Documentación Técnica
Wolfenstein 3D

[75.42-95.08] Taller de Programación I
Cátedra Veiga
Segundo cuatrimestre de 2020

Betz, Fontela, Kovnat, Rosenblatt
104348, 103924, 104429, 104105

Índice

1. Requerimientos de Software

Ademas de las librerias que utiliza el juego, el unico otro requisito que necesita el trabajo es contar con una computadora que utilice GNU/Linux como sistema operativo.

2. Descripcion General

2.1. Funcionamiento General del sistema

En un principio, teníamos la idea de que cada cliente pueda simular en su propia computadora el modelo y solo transmitirle al servidor lo necesario para retransmitirle al resto de los usuarios. Mientras que esto puede servir para ciertos juegos, en un FPS no es posible ya que podrían darse situaciones en las que dos usuarios generen el mismo input pero tengan resultados distintos, por lo que se generaría una desincronización entre los clientes. Es por esto que requerimos un modelo autoritario del lado del servidor. El servidor es quien simula el juego, mientras que los usuarios solo le pueden comunicar sus intenciones al servidor, que luego corre la simulación y genera la respuesta apropiada, dejando a todos los jugadores sincronizados.

Los clientes, a traves de sus teclas, envían por el socket un mensaje siguiendo un protocolo, que luego un handler del lado del servidor toma y encola el comando que simule lo que el usuario espera. Los comandos son sacados de la cola de manera FIFO por el Engine, quien es el encargado de tomar estos comandos, ejecutarlos y enviar las respuestas apropiadas a los usuarios. El motor del juego simula el paso del tiempo para aquellas partes del modelo que lo requieran (Por ejemplo, un cohete moviendose por el espacio del mapa) Estas notificaciones son luego recibidas por el usuario para actualizar su representacion visual del mapa, sea moviendo sprites de lugar o reproduciendo sonidos como respuesta a una acción.

3. Distribución de Clases

3.1. Cliente

El cliente se ocupa mayoritariamente de enviar, recibir y actuar ante comandos emitidos por el servidor. Posee varias clases que se ocupan de organizar, distribuir y renderizar la información que administra.

3.2. Dibujado

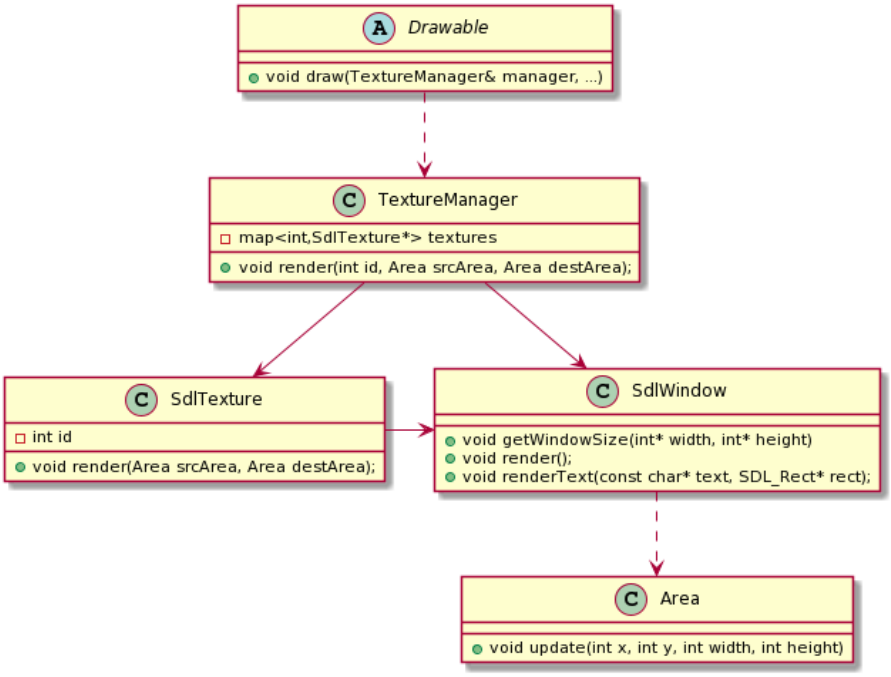

Figura 1: Renderizado de Imágenes

A lo largo de la ejecución del programa, clases como *Drawable* llaman constantemente a la clase *TextureManager* para dibujar texturas.

Esta clase siempre dibuja recibiendo un id y llamando al método *render()* de las texturas. El TextureManager mapea cada id con el recurso a dibujar, y todas las clases tienen un acceso sencillo a cada id utilizando las macros del protocolo del cliente.

Ejemplo de código donde se ve como se busca la textura a dibujar:

```
void TextureManager::render(int id, const Area& srcArea, const Area& destArea) {
  std::unordered_map<int, SdlTexture*>::iterator it = this->textures.find(id);
  if (it != this->textures.end())
    it->second->render(srcArea, destArea);
  else
    LOG_WITH_ID(TEXTURE_NOT_FOUND_ERROR);
}
```

Luego la textura llama la ventana para que la renderice en los espacios dictados por las instancias de *Area*

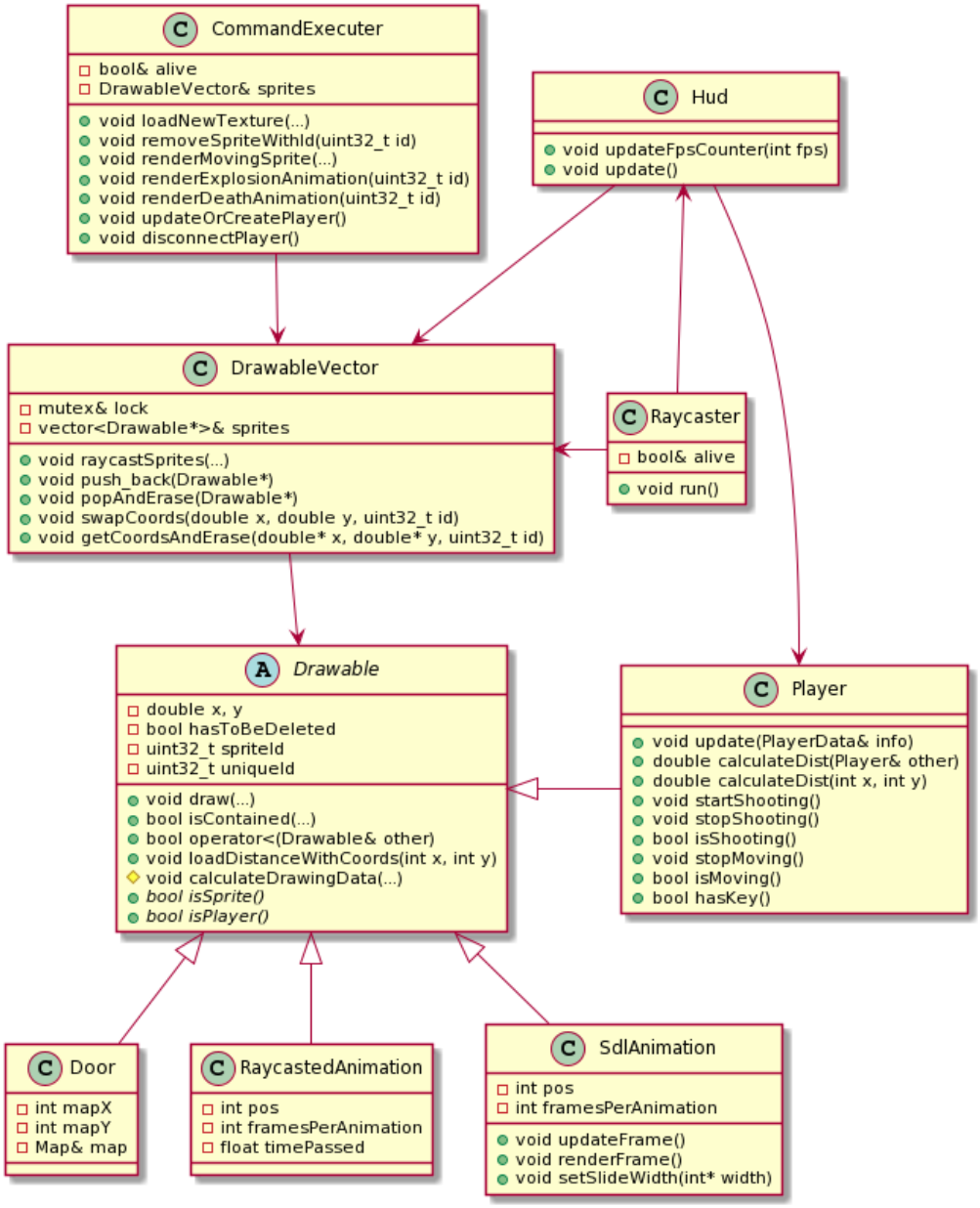

Figura 2: Cálculos y Generación de Gráficos

Las instancias de RayCaster, CommandExecuter y Hud poseen referencias al vector de dibujables. Este, que posee un mutex para evitar condiciones de carrera (ya que por ejemplo, el CommandExecuter y el Raycaster corren en hilos diferentes) administra un vector que tiene todas las referencias a los sprites. Estas pueden ser instancias de Player si se debe dibujar un enemigo en pantalla, RaycastedAnimation si se necesita raycastear una animación (muerte de un enemigo o explosión de un misil), SdlAnimation si se necesita dibujar una animación estática, Door si se quiere raycastear una tira de píxeles de la puerta o Drawable si es un simple dibujable sin mucho comportamiento extra.

3.2.1. Raycasting

El Raycaster dibuja dentro de un while que opera mientras el atomic bool (compartido por el CommandExecuter y el CommandSender) sea true.

Este necesita obtener datos de nuestro jugador, tales como sus coordenadas en el mapa $Pos = (x, y)$, el vector que representa la dirección a la que mira $Dir = (dirX, dirY)$ y otro vector al que llamaremos $Plane = (PlaneX, PlaneY)$ siendo que $Plane \perp Dir$.

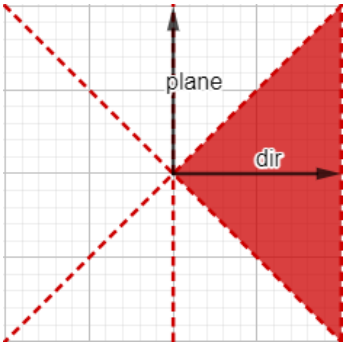

Figura 3: Campo de Visión del Jugador (Área Roja).

El campo de visión del jugador está definido por lo que se puede ver en $Pos + Dir + \alpha \cdot Plane$ con $\alpha \in [-1, 1]$ y cualquier rotación que se quiera generar se puede calcular sobre el vector a raycastear haciendo $R(\omega) \cdot (Pos + Dir + \alpha \cdot Plane)$ que corresponde a multiplicar por la matriz de rotación en función de ω (velocidad angular)

Una vez que obtiene estos datos comienza a, valga la redundancia, castear rayos para detectar cuál es el primer objeto con el que se encuentra para rayo casteado (uno por cada columna de píxeles en la pantalla) utilizando el algorítmo de DDA (que calcula, iteración a iteración, el desplazamiento que tiene que realizar el rayo, en cada eje, para llegar justo a interesectar con la próxima celda que se va a encontrar).

Dentro de la ejecución del DDA verificamos constantemente si lo que encontramos es una pared o una puerta (caso en el cual debemos guardar en una lista de puertas la tira que encontramos para renderizar, siempre y cuando la puerte esté lo "suficientemente" abierta). El DDA calculará la altura de cada elemento a dibujar y se lo guardará en un array (cuyo tamaño es el ancho de la pantalla) en el que cada posición representa el alto de la pared o puerta a dibujar.

Luego, las puertas, paredes y sprites y Hud reciben la información resultante relevante y para todas las clases que necesiten actualizar su tiempo, el while del raycaster tendrá un clock interno que podrá suministrar esta información.

3.2.2. Animaciones Estáticas

Para las animaciones estáticas, es decir, no raycasteables, renderizamos cada frame con el siguiente método:

```
void SdlAnimation::renderActualFrame(Area& destArea, int textureId) {
    int width, height;
    this->manager.getTextureSizeWithId(textureId, &width, &height);
    width /= this->picsPerAnimation;
```

```
        Area srcArea(width * pos + (pos - 1), 0, width, height);
        this->manager.render(textureId, srcArea, destArea);
    }
```

En el cual, el srcArea corresponde al área de la imagen a cortar y el destArea corresponde al área de la pantalla a dibujar. El efecto de animación se consigue haciendo que la textura que representa la misma tenga todos los frames uno al lado del otro. Es decir, si tengo una animación donde cada frame tiene 50x50px, la textura de la animación tendrá 150x50px, y se seleccionará uno de los 3 posibles frames a dibujar setteando la variable *pos* en algún valor entre 1 y la cantidad de frames.

Por ejemplo, en la siguiente textura que representa la animación del disparo de la pistola con sus 5 frames:

Digamos que la textura general es de 250x50px y queremos hacer que se muestre la animación en pantalla. Lo que debemos hacer es llamar al método *renderActualFrame()* de la animación de disparo, pasándole el ID de esta textura y el destArea (que define donde se dibuja). Para el TextureManager debemos pasarle un parámetro extra que será el espacio a cortar de la imagen, y esto es trivial ya que conocemos el alto y el ancho de cada frame ($width/picsPerAnimation$). Solo falta seleccionar el i-ésimo frame a cortar, cosa que se obtiene comenzando a cortar en el píxel $width * i + (i - 1)$

Para generar la sensación de animación solo queda actualizar el frame a renderizar cada T tiempo o iteraciones del loop del raycaster.

3.2.3. Animaciones Dinámicas

La lógica de renderizado de las animaciones dinámicas, es decir, las raycasteables, es la misma que la del raycasting de sprites (por eso hereda de Drawable), pero a diferencia de estos se debe actualizar el frame a dibujar. Esto acabamos de ver (en las animaciones estáticas) que no es difícil de implementar y solo consiste en desplazar el crop en función del frame a dibujar.

3.3. Esqueleto General

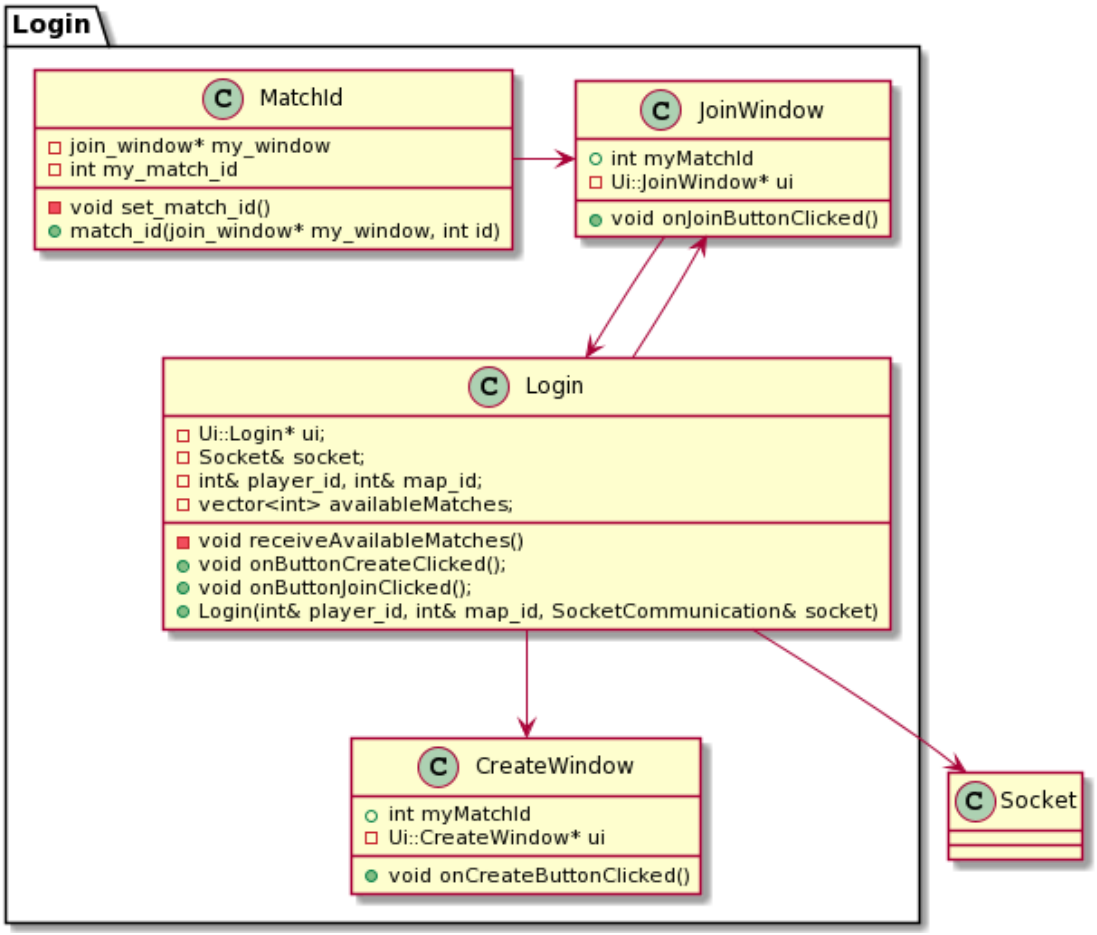

Figura 4: Login al Juego.

La ventana de Login, generada con *Qt* permite que el usuario pueda ingresar la ip y puerto necesarios para conectarse al servidor. La misma le permite al usuario elegir una partida de las disponibles o crear la suya.

Al introducir los datos, el Login se intenta conectar al servidor y cuando se conecta, puede recibir una lista de partidas vigentes a las que conectarse.

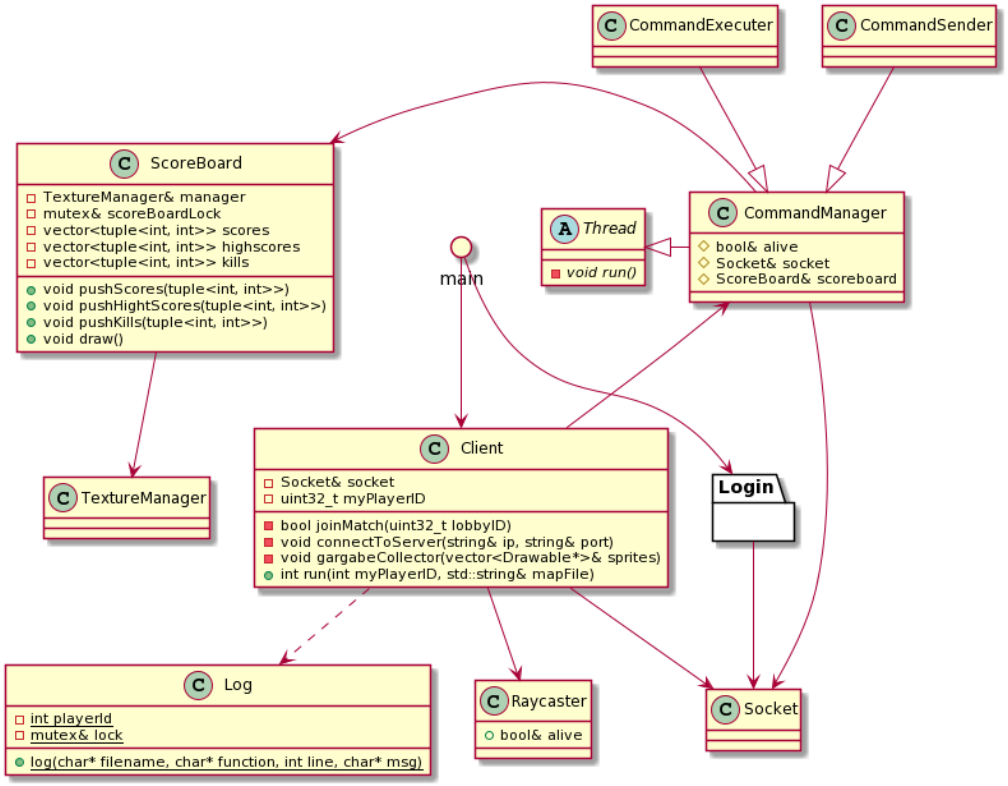

Figura 5: Esqueleto General del Cliente

Al iniciar el ejecutable del cliente se llama al Login, que conecta con el servidor y redirecciona los datos al mismo. Este y todas sus instancias que pueden usar el Log de errores, posee tres grandes pilares que soportan la jugabilidad del mismo. El Raycaster, que corre en el hilo principal, soporta el dibujado de clases. El CommandExecuter y CommandSender heredan de CommandManager ya que las dos referencian al Socket y ScoreBoard y deben poder correr en hilos distintos. El CommandExecuter recibe información del servidor y la redirecciona mientras que el CommandSender envía el input del teclado.

3.3.1. CommandExecuter

El método principal de esta clase es su run. Este inicializa la música y comienza a escuchar al servidor por sus opcodes. A medida que van llegando, redirecciona la información a otros métodos que contactan con los objetos correspondientes para manipular el estado del juego.

```
void CommandExecuter::run() {
  this->audiomanager.playWithId(MUSIC);
  while (!this->scoreboard->hasEnded()) {
    try {
      uint32_t opcode;
      socket.receive(&opcode, sizeof(opcode));
      if (opcode == PLAYER_UPDATE_PACKAGE) {
        this->updateOrCreatePlayer();
      } else if (opcode == PLAYER_DISCONNECT) {
        this->disconnectPlayer();
      } else if (opcode == SHOTS_FIRED) {
        this->shotsFired();
```

```
      } else if (opcode == OPEN_DOOR) {
        this->openDoor();
      } else if (opcode == PLAYER_PICKUP_ITEM) {
        this->pickUpItem();
      } else if (opcode == PLAYER_DIED) {
        this->playerDied();
      } else if (opcode == PLAYER_DROP_ITEM) {
        this->dropItem();
      } else if (opcode == ELEMENT_SWITCH_POSITION) {
        this->elementSwitchPosition();
      } else if (opcode == MISSILE_EXPLOTION) {
        this->explodeMissile();
      } else if (opcode == ENDING_MATCH) {
        this->saveScores();
        this->saveKills();
        this->saveShotsFired();
        alive = false;
      }
    } catch (SocketException& e) {
    } catch (std::exception& e) {
      LOG(e.what());
      break;
    }
  }
}
```

3.3.2. CommandSender

El CommandSender, que corre dentro de otro hilo, toma constantemente el input del jugador. Su método run constantemente llama a SDL_WaitEvent(event) el cual espera a que el jugador emita un evento que lo despierte. Esto puede disparado por apretar o soltar un botón, mover el mouse, querer cerrar la ventana, entre otros.

Por lo tanto, como también ocurre en el CommandExecuter, nuestra clase trabajará hasta que el atomic boolean que tiene adentro (compartido por el Raycaster y CommandExecuter) sea setteado en false.

Si se detecta que presiona alguna de las teclas que representan un comando de jugador (sección 4.3) el código correspondiente será enviado por el socket al servidor.

También puede detectar un evento SDL_QUIT que ocurre si de alguna forma la ventana recibe la directiva de cerrarse. Esto hace que el siguiente código se ejecute:

```
      SDL_Event event;
      SDL_WaitEvent(&event);
      if (event.type == SDL_QUIT) {
        uint32_t opcode = PLAYER_DISCONNECT;
        socket.send(&opcode, UINT32_SIZE);
        socket.readShutdown();
        socket.writeShutdown();
        socket.close();
        alive = false;
```

```
        this->scoreboard->stop();
        break;
    }
```

Donde el evento nos obliga a enviarle al servidor una señal de desconexión, a cerrar y destruir el socket, a settear en false el atomic boolean y parar el dibujado del scoreboard por si se estuviese dibujando.

3.4. Audio y Mapa

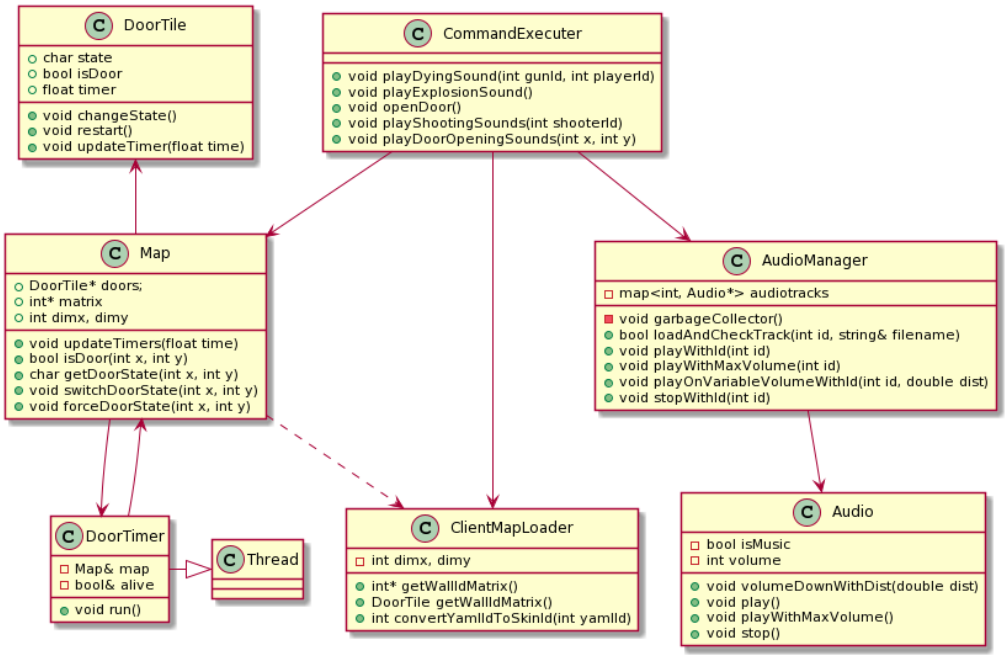

Figura 6: Data Flow para el Audio y Mapa

3.4.1. Mapa

El mapa en este juego se maneja principalmente mediante la clase Map. Esta tiene un vector de int y DoorTile que representan una matriz de enteros (cuyos valores, si son diferente de cero, mapean dentro del protocolo del cliente al id de la textura de una puerta) y una matriz de tiles para puertas respectivamente. La matriz de enteros se utiliza para saber que pared dibujar en cada iteración del raycasting y los DoorTiles conocen el estado de cada puerta del juego.

El DoorTimer, que corre en otro hilo, constantemente toma el tiempo y actualiza los timers de DoorTile a través de Map.

Map conoce constantemente, tanto donde están las paredes y sprites del mapa, como la distribución de puertas y el estado en que cada una de esas se encuentra cosntantemente. Esto permite poder obtener el tipo de sonido que tiene que hacer una puerta en función de si es secreta o no, y forzar un cerrado si fuera necesario, utilizando sus métodos.

El DoorTile que posee el mapa es una matriz representada por un puntero, lo que significa que para obtener el elemento de la posición $doors[x][y]$ se debe acceder mediante la siguiente fórmula:

```
doors + y + x * dimy
```

Donde el desplazamiento en y te ubica dentro de la fila deseada mientras que $x * dimy$ te deja en la columna correspondiente.

3.4.2. Audio

El CommandExecuter posee una referencia al Audio Manager. El mismo, similar al TextureManager, mapea cada archivo de audio con su id (que puede ser facilmente buscado en el protocolo del cliente) para evitar mútiples cargas innecesarias. Nuestra instancia de AudioManager sabe manipular cada objeto Audio para que pueda reproducirse, detenerse o cambiar el volumen como se desee.

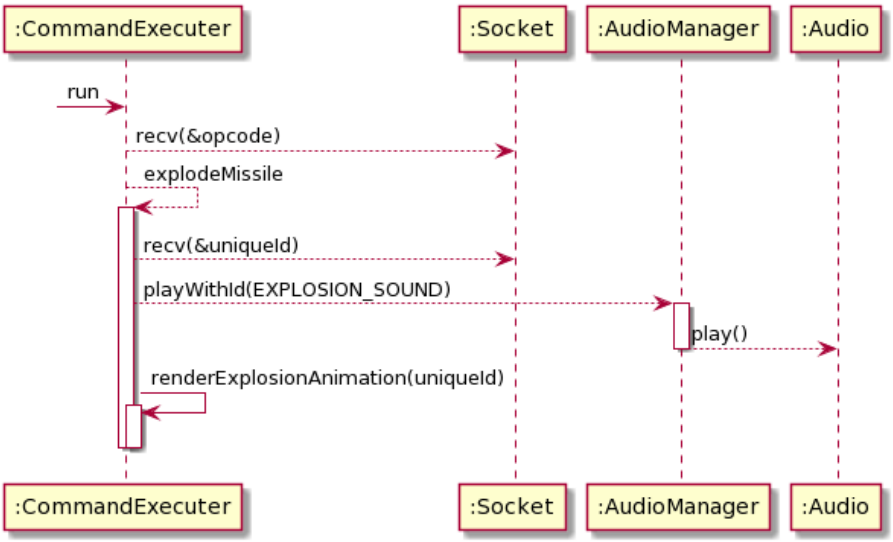

Figura 7: Como se Reproduce el Sonido de la Explosión

En el ejemplo de este diagrama de secuencia podemos apreciar como CommandExecuter hace que suene la explosión del misil cuando le llega la señal del server. Al saber que acción debe ejecutar sabe que automaticamente va a recibir un uniqueId para la textura de la explosión y justo después de darle play al sonido se comienza a animar la explosión.

3.5. Editor

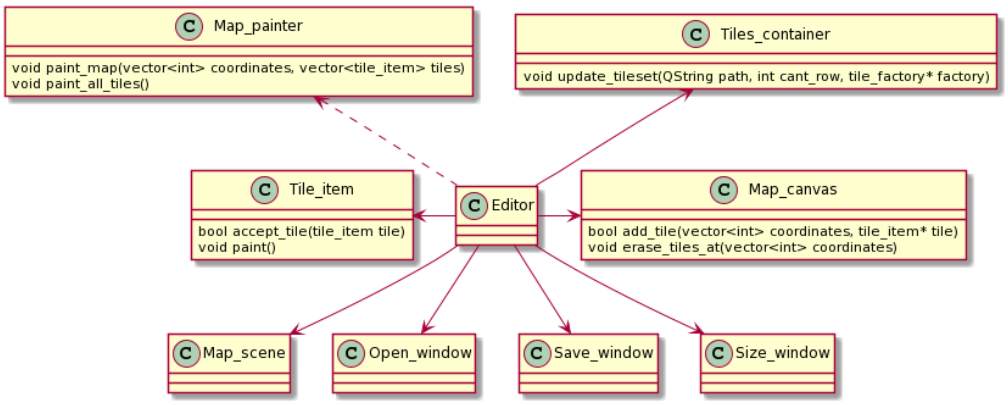

Figura 8: Primer Diagrama del Editor.

El editor se maneja alrededor de la ventana principal que es un objeto que hereda del mainwindow llamado Editor. Este se encarga de controlar todas las ventanas emergentes tales como el Size_Window que se crea al presionar la accion *new* y se encarga de pedirle los datos del tamaño de mapa deaseado al usuario, el Open_Window que se crea al presionar la accion *open* y se encarga de mostrarle al usuario los mapas guardados y abrir el seleccionado y el Save_Window que se crea al presionar la accion save (cuando el mapa abierto no tiene un nombre) y se encarga de guardar el mapa actual con el nombre pasado por el usuario.

El editor tambien tiene acciones de "Zoom_In" y "Zoom_Out" que permiten repintar el Map_Scene al tañano deseado y ademas una toolbar donde se ubican acciones con todos los tipos de tiles disponibles. Al presionar alguno de ellos este actualiza el Tiles_Container.

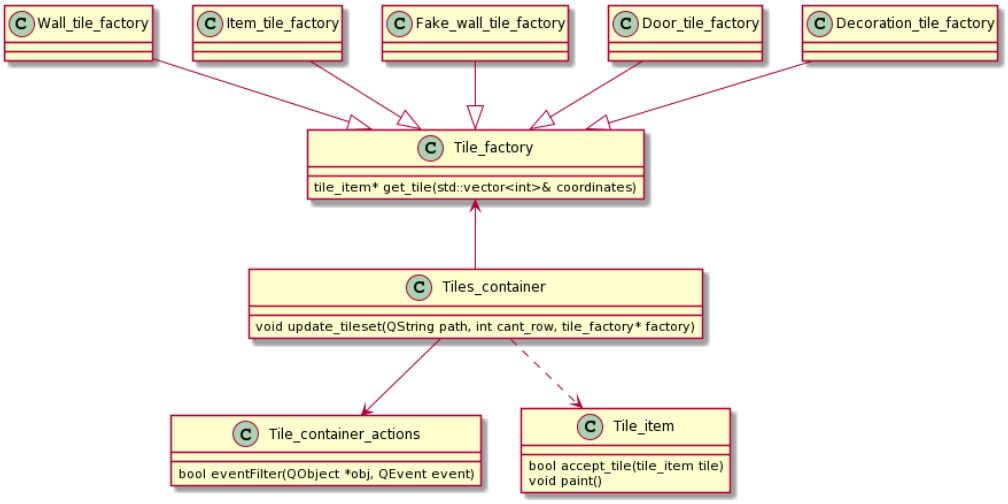

Figura 9: Segundo Diagrama del Editor.

El Tile_Container se encarga del manejo gráfico de los tilesets y contiene un atributo de clase Tile_Factory que se encarga de crear el Tile_Item correspondiente a la posicion del click del mouse y asignarselo al editor como Tile_Item_Selected.

Los eventos del Tile_Container estan regulados por su Eventfilter; Tile_Container_Actions.

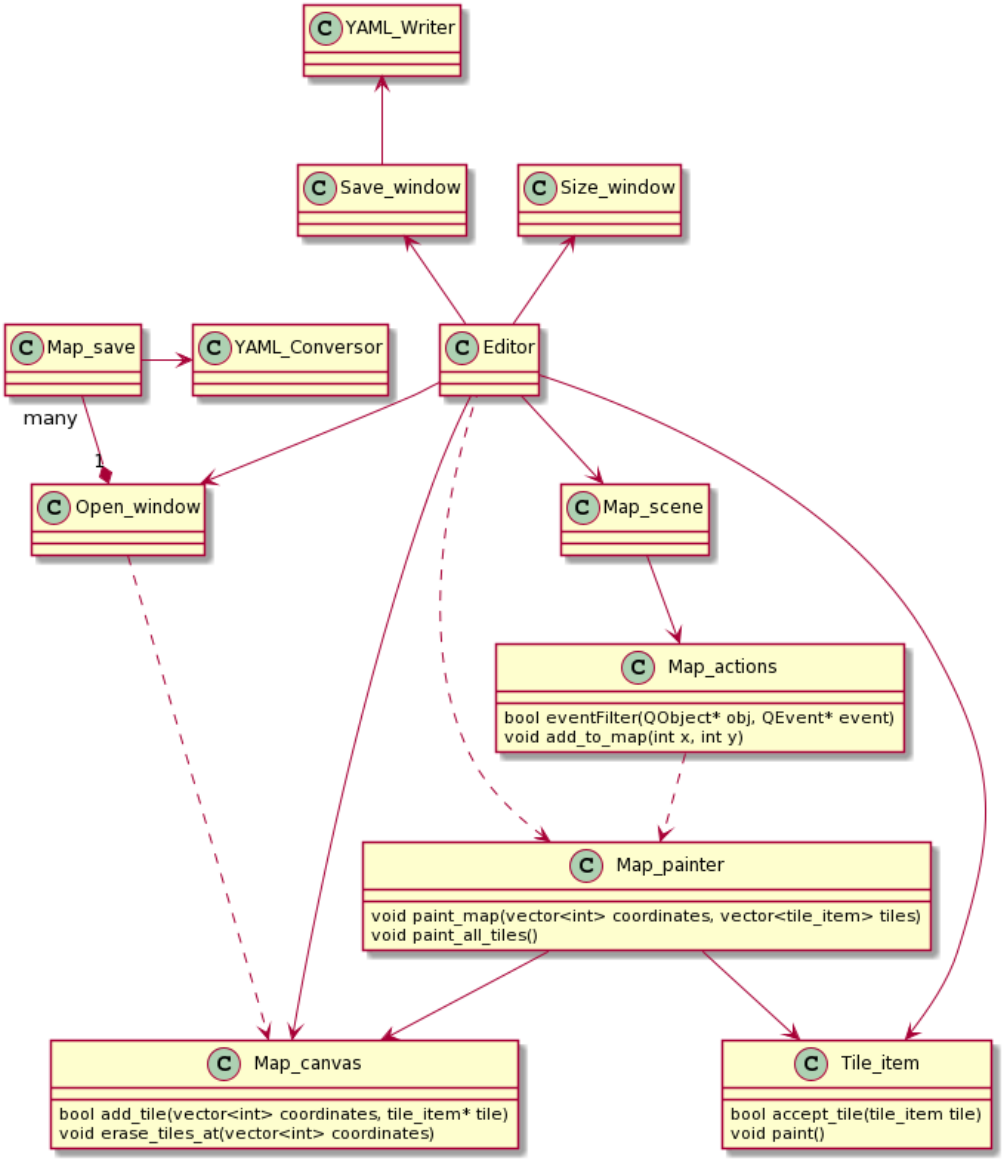

Figura 10: Tercer Diagrama del Editor.

El Editor guarda los cambios visualizados en el Map_Scene en una clase llamada Map_Canvas que se encarga de almacenar los Tiles_Items en la posicion correspondiente a la grilla mostrada en el Map_Scene.

Los eventos del Map_Scene estan regulados por su Eventfilter; Map_actions y este utiliza al Map_Painter para administrar los cambios visuales.
En la Open_Window se utiliza un objeto llamado YAML_Conversor para convertir el archivo Yaml en una matriz de Tile_Items. En la Save_Window se utiliza un objeto llamado YAML_Writer para convertir a una matriz de Tile_Items en un archivo yaml.

3.6. Servidor

3.6.1. Conectividad

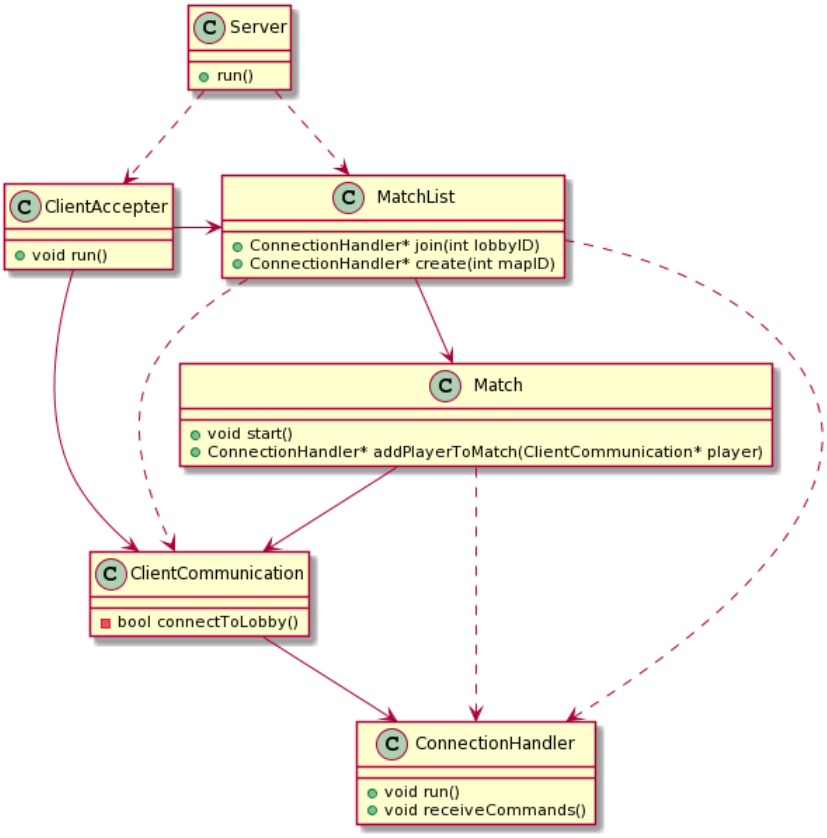

Figura 11: Connection Side del servidor

El servidor cuenta con un listener, una clase encargada de escuchar sobre un puerto y aceptar jugadores, creando una instancia de ClientCommunication que cuenta con información basica para comunicarse con dicho cliente: El socket por el cual comunicarse y su ID unico asignado por el servidor. Una vez establecida la conexión inicial se pasa al protocolo de iniciar sesión en el juego. El cliente puede elegir crear una partida o conectarse a una ya disponible. Si elige crear la partida, envia el opcode establecido en el protocolo junto con el ID del mapa con el que se desea crear la partida y el ClientCommunication le pide a una lista de partidas que cree una que cuente con el mapa especificado, ademas de brindarle derechos de administrador a ese jugador sobre la partida (Dandole asi la posibilidad de comenzarla cuando desee) En cambio, si decide unirse a una partida existente, el servidor primero le comunica al cliente una lista de partidas que tiene disponibles en ese momento para que el usuario tome su decision. Una vez que tome su decisión, el ClientCommunication le pide a la lista de partidas que agregue a ese jugador a la partida que especificó. En cualquiera de los dos casos, el servidor puede rechazar la conexión a la partida (Por ejemplo, porque el servidor no encuentra el mapa especificado por el usuario) y en ese caso el servidor le transmitirá al cliente dicho rechazo, siguiendo el protocolo diseñado. En caso de que la conexión sea aceptada, le asigna al ClientCommunication un ConnectionHandler que cuenta con la Queue de comandos y es quien sabe manejar los opcodes relacionados al modelo y que comandos generar

3.6.2. Motor

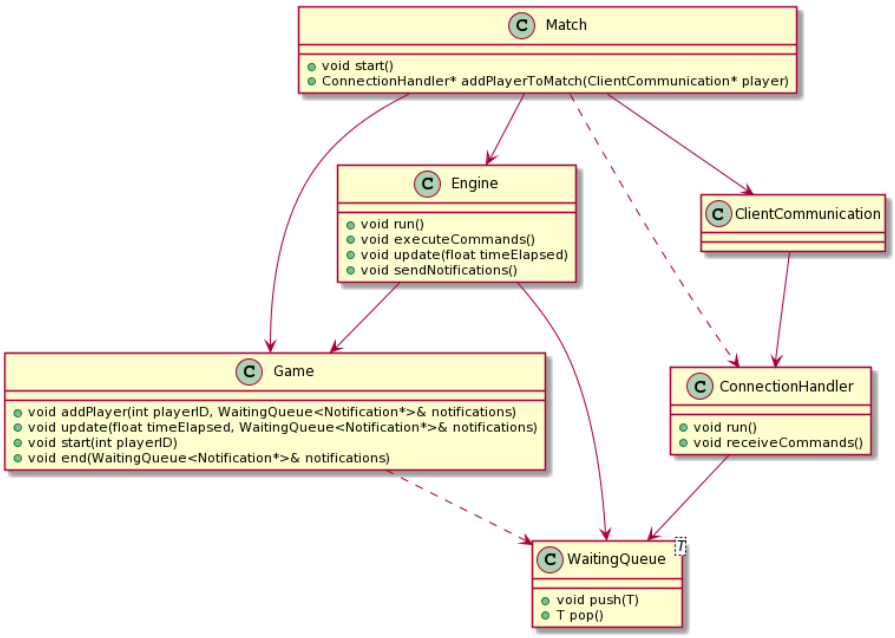

Figura 12: Relaciones con el motor del juego

El Engine cuenta con dos colas: una de comandos, donde todos los clientes pueden insertar los comandos que desean y otra de notificaciones generadas por el modelo como respuesta a los comandos del usuario. El motor constantemente esta sacando comandos y notificaciones, ejecutando la primera y enviando la segunda. Cada una de estas funciones corre en un hilo aparte manejado por el motor, mientras que su hilo principal simula el tiempo, corriendo a 30 ticks por segundo y en cada tick del mismo le pide al modelo del juego que actualize las variables dependientes del tiempo.

3.6.3. Juego

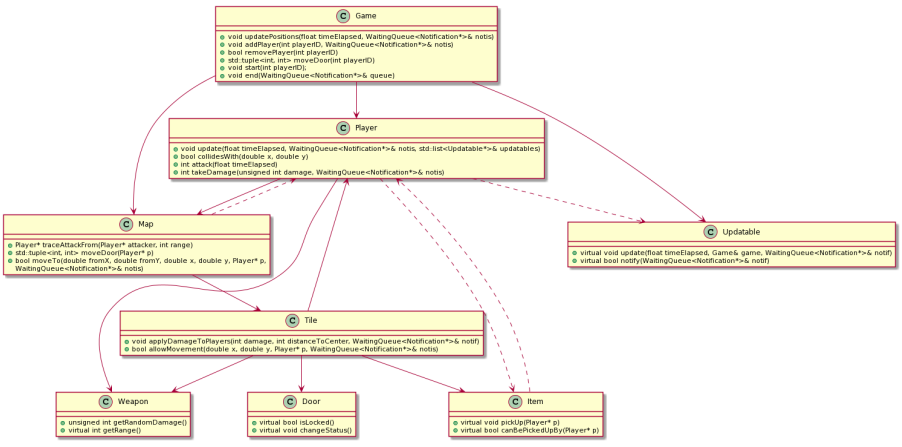

Figura 13: Esquema de la relacion entre clases del modelo

La clase Game es la clase que contiene los datos necesarios para el funcionamiento del juego y el responsable de modificar el modelo, sea delegando la tarea a otras clases o ejecutandolas ella misma. Esta clase cuenta con un mapa de jugadores y una lista de eventos actualizables.

El juego ademas cuenta con un mapa, que a su vez consiste de celdas que contienen los items y armas con los que el jugador puede interactuar si atraviesa dicha celda. Ademas, estas celdas contienen un puntero a los jugadores que actualmente estan ubicados en ellas. Cualquier movimiento que el jugador quiera hacer requiere una .ªutorización"previa del mapa, que revisa que no haya colisiones con paredes o con otros jugadores. Si el mapa aprueba del movimiento, se encarga de mover el jugador que hizo el movimiento de una celda a otra y le delega a la celda destino la responsabilidad de forzar la interacción entre el jugador y los items o armas que hayan en la misma.

3.6.4. Actualizables

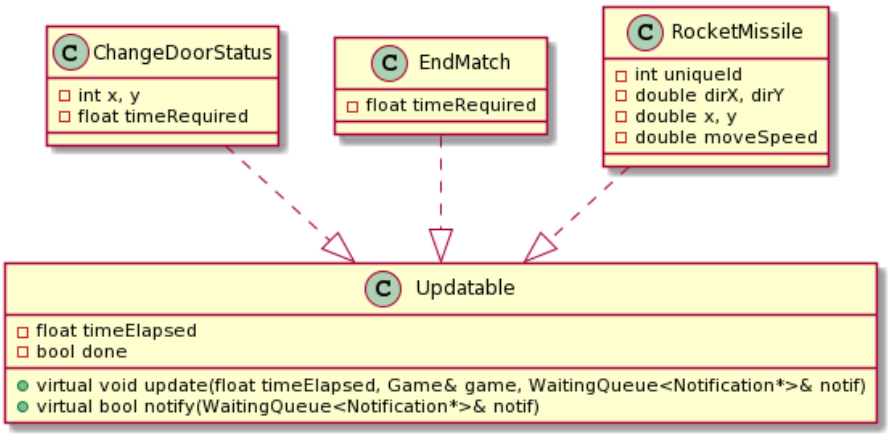

Figura 14: Relacion entre los actualizables

Como mencionamos anteriormente, el juego cuenta con una lista de actualizables que son eventos que requieren de un tiempo especificado para ejecutarse. Al encapsular todos estos eventos bajo la misma interfaz Updatable me permito añadir en un futuro cualquier evento que desee mientras que respete la interfaz que provee Updatable (Por ejemplo, se puede añadir un evento que sea una granada que explote despues de cierto tiempo)

En cada actualización que se le hace al modelo, el juego le pasa a cada item dentro de la lista el tiempo elapsado desde la ultima actualización para que el actualizable decida que hacer con ese tiempo y genere una acción si lo requiere, siempre notificando al jugador cuando sea necesario. Una vez que el evento termino, la función notify devolverá true indicando que el evento ya terminó y puede ser removido seguramente de la lista.

3.6.5. Armas

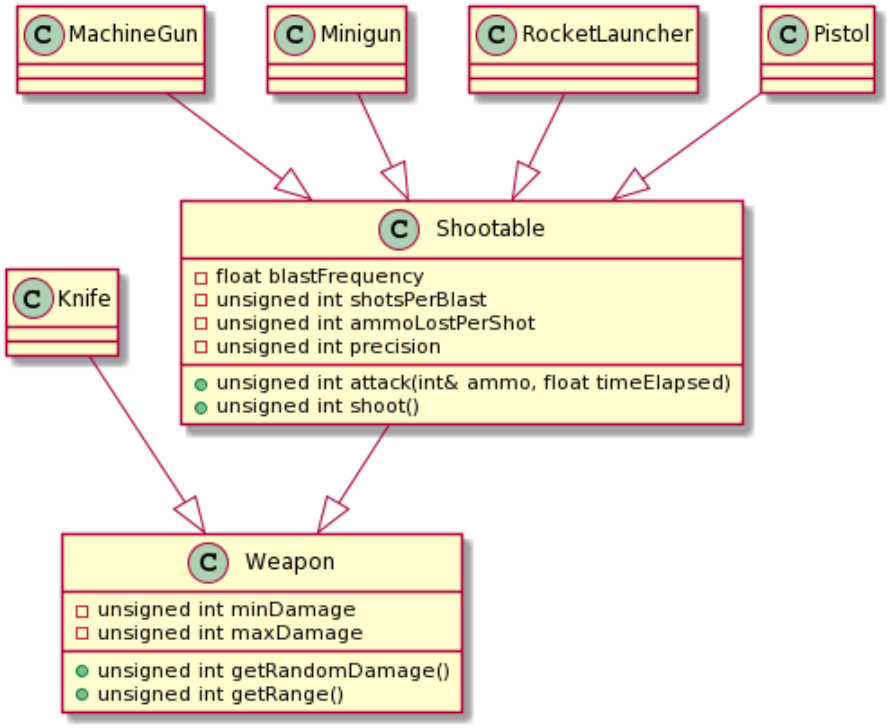

Figura 15: Relacion entre las armas

Similar al caso de los actualizables, decidimos segregar el diseño en ïnterfaces"que luego implementarían las armas particulares que requería el juego. Los jugadores tienen una lista de armas y tienen un arma activa, a la hora de disparar el jugador le pide al arma que gestione la cantidad de balas que dispara y el daño que generarían las mismas (Siempre basandose en el tiempo elapsado), evitando asi que el jugador sepa sobre el funcionamiento interno de como se dispara un armas y como se genera el daño.

4. Descripción de Archivos y Protocolos

El unico archivo externo que se requiere es el archivo del mapa que se quiera jugar. Por temas de simplicidad, se requiere que todo mapa tenga un nombre que siga el formato "map- id del mapa + ".yaml".

4.1. Protocolo del Cliente

El cliente posee un archivo clientprotocol.h que posee una lista de defines importantes utilizados para facilitar el manejo de opcodes de todo tipo y algunas macros imporantes.

Entre las más importanes están:

- LOG(...) Que loggea el error que se le pasa utilizando la función estática del Log. Viene bien para pasar macros relevantes como __FUNCTION__ o __LINE__ sin ocupar mucho espacio en la llamada al método.
- AUDIO AND TEXTURE CODES Que corresponden a los ids de los sonidos y texturas manejados por sus gestores correspondientes.

- DOOR STATES Conjunto de valores de un byte que representan el estado que puede tomar cualquier puerta.
- Paths a archivos.
- MACROS Para calcular los desplazamientos que se deben realizar en los crops de texturas para simular animaciones.
- RGB VALUES
- ERRORS, MORE MACROS, ...

4.2. Protocolo del Servidor

Todas las definiciones del protocolo se encuentran en el archivo protocol.h dentro de common/includes.

4.2.1. Conectividad

Para conectarse, se puede enviar cualquiera de estos dos opcodes:

- **JOIN LOBBY:** Una vez recibido este opcode, el server responde con las partidas disponibles y el cliente debera responder con el numero de partida elegido.
- **CREATE LOBBY:** Se debe enviar este opcode seguido del ID del mapa que se desea utilizar.

Como respuesta final del servidor en cuanto a conectividad, se pueden recibir dos respuestas.

- **CONNECTED OK:** Seguido de este opcode se envía el ID unico asignado al cliente y, en caso de haber enviado JOIN LOBBY se envia tambien el ID del mapa a cargar.
- **CONNECTION REFUSED:** Ante un rechazo del servidor, se envía este opcode.

4.3. Comandos del jugador

Los siguientes opcodes son bastante explicativos, envia el estado de las teclas del jugador

- **KEY A DOWN**
- **KEY D DOWN**
- **KEY S DOWN**
- **KEY W DOWN**
- **KEY A UP**
- **KEY S UP**
- **KEY D UP**
- **KEY W UP**
- **KEY 1 DOWN**
- **KEY 2 DOWN**
- **KEY 3 DOWN**
- **KEY 4 DOWN**
- **KEY 5 DOWN**
- **OPEN DOOR:** Mapeado al Key Down de la tecla E.
- **START MATCH:** Mapeado al Key Down tecla P, inicia la partida.

4.4. Notificaciones al cliente

Los siguientes opcodes son parte de las notificaciones que el servidor genera para los clientes.

- **PLAYER UPDATE PACKAGE:** A continuación, recibe el struct PlayerData serializado.
- **PLAYER PICKUP ITEM:** A continuación, envia el ID unico del item a borrar.
- **PLAYER DROP ITEM:** A continuación, envia sus coordenadas (X, Y), el ID de YAML y su ID unico.
- **PLAYER DIED:** A continuación, envia el ID del jugador.
- **ELEMENT SWITCH POSITION:** A continuación, el ID unico y sus coordenadas (X, Y).
- **MISSILE EXPLOTION:** A continuación, envia el ID unico.
- **PLAYER DISCONNECT:** A continuación, el ID del jugador.
- **SHOTS FIRED:** A continuación, recibe el ID del atacante.
- **OPEN DOOR:** A continuación, recibe las coordenadas (X, Y) de la puerta.