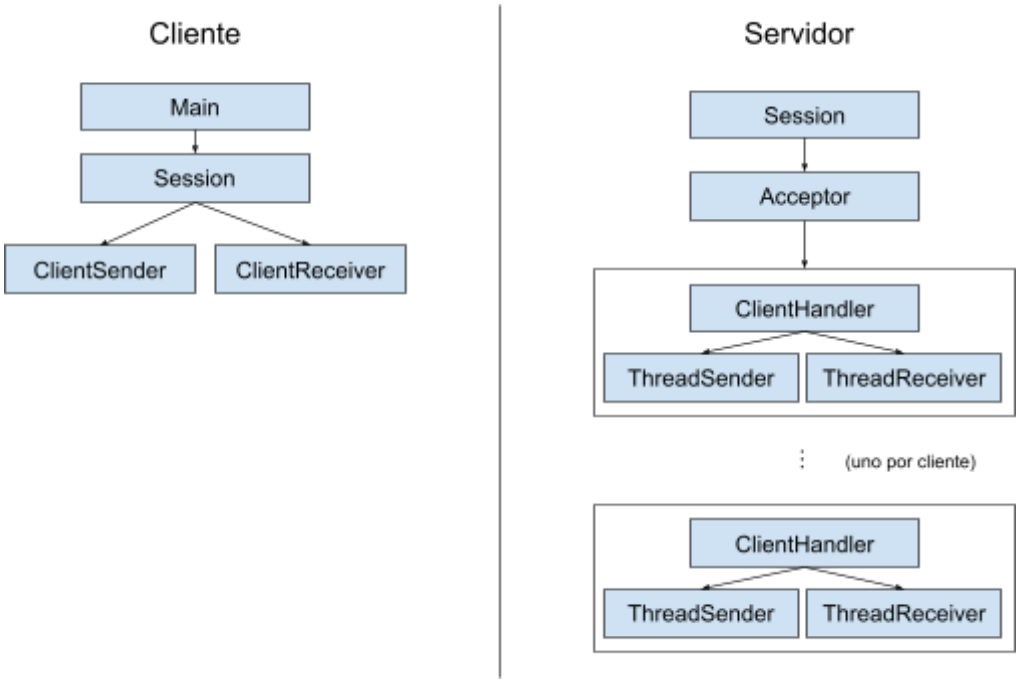
Trabajo Práctico Final
Informe Técnico
Grupo 26

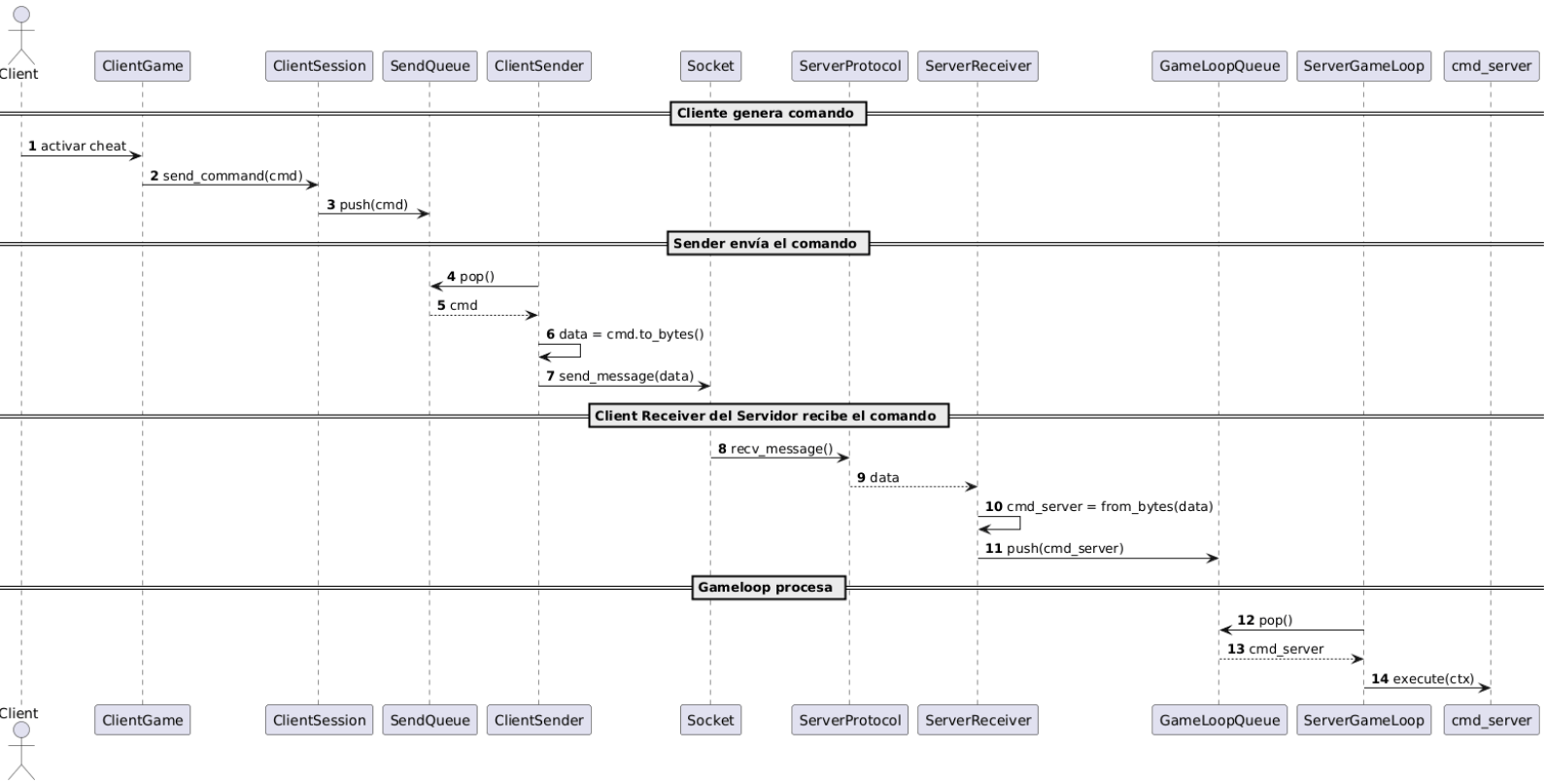
Nombre Integrante	**Padrón**
Máximo Giovanettoni	110303
Ulises Valentín Tripaldi	111919
Bautista Cánepa	111892
Máximo Calderón Vasil	111810

Estructura general del programa

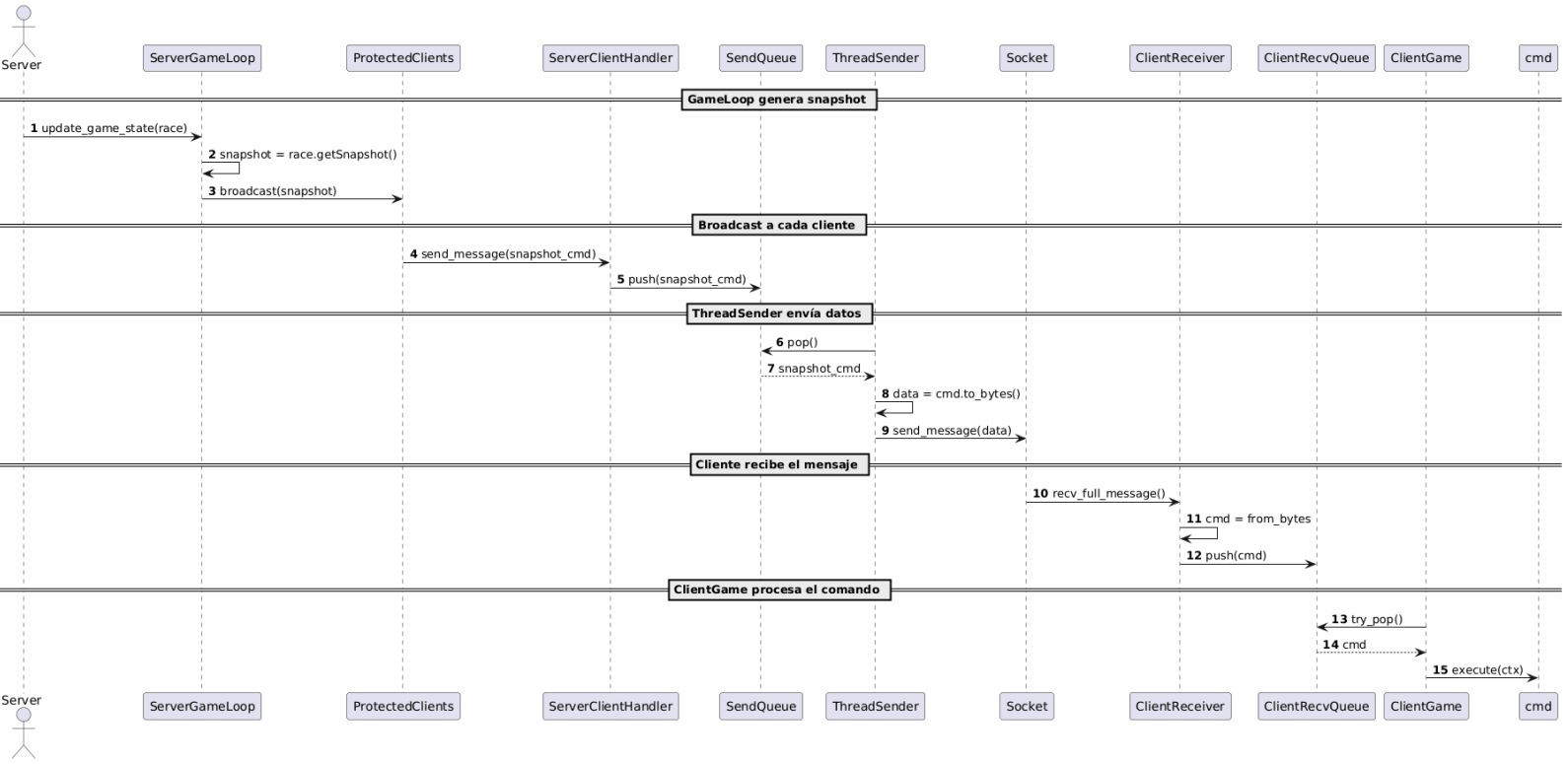
Lo primero es presentar un diagrama general de la arquitectura utilizada para el trabajo en general. Los diferentes hilos tanto del cliente como del servidor.



En el siguiente diagrama de secuencia se muestra el flujo completo de comunicación entre el cliente y el servidor durante el envío y procesamiento de un comando. En el lado del cliente, la interacción se inicia cuando el jugador solicita ejecutar una acción, la cual es gestionada por ClientGame y posteriormente enviada a través de ClientSession hacia la cola de envío (SendQueue), la cual almacena punteros de tipo ClientToServerCmd_Client*. El ClientSender extrae el comando, lo serializa y lo transmite utilizando el Protocol y este el Socket. Una vez en el servidor, el ServerProtocol recibe los datos y los entrega al ServerReceiver, que reconstruye el comando y lo introduce en la GameLoopQueue (Queue<ClientToServerCmd_Server*>). Finalmente, el ServerGameLoop recupera el comando desde la cola y lo ejecuta.



Por otro lado, el diagrama de secuencia a continuación describe el flujo completo mediante el cual el servidor genera y distribuye un snapshot del estado del juego hacia los clientes, así como la forma en que cada cliente lo recibe y procesa. El proceso comienza en el ServerGameLoop, que actualiza el estado de la carrera y construye un snapshot. Este snapshot es enviado a todos los clientes a través de ProtectedClients, el cual delega en cada ServerClientHandler la creación y encolado del comando correspondiente dentro de la SendQueue. Es importante destacar que esta cola utiliza std::shared_ptr<ServerToClientCmd_Server> para gestionar la memoria de los comandos compartidos que se broadcastean, de manera segura. El ThreadSender extrae el comando, lo serializa y lo transmite por el Socket. En el lado del cliente, el ClientReceiver recibe el mensaje completo, reconstruye el comando original y lo introduce en la ClientRecvQueue (Queue<ServerToClientCmd_Client*>). Finalmente, el ClientGame recupera el comando desde la cola y ejecuta la actualización del estado local.



Protocolo de comunicación

El sistema implementa un protocolo binario. La información se serializa directamente a una secuencia de bytes (std::vector<uint8_t>). La estructura fundamental de cada mensaje comienza invariablemente con un encabezado (header) de 1 byte (uint8_t), que actúa como identificador único del tipo de comando (por ejemplo, SNAPSHOT_COMMAND o MOVE_COMMAND).

La transformación de objetos a bytes y viceversa se realiza mediante utilidades de serialización (BufferUtils). El protocolo admite tipos de datos primitivos estrictamente tipados para asegurar la consistencia entre cliente y servidor. Se utilizan enteros sin signo de 8 bits para banderas booleanas (como isNPC o onBridge) y enumerados, enteros de 16 bits para longitudes de cadenas, enteros de 32 bits para identificadores de clientes (client_id), y flotantes (float) para datos de física como posición, velocidad y ángulo. Las cadenas de texto, como los nombres de usuario, se codifican precedidas por un uint16 que indica su longitud en bytes, permitiendo una lectura segura del buffer.

Para procesar los mensajes entrantes de manera dinámica, el sistema utiliza un Registro de Comandos basado en mapas hash (std::unordered_map). Tanto el cliente como el servidor mantienen un registro (ClientRegisteredCommands y ServerRegisteredCommands) que asocia cada byte de encabezado con una función lambda constructora. Este diseño implementa el patrón Factory Method: cuando llega un mensaje, el sistema extrae el primer byte, busca la función correspondiente en el registro e invoca el método estático from_bytes de la clase concreta (por ejemplo, ServerToClientSnapshot_Client::from_bytes), instanciando así el objeto polimórfico correcto sin necesidad de grandes bloques condicionales if/else.

Para que un comando sea considerado válido y pueda ejecutarse, debe cumplir con el contrato definido en las interfaces base abstractas (ClientToServerCmd_Server o ServerToClientCmd_Client). El ciclo de vida de un comando válido consta de tres etapas:

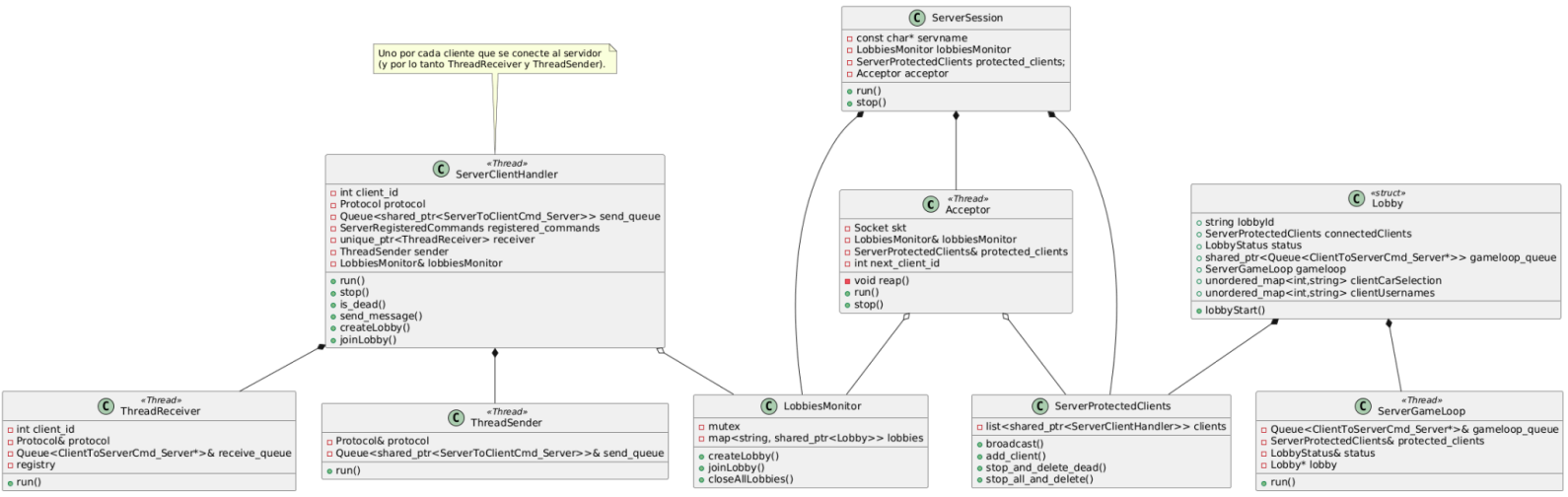
Validación de Existencia: El encabezado debe existir en el mapa de registro; de lo contrario, se lanza una excepción de "Unknown command header".

Validación de Integridad: Durante la deserialización (from_bytes), se verifica que el tamaño del vector de datos sea suficiente para contener todos los campos requeridos por ese comando específico, lanzando errores en caso de snapshots incompletos.

Ejecución Contextual: Una vez instanciado, el comando invoca su método virtual puro execute, el cual recibe un contexto específico (ClientContext o ServerContext). Esto permite que el comando interactúe de forma segura con los recursos del sistema, como modificar el estado del juego (ctx.game) o gestionar la lógica de la carrera (ctx.race), manteniendo un desacoplamiento total entre la red y la lógica de negocio.

Estructura de Servidor

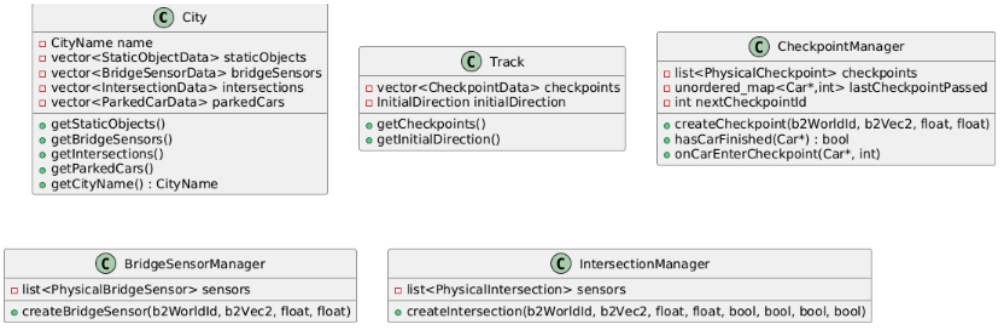
A continuación se presenta el diagrama de clases de las clases principales del servidor. Las mismas son las que manejan tanto el inicio del servidor como la conexión de múltiples clientes en simultáneo iniciando/uniéndose a los diferentes Lobbies que también puede haber múltiples de los mismos simultáneamente.



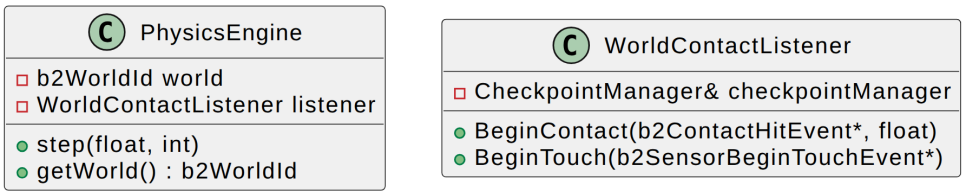
Por otro lado, el servidor se encarga de la simulación del juego. Para lo cual es necesario el sistema de físicas y lógica del juego. Su diseño combina Box2D como motor físico, una capa de objetos lógicos propios y una serie de managers especializados que permiten

detectar eventos, controlar autos, procesar colisiones y coordinar los elementos del entorno. La estructura central parte de la clase Race, que funciona como orquestador y representa una instancia completa de una carrera, desde la carga del circuito y la ciudad hasta la administración de jugadores, NPCs y los sensores físicos asociados al mapa.

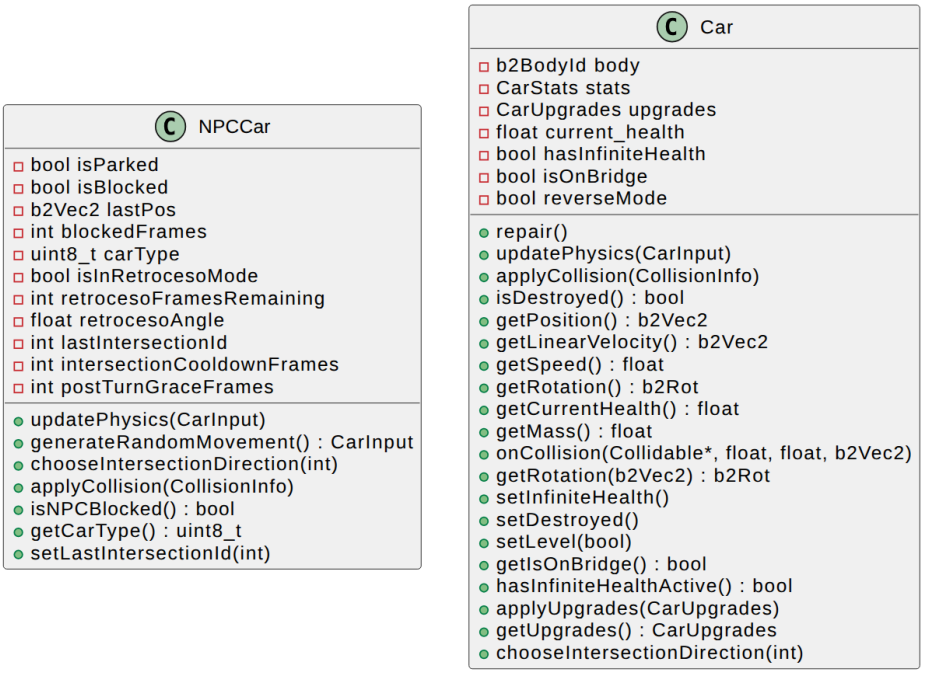
La pista de la carrera se carga a través del objeto Track, que provee información sobre checkpoints y dirección inicial de la misma. El entorno urbano proviene de City, que define objetos estáticos, sensores de puente, sensores de intersección y los lugares donde pueden aparecer los autos estacionados. Cada uno de estos elementos se traduce luego en cuerpos físicos de Box2D mediante managers especializados. El CheckpointManager crea y mantiene los cuerpos sensor para cada checkpoint del circuito y también registra el último checkpoint atravesado por cada jugador. Esto permite determinar cuándo un auto finaliza la carrera. El BridgeSensorManager genera sensores que detectan si un vehículo ingresa o sale de un puente. Esa información es utilizada por los autos para ajustar su comportamiento en colisiones. El IntersectionManager se encarga de los sensores que representan intersecciones de las calles donde los vehículos NPC deben decidir una dirección aleatoria para seguir circulando por el mapa. Todos los objetos estáticos se representan con la clase StaticObject, que implementa la interfaz Collidable y permite participar en el sistema de colisiones.



Las físicas las ejecuta dentro del PhysicsEngine, que crea el mundo Box2D, actualiza la simulación paso a paso y procesa eventos de contacto. Para esto utiliza un WorldContactListener, que intercepta dos tipos principales de eventos: colisiones entre cuerpos y activación de sensores. En las colisiones se distinguen autos, objetos estáticos y posibles impactos entre dos vehículos. El listener delega en cada objeto su propia reacción a la colisión, usando la lógica definida en la interfaz Collidable. Además aplica reglas específicas para autos: cálculo de fuerzas de impacto, determinación de ángulos de choque y reducciones de salud. En el caso de los sensores, se notifica al CheckpointManager si un auto cruza un checkpoint, lo que permite saber si un jugador completó el recorrido, mientras que los sensores de intersección y puente informan a los autos.

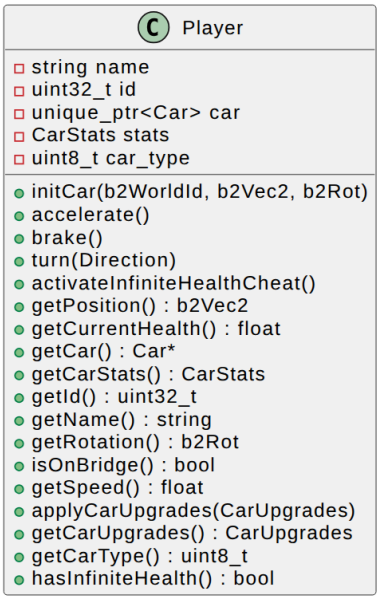


Los autos controlados por jugadores y NPCs comparten la implementación base Car. Cada instancia mantiene un cuerpo físico de Box2D, estadísticas, mejoras aplicadas y estado interno como salud, modo reversa o si se encuentra sobre un puente. La clase expone funciones para acelerar, frenar, girar, reparar y aplicar mejoras, así como métodos internos para limitar su velocidad según stats, manejar fuerzas aplicadas y procesar impactos. En colisiones, los autos calculan daños basados en la velocidad de aproximación y la masa del objeto involucrado. El estado de destrucción, vida infinita y comportamiento en puentes también se manejan desde esta clase base.

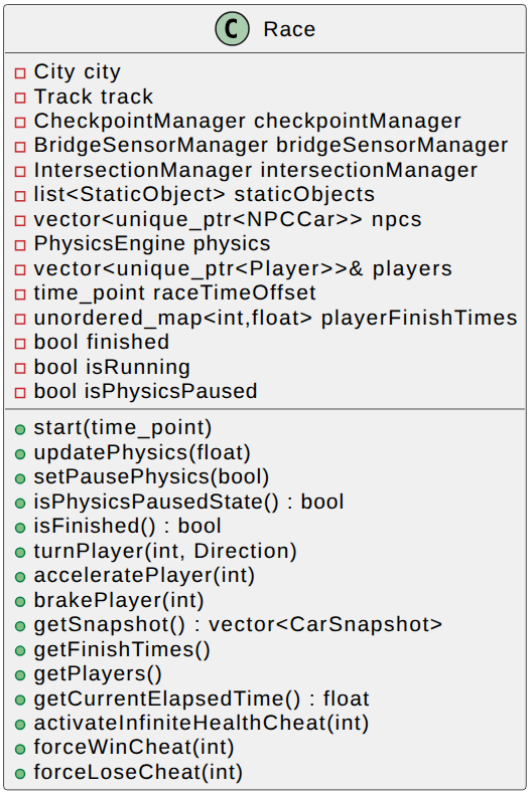


Los NPCs extienden Car con comportamiento autónomo. Implementan rutinas para detectar si están bloqueados, retroceder cuando quedan atrapados, decidir direcciones en intersecciones y generar movimientos pseudoaleatorios. También registran el último sensor de intersección utilizado para poder salir correctamente de maniobras de retroceso.

El servidor administra también a cada jugador a través de la clase Player, que encapsula la identidad del jugador, sus estadísticas, tipo de vehículo y una instancia de Car asociada. El Player sirve como interfaz de alto nivel para los comandos que envía el cliente, permitiendo acelerar, frenar, girar y activar cheats como vida infinita. El servidor se encarga de crear el auto al inicio de la carrera y reubicarlo según la pista.



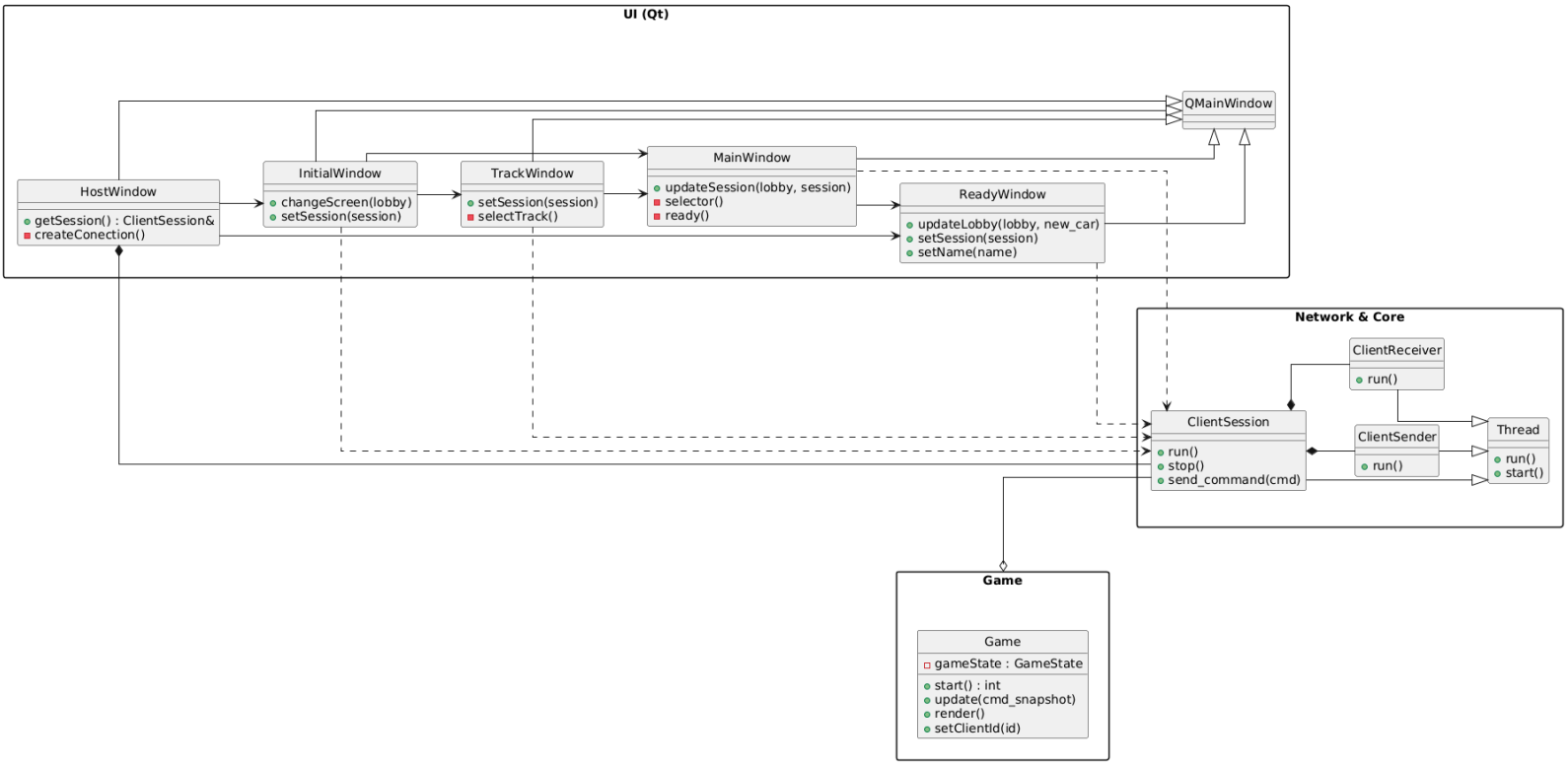
La clase Race integra todos estos componentes. Durante la inicialización carga la ciudad y el circuito, crea sensores, objetos estáticos, autos de jugadores y NPCs, define el mundo físico y prepara los estados de carrera como tiempos de inicio, pausa de simulación y finalización. En cada tick del servidor invoca updatePhysics, que coordina el avance de la simulación física y luego evalúa si algún jugador completó la carrera. La Race también mantiene el tiempo total transcurrido, los tiempos de llegada por jugador y expone snapshots consistentes del estado de todos los vehículos para ser enviados a los clientes.



El flujo completo consiste en construir el mundo con todos sus cuerpos físicos, vincular managers especializados para detección de interacciones, dejar que autos y NPCs actualicen su estado, resolver colisiones físicamente y aplicar reglas de juego que dependen de esas interacciones. La arquitectura separa claramente la responsabilidad entre física pura, lógica de colisiones, progresión de carrera y control de vehículos, permitiendo que el servidor modele un entorno dinámico, determinista y sincronizable para todos los clientes.

Estructura de Cliente

A continuación se presenta el diagrama de clases simplificado de las clases principales del cliente. En las mismas se maneja tanto la conexión con el servidor tanto para unirse a una nueva lobby, como para unirse a una existente, así como también la lógica de envío y recepción de comandos a través de los hilos sender y receiver.



La arquitectura del cliente se ha diseñado bajo un esquema modular dividido en tres paquetes principales: **Network & Core**, **UI (Qt)** y **Game**. Esta separación de responsabilidades permite desacoplar la lógica de red de la interfaz de usuario y del motor de renderizado.

En el paquete **Network & Core** residen los componentes troncales de la comunicación. Se implementa un modelo de concurrencia basado en hilos: un hilo para `ClientSender` (envío de comandos), otro para `ClientReceiver` (recepción de comandos) y el hilo principal de la `ClientSession`, que orquesta el ciclo de vida de la conexión.

El paquete **UI (Qt)** gestiona la interacción inicial con el usuario. A través de una serie de ventanas (`MainWindow`, `HostWindow`, `InitialWindow`), se abstrae la complejidad de la configuración, solicitando username y datos de conexión (IP y Puerto). Es en esta capa donde se define si el cliente se unirá a una partida existente o creará una nueva (Lobby), enviando la solicitud correspondiente al servidor a través de la cola de eventos.

Una vez finalizada la etapa de configuración en Qt y establecido el "handshake" con el servidor, el control se transfiere al paquete **Game**. La clase central, `Game`, asume el control del hilo principal para ejecutar el bucle de juego. Este módulo actúa como cliente del motor gráfico **SDL2pp**.

Para mantener la cohesión del diseño y evitar que la clase Game se convierta en una entidad monolítica, la responsabilidad del renderizado de ciertos elementos detallados a continuación y los efectos visuales se ha delegado en una serie de clases especializadas. Estas son instanciadas y orquestadas por el bucle principal de Game, el cual las alimenta con los snapshots de estado recibidos desde el servidor. A continuación se detalla la responsabilidad de cada componente:

Camera gestiona el viewport o área visible del mundo de juego. Su función principal es centrar la vista en el objeto seguido (el vehículo del jugador) mediante el método follow, ajustando las coordenadas x e y. Internamente, calcula el rectángulo de origen (SDL_Rect) para el renderizado, aplicando una lógica de "clamping" (restricción) que impide que la cámara muestre áreas fuera de los límites del mapa, asegurando que la vista se detenga al alcanzar los bordes del mundo.

HUD (Heads-Up Display) se encarga de renderizar la interfaz de usuario superpuesta utilizando la librería SDL_ttf. Esta clase recibe una estructura HUDData y formatea la información vital para el jugador mediante std::ostringstream: muestra la velocidad con precisión decimal, la salud como porcentaje, el progreso de checkpoints y el tiempo de carrera en formato MM:SS. Para garantizar la legibilidad del texto sobre cualquier fondo de juego, el HUD dibuja dinámicamente un rectángulo negro semitransparente con borde blanco detrás de las estadísticas antes de renderizar las texturas de texto.

Minimap procesa la transformación de coordenadas entre el servidor y la interfaz gráfica para generar una vista topográfica de la carrera. Utiliza factores de escala (scaleX, scaleY) para convertir la posición absoluta de los objetos en coordenadas relativas al minimap. Además de renderizar la textura de la pista, dibuja dinámicamente los checkpoints (distinguiendo la meta con un color diferente) y representa al jugador local mediante una flecha vectorial construida con primitivas geométricas (DrawLines), la cual rota en tiempo real según el ángulo de orientación del vehículo.

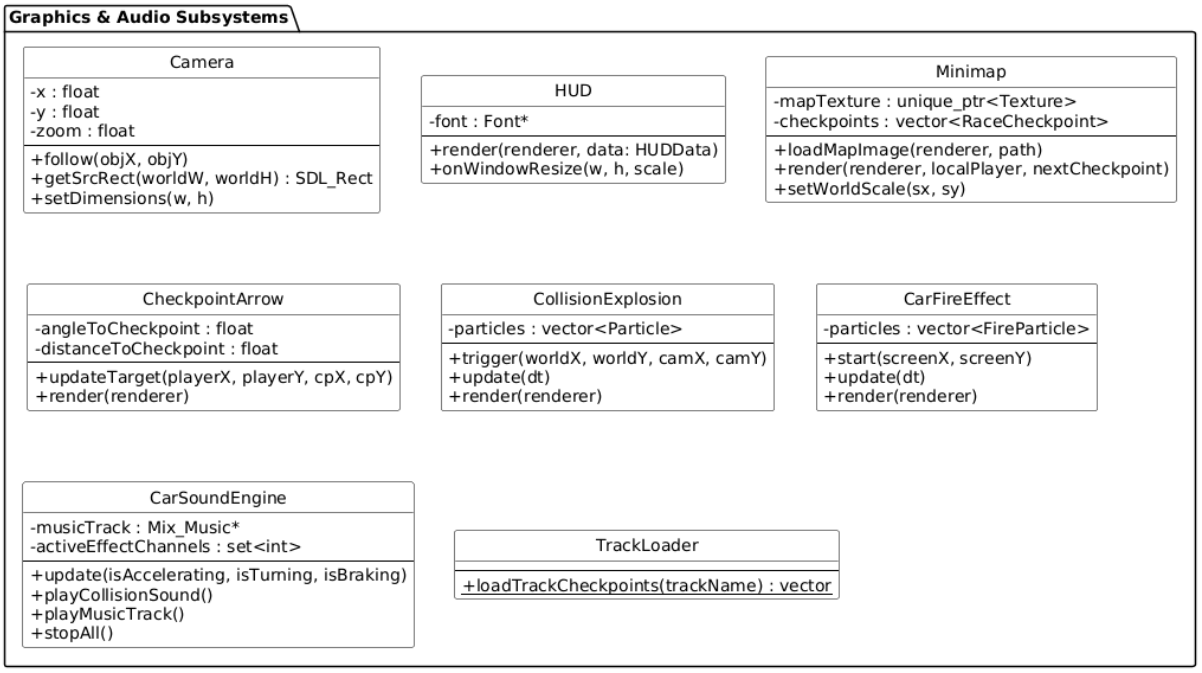
CollisionExplosion es un sistema de partículas que gestiona la retroalimentación visual de los impactos. Al activarse, genera múltiples partículas con velocidades y ángulos aleatorios que simulan una dispersión caótica. En su ciclo de actualización (update), aplica una gravedad simulada sobre la velocidad vertical de cada partícula y gestiona su ciclo de vida, desvaneciendo su color (alpha blending) progresivamente desde tonos brillantes hasta desaparecer, lo que otorga una sensación de peso y disipación a la explosión.

CarFireEffect maneja los efectos visuales de daño crítico y destrucción del vehículo. A diferencia de la explosión única, esta clase mantiene un emisor continuo (emissionTimer) que genera partículas simulando fuego y humo. Las partículas se inicializan con una paleta de colores cálidos (rojo, naranja, amarillo) y poseen una velocidad vertical negativa para simular el ascenso térmico del fuego, reduciendo su tamaño y velocidad horizontal a medida que consumen su tiempo de vida.

CheckpointArrow proporciona una ayuda visual de navegación esencial para el jugador. Mediante cálculos trigonométricos (std::atan2), determina el ángulo exacto entre la posición del jugador y el siguiente checkpoint objetivo. Renderiza una flecha estilizada en una posición fija de la pantalla que rota para señalar la dirección correcta, cambiando su estado visual si no hay un objetivo activo. El dibujo se realiza vectorialmente línea por línea, lo que permite un escalado nítido independientemente de la resolución de la ventana.

CarSoundEngine encapsula la gestión de audio utilizando la librería SDL_Mixer, administrando canales para música de fondo y efectos de sonido (SFX). Su método principal, update, monitorea el estado de las teclas (aceleración, frenado, giro) para reproducir bucles de sonido continuos o detenerlos según corresponda. Además, gestiona sonidos contextuales como colisiones, victorias o activación de trucos, implementando lógica para evitar la saturación de canales mediante temporizadores de enfriamiento (cooldowns).

TrackLoader funciona como una clase de utilidad estática diseñada para cargar los datos de los checkpoints para poder renderizarlos en la pantalla del cliente. Su método principal, loadTrackCheckpoints, utiliza la librería yaml-cpp para deserializar archivos de configuración externos ubicados en el directorio de activos. Este componente recorre la secuencia de nodos definidos en el archivo YAML para instanciar estructuras RaceCheckpoint, extrayendo atributos geométricos precisos como posición (x, y) y dimensiones (width, height). Adicionalmente, encapsula la lógica de determinación de meta, asignando automáticamente la bandera isFinish al último elemento de la secuencia.



Finalmente, la clase Game actúa como el núcleo integrador del cliente. Es la responsable de inicializar la ventana SDL y mantener el loop principal de Game, donde converge la lógica de renderización. En cada iteración del bucle, Game procesa la cola de recepción de la ClientSession para extraer los snapshots y actualizar el estado local de los vehículos, o ejecuta directamente el comando en caso de no ser un snapshot. Simultáneamente, captura los eventos de entrada del usuario (teclado) y los envía al servidor como comandos (ClientToServerMove o ClientToServerCheat). Además de coordinar a las clases gráficas mencionadas anteriormente, Game administra los estados globales de la aplicación, como las pantallas de fin de carrera, la tabla de posiciones acumulada y el menú de mejoras (upgrades), asegurando una transición fluida entre la competencia y las interfaces de gestión.