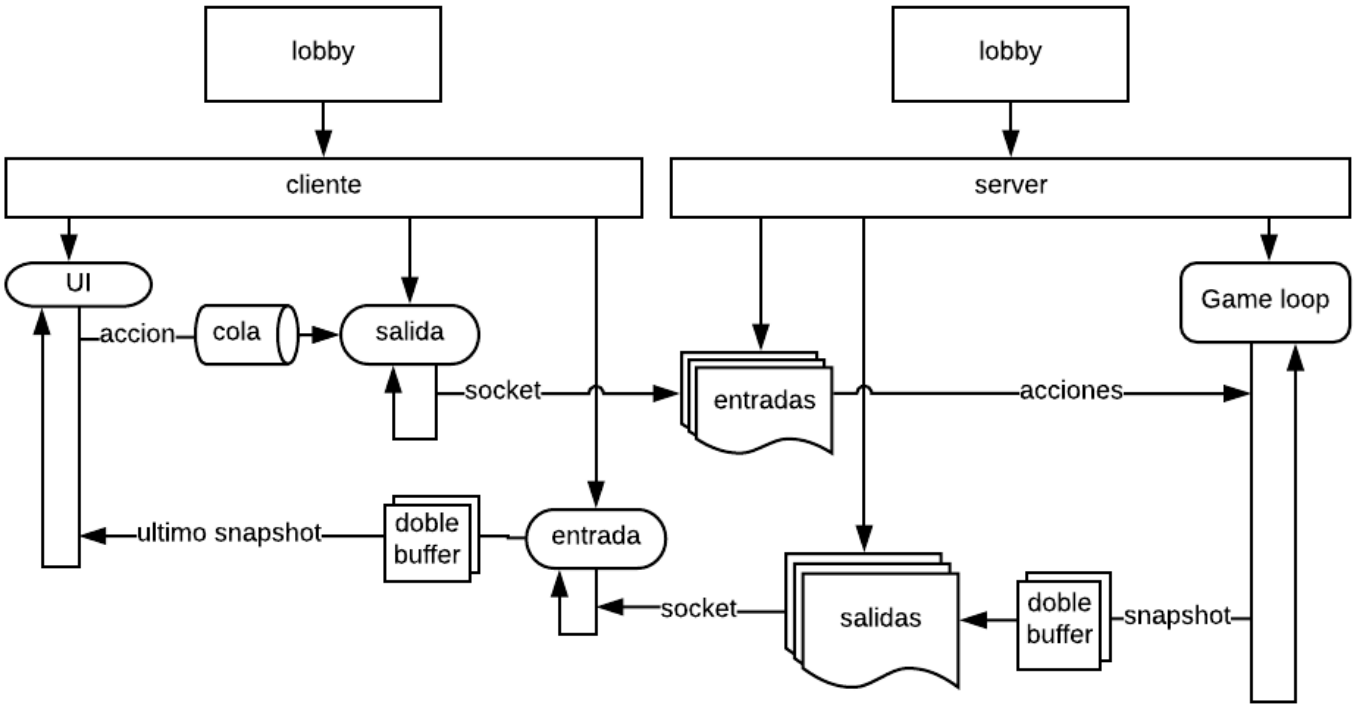

Figura 1: Comunicación entre cliente y servidor

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.

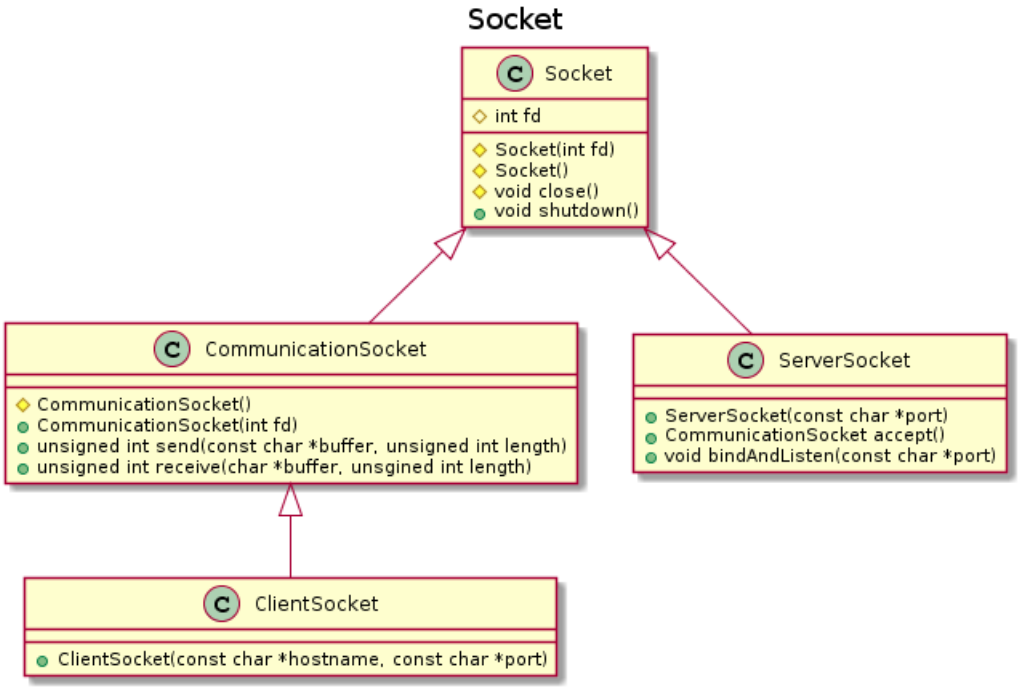

Figura 2: Diagrama de clase de Socket

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.

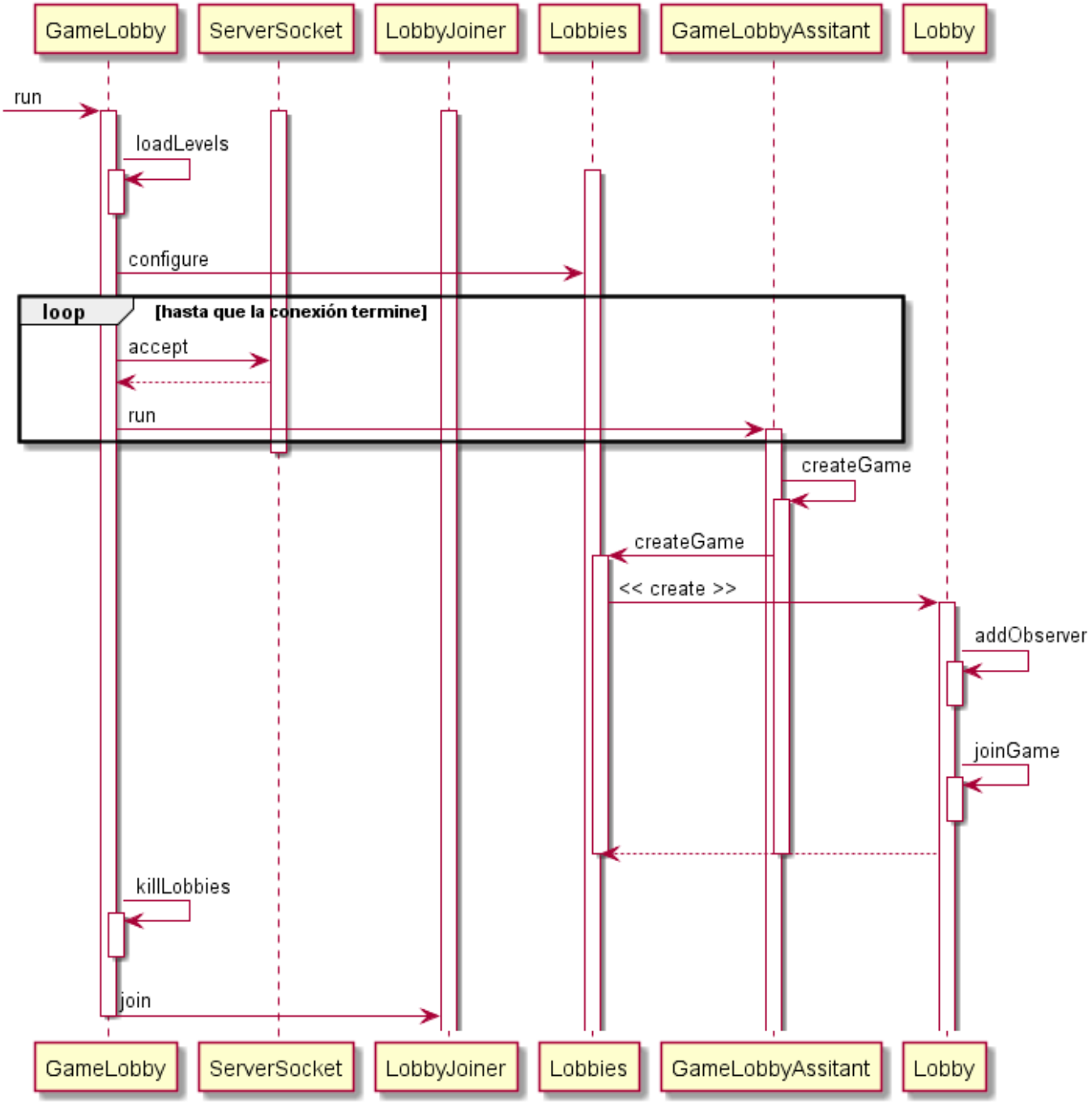

Figura 3: Diagrama de secuencia de la aceptación de una conexión y la creación de una partida.

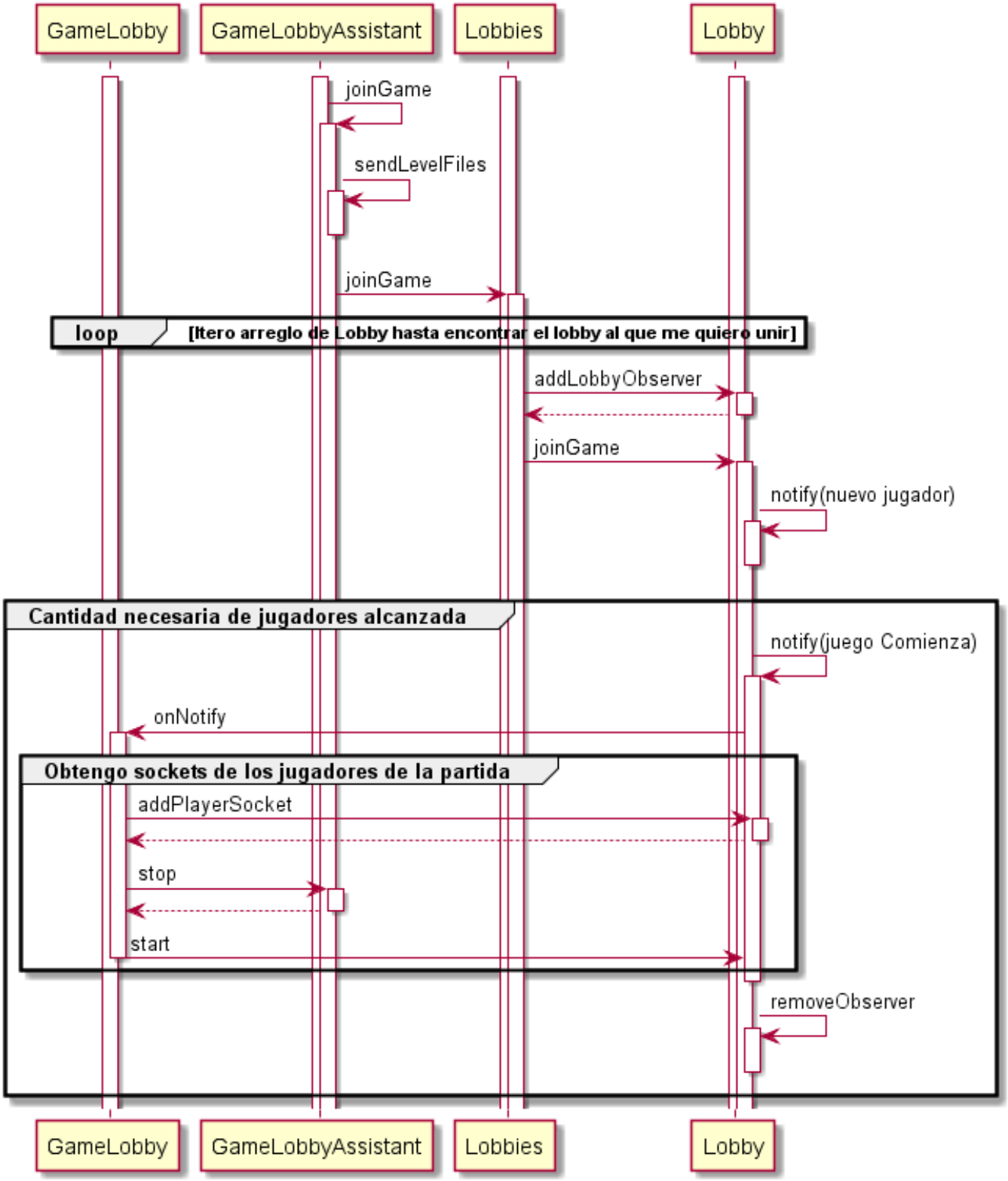

Figura 4: Diagrama de secuencia del proceso de unirse a una partida y el comienzo de la misma.

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.

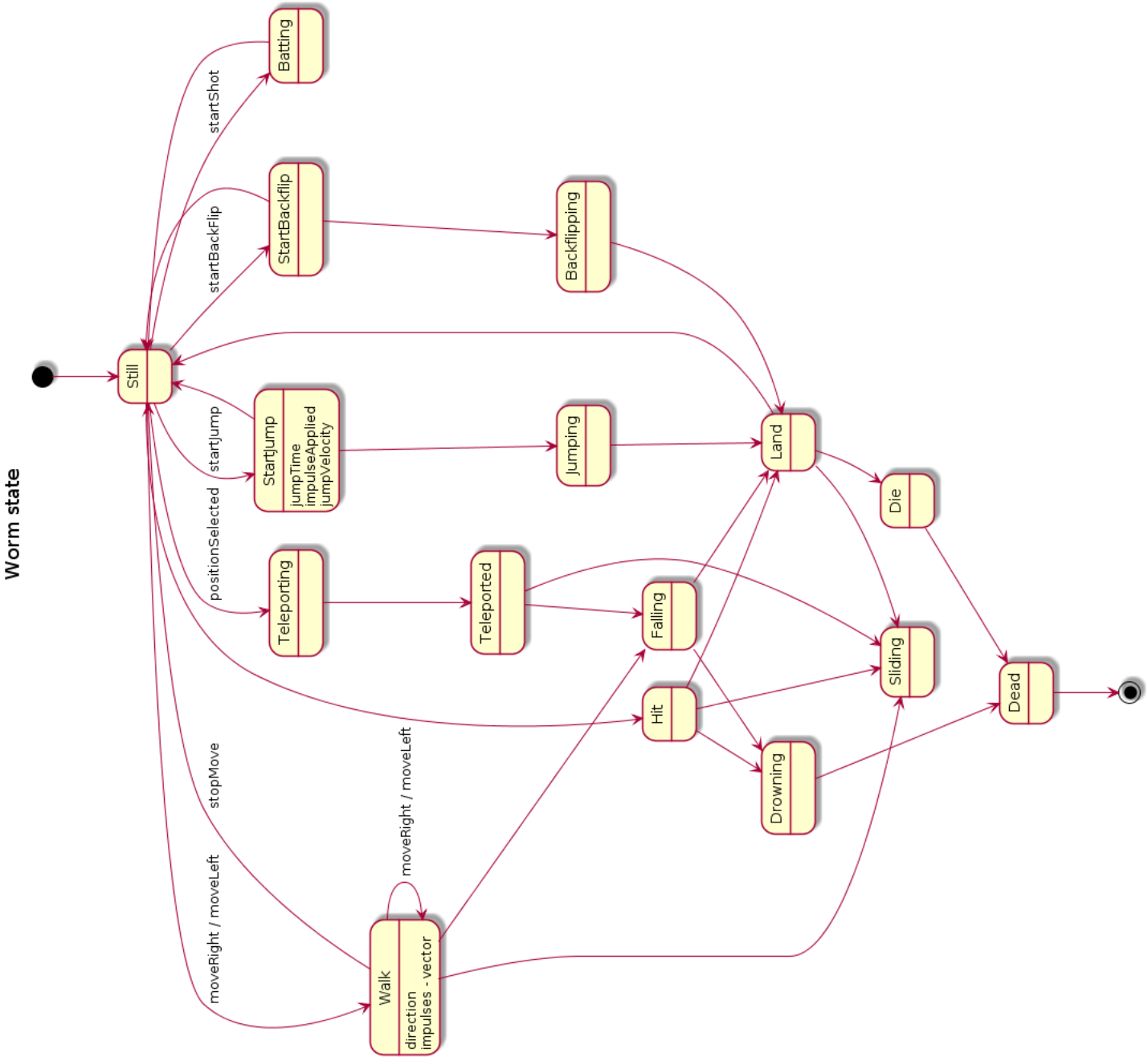

Figura 5: Diagrama de estados de los estados posibles del jugador.

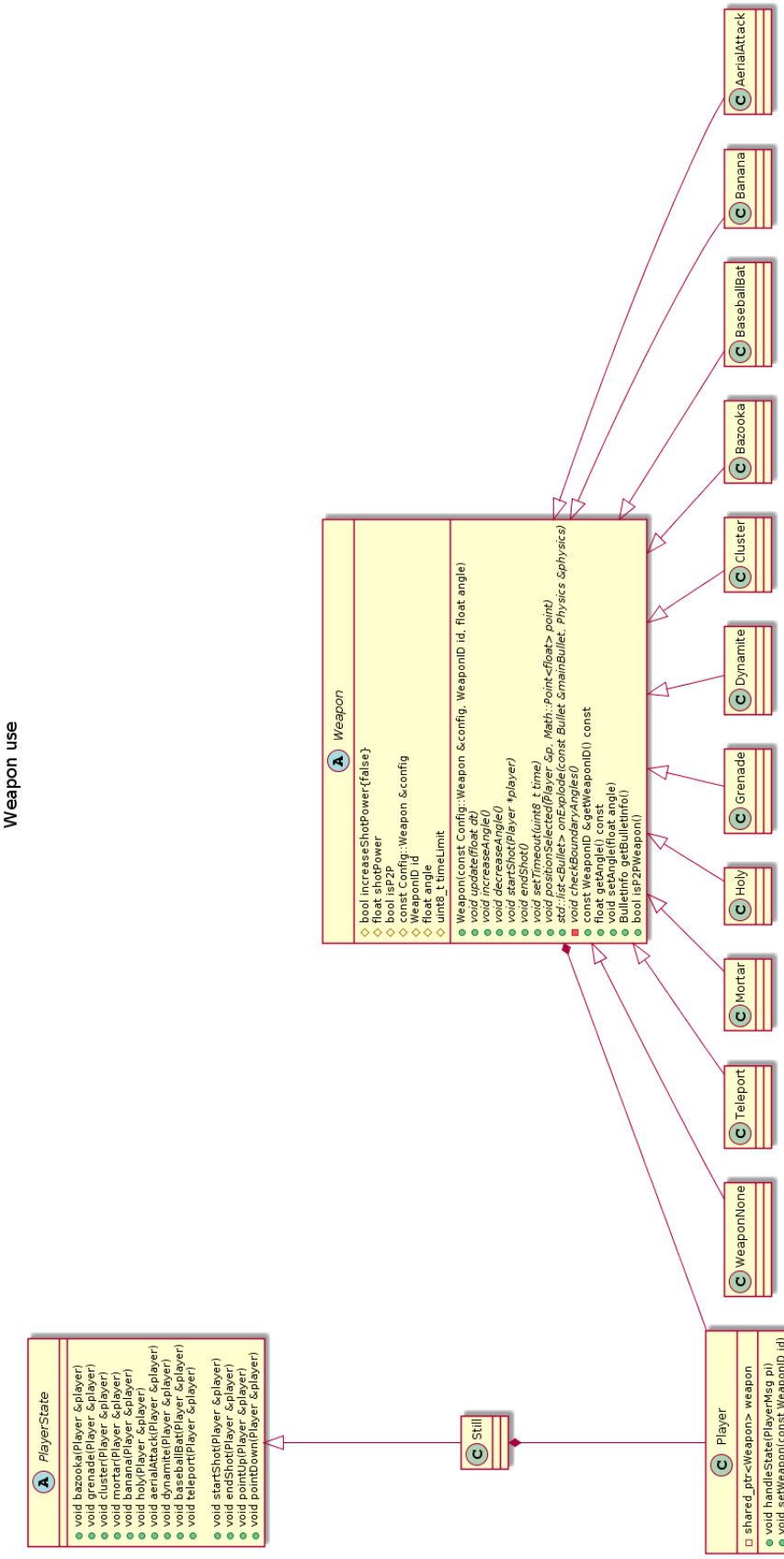

Figura 6: Diagrama de clase que muestra el uso de las armas.

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.

6. Código

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 16/06/18
 */

#include "AerialAttack.h"

Worm::AerialAttack::AerialAttack(const GUI::GameTextureManager &tex)
    : Weapon(tex, GUI::GameTextures::WormAirAttack, AERIAL_ATTACK_CENTER_FRAME,
WeaponID::WAerial) {
    this->weaponAnimation.setAnimateOnce();
}

void Worm::AerialAttack::update(float dt) {
    if (!this->weaponAnimation.finished()) {
        this->weaponAnimation.update(dt);
    } else {
        this->endAnimation();
    }
}

void Worm::AerialAttack::render(GUI::Position &p, GUI::Camera &cam, SDL_Renderer
Flip &flip) {
    this->weaponAnimation.render(p, cam, flip);
}

void Worm::AerialAttack::setAngle(float angle, Worm::Direction d) {}

void Worm::AerialAttack::startShot() {}

void Worm::AerialAttack::endShot() {}

bool Worm::AerialAttack::positionSelected() {
    this->weaponAnimation.setAutoUpdate(true);
    return true;
}

void Worm::AerialAttack::endAnimation() {
    this->weaponAnimation.setFrame(AERIAL_ATTACK_CENTER_FRAME);
    this->weaponAnimation.setAutoUpdate(false);
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 16/06/18
 */

#ifndef __AerialAttack_H__
#define __AerialAttack_H__

#define AERIAL_ATTACK_CENTER_FRAME 0

#include "Weapon.h"

namespace Worm {
class AerialAttack : public Weapon {
   public:
    explicit AerialAttack(const GUI::GameTextureManager &textureManager);
    ~AerialAttack() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;

   private:
    void endAnimation();
};
}  // namespace Worm

#endif  //__AerialAttack_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 10/06/18
 */

#include <sstream>

#include "Armory.h"

GUI::Armory::Armory(const GUI::GameTextureManager &textureManager, GUI::Camera &
cam,
                    GUI::Font &font)
    : manager(textureManager),
      camera(cam),
      font(font),
      weaponButton(font),
      ammunition(WEAPONS_QUANTITY, 0) {}

void GUI::Armory::render() {
    const Texture *temp = this->weaponIcons.back();
    ScreenPosition ammoPos{-temp->getWidth() / 2, 10};
    ScreenPosition iconPos{-temp->getWidth() / 2, 20 + temp->getHeight() / 2};
    ScreenPosition textPos{-temp->getWidth() / 2, 20 + temp->getHeight() * 3 / 2
};
    int i = 1;
    for (auto &weapon : this->weaponIcons) {
        ammoPos.x += weapon->getWidth();
        iconPos.x += weapon->getWidth();
        textPos.x += weapon->getWidth();

        std::int16_t weaponAmmo = this->ammunition[i - 1];
        std::ostringstream button;
        button << BUTTON_ROOT_STR << i++;

        if (weaponAmmo == -1) {
            weaponButton.set(std::string("inf"), SDL_Color{0, 0, 0}, 20);
            weaponButton.renderFixed(ammoPos, this->camera);
        } else {
            weaponButton.set(std::to_string(weaponAmmo), SDL_Color{0, 0, 0}, 20)
;
            weaponButton.renderFixed(ammoPos, this->camera);
        }

        weaponButton.set(button.str(), SDL_Color{0, 0, 0}, 25);
        weaponButton.renderFixed(textPos, this->camera);
        this->camera.drawLocal(*weapon, iconPos);
    }
}

void GUI::Armory::loadWeapons() {
    this->weaponIcons.emplace_back(&this->manager.get(GUI::GameTextures::Bazooka
Icon));
    this->weaponIcons.emplace_back(&this->manager.get(GUI::GameTextures::Grenade
Icon));
    this->weaponIcons.emplace_back(&this->manager.get(GUI::GameTextures::Cluster
Icon));
    this->weaponIcons.emplace_back(&this->manager.get(GUI::GameTextures::MortarI
con));
    this->weaponIcons.emplace_back(&this->manager.get(GUI::GameTextures::BananaI
con));
    this->weaponIcons.emplace_back(&this->manager.get(GUI::GameTextures::HolyIco
n));
```



```
    this->weaponIcons.emplace_back(&this->manager.get(GUI::GameTextures::AirIcon
));
    this->weaponIcons.emplace_back(&this->manager.get(GUI::GameTextures::Dynamit
eIcon));
    this->weaponIcons.emplace_back(&this->manager.get(GUI::GameTextures::Basebal
lBatIcon));
    this->weaponIcons.emplace_back(&this->manager.get(GUI::GameTextures::Telepor
tIcon));
}

void GUI::Armory::update(IO::GameStateMsg &msg) {
    for (int i = 0; i < WEAPONS_QUANTITY; i++) {
        this->ammunition[i] = msg.weaponAmmunition[i];
    }
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 10/06/18
 */

#ifndef __ARMORY_H__
#define __ARMORY_H__

#define BUTTON_ROOT_STR "F"

#include <vector>

#include <Animation.h>
#include <Font.h>
#include <GameStateMsg.h>
#include <Text.h>
#include "GameTextures.h"

namespace GUI {
class Armory {
   public:
    Armory(const GameTextureManager &textureManager, Camera &cam, Font &font);
    ~Armory() = default;
    void loadWeapons();
    void render();
    void update(IO::GameStateMsg &msg);

   private:
    const GameTextureManager &manager;
    Camera &camera;
    std::vector<const Texture *> weaponIcons;
    const Font &font;
    Text weaponButton;
    std::vector<std::int16_t> ammunition;
};
}  // namespace GUI

#endif  //__ARMORY_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 20/05/18
 */

#include "BackFlip.h"

Worm::BackFlip::BackFlip() : State(StateID::StartBackFlip) {}

Worm::BackFlip::~BackFlip() {}

void Worm::BackFlip::update(float dt) {}

IO::PlayerInput Worm::BackFlip::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::mortar(Worm &w) {
```



```
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlip::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 20/05/18
 */

#ifndef __WORM_BACK_FLIP_H__
#define __WORM_BACK_FLIP_H__

#include "GameStateMsg.h"
#include "WormState.h"

namespace Worm {
class BackFlip : public State {
   public:
    explicit BackFlip();
    virtual ~BackFlip();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;

    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;
    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__WORM_BACK_FLIP_H__
```

```cpp
//
// Created by rodrigo on 25/06/18.
//

#include "BackgroundMusic.h"
#include "Exception.h"

GUI::BackgroundMusic::BackgroundMusic(const std::string &filename) {
    this->backgroundMusic = Mix_LoadMUS(filename.c_str());
    if (!this->backgroundMusic) {
        throw Exception{"Error loading %s: %s", filename.c_str(), Mix_GetError()};
    }
}

GUI::BackgroundMusic::~BackgroundMusic() {
    if (this->backgroundMusic != nullptr) {
        Mix_FreeMusic(this->backgroundMusic);
    }
}

Mix_Music *GUI::BackgroundMusic::getMusic() const {
    return this->backgroundMusic;
}

GUI::BackgroundMusic::BackgroundMusic(GUI::BackgroundMusic &&other) {
    std::swap(this->backgroundMusic, other.backgroundMusic);
}

void GUI::BackgroundMusic::play() const {
    Mix_PlayMusic(this->backgroundMusic, -1);
}
```

```
//
// Created by rodrigo on 25/06/18.
//

#ifndef INC_4_WORMS_BACKGROUNDMUSIC_H
#define INC_4_WORMS_BACKGROUNDMUSIC_H


#include <SDL2/SDL.h>
#include <SDL2/SDL_mixer.h>
#include <string>

namespace GUI {
    class BackgroundMusic {
    public:
        BackgroundMusic(const std::string &filename);
        BackgroundMusic(BackgroundMusic &&other);
        ~BackgroundMusic();
        Mix_Music *getMusic() const;
        void play() const;

    private:
        Mix_Music *backgroundMusic{nullptr};
    };
}


#endif //INC_4_WORMS_BACKGROUNDMUSIC_H
```

```
//
// Created by rodrigo on 25/06/18.
//

#ifndef INC_4_WORMS_BACKGROUNDMUSICMANAGER_H
#define INC_4_WORMS_BACKGROUNDMUSICMANAGER_H

#include <SDL2/SDL.h>
#include <functional>
#include <string>
#include <unordered_map>
#include "BackgroundMusic.h"

namespace GUI {
    template <typename ID, typename HASH = std::hash<ID>>
    class BackgroundMusicManager {
    public:
        BackgroundMusicManager();
        ~BackgroundMusicManager();
        BackgroundMusicManager& operator=(BackgroundMusicManager& other) = delet
e;

        void load(ID id, const std::string& file_name);
        const BackgroundMusic& get(ID id) const;

    private:
        std::unordered_map<ID, BackgroundMusic, HASH> cache;
    };
}  // namespace GUI

template <typename ID, typename HASH>
GUI::BackgroundMusicManager<ID, HASH>::BackgroundMusicManager() {}

template <typename ID, typename HASH>
GUI::BackgroundMusicManager<ID, HASH>::~BackgroundMusicManager() {}

/**

 * @brief Loads a background music file.
 *
 * @param file_name The image file name.
 */
template <typename ID, typename HASH>
void GUI::BackgroundMusicManager<ID, HASH>::load(ID id, const std::string& file_
name) {
    GUI::BackgroundMusic backgroundMusic{file_name};
    this->cache.insert(std::make_pair(id, std::move(backgroundMusic)));
}

/**
 * @brief Gets a background music.
 *
 * @param file_name Name of the background music.
 */
template <typename ID, typename HASH>
const GUI::BackgroundMusic& GUI::BackgroundMusicManager<ID, HASH>::get(ID id) co
nst {
    return this->cache.at(id);
}

#endif //INC_4_WORMS_BACKGROUNDMUSICMANAGER_H
```

```cpp
//
// Created by rodrigo on 25/06/18.
//

#include "BackgroundMusicPlayer.h"

GUI::BackgroundMusicPlayer::BackgroundMusicPlayer(const GUI::BackgroundMusic &ba
ckgroundMusic)
        : backgroundMusic(&backgroundMusic) {}

GUI::BackgroundMusicPlayer::~BackgroundMusicPlayer() {}

void GUI::BackgroundMusicPlayer::play() {
    this->backgroundMusic->play();
}
```

```
//
// Created by rodrigo on 25/06/18.
//

#ifndef INC_4_WORMS_BACKGROUNDMUSICPLAYER_H
#define INC_4_WORMS_BACKGROUNDMUSICPLAYER_H


#include <SDL2/SDL.h>

#include "BackgroundMusic.h"

namespace GUI {
    class BackgroundMusicPlayer {
    public:
        bool loop{false};

        explicit BackgroundMusicPlayer(const GUI::BackgroundMusic &backgroundMus
ic);
        ~BackgroundMusicPlayer();
        void play();

    private:
        const BackgroundMusic *backgroundMusic;
    };
}


#endif //INC_4_WORMS_BACKGROUNDMUSICPLAYER_H
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#include <cmath>

#include "Banana.h"

Worm::Banana::Banana(const GUI::GameTextureManager &tex)
    : Weapon(tex, GUI::GameTextures::WormBanana, BANANA_CENTER_FRAME, WeaponID::
WBanana),
      scope(this->textureMgr),
      powerBar(this->textureMgr) {}

void Worm::Banana::update(float dt) {
    this->weaponAnimation.update(dt);
    this->scope.update(dt);
    this->powerBar.update(dt);
}

void Worm::Banana::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &
flip) {
    this->weaponAnimation.render(p, cam, flip);
    this->scope.render(p, cam, flip);
    this->powerBar.render(p, cam, flip);
}

void Worm::Banana::setAngle(float angle, Worm::Direction d) {
    this->weaponAnimation.setFrame((int)std::ceil(angle / ANGLE_STEP) + this->ce
nterFrame);
    this->scope.setAngle(angle, d);
    this->powerBar.setAngle(angle, d);
}

void Worm::Banana::startShot() {
    this->powerBar.startShot();
}

void Worm::Banana::endShot() {
    this->powerBar.endShot();
}

bool Worm::Banana::positionSelected() {
    return false;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#ifndef __BANANA_H__
#define __BANANA_H__

#include <vector>

#include "PowerBar.h"
#include "Scope.h"
#include "Weapon.h"

#define BANANA_CENTER_FRAME 14

namespace Worm {
class Banana : public Weapon {
   public:
    explicit Banana(const GUI::GameTextureManager &textureManager);
    ~Banana() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;

   private:
    ::Weapon::Scope scope;
    ::Weapon::PowerBar powerBar;
};
}  // namespace Worm

#endif  //__BANANA_H__
```

```
//
// Created by rodrigo on 16/06/18.
//

#include "BaseballBat.h"
#include <cmath>
#include <iostream>

Worm::BaseballBat::BaseballBat(const GUI::GameTextureManager &tex)
    : Weapon(tex, GUI::GameTextures::WormBaseballBat, BASEBALL_BAT_CENTER_FRAME,
             WeaponID::WBaseballBat),
      scope(this->textureMgr) {}

void Worm::BaseballBat::update(float dt) {
    this->weaponAnimation.update(dt);
    this->scope.update(dt);
}

void Worm::BaseballBat::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererF
lip &flip) {
    this->weaponAnimation.render(p, cam, flip);
    this->scope.render(p, cam, flip);
}

void Worm::BaseballBat::setAngle(float angle, Direction d) {
    this->weaponAnimation.setFrame((int)std::ceil(angle / ANGLE_STEP) + this->ce
nterFrame);
    this->scope.setAngle(angle, d);
}

void Worm::BaseballBat::startShot() {}

void Worm::BaseballBat::endShot() {}

bool Worm::BaseballBat::positionSelected() {
    return false;
}
```

```
//
// Created by rodrigo on 16/06/18.
//

#ifndef INC_4_WORMS_BASEBALLBAT_H
#define INC_4_WORMS_BASEBALLBAT_H

#include "Scope.h"
#include "Weapon.h"

#define BASEBALL_BAT_CENTER_FRAME 16

namespace Worm {
class BaseballBat : public Weapon {
   public:
    explicit BaseballBat(const GUI::GameTextureManager &textureManager);
    ~BaseballBat() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;

   private:
    ::Weapon::Scope scope;
};
}  // namespace Worm

#endif  // INC_4_WORMS_BASEBALLBAT_H
```

```
//
// Created by rodrigo on 23/06/18.
//

#include "Batting.h"

Worm::Batting::Batting() : State(StateID::Batting) {}

Worm::Batting::~Batting() {}

void Worm::Batting::update(float dt) {}

IO::PlayerInput Worm::Batting::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::mortar(Worm &w) {
    return IO::PlayerInput::moveNone;
```



```
}

IO::PlayerInput Worm::Batting::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Batting::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
//
// Created by rodrigo on 23/06/18.
//

#ifndef INC_4_WORMS_BATTING_H
#define INC_4_WORMS_BATTING_H

#include "../Worm.h"
#include "GameStateMsg.h"
#include "WormState.h"

namespace Worm {
class Batting : public State {
   public:
    Batting();

    ~Batting();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;

    virtual IO::PlayerInput moveLeft(Worm &w) override;

    virtual IO::PlayerInput stopMove(Worm &w) override;

    virtual IO::PlayerInput jump(Worm &w) override;

    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;

    virtual IO::PlayerInput grenade(Worm &w) override;

    virtual IO::PlayerInput cluster(Worm &w) override;

    virtual IO::PlayerInput mortar(Worm &w) override;

    virtual IO::PlayerInput banana(Worm &w) override;

    virtual IO::PlayerInput holy(Worm &w) override;

    virtual IO::PlayerInput aerialAttack(Worm &w) override;

    virtual IO::PlayerInput dynamite(Worm &w) override;

    virtual IO::PlayerInput baseballBat(Worm &w) override;

    virtual IO::PlayerInput teleport(Worm &w) override;

    virtual IO::PlayerInput positionSelected(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;

    virtual IO::PlayerInput endShot(Worm &w) override;

    virtual IO::PlayerInput pointUp(Worm &w) override;

    virtual IO::PlayerInput pointDown(Worm &w) override;

    virtual IO::PlayerInput backFlip(Worm &w) override;
};
```



```
}

#endif  // INC_4_WORMS_BATTING_H
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#include <cmath>

#include "Bazooka.h"

Worm::Bazooka::Bazooka(const GUI::GameTextureManager &tex)
    : Weapon(tex, GUI::GameTextures::Bazooka, BAZOOKA_CENTER_FRAME, WeaponID::WB
azooka),
      scope(this->textureMgr),
      powerBar(this->textureMgr) {}

void Worm::Bazooka::update(float dt) {
    this->weaponAnimation.update(dt);
    this->scope.update(dt);
    this->powerBar.update(dt);
}

void Worm::Bazooka::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip
&flip) {
    this->weaponAnimation.render(p, cam, flip);
    this->scope.render(p, cam, flip);
    this->powerBar.render(p, cam, flip);
}

void Worm::Bazooka::setAngle(float angle, Direction d) {
    this->weaponAnimation.setFrame((int)std::ceil(angle / ANGLE_STEP) + this->ce
nterFrame);
    this->scope.setAngle(angle, d);
    this->powerBar.setAngle(angle, d);
}

void Worm::Bazooka::startShot() {
    this->powerBar.startShot();
}

void Worm::Bazooka::endShot() {
    this->powerBar.endShot();
}

bool Worm::Bazooka::positionSelected() {
    return false;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#ifndef __BAZOOKA_H__
#define __BAZOOKA_H__

#include <vector>

#include "PowerBar.h"
#include "Scope.h"
#include "Weapon.h"

#define BAZOOKA_CENTER_FRAME 16

namespace Worm {
class Bazooka : public Weapon {
   public:
    explicit Bazooka(const GUI::GameTextureManager &textureManager);
    ~Bazooka() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;

   private:
    ::Weapon::Scope scope;
    ::Weapon::PowerBar powerBar;
};
}  // namespace Worm

#endif  //__BAZOOKA_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 26/05/18
 */

#include <cmath>
#include <iostream>

#include "../GameSoundEffects.h"
#include "Bullet.h"

Ammo::Bullet::Bullet(const GUI::GameTextureManager &texture_mgr,
                     const GUI::GameSoundEffectManager &sound_effect_mgr, Worm::
WeaponID id)
    : texture_mgr(texture_mgr),
      sound_effect_mgr(sound_effect_mgr),
      animation(this->texture_mgr.get(GUI::GameTextures::Missile), true, MISSILE
_0_DEG_FRAME,
                false),
      explosion(this->texture_mgr) {
    switch (id) {
        case Worm::WeaponID::WBazooka:
            this->animation = GUI::Animation(this->texture_mgr.get(GUI::GameText
ures::Missile),
                                             true, MISSILE_0_DEG_FRAME, false);
            break;
        case Worm::WeaponID::WGrenade:
            this->animation = GUI::Animation(this->texture_mgr.get(GUI::GameText
ures::Grenade),
                                             false, MISSILE_0_DEG_FRAME, false);
            break;
        case Worm::WeaponID::WCluster:
            this->animation = GUI::Animation(this->texture_mgr.get(GUI::GameText
ures::Cluster),
                                             false, MISSILE_0_DEG_FRAME, false);
            break;
        case Worm::WeaponID::WMortar:
            this->animation = GUI::Animation(this->texture_mgr.get(GUI::GameText
ures::Mortar),
                                             false, MISSILE_0_DEG_FRAME, false);
            break;
        case Worm::WeaponID::WBanana:
            this->animation = GUI::Animation(this->texture_mgr.get(GUI::GameText
ures::Banana),
                                             false, MISSILE_0_DEG_FRAME, false);
            break;
        case Worm::WeaponID::WHoly:
            this->animation = GUI::Animation(this->texture_mgr.get(GUI::GameText
ures::Holy), false,
                                             MISSILE_0_DEG_FRAME, false);
            break;
        case Worm::WeaponID::WAerial:
            this->animation = GUI::Animation(this->texture_mgr.get(GUI::GameText
ures::AirMissile),
                                             false, MISSILE_0_DEG_FRAME, false);
            break;
        case Worm::WeaponID::WBaseballBat:
            break;
        case Worm::WeaponID::WTeleport:
            break;
        case Worm::WeaponID::WExplode:
            break;
```



```
        case Worm::WeaponID::WFragment:
            this->animation =
                GUI::Animation(this->texture_mgr.get(GUI::GameTextures::Fragment
), false, 0, true);
            this->updateManually = false;
            break;
        case Worm::WeaponID::WDynamite:
            this->animation =
                GUI::Animation(this->texture_mgr.get(GUI::GameTextures::Dynamite
), false, 0, true);
            this->updateManually = false;
            break;
        case Worm::WeaponID::WNone:
            break;
    }
    this->wid = id;
}

void Ammo::Bullet::update(float dt) {
    if (!this->explode) {
        if (this->updateManually) {
            float angle = (this->angle - 90);
            if (angle >= 360) {
                angle -= 360;
            }
            float angleStep = MISSILE_ANGLE_STEP;
            this->animation.setFrame((int)std::floor(angle / angleStep));
        } else {
            this->animation.update(dt);
        }
    } else {
        this->explosion.update(dt);
    }
}

void Ammo::Bullet::render(GUI::Position p, GUI::Camera &cam) {
    if (!this->explode) {
        this->animation.render(p, cam, SDL_FLIP_HORIZONTAL);
    } else {
        this->explosion.render(cam);
    }
}

void Ammo::Bullet::setAngle(float angle) {
    this->angle = angle;
}

bool Ammo::Bullet::exploded() {
    return this->explosion.finished();
}

void Ammo::Bullet::madeImpact() {
    this->explode = true;
    this->soundEffectPlayer = std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::S
oundEffectPlayer{
        this->sound_effect_mgr.get(GUI::GameSoundEffects::Explosion), true});
    this->soundEffectPlayer->play();
}

void Ammo::Bullet::setPosition(GUI::Position p) {
    this->position = p;
    this->explosion.position = p;
```

```
}

bool Ammo::Bullet::exploding() {
    return this->explode;
}

GUI::Position Ammo::Bullet::getPosition() {
    return this->position;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 26/05/18
 */

#ifndef __Bullet_h__
#define __Bullet_h__

#include <GameStateMsg.h>
#include <memory>

#include "../GameSoundEffects.h"
#include "../GameTextures.h"
#include "../SoundEffectPlayer.h"
#include "Animation.h"
#include "Explosion.h"

#define MISSILE_0_DEG_FRAME 8
#define MISSILE_ANGLE_STEP 11.25f

namespace Ammo {
class Bullet {
   public:
    explicit Bullet(const GUI::GameTextureManager &texture_mgr,
                    const GUI::GameSoundEffectManager &sound_effect_mgr, Worm::W
eaponID id);
    ~Bullet() = default;
    void update(float dt);
    void render(GUI::Position p, GUI::Camera &cam);
    void setAngle(float angle);
    void setPosition(GUI::Position p);
    GUI::Position getPosition();
    void madeImpact();
    bool exploding();
    bool exploded();

   private:
    float angle{0};
    bool updateManually{true};
    const GUI::GameTextureManager &texture_mgr;
    const GUI::GameSoundEffectManager &sound_effect_mgr;
    GUI::Animation animation;
    GUI::Position position{0, 0};
    Worm::Explosion explosion;
    bool explode{false};
    Worm::WeaponID wid;
    std::shared_ptr<GUI::SoundEffectPlayer> soundEffectPlayer{nullptr};
};
}

#endif  //__Bullet_H__
```

```
//
// Created by rodrigo on 20/06/18.
//

#include <Font.h>
#include "Button.h"

GUI::Button::Button(ScreenPosition sp, int height, int width, const std::string
&msg, Font &font) :
        position(sp),
        height(height),
        width(width),
        msg(msg),
        textColor(SDL_Color{0xFF, 0xFF, 0xFF}),
        textSize(40),
        text(font) {
    this->text.set(this->msg, SDL_Color{0xFF, 0xFF, 0xFF}, 40);
}

GUI::Button::Button(const std::string &msg, GUI::Font &font, SDL_Color textColor
, int textSize) :
        msg(msg),
        textColor(textColor),
        textSize(textSize),
        text(font) {
    this->text.set(this->msg, textColor, textSize);
}

GUI::Button::Button(const std::string &msg, GUI::Font &font) :
        msg(msg),
        text(font) {
}

GUI::Button::Button(GUI::ScreenPosition sp, int height, int width, GUI::Font &fo
nt) :
        position(sp),
        height(height),
        width(width),
        text(font) {

}

void GUI::Button::render(GUI::Camera &cam) {
    SDL_Rect fillRect = {this->position.x - this->width / 2, this->position.y +
this->height / 2,
                         this->width / (int) cam.getScale(), this->height / (int
) cam.getScale()};
    cam.drawLocal(ScreenPosition{this->position.x, this->position.y}, fillRect,
SDL_Color{0, 0, 0});
    this->text.set(this->msg, this->textColor, this->textSize);
    this->text.renderFixed(this->position, cam);
}

bool GUI::Button::inside(GUI::ScreenPosition sp) {
    bool inside = true;

    if(sp.x < this->position.x - this->width / 2) {
        //Mouse is left of the button
        inside = false;
    } else if(sp.x > this->position.x + this->width / 2) {
        //Mouse is right of the button
        inside = false;
```



```
    } else if(sp.y < this->position.y - this->height / 2) {
        //Mouse below the button
        inside = false;
    } else if(sp.y > this->position.y + this->width / 2) {
        //Mouse above the button
        inside = false;
    }
    return inside;
}

void GUI::Button::setBackground(SDL_Color color) {
    this->text.setBackground(color);
}
```

```
//
// Created by rodrigo on 20/06/18.
//

#ifndef INC_4_WORMS_BUTTON_H
#define INC_4_WORMS_BUTTON_H


#include <Camera.h>
#include <Text.h>

namespace GUI {
    class Button {
    public:
        GUI::ScreenPosition position{0, 0};
        int height{0};
        int width{0};
        std::string msg;
        SDL_Color textColor{0, 0, 0};
        int textSize{10};

        Button(ScreenPosition sp, int height, int width, const std::string &msg,
 Font &font);
        Button(const std::string &msg, GUI::Font &font, SDL_Color textColor, int
 textSize);
        Button(const std::string &msg, Font &font);
        Button(ScreenPosition sp, int height, int width, Font &font);

        bool inside(ScreenPosition sp);
        void render(GUI::Camera &cam);

        void setBackground(SDL_Color color);

    private:
        Text text;
    };
}

#endif //INC_4_WORMS_BUTTON_H
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#include <netdb.h>
#include <unistd.h>
#include <cstring>

#include "ClientSocket.h"
#include "ErrorMessages.h"
#include "Exception.h"

ClientSocket::ClientSocket(const char *hostName, const char *port) {
    int status;
    bool is_connected = false;

    struct addrinfo hints = {AI_PASSIVE, AF_INET, SOCK_STREAM, 0, 0, nullptr, nu
llptr, nullptr};
    struct addrinfo *result, *ptr;

    status = getaddrinfo(hostName, port, &hints, &result);
    if (status != 0) {
        throw Exception(ERR_MSG_SOCKET_INVALID_HOST_OR_PORT, hostName, port, str
error(errno));
    }

    for (ptr = result; ptr != nullptr && !is_connected; ptr = ptr->ai_next) {
        this->fd = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
        /*
        *       si la creaciÃ³n del socket falla, no debo hacer nada mas
        *       en el ciclo (ya que no se abrio ningun fd)
        */
        if (this->fd == -1) {
            continue;
        }

        status = ::connect(this->fd, ptr->ai_addr, ptr->ai_addrlen);
        if (status == -1) {
            ::close(this->fd);
            this->fd = -1;
        } else {
            is_connected = true;
        }
    }

    freeaddrinfo(result);
    if (!is_connected) {
        throw Exception(ERR_MSG_CONNECTION_COULD_NOT_BE_STABLISHED, hostName, po
rt);
    }
}
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#ifndef __ClientSocket_H__
#define __ClientSocket_H__

#include <string>

#include "CommunicationSocket.h"

/**
 * Socket que tiene la capacidad de realizar una conexion con el servidor,
 * partiendo del dato del host y el port a donde conectarse
 */
class ClientSocket : public CommunicationSocket {
   public:
    ClientSocket(const char *hostName, const char *port);
};

#endif  //__ClientSocket_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#include <cmath>

#include "Cluster.h"

Worm::Cluster::Cluster(const GUI::GameTextureManager &tex)
    : Weapon(tex, GUI::GameTextures::WormCluster, CLUSTER_CENTER_FRAME, WeaponID
::WCluster),
      scope(this->textureMgr),
      powerBar(this->textureMgr) {}

void Worm::Cluster::update(float dt) {
    this->weaponAnimation.update(dt);
    this->scope.update(dt);
    this->powerBar.update(dt);
}

void Worm::Cluster::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip
&flip) {
    this->weaponAnimation.render(p, cam, flip);
    this->scope.render(p, cam, flip);
    this->powerBar.render(p, cam, flip);
}

void Worm::Cluster::setAngle(float angle, Worm::Direction d) {
    this->weaponAnimation.setFrame((int)std::ceil(angle / ANGLE_STEP) + this->ce
nterFrame);
    this->scope.setAngle(angle, d);
    this->powerBar.setAngle(angle, d);
}

void Worm::Cluster::startShot() {
    this->powerBar.startShot();
}

void Worm::Cluster::endShot() {
    this->powerBar.endShot();
}

bool Worm::Cluster::positionSelected() {
    return false;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#ifndef __CLUSTER_H__
#define __CLUSTER_H__

#include <vector>

#include "PowerBar.h"
#include "Scope.h"
#include "Weapon.h"

#define CLUSTER_CENTER_FRAME 15

namespace Worm {
class Cluster : public Weapon {
   public:
    explicit Cluster(const GUI::GameTextureManager &textureManager);
    ~Cluster() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;

   private:
    ::Weapon::Scope scope;
    ::Weapon::PowerBar powerBar;
};
}  // namespace Worm

#endif  //__CLUSTER_H__
```

```
//
// Created by rodrigo on 20/06/18.
//

#include <fstream>
#include <iostream>

#include "CommunicationProtocol.h"
#include "GameStateMsg.h"
#include "Stream.h"

IO::CommunicationProtocol::CommunicationProtocol(ClientSocket &socket, IO::Strea
m<IO::ClientGUIMsg> *clientStream,
                                                 IO::Stream<IO::ServerResponse>
*output)
        :
        protocol(socket),
        clientStream(clientStream),
        output(output) {
}

void IO::CommunicationProtocol::run() {
    try {
        while (!this->quit) {
            IO::ClientGUIMsg msg;
            if (clientStream->pop(msg)) {
                this->handleClientInput(msg);
            }
        }
    } catch (std::exception &e){
        if (!this->quit){
            std::cerr << "In CommunicationProtocol::run()" << std::endl;
            std::cerr << e.what() << std::endl;
            IO::ServerResponse sr{IO::ServerResponseAction::serverClosed};
            *this->output << sr;
        }
    } catch (...){
        std::cerr << "Unknown Error in CommunicationProtocol::run()" << std::endl;
    }
}

void IO::CommunicationProtocol::startCreateGame(){
    this->command = COMMAND_GET_LEVELS;
    this->protocol << this->command;
    this->protocol >> this->levelsInfo;
    *this->output << IO::ServerResponse{IO::ServerResponseAction::levelsInfo};
}

void IO::CommunicationProtocol::createGame() {
    this->command = COMMAND_CREATE_GAME;
    this->protocol << this->command;
    this->protocol << this->levelToCreate;
    this->getLevelFiles();
    this->waitGameStart(this->levelsInfo[this->levelToCreate].playersQuantity);
}

void IO::CommunicationProtocol::startJoinGame(){
    this->command = COMMAND_GET_GAMES;
    this->protocol << this->command;
    this->protocol >> this->gamesInfo;

    IO::ServerResponse sr;
```



```
    sr.action = IO::ServerResponseAction::gamesInfo;
    *this->output << sr;
}

void IO::CommunicationProtocol::joinGame() {
    this->command = COMMAND_JOIN_GAME;
    this->protocol << this->command;
    this->protocol << this->gameToJoin;
    this->protocol << this->levelOfGameToJoin;
    this->getLevelFiles();
    this->waitGameStart(this->gamesInfo[this->gameToJoin].numTotalPlayers);
}

ClientSocket IO::CommunicationProtocol::getSocket() {
    return std::move(this->protocol.getSocket());
}

void IO::CommunicationProtocol::waitGameStart(uint8_t playersQuantity) {
    while (this->playersQuantity < playersQuantity) {
        this->protocol >> this->playersQuantity;
        *this->output << IO::ServerResponse{IO::ServerResponseAction::playerConn
ected};
    }
    IO::ServerResponse sr{};
    sr.action = IO::ServerResponseAction::startGame;
    *this->output << sr;
}

void IO::CommunicationProtocol::stop() {
    this->quit = true;
    this->protocol.stopCommunication();
}

void IO::CommunicationProtocol::handleClientInput(IO::ClientGUIMsg &msg) {
    switch (msg.input) {
        case IO::ClientGUIInput::startCreateGame: {
            this->startCreateGame();
            break;
        }
        case IO::ClientGUIInput::levelSelected: {
            this->createGame();
            break;
        }
        case IO::ClientGUIInput::startJoinGame: {
            this->startJoinGame();
            break;
        }
        case IO::ClientGUIInput::joinGame: {
            this->joinGame();
            break;
        }
        case IO::ClientGUIInput::quit: {
            this->quit = true;
            break;
        }
        default: {
            break;
        }
    }
}

void IO::CommunicationProtocol::getLevelFiles() {
```

```
    this->protocol >> this->levelPath;
    std::ofstream levelFile(this->levelPath, std::ofstream::binary);
    this->protocol >> levelFile;

    this->protocol >> this->backgroundPath;
    for (auto &background : this->backgroundPath) {
        std::ofstream backgroundFile(background, std::ofstream::binary);
        this->protocol >> backgroundFile;
    }
}
```

```
//
// Created by rodrigo on 20/06/18.
//

#ifndef INC_4_WORMS_COMMUNICATIONPROTOCOL_H
#define INC_4_WORMS_COMMUNICATIONPROTOCOL_H


#include "ClientSocket.h"
#include <Protocol.h>
#include <Stream.h>
#include "Thread.h"

namespace IO {
    class CommunicationProtocol : public Thread {
    public:
        std::vector<LevelInfo> levelsInfo;
        uint8_t levelToCreate{0};
        std::vector<GameInfo> gamesInfo;
        uint8_t gameToJoin{0};
        uint8_t levelOfGameToJoin{0};
        std::string levelPath;
        std::vector<std::string> backgroundPath;


        explicit CommunicationProtocol(ClientSocket &socket, IO::Stream<IO::Clie
ntGUIMsg> *clientStream,
                                       IO::Stream<IO::ServerResponse> *output);

        void run() override;
        void stop() override;

        ClientSocket getSocket();

    private:
        Protocol<ClientSocket> protocol;
        unsigned char command{0};
        std::uint8_t playersQuantity{0};
        IO::Stream<IO::ClientGUIMsg> *clientStream;
        IO::Stream<IO::ServerResponse> *output;
        bool quit{false};

        void startCreateGame();
        void startJoinGame();
        void joinGame();
        void waitGameStart(uint8_t playersQuantity);

        void handleClientInput(ClientGUIMsg &msg);

        void createGame();

        void getLevelFiles();
    };
}


#endif //INC_4_WORMS_COMMUNICATIONPROTOCOL_H
```

```
//
// Created by rodrigo on 24/06/18.
//

#include "ConnectionWindow.h"

GUI::ConnectionWindow::ConnectionWindow(GUI::Window &window, GUI::Font &font, GU
I::Camera &cam) :
        GameWindow(window, font, cam) {
    std::string msg(CONNECT_MSG);
    this->buttons.emplace_back(msg, this->font, SDL_Color{0xFF, 0xFF, 0xFF}, thi
s->textSize);
    int x = this->window.getWidth() / 2;
    int y = this->window.getHeight() * 3 / 4;
    this->buttons.back().position = ScreenPosition{x, y};
    this->buttons.back().height = this->textSize * 3 / 2;
    this->buttons.back().width = this->buttons.back().msg.size() * 20 + 20;

    x = this->window.getWidth() * 6 / 10;
    y = this->window.getHeight() * 2 / 7;
    int textFieldHeight = this->textSize * 3 / 2;
    int textFieldWidth = 400;
    std::string emptyMsg(" ");
    this->textFields.emplace_back(emptyMsg, ScreenPosition{x, y}, textFieldHeigh
t, textFieldWidth, this->font);
    y = this->window.getHeight() * 4 / 7;
    emptyMsg = " ";
    this->textFields.emplace_back(emptyMsg, ScreenPosition{x, y}, textFieldHeigh
t, textFieldWidth, this->font);
}

void GUI::ConnectionWindow::start() {

}

void GUI::ConnectionWindow::render() {
    this->window.clear(SDL_Color{0xFF, 0xFF, 0xFF});

    SDL_Color black{0, 0, 0};

    Text ip{this->font};
    Text port{this->font};
    port.setBackground(black);
    int x = this->window.getWidth() * 3 / 10;
    int y = this->window.getHeight() * 2 / 7;
    ip.set("IP:", black, 50);
    ip.renderFixed(ScreenPosition{x, y}, this->cam);
    y = this->window.getHeight() * 4 / 7;
    ip.set("Server port:", black, 50);
    ip.renderFixed(ScreenPosition{x, y}, this->cam);

    for (auto &button : this->buttons) {
        button.render(this->cam);
    }

    for (auto &textField : this->textFields) {
        textField.render(this->cam);
    }

    this->window.render();
}
```



```
void GUI::ConnectionWindow::buttonPressed(GUI::ScreenPosition sp) {
    for (auto &textField : this->textFields) {
        textField.selected(sp);
    }

    if (this->buttons[0].inside(sp)) {
        this->notify(*this, Event::ConnectionToServer);
    }
}

void GUI::ConnectionWindow::appendCharacter(char *text) {
    for (auto &textField : this->textFields) {
        if (textField.focus) {
            textField.appendCharacter(text);
        }
    }
}

void GUI::ConnectionWindow::handleKeyDown(SDL_Keycode key) {
    switch (key) {
        case SDLK_BACKSPACE: {
            for (auto &textField : this->textFields) {
                if (textField.focus) {
                    textField.backSpace();
                }
            }
            break;
        }
    }
}

GUI::ConnectionInfo GUI::ConnectionWindow::getConnectionInfo() {
    return ConnectionInfo{this->textFields[0].inputText.msg.c_str(),
                          this->textFields[1].inputText.msg.c_str()};
}
```

```
//
// Created by rodrigo on 24/06/18.
//

#ifndef INC_4_WORMS_CONNECTIONWINDOW_H
#define INC_4_WORMS_CONNECTIONWINDOW_H


#include <vector>

#include "Window.h"
#include "Font.h"
#include "GameStateMsg.h"
#include "GameWindow.h"
#include "Button.h"

#define CONNECT_MSG "Connect"
#define IP_FOCUS 0
#define PORT_FOCUS 1

namespace GUI {
    struct ConnectionInfo {
        const char *ip;
        const char *port;
    };
    class ConnectionWindow : public GameWindow {
    public:
        uint8_t playersConnected{0};

        explicit ConnectionWindow(GUI::Window &window, GUI::Font &font, GUI::Cam
era &cam);

        void start() override;
        void render() override;
        void handleKeyDown(SDL_Keycode key) override;
        void appendCharacter(char text[32]) override;
        void buttonPressed(ScreenPosition sp) override;

        ConnectionInfo getConnectionInfo();

    private:
        std::vector<Button> buttons;
        int textSize{50};
    };
}


#endif //INC_4_WORMS_CONNECTIONWINDOW_H
```

```
//
// Created by rodrigo on 23/06/18.
//

#include <iostream>
#include "GameStateMsg.h"
#include "CreateGameWindow.h"

GUI::CreateGameWindow::CreateGameWindow(GUI::Window &window, GUI::Font &font, GUI::Camera &cam,
                                        std::vector<IO::LevelInfo> &levelsInfo)
:
        GameWindow(window, font, cam),
        levelsInfo(levelsInfo) {
    int height = this->levelInfoSize * 3 / 2;
    std::string msg(SELECT_LEVEL_MSG);
    int x = this->window.getWidth() / 2;
    int y = this->window.getHeight() * 3 / 4;
    this->buttons.emplace_back(msg, this->font, SDL_Color{0xFF, 0xFF, 0xFF}, this->levelInfoSize);
    this->buttons.back().position = ScreenPosition{x, y};
    this->buttons.back().height = height;
    this->buttons.back().width = this->buttons.back().msg.size() * 9 + 20;
    msg = NEXT_LEVEL_MSG;
    x = this->window.getWidth() * 3 / 4;
    y = this->window.getHeight() / 2;
    this->buttons.emplace_back(msg, this->font, SDL_Color{0xFF, 0xFF, 0xFF}, this->levelInfoSize);
    this->buttons.back().position = ScreenPosition{x, y};
    this->buttons.back().height = height;
    this->buttons.back().width = this->buttons.back().msg.size() * 9 + 20;
    msg = PREVIOUS_LEVEL_MSG;
    x = this->window.getWidth() / 4;
    y = this->window.getHeight() / 2;
    this->buttons.emplace_back(msg, this->font, SDL_Color{0xFF, 0xFF, 0xFF}, this->levelInfoSize);
    this->buttons.back().position = ScreenPosition{x, y};
    this->buttons.back().height = height;
    this->buttons.back().width = this->buttons.back().msg.size() * 9 + 20;
}

void GUI::CreateGameWindow::start() {

}

void GUI::CreateGameWindow::render() {
    this->window.clear(SDL_Color{0xFF, 0xFF, 0xFF});

    SDL_Color white{0xFF, 0xFF, 0xFF};
    SDL_Color black{0, 0, 0};

    Text levelName{this->font};
    Text levelPlayersQuantity{this->font};
    levelName.setBackground(black);
    levelPlayersQuantity.setBackground(black);
    int x = this->window.getWidth() * 4 / 10;
    int y = this->window.getHeight() * 3 / 7;
    levelName.set(LEVEL_MSG, white, 50);
    levelName.renderFixed(ScreenPosition{x, y - 50}, this->cam);
    x = this->window.getWidth() * 6 / 10;
    levelName.set(PLAYERS_MSG, white, 50);
    levelName.renderFixed(ScreenPosition{x, y - 50}, this->cam);
```



```
    levelName.setBackground(white);
    levelPlayersQuantity.setBackground(white);
    x = this->window.getWidth() * 4 / 10;
    y = this->window.getHeight() / 2;
    levelName.set(this->levelsInfo[this->buttonSelected].name, black, this->levelInfoSize);
    levelName.renderFixed(ScreenPosition{x, y}, this->cam);
    x = this->window.getWidth() * 6 / 10;
    levelName.set(std::to_string(this->levelsInfo[this->buttonSelected].playersQuantity), black, this->levelInfoSize);
    levelName.renderFixed(ScreenPosition{x, y}, this->cam);

    for (auto &button : this->buttons) {
        button.render(this->cam);
    }

    this->window.render();
}

void GUI::CreateGameWindow::buttonPressed(GUI::ScreenPosition sp) {
    if (this->buttons[0].inside(sp)) {
        this->notify(*this, Event::LevelSelected);
    }

    if (this->buttons[1].inside(sp)) {
        this->buttonSelected = (this->buttonSelected + 1) % this->levelsInfo.size();
    }

    if (this->buttons[2].inside(sp)) {
        this->buttonSelected = (this->buttonSelected == 0) ? this->levelsInfo.size() - 1 : this->buttonSelected - 1;
    }
}

void GUI::CreateGameWindow::appendCharacter(char *text) {

}

void GUI::CreateGameWindow::handleKeyDown(SDL_Keycode key) {

}
```

```
//
// Created by rodrigo on 23/06/18.
//

#ifndef INC_4_WORMS_CREATEGAMEWINDOW_H
#define INC_4_WORMS_CREATEGAMEWINDOW_H


#include <vector>

#include "Window.h"
#include "Font.h"
#include "GameStateMsg.h"
#include "GameWindow.h"
#include "Button.h"

#define SELECT_LEVEL_MSG "Select"
#define LEVEL_MSG "Level"
#define PLAYERS_MSG "Players"
#define NEXT_LEVEL_MSG "Next"
#define PREVIOUS_LEVEL_MSG "Previous"

namespace GUI {
    class CreateGameWindow : public GameWindow {
    public:
        std::vector<IO::LevelInfo> &levelsInfo;

        explicit CreateGameWindow(GUI::Window &window, GUI::Font &font, GUI::Cam
era &cam,
                                  std::vector<IO::LevelInfo> &levelsInfo);

        void start() override;
        void render() override;
        void handleKeyDown(SDL_Keycode key) override;
        void appendCharacter(char text[32]) override;
        void buttonPressed(ScreenPosition sp) override;

    private:
        std::vector<Button> buttons;
        int levelInfoSize{30};
    };
}


#endif //INC_4_WORMS_CREATEGAMEWINDOW_H
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#include <iostream>

#include "Dead.h"

Worm::Dead::Dead() : State(StateID::Dead) {}

Worm::Dead::~Dead() {}

void Worm::Dead::update(float dt) {
}

IO::PlayerInput Worm::Dead::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
```



```
}

IO::PlayerInput Worm::Dead::mortar(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Dead::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#ifndef __Dead_H__
#define __Dead_H__

#include "WormState.h"

namespace Worm {
class Dead : public State {
   public:
    Dead();
    ~Dead();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__Dead_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#include "Die.h"

Worm::Die::Die() : State(StateID::Die) {}

Worm::Die::~Die() {}

void Worm::Die::update(float dt) {}

IO::PlayerInput Worm::Die::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::mortar(Worm &w) {
```



```
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Die::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#ifndef __Die_H__
#define __Die_H__

#include "WormState.h"

namespace Worm {
class Die : public State {
   public:
    Die();
    ~Die();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__Die_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 29/05/18
 */

#include "Drowning.h"

Worm::Drowning::Drowning() : State(StateID::Drowning) {}

Worm::Drowning::~Drowning() {}

void Worm::Drowning::update(float dt) {}

IO::PlayerInput Worm::Drowning::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::mortar(Worm &w) {
```



```
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Drowning::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 29/05/18
 */

#ifndef __Drown_H__
#define __Drown_H__

#include "WormState.h"

namespace Worm {
class Drowning : public State {
   public:
    Drowning();
    ~Drowning();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__Drown_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 16/06/18
 */

#include "Dynamite.h"

Worm::Dynamite::Dynamite(const GUI::GameTextureManager &tex)
    : Weapon(tex, GUI::GameTextures::WormDynamite, DYNAMITE_CENTER_FRAME, Weapon
ID::WDynamite) {}

void Worm::Dynamite::update(float dt) {
    this->weaponAnimation.update(dt);
}

void Worm::Dynamite::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip
 &flip) {
    this->weaponAnimation.render(p, cam, flip);
}

void Worm::Dynamite::setAngle(float angle, Worm::Direction d) {}

void Worm::Dynamite::startShot() {}

void Worm::Dynamite::endShot() {}

bool Worm::Dynamite::positionSelected() {
    return false;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 16/06/18
 */

#ifndef __DYNAMITE_H__
#define __DYNAMITE_H__

#include "Weapon.h"

#define DYNAMITE_CENTER_FRAME 0

namespace Worm {
class Dynamite : public Weapon {
   public:
    explicit Dynamite(const GUI::GameTextureManager &textureManager);
    ~Dynamite() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;
};
}  // namespace Worm

#endif  //__DYNAMITE_H__
```

```
//
// Created by rodrigo on 2/06/18.
//

#include "Explosion.h"

Worm::Explosion::Explosion(const GUI::GameTextureManager &texture_mgr) : texture
_mgr(texture_mgr) {
    this->animations.emplace_back(this->texture_mgr.get(GUI::GameTextures::Explo
sion));
    this->animations.back().setAnimateOnce();
}
// TODO make observer in client side to clean exploded bullet
void Worm::Explosion::update(float dt) {
    for (auto &animation : this->animations) {
        animation.update(dt);
        this->explosionFinished = animation.finished();
    }
}

void Worm::Explosion::render(GUI::Camera &cam) {
    for (auto &animation : this->animations) {
        animation.render(this->position, cam, SDL_FLIP_HORIZONTAL);
    }
}

bool Worm::Explosion::finished() {
    return this->explosionFinished;
}
```

```
//
// Created by rodrigo on 2/06/18.
//

#ifndef INC_4_WORMS_EXPLOSION_H
#define INC_4_WORMS_EXPLOSION_H

#include <Animation.h>
#include <vector>
#include "../GameTextures.h"

namespace Worm {
class Explosion {
   public:
    explicit Explosion(const GUI::GameTextureManager &texture_mgr);
    ~Explosion() = default;
    void update(float dt);
    void render(GUI::Camera &cam);
    GUI::Position position{0, 0};
    bool finished();

   private:
    const GUI::GameTextureManager &texture_mgr;
    std::vector<GUI::Animation> animations;
    bool explosionFinished{false};
};
}

#endif  // INC_4_WORMS_EXPLOSION_H
```

```
//
// Created by rodrigo on 3/06/18.
//

#include "Falling.h"

Worm::Falling::Falling() : State(StateID::Falling) {}

Worm::Falling::~Falling() {}

void Worm::Falling::update(float dt) {}

IO::PlayerInput Worm::Falling::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::mortar(Worm &w) {
    return IO::PlayerInput::moveNone;
```



```
}

IO::PlayerInput Worm::Falling::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Falling::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
//
// Created by rodrigo on 3/06/18.
//

#ifndef INC_4_WORMS_FALLING_H
#define INC_4_WORMS_FALLING_H

#include "GameStateMsg.h"
#include "WormState.h"

namespace Worm {
class Falling : public State {
   public:
    Falling();
    ~Falling();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;

    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  // INC_4_WORMS_FALLING_H
```

```
//
// Created by rodrigo on 25/06/18.
//

#ifndef INC_4_WORMS_GAMEBACKGROUNDMUSIC_H
#define INC_4_WORMS_GAMEBACKGROUNDMUSIC_H

#include "BackgroundMusicManager.h"
#include "utils.h"

namespace GUI {
/** Different kinds of background music. */
    enum class GameBackgroundMusic {
        Original,
        MurderTrain
    };

/** Specialized BackgroundMusicManager class. */
    using GameBackgroundMusicManager = BackgroundMusicManager<GameBackgroundMusi
c, Utils::EnumClassHash>;
}  // namespace GUI

#endif //INC_4_WORMS_GAMEBACKGROUNDMUSIC_H
```

```
//
// Created by rodrigo on 26/06/18.
//

#include "GameEndWindow.h"

GUI::GameEndWindow::GameEndWindow(GUI::Window &window, GUI::Font &font, GUI::Cam
era &cam, bool youWin) :
        GameWindow(window, font, cam) {
    this->gameEndResultMsg = youWin ? "You Win!" : "You Lose!";
}

void GUI::GameEndWindow::start() {
    while (!this->quit) {
        SDL_Event e;
        while (SDL_PollEvent(&e) != 0) {
            switch (e.type) {
                case SDL_QUIT: {
                    this->quit = true;
//                    throw;
                    break;
                }
                default: {
                    break;
                }
            }
        }

        this->render();
    }
}

void GUI::GameEndWindow::render() {
    this->window.clear(SDL_Color{0xFF, 0xFF, 0xFF});

    SDL_Color black{0, 0, 0};

    Text gameResult{this->font};
    int x = this->window.getWidth() / 2;
    int y = this->window.getHeight() / 2;
    gameResult.set(this->gameEndResultMsg, black, 50);
    gameResult.renderFixed(ScreenPosition{x, y}, this->cam);

    this->window.render();
}

void GUI::GameEndWindow::buttonPressed(GUI::ScreenPosition sp) {
}

void GUI::GameEndWindow::appendCharacter(char *text) {
}

void GUI::GameEndWindow::handleKeyDown(SDL_Keycode key) {
}
```

```
//
// Created by rodrigo on 26/06/18.
//

#ifndef INC_4_WORMS_GAMEENDWINDOW_H
#define INC_4_WORMS_GAMEENDWINDOW_H


#include <vector>

#include "Window.h"
#include "Font.h"
#include "GameStateMsg.h"
#include "GameWindow.h"
#include "Button.h"


namespace GUI {
    class GameEndWindow : public GameWindow {
    public:
        explicit GameEndWindow(GUI::Window &window, GUI::Font &font, GUI::Camera
 &cam, bool youWin);

        void start() override;
        void render() override;
        void handleKeyDown(SDL_Keycode key) override;
        void appendCharacter(char text[32]) override;
        void buttonPressed(ScreenPosition sp) override;

    private:
        std::vector<Button> buttons;
        int textSize{50};
        std::string gameEndResultMsg;
    };
}


#endif //INC_4_WORMS_GAMEENDWINDOW_H
```

```
//
// Created by rodrigo on 4/06/18.
//

#ifndef INC_4_WORMS_GAMESOUNDEFFECTS_H
#define INC_4_WORMS_GAMESOUNDEFFECTS_H

#include "SoundEffectManager.h"
#include "utils.h"

namespace GUI {
/** Different kinds of sound effects. */
enum class GameSoundEffects {
    WalkCompress,
    Explosion,
    WormLanding,
    WormDrowning,
    Splash,
    WormJump,
    WormBackFlip,
    WormHit,
    WormDie,
    Holy,
    AirStrike,
    Teleport,
    Shot,
    Banana
};

/** Specialized SoundEffectManager class. */
using GameSoundEffectManager = SoundEffectManager<GameSoundEffects, Utils::EnumC
lassHash>;
}  // namespace GUI

#endif  // INC_4_WORMS_GAMESOUNDEFFECTS_H
```

```
#ifndef GAME_TEXTURES_H_
#define GAME_TEXTURES_H_

#include "TextureManager.h"
#include "utils.h"

namespace GUI {
/** Different kinds of textures. */
enum class GameTextures {
    WormWalk,
    WormIdle,
    LongGirder,
    ShortGirder,
    StartJump,
    Jumping,
    EndJump,
    BackFlipping,
    Bazooka,
    Missile,
    Fly,
    Die,
    Dead,
    Sliding,
    StaticBackground,
    Background1,
    Background2,
    Background3,
    WormGrenade,
    Grenade,
    WormCluster,
    Cluster,
    Mortar,
    Bazooka2,
    WormBanana,
    Banana,
    WormHoly,
    Holy,
    Explosion,
    Flame,
    Smoke,
    Falling,
    Scope,
    PowerBar,
    Fragment,
    BazookaIcon,
    GrenadeIcon,
    ClusterIcon,
    MortarIcon,
    BananaIcon,
    HolyIcon,
    WormAirAttack,
    AirMissile,
    AirIcon,
    WormDynamite,
    Dynamite,
    DynamiteIcon,
    WormTeleport,
    WormTeleporting,
    TeleportIcon,
    WormBaseballBat,
    WormBaseballBatting,
    BaseballBatIcon,
```



```
    WindLeft,
    WindRight,
    CurrentPlayerArrow,
    Water,
};

/** Specialized TextureManager class. */
using GameTextureManager = TextureManager<GameTextures, Utils::EnumClassHash>;
}  // namespace GUI

#endif
```

```
//
// Created by rodrigo on 19/06/18.
//

#include <Font.h>
#include <Camera.h>
#include "GameWindow.h"

GUI::GameWindow::GameWindow(Window &window, Font &font, Camera &cam) :
        window(window),
        font(font),
        cam(cam) {
}
```

```
//
// Created by rodrigo on 19/06/18.
//

#ifndef INC_4_WORMS_GAMEWINDOW_H
#define INC_4_WORMS_GAMEWINDOW_H


#include <vector>

#include "Button.h"
#include "Camera.h"
#include "Font.h"
#include "Subject.h"
#include "Window.h"

#define ASSETS_PATH "/var/Worms/assets"


namespace GUI {
    struct TextField {
        TextField(std::string &text, ScreenPosition sp, int height, int width, F
ont &font) :
                inputText(sp, height, width, text, font),
                focus(false) {};

        void selected(ScreenPosition sp) {
            this->focus = inputText.inside(sp);
        };

        void render(GUI::Camera &cam) {
            this->inputText.render(cam);
        };

        void appendCharacter(char *text) {
            if (this->emptyString) {
                this->inputText.msg = text;
                this->emptyString = false;
            } else {
                this->inputText.msg += text;
            }
        };

        void backSpace() {
            if (!this->emptyString) {
                this->inputText.msg.pop_back();
                if (this->inputText.msg.length() == 0) {
                    this->inputText.msg = " ";
                    this->emptyString = true;
                }
            }
        };

        Button inputText;
        bool focus;

    private:
        bool emptyString{true};
    };

    class GameWindow : public Subject {
    public:
```



```
        uint8_t buttonSelected{0};

        explicit GameWindow(Window &window, Font &font, Camera &cam);

        virtual void start() = 0;
        virtual void render() = 0;
        virtual void handleKeyDown(SDL_Keycode key) = 0;
        virtual void appendCharacter(char text[32]) = 0;
        virtual void buttonPressed(ScreenPosition sp) = 0;

    protected:
        Window &window;
        Font &font;
        Camera &cam;
        std::vector<TextField> textFields;
        bool quit{false};
    };
}


#endif //INC_4_WORMS_GAMEWINDOW_H
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#include <cmath>

#include "Grenade.h"

Worm::Grenade::Grenade(const GUI::GameTextureManager &tex)
    : Weapon(tex, GUI::GameTextures::WormGrenade, GRENADE_CENTER_FRAME, WeaponID
::WGrenade),
      scope(this->textureMgr),
      powerBar(this->textureMgr) {}

void Worm::Grenade::update(float dt) {
    this->weaponAnimation.update(dt);
    this->scope.update(dt);
    this->powerBar.update(dt);
}

void Worm::Grenade::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip
&flip) {
    this->weaponAnimation.render(p, cam, flip);
    this->scope.render(p, cam, flip);
    this->powerBar.render(p, cam, flip);
}

void Worm::Grenade::setAngle(float angle, Worm::Direction d) {
    this->weaponAnimation.setFrame((int)std::ceil(angle / ANGLE_STEP) + this->ce
nterFrame);
    this->scope.setAngle(angle, d);
    this->powerBar.setAngle(angle, d);
}

void Worm::Grenade::startShot() {
    this->powerBar.startShot();
}

void Worm::Grenade::endShot() {
    this->powerBar.endShot();
}

bool Worm::Grenade::positionSelected() {
    return false;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#ifndef __GRENADE_H__
#define __GRENADE_H__

#include <Camera.h>

#include "../GameTextures.h"
#include "Direction.h"
#include "PowerBar.h"
#include "Scope.h"
#include "Weapon.h"

#define GRENADE_CENTER_FRAME 15

namespace Worm {
class Grenade : public Weapon {
   public:
    explicit Grenade(const GUI::GameTextureManager &textureManager);
    ~Grenade() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;

   private:
    ::Weapon::Scope scope;
    ::Weapon::PowerBar powerBar;
};
}  // namespace Worm

#endif  //__GRENADE_H__
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 17/05/18.
 */

#include <SDL2/SDL.h>
#include <unistd.h>
#include <cmath>
#include <iostream>
#include <sstream>

#include "GameStateMsg.h"
#include "GameWindow.h"
#include "GUIGame.h"
#include "Stream.h"
#include "Text.h"
#include "Weapons/Bullet.h"
#include "Window.h"
#include "WrapTexture.h"


// TODO DEHARDCODE
GUI::Game::Game(Window &w, Worms::Stage &&stage, std::vector<std::string> &backg
roundPaths, ClientSocket &socket,
                std::uint8_t team)
    : window(w),
      texture_mgr(w.getRenderer()),
      sound_effect_mgr(),
      stage(stage),
      cam(w, this->scale, this->stage.getWidth(), this->stage.getHeight()),
      font(std::string(ASSETS_PATH) + "/fonts/gruen_lemonograf.ttf", 28),
      armory(this->texture_mgr, this->cam, this->font),
      socket(socket),
      team(team),
      wind(this->texture_mgr, this->cam),
      water(this->texture_mgr) {

    this->loadTextureManager();
    this->loadSoundManager();
    this->loadBackgroundManager();

    /* updates the armory */
    this->armory.loadWeapons();
    /* allocates space in the array to avoid the player addresses from changing
*/
    int num_worms = 0;
    this->worms.reserve(stage.getWorms().size());
    for (const auto &wormData : this->stage.getWorms()) {
        this->worms.emplace_back(num_worms, this->texture_mgr, this->sound_effec
t_mgr);
        this->snapshot.positions[num_worms * 2] = wormData.position.x;
        this->snapshot.positions[num_worms * 2 + 1] = wormData.position.y;
        this->snapshot.wormsHealth[num_worms] = wormData.health;
        num_worms += 1;
    }

    this->snapshot.num_worms = num_worms;
    //    this->snapshot.processingInputs = true;

    this->teamColors.push_back(SDL_Color{0xFF, 0, 0});
    this->teamColors.push_back(SDL_Color{0, 0xFF, 0});
    this->teamColors.push_back(SDL_Color{0, 0, 0xFF});
```



```
    this->teamColors.push_back(SDL_Color{0xFF, 0, 0xFF});

    this->currentPlayerArrow = std::unique_ptr<GUI::Animation>(
        new GUI::Animation(this->texture_mgr.get(GUI::GameTextures::CurrentPlaye
rArrow), false));
    this->inputThread = std::thread([this] { this->inputWorker(); });
    this->outputThread = std::thread([this] { this->outputWorker(); });

    this->backGroundMusicPlayer =
            std::unique_ptr<GUI::BackgroundMusicPlayer>(new GUI::BackgroundMusic
Player{
                    this->background_music_mgr.get(GUI::GameBackgroundMusic::Mur
derTrain)});
    this->backGroundMusicPlayer->play();
}

GUI::Game::~Game() {
    this->exit();
    this->outputThread.join();
    this->inputThread.join();
}

void GUI::Game::inputWorker() {
    IO::GameStateMsg msg;
    try {
        while (!this->quit) {
            /* receives the size of the msg */
            std::uint32_t size(0);
            socket.receive((char *)&size, sizeof(std::uint32_t));
            size = ntohl(size);

            std::vector<char> buffer(size, 0);
            /* reads the raw data from the buffer */
            socket.receive(buffer.data(), size);

            std::string buff(buffer.data(), size);

            /* sets the struct data from the buffer */
            msg.deserialize(buff);
            this->snapshotBuffer.set(msg);
            this->snapshotBuffer.swap();
        }
    } catch (const std::exception &e) {
        std::cerr << "GUI::Game::inputWorker:" << e.what() << std::endl;
    } catch (...) {
        std::cerr << "Unknown error in GUI::Game::inputWorker()" << std::endl;
    }
}

void GUI::Game::outputWorker() {
    IO::PlayerMsg msg;
    try {
        while (!this->quit) {
            this->output.pop(msg, true);
            std::string buff = msg.serialize();
            std::uint32_t size = buff.size();
            std::uint32_t netSize = htonl(size);

            this->socket.send((char *)&netSize, sizeof(std::uint32_t));
            this->socket.send(buff.c_str(), size);
        }
    } catch (const std::exception &e) {
```

```
        std::cerr << "GUI::Game::outputWorker:" << e.what() << std::endl;
    }
}

//void GUI::Game::inputWorker() {
//    IO::GameStateMsg msg;
//    char *buffer = new char[msg.getSerializedSize()];
//
//    try {
//        while (!this->quit) {
//            this->socket.receive(buffer, msg.getSerializedSize());
//            msg.deserialize(buffer, msg.getSerializedSize());
//            this->snapshotBuffer.set(msg);
//            this->snapshotBuffer.swap();
//        }
//    } catch (const std::exception &e) {
//        std::cerr << "GUI::Game::inputWorker:" << e.what() << std::endl;
//    } catch (...) {
//        std::cerr << "Unknown error in GUI::Game::inputWorker()" << std::endl;
//    }
//
//    delete[] buffer;
//}
//
//void GUI::Game::outputWorker() {
//    IO::PlayerMsg msg;
//    char *buffer = new char[msg.getSerializedSize()];
//
//    try {
//        while (!this->quit) {
//            this->output.pop(msg, true);
//            msg.serialize(buffer, msg.getSerializedSize());
//            this->socket.send(buffer, msg.getSerializedSize());
//        }
//    } catch (const std::exception &e) {
//        std::cerr << "GUI::Game::outputWorker:" << e.what() << std::endl;
//    } catch (...) {
//        std::cerr << "Unknown error in GUI::Game::outputWorker()" << std::endl
;
//    }
//
//    delete[] buffer;
//}

void GUI::Game::start() {
    try {
        uint32_t prev = SDL_GetTicks();
        while (!this->quit) {
            /* updates the snapshot */
            this->snapshot = this->snapshotBuffer.get();
            if (!this->snapshot.gameEnded) {
                Worm::Worm &cur = this->worms[this->snapshot.currentWorm];

                /* handle events on queue */
                SDL_Event e;
                while (SDL_PollEvent(&e) != 0) {
                    switch (e.type) {
                        case SDL_QUIT:
                            this->exit();
                            break;
                        case SDL_KEYDOWN:
                            if (this->snapshot.processingInputs &&
```



```
                                this->team == this->snapshot.currentTeam) {
                                cur.handleKeyDown(e.key.keysym.sym, &this->outpu
t);
                            }
                            break;
                        case SDL_KEYUP:
                            if (this->snapshot.processingInputs &&
                                this->team == this->snapshot.currentTeam) {
                                cur.handleKeyUp(e.key.keysym.sym, &this->output)
;
                            }
                            break;
                        case SDL_MOUSEBUTTONDOWN: {
                            if (this->snapshot.processingInputs &&
                                this->team == this->snapshot.currentTeam) {
                                int x, y;
                                SDL_GetMouseState(&x, &y);
                                GUI::Position global =
                                    this->cam.screenToGlobal(GUI::ScreenPosition
{x, y});
                                cur.mouseButtonDown(global, &this->output);
                                break;
                            }
                        }
                        default:
                            break;
                    }
                }

                /* synchronizes the worms states with the server's */
                for (std::size_t i = 0; i < this->worms.size(); i++) {
                    this->worms[i].setState(this->snapshot.stateIDs[i]);
                    this->worms[i].setWeapon((i != this->snapshot.currentWorm)
                                                 ? Worm::WeaponID::WNone
                                                 : this->snapshot.activePlayerWe
apon);
                }

                if (cur.getState() == Worm::StateID::Still &&
                    cur.getWeaponID() != Worm::WeaponID::WNone) {
                    cur.setWeaponAngle(this->snapshot.activePlayerAngle);
                }
                if (this->snapshot.bulletsQuantity == 0 && this->snapshot.player
UsedTool) {
                    this->bullets.erase(this->bullets.begin(), this->bullets.end
());
                    this->explodedQuantity = 0;
                    this->worms[this->snapshot.currentWorm].reset();
                }
                if (this->snapshot.bulletsQuantity > 0) {
                    for (int i = this->bullets.size(); i < this->snapshot.bullet
sQuantity; i++) {
                        std::shared_ptr<Ammo::Bullet> p(
                            new Ammo::Bullet(this->texture_mgr, this->sound_effe
ct_mgr,
                                             this->snapshot.bulletType[i]));
                        this->bullets.emplace_back(p);
                    }
                    int i = 0;
                    for (auto &bullet : this->bullets) {
                        if (this->snapshot.bulletType[i] == Worm::WeaponID::WExp
lode &&
```

```
                            !bullet->exploding()) {
                            bullet->madeImpact();
                            this->explodedQuantity++;
                        }
                        bullet->setAngle(this->snapshot.bulletsAngle[i++]);
                    }
                }

                uint32_t current = SDL_GetTicks();
                float dt = static_cast<float>(current - prev) / 1000.0f;
                prev = current;

                this->handleCamera(dt);
                this->update(dt);

                this->render();
            } else {
                this->youWin = this->snapshot.winner == this->team;
                this->quit = true;
            }
        }
    } catch (std::exception &e) {
        std::cerr << e.what() << std::endl << "In GUI::Game::start" << std::endl;
    } catch (...) {
        std::cerr << "Unkown error in GUI::Game::start()." << std::endl;
    }
}

void GUI::Game::update(float dt) {
    for (auto &worm : this->worms) {
        worm.health = this->snapshot.wormsHealth[static_cast<int>(worm.id)];
        worm.direction = this->snapshot.wormsDirection[static_cast<int>(worm.id)
];
        worm.update(dt);
    }
    if (this->snapshot.waitingForNextTurn) {
        this->armory.update(this->snapshot);
        this->currentPlayerArrow->update(dt);
    } else {
        this->currentPlayerArrow->setFrame(0);
    }

    this->cam.update(dt);

    for (auto &bullet : this->bullets) {
        bullet->update(dt);
    }

    this->water.update(dt);
}

void GUI::Game::render() {
    this->renderBackground();

    for (uint8_t i = 0; i < this->snapshot.num_worms; i++) {
        float cur_x = this->snapshot.positions[i * 2];
        float cur_y = this->snapshot.positions[i * 2 + 1];

        GUI::Position p{cur_x, cur_y};
        this->worms[i].setPosition(p);
        this->worms[i].render(p, this->cam);
    }
```



```
    for (auto &girder : this->stage.getGirders()) {
        const GUI::Texture &texture = this->texture_mgr.get(GUI::GameTextures::L
ongGirder);

        GUI::WrapTexture wt{texture, girder.length, girder.height};
        wt.render(GUI::Position{girder.pos.x, girder.pos.y}, girder.angle, this-
>cam);
    }

    int i = 0, j = 0;
    for (auto &bullet : this->bullets) {
        float local_x = this->snapshot.bullets[i++];
        float local_y = this->snapshot.bullets[i++];
        if (!bullet->exploding()) {
            bullet->setAngle(this->snapshot.bulletsAngle[j++]);
            bullet->setPosition(GUI::Position{local_x, local_y});
        }

        if (!bullet->exploded()) {
            bullet->render(GUI::Position{local_x, local_y}, this->cam);
        }
    }

    /* health bars are renderer after the worms so they appear on top */
    for (uint8_t i = 0; i < this->snapshot.num_worms; i++) {
        float cur_x = this->snapshot.positions[i * 2];
        float cur_y = this->snapshot.positions[i * 2 + 1];
        if (this->worms[i].getState() != Worm::StateID::Dead) {
            Text health{this->font};
            health.setBackground(SDL_Color{0, 0, 0});
            health.set(std::to_string(static_cast<int>(this->worms[i].health)),
                       this->teamColors[this->snapshot.wormsTeam[i]], 20);
            health.render(GUI::Position{cur_x, cur_y + 2.2f}, this->cam);
        }
    }

    this->water.render(this->cam);

    this->renderStatic();

    this->window.render();
}

/**
 * @brief interrupts all current game operations and leaves the main loop.
 *
 */
void GUI::Game::exit() {
    this->quit = true;
    this->output.close();
    this->socket.shutdown();
}

/**
 * @brief Renders the background images using a parallax effect.
 *
 */
void GUI::Game::renderBackground() {
    SDL_Color bgColor{this->stage.backgroundColor.r, this->stage.backgroundColor
.g, this->stage.backgroundColor.b};
    this->window.clear(bgColor);
```

```
    /* draws moving image further in the background */
    const Texture &Bg1Tex = this->texture_mgr.get(GameTextures::Background1);
    // TODO: use the stage size
    WrapTexture bg1{Bg1Tex, this->stage.getWidth(), Bg1Tex.getHeight() / this->c
am.getScale()};

    Position pos{0.0f, (Bg1Tex.getHeight() / this->cam.getScale()) / 2};
    pos.x += this->cam.getPosition().x * 0.8f;
    bg1.render(pos, this->cam);

    /* draws a moving image in the background at intermediate distance */
    const Texture &Bg2Tex = this->texture_mgr.get(GameTextures::Background2);
    // TODO: use the stage size
    WrapTexture bg2{Bg2Tex, this->stage.getWidth(), Bg2Tex.getHeight() / this->c
am.getScale()};

    pos = {0.0f, (Bg2Tex.getHeight() / this->cam.getScale()) / 2};
    pos.x += this->cam.getPosition().x * 0.6f;
    bg2.render(pos, this->cam);

    /* draws a moving image in the background at a closer distance */
    const Texture &Bg3Tex = this->texture_mgr.get(GameTextures::Background3);
    // TODO: use the stage size
    WrapTexture bg3{Bg3Tex, this->stage.getWidth(), Bg3Tex.getHeight() / this->c
am.getScale()};

    pos = {0.0f, (Bg3Tex.getHeight() / this->cam.getScale()) / 2};
    pos.x += this->cam.getPosition().x * 0.25f;
    bg3.render(pos, this->cam);
}

/**
 * @brief Draws the game controls.
 */
void GUI::Game::renderStatic() {

    /* render the arrow to notify the current player when wainting for next turn
*/
    if (this->snapshot.waitingForNextTurn) {
        float cur_x = this->snapshot.positions[this->snapshot.currentWorm * 2];
        float cur_y = this->snapshot.positions[this->snapshot.currentWorm * 2 +
1];
        GUI::Position position = GUI::Position{cur_x, cur_y + 4.4f};
        this->currentPlayerArrow->render(position, this->cam, SDL_FLIP_NONE);
    }

    /* health bars of the team */
    uint8_t numTeams = this->snapshot.num_teams;
    int textHeight = 25;
    for (uint8_t i = 0; i < numTeams; i++) {
        Text health{this->font};
        std::ostringstream oss;
        oss << "Team " << i + 1 << ":" << this->snapshot.teamHealths[i];

        health.setBackground(SDL_Color{0, 0, 0});
        health.set(oss.str(), this->teamColors[i], textHeight);
        int x = this->window.getWidth() / 2;
        int y = this->window.getHeight() - (textHeight * (numTeams - i));
        health.renderFixed(ScreenPosition{x, y}, this->cam);
    }
```



```
    /* displays the remaining turn time */
    std::int16_t turnTimeLeft =
        this->snapshot.currentPlayerTurnTime - this->snapshot.elapsedTurnSeconds
;
    turnTimeLeft = (turnTimeLeft < 0) ? 0 : turnTimeLeft;

    int x = this->window.getWidth() / 2;
    int y = 20;

    SDL_Color color = {0, 0, 0};
    Text text{this->font};
    text.set(std::to_string(turnTimeLeft), color);
    text.renderFixed(ScreenPosition{x, y}, this->cam);

    /* renders armory */
    this->armory.render();

    this->wind.render(this->snapshot.windIntensity, this->window.getWidth());
}

/**
 * @brief Handles the camera actions.
 *
 * @param dt Seconds elapsed since the last call to this function.
 */
void GUI::Game::handleCamera(float dt) {
    this->lastCameraUpdate += dt;

    /* checks the mouse to see if the user wishes to move the camera */
    int mx, my;
    SDL_GetMouseState(&mx, &my);

    const float cameraSpeed = 15.0f;
    const int cameraMargin = 50;

    /* checks if the camera should be moved horizontally */
    if (this->window.containsMouse()) {
        if (mx < cameraMargin) {
            auto p = this->cam.getPosition() - GUI::Position{cameraSpeed, 0.0f}
* dt;
            this->cam.moveTo(this->cam.getPosition() - GUI::Position{cameraSpeed
, 0.0f} * dt);
            this->lastCameraUpdate = 0.0f;
        } else if (mx > this->window.getWidth() - cameraMargin) {
            this->cam.moveTo(this->cam.getPosition() + GUI::Position{cameraSpeed
, 0.0f} * dt);
            this->lastCameraUpdate = 0.0f;
        }

        /* checks if the camera should be moved vertically */
        if (my < cameraMargin) {
            this->cam.moveTo(this->cam.getPosition() + GUI::Position{0.0f, camer
aSpeed} * dt);
            this->lastCameraUpdate = 0.0f;
        } else if (my > this->window.getHeight() - cameraMargin) {
            this->cam.moveTo(this->cam.getPosition() - GUI::Position{0.0f, camer
aSpeed} * dt);
            this->lastCameraUpdate = 0.0f;
        }
    }

    /* if the user hasn't changed the camera in a while, it becomes automatic ag
```

```
ain */
    if (this->lastCameraUpdate < 2.0f) {
        return;
    } else {
        /* avoids overflow */
        this->lastCameraUpdate = 2.0f;
    }

    /* move the camera to the current player */
    if (this->snapshot.bulletsQuantity > this->explodedQuantity) {
        float cur_x{0};
        float cur_y{0};
        int i{0};
        for (int j = 0; i < this->snapshot.bulletsQuantity; i++) {
            if (this->snapshot.bulletType[i] != Worm::WExplode) {
                cur_x = this->snapshot.bullets[j++];
                cur_y = this->snapshot.bullets[j];
                break;
            }
            j += 2;
        }

        this->cam.moveTo(GUI::Position{cur_x, cur_y});
    } else {
        float cur_follow_x = this->snapshot.positions[this->snapshot.currentWorm
ToFollow * 2];
        float cur_follow_y = this->snapshot.positions[this->snapshot.currentWorm
ToFollow * 2 + 1];

        /* move the camera to the current player */
        this->cam.moveTo(GUI::Position{cur_follow_x, cur_follow_y});
    }
}

void GUI::Game::loadTextureManager(){
    std::string path(ASSETS_PATH);
    /* loads the required textures */
    this->texture_mgr.load(GUI::GameTextures::CurrentPlayerArrow, path + "/img/Mi
sc/arrowdnb.png",
                           GUI::Color{0x40, 0x40, 0x80});
    this->texture_mgr.load(GUI::GameTextures::WindLeft, path + "/img/Misc/windl.png"
,
                           GUI::Color{0x00, 0x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::WindRight, path + "/img/Misc/windr.png
",
                           GUI::Color{0x00, 0x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::WormWalk, path + "/img/Worms/wwalk2.p
ng",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::WormIdle, path + "/img/Worms/wbrth1.pn
g",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::LongGirder, path + "/img/Weapons/grdl4
.png",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::StartJump, path + "/img/Worms/wjump.p
ng",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Jumping, path + "/img/Worms/wflyup.png
",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::EndJump, path + "/img/Worms/wland2.png
```



```
",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::BackFlipping, path + "/img/Worms/wba
ckflp.png",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Falling, path + "/img/Worms/wfall.png"
,
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Bazooka, path + "/img/Worms/wbaz.png"
,
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Fly, path + "/img/Worms/wfly1.png",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Die, path + "/img/Worms/wdie.png",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Sliding, path + "/img/Worms/wslided.png
",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Dead, path + "/img/Misc/grave4.png",
                           GUI::Color{0xC0, 0xC0, 0x80});
    this->texture_mgr.load(GUI::GameTextures::Missile, path + "/img/Weapons/missile.pn
g",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Explosion, path + "/img/Effects/circle25.
png",
                           GUI::Color{0x80, 0x80, 0xC0});
    this->texture_mgr.load(GUI::GameTextures::Flame, path + "/img/Effects/flame1.png",
                           GUI::Color{0x80, 0x80, 0xC0});
    this->texture_mgr.load(GUI::GameTextures::Smoke, path + "/img/Effects/smkdrk20.png
",
                           GUI::Color{0xC0, 0xC0, 0x80});
    this->texture_mgr.load(GUI::GameTextures::Background1, path + "/img/background/b
g1.png",
                           GUI::Color{0xff, 0xff, 0xff});
    this->texture_mgr.load(GUI::GameTextures::Background2, path + "/img/background/b
g2.png",
                           GUI::Color{0xff, 0xff, 0xff});
    this->texture_mgr.load(GUI::GameTextures::Background3, path + "/img/background/b
g3.png",
                           GUI::Color{0xff, 0xff, 0xff});
    this->texture_mgr.load(GUI::GameTextures::WormGrenade, path + "/img/Worms/wthrg
rn.png",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Grenade, path + "/img/Weapons/grenade.p
ng",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::WormCluster, path + "/img/Worms/wthrc
ls.png",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Cluster, path + "/img/Weapons/cluster.pn
g",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Mortar, path + "/img/Weapons/mortar.png
",
                           GUI::Color{0xc0, 0xc0, 0x80});
    this->texture_mgr.load(GUI::GameTextures::Bazooka2, path + "/img/Worms/wbaz2.pn
g",
                           GUI::Color{0xc0, 0xc0, 0x80});
    this->texture_mgr.load(GUI::GameTextures::Banana, path + "/img/Weapons/banana.png
",
                           GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::WormBanana, path + "/img/Worms/wthrba
```

```
n.png",
                            GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Holy, path + "/img/Weapons/hgrenade.png
",
                            GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::WormHoly, path + "/img/Worms/wthrhgrd.
png",
                            GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Scope, path + "/img/Misc/crshairb.png",
                            GUI::Color{0x40, 0x40, 0x80});
    this->texture_mgr.load(GUI::GameTextures::Scope, path + "/img/Misc/crshairb.png",
                            GUI::Color{0x40, 0x40, 0x80});
    this->texture_mgr.load(GUI::GameTextures::PowerBar, path + "/img/Effects/blob.png"
,
                            GUI::Color{0x80, 0x80, 0xC0});
    this->texture_mgr.load(GUI::GameTextures::Fragment, path + "/img/Weapons/clustlet.
png",
                            GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::WormAirAttack, path + "/img/Worms/wa
irtlk.png",
                            GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::AirMissile, path + "/img/Weapons/airmi
sl.png",
                            GUI::Color{0xc0, 0xc0, 0x80});
    this->texture_mgr.load(GUI::GameTextures::WormDynamite, path + "/img/Worms/wdy
nbak.png",
                            GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::Dynamite, path + "/img/Weapons/dynamit
e.png",
                            GUI::Color{0x7f, 0x7f, 0xbb});
    this->texture_mgr.load(GUI::GameTextures::WormBaseballBat, path + "/img/Worms/
wbsbaim.png",
                            GUI::Color{0xc0, 0xc0, 0x80});
    this->texture_mgr.load(GUI::GameTextures::WormBaseballBatting, path + "/img/
Worms/wbsbswn.png",
                            GUI::Color{0xc0, 0xc0, 0x80});
    this->texture_mgr.load(GUI::GameTextures::WormTeleport, path + "/img/Worms/wtel
tlk.png",
                            GUI::Color{0xc0, 0xc0, 0x80});
    this->texture_mgr.load(GUI::GameTextures::WormTeleporting, path + "/img/Worms/
wteldsv.png",
                            GUI::Color{0xc0, 0xc0, 0x80});
    this->texture_mgr.load(GUI::GameTextures::BazookaIcon, path + "/img/Weapon Icon
s/bazooka.2.png",
                            GUI::Color{0x00, 0x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::GrenadeIcon, path + "/img/Weapon Icon
s/grenade.2.png",
                            GUI::Color{0x00, 0x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::ClusterIcon, path + "/img/Weapon Icon
s/cluster.2.png",
                            GUI::Color{0x00, 0x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::MortarIcon, path + "/img/Weapon Icons/
mortar.2.png",
                            GUI::Color{0x00, 0x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::BananaIcon, path + "/img/Weapon Icons/
banana.2.png",
                            GUI::Color{0x00, 0x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::HolyIcon, path + "/img/Weapon Icons/hgr
enade.2.png",
                            GUI::Color{0x00, 0x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::AirIcon, path + "/img/Weapon Icons/airstr
ke.1.png",
```



```
                            GUI::Color{0x00, 0x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::DynamiteIcon,
                            path + "/img/Weapon Icons/dynamite.1.png", GUI::Color{0x00, 0
x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::BaseballBatIcon,
                            path + "/img/Weapon Icons/baseball.1.png", GUI::Color{0x00, 0
x00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::TeleportIcon,
                            path + "/img/Weapon Icons/teleport.1.png", GUI::Color{0x00, 0x
00, 0x00});
    this->texture_mgr.load(GUI::GameTextures::Water,
                            path + "/img/background/water.png", GUI::Color{0x00, 0x00,
0x00});
}

void GUI::Game::loadSoundManager(){
    std::string path(ASSETS_PATH);
    this->sound_effect_mgr.load(GUI::GameSoundEffects::WalkCompress,
                                path + "/sound/Effects/Walk-Compress.wav");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::WormJump,
                                path + "/sound/Soundbanks/JUMP1.WAV");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::WormBackFlip,
                                path + "/sound/Soundbanks/JUMP2.WAV");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::WormLanding,
                                path + "/sound/Effects/WormLanding.wav");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::WormHit, path + "/sound/Sou
ndbanks/OUCH.WAV");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::WormDrowning,
                                path + "/sound/Effects/UnderWaterLoop.wav");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::WormDie,
                                path + "/sound/Soundbanks/BYEBYE.WAV");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::Splash, path + "/sound/Effect
s/Splash.wav");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::Explosion,
                                path + "/sound/Effects/Explosion1.wav");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::Holy,
                                path + "/sound/Effects/HOLYGRENADE.WAV");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::AirStrike,
                                path + "/sound/Effects/Airstrike.wav");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::Teleport,
                                path + "/sound/Effects/TELEPORT.WAV");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::Shot,
                                path + "/sound/Effects/ROCKETRELEASE.WAV");
    this->sound_effect_mgr.load(GUI::GameSoundEffects::Banana,
                                path + "/sound/Effects/BananaImpact.wav");
}

void GUI::Game::loadBackgroundManager(){
    std::string path(ASSETS_PATH);
    this->background_music_mgr.load(GUI::GameBackgroundMusic::Original,
                                    path + "/sound/Background/background.wav");
    this->background_music_mgr.load(GUI::GameBackgroundMusic::MurderTrain, path
+ "/sound/Background/MurderTrain.wav");
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#ifndef __GUIGame_H__
#define __GUIGame_H__

#include <atomic>
#include <list>
#include <thread>
#include <vector>

#include "Animation.h"
#include "Armory.h"
#include "Camera.h"
#include "ClientSocket.h"
#include "DoubleBuffer.h"
#include "Font.h"
#include "GameSoundEffects.h"
#include "GameStateMsg.h"
#include "GameTextures.h"
#include "Stage.h"
#include "Stream.h"
#include "TextureManager.h"
#include "Water.h"
#include "Weapons/Bullet.h"
#include "Weapons/Explosion.h"
#include "Wind.h"
#include "Window.h"
#include "Worm.h"
#include "BackgroundMusic.h"
#include "GameBackgroundMusic.h"
#include "BackgroundMusicPlayer.h"

namespace GUI {
using GameOutput = IO::Stream<IO::PlayerMsg>;
class Game {
   public:
    bool youWin{false};

    Game(Window &w, Worms::Stage &&stage, std::vector<std::string> &backgroundPa
ths, ClientSocket &socket,
              std::uint8_t team);
    ~Game();
    void start();
    void update(float dt);
    void render();

    void exit();

   private:
    void renderStatic();
    void renderBackground();
    void handleCamera(float dt);

    void inputWorker();
    void outputWorker();

    std::atomic<bool> quit{false};
    float scale{13.0f};           // pixels per meter
    float lastCameraUpdate{0.0f};  // pixels per meter
```



```
    Window &window;
    GameTextureManager texture_mgr;
    GameSoundEffectManager sound_effect_mgr;
    GameBackgroundMusicManager background_music_mgr;
    std::vector<Worm::Worm> worms;
    Worms::Stage stage;
    std::list<std::shared_ptr<Ammo::Bullet>> bullets;
    Camera cam;
    Font font;
    SDL_Color backgroundColor{0xba, 0x8d, 0xc6};
    std::vector<SDL_Color> teamColors;
    Armory armory;
    std::thread inputThread;
    std::thread outputThread;
    IO::DoubleBuffer<IO::GameStateMsg> snapshotBuffer;
    IO::GameStateMsg snapshot;
    GameOutput output;
    CommunicationSocket &socket;
    std::uint8_t team{0};
    uint8_t explodedQuantity{0};
    GUI::Wind wind;
    GUI::Water water;
    std::unique_ptr<Animation> currentPlayerArrow{nullptr};
    std::unique_ptr<GUI::BackgroundMusicPlayer> backGroundMusicPlayer{nullptr};

    void loadTextureManager();
    void loadBackgroundManager();
    void loadSoundManager();
};
}  // namespace GUI

#endif  //__GUIGame_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#include "Hit.h"

Worm::Hit::Hit() : State(StateID::Hit) {}

Worm::Hit::~Hit() {}

void Worm::Hit::update(float dt) {}

IO::PlayerInput Worm::Hit::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::mortar(Worm &w) {
```



```
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Hit::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#ifndef __Hit_H__
#define __Hit_H__

#include "WormState.h"

namespace Worm {
class Hit : public State {
   public:
    explicit Hit();
    virtual ~Hit();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;

    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;
    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__Hit_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#include <cmath>

#include "Holy.h"

Worm::Holy::Holy(const GUI::GameTextureManager &tex)
    : Weapon(tex, GUI::GameTextures::WormHoly, HOLY_CENTER_FRAME, WeaponID::WHol
y),
      scope(this->textureMgr),
      powerBar(this->textureMgr) {}

void Worm::Holy::update(float dt) {
    this->weaponAnimation.update(dt);
    this->scope.update(dt);
    this->powerBar.update(dt);
}

void Worm::Holy::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &fl
ip) {
    this->weaponAnimation.render(p, cam, flip);
    this->scope.render(p, cam, flip);
    this->powerBar.render(p, cam, flip);
}

void Worm::Holy::setAngle(float angle, Worm::Direction d) {
    this->weaponAnimation.setFrame((int)std::ceil(angle / ANGLE_STEP) + this->ce
nterFrame);
    this->scope.setAngle(angle, d);
    this->powerBar.setAngle(angle, d);
}

void Worm::Holy::startShot() {
    this->powerBar.startShot();
}

void Worm::Holy::endShot() {
    this->powerBar.endShot();
}

bool Worm::Holy::positionSelected() {
    return false;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#ifndef __HOLY_H__
#define __HOLY_H__

#include <vector>

#include "PowerBar.h"
#include "Scope.h"
#include "Weapon.h"

#define HOLY_CENTER_FRAME 15

namespace Worm {
class Holy : public Weapon {
   public:
    explicit Holy(const GUI::GameTextureManager &textureManager);
    ~Holy() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;

   private:
    ::Weapon::Scope scope;
    ::Weapon::PowerBar powerBar;
};
}  // namespace Worm

#endif  //__HOLY_H__
```

```
#include "JoinGameWindow.h"

const SDL_Color WHITE = {0xff, 0xff, 0xff};
const SDL_Color BLACK = {0, 0, 0};

const int TEXT_SIZE = 30;

GUI::JoinGameWindow::JoinGameWindow(Window &window, Font &font, Camera &cam,
                                    std::vector<IO::GameInfo> &info)
    : GameWindow(window, font, cam),
      info(info),
      gameName(font),
      numPlayers(font),
      prev("Previous", font),
      next("Next", font),
      join("Join", font) {
    int height = TEXT_SIZE * 3 / 2;
    this->prev.textColor = WHITE;
    this->prev.textSize = TEXT_SIZE;
    this->prev.position = {this->window.getWidth() / 4, this->window.getHeight()
 / 2};
    this->prev.height = height;
    this->prev.width = this->prev.msg.size() * 9 + 20;

    this->next.textColor = WHITE;
    this->next.textSize = TEXT_SIZE;
    this->next.position = {this->window.getWidth() * 3 / 4, this->window.getHeig
ht() / 2};
    this->next.height = height;
    this->next.width = this->next.msg.size() * 9 + 20;

    this->join.textColor = WHITE;
    this->join.textSize = TEXT_SIZE;
    this->join.position = {this->window.getWidth() / 2, this->window.getHeight()
 * 3 / 4};
    this->join.height = height;
    this->join.width = this->join.msg.size() * 9 + 20;
}

/**
 * @brief Called when the window is started.
 *
 */
void GUI::JoinGameWindow::start() {}

/**
 * @brief Renders the window.
 *
 */
void GUI::JoinGameWindow::render() {
    this->window.clear(SDL_Color{0xFF, 0xFF, 0xFF});

    const ScreenPosition center{this->window.getWidth() / 2, this->window.getHei
ght() / 2};

    this->prev.render(this->cam);
    this->next.render(this->cam);

    if (this->info.size() > 0) {
        const IO::GameInfo &info = this->info.at(this->currentGameIndex);

        this->gameName.set("Game #" + std::to_string(info.gameID), BLACK, TEXT_S
```



```
IZE * 2);
        this->gameName.renderFixed(center - ScreenPosition{0, this->window.getHe
ight() / 4},
                                   this->cam);

        std::string msg =
            std::to_string(info.numCurrentPlayers) + "/" + std::to_string(info.n
umTotalPlayers);
        this->numPlayers.set(msg, BLACK, TEXT_SIZE * 2);
        this->numPlayers.renderFixed(center, this->cam);

        if (info.numCurrentPlayers < info.numTotalPlayers) {
            this->join.render(this->cam);
        }
    }

    this->window.render();
}

/**
 * @brief Checks if a button was pressed.
 *
 * @param sp Position where there was a click.
 */
void GUI::JoinGameWindow::buttonPressed(ScreenPosition sp) {
    if (this->prev.inside(sp)) {
        if (this->currentGameIndex == 0) {
            this->currentGameIndex = static_cast<uint8_t>(this->info.size()) - 1
;
        } else {
            this->currentGameIndex--;
        }
    } else if (this->next.inside(sp)) {
        this->currentGameIndex = (this->currentGameIndex + 1) % this->info.size(
);
    } else if (this->join.inside(sp)) {
        const IO::GameInfo &info = this->info.at(this->currentGameIndex);
        if (info.numCurrentPlayers < info.numTotalPlayers) {
            this->notify(*this, Event::LobbyToJoinSelected);
        }
    }
}

/**
 * @brief Handles key press events.
 *
 * @param key Key pressed.
 */
void GUI::JoinGameWindow::handleKeyDown(SDL_Keycode key) {}

void GUI::JoinGameWindow::appendCharacter(char text[32]) {}
```

```
#ifndef JOIN_GAME_WINDOW_H_
#define JOIN_GAME_WINDOW_H_

#include <vector>
#include "../Button.h"
#include "../GameWindow.h"
#include "GameStateMsg.h"
#include "Text.h"
#include "Texture.h"

namespace GUI {
class JoinGameWindow : public GameWindow {
   public:
    JoinGameWindow(Window &window, Font &font, Camera &cam, std::vector<IO::Game
Info> &info);
    std::vector<IO::GameInfo> &info;
    uint8_t currentGameIndex{0};

    void start() override;
    void render() override;
    void handleKeyDown(SDL_Keycode key) override;
    void appendCharacter(char text[32]) override;
    void buttonPressed(ScreenPosition sp) override;

   private:
    Text gameName;
    Text numPlayers;
    Button prev;
    Button next;
    Button join;
};
}  // namespace GUI

#endif
```

```
//
// Created by rodrigo on 3/06/18.
//

#include "Land.h"

Worm::Land::Land() : State(StateID::Land) {}

Worm::Land::~Land() {}

void Worm::Land::update(float dt) {}

IO::PlayerInput Worm::Land::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::mortar(Worm &w) {
    return IO::PlayerInput::moveNone;
```



```
}

IO::PlayerInput Worm::Land::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Land::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
//
// Created by rodrigo on 3/06/18.
//

#ifndef INC_4_WORMS_LAND_H
#define INC_4_WORMS_LAND_H

#include "GameStateMsg.h"
#include "WormState.h"

namespace Worm {
class Land : public State {
   public:
    Land();
    ~Land();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;
    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;

    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  // INC_4_WORMS_LAND_H
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 17/06/18
 */

#include <iostream>
#include <GameStateMsg.h>
#include <SDL2/SDL.h>
#include <zconf.h>

#include "GameWindow.h"
#include "LobbyAssistant.h"
#include "Text.h"
#include "Window.h"
#include "SelectActionWindow.h"
#include "CreateGameWindow.h"
#include "WaitingPlayersWindow.h"
#include "Lobby/JoinGameWindow.h"
#include "ConnectionWindow.h"

GUI::LobbyAssistant::LobbyAssistant(Window &window) :
        window(window),
        font(std::string(ASSETS_PATH) + "/fonts/gruen_lemonograf.ttf", 28),
        cam(window, this->scale, 600, 600) {
    this->gameWindow = std::shared_ptr<GameWindow>(new ConnectionWindow{this->wi
ndow, this->font, this->cam});
    this->gameWindow->addObserver(this);
}

void GUI::LobbyAssistant::run() {
    while (!this->quit) {
        SDL_Event e;
        while (SDL_PollEvent(&e) != 0) {
            switch (e.type) {
                case SDL_QUIT: {
                    this->quit = true;
                    this->exit = true;
                    break;
                }
                case SDL_KEYDOWN: {
                    this->gameWindow->handleKeyDown(e.key.keysym.sym);
                    break;
                }
                case SDL_KEYUP: {
                    break;
                }
                case SDL_TEXTINPUT: {
                    if(!((e.text.text[0] == 'c' || e.text.text[0] == 'C') && (e.
text.text[0] == 'v' || e.text.text[0] == 'V') && SDL_GetModState() & KMOD_CTRL)
) {
                        //Append character
                        this->gameWindow->appendCharacter(e.text.text);
                    }
                    break;
                }
                case SDL_MOUSEBUTTONDOWN: {
                    int x, y;
                    SDL_GetMouseState(&x, &y);
                    GUI::Position global = this->cam.screenToGlobal(GUI::ScreenP
osition{x, y});
                    this->gameWindow->buttonPressed(ScreenPosition{x, y});
                }
```



```
            }
        }

        IO::ServerResponse sr{};
        if (this->serverStream.pop(sr, false)) {
            this->handleServerResponse(sr);
        }

        if (this->nextGameWindow) {
            this->gameWindow = this->nextGameWindow;
            this->nextGameWindow = nullptr;
        }

        if (this->gameWindow != nullptr) {
            this->gameWindow->render();
        }

        usleep(50 * 1000);
    }
}



void GUI::LobbyAssistant::onNotify(Subject &subject, Event event) {
    switch (event) {
        case Event::ConnectionToServer: {
            auto connectionWindow = dynamic_cast<ConnectionWindow *>(this->gameW
indow.get());
            ConnectionInfo info = connectionWindow->getConnectionInfo();
            ClientSocket socket(info.ip, info.port);
            this->communicationProtocol = std::shared_ptr<IO::CommunicationProto
col>(
                    new IO::CommunicationProtocol(socket, &this->output, &this->
serverStream));
            this->communicationProtocol->start();

            this->nextGameWindow = std::shared_ptr<GameWindow>(new SelectActionW
indow{this->window, this->font, this->cam});
            this->nextGameWindow->addObserver(this);
            break;
        }
        case Event::CreateGame: {
            this->output << IO::ClientGUIMsg{IO::ClientGUIInput::startCreateGame
};
            break;
        }
        case Event::LevelSelected: {
            auto createGameWindow = dynamic_cast<CreateGameWindow *>(this->gameW
indow.get());
            this->communicationProtocol->levelToCreate = createGameWindow->butto
nSelected;
            this->output << IO::ClientGUIMsg{IO::ClientGUIInput::levelSelected};
            this->nextGameWindow = std::shared_ptr<GameWindow>(new WaitingPlayer
sWindow{this->window,

    this->font,

    this->cam,

    createGameWindow->levelsInfo[createGameWindow->buttonSelected].playersQuanti
ty});
            this->nextGameWindow->addObserver(this);
```

```
            break;
        }
        case Event::JoinGame: {
            this->output << IO::ClientGUIMsg{IO::ClientGUIInput::startJoinGame};
            break;
        }
        case Event::LobbyToJoinSelected: {
            auto joinGameWindow = dynamic_cast<JoinGameWindow *>(this->gameWindo
w.get());
            this->communicationProtocol->gameToJoin = joinGameWindow->currentGam
eIndex;
            this->communicationProtocol->levelOfGameToJoin = joinGameWindow->inf
o[joinGameWindow->currentGameIndex].levelID;
            this->output << IO::ClientGUIMsg{IO::ClientGUIInput::joinGame};
            this->nextGameWindow = std::shared_ptr<GameWindow>(new WaitingPlayer
sWindow{this->window,

        this->font,

        this->cam,

        joinGameWindow->info[joinGameWindow->currentGameIndex].numTotalPlayers,

        joinGameWindow->info[joinGameWindow->currentGameIndex].numCurrentPlayers
});
            this->nextGameWindow->addObserver(this);
            break;
        }
        default: {
            break;
        }
    }
}

ClientSocket GUI::LobbyAssistant::getSocket() {
    return std::move(this->communicationProtocol->getSocket());
}

void GUI::LobbyAssistant::handleServerResponse(IO::ServerResponse &response) {
    switch (response.action) {
        case IO::ServerResponseAction::startGame: {
            this->levelPath = std::move(this->communicationProtocol->levelPath);
            this->backgroundPath = std::move(this->communicationProtocol->backgr
oundPath);
            this->output << IO::ClientGUIMsg{IO::ClientGUIInput::quit};
            this->quit = true;
            break;
        }
        case IO::ServerResponseAction::levelsInfo: {
            this->nextGameWindow = std::shared_ptr<GameWindow>(new CreateGameWin
dow{this->window,

this->font,

this->cam ,

this->communicationProtocol->levelsInfo});
            this->nextGameWindow->addObserver(this);
            break;
        }
        case IO::ServerResponseAction::gamesInfo: {
            this->nextGameWindow = std::shared_ptr<GameWindow>(new JoinGameWindo
```



```
w{this->window,

  this->font,

  this->cam ,

  this->communicationProtocol->gamesInfo});
            this->nextGameWindow->addObserver(this);
            break;
        }
        case IO::ServerResponseAction::playerConnected: {
            dynamic_cast<WaitingPlayersWindow *>(this->gameWindow.get())->player
sConnected++;
            break;
        }
        case IO::ServerResponseAction::serverClosed: {
            this->quit = true;
            this->exit = true;
            this->gameWindow = nullptr;
            break;
        }
        default: {
            break;
        }
    }
}

GUI::Font & GUI::LobbyAssistant::getFont() {
    return this->font;
}

GUI::Camera & GUI::LobbyAssistant::getCam() {
    return this->cam;
}

GUI::LobbyAssistant::~LobbyAssistant() {
    this->output.close();
    this->serverStream.close();
    if (this->communicationProtocol != nullptr) {
        this->communicationProtocol->stop();
        this->communicationProtocol->join();
    }
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 17/06/18
 */

#ifndef __LOBBY_ASSISTANT_H__
#define __LOBBY_ASSISTANT_H__

#include <Protocol.h>
#include <memory>
#include <Stream.h>
#include <Font.h>
#include <Camera.h>
#include "ClientSocket.h"
#include "CommunicationProtocol.h"
#include "Observer.h"
#include "Thread.h"
#include "GameWindow.h"
#include "GameStateMsg.h"

namespace GUI { // HabÃ-a una forward declaration con GameWindow pero no hace fa
lta parece.
    class LobbyAssistant : public Observer {
    public:
        std::string levelPath;
        std::vector<std::string> backgroundPath;
        bool exit{false};

        explicit LobbyAssistant(Window &window);
        ~LobbyAssistant();
        //TODO overrrite
        void run();
        void onNotify(Subject &subject, Event event) override;

        ClientSocket getSocket();

        Font & getFont();

        Camera & getCam();

    private:
        Window &window;
        float scale{13.0f};
        bool quit{false};
        std::shared_ptr<GameWindow> gameWindow{nullptr};
        std::shared_ptr<GameWindow> nextGameWindow{nullptr};
        Font font;
        Camera cam;
        std::shared_ptr<IO::CommunicationProtocol> communicationProtocol;
        IO::Stream<IO::ClientGUIMsg> output;
        IO::Stream<IO::ServerResponse> serverStream;

        void handleServerResponse(IO::ServerResponse &response);
    };
} //namespace Worm


#endif //__LOBBY_ASSISTANT_H__
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */
#include <cstdlib>
#include <iostream>
#include <string>

#include "ClientSocket.h"
#include "GUIGame.h"
#include "LobbyAssistant.h"
#include "GameEndWindow.h"

int main(int argc, const char *argv[]) {
    if (argc != 1) {
        std::cout << "Usage: ./client" << std::endl;
        return EXIT_FAILURE;
    }

    try {
        GUI::Window window{};
        window.clear();
        GUI::LobbyAssistant lobby(window);
        lobby.run();

        if (!lobby.exit) {
            ClientSocket socket = std::move(lobby.getSocket());

            char buffer[1];
            socket.receive(buffer, sizeof(buffer));

            GUI::Game game{window, Worms::Stage::fromFile(lobby.levelPath), lobb
y.backgroundPath, socket,
                           (std::uint8_t) buffer[0]};
            game.start();

            GUI::GameEndWindow gameEndWindow(window, lobby.getFont(), lobby.getC
am(), game.youWin);
            gameEndWindow.start();
        }
    } catch (std::exception &e) {
        std::cerr << "In main()" << std::endl;
        std::cerr << e.what() << std::endl;
        return 1;
    } catch (...) {
        std::cerr << "Unkown error in main thread" << std::endl;
        return 1;
    }
    return 0;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#include <cmath>

#include "Mortar.h"

Worm::Mortar::Mortar(const GUI::GameTextureManager &tex)
    : Weapon(tex, GUI::GameTextures::Bazooka2, MORTAR_CENTER_FRAME, WeaponID::WM
ortar),
      scope(this->textureMgr),
      powerBar(this->textureMgr) {}

void Worm::Mortar::update(float dt) {
    this->weaponAnimation.update(dt);
    this->scope.update(dt);
    this->powerBar.update(dt);
}

void Worm::Mortar::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &
flip) {
    this->weaponAnimation.render(p, cam, flip);
    this->scope.render(p, cam, flip);
    this->powerBar.render(p, cam, flip);
}

void Worm::Mortar::setAngle(float angle, Worm::Direction d) {
    this->weaponAnimation.setFrame((int)std::ceil(angle / ANGLE_STEP) + this->ce
nterFrame);
    this->scope.setAngle(angle, d);
    this->powerBar.setAngle(angle, d);
}

void Worm::Mortar::startShot() {
    this->powerBar.startShot();
}

void Worm::Mortar::endShot() {
    this->powerBar.endShot();
}

bool Worm::Mortar::positionSelected() {
    return false;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#ifndef __MORTAR_H__
#define __MORTAR_H__

#include <vector>

#include "PowerBar.h"
#include "Scope.h"
#include "Weapon.h"

#define MORTAR_CENTER_FRAME 16

namespace Worm {
class Mortar : public Weapon {
   public:
    explicit Mortar(const GUI::GameTextureManager &textureManager);
    ~Mortar() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;

   private:
    ::Weapon::Scope scope;
    ::Weapon::PowerBar powerBar;
};
}  // namespace Worm

#endif  //__MORTAR_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#include "PowerBar.h"
#include "Weapon.h"

Weapon::PowerBar::PowerBar(const GUI::GameTextureManager &tex) : textureManager(
tex) {
    this->animations.reserve(POWER_FRAMES_QUANTITY);
}

void Weapon::PowerBar::setAngle(float angle, Worm::Direction d) {
    this->angle = d == Worm::Direction::right ? angle : 180 - angle;
}

void Weapon::PowerBar::update(float dt) {
    if (this->shotStarted) {
        this->elapsedTime += dt;
        if (this->power < POWER_FRAMES_QUANTITY && this->elapsedTime < POWER_CHA
RGE_TIME) {
            this->animations.emplace_back(this->textureManager.get(GUI::GameText
ures::PowerBar),
                                          false, this->power, false);
            this->power++;
        }
    }
}

void Weapon::PowerBar::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFl
ip &flip) {
    for (int i = 0; i < this->power; i++) {
        GUI::Position powerPos =
            GUI::Position((SCOPE_DISTANCE * (log10(10 * i / 17))) * cos(this->an
gle * PI / 180),
                          (SCOPE_DISTANCE * (log10(10 * i / 17))) * sin(this->an
gle * PI / 180)) +
            p;
        this->animations[i].render(powerPos, cam, flip);
    }
}

void Weapon::PowerBar::startShot() {
    this->shotStarted = true;
}

void Weapon::PowerBar::endShot() {
    this->shotStarted = false;
    this->animations.erase(this->animations.begin(), this->animations.end());
    this->power = 0;
    this->elapsedTime = 0.0f;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#ifndef __PowerBar_H__
#define __PowerBar_H__

#include <Animation.h>
#include <Camera.h>
#include <vector>

#include "../GameTextures.h"
#include "Direction.h"

#define POWER_FRAMES_QUANTITY 16

namespace Weapon {
class PowerBar {
   public:
    explicit PowerBar(const GUI::GameTextureManager &tex);
    ~PowerBar() = default;
    void setAngle(float angle, Worm::Direction d);
    void update(float dt);
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip);
    void startShot();
    void endShot();

   private:
    bool shotStarted{false};
    float angle{0.0f};
    float elapsedTime{0.0f};
    uint16_t power{0};
    std::vector<GUI::Animation> animations;
    const GUI::GameTextureManager &textureManager;
};
}

#endif  //__PowerBar_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#include "Scope.h"
#include "Direction.h"
#include "Weapon.h"

Weapon::Scope::Scope(const GUI::GameTextureManager &tex)
    : animation(tex.get(GUI::GameTextures::Scope), false, 0, false) {}

void Weapon::Scope::setAngle(float angle, Worm::Direction d) {
    this->angle = d == Worm::Direction::right ? angle : 180 - angle;
}

void Weapon::Scope::update(float dt) {
    this->animation.update(dt);
}

void Weapon::Scope::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip
&flip) {
    GUI::Position scopePos = GUI::Position(SCOPE_DISTANCE * cos(this->angle * PI
 / 180),
                                           SCOPE_DISTANCE * sin(this->angle * PI
 / 180)) +
                             p;
    this->animation.render(scopePos, cam, flip);
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#ifndef __Scope_H__
#define __Scope_H__

#include <Animation.h>
#include <Camera.h>
#include "../GameTextures.h"
#include "Direction.h"

namespace Weapon {
class Scope {
   public:
    Scope(const GUI::GameTextureManager &tex);
    ~Scope() = default;
    void setAngle(float angle, Worm::Direction d);
    void update(float dt);
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip);

   private:
    float angle{0.0f};
    GUI::Animation animation;
};
}

#endif  //__Scope_H__
```

```
//
// Created by rodrigo on 19/06/18.
//

#include <SDL2/SDL.h>
#include <iostream>

#include "SelectActionWindow.h"
#include "Text.h"
#include "Window.h"

#define MSG_CREATE_GAME "Create game"
#define MSG_JOIN_GAME "Join game"

GUI::SelectActionWindow::SelectActionWindow(Window &window, Font &font, Camera &
cam) :
        GameWindow(window, font, cam) {
    std::string msg(MSG_CREATE_GAME);
    this->buttons.emplace_back(ScreenPosition{this->window.getWidth() / 4, this-
>window.getHeight() / 2},
                               50, 300, msg, this->font);
    msg = MSG_JOIN_GAME;
    this->buttons.emplace_back(ScreenPosition{this->window.getWidth() * 3 / 4, t
his->window.getHeight() / 2},
                               50, 300, msg, this->font);
}

void GUI::SelectActionWindow::start() {

}

void GUI::SelectActionWindow::render() {
    this->window.clear(SDL_Color{0xFF, 0xFF, 0xFF});
    for (auto &button : this->buttons) {
        button.render(this->cam);
    }

    this->window.render();
}

void GUI::SelectActionWindow::buttonPressed(GUI::ScreenPosition sp) {
    if (this->buttons[0].inside(sp)) {
        this->notify(*this, Event::CreateGame);
    }

    if (this->buttons[1].inside(sp)) {
        this->notify(*this, Event::JoinGame);
    }
}

void GUI::SelectActionWindow::appendCharacter(char *text) {

}

void GUI::SelectActionWindow::handleKeyDown(SDL_Keycode key) {

}
```

```
//
// Created by rodrigo on 19/06/18.
//

#ifndef INC_4_WORMS_SELECTACTIONWINDOW_H
#define INC_4_WORMS_SELECTACTIONWINDOW_H


#include <vector>

#include "Window.h"
#include "Font.h"
#include "GameWindow.h"
#include "Button.h"

namespace GUI {
    class SelectActionWindow : public GameWindow {
    public:
        explicit SelectActionWindow(Window &window, Font &font, Camera &cam);

        void start() override;
        void render() override;
        void handleKeyDown(SDL_Keycode key) override;
        void appendCharacter(char text[32]) override;
        void buttonPressed(ScreenPosition sp) override;

    private:
        std::vector<Button> buttons;
    };
}


#endif //INC_4_WORMS_SELECTACTIONWINDOW_H
```

```
#include "Sliding.h"

Worm::Sliding::Sliding() : State(StateID::Sliding) {}

Worm::Sliding::~Sliding() {}

void Worm::Sliding::update(float dt) {}

IO::PlayerInput Worm::Sliding::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::mortar(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
```



```
}

IO::PlayerInput Worm::Sliding::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Sliding::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
#ifndef PLAYER_SLIDING_H_
#define PLAYER_SLIDING_H_

#include "WormState.h"

namespace Worm {
class Sliding : public State {
   public:
    Sliding();
    ~Sliding();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;

    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;
};
}  // namespace Worm

#endif  // INC_4_WORMS_FALLING_H
```

```cpp
//
// Created by rodrigo on 4/06/18.
//

#include "SoundEffect.h"
#include "Exception.h"

GUI::SoundEffect::SoundEffect(const std::string &filename) {
    this->soundEffect = Mix_LoadWAV(filename.c_str());
    if (!this->soundEffect) {
        throw Exception{"Error loading %s: %s", filename.c_str(), Mix_GetError()};
    }
}

GUI::SoundEffect::~SoundEffect() {
    if (this->soundEffect != nullptr) {
        Mix_FreeChunk(this->soundEffect);
    }
}

Mix_Chunk *GUI::SoundEffect::getChunk() const {
    return this->soundEffect;
}

GUI::SoundEffect::SoundEffect(GUI::SoundEffect &&other) {
    std::swap(this->soundEffect, other.soundEffect);
}

void GUI::SoundEffect::play(bool loop) const {
    Mix_PlayChannel(-1, this->soundEffect, -1 * loop);
}
```

```
//
// Created by rodrigo on 4/06/18.
//

#ifndef INC_4_WORMS_SOUNDEFFECT_H
#define INC_4_WORMS_SOUNDEFFECT_H

#include <SDL2/SDL.h>
#include <SDL2/SDL_mixer.h>
#include <string>

namespace GUI {
class SoundEffect {
   public:
    SoundEffect(const std::string &filename);
    SoundEffect(SoundEffect &&other);
    ~SoundEffect();
    Mix_Chunk *getChunk() const;
    void play(bool loop) const;

   private:
    Mix_Chunk *soundEffect{nullptr};
};
}

#endif  // INC_4_WORMS_SOUNDEFFECT_H
```

```
//
// Created by rodrigo on 4/06/18.
//

#ifndef INC_4_WORMS_SOUNDEFFECTMANAGER_H
#define INC_4_WORMS_SOUNDEFFECTMANAGER_H

#include <SDL2/SDL.h>
#include <functional>
#include <string>
#include <unordered_map>
#include "SoundEffect.h"

namespace GUI {
template <typename ID, typename HASH = std::hash<ID>>
class SoundEffectManager {
   public:
    SoundEffectManager();
    ~SoundEffectManager();
    SoundEffectManager& operator=(SoundEffectManager& other) = delete;

    void load(ID id, const std::string& file_name);
    const SoundEffect& get(ID id) const;

   private:
    std::unordered_map<ID, SoundEffect, HASH> cache;
};
}  // namespace GUI

template <typename ID, typename HASH>
GUI::SoundEffectManager<ID, HASH>::SoundEffectManager() {}

template <typename ID, typename HASH>
GUI::SoundEffectManager<ID, HASH>::~SoundEffectManager() {}

/**

 * @brief Loads a sound effect.
 *
 * @param file_name The image file name.
 */
template <typename ID, typename HASH>
void GUI::SoundEffectManager<ID, HASH>::load(ID id, const std::string& file_name
) {
    GUI::SoundEffect soundEffect{file_name};
    this->cache.insert(std::make_pair(id, std::move(soundEffect)));
}

/**
 * @brief Gets a sound effect.
 *
 * @param file_name Name of the sound effect.
 */
template <typename ID, typename HASH>
const GUI::SoundEffect& GUI::SoundEffectManager<ID, HASH>::get(ID id) const {
    return this->cache.at(id);
}

#endif  // INC_4_WORMS_SOUNDEFFECTMANAGER_H
```

```
//
// Created by rodrigo on 5/06/18.
//

#include "SoundEffectPlayer.h"

GUI::SoundEffectPlayer::SoundEffectPlayer(const GUI::SoundEffect &soundEffect)
    : soundEffect(&soundEffect) {}

GUI::SoundEffectPlayer::SoundEffectPlayer(const GUI::SoundEffect &soundEffect, f
loat duration)
    : soundEffect(&soundEffect), duration(duration) {
    //    this->soundEffect->play();
}

GUI::SoundEffectPlayer::SoundEffectPlayer(const GUI::SoundEffect &soundEffect, b
ool autoUpdate)
    : soundEffect(&soundEffect), autoUpdate(autoUpdate) {}

GUI::SoundEffectPlayer::~SoundEffectPlayer() {}

void GUI::SoundEffectPlayer::update(float dt) {
    if (!this->autoUpdate) {
        this->timeElapsed += dt;
        if (this->timeElapsed > this->duration) {
            this->play();
            this->timeElapsed = 0.0f;
        }
    }
}

void GUI::SoundEffectPlayer::play() {
    this->soundEffect->play(this->loop);
}
```

```
//
// Created by rodrigo on 5/06/18.
//

#ifndef INC_4_WORMS_SOUNDEFFECTPLAYER_H
#define INC_4_WORMS_SOUNDEFFECTPLAYER_H

#include <SDL2/SDL.h>

#include "SoundEffect.h"

namespace GUI {
class SoundEffectPlayer {
   public:
    bool loop{false};

    explicit SoundEffectPlayer(const GUI::SoundEffect &soundEffect);
    SoundEffectPlayer(const SoundEffect &soundEffect, float duration);
    SoundEffectPlayer(const GUI::SoundEffect &soundEffect, bool autoUpdate);
    ~SoundEffectPlayer();
    void update(float dt);
    void play();

   private:
    const SoundEffect *soundEffect;
    float duration{0.0f};
    float timeElapsed{0.0f};
    bool autoUpdate{false};
};
}

#endif  // INC_4_WORMS_SOUNDEFFECTPLAYER_H
```

```
//
// Created by rodrigo on 16/06/18.
//

#include "Teleport.h"

Worm::Teleport::Teleport(const GUI::GameTextureManager &tex)
    : Weapon(tex, GUI::GameTextures::WormTeleport, TELEPORT_CENTER_FRAME, Weapon
ID::WTeleport) {
    this->weaponAnimation.setAnimateOnce();
}

void Worm::Teleport::update(float dt) {
    if (!this->weaponAnimation.finished()) {
        this->weaponAnimation.update(dt);
    } else {
        this->endAnimation();
    }
}

void Worm::Teleport::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip
 &flip) {
    this->weaponAnimation.render(p, cam, flip);
}

void Worm::Teleport::setAngle(float angle, Worm::Direction d) {}

void Worm::Teleport::startShot() {}

void Worm::Teleport::endShot() {}

bool Worm::Teleport::positionSelected() {
    this->weaponAnimation.setAutoUpdate(true);
    return true;
}

void Worm::Teleport::endAnimation() {
    this->weaponAnimation.setFrame(TELEPORT_CENTER_FRAME);
    this->weaponAnimation.setAutoUpdate(false);
}
```

```
//
// Created by rodrigo on 16/06/18.
//

#include "Teleported.h"

Worm::Teleported::Teleported() : State(StateID::Teleported) {}

Worm::Teleported::~Teleported() {}

void Worm::Teleported::update(float dt) {}

IO::PlayerInput Worm::Teleported::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::mortar(Worm &w) {
    return IO::PlayerInput::moveNone;
```



```
}

IO::PlayerInput Worm::Teleported::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleported::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
//
// Created by rodrigo on 16/06/18.
//

#ifndef INC_4_WORMS_TELEPORTED_H
#define INC_4_WORMS_TELEPORTED_H

#include "WormState.h"

namespace Worm {
class Teleported : public State {
   public:
    Teleported();
    ~Teleported();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  // INC_4_WORMS_TELEPORTED_H
```

```
//
// Created by rodrigo on 16/06/18.
//

#ifndef INC_4_WORMS_TELEPORT_H
#define INC_4_WORMS_TELEPORT_H

#define TELEPORT_CENTER_FRAME 0

#include "Weapon.h"

namespace Worm {
class Teleport : public Weapon {
   public:
    explicit Teleport(const GUI::GameTextureManager &textureManager);
    ~Teleport() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;

   private:
    void endAnimation();
};
}  // namespace Worm

#endif  // INC_4_WORMS_TELEPORT_H
```

```
//
// Created by rodrigo on 16/06/18.
//

#include "Teleporting.h"

Worm::Teleporting::Teleporting() : State(StateID::Teleporting) {}

Worm::Teleporting::~Teleporting() {}

void Worm::Teleporting::update(float dt) {}

IO::PlayerInput Worm::Teleporting::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::mortar(Worm &w) {
    return IO::PlayerInput::moveNone;
```



```
}

IO::PlayerInput Worm::Teleporting::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Teleporting::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
//
// Created by rodrigo on 16/06/18.
//

#ifndef INC_4_WORMS_TELEPORTING_H
#define INC_4_WORMS_TELEPORTING_H

#include "WormState.h"

namespace Worm {
class Teleporting : public State {
   public:
    Teleporting();
    ~Teleporting();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  // INC_4_WORMS_TELEPORTING_H
```

```
//
// Created by rodrigo on 24/06/18.
//

#include "WaitingPlayersWindow.h"

GUI::WaitingPlayersWindow::WaitingPlayersWindow(GUI::Window &window, GUI::Font &
font, GUI::Camera &cam,
                                                uint8_t playersQuantity) :
        GameWindow(window, font, cam),
        playersQuantity(playersQuantity) {
}

GUI::WaitingPlayersWindow::WaitingPlayersWindow(GUI::Window &window, GUI::Font &
font, GUI::Camera &cam,
                                                uint8_t playersQuantity, uint8_t
 playersConnected) :
        WaitingPlayersWindow(window, font, cam, playersQuantity) {
    this->playersConnected = playersConnected;
}

void GUI::WaitingPlayersWindow::start() {

}

void GUI::WaitingPlayersWindow::render() {
    this->window.clear(SDL_Color{0xFF, 0xFF, 0xFF});

    Text playersConnected{this->font};
    int x = this->window.getWidth() * 2 / 5;
    int y = this->window.getHeight() / 2;
    playersConnected.set("Players connected", SDL_Color{0, 0, 0}, 50);
    playersConnected.renderFixed(ScreenPosition{x, y}, this->cam);
    x = this->window.getWidth() * 3 / 5;
    y = this->window.getHeight() / 2;
    playersConnected.setBackground(SDL_Color{0, 0, 0});
    playersConnected.set(std::to_string(this->playersConnected) + "/" + std::to_
string(this->playersQuantity), SDL_Color{0xFF, 0xFF, 0xFF}, 50);
    playersConnected.renderFixed(ScreenPosition{x, y}, this->cam);

    this->window.render();
}

void GUI::WaitingPlayersWindow::buttonPressed(GUI::ScreenPosition sp) {

}

void GUI::WaitingPlayersWindow::appendCharacter(char *text) {

}

void GUI::WaitingPlayersWindow::handleKeyDown(SDL_Keycode key) {

}
```

```
//
// Created by rodrigo on 24/06/18.
//

#ifndef INC_4_WORMS_WAITINGPLAYERSWINDOW_H
#define INC_4_WORMS_WAITINGPLAYERSWINDOW_H


#include <vector>

#include "Window.h"
#include "Font.h"
#include "GameStateMsg.h"
#include "GameWindow.h"
#include "Button.h"

namespace GUI {
    class WaitingPlayersWindow : public GameWindow {
    public:
        uint8_t playersConnected{0};

        WaitingPlayersWindow(GUI::Window &window, GUI::Font &font, GUI::Camera &
cam, uint8_t playersQuantity);
        WaitingPlayersWindow(Window &window, Font &font, Camera &cam, uint8_t pl
ayersQuantity, uint8_t playersConnected);

        void start() override;
        void render() override;
        void handleKeyDown(SDL_Keycode key) override;
        void appendCharacter(char text[32]) override;
        void buttonPressed(ScreenPosition sp) override;

    private:
        std::vector<Button> buttons;
        unsigned int playersQuantity{0};
    };
}


#endif //INC_4_WORMS_WAITINGPLAYERSWINDOW_H
```

```cpp
#include "Water.h"
#include <cmath>
#include "WrapTexture.h"

GUI::Water::Water(const GUI::GameTextureManager &tm) : textureManager(tm) {}

/**
 * @brief Updates the water animation state
 *
 * @param dt Time elapsed since the last call to this function.
 */
void GUI::Water::update(float dt) {
    this->elapsed += dt;
    this->yDelta = std::sin(this->elapsed) * 1;
}

/**
 * @brief Renders the water.
 *
 * @param camera Camera where the water is rendered.
 */
void GUI::Water::render(GUI::Camera &camera) {
    const GUI::Texture &texture = this->textureManager.get(GUI::GameTextures::Wa
ter);
    GUI::WrapTexture water{texture, camera.screenWidth(), texture.getHeight() /
camera.getScale()};
    water.render(Position{camera.getPosition().x, -6.5f + this->yDelta}, camera)
;
}
```

```
#ifndef WATER_H_
#define WATER_H_

#include "Camera.h"
#include "GameTextures.h"

namespace GUI {
class Water {
   public:
    Water(const GameTextureManager &tm);
    ~Water() = default;

    void update(float dt);
    void render(Camera &camera);

   private:
    const GUI::GameTextureManager &textureManager;
    float elapsed{0};
    float yDelta{0};
};
}  // namespace GUI

#endif
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 27/05/18
 */

#include <iostream>

#include "GameStateMsg.h"
#include "Weapon.h"

Worm::Weapon::Weapon(const GUI::GameTextureManager &texMgr, GUI::GameTextures te
x,
                     uint16_t centerFrame, WeaponID id)
    : textureMgr(texMgr),
      current(id),
      centerFrame(centerFrame),
      weaponAnimation(texMgr.get(tex), false, centerFrame, false) {}

const Worm::WeaponID &Worm::Weapon::getWeaponID() const {
    return this->current;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 27/05/18
 */

#ifndef __Weapon_H__
#define __Weapon_H__

#include "../GameTextures.h"
#include "Animation.h"
#include "Camera.h"
#include "Direction.h"
#include "GameStateMsg.h"
#include "TextureManager.h"

#define ANGLE_STEP 5.625f
#define SCOPE_DISTANCE 4

namespace Worm {
class Weapon {
   public:
    explicit Weapon(const GUI::GameTextureManager &texMgr, GUI::GameTextures tex
,
                    uint16_t centerFrame, WeaponID id);
    virtual ~Weapon() = default;
    /**
     * updates all its animations.
     * @param dt
     */
    virtual void update(float dt) = 0;
    /**
     * renders all its animations.
     * @param p
     * @param cam
     * @param flip
     */
    virtual void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &fl
ip) = 0;
    const WeaponID &getWeaponID() const;
    /**
     * updates animations' frame depending on the angle.
     * @param angle
     */
    virtual void setAngle(float angle, Direction d) = 0;
    /**
     * Starts the PowerBar's rendering, adding animations in its container
     */
    virtual void startShot() = 0;
    /**
     * End PowerBar's rendering, freeing its container
     */
    virtual void endShot() = 0;
    /**
     * When using remoteControl weapons, starts the animation of the worm
     * and return
     */
    virtual bool positionSelected() = 0;

   protected:
    const GUI::GameTextureManager &textureMgr;
    WeaponID current;
    uint16_t centerFrame;
```



```
    GUI::Animation weaponAnimation;
    float angle{0.0f};
};
}  // namespace Weapon

#endif  //__Weapon_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#include "WeaponNone.h"

Worm::WeaponNone::WeaponNone(const GUI::GameTextureManager &textureManager)
    : Weapon(textureManager, GUI::GameTextures::WormIdle, 0, WeaponID::WNone) {}

void Worm::WeaponNone::update(float dt) {}

void Worm::WeaponNone::render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFl
ip &flip) {}

void Worm::WeaponNone::setAngle(float angle, Worm::Direction d) {}

void Worm::WeaponNone::startShot() {}

void Worm::WeaponNone::endShot() {}

bool Worm::WeaponNone::positionSelected() {
    return false;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#ifndef __WEAPON_NONE_H__
#define __WEAPON_NONE_H__

#include <vector>

#include "Weapon.h"

namespace Worm {
class WeaponNone : public Weapon {
   public:
    explicit WeaponNone(const GUI::GameTextureManager &textureManager);
    ~WeaponNone() = default;
    void update(float dt) override;
    void render(GUI::Position &p, GUI::Camera &cam, SDL_RendererFlip &flip) over
ride;
    void setAngle(float angle, Direction d) override;
    void startShot() override;
    void endShot() override;
    bool positionSelected() override;
};
}  // namespace Worm

#endif  //__WEAPON_NONE_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 20/06/18
 */

#include "Wind.h"
#include <WrapTexture.h>
#include "Texture.h"

GUI::Wind::Wind(const GUI::GameTextureManager &tex, GUI::Camera &cam) : tex(tex)
, cam(cam) {}

void GUI::Wind::render(std::int8_t intensity, int windowWidth) {
    const GUI::Texture &toUse = (intensity > 0) ? this->tex.get(GameTextures::Wi
ndRight)
                                                : this->tex.get(GameTextures::Wi
ndLeft);
    float scaledIntensity = (float)std::abs(intensity) / 127 * this->cam.getScal
e();
    GUI::WrapTexture wt{toUse, scaledIntensity, (float)toUse.getHeight() / this-
>cam.getScale()};
    GUI::ScreenPosition p{windowWidth, toUse.getHeight()};
    wt.renderFixed(p, this->cam);
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 20/06/18
 */

#ifndef __WIND_H__
#define __WIND_H__

#include <Camera.h>
#include "GameTextures.h"

namespace GUI {
/**
 *  @brief receives the snapshot's intensity and draws the help interface
 *  to show the wind's intensity.
 */
class Wind {
   public:
    Wind(const GameTextureManager &textureManager, Camera &cam);
    ~Wind() = default;
    void render(std::int8_t intensity, int windowWidth);

   private:
    const GameTextureManager &tex;
    Camera &cam;
};
}

#endif  //__WIND_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#include "WormBackFlipping.h"

Worm::BackFlipping::BackFlipping() : State(StateID::BackFlipping) {}

Worm::BackFlipping::~BackFlipping() {}

void Worm::BackFlipping::update(float dt) {}

IO::PlayerInput Worm::BackFlipping::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::mortar(Worm &w) {
```



```
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::BackFlipping::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#ifndef __WORM_BACK_FLIPPING_H__
#define __WORM_BACK_FLIPPING_H__

#include "GameStateMsg.h"
#include "WormState.h"

namespace Worm {
class BackFlipping : public State {
   public:
    BackFlipping();
    ~BackFlipping();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;

    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;
    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__WORM_BACK_FLIPPING_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#include <SDL2/SDL_system.h>
#include <cmath>
#include <iostream>

#include "GameStateMsg.h"
#include "Text.h"
#include "Weapons/AerialAttack.h"
#include "Weapons/Banana.h"
#include "Weapons/BaseballBat.h"
#include "Weapons/Bazooka.h"
#include "Weapons/Cluster.h"
#include "Weapons/Dynamite.h"
#include "Weapons/Grenade.h"
#include "Weapons/Holy.h"
#include "Weapons/Mortar.h"
#include "Weapons/Teleport.h"
#include "Weapons/WeaponNone.h"
#include "Worm.h"
#include "WormState/BackFlip.h"
#include "WormState/Batting.h"
#include "WormState/Dead.h"
#include "WormState/Die.h"
#include "WormState/Drowning.h"
#include "WormState/Falling.h"
#include "WormState/Hit.h"
#include "WormState/Land.h"
#include "WormState/Sliding.h"
#include "WormState/Teleported.h"
#include "WormState/Teleporting.h"
#include "WormState/WormBackFlipping.h"
#include "WormState/WormEndBackFlip.h"
#include "WormState/WormEndJump.h"
#include "WormState/WormJumping.h"
#include "WormState/WormStartJump.h"
#include "WormState/WormStill.h"
#include "WormState/WormWalk.h"

Worm::Worm::Worm(ID id, const GUI::GameTextureManager &texture_mgr,
                 const GUI::GameSoundEffectManager &sound_effect_mgr)
    : id(id),
      texture_mgr(texture_mgr),
      sound_effect_mgr(sound_effect_mgr),
      animation(texture_mgr.get(GUI::GameTextures::WormIdle)) {
    this->setState(::Worm::StateID::Still);
    this->weapon = std::shared_ptr<Weapon>(new Bazooka(texture_mgr));
}

void Worm::Worm::handleKeyDown(SDL_Keycode key, IO::Stream<IO::PlayerMsg> *out)
{
    IO::PlayerInput i = IO::PlayerInput::moveNone;
    switch (key) {
        case SDLK_RIGHT:
            i = this->state->moveRight(*this);
            break;
        case SDLK_LEFT:
            i = this->state->moveLeft(*this);
            break;
```



```
        case SDLK_UP:
            i = this->state->pointUp(*this);
            break;
        case SDLK_DOWN:
            i = this->state->pointDown(*this);
            break;
        case SDLK_RETURN:
            i = this->state->jump(*this);
            break;
        case SDLK_BACKSPACE:
            i = this->state->backFlip(*this);
            break;
        case SDLK_1:
            i = this->state->setTimeoutTo(*this, 1);
            break;
        case SDLK_2:
            i = this->state->setTimeoutTo(*this, 2);
            break;
        case SDLK_3:
            i = this->state->setTimeoutTo(*this, 3);
            break;
        case SDLK_4:
            i = this->state->setTimeoutTo(*this, 4);
            break;
        case SDLK_5:
            i = this->state->setTimeoutTo(*this, 5);
            break;
        case SDLK_F1:
            i = this->state->bazooka(*this);
            break;
        case SDLK_F2:
            i = this->state->grenade(*this);
            break;
        case SDLK_F3:
            i = this->state->cluster(*this);
            break;
        case SDLK_F4:
            i = this->state->mortar(*this);
            break;
        case SDLK_F5:
            i = this->state->banana(*this);
            break;
        case SDLK_F6:
            i = this->state->holy(*this);
            break;
        case SDLK_F7:
            i = this->state->aerialAttack(*this);
            break;
        case SDLK_F8:
            i = this->state->dynamite(*this);
            break;
        case SDLK_F9:
            i = this->state->baseballBat(*this);
            break;
        case SDLK_F10:
            i = this->state->teleport(*this);
            break;
        case SDLK_SPACE:
            i = this->state->startShot(*this);
            break;
    }
    if (i != IO::PlayerInput::moveNone) {
```

```
        IO::PlayerMsg msg;
        msg.input = i;
        *out << msg;
    }
}

void Worm::Worm::handleKeyUp(SDL_Keycode key, IO::Stream<IO::PlayerMsg> *out) {
    IO::PlayerInput i = IO::PlayerInput::moveNone;
    switch (key) {
        case SDLK_RIGHT:
            i = this->state->stopMove(*this);
            break;
        case SDLK_LEFT:
            i = this->state->stopMove(*this);
            break;
        case SDLK_SPACE:
            i = this->state->endShot(*this);
            break;
    }
    if (i != IO::PlayerInput::moveNone) {
        IO::PlayerMsg msg;
        msg.input = i;
        *out << msg;
    }
}

void Worm::Worm::render(GUI::Position &p, GUI::Camera &cam) {
    SDL_RendererFlip flipType =
        this->direction == Direction::left ? SDL_FLIP_NONE : SDL_FLIP_HORIZONTAL
;
    if (this->state->getState() != StateID::Still ||
        this->weapon->getWeaponID() == WeaponID::WNone) {
        this->animation.render(p, cam, flipType);
    } else {
        this->weapon->render(p, cam, flipType);
    }
    if (this->explosion != nullptr) {
        this->explosion->render(cam);
        if (this->explosion->finished()) {
            this->explosion = nullptr;
        }
    }
}

void Worm::Worm::update(float dt) {
    this->state->update(dt);
    this->animation.update(dt);
    this->weapon->update(dt);
    if (this->explosion != nullptr) {
        this->explosion->update(dt);
    }
    if (this->soundEffectPlayer != nullptr) {
        this->soundEffectPlayer->update(dt);
    }
}

GUI::Animation Worm::Worm::getAnimation(StateID state) const {
    switch (state) {
        case StateID::Still:
            break;
        case StateID::Walk:
            return GUI::Animation{this->texture_mgr.get(GUI::GameTextures::WormW
```



```
alk)};
        case StateID::StartBackFlip:
        case StateID::StartJump:
            return GUI::Animation{this->texture_mgr.get(GUI::GameTextures::Start
Jump), true};
        case StateID::Jumping:
            return GUI::Animation{this->texture_mgr.get(GUI::GameTextures::Jumpi
ng)};
        case StateID::Land:
        case StateID::EndBackFlip:
        case StateID::EndJump:
            return GUI::Animation{this->texture_mgr.get(GUI::GameTextures::EndJu
mp), true};
        case StateID::BackFlipping: {
            GUI::Animation animation{this->texture_mgr.get(GUI::GameTextures::Ba
ckFlipping)};
            animation.setAnimateOnce();
            return animation;
        }
        case StateID::Falling: {
            GUI::Animation animation{this->texture_mgr.get(GUI::GameTextures::Fa
lling), true};
            animation.setAnimateOnce();
            return animation;
        }
        case StateID::Batting: {
            GUI::Animation animation{this->texture_mgr.get(GUI::GameTextures::Wo
rmBaseballBatting),
                                     false, 25, false};
            //            animation.setAnimateOnce();
            return animation;
        }
        case StateID::Teleporting: {
            GUI::Animation animation{this->texture_mgr.get(GUI::GameTextures::Wo
rmTeleporting),
                                     true};
            animation.setAnimateOnce();
            return animation;
        }
        case StateID::Teleported: {
            GUI::Animation animation{this->texture_mgr.get(GUI::GameTextures::Wo
rmTeleporting),
                                     true};
            animation.setPlayInverse();
            return animation;
        }
        case StateID::Hit:
            return GUI::Animation{this->texture_mgr.get(GUI::GameTextures::Fly),
 true,
                                  FLY_CENTER_FRAME, false};
        case StateID::Die: {
            GUI::Animation animation{this->texture_mgr.get(GUI::GameTextures::Di
e)};
            animation.setAnimateOnce();
            return animation;
        }
        case StateID::Drowning:
            return GUI::Animation{this->texture_mgr.get(GUI::GameTextures::Fly),
 true,
                                  DROWN_CENTER_FRAME, false};
        case StateID::Dead:
            return GUI::Animation{this->texture_mgr.get(GUI::GameTextures::Dead)
```

```
, true};
        case StateID::Sliding:
            return GUI::Animation{this->texture_mgr.get(GUI::GameTextures::Slidi
ng), true};
    }
    return GUI::Animation{this->texture_mgr.get(GUI::GameTextures::WormIdle), tr
ue};
}

void Worm::Worm::playSoundEffect(StateID state) {
    this->soundEffectPlayer = nullptr;
    switch (state) {
        case StateID::Still:
            break;
        case StateID::Walk:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::WalkCompre
ss), 0.7f});
            this->soundEffectPlayer->update(0.3f);
            break;
        case StateID::StartBackFlip:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::WormBackFl
ip), true});
            this->soundEffectPlayer->play();
            break;
        case StateID::StartJump:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::WormJump),
 true});
            this->soundEffectPlayer->play();
            break;
        case StateID::Jumping:
            break;
        case StateID::EndBackFlip:
        case StateID::EndJump:
        case StateID::Land:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::WormLandin
g), true});
            this->soundEffectPlayer->play();
            break;
        case StateID::BackFlipping:
            break;
        case StateID::Falling:
            break;
        case StateID::Batting:
            break;
        case StateID::Teleporting:
            break;
        case StateID::Teleported:
            break;
        case StateID::Hit:
            this->soundEffectPlayer =
```



```
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::WormHit),
true});
            this->soundEffectPlayer->play();
            break;
        case StateID::Die:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::WormDie),
true});
            this->soundEffectPlayer->play();
            break;
        case StateID::Drowning:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::WormDrowni
ng)});
            break;
        case StateID::Dead:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::Explosion)
, true});
            this->soundEffectPlayer->play();
            break;
        case StateID::Sliding:
            break;
    }
}

void Worm::Worm::setState(StateID state) {
    if (this->state == nullptr || this->state->getState() != state) {
        this->animation = this->getAnimation(state);
        this->playSoundEffect(state);

        /* creates the right state type */
        switch (state) {
            case StateID::Still:
                this->state = std::shared_ptr<State>(new Still());
                break;
            case StateID::Walk:
                this->state = std::shared_ptr<State>(new Walk());
                break;
            case StateID::StartJump:
                this->state = std::shared_ptr<State>(new StartJump());
                break;
            case StateID::Jumping:
                this->state = std::shared_ptr<State>(new Jumping());
                break;
            case StateID::EndJump:
                this->state = std::shared_ptr<State>(new EndJump());
                break;
            case StateID::StartBackFlip:
                this->state = std::shared_ptr<State>(new BackFlip());
                break;
            case StateID::BackFlipping:
                this->state = std::shared_ptr<State>(new BackFlipping());
                break;
```

```
            case StateID::EndBackFlip:
                this->state = std::shared_ptr<State>(new EndBackFlip());
                break;
            case StateID::Falling:
                this->state = std::shared_ptr<State>(new Falling());
                break;
            case StateID::Land:
                this->state = std::shared_ptr<State>(new Land());
                break;
            case StateID::Batting:
                this->state = std::shared_ptr<State>(new Batting());
                break;
            case StateID::Teleporting:
                this->state = std::shared_ptr<State>(new Teleporting());
                break;
            case StateID::Teleported:
                this->state = std::shared_ptr<State>(new Teleported());
                break;
            case StateID::Hit:
                this->state = std::shared_ptr<State>(new Hit());
                break;
            case StateID::Die:
                this->state = std::shared_ptr<State>(new Die());
                break;
            case StateID::Drowning:
                this->state = std::shared_ptr<State>(new Drowning());
                break;
            case StateID::Dead:
                this->state = std::shared_ptr<State>(new Dead());
                this->explosion = std::shared_ptr<Explosion>(new Explosion(this-
>texture_mgr));
                this->explosion->position = this->position;
                break;
            case StateID::Sliding:
                this->state = std::shared_ptr<State>(new Sliding());
                break;
        }
    }
}

Worm::StateID &Worm::Worm::getState() const {
    return this->state->getState();
}

void Worm::Worm::setWeapon(const WeaponID &id) {
    //    this->weapon.setWeapon(id);
    if (this->weapon->getWeaponID() != id) {
        switch (id) {
            case WeaponID::WBazooka:
                this->weapon = std::shared_ptr<Weapon>(new Bazooka(this->texture
_mgr));
                break;
            case WeaponID::WGrenade:
                this->weapon = std::shared_ptr<Weapon>(new Grenade(this->texture
_mgr));
                break;
            case WeaponID::WCluster:
                this->weapon = std::shared_ptr<Weapon>(new Cluster(this->texture
_mgr));
                break;
            case WeaponID::WMortar:
                this->weapon = std::shared_ptr<Weapon>(new Mortar(this->texture_
```



```
mgr));
                break;
            case WeaponID::WBanana:
                this->weapon = std::shared_ptr<Weapon>(new Banana(this->texture_
mgr));
                break;
            case WeaponID::WHoly:
                this->weapon = std::shared_ptr<Weapon>(new Holy(this->texture_mg
r));
                break;
            case WeaponID::WAerial:
                this->weapon = std::shared_ptr<Weapon>(new AerialAttack(this->te
xture_mgr));
                break;
            case WeaponID::WDynamite:
                this->weapon = std::shared_ptr<Weapon>(new Dynamite(this->textur
e_mgr));
                break;
            case WeaponID::WBaseballBat:
                this->weapon = std::shared_ptr<Weapon>(new BaseballBat(this->tex
ture_mgr));
                break;
            case WeaponID::WTeleport:
                this->weapon = std::shared_ptr<Weapon>(new Teleport(this->textur
e_mgr));
                break;
            case WeaponID::WNone:
                this->weapon = std::shared_ptr<Weapon>(new WeaponNone(this->text
ure_mgr));
                break;
            case WeaponID::WExplode:
                break;
            case WeaponID::WFragment:
                break;
        }
    }
}

const Worm::WeaponID &Worm::Worm::getWeaponID() const {
    return this->weapon->getWeaponID();
}

void Worm::Worm::setWeaponAngle(float angle) {
    this->weapon->setAngle(angle, this->direction);
}

void Worm::Worm::setPosition(GUI::Position p) {
    this->position = p;
}

void Worm::Worm::startShot() {
    if (!this->hasFired) {
        this->weapon->startShot();
    }
}

void Worm::Worm::endShot() {
    if (this->weapon->getWeaponID() != WeaponID::WAerial &&
        this->weapon->getWeaponID() != WeaponID::WTeleport &&
        this->weapon->getWeaponID() != WeaponID::WNone) {
        if (!this->hasFired) {
            this->weapon->endShot();
```

```
            this->playWeaponSoundEffect(this->getWeaponID());
            this->hasFired = true;
        }
    }
}

void Worm::Worm::mouseButtonDown(GUI::Position position, IO::Stream<IO::PlayerMs
g> *out) {
    IO::PlayerInput i = this->state->positionSelected(*this);
    if (i != IO::PlayerInput::moveNone && !this->hasFired && this->weapon->posit
ionSelected()) {
        this->playWeaponSoundEffect(this->weapon->getWeaponID());
        IO::PlayerMsg msg;
        msg.input = i;
        msg.position = position;
        *out << msg;
    }
}

void Worm::Worm::playWeaponSoundEffect(const WeaponID &id) {
    this->soundEffectPlayer = nullptr;

    switch (id) {
        case WeaponID::WBazooka:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::Shot), tru
e});
            this->soundEffectPlayer->play();
            break;
        case WeaponID::WGrenade:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::Shot), tru
e});
            this->soundEffectPlayer->play();
            break;
        case WeaponID::WCluster:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::Shot), tru
e});
            this->soundEffectPlayer->play();
            break;
        case WeaponID::WMortar:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::Shot), tru
e});
            this->soundEffectPlayer->play();
            break;
        case WeaponID::WBanana:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::Shot), tru
e});
            this->soundEffectPlayer->play();
```



```
            break;
        case WeaponID::WHoly:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::Holy), tru
e});
            this->soundEffectPlayer->play();
            break;
        case WeaponID::WAerial:
            this->soundEffectPlayer =
                std::shared_ptr<GUI::SoundEffectPlayer>(new GUI::SoundEffectPlay
er{
                    this->sound_effect_mgr.get(GUI::GameSoundEffects::AirStrike)
, true});
            this->soundEffectPlayer->play();
            break;
        case WeaponID::WDynamite:
            break;
        case WeaponID::WBaseballBat:
            break;
        case WeaponID::WTeleport:
            break;
        case WeaponID::WNone:
            break;
        case WeaponID::WExplode:
            break;
        case WeaponID::WFragment:
            break;
    }
}

void Worm::Worm::reset() {
    this->hasFired = false;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#include "WormEndBackFlip.h"

Worm::EndBackFlip::EndBackFlip() : State(StateID::EndBackFlip) {}

Worm::EndBackFlip::~EndBackFlip() {}

void Worm::EndBackFlip::update(float dt) {}

IO::PlayerInput Worm::EndBackFlip::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::mortar(Worm &w) {
```



```
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndBackFlip::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#ifndef __WORM_END_BACKFLIP_H__
#define __WORM_END_BACKFLIP_H__

#include "GameStateMsg.h"
#include "WormState.h"

namespace Worm {
class EndBackFlip : public State {
   public:
    EndBackFlip();
    ~EndBackFlip();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;

    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__WORM_END_BACKFLIP_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#include "WormEndJump.h"

Worm::EndJump::EndJump() : State(StateID::EndJump) {}

Worm::EndJump::~EndJump() {}

void Worm::EndJump::update(float dt) {}

IO::PlayerInput Worm::EndJump::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::mortar(Worm &w) {
```



```
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::EndJump::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#ifndef __END_JUMP_H__
#define __END_JUMP_H__

#include "GameStateMsg.h"
#include "WormState.h"

namespace Worm {
class EndJump : public State {
   public:
    EndJump();
    ~EndJump();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__END_JUMP_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#ifndef __Worm_H__
#define __Worm_H__

#define FLY_CENTER_FRAME 16
#define DROWN_CENTER_FRAME 0
#define ANGLE_STEP 5.625f

#include <SDL2/SDL.h>
#include <memory>

#include "Animation.h"
#include "Camera.h"
#include "Direction.h"
#include "GameSoundEffects.h"
#include "GameStateMsg.h"
#include "GameTextures.h"
#include "SoundEffectPlayer.h"
#include "Stream.h"
#include "Weapons/Explosion.h"
#include "Weapons/Weapon.h"
#include "WormState/WormState.h"
#include "utils.h"

namespace Worm {
using ID = char;

class Worm {
    /**
     * Fundamental class of the game, it is in charge of handling the user's
     * entries, and delegate in their attributes the rendering and animation
     */
   public:
    Direction direction{Direction::left};
    std::uint8_t health{0};
    const ID id;

    explicit Worm(ID id, const GUI::GameTextureManager &texture_mgr,
                  const GUI::GameSoundEffectManager &sound_effect_mgr);
    ~Worm() {}
    /**
     * @brief Calls State::update to change frame of animation
     * @param dt
     */
    void update(float dt);
    /**
     * Render worm in position (x,y)
     * @param x
     * @param y
     */
    void render(GUI::Position &p, GUI::Camera &cam);
    /**
     * @brief Using a state pattern, change its state depending on the input, an
d
     * sends it to the server
     * @param key
     * @param out
     */
```



```
    void handleKeyDown(SDL_Keycode key, IO::Stream<IO::PlayerMsg> *out);
    /**
     * @brief Same as handleKeyDown, but stops its current status.
     * @param key
     * @param out
     */
    void handleKeyUp(SDL_Keycode key, IO::Stream<IO::PlayerMsg> *out);
    /**
     * @brief Receives a position in global coordinates and sends it to the stat
e
     * so it can handle it.
     * @param position
     */
    void mouseButtonDown(GUI::Position position, IO::Stream<IO::PlayerMsg> *pStr
eam);
    GUI::Animation getAnimation(StateID state) const;
    /**
     * @brief Attributte that implements state pattern to change the behavior
     * of the class polymorphically.
     */
    void setState(StateID state);
    StateID &getState() const;
    /**
     * @brief Update the animation with weapons, depending on the
     * worm's angle.
     * @param angle
     */
    void setWeaponAngle(float angle);
    /**
     * @brief Update the used weapon
     * @param id
     */
    void setWeapon(const WeaponID &id);
    const WeaponID &getWeaponID() const;
    void setPosition(GUI::Position p);
    /**
     * @brief Starts the PowerBar's rendering, adding animations in its containe
r
     */
    void startShot();
    /**
     * @brief End PowerBar's rendering, freeing its container
     */
    void endShot();
    /**
     * @brief resets some attributes when the turn ends
     */
    void reset();

   private:
    const GUI::GameTextureManager &texture_mgr;
    const GUI::GameSoundEffectManager &sound_effect_mgr;
    std::shared_ptr<State> state{nullptr};
    GUI::Animation animation;
    std::shared_ptr<Weapon> weapon{nullptr};
    bool active{false};
    GUI::Position position{0, 0};
    std::shared_ptr<Explosion> explosion{nullptr};
    bool hasFired{false};
    std::shared_ptr<GUI::SoundEffectPlayer> soundEffectPlayer{nullptr};
    void playSoundEffect(StateID state);
    void playWeaponSoundEffect(const WeaponID &id);
```

```
};
}  // namespace Worm

#endif  //__Worm__H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#include "WormJumping.h"

Worm::Jumping::Jumping() : State(StateID::Jumping) {}

Worm::Jumping::~Jumping() {}

void Worm::Jumping::update(float dt) {}

IO::PlayerInput Worm::Jumping::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::mortar(Worm &w) {
```



```
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Jumping::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#ifndef __JUMPING_H__
#define __JUMPING_H__

#include "../Worm.h"
#include "GameStateMsg.h"

namespace Worm {
class Jumping : public State {
   public:
    explicit Jumping();
    virtual ~Jumping();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__JUMPING_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 19/05/18
 */

#include "WormStartJump.h"

Worm::StartJump::StartJump() : State(StateID::StartJump) {}

Worm::StartJump::~StartJump() {}

void Worm::StartJump::update(float dt) {}

IO::PlayerInput Worm::StartJump::moveRight(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::moveLeft(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::stopMove(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::jump(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::backFlip(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::mortar(Worm &w) {
```



```
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::StartJump::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Rodrigo.
 *  date: 19/05/18
 */

#ifndef __WORM_START_JUMP_H__
#define __WORM_START_JUMP_H__

#include "../Worm.h"
#include "GameStateMsg.h"
#include "WormState.h"

namespace Worm {
class StartJump : public State {
   public:
    StartJump();
    ~StartJump();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
};
}  // namespace Worm

#endif  //__WORM_START_JUMP_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#ifndef __WORM_STATE_H__
#define __WORM_STATE_H__

#include "Animation.h"
#include "GameStateMsg.h"

namespace Worm {

class Worm;
/**
 * Worm status interface. It is used to implement the state pattern and
 * thus obtain a polymorphic behavior and at the same time treat the
 * animation as a state machine
 */
class State {
   public:
    State(StateID stateID) : stateID(stateID){};
    virtual ~State() = default;

    virtual void update(float dt) = 0;

    virtual IO::PlayerInput moveRight(Worm &w) = 0;
    virtual IO::PlayerInput moveLeft(Worm &w) = 0;
    virtual IO::PlayerInput stopMove(Worm &w) = 0;
    virtual IO::PlayerInput pointUp(Worm &w) = 0;
    virtual IO::PlayerInput pointDown(Worm &w) = 0;
    virtual IO::PlayerInput jump(Worm &w) = 0;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) = 0;

    virtual IO::PlayerInput bazooka(Worm &w) = 0;
    virtual IO::PlayerInput grenade(Worm &w) = 0;
    virtual IO::PlayerInput cluster(Worm &w) = 0;
    virtual IO::PlayerInput mortar(Worm &w) = 0;
    virtual IO::PlayerInput banana(Worm &w) = 0;
    virtual IO::PlayerInput holy(Worm &w) = 0;
    virtual IO::PlayerInput aerialAttack(Worm &w) = 0;
    virtual IO::PlayerInput dynamite(Worm &w) = 0;
    virtual IO::PlayerInput baseballBat(Worm &w) = 0;
    virtual IO::PlayerInput teleport(Worm &w) = 0;

    virtual IO::PlayerInput startShot(Worm &w) = 0;
    virtual IO::PlayerInput endShot(Worm &w) = 0;
    virtual IO::PlayerInput backFlip(Worm &w) = 0;
    virtual IO::PlayerInput positionSelected(Worm &w) = 0;

    virtual StateID &getState() {
        return this->stateID;
    };

   protected:
    StateID stateID;
};
}  // namespace Worm

#endif  //__WORM_STATE_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#include "WormStill.h"
#include <iostream>
#include "Texture.h"

Worm::Still::Still() : State(StateID::Still) {}

Worm::Still::~Still() {}

void Worm::Still::update(float dt) {}

IO::PlayerInput Worm::Still::moveRight(Worm &w) {
    return IO::PlayerInput::moveRight;
}

IO::PlayerInput Worm::Still::moveLeft(Worm &w) {
    return IO::PlayerInput::moveLeft;
}

IO::PlayerInput Worm::Still::stopMove(Worm &w) {
    return IO::PlayerInput::stopMove;
}

IO::PlayerInput Worm::Still::jump(Worm &w) {
    return IO::PlayerInput::startJump;
}

IO::PlayerInput Worm::Still::backFlip(Worm &w) {
    return IO::PlayerInput::startBackFlip;
}

IO::PlayerInput Worm::Still::bazooka(Worm &w) {
    return IO::PlayerInput::bazooka;
}

IO::PlayerInput Worm::Still::pointUp(Worm &w) {
    return IO::PlayerInput::pointUp;
}

IO::PlayerInput Worm::Still::pointDown(Worm &w) {
    return IO::PlayerInput::pointDown;
}

IO::PlayerInput Worm::Still::startShot(Worm &w) {
    w.startShot();
    return IO::PlayerInput::startShot;
}

IO::PlayerInput Worm::Still::endShot(Worm &w) {
    w.endShot();
    return IO::PlayerInput::endShot;
}

IO::PlayerInput Worm::Still::grenade(Worm &w) {
    return IO::PlayerInput::grenade;
}

IO::PlayerInput Worm::Still::cluster(Worm &w) {
```



```
    return IO::PlayerInput::cluster;
}

IO::PlayerInput Worm::Still::mortar(Worm &w) {
    return IO::PlayerInput::mortar;
}

IO::PlayerInput Worm::Still::banana(Worm &w) {
    return IO::PlayerInput::banana;
}

IO::PlayerInput Worm::Still::holy(Worm &w) {
    return IO::PlayerInput::holy;
}

IO::PlayerInput Worm::Still::setTimeoutTo(Worm &w, int time) {
    switch (time) {
        case 1:
            return IO::PlayerInput::timeout1;
        case 2:
            return IO::PlayerInput::timeout2;
        case 3:
            return IO::PlayerInput::timeout3;
        case 4:
            return IO::PlayerInput::timeout4;
        case 5:
            return IO::PlayerInput::timeout5;
        default:
            return IO::PlayerInput::moveNone;
    }
}

IO::PlayerInput Worm::Still::aerialAttack(Worm &w) {
    return IO::PlayerInput::aerialAttack;
}

IO::PlayerInput Worm::Still::positionSelected(Worm &w) {
    return IO::PlayerInput::positionSelected;
}

IO::PlayerInput Worm::Still::dynamite(Worm &w) {
    return IO::PlayerInput::dynamite;
}

IO::PlayerInput Worm::Still::teleport(Worm &w) {
    return IO::PlayerInput::teleport;
}

IO::PlayerInput Worm::Still::baseballBat(Worm &w) {
    return IO::PlayerInput::baseballBat;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#ifndef __WORM_QUIET_H__
#define __WORM_QUIET_H__

#include <SDL2/SDL_system.h>

#include "../Worm.h"
#include "GameStateMsg.h"
#include "WormState.h"

namespace Worm {
class Still : public State {
   public:
    Still();
    ~Still();
    virtual void update(float dt) override;
    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__WORM_QUIET_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#include "WormWalk.h"
#include <iostream>

Worm::Walk::Walk() : State(StateID::Walk) {}

Worm::Walk::~Walk() {}

void Worm::Walk::update(float dt) {}

IO::PlayerInput Worm::Walk::moveLeft(Worm &w) {
    if (w.direction == Direction::left) {
        return IO::PlayerInput::moveNone;
    }
    return IO::PlayerInput::moveLeft;
}

IO::PlayerInput Worm::Walk::moveRight(Worm &w) {
    if (w.direction == Direction::right) {
        return IO::PlayerInput::moveNone;
    }
    return IO::PlayerInput::moveRight;
}

IO::PlayerInput Worm::Walk::stopMove(Worm &w) {
    return IO::PlayerInput::stopMove;
}

IO::PlayerInput Worm::Walk::jump(Worm &w) {
    return IO::PlayerInput::startJump;
}

IO::PlayerInput Worm::Walk::backFlip(Worm &w) {
    return IO::PlayerInput::startBackFlip;
}

IO::PlayerInput Worm::Walk::bazooka(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::pointUp(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::pointDown(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::startShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::endShot(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::grenade(Worm &w) {
    return IO::PlayerInput::moveNone;
```



```
}

IO::PlayerInput Worm::Walk::cluster(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::mortar(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::banana(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::holy(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::setTimeoutTo(Worm &w, int t) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::aerialAttack(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::positionSelected(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::dynamite(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::teleport(Worm &w) {
    return IO::PlayerInput::moveNone;
}

IO::PlayerInput Worm::Walk::baseballBat(Worm &w) {
    return IO::PlayerInput::moveNone;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#ifndef __WORM_WALK_H__
#define __WORM_WALK_H__

#include <SDL2/SDL_system.h>

#include "../Worm.h"
#include "GameStateMsg.h"
#include "WormState.h"

namespace Worm {
class Walk : public State {
   public:
    explicit Walk();
    virtual ~Walk();

    virtual void update(float dt) override;

    virtual IO::PlayerInput moveRight(Worm &w) override;
    virtual IO::PlayerInput moveLeft(Worm &w) override;
    virtual IO::PlayerInput stopMove(Worm &w) override;
    virtual IO::PlayerInput jump(Worm &w) override;
    virtual IO::PlayerInput backFlip(Worm &w) override;
    virtual IO::PlayerInput setTimeoutTo(Worm &w, int t) override;

    virtual IO::PlayerInput bazooka(Worm &w) override;
    virtual IO::PlayerInput grenade(Worm &w) override;
    virtual IO::PlayerInput cluster(Worm &w) override;
    virtual IO::PlayerInput mortar(Worm &w) override;
    virtual IO::PlayerInput banana(Worm &w) override;
    virtual IO::PlayerInput holy(Worm &w) override;
    virtual IO::PlayerInput aerialAttack(Worm &w) override;
    virtual IO::PlayerInput dynamite(Worm &w) override;
    virtual IO::PlayerInput baseballBat(Worm &w) override;
    virtual IO::PlayerInput teleport(Worm &w) override;
    virtual IO::PlayerInput positionSelected(Worm &w) override;

    virtual IO::PlayerInput startShot(Worm &w) override;
    virtual IO::PlayerInput endShot(Worm &w) override;
    virtual IO::PlayerInput pointUp(Worm &w) override;
    virtual IO::PlayerInput pointDown(Worm &w) override;
};
}  // namespace Worm

#endif  //__WORM_WALK_H__
```

```cpp
#include "editor.h"

Editor::Editor::Editor(QWidget *parent) : QGraphicsView(parent) {}

Editor::Editor::~Editor() {}

void Editor::Editor::wheelEvent(QWheelEvent *event) {
    QGraphicsView::wheelEvent(event);
    if (event->isAccepted()) {
        return;
    }

    static qreal factor = 1.1;

    if (event->angleDelta().y() > 0) {
        scale(factor, factor);
    } else {
        scale(1 / factor, 1 / factor);
    }

    event->accept();
}

void Editor::Editor::mousePressEvent(QMouseEvent *event) {
    QGraphicsView::mousePressEvent(event);
    if (event->isAccepted()) {
        return;
    }

    switch (event->button()) {
        case Qt::LeftButton:
            break;

        case Qt::RightButton:
            break;

        default:
            break;
    }

    event->accept();
}
```

```
#ifndef EDITOR_H
#define EDITOR_H

#include <QGraphicsView>
#include <QWheelEvent>

namespace Editor {
class Editor : public QGraphicsView {
   public:
    Editor(QWidget *parent);
    ~Editor();

    void wheelEvent(QWheelEvent *event);
    void mousePressEvent(QMouseEvent *event);
};
}

// Q_DECLARE_METATYPE(Editor::Editor);

#endif  // EDITOR_H
```

```
#-------------------------------------------------
#
# Project created by QtCreator 2018-06-02T20:55:03
#
#-------------------------------------------------

QT       += core gui
QMAKE_CXXFLAGS += -std=c++0x

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = Editor
TEMPLATE = app


SOURCES += main.cpp\
        mainwindow.cpp \
    editor.cpp \
    editorview.cpp \
    editorscene.cpp \
    stagedata.cpp \
    stageelement.cpp \
    stageelementworm.cpp \
    stageelemlonggirder.cpp \
    stageelemshortgirder.cpp \
    qgraphicsitemlayer.cpp \
    ../../../libs/yaml-cpp/contrib/graphbuilder.cpp \
    ../../../libs/yaml-cpp/contrib/graphbuilderadapter.cpp \
    ../../../libs/yaml-cpp/binary.cpp \
    ../../../libs/yaml-cpp/convert.cpp \
    ../../../libs/yaml-cpp/directives.cpp \
    ../../../libs/yaml-cpp/emit.cpp \
    ../../../libs/yaml-cpp/emitfromevents.cpp \
    ../../../libs/yaml-cpp/emitter.cpp \
    ../../../libs/yaml-cpp/emitterstate.cpp \
    ../../../libs/yaml-cpp/emitterutils.cpp \
    ../../../libs/yaml-cpp/exceptions.cpp \
    ../../../libs/yaml-cpp/exp.cpp \
    ../../../libs/yaml-cpp/memory.cpp \
    ../../../libs/yaml-cpp/node.cpp \
    ../../../libs/yaml-cpp/node_data.cpp \
    ../../../libs/yaml-cpp/nodebuilder.cpp \
    ../../../libs/yaml-cpp/nodeevents.cpp \
    ../../../libs/yaml-cpp/null.cpp \
    ../../../libs/yaml-cpp/ostream_wrapper.cpp \
    ../../../libs/yaml-cpp/parse.cpp \
    ../../../libs/yaml-cpp/parser.cpp \
    ../../../libs/yaml-cpp/regex_yaml.cpp \
    ../../../libs/yaml-cpp/scanner.cpp \
    ../../../libs/yaml-cpp/scanscalar.cpp \
    ../../../libs/yaml-cpp/scantag.cpp \
    ../../../libs/yaml-cpp/scantoken.cpp \
    ../../../libs/yaml-cpp/simplekey.cpp \
    ../../../libs/yaml-cpp/singledocparser.cpp \
    ../../../libs/yaml-cpp/stream.cpp \
    ../../../libs/yaml-cpp/tag.cpp

HEADERS  += mainwindow.h \
    editor.h \
    editorview.h \
    editorscene.h \
    stagedata.h \
```



```
    stageelement.h \
    stageelementworm.h \
    stageelemlonggirder.h \
    stageelemshortgirder.h \
    qgraphicsitemlayer.h \
    ../../../libs/yaml-cpp/contrib/anchordict.h \
    ../../../libs/yaml-cpp/contrib/graphbuilder.h \
    ../../../libs/yaml-cpp/contrib/graphbuilderadapter.h \
    ../../../libs/yaml-cpp/node/detail/bool_type.h \
    ../../../libs/yaml-cpp/node/detail/impl.h \
    ../../../libs/yaml-cpp/node/detail/iterator.h \
    ../../../libs/yaml-cpp/node/detail/iterator_fwd.h \
    ../../../libs/yaml-cpp/node/detail/memory.h \
    ../../../libs/yaml-cpp/node/detail/node.h \
    ../../../libs/yaml-cpp/node/detail/node_data.h \
    ../../../libs/yaml-cpp/node/detail/node_iterator.h \
    ../../../libs/yaml-cpp/node/detail/node_ref.h \
    ../../../libs/yaml-cpp/node/convert.h \
    ../../../libs/yaml-cpp/node/emit.h \
    ../../../libs/yaml-cpp/node/impl.h \
    ../../../libs/yaml-cpp/node/iterator.h \
    ../../../libs/yaml-cpp/node/node.h \
    ../../../libs/yaml-cpp/node/parse.h \
    ../../../libs/yaml-cpp/node/ptr.h \
    ../../../libs/yaml-cpp/node/type.h \
    ../../../libs/yaml-cpp/anchor.h \
    ../../../libs/yaml-cpp/binary.h \
    ../../../libs/yaml-cpp/collectionstack.h \
    ../../../libs/yaml-cpp/directives.h \
    ../../../libs/yaml-cpp/dll.h \
    ../../../libs/yaml-cpp/emitfromevents.h \
    ../../../libs/yaml-cpp/emitter.h \
    ../../../libs/yaml-cpp/emitterdef.h \
    ../../../libs/yaml-cpp/emittermanip.h \
    ../../../libs/yaml-cpp/emitterstate.h \
    ../../../libs/yaml-cpp/emitterstyle.h \
    ../../../libs/yaml-cpp/emitterutils.h \
    ../../../libs/yaml-cpp/eventhandler.h \
    ../../../libs/yaml-cpp/exceptions.h \
    ../../../libs/yaml-cpp/exp.h \
    ../../../libs/yaml-cpp/indentation.h \
    ../../../libs/yaml-cpp/mark.h \
    ../../../libs/yaml-cpp/nodebuilder.h \
    ../../../libs/yaml-cpp/nodeevents.h \
    ../../../libs/yaml-cpp/noncopyable.h \
    ../../../libs/yaml-cpp/null.h \
    ../../../libs/yaml-cpp/ostream_wrapper.h \
    ../../../libs/yaml-cpp/parser.h \
    ../../../libs/yaml-cpp/ptr_vector.h \
    ../../../libs/yaml-cpp/regex_yaml.h \
    ../../../libs/yaml-cpp/regeximpl.h \
    ../../../libs/yaml-cpp/scanner.h \
    ../../../libs/yaml-cpp/scanscalar.h \
    ../../../libs/yaml-cpp/scantag.h \
    ../../../libs/yaml-cpp/setting.h \
    ../../../libs/yaml-cpp/singledocparser.h \
    ../../../libs/yaml-cpp/stlemitter.h \
    ../../../libs/yaml-cpp/stream.h \
    ../../../libs/yaml-cpp/streamcharsource.h \
    ../../../libs/yaml-cpp/stringsource.h \
    ../../../libs/yaml-cpp/tag.h \
    ../../../libs/yaml-cpp/token.h \
```

```
    ../../../libs/yaml-cpp/traits.h \
    ../../../libs/yaml-cpp/yaml.h

FORMS    += mainwindow.ui

RESOURCES += \
    resources.qrc

INCLUDEPATH += ../../../libs
```

```
#include "editorscene.h"
#include <QDebug>
#include <QGraphicsPixmapItem>
#include <QImage>
#include <QMouseEvent>
#include <QPainter>

EditorScene::EditorScene(QRectF rect) : QGraphicsScene(nullptr), rect(rect) {
    this->setSceneRect(rect);
}

void EditorScene::setCursor(StageElement *newCursor) {
    if (this->cursor) {
        delete this->cursor;
    }
    this->cursor = newCursor;
    QGraphicsScene::addItem(this->cursor);
}

void EditorScene::hideCursor() {
    if (this->cursor) {
        QGraphicsScene::removeItem(this->cursor);
    }
}

void EditorScene::showCursor() {
    if (this->cursor) {
        QGraphicsScene::addItem(this->cursor);
    }
}

void EditorScene::addItem(QGraphicsItem *elem) {
    if (elem->scene() != this) {
        QGraphicsScene::addItem(elem);
    }
}

void EditorScene::addItem(StageElement *elem) {
    if (elem->scene() != this) {
        if (!this->rect.contains(elem->getPosition())) {
            return;
        }
        QGraphicsScene::addItem(elem);
        this->elements.insert(elem);
    }
}

void EditorScene::removeItem(StageElement *elem) {
    if (this->contains(elem)) {
        this->elements.erase(this->elements.find(elem));

        if (elem->scene()) {
            QGraphicsScene::removeItem(elem);
        }
    }
}

void EditorScene::removeItem(QGraphicsItem *elem) {
    if (elem->scene()) {
        QGraphicsScene::removeItem(elem);
    }
}
```



```
void EditorScene::serialize(StageData &sd) {
    sd.bgColor = this->bgColor;
    for (auto *elem : this->elements) {
        elem->serialize(sd);
    }
}

QList<StageElement *> EditorScene::collidingItems(StageElement *elem) {
    QList<StageElement *> rv;
    for (QGraphicsItem *other : QGraphicsScene::collidingItems(elem)) {
        if (other == elem) {
            continue;
        }

        if (this->contains(dynamic_cast<StageElement *>(other))) {
            rv.append(dynamic_cast<StageElement *>(other));
        }
    }
    return rv;
}

bool EditorScene::contains(StageElement *elem) {
    auto it = this->elements.find(elem);
    return (it != this->elements.end());
}

void EditorScene::setBgColor(QColor color) {
    this->bgColor = color;
    if (this->bgColorLayer) {
        this->removeItem(this->bgColorLayer);
        delete this->bgColorLayer;
    }

    this->bgColorLayer = new QGraphicsItemLayer;
    this->bgColorLayer->setZValue(-4);
    this->addItem(this->bgColorLayer);
    QGraphicsRectItem *bg = new QGraphicsRectItem(this->rect, this->bgColorLayer
);
    bg->setBrush(QBrush{color});
}

void EditorScene::setFartherBg(QImage image) {
    this->setBackground(image, &this->fartherBg, -3);
}

void EditorScene::setMedianBg(QImage image) {
    this->setBackground(image, &this->medianBg, -2);
}

void EditorScene::setCloserBg(QImage image) {
    this->setBackground(image, &this->closeBg, -1);
}

void EditorScene::setBackground(QImage image, QGraphicsItemLayer **layerPtr, qre
al zValue) {
    if (*layerPtr) {
        this->removeItem(*layerPtr);
        delete *layerPtr;
    }

    *layerPtr = new QGraphicsItemLayer;
```

```
    QGraphicsItemLayer *layer = *layerPtr;

    layer->setZValue(zValue);
    this->addItem(layer);

    for (int i = 0; i < this->rect.width() / image.width() + 1; i++) {
        QGraphicsPixmapItem *pix = new QGraphicsPixmapItem{layer};
        pix->setPixmap(QPixmap::fromImage(image));
        pix->setPos(image.width() * i, this->rect.height() - image.height());
    }
}
```

```
#ifndef EDITORSCENE_H
#define EDITORSCENE_H

#include <QColor>
#include <QGraphicsScene>
#include <QImage>
#include <QObject>
#include <QWidget>
#include <set>
#include <string>
#include "qgraphicsitemlayer.h"
#include "stageelement.h"

class EditorScene : public QGraphicsScene {
    Q_OBJECT

   public:
    EditorScene(QRectF rect);

    void setCursor(StageElement *newCursor);
    void hideCursor();
    void showCursor();

    void addItem(QGraphicsItem *elem);
    void addItem(StageElement *elem);
    void removeItem(QGraphicsItem *elem);
    void removeItem(StageElement *elem);

    virtual QList<StageElement *> collidingItems(StageElement *elem);
    bool contains(StageElement *elem);

    void serialize(StageData &sd);

    /* background */
    void setBgColor(QColor color);
    void setFartherBg(QImage image);
    void setMedianBg(QImage image);
    void setCloserBg(QImage image);

   private:
    void setBackground(QImage image, QGraphicsItemLayer **layerPtr, qreal zValue
);

    QRectF rect;
    QColor bgColor{Qt::white};
    QGraphicsItemLayer *closeBg{nullptr};
    QGraphicsItemLayer *medianBg{nullptr};
    QGraphicsItemLayer *fartherBg{nullptr};
    QGraphicsItemLayer *bgColorLayer{nullptr};
    StageElement *cursor{nullptr};
    std::string resource;
    std::set<StageElement *> elements;
};

#endif  // EDITORSCENE_H
```

```
#include "editorview.h"
#include <QGraphicsPixmapItem>
#include <QImage>
#include <QtDebug>
#include <QScrollBar>
#include <cmath>
#include "stageelementworm.h"
#include "stageelemlonggirder.h"
#include "stageelemshortgirder.h"

const qreal cursorOpacity = 0.7;

EditorView::EditorView(QWidget *parent) : QGraphicsView(parent) {
    this->setAttribute(Qt::WA_Hover, true);
    this->setTransformationAnchor(QGraphicsView::AnchorUnderMouse);
    this->setLongGirder();
}

void EditorView::drawCloseBg(QString &) {}

void EditorView::setScene(EditorScene *scene) {
    QGraphicsView::setScene(scene);
    this->escene = scene;
    this->horizontalScrollBar()->setValue(this->horizontalScrollBar()->maximum()
 / 2);
    this->verticalScrollBar()->setValue(this->verticalScrollBar()->maximum());
}

void EditorView::setWorm() {
    if (this->stageElem) {
        delete this->stageElem;
    }

    this->stageElem = new StageElementWorm{cursorOpacity};
}

void EditorView::setShortGirder() {
    if (this->stageElem) {
        delete this->stageElem;
    }

    this->stageElem = new StageElemShortGirder{cursorOpacity};
}

void EditorView::setLongGirder() {
    if (this->stageElem) {
        delete this->stageElem;
    }

    this->stageElem = new StageElemLongGirder{cursorOpacity};
}

void EditorView::mousePressEvent(QMouseEvent *) {}

void EditorView::deleteAt(QPoint pos) {
    this->scene()->removeItem(this->stageElem);
    QGraphicsItem *item = this->itemAt(pos);
    if (item) {
        this->escene->removeItem(static_cast<StageElement *>(item));
    }
    this->escene->addItem(this->stageElem);
}
```



```
void EditorView::keyPressEvent(QKeyEvent *event) {
    if (event->key() == Qt::Key_Plus) {
        this->stageElem->increaseAngle();
    } else if (event->key() == Qt::Key_Minus) {
        this->stageElem->decreaseAngle();
    }
}

void EditorView::createAt(QPoint pos) {
    if (!this->stageElem) {
        return;
    }

    if (this->collides()) {
        return;
    }

    StageElement *newElem = this->stageElem->clone();

    QPointF lpos = this->mapToScene(pos);
    lpos.rx() -= newElem->pixmap().width() / 2;
    lpos.ry() -= newElem->pixmap().height() / 2;

    newElem->setPos(lpos);
    this->escene->addItem(newElem);
}

void EditorView::mouseReleaseEvent(QMouseEvent *event) {
    event->accept();

    if (event->button() & Qt::RightButton) {
        this->deleteAt(event->pos());
    } else {
        this->createAt(event->pos());
        this->stageElem->setZValue(1);
    }
}

void EditorView::mouseMoveEvent(QMouseEvent *event) {
    if (!this->stageElem) {
        return;
    }

    /* set the position of the hint image under the mouse */
    QPointF pos = this->mapToScene(event->pos());
    pos.rx() -= this->stageElem->pixmap().width() / 2;
    pos.ry() -= this->stageElem->pixmap().height() / 2;

    this->stageElem->setPos(pos);
    event->accept();
}

bool EditorView::event(QEvent *event) {
    switch (event->type()) {
        case QEvent::HoverEnter:
            if (this->stageElem) {
                this->setFocus();
                this->escene->addItem(dynamic_cast<QGraphicsItem *>(this->stageE
lem));
            }
            return true;
```

```
        case QEvent::HoverLeave:
            if (this->stageElem) {
                this->escene->removeItem(dynamic_cast<QGraphicsItem *>(this->sta
geElem));
            }
            return true;
        default:
            break;
    }

    return QGraphicsView::event(event);
}

void EditorView::wheelEvent(QWheelEvent *event) {
    static qreal factor = 1.1;

    if (event->delta() > 0) {
        this->scale(factor, factor);
    } else {
        this->scale(1.0 / factor, 1.0 / factor);
    }

    /* set the position of the hint image under the mouse */
    QPointF pos = this->mapToScene(event->pos());
    pos.rx() -= this->stageElem->pixmap().width() / 2;
    pos.ry() -= this->stageElem->pixmap().height() / 2;

    this->stageElem->setPos(pos);
    event->accept();
}

bool EditorView::collides() {
    for (StageElement *other : this->escene->collidingItems(this->stageElem)) {
        if (!this->stageElem->canOverlap(other)) {
            return true;
        }
    }

    return false;
}

void EditorView::serialize(StageData &sd) const {
    if (this->stageElem) {
        this->escene->removeItem(this->stageElem);
    }

    this->escene->serialize(sd);

    if (this->stageElem) {
        this->escene->addItem(this->stageElem);
    }
}
```

```
#ifndef EDITORVIEW_H
#define EDITORVIEW_H

#include <QEvent>
#include <QGraphicsView>
#include <QObject>
#include <QWheelEvent>
#include <QWidget>
#include "editorscene.h"
#include "stagedata.h"
#include "stageelement.h"

class EditorView : public QGraphicsView {
    Q_OBJECT

   public:
    EditorView(QWidget *parent);
    virtual void setScene(EditorScene *scene);

    void drawCloseBg(QString &fileName);

   public slots:
    void setWorm();
    void setShortGirder();
    void setLongGirder();

    void serialize(StageData &sd) const;

    // QWidget interface
   protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *);
    void mouseMoveEvent(QMouseEvent *);
    void wheelEvent(QWheelEvent *);
    bool event(QEvent *event);
    void hoverEvent(QHoverEvent *event);
    void keyPressEvent(QKeyEvent *event);

    bool collides();
    void deleteAt(QPoint pos);
    void createAt(QPoint pos);

   private:
    StageElement *stageElem{nullptr};
    EditorScene *escene{nullptr};
};

#endif  // EDITORVIEW_H
```

```
#include <QApplication>
#include "mainwindow.h"

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

```
#include "mainwindow.h"
#include <QColor>
#include <QColorDialog>
#include <QErrorMessage>
#include <QFileDialog>
#include <fstream>
#include "stagedata.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWi
ndow) {
    ui->setupUi(this);

    this->scene = new EditorScene{this->stageSize};
    this->ui->editorView->setScene(this->scene);

    this->ui->colorPreview->setScene(new QGraphicsScene);

    /* toolbar */
    connect(this->ui->actionAdd_Worm, SIGNAL(triggered(bool)), this->ui->editorV
iew,
            SLOT(setWorm()));
    connect(this->ui->actionAdd_Long_Girder, SIGNAL(triggered(bool)), this->ui->
editorView,
            SLOT(setLongGirder()));
    connect(this->ui->actionShort_Girder, SIGNAL(triggered(bool)), this->ui->edi
torView,
            SLOT(setShortGirder()));

    QColor defaultColor{0xba, 0x8d, 0xc6};
    this->scene->setBgColor(defaultColor);
    this->ui->colorPreview->setBackgroundBrush(QBrush(defaultColor));

    this->showMaximized();
}

MainWindow::~MainWindow() {
    delete ui;
    delete scene;
}

void MainWindow::on_actionLejano_triggered() {
    QString fileName = QFileDialog::getOpenFileName(
        this, tr("Seleccione una imagen para el fondo lejano"), "/home", tr("Image Files (*.png)"))
;
    if (!fileName.isEmpty()) {
        this->fartherBgFile = fileName;
        this->scene->setFartherBg(QImage(fileName));
    }
}

void MainWindow::on_actionMedio_triggered() {
    QString fileName = QFileDialog::getOpenFileName(
        this, tr("Seleccione una imagen para el fondo medio"), "/home", tr("Image Files (*.png)"))
;
    if (!fileName.isEmpty()) {
        this->midBgFile = fileName;
        this->scene->setMedianBg(QImage(fileName));
    }
}

void MainWindow::on_actionCercano_triggered() {
```



```
    QString fileName =
        QFileDialog::getOpenFileName(this, tr("Seleccione una imagen para el fondo cercano")
,
                                     "/home", tr("Image Files (*.png)"));
    if (!fileName.isEmpty()) {
        this->closeBgFile = fileName;
        this->scene->setCloserBg(QImage(fileName));
    }
}

void MainWindow::on_bgColorButton_clicked() {
    QColor color = QColorDialog::getColor(Qt::white, this);
    if (color.isValid()) {
        this->scene->setBgColor(color);
        this->ui->colorPreview->setBackgroundBrush(QBrush(color));
    }
}

void MainWindow::on_actionOpen_triggered() {
    /* serializes the stage */
    StageData sd{this->stageSize.width(), stageSize.height()};

    sd.closeBgFile = this->closeBgFile;
    sd.medianBgFile = this->midBgFile;
    sd.fartherBgFile = this->fartherBgFile;
    sd.wormsHealth = this->ui->wormsHP->value();
    sd.numPlayers= this->ui->numPlayers->value();

    /* weapon ammo */
    const QString WEAPON_PREFIX = "wpn_";
    for(auto *child : this->ui->stageParams->children()) {
        if(child->objectName().startsWith(WEAPON_PREFIX)) {
            QString weaponName = child->objectName().remove(0, WEAPON_PREFIX.siz
e());

            QSpinBox *widget = dynamic_cast<QSpinBox *>(child);
            sd.addWeaponAmmo(weaponName, widget->value());
        }
    }

    this->ui->editorView->serialize(sd);

    if (static_cast<int>(sd.numWorms()) < sd.numPlayers) {
        QErrorMessage::qtHandler()->showMessage("Se necesita al menos 1 worm por jugador");
        return;
    }

    /* gets the output file name */
    QString fileName = QFileDialog::getSaveFileName(this, tr("Nombre de archivo de sali
da"),
                                                    "/home", tr("YAML (*.yml)"));

    /* checks if a file was selected */
    if (fileName.isEmpty()) {
        return;
    }

    std::ofstream file;
    file.open(fileName.toStdString(), std::ios::out | std::ios::trunc);
    if (!file) {
        QErrorMessage::qtHandler()->showMessage("Error al abrir el archivo");
        return;
```

```
    }
    /* gets the base name of the file */
    QStringList list = fileName.split('/');
    QList<QString>::Iterator it = list.end();
    it--;
    QStringList list2 = it->split('.');
    sd.dump(file,list2[0].toStdString());
}
```

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QGraphicsScene>
#include <QGraphicsView>
#include <QMainWindow>
#include <QString>
#include "editorscene.h"

namespace Ui {

class MainWindow;
}

class MainWindow : public QMainWindow {
    Q_OBJECT

   public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

   private slots:
    void on_actionLejano_triggered();

    void on_actionMedio_triggered();

    void on_actionCercano_triggered();

    void on_bgColorButton_clicked();

    void on_actionOpen_triggered();

   private:
    Ui::MainWindow *ui;
    QRectF stageSize{0, 0, 13 * 250, 13 * 250};
    EditorScene *scene;
    QString closeBgFile;
    QString midBgFile;
    QString fartherBgFile;
};

#endif  // MAINWINDOW_H
```

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>1065</width>
    <height>609</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralWidget">
   <property name="sizePolicy">
    <sizepolicy hsizetype="Maximum" vsizetype="Maximum">
     <horstretch>0</horstretch>
     <verstretch>0</verstretch>
    </sizepolicy>
   </property>
   <layout class="QGridLayout" name="gridLayout">
    <item row="0" column="1">
     <widget class="QScrollArea" name="stageParamsContainer">
      <property name="minimumSize">
       <size>
        <width>250</width>
        <height>200</height>
       </size>
      </property>
      <property name="maximumSize">
       <size>
        <width>250</width>
        <height>16777215</height>
       </size>
      </property>
      <property name="widgetResizable">
       <bool>true</bool>
      </property>
      <widget class="QWidget" name="stageParams">
       <property name="geometry">
        <rect>
         <x>0</x>
         <y>0</y>
         <width>235</width>
         <height>617</height>
        </rect>
       </property>
       <layout class="QFormLayout" name="formLayout">
        <item row="0" column="0" colspan="2">
         <widget class="QLabel" name="label_2">
          <property name="frameShadow">
           <enum>QFrame::Plain</enum>
          </property>
          <property name="lineWidth">
           <number>1</number>
          </property>
          <property name="text">
           <string>&lt;b&gt;Worms&lt;/b&gt;</string>
          </property>
          <property name="textFormat">
```



```
           <enum>Qt::RichText</enum>
          </property>
          <property name="scaledContents">
           <bool>false</bool>
          </property>
          <property name="alignment">
           <set>Qt::AlignCenter</set>
          </property>
          <property name="wordWrap">
           <bool>false</bool>
          </property>
          <property name="textInteractionFlags">
           <set>Qt::NoTextInteraction</set>
          </property>
         </widget>
        </item>
        <item row="2" column="0">
         <widget class="QLabel" name="label">
          <property name="text">
           <string>HP</string>
          </property>
         </widget>
        </item>
        <item row="2" column="1">
         <widget class="QSpinBox" name="wormsHP">
          <property name="maximum">
           <number>250</number>
          </property>
          <property name="singleStep">
           <number>10</number>
          </property>
          <property name="value">
           <number>100</number>
          </property>
         </widget>
        </item>
        <item row="4" column="0" colspan="2">
         <widget class="Line" name="line">
          <property name="orientation">
           <enum>Qt::Horizontal</enum>
          </property>
         </widget>
        </item>
        <item row="5" column="0">
         <spacer name="verticalSpacer">
          <property name="orientation">
           <enum>Qt::Vertical</enum>
          </property>
          <property name="sizeHint" stdset="0">
           <size>
            <width>20</width>
            <height>10</height>
           </size>
          </property>
         </spacer>
        </item>
        <item row="6" column="0" colspan="2">
         <widget class="QLabel" name="label_3">
          <property name="text">
           <string>&lt;html&gt;&lt;head/&gt;&lt;body&gt;&lt;p&gt;&lt;span style=
&quot; font-weight:600;&quot;&gt;Turno&lt;/span&gt;&lt;/p&gt;&lt;/body&gt;&lt;/h
tml&gt;</string>
```

```
    </property>
    <property name="alignment">
     <set>Qt::AlignCenter</set>
    </property>
   </widget>
  </item>
  <item row="7" column="0">
   <widget class="QLabel" name="label_4">
    <property name="text">
     <string>DuraciÃ³n [s]</string>
    </property>
   </widget>
  </item>
  <item row="7" column="1">
   <widget class="QSpinBox" name="spinBox_2">
    <property name="maximum">
     <number>120</number>
    </property>
    <property name="singleStep">
     <number>5</number>
    </property>
    <property name="value">
     <number>40</number>
    </property>
   </widget>
  </item>
  <item row="8" column="0" colspan="2">
   <widget class="Line" name="line_2">
    <property name="minimumSize">
     <size>
      <width>50</width>
      <height>0</height>
     </size>
    </property>
    <property name="orientation">
     <enum>Qt::Horizontal</enum>
    </property>
   </widget>
  </item>
  <item row="9" column="0">
   <spacer name="verticalSpacer_2">
    <property name="orientation">
     <enum>Qt::Vertical</enum>
    </property>
    <property name="sizeHint" stdset="0">
     <size>
      <width>20</width>
      <height>10</height>
     </size>
    </property>
   </spacer>
  </item>
  <item row="10" column="0" colspan="2">
   <widget class="QLabel" name="label_5">
    <property name="text">
     <string>&lt;b&gt;Armas&lt;/b&gt;</string>
    </property>
    <property name="alignment">
     <set>Qt::AlignCenter</set>
    </property>
   </widget>
  </item>
```



```
  <item row="11" column="0">
   <widget class="QLabel" name="label_6">
    <property name="text">
     <string>Bazooka</string>
    </property>
   </widget>
  </item>
  <item row="11" column="1">
   <widget class="QSpinBox" name="wpn_bazooka">
    <property name="maximum">
     <number>999</number>
    </property>
    <property name="value">
     <number>999</number>
    </property>
   </widget>
  </item>
  <item row="12" column="0">
   <widget class="QLabel" name="label_7">
    <property name="text">
     <string>Granada Verde</string>
    </property>
   </widget>
  </item>
  <item row="12" column="1">
   <widget class="QSpinBox" name="wpn_grenade">
    <property name="maximum">
     <number>999</number>
    </property>
    <property name="value">
     <number>999</number>
    </property>
   </widget>
  </item>
  <item row="13" column="0">
   <widget class="QLabel" name="label_8">
    <property name="text">
     <string>Granada Roja</string>
    </property>
   </widget>
  </item>
  <item row="13" column="1">
   <widget class="QSpinBox" name="wpn_cluster">
    <property name="maximum">
     <number>999</number>
    </property>
    <property name="value">
     <number>5</number>
    </property>
   </widget>
  </item>
  <item row="14" column="0">
   <widget class="QLabel" name="label_9">
    <property name="text">
     <string>Mortero</string>
    </property>
   </widget>
  </item>
  <item row="14" column="1">
   <widget class="QSpinBox" name="wpn_mortar">
    <property name="value">
     <number>5</number>
```

```
   </property>
  </widget>
 </item>
 <item row="15" column="0">
  <widget class="QLabel" name="label_10">
   <property name="text">
    <string>Banana</string>
   </property>
  </widget>
 </item>
 <item row="15" column="1">
  <widget class="QSpinBox" name="wpn_banana">
   <property name="maximum">
    <number>999</number>
   </property>
   <property name="value">
    <number>5</number>
   </property>
  </widget>
 </item>
 <item row="16" column="0">
  <widget class="QLabel" name="label_11">
   <property name="text">
    <string>Granada Santa</string>
   </property>
  </widget>
 </item>
 <item row="16" column="1">
  <widget class="QSpinBox" name="wpn_holy">
   <property name="maximum">
    <number>999</number>
   </property>
   <property name="value">
    <number>2</number>
   </property>
  </widget>
 </item>
 <item row="17" column="0">
  <widget class="QLabel" name="label_12">
   <property name="text">
    <string>Dinamita</string>
   </property>
  </widget>
 </item>
 <item row="17" column="1">
  <widget class="QSpinBox" name="wpn_dynamite">
   <property name="maximum">
    <number>999</number>
   </property>
   <property name="value">
    <number>2</number>
   </property>
  </widget>
 </item>
 <item row="18" column="0">
  <widget class="QLabel" name="label_13">
   <property name="text">
    <string>Aereo</string>
   </property>
  </widget>
 </item>
 <item row="18" column="1">
```



```
  <widget class="QSpinBox" name="wpn_aerialAttack">
   <property name="maximum">
    <number>999</number>
   </property>
   <property name="value">
    <number>1</number>
   </property>
  </widget>
 </item>
 <item row="19" column="0">
  <widget class="QLabel" name="label_14">
   <property name="text">
    <string>Bate </string>
   </property>
  </widget>
 </item>
 <item row="19" column="1">
  <widget class="QSpinBox" name="wpn_baseballBat">
   <property name="maximum">
    <number>999</number>
   </property>
   <property name="value">
    <number>1</number>
   </property>
  </widget>
 </item>
 <item row="20" column="0">
  <widget class="QLabel" name="label_15">
   <property name="text">
    <string>TeletransportaciÃ³n</string>
   </property>
  </widget>
 </item>
 <item row="20" column="1">
  <widget class="QSpinBox" name="wpn_teleport">
   <property name="maximum">
    <number>999</number>
   </property>
   <property name="value">
    <number>2</number>
   </property>
  </widget>
 </item>
 <item row="21" column="0" colspan="2">
  <widget class="Line" name="line_3">
   <property name="minimumSize">
    <size>
     <width>100</width>
     <height>0</height>
    </size>
   </property>
   <property name="orientation">
    <enum>Qt::Horizontal</enum>
   </property>
  </widget>
 </item>
 <item row="22" column="0">
  <spacer name="verticalSpacer_4">
   <property name="orientation">
    <enum>Qt::Vertical</enum>
   </property>
   <property name="sizeHint" stdset="0">
```

```
          <size>
           <width>20</width>
           <height>10</height>
          </size>
         </property>
        </spacer>
       </item>
       <item row="23" column="0" colspan="2">
        <widget class="QLabel" name="label_16">
         <property name="text">
          <string>&lt;b&gt;Fondo&lt;/b&gt;</string>
         </property>
         <property name="alignment">
          <set>Qt::AlignCenter</set>
         </property>
        </widget>
       </item>
       <item row="24" column="0">
        <widget class="QPushButton" name="bgColorButton">
         <property name="text">
          <string>Color</string>
         </property>
        </widget>
       </item>
       <item row="24" column="1">
        <widget class="QGraphicsView" name="colorPreview">
         <property name="enabled">
          <bool>true</bool>
         </property>
         <property name="sizePolicy">
          <sizepolicy hsizetype="Ignored" vsizetype="Ignored">
           <horstretch>0</horstretch>
           <verstretch>0</verstretch>
          </sizepolicy>
         </property>
         <property name="backgroundBrush">
          <brush brushstyle="SolidPattern">
           <color alpha="255">
            <red>255</red>
            <green>255</green>
            <blue>255</blue>
           </color>
          </brush>
         </property>
         <property name="foregroundBrush">
          <brush brushstyle="NoBrush">
           <color alpha="255">
            <red>186</red>
            <green>141</green>
            <blue>198</blue>
           </color>
          </brush>
         </property>
        </widget>
       </item>
       <item row="3" column="1">
        <widget class="QSpinBox" name="numPlayers">
         <property name="minimum">
          <number>1</number>
         </property>
         <property name="maximum">
          <number>4</number>
```



```
         </property>
        </widget>
       </item>
       <item row="3" column="0">
        <widget class="QLabel" name="label_17">
         <property name="text">
          <string>Jugadores</string>
         </property>
        </widget>
       </item>
      </layout>
     </widget>
    </widget>
   </item>
   <item row="0" column="0">
    <widget class="EditorView" name="editorView">
     <property name="sizePolicy">
      <sizepolicy hsizetype="Expanding" vsizetype="Expanding">
       <horstretch>0</horstretch>
       <verstretch>0</verstretch>
      </sizepolicy>
     </property>
     <property name="minimumSize">
      <size>
       <width>500</width>
       <height>500</height>
      </size>
     </property>
     <property name="cursor" stdset="0">
      <cursorShape>BlankCursor</cursorShape>
     </property>
     <property name="layoutDirection">
      <enum>Qt::RightToLeft</enum>
     </property>
     <property name="sizeAdjustPolicy">
      <enum>QAbstractScrollArea::AdjustToContents</enum>
     </property>
     <property name="alignment">
      <set>Qt::AlignCenter</set>
     </property>
     <property name="renderHints">
      <set>QPainter::SmoothPixmapTransform|QPainter::TextAntialiasing</set>
     </property>
     <property name="dragMode">
      <enum>QGraphicsView::NoDrag</enum>
     </property>
     <property name="transformationAnchor">
      <enum>QGraphicsView::AnchorUnderMouse</enum>
     </property>
     <property name="resizeAnchor">
      <enum>QGraphicsView::AnchorUnderMouse</enum>
     </property>
     <property name="viewportUpdateMode">
      <enum>QGraphicsView::BoundingRectViewportUpdate</enum>
     </property>
     <property name="rubberBandSelectionMode">
      <enum>Qt::IntersectsItemBoundingRect</enum>
     </property>
    </widget>
   </item>
  </layout>
 </widget>
```

```
  <widget class="QMenuBar" name="menuBar">
   <property name="geometry">
    <rect>
     <x>0</x>
     <y>0</y>
     <width>1065</width>
     <height>25</height>
    </rect>
   </property>
   <widget class="QMenu" name="menuEditor">
    <property name="title">
     <string>File</string>
    </property>
    <addaction name="actionOpen"/>
   </widget>
   <widget class="QMenu" name="menuFondo">
    <property name="title">
     <string>Fondo</string>
    </property>
    <addaction name="actionLejano"/>
    <addaction name="actionMedio"/>
    <addaction name="actionCercano"/>
   </widget>
   <addaction name="menuEditor"/>
   <addaction name="menuFondo"/>
  </widget>
  <widget class="QToolBar" name="mainToolBar">
   <property name="sizePolicy">
    <sizepolicy hsizetype="Preferred" vsizetype="Fixed">
     <horstretch>0</horstretch>
     <verstretch>0</verstretch>
    </sizepolicy>
   </property>
   <property name="iconSize">
    <size>
     <width>50</width>
     <height>50</height>
    </size>
   </property>
   <attribute name="toolBarArea">
    <enum>TopToolBarArea</enum>
   </attribute>
   <attribute name="toolBarBreak">
    <bool>false</bool>
   </attribute>
   <addaction name="actionAdd_Worm"/>
   <addaction name="actionAdd_Long_Girder"/>
   <addaction name="actionShort_Girder"/>
  </widget>
  <action name="actionOpen">
   <property name="text">
    <string>Export</string>
   </property>
  </action>
  <action name="actionAdd_Worm">
   <property name="icon">
    <iconset resource="resources.qrc">
     <normaloff>:/assets/buttons/worm.png</normaloff>:/assets/buttons/worm.png</
iconset>
   </property>
   <property name="text">
    <string>Add Worm</string>
```



```
   </property>
   <property name="toolTip">
    <string>&lt;html&gt;&lt;head/&gt;&lt;body&gt;&lt;p&gt;Add Worm&lt;/p&gt;&lt;
/body&gt;&lt;/html&gt;</string>
   </property>
   <property name="shortcut">
    <string>W</string>
   </property>
  </action>
  <action name="actionAdd_Long_Girder">
   <property name="icon">
    <iconset resource="resources.qrc">
     <normaloff>:/assets/buttons/long-girder.png</normaloff>:/assets/buttons/lon
g-girder.png</iconset>
   </property>
   <property name="text">
    <string>Long Girder</string>
   </property>
   <property name="toolTip">
    <string>&lt;html&gt;&lt;head/&gt;&lt;body&gt;&lt;p&gt;Add long girder&lt;/p&
gt;&lt;/body&gt;&lt;/html&gt;</string>
   </property>
   <property name="shortcut">
    <string>L</string>
   </property>
  </action>
  <action name="actionShort_Girder">
   <property name="icon">
    <iconset resource="resources.qrc">
     <normaloff>:/assets/buttons/short-girder.png</normaloff>:/assets/buttons/sh
ort-girder.png</iconset>
   </property>
   <property name="text">
    <string>Short Girder</string>
   </property>
   <property name="toolTip">
    <string>&lt;html&gt;&lt;head/&gt;&lt;body&gt;&lt;p&gt;Add short girder&lt;/p
&gt;&lt;/body&gt;&lt;/html&gt;</string>
   </property>
   <property name="shortcut">
    <string>S</string>
   </property>
  </action>
  <action name="actionLejano">
   <property name="text">
    <string>Lejano</string>
   </property>
  </action>
  <action name="actionMedio">
   <property name="text">
    <string>Medio</string>
   </property>
  </action>
  <action name="actionCercano">
   <property name="text">
    <string>Cercano</string>
   </property>
  </action>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <customwidgets>
  <customwidget>
```

```
   <class>EditorView</class>
   <extends>QGraphicsView</extends>
   <header>editorview.h</header>
   <slots>
    <slot>setWorm()</slot>
   </slots>
  </customwidget>
 </customwidgets>
 <resources>
  <include location="resources.qrc"/>
 </resources>
 <connections/>
</ui>
```

```
#include "qgraphicsitemlayer.h"

QGraphicsItemLayer::QGraphicsItemLayer() : QGraphicsItem(nullptr) {}

QRectF QGraphicsItemLayer::boundingRect() const {
    return QRectF(0, 0, 0, 0);
}

void QGraphicsItemLayer::paint(QPainter *, const QStyleOptionGraphicsItem *, QWi
dget *) {}
```

```
#ifndef QGRAPHICSITEMLAYER_H
#define QGRAPHICSITEMLAYER_H

#include <QGraphicsItem>
#include <QObject>

class QGraphicsItemLayer : public QGraphicsItem {
   public:
    QGraphicsItemLayer();

    virtual QRectF boundingRect() const;
    virtual void paint(QPainter *, const QStyleOptionGraphicsItem *, QWidget *);
};

#endif  // QGRAPHICSITEMLAYER_H
```

```
<RCC>
    <qresource prefix="/">
        <file>assets/buttons/worm.png</file>
        <file>assets/buttons/long-girder.png</file>
        <file>assets/buttons/short-girder.png</file>
        <file>assets/stage/long_girder.png</file>
        <file>assets/stage/short_girder.png</file>
        <file>assets/stage/worm.png</file>
    </qresource>
</RCC>
```

```
#include "stagedata.h"
#include <QDebug>
#include <cassert>

const qreal scale = 13.0;

YAML::Emitter& operator<<(YAML::Emitter& out, const QColor& v) {
    out << YAML::Flow;
    out << YAML::BeginSeq << v.red() << v.green() << v.blue() << YAML::EndSeq;
    return out;
}

YAML::Emitter& operator<<(YAML::Emitter& out, const QPointF& v) {
    out << YAML::Flow;
    out << YAML::BeginSeq << v.x() << v.y() << YAML::EndSeq;
    return out;
}

YAML::Emitter& operator<<(YAML::Emitter& out, const WormData& v) {
    out << YAML::BeginMap;

    out << YAML::Key << "position";
    out << YAML::Value << v.position;

    out << YAML::EndMap;
    return out;
}

YAML::Emitter& operator<<(YAML::Emitter& out, const GirderData& v) {
    out << YAML::BeginMap;

    out << YAML::Key << "position";
    out << YAML::Value << v.position;

    out << YAML::Key << "angle";
    out << YAML::Value << v.angle;

    out << YAML::Key << "length";
    out << YAML::Value << v.length;

    out << YAML::EndMap;
    return out;
}

StageData::StageData(qreal width, qreal height) : width(width / scale), height(h
eight / scale) {}

QPointF StageData::toGameCoords(const QPointF& point) const {
    qreal xpos = (point.x() / scale - this->width / 2.0);
    qreal ypos = this->height - point.y() / scale;
    return QPointF(xpos, ypos);
}

std::size_t StageData::numWorms() const {
    return this->worms.size();
}

void StageData::dump(std::ostream& output, std::string fileName) {
    YAML::Emitter emitter;

    emitter << YAML::BeginMap;
```



```
    emitter << YAML::Key << "name";
    emitter << YAML::Value << fileName;

    emitter << YAML::Key << "numPlayers";
    emitter << YAML::Value << this->numPlayers;

    emitter << YAML::Key << "weaponsAmmo";
    emitter << YAML::Value << this->weapons;

    emitter << YAML::Key << "width";
    emitter << YAML::Value << this->width;

    emitter << YAML::Key << "height";
    emitter << YAML::Value << this->height;

    emitter << YAML::Key << "wormsHealth";
    emitter << YAML::Value << this->wormsHealth;

    emitter << YAML::Key << "worms";
    emitter << YAML::Value << this->worms;

    emitter << YAML::Key << "girders";
    emitter << YAML::Value << this->girders;

    emitter << YAML::Key << "background";
    emitter << YAML::Value;
    {
        emitter << YAML::BeginMap;
        emitter << YAML::Key << "closeBackgroundFile";
        emitter << YAML::Value << this->closeBgFile.toStdString();

        emitter << YAML::Key << "midBackgroundFile";
        emitter << YAML::Value << this->medianBgFile.toStdString();

        emitter << YAML::Key << "fartherBackgroundFile";
        emitter << YAML::Value << this->fartherBgFile.toStdString();

        emitter << YAML::Key << "color";
        emitter << YAML::Value << this->bgColor;

        emitter << YAML::EndMap;
    }

    emitter << YAML::EndMap;

    assert(emitter.good());

    output << emitter.c_str();
}

void StageData::addWorm(QPointF position) {
    this->worms.push_back(WormData{this->toGameCoords(position)});
}

void StageData::addShortGirder(QPointF position, qreal angle) {
    this->girders.push_back(GirderData{this->toGameCoords(position), -angle, 5.3
845});
}

void StageData::addLongGirder(QPointF position, qreal angle) {
    this->girders.push_back(GirderData{this->toGameCoords(position), -angle, 10.
769});
```

```
}

void StageData::addWeaponAmmo(QString weaponName, int ammo) {
    this->weapons[weaponName.toStdString()] = ammo;
}
```

```
#ifndef STAGEDATA_H
#define STAGEDATA_H

#include <QColor>
#include <QDebug>
#include <QPoint>
#include <QString>
#include <Qt>
#include <iostream>
#include <vector>
#include <map>
#include "yaml-cpp/yaml.h"

struct GirderData {
    QPointF position;
    qreal angle;
    qreal length;
};

struct WormData {
    QPointF position;
};

class StageData {
   public:
    QString fartherBgFile;
    QString medianBgFile;
    QString closeBgFile;
    QColor bgColor;
    int wormsHealth;
    int numPlayers;

    StageData(qreal width, qreal height);

    void dump(std::ostream &output, std::string fileName);

    std::size_t numWorms() const;

    void addWorm(QPointF position);
    void addShortGirder(QPointF position, qreal angle);
    void addLongGirder(QPointF position, qreal angle);
    void addWeaponAmmo(QString weaponName, int ammo);

   private:
    QPointF toGameCoords(const QPointF &point) const;

    qreal width;
    qreal height;
    std::vector<GirderData> girders;
    std::vector<WormData> worms;
    std::map<std::string, int> weapons;
};

#endif  // STAGEDATA_H
```

```cpp
#include "stageelement.h"
#include <QPainter>

const double PI = 3.141592653589793;

StageElement::StageElement(const std::string &resource, ItemType type, qreal opa
city)
    : QGraphicsPixmapItem(nullptr), type(type), resource(resource) {
    this->setPixmap(this->getResource(opacity));
    this->setTransformOriginPoint(this->pixmap().width() / 2, this->pixmap().hei
ght() / 2);
    this->setFlag(QGraphicsItem::ItemIsMovable);
}

ItemType StageElement::getType() {
    return this->type;
}

qreal StageElement::getAngle() const {
    return this->angle;
}

QPointF StageElement::getPosition() const {
    qreal hw = this->pixmap().width() / 2.0;
    qreal hh = this->pixmap().height() / 2.0;
    return QPointF{this->pos().x() + hw, this->pos().y() + hh};
}

void StageElement::increaseAngle() {
    this->angle += 90.0f / 10.0f;

    if (this->angle > 90.0f) {
        this->angle = 90.0f;
    }

    this->setRotation(this->angle);
}

void StageElement::decreaseAngle() {
    this->angle -= 90.0f / 10.0f;

    if (this->angle < -90.0f) {
        this->angle = -90.0f;
    }

    this->setRotation(this->angle);
}

QPixmap StageElement::getResource(qreal opacity) {
    QImage image;
    image.load(this->resource.c_str());
    image = image.convertToFormat(QImage::Format_ARGB32);

    QImage image2(image.size(), QImage::Format_ARGB32);
    image2.fill(Qt::transparent);

    QPainter painter(&image2);
    painter.setOpacity(opacity);
    painter.drawImage(image.rect(), image);

    return QPixmap::fromImage(image2);
}
```



```cpp
bool StageElement::canOverlap(StageElement *) {
    return false;
}
```

```
#ifndef STAGEELEMENT_H
#define STAGEELEMENT_H

#include <QGraphicsItem>
#include <QGraphicsPixmapItem>
#include <QObject>
#include <QWidget>
#include <QtDebug>
#include <string>
#include "stagedata.h"

enum class ItemType {
    Worm,
    ShortGirder,
    LongGirder,
};

class StageElement : public QGraphicsPixmapItem {
   public:
    StageElement(const std::string &resource, ItemType type, qreal opacity);

    ItemType getType();

    qreal getAngle() const;
    QPointF getPosition() const;

    virtual StageElement *clone() = 0;
    virtual bool canOverlap(StageElement *other);

    virtual void increaseAngle();
    virtual void decreaseAngle();

    void setRotationEnabled(bool);

    virtual void serialize(StageData &sd) = 0;

   protected:
    QPixmap getResource(qreal opacity = 1.0);
    qreal angle{0.0f};
    ItemType type;

   private:
    std::string resource;
};

#endif  // STAGEELEMENT_H
```

```cpp
#include "stageelement.h"
#include "stageelementworm.h"

StageElementWorm::StageElementWorm(qreal opacity)
    : StageElement(":/assets/stage/worm.png", ItemType::Worm, opacity) {}

void StageElementWorm::increaseAngle() {}

void StageElementWorm::decreaseAngle() {}

StageElement *StageElementWorm::clone() {
    auto *e = new StageElementWorm;
    e->angle = this->angle;
    return e;
}

void StageElementWorm::serialize(StageData &sd) {
    sd.addWorm(this->getPosition());
}
```

```
#ifndef STAGEELEMENTWORM_H
#define STAGEELEMENTWORM_H

#include "stageelement.h"

class StageElementWorm : public StageElement {
   public:
    StageElementWorm(qreal opacity = 1.0);

    virtual StageElement *clone();

    virtual void increaseAngle() override;
    virtual void decreaseAngle() override;
    virtual void serialize(StageData &sd);
};

#endif  // STAGEELEMENTWORM_H
```

```cpp
#include "stageelemlonggirder.h"

StageElemLongGirder::StageElemLongGirder(qreal opacity)
    : StageElement(":/assets/stage/long_girder.png", ItemType::LongGirder, opacity) {}

StageElement *StageElemLongGirder::clone() {
    auto *e = new StageElemLongGirder;
    e->angle = this->angle;
    e->setRotation(this->angle);
    return e;
}

void StageElemLongGirder::serialize(StageData &sd) {
    sd.addLongGirder(this->getPosition(), this->getAngle());
}

bool StageElemLongGirder::canOverlap(StageElement *other) {
    return (other->getType() != ItemType::Worm);
}
```

```
#ifndef STAGEELEMLONGGIRDER_H
#define STAGEELEMLONGGIRDER_H

#include "stageelement.h"

class StageElemLongGirder : public StageElement {
   public:
    StageElemLongGirder(qreal opacity = 1.0);

    virtual StageElement *clone();
    virtual bool canOverlap(StageElement *other);
    virtual void serialize(StageData &sd);
};

#endif  // STAGEELEMLONGGIRDER_H
```

```cpp
#include "stageelemshortgirder.h"

StageElemShortGirder::StageElemShortGirder(qreal opacity)
    : StageElement(":/assets/stage/short_girder.png", ItemType::ShortGirder, opacity) {}

StageElement *StageElemShortGirder::clone() {
    auto *e = new StageElemShortGirder;
    e->angle = this->angle;
    e->setRotation(this->angle);
    return e;
}

void StageElemShortGirder::serialize(StageData &sd) {
    sd.addShortGirder(this->getPosition(), this->getAngle());
}

bool StageElemShortGirder::canOverlap(StageElement *other) {
    return (other->getType() != ItemType::Worm);
}
```

```
#ifndef STAGEELEMSHORTGIRDER_H
#define STAGEELEMSHORTGIRDER_H

#include <stageelement.h>

class StageElemShortGirder : public StageElement {
   public:
    StageElemShortGirder(qreal opacity = 1.0);

    StageElement *clone();
    virtual bool canOverlap(StageElement *other);
    void serialize(StageData &sd);
};

#endif  // STAGEELEMSHORTGIRDER_H
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 17/05/18.
 */
#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>

#include <algorithm>
#include <cassert>
#include <iostream>
#include <string>

#include "Animation.h"
#include "Texture.h"

GUI::Animation::Animation(const Texture &texture) : texture(&texture) {
    /* assumes the frames are squares */
    this->numFrames = this->texture->getHeight() / this->texture->getWidth();
    this->size = this->texture->getWidth();

    assert(this->numFrames > 0);
    assert(this->size > 0);
}

GUI::Animation::Animation(const Texture &texture, bool playReversed) : Animation
(texture) {
    this->playReversed = playReversed;
}

GUI::Animation::Animation(const GUI::Texture &texture, bool playReversed, int in
itialFrame,
                          bool autoUpdate)
    : Animation(texture, playReversed) {
    assert(this->numFrames > initialFrame);
    this->currentFrame = initialFrame;
    this->autoUpdate = autoUpdate;
}

GUI::Animation::~Animation() {}

void GUI::Animation::update(float dt) {
    if (this->autoUpdate) {
        this->elapsed += dt;
        /* checks if the frame should be updated based on the time elapsed since
 the last update */
        while (this->elapsed > this->frameRate) {
            this->advanceFrame();
            this->elapsed -= this->frameRate;
        }
    }
}

/**
 * @brief Renders the animation in the given coordinates.
 *
 * @param renderer Renderer.
 * @param x X coordinate.
 * @param y Y corrdinate.
 */
void GUI::Animation::render(Position &p, Camera &cam, const SDL_RendererFlip &fl
ipType) {
    this->setFlip(flipType);
```



```
    SDL_Rect clip = {0, this->size * this->currentFrame, this->size, this->size}
;
    cam.draw(*this->texture, p, clip, this->flipType);
}

/**
 * @brief Resets the animation.
 *
 */
void GUI::Animation::reset() {
    this->currentFrame = 0;
}

void GUI::Animation::advanceFrame() {
    if (!this->playInverse) {
        if (this->playReversed) {
            if (this->currentFrame == 0 && this->step < 0) {
                this->step = 1;
            } else if (this->currentFrame == this->numFrames - 1 && this->step >
 0) {
                this->step = -1;
            }
        }

        if (this->playOnce &&
            this->currentFrame == this->numFrames - 1) {  // Por ahora ignora el
 playReversed
            this->animationFinished = true;
        }

        if (!this->animationFinished) {
            this->currentFrame = (this->currentFrame + this->step) % this->numFr
ames;
        }
    } else {
        if (this->currentFrame == 0) {
            this->animationFinished = true;
        } else {
            this->currentFrame -= 1;
        }
    }
}

void GUI::Animation::setFlip(SDL_RendererFlip flip_type) {
    this->flipType = flip_type;
}

SDL_RendererFlip GUI::Animation::getFlip() {
    return this->flipType;
}

void GUI::Animation::setFrame(int frame) {
    assert(frame < this->numFrames);
    assert(frame >= 0);
    this->currentFrame = frame;
}

void GUI::Animation::setAnimateOnce() {
    this->playOnce = true;
}

bool GUI::Animation::finished() {
```

```
    return this->animationFinished;
}

void GUI::Animation::setAutoUpdate(bool autoUpdate) {
    this->autoUpdate = autoUpdate;
}

void GUI::Animation::setPlayInverse() {
    this->playInverse = true;
    this->currentFrame = this->numFrames - 1;
}
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 17/05/18.
 */

#ifndef __ANIMATION_H__
#define __ANIMATION_H__

#include <SDL2/SDL.h>

#include "Camera.h"
#include "Color.h"
#include "Exception.h"
#include "Texture.h"

namespace GUI {
class Animation {
   public:
    Animation(const Texture &texture);
    Animation(const Texture &texture, bool playReversed);
    Animation(const Texture &texture, bool playReversed, int initialFrame, bool
autoUpdate);
    ~Animation();
    void update(float dt);
    void render(Position &p, Camera &cam, const SDL_RendererFlip &flipType);
    void reset();
    void advanceFrame();
    /**
     * Sets frame manually. Commonly used when the worm is aiming with a weapon
     * @param frame
     */
    void setFrame(int frame);
    void setFlip(SDL_RendererFlip flipType);
    SDL_RendererFlip getFlip();
    void setAnimateOnce();
    void setAutoUpdate(bool autoUpdate);
    bool finished();

    void setPlayInverse();

   private:
    /** SDL texture of the raw image. */
    const Texture *texture;
    /**
     * Disable Automatic update (when worm uses a bazooka, the frame changes
     * when user presses up or down)
     */
    bool autoUpdate{true};
    /** Current animation frame. */
    int currentFrame{0};
    /** Total number of frames in the sprite. */
    int numFrames;
    /** Size of the sprite (height and width). */
    int size;
    /** Time elapsed since last update. */
    float elapsed{0.0f};
    /** Frames per seconds (should this be elsewere?). */
    float frameRate{1.0f / 25.0f};
    SDL_RendererFlip flipType{SDL_FLIP_NONE};
    /** If true, when the animation finishes, it's played reversed instead of re
starting. */
    bool playReversed{false};
```



```
    /** If true, plays the animation once. */
    bool playOnce{false};
    /** Frame step. */
    int step{1};
    bool animationFinished{false};
    bool playInverse{false};
};
}  // namespace GUI

#endif  //__ANIMATION_H__
```

```
#include "Buffer.h"
#include <cstdlib>
#include <cstring>

IO::Buffer::Buffer() {
    this->dataSize = 2;
    this->data = (uint8_t *)malloc(this->dataSize);
}

IO::Buffer::Buffer(const std::string &data) {
    this->dataSize = data.size();
    this->data = (uint8_t *)malloc(this->dataSize);

    memcpy(this->data, data.data(), data.size());
}

IO::Buffer::~Buffer() {
    if (this->data) {
        free(this->data);
    }
}

std::string IO::Buffer::asString() {
    std::string output;
    output.reserve(this->dataSize);
    output.append((char *)this->data, this->dataSize);
    return output;
}

void IO::Buffer::appendFloat(float v) {
    static_assert(sizeof(v) == 4, "needs 32 bit float");

    union {
        uint32_t u32;
        float f;
    } u;
    u.f = v;
    this->append(u.u32);
}

void IO::Buffer::appendBuffer(void *data, std::size_t dataSize) {
    while (this->dataSize - this->offset < dataSize) {
        this->grow();
    }
    memcpy(this->data + this->offset, data, dataSize);
    this->offset += dataSize;
}

void IO::Buffer::grow() {
    this->dataSize *= 2;
    this->data = static_cast<uint8_t *>(realloc(this->data, this->dataSize));
}

void IO::Buffer::getData(const uint8_t **data, std::size_t *size) {
    *data = this->data;
    *size = this->offset;
}

float IO::Buffer::extractFloat() {
    uint32_t v = this->extract<uint32_t>();
    union {
        uint32_t u32;
```



```
        float f;
    } u = {v};
    return u.f;
}
```

```
#ifndef BUFFER_H_
#define BUFFER_H_

#include <arpa/inet.h>
#include <cstdint>
#include <cstring>
#include <string>
#include "Exception.h"

namespace IO {
class Buffer {
   public:
    Buffer();
    explicit Buffer(const std::string &data);
    ~Buffer();

    template <typename NUMERIC>
    void append(NUMERIC v);
    void appendFloat(float v);
    void appendBuffer(void *data, std::size_t dataSize);

    std::string asString();

    template <typename NUMERIC>
    NUMERIC extract();
    float extractFloat();

    void getData(const uint8_t **data, std::size_t *size);

   private:
    void grow();

    uint8_t *data{nullptr};
    std::size_t dataSize{0};
    std::size_t offset{0};
};
}  // namespace IO

template <typename NUMERIC>
void IO::Buffer::append(NUMERIC v) {
    switch (sizeof(NUMERIC)) {
        case 1: {
            this->appendBuffer(&v, sizeof(v));
            return;
        }
        case 2: {
            uint16_t vt = htons(v);
            this->appendBuffer(&vt, sizeof(vt));
            return;
        }
        case 4: {
            uint32_t vt = htonl(v);
            this->appendBuffer(&vt, sizeof(vt));
            return;
        }
    }

    /* if it got here, the value couldn't be handled */
    throw Exception{"Invalid type"};
}

template <typename NUMERIC>
```



```
NUMERIC IO::Buffer::extract() {
    NUMERIC vt;
    memcpy(&vt, this->data + this->offset, sizeof(NUMERIC));

    if (this->dataSize - this->offset < sizeof(NUMERIC)) {
        throw Exception{"out of buffer"};
    }

    switch (sizeof(NUMERIC)) {
        case 1: {
            this->offset += sizeof(NUMERIC);
            return vt;
        }
        case 2: {
            this->offset += sizeof(NUMERIC);
            return ntohs(vt);
        }
        case 4: {
            this->offset += sizeof(NUMERIC);
            return ntohl(vt);
        }
    }

    /* if it got here, the value couldn't be handled */
    throw Exception{"Invalid type"};
}

#endif
```

```cpp
#include "Camera.h"
#include <cmath>
#include <iostream>

/**
 * @brief Construct a new Camera with the given initial coordinates.
 *
 * @param window The window where the camera renders.
 * @param scale The pixel/meters relation.
 * @param width The width of the area where the camera can go.
 * @param width The height of the area where the camera can go.
 */
GUI::Camera::Camera(GUI::Window &window, float scale, float width, float height)
    : window(window),
      start(this->cur),
      dst(this->cur),
      scale(scale),
      renderer(window.getRenderer()),
      width(width),
      height(height) {
    this->setTo(Position{0, float(this->window.getHeight()) / this->scale / 2.0f
});
}

GUI::Camera::~Camera() {}

/**
 * @brief Whether the camera is currently moving to or not.
 */
bool GUI::Camera::isMoving() const {
    return (this->dst != this->cur);
}

/**
 * @brief Returns the internal renderer.
 *
 * @return Renderer.
 */
SDL_Renderer &GUI::Camera::getRenderer() const {
    return this->renderer;
}

/**
 * @brief Returns the pixel/meters scale.
 *
 * @return Scale.
 */
float GUI::Camera::getScale() const {
    return this->scale;
}

/**
 * @brief Gets the size of the screen in game coordinates.
 *
 * @return float Screen width.
 */
float GUI::Camera::screenWidth() const {
    return float(this->window.getWidth()) / this->scale;
}

/**
 * @brief Gets the size of the screen in game coordinates.
```



```cpp
 *
 * @return float Screen height.
 */
float GUI::Camera::screenHeight() const {
    return float(this->window.getHeight()) / this->scale;
}

/**
 * @brief Returns the camera position in global coordinates.
 *
 * @return Position Camera position.
 */
GUI::Position GUI::Camera::getPosition() const {
    Position offset{(this->window.getWidth() / 2) / this->scale,
                    -(this->window.getHeight() / 2) / this->scale};
    return this->cur + offset;
}

/**
 * @brief Instantly moves the camera to the given coordinates.
 *
 * @param coord Coordinates of the new camera position.
 */
void GUI::Camera::setTo(GUI::Position coord) {
    this->elapsed = 0.0f;

    coord = this->clamp(coord);

    coord.x -= (this->window.getWidth() / 2) / this->scale;
    coord.y += (this->window.getHeight() / 2) / this->scale;
    this->dst = this->start = this->cur = coord;
}

/**
 * @brief Smoothly moves the camera to the given coordinates.
 *
 * @param coord Coordinates of the final camera position.
 */
void GUI::Camera::moveTo(GUI::Position coord) {
    coord = this->clamp(coord);

    coord.x -= (this->window.getWidth() / 2) / this->scale;
    coord.y += (this->window.getHeight() / 2) / this->scale;

    if (this->dst.distance(coord) < 7.0f) {
        this->dst = coord;
        return;
    }

    this->elapsed = 0.0f;
    this->dst = coord;
    this->start = this->cur;
}

/**
 * @brief Converts some global coordinates to screen coordinates.
 *
 * @param global Global coordinates.
 * @return Corresponding screen coordinates.
 */
GUI::ScreenPosition GUI::Camera::globalToScreen(GUI::Position global) {
    /* converts from global to local/camera coordinates */
```

```
    Position local = (global - this->cur);
    local.y *= -1;

    /* calculates the screen coordinates */
    return ScreenPosition{int(local.x * this->scale), int(local.y * this->scale)
};
}

/**
 * @brief Converts some screen coordinates to global coordinates.
 *
 * @param global Screen coordinates.
 * @return Corresponding global coordinates.
 */
GUI::Position GUI::Camera::screenToGlobal(GUI::ScreenPosition screen) {
    /* converts from screen to global coordinate */
    Position global = {screen.x / this->scale, screen.y / this->scale};
    global.y *= -1;
    global += this->cur;
    return global;
}

/**
 * @brief Updates the camera according to the elapsed time since the last update
.
 *
 * @param dt Seconds elapsed since the last update.
 */
void GUI::Camera::update(float dt) {
    if (this->dst == this->cur) {
        return;
    }

    /* total animation duration in seconds */
    const float duration = 5.0f;

    this->elapsed += dt;

    /* if the distance is less than the threshold, then it's set immediately */
    if (this->elapsed > duration || this->cur.distance(this->dst) < 0.05) {
        this->cur = this->dst;
        return;
    }

    /* calculates the new position */
    const float framerate = duration / 60.0f;
    float scale = 1.0f - 1.0f / std::pow(2, this->elapsed / framerate);
    this->cur = this->start + (this->dst - this->start) * scale;
}

void GUI::Camera::draw(const Texture &texture, Position p) {
    SDL_Rect clip = {0, 0, texture.getWidth(), texture.getHeight()};
    this->draw(texture, p, clip);
}

/**
 * @brief Draws a texture in the screen.
 *
 * @param texture The texture to render.
 * @param p Position where to draw the texture (global coordinates).
 * @param clip Portion of the texture to render.
 */
```



```
void GUI::Camera::draw(const Texture &texture, Position p, const SDL_Rect &clip)
 {
    this->draw(texture, p, clip, SDL_FLIP_NONE);
}

/**
 * @brief Draws a texture in the screen specifying the flip.
 *
 * @param texture The texture to render.
 * @param p Position where to draw the texture (global coordinates).
 * @param clip Portion of the texture to render.
 * @param flip Flip mode.
 * @param scale Scale factor of the final rendered texture size.
 */
void GUI::Camera::draw(const Texture &texture, Position p, const SDL_Rect &clip,
                       SDL_RendererFlip flip, float scale) {
    /* converts from global to local/camera coordinates */
    Position local = (p - this->cur);
    local.y *= -1;

    /* calculates the screen coordinates */
    ScreenPosition screen_local{int(local.x * this->scale), int(local.y * this->
scale)};

    /* draws in screen coordinates */
    this->drawLocal(texture, screen_local, clip, flip, scale);
}

/**
 * @brief Draws a texture in the screen in camera/screen coordinates.
 * The texture is fully rendered based in it's dimensions.
 *
 * @param texture Texture to draw.
 * @param p Position where to draw it.
 */
void GUI::Camera::drawLocal(const Texture &texture, ScreenPosition p) {
    SDL_Rect clip = {0, 0, texture.getWidth(), texture.getHeight()};
    this->drawLocal(texture, p, clip);
}

/**
 * @brief Draws a texture in the screen in camera/screen coordinates.
 *
 * @param texture The texture to render.
 * @param p Position where to draw the texture (camera coordinates).
 * @param clip Portion of the texture to render.
 */
void GUI::Camera::drawLocal(const Texture &texture, ScreenPosition p, const SDL_
Rect &clip) {
    this->drawLocal(texture, p, clip, SDL_FLIP_NONE);
}

/**
 * @brief Draws a texture in the screen in camera/screen coordinates specifying
the flip.
 *
 * @param texture The texture to render.
 * @param p Position where to draw the texture (camera coordinates).
 * @param clip Portion of the texture to render.
 * @param flip Flip mode.
 * @param scale Scale factor of the final texture size.
 */
```

```
void GUI::Camera::drawLocal(const Texture &texture, ScreenPosition p, const SDL_
Rect &clip,
                            SDL_RendererFlip flip, float scale) {
    /* calculates the scaled size */
    int w = int(clip.w * scale);
    int h = int(clip.h * scale);

    SDL_Rect dst = {};
    dst.x = p.x - w / 2;
    dst.y = p.y - h / 2;
    dst.w = w;
    dst.h = h;

    SDL_RenderCopyEx(&this->renderer, texture.get(), &clip, &dst, 0, nullptr, fl
ip);
}

/**
 * @brief Clamps a given position to the camera limits.
 *
 * @param p Position to clamp.
 * @return Position Clamped position.
 */
GUI::Position GUI::Camera::clamp(Position p) const {
    float wWidth = this->window.getWidth() / this->scale;
    float wHeight = this->window.getHeight() / this->scale;

    /* avoids the camera from going way below the zero or above the level limits
 */
    p.y = std::max(p.y, wHeight / 2 - 10.0f);
    p.y = std::min(p.y, this->height - wHeight / 2.0f);

    /* avoids the camera from going out of the level limits in the X axis */
    p.x = std::max(p.x, (-this->width / 2.0f) + wWidth / 2.0f);
    p.x = std::min(p.x, (this->width / 2.0f) - wWidth / 2.0f);

    return p;
}

/**
 * @brief Draws a sdl rect in the screen in camera/screen coordinates specifying
 the flip.
 *
 * @param p Position where to draw the texture (camera coordinates).
 * @param clip sdl rect to render.
 * @param scale Scale factor of the final sdl rect size.
 */
void GUI::Camera::drawLocal(ScreenPosition p, const SDL_Rect &clip, SDL_Color co
lor) {
    /* calculates the scaled size */
    float scale = this->scale;
    int w = int(clip.w * scale);
    int h = int(clip.h * scale);

    SDL_Rect dst = {};
    dst.x = p.x - w / 2;
    dst.y = p.y - h / 2;
    dst.w = w;
    dst.h = h;

    SDL_SetRenderDrawColor(&this->renderer, color.r, color.g, color.b, 0xFF);
    SDL_RenderFillRect(&this->renderer, &dst);
```



```
}
```

```
#ifndef CAMERA_H_
#define CAMERA_H_

#include <SDL2/SDL.h>

#include "Point.h"
#include "Texture.h"
#include "Window.h"

namespace GUI {
using Position = Math::Point<float>;
using ScreenPosition = Math::Point<int>;

class Camera {
   public:
    Camera(GUI::Window &window, float scale, float width, float height);
    ~Camera();

    bool isMoving() const;

    void setTo(Position coords);
    void moveTo(Position coords);

    ScreenPosition globalToScreen(Position);
    Position screenToGlobal(ScreenPosition);

    float getScale() const;
    Position getPosition() const;
    SDL_Renderer &getRenderer() const;
    float screenWidth() const;
    float screenHeight() const;

    void draw(const Texture &texture, Position p);
    void draw(const Texture &texture, Position p, const SDL_Rect &clip);
    void draw(const Texture &texture, Position p, const SDL_Rect &clip, SDL_Rend
ererFlip flip,
              float scale = 1);
    void drawLocal(const Texture &texture, ScreenPosition p);
    void drawLocal(const Texture &texture, ScreenPosition p, const SDL_Rect &cli
p);
    void drawLocal(const Texture &texture, ScreenPosition p, const SDL_Rect &cli
p,
                   SDL_RendererFlip flip, float scale = 1);
    void drawLocal(ScreenPosition p, const SDL_Rect &clip, SDL_Color color);

    Position clamp(Position p) const;

    void update(float dt);

   private:
    GUI::Window &window;
    /* current camera coordinates. */
    Position cur{0, 0};
    /* the position that the camera is moving towards to and the one that it sta
rted moving from. */
    Position start, dst;
    /* Distance scale factor. */
    float scale;
    /* elapsed time accumulator. */
    float elapsed{0};
    /* The SDL renderer. */
    SDL_Renderer &renderer;
```



```
    float width, height;
};
}  // namespace GUI

#endif
```

```cpp
#include "Chronometer.h"

Utils::Chronometer::Chronometer() {
    this->prev = std::chrono::high_resolution_clock::now();
}

Utils::Chronometer::~Chronometer() {}

/**
 * @brief Returns the number of milliseconds elapsed since the last call to this
 function.
 *
 * @return double Milliseconds elapsed.
 */
double Utils::Chronometer::elapsed() {
    std::chrono::high_resolution_clock::time_point current =
        std::chrono::high_resolution_clock::now();

    double dt = std::chrono::duration_cast<std::chrono::duration<double>>(curren
t - prev).count();
    this->prev = current;
    return dt;
}
```

```
#include <chrono>

namespace Utils {
class Chronometer {
   public:
    Chronometer();
    ~Chronometer();

    double elapsed();

   private:
    std::chrono::high_resolution_clock::time_point prev;
};
}  // namespace Utils
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 17/05/18.
 */

#ifndef __COLOR_H__
#define __COLOR_H__

#include <SDL2/SDL_stdinc.h>
namespace GUI {
struct Color {
    Uint8 r, g, b;
};
}  // namespace GUI

#endif  //__COLOR_H__
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#include <errno.h>
#include <string.h>
#include <sys/socket.h>

#include "CommunicationSocket.h"
#include "ErrorMessages.h"
#include "Exception.h"

CommunicationSocket::CommunicationSocket(int fd) : Socket(fd) {}

unsigned int CommunicationSocket::send(const char *buffer, unsigned int length)
{
    unsigned int sent = 0;
    int status = 0;

    while (sent < length) {
        status = ::send(this->fd, &buffer[sent], length - sent, MSG_NOSIGNAL);
        /*
         * Inspirado en el ejemplo del ejemplo de ayuda echoserver
         * status == 0: cerraron el socket, lanzo una exepcion de socket cerrado
         * para que no termine correctamente el servidor.
         * status < 0:  hubo un error
         * status > 0:  se enviaron status bytes
         */
        if (status == 0) {
            throw Exception(ERR_MSG_SOCKET_CLOSED);
        } else if (status < 0) {
            throw Exception(strerror(errno));
        } else {
            sent += status;
        }
    }

    return sent;
}

unsigned int CommunicationSocket::receive(char *buffer, unsigned int length) {
    unsigned int received = 0;
    int status = 0;

    while (received < length) {
        status = ::recv(this->fd, &buffer[received], length - received, MSG_NOSI
GNAL);
        /*
         * Inspirado en el ejemplo del ejemplo de ayuda echoserver
         * status == 0: cerraron el socket, lanzo una exepcion de socket cerrado
         * para que no termine correctamente el servidor.
         * status < 0:  hubo un error
         * status > 0:  se enviaron status bytes
         */
        if (status == 0) {
            throw Exception(ERR_MSG_SOCKET_CLOSED);
        } else if (status < 0) {
            throw Exception(strerror(errno));
        } else {
            received += status;
        }
```



```
    }
    return received;
}
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#ifndef __COMMUNICATIONSOCKET_H__
#define __COMMUNICATIONSOCKET_H__

#include "Socket.h"
/**
 * Clase que se usa directamente en el servidor para comunicarse con el cliente.
 * Tiene la posibilidad de enviar y recibir mensajes, y es devuelta por
 * movimiento cuando se acepta una conexion.
 *
 * 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.
 */
class CommunicationSocket : public Socket {
protected:
    CommunicationSocket() = default;

public:
    explicit CommunicationSocket(int fd);
    /**
     * Envia length bytes, contenidos en el buffer. Si la conexion falla, se
     * lanza una Exception.Retorna la cantidad de bytes eviados.
     * @param buffer
     * @param length
     * @return
     */
    unsigned int send(const char *buffer, unsigned int length);
    /**
     * Recibe length bytes y los guarda en el buffer. Si la conexion falla, se
     * lanza una Exception. Retorna la cantidad de bytes recibidos.
     * @param buffer
     * @param length
     * @return
     */
    unsigned int receive(char *buffer, unsigned int length);
};

#endif  //__COMMUNICATIONSOCKET_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 04/06/18
 */

#ifndef __DIRECTION_H__
#define __DIRECTION_H__

namespace Worm {
enum class Direction { right, left };
}

#endif  //__DIRECTION_H__
```

```
#ifndef DOUBLEBUFFER_H_
#define DOUBLEBUFFER_H_

#include <condition_variable>
#include <mutex>
#include "IO.h"

namespace IO {
template <typename T>
class DoubleBuffer {
   public:
    DoubleBuffer() {}
    ~DoubleBuffer() {}

    void swap();
    void set(const T& instance);
    T get(bool waitNew = false);

    void interrupt();

   private:
    bool hasData{false};
    bool interrupted{false};
    T copies[2];
    int current{0};
    std::mutex mutex;
    std::condition_variable dataSet;
};
}  // namespace IO

/**
 * @brief Swaps the background copy and the current copy.
 */
template <typename T>
void IO::DoubleBuffer<T>::swap() {
    std::unique_lock<std::mutex> lock(this->mutex);
    this->hasData = true;
    this->current = !this->current;
    this->dataSet.notify_all();
}

/**
 * @brief Sets the background copy to the given value.
 *
 * @tparam T Instance type.
 * @param instance The new copy.
 */
template <typename T>
void IO::DoubleBuffer<T>::set(const T& instance) {
    std::unique_lock<std::mutex> lock(this->mutex);
    /* sets the new value into the hidden instance */
    this->copies[!this->current] = instance;
}

/**
 * @brief Gets the current copy.
 *
 * @param waitNew if true, waits until a swap was done from the last time this f
unction was called.
 *                if false, returns the copy eitherway.
 * @return The current copy.
 */
```



```
template <typename T>
T IO::DoubleBuffer<T>::get(bool waitNew) {
    std::unique_lock<std::mutex> lock(this->mutex);
    while (!this->hasData && !this->interrupted) {
        this->dataSet.wait(lock);
    }

    if (this->interrupted) {
        throw IO::Interrupted{};
    }

    if (waitNew) {
        this->hasData = false;
    }

    /* returns a copy of the visible instance */
    return this->copies[this->current];
}

/**
 * @brief Interrupts all the threads waiting in "get" launching an exception.
 *        Useful to unblock all the observers.
 */
template <typename T>
void IO::DoubleBuffer<T>::interrupt() {
    this->interrupted = true;
    this->dataSet.notify_all();
}

#endif
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#ifndef __ERROR_MESSAGES_H__
#define __ERROR_MESSAGES_H__

#define ERR_MSG_SOCKET_INVALID_PORT "Puerto %s invÃ¡lido: %s."
#define ERR_MSG_SOCKET_BINDING "No se pudo bindear el socket al puerto %s:"
#define ERR_MSG_SOCKET_LISTEN                                   \
    "No se pudo establecer la cantidad de conexiones" \
    "en espera: %s."
#define ERR_MSG_SOCKET_INVALID_HOST_OR_PORT "Host %s o puerto %s invÃ¡lido: %s."
#define ERR_MSG_CONNECTION_COULD_NOT_BE_STABLISHED \
    "La conexiÃ³n a %s:%s no "                          \
    "pudo ser establecida."
#define ERR_MSG_SOCKET_ACCEPT "No se pudo aceptar la conexiÃ³n: %s"
#define ERR_MSG_SOCKET_CLOSED "ConexiÃ³n cerrada."
#define ERR_MSG_SOCKET_CLOSED_UNEXPECTLY \
    "Error: terminaciÃ³n inesperada al " \
    "recibir un mensaje."

#endif  //__ERROR_MESSAGES_H__
```

```cpp
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#include <cstdarg>
#include <sstream>
#include <string>

#include "Exception.h"
/**
 * Se adapto la clase OSError, usando funciones estandar de C++11, para hacer
 * una clase genÃ©rica Exception
 * @param fmt = Formato al cual completar
 * @param ... = argumentos para completar fmt
 */
Exception::Exception(const char *fmt, ...) noexcept {
    va_list args, args_copy;
    va_start(args, fmt);
    va_copy(args_copy, args);

    try {
        std::size_t size = std::vsnprintf(nullptr, 0, fmt, args) + 1;

        this->msg_error.reserve(size);
        std::vsnprintf(&this->msg_error.front(), size, fmt, args_copy);

        va_end(args_copy);
        va_end(args);
    } catch (...) {
        va_end(args_copy);
        va_end(args);
    }
}

const char *Exception::what() const noexcept {
    return this->msg_error.c_str();
}
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#ifndef __Exception_H__
#define __Exception_H__

#include <exception>
#include <string>
/**
 * Clase que se inspira de las filminas de la catedra de exception.
 * recibe en su constructor un string con el formato (al estilo del formato
 * que recibe printf), y una cantidad indefinida de parÃ¡metros adicionales,
 * para agregarselos al esqueleto de fmt.
 */
class Exception : public std::exception {
   private:
    std::string msg_error;

   public:
    explicit Exception(const char* fmt, ...) noexcept;
    Exception() = delete;
    virtual const char* what() const noexcept;
    virtual ~Exception() noexcept = default;
};

#endif  //__Exception_H__
```

```cpp
#include "Font.h"
#include "Exception.h"

GUI::Font::Font(const std::string &file, int size) : size(size) {
    this->font = TTF_OpenFont(file.c_str(), size);
    if (!this->font) {
        throw Exception{"Failed loading the font: %s", TTF_GetError()};
    }
}

GUI::Font::~Font() {
    TTF_CloseFont(this->font);
}

/**
 * @brief Gets the SDL font pointer.
 */
TTF_Font *GUI::Font::get() {
    return this->font;
}
```

```
#ifndef FONT_H_
#define FONT_H_

#include <SDL2/SDL_ttf.h>
#include <string>

namespace GUI {
class Font {
   public:
    const int size;

    Font(const std::string &file, int size);
    ~Font();

    TTF_Font *get();

   private:
    TTF_Font *font;
};

}  // namespace GUI

#endif
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 17/05/18.
 */
#ifndef __GAME_STATE_MSG_H__
#define __GAME_STATE_MSG_H__

#define WORMS_QUANTITY 20
#define BULLETS_QUANTITY 7
#define TOTAL_TEAM_QUANTITY 5
#define WEAPONS_QUANTITY 10

#define POWER_CHARGE_TIME 5.0f

#define COMMAND_GET_LEVELS 0
#define COMMAND_JOIN_GAME 1
#define COMMAND_GET_GAMES 2
#define COMMAND_CREATE_GAME 3
#define COMMAND_QUIT 5

#include "Buffer.h"
#include <netinet/in.h>
#include <stdint.h>
#include <yaml-cpp/node/node.h>
#include <yaml-cpp/yaml.h>
#include <cstring>
#include <vector>
#include <cstdint>
#include <stdint.h>

#include "Direction.h"
#include "Exception.h"
#include "Point.h"

namespace Worm {
enum class StateID {
    Walk,
    Still,
    StartJump,
    Jumping,
    EndJump,
    StartBackFlip,
    BackFlipping,
    EndBackFlip,
    Hit,
    Die,
    Dead,
    Drowning,
    Falling,
    Land,
    Sliding,
    Teleporting,
    Teleported,
    Batting
};
enum WeaponID {
    WNone,
    WBazooka,
    WGrenade,
    WCluster,
    WMortar,
    WBanana,
```



```
    WHoly,
    WExplode,
    WFragment,
    WAerial,
    WDynamite,
    WTeleport,
    WBaseballBat
};
}  // namespace Worm

namespace IO {
    enum class ClientGUIInput {
        startCreateGame,
        startJoinGame,
        quit,
        levelSelected,
        joinGame
    };
    enum class ServerResponseAction {
        startGame,
        levelsInfo,
        playerConnected,
        gamesInfo,
        serverClosed
    };

    struct ClientGUIMsg {
        ClientGUIMsg() = default;
        explicit ClientGUIMsg(ClientGUIInput input) :
                input(input) {};
        ClientGUIInput input;
    };
    struct ServerResponse {
        ServerResponseAction action;
    };

    //TODO put this in dedicated file.
    enum class ServerInternalAction {
        lobbyFinished,
        quit
    };

    struct ServerInternalMsg {
        ServerInternalAction action;
    };

    struct LevelInfo {
        uint8_t id;
        std::string name;
        uint8_t playersQuantity;
    };

    struct GameInfo {
        uint8_t gameID;
        uint8_t levelID;
        std::string levelName;
        uint8_t numCurrentPlayers;
        uint8_t numTotalPlayers;
    };

    struct LevelData {
        std::string levelPath;
```

```
        std::string levelName;
        std::vector<std::string> backgroundPath;
        std::vector<std::string> backgroundName;
    };

    struct LevelsInfo : public ServerResponse {
        LevelsInfo(ServerResponseAction action, std::vector<LevelInfo> &levelsIn
fo) :
                action(action),
                levelsInfo(std::move(levelsInfo)) {}
        ServerResponseAction action;
        std::vector<LevelInfo> levelsInfo;
    };
    struct LevelSelected : public ClientGUIMsg {
        LevelSelected(ClientGUIInput input, unsigned int levelSelected) :
                ClientGUIMsg(input),
                levelSelected(levelSelected) {};
        unsigned int levelSelected;
    };
enum class PlayerInput {
    moveNone,
    moveRight,
    moveLeft,
    startJump,
    stopMove,
    startBackFlip,
    bazooka,
    pointUp,
    pointDown,
    startShot,
    endShot,
    grenade,
    cluster,
    mortar,
    banana,
    holy,
    timeout1,
    timeout2,
    timeout3,
    timeout4,
    timeout5,
    positionSelected,
    aerialAttack,
    dynamite,
    teleport,
    baseballBat,
    disconnected
};

struct PlayerMsg {
    PlayerInput input;
    Math::Point<float> position{0.0f, 0.0f};

    std::string serialize() {
        IO::Buffer buffer;

        buffer.append(static_cast<uint8_t>(this->input));
        buffer.appendFloat(this->position.x);
        buffer.appendFloat(this->position.y);

        return buffer.asString();
    }
```



```
    void deserialize(const std::string &data) {
        IO::Buffer buffer{data};

        this->input = static_cast<PlayerInput>(buffer.extract<uint8_t>());
        this->position.x = buffer.extractFloat();
        this->position.y = buffer.extractFloat();
    }
};


struct GameStateMsg {
    std::uint16_t elapsedTurnSeconds;
    std::int8_t windIntensity;
    std::uint8_t currentWorm;
    std::uint8_t currentWormToFollow;
    std::uint8_t currentTeam;
    std::uint8_t num_worms;
    std::uint8_t num_teams;
    std::uint8_t wormsTeam[WORMS_QUANTITY];
    Worm::Direction wormsDirection[WORMS_QUANTITY];
    std::uint16_t wormsHealth[WORMS_QUANTITY];
    std::uint32_t teamHealths[TOTAL_TEAM_QUANTITY];
    float positions[WORMS_QUANTITY * 2];
    Worm::StateID stateIDs[WORMS_QUANTITY];
    Worm::WeaponID activePlayerWeapon;
    float activePlayerAngle;
    uint8_t bulletsQuantity;
    float bullets[2 * BULLETS_QUANTITY];
    float bulletsAngle[BULLETS_QUANTITY];
    Worm::WeaponID bulletType[BULLETS_QUANTITY];
    std::int16_t weaponAmmunition[WEAPONS_QUANTITY];

    bool processingInputs;
    std::uint16_t currentPlayerTurnTime;
    bool gameEnded;
    std::uint8_t winner;
    bool playerUsedTool;
    bool waitingForNextTurn;

    std::string serialize() {
        IO::Buffer buffer;

        buffer.append(this->elapsedTurnSeconds);
        buffer.append(windIntensity);
        buffer.append(currentWorm);
        buffer.append(currentWormToFollow);
        buffer.append(currentTeam);
        buffer.append(num_worms);
        buffer.append(num_teams);

        for(std::size_t i = 0; i < this->num_worms; i++) {
            buffer.append(this->wormsTeam[i]);
            buffer.append(static_cast<uint8_t>(this->wormsDirection[i]));
            buffer.append(this->wormsHealth[i]);
            buffer.appendFloat(this->positions[i * 2]);
            buffer.appendFloat(this->positions[i * 2 + 1]);
            buffer.append(static_cast<uint8_t>(this->stateIDs[i]));
        }

        for(std::size_t i = 0; i < this->num_teams; i++) {
            buffer.append(this->teamHealths[i]);
```

```
        }

        buffer.append(static_cast<uint8_t>(this->activePlayerWeapon));
        buffer.appendFloat(this->activePlayerAngle);
        buffer.append(this->bulletsQuantity);

        for(std::size_t i = 0; i < this->bulletsQuantity; i++) {
            buffer.appendFloat(this->bullets[i * 2]);
            buffer.appendFloat(this->bullets[i * 2 + 1]);
            buffer.appendFloat(this->bulletsAngle[i]);
            buffer.append(static_cast<uint8_t>(this->bulletType[i]));
        }

        for(std::size_t i = 0; i < WEAPONS_QUANTITY; i++) {
            buffer.append(weaponAmmunition[i]);
        }

        buffer.append(static_cast<uint8_t>(this->processingInputs));
        buffer.append(this->currentPlayerTurnTime);
        buffer.append(static_cast<uint8_t>(this->gameEnded));
        buffer.append(this->winner);
        buffer.append(static_cast<uint8_t>(this->playerUsedTool));
        buffer.append(static_cast<uint8_t>(this->waitingForNextTurn));

        return buffer.asString();
    }

    void deserialize(const std::string &data) {
        IO::Buffer buffer{data};

        this->elapsedTurnSeconds = buffer.extract<uint16_t>();
        this->windIntensity = buffer.extract<uint8_t>();
        this->currentWorm = buffer.extract<uint8_t>();
        this->currentWormToFollow = buffer.extract<uint8_t>();
        this->currentTeam = buffer.extract<uint8_t>();
        this->num_worms = buffer.extract<uint8_t>();
        this->num_teams = buffer.extract<uint8_t>();

        for(std::size_t i = 0; i < this->num_worms; i++) {
            this->wormsTeam[i] = buffer.extract<uint8_t>();
            this->wormsDirection[i] = static_cast<Worm::Direction>(buffer.extrac
t<uint8_t>());
            this->wormsHealth[i] = buffer.extract<uint16_t>();
            this->positions[i * 2] = buffer.extractFloat();
            this->positions[i * 2 + 1] = buffer.extractFloat();
            this->stateIDs[i] = static_cast<Worm::StateID>(buffer.extract<uint8_
t>());
        }

        for(std::size_t i = 0; i < this->num_teams; i++) {
            this->teamHealths[i] = buffer.extract<uint32_t>();
        }

        this->activePlayerWeapon = static_cast<Worm::WeaponID>(buffer.extract<ui
nt8_t>());
        this->activePlayerAngle = buffer.extractFloat();
        this->bulletsQuantity = buffer.extract<uint8_t>();

        for(std::size_t i = 0; i < this->bulletsQuantity; i++) {
            this->bullets[i * 2] = buffer.extractFloat();
            this->bullets[i * 2 + 1] = buffer.extractFloat();
            this->bulletsAngle[i] = buffer.extractFloat();
```



```
            this->bulletType[i] = static_cast<Worm::WeaponID>(buffer.extract<uin
t8_t>());
        }

        for(std::size_t i = 0; i < WEAPONS_QUANTITY; i++) {
            this->weaponAmmunition[i] = buffer.extract<uint16_t>();
        }

        this->processingInputs = static_cast<bool>(buffer.extract<uint8_t>());
        this->currentPlayerTurnTime = buffer.extract<uint16_t>();
        this->gameEnded = static_cast<bool>(buffer.extract<uint8_t>());
        this->winner = buffer.extract<uint8_t>();
        this->playerUsedTool = static_cast<bool>(buffer.extract<uint8_t>());
        this->waitingForNextTurn = static_cast<bool>(buffer.extract<uint8_t>());
    }
};
}  // namespace IO

#endif  //__GAME_STATE_MSG_H__
```

```
#ifndef IO_H_
#define IO_H_

#include <exception>

namespace IO {
/**
 * @brief Interrupt exception.
 */
class Interrupted : public std::exception {
   public:
    Interrupted() = default;
    ~Interrupted() = default;
};
}  // namespace IO

#endif
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 06/06/18
 */

#ifndef __OBSERVER_H__
#define __OBSERVER_H__

enum class Event {
    Explode,
    OnExplode,
    Shot,
    Drowning,
    Drowned,
    EndTurn,
    Hit,
    EndHit,
    NewWormToFollow,
    ImpactEnd,
    TurnEnded,
    WormFalling,
    WormLanded,
    Dead,
    Dying,
    DamageOnLanding,
    NextTurn,
    Teleported,
    P2PWeaponUsed,
    StartGame,
    NewPlayer,
    EndGame,
    CreateGame,
    JoinGame,
    LevelSelected,
    ConnectionToServer,
    LobbyToJoinSelected,
    DyingDueToDisconnection,
    DeadDueToDisconnection
};

class Subject;

class Observer {
   public:
    virtual ~Observer() = default;
    virtual void onNotify(Subject &subject, Event event) = 0;
};

#endif  //__OBSERVER_H__
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 17/05/18.
 */

#ifndef __POINT_H__
#define __POINT_H__

#define PI 3.14159265

#include <math.h>

// TODO change template for fixed class with float?
namespace Math {
template <typename Numeric>
class Point {
   public:
    Point(Numeric x, Numeric y) : x(x), y(y) {}
    ~Point() {}

    Point<Numeric> &operator=(const Point<Numeric> &other) {
        this->x = other.x;
        this->y = other.y;
        return *this;
    }

    Point<Numeric> &operator-=(const Point<Numeric> &other) {
        this->x -= other.x;
        this->y -= other.y;
        return *this;
    }

    Point<Numeric> &operator+=(const Point<Numeric> &other) {
        this->x += other.x;
        this->y += other.y;
        return *this;
    }

    double distance(const Point<Numeric> &other) const {
        Point<Numeric> dist = *this - other;
        return sqrt(dist.x * dist.x + dist.y * dist.y);
    }

    friend Point<Numeric> operator/(const Point<Numeric> &p, Numeric n) {
        return Point<Numeric>{p.x / n, p.y / n};
    }

    friend Point<Numeric> operator*(const Point<Numeric> &p, Numeric n) {
        return Point<Numeric>{p.x * n, p.y * n};
    }

    friend bool operator==(const Point<Numeric> &a, const Point<Numeric> &b) {
        return ((a.x == b.x) && (a.y == b.y));
    }

    friend bool operator!=(const Point<Numeric> &a, const Point<Numeric> &b) {
        return !(a == b);
    }

    friend Point<Numeric> operator+(Point<Numeric> a, const Point<Numeric> &b) {
        /* 'a' is passed by value! */
        a += b;
```



```
        return a;
    }

    friend Point<Numeric> operator-(Point<Numeric> a, const Point<Numeric> &b) {
        /* 'a' is passed by value! */
        a -= b;
        return a;
    }

    Numeric x, y;
};
}  // namespace Math

#endif  //__POINT_H__
```

```
//
// Created by rodrigo on 15/06/18.
//

#ifndef INC_4_WORMS_PROTOCOL_H
#define INC_4_WORMS_PROTOCOL_H


#include <fstream>
#include <netinet/in.h>
#include <string>
#include <vector>

#include "GameStateMsg.h"

#define COMMAND_LENGTH 1
#define INT_LENGTH 4
#define FILE_CHUNK_LENGTH 512

template <typename SOCKET>
class Protocol {
private:
    SOCKET socket;

public:
    explicit Protocol(SOCKET &socket) : socket(std::move(socket)) {};

//    /* Constructor por movimiento
//     */
//    Protocol(Protocol &&protocol);

//    /* Termina la comunicaciÃ³n, cerrando el socket.
//     */
//    void stopCommunication();

    /* El tipo char en este protocolo estÃ¡ asociado a los comandos.
     * En este caso se envÃa un comando.
     */
    void operator<<(unsigned char command){
        this->socket.send((const char *) &command, sizeof(command));
    };

    /* EnvÃa un entero de 4 bytes sin signo en big endian.
     */
    void operator<<(uint32_t i){
        uint32_t networkInt = htonl(i);
        this->socket.send((char *) &networkInt, sizeof(uint32_t));
    };

    /* EnvÃa un unsigned long tratÃ¡ndolo como un entero de 4 bytes
     * sin signo en big endian.
     */
    void operator<<(unsigned long i){
        *this << (unsigned int) i;
    };

    /* EnvÃa una cadena de caracteres sin el caracter de fin de cadena,
     * primero enviando su longitud como un entero con el mÃ©todo ya definido.
     */
    void operator<<(const std::string &string){
        *this << (unsigned int) string.length();
        this->socket.send(string.c_str(), string.length());
```



```
    };

    void operator<<(const IO::LevelInfo &levelInfo){
        *this << levelInfo.id;
        *this << levelInfo.name;
        *this << levelInfo.playersQuantity;
    };

    void operator<<(const IO::GameInfo &info) {
        *this << info.gameID;
        *this << info.levelID;
        *this << info.levelName;
        *this << info.numCurrentPlayers;
        *this << info.numTotalPlayers;
    };

    template <typename T> void operator<<(const std::vector<T> &vec) {
        *this << vec.size();
        for (const auto &elem : vec) {
            *this << elem;
        }
    }

    void operator<<(std::ifstream &file) {
        std::vector<char> chunk(FILE_CHUNK_LENGTH);

        file.seekg(0, file.end);
        uint32_t length = file.tellg();
        file.seekg(0, file.beg);

        *this << length;

        uint32_t charactersToRead = length;

        while (charactersToRead > 0) {
            file.read(chunk.data(), FILE_CHUNK_LENGTH);
            uint32_t charactersRead = file.gcount();
            this->socket.send(chunk.data(), charactersRead);
            charactersToRead -= charactersRead;
        }
    }


    /* Recibe un comando.
     */
    Protocol & operator>>(unsigned char &command){
        char buffer;
        this->socket.receive(&buffer, COMMAND_LENGTH);
        command = buffer;
        return *this;
    };

    /* Recibe un entero de 4 bytes sin signo en big endian.
     */
    Protocol & operator>>(unsigned int &i) {
        std::vector<char> buffer(INT_LENGTH);
        this->socket.receive(buffer.data(), INT_LENGTH);
        i = ntohl(*(unsigned int *) buffer.data());
        return *this;
    };

    /* Recibe una cadena de caracteras recibiendo primero su longitud
```

```
     * como un entero de la forma ya especificada y luego los caracteres
     * sin el caracter de fin de cadena.
     */
    Protocol & operator>>(std::string &string) {
        unsigned int length;
        *this >> length;
        std::vector<char> buffer(length);
        this->socket.receive(buffer.data(), length);
        std::string localString(buffer.data(), length);
        string = std::move(localString);
        return *this;
    };

    Protocol & operator>>(IO::LevelInfo &levelInfo) {
        *this >> levelInfo.id;
        *this >> levelInfo.name;
        *this >> levelInfo.playersQuantity;
        return *this;
    };

    Protocol &operator>>(IO::GameInfo &info){
        *this >> info.gameID;
        *this >> info.levelID;
        *this >> info.levelName;
        *this >> info.numCurrentPlayers;
        *this >> info.numTotalPlayers;
        return *this;
    };

    template <typename T> Protocol &operator>>(std::vector<T> &vec) {
        unsigned int elements{0};
        *this >> elements;
        for (unsigned int i = 0; i < elements; i++) {
            T elem;
            *this >> elem;
            vec.emplace_back(std::move(elem));
        }

        return *this;
    }

    Protocol &operator>>(std::ofstream &file) {
        std::vector<char> chunk(FILE_CHUNK_LENGTH);
        uint32_t fileLength;
        *this >> fileLength;
        uint32_t fileBytesWritten = 0;

        /* La cantidad de bytes a recibir es el mÃ-nimo entre los bytes
         * que quedan por recibir y el tamaÃ±o mÃ¡ximo de chunk.
         */
        while (fileBytesWritten < fileLength) {
            size_t bytesReceived = this->socket.receive(
                    chunk.data(),
                    std::min(fileLength - fileBytesWritten,
                             (uint32_t) FILE_CHUNK_LENGTH));
            file.write(chunk.data(), bytesReceived);
            fileBytesWritten += bytesReceived;
        }

        return *this;
    }
```



```
    /**
     * @brief returns socket by move semantic
     * @return
     */
    SOCKET getSocket() {
        this->socketRemoved = true;
        return std::move(this->socket);
    };

    void stopCommunication() {
        if (!this->socketRemoved) {
            this->socket.shutdown();
        }
    }

    bool socketRemoved{false};
};


#endif //INC_4_WORMS_PROTOCOL_H
```

```cpp
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#include <cstring>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

#include "ErrorMessages.h"
#include "Exception.h"
#include "Socket.h"


Socket::Socket() : fd(-1) {}

Socket::Socket(int fd) : fd(fd) {}

Socket::Socket(Socket &&other) noexcept : fd(other.fd) {
    other.fd = -1;
}

Socket &Socket::operator=(Socket &&other) noexcept {
    if (this != &other) {
        this->fd = other.fd;
        other.fd = -1;
    }
    return *this;
}

Socket::~Socket() {
    if (this->fd != -1) {
        this->close();
    }
}

void Socket::close() {
    ::close(this->fd);
    this->fd = -1;
}

void Socket::shutdown() {
    ::shutdown(this->fd, SHUT_RDWR);
}
```

```
/*
 * Created by Federico Manuel Gomez Peter on 02/05/2018.
 */

#ifndef __SOCKET_H__
#define __SOCKET_H__

class Socket {
   protected:
    int fd;
    /**
     * Constructor protegido, las clases hijas ClientSocket, ServerSocket y
     * DataSocket lo usaran para asignar el fd.
     */
    Socket();
    explicit Socket(int fd);
    void close();

   public:
    virtual ~Socket();
    /*
     * La superclase Socket se va a poder mover, pero no copiar.
     */
    Socket(Socket &&other) noexcept;
    Socket &operator=(Socket &&other) noexcept;
    Socket(Socket &other) = delete;
    Socket &operator=(Socket &other) = delete;
    /**
     *  Termina la comunicacion del socket.
     */
    void shutdown();
};

#endif  //__SOCKET_H__
```

```cpp
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#include "Stage.h"
#include <fstream>
#include "Exception.h"
#include "GameStateMsg.h"
#include "Point.h"
#include "yaml-cpp/yaml.h"

/**
 * @brief Parses a Point from a YAML node.
 *
 * @param node Node containing the data.
 * @return Point.
 */
static Math::Point<float> _parsePoint(YAML::Node &node) {
    if (!node.IsSequence() || node.size() != 2) {
        throw Exception{"Invalid stage: Point should be a sequence of 2 floats"};
    }

    float posX = node[0].as<float>();
    float posY = node[1].as<float>();
    return Math::Point<float>{posX, posY};
}

/**
 * @brief Parses a worm's data from a YAML node.
 *
 * @param health Worm's health.
 * @param node Node containing the data.
 * @return Worm data.
 */
static Worms::WormData _parseWorm(uint16_t health, YAML::Node &node) {
    if (node.Type() != YAML::NodeType::Map) {
        throw Exception{"Invalid stage: worm data expected as Map"};
    }

    if (!node["position"]) {
        throw Exception{"Invalid stage: worm requires 'position'"};
    }

    YAML::Node positionNode = node["position"];
    return Worms::WormData{health, _parsePoint(positionNode)};
}

/**
 * @brief Parses a girder's data from a YAML node.
 *
 * @param node Node containing the data.
 * @return Girder data.
 */
static Worms::GirderData _parseGirder(YAML::Node &node) {
    if (node.Type() != YAML::NodeType::Map) {
        throw Exception{"Invalid stage: girder data expected as Map"};
    }

    if (!node["position"]) {
        throw Exception{"Invalid stage: girder requires 'position'"};
    }
```



```cpp
    if (!node["angle"]) {
        throw Exception{"Invalid stage: girder requires 'angle'"};
    }

    YAML::Node positionNode = node["position"];
    Math::Point<float> position = _parsePoint(positionNode);

    float angle = node["angle"].as<float>();
    float length = node["length"].as<float>();

    return Worms::GirderData{length, 1.42f, angle, position};
}

/**
 * @brief Creates a Stage object populating it from the contents of the given fi
le.
 *
 * @param filename Name of the file with the Stage data.
 * @return Stage.
 */
Worms::Stage Worms::Stage::fromFile(const std::string &filename) {
    YAML::Node data = YAML::LoadFile(filename);

    if (!data["wormsHealth"]) {
        throw Exception{"Invalid stage: expected float 'wormsHealth'"};
    }

    uint16_t wormsHealth = data["wormsHealth"].as<unsigned short>();

    Stage stage;
    if (!data["width"] || !data["height"]) {
        throw Exception{"Invalid stage: expected stage 'width' and 'height'"};
    }
    stage.width = data["width"].as<float>();
    stage.height = data["height"].as<float>();

    if (!data["worms"] || !data["worms"].IsSequence()) {
        throw Exception{"Invalid stage: expected a sequence of 'worms'"};
    }

    if (!data["numPlayers"]) {
        throw Exception{"Invalid stage: expected integer 'numPlayers'"};
    }
    stage.numPlayers = data["numPlayers"].as<int>();

    /* loads the worms */
    for (std::size_t i = 0; i < data["worms"].size(); i++) {
        YAML::Node wormNode = data["worms"][i];
        stage.players.push_back(_parseWorm(wormsHealth, wormNode));
    }

    /* loads the girders */
    if (!data["girders"] || !data["girders"].IsSequence()) {
        throw Exception{"Invalid stage: expected a sequence of 'girders'"};
    }
    for (std::size_t i = 0; i < data["girders"].size(); i++) {
        YAML::Node girderNode = data["girders"][i];
        stage.girders.push_back(_parseGirder(girderNode));
    }

    /* loads the weapons ammo */
    YAML::Node weaponsNode;
```

```
    if (!data["weaponsAmmo"] || !data["weaponsAmmo"].IsMap()){
        weaponsNode = YAML::LoadFile(DEFAULT_WEAPON_CONFIG_PATH);
    } else{
        weaponsNode = data["weaponsAmmo"];
    }

    for (const auto &elem : weaponsNode) {
        auto weaponID = stage.weaponStrToID.at(elem.first.as<std::string>());
        int ammo = elem.second.as<uint16_t>();
        stage.ammunitionCounter.emplace(weaponID, ammo);
    }

    // -1 indicates infinite uses
    stage.ammunitionCounter.emplace(Worm::WNone, -1);

    /* background */
    if (data["background"]) {
        YAML::Node bg = data["background"];
        if (bg["color"] && bg["color"].IsSequence()) {
            stage.backgroundColor.r = bg["color"][0].as<int>();
            stage.backgroundColor.g = bg["color"][1].as<int>();
            stage.backgroundColor.b = bg["color"][2].as<int>();
        }
        if (bg["closeBackgroundFile"]) {
            stage.closerBackgroundFile = bg["closeBackgroundFile"].as<std::string>()
;
        }
        if (bg["midBackgroundFile"]) {
            stage.midBackgroundFile = bg["midBackgroundFile"].as<std::string>();
        }
        if (bg["fartherBackgroundFile"]) {
            stage.fartherBackgroundFile = bg["fartherBackgroundFile"].as<std::string>
();
        }
    }

    return stage;
}

Worms::Stage::Stage() {
    this->weaponStrToID.emplace("aerialAttack", Worm::WAerial);
    this->weaponStrToID.emplace("banana", Worm::WBanana);
    this->weaponStrToID.emplace("baseballBat", Worm::WBaseballBat);
    this->weaponStrToID.emplace("bazooka", Worm::WBazooka);
    this->weaponStrToID.emplace("cluster", Worm::WCluster);
    this->weaponStrToID.emplace("dynamite", Worm::WDynamite);
    this->weaponStrToID.emplace("grenade", Worm::WGrenade);
    this->weaponStrToID.emplace("holy", Worm::WHoly);
    this->weaponStrToID.emplace("mortar", Worm::WMortar);
    this->weaponStrToID.emplace("teleport", Worm::WTeleport);
    return;
}

const std::vector<Worms::WormData> &Worms::Stage::getWorms() const {
    return this->players;
}

const std::vector<Worms::GirderData> &Worms::Stage::getGirders() const {
    return this->girders;
}

float Worms::Stage::getHeight() const {
```



```
    return this->height;
}

float Worms::Stage::getWidth() const {
    return this->width;
}

const std::map<Worm::WeaponID, int16_t> &Worms::Stage::getAmmoCounter() const {
    return this->ammunitionCounter;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#ifndef __STAGE_H__
#define __STAGE_H__

#define DEFAULT_WEAPON_CONFIG_PATH "/etc/Worms/defaultWeapons.yaml"

#include <cstdint>
#include <map>
#include <string>
#include <unordered_map>
#include <vector>

#include "GameStateMsg.h"
#include "Point.h"

namespace Worms {
struct GirderData {
    float length;
    float height;
    float angle;
    Math::Point<float> pos;
};

struct WormData {
    uint16_t health;
    Math::Point<float> position;
};

struct Color {
    uint8_t r, g, b;
};

class Stage {
   public:
    static Stage fromFile(const std::string &filename);

    uint8_t turnTime{10};
    Color backgroundColor{255, 255, 255};
    std::string fartherBackgroundFile;
    std::string midBackgroundFile;
    std::string closerBackgroundFile;
    int numPlayers;

    Stage();
    ~Stage() = default;

    const std::vector<WormData> &getWorms() const;
    const std::vector<GirderData> &getGirders() const;

    float getHeight() const;
    float getWidth() const;
    const std::map<Worm::WeaponID, std::int16_t> &getAmmoCounter() const;

   private:
    std::vector<WormData> players;
    std::vector<GirderData> girders;
    std::unordered_map<std::string, Worm::WeaponID> weaponStrToID;
    std::map<Worm::WeaponID, std::int16_t> ammunitionCounter;
```



```
    float width{100.0f};
    float height{30.0f};
};
}  // namespace Worms

#endif  //__STAGE_H__
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 17/05/18.
 */

#ifndef __STREAM_H__
#define __STREAM_H__

#include <atomic>
#include <condition_variable>
#include <queue>

#include "Exception.h"

namespace IO {
template <typename Msg>
class Stream {
   public:
    Stream() {}
    Stream(Stream &&other) = default;
    Stream &operator=(Stream &&other) = default;
    Stream &operator=(Stream &other) = delete;

    ~Stream() {
        this->close();
    }

    void push(const Msg &m) {
        std::unique_lock<std::mutex> lock(this->mutex);
        this->q.push(m);
        this->notEmpty.notify_all();
    }

    bool pop(Msg &m, bool block = true) {
        std::unique_lock<std::mutex> lock(this->mutex);
        while (this->q.empty() && !this->closed) {
            if (!block) {
                return false;
            }
            this->notEmpty.wait(lock);
        }

        if (this->closed) {
            throw Exception{"closed"};
        }

        m = this->q.front();
        this->q.pop();
        return true;
    }

    Stream &operator<<(const Msg &m) {
        this->push(m);
        return *this;
    }

    Stream &operator>>(Msg &m) {
        this->pop(m);
        return *this;
    }

    void close() {
```



```
        if (this->closed) {
            return;
        }
        this->closed = true;
        this->notEmpty.notify_all();
    }

   private:
    std::queue<Msg> q;
    std::mutex mutex;
    std::condition_variable notEmpty;
    std::atomic<bool> closed{false};
};
}  // namespace IO
#endif  //__STREAM_H__
```

```cpp
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 06/06/18
 */

#include "Subject.h"

void Subject::addObserver(Observer *obs) {
    this->observers.emplace(obs);
}
// TODO check possibly raceCondition
void Subject::removeObserver(Observer *obs) {
    this->observers.erase(this->observers.find(obs));
}

void Subject::notify(Subject &subject, Event event) {
    for (auto &observer : this->observers) {
        observer->onNotify(subject, event);
    }
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 06/06/18
 */

#ifndef __Subject_H__
#define __Subject_H__

#include <set>

#include "Observer.h"

class Subject {
   public:
    Subject() = default;
    Subject(Subject &copy) = delete;
    virtual ~Subject() = default;
    void addObserver(Observer *obs);
    void removeObserver(Observer *obs);
    /**
     * Notify all observers with the Event id, so the Observer can do
     * what is necessary
     * @param subject
     * @param event
     */
    void notify(Subject &subject, Event event);

   protected:
    std::set<Observer *> observers;
    int numObservers;
};

#endif  //__Subject_H__
```

```cpp
#include "Text.h"
#include <random>
#include "Exception.h"

std::unordered_map<GUI::TextCacheKey, GUI::TexturePtr, GUI::TextKeyHash> GUI::Te
xt::cache;

GUI::Text::Text(GUI::Font& font) : font(font) {}

GUI::Text::~Text() {}

void GUI::Text::setBackground(SDL_Color color) {
    this->hasBackground = true;
    this->background = color;
    this->needs_render = true;
}

void GUI::Text::set(const std::string& text, SDL_Color color) {
    this->set(text, color, this->font.size);
}

void GUI::Text::set(const std::string& text, SDL_Color color, int size) {
    this->size = size;
    this->text = text;
    this->color = color;
    this->needs_render = true;
}

void GUI::Text::render(GUI::Position p, GUI::Camera& camera) {
    if (this->needs_render) {
        this->createTexture(&camera.getRenderer());
        this->needs_render = false;
    }

    float scale = float(this->size) / float(this->font.size);

    if (this->hasBackground) {
        GUI::ScreenPosition sp = camera.globalToScreen(p);

        SDL_Renderer& renderer = camera.getRenderer();
        SDL_SetRenderDrawColor(&renderer, this->background.r, this->background.g
,
                               this->background.b, 255);

        SDL_Rect r = {};
        r.x = sp.x - this->texture->getWidth() * scale / 2;
        r.y = sp.y - (this->texture->getHeight() * scale - 10) / 2;
        r.w = this->texture->getWidth() * scale;
        r.h = this->texture->getHeight() * scale - 10;

        SDL_RenderFillRect(&renderer, &r);
    }

    SDL_Rect clip = {0, 0, this->texture->getWidth(), this->texture->getHeight()
};
    camera.draw(*this->texture, p, clip, SDL_FLIP_NONE, scale);
}

/**
 * @brief Renders the text in a screen coordintates.
 *
 * @param p Position where the text will be rendered in screen coordintaes.
```



```cpp
 * @param camera Camera.
 */
void GUI::Text::renderFixed(GUI::ScreenPosition p, GUI::Camera& camera) {
    this->render(camera.screenToGlobal(p), camera);
}

/**
 * @brief Creates a texture for the current text.
 *
 * @param renderer Renderer.
 */
void GUI::Text::createTexture(SDL_Renderer* renderer) {
    /* checks if the texture is already in the cache */
    uint32_t color = 0 | this->color.r << 16 | this->color.g << 8 | this->color.
b;
    TextCacheKey key{this->text, color};
    if (GUI::Text::cache.find(key) != GUI::Text::cache.end()) {
        this->texture = GUI::Text::cache.at(key);
        return;
    }

    /* if the cache is full, removes a random entry */
    if (Text::cache.size() >= 150) {
        std::mt19937 rng;
        rng.seed(std::random_device()());
        std::uniform_int_distribution<std::mt19937::result_type> r(0, Text::cach
e.size() - 1);
        auto random_it = std::next(std::begin(Text::cache), r(rng));
        Text::cache.erase(random_it);
    }

    /* render text surface */
    SDL_Surface* surface = TTF_RenderText_Solid(this->font.get(), text.c_str(),
this->color);
    if (surface == NULL) {
        throw Exception{"Failed rendering text surface: %s", TTF_GetError()};
    }

    /* convert to a texture */
    SDL_Texture* texture = SDL_CreateTextureFromSurface(renderer, surface);
    if (texture == NULL) {
        SDL_FreeSurface(surface);
        throw Exception{"Failed create texture from rendered text: %s", SDL_GetError()};
    }

    GUI::Text::cache[key] = TexturePtr(new Texture{texture, surface->w, surface-
>h});
    this->texture = GUI::Text::cache[key];

    SDL_FreeSurface(surface);
}
```

```
#ifndef TEXT_H_
#define TEXT_H_

#include <SDL2/SDL.h>
#include <cstdint>
#include <functional>
#include <list>
#include <memory>
#include <string>
#include <unordered_map>
#include "Camera.h"
#include "Font.h"

namespace GUI {
using TextCacheKey = std::pair<std::string, uint32_t>;

struct TextKeyHash {
    size_t operator()(const TextCacheKey &p) const {
        return std::hash<std::string>{}(p.first) ^ p.second;
    }
};

using TexturePtr = std::shared_ptr<Texture>;

class Text {
   public:
    Text(Font &font);
    ~Text();

    void set(const std::string &text, SDL_Color color);
    void set(const std::string &text, SDL_Color color, int size);
    void setBackground(SDL_Color color);

    void render(Position p, Camera &camera);
    void renderFixed(ScreenPosition p, Camera &camera);

   protected:
    static std::unordered_map<TextCacheKey, TexturePtr, TextKeyHash> cache;

   private:
    void createTexture(SDL_Renderer *renderer);

    bool hasBackground{false};
    SDL_Color background;
    /* internal texture to represent the text in screen. */
    TexturePtr texture;
    /* text content to render. */
    std::string text;
    /* whether the texture should be rendered. */
    bool needs_render{true};
    /* the font to use. */
    Font &font;
    SDL_Color color;
    int size{-1};
};
}  // namespace GUI

#endif
```

```cpp
#include "Texture.h"
#include <SDL2/SDL_image.h>
#include <cassert>
#include "Exception.h"

GUI::Texture::Texture(const std::string &filename, SDL_Renderer &renderer, GUI::
Color key) {
    /* loads the image into a temporary surface */
    SDL_Surface *tmp = IMG_Load(filename.c_str());
    if (!tmp) {
        throw Exception{"Error loading %s: %s", filename.c_str(), IMG_GetError()};
    }

    SDL_SetColorKey(tmp, SDL_TRUE, SDL_MapRGB(tmp->format, key.r, key.g, key.b))
;

    /* creates a texture from the surface */
    this->texture = SDL_CreateTextureFromSurface(&renderer, tmp);
    if (!this->texture) {
        SDL_FreeSurface(tmp);
        throw Exception{"Error creating texture for %s: %s", filename.c_str(), SDL_GetErro
r()};
    }

    this->height = tmp->h;
    this->width = tmp->w;

    /* releases the temporary surface */
    SDL_FreeSurface(tmp);
}

GUI::Texture::Texture(SDL_Texture *texture, int width, int height)
    : height(height), width(width), texture(texture) {
    if (!texture) {
        throw Exception{"Texture cannot be NULL"};
    }
}

/**
 * @brief Move constructor.
 *
 * @param other Instance to move.
 */
GUI::Texture::Texture(Texture &&other) {
    this->height = other.height;
    this->width = other.width;
    std::swap(this->texture, other.texture);
}

GUI::Texture::~Texture() {
    if (this->texture) {
        SDL_DestroyTexture(this->texture);
    }
}

/**
 * @brief Gets the texture's width.
 *
 * @return int width.
 */
int GUI::Texture::getWidth() const {
    return this->width;
```



```cpp
}

/**
 * @brief Gets the texture's height.
 *
 * @return int Height.
 */
int GUI::Texture::getHeight() const {
    return this->height;
}

/**
 * @brief Returns the internal SDL texture.
 *
 * @return SDL texture.
 */
SDL_Texture *GUI::Texture::get() const {
    return this->texture;
}
```

```
#ifndef TEXTURE_H_
#define TEXTURE_H_

#include <SDL2/SDL.h>
#include <string>
#include "Color.h"

namespace GUI {
class Texture {
   public:
    Texture(const std::string &filename, SDL_Renderer &renderer, Color key);
    Texture(SDL_Texture *texture, int width, int height);
    Texture(Texture &&other);
    ~Texture();

    int getWidth() const;
    int getHeight() const;
    SDL_Texture *get() const;

   private:
    int height{0}, width{0};
    SDL_Texture *texture{nullptr};
};
}  // namespace GUI

#endif
```

```
#ifndef TEXTURE_MANAGER_H_
#define TEXTURE_MANAGER_H_

#include <SDL2/SDL.h>
#include <functional>
#include <string>
#include <unordered_map>
#include "Texture.h"

namespace GUI {
template <typename ID, typename HASH = std::hash<ID>>
class TextureManager {
   public:
    TextureManager(SDL_Renderer& renderer);
    ~TextureManager();
    TextureManager& operator=(TextureManager& other) = delete;

    void load(ID id, const std::string& file_name, Color key);
    const Texture& get(ID id) const;

   private:
    std::unordered_map<ID, Texture, HASH> cache;
    SDL_Renderer& renderer;
};
}  // namespace GUI

template <typename ID, typename HASH>
GUI::TextureManager<ID, HASH>::TextureManager(SDL_Renderer& renderer) : renderer
(renderer) {}

template <typename ID, typename HASH>
GUI::TextureManager<ID, HASH>::~TextureManager() {}

/**

 * @brief Loads a texture.
 *
 * @param file_name The image file name.
 * @param renderer Renderer.
 */
template <typename ID, typename HASH>
void GUI::TextureManager<ID, HASH>::load(ID id, const std::string& file_name, GU
I::Color key) {
    GUI::Texture texture{file_name, this->renderer, key};
    this->cache.insert(std::make_pair(id, std::move(texture)));
}

/**
 * @brief Gets a texture.
 *
 * @param file_name Name of the texture.
 */
template <typename ID, typename HASH>
const GUI::Texture& GUI::TextureManager<ID, HASH>::get(ID id) const {
    return this->cache.at(id);
}

#endif
```

```
//
// Created by rodrigo on 15/06/18.
//

#include <thread>
#include "Thread.h"

Thread::Thread(Thread &&thread) noexcept {
    this->thread = std::move(thread.thread);
}

Thread & Thread::operator=(Thread &&thread) noexcept {
    this->thread = std::move(thread.thread);
    return *this;
}

void Thread::start() {
    this->thread = std::thread(&Thread::run, this);
}

void Thread::join() {
    this->thread.join();
}
```

```
//
// Created by rodrigo on 15/06/18.
//

#ifndef INC_4_WORMS_THREAD_H
#define INC_4_WORMS_THREAD_H


#include <thread>

class Thread {
private:
    std::thread thread;

public:
    Thread() = default;

    /* El thread del objeto recibido de asigna al propio
     * mediante move semantics.
     */
    Thread(Thread &&thread) noexcept;

    /* El thread del objeto recibido de asigna al propio
     * mediante move semantics.
     */
    Thread & operator=(Thread &&thread) noexcept;

    /* Se elimina el constructor copia ya que no debe usarse para un thread.
     */
    Thread(const Thread&) = delete;

    /* Se elimina el operador asignaciÃ³n ya que no debe usarse para un thread.
     */
    Thread & operator=(const Thread&) = delete;

    /* Comienza la ejecuciÃ³n del hilo.
     */
    void start();

    /* Hace el join del hilo.
     */
    void join();

    /* Este mÃ©todo deben implementarlo todas las clases que hereden de esta,
     * ya que es el mÃ©todo que el thread ejecutarÃ¡.
     */
    virtual void run() = 0;

    /* Este mÃ©todo deben implementarlo todas las clases que hereden de esta,
     * ya que debe terminar la ejecuciÃ³n del hilo de una forma ordenada.
     */
    virtual void stop() = 0;

    /* El destructor es el default.
     */
    virtual ~Thread() = default;
};


#endif //INC_4_WORMS_THREAD_H
```

```
#ifndef UTILS_H_
#define UTILS_H_

namespace Utils {

/**
 * @brief This class allows using enums classes as hash keys.
 *
 */
struct EnumClassHash {
    template <typename T>
    std::size_t operator()(T t) const {
        return static_cast<std::size_t>(t);
    }
};
}  // namespace Utils

#endif
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 17/05/18.
 */

#include <SDL2/SDL.h>
#include <SDL2/SDL_image.h>
#include <SDL2/SDL_mixer.h>
#include <SDL2/SDL_ttf.h>
#include <iostream>

#include "Exception.h"
#include "Window.h"

GUI::Window::Window() : Window(WINDOW_WIDTH, WINDOW_HEIGHT) {}

GUI::Window::Window(int width, int height) {
    if (SDL_Init(SDL_INIT_EVERYTHING) < 0) {
        throw Exception{"SDL could not initialize: %s", SDL_GetError()};
    }

    if (!SDL_SetHint(SDL_HINT_RENDER_SCALE_QUALITY, "1")) {
        std::cerr << "Warning: Linear texture filtering not enabled!" << std::endl;
    }

    this->window = SDL_CreateWindow("Worms", SDL_WINDOWPOS_UNDEFINED, SDL_WINDOW
POS_UNDEFINED,
                                    width, height, SDL_WINDOW_SHOWN | SDL_WINDOW
_RESIZABLE);
    if (!this->window) {
        this->close();
        throw Exception{"Window could not be created: %s", SDL_GetError()};
    }

    this->renderer =
        SDL_CreateRenderer(this->window, -1, SDL_RENDERER_ACCELERATED | SDL_REND
ERER_PRESENTVSYNC);
    if (!this->renderer) {
        this->close();
        throw Exception{"Failed creating the render: %s", SDL_GetError()};
    }

    if ((!IMG_Init(IMG_INIT_PNG)) & IMG_INIT_PNG) {
        this->close();
        throw Exception{"Failed initialiing IMG: %s", IMG_GetError()};
    }

    if (TTF_Init() == -1) {
        this->close();
        throw Exception{"SDL_ttf could not initialize! SDL_ttf Error: %s", TTF_GetError()};
    }

    SDL_SetWindowSize(this->window, width, height);

    if (Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 2048) < 0) {
        throw Exception("SDL_mixer could not initialize! SDL_mixer Error: %s\n", Mix_GetError()
);
    }
}

GUI::Window::~Window() {
    this->close();
```



```
}

/**
 * @brief Maximizes the window.
 *
 */
void GUI::Window::maximize() {
    SDL_MaximizeWindow(this->window);
}

void GUI::Window::close() {
    if (this->renderer != nullptr) {
        SDL_DestroyRenderer(this->renderer);
        this->renderer = nullptr;
    }

    if (this->window != nullptr) {
        SDL_DestroyWindow(this->window);
        this->window = nullptr;
    }

    Mix_Quit();
    TTF_Quit();
    IMG_Quit();
    SDL_Quit();
}

void GUI::Window::clear() {
    this->clear({0xFF, 0xFF, 0xFF, 0xFF});
}

void GUI::Window::clear(SDL_Color color) {
    SDL_SetRenderDrawColor(this->renderer, color.r, color.g, color.b, color.a);
    SDL_RenderClear(this->renderer);
}

void GUI::Window::render() {
    SDL_RenderPresent(this->renderer);
}

SDL_Renderer& GUI::Window::getRenderer() {
    return *this->renderer;
}

/**
 * @brief Gets the window's height.
 *
 * @return int Window height.
 */
int GUI::Window::getHeight() const {
    int h, w;
    SDL_GL_GetDrawableSize(this->window, &w, &h);
    return h;
}

/**
 * @brief Gets the window's width.
 *
 * @return int Window width.
 */
int GUI::Window::getWidth() const {
    int h, w;
```

```
    SDL_GL_GetDrawableSize(this->window, &w, &h);
    return w;
}

/**
 * @brief Checks if the mouse cursor is contained in this window.
 *
 * @return true is the mouse is in the current window.
 */
bool GUI::Window::containsMouse() const {
    return (SDL_GetMouseFocus() == this->window);
}
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 17/05/18.
 */

#ifndef __Window_H__
#define __Window_H__

#include <SDL2/SDL.h>

#define WINDOW_WIDTH 1200
#define WINDOW_HEIGHT 800

namespace GUI {
class Window {
   public:
    Window();
    Window(int width, int height);
    ~Window();

    void maximize();
    void clear();
    void clear(SDL_Color color);
    void render();
    SDL_Renderer &getRenderer();

    int getWidth() const;
    int getHeight() const;
    bool containsMouse() const;

   private:
    SDL_Window *window{nullptr};
    SDL_Renderer *renderer{nullptr};
    void close();
};
}  // namespace GUI

#endif  //__Window_H__
```

```cpp
#include "WrapTexture.h"

GUI::WrapTexture::WrapTexture(const GUI::Texture& texture, float width, float he
ight)
    : texture(texture), width(width), height(height) {}

GUI::WrapTexture::~WrapTexture() {}

/**
 * @brief Renders the texture wrapped.
 *
 * @param p Position where to render.
 * @param camera Camera.
 */
void GUI::WrapTexture::render(GUI::Position p, Camera& camera) {
    this->render(p, 0.0f, camera);
}

/**
 * @brief Renders the texture wrapped with the given angle.
 *
 * @param p Position where to render.
 * @param angle Render angle.
 * @param camera Camera.
 */
void GUI::WrapTexture::render(GUI::Position p, float angle, Camera& camera) {
    int width = this->width * camera.getScale();
    int height = this->height * camera.getScale();

    int cols = width / this->texture.getWidth();
    int rows = height / this->texture.getHeight();

    int x_remainder = width % this->texture.getWidth();
    int y_remainder = height % this->texture.getHeight();

    ScreenPosition sp = camera.globalToScreen(p);
    sp.x -= width / 2;
    sp.y -= height / 2;

    SDL_Renderer* renderer = &camera.getRenderer();
    int tw = this->texture.getWidth();
    int th = this->texture.getHeight();

    for (int i = 0; i < cols; i++) {
        for (int j = 0; j < rows; j++) {
            ScreenPosition pos{sp.x + tw * i, sp.y - th * j};
            ScreenPosition rcenter = sp - pos;
            SDL_Point center = {rcenter.x + width / 2, rcenter.y + height / 2};

            SDL_Rect dst = {pos.x, pos.y, tw, th};
            SDL_RenderCopyEx(renderer, this->texture.get(), NULL, &dst, -angle,
&center,
                             SDL_FLIP_NONE);
        }

        if (y_remainder) {
            ScreenPosition pos{sp.x + tw * i, sp.y - th * rows};
            ScreenPosition rcenter = sp - pos;
            SDL_Point center = {rcenter.x + width / 2, rcenter.y + height / 2};

            SDL_Rect clip = {0, 0, tw, y_remainder};
            SDL_Rect dst = {pos.x, pos.y, tw, y_remainder};
```



```cpp
            SDL_RenderCopyEx(renderer, this->texture.get(), &clip, &dst, -angle,
 &center,
                             SDL_FLIP_NONE);
        }
    }

    if (x_remainder > 0) {
        for (int i = 0; i < rows; i++) {
            ScreenPosition pos{sp.x + tw * cols, sp.y - th * i};
            ScreenPosition rcenter = sp - pos;
            SDL_Point center = {rcenter.x + width / 2, rcenter.y + height / 2};

            SDL_Rect clip = {0, 0, x_remainder, th};
            SDL_Rect dst = {pos.x, pos.y, x_remainder, th};
            SDL_RenderCopyEx(renderer, this->texture.get(), &clip, &dst, -angle,
 &center,
                             SDL_FLIP_NONE);
        }

        if (y_remainder) {
            ScreenPosition pos{sp.x + tw * cols, sp.y - th * rows};
            ScreenPosition rcenter = sp - pos;
            SDL_Point center = {rcenter.x + width / 2, rcenter.y + height / 2};

            SDL_Rect clip = {0, 0, x_remainder, y_remainder};
            SDL_Rect dst = {pos.x, pos.y, x_remainder, y_remainder};
            SDL_RenderCopyEx(renderer, this->texture.get(), &clip, &dst, -angle,
 &center,
                             SDL_FLIP_NONE);
        }
    }
}

void GUI::WrapTexture::renderFixed(GUI::ScreenPosition sp, Camera& camera) {
    sp.x -= this->width * camera.getScale() / 2;
    sp.y -= this->height * camera.getScale() / 2;
    this->render(camera.screenToGlobal(sp), camera);
}
```

```
#ifndef WRAP_TEXTURE_H_
#define WRAP_TEXTURE_H_

#include "Camera.h"
#include "Texture.h"

namespace GUI {
class WrapTexture {
   public:
    WrapTexture(const Texture &texture, float width, float height);
    ~WrapTexture();

    void render(Position p, Camera &camera);
    void render(Position p, float angle, Camera &camera);
    void renderFixed(ScreenPosition p, Camera &camera);

   private:
    const Texture &texture;
    float width, height;
};
}  // namespace GUI

#endif
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 16/06/18
 */

#include "AerialAttack.h"
#define CONFIG Game::Config::getInstance()

Weapon::AerialAttack::AerialAttack()
    : Weapon::Weapon(CONFIG.getAerialAttackConfig(), Worm::WeaponID::WAerial, 0.
0),
      bulletsQuantity(CONFIG.getAerialAttackMissileQuantity()),
      missileSeparation(CONFIG.getAerialAttackMissileSeparation()) {}

void Weapon::AerialAttack::update(float dt) {}

void Weapon::AerialAttack::startShot(Worms::Player *player) {}

void Weapon::AerialAttack::endShot() {}

void Weapon::AerialAttack::setTimeout(uint8_t time) {}

std::list<Worms::Bullet> Weapon::AerialAttack::onExplode(const Worms::Bullet &ma
inBullet,
                                                         Worms::Physics &physics
) {
    return std::move(std::list<Worms::Bullet>());
}

void Weapon::AerialAttack::positionSelected(Worms::Player &p, Math::Point<float>
 point) {
    point.y += CONFIG.getAerialAttackLaunchHeight();
    point.x -= this->missileSeparation * (this->bulletsQuantity + 1) / 2;

    Worms::BulletInfo bulletInfo = {this->config.dmgInfo,
                                    point,
                                    this->config.minAngle,
                                    (float)this->config.maxShotPower,
                                    0,
                                    this->config.restitution,
                                    this->config.friction,
                                    this->config.explotionInitialTimeout,
                                    Event::Explode,
                                    this->config.bulletRadius,
                                    this->config.bulletDampingRatio,
                                    this->config.windAffected};

    std::list<Worms::Bullet> ret;
    for (int i = 0; i < this->bulletsQuantity; i++) {
        point.x += this->missileSeparation;
        bulletInfo.point = point;
        ret.emplace_back(bulletInfo, p.getPhysics(), Worm::WeaponID::WAerial);
    }

    p.endShot(ret);
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 16/06/18
 */

#ifndef __AERIAL_ATTACK_H__
#define __AERIAL_ATTACK_H__

#include "../Player.h"
#include "Weapon.h"

namespace Weapon {
class AerialAttack : public Worms::Weapon {
   public:
    AerialAttack();
    ~AerialAttack() override = default;
    void update(float dt) override;
    void startShot(Worms::Player *player) override;
    void endShot() override;
    void setTimeout(uint8_t time) override;
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &mainBullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override;

   private:
    const uint8_t bulletsQuantity{0};
    const float missileSeparation{0};
};
}  // namespace Weapon

#endif  //__AERIAL_ATTACK_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#include "BackFlipping.h"
#include "../Player.h"

Worms::BackFlipping::BackFlipping(GUI::Position p)
    : State(Worm::StateID::BackFlipping), startPosition(p) {}

void Worms::BackFlipping::update(Worms::Player &p, float dt, b2Body *body) {
    /*
     * when the worm lands (there was a collision between the worm and the
     * girder) it has to changes its state to endJump, and take an impulse
     * of equal absolute value and different sign of the impulse taken in
     * startJump stage (remember, the worm has a friction coefficient 0).
     *
     * In the y-axis there will be no impulse because its velocity was
     * cancelled because of the collision with the girder.
     */
    this->timeElapsed += dt;

    if (p.isOnGround()) {
        float32 mass = body->GetMass();
        b2Vec2 previousVel = body->GetLinearVelocity();
        b2Vec2 impulses = {mass * (0.0f - previousVel.x), 0.0f};
        body->ApplyLinearImpulseToCenter(impulses, true);

        p.landDamage(this->startPosition.y - p.getPosition().y);
        p.setState(Worm::StateID::Land);
        //        p.setState(Worm::StateID::EndBackFlip);
    }
}

void Worms::BackFlipping::moveRight(Worms::Player &p) {}

void Worms::BackFlipping::moveLeft(Worms::Player &p) {}

void Worms::BackFlipping::jump(Worms::Player &p) {}

void Worms::BackFlipping::stopMove(Worms::Player &p) {}

void Worms::BackFlipping::backFlip(Worms::Player &p) {}

void Worms::BackFlipping::bazooka(Worms::Player &p) {}

void Worms::BackFlipping::pointUp(Worms::Player &p) {}

void Worms::BackFlipping::pointDown(Worms::Player &p) {}

void Worms::BackFlipping::startShot(Worms::Player &p) {}

void Worms::BackFlipping::endShot(Worms::Player &p) {}

void Worms::BackFlipping::grenade(Worms::Player &p) {}

void Worms::BackFlipping::cluster(Worms::Player &p) {}

void Worms::BackFlipping::mortar(Worms::Player &p) {}

void Worms::BackFlipping::banana(Worms::Player &p) {}
```



```
void Worms::BackFlipping::holy(Worms::Player &p) {}

void Worms::BackFlipping::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::BackFlipping::aerialAttack(Worms::Player &p) {}

void Worms::BackFlipping::dynamite(Worms::Player &p) {}

void Worms::BackFlipping::teleport(Worms::Player &p) {}

void Worms::BackFlipping::baseballBat(Worms::Player &p) {}
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#ifndef __PLAYER_BACK_FLIPPING_H__
#define __PLAYER_BACK_FLIPPING_H__

#include <Camera.h>
#include <cstdint>
#include "PlayerState.h"

namespace Worms {
class BackFlipping : public State {
   public:
    BackFlipping(GUI::Position p);
    ~BackFlipping() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    GUI::Position startPosition;
};
}  // namespace Worms

#endif  //__PLAYER_BACK_FLIPPING_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#include "Banana.h"
#include "../Player.h"

Weapon::Banana::Banana(float angle)
    : Worms::Weapon(Game::Config::getInstance().getBananaConfig(), Worm::WeaponI
D::WBanana, angle) {
    this->powerChargeTime = Game::Config::getInstance().getPowerChargeTime();
}

void Weapon::Banana::update(float dt) {
    if (this->increaseShotPower) {
        if (this->shotPower < this->config.maxShotPower) {
            this->shotPower += dt / this->powerChargeTime * this->config.maxShot
Power;
        }
    }
}

void Weapon::Banana::startShot(Worms::Player *player) {
    this->increaseShotPower = true;
}

void Weapon::Banana::endShot() {
    this->increaseShotPower = false;
    this->shotPower = 0;
}

void Weapon::Banana::setTimeout(uint8_t time) {
    this->timeLimit = time;
}

std::list<Worms::Bullet> Weapon::Banana::onExplode(const Worms::Bullet &mainBull
et,
                                                   Worms::Physics &physics) {
    return std::move(std::list<Worms::Bullet>());
}

void Weapon::Banana::positionSelected(Worms::Player &p, Math::Point<float> point
) {}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#ifndef __Banana_H__
#define __Banana_H__

#include "Weapon.h"

namespace Weapon {
class Banana : public Worms::Weapon {
   public:
    Banana(float angle);
    ~Banana() override = default;
    void update(float dt) override;
    void startShot(Worms::Player *player) override;
    void endShot() override;
    void setTimeout(uint8_t time) override;
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &mainBullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override;

   private:
    float powerChargeTime{0.0f};
};
}  // namespace Weapon

#endif  //__Banana_H__
```

```
//
// Created by rodrigo on 16/06/18.
//

#include "BaseballBat.h"
#include "../Player.h"
#include "Direction.h"

Weapon::BaseballBat::BaseballBat(float angle)
    : Worms::Weapon(Game::Config::getInstance().getBaseballBatConfig(),
                    Worm::WeaponID::WBaseballBat, angle),
      weaponInfo{this->config.dmgInfo, Worm::Direction::left, {0, 0}} {
    this->isP2P = true;
}

void Weapon::BaseballBat::update(float dt) {}

void Weapon::BaseballBat::startShot(Worms::Player *player) {
    this->weaponInfo.position = player->getPosition();
    this->weaponInfo.direction = player->direction;
    this->weaponInfo.angle = this->angle;
}

void Weapon::BaseballBat::endShot() {}

void Weapon::BaseballBat::setTimeout(uint8_t time) {}

std::list<Worms::Bullet> Weapon::BaseballBat::onExplode(const Worms::Bullet &mai
nBullet,
                                                        Worms::Physics &physics)
 {
    return std::move(std::list<Worms::Bullet>());
}

void Weapon::BaseballBat::positionSelected(Worms::Player &p, Math::Point<float>
point) {}

Config::P2PWeapon &Weapon::BaseballBat::getWeaponInfo() {
    return this->weaponInfo;
}
```

```
//
// Created by rodrigo on 16/06/18.
//

#ifndef INC_4_WORMS_BASEBALLBAT_H
#define INC_4_WORMS_BASEBALLBAT_H

#include "../Config/P2PWeapon.h"
#include "../Physics.h"
#include "Weapon.h"

namespace Weapon {
class BaseballBat : public Worms::Weapon {
   public:
    BaseballBat(float angle);
    ~BaseballBat() = default;
    void update(float dt) override;
    void startShot(Worms::Player *player) override;
    void endShot() override;
    void setTimeout(uint8_t time) override;
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &mainBullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override;
    Config::P2PWeapon &getWeaponInfo();

   private:
    Config::P2PWeapon weaponInfo;
};
}  // namespace Weapon

#endif  // INC_4_WORMS_BASEBALLBAT_H
```

```
//
// Created by rodrigo on 23/06/18.
//

#include "Batting.h"
#include "../Config/Config.h"
#include "../Player.h"

Worms::Batting::Batting()
    : State(Worm::StateID::Batting), battingTime(Game::Config::getInstance().get
BattingTime()) {}

void Worms::Batting::update(Worms::Player &p, float dt, b2Body *body) {
    this->timeElapsed += dt;
    if (this->timeElapsed >= this->battingTime) {
        p.setState(Worm::StateID::Still);
    }
}

void Worms::Batting::moveRight(Worms::Player &p) {}

void Worms::Batting::moveLeft(Worms::Player &p) {}

void Worms::Batting::jump(Worms::Player &p) {}

void Worms::Batting::stopMove(Worms::Player &p) {}

void Worms::Batting::backFlip(Worms::Player &p) {}

void Worms::Batting::bazooka(Worms::Player &p) {}

void Worms::Batting::pointUp(Worms::Player &p) {}

void Worms::Batting::pointDown(Worms::Player &p) {}

void Worms::Batting::startShot(Worms::Player &p) {}

void Worms::Batting::endShot(Worms::Player &p) {}

void Worms::Batting::grenade(Worms::Player &p) {}

void Worms::Batting::cluster(Worms::Player &p) {}

void Worms::Batting::mortar(Worms::Player &p) {}

void Worms::Batting::banana(Worms::Player &p) {}

void Worms::Batting::holy(Worms::Player &p) {}

void Worms::Batting::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Batting::aerialAttack(Worms::Player &p) {}

void Worms::Batting::dynamite(Worms::Player &p) {}

void Worms::Batting::teleport(Worms::Player &p) {}

void Worms::Batting::baseballBat(Worms::Player &p) {}
```

```
//
// Created by rodrigo on 23/06/18.
//

#ifndef INC_4_WORMS_BATTING_H
#define INC_4_WORMS_BATTING_H

#include "PlayerState.h"

namespace Worms {
class Batting : public State {
   public:
    Batting();
    ~Batting() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    float battingTime;
};
}

#endif  // INC_4_WORMS_BATTING_H
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#include "Bazooka.h"
#include "../Player.h"

Weapon::Bazooka::Bazooka(float angle)
    : Worms::Weapon(Game::Config::getInstance().getBazookaConfig(), Worm::Weapon
ID::WBazooka,
                    angle) {
    this->powerChargeTime = Game::Config::getInstance().getPowerChargeTime();
}

void Weapon::Bazooka::update(float dt) {
    if (this->increaseShotPower) {
        if (this->shotPower < this->config.maxShotPower) {
            this->shotPower += dt / this->powerChargeTime * this->config.maxShot
Power;
        } else {
            this->player->endShot();
        }
    }
}

void Weapon::Bazooka::startShot(Worms::Player *player) {
    this->increaseShotPower = true;
    this->player = player;
}

void Weapon::Bazooka::endShot() {
    this->increaseShotPower = false;
    this->shotPower = 0;
}

void Weapon::Bazooka::setTimeout(uint8_t time) {}

std::list<Worms::Bullet> Weapon::Bazooka::onExplode(const Worms::Bullet &mainBul
let,
                                                    Worms::Physics &physics) {
    return std::move(std::list<Worms::Bullet>());
}

void Weapon::Bazooka::positionSelected(Worms::Player &p, Math::Point<float> poin
t) {}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#ifndef __BAZOOKA_H__
#define __BAZOOKA_H__

#include "Weapon.h"

namespace Weapon {
class Bazooka : public Worms::Weapon {
   public:
    Bazooka(float angle);
    ~Bazooka() = default;
    void update(float dt) override;
    void startShot(Worms::Player *player) override;
    void endShot() override;
    void setTimeout(uint8_t time) override;
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &mainBullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override;

   private:
    float powerChargeTime{0.0f};
    Worms::Player *player;
};
}  // namespace Weapon

#endif  //__BAZOOKA_H__
```

```cpp
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 22/06/18
 */

#include "BulletConfig.h"
#include "ConfigDefines.h"

Config::Bullet::DamageInfo::DamageInfo(const YAML::Node &config)
    : damage((std::uint16_t)config[DAMAGE].as<unsigned int>()),
      radius(config[RADIUS].as<float>()),
      impulseDampingRatio(config[IMPULSE_DAMPING_RATIO].as<float>()) {}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 22/06/18
 */

#ifndef __BULLET_CONFIG_H__
#define __BULLET_CONFIG_H__

#include <cstdint>
#include "yaml-cpp/yaml.h"

namespace Config {
namespace Bullet {
struct DamageInfo {
    std::uint16_t damage;
    float radius;
    float impulseDampingRatio;

    explicit DamageInfo(const YAML::Node &config);
};
}  // namespace Bullet
}  // namespace Bullet

#endif  //__BULLET_CONFIG_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 26/05/18
 */

#include <cmath>
#include <iostream>

#include "../Config/Config.h"
#include "Bullet.h"
#include "Weapon.h"
#include "../Physics.h"
#include "../PhysicsEntity.h"

Worms::Bullet::Bullet(BulletInfo &info, Worms::Physics &physics, Worm::WeaponID
weapon)
    : PhysicsEntity(Worms::EntityID::EtBullet), physics(physics), weaponID(weapo
n), info(info) {
    float distance = info.safeNonContactDistance + info.radius;
    this->bodyDef.type = b2_dynamicBody;
    this->bodyDef.position.Set(info.point.x + distance * cos(info.angle * PI / 1
80.0f),
                               info.point.y + distance * sin(info.angle * PI / 1
80.0f));
    this->bodyDef.fixedRotation = true;

    this->body = this->physics.createBody(this->bodyDef);
    this->shape.m_p.Set(0.0f, 0.0f);
    this->shape.m_radius = info.radius;
    this->fixture.shape = &this->shape;
    this->fixture.density = 1.0f;
    this->fixture.restitution = info.restitution;
    this->fixture.friction = info.friction;

    this->body->CreateFixture(&this->fixture);
    this->body->SetUserData(this);

    //    this->body->SetTransform(this->body->GetPosition(), info.angle);
}

void Worms::Bullet::update(float dt, Config::Wind wind) {
    if (this->keepUpdating) {
        this->timeElapsed += dt;
        if (!this->impulseApplied) {
            float32 mass = this->body->GetMass();
            b2Vec2 impulses = {mass * float32(this->info.power * this->info.damp
ingRatio *
                                              cos(this->info.angle * PI / 180.0f
)),
                               mass * float32(this->info.power * this->info.damp
ingRatio *
                                              sin(this->info.angle * PI / 180.0f
))};
            b2Vec2 position = this->body->GetWorldCenter();
            this->body->ApplyLinearImpulse(impulses, position, true);
            this->impulseApplied = true;
        } else {
            b2Vec2 velocity = this->body->GetLinearVelocity();
            this->info.angle = atan2(velocity.y, velocity.x) * 180.0f / PI;
            if (this->info.angle < 0) {
                this->info.angle += 360.0f;
            }
```



```
        }

        if (this->info.windAffected) {
            this->body->ApplyForceToCenter(b2Vec2{wind.instensity * wind.xDirect
ion, 0.0f}, true);
        }

        if (this->hasExploded()) {
            this->notify(*this, this->info.explodeEvent);
            this->weaponID = Worm::WeaponID::WExplode;
            this->keepUpdating = false;
            b2Vec2 lastP = this->body->GetPosition();
            this->lastPosition = {lastP.x, lastP.y};
            this->destroyBody();
        }
    }
}

Math::Point<float> Worms::Bullet::getPosition() const {
    if (this->keepUpdating) {
        b2Vec2 p = this->body->GetPosition();
        return Math::Point<float>(p.x, p.y);
    } else {
        return this->lastPosition;
    }
}

float Worms::Bullet::getAngle() const {
    return (this->info.angle >= 0 && this->info.angle < 90) ? this->info.angle +
 360.0f
                                                            : this->info.angle;
}

void Worms::Bullet::startContact(Worms::PhysicsEntity *physicsEntity) {
    this->madeImpact = true;
}

void Worms::Bullet::endContact(Worms::PhysicsEntity *physicsEntity) {}

Worms::Bullet::~Bullet() {
    this->destroyBody();
}

bool Worms::Bullet::hasExploded() const {
    if (this->getPosition().y < Game::Config::getInstance().getWaterLevel()) {
        return true;
    }
    if (this->info.explotionTimeout > 0) {
        return this->timeElapsed >= this->info.explotionTimeout;
    } else {
        return this->madeImpact;
    }
}

Config::Bullet::DamageInfo Worms::Bullet::getDamageInfo() const {
    return this->info.dmgInfo;
}

bool Worms::Bullet::operator<(Worms::Bullet &other) {
    return this->timeElapsed > other.timeElapsed;
}
```

```
Worm::WeaponID Worms::Bullet::getWeaponID() const {
    return this->weaponID;
}

void Worms::Bullet::destroyBody() {
    if (this->body != nullptr) {
        this->body->GetWorld()->DestroyBody(this->body);
        this->body = nullptr;
    }
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 26/05/18
 */

#ifndef __BULLET_H__
#define __BULLET_H__

#include <GameStateMsg.h>

#include "../Config/Config.h"
#include "../Config/WindConfig.h"
#include "../../../libs/Observer.h"
#include "../Physics.h"
#include "../PhysicsEntity.h"
#include "Point.h"

namespace Worms {
struct BulletInfo {
    Config::Bullet::DamageInfo dmgInfo;
    Math::Point<float> point;
    float angle;
    float power;
    float safeNonContactDistance;
    float restitution;
    float friction;
    uint8_t explotionTimeout;
    Event explodeEvent;
    float radius;
    float dampingRatio;
    bool windAffected;
};
/**
 * forward declaration of weapon.
 */
class Weapon;
class Bullet : public PhysicsEntity {
   public:
    Bullet(BulletInfo &i, Worms::Physics &physics, Worm::WeaponID weaponID);
    ~Bullet();
    /**
     * Apply initial impulse in the first iteration, or estimate the
     * bullet's tangential velocity to guide the animation. Finally, checks if
     * an Explode event ocurred, and notify his observer if so.
     * @param dt
     * @param w
     */
    void update(float dt, Config::Wind wind);
    Math::Point<float> getPosition() const;
    float getAngle() const;
    /**
     * Sets its impact boolean to true. Usefull for detecting explosion in
     * bullets that explode on first impact.
     * @param physicsEntity
     */
    virtual void startContact(Worms::PhysicsEntity *physicsEntity) override;
    virtual void endContact(Worms::PhysicsEntity *physicsEntity) override;
    /**
     * return true if the bullet is under the water, if its timeout (in the
     * case that it have it) has been reached, or if it has collided with
     * something
     * @return
```



```
     */
    bool hasExploded() const;
    Config::Bullet::DamageInfo getDamageInfo() const;
    bool operator<(Worms::Bullet &other);
    Worm::WeaponID getWeaponID() const;

   private:
    b2Body *body{nullptr};
    b2BodyDef bodyDef;
    b2CircleShape shape;
    b2FixtureDef fixture;
    Worms::Physics &physics;
    bool impulseApplied{false};
    float timeElapsed{0.0f};
    bool madeImpact{false};
    Worm::WeaponID weaponID;
    BulletInfo info;
    bool keepUpdating{true};
    Math::Point<float> lastPosition{0, 0};

    void destroyBody();
};
}  // namespace Worms

#endif  //__BULLET_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#include "Cluster.h"

#define CONFIG Game::Config::getInstance()

Weapon::Cluster::Cluster(float angle)
    : Worms::Weapon(CONFIG.getClusterConfig(), Worm::WeaponID::WCluster, angle),
      fragmentConfig(CONFIG.getClusterFragmentConfig()) {
    this->powerChargeTime = CONFIG.getPowerChargeTime();
}

void Weapon::Cluster::update(float dt) {
    if (this->increaseShotPower) {
        if (this->shotPower < this->config.maxShotPower) {
            this->shotPower += dt / this->powerChargeTime * this->config.maxShot
Power;
        }
    }
}

void Weapon::Cluster::startShot(Worms::Player *player) {
    this->increaseShotPower = true;
}

void Weapon::Cluster::endShot() {
    this->increaseShotPower = false;
    this->shotPower = 0;
}

void Weapon::Cluster::setTimeout(uint8_t time) {
    this->timeLimit = time;
}

std::list<Worms::Bullet> Weapon::Cluster::onExplode(const Worms::Bullet &mainBul
let,
                                                    Worms::Physics &physics) {
    uint8_t fragmentQuantity = CONFIG.getClusterFragmentQuantity();
    Math::Point<float> p = mainBullet.getPosition();
    Worms::BulletInfo bulletInfo = {this->fragmentConfig.dmgInfo,
                                    p,
                                    this->fragmentConfig.minAngle,
                                    (float)this->fragmentConfig.maxShotPower,
                                    this->fragmentConfig.bulletRadius * 6,
                                    this->fragmentConfig.restitution,
                                    this->fragmentConfig.friction,
                                    this->fragmentConfig.explotionInitialTimeout
,
                                    Event::Explode,
                                    this->fragmentConfig.bulletRadius,
                                    this->fragmentConfig.bulletDampingRatio,
                                    this->config.windAffected};

    std::list<Worms::Bullet> ret;
    for (int i = 0; i < fragmentQuantity; i++) {
        bulletInfo.angle = i * this->fragmentConfig.angleStep + this->fragmentCo
nfig.minAngle;
        ret.emplace_back(bulletInfo, physics, Worm::WeaponID::WFragment);
    }
```



```
    return std::move(ret);
}

void Weapon::Cluster::positionSelected(Worms::Player &p, Math::Point<float> poin
t) {}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#ifndef __CLUSTER_H__
#define __CLUSTER_H__

#include "../Physics.h"
#include "../Player.h"
#include "Weapon.h"

namespace Weapon {
class Cluster : public Worms::Weapon {
   public:
    Cluster(float angle);
    ~Cluster() override = default;
    void update(float dt) override;
    void startShot(Worms::Player *player) override;
    void endShot() override;
    void setTimeout(uint8_t time) override;
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &mainBullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override;

   private:
    const Config::Weapon &fragmentConfig;
    float powerChargeTime{0.0f};
};
}  // namespace Weapon

#endif  //__CLUSTER_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 01/06/18
 */

#include "Config.h"
#include <iostream>
#include "ConfigDefines.h"
#include "yaml-cpp/yaml.h"

/**
 * Meyer's singleton implementation.
 * @return
 */
Game::Config &Game::Config::getInstance() {
    static Config instance(YAML::LoadFile(CONFIG_PATH));
    return instance;
}

Game::Config::Config(const YAML::Node &node)
        : jumpVelocity(node[JUMP][VELOCITY][X].as<float>(), node[JUMP][VELOCITY]
[Y].as<float>()),
          backflipVelocity(node[BACKFLIP][VELOCITY][X].as<float>(),
                           node[BACKFLIP][VELOCITY][Y].as<float>()),
          startJumpTime(node[JUMP][START_TIME].as<float>()),
          landTime(node[JUMP][LAND_TIME].as<float>()),
          walkVelocity(node[WALK][VELOCITY].as<float>()),
          safeFallDistance(node[GAME][SAFE_FALL_DISTANCE].as<float>()),
          maxFallDamage(node[GAME][MAX_FALL_DAMAGE].as<float>()),
          turnTime((std::uint8_t)node[GAME][TURN_TIME].as<unsigned int>()),
          extraTurnTime(node[GAME][EXTRA_TURN_TIME].as<float>()),
          waitForNextTurnTime(node[GAME][WAIT_FOR_NEXT_TURN_TIME].as<float>()),
          powerChargeTime(node[GAME][POWER_CHARGE_MAX_TIME].as<float>()),
          dyingTime(node[GAME][DYING_TIME].as<float>()),
          drowningTime(node[GAME][DROWNING_TIME].as<float>()),
          battingTime(node[GAME][BATTING_TIME].as<float>()),
          teleportTime(node[GAME][TELEPORT_TIME].as<float>()),
          waterLevel(node[GAME][WATER_LEVEL].as<int>()),
          minWindIntensity(node[WIND_INTENSITY][MIN].as<float>()),
          maxWindIntensity(node[WIND_INTENSITY][MAX].as<float>()),
          bazooka(node[BAZOOKA]),
          greenGrenade(node[GRENADE]),
          cluster(node[CLUSTER]),
          clusterFragments(node[CLUSTER][FRAGMENT]),
          clusterFragmentQuantity((std::uint8_t)node[CLUSTER][FRAGMENT][QUANTITY
].as<unsigned int>()),
          mortar(node[MORTAR]),
          mortarFragments(node[MORTAR][FRAGMENT]),
          mortarFragmentQuantity((std::uint8_t)node[MORTAR][FRAGMENT][QUANTITY].
as<unsigned int>()),
          banana(node[BANANA]),
          holy(node[HOLY]),
          aerialAttackMissileQuantity(
                  (std::uint8_t)node[AERIAL_ATTACK][BULLET][QUANTITY].as<unsigne
d int>()),
          aerialAttackMissileSeparation(node[AERIAL_ATTACK][BULLET][SEPARATION].
as<float>()),
          aerialAttack(node[AERIAL_ATTACK]),
          aerialAttackLaunchHeight(node[AERIAL_ATTACK][LAUNCH_HEIGHT].as<float>(
)),
          dynamite(node[DYNAMITE]),
          teleport(node[TELEPORT]),
```



```
          baseballBat(node[BASEBALL_BAT]) {}

float Game::Config::getSafeFallDistance() const {
    return this->safeFallDistance;
}

float Game::Config::getMaxFallDamage() const {
    return this->maxFallDamage;
}

const Math::Vector Game::Config::getJumpVelocity() const {
    return this->jumpVelocity;
}

const float Game::Config::getStartJumpTime() const {
    return this->startJumpTime;
}

const float Game::Config::getLandTime() const {
    return this->landTime;
}

const Math::Vector Game::Config::getBackflipVelocity() const {
    return this->backflipVelocity;
}

const uint8_t Game::Config::getTurnTime() const {
    return this->turnTime;
}

const float Game::Config::getGameWidth() const {
    return this->gameWidth;
}

const float Game::Config::getGameHeight() const {
    return this->gameHeight;
}

const uint16_t Game::Config::getWormHealth() const {
    return this->wormHealth;
}

const Config::Weapon &Game::Config::getBazookaConfig() const {
    return this->bazooka;
}

const float Game::Config::getDyingTime() const {
    return this->dyingTime;
}

const float Game::Config::getDrowningTime() const {
    return this->drowningTime;
}

const float Game::Config::getExtraTurnTime() const {
    return this->extraTurnTime;
}

const int Game::Config::getWaterLevel() const {
    return this->waterLevel;
}
```

```
const float Game::Config::getWalkVelocity() const {
    return this->walkVelocity;
}

const Config::Weapon &Game::Config::getGreenGrenadeConfig() const {
    return this->greenGrenade;
}

const Config::Weapon &Game::Config::getClusterConfig() const {
    return this->cluster;
}

const Config::Weapon &Game::Config::getMortarConfig() const {
    return this->mortar;
}

const Config::Weapon &Game::Config::getBananaConfig() const {
    return this->banana;
}

const Config::Weapon &Game::Config::getHolyConfig() const {
    return this->holy;
}

const float Game::Config::getPowerChargeTime() const {
    return this->powerChargeTime;
}

const Config::Weapon &Game::Config::getClusterFragmentConfig() const {
    return this->clusterFragments;
}

const uint8_t Game::Config::getClusterFragmentQuantity() const {
    return this->clusterFragmentQuantity;
}

const Config::Weapon &Game::Config::getMortarFragmentConfig() const {
    return this->mortarFragments;
}

const uint8_t Game::Config::getMortarFragmentQuantity() const {
    return this->mortarFragmentQuantity;
}

const float Game::Config::getWaitForNextTurnTime() const {
    return this->waitForNextTurnTime;
}

const Config::Weapon &Game::Config::getAerialAttackConfig() const {
    return this->aerialAttack;
}

const uint8_t Game::Config::getAerialAttackMissileQuantity() const {
    return this->aerialAttackMissileQuantity;
}

const float Game::Config::getAerialAttackMissileSeparation() const {
    return this->aerialAttackMissileSeparation;
}

const float Game::Config::getAerialAttackLaunchHeight() const {
    return this->aerialAttackLaunchHeight;
```



```
}

const float Game::Config::getBattingTime() const {
    return this->battingTime;
}

const Config::Weapon &Game::Config::getTeleportConfig() const {
    return this->teleport;
}

const float Game::Config::getTeleportTime() const {
    return this->teleportTime;
}

const Config::Weapon &Game::Config::getDynamiteConfig() const {
    return this->dynamite;
}

const Config::Weapon &Game::Config::getBaseballBatConfig() const {
    return this->baseballBat;
}

float Game::Config::getMinWindIntensity() const {
    return this->minWindIntensity;
}

float Game::Config::getMaxWindIntensity() const {
    return this->maxWindIntensity;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 22/06/18
 */

#ifndef __CONFIG_DEFINES_H__
#define __CONFIG_DEFINES_H__

#define CONFIG_PATH "/etc/Worms/serverConfig.yaml"

#define JUMP "jump"
#define VELOCITY "velocity"
#define X "x"
#define Y "y"
#define BACKFLIP "backflip"
#define START_TIME "startTime"
#define LAND_TIME "landTime"
#define WALK "walk"
#define GAME "game"
#define SAFE_FALL_DISTANCE "safeFallDistance"
#define MAX_FALL_DAMAGE "maxFallDamage"
#define TURN_TIME "turnTime"
#define EXTRA_TURN_TIME "extraTurnTime"
#define WAIT_FOR_NEXT_TURN_TIME "waitForNextTurnTime"
#define POWER_CHARGE_MAX_TIME "powerChargeMaxTime"
#define DYING_TIME "dyingTime"
#define DROWNING_TIME "drowningTime"
#define BATTING_TIME "battingTime"
#define TELEPORT_TIME "teleportTime"
#define WATER_LEVEL "waterLevel"
#define WIND_INTENSITY "windIntensity"
#define MIN "min"
#define MAX "max"
#define BAZOOKA "bazooka"
#define GRENADE "grenade"
#define CLUSTER "cluster"
#define FRAGMENT "fragment"
#define QUANTITY "quantity"
#define MORTAR "mortar"
#define BANANA "banana"
#define HOLY "holy"
#define AERIAL_ATTACK "aerialAttack"
#define BULLET "bullet"
#define SEPARATION "separation"
#define LAUNCH_HEIGHT "launchHeight"
#define DYNAMITE "dynamite"
#define TELEPORT "teleport"
#define BASEBALL_BAT "baseballBat"

#define DAMAGE "damage"
#define RADIUS "radius"
#define IMPULSE_DAMPING_RATIO "impulseDampingRatio"
#define ANGLE "angle"
#define STEP "step"
#define MAX_SHOT_POWER "maxShotPower"
#define RESTITUTION "restitution"
#define FRICTION "friction"
#define EXPLOTION_INITIAL_TIMEOUT "explotionInitialTimeout"
#define HAS_AFTER_EXPLODE "hasAfterExplode"
#define DAMPING_RATIO "dampingRatio"
#define WIND_AFFECTED "windAffected"
```



```
#endif  //__CONFIG_DEFINES_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 01/06/18
 */

#ifndef __GAMECONFIG_H__
#define __GAMECONFIG_H__

#include <stdint.h>
#include <mutex>

#include "Direction.h"
#include "Point.h"
#include "WeaponConfig.h"
#include "WindConfig.h"

#define NUM_TEAMS 2
#define GAME_HEIGHT 30.0f
#define GAME_WIDTH 30.0f
#define WORM_HEALTH 100

namespace Math {
using Vector = Math::Point<float>;
}

namespace Game {

/**
 *  Singleton class with all the game configuration (Velocity constants,
 *  Weapons attributes, etc)
 */
class Config {
   public:
    static Config &getInstance();
    ~Config() = default;

    const Math::Vector getJumpVelocity() const;
    const Math::Vector getBackflipVelocity() const;
    const float getStartJumpTime() const;
    const float getLandTime() const;

    const float getWalkVelocity() const;
    float getSafeFallDistance() const;
    float getMaxFallDamage() const;
    float getMinWindIntensity() const;
    float getMaxWindIntensity() const;

    const uint8_t getTurnTime() const;
    const float getExtraTurnTime() const;
    const float getWaitForNextTurnTime() const;
    const float getPowerChargeTime() const;
    const float getGameWidth() const;
    const float getGameHeight() const;
    const float getDyingTime() const;
    const float getDrowningTime() const;
    const float getBattingTime() const;
    const float getTeleportTime() const;
    const int getWaterLevel() const;
    const uint16_t getWormHealth() const;

    const ::Config::Weapon &getBazookaConfig() const;
    const ::Config::Weapon &getGreenGrenadeConfig() const;
```



```
    const uint8_t getClusterFragmentQuantity() const;
    const ::Config::Weapon &getClusterConfig() const;
    const ::Config::Weapon &getMortarConfig() const;
    const ::Config::Weapon &getBananaConfig() const;
    const ::Config::Weapon &getHolyConfig() const;
    const ::Config::Weapon &getClusterFragmentConfig() const;
    const ::Config::Weapon &getMortarFragmentConfig() const;
    const uint8_t getMortarFragmentQuantity() const;
    const ::Config::Weapon &getAerialAttackConfig() const;
    const ::Config::Weapon &getDynamiteConfig() const;
    const uint8_t getAerialAttackMissileQuantity() const;
    const float getAerialAttackMissileSeparation() const;
    const float getAerialAttackLaunchHeight() const;
    const ::Config::Weapon &getTeleportConfig() const;
    const ::Config::Weapon &getBaseballBatConfig() const;

  private:
    /**
     * Constructor hidden because is a singleton.
     * TODO change constructor so it loads information from yaml file
     */
    //    Config();
    explicit Config(const YAML::Node &node);
    Config(Config &copy) = delete;
    Config(Config &&other) = delete;
    Config &operator=(Config &copy) = delete;
    Config &operator=(Config &&other) = delete;

    // jump
    const Math::Vector jumpVelocity;
    const Math::Vector backflipVelocity;
    const float startJumpTime;
    const float landTime;

    // moving
    const float walkVelocity;
    // game
    const float safeFallDistance;
    const float maxFallDamage;
    const std::uint8_t turnTime;
    const float extraTurnTime;
    const float waitForNextTurnTime;
    const float powerChargeTime;
    uint8_t numTeams{NUM_TEAMS};
    float gameWidth{GAME_WIDTH};
    float gameHeight{GAME_HEIGHT};
    uint16_t wormHealth{WORM_HEALTH};
    const float dyingTime;
    const float drowningTime;
    const float battingTime;
    const float teleportTime;
    const int waterLevel;
    const float minWindIntensity;
    const float maxWindIntensity;
    // weapons
    const ::Config::Weapon bazooka;
    const ::Config::Weapon greenGrenade;
    const ::Config::Weapon cluster;
    const ::Config::Weapon clusterFragments;
    const uint8_t clusterFragmentQuantity;
    const ::Config::Weapon mortar;
    const ::Config::Weapon mortarFragments;
```

```
    const uint8_t mortarFragmentQuantity;
    const ::Config::Weapon banana;
    const ::Config::Weapon holy;
    const uint8_t aerialAttackMissileQuantity;
    const float aerialAttackMissileSeparation;
    const ::Config::Weapon aerialAttack;
    const float aerialAttackLaunchHeight;
    const ::Config::Weapon dynamite;
    const ::Config::Weapon teleport;
    const ::Config::Weapon baseballBat;
};

void endTurn();
}  // namespace Game

#endif  //__GAMECONFIG_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 20/05/18
 */

#include <iostream>

#include "ContactEventListener.h"
#include "Player.h"

/**
 * @brief Pre collision solver handler for Box2D. Notifies colliding objects so
the can act
 * appropriately.
 *
 * @param contact Collision contact.
 * @param oldManifold Manifold.
 */
void ContactEventListener::PreSolve(b2Contact *contact, const b2Manifold *oldMan
ifold) {
    Worms::PhysicsEntity *e1 =
        static_cast<Worms::PhysicsEntity *>(contact->GetFixtureA()->GetBody()->G
etUserData());
    Worms::PhysicsEntity *e2 =
        static_cast<Worms::PhysicsEntity *>(contact->GetFixtureB()->GetBody()->G
etUserData());

    if (!e1 || !e2) {
        return;
    }

    e1->contactWith(*e2, *contact);
    e2->contactWith(*e1, *contact);
}

void ContactEventListener::BeginContact(b2Contact *contact) {
    Worms::PhysicsEntity *playerA =
        static_cast<Worms::PhysicsEntity *>(contact->GetFixtureA()->GetBody()->G
etUserData());
    Worms::PhysicsEntity *playerB =
        static_cast<Worms::PhysicsEntity *>(contact->GetFixtureB()->GetBody()->G
etUserData());
    /*
     * If fixture A is a Worm, then call startContact. This will delegate
     * the action to the internal state. For example, when a worm jump,
     * it run with a state startJump, after a few seconds (so the clients
     * could animate the impulse the worm takes to jump), it changes its
     * state to Jumping. The moment the state changes to endJump will be
     * when box2d detects a collision between the worm and the girder.
     */
    if (playerA) {
        playerA->startContact(playerB);
    }
    if (playerB) {
        playerB->startContact(playerA);
    }

    void *fixtureData = contact->GetFixtureA()->GetUserData();
    if (fixtureData) {
        Worms::PhysicsEntity *sensor = static_cast<Worms::TouchSensor *>(fixture
Data);
        sensor->startContact(playerB, *contact);
    }

    fixtureData = contact->GetFixtureB()->GetUserData();
    if (fixtureData) {
        Worms::PhysicsEntity *sensor = static_cast<Worms::TouchSensor *>(fixture
Data);
        sensor->startContact(playerA, *contact);
    }
}

void ContactEventListener::EndContact(b2Contact *contact) {
    Worms::PhysicsEntity *playerA =
        static_cast<Worms::PhysicsEntity *>(contact->GetFixtureA()->GetBody()->G
etUserData());
    Worms::PhysicsEntity *playerB =
        static_cast<Worms::PhysicsEntity *>(contact->GetFixtureB()->GetBody()->G
etUserData());

    if (playerA) {
        playerA->endContact(playerB);
    }
    if (playerB) {
        playerB->endContact(playerA);
    }

    void *fixtureData = contact->GetFixtureA()->GetUserData();
    if (fixtureData) {
        Worms::PhysicsEntity *sensor = static_cast<Worms::TouchSensor *>(fixture
Data);
        sensor->endContact(playerB, *contact);
    }

    fixtureData = contact->GetFixtureB()->GetUserData();
    if (fixtureData) {
        Worms::PhysicsEntity *sensor = static_cast<Worms::TouchSensor *>(fixture
Data);
        sensor->endContact(playerA, *contact);
    }
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 20/05/18
 */

#ifndef __ContactEventListener_H__
#define __ContactEventListener_H__

#include "Box2D/Box2D.h"

class ContactEventListener : public b2ContactListener {
   public:
    ContactEventListener() = default;
    ~ContactEventListener() = default;

    void PreSolve(b2Contact* contact, const b2Manifold* oldManifold) override;
    void BeginContact(b2Contact* contact) override;
    void EndContact(b2Contact* contact) override;
};

#endif  //__ContactEventListener_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#include "Dead.h"
#include "../Player.h"

Worms::Dead::Dead() : State(Worm::StateID::Dead) {}

void Worms::Dead::update(Worms::Player &p, float dt, b2Body *body) {}

void Worms::Dead::moveRight(Worms::Player &p) {}

void Worms::Dead::moveLeft(Worms::Player &p) {}

void Worms::Dead::jump(Worms::Player &p) {}

void Worms::Dead::stopMove(Worms::Player &p) {}

void Worms::Dead::backFlip(Worms::Player &p) {}

void Worms::Dead::bazooka(Worms::Player &p) {}

void Worms::Dead::pointUp(Worms::Player &p) {}

void Worms::Dead::pointDown(Worms::Player &p) {}

void Worms::Dead::startShot(Worms::Player &p) {}

void Worms::Dead::endShot(Worms::Player &p) {}

void Worms::Dead::grenade(Worms::Player &p) {}

void Worms::Dead::cluster(Worms::Player &p) {}

void Worms::Dead::mortar(Worms::Player &p) {}

void Worms::Dead::banana(Worms::Player &p) {}

void Worms::Dead::holy(Worms::Player &p) {}

void Worms::Dead::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Dead::aerialAttack(Worms::Player &p) {}

void Worms::Dead::dynamite(Worms::Player &p) {}

void Worms::Dead::teleport(Worms::Player &p) {}

void Worms::Dead::baseballBat(Worms::Player &p) {}
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#ifndef __Dead_H__
#define __Dead_H__

#include <cstdint>
#include "PlayerState.h"

namespace Worms {
class Dead : public State {
   public:
    Dead();
    ~Dead() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;
};
}  // namespace Worms

#endif  //__Dead_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#include "Die.h"
#include "../Player.h"

Worms::Die::Die() : State(Worm::StateID::Die) {}

void Worms::Die::update(Worms::Player &p, float dt, b2Body *body) {
    this->timeElapsed += dt;
    if (this->timeElapsed >= this->dyingTime) {
        if (p.dyingDisconnected) {
            p.notify(p, Event::DeadDueToDisconnection);
        } else {
            p.notify(p, Event::Dead);
        }
        p.setState(Worm::StateID::Dead);
    }
}

void Worms::Die::moveRight(Worms::Player &p) {}

void Worms::Die::moveLeft(Worms::Player &p) {}

void Worms::Die::jump(Worms::Player &p) {}

void Worms::Die::stopMove(Worms::Player &p) {}

void Worms::Die::backFlip(Worms::Player &p) {}

void Worms::Die::bazooka(Worms::Player &p) {}

void Worms::Die::pointUp(Worms::Player &p) {}

void Worms::Die::pointDown(Worms::Player &p) {}

void Worms::Die::startShot(Worms::Player &p) {}

void Worms::Die::endShot(Worms::Player &p) {}

void Worms::Die::grenade(Worms::Player &p) {}

void Worms::Die::cluster(Worms::Player &p) {}

void Worms::Die::mortar(Worms::Player &p) {}

void Worms::Die::banana(Worms::Player &p) {}

void Worms::Die::holy(Worms::Player &p) {}

void Worms::Die::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Die::aerialAttack(Worms::Player &p) {}

void Worms::Die::dynamite(Worms::Player &p) {}

void Worms::Die::teleport(Worms::Player &p) {}

void Worms::Die::baseballBat(Worms::Player &p) {}
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#ifndef __DIE_H__
#define __DIE_H__

#include <cstdint>
#include "../Config/Config.h"
#include "PlayerState.h"

namespace Worms {
class Die : public State {
   public:
    Die();
    ~Die() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    float dyingTime{Game::Config::getInstance().getDyingTime()};
};
}  // namespace Worms

#endif  //__DIE_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 29/05/18
 */

#include "Drowning.h"
#include "../Player.h"

Worms::Drowning::Drowning()
    : State(Worm::StateID::Drowning), drowningTime(Game::Config::getInstance().g
etDrowningTime()) {}

void Worms::Drowning::update(Worms::Player &p, float dt, b2Body *body) {
    this->timeElapsed += dt;
    if (this->timeElapsed >= this->drowningTime) {
        p.setState(Worm::StateID::Dead);
        p.notify(p, Event::Drowned);
    }
}

void Worms::Drowning::moveRight(Worms::Player &p) {}

void Worms::Drowning::moveLeft(Worms::Player &p) {}

void Worms::Drowning::jump(Worms::Player &p) {}

void Worms::Drowning::stopMove(Worms::Player &p) {}

void Worms::Drowning::backFlip(Worms::Player &p) {}

void Worms::Drowning::bazooka(Worms::Player &p) {}

void Worms::Drowning::pointUp(Worms::Player &p) {}

void Worms::Drowning::pointDown(Worms::Player &p) {}

void Worms::Drowning::startShot(Worms::Player &p) {}

void Worms::Drowning::endShot(Worms::Player &p) {}

void Worms::Drowning::grenade(Worms::Player &p) {}

void Worms::Drowning::cluster(Worms::Player &p) {}

void Worms::Drowning::mortar(Worms::Player &p) {}

void Worms::Drowning::banana(Worms::Player &p) {}

void Worms::Drowning::holy(Worms::Player &p) {}

void Worms::Drowning::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Drowning::aerialAttack(Worms::Player &p) {}

void Worms::Drowning::dynamite(Worms::Player &p) {}

void Worms::Drowning::teleport(Worms::Player &p) {}

void Worms::Drowning::baseballBat(Worms::Player &p) {}
```

```
/*
 *  Created by Rodrigo.
 *  date: 29/05/18
 */

#ifndef __Drown_H__
#define __Drown_H__

#include <cstdint>

#include "../Config/Config.h"
#include "PlayerState.h"

namespace Worms {
class Drowning : public State {
   public:
    Drowning();
    ~Drowning() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

    float timeElapsed{0.0f};
    float drowningTime;
};
}  // namespace Worms

#endif  //__Drown_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 16/06/18
 */

#include "Dynamite.h"
#include "../Player.h"

#define CONFIG Game::Config::getInstance()

Weapon::Dynamite::Dynamite()
    : Worms::Weapon(CONFIG.getDynamiteConfig(), Worm::WeaponID::WDynamite, 0.0)
{}

void Weapon::Dynamite::update(float dt) {}

void Weapon::Dynamite::startShot(Worms::Player *player) {}

void Weapon::Dynamite::endShot() {}

void Weapon::Dynamite::setTimeout(uint8_t time) {
    this->timeLimit = time;
}

std::list<Worms::Bullet> Weapon::Dynamite::onExplode(const Worms::Bullet &mainBu
llet,
                                                     Worms::Physics &physics) {
    return std::list<Worms::Bullet>();
}

void Weapon::Dynamite::positionSelected(Worms::Player &p, Math::Point<float> poi
nt) {}

void Weapon::Dynamite::increaseAngle() {}

void Weapon::Dynamite::decreaseAngle() {}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 16/06/18
 */

#ifndef __TNT_H__
#define __TNT_H__

#include "Weapon.h"

namespace Weapon {
class Dynamite : public Worms::Weapon {
   public:
    Dynamite();
    ~Dynamite() override = default;
    void update(float dt) override;
    void startShot(Worms::Player *player) override;
    void endShot() override;
    void setTimeout(uint8_t time) override;
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &mainBullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override;
    void increaseAngle() override;
    void decreaseAngle() override;
};
}  // namespace Weapon

#endif  //__TNT_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#include "EndBackFlip.h"
#include "../Config/Config.h"
#include "../Player.h"
#include "PlayerState.h"

Worms::EndBackFlip::EndBackFlip()
    : State(Worm::StateID::EndBackFlip), landTime(Game::Config::getInstance().ge
tLandTime()) {}

void Worms::EndBackFlip::update(Worms::Player &p, float dt, b2Body *body) {
    this->timeElapsed += dt;
    if (this->timeElapsed > this->landTime) {
        p.setState(Worm::StateID::Still);
    }
}

void Worms::EndBackFlip::moveRight(Worms::Player &p) {}

void Worms::EndBackFlip::moveLeft(Worms::Player &p) {}

void Worms::EndBackFlip::jump(Worms::Player &p) {}

void Worms::EndBackFlip::stopMove(Worms::Player &p) {}

void Worms::EndBackFlip::backFlip(Worms::Player &p) {}

void Worms::EndBackFlip::bazooka(Worms::Player &p) {}

void Worms::EndBackFlip::pointUp(Worms::Player &p) {}

void Worms::EndBackFlip::pointDown(Worms::Player &p) {}

void Worms::EndBackFlip::startShot(Worms::Player &p) {}

void Worms::EndBackFlip::endShot(Worms::Player &p) {}

void Worms::EndBackFlip::grenade(Worms::Player &p) {}

void Worms::EndBackFlip::cluster(Worms::Player &p) {}

void Worms::EndBackFlip::mortar(Worms::Player &p) {}

void Worms::EndBackFlip::banana(Worms::Player &p) {}

void Worms::EndBackFlip::holy(Worms::Player &p) {}

void Worms::EndBackFlip::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::EndBackFlip::aerialAttack(Worms::Player &p) {}

void Worms::EndBackFlip::dynamite(Worms::Player &p) {}

void Worms::EndBackFlip::teleport(Worms::Player &p) {}

void Worms::EndBackFlip::baseballBat(Worms::Player &p) {}
```

```
/*
 *  Created by Rodrigo.
 *  date: 21/05/18
 */

#ifndef __PLAYER_END_BACKFLIP_H__
#define __PLAYER_END_BACKFLIP_H__

#include <cstdint>
#include "PlayerState.h"

namespace Worms {
class EndBackFlip : public State {
   public:
    EndBackFlip();
    ~EndBackFlip() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    float landTime;
};
}  // namespace Worms

#endif  //__PLAYER_END_BACKFLIP_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 20/05/18
 */

#include "EndJump.h"
#include "../Config/Config.h"
#include "../Player.h"

Worms::EndJump::EndJump()
    : State(Worm::StateID::EndJump), landTime(Game::Config::getInstance().getLan
dTime()) {}

void Worms::EndJump::update(Worms::Player &p, float dt, b2Body *body) {
    this->timeElapsed += dt;
    if (this->timeElapsed > this->landTime) {
        p.setState(Worm::StateID::Still);
    }
}

void Worms::EndJump::moveRight(Worms::Player &p) {}

void Worms::EndJump::moveLeft(Worms::Player &p) {}

void Worms::EndJump::jump(Worms::Player &p) {}

void Worms::EndJump::stopMove(Worms::Player &p) {}

void Worms::EndJump::bazooka(Worms::Player &p) {}

void Worms::EndJump::pointUp(Worms::Player &p) {}

void Worms::EndJump::pointDown(Worms::Player &p) {}

void Worms::EndJump::backFlip(Worms::Player &p) {}

void Worms::EndJump::startShot(Worms::Player &p) {}

void Worms::EndJump::endShot(Worms::Player &p) {}

void Worms::EndJump::grenade(Worms::Player &p) {}

void Worms::EndJump::cluster(Worms::Player &p) {}

void Worms::EndJump::mortar(Worms::Player &p) {}

void Worms::EndJump::banana(Worms::Player &p) {}

void Worms::EndJump::holy(Worms::Player &p) {}

void Worms::EndJump::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::EndJump::aerialAttack(Worms::Player &p) {}

void Worms::EndJump::dynamite(Worms::Player &p) {}

void Worms::EndJump::teleport(Worms::Player &p) {}

void Worms::EndJump::baseballBat(Worms::Player &p) {}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 20/05/18
 */

#ifndef __PLAYER_END_JUMP_H__
#define __PLAYER_END_JUMP_H__

#include "PlayerState.h"

namespace Worms {
class EndJump : public State {
   public:
    EndJump();
    ~EndJump() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    virtual void pointUp(Player &p) override;
    virtual void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    const float landTime;
};
}  // namespace Worms

#endif  //__PLAYER_END_JUMP_H__
```

```
//
// Created by rodrigo on 3/06/18.
//

#include "Falling.h"

Worms::Falling::Falling(GUI::Position p) : State(Worm::StateID::Falling), startP
osition(p) {}

void Worms::Falling::update(Player &p, float dt, b2Body *body) {
    if (p.isOnGround()) {
        p.landDamage(this->startPosition.y - p.getPosition().y);
        p.setState(Worm::StateID::Land);
    }
}

void Worms::Falling::moveRight(Worms::Player &p) {}

void Worms::Falling::moveLeft(Worms::Player &p) {}

void Worms::Falling::jump(Worms::Player &p) {}

void Worms::Falling::stopMove(Worms::Player &p) {}

void Worms::Falling::backFlip(Worms::Player &p) {}

void Worms::Falling::bazooka(Worms::Player &p) {}

void Worms::Falling::pointUp(Worms::Player &p) {}

void Worms::Falling::pointDown(Worms::Player &p) {}

void Worms::Falling::startShot(Worms::Player &p) {}

void Worms::Falling::endShot(Worms::Player &p) {}

void Worms::Falling::grenade(Worms::Player &p) {}

void Worms::Falling::cluster(Worms::Player &p) {}

void Worms::Falling::mortar(Worms::Player &p) {}

void Worms::Falling::banana(Worms::Player &p) {}

void Worms::Falling::holy(Worms::Player &p) {}

void Worms::Falling::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Falling::aerialAttack(Worms::Player &p) {}

void Worms::Falling::dynamite(Worms::Player &p) {}

void Worms::Falling::teleport(Worms::Player &p) {}

void Worms::Falling::baseballBat(Worms::Player &p) {}
```

```
//
// Created by rodrigo on 3/06/18.
//

#ifndef INC_4_WORMS_FALLING_H
#define INC_4_WORMS_FALLING_H

#include <Camera.h>
#include <cstdint>
#include "../Player.h"

namespace Worms {
class Falling : public State {
   public:
    Falling(GUI::Position p);
    ~Falling() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;
    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    GUI::Position startPosition;
};
}  // namespace Worms

#endif  // INC_4_WORMS_FALLING_H
```

```
//
// Created by rodrigo on 10/06/18.
//

#include "GameClock.h"

GameClock::GameClock()
    : turnTime(Game::Config::getInstance().getTurnTime()),
      extraTurnTime(Game::Config::getInstance().getExtraTurnTime()),
      currentTurnTime(turnTime),
      waitForNextTurnTime(Game::Config::getInstance().getWaitForNextTurnTime())
{}

void GameClock::playerShot() {
    this->currentTurnTime = this->extraTurnTime;
    this->timeElapsed = 0.0f;
}

void GameClock::update(float dt) {
    this->timeElapsed += dt;
    if (this->timeElapsed > this->currentTurnTime) {
        if (this->waitingForNextTurn) {
            this->notify(*this, Event::NextTurn);
        } else {
            this->notify(*this, Event::EndTurn);
        }
    }
}

double GameClock::getTimeElapsed() const {
    return this->timeElapsed;
}

double GameClock::getTurnTime() const {
    return this->currentTurnTime;
}

void GameClock::restart() {
    this->waitingForNextTurn = false;
    this->timeElapsed = 0.0f;
    this->currentTurnTime = this->turnTime;
}

void GameClock::endTurn() {
    this->timeElapsed = this->currentTurnTime + 1.0f;
    this->notify(*this, Event::EndTurn);
}

void GameClock::waitForNextTurn() {
    this->timeElapsed = 0.0f;
    this->currentTurnTime = this->waitForNextTurnTime;
    this->waitingForNextTurn = true;
}
```

```
//
// Created by rodrigo on 10/06/18.
//

#ifndef INC_4_WORMS_GAMECLOCK_H
#define INC_4_WORMS_GAMECLOCK_H

#include "Config/Config.h"
#include "Subject.h"

class GameClock : public Subject {
   public:
    GameClock();
    ~GameClock() = default;
    void update(float dt);
    void playerShot();
    double getTimeElapsed() const;
    double getTurnTime() const;
    void waitForNextTurn();
    void restart();
    void endTurn();

   private:
    float timeElapsed{0.0f};
    float turnTime;
    float extraTurnTime;
    float currentTurnTime;
    float waitForNextTurnTime;
    bool waitingForNextTurn{false};
};

#endif  // INC_4_WORMS_GAMECLOCK_H
```

```cpp
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#include <Stage.h>
#include <zconf.h>
#include <atomic>
#include <cassert>
#include <chrono>
#include <iostream>
#include <random>
#include "Box2D/Box2D.h"
#include "Chronometer.h"

#include "Config/Config.h"
#include "Direction.h"
#include "Game.h"
#include "GameStates/ImpactOnCourse.h"
#include "Player.h"
#include "Stage.h"
#include "Weapons/BaseballBat.h"

#define CONFIG ::Game::Config::getInstance()
#define TIME_STEP (1.0f / 30.0f)

Worms::Game::Game(Stage &&stage, std::vector<CommunicationSocket> &sockets)
    : physics(b2Vec2{0.0f, -10.0f}, TIME_STEP),
      stage(std::move(stage)),
      maxTurnTime(::Game::Config::getInstance().getExtraTurnTime()),
      gameTurn(*this),
      sockets(sockets),
      inputs(sockets.size()),
      snapshots(sockets.size()),
      playersConnected(sockets.size()) {
    this->inputThreads.reserve(sockets.size());
    this->outputThreads.reserve(sockets.size());
    for (std::size_t i = 0; i < sockets.size(); i++) {
        this->inputThreads.emplace_back([this, i] { this->inputWorker(i); });
        this->outputThreads.emplace_back([this, i] { this->outputWorker(i); });
    }
    /* reserves the required space to avoid reallocations that may move the worm
 addresses */
    this->players.reserve(this->stage.getWorms().size());
    uint8_t id = 0;
    for (auto &wormData : this->stage.getWorms()) {
        /* initializes the instances */
        this->players.emplace_back(this->physics);
        this->players.back().setPosition(wormData.position);
        this->players.back().health = wormData.health;
        this->players.back().setId(id);
        this->players.back().addObserver(this);
        id++;
    }

    this->teams.makeTeams(this->players, (uint8_t)sockets.size(), this->stage.ge
tAmmoCounter());
    //    this->wind.range = CONFIG.getWindIntensityRange();
    this->wind.minIntensity = CONFIG.getMinWindIntensity();
    this->wind.maxIntensity = CONFIG.getMaxWindIntensity();
    this->calculateWind();
```



```cpp
    /* sets the girders */
    this->girders.reserve(this->stage.getGirders().size());
    for (auto &girder : this->stage.getGirders()) {
        this->girders.emplace_back(girder, this->physics);
    }

    /* calculate the initial team's healths */
    this->teamHealths = this->teams.getTotalHealth(this->players);

    this->currentWorm = this->teams.getCurrentPlayerID();
    this->currentWormToFollow = this->currentWorm;

    this->gameClock.addObserver(this);
    this->gameClock.waitForNextTurn();
}

Worms::Game::~Game() {
    this->exit();
    for (auto &t : this->outputThreads) {
        t.join();
    }

    for (auto &t : this->inputThreads) {
        t.join();
    }
}
///**
// * @brief Reads player messages from a socket and pushes them into the input q
ueue.
// *
// * @param playerIndex The index of the player.
// */
//void Worms::Game::inputWorker(std::size_t playerIndex) {
//    PlayerInput &input = this->inputs.at(playerIndex);
//    CommunicationSocket &socket = this->sockets.at(playerIndex);
//
//    /* TODO: avoid hardcoding the size */
//    IO::PlayerMsg msg;
//    char *buffer = new char[msg.getSerializedSize()];
//
//    try {
//        while (!this->quit) {
//            /* reads the raw data from the buffer */
//            socket.receive(buffer, msg.getSerializedSize());
//
//            /* sets the struct data from the buffer */
//            msg.deserialize(buffer, msg.getSerializedSize());
//
//            /* pushes the message into the player's input queue if it's the cu
rrent player */
//            if (this->currentTeam == playerIndex) {
//                input.push(msg);
//            }
//        }
//    } catch (const std::exception &e) {
//        std::cerr << "Worms::Game::inputWorker:" << e.what() << std::endl;
//        msg.input = IO::PlayerInput::disconnected;
//        msg.position = Math::Point<float>{0, 0};
//        input.push(msg);
//    } catch (...) {
//        std::cerr << "Unknown error in Worms::Game::inputWorker()" << std::end
l;
```

```
//     }
//
//     delete[] buffer;
//}
//
///**
// * @brief Sends model snapshot messages to a socket.
// *
// * @param playerIndex The index of the player to send the spanshots to.
// */
//void Worms::Game::outputWorker(std::size_t playerIndex) {
//     CommunicationSocket &socket = this->sockets.at(playerIndex);
//     GameSnapshot &snapshot = this->snapshots.at(playerIndex);
//
//     IO::GameStateMsg msg;
//     char *buffer = new char[msg.getSerializedSize()];
//
//     try {
//         while (!this->quit) {
//             msg = snapshot.get(true);
//             msg.serialize(buffer, msg.getSerializedSize());
//             socket.send(buffer, msg.getSerializedSize());
//         }
//     } catch (const IO::Interrupted &e) {
//         /* this means that the game is ready to exit */
//     } catch (const std::exception &e) {
//         std::cerr << "Worms::Game::outputWorker:" << e.what() << std::endl;
//     } catch (...) {
//         std::cerr << "Unknown error in Worms::Game::outputWorker()" << std::en
dl;
//     }
//
//     delete[] buffer;
//}


/**
 * @brief Reads player messages from a socket and pushes them into the input que
ue.
 *
 * @param playerIndex The index of the player.
 */
void Worms::Game::inputWorker(std::size_t playerIndex) {
    PlayerInput &input = this->inputs.at(playerIndex);
    CommunicationSocket &socket = this->sockets.at(playerIndex);

    /* TODO: avoid hardcoding the size */
    IO::PlayerMsg msg;
    try {
        while (!this->quit) {
            /* receives the size of the msg */
            std::uint32_t size(0);
            socket.receive((char *)&size, sizeof(std::uint32_t));
            size = ntohl(size);

            std::vector<char> buffer(size, 0);
            /* reads the raw data from the buffer */
            socket.receive(buffer.data(), size);

            std::string buff(buffer.data(), size);

            /* sets the struct data from the buffer */
```



```
            msg.deserialize(buff);

            /* pushes the message into the player's input queue if it's the curr
ent player */
            if (this->currentTeam == playerIndex) {
                input.push(msg);
            }
        }
    } catch (const std::exception &e) {
        std::cerr << "Worms::Game::inputWorker:" << e.what() << std::endl;
        msg.input = IO::PlayerInput::disconnected;
        msg.position = Math::Point<float>{0, 0};
        input.push(msg);
    } catch (...) {
        std::cerr << "Unknown error in Worms::Game::inputWorker()" << std::endl;
    }
}

/**
 * @brief Sends model snapshot messages to a socket.
 *
 * @param playerIndex The index of the player to send the spanshots to.
 */
void Worms::Game::outputWorker(std::size_t playerIndex) {
    CommunicationSocket &socket = this->sockets.at(playerIndex);
    GameSnapshot &snapshot = this->snapshots.at(playerIndex);

    IO::GameStateMsg msg;
    try {
        while (!this->quit) {
            msg = snapshot.get(true);
            std::string buff = msg.serialize();
            std::uint32_t size = buff.size();
            std::uint32_t netInt = htonl(size);

            socket.send((char *)&netInt, sizeof(std::uint32_t));
            socket.send(buff.data(), size);
        }
    } catch (const IO::Interrupted &e) {
        /* this means that the game is ready to exit */
    } catch (const std::exception &e) {
        std::cerr << "Worms::Game::outputWorker:" << e.what() << std::endl;
    }
}

void Worms::Game::start() {
    try {
        /* game loop */
        Utils::Chronometer chronometer;
        while (!quit) {
            double dt = chronometer.elapsed();

            this->gameClock.update(dt);
            this->gameTurn.update(dt);

            IO::PlayerMsg pMsg;
            if (this->inputs.at(this->currentTeam).pop(pMsg, false)) {
                if (pMsg.input == IO::PlayerInput::disconnected) {
                    this->playerDisconnected(this->currentTeam);
                } else {
                    if (this->processingClientInputs) {
                        if (this->currentPlayerShot) {
```

```
                            if (pMsg.input != IO::PlayerInput::startShot &&
                                pMsg.input != IO::PlayerInput::endShot &&
                                pMsg.input != IO::PlayerInput::positionSelected)
 {
                                this->players.at(this->currentWorm).handleState(
pMsg);
                            }
                        } else {
                            this->players.at(this->currentWorm).handleState(pMsg
);
                        }
                    } else {
                        this->players.at(this->currentWorm).handleState(pMsg);
                    }
                }
            }

            /* updates the actors */
            for (auto &worm : this->players) {
                worm.update(dt);
            }

            for (auto &bullet : this->bullets) {
                bullet.update(dt, this->wind);
            }

            this->physics.update(dt);

            /* serializes and updates the game state */
            auto msg = this->serialize();
            for (auto &snapshot : this->snapshots) {
                snapshot.set(msg);
                snapshot.swap();
            }

            if (this->gameEnded) {
                this->quit = true;
            }

            if (TIME_STEP > dt) {
                usleep((TIME_STEP - dt) * 1000000);
            }
        }
    } catch (std::exception &e) {
        std::cerr << e.what() << std::endl << "In Worms::Game::start" << std::endl;
    } catch (...) {
        std::cerr << "Unkown error in Worms::Game::start()" << std::endl;
    }
}

void Worms::Game::endTurn() {
    this->waitingForNextTurn = false;
    this->processingClientInputs = true;
    this->gameClock.restart();
    this->gameTurn.restart();
    this->calculateWind();
}

void Worms::Game::calculateCurrentPlayer() {
    this->waitingForNextTurn = true;
    this->players[this->currentWorm].reset();
    this->gameEnded = this->teams.endTurn(this->players);
```



```
    if (this->gameEnded) {
        this->winnerTeam = this->teams.getWinner();
    }
    this->currentTeam = this->teams.getCurrentTeamID();
    this->currentWorm = this->teams.getCurrentPlayerID();
    this->currentWormToFollow = this->currentWorm;
}

IO::GameStateMsg Worms::Game::serialize() const {
    assert(this->players.size() <= 20);

    IO::GameStateMsg m;
    memset(&m, 0, sizeof(m));

    m.num_worms = 0;
    m.num_teams = this->teams.getTeamQuantity();
    for (const auto &worm : this->players) {
        m.positions[m.num_worms * 2] = worm.getPosition().x;
        m.positions[m.num_worms * 2 + 1] = worm.getPosition().y;
        m.stateIDs[m.num_worms] = worm.getStateId();
        m.wormsHealth[m.num_worms] = worm.health;
        m.wormsTeam[m.num_worms] = worm.getTeam();
        m.wormsDirection[m.num_worms] = worm.direction;
        m.num_worms++;
    }

    /* sets team health*/
    uint8_t i{0};
    for (auto health : this->teamHealths) {
        m.teamHealths[i++] = health;
    }
    /* sets wind data */
    m.windIntensity =
        (char)(127.0f * this->wind.instensity /
                (this->wind.maxIntensity - this->wind.minIntensity) * this->wind.
xDirection);

    /* sets the current player's data */
    m.elapsedTurnSeconds = static_cast<std::uint16_t>(std::floor(this->gameClock
.getTimeElapsed()));
    m.currentPlayerTurnTime = static_cast<std::uint16_t>(std::floor(this->gameCl
ock.getTurnTime()));
    m.currentWorm = this->currentWorm;
    m.currentWormToFollow = this->currentWormToFollow;
    m.currentTeam = this->currentTeam;
    m.activePlayerAngle = this->players[this->currentWorm].getWeaponAngle();
    m.activePlayerWeapon = this->players[this->currentWorm].getWeaponID();

    m.bulletsQuantity = this->bullets.size();
    i = 0;
    uint8_t j = 0;
    for (auto &bullet : this->bullets) {
        Math::Point<float> p = bullet.getPosition();
        m.bullets[i++] = p.x;
        m.bullets[i++] = p.y;
        m.bulletsAngle[j] = bullet.getAngle();
        m.bulletType[j++] = bullet.getWeaponID();
    }
    /*
     * serialize the ammunition counter
     */
    this->teams.serialize(m);
```

```
    m.processingInputs = this->processingClientInputs;
    m.playerUsedTool = this->currentPlayerShot;
    m.waitingForNextTurn = this->waitingForNextTurn;
    m.gameEnded = this->gameEnded;
    m.winner = this->winnerTeam;

    return m;
}

void Worms::Game::exit() {
    this->quit = true;
    for (auto &snapshot : this->snapshots) {
        snapshot.interrupt();
    }
    for (auto &socket : this->sockets) {
        socket.shutdown();
    }
}

void Worms::Game::onNotify(Subject &subject, Event event) {
    switch (event) {
        /**
         * Because i didnt want to move all responsability of the bullets to
         * the game (until the refactor of the start), i added this function
         * that delegates to the player the responsability to iterate all over
         * the bullets and add the game as an observer
         */
        case Event::Shot: {
            //             this->players[this->currentWorm].addObserverToBullets
(this);
            this->bullets.merge(this->players[this->currentWorm].getBullets());
            for (auto &bullet : this->bullets) {
                bullet.addObserver(this);
            }
            this->gameClock.playerShot();
            this->gameTurn.playerShot(this->players[this->currentWorm].getWeapon
ID());
            this->currentPlayerShot = true;
            break;
        }
        /**
         * On explode, the game must check worms health.
         */
        case Event::Explode: {
            auto &bullet = dynamic_cast<const Bullet &>(subject);
            this->gameTurn.explosion();
            this->calculateDamage(bullet);
            break;
        }
        case Event::P2PWeaponUsed: {
            auto &player = dynamic_cast<const Worms::Player &>(subject);
            const std::shared_ptr<Worms::Weapon> weapon = player.getWeapon();
            this->gameClock.playerShot();
            this->gameTurn.playerShot(this->players[this->currentWorm].getWeapon
ID());
            this->currentPlayerShot = true;
            this->gameTurn.explosion();
            this->calculateDamage(weapon, player.getPosition(), player.direction
);
            break;
        }
        /**
```



```
         * onExplode will create new Bullets in player's container, and we
         * need to listen to them.
         */
        case Event::OnExplode: {
            auto &bullet = dynamic_cast<const Bullet &>(subject);
            this->calculateDamage(bullet);

            this->bullets.merge(this->players[this->currentWorm].onExplode(bulle
t, this->physics));

            for (auto &fragment : this->bullets) {
                fragment.addObserver(this);
            }
            //             this->players[this->currentWorm].addObserverToBullets(
this);
            break;
        }
        case Event::DyingDueToDisconnection: {
            this->gameTurn.playerDisconnected(dynamic_cast<const Player &>(subje
ct).getId());
            break;
        }
        case Event::DeadDueToDisconnection: {
            this->gameTurn.playerDisconnectedDead(dynamic_cast<const Player &>(s
ubject).getId());
            break;
        }
        case Event::Teleported: {
            this->gameClock.playerShot();
            this->currentPlayerShot = true;
            this->teams.weaponUsed(this->players[this->currentWorm].getWeaponID(
));
            break;
        }
        case Event::WormFalling: {
            this->gameTurn.wormFalling(dynamic_cast<const Player &>(subject).get
Id());
            break;
        }
        case Event::WormLanded: {
            this->gameTurn.wormLanded(dynamic_cast<const Player &>(subject).getI
d());
            break;
        }
        case Event::Hit: {
            this->gameTurn.wormHit(dynamic_cast<const Player &>(subject).getId()
);
            break;
        }
        case Event::EndHit: {
            this->gameTurn.wormEndHit(dynamic_cast<const Player &>(subject).getI
d());
            break;
        }
        case Event::Drowning: {
            this->gameTurn.wormDrowning(dynamic_cast<const Player &>(subject).ge
tId());
            break;
        }
        case Event::Drowned: {
            this->gameTurn.wormDrowned(dynamic_cast<const Player &>(subject).get
Id());
```

```
            break;
        }
        case Event::Dying: {
            this->gameTurn.wormDying();
            break;
        }
        case Event::Dead: {
            this->gameTurn.wormDead();
            this->gameTurn.endTurn();
            break;
        }
        case Event::NewWormToFollow: {
            this->currentWormToFollow =
                dynamic_cast<const GameTurnState &>(subject).getWormToFollow();
            break;
        }
        case Event::DamageOnLanding: {
            this->gameClock.endTurn();
            break;
        }
        case Event::ImpactEnd: {
            auto &wormsHit = dynamic_cast<ImpactOnCourse &>(subject).getWormsHit
();
            for (auto worm : wormsHit) {
                Worm::StateID wormState = this->players[worm].getStateId();
                if (this->players[worm].health == 0) {
                    if (wormState != Worm::StateID::Die && wormState != Worm::St
ateID::Dead) {
                        this->players[worm].notify(this->players[worm], Event::D
ying);
                        this->players[worm].setState(Worm::StateID::Die);
                    }
                }
            }
            break;
        }
        case Event::EndTurn: {
            this->processingClientInputs = false;
            this->gameTurn.endTurn();
            break;
        }
        case Event::TurnEnded: {
            if (this->players[this->currentWorm].getStateId() != Worm::StateID::
Dead) {
                this->players[this->currentWorm].setState(Worm::StateID::Still);
            }
            this->bullets.erase(this->bullets.begin(), this->bullets.end());
            this->gameClock.waitForNextTurn();
            this->teamHealths = this->teams.getTotalHealth(this->players);
            this->calculateCurrentPlayer();
            break;
        }
        case Event::NextTurn: {
            this->currentPlayerShot = false;
            this->endTurn();
            break;
        }
        default: {
            break;
        }
    }
}
```



```
void Worms::Game::calculateDamage(const Worms::Bullet &bullet) {
    Config::Bullet::DamageInfo damageInfo = bullet.getDamageInfo();
    for (auto &worm : this->players) {
        worm.acknowledgeDamage(damageInfo, bullet.getPosition());
    }
    this->removeBullets = true;
}
/**
 * @brief calculate damage for p2p weapons. Because the only one is the
 * baseball bat and because we are running out of time, there will be
 * a cast to a baseballWeapon.
 * TODO make a class between weapon and baseballBat, that represents a
 * p2pWeapon.
 * @param weapon
 */
void Worms::Game::calculateDamage(std::shared_ptr<Worms::Weapon> weapon,
                                  Math::Point<float> shooterPosition,
                                  Worm::Direction shooterDirection) {
    auto *baseball = (::Weapon::BaseballBat *)weapon.get();
    Config::P2PWeapon &weaponInfo = baseball->getWeaponInfo();
    for (auto &worm : this->players) {
        worm.acknowledgeDamage(weaponInfo, shooterPosition, shooterDirection);
    }
    this->removeBullets = true;
}

void Worms::Game::calculateWind() {
        std::random_device rnd_device;
        std::mt19937 mersenne_engine(rnd_device());
        std::uniform_real_distribution<> distr(this->wind.minIntensity, this->wi
nd.maxIntensity);

        this->wind.xDirection =
            (distr(mersenne_engine) > (this->wind.maxIntensity - this->wind.minI
ntensity) / 2.0f)
            ? 1

                  : -1;
        this->wind.instensity = (float) distr(mersenne_engine);
}

void Worms::Game::playerDisconnected(uint8_t teamDisconnected) {
    this->playersConnected--;
    this->teams.kill(teamDisconnected, this->players);
    if (this->playersConnected <= 1) {
        this->winnerTeam = this->teams.getWinner();
        this->gameEnded = true;
    }
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#ifndef __GAME_H__
#define __GAME_H__

#include <atomic>
#include <list>
#include <thread>
#include <unordered_map>

#include "CommunicationSocket.h"
#include "Direction.h"
#include "DoubleBuffer.h"
#include "GameClock.h"
#include "GameTeams.h"
#include "GameTurn.h"
#include "Girder.h"
#include "Observer.h"
#include "Player.h"
#include "Stage.h"
#include "Weapons/Bullet.h"

namespace Worms {
using PlayerInput = IO::Stream<IO::PlayerMsg>;
using GameSnapshot = IO::DoubleBuffer<IO::GameStateMsg>;

struct Teamasd {
    std::vector<uint8_t> players;
    uint8_t currentPlayer;
    bool alive;
};

class Game : Observer {
   public:
    std::atomic<bool> quit{false};

    Game(Stage &&stage, std::vector<CommunicationSocket> &sockets);
    virtual ~Game();

    Game(Game &&other) = delete;

    void start();
    IO::GameStateMsg serialize() const;
    void onNotify(Subject &subject, Event event) override;
    /**
     * @brief calculates damage for weapons that throw bullets. It gives
     * information of the bullet to all players so them can calculate his damage
     * and apply an impulse if this was hitted.
     * @param bullet
     */
    void calculateDamage(const Bullet &bullet);
    /**
     * @brief calculates damage for p2p weapons (baseball). It gives
     * information of the weapon (direction, point and damageInfo) to the
     * players so that they can calculate his damage and apply an impulse if
     * this was hitted.
     * @param weapon
     */
    void calculateDamage(std::shared_ptr<Worms::Weapon> weapon, Math::Point<floa
```



```
t> shooterPosition,
                         Worm::Direction shooterDirection);
    void calculateWind();
    void exit();
    void endTurn();

   private:
    void inputWorker(std::size_t playerIndex);
    void outputWorker(std::size_t playerIndex);
    void calculateCurrentPlayer();

    uint8_t currentWorm;
    uint8_t currentTeam{0};
    Physics physics;
    Stage stage;
    std::vector<Girder> girders;
    std::vector<Player> players;
    std::vector<std::uint32_t> teamHealths;
    const double maxTurnTime;
    bool processingClientInputs{false};
    uint8_t currentWormToFollow{0};
    bool currentPlayerShot{false};
    GameTeams teams;
    std::list<Bullet> bullets;
    Config::Wind wind;

    std::vector<uint8_t> deadTeams;
    GameClock gameClock;
    GameTurn gameTurn;

    /* communication */
    std::vector<std::thread> inputThreads;
    std::vector<std::thread> outputThreads;
    std::vector<CommunicationSocket> &sockets;
    std::vector<PlayerInput> inputs;
    std::vector<GameSnapshot> snapshots;
    std::uint8_t playersConnected;
    bool removeBullets{false};
    bool gameEnded{false};
    std::uint8_t winnerTeam{0};
    bool waitingForNextTurn{true};

void playerDisconnected(uint8_t teamDisconnected);
};
}  // namespace Worms

#endif  //__GAME_H__
```

```
//
// Created by rodrigo on 15/06/18.
//

#include <fstream>
#include <iostream>

#include "GameLobbyAssistant.h"
#include <GameStateMsg.h>
#include "Protocol.h"
#include "Lobbies.h"
#include "GamesGetter.h"

Worms::GameLobbyAssistant::GameLobbyAssistant(CommunicationSocket &&communicatio
nSocket, Lobbies &lobbies, int id,
                                              Observer *lobbyObs) :
        protocol(communicationSocket),
        lobbies(lobbies),
        playerID(id) {
    this->lobbyObservers.emplace_back(lobbyObs);
    this->lobbyObservers.emplace_back(this);
}

void Worms::GameLobbyAssistant::run() {
    try {
        std::uint8_t command{COMMAND_GET_LEVELS};
        while (!this->quit) {
            this->protocol >> command;
            switch (command) {
                case COMMAND_GET_LEVELS:
                    this->getLevels();
                    break;
                case COMMAND_CREATE_GAME:
                    this->createGame();
                    break;
                case COMMAND_GET_GAMES:
                    this->getGames();
                    break;
                case COMMAND_JOIN_GAME:
                    this->joinGame();
                    break;
            }
        }
//        this->createGame();
//        this->createGame();
//        this->createGame();
//        this->getGames();
    } catch (std::exception &e) {
        std::cerr << "In GameLobbyAssistant::run()" << std::endl;
        std::cerr << e.what() << std::endl;
    } catch (...) {
        std::cerr << "Unkown error in GameLobbyAssistant::run()" << std::endl;
    }
}

void Worms::GameLobbyAssistant::stop() {
    this->finished = true;
    this->protocol.stopCommunication();
}

void Worms::GameLobbyAssistant::getLevels() {
//    std::vector<IO::LevelInfo> levelsInfo;
```



```
//    IO::LevelInfo levelInfo{"First Stage", 2};
//    levelsInfo.emplace_back(levelInfo);
//    levelInfo = {"Second Stage", 3};
//    levelsInfo.emplace_back(levelInfo);
//    levelInfo = {"Third Stage", 4};
//    levelsInfo.emplace_back(levelInfo);


    this->protocol << this->lobbies.getLevels();
}

void Worms::GameLobbyAssistant::createGame() {
    uint8_t levelSelected{0};
    this->protocol >> levelSelected;
    this->sendLevelFiles(levelSelected);

    this->lobbies.createGame(this->playerID, this->lobbyObservers, levelSelected
);
    this->quit = true;
}

void Worms::GameLobbyAssistant::getGames() {
    GamesGetter getter;
    this->lobbies.getGames(getter);
    this->protocol << getter.gamesInfo;
}

void Worms::GameLobbyAssistant::joinGame() {
    std::uint8_t gameID{0};
    std::uint8_t levelID{0};
    this->protocol >> gameID;
    this->protocol >> levelID;
    this->sendLevelFiles(levelID);
    this->lobbies.joinGame(gameID, this->playerID, this);
    this->quit = true;
}

void Worms::GameLobbyAssistant::onNotify(Subject &subject, Event event) {
    switch (event) {
        case Event::NewPlayer: {
            auto &lobby = dynamic_cast<Lobby &>(subject);
            this->protocol << lobby.getActualPlayers();
            break;
        }
        default: {
            break;
        }
    }
}

CommunicationSocket Worms::GameLobbyAssistant::getSocket() {
    return std::move(this->protocol.getSocket());
}

int Worms::GameLobbyAssistant::getPlayerID() const{
    return this->playerID;
}

bool Worms::GameLobbyAssistant::itsOver() const{
    return this->finished;
}
```

```
void Worms::GameLobbyAssistant::sendLevelFiles(uint8_t level) {
    const IO::LevelData &levelData = this->lobbies.getLevelData(level);
    this->protocol << levelData.levelName;
    std::ifstream levelFile(levelData.levelPath, std::ifstream::binary);
    this->protocol << levelFile;

    this->protocol << levelData.backgroundName;
    for (auto &background : levelData.backgroundPath) {
        std::ifstream backgroundFile(background, std::ifstream::binary);
        if (!backgroundFile) {
        }
        this->protocol << backgroundFile;
    }
}
```

```
//
// Created by rodrigo on 15/06/18.
//

#ifndef INC_4_WORMS_GAMELOBBYASSISTANT_H
#define INC_4_WORMS_GAMELOBBYASSISTANT_H


#include <Protocol.h>
#include <sstream>

#include "Thread.h"
#include "Lobbies.h"
#include "Observer.h"

namespace Worms {
    class GameLobbyAssistant : public Thread, public Observer {
    public:
        explicit GameLobbyAssistant(CommunicationSocket &&communicationSocket, L
obbies &lobbies, int id,
                                    Observer *lobbyObs);
        GameLobbyAssistant(GameLobbyAssistant &copy) = delete;
        void run() override;
        void stop() override;
        bool itsOver() const;
        void onNotify(Subject &subject, Event event) override;
        int getPlayerID() const;
        CommunicationSocket getSocket();

    private:
        Protocol<CommunicationSocket> protocol;
        Lobbies &lobbies;
        int playerID;
        std::vector<Observer *> lobbyObservers;
        bool finished{false};

        void getLevels();
        void getGames();
        void joinGame();

        void createGame();

        bool quit{false};

        void sendLevelFiles(uint8_t level);
    };
}

#endif //INC_4_WORMS_GAMELOBBYASSISTANT_H
```

```
//
// Created by rodrigo on 15/06/18.
//

#include <iostream>
#include <dirent.h>

#include "GameLobby.h"
#include "ServerSocket.h"
#include "Lobbies.h"
#include "Game.h"
#include "LobbyJoiner.h"
#include "Stage.h"

Worms::GameLobby::GameLobby(std::string port) :
        serverSocket(port.c_str()) {
    std::cout << "Se bindeo" << std::endl;
}

void Worms::GameLobby::run() {
    std::string path(RESOURCE_PATH);
    std::vector<IO::LevelData> levels;

    Lobbies lobbies{levels};

    LobbyJoiner lobbyJoiner{lobbies, this->msgToJoiner};
    try {
        this->loadLevels(path, levels);
        lobbies.configure();
        lobbyJoiner.start();
        int id = 0;

        while (!quit) {
            this->players.emplace_back(this->serverSocket.accept(), lobbies, id,
 this);
            this->players.back().start();
            id++;

            this->removePlayers();

            std::cout << "hubo una conexiÃ³n" << std::endl;
        }

    } catch (std::exception &e) {
        if (!this->quit){
            std::cerr << "In GameLobby::run()" << std::endl;
            std::cerr << e.what() << std::endl;
        }
    } catch (...) {
        std::cerr << "Unkown error in GameLobby::run()" << std::endl;
    }

    this->killPlayers();
    this->msgToJoiner << IO::ServerInternalMsg{IO::ServerInternalAction::quit};
    lobbyJoiner.join();
}

void Worms::GameLobby::stop() {
    this->quit = true;
    this->serverSocket.shutdown();
}
```



```
void Worms::GameLobby::onNotify(Subject &subject, Event event) {
    switch (event) {
        case Event::StartGame: {
            auto &lobby = dynamic_cast<Lobby &>(subject);

            /** En algÃºn momento le tengo que sacar el socket al GameLobbyAssis
tant
             *  para crear un vector con los sockets de todos los jugadores, que
 es lo que
             *  recibe Game, entonces pienso que es mejor que sea al momento de
iniciar la partida
             *  por si el jugador se arrepiente antes y quiere salir, que el Ass
istant lo pueda
             *  manejar.
             */

            const std::vector<int> &playerIDs = lobby.getPlayerIDs();
            for (auto &playerID : playerIDs) {
                for (auto &player : this->players){
                    if (player.getPlayerID() == playerID){
                        //TODO revisar el lugar donde se setea terminado el hilo
                        lobby.addPlayerSocket(std::move(player.getSocket()));
                        player.stop();
                    };
                }
            }
            lobby.start();

            break;
        }
        case Event::EndGame: {
            this->msgToJoiner << IO::ServerInternalMsg{IO::ServerInternalAction:
:lobbyFinished};
            break;
        }
        default: {
            break;
        }
    }
}

void Worms::GameLobby::removePlayers(){
    std::list<GameLobbyAssistant>::iterator playerIt;
    playerIt = this->players.begin();
    while (playerIt != this->players.end()){
        if (playerIt->itsOver()){
            playerIt->join();
            playerIt = this->players.erase(playerIt);
        } else {
            playerIt++;
        }
    }
}

void Worms::GameLobby::loadLevels(std::string &path, std::vector<IO::LevelData>
&levels) {
    DIR *dir;
    struct dirent *ent;
    if ((dir = opendir(path.c_str())) != NULL) {
        /* print all the files and directories within directory */
        while ((ent = readdir(dir)) != NULL) {
            if (std::string(ent->d_name)[0] != '.') {
```

```
                std::string levelPath(path + ent->d_name + "/");
                this->loadLevel(levelPath, levels);
            }
        }
        closedir (dir);
    } else {
        /* could not open directory */
        throw Exception("Could not open directory: %s", path.c_str());
    }
}

void Worms::GameLobby::loadLevel(std::string &path, std::vector<IO::LevelData> &
levels) {
    DIR *dir;
    struct dirent *ent;
    IO::LevelData level;
    if ((dir = opendir(path.c_str())) != NULL) {
        /* print all the files and directories within directory */
        while ((ent = readdir(dir)) != NULL) {
            if (std::string(ent->d_name)[0] != '.') {
                std::string levelPath(path + ent->d_name);
//                  if (std::string(ent->d_name) == "Background") {
//                      std::string backgroundsPath(levelPath + "/");
//                      this->loadLevelBackground(backgroundsPath, level);
//                  } else {
                std::string levelName(ent->d_name);
                YAML::Node data = YAML::LoadFile(levelPath);
                std::set<char> delims{'/'};
                std::string closeBackgroundFile = data["background"]["closeBackground
File"].as<std::string>();
                level.backgroundName.emplace_back(std::move(this->splitpath(clos
eBackgroundFile, delims)));
                level.backgroundPath.emplace_back(std::move(closeBackgroundFile)
);
                std::string midBackgroundFile = data["background"]["midBackgroundFile
"].as<std::string>();
                level.backgroundName.emplace_back(std::move(this->splitpath(midB
ackgroundFile, delims)));
                level.backgroundPath.emplace_back(std::move(midBackgroundFile));
                std::string fartherBackgroundFile = data["background"]["fartherBackgro
undFile"].as<std::string>();
                level.backgroundName.emplace_back(std::move(this->splitpath(fart
herBackgroundFile, delims)));
                level.backgroundPath.emplace_back(std::move(fartherBackgroundFil
e));



//                      levelName = levelName.substr(0, levelName.find('.'));

                level.levelPath = std::move(levelPath);
                level.levelName = std::move(levelName);
//                  }
            }
        }
        closedir (dir);
        levels.emplace_back(std::move(level));
    } else {
        /* could not open directory */
        throw Exception("Could not open directory: %s", path.c_str());
    }
}
```



```
std::string Worms::GameLobby::splitpath(const std::string &str, const std::set<c
har> &delimiters) {
    std::vector<std::string> result;

    char const* pch = str.c_str();
    char const* start = pch;
    for(; *pch; ++pch) {
        if (delimiters.find(*pch) != delimiters.end()) {
            if (start != pch) {
                std::string str(start, pch);
                result.push_back(str);
            } else {
                result.emplace_back("");
            }
            start = pch + 1;
        }
    }
    result.emplace_back(start);

    return result.back();
}

void Worms::GameLobby::loadLevelBackground(std::string &path, IO::LevelData &lev
el) {
    DIR *dir;
    struct dirent *ent;
    std::vector<std::string> backgrounds;
    if ((dir = opendir(path.c_str())) != NULL) {
        /* print all the files and directories within directory */
        while ((ent = readdir(dir)) != NULL) {
            if (std::string(ent->d_name)[0] != '.') {
                std::string backgroundPath(path + ent->d_name);
                std::string backgroundName(ent->d_name);

                level.backgroundPath.emplace_back(std::move(backgroundPath));
                level.backgroundName.emplace_back(std::move(backgroundName));
            }
        }
        closedir (dir);
    } else {
        /* could not open directory */
        throw Exception("Could not open directory: %s", path.c_str());
    }
}

void Worms::GameLobby::killPlayers(){
    std::list<GameLobbyAssistant>::iterator playerIt;
    playerIt = this->players.begin();
    while (playerIt != this->players.end()){
        playerIt->stop();
        playerIt->join();
        playerIt++;
    }
    this->players.erase(this->players.begin(), this->players.end());
}
```

```
//
// Created by rodrigo on 15/06/18.
//

#ifndef INC_4_WORMS_GAMELOBBY_H
#define INC_4_WORMS_GAMELOBBY_H


#define RESOURCE_PATH "/var/Worms/res/"

#include <list>
#include <string>

#include <CommunicationSocket.h>
#include <thread>
#include <GameStateMsg.h>
#include <Stream.h>
#include "ServerSocket.h"
#include "GameLobbyAssistant.h"

namespace Worms {
    class GameLobby : public Observer, public Thread {
    public:
        GameLobby(std::string port);
        GameLobby(GameLobby &copy) = delete;

        void run() override;
        void onNotify(Subject &subject, Event event) override;
        void stop() override;

    private:
        ServerSocket serverSocket;
        IO::Stream<IO::ServerInternalMsg> msgToJoiner;
        std::list<GameLobbyAssistant> players;
        bool quit{false};
        /**
         * @brief check if the GameLobbyAssistant thread is over. If so, join
         * it and erase it (because the sockets was already moved to the Lobby)
         */
        void removePlayers();

        void loadLevels(std::string &path, std::vector<IO::LevelData> &levels);
        void loadLevel(std::string &path, std::vector<IO::LevelData> &levels);
        std::string splitpath(const std::string &str, const std::set<char> &deli
miters);
        void loadLevelBackground(std::string &path, IO::LevelData &level);

        void killPlayers();
    };
}


#endif //INC_4_WORMS_GAMELOBBY_H
```

```cpp
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 17/06/18
 */

#include "GamesGetter.h"

void GamesGetter::operator()(const std::list<Worms::Lobby> &lobbies){
    for (auto &lobby : lobbies) {
        auto &levelInfo = lobby.getLevelInfo();
        IO::GameInfo gameInfo{lobby.getID(),
                              levelInfo.id,
                              levelInfo.name,
                              lobby.getActualPlayers(),
                              levelInfo.playersQuantity};
        this->gamesInfo.emplace_back(gameInfo);
    }
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 17/06/18
 */

#ifndef __GamesGetter_H__
#define __GamesGetter_H__

#include <list>
#include <string>

#include "GameStateMsg.h"
#include "Lobby.h"

struct GamesGetter{
public:
    void operator()(const std::list<Worms::Lobby> &lobbies);
    std::vector<IO::GameInfo> gamesInfo;
};


#endif //__GamesGetter_H__
```

```
//
// Created by rodrigo on 3/06/18.
//

#include "GameTeams.h"
#include <random>

void Worms::GameTeams::makeTeams(std::vector<Worms::Player> &players, uint8_t nu
mTeams,
                                 const std::map<Worm::WeaponID, std::int16_t> &a
mmoCounter) {
    uint8_t numPlayers = players.size();

    this->teams.reserve(numTeams);
    std::vector<uint8_t> playersNum(numPlayers);
    for (uint8_t i = 0; i < numPlayers; i++) {
        playersNum[i] = i;
    }

    std::random_device rnd_device;
    std::mt19937 mersenne_engine(rnd_device());

    shuffle(playersNum.begin(), playersNum.end(), mersenne_engine);

    uint8_t maxTeamPlayers =
        (numPlayers % numTeams == 0) ? numPlayers / numTeams : numPlayers / numT
eams + 1;
    std::vector<uint8_t> numPlayersPerTeam(numTeams);
    for (uint8_t i = 0, nP = numPlayers, nT = numTeams; i < numPlayersPerTeam.si
ze(); i++) {
        numPlayersPerTeam[i] = nP / nT;
        nP -= numPlayersPerTeam[i];
        nT--;
    }
    std::vector<uint8_t> playerIDs;
    for (uint8_t i = 0, currentTeam = 0; i < numPlayers; i++) {
        //        this->teams[currentTeam].players.emplace_back(players[playersN
um[i]].getId());
        playerIDs.emplace_back(players[playersNum[i]].getId());
        players[playersNum[i]].setTeamID(currentTeam);
        if (numPlayersPerTeam[currentTeam] < maxTeamPlayers) {
            players[playersNum[i]].increaseHealth(25.0f);
        }
        if (playerIDs.size() == numPlayersPerTeam[currentTeam]) {
            this->teams.emplace_back(playerIDs, players, ammoCounter);
            playerIDs.clear();
            currentTeam++;
        }
    }
}

void Worms::GameTeams::checkAlive(std::vector<Worms::Player> &players) {
    std::uint8_t teamID{0};
    for (auto &team : this->teams) {
        team.checkAlive(players);
        if (!team.isAlive() && std::find(this->deadTeams.begin(), this->deadTeam
s.end(), teamID) == this->deadTeams.end()) {
            this->deadTeams.emplace_back(teamID);
        }
        teamID++;
    }
}
```



```
bool Worms::GameTeams::endTurn(std::vector<Player> &players) {
    this->checkAlive(players);

    do {
        this->currentTeam = (this->currentTeam + 1) % this->teams.size();
    } while (!this->teams[this->currentTeam].isAlive());

    this->teams[this->currentTeam].endTurn(players);

    if (this->deadTeams.size() >= this->teams.size() - 1) {
        return true;
    } else {
        return false;
    }
}

std::vector<std::uint32_t> Worms::GameTeams::getTotalHealth(std::vector<Worms::P
layer> &players) {
    uint8_t i{0};
    std::vector<std::uint32_t> teamHealths;
    for (auto &team : this->teams) {
        teamHealths.emplace_back(team.calculateTotalHealth(players));
        i++;
    }
    return std::move(teamHealths);
}

uint8_t Worms::GameTeams::getCurrentPlayerID() {
    return this->teams[this->currentTeam].getCurrentPlayerID();
}

uint8_t Worms::GameTeams::getCurrentTeamID() {
    return this->currentTeam;
}

uint8_t Worms::GameTeams::getWinner() {
    std::uint8_t winner{0};
    for (auto &team : this->teams) {
        if (team.isAlive()) {
            return winner;
        }
        winner++;
    }
    return winner;
}

std::uint8_t Worms::GameTeams::getTeamQuantity() const {
    return (std::uint8_t) this->teams.size();
}

Worms::Team &Worms::GameTeams::getCurrentTeam() {
    return this->teams[this->currentTeam];
}

void Worms::GameTeams::weaponUsed(const Worm::WeaponID weaponID) {
    this->teams[this->currentTeam].weaponUsed(weaponID);
}

void Worms::GameTeams::serialize(IO::GameStateMsg &msg) const {
    this->teams[this->currentTeam].serialize(msg);
}
```

```
void Worms::GameTeams::kill(uint8_t team, std::vector<Worms::Player> &players) {
    this->teams[team].kill(players);
}
```

```
//
// Created by rodrigo on 3/06/18.
//

#ifndef INC_4_WORMS_TEAMS_H
#define INC_4_WORMS_TEAMS_H

#include <vector>
#include "Player.h"
#include "Team.h"

namespace Worms {
class GameTeams {
   public:
    GameTeams() = default;
    ~GameTeams(){};
    void makeTeams(std::vector<Player> &players, uint8_t numTeams,
                   const std::map<Worm::WeaponID, std::int16_t> &ammoCounter);
    void checkAlive(std::vector<Player> &players);
    bool endTurn(std::vector<Player> &players);
    uint8_t getCurrentPlayerID();
    std::uint8_t getTeamQuantity() const;
    uint8_t getCurrentTeamID();
    Team &getCurrentTeam();
    std::uint8_t getWinner();
    std::vector<std::uint32_t> getTotalHealth(std::vector<Worms::Player> &player
s);
    void weaponUsed(const Worm::WeaponID weaponID);
    void serialize(IO::GameStateMsg &msg) const;

    void kill(uint8_t team, std::vector<Player> &players);

private:
    std::vector<Team> teams;
    std::vector<std::uint8_t> deadTeams;
    uint8_t currentTeam{0};
};
}

#endif  // INC_4_WORMS_TEAMS_H
```

```
//
// Created by rodrigo on 10/06/18.
//

#include "GameTurn.h"
#include "Config/Config.h"
#include "GameStateMsg.h"
#include "GameStates/ImpactOnCourse.h"
#include "GameStates/PlayerShot.h"
#include "GameStates/StartTurn.h"

Worms::GameTurn::GameTurn(Observer &game) : game(game) {
    this->state = std::shared_ptr<GameTurnState>(new StartTurn());
    this->state->addObserver(&this->game);
}

void Worms::GameTurn::playerShot(Worm::WeaponID weaponID) {
    this->stateID = GameTurnStateID::PlayerShot;
    this->newState = true;

    switch (weaponID) {
        case Worm::WeaponID::WMortar:
            this->bulletFragments = ::Game::Config::getInstance().getMortarFragm
entQuantity();
            break;
        case Worm::WeaponID::WCluster:
            this->bulletFragments = ::Game::Config::getInstance().getClusterFrag
mentQuantity();
            break;
        case Worm::WeaponID::WAerial:
            this->bulletFragments = ::Game::Config::getInstance().getAerialAttac
kMissileQuantity();
            break;
        default:
            break;
    }
}

void Worms::GameTurn::endTurn() {
    this->state->endTurn(*this);
}

void Worms::GameTurn::wormHit(uint8_t wormId) {
    this->state->wormHit(*this, wormId);
}

void Worms::GameTurn::explosion() {
    if (this->stateID != GameTurnStateID::ImpactOnCourse) {
        this->stateID = GameTurnStateID::ImpactOnCourse;
        this->state = std::shared_ptr<GameTurnState>(new ImpactOnCourse(this->bu
lletFragments));
        this->state->addObserver(&this->game);
    }
    this->state->explosion();
}

void Worms::GameTurn::wormEndHit(uint8_t wormId) {
    this->state->wormEndHit(*this, wormId);
}

void Worms::GameTurn::wormDrowning(uint8_t wormId) {
    this->state->wormDrowning(*this, wormId);
```



```
}

void Worms::GameTurn::wormDrowned(uint8_t wormId) {
    this->state->wormDrowned(*this, wormId);
}

void Worms::GameTurn::restart() {
    this->stateID = GameTurnStateID::StartTurn;
    this->newState = true;
    this->bulletFragments = 1;
}

void Worms::GameTurn::update(float dt) {
    if (this->newState) {
        switch (this->stateID) {
            case GameTurnStateID::StartTurn:
                this->state = std::shared_ptr<GameTurnState>(new StartTurn());
                break;
            case GameTurnStateID::PlayerShot:
                this->state = std::shared_ptr<GameTurnState>(new PlayerShot());
                break;
            case GameTurnStateID::ImpactOnCourse:
                break;
        }
        this->state->addObserver(&this->game);
        this->newState = false;
    }
    this->state->update(dt);
}

void Worms::GameTurn::wormFalling(uint8_t wormId) {
    this->state->wormFalling(wormId);
}

void Worms::GameTurn::wormLanded(uint8_t wormId) {
    this->state->wormLanded(wormId);
}

void Worms::GameTurn::wormDead() {
    this->state->wormDead();
}

void Worms::GameTurn::wormDying() {
    this->state->wormDying();
}

void Worms::GameTurn::playerDisconnected(uint8_t wormId) {
    this->state->wormDisconnectedDying(wormId);
}

void Worms::GameTurn::playerDisconnectedDead(uint8_t wormId) {
    this->state->wormDisconnectedDead(wormId);
}
```

```
//
// Created by rodrigo on 10/06/18.
//

#ifndef INC_4_WORMS_GAMETURN_H
#define INC_4_WORMS_GAMETURN_H

#include <memory>
#include "GameStates/GameTurnState.h"
#include "Subject.h"

namespace Worms {
enum class GameTurnStateID { StartTurn, PlayerShot, ImpactOnCourse };
class Game;
class GameTurn : public Subject {
   public:
    GameTurn(Observer &game);
    ~GameTurn() override = default;

    void playerShot(Worm::WeaponID weaponID);
    void endTurn();
    void wormHit(uint8_t wormId);
    void explosion();
    void wormEndHit(uint8_t wormId);
    void wormDrowning(uint8_t wormId);
    void wormDrowned(uint8_t wormId);
    void restart();
    void update(float dt);
    void wormFalling(uint8_t wormId);
    void wormLanded(uint8_t wormId);
    void wormDead();
    void wormDying();
    void playerDisconnected(uint8_t wormId);
    void playerDisconnectedDead(uint8_t wormId);

private:
    std::shared_ptr<GameTurnState> state{nullptr};
    Observer &game;
    GameTurnStateID stateID;
    bool newState{false};
    uint8_t bulletFragments{1};
};
}

#endif  // INC_4_WORMS_GAMETURN_H
```

```
//
// Created by rodrigo on 10/06/18.
//

#include <algorithm>

#include "GameTurnState.h"

Worms::GameTurnState::GameTurnState() {}

void Worms::GameTurnState::wormFalling(uint8_t wormId) {
    this->wormsFalling.emplace_back(wormId);
}

void Worms::GameTurnState::wormLanded(uint8_t wormId) {
    this->wormsFalling.erase(
        std::remove(this->wormsFalling.begin(), this->wormsFalling.end(), wormId
),
        this->wormsFalling.end());
}

void Worms::GameTurnState::wormDead() {
    this->wormsDying--;
}

void Worms::GameTurnState::wormDying() {
    this->wormsDying++;
}

uint8_t Worms::GameTurnState::getWormToFollow() const {
    return this->wormToFollow;
}
```

```
//
// Created by rodrigo on 10/06/18.
//

#ifndef INC_4_WORMS_GAMETURNSTATE_H
#define INC_4_WORMS_GAMETURNSTATE_H

#include <cstdint>
#include <vector>

#include "../../../libs/Subject.h"
#include "GameStateMsg.h"

namespace Worms {
class GameTurn;
class GameTurnState : public Subject {
   public:
    GameTurnState();
    virtual ~GameTurnState() = default;

    virtual void endTurn(GameTurn &gt) = 0;
    virtual void update(float dt) = 0;
    virtual void wormHit(GameTurn &gt, uint8_t wormId) = 0;
    virtual void wormEndHit(GameTurn &gt, uint8_t wormId) = 0;
    virtual void wormDrowning(GameTurn &gt, uint8_t wormId) = 0;
    virtual void wormDrowned(GameTurn &gt, uint8_t wormId) = 0;
    virtual void explosion() = 0;
    virtual void wormFalling(uint8_t wormId);
    virtual void wormLanded(uint8_t wormId);
    virtual void wormDying();
    virtual void wormDead();
    virtual void wormDisconnectedDying(uint8_t wormId) = 0;
    virtual void wormDisconnectedDead(uint8_t wormId) = 0;
    virtual uint8_t getWormToFollow() const;

   protected:
    std::vector<uint8_t> wormsFalling;
    std::vector<uint8_t> wormsDrowning;
    uint8_t wormsDying{0};
    std::vector<uint8_t> wormsDisconnectedDying;
    uint8_t wormToFollow{0};
};
}

#endif  // INC_4_WORMS_GAMETURNSTATE_H
```

```
#include "Girder.h"

Worms::Girder::Girder(const Worms::GirderData &data, Worms::Physics &physics)
    : PhysicsEntity(EntityID::EtGirder), angle(data.angle) {
    b2PolygonShape poly;

    b2BodyDef bdef;
    bdef.type = b2_staticBody;
    bdef.position.Set(0.0f, 0.0f);
    b2Body *staticBody = physics.createBody(bdef);

    b2FixtureDef fixture;
    fixture.density = 1;
    fixture.shape = &poly;

    poly.SetAsBox(data.length / 2, data.height / 2, b2Vec2(data.pos.x, data.pos.
y),
                  data.angle * (PI / 180.0f));
    staticBody->CreateFixture(&fixture);

    staticBody->SetUserData(this);
}

Worms::Girder::Girder(Worms::Girder &&other) noexcept :
        PhysicsEntity(other.id), angle(other.angle){
    this->handlingContact = other.handlingContact;
    this->numObservers = other.numObservers;
    this->observers = std::move(other.observers);
}
```

```
#ifndef GIRDER_H_
#define GIRDER_H_

#include "Physics.h"
#include "PhysicsEntity.h"
#include "Stage.h"

namespace Worms {
class Girder : public PhysicsEntity {
   public:
    const float angle;

    Girder(const Worms::GirderData &data, Physics &physics);
    Girder(Girder &copy) = delete;
    Girder(Girder &&other) noexcept;
    ~Girder() = default;
};
}  // namespace Worms

#endif
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#include "Grenade.h"
#include "../Player.h"

Weapon::Grenade::Grenade(float angle)
    : Worms::Weapon(Game::Config::getInstance().getGreenGrenadeConfig(), Worm::W
eaponID::WGrenade,
                    angle) {
    this->powerChargeTime = Game::Config::getInstance().getPowerChargeTime();
}

void Weapon::Grenade::update(float dt) {
    if (this->increaseShotPower) {
        if (this->shotPower < this->config.maxShotPower) {
            this->shotPower += dt / this->powerChargeTime * this->config.maxShot
Power;
        }
    }
}

void Weapon::Grenade::startShot(Worms::Player *player) {
    this->increaseShotPower = true;
}

void Weapon::Grenade::endShot() {
    this->increaseShotPower = false;
    this->shotPower = 0;
}

void Weapon::Grenade::setTimeout(uint8_t time) {
    this->timeLimit = time;
}

std::list<Worms::Bullet> Weapon::Grenade::onExplode(const Worms::Bullet &bullet,
                                                    Worms::Physics &physics) {
    return std::move(std::list<Worms::Bullet>());
}

void Weapon::Grenade::positionSelected(Worms::Player &p, Math::Point<float> poin
t) {}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#ifndef __GRENADE_H__
#define __GRENADE_H__

#include "Weapon.h"

namespace Weapon {
class Grenade : public Worms::Weapon {
   public:
    Grenade(float angle);
    ~Grenade() override = default;
    void update(float dt) override;
    void startShot(Worms::Player *player) override;
    void endShot() override;
    void setTimeout(uint8_t time) override;
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &bullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override;

   private:
    float powerChargeTime{0.0f};
};
}  // namespace Weapon

#endif  //__GRENADE_H__
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#include "Hit.h"
#include "../Player.h"

Worms::Hit::Hit() : State(Worm::StateID::Hit) {}

void Worms::Hit::update(Worms::Player &p, float dt, b2Body *body) {
    /*
     * when the worm lands (there was a collision between the worm and the
     * girder) it has to change its state to still, and take an impulse
     * of equal absolute value and different sign of the impulse taken in
     * hit stage (remember, the worm has a friction coefficient 0).
     *
     * In the y-axis there will be no impulse because its velocity was
     * cancelled because of the collision with the girder.
     */
    if (p.isOnGround()) {
        this->timeElapsed += dt;
        if (this->timeElapsed > 0.7f) {
            float32 mass = body->GetMass();
            b2Vec2 previousVel = body->GetLinearVelocity();
            b2Vec2 impulses = {mass * (0.0f - previousVel.x), 0.0f};
            body->ApplyLinearImpulseToCenter(impulses, true);

            p.notify(p, Event::EndHit);
            p.setState(Worm::StateID::Land);
        }
    } else {
        this->timeElapsed = 0.0f;
    }
}

void Worms::Hit::moveRight(Worms::Player &p) {}

void Worms::Hit::moveLeft(Worms::Player &p) {}

void Worms::Hit::jump(Worms::Player &p) {}

void Worms::Hit::stopMove(Worms::Player &p) {}

void Worms::Hit::backFlip(Worms::Player &p) {}

void Worms::Hit::bazooka(Worms::Player &p) {}

void Worms::Hit::pointUp(Worms::Player &p) {}

void Worms::Hit::pointDown(Worms::Player &p) {}

void Worms::Hit::startShot(Worms::Player &p) {}

void Worms::Hit::endShot(Worms::Player &p) {}

void Worms::Hit::grenade(Worms::Player &p) {}

void Worms::Hit::cluster(Worms::Player &p) {}

void Worms::Hit::mortar(Worms::Player &p) {}
```



```
void Worms::Hit::banana(Worms::Player &p) {}

void Worms::Hit::holy(Worms::Player &p) {}

void Worms::Hit::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Hit::aerialAttack(Worms::Player &p) {}

void Worms::Hit::dynamite(Worms::Player &p) {}

void Worms::Hit::teleport(Worms::Player &p) {}

void Worms::Hit::baseballBat(Worms::Player &p) {}
```

```
/*
 *  Created by Rodrigo.
 *  date: 28/05/18
 */

#ifndef __Hit_H__
#define __Hit_H__

#include <cstdint>
#include "PlayerState.h"

namespace Worms {
class Hit : public State {
   public:
    Hit();
    ~Hit() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
};
}  // namespace Worms

#endif  //__Hit_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#include "Holy.h"
#include "../Player.h"

Weapon::Holy::Holy(float angle)
    : Worms::Weapon(Game::Config::getInstance().getHolyConfig(), Worm::WeaponID:
:WHoly, angle) {
    this->powerChargeTime = Game::Config::getInstance().getPowerChargeTime();
}

void Weapon::Holy::update(float dt) {
    if (this->increaseShotPower) {
        if (this->shotPower < this->config.maxShotPower) {
            this->shotPower += dt / this->powerChargeTime * this->config.maxShot
Power;
        }
    }
}

void Weapon::Holy::startShot(Worms::Player *player) {
    this->increaseShotPower = true;
}

void Weapon::Holy::endShot() {
    this->increaseShotPower = false;
    this->shotPower = 0;
}

void Weapon::Holy::setTimeout(uint8_t time) {
    this->timeLimit = time;
}

std::list<Worms::Bullet> Weapon::Holy::onExplode(const Worms::Bullet &bullet,
                                                 Worms::Physics &physics) {
    return std::move(std::list<Worms::Bullet>());
}

void Weapon::Holy::positionSelected(Worms::Player &p, Math::Point<float> point)
{}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#ifndef __Holy_H__
#define __Holy_H__

#include "Weapon.h"

namespace Weapon {
class Holy : public Worms::Weapon {
   public:
    Holy(float angle);
    ~Holy() override = default;
    void update(float dt) override;
    void startShot(Worms::Player *player) override;
    void endShot() override;
    void setTimeout(uint8_t time) override;
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &bullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override;

   private:
    float powerChargeTime{0.0f};
};
}  // namespace Weapon

#endif  //__Holy_H__
```

```
//
// Created by rodrigo on 10/06/18.
//

#include <algorithm>
#include <iostream>

#include "GameStateMsg.h"
#include "ImpactOnCourse.h"

void Worms::ImpactOnCourse::endTurn(GameTurn &gt) {
    if (this->impactEnded && this->wormsFalling.size() == 0 && this->wormsDrowni
ng.size() == 0 &&
        !this->wormsDying && this->wormsDisconnectedDying.size() == 0) {
        this->notify(*this, Event::TurnEnded);
    }
}

void Worms::ImpactOnCourse::wormHit(GameTurn &gt, uint8_t wormId) {
    this->wormsStillHit.emplace_back(wormId);
    this->wormsHit.emplace_back(wormId);
    if (this->wormToFollow != this->wormsStillHit[0]) {
        this->wormToFollow = this->wormsStillHit[0];
        this->notify(*this, Event::NewWormToFollow);
    }
}

void Worms::ImpactOnCourse::wormEndHit(Worms::GameTurn &gt, uint8_t wormId) {
    this->wormsStillHit.erase(
        std::remove(this->wormsStillHit.begin(), this->wormsStillHit.end(), worm
Id),
        this->wormsStillHit.end());
    if (this->wormToFollow == wormId) {
        this->wormToFollow = this->wormsStillHit[0];
        this->notify(*this, Event::NewWormToFollow);
    }
}

void Worms::ImpactOnCourse::wormDrowning(Worms::GameTurn &gt, uint8_t wormId) {
    this->wormsDrowning.emplace_back(wormId);
    this->wormLanded(wormId);
    if (this->wormsStillHit.size() == 0) {
        if (this->wormToFollow != this->wormsDrowning[0]) {
            this->wormToFollow = this->wormsDrowning[0];
            this->notify(*this, Event::NewWormToFollow);
        }
    }
}

void Worms::ImpactOnCourse::wormDrowned(Worms::GameTurn &gt, uint8_t wormId) {
    this->wormsDrowning.erase(
        std::remove(this->wormsDrowning.begin(), this->wormsDrowning.end(), worm
Id),
        this->wormsDrowning.end());
    if (this->wormsStillHit.size() == 0) {
        if (this->wormToFollow != this->wormsDrowning[0]) {
            this->wormToFollow = this->wormsDrowning[0];
            this->notify(*this, Event::NewWormToFollow);
        }
    }
}
```



```
std::vector<uint8_t> &Worms::ImpactOnCourse::getWormsHit() {
    return this->wormsHit;
}

void Worms::ImpactOnCourse::impactNotEnded() {
    this->impactEnded = false;
}

Worms::ImpactOnCourse::ImpactOnCourse(uint8_t bulletFragments) {
    this->bulletFragments = bulletFragments;
}

void Worms::ImpactOnCourse::explosion() {
    this->fragmentExplosions++;
}

void Worms::ImpactOnCourse::update(float dt) {
    if (!this->impactEnded) {
        if (this->wormsStillHit.size() == 0 && this->wormsDrowning.size() == 0 &
&
            this->fragmentExplosions == this->bulletFragments) {
            this->impactEnded = true;
            this->notify(*this, Event::ImpactEnd);
        }
    }
}

void Worms::ImpactOnCourse::wormDisconnectedDying(uint8_t wormId) {
    this->wormsDisconnectedDying.emplace_back(wormId);
    if (this->wormToFollow != this->wormsDisconnectedDying[0] && this->wormsFall
ing.size() == 0 && this->wormsDrowning.size() == 0) {
        this->wormToFollow = this->wormsDisconnectedDying[0];
        this->notify(*this, Event::NewWormToFollow);
    }
}

void Worms::ImpactOnCourse::wormDisconnectedDead(uint8_t wormId) {
    this->wormsDisconnectedDying.erase(
            std::remove(this->wormsDisconnectedDying.begin(), this->wormsDisconn
ectedDying.end(), wormId),
            this->wormsDisconnectedDying.end());
    if (this->wormToFollow == wormId) {
        this->wormToFollow = this->wormsDisconnectedDying[0];
        this->notify(*this, Event::NewWormToFollow);
    }
}
```

```
//
// Created by rodrigo on 10/06/18.
//

#ifndef INC_4_WORMS_IMPACTONCOURSE_H
#define INC_4_WORMS_IMPACTONCOURSE_H

#include <GameStateMsg.h>
#include <vector>
#include "../../../libs/Observer.h"
#include "GameTurnState.h"

namespace Worms {
class ImpactOnCourse : public GameTurnState {
   public:
    ImpactOnCourse(uint8_t bulletFragments);
    ~ImpactOnCourse() = default;

    void endTurn(GameTurn &gt) override;
    void update(float dt) override;
    void wormHit(GameTurn &gt, uint8_t wormId) override;
    void wormEndHit(GameTurn &gt, uint8_t wormId) override;
    void wormDrowning(GameTurn &gt, uint8_t wormId) override;
    void wormDrowned(GameTurn &gt, uint8_t wormId) override;
    void explosion() override;
    void wormDisconnectedDying(uint8_t wormId) override;
    void wormDisconnectedDead(uint8_t wormId) override;
    std::vector<uint8_t> &getWormsHit();
    void impactNotEnded();

   private:
    std::vector<uint8_t> wormsStillHit;
    std::vector<uint8_t> wormsHit;
//    uint8_t wormToFollow{0};
    bool impactEnded{false};
    uint8_t bulletFragments{0};
    uint8_t fragmentExplosions{0};
};
}

#endif  // INC_4_WORMS_IMPACTONCOURSE_H
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 20/05/18
 */

#include <Box2D/Dynamics/b2Body.h>
#include <iostream>
#include <vector>

#include "../Player.h"
#include "Jumping.h"

Worms::Jumping::Jumping(GUI::Position p) : State(Worm::StateID::Jumping), startP
osition(p) {}

void Worms::Jumping::update(Worms::Player &p, float dt, b2Body *body) {
    /*
     * when the worm lands (there was a collision between the worm and the
     * girder) it has to changes its state to endJump, and take an impulse
     * of equal absolute value and different sign of the impulse taken in
     * startJump stage (remember, the worm has a friction coefficient 0).
     *
     * In the y-axis there will be no impulse because its velocity was
     * cancelled because of the collision with the girder.
     */
    if (p.isOnGround()) {
        this->timeElapsed += dt;
    } else {
        this->timeElapsed = 0.0f;
    }
    if (p.isOnGround() || this->timeElapsed > 0.2f) {
        float32 mass = body->GetMass();
        b2Vec2 previousVel = body->GetLinearVelocity();
        b2Vec2 impulses = {mass * (0.0f - previousVel.x), 0.0f};
        body->ApplyLinearImpulseToCenter(impulses, true);

        p.landDamage(this->startPosition.y - p.getPosition().y);
        p.setState(Worm::StateID::Land);
        //        p.setState(Worm::StateID::EndJump);
    }
}

void Worms::Jumping::moveRight(Worms::Player &p) {}

void Worms::Jumping::moveLeft(Worms::Player &p) {}

void Worms::Jumping::jump(Worms::Player &p) {}

void Worms::Jumping::stopMove(Worms::Player &p) {}

void Worms::Jumping::backFlip(Worms::Player &p) {}

void Worms::Jumping::bazooka(Worms::Player &p) {}

void Worms::Jumping::pointUp(Worms::Player &p) {}

void Worms::Jumping::pointDown(Worms::Player &p) {}

void Worms::Jumping::startShot(Worms::Player &p) {}

void Worms::Jumping::endShot(Worms::Player &p) {}
```



```
void Worms::Jumping::grenade(Worms::Player &p) {}

void Worms::Jumping::cluster(Worms::Player &p) {}

void Worms::Jumping::mortar(Worms::Player &p) {}

void Worms::Jumping::banana(Worms::Player &p) {}

void Worms::Jumping::holy(Worms::Player &p) {}

void Worms::Jumping::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Jumping::aerialAttack(Worms::Player &p) {}

void Worms::Jumping::dynamite(Worms::Player &p) {}

void Worms::Jumping::teleport(Worms::Player &p) {}

void Worms::Jumping::baseballBat(Worms::Player &p) {}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 20/05/18
 */

#ifndef __PLAYER_JUMPING_H__
#define __PLAYER_JUMPING_H__

#include <Box2D/Dynamics/b2Body.h>
#include <Camera.h>

#include "PlayerState.h"

namespace Worms {
class Jumping : public State {
   public:
    Jumping(GUI::Position p);
    ~Jumping() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    virtual void pointUp(Player &p) override;
    virtual void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    GUI::Position startPosition;
};
}  // namespace Worms

#endif  //__PLAYER_JUMPING_H__
```

```
//
// Created by rodrigo on 3/06/18.
//

#include "Land.h"
#include "../Config/Config.h"
#include "../Player.h"
#include "PlayerState.h"

Worms::Land::Land()
    : State(Worm::StateID::Land), landTime(Game::Config::getInstance().getLandTi
me()) {}

void Worms::Land::update(Worms::Player &p, float dt, b2Body *body) {
    this->timeElapsed += dt;
    if (this->timeElapsed > this->landTime) {
        p.notify(p, Event::WormLanded);
        if (p.health <= 0) {
            p.notify(p, Event::Dying);
            p.setState(Worm::StateID::Die);
        } else {
            p.setState(Worm::StateID::Still);
        }
    }
}

void Worms::Land::moveRight(Worms::Player &p) {}

void Worms::Land::moveLeft(Worms::Player &p) {}

void Worms::Land::jump(Worms::Player &p) {}

void Worms::Land::stopMove(Worms::Player &p) {}

void Worms::Land::backFlip(Worms::Player &p) {}

void Worms::Land::bazooka(Worms::Player &p) {}

void Worms::Land::pointUp(Worms::Player &p) {}

void Worms::Land::pointDown(Worms::Player &p) {}

void Worms::Land::startShot(Worms::Player &p) {}

void Worms::Land::endShot(Worms::Player &p) {}

void Worms::Land::grenade(Worms::Player &p) {}

void Worms::Land::cluster(Worms::Player &p) {}

void Worms::Land::mortar(Worms::Player &p) {}

void Worms::Land::banana(Worms::Player &p) {}

void Worms::Land::holy(Worms::Player &p) {}

void Worms::Land::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Land::aerialAttack(Worms::Player &p) {}

void Worms::Land::dynamite(Worms::Player &p) {}
```



```
void Worms::Land::teleport(Worms::Player &p) {}

void Worms::Land::baseballBat(Worms::Player &p) {}
```

```
//
// Created by rodrigo on 3/06/18.
//

#ifndef INC_4_WORMS_LAND_H
#define INC_4_WORMS_LAND_H

#include <cstdint>
#include "PlayerState.h"

namespace Worms {
class Land : public State {
   public:
    Land();
    ~Land() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    float landTime;
};
}  // namespace Worms

#endif  // INC_4_WORMS_LAND_H
```

```
//
// Created by rodrigo on 15/06/18.
//

#include <iostream>
#include <yaml-cpp/yaml.h>
#include "GamesGetter.h"
#include "Lobbies.h"

void Worms::Lobbies::createGame(int playerID, std::vector<Observer *> lobbyObser
vers, uint8_t levelSelected) {
    std::lock_guard<std::mutex> lock(this->mutex);
    this->lobbies.emplace_back(playerID, this->idLobby, lobbyObservers, this->le
vels[levelSelected], this->levelsInfo[levelSelected]);
    this->idLobby++;
}

void Worms::Lobbies::getGames(GamesGetter &getter) {
    std::lock_guard<std::mutex> lock(this->mutex);
    getter(this->lobbies);
}

void Worms::Lobbies::joinGame(int gameID, int playerID, Observer *lobbyObserver)
 {
    std::lock_guard<std::mutex> lock(this->mutex);
    auto it = this->lobbies.begin();
    while ((*it).getID() != gameID && it != this->lobbies.end()){
        it++;
    }
    (*it).addLobbyObserver(lobbyObserver);
    (*it).joinGame(playerID);
}

std::list<Worms::Lobby> &Worms::Lobbies::getLobbies() {
    return this->lobbies;
}

Worms::Lobbies::Lobbies(const std::vector<IO::LevelData> &levels) :
        levels(levels) {}

const std::vector<IO::LevelInfo> &Worms::Lobbies::getLevels() {
    return this->levelsInfo;
}

const IO::LevelData & Worms::Lobbies::getLevelData(uint8_t levelSelected) {
    return this->levels[levelSelected];
}

void Worms::Lobbies::configure(){
    uint8_t id{0};
    for (auto &level : this->levels) {
        YAML::Node data = YAML::LoadFile(level.levelPath);
        std::string name = data["name"].as<std::string>();
        uint8_t playersQuantity = static_cast<uint8_t>(data["numPlayers"].as<int>(
));
        IO::LevelInfo levelInfo{id, name, playersQuantity};
        this->levelsInfo.emplace_back(levelInfo);
        id++;
    }
}

Worms::Lobbies::~Lobbies(){}
```

```
//
// Created by rodrigo on 15/06/18.
//

#ifndef INC_4_WORMS_LOBBIES_H
#define INC_4_WORMS_LOBBIES_H


#include <list>
#include <mutex>

#include "GamesGetter.h"
#include "Lobby.h"
#include "Observer.h"

namespace Worms {
    class Lobbies {
    public:
        explicit Lobbies(const std::vector<IO::LevelData> &levels);
        ~Lobbies();
        void configure();
        void createGame(int playerID, std::vector<Observer *> lobbyObservers, ui
nt8_t levelSelected);
        void getGames(GamesGetter &getter);
        void joinGame(int gameID, int playerID, Observer *lobbyObserver);
        const std::vector<IO::LevelInfo> &getLevels();
        const IO::LevelData & getLevelData(uint8_t levelSelected);
        std::list<Lobby> &getLobbies();

    private:
        std::mutex mutex;
        std::list<Lobby> lobbies;
        uint8_t idLobby{0};
        const std::vector<IO::LevelData> &levels;
        std::vector<IO::LevelInfo> levelsInfo;
    };
}


#endif //INC_4_WORMS_LOBBIES_H
```

```
//
// Created by rodrigo on 16/06/18.
//

#include <iostream>
#include <string>

#include "Lobby.h"
#include "Stage.h"
#include "Game.h"


/** Copio por Â¿posible race condition?
 */
Worms::Lobby::Lobby(int playerID, std::uint8_t id, std::vector<Observer *> obs,
const IO::LevelData level,
                    const IO::LevelInfo levelInfo) :
        id(id),
        level(level),
        levelInfo(levelInfo) {
    for (auto *lobbyObserver : obs) {
        this->obs.emplace_back(lobbyObserver);
        this->addObserver(lobbyObserver);
    }
    this->joinGame(playerID);
}

void Worms::Lobby::joinGame(int playerID) {
//    std::lock_guard<std::mutex> lock(this->mutex);
    this->playerIDs.emplace_back(playerID);
    this->actualPlayers++;
    this->notify(*this, Event::NewPlayer);
    if (this->actualPlayers == levelInfo.playersQuantity) {
        this->notify(*this, Event::StartGame);
        std::uint8_t i{0};
        for (auto *obs : this->obs) {
            if (i != 0) {
                this->removeObserver(obs);
            }
            i++;
        }
        this->gameStarted = true;
    }
}

const IO::LevelInfo & Worms::Lobby::getLevelInfo() const{
    return this->levelInfo;
}

std::uint8_t Worms::Lobby::getActualPlayers() const{
    return this->actualPlayers;
}

const std::vector<int> &Worms::Lobby::getPlayerIDs() const{
    return this->playerIDs;
}

std::uint8_t Worms::Lobby::getID() const{
    return this->id;
}

void Worms::Lobby::addPlayerSocket(CommunicationSocket &&player) {
```



```
    this->players.emplace_back(std::move(player));
}

Worms::Lobby::Lobby(Worms::Lobby &&other) noexcept :
        id(other.id),
        level(other.level),
        levelInfo(other.levelInfo) {
    if (this != &other){
        this->actualPlayers = other.actualPlayers;
        this->playerIDs = std::move(other.playerIDs);
        this->players = std::move(other.players);
    }
}

void Worms::Lobby::run() {
    try {
        while (!this->finished) {
            if (this->gameStarted) {
                for (std::uint8_t i = 0; i < levelInfo.playersQuantity; i++) {
                    char buffer[1];
                    buffer[0] = i;
                    this->players[i].send(buffer, sizeof(buffer));
                }
                Worms::Game game{Worms::Stage::fromFile(this->level.levelPath),
this->players};
                game.start();
                this->notify(*this, Event::EndGame);
                this->gameStarted = false;
                this->finished = true;
            }
        }
    } catch (std::exception &e){
        if (!this->finished){
            std::cerr << "In Lobby::run()" << std::endl;
            std::cerr << e.what() << std::endl;
        }
    } catch (...){
        std::cerr << "Unkown error in Lobby::run()" << std::endl;
    }
}

void Worms::Lobby::stop() {
    this->finished = true;
    for(auto &player: this->players){
        player.shutdown();
    }
}

bool Worms::Lobby::itsOver() {
    return this->finished;
}

void Worms::Lobby::addLobbyObserver(Observer *lobbyObserver) {
    this->obs.emplace_back(lobbyObserver);
    this->addObserver(lobbyObserver);
}

bool Worms::Lobby::started() {
    return this->gameStarted;
}
```

```
//
// Created by rodrigo on 16/06/18.
//

#ifndef INC_4_WORMS_LOBBY_H
#define INC_4_WORMS_LOBBY_H


#include <stdint-gcc.h>
#include <string>
#include <vector>
#include <mutex>
#include <GameStateMsg.h>

#include "CommunicationSocket.h"
#include "Subject.h"
#include "Thread.h"

namespace Worms {
    class Lobby : public Thread, public Subject {
    public:
        Lobby(int playerID, std::uint8_t id, std::vector<Observer *> obs, const
IO::LevelData level,
              const IO::LevelInfo levelInfo);
        Lobby(Lobby &&other) noexcept;
        Lobby(Lobby &copy) = delete;

        void run() override;
        void stop() override;
        bool itsOver();

        void joinGame(int playerID);
        const IO::LevelInfo & getLevelInfo() const;
        std::uint8_t getActualPlayers() const;
        const std::vector<int> &getPlayerIDs() const;
        std::uint8_t getID() const;
        void addLobbyObserver(Observer *lobbyObserver);
        bool started();
        void addPlayerSocket(CommunicationSocket &&player);

    private:
        std::mutex mutex;
        const std::uint8_t id;
        std::uint8_t actualPlayers{0};
        std::vector<int> playerIDs;
        std::vector<CommunicationSocket> players;
        std::vector<Observer *> obs;
        const IO::LevelData level;
        const IO::LevelInfo levelInfo;

        bool finished{false};
        bool gameStarted{false};
    };
}


#endif //INC_4_WORMS_LOBBY_H
```

```
//
// Created by rodrigo on 19/06/18.
//

#include <GameStateMsg.h>
#include <iostream>
#include "LobbyJoiner.h"

Worms::LobbyJoiner::LobbyJoiner(Worms::Lobbies &lobbies, IO::Stream<IO::ServerIn
ternalMsg> &serverInput) :
        lobbies(lobbies.getLobbies()),
        serverInput(serverInput) {
}

void Worms::LobbyJoiner::run() {
    try{
        while (!this->finished) {
            IO::ServerInternalMsg msg;
            if (this->serverInput.pop(msg)) {
                this->handleServerInput(msg);
            }
        }
    } catch (std::exception &e){
        if(!this->finished){
            std::cerr << "In LobbyJoiner::run()" << std::endl;
            std::cerr << e.what() << std::endl;
        }
    } catch (...){
        std::cerr << "Unknown error in LobbyJoiner::run()" << std::endl;
    }

    this->killLobbies();
}

void Worms::LobbyJoiner::stop() {
    this->finished = true;
}

void Worms::LobbyJoiner::handleServerInput(IO::ServerInternalMsg &msg) {
    switch (msg.action) {
        case IO::ServerInternalAction::lobbyFinished: {
            std::list<Lobby>::iterator lobbyIt;
            lobbyIt = this->lobbies.begin();
            while (lobbyIt != this->lobbies.end()) {
                if (lobbyIt->itsOver()) {
                    lobbyIt->join();
                    lobbyIt = this->lobbies.erase(lobbyIt);
                } else {
                    lobbyIt++;
                }
            }
            break;
        }
        case IO::ServerInternalAction::quit: {
            this->finished = true;
            break;
        }
    }
}

void Worms::LobbyJoiner::killLobbies(){
    for (auto &lobby: this->lobbies){
```



```
        if (lobby.started()) {
            lobby.stop();
            lobby.join();
        }
    }
    this->lobbies.erase(this->lobbies.begin(), this->lobbies.end());
}
```

```
//
// Created by rodrigo on 19/06/18.
//

#ifndef INC_4_WORMS_LOBBYJOINER_H
#define INC_4_WORMS_LOBBYJOINER_H


#include <GameStateMsg.h>
#include <Stream.h>
#include "Thread.h"
#include "Lobbies.h"

namespace Worms {
    class LobbyJoiner : public Thread {
    public:
        explicit LobbyJoiner(Worms::Lobbies &lobbies, IO::Stream<IO::ServerInter
nalMsg> &serverInput);

        void run() override;
        void stop() override;

    private:
        std::list<Lobby> &lobbies;
        IO::Stream<IO::ServerInternalMsg> &serverInput;
        bool finished{false};

        void handleServerInput(IO::ServerInternalMsg &msg);

        void killLobbies();
    };
}


#endif //INC_4_WORMS_LOBBYJOINER_H
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#include <signal.h>
#include <unistd.h>
#include <cstdlib>
#include <iostream>
#include <string>
#include <thread>
#include <vector>

#include "CommunicationSocket.h"
#include "Game.h"
#include "ServerSocket.h"
#include "GameLobby.h"

static volatile bool quit = false;

/**
 * @brief Signal handler.
 *
 * @param _ unused.
 */
static void _signal_handler(int _) {
    quit = true;
}

///**
// * @brief Thread handler that signals the Game to exit.
// *
// * @param game
// */
//static void _exit_handler(Worms::Game &game) {
//    while (!quit) {
//        usleep(100000);
//    }
//    game.exit();
//}

int main(int argc, const char *argv[]) {
    if (argc != 2) {
        std::cout << "Usage: ./server PORT" << std::endl;
        return EXIT_FAILURE;
    }

    try {
        /* sets a signal handler to exit the program gracefully */
        signal(SIGINT, _signal_handler);
        signal(SIGTERM, _signal_handler);

        std::string port(argv[1]);
        Worms::GameLobby gameLobby{port};

        gameLobby.start();
        char quit{0};
        while (quit != 'q'){
            std::cin >> quit;
        }

        gameLobby.stop();
```



```
        gameLobby.join();

    } catch (std::exception &e) {
        std::cerr << "In main()" << std::endl;
        std::cerr << e.what() << std::endl;
        return 1;
    } catch (...) {
        std::cerr << "Unkown error in main thread" << std::endl;
        return 1;
    }

    return EXIT_SUCCESS;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#include "Mortar.h"
#include "../Player.h"

Weapon::Mortar::Mortar(float angle)
    : Worms::Weapon(Game::Config::getInstance().getMortarConfig(), Worm::WeaponI
D::WMortar, angle),
      fragmentConfig(Game::Config::getInstance().getMortarFragmentConfig()) {}

void Weapon::Mortar::update(float dt) {
    if (this->increaseShotPower) {
        if (this->shotPower >= this->config.maxShotPower) {
            this->shotPower = this->config.maxShotPower;
        } else {
            this->shotPower++;
        }
    }
}

void Weapon::Mortar::startShot(Worms::Player *player) {
    this->increaseShotPower = true;
}

void Weapon::Mortar::endShot() {
    this->increaseShotPower = false;
    this->shotPower = 0;
}

void Weapon::Mortar::setTimeout(uint8_t time) {}

std::list<Worms::Bullet> Weapon::Mortar::onExplode(const Worms::Bullet &mainBull
et,
                                                   Worms::Physics &physics) {
    uint8_t fragmentQuantity = Game::Config::getInstance().getMortarFragmentQuan
tity();
    Math::Point<float> p = mainBullet.getPosition();
    Worms::BulletInfo bulletInfo = {this->fragmentConfig.dmgInfo,
                                    p,
                                    this->fragmentConfig.minAngle,
                                    (float)this->fragmentConfig.maxShotPower,
                                    this->fragmentConfig.bulletRadius * 6,
                                    this->fragmentConfig.restitution,
                                    this->fragmentConfig.friction,
                                    this->fragmentConfig.explotionInitialTimeout
,
                                    Event::Explode,
                                    this->fragmentConfig.bulletRadius,
                                    this->fragmentConfig.bulletDampingRatio,
                                    this->config.windAffected};

    std::list<Worms::Bullet> ret;
    for (int i = 0; i < fragmentQuantity; i++) {
        bulletInfo.angle = i * this->fragmentConfig.angleStep + this->fragmentCo
nfig.minAngle;
        ret.emplace_back(bulletInfo, physics, Worm::WeaponID::WFragment);
    }

    return std::move(ret);
```



```
}

void Weapon::Mortar::positionSelected(Worms::Player &p, Math::Point<float> point
) {}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 03/06/18
 */

#ifndef __Mortar_H__
#define __Mortar_H__

#include "Weapon.h"

namespace Weapon {
class Mortar : public Worms::Weapon {
   public:
    Mortar(float angle);
    ~Mortar() override = default;
    void update(float dt) override;
    void startShot(Worms::Player *player) override;
    void endShot() override;
    void setTimeout(uint8_t time) override;
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &bullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override;

   private:
    const Config::Weapon &fragmentConfig;
};
}  // namespace Weapon

#endif  //__Mortar_H__
```

```cpp
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 22/06/18
 */

#include "P2PWeapon.h"
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 22/06/18
 */

#ifndef __P2PWeapon_H__
#define __P2PWeapon_H__

#include <Direction.h>
#include <Point.h>

#include "BulletConfig.h"

namespace Config {
struct P2PWeapon {
    Bullet::DamageInfo dmgInfo;
    Worm::Direction direction;
    Math::Point<float> position;
    float angle;
};
}  // namespace Config

#endif  //__P2PWeapon_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#include "Physics.h"

Worms::Physics::Physics(b2Vec2 gravity, float timeStep)
    : timeStep(timeStep),
      gravity(gravity),
      world(this->gravity),
      contactEventListener(new ContactEventListener) {
    this->world.SetContactListener(this->contactEventListener.get());
}

/**
 * @brief Updates the physics engine.
 *
 * @param dt Seconds elapsed since last call to this function.
 */
void Worms::Physics::update(float dt) {
    this->accumTime += dt;

    /* updates the physics engine */
    for (int i = 0; i < 5 && this->accumTime > this->timeStep; i++) {
        this->world.Step(this->timeStep, this->vIterations, this->pIterations);
        this->accumTime -= this->timeStep;
    }
}

/**
 * @brief Creates a new physical body.
 *
 * @param bodyDef Body definition.
 * @return new body.
 */
b2Body* Worms::Physics::createBody(b2BodyDef& bodyDef) {
    return this->world.CreateBody(&bodyDef);
}
```

```cpp
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 26/05/18
 */

#include "PhysicsEntity.h"

Worms::PhysicsEntity::PhysicsEntity(Worms::EntityID id) : id(id) {}

Worms::EntityID Worms::PhysicsEntity::getEntityId() {
    return this->id;
}

Worms::PhysicsEntity::PhysicsEntity(Worms::PhysicsEntity &&other){
    this->id = other.id;
    this->handlingContact = other.handlingContact;

    other.handlingContact = false;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 26/05/18
 */

#ifndef PHYSICS_ENTITY_H_
#define PHYSICS_ENTITY_H_

#include "Box2D/Box2D.h"

#include "Subject.h"

namespace Worms {
enum EntityID { EtWorm, EtBullet, Sensor, EtGirder };
class PhysicsEntity : public Subject {
   public:
    explicit PhysicsEntity(EntityID id);
    PhysicsEntity(PhysicsEntity &&other);
    PhysicsEntity(PhysicsEntity &copy) = delete;

    virtual EntityID getEntityId();
    virtual void startContact(Worms::PhysicsEntity *physicsEntity) {}
    virtual void startContact(Worms::PhysicsEntity *physicsEntity, b2Contact &co
ntact) {}
    virtual void endContact(Worms::PhysicsEntity *physicsEntity) {}
    virtual void endContact(Worms::PhysicsEntity *physicsEntity, b2Contact &cont
act) {}
    virtual void contactWith(PhysicsEntity &physicsEntity, b2Contact &contact) {
}

   protected:
    EntityID id;
    bool handlingContact{false};
};
}  // namespace Worms

#endif
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#ifndef __Physics_H__
#define __Physics_H__

#include "Box2D/Box2D.h"

#include <memory.h>
#include "ContactEventListener.h"

namespace Worms {
class Physics {
   public:
    Physics(b2Vec2 gravity, float timeStep);
    ~Physics() = default;
    void update(float dt);
    b2Body *createBody(b2BodyDef &bodyDef);

   private:
    float timeStep;
    float accumTime{0.0f};
    b2Vec2 gravity;
    b2World world;
    std::shared_ptr<ContactEventListener> contactEventListener;
    int32 vIterations{6};
    int32 pIterations{2};
};
}  // namespace Worms

#endif  //__Physics_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#include <Box2D/Box2D.h>
#include <iostream>

#include "Direction.h"
#include "Girder.h"
#include "Physics.h"
#include "Player.h"
#include "Weapons/AerialAttack.h"
#include "Weapons/Banana.h"
#include "Weapons/BaseballBat.h"
#include "Weapons/Bazooka.h"
#include "Weapons/Cluster.h"
#include "Weapons/Dynamite.h"
#include "Weapons/Grenade.h"
#include "Weapons/Holy.h"
#include "Weapons/Mortar.h"
#include "Weapons/Teleport.h"
#include "Weapons/Weapon.h"
#include "WormStates/BackFlipping.h"
#include "WormStates/Batting.h"
#include "WormStates/Dead.h"
#include "WormStates/Die.h"
#include "WormStates/Drowning.h"
#include "WormStates/EndBackFlip.h"
#include "WormStates/EndJump.h"
#include "WormStates/Falling.h"
#include "WormStates/Hit.h"
#include "WormStates/Jumping.h"
#include "WormStates/Land.h"
#include "WormStates/Sliding.h"
#include "WormStates/StartBackFlip.h"
#include "WormStates/StartJump.h"
#include "WormStates/Still.h"
#include "WormStates/Teleported.h"
#include "WormStates/Teleporting.h"
#include "WormStates/Walk.h"
#include "Weapons/WeaponNone.h"

#define CONFIG Game::Config::getInstance()

Worms::Player::Player(Physics &physics)
    : PhysicsEntity(Worms::EntityID::EtWorm), physics(physics), waterLevel(CONFI
G.getWaterLevel()) {
    /* creates 2 bodies so players cannot move each other */
    this->body = this->createBody(b2_dynamicBody);
    this->body_kinematic = this->createBody(b2_kinematicBody);

    /* creates the sensor as a circle */
    b2CircleShape sensorShape;
    sensorShape.m_radius = PLAYER_HEIGHT / 4;
    sensorShape.m_p.Set(0.0f, -PLAYER_HEIGHT / 4 - 0.2);

    /* allocated in heap because it's address shouldn't change */
    this->footSensor = new TouchSensor{*this->body, sensorShape};
    this->footSensor->ignore(*this);

    this->setState(Worm::StateID::Falling);
```



```
    this->weapon = std::shared_ptr<Worms::Weapon>(new ::Weapon::Bazooka(0.0f));
}

Worms::Player::~Player() {
    delete this->footSensor;
}

/**
 * @brief "Not equal" operator.
 *
 * @param other Other instance to compare.
 * @return true if not equal.
 */
bool Worms::Player::operator!=(const Player &other) {
    return !(*this == other);
}

/**
 * @brief Comparisson operator.
 *
 * @param other Other instance to compare.
 * @return true if equal.
 */
bool Worms::Player::operator==(const Player &other) {
    return (this->id == other.id) && (this->teamID == other.teamID);
}

/**
 * @brief Handles player-entity contact.
 *
 * @param other Other player that made contact.
 * @param contact box2D collision contact.
 */
void Worms::Player::contactWith(PhysicsEntity &entity, b2Contact &contact) {
    if (entity.getEntityId() == Worms::EntityID::EtGirder) {
        Worms::Girder &girder = dynamic_cast<Worms::Girder &>(entity);
        if (std::abs(girder.angle) > PI / 4.0f) {
            this->lastGroundNormal = contact.GetManifold()->localNormal;
        } else {
            this->lastGroundNormal = {0.0f, 0.1f};
        }
    }

    if (entity.getEntityId() != Worms::EntityID::EtWorm) {
        return;
    }

    /* checks if it's the player itself */
    if (&entity == this) {
        /* checks if it's the kinematic and dynamic bodies colliding */
        if (contact.GetFixtureA()->GetBody()->GetType() !=
            contact.GetFixtureB()->GetBody()->GetType()) {
            contact.SetEnabled(false);
        }
    }
}

void Worms::Player::update(float dt) {
    /* sets the kinematic body to the position of the dynamic body */
    this->body_kinematic->SetTransform(this->body->GetTransform().p, this->body-
>GetAngle());
```

```
    this->state->update(*this, dt, this->body);
    this->weapon->update(dt);

    if (this->getPosition().y <= this->waterLevel && this->getStateId() != Worm:
:StateID::Dead &&
        this->getStateId() != Worm::StateID::Drowning) {
        this->health = 0;
        if (this->getStateId() == Worm::StateID::Hit) {
            this->notify(*this, Event::EndHit);
        }
        this->setState(Worm::StateID::Drowning);
        this->notify(*this, Event::Drowning);
    } else if (this->isOnGround()) {
        /* checks if the ground slope is too tilted */
        try {
            b2Vec2 normal = this->getGroundNormal();
            float slope = std::abs(std::atan2(normal.y, normal.x));
            if ((slope < PI / 4.0f) || (slope > (PI * 3.0f) / 4.0f)) {
                if (this->getStateId() == Worm::StateID::Hit) {
                    this->notify(*this, Event::EndHit);
                }
                this->setState(Worm::StateID::Sliding);
                return;
            }
        } catch (const Exception &e) {
        }
    }
}

/**
 * @brief Whether the player is touching the ground or not.
 *
 * @return true is touching the ground.
 */
bool Worms::Player::isOnGround() const {
    return this->footSensor->isActive();
}

void Worms::Player::setPosition(const Math::Point<float> &new_pos) {
    this->body->SetTransform(b2Vec2(new_pos.x, new_pos.y), body->GetAngle());
}

/**
 * @brief Returns a unit vector with the direction normal to the floor where the
 player is standing.
 *
 * @return b2Vec2 Floor normal.
 */
b2Vec2 Worms::Player::getGroundNormal() const {
    for (auto &contact : *this->footSensor) {
        if (contact.first->getEntityId() == Worms::EntityID::EtGirder) {
            return this->lastGroundNormal;
        }
    }
    throw Exception{"No ground normal"};
}

void Worms::Player::startContact(Worms::PhysicsEntity *physicsEntity, b2Contact
&contact) {}

Math::Point<float> Worms::Player::getPosition() const {
    const b2Vec2 &pos = this->body->GetPosition();
```



```
    return Math::Point<float>{pos.x, pos.y};
}

Worm::StateID Worms::Player::getStateId() const {
    return this->state->getState();
}

void Worms::Player::handleState(IO::PlayerMsg pi) {
    switch (pi.input) {
        case IO::PlayerInput::moveLeft:
            this->state->moveLeft(*this);
            break;
        case IO::PlayerInput::moveRight:
            this->state->moveRight(*this);
            break;
        case IO::PlayerInput::startJump:
            this->state->jump(*this);
            break;
        case IO::PlayerInput::startBackFlip:
            this->state->backFlip(*this);
            break;
        case IO::PlayerInput::stopMove:
            this->state->stopMove(*this);
            break;
        case IO::PlayerInput::bazooka:
            this->state->bazooka(*this);
            break;
        case IO::PlayerInput::grenade:
            this->state->grenade(*this);
            break;
        case IO::PlayerInput::cluster:
            this->state->cluster(*this);
            break;
        case IO::PlayerInput::mortar:
            this->state->mortar(*this);
            break;
        case IO::PlayerInput::banana:
            this->state->banana(*this);
            break;
        case IO::PlayerInput::holy:
            this->state->holy(*this);
            break;
        case IO::PlayerInput::moveNone:
            break;
        case IO::PlayerInput::pointUp:
            this->state->pointUp(*this);
            break;
        case IO::PlayerInput::pointDown:
            this->state->pointDown(*this);
            break;
        case IO::PlayerInput::startShot:
            this->state->startShot(*this);
            break;
        case IO::PlayerInput::endShot:
            this->state->endShot(*this);
            break;
        case IO::PlayerInput::timeout1:
            this->state->setTimeout(*this, 1);
            break;
        case IO::PlayerInput::timeout2:
            this->state->setTimeout(*this, 2);
            break;
```

```
        case IO::PlayerInput::timeout3:
            this->state->setTimeout(*this, 3);
            break;
        case IO::PlayerInput::timeout4:
            this->state->setTimeout(*this, 4);
            break;
        case IO::PlayerInput::timeout5:
            this->state->setTimeout(*this, 5);
            break;
        case IO::PlayerInput::positionSelected:
            this->weapon->positionSelected(*this, pi.position);
            break;
        case IO::PlayerInput::aerialAttack:
            this->state->aerialAttack(*this);
            break;
        case IO::PlayerInput::dynamite:
            this->state->dynamite(*this);
            break;
        case IO::PlayerInput::baseballBat:
            this->state->baseballBat(*this);
            break;
        case IO::PlayerInput::teleport:
            this->state->teleport(*this);
            break;
        default:
            break;
    }
}

void Worms::Player::setState(Worm::StateID stateID) {
    if (this->state == nullptr || this->state->getState() != stateID) {
        /* creates the right state type */
        this->body->SetType(b2_dynamicBody);
        switch (stateID) {
            case Worm::StateID::Still:
                //                this->body->SetType(b2_staticBody);
                this->state = std::shared_ptr<State>(new Still());
                break;
            case Worm::StateID::Walk:
                this->state = std::shared_ptr<State>(new Walk());
                break;
            case Worm::StateID::StartJump:
                this->state = std::shared_ptr<State>(new StartJump());
                break;
            case Worm::StateID::Jumping:
                this->state = std::shared_ptr<State>(new Jumping(this->getPositi
on()));
                break;
            case Worm::StateID::EndJump:
                this->state = std::shared_ptr<State>(new EndJump());
                break;
            case Worm::StateID::StartBackFlip:
                this->state = std::shared_ptr<State>(new StartBackFlip());
                break;
            case Worm::StateID::BackFlipping:
                this->state = std::shared_ptr<State>(new BackFlipping(this->getP
osition()));
                break;
            case Worm::StateID::EndBackFlip:
                this->state = std::shared_ptr<State>(new EndBackFlip());
                break;
            case Worm::StateID::Falling:
```



```
                this->state = std::shared_ptr<State>(new Falling(this->getPositi
on()));
                break;
            case Worm::StateID::Land:
                this->state = std::shared_ptr<State>(new Land());
                break;
            case Worm::StateID::Batting:
                this->state = std::shared_ptr<State>(new Batting());
                break;
            case Worm::StateID::Teleporting:
                this->state = std::shared_ptr<State>(new Teleporting(this->telep
ortPosition));
                break;
            case Worm::StateID::Teleported:
                this->state = std::shared_ptr<State>(new Teleported());
                break;
            case Worm::StateID::Hit:
                this->state = std::shared_ptr<State>(new Hit());
                break;
            case Worm::StateID::Die:
                this->state = std::shared_ptr<State>(new Die());
                break;
            case Worm::StateID::Drowning:
                this->state = std::shared_ptr<State>(new Drowning());
                break;
            case Worm::StateID::Dead:
                this->state = std::shared_ptr<State>(new Dead());
                this->body->SetType(b2_staticBody);
                break;
            case Worm::StateID::Sliding:
                this->notify(*this, Event::WormFalling);
                this->state = std::shared_ptr<State>(new Sliding());
                break;
        }
    }
}

std::list<Worms::Bullet> Worms::Player::getBullets() {
    return std::move(this->bullets);
}

void Worms::Player::acknowledgeDamage(Config::Bullet::DamageInfo damageInfo,
                                      Math::Point<float> epicenter) {
    if (this->getStateId() != Worm::StateID::Dead) {
        double distanceToEpicenter = this->getPosition().distance(epicenter);
        if (distanceToEpicenter <= damageInfo.radius) {
            this->body->SetType(b2_dynamicBody);
            double inflictedDamage =
                (1.0f - (distanceToEpicenter / (damageInfo.radius * 1.01f))) * d
amageInfo.damage;
            this->health -= inflictedDamage;

            Math::Point<float> positionToEpicenter = this->getPosition() - epice
nter;
            float xImpactDirection = (positionToEpicenter.x > 0) - (positionToEp
icenter.x < 0);
            float yImpactDirection = (positionToEpicenter.y > 0) - (positionToEp
icenter.y < 0);
            float32 mass = this->body->GetMass();
            b2Vec2 impulses = {
                mass * float32(inflictedDamage) * xImpactDirection * damageInfo.
impulseDampingRatio,
```

```
                mass * float32(inflictedDamage) * yImpactDirection *
                    damageInfo.impulseDampingRatio};
            b2Vec2 position = this->body->GetWorldCenter();
            this->body->ApplyLinearImpulse(impulses, position, true);
            this->notify(*this, Event::Hit);
            this->setState(Worm::StateID::Hit);
            this->health = (this->health < 0) ? 0 : this->health;
        }
    }
}

void Worms::Player::acknowledgeDamage(const Config::P2PWeapon &info,
                                      Math::Point<float> shooterPosition,
                                      Worm::Direction shooterDirection) {
    if (this->getStateId() != Worm::StateID::Dead) {
        if ((shooterDirection == Worm::Direction::right &&
             this->getPosition().x - shooterPosition.x > 0) ||
            (shooterDirection == Worm::Direction::left &&
             this->getPosition().x - shooterPosition.x < 0)) {
            double distanceToTheWeapon = this->getPosition().distance(info.posit
ion);
            if (distanceToTheWeapon <= info.dmgInfo.radius && distanceToTheWeapo
n > 0) {
                this->body->SetType(b2_dynamicBody);
                this->health -= info.dmgInfo.damage;
                this->health = (this->health < 0) ? 0 : this->health;

                float32 mass = this->body->GetMass();
                Math::Point<float> direction{0, 0};
                direction.x = info.dmgInfo.radius * cos(info.angle * PI / 180.0f
);
                direction.y = info.dmgInfo.radius * sin(info.angle * PI / 180.0f
);
                Math::Point<float> positionToShooter = this->getPosition() - sho
oterPosition;
                float xImpactDirection = (positionToShooter.x > 0) - (positionTo
Shooter.x < 0);
                float yImpactDirection = (direction.y > 0) - (direction.y < 0);
                b2Vec2 impulses = {mass * float32(info.dmgInfo.damage) * directi
on.x *
                                       xImpactDirection * info.dmgInfo.impulseDa
mpingRatio,
                                   mass * float32(info.dmgInfo.damage) * directi
on.y *
                                       yImpactDirection * info.dmgInfo.impulseDa
mpingRatio};

                this->body->ApplyLinearImpulse(impulses, this->body->GetWorldCen
ter(), true);
                this->notify(*this, Event::Hit);
                this->setState(Worm::StateID::Hit);
            }
        }
    }
}

float Worms::Player::getWeaponAngle() const {
    return this->weapon->getAngle();
}

const Worm::WeaponID &Worms::Player::getWeaponID() const {
    return this->weapon->getWeaponID();
```



```
}

void Worms::Player::setWeapon(const Worm::WeaponID &id) {
    // keep the last angle
    float lastAngle = this->weapon->getAngle();
    this->weapon = this->team->getWeapon(id);
    this->weapon->setAngle(lastAngle);
    this->isP2PWeapon = this->weapon->isP2PWeapon();
}

void Worms::Player::increaseWeaponAngle() {
    this->weapon->increaseAngle();
}

void Worms::Player::decreaseWeaponAngle() {
    this->weapon->decreaseAngle();
}

void Worms::Player::startShot() {
    this->weapon->startShot(this);
}

void Worms::Player::endShot() {
    if (this->weapon->getWeaponID() != Worm::WeaponID::WTeleport &&
        this->weapon->getWeaponID() != Worm::WeaponID::WAerial &&
            this->weapon->getWeaponID() != Worm::WeaponID::WNone) {
        if (!this->isP2PWeapon) {
            Math::Point<float> position = this->getPosition();
            float safeNonContactDistance = sqrt((PLAYER_WIDTH / 2) * (PLAYER_WID
TH / 2) +
                                                (PLAYER_HEIGHT / 2) * (PLAYER_HE
IGHT / 2)) + 0.1;
            BulletInfo info = this->weapon->getBulletInfo();
            info.point = position;
            info.safeNonContactDistance = safeNonContactDistance;
            if (this->direction == Worm::Direction::right) {
                if (info.angle < 0.0f) {
                    info.angle += 360.0f;
                }
            } else {
                info.angle = 180.0f - info.angle;
            }
            this->bullets.emplace_back(info, this->physics, this->weapon->getWea
ponID());
            this->weapon->endShot();
            this->notify(*this, Event::Shot);
        } else {
            this->setState(Worm::StateID::Batting);
            this->notify(*this, Event::P2PWeaponUsed);
        }
        this->team->weaponUsed(this->getWeaponID());
    }
}

void Worms::Player::endShot(std::list<Worms::Bullet> &bullets) {
    this->bullets = std::move(bullets);
    this->notify(*this, Event::Shot);
    this->team->weaponUsed(this->getWeaponID());
}

void Worms::Player::setTeamID(uint8_t team) {
    this->teamID = team;
```

```
}

void Worms::Player::increaseHealth(float extraPoints) {
//    this->health += (percentage / 100.0f) * this->health; //    25% more
    this->health += extraPoints; //    25 points more
}

uint8_t Worms::Player::getTeam() const {
    return this->teamID;
}

void Worms::Player::setId(uint8_t id) {
    this->id = id;
}

uint8_t Worms::Player::getId() const {
    return this->id;
}

void Worms::Player::setWeaponTimeout(uint8_t time) {
    this->weapon->setTimeout(time);
}

void Worms::Player::landDamage(float yDistance) {
    if (yDistance > CONFIG.getSafeFallDistance()) {
        this->health -=
            (yDistance > CONFIG.getMaxFallDamage()) ? CONFIG.getMaxFallDamage()
: yDistance;
        this->health = (this->health < 0) ? 0 : this->health;
        if (this->health > 0) {
            this->notify(*this, Event::DamageOnLanding);
        }
    }
}

/**
 * @brief Creates a player's body with the given type.
 *
 * @param type Body type.
 * @return Created body.
 */
b2Body *Worms::Player::createBody(b2BodyType type) {
    /* the players consists of a rectangle as the upper part of the body and a c
icle for the
     * bottom */
    b2BodyDef bodyDef;
    bodyDef.type = type;
    bodyDef.position.Set(0.0f, 0.0f);
    bodyDef.fixedRotation = true;
    b2Body *new_body = this->physics.createBody(bodyDef);

    b2PolygonShape shape;
    shape.SetAsBox(PLAYER_WIDTH / 2, PLAYER_HEIGHT / 4, b2Vec2{0.0f, PLAYER_HEIG
HT / 4}, 0.0f);

    /* creates the upper square */
    b2FixtureDef fixture;
    fixture.shape = &shape;
    fixture.density = 1.0f;
    fixture.restitution = 0.1f;
    fixture.friction = 1.0f;
    new_body->CreateFixture(&fixture);
```



```
    /* creates the bottom circle */
    b2CircleShape bottom;
    bottom.m_radius = PLAYER_HEIGHT / 4;
    bottom.m_p.Set(0.0f, -PLAYER_HEIGHT / 4);
    fixture.shape = &bottom;
    new_body->CreateFixture(&fixture);

    new_body->SetUserData(this);
    return new_body;
}

std::list<Worms::Bullet> Worms::Player::onExplode(const Bullet &b, Physics &phys
ics) {
    return std::move(this->weapon->onExplode(b, physics));
}

void Worms::Player::reset() {
    this->weapon->endShot();
    /*
     * If the weapon has no more ammunition, returns weaponNone
     */
    this->setWeapon(this->getWeaponID());
    this->bullets.erase(this->bullets.begin(), this->bullets.end());
}

Worms::Physics &Worms::Player::getPhysics() {
    return this->physics;
}

const std::shared_ptr<Worms::Weapon> Worms::Player::getWeapon() const {
    return this->weapon;
}

void Worms::Player::setTeam(Worms::Team *team) {
    this->team = team;
}

Worms::Player::Player(Worms::Player &&player) noexcept: PhysicsEntity(std::move(
player)), physics(player.physics), waterLevel(player.waterLevel) {

    this->body = player.body;
    this->body_kinematic = player.body_kinematic;
    this->footSensor = player.footSensor;

    this->state = player.state;
    this->weapon = player.weapon;
    this->team = player.team;
    this->id = player.id;
    this->bullets = std::move(player.bullets);

    player.body = nullptr;
    player.body_kinematic = nullptr;
    player.footSensor = nullptr;
    player.state = nullptr;
    player.weapon = nullptr;
    player.team = 0;
    player.id = 0;
}

void Worms::Player::die() {
    this->setState(Worm::StateID::Die);
```

```
    this->health = 0;
    this->dyingDisconnected = true;
    this->notify(*this, Event::DyingDueToDisconnection);
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 18/05/18
 */

#ifndef __PLAYER_H__
#define __PLAYER_H__

#define PLAYER_WIDTH 0.8f

#define PLAYER_HEIGHT 2.0f

#include <list>

#include "Config/Config.h"
#include "Config/P2PWeapon.h"
#include "Direction.h"
#include "GameStateMsg.h"
#include "Physics.h"
#include "Point.h"
#include "Stream.h"
#include "Team.h"
#include "TouchSensor.h"
#include "Weapons/Bullet.h"
#include "Weapons/Weapon.h"
#include "WormStates/PlayerState.h"

enum class PlayerState { movingRight, movingLeft, still };

namespace Worms {

class Player : public PhysicsEntity {
   public:
    Worm::Direction direction{Worm::Direction::left};
    Worm::Direction lastWalkDirection;
    std::uint16_t health{0};
    Math::Point<float> teleportPosition{0.0f, 0.0f};
    bool dyingDisconnected{false};

    explicit Player(Physics &physics);
    Player(Player &&player) noexcept;
    Player(Player &copy) = delete;

    ~Player();

    /* contact handlers */
    virtual void contactWith(PhysicsEntity &other, b2Contact &contact);

    bool isOnGround() const;

    /**
     * Updates its state, its weapon
     * @param dt
     */
    void update(float dt);
    void serialize(IO::Stream<IO::GameStateMsg> &s) const {}
    /**
     * @brief moves the player to newPos position
     * @param newPos
     */
    void setPosition(const Math::Point<float> &newPos);
    b2Vec2 getGroundNormal() const;
```



```
    void startContact(Worms::PhysicsEntity *physicsEntity, b2Contact &contact);

    /**
     * @brief asks box2D from current position.
     * @return
     */
    Math::Point<float> getPosition() const;
    /**
     * @brief given playerInput, changes its state (or its weapon) accordingly
     * @param pi
     */
    void handleState(IO::PlayerMsg pi);
    const std::shared_ptr<Worms::Weapon> getWeapon() const;
    Worm::StateID getStateId() const;
    void setState(Worm::StateID stateID);
    float getWeaponAngle() const;
    const Worm::WeaponID &getWeaponID() const;
    void setWeapon(const Worm::WeaponID &id);
    /**
     * @brief delegates on its weapon the action of increase the angle, if
     * the weapon handles it.
     */
    void increaseWeaponAngle();
    /**
     * @brief delegates on its weapon the action of decrease the angle, if
     * the weapon handles it.
     */
    void decreaseWeaponAngle();
    /**
     * @brief delegates on its weapon the action of starting a shot, increasing
     * its powerShot if it handles it
     */
    void startShot();
    /**
     * @brief creates a bullet that needs to be moved using getBullet()
     */
    void endShot();
    void acknowledgeDamage(Config::Bullet::DamageInfo damageInfo, Math::Point<fl
oat> epicenter);
    void acknowledgeDamage(const Config::P2PWeapon &info, Math::Point<float> sho
oterPosition,
                           Worm::Direction shooterDirection);
    void landDamage(float yDistance);
    void setTeamID(uint8_t team);
    void setTeam(Worms::Team *team);
    void increaseHealth(float extraPoints);
    uint8_t getTeam() const;
    void setId(uint8_t id);
    uint8_t getId() const;
    Physics &getPhysics();
    void setWeaponTimeout(uint8_t time);
    /**
     * Moves the bullets to the caller (the Game)
     * @return bullets
     */
    std::list<Bullet> getBullets();
    /**
     * Resets the weapon's powershot and erase every possible bullet
     * inside his container.
     */
    void reset();
    /**
```

```
     * calls weapon's onExplode and get new bullets if it is necesary.
     */
    std::list<Bullet> onExplode(const Bullet &bullet, Physics &physics);

    bool operator!=(const Player &other);
    bool operator==(const Player &other);

    void endShot(std::list<Worms::Bullet> &bullets);
    void die();

   private:
    b2Body *createBody(b2BodyType type);

    b2Body *body{nullptr};
    b2Body *body_kinematic{nullptr};
    TouchSensor *footSensor;

    std::shared_ptr<Worms::State> state{nullptr};
    std::shared_ptr<Worms::Weapon> weapon{nullptr};
    Physics &physics;
    const int waterLevel;
    uint8_t teamID;
    uint8_t id;
    std::list<Bullet> bullets;
    bool isP2PWeapon{false};
    b2Vec2 lastGroundNormal{0.0f, 0.0f};
    Team *team{nullptr};
};
}  // namespace Worms

#endif  //__PLAYER_H__
```

```
//
// Created by rodrigo on 10/06/18.
//

#include "PlayerShot.h"

void Worms::PlayerShot::endTurn(GameTurn &gt) {}

void Worms::PlayerShot::wormHit(GameTurn &gt, uint8_t wormId) {}

void Worms::PlayerShot::wormEndHit(Worms::GameTurn &gt, uint8_t wormId) {}

void Worms::PlayerShot::wormDrowning(Worms::GameTurn &gt, uint8_t wormId) {}

void Worms::PlayerShot::wormDrowned(Worms::GameTurn &gt, uint8_t wormId) {}

Worms::PlayerShot::PlayerShot() {}

void Worms::PlayerShot::explosion() {}

void Worms::PlayerShot::update(float dt) {}

void Worms::PlayerShot::wormDisconnectedDying(uint8_t wormId) {}

void Worms::PlayerShot::wormDisconnectedDead(uint8_t wormId) {}
```

```
//
// Created by rodrigo on 10/06/18.
//

#ifndef INC_4_WORMS_PLAYERSHOT_H
#define INC_4_WORMS_PLAYERSHOT_H

#include "../../../libs/Observer.h"
#include "GameTurnState.h"

namespace Worms {
class PlayerShot : public GameTurnState {
   public:
    PlayerShot();
    ~PlayerShot() = default;

    void endTurn(GameTurn &gt) override;
    void update(float dt) override;
    void wormHit(GameTurn &gt, uint8_t wormId) override;
    void wormEndHit(GameTurn &gt, uint8_t wormId) override;
    void wormDrowning(GameTurn &gt, uint8_t wormId) override;
    void wormDrowned(GameTurn &gt, uint8_t wormId) override;
    void explosion() override;
    void wormDisconnectedDying(uint8_t wormId) override;
    void wormDisconnectedDead(uint8_t wormId) override;
};
}

#endif  // INC_4_WORMS_PLAYERSHOT_H
```

```cpp
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 20/05/18
 */

#include "PlayerState.h"
#include "GameStateMsg.h"

Worms::State::State(Worm::StateID id) : stateID(id) {}

Worm::StateID Worms::State::getState() const {
    return this->stateID;
}
```

```
#ifndef _PLAYERSTATE_H
#define _PLAYERSTATE_H

#include <Box2D/Common/b2Math.h>
#include <Box2D/Dynamics/b2Body.h>
#include <vector>

#include "GameStateMsg.h"

namespace Worms {
class Player;
class State {
   public:
    explicit State(Worm::StateID id);
    virtual ~State() = default;
    virtual void update(Player &p, float dt, b2Body *body) = 0;
    virtual void moveRight(Player &p) = 0;
    virtual void moveLeft(Player &p) = 0;
    virtual void jump(Player &p) = 0;
    virtual void setTimeout(Player &p, uint8_t time) = 0;

    virtual void bazooka(Player &p) = 0;
    virtual void grenade(Player &p) = 0;
    virtual void cluster(Player &p) = 0;
    virtual void mortar(Player &p) = 0;
    virtual void banana(Player &p) = 0;
    virtual void holy(Player &p) = 0;
    virtual void aerialAttack(Player &p) = 0;
    virtual void dynamite(Player &p) = 0;
    virtual void baseballBat(Player &p) = 0;
    virtual void teleport(Player &p) = 0;

    virtual void startShot(Player &p) = 0;
    virtual void endShot(Player &p) = 0;
    virtual void backFlip(Player &p) = 0;
    virtual void stopMove(Player &p) = 0;
    virtual void pointUp(Player &p) = 0;
    virtual void pointDown(Player &p) = 0;
    virtual Worm::StateID getState() const;

   protected:
    Worm::StateID stateID;
    std::vector<float> impulses{0.0f, 0.0f};
};
}

#endif  //_PLAYERSTATE_H
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#include <netdb.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <cstring>

#include "ErrorMessages.h"
#include "Exception.h"
#include "ServerSocket.h"

ServerSocket::ServerSocket(const char *port) {
    this->bindAndListen(port);
}

void ServerSocket::bindAndListen(const char *port) {
    int status = 0;
    int option_value = 1;
    bool is_bound = false;
    /*
    *   inicializo el bloque de memoria de addrinfo,
    *   lo configuro para que result sea una lista de
    *   address pertenecientes a IPv4, y que sean TCP.
    */
    struct addrinfo hints = {AI_PASSIVE, AF_INET, SOCK_STREAM, 0, 0, nullptr, nu
llptr, nullptr};
    struct addrinfo *result, *ptr;

    status = getaddrinfo(nullptr, port, &hints, &result);
    if (status != 0) {
        throw Exception(ERR_MSG_SOCKET_INVALID_PORT, port, gai_strerror(status))
;
    }
    /*
    *   Recorro los resultados posibles, hasta poder bindear
    */
    for (ptr = result; ptr != nullptr && !is_bound; ptr = ptr->ai_next) {
        this->fd = ::socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
        /*
        *       si la creaciÃ³n del socket falla, no debo hacer nada mas
        *       en el ciclo (ya que no se abrio ningun fd)
        */
        if (this->fd == -1) {
            continue;
        }
        /*
        *       Del ejemplo del echoserver, se obtuvo la forma de
        *       configurar la reutilizaciÃ³n de la direcciÃ³n
        *       que no se encuentre disponible por un TIME_WAIT.
        *       si la configuraciÃ³n falla, debo liberar
        *       el socket (segun la documentaciÃ³n y el ejemplo
        *       que se encuentra en el manual de getaddrinfo)
        */
        status =
            setsockopt(this->fd, SOL_SOCKET, SO_REUSEADDR, &option_value, sizeof
(option_value));
        if (status == -1) {
```



```
            this->close();
            continue;
        }
        /*
        *       Si logro bindear, salgo del ciclo, sino, cierro el socket
        *       y pruebo en el siguiente resultado.
        */
        status = bind(this->fd, result->ai_addr, result->ai_addrlen);
        if (status == -1) {
            this->close();
        } else {
            is_bound = true;
        }
    }
    freeaddrinfo(result);

    if (!is_bound) {
        throw Exception(ERR_MSG_SOCKET_BINDING, port);
    }

    status = listen(this->fd, 20);
    if (status == -1) {
        throw Exception(ERR_MSG_SOCKET_LISTEN, strerror(errno));
    }
}

CommunicationSocket ServerSocket::accept() {
    int fd = ::accept(this->fd, nullptr, nullptr);
    if (fd == -1) {
        throw Exception(ERR_MSG_SOCKET_ACCEPT, strerror(errno));
    }
    return std::move(CommunicationSocket(fd));
}
```

```
/*
 * Created by Federico Manuel Gomez Peter
 * Date: 02/05/2018.
 */

#ifndef __SERVERSOCKET_H__
#define __SERVERSOCKET_H__

#include <string>

#include "CommunicationSocket.h"
#include "Socket.h"

class ServerSocket : public Socket {
   public:
    explicit ServerSocket(const char *port);
    /**
     * Acepta una conexiÃ³n y devuelve un CommunicationSocket por movimiento.
     * @return Socket para comunicacion
     */
    CommunicationSocket accept();
    void bindAndListen(const char *port);
};

#endif  //__SERVERSOCKET_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 20/05/18
 */

#include <iostream>
#include <vector>

#include "../Player.h"
#include "Sliding.h"

Worms::Sliding::Sliding() : State(Worm::StateID::Sliding) {}

void Worms::Sliding::update(Worms::Player &p, float dt, b2Body *body) {
    if (!p.isOnGround()) {
        p.setState(Worm::StateID::Falling);
        return;
    }

    float final_vel{0.0f};

    try {
        b2Vec2 normal = p.getGroundNormal();
        float slope = std::abs(std::atan2(normal.y, normal.x));
        if ((slope < PI / 4.0f) || (slope > (PI * 3.0f) / 4.0f)) {
            final_vel = 3.0f * normal.x;
            float impulse = body->GetMass() * (final_vel - body->GetLinearVeloci
ty().x);

            body->ApplyLinearImpulse(b2Vec2(impulse, 0.0f), body->GetWorldCenter
(), true);
        } else {
            p.setState(Worm::StateID::Land);
        }
    } catch (const Exception &e) {
    }
}

void Worms::Sliding::moveRight(Worms::Player &p) {}

void Worms::Sliding::moveLeft(Worms::Player &p) {}

void Worms::Sliding::jump(Worms::Player &p) {}

void Worms::Sliding::stopMove(Worms::Player &p) {}

void Worms::Sliding::backFlip(Worms::Player &p) {}

void Worms::Sliding::bazooka(Worms::Player &p) {}

void Worms::Sliding::pointUp(Worms::Player &p) {}

void Worms::Sliding::pointDown(Worms::Player &p) {}

void Worms::Sliding::startShot(Worms::Player &p) {}

void Worms::Sliding::endShot(Worms::Player &p) {}

void Worms::Sliding::grenade(Worms::Player &p) {}

void Worms::Sliding::cluster(Worms::Player &p) {}
```



```
void Worms::Sliding::mortar(Worms::Player &p) {}

void Worms::Sliding::banana(Worms::Player &p) {}

void Worms::Sliding::holy(Worms::Player &p) {}

void Worms::Sliding::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Sliding::aerialAttack(Worms::Player &p) {}

void Worms::Sliding::dynamite(Worms::Player &p) {}

void Worms::Sliding::teleport(Worms::Player &p) {}

void Worms::Sliding::baseballBat(Worms::Player &p) {}
```

```
#ifndef _PLAYER_SLIDING_H
#define _PLAYER_SLIDING_H

#include "../Config/Config.h"
#include "PlayerState.h"

namespace Worms {
class Sliding : public State {
   public:
    Sliding();
    ~Sliding() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    virtual void pointUp(Player &p) override;
    virtual void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
};
}  // namespace Worms

#endif  //_PLAYER_SLIDING_H
```

```
/*
 *  Created by Rodrigo.
 *  date: 20/05/18
 */

#include "StartBackFlip.h"
#include "Direction.h"

Worms::StartBackFlip::StartBackFlip()
    : State(Worm::StateID::StartBackFlip),
      backflipVelocity(Game::Config::getInstance().getBackflipVelocity()),
      startJumpTime(Game::Config::getInstance().getStartJumpTime()) {}

void Worms::StartBackFlip::update(Worms::Player &p, float dt, b2Body *body) {
    this->timeElapsed += dt;
    if (this->timeElapsed >= this->startJumpTime) {
        if (!this->impulseApplied) {
            float32 mass = body->GetMass();
            b2Vec2 impulses = {mass * this->backflipVelocity.x, mass * this->bac
kflipVelocity.y};
            if (p.direction == Worm::Direction::left) {
                impulses.x *= -1;
            }
            /* When the worm jumps, it needs an initial impulse in the y axis
             * that will never will be applied again. In the x axis, the worms
             * moves in RUM, so it needs an initial impulse (because his frictio
n
             * coeficient is 0) and then needs an end impulse, of equal absolute
             * value and different sign.
             */
            body->ApplyLinearImpulse(impulses, body->GetWorldCenter(), true);
            this->impulseApplied = true;
        } else if (!p.isOnGround()) {
            p.setState(Worm::StateID::BackFlipping);
        } else if (this->timeElapsed > 0.9f) {
            p.setState(Worm::StateID::Still);
        }
    }
}

void Worms::StartBackFlip::moveRight(Worms::Player &p) {}

void Worms::StartBackFlip::moveLeft(Worms::Player &p) {}

void Worms::StartBackFlip::jump(Worms::Player &p) {}

void Worms::StartBackFlip::backFlip(Worms::Player &p) {}

void Worms::StartBackFlip::stopMove(Worms::Player &p) {}

void Worms::StartBackFlip::bazooka(Worms::Player &p) {}

void Worms::StartBackFlip::pointUp(Worms::Player &p) {}

void Worms::StartBackFlip::pointDown(Worms::Player &p) {}

void Worms::StartBackFlip::startShot(Worms::Player &p) {}

void Worms::StartBackFlip::endShot(Worms::Player &p) {}

void Worms::StartBackFlip::grenade(Worms::Player &p) {}
```



```
void Worms::StartBackFlip::cluster(Worms::Player &p) {}

void Worms::StartBackFlip::mortar(Worms::Player &p) {}

void Worms::StartBackFlip::banana(Worms::Player &p) {}

void Worms::StartBackFlip::holy(Worms::Player &p) {}

void Worms::StartBackFlip::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::StartBackFlip::aerialAttack(Worms::Player &p) {}

void Worms::StartBackFlip::dynamite(Worms::Player &p) {}

void Worms::StartBackFlip::teleport(Worms::Player &p) {}

void Worms::StartBackFlip::baseballBat(Worms::Player &p) {}
```

```
/*
 *  Created by Rodrigo.
 *  date: 20/05/18
 */

#ifndef __PLAYER_START_BACK_FLIP_H__
#define __PLAYER_START_BACK_FLIP_H__

#include <stdint-gcc.h>
#include <cstdint>
#include "../Config/Config.h"
#include "../Player.h"

namespace Worms {
class StartBackFlip : public State {
   public:
    StartBackFlip();
    ~StartBackFlip() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    bool impulseApplied{false};
    const Math::Vector backflipVelocity;
    const float startJumpTime;
};
}

#endif  //__PLAYER_START_BACK_FLIP_H__
```

```
//
// Created by Gorco on 19/05/18.
//

#include <iostream>

#include "../Config/Config.h"
#include "Direction.h"
#include "StartJump.h"

Worms::StartJump::StartJump()
    : State(Worm::StateID::StartJump),
      jumpTime(Game::Config::getInstance().getStartJumpTime()),
      jumpVelocity(Game::Config::getInstance().getJumpVelocity()) {}

void Worms::StartJump::update(Player &p, float dt, b2Body *body) {
    this->timeElapsed += dt;
    if (this->timeElapsed >= this->jumpTime) {
        if (!this->impulseApplied) {
            float32 mass = body->GetMass();
            b2Vec2 impulses = {mass * this->jumpVelocity.x, mass * this->jumpVel
ocity.y};
            if (p.direction == Worm::Direction::left) {
                impulses.x *= -1;
            }
            /* When the worm jumps, it needs an initial impulse in the y axis
             * that will never will be applied again. In the x axis, the worms
             * moves in RUM, so it needs an initial impulse (because his frictio
n
             * coeficient is 0) and then needs an end impulse, of equal absolute
             * value and different sign.
             */
            body->ApplyLinearImpulse(impulses, body->GetWorldCenter(), true);
            this->impulseApplied = true;
        } else if (!p.isOnGround()) {
            p.setState(Worm::StateID::Jumping);
        } else if (this->timeElapsed > 0.9f) {
            p.setState(Worm::StateID::Still);
        }
    }
}

void Worms::StartJump::moveRight(Worms::Player &p) {}

void Worms::StartJump::moveLeft(Worms::Player &p) {}

void Worms::StartJump::jump(Worms::Player &p) {}

void Worms::StartJump::stopMove(Worms::Player &p) {}

void Worms::StartJump::backFlip(Worms::Player &p) {}

void Worms::StartJump::bazooka(Worms::Player &p) {}

void Worms::StartJump::pointUp(Worms::Player &p) {}

void Worms::StartJump::pointDown(Worms::Player &p) {}

void Worms::StartJump::startShot(Worms::Player &p) {}

void Worms::StartJump::endShot(Worms::Player &p) {}
```



```
void Worms::StartJump::grenade(Worms::Player &p) {}

void Worms::StartJump::cluster(Worms::Player &p) {}

void Worms::StartJump::mortar(Worms::Player &p) {}

void Worms::StartJump::banana(Worms::Player &p) {}

void Worms::StartJump::holy(Worms::Player &p) {}

void Worms::StartJump::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::StartJump::aerialAttack(Worms::Player &p) {}

void Worms::StartJump::dynamite(Worms::Player &p) {}

void Worms::StartJump::teleport(Worms::Player &p) {}

void Worms::StartJump::baseballBat(Worms::Player &p) {}
```

```
//
// Created by Gorco on 19/05/18.
//

#ifndef __WORMS_PLAYER_JUMP_RIGHT_H__
#define __WORMS_PLAYER_JUMP_RIGHT_H__

#include <stdint-gcc.h>
#include <cstdint>
#include "../Config/Config.h"
#include "../Player.h"

namespace Worms {
class StartJump : public State {
   public:
    StartJump();
    ~StartJump() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    bool impulseApplied{false};
    const float jumpTime;
    const Math::Vector jumpVelocity;
};
}  // namespace Worms

#endif  // __WORMS_PLAYER_JUMP_RIGHT_H__
```

```
//
// Created by rodrigo on 10/06/18.
//

#include <algorithm>

#include "StartTurn.h"

void Worms::StartTurn::endTurn(GameTurn &gt) {
    if (this->wormsFalling.size() == 0 && this->wormsDrowning.size() == 0 && !th
is->wormsDying && this->wormsDisconnectedDying.size() == 0) {
        this->notify(*this, Event::TurnEnded);
    }
}

void Worms::StartTurn::wormHit(GameTurn &gt, uint8_t wormId) {}

void Worms::StartTurn::wormEndHit(Worms::GameTurn &gt, uint8_t wormId) {}

void Worms::StartTurn::wormDrowning(Worms::GameTurn &gt, uint8_t wormId) {
    this->wormsDrowning.emplace_back(wormId);
    this->wormLanded(wormId);
}

void Worms::StartTurn::wormDrowned(Worms::GameTurn &gt, uint8_t wormId) {
    this->wormsDrowning.erase(
        std::remove(this->wormsDrowning.begin(), this->wormsDrowning.end(), worm
Id),
        this->wormsDrowning.end());
}

Worms::StartTurn::StartTurn() {}

void Worms::StartTurn::explosion() {}

void Worms::StartTurn::update(float dt) {}

void Worms::StartTurn::wormDisconnectedDying(uint8_t wormId) {
    this->wormsDisconnectedDying.emplace_back(wormId);
    if (this->wormToFollow != this->wormsDisconnectedDying[0] && this->wormsFall
ing.size() == 0 && this->wormsDrowning.size() == 0) {
        this->wormToFollow = this->wormsDisconnectedDying[0];
        this->notify(*this, Event::NewWormToFollow);
    }
}

void Worms::StartTurn::wormDisconnectedDead(uint8_t wormId) {
    this->wormsDisconnectedDying.erase(
            std::remove(this->wormsDisconnectedDying.begin(), this->wormsDisconn
ectedDying.end(), wormId),
            this->wormsDisconnectedDying.end());
    if (this->wormToFollow == wormId) {
        this->wormToFollow = this->wormsDisconnectedDying[0];
        this->notify(*this, Event::NewWormToFollow);
    }
}
```

```
//
// Created by rodrigo on 10/06/18.
//

#ifndef INC_4_WORMS_STARTTURN_H
#define INC_4_WORMS_STARTTURN_H

#include "../../../libs/Observer.h"
#include "GameTurnState.h"

namespace Worms {
class StartTurn : public GameTurnState {
   public:
    StartTurn();
    ~StartTurn() = default;

    void endTurn(GameTurn &gt) override;
    void update(float dt) override;
    void wormHit(GameTurn &gt, uint8_t wormId) override;
    void wormEndHit(GameTurn &gt, uint8_t wormId) override;
    void wormDrowning(GameTurn &gt, uint8_t wormId) override;
    void wormDrowned(GameTurn &gt, uint8_t wormId) override;
    void explosion() override;
    void wormDisconnectedDying(uint8_t wormId) override;
    void wormDisconnectedDead(uint8_t wormId) override;
};
}

#endif  // INC_4_WORMS_STARTTURN_H
```

```
//
// Created by Gorco on 19/05/18.
//

#include <cstdint>
#include <iostream>
#include <memory>

#include "../Player.h"
#include "Still.h"
#include "Walk.h"

Worms::Still::Still() : State(Worm::StateID::Still) {}

void Worms::Still::update(Player &p, float dt, b2Body *body) {
    float32 mass = body->GetMass();
    b2Vec2 vel = body->GetLinearVelocity();

    this->impulses[0] = -vel.x * mass;
    body->ApplyLinearImpulse(b2Vec2(impulses[0], impulses[1]), body->GetWorldCen
ter(), true);
}

void Worms::Still::moveRight(Worms::Player &p) {
    p.direction = Worm::Direction::right;
    p.setState(Worm::StateID::Walk);
}

void Worms::Still::moveLeft(Worms::Player &p) {
    p.direction = Worm::Direction::left;
    p.setState(Worm::StateID::Walk);
}

void Worms::Still::stopMove(Worms::Player &p) {}

void Worms::Still::jump(Worms::Player &p) {
    p.notify(p, Event::WormFalling);
    p.setState(Worm::StateID::StartJump);
}

void Worms::Still::backFlip(Worms::Player &p) {
    p.notify(p, Event::WormFalling);
    p.setState(Worm::StateID::StartBackFlip);
}

void Worms::Still::bazooka(Worms::Player &p) {
    p.setWeapon(Worm::WeaponID::WBazooka);
}

void Worms::Still::pointUp(Worms::Player &p) {
    p.increaseWeaponAngle();
}

void Worms::Still::pointDown(Worms::Player &p) {
    p.decreaseWeaponAngle();
}

void Worms::Still::startShot(Worms::Player &p) {
    p.startShot();
}

void Worms::Still::endShot(Worms::Player &p) {
```



```
    p.endShot();
}

void Worms::Still::grenade(Worms::Player &p) {
    p.setWeapon(Worm::WeaponID::WGrenade);
}

void Worms::Still::cluster(Worms::Player &p) {
    p.setWeapon(Worm::WeaponID::WCluster);
}

void Worms::Still::mortar(Worms::Player &p) {
    p.setWeapon(Worm::WeaponID::WMortar);
}

void Worms::Still::banana(Worms::Player &p) {
    p.setWeapon(Worm::WeaponID::WBanana);
}

void Worms::Still::holy(Worms::Player &p) {
    p.setWeapon(Worm::WeaponID::WHoly);
}

void Worms::Still::setTimeout(Worms::Player &p, uint8_t time) {
    p.setWeaponTimeout(time);
}

void Worms::Still::aerialAttack(Worms::Player &p) {
    p.setWeapon(Worm::WeaponID::WAerial);
}

void Worms::Still::dynamite(Worms::Player &p) {
    p.setWeapon(Worm::WeaponID::WDynamite);
}

void Worms::Still::teleport(Worms::Player &p) {
    p.setWeapon(Worm::WeaponID::WTeleport);
}

void Worms::Still::baseballBat(Worms::Player &p) {
    p.setWeapon(Worm::WeaponID::WBaseballBat);
}
```

```
//
// Created by Gorco on 19/05/18.
//

#ifndef INC_4_WORMS_STOPMOVE_H
#define INC_4_WORMS_STOPMOVE_H

#include <Box2D/Common/b2Math.h>
#include <vector>

#include "PlayerState.h"

namespace Worms {
class Still : public State {
   public:
    Still();
    ~Still() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;
};
}

#endif  // INC_4_WORMS_STOPMOVE_H
```

```
//
// Created by rodrigo on 3/06/18.
//

#include "Team.h"
#include "Weapons/AerialAttack.h"
#include "Weapons/Banana.h"
#include "Weapons/BaseballBat.h"
#include "Weapons/Bazooka.h"
#include "Weapons/Cluster.h"
#include "Weapons/Dynamite.h"
#include "Weapons/Grenade.h"
#include "Weapons/Holy.h"
#include "Weapons/Mortar.h"
#include "Weapons/Teleport.h"
#include "Weapons/WeaponNone.h"

Worms::Team::Team(std::vector<uint8_t> &playerIDs, std::vector<Player> &players,
                  const std::map<Worm::WeaponID, std::int16_t> &stageAmmo)
    : playerIDs(std::move(playerIDs)), ammunitionCounter(stageAmmo) {
    for (auto id : this->playerIDs) {
        players[id].setTeam(this);
    }
    this->initializeWeapons();
}

void Worms::Team::checkAlive(std::vector<Player> &players) {
    if (this->alive) {
        bool teamAlive = false;
        for (auto teamPlayerID : this->playerIDs) {
            if (players[teamPlayerID].getStateId() != Worm::StateID::Dead) {
                teamAlive = true;
            }
        }
        if (!teamAlive) {
            this->alive = false;
        }
    }
}

bool Worms::Team::isAlive() {
    return this->alive;
}

uint8_t Worms::Team::getCurrentPlayerID() {
    return this->playerIDs[this->currentPlayer];
}

void Worms::Team::setCurrentPlayer(uint8_t currentPlayer) {
    this->currentPlayer = currentPlayer;
}

void Worms::Team::endTurn(std::vector<Worms::Player> &players) {
    do {
        this->currentPlayer = (this->currentPlayer + 1) % this->playerIDs.size()
;
    } while (players[this->getCurrentPlayerID()].getStateId() == Worm::StateID::
Dead);
}

std::uint32_t Worms::Team::calculateTotalHealth(std::vector<Worms::Player> &play
ers) {
```



```
    std::uint32_t total{0};
    for (auto playerID : this->playerIDs) {
        for (auto &player : players) {
            if (player.getId() == playerID) {
                total += (std::uint32_t)std::floor(player.health);
            }
        }
    }
    return total;
}

std::shared_ptr<Worms::Weapon> Worms::Team::getWeapon(const Worm::WeaponID &id)
{
    if (this->ammunitionCounter.at(id) == 0) {
        return this->weaponNone;
    }

    switch (id) {
        case Worm::WeaponID::WBazooka:
            return this->bazooka;
        case Worm::WeaponID::WGrenade:
            return this->grenade;
        case Worm::WeaponID::WCluster:
            return this->cluster;
        case Worm::WeaponID::WMortar:
            return this->mortar;
        case Worm::WeaponID::WBanana:
            return this->banana;
        case Worm::WeaponID::WHoly:
            return this->holy;
        case Worm::WeaponID::WAerial:
            return this->aerialAttack;
        case Worm::WeaponID::WDynamite:
            return this->dynamite;
        case Worm::WeaponID::WBaseballBat:
            return this->baseballBat;
        case Worm::WeaponID::WTeleport:
            return this->teleport;
        default:
            return this->weaponNone;
    }
}

void Worms::Team::initializeWeapons() {
    this->aerialAttack = std::shared_ptr<Worms::Weapon>(new ::Weapon::AerialAtta
ck());
    this->banana = std::shared_ptr<Worms::Weapon>(new ::Weapon::Banana(0.0f));
    this->baseballBat = std::shared_ptr<Worms::Weapon>(new ::Weapon::BaseballBat
(0.0f));
    this->bazooka = std::shared_ptr<Worms::Weapon>(new ::Weapon::Bazooka(0.0f));
    this->cluster = std::shared_ptr<Worms::Weapon>(new ::Weapon::Cluster(0.0f));
    this->dynamite = std::shared_ptr<Worms::Weapon>(new ::Weapon::Dynamite());
    this->grenade = std::shared_ptr<Worms::Weapon>(new ::Weapon::Grenade(0.0f));
    this->holy = std::shared_ptr<Worms::Weapon>(new ::Weapon::Holy(0.0f));
    this->mortar = std::shared_ptr<Worms::Weapon>(new ::Weapon::Mortar(0.0f));
    this->teleport = std::shared_ptr<Worms::Weapon>(new ::Weapon::Teleport());
    this->weaponNone = std::shared_ptr<Worms::Weapon>(new ::Weapon::WeaponNone()
);
}

void Worms::Team::weaponUsed(const Worm::WeaponID weaponID) {
    if (this->ammunitionCounter.at(weaponID) > 0) {
```

```
        this->ammunitionCounter.at(weaponID)--;
    }
}

void Worms::Team::serialize(IO::GameStateMsg &msg) const {
    Worm::WeaponID weapons[] = {Worm::WBazooka,     Worm::WGrenade, Worm::WClust
er, Worm::WMortar,
                                Worm::WBanana,      Worm::WHoly,    Worm::WAeria
l,  Worm::WDynamite,
                                Worm::WBaseballBat, Worm::WTeleport};

    for (int i = 0; i < 10; i++) {
        msg.weaponAmmunition[i] = this->ammunitionCounter.at(weapons[i]);
    }
}

void Worms::Team::kill(std::vector<Worms::Player> &players) {
    for (auto &playerID : this->playerIDs) {
        players[playerID].die();
    }
    this->alive = false;
}
```

```
//
// Created by rodrigo on 3/06/18.
//

#ifndef INC_4_WORMS_TEAM_H
#define INC_4_WORMS_TEAM_H

#include <stdint.h>
#include <cstdint>
#include <map>
#include <vector>

#include "Weapons/Weapon.h"

namespace Worms {
class Player;
class Team {
   public:
    Team(std::vector<uint8_t> &playerIDs, std::vector<Player> &players,
         const std::map<Worm::WeaponID, std::int16_t> &stageAmmo);
    ~Team() = default;
    void checkAlive(std::vector<Player> &players);
    bool isAlive();
    uint8_t getCurrentPlayerID();
    void setCurrentPlayer(uint8_t currentPlayer);
    void endTurn(std::vector<Worms::Player> &players);
    std::uint32_t calculateTotalHealth(std::vector<Worms::Player> &players);
    std::shared_ptr<Weapon> getWeapon(const Worm::WeaponID &id);
    void weaponUsed(const Worm::WeaponID weaponID);
    void serialize(IO::GameStateMsg &msg) const;
    void kill(std::vector<Worms::Player> &players);

   private:
    std::vector<uint8_t> playerIDs;
    uint8_t currentPlayer{0};
    bool alive{true};
    std::shared_ptr<Weapon> aerialAttack{nullptr};
    std::shared_ptr<Weapon> banana{nullptr};
    std::shared_ptr<Weapon> baseballBat{nullptr};
    std::shared_ptr<Weapon> bazooka{nullptr};
    std::shared_ptr<Weapon> cluster{nullptr};
    std::shared_ptr<Weapon> dynamite{nullptr};
    std::shared_ptr<Weapon> grenade{nullptr};
    std::shared_ptr<Weapon> holy{nullptr};
    std::shared_ptr<Weapon> mortar{nullptr};
    std::shared_ptr<Weapon> teleport{nullptr};
    std::map<Worm::WeaponID, std::int16_t> ammunitionCounter;
    std::shared_ptr<Weapon> weaponNone;

    void initializeWeapons();
};
}  // namespace Worms

#endif  // INC_4_WORMS_TEAM_H
```

```
//
// Created by rodrigo on 16/06/18.
//

#include "Teleport.h"

#define CONFIG Game::Config::getInstance()

Weapon::Teleport::Teleport()
    : Weapon::Weapon(CONFIG.getTeleportConfig(), Worm::WeaponID::WTeleport, 0.0)
 {}

void Weapon::Teleport::update(float dt) {}

void Weapon::Teleport::startShot(Worms::Player *player) {}

void Weapon::Teleport::endShot() {}

void Weapon::Teleport::setTimeout(uint8_t time) {}

std::list<Worms::Bullet> Weapon::Teleport::onExplode(const Worms::Bullet &mainBu
llet,
                                                     Worms::Physics &physics) {
    return std::move(std::list<Worms::Bullet>());
}

void Weapon::Teleport::positionSelected(Worms::Player &p, Math::Point<float> poi
nt) {
    p.teleportPosition = point;
    p.notify(p, Event::Teleported);
    p.setState(Worm::StateID::Teleporting);
}

void Weapon::Teleport::increaseAngle() {}

void Weapon::Teleport::decreaseAngle() {}
```

```
//
// Created by rodrigo on 16/06/18.
//

#include "Teleported.h"
#include "../Config/Config.h"
#include "../Player.h"

Worms::Teleported::Teleported()
    : State(Worm::StateID::Teleported),
      teleportTime(Game::Config::getInstance().getTeleportTime()) {}

void Worms::Teleported::update(Worms::Player &p, float dt, b2Body *body) {
    this->timeElapsed += dt;
    if (this->timeElapsed >= this->teleportTime){
        p.setState(Worm::StateID::Falling);
    }
}

void Worms::Teleported::moveRight(Worms::Player &p) {}

void Worms::Teleported::moveLeft(Worms::Player &p) {}

void Worms::Teleported::jump(Worms::Player &p) {}

void Worms::Teleported::stopMove(Worms::Player &p) {}

void Worms::Teleported::backFlip(Worms::Player &p) {}

void Worms::Teleported::bazooka(Worms::Player &p) {}

void Worms::Teleported::pointUp(Worms::Player &p) {}

void Worms::Teleported::pointDown(Worms::Player &p) {}

void Worms::Teleported::startShot(Worms::Player &p) {}

void Worms::Teleported::endShot(Worms::Player &p) {}

void Worms::Teleported::grenade(Worms::Player &p) {}

void Worms::Teleported::cluster(Worms::Player &p) {}

void Worms::Teleported::mortar(Worms::Player &p) {}

void Worms::Teleported::banana(Worms::Player &p) {}

void Worms::Teleported::holy(Worms::Player &p) {}

void Worms::Teleported::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Teleported::aerialAttack(Worms::Player &p) {}

void Worms::Teleported::dynamite(Worms::Player &p) {}

void Worms::Teleported::teleport(Worms::Player &p) {}

void Worms::Teleported::baseballBat(Worms::Player &p) {}
```

```
//
// Created by rodrigo on 16/06/18.
//

#ifndef INC_4_WORMS_TELEPORTED_H
#define INC_4_WORMS_TELEPORTED_H

#include <stdint-gcc.h>
#include <cstdint>
#include "PlayerState.h"

namespace Worms {
class Teleported : public State {
   public:
    Teleported();
    ~Teleported() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    float teleportTime;
};
}

#endif  // INC_4_WORMS_TELEPORTED_H
```

```
//
// Created by rodrigo on 16/06/18.
//

#ifndef INC_4_WORMS_TELEPORT_H
#define INC_4_WORMS_TELEPORT_H

#include "../Player.h"
#include "Weapon.h"

namespace Weapon {
class Teleport : public Worms::Weapon {
   public:
    Teleport();
    ~Teleport() override = default;
    void update(float dt) override;
    void increaseAngle() override;
    void decreaseAngle() override;
    void startShot(Worms::Player *player) override;
    void endShot() override;
    void setTimeout(uint8_t time) override;
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &mainBullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override;
};
}  // namespace Weapon

#endif  // INC_4_WORMS_TELEPORT_H
```

```
//
// Created by rodrigo on 16/06/18.
//

#include "Teleporting.h"
#include <Camera.h>
#include "../Config/Config.h"
#include "../Player.h"

Worms::Teleporting::Teleporting(GUI::Position p)
    : State(Worm::StateID::Teleporting),
      newPosition(p),
      teleportTime(Game::Config::getInstance().getTeleportTime()) {}

void Worms::Teleporting::update(Worms::Player &p, float dt, b2Body *body) {
    this->timeElapsed += dt;
    if (this->timeElapsed >= this->teleportTime) {
        p.setPosition(this->newPosition);
        p.setState(Worm::StateID::Teleported);
    }
}

void Worms::Teleporting::moveRight(Worms::Player &p) {}

void Worms::Teleporting::moveLeft(Worms::Player &p) {}

void Worms::Teleporting::jump(Worms::Player &p) {}

void Worms::Teleporting::stopMove(Worms::Player &p) {}

void Worms::Teleporting::backFlip(Worms::Player &p) {}

void Worms::Teleporting::bazooka(Worms::Player &p) {}

void Worms::Teleporting::pointUp(Worms::Player &p) {}

void Worms::Teleporting::pointDown(Worms::Player &p) {}

void Worms::Teleporting::startShot(Worms::Player &p) {}

void Worms::Teleporting::endShot(Worms::Player &p) {}

void Worms::Teleporting::grenade(Worms::Player &p) {}

void Worms::Teleporting::cluster(Worms::Player &p) {}

void Worms::Teleporting::mortar(Worms::Player &p) {}

void Worms::Teleporting::banana(Worms::Player &p) {}

void Worms::Teleporting::holy(Worms::Player &p) {}

void Worms::Teleporting::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Teleporting::aerialAttack(Worms::Player &p) {}

void Worms::Teleporting::dynamite(Worms::Player &p) {}

void Worms::Teleporting::teleport(Worms::Player &p) {}

void Worms::Teleporting::baseballBat(Worms::Player &p) {}
```

```
//
// Created by rodrigo on 16/06/18.
//

#ifndef INC_4_WORMS_TELEPORTING_H
#define INC_4_WORMS_TELEPORTING_H

#include <Camera.h>
#include <stdint-gcc.h>
#include <cstdint>
#include "PlayerState.h"

namespace Worms {
class Teleporting : public State {
   public:
    Teleporting(GUI::Position p);
    ~Teleporting() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    void pointUp(Player &p) override;
    void pointDown(Player &p) override;

   private:
    float timeElapsed{0.0f};
    GUI::Position newPosition;
    float teleportTime;
};
}

#endif  // INC_4_WORMS_TELEPORTING_H
```

```cpp
#include "TouchSensor.h"
#include <iostream>

/**
 * @brief Construct a TouchSensor for the given body and shape.
 *
 * @param body Body that the touch sensor belongs to.
 * @param shape Sensor shape.
 */
Worms::TouchSensor::TouchSensor(b2Body &body, b2Shape &shape) : PhysicsEntity(En
tityID::Sensor) {
    /* fixture definition using the given shape */
    b2FixtureDef fixtureDef;
    fixtureDef.shape = &shape;
    fixtureDef.density = 1;
    fixtureDef.isSensor = true;

    this->fixture = body.CreateFixture(&fixtureDef);
    this->fixture->SetUserData(this);
}

Worms::TouchSensor::iterator Worms::TouchSensor::begin() {
    return this->contacts.begin();
}

Worms::TouchSensor::iterator Worms::TouchSensor::end() {
    return this->contacts.end();
}

/**
 * @brief Called whenever the sensor started contacting another entity.
 *
 * @param Contacted entity.
 */
void Worms::TouchSensor::startContact(PhysicsEntity *physicsEntity, b2Contact &c
ontact) {
    /* checks if the entity is in the ignore list */
    if (!this->isIgnored(physicsEntity)) {
        b2Manifold manifold;
        const b2Transform t1 = contact.GetFixtureA()->GetBody()->GetTransform();
        const b2Transform t2 = contact.GetFixtureB()->GetBody()->GetTransform();

        contact.Evaluate(&manifold, t1, t2);

        if (contact.GetFixtureA()->GetUserData() == physicsEntity) {
            this->contacts[physicsEntity] = -manifold.localNormal;
        } else {
            this->contacts[physicsEntity] = manifold.localNormal;
        }
    }
}

/**
 * @brief Called whenever the sensor stopped contacting another entity.
 *
 * @param Entity.
 */
void Worms::TouchSensor::endContact(PhysicsEntity *physicsEntity, b2Contact &con
tact) {
    /* checks if the entity is in the ignore list */
    if (!this->isIgnored(physicsEntity)) {
        this->contacts.erase(physicsEntity);
```



```cpp
    }
}

/**
 * @brief Whether the sensor is active or not (i.e. touching another body).
 *
 * @return true is active.
 */
bool Worms::TouchSensor::isActive() const {
    return (this->contacts.size() > 0);
}

/**
 * @brief Adds an entity that should be ignored by the sensor.
 *
 * @param other Entity to ignore.
 */
void Worms::TouchSensor::ignore(PhysicsEntity &other) {
    this->ignoredEntities.push_back(&other);
}

/**
 * @brief Checks if a given entity is in the ignore list.
 *
 * @param entity Entity to check.
 * @return true if the given entity is ignored by this sensor.
 */
bool Worms::TouchSensor::isIgnored(PhysicsEntity *entity) {
    return std::find(this->ignoredEntities.begin(), this->ignoredEntities.end(),
 entity) !=
           this->ignoredEntities.end();
}
```

```
#ifndef TOUCH_SENSOR_H_
#define TOUCH_SENSOR_H_

#include <unordered_map>
#include <vector>

#include "Physics.h"
#include "PhysicsEntity.h"

namespace Worms {
class TouchSensor : public PhysicsEntity {
   public:
    using iterator = std::unordered_map<PhysicsEntity *, b2Vec2>::iterator;

    TouchSensor(b2Body &body, b2Shape &shape);
    ~TouchSensor() = default;

    iterator begin();
    iterator end();

    bool isActive() const;
    void ignore(PhysicsEntity &other);

    void startContact(PhysicsEntity *physicsEntity, b2Contact &contact);
    void endContact(PhysicsEntity *physicsEntity, b2Contact &contact);

   private:
    bool isIgnored(PhysicsEntity *entity);

    b2Fixture *fixture{nullptr};
    std::vector<PhysicsEntity *> ignoredEntities;
    std::unordered_map<PhysicsEntity *, b2Vec2> contacts;
    std::unordered_map<PhysicsEntity *, b2Fixture *> contactFixtures;
};
}  // namespace Worms

#endif
```

```
#include <cmath>
#include <iostream>
#include <memory>

#include "../Player.h"
#include "Still.h"
#include "Walk.h"

void Worms::Walk::update(Player &p, float dt, b2Body *body) {
    float32 mass = body->GetMass();
    b2Vec2 vel = body->GetLinearVelocity();

    float final_vel{0.0f};

    if (!p.isOnGround()) {
        this->impulses[0] = -vel.x * mass;
        body->ApplyLinearImpulse(b2Vec2(impulses[0], impulses[1]), body->GetWorl
dCenter(), true);
        p.notify(p, Event::WormFalling);
        p.setState(Worm::StateID::Falling);
        return;
    }

    if (p.direction == Worm::Direction::left) {
        final_vel = -this->walkVelocity;
    } else {
        final_vel = this->walkVelocity;
    }

    this->impulses[0] = mass * (final_vel - vel.x);
    body->ApplyLinearImpulse(b2Vec2(this->impulses[0], this->impulses[1]), body-
>GetWorldCenter(),
                             true);

    p.lastWalkDirection = p.direction;

    this->timeElapsed += dt;
}

void Worms::Walk::moveRight(Worms::Player &p) {
    p.direction = Worm::Direction::right;
}

void Worms::Walk::moveLeft(Worms::Player &p) {
    p.direction = Worm::Direction::left;
}

void Worms::Walk::stopMove(Worms::Player &p) {
    p.setState(Worm::StateID::Still);
}

void Worms::Walk::jump(Worms::Player &p) {}

Worms::Walk::Walk()
    : State(Worm::StateID::Walk), walkVelocity(Game::Config::getInstance().getWa
lkVelocity()) {}

void Worms::Walk::backFlip(Worms::Player &p) {}

void Worms::Walk::bazooka(Worms::Player &p) {}
```



```
void Worms::Walk::pointUp(Worms::Player &p) {}

void Worms::Walk::pointDown(Worms::Player &p) {}

void Worms::Walk::startShot(Worms::Player &p) {}

void Worms::Walk::endShot(Worms::Player &p) {}

void Worms::Walk::grenade(Worms::Player &p) {}

void Worms::Walk::cluster(Worms::Player &p) {}

void Worms::Walk::mortar(Worms::Player &p) {}

void Worms::Walk::banana(Worms::Player &p) {}

void Worms::Walk::holy(Worms::Player &p) {}

void Worms::Walk::setTimeout(Worms::Player &p, uint8_t time) {}

void Worms::Walk::aerialAttack(Worms::Player &p) {}

void Worms::Walk::dynamite(Worms::Player &p) {}

void Worms::Walk::teleport(Worms::Player &p) {}

void Worms::Walk::baseballBat(Worms::Player &p) {}
```

```
#ifndef _PLAYERWALKLEFT_H
#define _PLAYERWALKLEFT_H

#include "../Config/Config.h"
#include "PlayerState.h"

namespace Worms {
class Walk : public State {
   public:
    Walk();
    ~Walk() = default;
    void update(Player &p, float dt, b2Body *body) override;
    void moveRight(Player &p) override;
    void moveLeft(Player &p) override;
    void jump(Player &p) override;
    void setTimeout(Player &p, uint8_t time) override;

    void bazooka(Player &p) override;
    void grenade(Player &p) override;
    void cluster(Player &p) override;
    void mortar(Player &p) override;
    void banana(Player &p) override;
    void holy(Player &p) override;
    void aerialAttack(Player &p) override;
    void dynamite(Player &p) override;
    void baseballBat(Player &p) override;
    void teleport(Player &p) override;

    void startShot(Player &p) override;
    void endShot(Player &p) override;
    void backFlip(Player &p) override;
    void stopMove(Player &p) override;
    virtual void pointUp(Player &p) override;
    virtual void pointDown(Player &p) override;

   private:
    const float walkVelocity;
    float timeElapsed{0.0f};
};
}

#endif  //_PLAYERWALKLEFT_H
```

```cpp
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 22/06/18
 */

#include "WeaponConfig.h"
#include "ConfigDefines.h"

Config::Weapon::Weapon(const YAML::Node &config)
    : dmgInfo(config[BULLET][DAMAGE]),
      minAngle(config[ANGLE][MIN].as<float>()),
      maxAngle(config[ANGLE][MAX].as<float>()),
      angleStep(config[ANGLE][STEP].as<float>()),
      maxShotPower((std::uint16_t)config[MAX_SHOT_POWER].as<unsigned int>()),
      restitution(config[BULLET][RESTITUTION].as<float>()),
      friction(config[BULLET][FRICTION].as<float>()),
      explotionInitialTimeout(
          (std::uint8_t)config[BULLET][EXPLOTION_INITIAL_TIMEOUT].as<unsigned in
t>()),
      hasAfterExplode(config[HAS_AFTER_EXPLODE].as<bool>()),
      bulletRadius(config[BULLET][RADIUS].as<float>()),
      bulletDampingRatio(config[BULLET][DAMAGE][DAMPING_RATIO].as<float>()),
      windAffected(config[BULLET][WIND_AFFECTED].as<bool>()) {}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 22/06/18
 */

#ifndef __WeaponConfig_H__
#define __WeaponConfig_H__

#include <cstdint>
#include "yaml-cpp/node/node.h"

#include "BulletConfig.h"

namespace Config {
struct Weapon {
    Bullet::DamageInfo dmgInfo;
    float minAngle;
    float maxAngle;
    float angleStep;
    std::uint16_t maxShotPower;
    float restitution;
    float friction;
    std::uint8_t explotionInitialTimeout;
    bool hasAfterExplode;
    float bulletRadius;
    float bulletDampingRatio;
    bool windAffected;

    explicit Weapon(const YAML::Node &config);
};
}  // namespace Config

#endif  //__WeaponConfig_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 28/05/18
 */

#include "Weapon.h"
#include "../Config/Config.h"
#include "../Player.h"
#include "../Config/WeaponConfig.h"

Worms::Weapon::Weapon(const Config::Weapon &config, Worm::WeaponID id, float ang
le)
    : config(config), id(id), angle(angle) {
    this->angle = angle;
    this->timeLimit = this->config.explotionInitialTimeout;
    /*
     * Because the limit angles between weapons are
     * differents, it is necesary to check boundaries angles.
     * If not, the game could crash in rendering time.
     */
    this->checkBoundaryAngles();
}

const Worm::WeaponID &Worms::Weapon::getWeaponID() const {
    return this->id;
}

void Worms::Weapon::decreaseAngle() {
    this->angle -= this->config.angleStep;
    if (this->angle < this->config.minAngle) {
        this->angle = this->config.minAngle;
    }
}

void Worms::Weapon::increaseAngle() {
    this->angle += this->config.angleStep;
    if (this->angle > this->config.maxAngle) {
        this->angle = this->config.maxAngle;
    }
}

float Worms::Weapon::getAngle() const {
    return this->angle;
}

void Worms::Weapon::checkBoundaryAngles() {
    if (this->angle > this->config.maxAngle) {
        this->angle = this->config.maxAngle;
    } else if (this->angle < this->config.minAngle) {
        this->angle = this->config.minAngle;
    }
}

Worms::BulletInfo Worms::Weapon::getBulletInfo() {
    return Worms::BulletInfo{this->config.dmgInfo,
                             Math::Point<float>{0, 0},
                             angle,
                             this->shotPower,
                             0,
                             this->config.restitution,
                             this->config.friction,
                             this->timeLimit,
```



```
                             this->config.hasAfterExplode ? Event::OnExplode : E
vent::Explode,
                             this->config.bulletRadius,
                             this->config.bulletDampingRatio,
                             this->config.windAffected};
}

void Worms::Weapon::setAngle(float angle) {
    this->angle = angle;
}

bool Worms::Weapon::isP2PWeapon() {
    return this->isP2P;
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 28/05/18
 */

#ifndef __WEAPON_H__
#define __WEAPON_H__

#include <GameStateMsg.h>
#include <list>
#include <memory>

#include "../Config/Config.h"
#include "Bullet.h"
#include "../Config/WeaponConfig.h"

namespace Worms {
class Player;
class Weapon {
   public:
    Weapon(const Config::Weapon &config, Worm::WeaponID id, float angle);
    virtual ~Weapon() = default;

    const Worm::WeaponID &getWeaponID() const;
    /**
     * If was an event of startShot, then increase its power shot until
     * reach its limit.
     * @param dt
     */
    virtual void update(float dt) = 0;
    /**
     * @brief increases the angle of the aim. If the angle exceeds the limit
     * then it will be changed to the maximum possible
     */
    virtual void increaseAngle();
    /**
     * @brief decreases the angle of the aim. If the angle exceeds the limit
     * then it will be changed to the maximum possible
     */
    virtual void decreaseAngle();
    float getAngle() const;
    void setAngle(float angle);
    virtual void startShot(Worms::Player *player) = 0;
    virtual void endShot() = 0;
    BulletInfo getBulletInfo();
    virtual void setTimeout(uint8_t time) = 0;
    /**
     * @brief check if the weapon is person to preson or not
     * @return
     */
    bool isP2PWeapon();
    /**
     * Used by te remote control weapons. Sends to the weapon the coordinates
     * of the deploy of the bullets, and a reference of Player so that the
     * weapons, if they are remote control. calls the appropiate method.
     * @param player to call deploy method (if the weapon has this feature)
     * @param point
     */
    virtual void positionSelected(Worms::Player &p, Math::Point<float> point) =
0;
    /**
     * Function that returns, using move semantics, a list of bullets
```



```
     * depending on weapon's behavior after the main bullet explode.
     * @return
     */
    virtual std::list<Worms::Bullet> onExplode(const Worms::Bullet &mainBullet,
                                               Worms::Physics &physics) = 0;

   protected:
    bool increaseShotPower{false};
    float shotPower{0};
    bool isP2P{false};
    const Config::Weapon &config;
    Worm::WeaponID id;
    float angle{0};
    uint8_t timeLimit;

   private:
    /**
     * When weapons change, their own limit angles may crash the game.
     * To avoid this, this function checks and correct angles between changes.
     */
    virtual void checkBoundaryAngles();
};
}  // namespace Worms

#endif  //_WEAPON_H__
```

```cpp
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 24/06/18
 */

#include "WeaponNone.h"

#define CONFIG Game::Config::getInstance()

Weapon::WeaponNone::WeaponNone()
    : Weapon::Weapon(CONFIG.getTeleportConfig(), Worm::WeaponID::WNone, 0.0) {}

std::list<Worms::Bullet> Weapon::WeaponNone::onExplode(const Worms::Bullet &main
Bullet,
                                                       Worms::Physics &physics)
{
    return std::move(std::list<Worms::Bullet>());
}
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 24/06/18
 */

#ifndef __WEAPON_NONE_H__
#define __WEAPON_NONE_H__

#include "Weapon.h"

namespace Weapon {
class WeaponNone : public Worms::Weapon {
   public:
    WeaponNone();
    ~WeaponNone() override = default;
    void update(float dt) override{};
    void increaseAngle() override{};
    void decreaseAngle() override{};
    void checkBoundaryAngles() override{};
    void startShot(Worms::Player *player) override{};
    void endShot() override{};
    void setTimeout(uint8_t time) override{};
    std::list<Worms::Bullet> onExplode(const Worms::Bullet &mainBullet,
                                       Worms::Physics &physics) override;
    void positionSelected(Worms::Player &p, Math::Point<float> point) override{}
;
};
}  // namespace Weapon

#endif  //__WEAPON_NONE_H__
```

```
/*
 *  Created by Federico Manuel Gomez Peter.
 *  date: 22/06/18
 */

#ifndef __WIND_CONFIG_H__
#define __WIND_CONFIG_H__

#include "yaml-cpp/node/node.h"

namespace Config {
struct Wind {
    float minIntensity;
    float maxIntensity;
    int xDirection;
    float instensity;
};
}  // namespace Config

#endif  //__WIND_CONFIG_H__
```

Table of Contents