Duck Game

Documentación Técnica

Integrantes del Grupo:

Julio Mateo Fernandez (107491)
Lucía Belén Napoli (101562)
Leticia Isabel Aab (106053)
Máximo Palopoli (108755)

Fecha: 3 de Diciembre de 2024

1. Arquitectura del sistema

Las tecnologías utilizadas en el desarrollo son:

- **C++**: Para la lógica general.
- **SDL2**: Para la visualización gráfica.

2. Servidor

2.1. Funcionamieno inicial

El servidor inicia levantando un hilo aceptador, encargado de recibir nuevas conexiones a través del socket del servidor. Este aceptador realiza las siguientes acciones:

- Crea un hilo de `lobby`, que gestiona la distribución de las partidas y los jugadores.
- Genera instancias de `lobby_player` para cada jugador que se conecta, añadiéndolos al `lobby`.

Cada `lobby_player` tiene un hilo interno llamado `lobby_receiver`, que:

- Recibe comandos del cliente.
- Encola los comandos en la `queue` del `lobby`.

2.2. Lobby

El hilo del `lobby` contiene una `queue` de comandos, bloqueante hasta recibir instrucciones desde el cliente. Los comandos posibles son:

- Crear una partida.
- Unirse a una partida.
- Iniciar una partida.

Cuando se inicia una partida, el `lobby`:

- Notifica a los jugadores conectados a la partida.
- Mata los hilos asociados a esos jugadores.

- Devuelve las instancias de la clase `player`, que contienen dos hilos por jugador:
  - **Sender**: Para enviar mensajes del servidor al cliente.
  - **Receiver**: Para recibir comandos del cliente al servidor.

2.3. Inicio de partida

Al comenzar una partida:

- Se genera una instancia de `match`.
- `Match` crea un hilo para el `game_loop`, que controla toda la interacción entre los jugadores durante la partida.
- Cada `player` tiene asociada una `message_queue`, donde se encolan mensajes que salen del servidor hacia el cliente.
- Los comandos recibidos por el `receiver` del jugador se encolan en la `game_queue`, que es la principal del `game_loop`.

2.4. Game loop

La `game_queue` está protegida por un monitor que:

- Garantiza la integridad y el orden de los comandos.
- Evita *race conditions*.

Los comandos encolados son ejecutados en lotes de 10, y cada uno está representado por la superclase `Executable`. Las acciones posibles incluyen por ejemplo:

- Iniciar o detener el movimiento de un pato.
- Tomar un objeto.
- Disparar.

2.5. Componentes destacados

2.5.1. Matriz de colisiones (`game_map`)

Es una matriz que representa las colisiones en el juego. Los elementos con `hitbox` (como patos, plataformas, balas y cajas) actualizan sus posiciones dentro de la matriz para detectar interacciones. Los eventos de colisión son:

- Notificados a las clases correspondientes.
- Enviados a la clase principal (`Game.cpp`), que transmite la información a los clientes a través de las `queues` de los jugadores.

2.5.2. Items

Los objetos del juego (cascos, armaduras, armas y lanzables) heredan de la superclase `Item`. Esta unificación permite:

- Interacción consistente con los objetos, como recogerlos, equiparlos o utilizarlos.

2.6. Finalización de partida

Al terminar una partida:

- Todos los jugadores reciben un mensaje indicando el final de la partida.
- `Game` notifica a `Match` sobre el fin de la partida.
- El `lobby` realiza una verificación periódica (*polling*) para detectar partidas finalizadas y liberar los recursos correspondientes.

2.7. Diagrama de secuencia

En pos de entender mejor el flujo de como funciona el procesamiendo de los comandos recibidos por el server compartimos un diagrama de secuencia que representa que es lo que pasa cuando queremos que un pato salte.

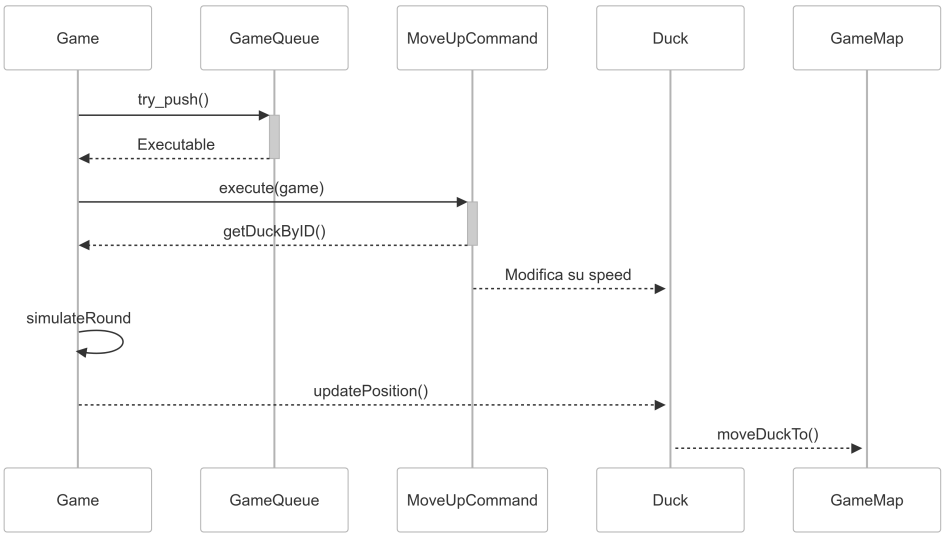

Figura 1: Diagrama secuencia del servidor.

2.8. Diagrama de clases

A continuación brindamos un diagrama de clases de la estructura del servidor, para analizar teniendo en cuenta todo lo definido anteriormente.

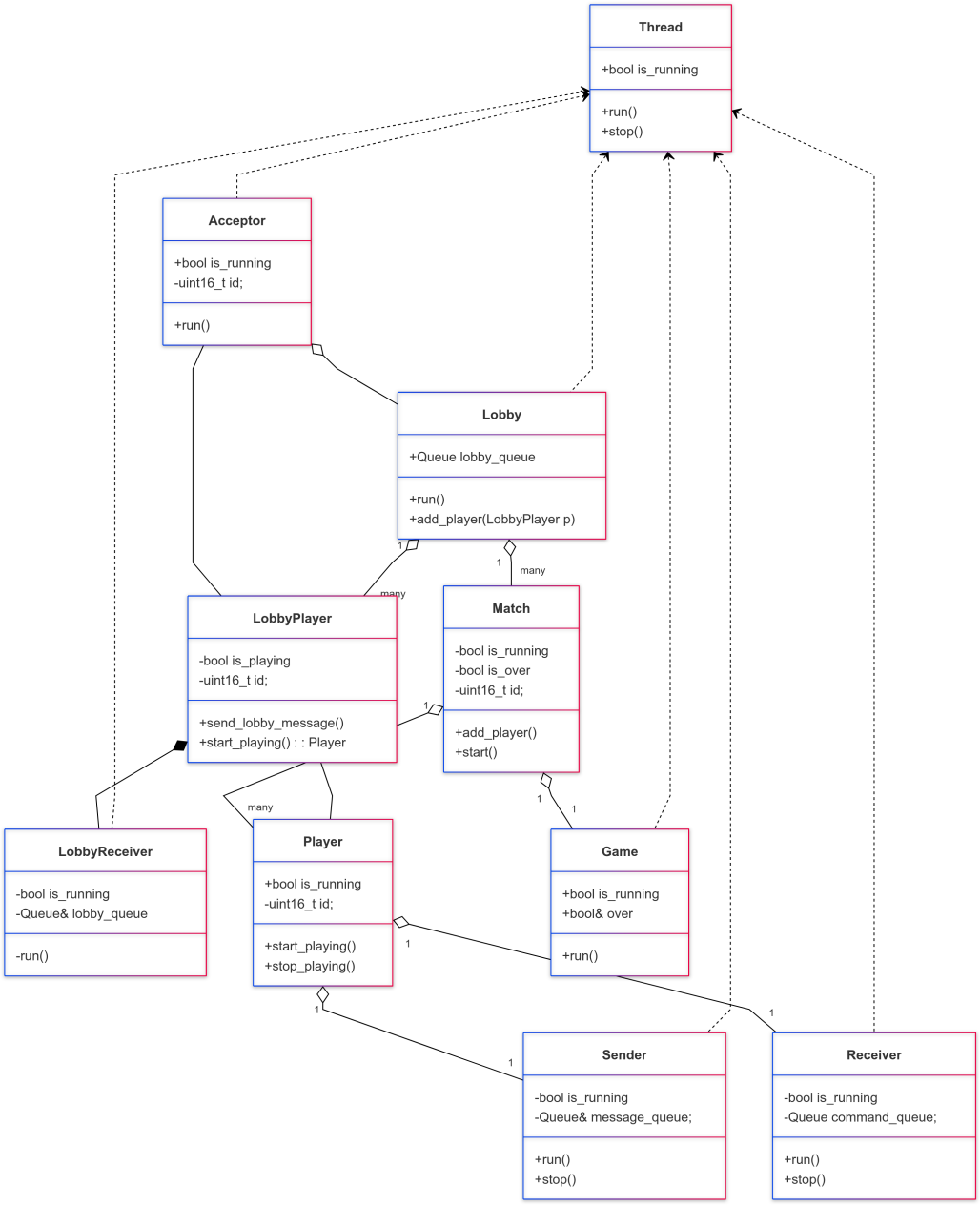

Figura 2: Diagrama de clases del servidor.

3. Cliente

Al iniciar el cliente, se crea la clase `Client`, que contiene dos hilos: `sender` y `receiver`, los cuales manejan una lógica similar a la del `player` en el servidor. Además, se crea un `SDLHandler`, encargado de:

- Cargar las imágenes necesarias.
- Manejar los mensajes recibidos en la *message queue*.
- Inicializar las ventanas del lobby y del juego.

Dentro del `SDLHandler` se instancian todas las clases necesarias para la inicialización, renderizado, procesamiento de eventos, presentación de ventanas y manejo de sonido. Asimismo, el `SDLHandler` ejecuta el `gameloop`, gestionando eventos, actualizando el estado del juego y renderizando las vistas.

La clase `RendererManager` se encarga de renderizar los distintos elementos del juego, incluyendo:

- **Renderizado estático:** Renderiza los elementos estáticos del juego, como el fondo y las plataformas.
- **Renderizado dinámico:** Renderiza los elementos que cambian durante el juego, como los patos, proyectiles, cajas y objetos arrojados.
- **Transformación de coordenadas:** Convierte las coordenadas de los objetos al espacio de la cámara para que se representen correctamente en pantalla.
- **Actualización de la cámara:** Ajusta la posición y el zoom de la cámara para seguir a los patos.
- **Renderizado de estadísticas:** Muestra información como la vida y las armas equipadas del jugador.

El `TextureHandler` gestiona las imágenes y fuentes utilizadas en el juego. Sus responsabilidades incluyen:

- Cargar y almacenar imágenes, ya sean estáticas o animaciones.
- Manejar las fuentes necesarias para el juego.
- Definir un *target* para renderizar, utilizado para dibujar todos los elementos estáticos.

- Eliminar correctamente todas las texturas al finalizar.

La clase `ScreenManager` es responsable de gestionar y mostrar las diferentes pantallas del juego, que incluyen:

- **Start:** Pantalla inicial del juego.
- **GetReady:** Pantalla que indica que el juego está por comenzar.
- **ServerIsDown:** Pantalla que se muestra cuando el servidor no está disponible.
- **NextRound:** Pantalla que aparece entre rondas.
- **EndMatch:** Pantalla de finalización del partido.
- **ScoreBoard:** Pantalla que muestra el marcador de los jugadores.
- **Lobby:** Pantalla del lobby del juego.

La clase `EventProcessor` utiliza `SDL` para capturar eventos de teclado y mouse. Además, se comunica con otras clases para actualizar el estado del juego y renderizar los cambios.

4. Editor de Niveles

La estructura del editor de niveles es sencilla pero eficaz. Está basado en una clase principal que gestiona todos los elementos que se pueden renderizar en el juego. La implementación se apoya en varias estructuras `unordered_map`, donde la clave corresponde a la ubicación en la grilla, y el valor es el elemento presente en dicha ubicación. Además, se utiliza una matriz auxiliar de ocupación, que permite verificar en tiempo constante $O(1)$ si una posición ya está ocupada por otro elemento.

El editor de niveles genera un archivo de texto (`.txt`) con el siguiente formato:

```
n
m

CRATES
x,y
x,y

SPAWNPLACE
x,y
x,y

BOX
x,y
x,y

ITEM
x,y, id_del_item
x,y, id_del_item

SPAWN DUCK
x,y
x,y
x,y
x,y
x,y
x,y
```

En este formato:

- `n` y `m`: Representan las dimensiones de la matriz ($n \times m$) que define el mapa.
- `CRATES`, `SPAWNPLACE`, `BOX`, `ITEM`, y `SPAWN DUCK`: Son las categorías de los elementos que se deben colocar en el mapa.
- `x, y`: Son las coordenadas de la grilla donde se ubicará el elemento correspondiente.
- `id_del_item`: Es el identificador único del objeto en el caso de los ítems, que permite diferenciarlos.

Este archivo es utilizado por el servidor para cargar los elementos en el mapa de juego, permitiendo la creación de niveles de manera estructurada y eficiente.

Se debe tener en cuenta que siempre deben crearse 6 spawns para los posibles patos en una partida. Logrando así alta consistencia.

5. Protocolo de Comunicación

La comunicación entre cliente y servidor en *DUCK GAME* se realiza mediante sockets TCP/IP, un protocolo confiable y orientado a la conexión que garantiza la entrega ordenada y completa de los datos. A través de este protocolo, se intercambian comandos y mensajes, que estructuran y organizan las interacciones entre cliente y servidor.

5.1. Estructura de comandos y mensajes

Tanto los comandos como los mensajes comparten atributos clave, entre ellos:

- `player_id`: Identifica al jugador que envía o recibe la información.
- `match_id`: Especifica la partida asociada al comando o mensaje.
- Posición (`x`, `y`, `z`): Define las coordenadas relevantes dentro del mapa del juego.
- `Flags de estado y renderizado`: Representan condiciones del juego, como el estado del pato, interacciones activas o eventos específicos.

El atributo más relevante es el `type`, representado como un `uint8_t`.

5.2. Uso del uint8_t type

El `type` es un byte que actúa como el identificador principal de cada comando o mensaje. Su funcionamiento es:

- Al enviar un comando o mensaje, el `type` es el primer byte transmitido.
- Al recibirlo, este byte determina cómo interpretar el contenido restante del mensaje.

Todos los posibles `types` están definidos en un archivo centralizado: `common/constants.h`. Esto garantiza que el protocolo sea fácil de mantener y escalar. Si se requiere agregar nuevas funcionalidades, basta con definir un nuevo `type` e implementar su lógica en cliente y servidor.

5.3. Manejo de errores

En caso de que se produzca un error de comunicación, como el cierre inesperado de un socket, se lanza una excepción que es capturada en los contextos correspondientes. Cada contexto maneja la excepción según el caso.