Taller de programación

Curso Veiga

Documentación técnica
Duck game

3 de diciembre de 2024

Matias Besmedrisnik 110487

Lourdes Ramirez Almada 105900

Ignacio Ramirez 111167

Ezequiel Aragon 110643

Índice

1. Introducción

Les presentamos la documentación tecnica del juego *DuckGame*, realizado para la materia de Taller de Programación, Cátedra Veiga, de la Facultad de Ingeniería de la Universidad de Buenos Aires. Este documento ayudará a enteder las bases del juego, desde protocolo hasta diagramas de clases y de secuencia.

2. Sockets

En el diseño del sistema de comunicación entre el servidor y el cliente, se utilizaron **sockets TCP**, los utilizados durante toda la cursada y requeridos para este trabajo. El uso de **TCP** (Transmission Control Protocol) asegura que los datos enviados desde el servidor sean recibidos de manera correcta y en el orden adecuado por el cliente. Este protocolo es ideal donde la integridad de los datos es crucial y donde las conexiones pueden mantenerse abiertas durante largos períodos de tiempo. Sus caracteristicas princpales son:

- **Orientado a la conexión**: TCP establece una conexión confiable entre el cliente y el servidor antes de iniciar la transmisión de datos.
- **Recepción de paquetes**: Los paquetes se reciben **todos** y en **orden**.
- **Acknowledgment**: Se envía un .ªcknowledge"(reconocimiento) para cada paquete recibido, lo que garantiza que los datos han llegado correctamente.
- **Validación mediante Checksum**: Cada paquete se valida mediante un **Checksum** para asegurar que no haya corrupción de datos durante la transmisión.
- **Transmisión simultánea**: Los datos pueden ser transmitidos simultáneamente en ambas direcciones, lo que permite una comunicación bidireccional eficiente.

3. Protocolo

3.1. Abstracción en el Envío y Recepción de Datos

El protocolo abstrae las tareas específicas de envío y recepción de distintos tipos de datos, como cadenas de texto (`string`), bytes (`uint8_t`), enteros cortos (`shorts`) y `float`.

De esta manera, tanto el `ProtocoloServer` como el `ProtocoloClient` se encargan únicamente de asegurar que los datos sean enviados o recibidos correctamente.

Los detalles de cómo se envían los distintos tipos de datos se pueden ver en funciones como `sendString`, `sendByte`, `sendShort` y `sendFloat`. De la misma forma, en recepción, funciones como `receiveString`, `receiveByte`, `receiveShort` y `receiveFloat` se encargan de recibir los datos enviados, interpretarlos correctamente y devolverlos al formato esperado. En el caso de las listas, será necesario enviar de antemano el tamaño de estas.

3.2. Envios por parte del servidor

Los envios por parte del servidor seran diferenciados por un primer byte de accion, este byte puede representar:

- `<0x24>` - Este byte se utiliza para notificar el color asignado a cada jugador. En este caso, el servidor envía el color de todos los jugadores para ser presentados por el cliente.

- `<0x20>` - Representa el envío de la escena de una partida en curso. Luego de este byte se enviará información detallada sobre el estado del juego, como el background, las plataformas, los personajes, las balas, armas y otros elementos relevantes.
- `<0x21>` - Este byte indica la finalización de una ronda dentro de una serie de X rondas. Se envía junto con un resumen de los resultados parciales del juego, es decir, la cantidad de victorias por jugador.
- `<0x22>` - El servidor lo utiliza para enviar la escena de victoria, indicando luego qué jugador ha ganado el juego.
- `<0x23>` - Este mensaje se utiliza para cerrar la partida, notificando al cliente que el juego ha terminado.

El ProtocoloClient sera encargado de recivir de forma adecuada segun el byte de accion.

3.3. Envíos por parte del cliente

Mientras el cliente no esté formando parte de una partida, se enviará un byte de acceso:

- `<0xCC>` - Se enviará cuando el cliente intente unirse a un juego.
- `<0xDD>` - Se enviará cuando el cliente cree una nueva partida.
- `<0xEE>` - Se enviará cuando el cliente inicie la partida.
- `<0xFF>` - Se enviará cuando el cliente solicite la lista de partidas disponibles.

Cada uno de estos bytes será seguido de los datos necesarios para la acción, que pueden incluir el nombre del jugador, el nombre de la partida o no contener datos adicionales.

En caso de que el cliente ya esté dentro de una partida, se enviará un byte de acción:

- `<0x08>` - Se enviará cuando el cliente realice una acción con el arma.
- `<0x1A>` - Se enviará cuando el cliente realice una acción de trampa.
- `<0x03>` - Se enviará cuando el cliente realice una acción de movimiento.

Cada uno de estos bytes será seguido por el tipo de movimiento o acción correspondiente.

4. Servidor

El servidor es el componente central de la aplicación, encargado de gestionar la comunicación entre distintas computadoras a través de sockets TCP. Su diseño busca garantizar una comunicación ágil entre los clientes y el servidor, asegurando un flujo eficiente de datos en tiempo real.

4.1. Principales clases

Acceptor maneja las conexiones entrantes al servidor, aceptando nuevos clientes y asignándoles hilos para su manejo. Se encarga de eliminar clientes inactivos y cerrar el socket de manera segura, optimizando los recursos del servidor y manteniendo la estabilidad del sistema en tiempo real.

Receiver recibe y procesa los comandos de los jugadores. Actúa como un hilo independiente que coordina las interacciones entre los jugadores y el `gameloop`, gestionando el inicio de las partidas y la detección de desconexiones para mantener la estabilidad del juego.

Sender maneja la comunicación saliente del servidor, enviando los estados del juego a los clientes. Utiliza una cola bloqueante para transmitir los datos de manera eficiente, sin bloquear

otras operaciones del servidor. También supervisa la conexión para asegurar la integridad de la comunicación.

LobbyPartidas gestiona la creación, administración y eliminación de partidas. Permite que los jugadores se unan a partidas existentes o creen nuevas, asignando roles como el de host. Es el encargado de proveerle al `gameloop` la queue para procesar los comandos del cliente y el mapa de colas protegidas de todos los integrantes de la partida.

GameLoop es el componente principal encargado de gestionar el desarrollo del juego en tiempo real. Se ocupa de procesar los comandos de los jugadores, controlar las acciones de los personajes, las armas, las balas y otros objetos del juego, y mantener el estado de la partida. Además, gestiona las plataformas, cajas y la reaparición de objetos, asegurando que el juego avance correctamente. El ciclo se basa en recibir y procesar los comandos de los jugadores, permitiéndoles moverse, disparar y usar armas dentro del juego. También se encarga de controlar los tiempos de reaparición de armas y defensas, así como de verificar si hay un ganador o si se ha terminado una ronda. Además, permite activar ciertos trucos que modifican el estado del juego. Funciona en un hilo independiente, lo que permite que el servidor maneje múltiples juegos a la vez sin bloquear otras tareas. Esto asegura que el juego siga funcionando sin interrupciones, enviando actualizaciones constantes a los jugadores sobre el estado del juego y permitiendo una experiencia fluida y dinámica.

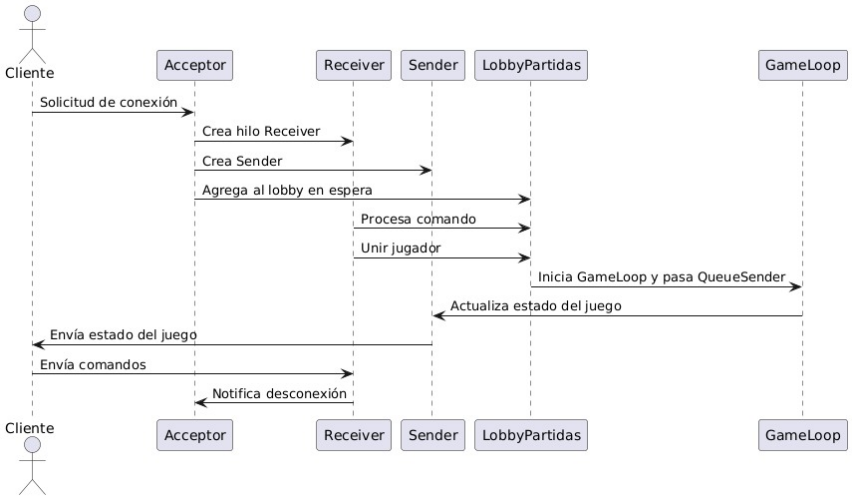

Figura 1: Diagrama de secuencia del server

5. Cliente

5.1. Principales clases

- **Client**: Controla el flujo principal del cliente, iniciando y gestionando el juego. Se encarga de establecer conexiones con el servidor y de gestionar las interacciones del usuario.
- **GameRunner**: Maneja el bucle principal del juego y coordina las partes importantes, como la gestión de eventos, el renderizado gráfico y la reproducción de sonidos.
- **GameRenderer**: Encargado de representar los objetos del juego en pantalla. Realiza el dibujo de elementos en base al estado actual del juego y mantiene actualizada la vista para el usuario.
- **EventHandler**: Captura y procesa los eventos del jugador, como movimientos, disparos y otras interacciones. Traduce estas acciones en comandos que el sistema entiende.
- **Graficos**: Proporciona soporte para la representación visual, incluyendo el manejo de texturas, renderizado de objetos y otros elementos gráficos.

- **Sound**: Brinda soporte para la reproducción de música de fondo y efectos sonoros, añadiendo una capa auditiva que complementa la experiencia del jugador.

5.2. Flujo del programa

A continuación, se presenta el diagrama UML que detalla la estructura y relaciones entre las clases del flujo del cliente gráfico y su lógica de ejecución:

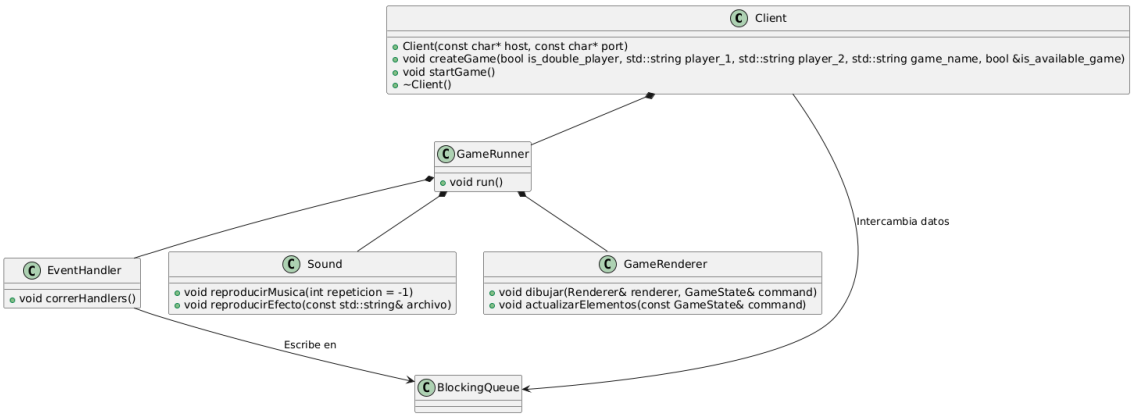

Figura 2: Diagrama UML del flujo del programa del cliente gráfico.

5.3. Renderizador

El **GameRenderer** utiliza la funcionalidad proporcionada por **Graficos** para acceder al *Renderer* y cargar texturas necesarias para la representación de los objetos. A su vez, administra el renderizado en pantalla de los elementos del juego, incluyendo personajes, armas y otros componentes visuales.

Los **Objetos Gráficos** representan los distintos elementos visibles en el juego, como el personaje del jugador (**ClientDuck**), balas (**Bullet**), armas (**Gun**), armaduras (**Armor**), cascos (**Helmet**), plataformas (**Platform**) y cajas (**Box**). El *GameRenderer* actualiza y dibuja estos elementos en base al estado actual del juego, asegurando una experiencia visual fluida y consistente para el usuario.

El siguiente diagrama UML detalla las relaciones entre el renderizador y los objetos gráficos involucrados:

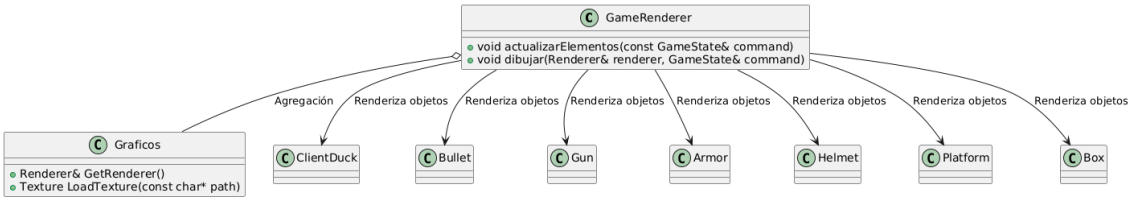

Figura 3: Diagrama UML del renderizador gráfico y sus objetos.

5.4. Menu

El menú principal del juego fue realizado con QT.

5.4.1. Clases principales

- **Menu**: Se encarga de la parte visual del menú, o sea, crea escenas y presenta en cada momento la correspondiente.
- **MenuController**: Es un intermediario entre la parte visual, Menu, y el juego representado en la clase Client. Cuenta con dos métodos fundamentales: start game, que se encarga de inicializar lo necesario para el juego a través del menú visual a través de Menu, y game, encargado de iniciar el juego como tal a través de Client.

5.4.2. Diagrama de clases

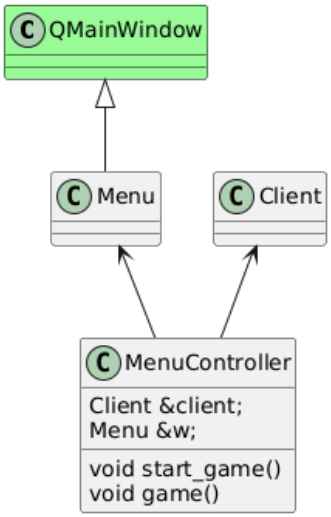

Figura 4: Diagrama UML de las clases del menú

6. Editor de niveles

El editor de niveles es una aplicación que, por medio de archivos YAML, es capaz de crear y editar mapas.

6.1. Principales clases

- **LevelEditor**: Controla la parte visual del editor, especificamente lo referido al menú principal y el menú contextual que permite elegir entre los distintos elementos disponibles.
- **LevelEditorController**: Es el handler de los eventos del editor. Se encarga de agregar (Esto a partir de los métodos set), eliminar y mover elementos, como también de guardar y cargar los archivos YAML. Tiene como atributos listas a los objetos añadidos, lo cual permite posteriormente guardarlos en un YAML, y una referencia a la escena que permite añadir y eliminar objetos en la misma.
- **MapObject**: Es la clase que representa *casi* todos los objetos del mapa dentro del editor (La excepción son los spawn que son representados con la clase QGraphicsPixmapItem perteneciente a QT). Hereda de QGraphicsPixmapItem, lo cual permite funcionalidades como el arrastre y el manejo de eventos de cada objeto.
- **Clases "Maker"**: Estas clases contienen métodos que, a partir de cierto argumento, devuelven información

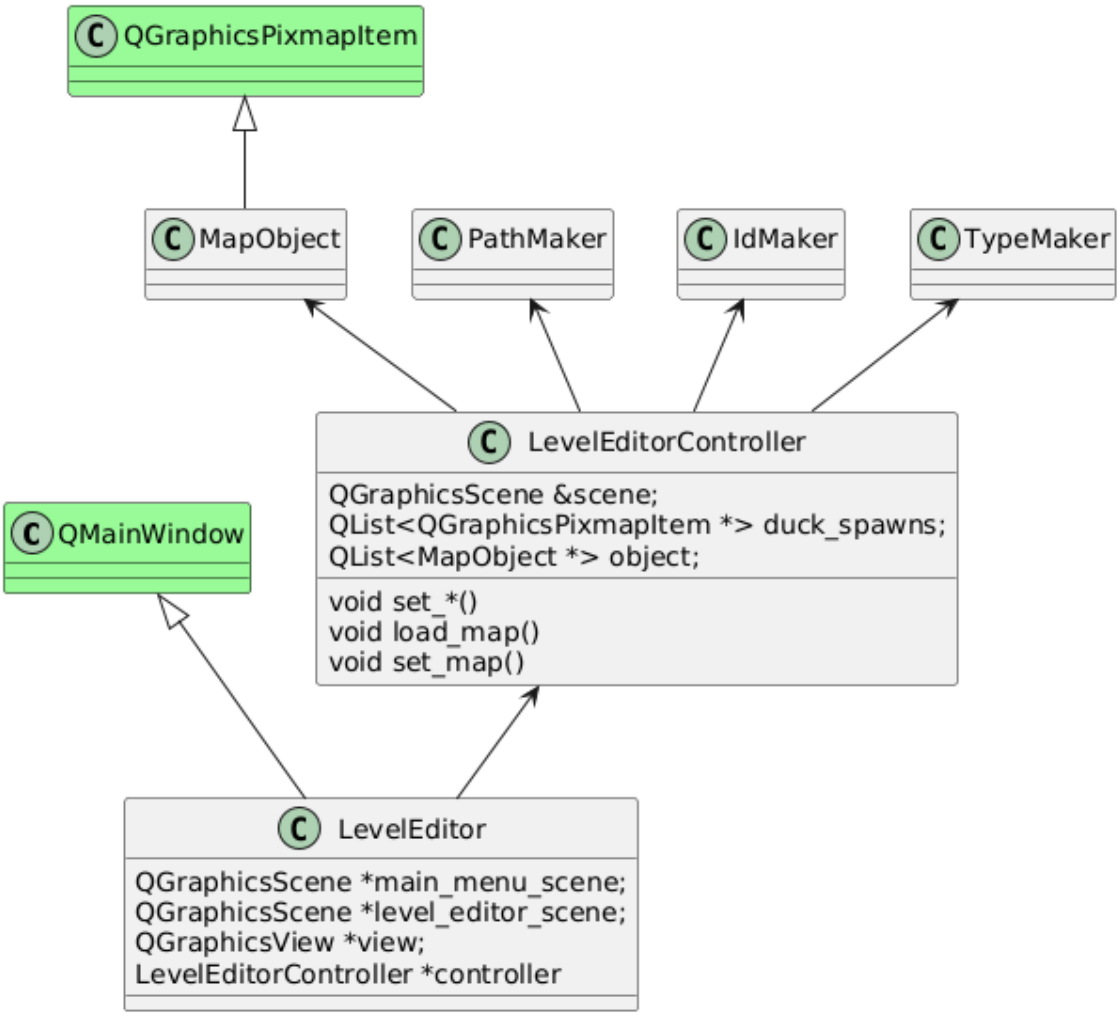

Figura 5: Diagrama UML de las clases del editor de niveles

- **Clase PathMaker**: A partir del nombre del tipo del objeto se devuelve el path a la imagen (Devuelven sus métodos datos de tipo string)
- **Clase IdMaker**: A partir del nombre del tipo del objeto se devuelve su ID por el cual es identificado en el archivo YAML del mapa correspondiente (Devuelven sus métodos datos de tipo int)
- **Clase TypeMaker**: A partir del ID devuelve el nombre del tipo del objeto (Devuelven sus métodos datos de tipo string)

6.2. Diagrama de clases

Nota: Las clases en color verde representan clases pertenecientes a QT.