🧾 Documentación Técnica – Counter Strike 2D

📌 Introducción

Este proyecto consiste en el desarrollo de un videojuego multijugador tipo Counter Strike en 2D, con cliente y servidor implementados en C++.

El juego incluye lógica de lobby, rondas, disparos, economía, bomba, físicas, y comunicación por sockets utilizando un protocolo binario.

👨‍💻 Desarrolladores

- Sebastián Kraglievich - 109038
- Agustin Perez Romano - 109367
- Morena Sandroni - 110205
- Mateo Bulnes - 106211

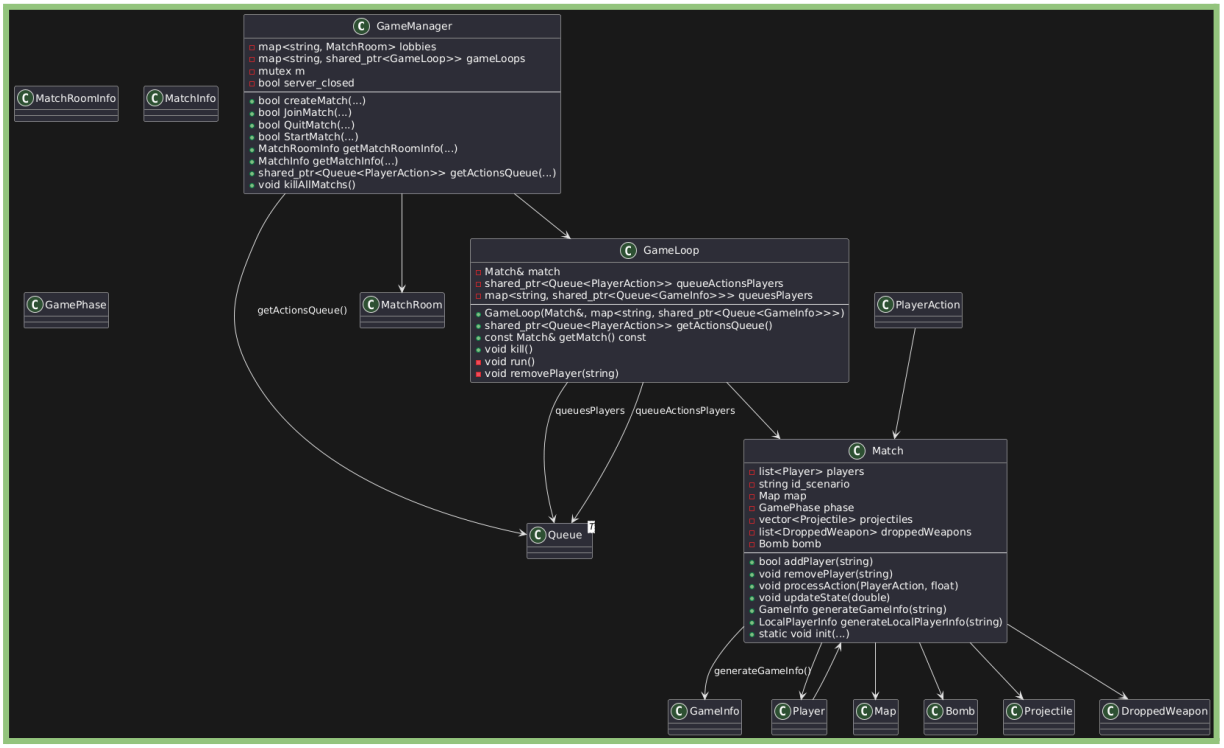
🧱 Arquitectura General

- El proyecto se estructura en **capas lógicas**:
    - **Capa Baja:** manejo de sockets y transporte binario.
    - **Capa Media:** protocolo de comunicación (serialización/deserialización).
    - **Capa Alta:** lógica del juego (partida, jugadores, armas, tienda).
- Comunicación cliente-servidor full dúplex con manejo de múltiples partidas concurrentes a través de hilos (uno por cliente).

Servidor

🕹️ Flujo de juego

En el siguiente diagrama se muestran las clases que interactúan en el flujo de juego, desde la gestión de partidas, el gameloop y el manejo de acciones y estados durante la partida.



Clases principales:

- **GameManager:** Gestiona las acciones relacionadas con las partidas (crear o unirse a una partida, salir de la partida, comenzar la partida)
- **Match:** Controla la lógica completa de una partida
- **Gameloop:** Ejecuta iterativamente la lógica del Match asociado, desencoplando PlayerActions, mandandolas a procesar y actualizando el estado con Match y encolando la información para el cliente en forma de GameInfo

Funciones destacadas:

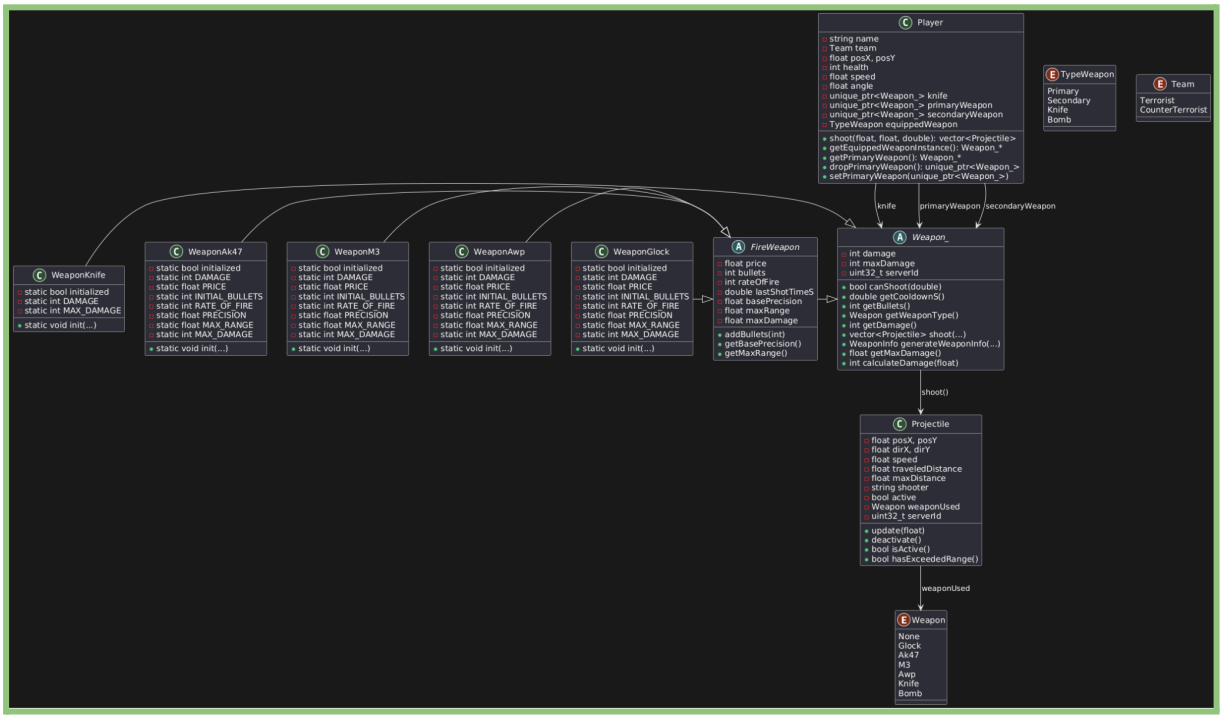
- **Match::processAction():** Procesa acciones del jugador (moverse, disparar, plantar la bomba, comprar, pickup, etc)

- **Match::updateState(elapsedTime):** Avanza el estado del juego (disparos, bomba, proyectiles)
- **Match::advancePhase():** Gestiona la transición entre fases de una ronda

---

🔫 Sistema de armas

El siguiente diagrama muestra las clases principales encargadas del sistema de armas y disparos.



Clases principales:

- **Weapon_:** Clase abstracta base para todas las armas en el juego
- **FireWeapon:** Clase abstracta que hereda de Weapon_ y a su vez es base para todas las armas de fuego del juego.
- **WeaponAk47, WeaponM3, WeaponAwp, WeaponGlock:** Clase representando a un arma de fuego particular. Hereda de FireWeapon y define el método shoot() con el tipo de disparo específico del arma y sus configuraciones específicas obtenidas del archivo de configuración.
- **WeaponKnife**: Clase que representa al cuchillo. Hereda de Weapon_.
- **Projectile**: Clase que representa a una bala, contiene quien la disparó y con que arma y funciona independiente del resto de las balas

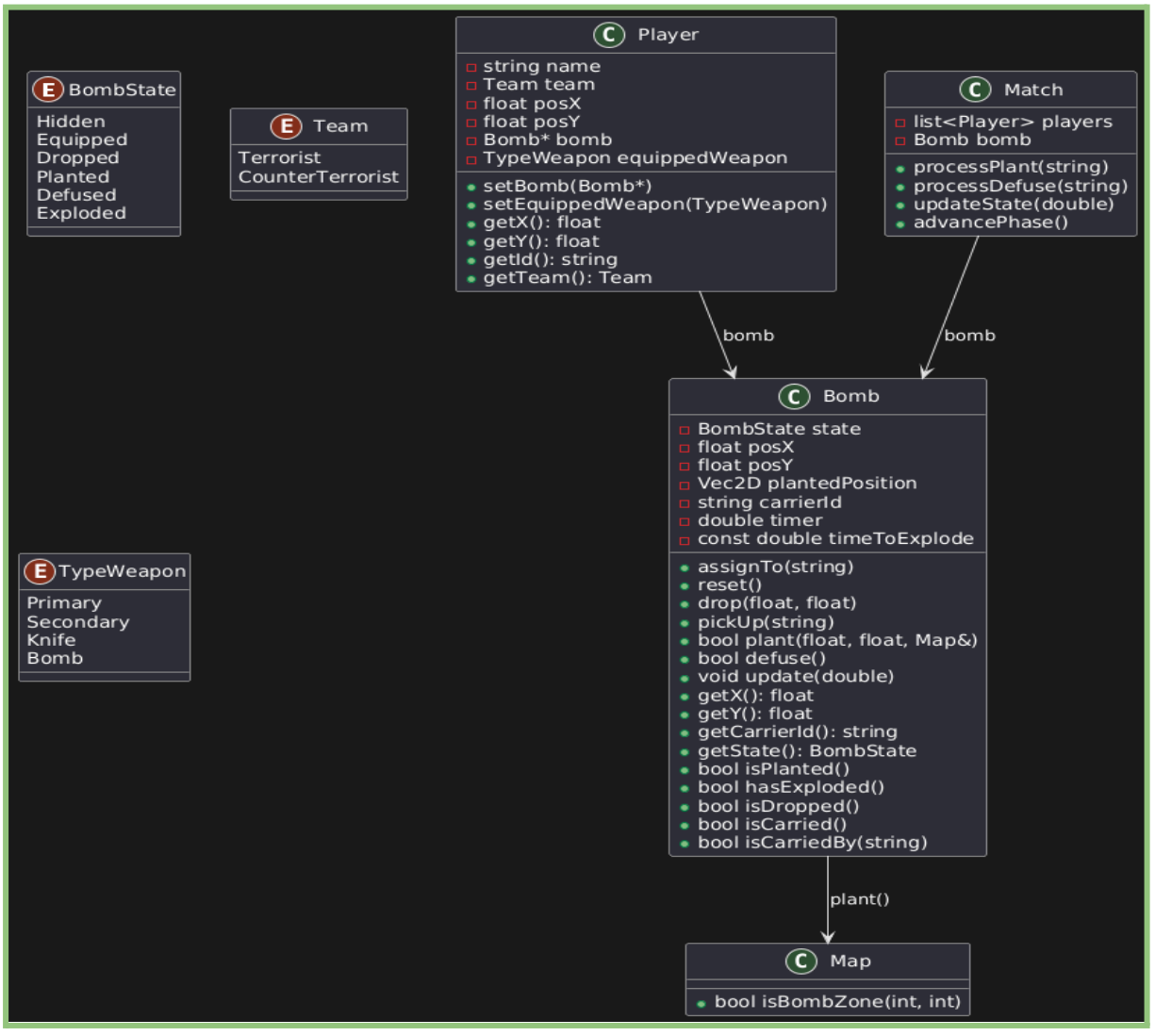
Funciones destacadas:

- **WeaponAk47::shoot() (aplica para todas las armas):** Implementa el disparo específico del arma (ak47 en rafaga de a 3, M3 disparo de dispersión, etc)
- **calculateDamage():** hace el calculo de daño específico para cada arma, aplicando factores de aleatoriedad según corresponda.
- **canShoot()**: Indica si un arma puede disparar o no, dependiendo del cooldown

---

💣 Sistema de bomba

El siguiente diagrama representa el sistema de la bomba, con las clases involucradas en el flujo de la funcionalidad de la bomba.



Clases principales:

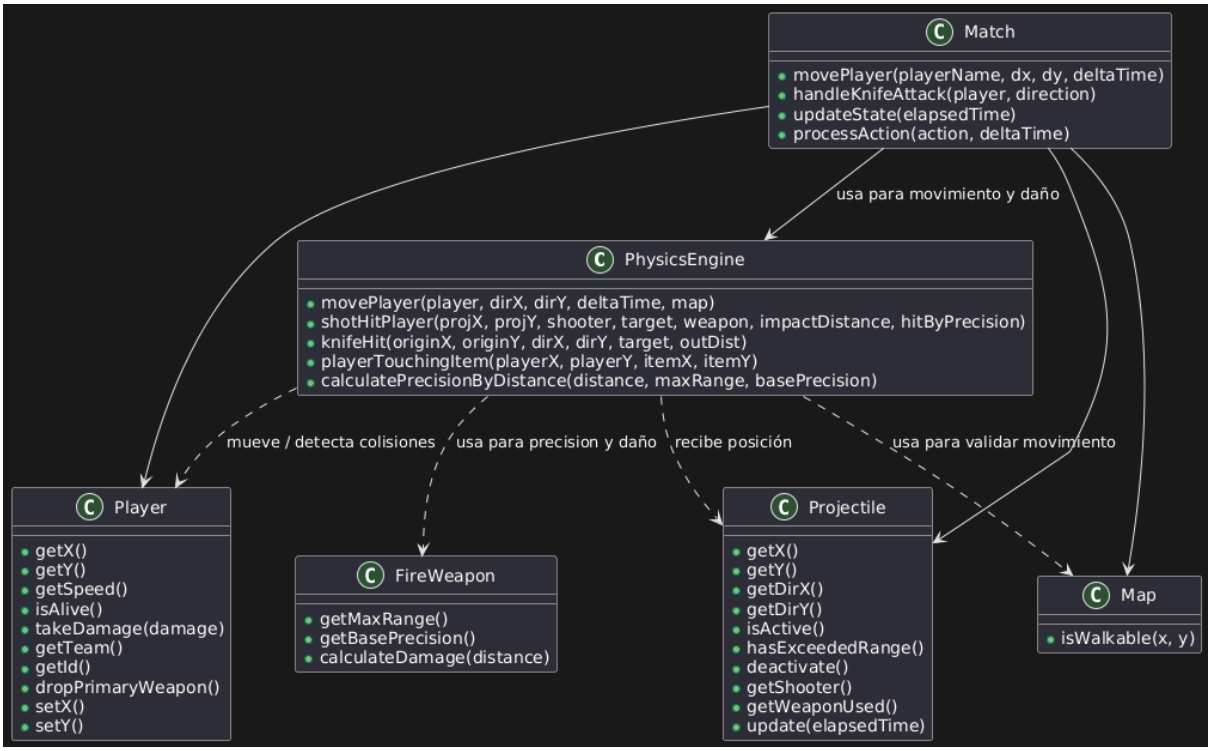
- **Bomb:** Maneja toda la lógica de la bomba, colocación, recolección, reseteo, explosión y desactivación.

Funciones destacadas:

- **Bomb::plant():** Planta la bomba validando si la posición donde se quiere plantar es la zona de la bomba
- **Bomb::defuse()**: Desactivar la bomba
- **Bomb::update():** Actualiza el estado de la bomba, restando al timer si la misma está plantada y explotandola si se termino el tiempo.

⚛ Sistema de Físicas

En el siguiente diagrama se muestran las clases relacionadas con el motor de físicas utilizado para la detección de colisiones y distancias del juego.



Clases principales:

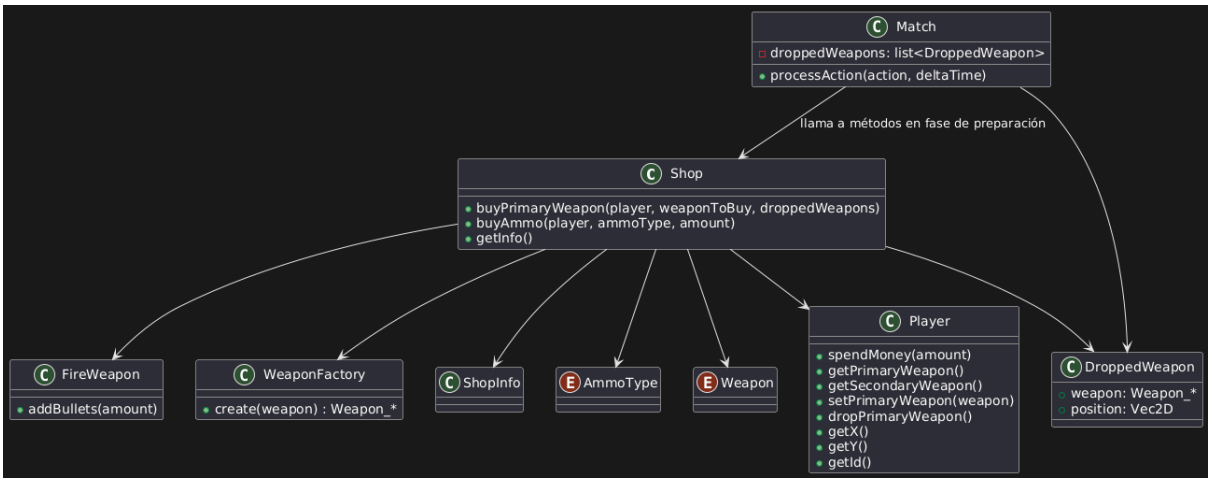
- **PhysicsEngine:** Contiene las funciones necesarias para la detección de colisiones con objetos, colisiones con balas y contacto entre el jugador e items. Además contiene funciones para calcular precisión y daño proporcional a la distancia del disparo

Funciones destacadas:

- **PhysicsEngine::movePlayer():** Función encargada de validar si un jugador puede moverse a una posición detectando colisiones con obstáculos en el mapa.
- **PhysicsEngine::shotHitPlayer()**: En base a la posición de un proyectil y la posición de un jugador, detecta si colisionan o no. Luego calcula la precisión en base a la distancia desde donde se disparó y define si el disparo fue acertado o no.
- **PhysicsEngine::calculatePrecisionByDistance()**: Dada la distancia desde donde se realizo el disparo y el alcance máximo del arma, se define la precisión del disparo.
- **PhysicsEngine::playerTouchingItem()**: Dada la posición de un objeto (lo utilizamos para pickup de armas y bomba) y la posición del jugador, determina si estan en contacto.

🛒 Sistema de Tienda

En el siguiente diagrama se muestran las clases involucradas en el sistema de la tienda para compra de armas y munición



Clases principales:

- **Shop:** Contiene los maps con el precio de cada arma y munición y la cantidad de balas en un cartucho por arma. Además de la lógica para la compra de armas y municiones

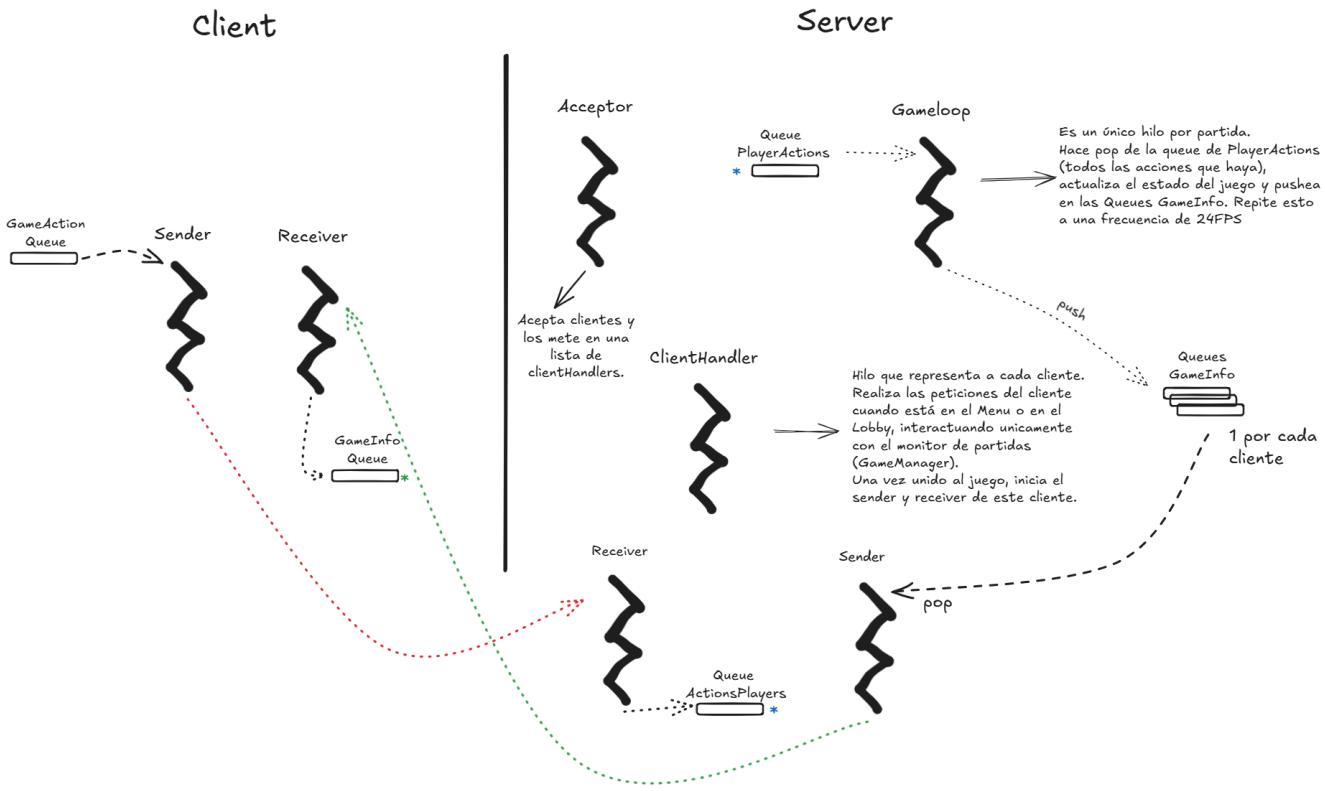
Funciones destacadas:

- **Shop::buyWeapon():** Función encargada de validar si un jugador tiene saldo suficiente para comprar el arma seleccionada, en caso positivo realiza el cambio de arma, dropeando la actual y equipando la nueva, restando el saldo correspondiente al jugador

- **Shop::buyAmmo()**: Misma lógica que buyWeapon, hace la validación de si la compra de municiones se puede realizar y aumenta la cantidad de balas del arma correspondiente, restando el saldo acorde.

🛜 Conexión Cliente - Servidor

En el siguiente diagrama se muestra el manejo de hilos en la comunicación entre el cliente y el servidor.



...

👤 Cliente

Responsabilidad

En el presente proyecto, tomamos la decisión de que toda la lógica de negocio del juego esté encapsulada en el servidor, quitándole dicha responsabilidad al cliente y logrando que el mismo sólo se encargue de recibir la información desde el servidor y renderice el juego de forma correcta.

Estados del cliente

Como se mencionó previamente, cuando el cliente ejecuta el programa, pasa a través de diferentes etapas. En una primera instancia debe loguearse en la aplicación, luego pasa al menú (donde puede crear o unirse a una partida), y por último juega la partida propiamente dicha.

Para modelar esto, hemos decidido implementar una especie de máquina de estados, en donde se transiciona de un estado a otro dependiendo de la interacción del cliente con la aplicación.

En este punto las clases relevantes son:

- **AppState**: Abstracción que representa un estado posible en el que se encuentra el cliente en un momento dado.
- **AppStateController**: Encargado de controlar el pasaje de un estado al siguiente, cumpliendo con los conceptos RAII.
- **LoginAppState, MainMenuAppState, LobbyAppState, GameMatchAppState**:
    - Estos son los estados concretos que encapsulan la lógica particular propia de cada estado. Esto nos permitió implementar por separado las features de cada estado y poder utilizar librerías gráficas diferentes para cada estadío.

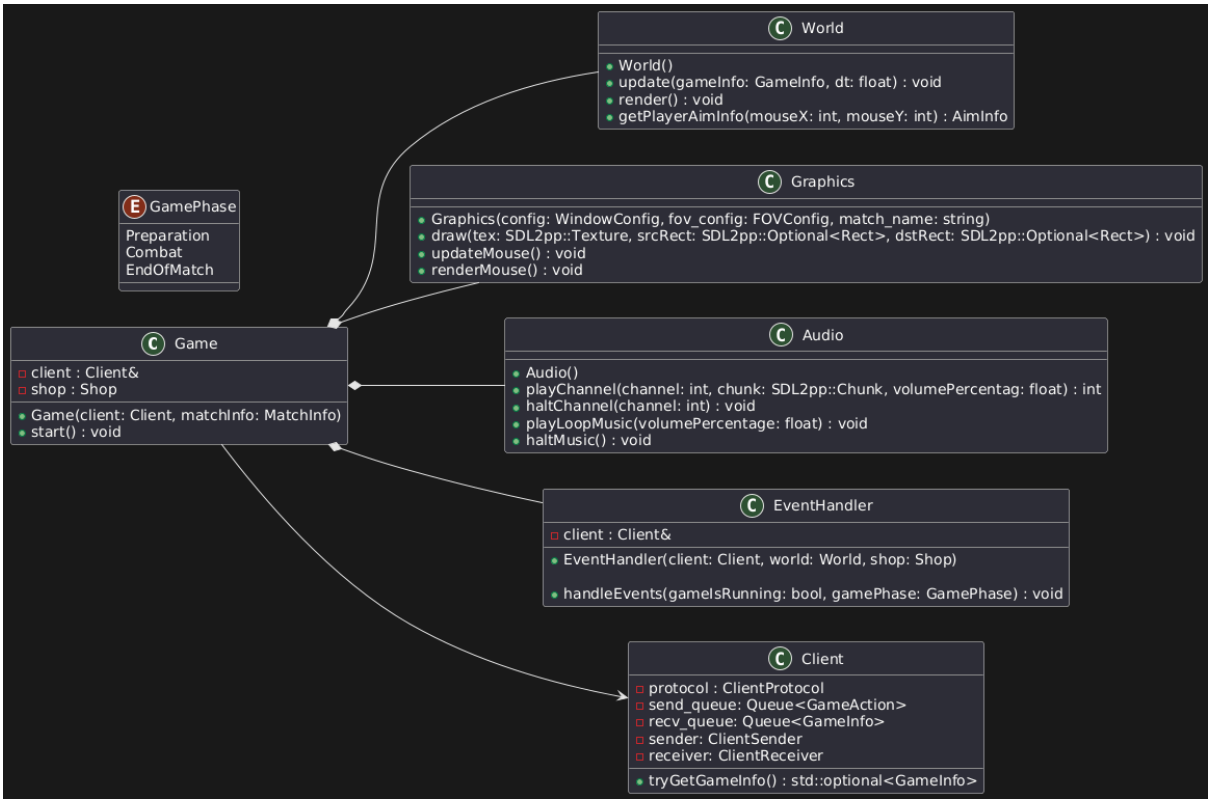
A su vez, tenemos la clase **Client** que representa al cliente en sí y funciona como un wrapper del protocolo del cliente, otorgando una interfaz amigable para poder enviar acciones de juego concretas al servidor.

🏗️ Detalles de implementación (GameMatchAppState)

En términos generales, las clases más relevantes del lado del cliente, al iniciar una partida, son las siguientes:

- **Game**: Clase que encapsula toda la lógica relativa a una partida. Ofrece una interfaz pública sencilla para iniciar la partida en base a la información de configuración que fue proporcionada al momento de crear una instancia con su constructor. En particular, al ejecutar Game::start() se inicia el gameloop del cliente.
- **World**: Modela el mundo del juego, encapsulando la lógica que opera sobre las entidades del mismo.
- **Graphics**: Encargado de la inicialización correcta del motor gráfico. Ofrece una API para realizar operaciones de renderizado u operaciones gráficas.
- **Audio**: Encargado de la inicialización correcta de los sistemas de audio. Ofrece una API para operar sobre los canales de reproducción de audio.
- **EventHandler**: Encargado de recibir e interpretar los eventos reconocidos por el motor gráfico (en nuestro caso SDL2), con el fin de manejarlos adecuadamente, indicando a la instancia de Client que acción se desea realizar.

Aquí podemos ver un diagrama de clases que muestra sus relaciones básicas y algunos métodos importantes:



Por otro lado, para modelar las entidades del juego decidimos implementar un ***híbrido*** entre el patrón Component y el diseño Entity Component System (ECS). Obteniendo lo mejor de ambos mundos. En todo momento estuvimos buscando lograr que nuestro código sea cache friendly y aproveche la data locality. De esta forma, para optimizar el uso de la

memoria y reducir el tiempo de búsqueda a la misma decidimos utilizar pools de componentes.

Particularmente, hemos decidido representar a las entidades como simples IDs (*uint32_t*) los cuales están asociados a ciertos componentes, dependiendo del tipo de entidad que sea. A su vez, definimos ciertos componentes que encapsulan un cierto estado específico de la entidad y contienen lógica que trabaja sobre dichos datos. En adición hemos implementado únicamente 2 sistemas: RenderSystem y AudioSystem; que encapsulan la lógica para renderizar las entidades y para reproducir los sonidos de efecto del juego, respectivamente.

Con esto en mente, las clases relevantes en cuanto a este punto son:

- **EntityManage**r: Encargada de crear una entidad de forma controlada. Le asigna un ID válido y disponible, delegando la asignación de componentes específicos al tipo de entidad a una instancia de la clase *EntityFactory*.
- **EntityFactory**: Encargada de asignar de forma adecuada los componentes específicos a un tipo de entidad.
- **ComponentPool**: Clase template que representa un pool de componentes y define la API para trabajar sobre el mismo. El tamaño de los pools es configurable al momento de su creación. Esto es útil ya que si se desea limitar la cantidad de entidades de un tipo en específico, entonces permite reservar una cantidad de memoria justa para lo que se requiere. Actualmente, si no se especifica un valor en su creación se asigna un valor de MAX_ENTITIES por default.
    - MAX_ENTITIES se puede encontrar en Entity.h
- **ComponentManager**: Clase encargada de administrar de forma controlada los pools de componentes. Ofrece una API con métodos template que permiten indicar específicamente con que componente trabajar.
- **ComponentUpdater**: Encargado de actualizar los datos de los componentes de cada entidad, según el snapshot del juego enviado desde el servidor. Dentro de su responsabilidad, está incluida la sincronización de las entidades del mundo que existen en el cliente, con las entidades que continúan activas en el servidor.

Aquí tenemos un diagrama de clases que muestra cómo se relacionan estas clases con World:

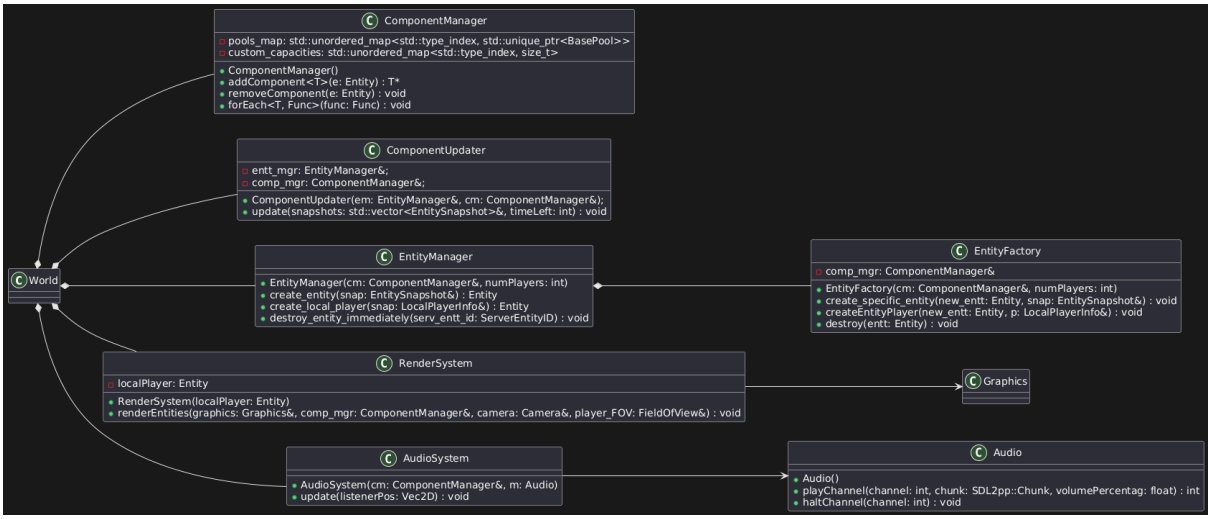
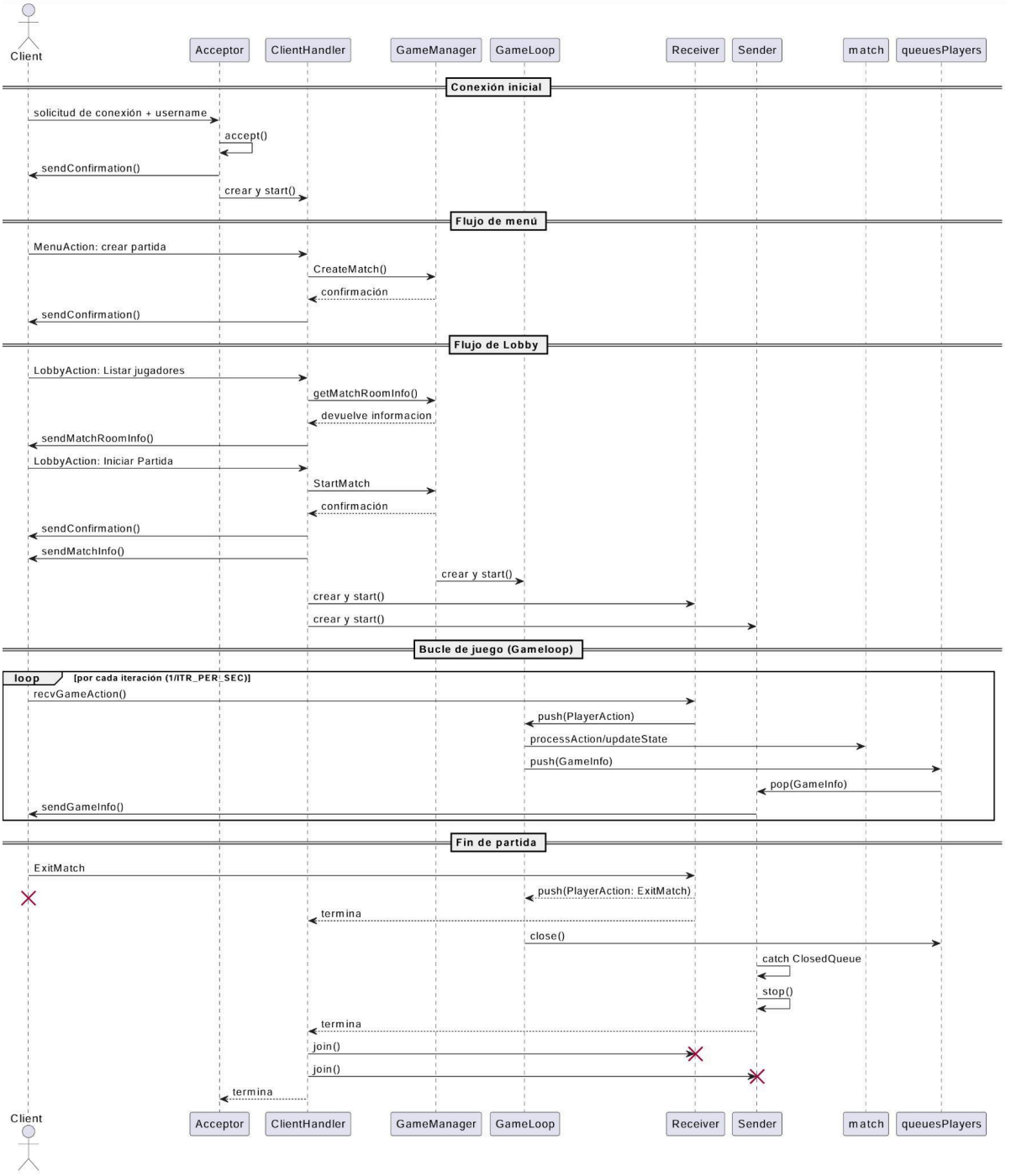


Diagrama de secuencia.

Para un cliente que crea una partida, la inicia, juega y se desconecta. (se puede exportar en pdf y se ve mejor)


Client
Acceptor
ClientHandler
GameManager
GameLoop
Receiver
Sender
match
queuesPlayers
Conexión inicial
solicitud de conexión + username
accept()
sendConfirmation()
crear y start()
Flujo de menú
MenuAction: crear partida
CreateMatch()
confirmación
sendConfirmation()
Flujo de Lobby
LobbyAction: Listar jugadores
getMatchRoomInfo()
devuelve informacion
sendMatchRoomInfo()
LobbyAction: Iniciar Partida
StartMatch
confirmación
sendConfirmation()
sendMatchInfo()
crear y start()
crear y start()
crear y start()
Bucle de juego (Gameloop)
loop
[por cada iteración (1/ITR_PER_SEC)]
recvGameAction()
push(PlayerAction)
processAction/updateState
push(GameInfo)
pop(GameInfo)
sendGameInfo()
Fin de partida
ExitMatch
push(PlayerAction: ExitMatch)
termina
close()
catch ClosedQueue
stop()
termina
join()
join()
termina
Client
Acceptor
ClientHandler
GameManager
GameLoop
Receiver
Sender
match
queuesPlayers

🤔Próximos pasos…

Es conveniente hacer una limpieza general de comentarios innecesarios. Esto no es crítico pero ayuda a hacer el código más legible.

Por otro lado, del lado del cliente sería positivo realizar los siguientes refactors:

- Crear una abstracción que encapsule la lógica relacionada al manejo del tiempo en el game loop (en Game::start()). Esto mejoraría la legibilidad, mantenibilidad y favorece a la separación de responsabilidades (recordar SRP). A su vez, se podría reutilizar en el gameloop del servidor, adaptando la API la abstracción de ser necesario. Un buen nombre para la abstracción puede ser “Timer”.
- Implementar la interpolación en el renderizado de las entidades. De esta manera, se suaviza el movimiento de las entidades en la pantalla, mejorando la UX.
- Implementar animaciones. Para ello se puede crear un AnimationComponent, el cual se encargue de actualizar el sprite que se debe mostrar de la textura que contiene el SpriteComponent de una entidad en particular.
- Crear un archivo de configuración para el cliente (parecido al del servidor, pero con otros parámetros). Por ejemplo, uno de los parámetros que pueden sacarse del archivo de configuración del servidor y colocarlo en el cliente es si se desea jugar en Fullscreen o no. Para ello hay que implementar un parseador de yaml, de manera análoga a como se hizo en el servidor, pero del lado del cliente.

Diseño del protocolo

En nuestro juego, toda la lógica del mismo se encuentra del lado del servidor. Esto significa que, para realizar cualquier acción, el cliente debe enviar un mensaje al servidor, esperar a que este lo interprete y que actualice el estado del juego en consecuencia.
Para habilitar esta comunicación, el programa establece una conexión cliente-servidor utilizando el protocolo TCP mediante **sockets**. Una vez establecida la conexión, tanto el cliente como el servidor cuentan con un conjunto de mensajes que pueden enviar y recibir. Estos mensajes están definidos en nuestro protocolo de aplicación, representado por las clases *ClientProtocol* y *ServerProtocol*.

Conexión inicial

El primer mensaje que envía el cliente es su nombre de usuario:

Envía: 0x01 <length:2B> <username>

- 0x01: Identificador del tipo de mensaje (1 byte)
- <length>: Tamaño del nombre de usuario en bytes, codificado en big endian (2 bytes)
- <username>: Nombre de usuario (buffer del string (cada caracter es un byte) con la cantidad de bytes indicada por <length>)

El servidor responde con un byte de confirmación:

- 0x06 → **OK**
- 0x15 → **FAIL**

Si la respuesta es OK, el cliente pasa al estado **InMenu**.

Todo el programa está dividido en fases o estados, según el momento en el que se encuentre el cliente. Estos estados son:

- ***Disconnected***
- ***InMenu***
- ***InLobby***
- ***InGame***

Por lo tanto el protocolo también separa sus mensajes para enviar una acción correspondiente al estado en el que se encuentre el cliente. Estas acciones se definieron como *MenuAction*, *LobbyAction* y *GameAction (ver estructuras en common/types.h)*.

Mensajes del Cliente

El cliente (Client) puede enviar los siguientes mensajes:

Acciones de menú (MenuActions)

- **Crear partida:**
  Envía: 0xA1 <length_match_name:2B> <match_name> <length_scenario_name:2B> <scenario_name>

  Recibe una confirmación:

  - Si recibe OK → el cliente cambia su estado a **InLobby**
  - Si recibe FAIL → el cliente permanece en estado **InMenu**

- **Unirse a una partida:**
  Envía: 0xA2 <length_match_name:2B> <match_name>

  Recibe una confirmación:

  - Si recibe OK → el cliente cambia su estado a **InLobby**
  - Si recibe FAIL → el cliente permanece en estado **InMenu**

- **Listar partidas:**
  Envía: 0xA3

  Espera recibir: 0xC1 <length:2B> <matches_list>
  Donde matches_list es un string con los nombres de las partidas separados por '\n'.

- **Listar escenarios:**
  Envía: 0xC4

  Espera recibir: 0xC4 <length:2B> <scenarios_list>
  Donde scenarios_list es un string con los nombres de escenarios separados por '\n'.

- **Salir:**
  Envía: 0xF0

  No espera confirmación y cambia el estado del cliente a **Disconnected**.

Acciones de Lobby (LobbyActions)

- **Iniciar Partida:**

  Envía: 0xB1

  Recibe una confirmación:

  - Si recibe OK → el cliente cambia su estado a **InGame** y recibe el MatchInfo
  - Si recibe FAIL → el cliente permanece en estado **InLobby**

- **Listar Jugadores en la sala:**

  Envía: 0xB2

  Recibe:

  0xC2 <players_count:2B>
  <length_username1:2B> <username1> <code_Team:1B> <bool_isHost:1B>
  <length_username2:2B> <username2> <code_Team:1B> <bool_isHost:1B>
  ...
  <bool_matchStarted:1B>

  - Si la partida ha comenzado (match started) el cliente cambia su estado a **InGame** y recibe el *MatchInfo* (ver más abajo).

  Los bytes correspondientes a los equipos (**Team**) son:
  - **Terrorist** → 0x01
  - **CounterTerrorist** → 0x02

  Los valores booleanos están representados así:
  - **true** → 0x01
  - **false** → 0x00

- **Abandonar Partida:**

  Envía: 0xF1

  Recibe una confirmación:

  - Si recibe OK → el cliente cambia su estado a InMenu
  - Si recibe FAIL → el cliente permanece en estado InLobby

Una vez que el cliente pasa del estado **InLobby** al estado **InGame**, inmediatamente el servidor le envía un mensaje con información necesaria para comenzar el juego. Esta información está encapsulada en una estructura llamada MatchInfo (*ver common/dtos/MatchInfo.h*) y se enviará una única vez a cada cliente.

En resumen, en esta estructura se incluye:

- nombre de la partida.
- configuración de tamaño de la ventana del juego.
- configuración del FOV del jugador.
- tilemap (matriz de números enteros, donde cada número representa el id de una imagen de un tile específico).
- número de jugadores al iniciar la partida.
- información del estado inicial del jugador local (diferente para cada cliente)
- información de la tienda (armas disponibles, precios, etc.).

El servidor envía y el cliente espera recibir el siguiente buffer:

```
0xC0 <length_match_name:2B> <match_name>

<window_width:2B> <window_heigth:2B> <window_flags:4B>

<bool_fov_is_active:1B> <screen_width:2B> <screen_heigth:2B>
<fov_radius_circle:2B> <fov_angle:4B> <fov_visibility:4B>
<fov_transparency:4B>

<size_buffer_tilemap:4B> <code_type_tilemap:1B> <row_count:4B>
<col_count:4B>
<id_tile_f0_c0:1B> <id_tile_f0_c1:1B> <id_tile_f0_c2:1B>...
<id_tile_fN_cN:1B>

<players_count:2B>

<size_buffer_local_player:2B> <player_entity_id:4B>  <code_Team:1B>
<code_PlayerSkin:1B> <code_PlayerState:1B>  <pos_x:4B> <pos_y:4B>
<angle_direction:4B> <code_TypeWeapon:1B> <code_Weapon:1B>
<player_health:2B> <player_money:2B> <ammo_weapon:2B>
<weapon_entity_id:4B>

<size_buffer_shop:2B> <weapon_count:2B> <code_Weapon:1B>
<price_weapon1:2B> <code_Weapon:1B> <price_weapon2:2B> ...
<ammo_type_count:2B>  <code_AmmoType:1B>  <price_ammo1:2B>
<code_AmmoType:1B>  <price_ammo2:2B> ...
```

(nota. Más abajo se especifican los códigos de bytes para cada elemento.)

Acciones de Juego (GameActions)

En esta etapa, los mensajes que envía el cliente no esperan respuesta ni confirmación. Simplemente recibe el estado del juego constantemente.

- **Comprar arma:**

  Envía: 0x02 <code_Weapon:1B>

  Los bytes correspondientes a cada arma (**Weapon**) son:
  - None → 0x00
  - Glock → 0x01
  - AK47 →0x02
  - M3 →0x03
  - AWP → 0x04
  - M3 →0x05
  - AWP → 0x06
- **Comprar munición:**

  Envía: 0x03 <code_AmmoType:1B> <ammo_count:2B>

  Los bytes correspondientes a los tipos de munición (**AmmoType**) son:
  - PrimaryAmmo → 0x01
  - SecondaryAmmo → 0x02
  - None → 0x03
- **Atacar / colocar bomba:**

  0x04 <direction_x:3B> <direction_y:3B>

  La dirección se representa con dos floats normalizados (con valores entre -1.0 y 1.0)
- **Moverse a una dirección:**

  Envía: 0x05 <direction_x:3B> <direction_y:3B>
- **Cambiar Arma:**

Envía: 0x06 <code_TypeWeapon:1B>

Los bytes correspondientes a los tipos de arma (**TypeWeapon**) son:

- Primary → 0x
- Secondary→ 0x
- Knife→ 0x
- Bomb→ 0x

- **Agarrar objeto (Pick Up):**

  Envía: 0x07

- **Rotar ángulo del jugador:**
  Envía: 0x08 <angle_direction:4B>

  El ángulo se representa con un float (4 bytes) (con valores entre 0.0 y 360.0)

- **Desactivar la bomba:**

  Envía: 0x09

- **Salir de la partida:**

  Envía: 0x10

  El cliente pasa al estado **Disconnected** y sale de la aplicación.

- **Recibir GameInfo (snapshot):**

  Además de poder enviar estos mensajes, cada cierta frecuencia (en nuestro caso 24FPS), el cliente espera recibir un buffer con la información del estado actual del juego, la cual utilizará la interfaz para poder mostrar en pantalla lo que corresponda. Esta información está encapsulada en un clase llamada **GameInfo** (ver *common/game_info/game_info.h*). A continuación detallamos qué información guarda y cómo se envía.

Mensajes del Servidor

Durante la etapa del juego, el servidor enviará constantemente (en cada frame) a cada cliente un buffer de un **GameInfo**.

En detalle, esta clase contiene los siguientes datos:

- ★ Fase actual del juego (preparación, combate o final de partida).
- ★ Tiempo restante (TimeLeft).
- ★ Información de la bomba:
  - id de entidad
  - estado de la bomba (equipada, plantada, explotada, etc.)
  - posición en el mundo
- ★ Información del jugador local:
  - id de entidad
  - equipo
  - skin actual
  - estado actual (inactivo, caminando, atacando, etc.)
  - posición en el mundo
  - ángulo (a dónde mira el jugador)
  - tipo de arma equipada (primaria, secundaria, etc)
  - arma específica equipada.
  - vida actual
  - dinero actual
  - municiones actuales (del arma equipada)
  - id del arma equipada
- ★ Información de los otros jugadores en la partida:
  - similar a la información anterior pero sin incluir la vida, el dinero y las municiones, y agregando el nombre de usuario para cada uno.
- ★ Información de las balas o proyectiles:
  - id de la entidad.
  - tipo de bala (según el arma).
  - posición.
  - dirección.
  - si está activada o no (si ya colisionó).
- ★ Información de las armas.
  - id de la entidad
  - arma específica (Glock, AK-47, AWP, etc.).
  - estado del arma (equipada, oculta o tirada).
  - municiones.
  - posición.
- ★ Información de las estadísticas.
  - Dos listas (una por cada Team), donde se incluye nombre de usuario, cantidad de kills, cantidad de muertes y dinero recolectado de cada miembro del equipo.

Por último, la clase cuenta con un método llamado toBytes() que genera el buffer del objeto (un vector de bytes), y un constructor a partir de ese buffer. De esta forma el protocolo simplemente debe invocar a ese método para poder enviar la información y puede reconstruir el objeto rápidamente usando ese constructor.

El buffer generado por el método toBytes() será lo que el servidor le envíe a cada cliente por la red. A continuación detallamos su contenido:

```
0xC3 <size_buffer:2B>

<code_GamePhase:1B>

<id_bomb:4B> <code_BombState:1B> <pos_x:4B> <pos_y:4B>

<time_left:4B>

<id_local_player:4B> <code_Team:1B> <code_Skin:1B> <code_State:1B>
<pos_x:4B> <pos_y:4B> <angle:4B> <code_TypeWeapon:1B> <code_Weapon:1B>
<health:2B> <money:2B> <ammo_weapon:2B> <id_weapon_equipped:4B>

<other_players_count:2B>
[<size_buffer_player1:2B> <id_player1:4B> <length_username1:2B>
<user_name1>
<code_Team:1B> <code_Skin:1B> <code_State:1B> <pos_x:4B> <pos_y:4B>
<angle:4B> <code_TypeWeapon:1B> <code_Weapon:1B> <id_weapon_equipped:4B>]
...

<bullets_count:2B>
[<id_bullet1:4B> <code_Weapon:1B> <pos_x:4B> <pos_y:4B> <direction_x:3B>
<direction_y:3B> <bool_isActive:1B>]
...

<weapons_count:2B>
[<id_weapon:4B> <code_Weapon:1B> <code_WeaponState:1B> <ammo_weapon:2B>
<pos_x:4B> <pos_y:4B>]
...

(si el juego se encuentra en la fase EndOfMatch, también se envía)

<size_buffer_stats:2B>
<terrorists_count:2B>
[<size_buffer_player_stat:2B> <length_username:2B> <username> <kills:4B>
<deaths:4B> <moneyEarned:4B>] ...
<counter_terrorists_count:2B>
[<size_buffer_player_stat:2B> <length_username:2B> <username> <kills:4B>
<deaths:4B> <moneyEarned:4B>] ...
```

Todos los códigos de los bytes mencionados y algunos que faltan mencionar están definidos en el archivo *common/constants_protocol.h*