

Informe TP Final

Taller de programación

Grupo 17

Integrantes:

Dante Alejandro Finci - 108456

Ezequiel Tosto Valenzuela - 108321

Matías Morales - 106793

Marcos Navarro Cafferata - 106974





Manual del usuario

Instalación

Para compilar el proyecto en un sistema operativo Linux primero tenemos que instalar las dependencias necesarias, estas son las librerías con las que se compila el juego y las librerías externas que nos ayudan al renderizado.

1. Tener una terminal en la ruta del filesystem o donde quiera instalarse las librerías
2. En la raíz del proyecto. Ejecutar el comando “**chmod +x installer.sh**” y luego “**./installer.sh**”
2.1 (En caso de requerir la clave del sistema, ingresarla ya que se aplican operaciones sudo al momento de instalar)
3. Esto va a generar una carpeta **build** la cual va a tener el ejecutable tanto del server como de los clientes.
4. Para levantar el server. Parado en la carpeta build con “**cd build**”, ejecutar el comando **./taller_server {PORT}** dónde PORT es el puerto elegido para que se ejecute el server.
5. Para levantar un cliente. Parado en la carpeta build, ejecutar el comando **./taller_server {IP_SERVER} {PORT}** donde IP_SERVER si se corre en la misma computadora será **localhost** y PORT el puerto elegido para el server.
6. (En caso de no poder ejecutar un cliente, ejecutar en la ruta principal el comando
`unset GTK_PATH`)
7. Para levantar el editor. Parado en la carpeta build, ejecutar el comando **./taller_editor**.

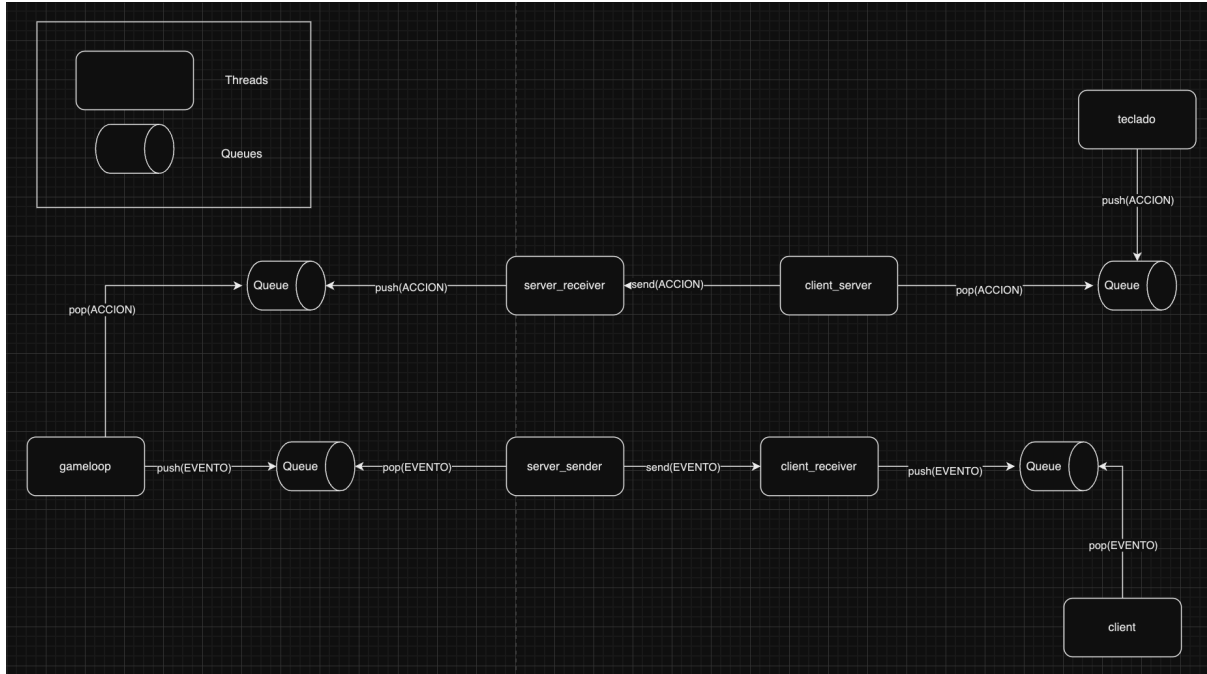
Controles

- Flecha : Agacharse
- Flecha : Apuntar para arriba
- Spacebar: Salto
- Flecha  : Moverse izquierda derecha
- Repetidamente Spacebar: Aleteo
- R: Reload
- V: Disparo

Documentación Técnica

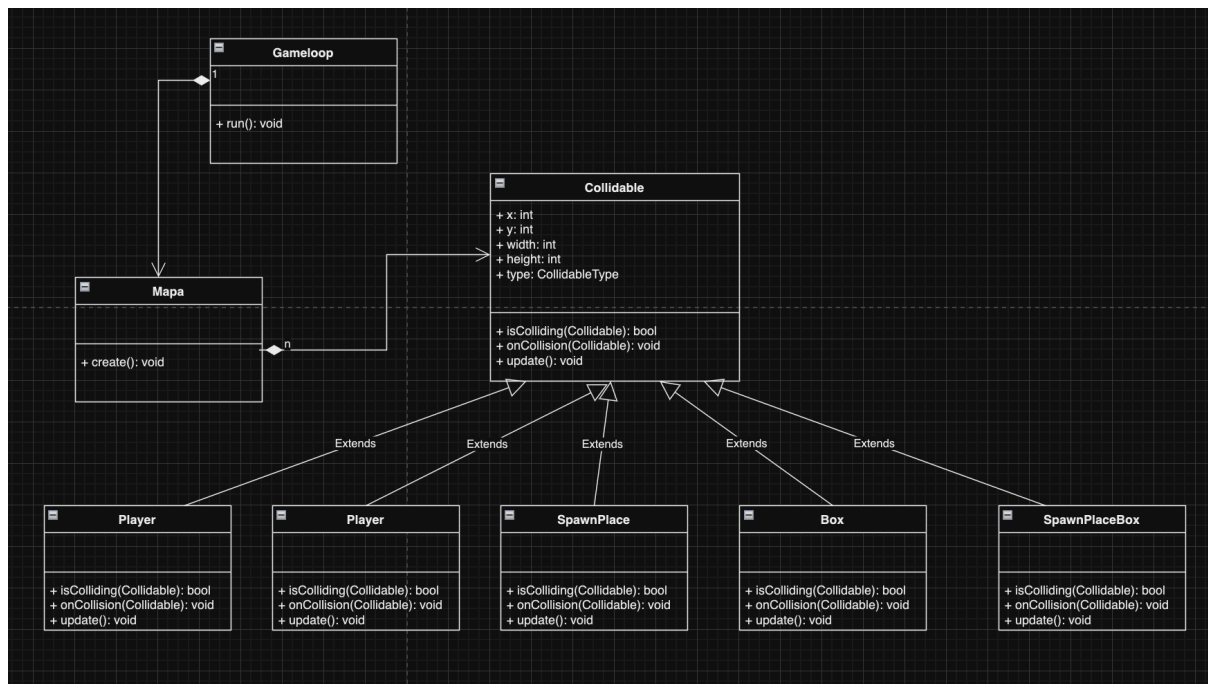
El proyecto está dividido en 3 partes. El servidor, los common (recursos compartidos entre servidor y cliente) y el cliente.

Arquitectura de comunicación



Ambos receivers y senders utilizan una implementación de Sockets IPv4 para TCP. Las queues tienen el comportamiento en el que pueden ser tanto bloqueantes como no bloqueantes. Para la simplificación del diagrama no se especifica cuáles son bloqueantes y cuáles no. Pero en definitiva los hilos que no podían dejar de ejecutarse como el gameloop o el cliente tienen comportamientos no bloqueantes con respecto a las queues.

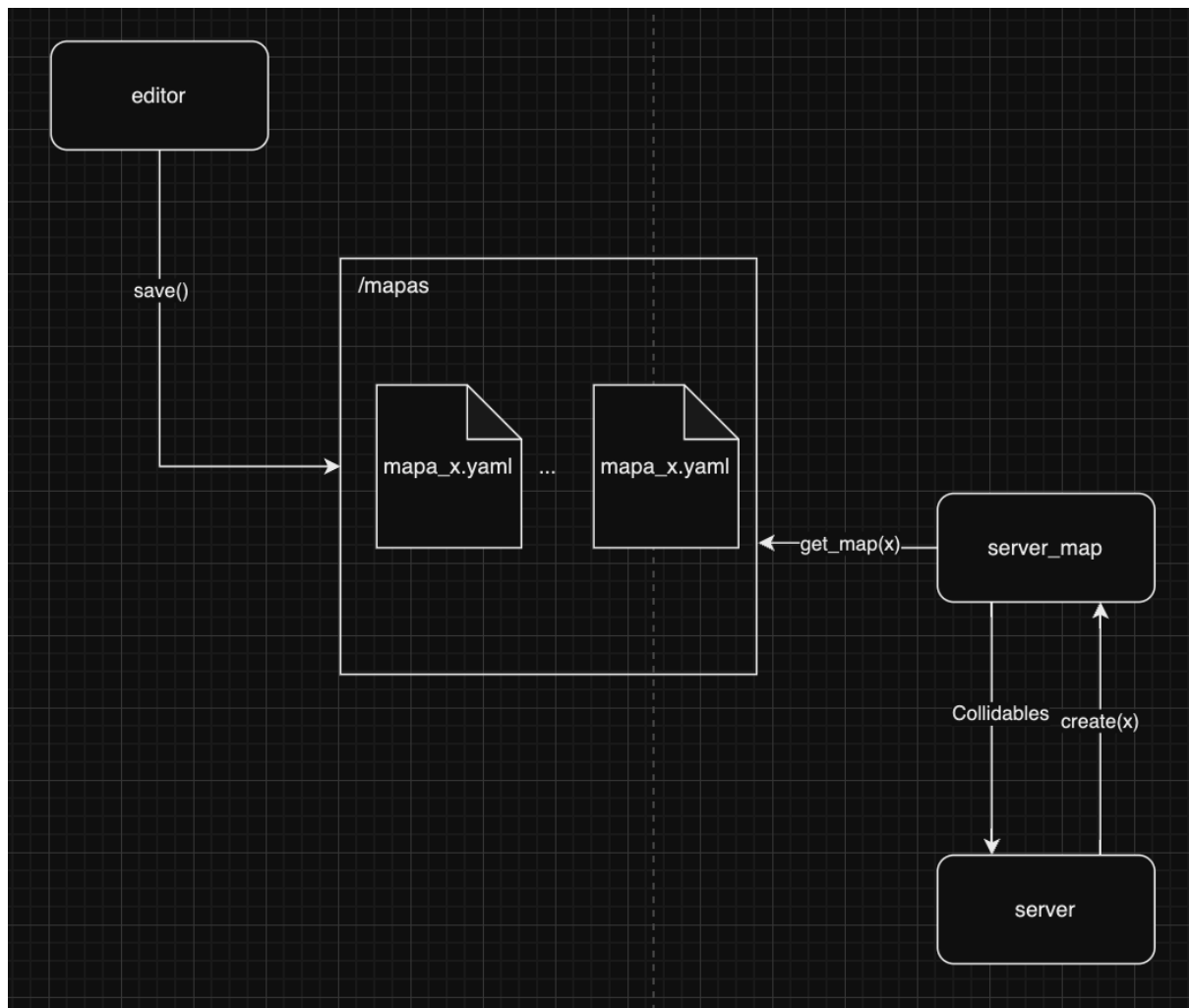
Lógica del juego



Sobre la lógica del juego, el mapa se trataba de una lista de colisionables, los cuales se actualizaban los unos con los otros dependiendo del tipo o la interacción que tenían. No implementamos ningún patrón de diseño para esta solución por falta de planeamiento pero seguramente algo más estructurado nos hubiera servido. Lo que generaba cada tick del servidor es que ocurran eventos según las posiciones o acciones de los jugadores (EventoWinRound, EventoMovimiento, EventoWinMatch, etc).

Editor

Se mantiene el estado de un mapa guardado en un directorio del servidor, donde después el servidor elige un mapa al azar dada la cantidad de mapas para intercambiar de mapa entre rondas. Esta selección NO puede darse en tiempo de ejecución (dado que el contado de los mapas según el directorio se hace en cuanto empieza el gameloop) por ende el rango va a ser como se disparó el



Lobby / Múltiples partidas

Para implementar las múltiples partidas tuvimos que subir un nivel de abstracción más para tener una lista de gameloops. El hilo aceptador (que es el que recibe las conexiones) envía al procesador del lobby del server 2 eventos. O que cargue las partidas o que cree una.

Si crea una partida entonces inicializa un gameloop y lo deja en espera hasta que se unan la cantidad de jugadores necesarios para empezar el juego.

En caso de que cargue una partida se envía el ID de la partida seleccionada y se agrega al jugador a esa partida.

Protocolo de comunicación

Para la implementación del mismo, el server el tipo de evento en 8 bytes, indicando el mensaje que debe leerse.

Mientras que del lado del cliente, se leen los primeros 8 bytes del mensaje, se identifica a que tipo de evento se refiere, y lee el mensaje correspondiente.

Por el lado del cliente, envía acciones con la misma lógica que los eventos, pero donde el tipo de acción es enviado en 6 bytes.

Eventos

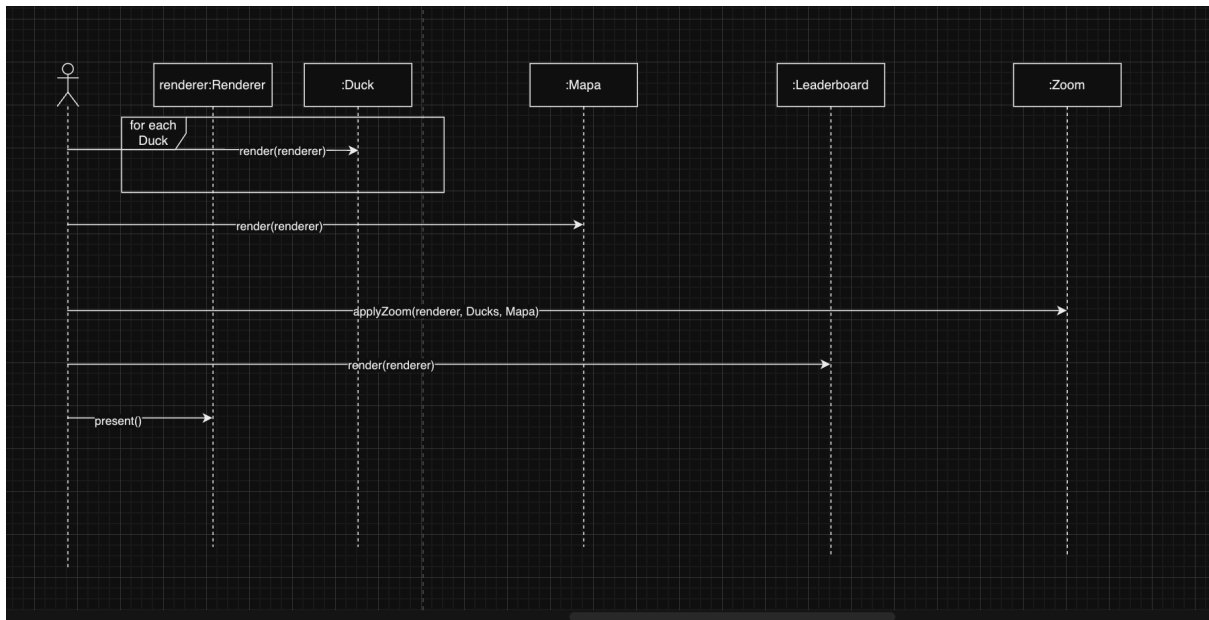
Los eventos son contextos para que el cliente actualice su renderizado en base a lo que pasa y el estado que guarda, dado que el cliente guarda el estado, solo hay que enviar donde ocurrió tal evento para que el cliente sepa que actualizar. No va a actualizar todo el estado todo el tiempo.

Aclaraciones importantes: El cliente mantiene la lista de colisionables y los eventos se basan en un x,y por ende ahí es donde se hace el matcheo donde accionar del lado del cliente.

Renderización

Para renderizar realizamos un approach bastante no ortodoxo. En vez de recalcular las dimensiones del servidor y pasarlas a un contexto de píxeles nuestra arquitectura tiene una relación 1:1, es decir 1 píxel equivale a 1 unidad de movimiento del lado del server.

Entonces para implementar la camara de los jugadores, primero renderizamos el contexto del servidor en el total de la pantalla (utilizamos 800x600 de medidas) y luego calculamos la textura generada para poder crear una cámara que mantenga el aspect ratio y siga renderizando el contexto del servidor.



Manual de proyecto

Ezequiel:

Trabajé en el motor de físicas de los patos (tanto en el server como en el renderizado), en las colisiones entre todos los colisionables detallados previamente y el armado del mapa del lado del server.

De parte de la lógica también realicé el desarrollo de la finalización del juego, el manejo de rondas y los múltiples patos por partida del lado del servidor. También trabajé en el editor de mapas y el zoom de la cámara.

Del lado del cliente también trabajé en el renderizado del Leaderboard o contador para la visualización de los puntos.

Hice la primera implementación de un arma y del SpawnPlace y la lógica de aparición del arma cada X tiempo + un offset random.

También realicé la documentación del proyecto mencionada previamente.

Utilizé Ubuntu 24.04 y VSCode como IDE, de linters utilicé el proporcionado por las librerías de vscode.

Sobre la documentación consultada fue una mezcla entre “repos que utilicen estas librerías” y “documentación de la librería” dado que SDL2 no tiene tanto open source o varía mucho la implementación y necesidad.

Matias:

Trabaje en la estructura general del cliente, tanto la implementación de los threads, como las queues y los eventos y acciones recibidos y enviados. Implemente la lógica para manejar múltiples clientes, la lógica de disparos, las Box y sus distintos Spawns de armas, entre otras cosas. También la estructura del protocolo, y la estructura de los eventos a enviar. Por otro lado, también trabaja en múltiples tareas de SDL como el renderizado del mapa, la música, entre otras cosas.

Por mi parte, tambien utilize Ubuntu 24.04 y VSCode com IDE.

Dante:

Estuve trabajando en las renderizaciones del pato al principio del tp, subí una ventana por ejemplo y manejé un poco los threads del cliente. Luego pasé a trabajar en el lobby tanto en la parte del cliente como en la del servidor. Trabaje en la parte del reseteo de la cola de comandos cuando el jugador comienza la partida. Y trabaje en la interfaz gráfica del lobby utilizando qt. En la parte del servidor también estuve implementado la estructuras del gameloop, aceptador y los senders y recievers de los jugadores

Marcos:

Desarrollé el Disparo Manager, encargado de controlar toda la lógica relacionada con los disparos del jugador. Implementé la detección de colisiones de las balas para identificar qué objeto fue impactado primero y definí la lógica de las consecuencias según cada caso, incluyendo los eventos necesarios para la comunicación con los clientes.

Trabajé en la lógica de los disparos, definiendo las responsabilidades del arma y su física. Desarrollé la lógica de dispersión, la funcionalidad de disparo automático y las mecánicas específicas de las armas, como los diferentes tipos de recarga y funcionalidades particulares. Además, implementé la capacidad de disparar hacia arriba, considerando la dispersión en el eje horizontal y separando la lógica del movimiento del jugador.

En el player, trabajé en la lógica relacionada con los disparos, el apuntado, las armas y las recompensas obtenidas. También implementé una nueva SpawnBox, incorporando las mecánicas de protecciones. Esto incluyó tareas en SDL para la representación visual de los diferentes estados de las armas, protecciones y del pato.

Utilicé Ubuntu 22.04 y VSCode con IDE.

Mayores pain points:

- Desconocimiento de las librerías de renderizado como QT5 y SDL2.
- Utilización de SDL2 puro y no el wrapper SDL2pp.
- Poca documentación para integrarse directamente desde el cmake con SDL2 y QT5 y no depender de instalaciones externas.
- Realizar el sistema de referencia en el servidor en bottom left mientras que SDL2 QT5 trabajan en top left ("es solo restarle la altura" hasta que no).
La justificación fue "No quiero que el pato salte para -y" que tuvo sentido hasta que tuvimos que integrarlo con el resto de librerías.
- Organización y división del trabajo.

Pendings

- Lobby con multipartidas
- Unit testing
- Animación smooth de disparo
- Asignación de colores on movement (tech debt)
- Valgrind y leaks.

Que nos hubiera servido de antemano

- Saber cómo dividir tareas entre server y client.

- Tener las dependencias antes de su utilización (que el CMake que provee la cátedra funcione para todos los casos).
- Priorización de importancia de features.
- Definir entregables con los correctores.
- Practicar un protocolo en binario. (No lo hicimos en los TPs individuales)