Taller de Programación I

Duck Game

Integrantes:

José Manuel Dieguez - 106146

Trinidad Angeles Bucciarelli - 105494

Matias Daniel Rueda - 106963

Índice:

Introducción

En esta documentación se explicará en detalle la arquitectura utilizada tanto para el cliente como para el servidor, abarcando todos los aspectos relevantes de su diseño y funcionamiento. Se detallara cómo se utilizaron los **sockets** para la comunicación bidireccional entre ambos. Además, se abordarán los mensajes enviados y recibidos, los patrones de diseño y las estrategia utilizadas, así como las ventajas obtenidas

Cliente-servidor

Interacción

La comunicación entre cliente y servidor se realizará mediante sockets utilizando el protocolo TCP/IP, garantizando una conexión estable y segura para el intercambio de datos.

Crear partida y unirse a partida:

Independientemente si se quiere crear o unirse a una partida el cliente debe enviar un mensaje indicando el tipo modo de juego y el número de jugadores que se quieren unir el mensaje tendrá el siguiente estilo:

- Game mode
- Números de jugadores

La respuesta por parte del servidor será esta:

- ID
- Color

Esta información será utilizada en dos situaciones. Para mostrar al usuario que color le fue seleccionado y el id será utilizado para diferenciar a los jugadores cuando se esté jugando localmente de a dos.

Lista de partidas:

Además, el jugador podrá solicitar las partidas disponibles cuando desee unirse a una. En el lado del cliente, se enviará una acción denominada **Game List**:

- Acción

La respuesta por parte del servidor será esta:

- Lista de partidas

Esta información será utilizada para mostrarle al usuario que partidas están disponibles.

Juego:

Durante el juego la interacción cliente-servidor se realizará a través de snapshots.
Estos serán enviados constantemente por el servidor con toda la información que el cliente necesita para renderizar. En particular se enviará la siguiente información:

- Patos
- Armas
- Projectiles
- Explosiones
- Cajas
- Armaduras
- Puntajes
- Mapa
- Cámara
- Finalizado
- Rounds
- Color ganador

Cliente

Arquitectura

El cliente contará con tres hilos en total y dos colas para los mensajes. El hilo principal se encargará principalmente de mostrar la interfaz de usuario, gestionando la interacción y actualización de los elementos en pantalla. El segundo hilo será responsable de recibir mensajes del servidor, procesando la información en tiempo real, mientras que el tercer hilo se encargará de enviar los mensajes hacia el servidor.

Las colas, tanto para enviar como para recibir mensajes, serán no bloqueantes, así se permite que el juego siga ejecutándose sin interrupciones, incluso si hay un retraso en el procesamiento de mensajes. Este enfoque es crucial para mantener la fluidez del juego y asegurar que la comunicación entre cliente y servidor no afecte el rendimiento.

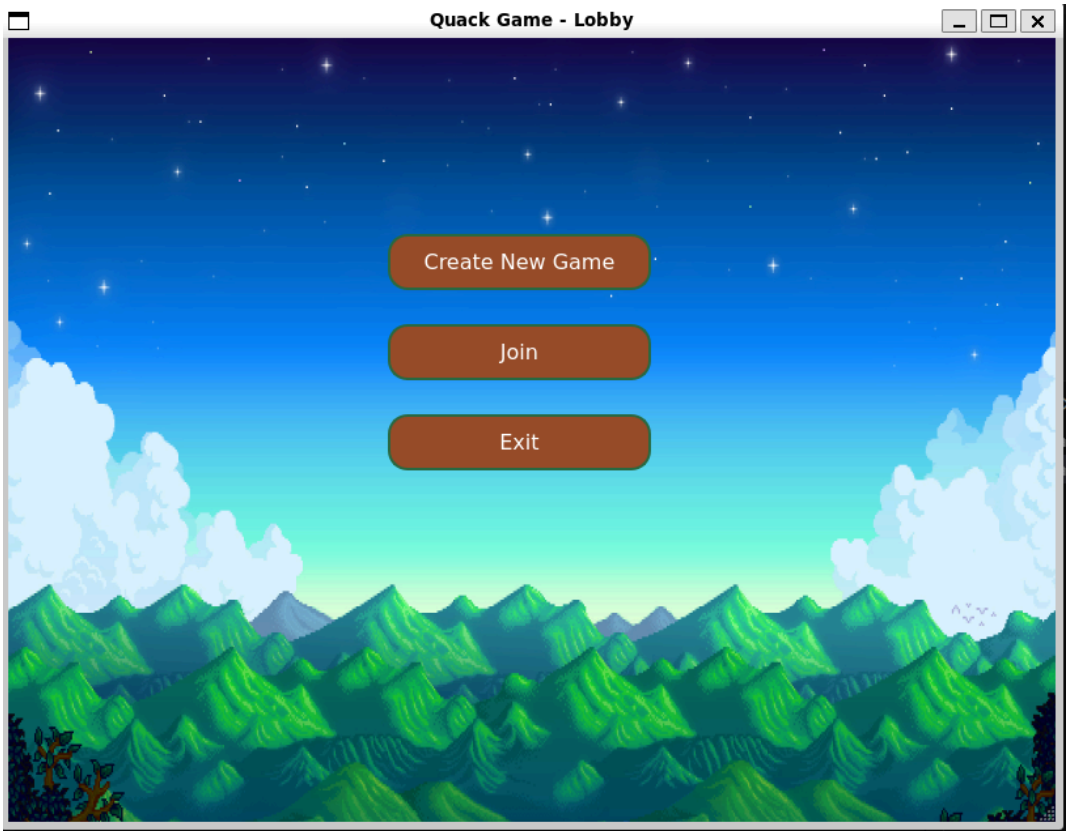
Patrón de arquitectura

Utilizamos el patrón de arquitectura MVC (Modelo-Vista-Controlador) para mejorar la modularización en el lado del cliente.

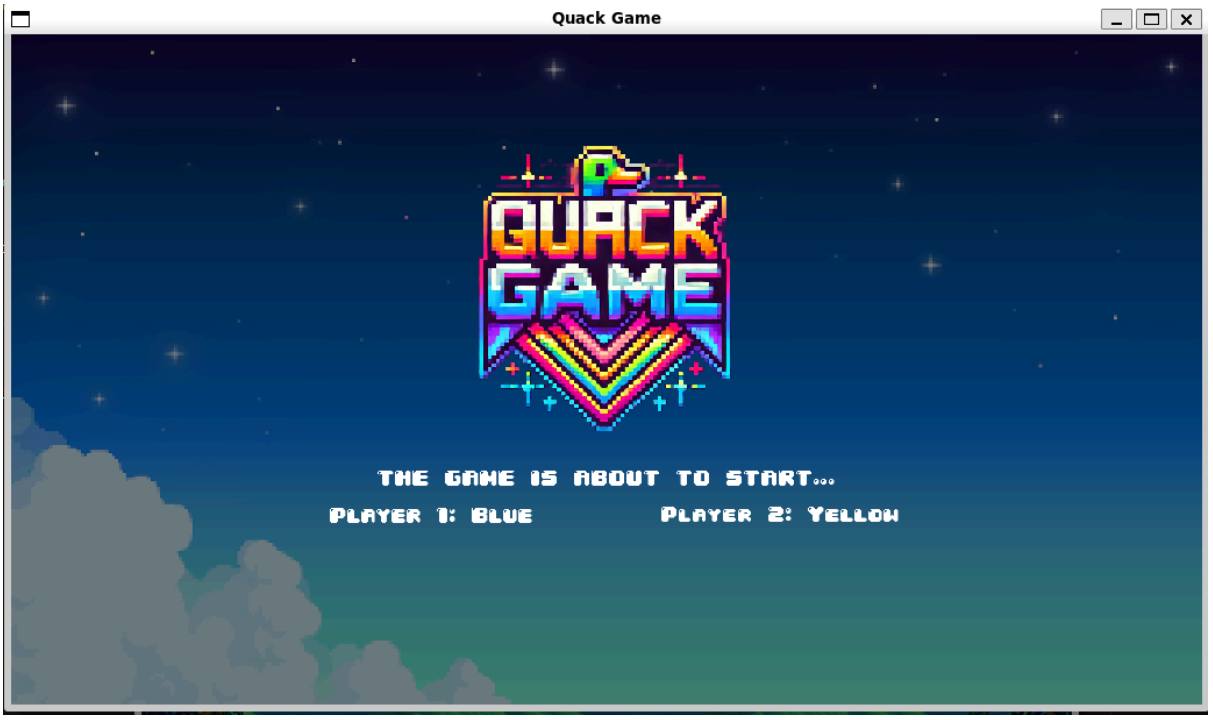
Vista (View): Aquí se incluyen todos los elementos relacionados con la interfaz gráfica, como la pantalla de inicio, la pantalla de resultados del juego, la pantalla de carga, la pantalla de inicio del juego, la tabla de puntuaciones, la sala para crear partidas y el listado de partidas disponibles.

Algunas pantallas:

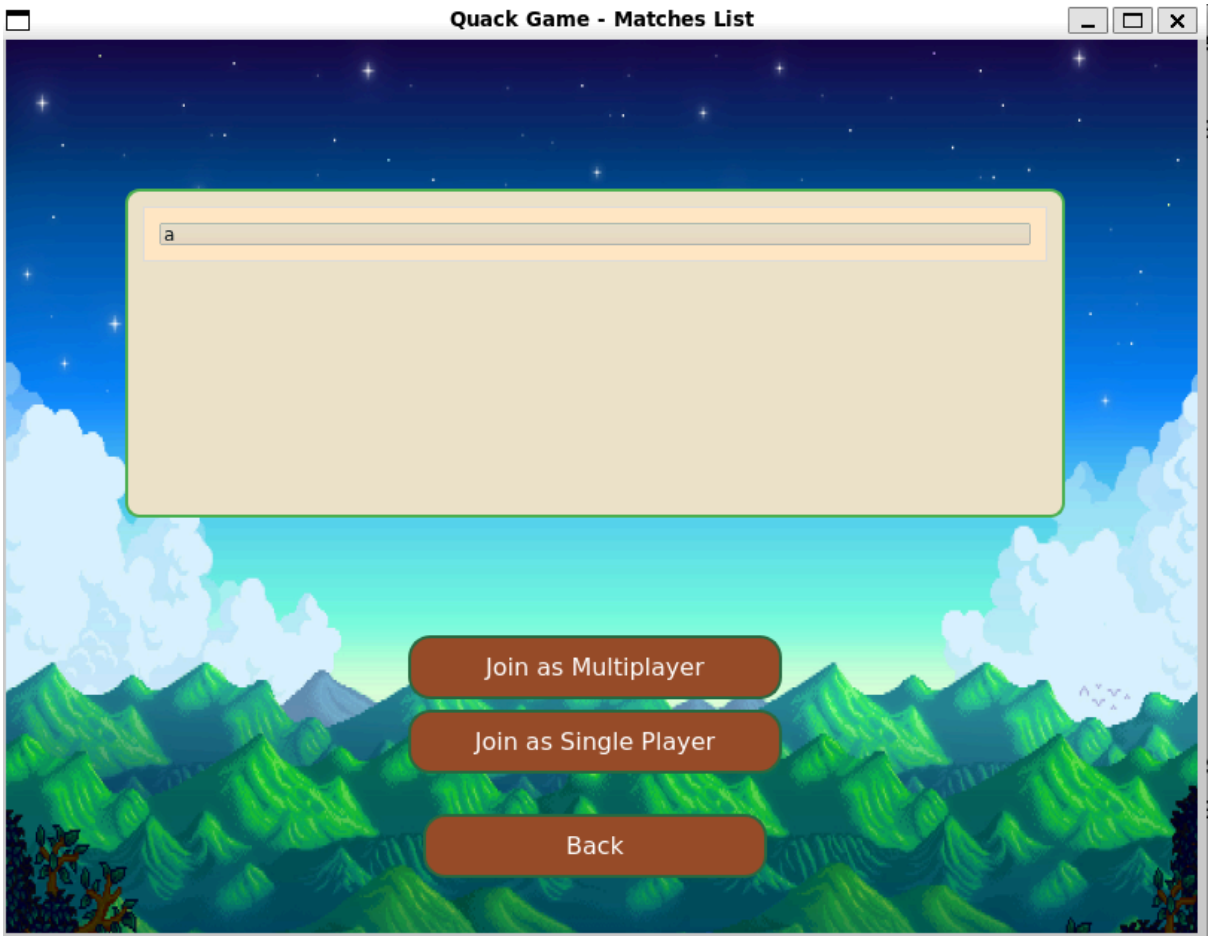
Pantalla de inicio:



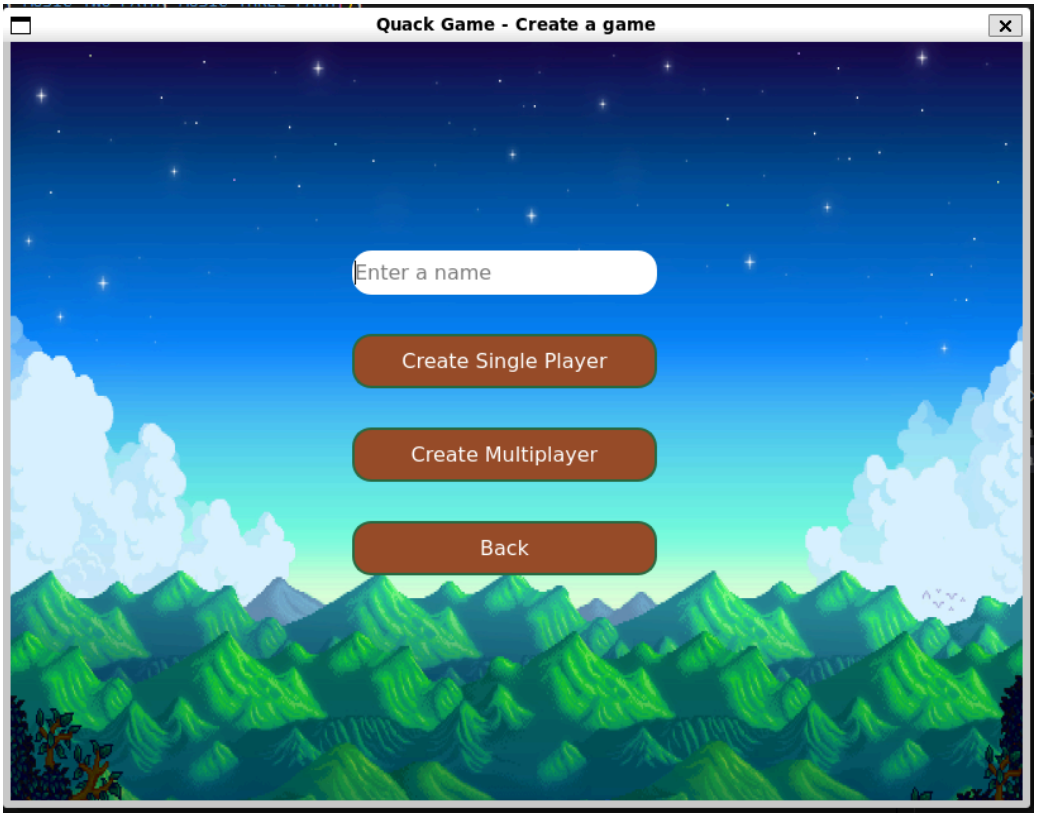
Pantalla de carga:



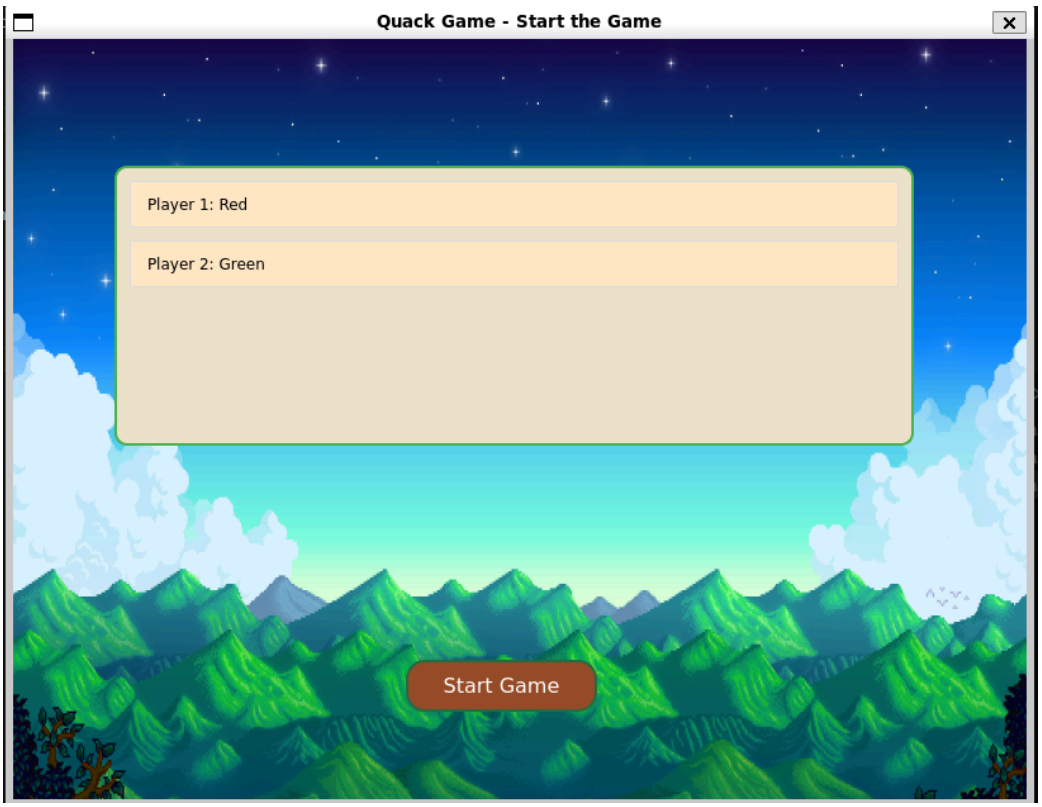
Pantalla con la lista de partidas:



Pantalla para crear partida:



Pantalla para iniciar partida:



Modelo (Model): Aca se encuentra toda la lógica utilizada por el cliente, como la gestión del lobby, los recursos empleados (sonidos, texturas), el protocolo de comunicación, y las colas de mensajes (tanto de recepción como de envío).

Controlador (Controller): En esta capa se gestiona la lógica relacionada con SDL, como la música y los efectos de sonido. También se incluyen los controladores de los renderizadores, la interacción del usuario y las funcionalidades de trucos del juego.

Patrones de diseño

Utilizamos distintos patrones de diseño, como Flyweight, Command y Singleton:

Flyweight: Un ejemplo de donde fue utilizado este patrón es en las texturas. Utilizamos este patrón para almacenar las texturas que ya han sido cargadas una vez, y así poder utilizarlas. De esta manera, mejoramos significativamente el rendimiento, ya que no es necesario cargar las texturas cada vez que se las necesita.

Command: Este patrón fue utilizado para implementar las teclas que los usuarios utilizarán. De esta manera, conseguimos modularidad en la asignación de teclas, facilitando su extensión si en algún momento lo requerimos.

Singleton: Utilizamos este patrón en varias ocasiones, por ejemplo, para crear las clases TextureStorage y Config. De esta manera, nos aseguramos de que solo haya una única instancia de estas clases y puedan ser accedidas desde cualquier lugar.

Renderizado

En el proceso de renderizado, nos encargamos de mostrar cómo deben visualizarse los distintos elementos que se presentan en pantalla. Con la ayuda del snapshot, que nos proporciona las coordenadas de estos elementos, y las clases Size, podemos colocar y renderizar los objetos en el renderer.
Además, empleamos estrategias basadas en la información de la cámara para asegurarnos de que solo se muestren aquellos elementos que están dentro del rango visual de la misma.

Interacción del usuario

En el lobby

El usuario tendrá la opción de crear partidas o unirse a ellas.
Cuando desee unirse a una partida, el cliente enviará un mensaje solicitando las partidas disponibles, que recibirá como respuesta del servidor y, de esta forma, podrá mostrar la lista en pantalla.

Al crear o unirse a una partida, se deberá especificar cuántos usuarios jugarán de forma local (1 o 2). Esta información será enviada al servidor a través de sockets, el cual responderá con un mensaje que incluirá el color y el ID asignado a cada jugador. Esta información es útil para mostrar al jugador su color asignado, y el ID es necesario para identificar qué usuario está enviando mensajes durante el juego.

Posteriormente, se pasará a la pantalla de carga si se unió a una partida, o a la pantalla de inicio de partida si la creó. Además, tendrá la opción de enviar un mensaje al servidor para dar comienzo a la partida mediante el botón "Start Game".

En el juego

El usuario podrá interactuar enviando información al servidor utilizando las teclas ya configuradas, se implementaron varias estrategias para optimizar el proceso. Un ejemplo de esto es la restricción de enviar más de un mensaje consecutivo para una misma acción. Por ejemplo, si el usuario envía un mensaje indicando que se mueve a la derecha, mantener presionada esa tecla no generará más mensajes repetidos. Esto mejora el rendimiento, tanto del cliente, ya que no envía mensajes innecesarios, como del servidor, que no tendrá que procesar información redundante.

Manejo de estado del juego

El estado del juego se actualizará utilizando la información recibida del servidor. Con esta información, se podrá actualizar el color del pato, las rondas, el jugador ganador, si se debe mostrar la pantalla de carga o si es el momento de mostrar la tabla de resultados, entre otros aspectos.

Música y sonidos

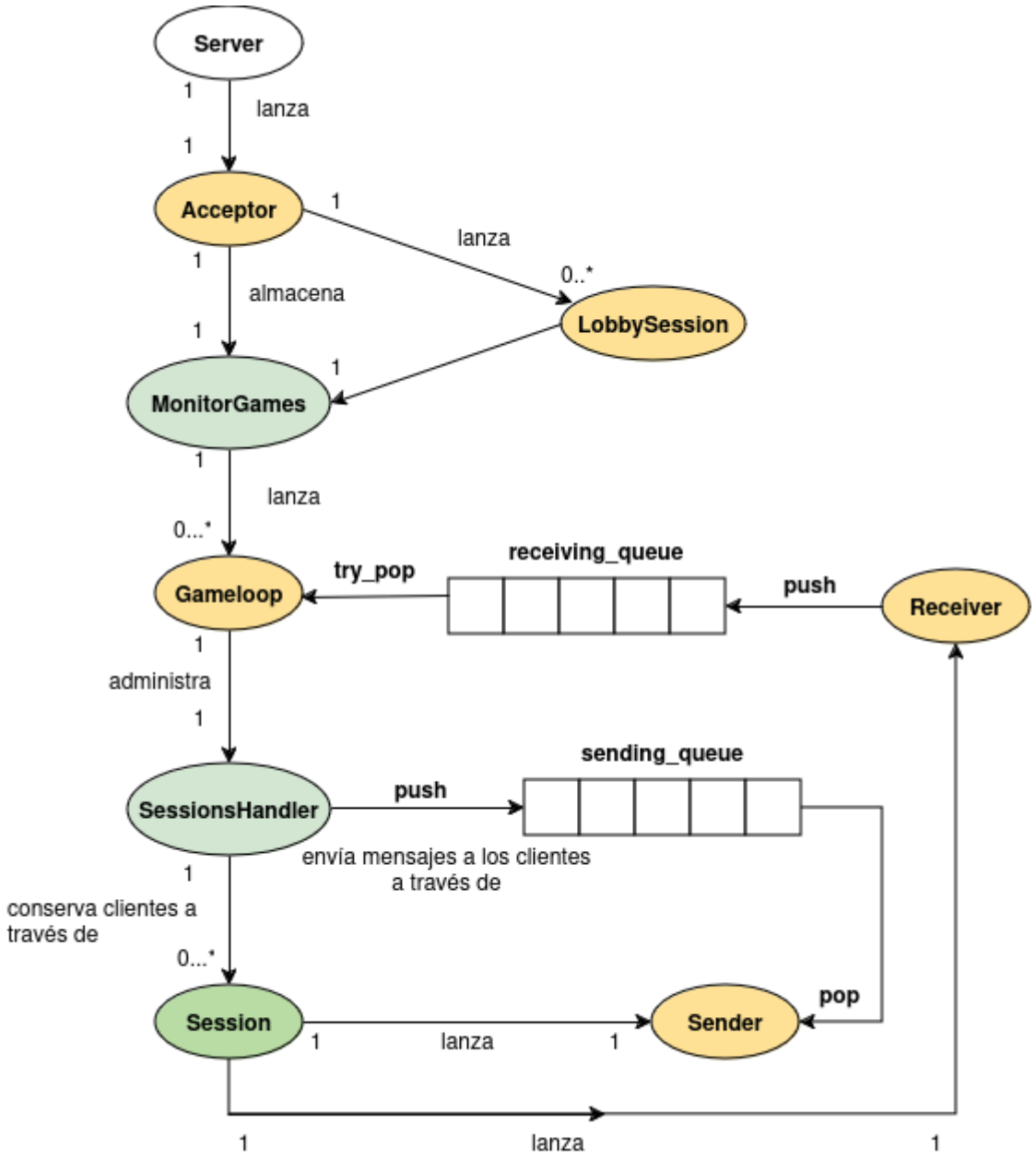
Para los sonidos, utilizamos una estrategia similar a la de las texturas: una vez cargados, se almacenan para poder reutilizarlos. Además, aplicamos un control de sonidos para evitar saturar al usuario con demasiada información.
Para esto es necesario que cada elemento que vaya a emitir un sonido tenga un ID asociado, lo que nos permite identificar qué elementos ya han emitido su sonido.
En cuanto a la música, se decidió implementar una clase MusicBox, que se encargará de reproducir las tres músicas establecidas durante el juego.

Servidor

El servidor será el encargado de gestionar la lógica central del juego, desde recibir nuevas conexiones, almacenar partidas, procesar las acciones de los clientes y mantener la integridad del estado global del juego.

Administración de clientes

A continuación se muestra un diagrama de la arquitectura del **Servidor**.



El Servidor lanza un hilo **Acceptor** responsable de recibir las nuevas conexiones de los clientes. A su vez, asociado a este último, se lanza un hilo **LobbySession** que se encargará de administrar las acciones pre partida de un usuario, como lo son:

- Crear una partida
- Listar las partidas existentes
- Unirse a una partida

Esto se hace apoyándose en el **MonitorGames**, que de forma thread-safe, puede almacenar múltiples hilos **Gameloop,** los cuales se encargan de almacenar y administrar un **Game** según las acciones realizadas por los clientes. Cada Gameloop contiene un **SessionsHandler**, clase monitor que controla de forma thread-safe las conexiones de clientes (**Sessions**) asociados a una misma partida.

Cada **Session**, correspondiente a un cliente en particular, a su vez lanza dos hilos: Un **Sender**, encargado de enviar el estado actual del juego al cliente, y un **Receiver**, que se encarga de recibir las acciones de los clientes durante la partida.

Esta arquitectura permite la liberación de recursos en forma de cascada. Ante una nueva conexión de un cliente, el Acceptor elimina las partidas que han terminado a través del MonitorGames.

A su vez las partidas liberan los recursos asociados a los clientes a través de su SessionsHandler una vez terminada la partida, o bien ante la desconexión del mismo durante el transcurso del juego. También se puede interrumpir una partida si todos los usuarios se desconectaron, o bien si se cierra el Servidor manualmente.

Lógica del juego

El **Gameloop** cuenta con un **Constant Rate Loop** con el cual puede ejecutar una función (step) cada frame de forma constante.

Dentro de este step se realiza lo siguiente:

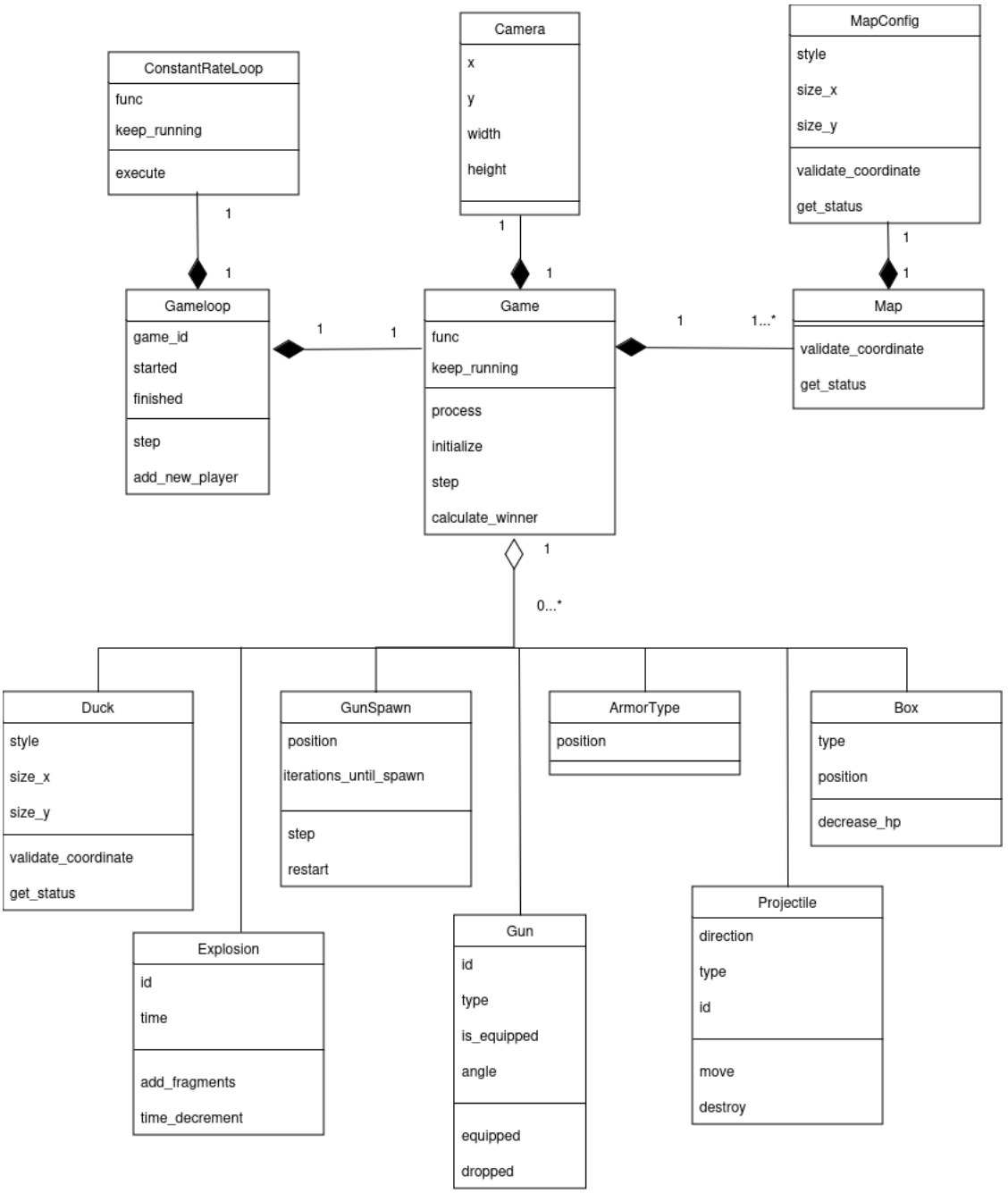
- Se corrobora que la partida no haya terminado o que sigan usuarios conectados.
- Se reciben comandos de los clientes (por ej, disparar, moverse, etc)
- Se procesan los comandos de los clientes.
- Se simula una iteración del **Game**, a través de su función step.
- Se obtiene el estado actual del juego.
- Se eliminan las sesiones desconectadas.
- Se envía el estado actual del juego a todas las sesiones activas.

Durante la simulación del **Game**, se realiza lo siguiente:

- Se corrobora que la partida no haya finalizado.
- Se inicializa el siguiente mapa en caso de ser necesario.
- Se simulan las físicas de los proyectiles y explosiones, rompiendo cajas y spawneando o no (una caja puede estar vacía) armas/armaduras/granadas en caso de ser necesario.
- Se simulan las acciones realizadas por los patos (jugadores), como moverse, disparar, saltar, aletear, intentar recoger o tirar un arma, etc.
- Se corrobora la colisión de patos y armaduras.

- Se spawnean y mueven armas.
- Se actualiza la cámara según la distribución de los patos en el mapa.
- Se corrobora que haya un ganador.

A continuación se muestra un diagrama de clases resumido de las clases involucradas en la lógica del juego.



Consideraciones: Para el caso de los proyectiles y explosiones, no están vinculadas directamente con el juego tal como se muestra en el gráfico, sino que se cuenta con un **ProjectileManager** y un **ExplosionManager**. A fin de resumir el gráfico se omitieron estas clases. **Game** colecciona los **GunSpawn**, **ArmorType** y **Box** en diccionarios con key igual a la posición de los mismos. En este diagrama se muestran las posiciones como atributos de la clase en sí, aunque no lo son.

Estado actual del juego

Cada iteración el servidor envía a los clientes la siguiente información:

- La información relacionada a cada pato, como su: id, posición, tipo, posición y ángulo del arma, su color, cuántas victorias lleva, y también su estado (si se está moviendo, disparando, si tiene casco, armadura, etc).
- Las armas tiradas en el mapa.
- Los proyectiles y explosiones.
- Las cajas restantes.
- Las armaduras y cascos spawneadas en el mapa.
- El mapa (refactorizable, podría enviarse una vez por ronda).
- La cámara.
- El número de ronda.
- Si terminó o no la partida.
- El color del pato vencedor.

Créditos:

Gracias al ElDipa por permitirnos utilizar su implementación de Socket , Queue y Thread, a continuación dejo el link a su repositorio donde se podrá encontrar su implementación:

Socket: https://github.com/eldipa/hands-on-sockets-in-cpp

Thread y Queue: https://github.com/eldipa/hands-on-threads