FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico Final
Jazzjack Rabbit 2
Manual Técnico

[75.42] Taller De Programación I
Primer cuatrimestre de 2024

Apellido/s	Nombres	Padrón	Correo electrónico
Brizuela	Valentin	108071	vabrizuela@fi.uba.ar
Cabrera	Isaias A.	108885	isaiascabrera20@gmail.com
Mokorel	Pablo A.	103029	pmokorel@fi.uba.ar
Perez Esnaola	Lucas	107990	lpereze@fi.uba.ar

Índice

1. Introducción

Este documento proporciona una descripción de la arquitectura del proyecto, incluidos los diagramas de clase y librerias utilizadas, así como el formato de los archivos y el protocolo de comunicación. Está dirigido a desarrolladores que deseen comprender, mantener y ampliar el código fuente del proyecto.

2. Requerimientos del Sistema

El proyecto usa varias librerias para poder ejecutarse, así como también un sistema operativos basado en alguna distribución GNU/Linux. A continuación se mencionaran las librerias necesarias para poder compilar y ejecutar el juego.

2.1. SDL 2

Esta librería es el motor gráfico del juego, se encarga de todo lo relacionado con la renderización, musicalización e interacción con el usuario (Cliente), determinando cada acción del usuario para luego mostrarla en pantalla.

A su vez esta libreríia cuenta con librerias adicionales, que complementan y forman parte del motor gráfico. Estas librerias son de SDL2 Mixer y SDL2 TTF, encargandose de la reproducción de la música del juego, y la visualización de fuentes respectivamente.

Figura 1: SDL

2.2. Qt5

Qt5, es una librería para trabajar con interfaz gráfica, la ventaja de esta librería es que utiliza el lenguaje de programación C++ de forma nativa, lo que cual fue una gran ventaja para el proyecto.

Cuenta con diversos módulos. Dentro de estos, para el proyecto se requirieron el módulo de Widgets y de MessageBox, siendo el primero para la creación de la interfaz, mediante el uso de buttons, labels, lineEdits, etc, y el segundo para los carteles de advertencia.
Cabe resaltar que Qt se empleó también para la creación del editor de mapas del proyecto, dandonós un uso más a esta librería.

Figura 2: Qt

2.3. YAML-cpp

El nombre de la libreria nos da una pista sobre el uso de la misma. Su funcionalidad radica en ser un parser de archivos YAML, el cual se usa unicamente para la carga de configuración del juego, ya que para la carga del mapa se utilizaran archivos CSV con funciones build in de c++.

3. Descripción General

Inicialmente, planteamos que cada cliente pudiera transmitir sus movimientos al servidor, para que este los retransmita al resto de los usuarios. Para lograr esto, adoptamos un modelo autoritario del lado del servidor. En este modelo, el servidor es el encargado de simular el juego, mientras que los usuarios solo pueden comunicarle sus intenciones. El servidor ejecuta la simulación y genera las respuestas apropiadas, asegurando que todos los jugadores permanezcan sincronizados.

3.1. Aceptador y Creador de Juegos

El servidor incluye un componente llamado *Acceptor* que se encarga de aceptar nuevas conexiones de clientes. Este componente mantiene una lista de clientes activos y se asegura de eliminar aquellos que se desconectan. Cuando un nuevo cliente se conecta, el *Acceptor* crea un *ClientHandler* para gestionar la comunicación con ese cliente, asignándole un identificador único y notificando a los gestores de juegos sobre el nuevo cliente.

El servidor también incluye un *GamesManager* que gestiona la creación, unión y eliminación de juegos. El *GamesManager* carga la configuración del juego desde un archivo YAML, maneja los mapas del juego y permite a los clientes unirse a juegos existentes o crear nuevos. Los comandos de los clientes son procesados en una cola y ejecutados por el *GamesManager*, que también maneja los eventos de juego, como la activación de trucos y la actualización de los estados de los juegos.

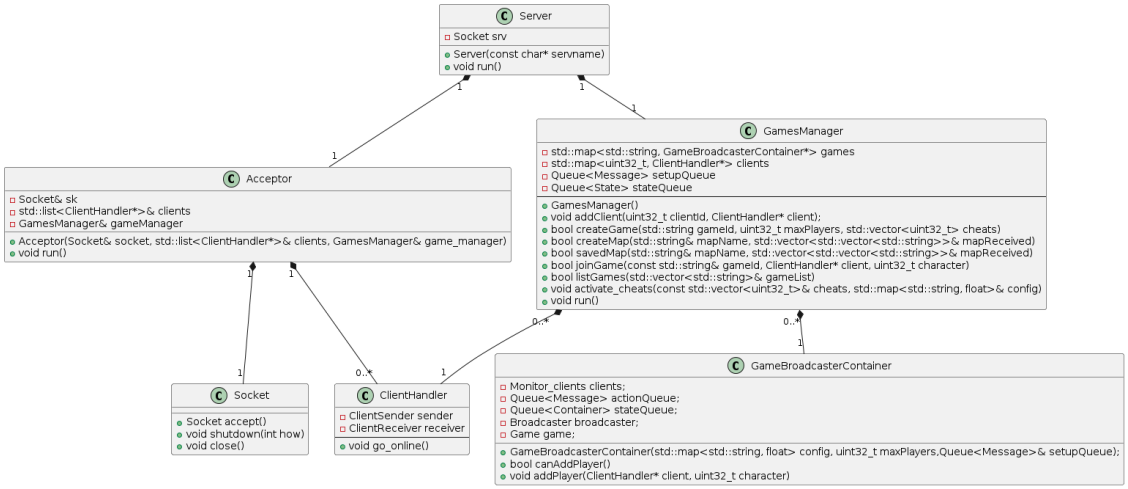

Figura 3: Diagrama de clases basico del servidor

3.2. Flujo de Mensajes entre Clientes y Servidor

Los clientes, a través de sus movimientos, envían un mensaje al servidor utilizando un protocolo específico a través de un socket. Un manejador (receiver) del lado del servidor recibe estos mensajes y encola los comandos correspondientes, simulando las acciones esperadas por los usuarios. Estos comandos son extraídos de la cola de manera FIFO (First-In-First-Out) por el motor del juego (Engine), el cual es responsable de ejecutar los comandos y mediante un broadcaster enviar las respuestas apropiadas a los usuarios mediante otro manejador (sender).

El motor del juego también se encarga de simular el paso del tiempo para las partes del modelo que lo requieren. Las notificaciones generadas por el motor del juego son recibidas por los usuarios, quienes actualizan su representación visual del mapa en función de estas notificaciones. Esto puede incluir mover sprites de lugar o reproducir sonidos en respuesta a una acción.

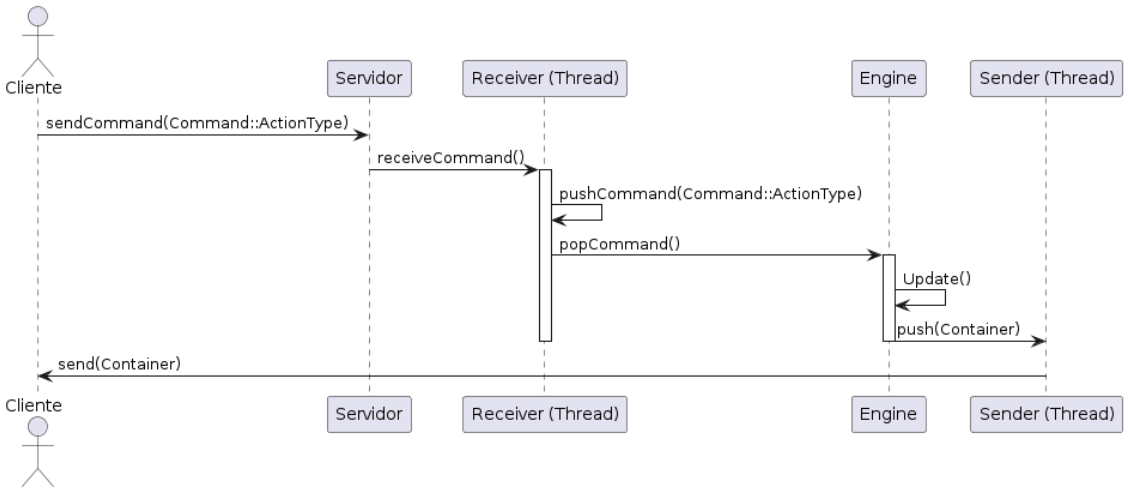

Figura 4: Diagrama de secuencia basico del flujo de comandos de un cliente

**Nota: A engine nos referimos a toda la clase que contiene el juego y broadcaster donde ejecuta los comando un cliente.

4. Descripción de Ejecutables

En esta sección hablaremos de los ejecutalbes (Cliente-Servidor-Editor) para la resolución del proyecto, algunas de sus características y detalles de implementación.

4.1. Cliente

El ejecutable *Cliente* es responsable de manejar la conexión y comunicación con el servidor de juego. Algunas de sus principales funciones incluyen:

- Inicialización de la interfaz gráfica de usuario (GUI) para la configuración inicial del juego, como la selección de nombre de usuario y la conexión al servidor.
- Gestión de la comunicación con el servidor mediante el uso de la clase Client para enviar y recibir mensajes de juego.
- Interacción con la clase MultiplayerMenu para mostrar y manejar la lista de juegos disponibles, así como para crear y unirse a partidas.
- Gestión de la inicialización y cierre de los subsistemas de audio y de la biblioteca SDL para la reproducción de sonidos.

El *Cliente* utiliza eventos de la GUI y señales de Qt para manejar la interacción del usuario y la actualización de la interfaz de usuario en función de la respuesta del servidor. Además, controla el ciclo de vida del juego en modo multijugador, permitiendo al usuario unirse y crear partidas, así como jugar una vez que se haya unido a una partida existente.

Por último, la clase Cliente también maneja posibles excepciones que puedan ocurrir durante la ejecución del juego, mostrando mensajes de error adecuados en caso de fallos.

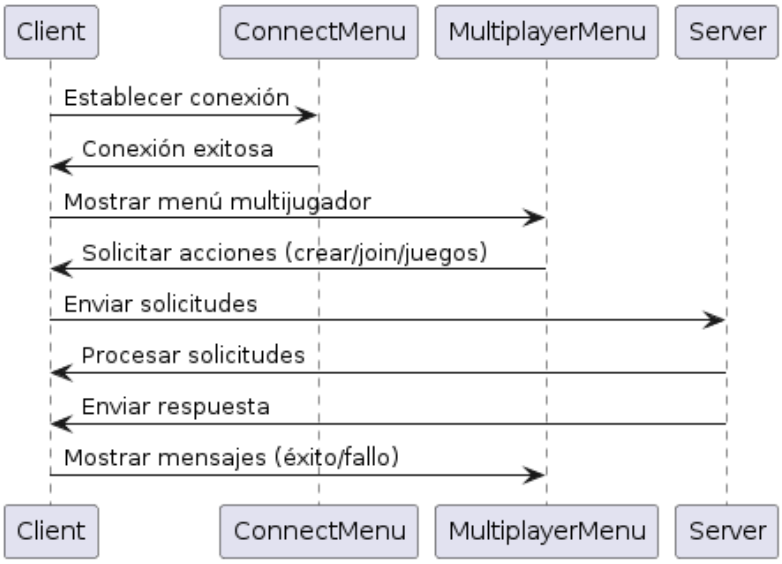

Figura 5: Diagrama de secuencia básico del flujo de comunicación entre el cliente y la UI

4.1.1. Renderizado

El *Cliente* utiliza la biblioteca SDL para renderizar gráficos en la pantalla. El proceso de renderizado se realiza en el bucle principal del juego, donde se actualiza la pantalla en cada iteración. El renderizado incluye la representación de los elementos del juego, como el mapa, los personajes, los objetos y la interfaz de usuario.

El renderizado se realiza en función de la posición y el estado de los elementos del juego, como la posición de los personajes, la posición de la cámara y el estado de los objetos en la pantalla. Además, el renderizado también incluye efectos visuales, como animaciones y transiciones para mejorar la experiencia visual del juego.

4.2. Servidor

El ejecutable del *Servidor* es responsable de gestionar la conexión y comunicación con los clientes del juego. Algunas de sus funciones principales incluyen:

- Inicialización y gestión de la comunicación con los clientes mediante el uso de la clase Acceptor para aceptar nuevas conexiones y la clase GamesManager para gestionar los juegos y clientes conectados.
- Creación y mantenimiento de juegos a través de la clase GamesManager, permitiendo a los clientes unirse a partidas existentes o crear nuevas partidas.

4.2.1. Gestión de Clientes

El servidor acepta nuevas conexiones de clientes a través de la clase *Acceptor*.

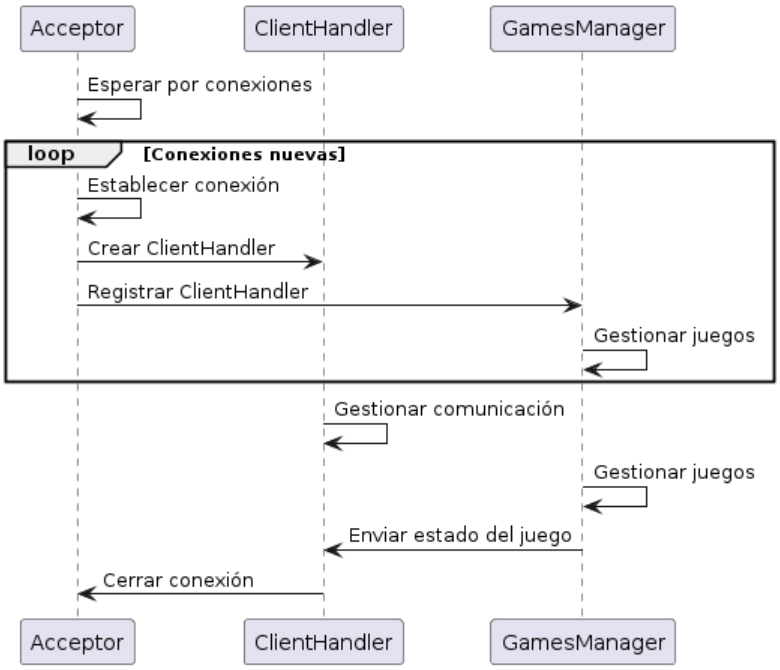

Figura 6: Diagrama de Secuencia de nueva conexiones

**Nota: Por cada cliente nuevo se lanzaran dos threads (sender y receiver).

4.2.2. Gestión de Juegos

El servidor gestiona los juegos a través de la clase *GamesManager*.

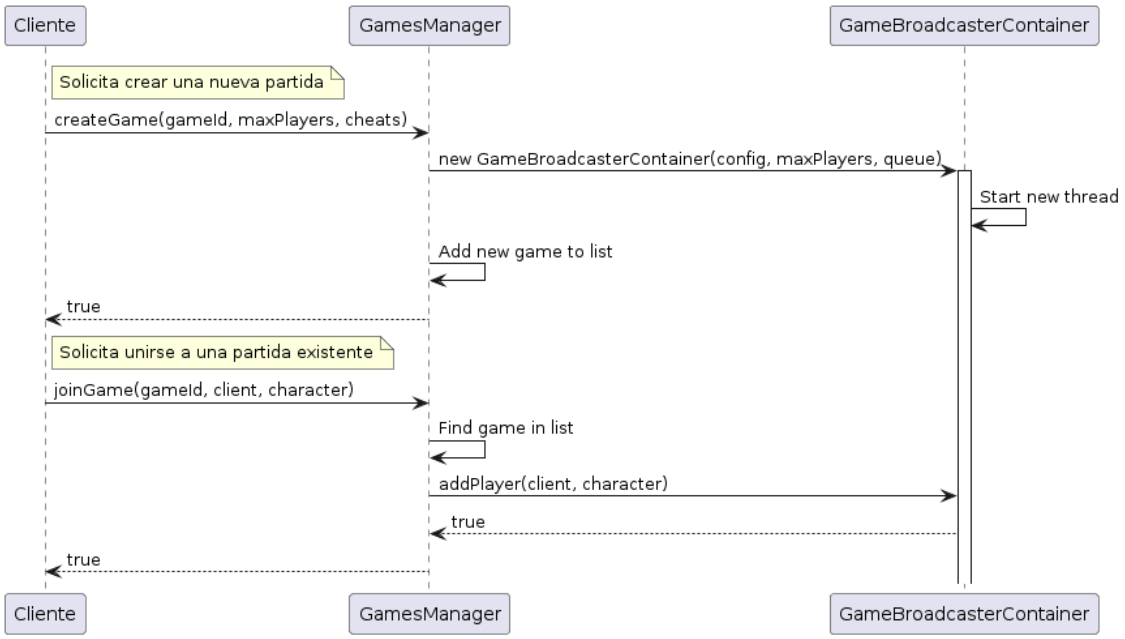

Figura 7: Diagrama de Secuencia para la creacion y conexión a un juego.

**Nota: Cada GameBroadcasterContainer nuevo lanzara dos threads ,game y broadcaster.

4.2.3. Modelo de Juego

El modelo de juego se gestiona principalmente a través de la clase *Game*, que maneja la lógica del juego, incluyendo la gestión de jugadores, enemigos y objetos del juego. Los jugadores pueden realizar acciones como moverse, saltar y disparar, y estas acciones se procesan y actualizan en tiempo real durante el ciclo del juego.

- La clase *Game* es responsable de agregar jugadores y actualizar el estado del juego.
- Las acciones de los jugadores se reciben y procesan desde una cola de acciones (*ActionQueue*).
- El estado del juego se actualiza en tiempo real y se envía a través de una cola de estado (*StateQueue*) al *Broadcaster* para su transmisión a los clientes conectados.
- El juego se ejecuta en un ciclo continuo mientras esté activo, procesando acciones, actualizando el estado y manejando colisiones y eventos del juego.

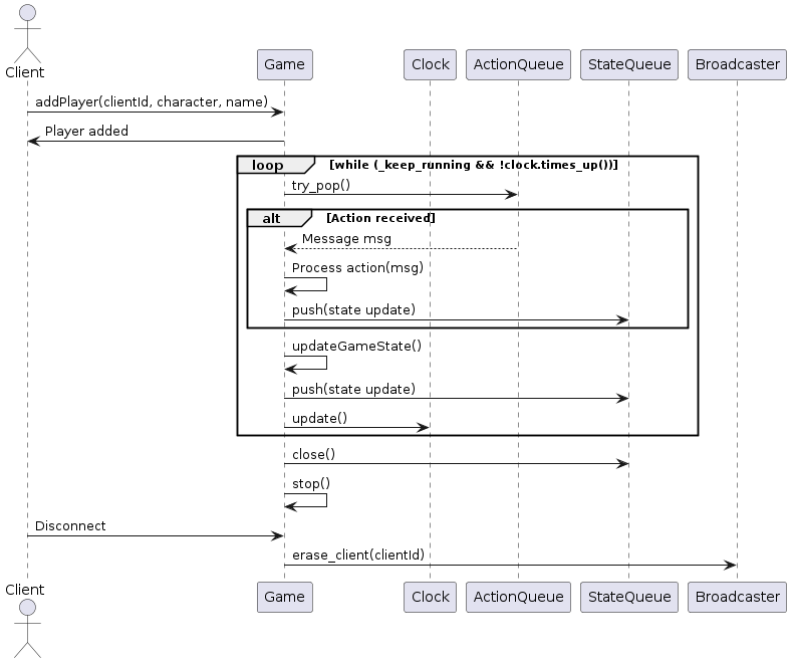

Figura 8: Diagrama de Secuencia para la gestión del juego.

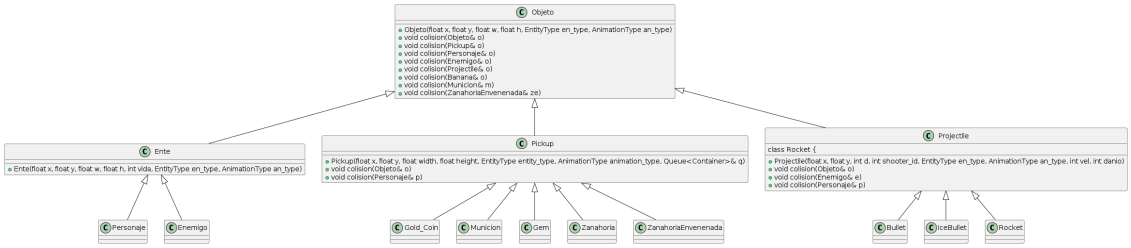

Figura 9: Diagrama de Clases objetos del juego.

Nota: Como ultima actualizacion al proyecto, los nombres de las clases 'Objecto' y 'Ente', se cambiaron a 'Object' y 'Entity'

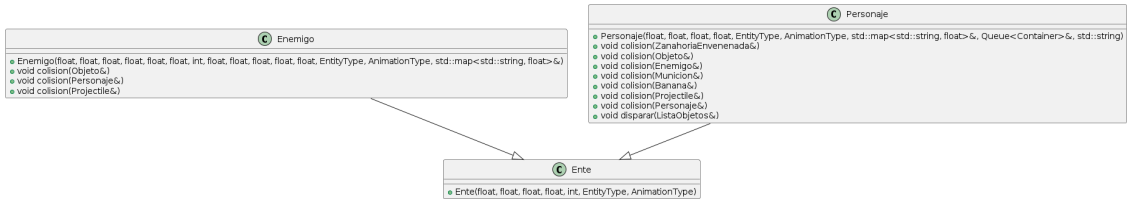

Figura 10: Diagrama de Clases personajes y enemigos del juego.

Nota: Como ultima actualizacion al proyecto, los nombres de las clases 'Personaje', 'Enemigo' y 'Ente', se cambiaron a 'Character', 'Enemy' y 'Entity'

4.2.4. Broadcaster

El *Broadcaster* es responsable de enviar el estado del juego a todos los clientes conectados. Utiliza una cola para recibir actualizaciones de estado del juego y las distribuye a los clientes. Además, gestiona la eliminación de clientes cuando se desconectan.

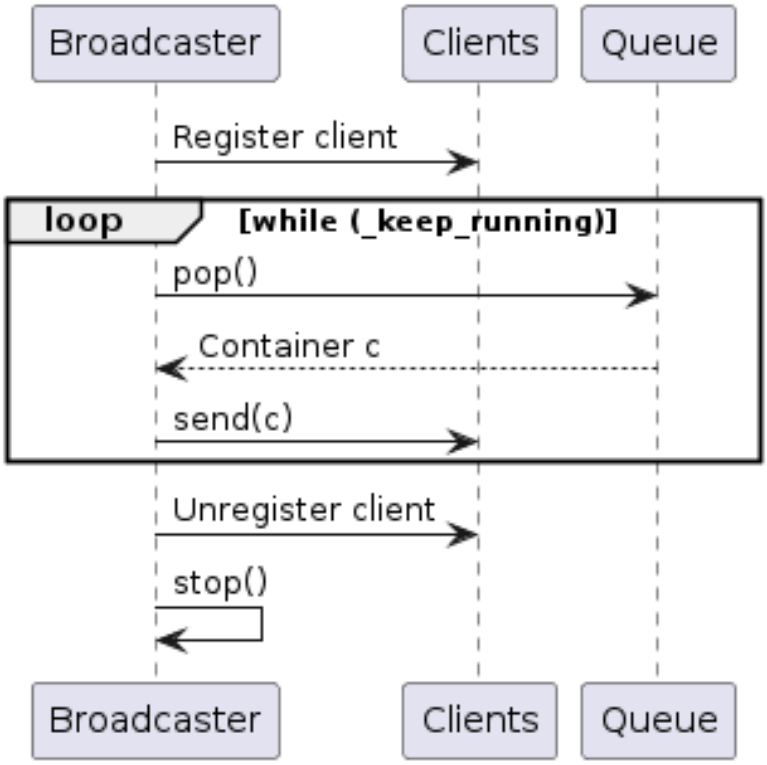

Figura 11: Diagrama de Secuencia del Broadcaster

4.3. Editor de Mapas

El ejecutable del *Editor de Mapas* es responsable de permitir al usuario crear y modificar mapas para el juego. Algunas de sus funciones principales incluyen:

- Interfaz gráfica de usuario (GUI) para la creación y edición de mapas, incluida la selección de elementos del mapa y la disposición en la cuadrícula del mapa.
- Gestión de la comunicación con el servidor para enviar los mapas creados o modificados.
- Renderizado de la paleta de assets para que el usuario pueda seleccionar los elementos del mapa.
- Guardado de los mapas creados en archivos CSV, incluyendo la información de los tiles y los spawns de jugadores y enemigos.

El *Editor de Mapas* utiliza la biblioteca SDL para renderizar la paleta de assets y los mapas en la pantalla. Permite al usuario seleccionar los tiles de la paleta y colocarlos en la cuadrícula del mapa, además de gestionar los spawns de jugadores y enemigos.

4.3.1. Creación de un nuevo mapa

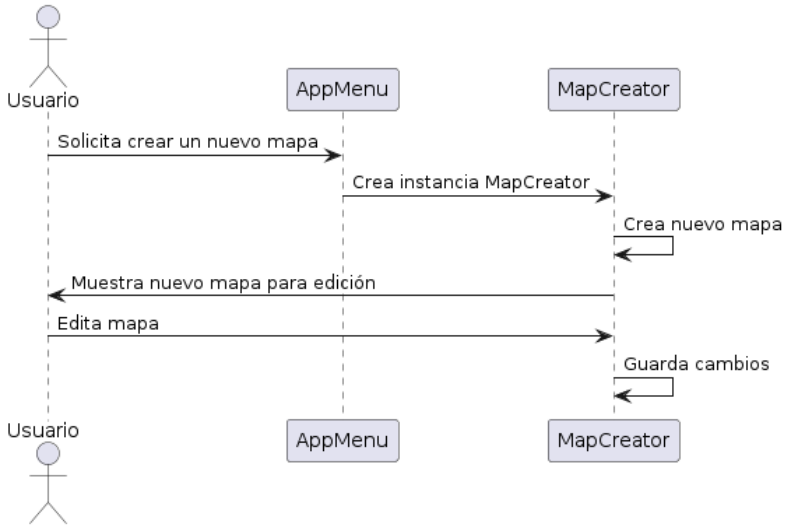

Figura 12: Creación de un nuevo mapa

4.3.2. Modificación de un mapa existente

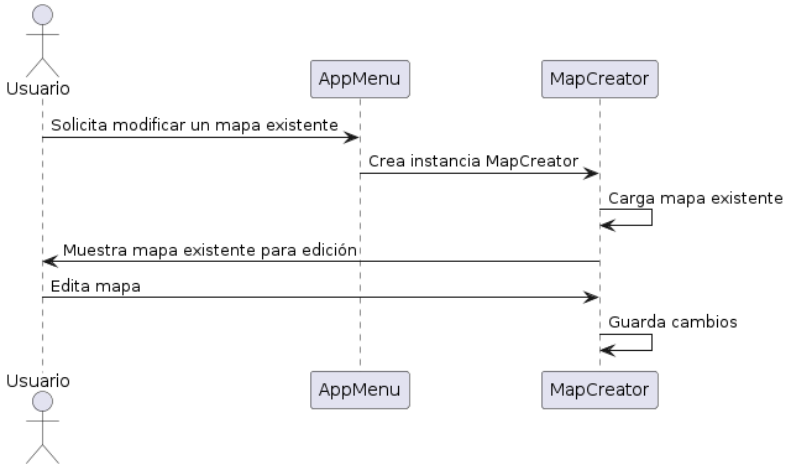

Figura 13: Modificación de un mapa existente

5. Descripción de Archivos y Protocolos

5.1. Archivos de configuracón del juego

Se emplearon archivos tipo YAML para la configuración del juego. Este archivo de configuración define los parámetros de juego y características de los personajes y objetos en el juego. A continuación, se describen los campos y sus propósitos:

5.1.1. Juego

- **game_time**: Duración del juego en segundos.

- **gravity**: Fuerza de gravedad aplicada en el juego.

5.1.2. Personaje

- **player**: Configuración del personaje principal.
  - **life**: Vida inicial del personaje.
  - **special_attack_dmg**: Daño del ataque especial.
  - **speed**: Velocidad normal de movimiento.
  - **run_speed**: Velocidad de carrera.
  - **jump**: Fuerza del salto.

5.1.3. Enemigos

- **ghost**: Configuración del enemigo tipo fantasma.
  - **life**: Vida del fantasma.
  - **damage**: Daño que inflige.
  - **speed**: Velocidad de movimiento.
  - **prob_carrot**, **prob_ammo**, **prob_goldcoin**, **prob_rocket**: Probabilidades de soltar zanahorias, munición (bullet y rocket) y monedas de oro al morir.
- **bat**: Configuración del enemigo tipo murciélago.
  - **life**, **damage**, **speed**, **prob_carrot**, **prob_ammo**, **prob_goldcoin**, **prob_rocket**: Similares a las del fantasma, pero con diferentes valores.
- **monkey**: Configuración del enemigo tipo mono.
  - **life**, **damage**, **speed**, **prob_carrot**, **prob_ammo**, **prob_goldcoin**, **prob_rocket**: Similares a las del fantasma, pero con diferentes valores.

5.1.4. Pickups

- **goldcoin**: Configuración de las monedas de oro.
  - **add_score**: Puntos añadidos al recoger una moneda.
- **gem**: Configuración de las gemas.
  - **add_score**: Puntos añadidos al recoger una gema.
- **carrot**: Configuración de las zanahorias.
  - **add_life**: Vida añadida al recoger una zanahoria.
- **ammo**: Configuración del pickup de munición tipo bullet.
  - **add_ammo**: Cantidad de munición añadida al recogerla.
- **rocket_pickup**: Configuración del pickup de municion tipo rocket.
  - **add_ammo**: Cantidad de municion añadida al recogerla.
- **ice_bullet_pickup**: Configuración del pickup de municion tipo ice bullet.
  - **add_ammo**: Cantidad de municion añadida al recogerla.

5.1.5. Tipos de municion

- **bullet**: Configuración de las balas.
  - **speed**: Velocidad de movimiento de la bala.
  - **damage**: Daño infligido por la bala.
  - **fire_rate**: Velocidad de disparo de la bala.
  - **initial_ammo**: Cantidad de municion inicial.
- **rocket**: Configuración de los cohetes.
  - **speed**, **damage**, **fire_rate**, **initial_ammo**: Similares a las de las balas, pero con diferentes valores.
- **ice_bullet**: Configuración de las balas de hielo.
  - **speed**, **damage**, **fire_rate**, **initial_ammo**: Similares a las de las balas, pero con diferentes valores.
  - **frozen_time**: Tiempo de congelamiento.

5.1.6. Municion de enemigos

- **banana**: Configuración de las bananas lanzadas por el enemigo Monkey.
  - **speed**: Velocidad de movimiento de la bala.
  - **damage**: Daño infligido por la bala.

5.2. Protocolos

El protocolo de comunicación está basado en el envío y recepción de mensajes entre el cliente y el servidor. Los mensajes pueden ser de dos tipos principales: SETUP y COMMAND. Cada tipo de mensaje tiene subtipos específicos que determinan la acción a realizar.

5.2.1. Protocolo base

Este protocolo contiene la implementación del protocolo de comunicación entre el cliente y el servidor. Esta clase base define métodos para enviar y recibir datos a través de un socket que podrán ser utilizados tanto del lado del cliente como del lado del servidor. Se describen los métodos principales:

- `sendUChar`, `receiveUChar`: Envío y recepción de caracteres/bits.
- `send16`, `receive16`: Envío y recepción de enteros de 16 bits.
- `send32`, `receiveUInt32`: Envío y recepción de enteros de 32 bits.
- `sendString`, `receiveString`: Envío y recepción de cadenas de texto.
- `sendBool`, `receiveBool`: Envío y recepción de valores booleanos.
- `sendVectorString`, `receiveVectorString`: Envío y recepción de vectores de cadenas de texto.
- `sendMap`, `receiveMap`: Envío y recepción de mapas bidimensionales de cadenas de texto.
- `sendVectorUint32`, `receiveVectorUint32`: Envío y recepción de vectores de enteros de 32 bits.

5.2.2. Server Protocol

La clase ServerProtocol hereda de Protocol (Protocolo base) y añade métodos específicos para enviar y recibir contenedores de datos:

- **Envío de Datos**
  - `send_container`: Envía un contenedor de datos, dependiendo de su tipo (SETUP, GAME, SOUND).
  - `send_setup_container`, `send_game_container`, `send_sound_container`: Métodos específicos para enviar contenedores de configuración, de juego y de sonido, respectivamente.
- **Recepción de Datos**
  - `receive_message`: Recibe un mensaje, identificando su tipo (SETUP, COMMAND).
  - `receive_setup_message`: Recibe un mensaje de configuración, delegando a métodos específicos según el tipo de acción de configuración.
  - `receive_command_message`: Recibe un mensaje de comando.

5.2.3. Client Protocol

La clase ClientProtocol hereda de Protocol (Protocolo base) y añade métodos específicos para enviar y recibir contenedores de datos:

- **Envío de Datos**
  - `send_message`: Envía un mensaje, dependiendo de su tipo (SETUP, COMMAND).
  - `send_command`: Envía un comando.
  - `send_setup`: Envía una configuración, delegando a métodos específicos según el tipo de acción de configuración.
- **Recepción de Datos**
  - `receive_container`: Recibe un contenedor de datos, identificando su tipo (SETUP, GAME, SOUND).
  - `receive_setup`: Recibe un mensaje de configuración, delegando a métodos específicos según el tipo de acción de configuración.

5.2.4. Tipo de Container

Se definen tres tipos de contenedores de datos en el protocolo:

- **SetupContainer**: Contiene información relacionada con la configuración del juego, como el tipo de configuración, el ID del juego, el número máximo de jugadores, trucos disponibles, etc.
- **GameContainer**: Contiene información sobre el estado del juego, como el código del mensaje, ID del jugador, posición, dirección, tipo de animación, tipo de entidad, salud, munición, puntuación, nombre del jugador, etc.
- **SoundContainer**: Contiene información sobre los sonidos del juego, como el tipo de entidad asociada al sonido, el tipo de sonido y el ID del sonido.

5.2.5. Tipos de Messages

Se definen dos tipos principales de mensajes en el protocolo:

- **Command**: Representa un comando enviado por el cliente al servidor para controlar el juego, como moverse, saltar, disparar, etc. El mensaje incluye el tipo de acción y el ID del cliente.
- **Setup**: Representa un mensaje de configuración enviado por el cliente al servidor para configurar el juego, como unirse a un juego, crear un juego, obtener la lista de juegos, establecer un nombre, etc. El mensaje incluye el tipo de acción, el ID del cliente, el ID del juego, el número máximo de jugadores, trucos disponibles, etc.

5.2.6. Common Commands

En el archivo `commands.h` se definen dos estructuras para representar comandos comunes en el protocolo de comunicación:

- **Command**: Define los posibles comandos de acción que un cliente puede enviar al servidor, como moverse, saltar, disparar, etc. Cada comando tiene un identificador único y opcionalmente puede incluir un ID de cliente para identificar al jugador que envía el comando.
  - **NONE (0x00)**: No se realiza ninguna acción.
  - **UP (0x01)**: Mover hacia arriba.
  - **DOWN (0x02)**: Mover hacia abajo.
  - **LEFT (0x03)**: Mover hacia la izquierda.
  - **RIGHT (0x04)**: Mover hacia la derecha.
  - **RUN (0x05)**: Correr.
  - **RUNFAST (0x06)**: Correr más rápido.
  - **JUMP (0x07)**: Saltar.
  - **FIRE (0x08)**: Disparar.
  - **STOPLEFT (0x09)**: Detener movimiento hacia la izquierda.
  - **STOPRIGHT (0x10)**: Detener movimiento hacia la derecha.
  - **STOPFIRE (0x11)**: Detener disparo.
  - **SPECIAL (0x12)**: Ejecutar una acción especial.
  - **QUIT (0x13)**: Salir del juego.
- **Setup**: Define los posibles comandos de configuración que un cliente puede enviar al servidor, como unirse a un juego, crear un juego, obtener la lista de juegos, etc. Cada comando de configuración tiene un identificador único y opcionalmente puede incluir un ID de cliente, un ID de juego, el número máximo de jugadores, etc.
  - **JOIN_GAME (0x31)**: Unirse a una partida.
  - **CREATE_GAME (0x32)**: Crear una nueva partida.
  - **GET_GAME_LIST (0x33)**: Obtener la lista de partidas disponibles.
  - **CLIENT_ID (0x34)**: Identificador de cliente.
  - **CREATE_MAP (0x35)**: Crear un nuevo mapa.
  - **SET_NAME (0x37)**: Establecer nombre de usuario.

Los identificadores de los comandos y comandos de configuración se especifican como valores hexadecimales para distinguirlos de manera única.