Taller de Programación I - 75.42
Micromachines

Manual Técnico

Índice

1. Requerimientos de Software **2**

2. Arquitectura general **2**
2.1. Librerias externas . . . 2
2.2. Diagrama de paquetes . . . 3

3. Server **4**
3.1. Proceso de creación de una partida . . . 5
3.2. Proceso de unión a una partida . . . 7
3.3. Durante la partida . . . 7
3.4. Envío y recepción de actualizaciones . . . 8
3.5. Modificadores . . . 9
3.5.1. Búsqueda y carga . . . 9
3.5.2. Ejecución . . . 9
3.5.3. Destrucción . . . 9

4. Common **10**
4.1. Protocolo . . . 10
4.2. Constantes . . . 10

5. Cliente **10**
5.1. Qt . . . 10
5.2. SDL2 . . . 13
5.3. Hilo que recibe del servidor . . . 14
5.4. Hilo que dibuja . . . 15
5.5. Hilo que espera teclas . . . 17
5.6. Bot . . . 18

1. Requerimientos de Software

Para compilar, ejecutar y depurar el programa se requiere tener instaladas las siguientes librerías:

- SDL2
- SDL_image
- SDL_mixer
- SDL_ttf
- Qt5
- ffmpeg
- Lua 5.3.4

La instalación de estas librerías se detalla en el Manual de Usuario.

2. Arquitectura general

El proyecto esta dividido en cuatro secciones principales:

- **Server:** El modulo *Server* se encarga del proceso de establecer múltiples conexiones con los clientes, separarlos correctamente en las partidas, además de realizar de ser el intermediario entre la comunicación de los jugadores con el modelo del juego.
- **Model:** El modulo *Model* es el encargado de ejecutar y procesar tanto las físicas como la lógica de una carrera. Cada clase que posee simula alguno de los objetos con los cuales el cliente puede interactuar o visualizar.
- **Client:** El modulo *Client* es la interfaz entre el jugador y los módulos *Server/Model*. Su funcion es recibir los comandos de un jugador, enviarlos al servidor para luego dibujar en pantalla la interpretación de la respuesta.
- **Common:** Si bien es el modulo mas corto, este modulo posee una serie de clases y/o definiciones que permiten la comunicación entre el servidor y el cliente. Establece el protocolo de envio, además de establecer los tipos de mensajes que el cliente puede procesar,

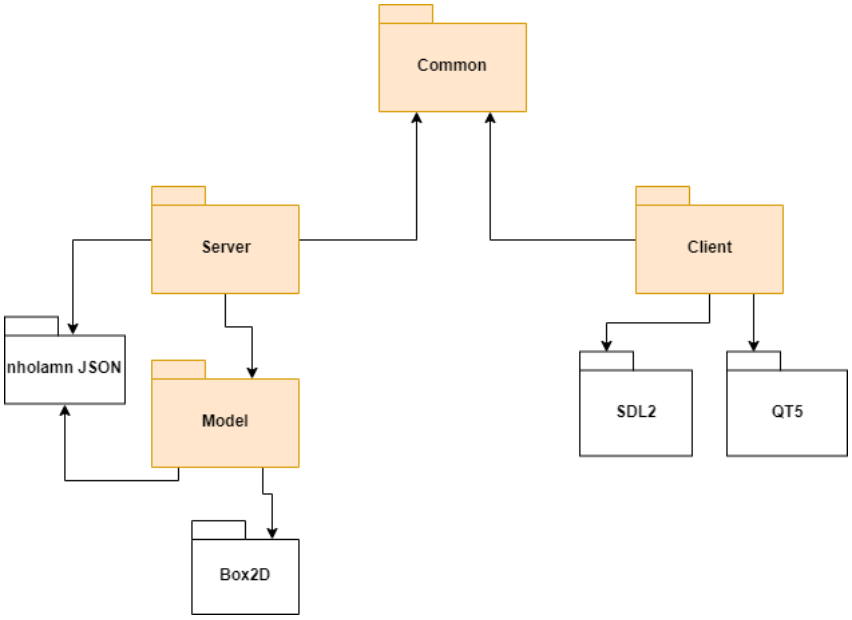
2.1. Librerias externas

Durante el proyecto se realizo el uso de tres bibliotecas externas. Para la interfaz gráfica, se utilizo la biblioteca *SDL2* junto con *QT5*.

Respecto a la simulación de las físicas involucradas en la carrera, se utilizo la biblioteca Box2D.

Por ultimo, se utilizo la biblioteca *nlohman JSON* para el parseo de los archivos de mapas y de configuraciones del servidor.

2.2. Diagrama de paquetes



El color naranja indica que son de implementación propia. El color blanco indica una biblioteca externa.

3. Server

El server esta compuesto por tres hilos principales, *ThreadAcceptor*, *ThreadPlayerLocator* y *ThreadMatchStarter*. A su vez, cada hilo lanza otros hilos para esperar respuestas del cliente (ya que son operaciones bloqueantes) permitiendo atender varios clientes simultáneamente.

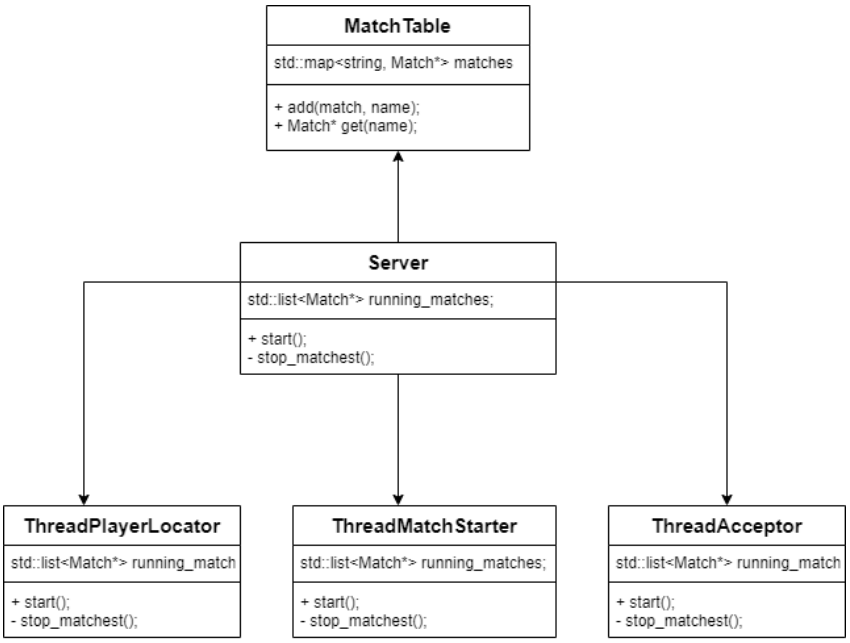


Figura 1: Diagrama general del servidor

En las secciones siguientes se procederá a explicar el funcionamiento de cada hilo en cada etapa, tanto en la creación de una nueva partida como en el proceso de unión a una partida ya existente, detallando las interacciones con otros objetos e hilos.

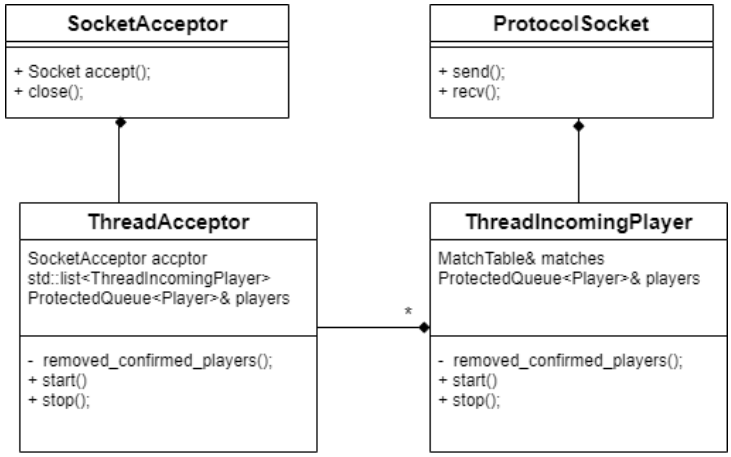


Figura 2: Estructura del ThreadAcceptor

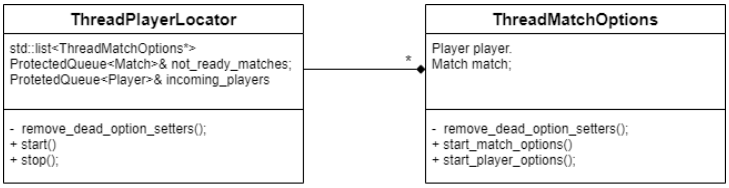


Figura 3: Estructura del ThreadPlayerLocator

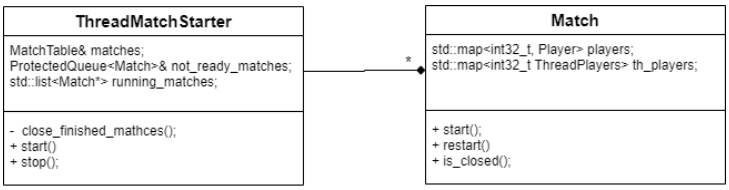


Figura 4: Estructura del ThreadMatchStarter

3.1. Proceso de creación de una partida

El primer contacto del jugador con el servidor es por medio del *ThreadAcceptor*. Este hilo se encarga de aceptar los nuevos jugadores y establecer la conexion creando un nuevo socket. Luego de creado el socket, un nuevo hilo (*ThreadIncomingPlayer*) es lanzado para esperar los siguientes comandos del jugador. En esta etapa, el jugador entrante puede elegir entre crear una partida o unirse a una. En ambos caso debe seleccionar un nombre de usuario con el cual se lo identificara durante el desarrollo de la carrera. Caso este elija crear una partida, deberá además elegir un nombre (que podrá ser aceptado o no caso exista una partida con ese nombre). Una vez finalizado el proceso de elección, este hilo crea un objeto de la clase *¨Player* con la información ingresada previamente y el socket. Esta clase se utilizara durante toda la ejecucion como la interfaz de comunicación con el cliente. Finalmente, se encola al objeto en una cola de jugadores entrantes, para que el *ThreadPlayerLocator* pueda ubicarlo donde corresponda.

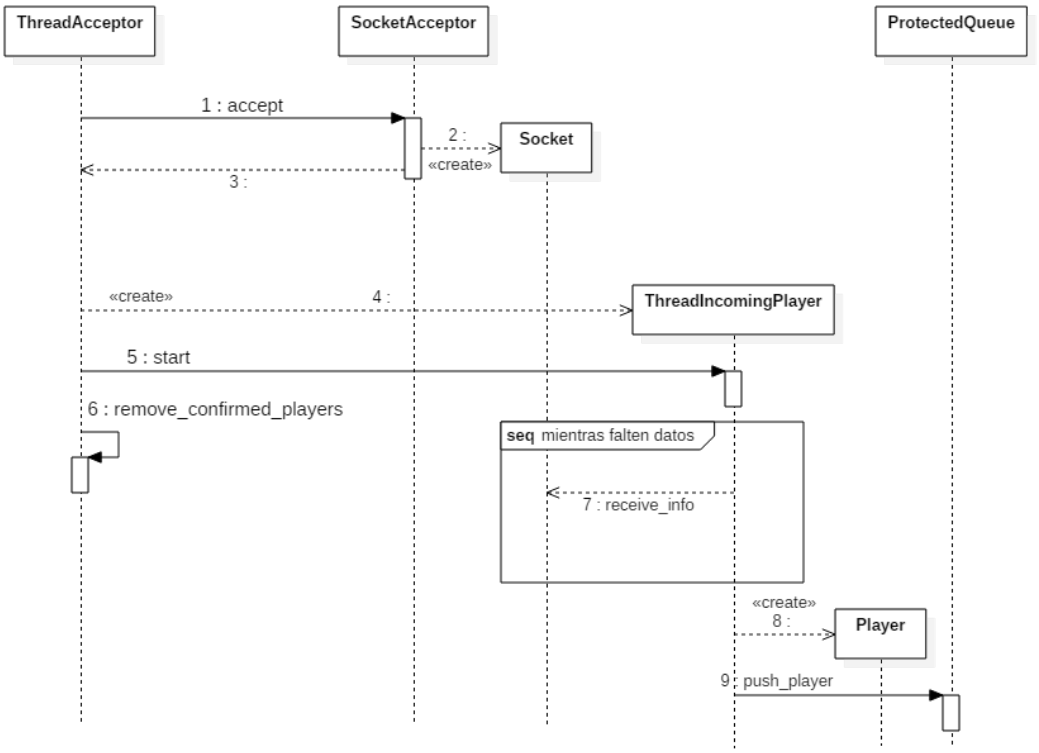


Figura 5: Secuencia de aceptar un nuevo cliente

La funcion del *ThreadPlayerLocator* sera ubicar al jugador según las opciones elegidas previamente. Si el jugador ingreso en modo çreación de partida", este hilo se encargará de crear la partida, representada con un objeto de la clase *Match* y lanzar otro hilo

ThreadMatchOptions, el cual esperará a que el cliente envíe un flag para empezar la partida. Si bien en la versión actual solo espera la señal para comenzar la carrera, el diseño de este hilo está pensado de manera que en versiones posteriores se puedan agregar opciones a la creación de la partida, como elegir otro mapa. Una vez recibida la señal para comenzar la partida, se agrega el creador a la partida y se enconla el partido con todos los jugadores a una cola se partidos listos para ser iniciados. En este momento entra en juego el hilo *ThreadMatchStarter*.

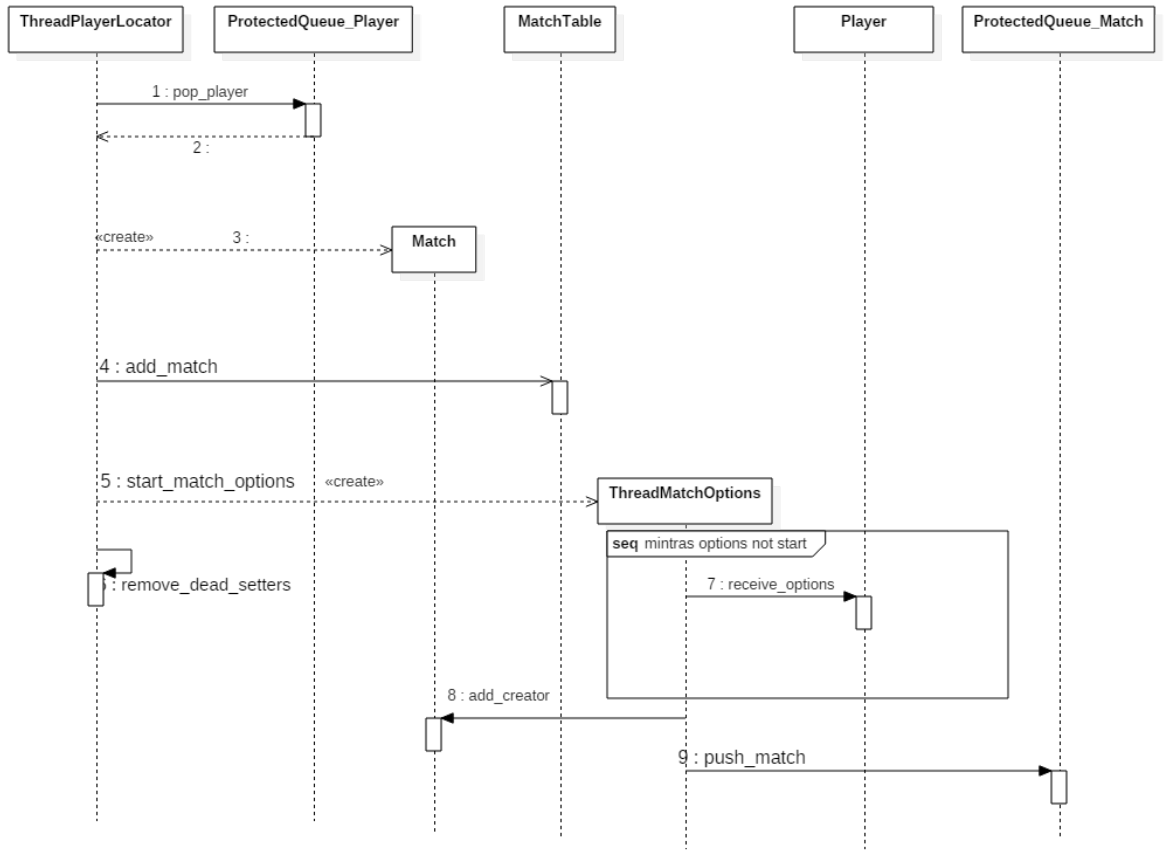


Figura 6: Secuencia de crear una nueva partida

El hilo *ThreadMatchStarter* se encarga de manejar las partidas en ejecucion. Este desencola una partida, la inicia (cada partida corre en un hilo por separado) y se la guarda en una lista de partidas en ejecucion. Luego de cada partida aceptada, recorre la lista de partidas en ejecucion controlando si alguna ya finalizo, cerrando y liberando la memoria de las partidas ya finalizadas.

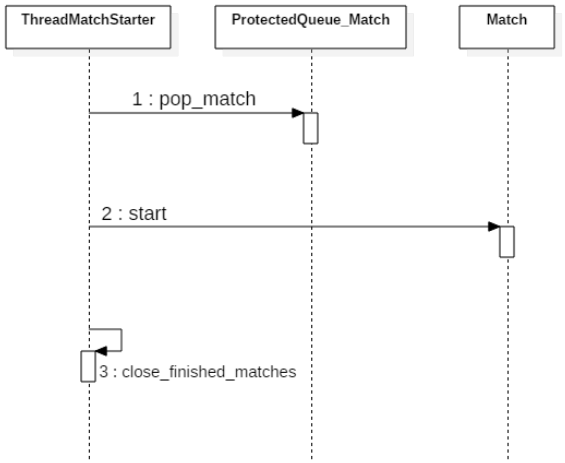


Figura 7: Secuencia de empezar una nueva partida

3.2. Proceso de unión a una partida

El proceso de unión a una partida ya creada, comienza de igual manera que la creación de una partida. Las diferencias comienzan cuando se llega al hilo *ThreadPlayerLocator*. En este caso, este hilo se encargara de agregar el jugador a la partida ya creada. Aprovechando la existencia del hilo *ThreadMatchOptions*, se realizo un diseño de manera que al igual sucede con la creación de una partida, sea posible en un futuro implementar opciones para el jugador entrante.

Luego de que el jugador es agregado a la partida, se sigue el mismo flujo que en el caso de la creación de una nueva partida.

3.3. Durante la partida

Para el modelado de una partida existes cuatro clases principales.

La primer clase es *Match*. Esta clase se crea desde el momento en que un jugador ingresa en modo crear partida y vive hasta el final del juego. Se encarga de administrar todas las cuestiones que no están relacionas a la simulación de la carrera, como recibir las actualizaciones de los jugadores y distribuir las actualizaciones de la carrera correctamente. También se encarga de validar constantemente los jugadores conectados y manejar los casos de desconexión de jugadores durante la carrera.

La segunda clase es *Race*. A diferencia de lo que hace la clase *Match*, se encarga de actualizar el modelo con las actualizaciones que le entrega la clase *Match*. Actualiza cada vehículo, los objetos dinámicos, genera los modificadores, actualiza las vueltas y las posiciones. Todos esos cambios son informados al cliente cuando *Match* se los solicita.

En tercer lugar, esta la clase *Player*. Al igual que se utilizaba durante en proceso de unión/creación de una partida, esta clase sirve como interfaz para enviar y recibir las actualizaciones.

La ultima clase es *RacingTrack*. Esta clase se encarga de manejar toda la estructura del mundo. Durante el cargado del mapa, se le indican las posiciones de la pista, los objetos estáticos, los objetos dinámicos, las posiciones de inicio y la linea de llegada. Además administra la clase b2World (que es la clase que utiliza la librería Box2D para permitir que todos los objetos interactúen entre si).

3.4. Envío y recepción de actualizaciones

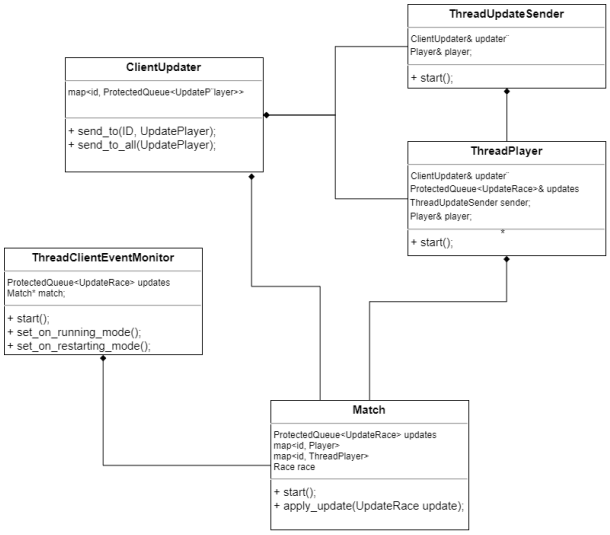


Figura 8: Estructura del sistema de actualizaciones

Durante la ejecucion de la partida, las actualizaciones se reciben mediante la clase *Player*. Cada jugador posee asociado un hilo *ThreadPlayer*, que a su vez posee una referencia a la unica cola de actualizaciones (esta cola es una cola bloqueante) donde encola las nuevas actualizaciones recibidas. Paralelamente, otro hilo *ThreadClientEventMonitor* desencola las actualizaciones y se las aplica a *Match*. Según el estado de la partida, la actualización se redirección a dos lugares diferentes.

Si la carrera no finalizo, se aplica la actualización *Race*. Si la carrera ya finalizo, se procesa la actualización en la la misma clase *Match* para interpretar el comando de reiniciar o salir de la carrera.

Respecto al envio de actualizaciones, cada jugador posee una cola de actualizaciones asociada. Todas las colas son manejas por la clase *ClientUpdater*. A su vez, existe un hilo para por cada jugador, encargado de realizar los envíos, el *ThreadUpdateSender*. Cada objeto de la pista (podios, contador de vuelta, etc) recibe una vez por iteración el *ClientUpdater* para crear el mensaje de actualización indicado.

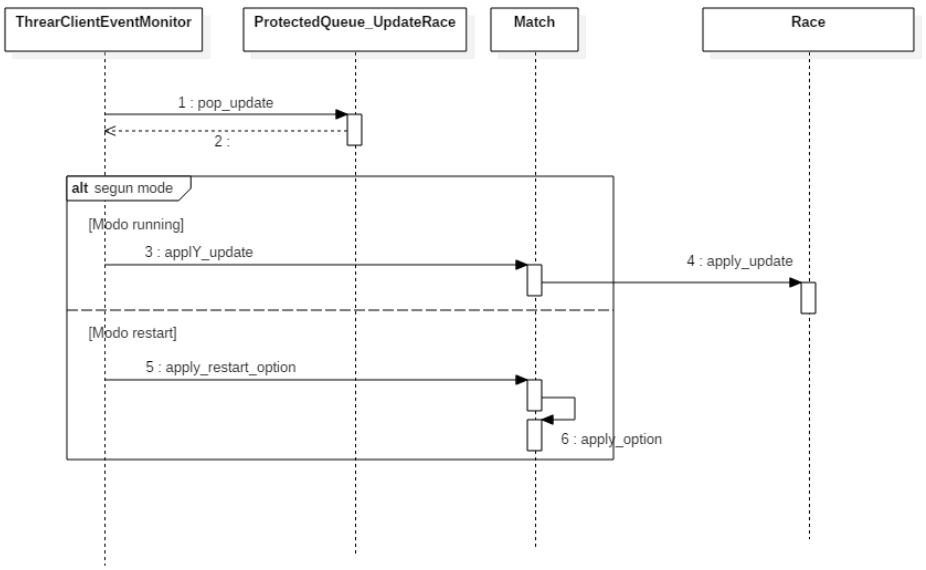


Figura 9: Estructura del sistema de actualizaciones

3.5. Modificadores

Los modificadores nos dan la posibilidad de modificar las reglas del juego sin necesidad de modificar el código ya implementado del mismo, esto se logra a partir de la implementación de librerías dinámicas las cuales deben ser guardadas en un lugar especifico. Para lograr el acople con estas librerías dinámicas, se implemento la clase PluginsManager la cual utiliza la libreria dlfcn.

3.5.1. Búsqueda y carga

El método load_plugins se encarga de recorrer todos los archivos que se encuentre en cierta carpeta, cuya ruta ser recibe por parámetro al inicializar la clase. En el momento que recorre todos los archivos, trata de cargar las librerías en memoria utilizando la función dlopen y guardando el void* que apunta a esta librería en un vector. Luego se trata de llamar a la función de inicialización del plugin el cual debe devolver otro void* y también es guardado en otro vector.

3.5.2. Ejecución

La ejecución de los plugins encontrados se realiza 3 veces por segundo. Para realizar la ejecución de los plugins se recorre el vector donde están guardados y se llama la función de ejecucion del plugin, pasándole como parámetro el void* que nos devolvió la función de inicialización y un DTO que representa la información de la carrera. Una vez que todos los plugins son ejecutados, se llama al metodo apply_plugin el cual aplica los cambios realizados en el DTO a través del plugin.

3.5.3. Destrucción

En el destructor de la clase PluginsManager, es necesario llamar a todas las funciones destructoras de los plugins, de esta forma en caso de que los mismos hayan solicitado memoria podrán liberarla. Es de suma importancia que la memoria pedida por los plugins sea liberada por ellos mismos, ya que de lo contrario se perdera memoria.

4. Common

Este módulo contiene todo lo que se usa tanto desde el cliente como desde el servidor.

- **Thread**: Es una clase abstracta que representa un hilo de ejecución, contiene un metodo `run()` abstracto que podemos redefinir en las clases hijas, el cual se va a correr en un hilo aparte. Entonces cuando llamamos al método `start()` se lanza un hilo que ejecuta run.
- **Socket**: Encapsula a un socket de C. Contiene métodos para conectarse, enviar y recibir cadenas de caracteres de longitud fija.
- **ProtectedQueue**: Es una cola bloqueante thread-safe con buffer fijo.

4.1. Protocolo

El protocolo de comunicación se encuentra implementado en la clase **ProtocolSocket**, la cual utiliza un **Socket**. Esta clase permite enviar y recibir distintos tipos de dato a través del socket, puede ser un byte, un entero, un string o un vector de enteros. Durante el ingreso a la partida se envían y reciben datos mediante todos estos métodos ya que en cada momento se sabe que se desea recibir. En cambio durante la partida, lo que se hace es recibir siempre vectores de enteros, que siguen la sintaxis:

```
tipo_de_mensaje, parametro_1, parametro_2, ..., parametro_n
```

Cada mensaje posible tiene un cantidad de parámetros fija. En ciertos tipos de mensaje va a ser necesario recibir otro dato, como por ejemplo cuando se va a recibir el nombre de un jugador que terminó la carrera, en ese caso se hace otro `receive()` para recibir el string.

4.2. Constantes

En este módulo tambien se encuentran los archivos que definen constantes compartidas por el cliente y el servidor:

- **EntityType.h**: IDs de los tipos de entidades, que puede ser coche, rueda, entidad dinámica, entidad estática, modificador de boost, etc.
- **MsgType.h**: IDs de los tipos de mensaje que puede enviar el cliente al servidor y viceversa.
- **Key.h**: IDs de las teclas que puede enviar el cliente al servidor para controlar el coche.
- **Sizes.h**: Constantes numericas para los tamaños de las cosas, no solo objetos fisicos.

5. Cliente

El cliente del juego consta de dos etapas. La primera hecha en Qt y la segunda hecha en SDL2.

5.1. Qt

Para realizar el ingreso a la partida se utilizo QT. Para esto al iniciar el cliente se inicializa una instancia de la clase ViewManager que organiza las ejecuciones de las vistas. Esta clase también se ocupa de aplicar la hoja de estilos a la QAplication la cual es un atributo de esta clase.

La entrada al juego consta de dos vistas, cada una de las vistas esta implementada como una clase:

- **ConnectView:** Se encarga de realizar la conexion del socket con el servidor.
- **MenuWindow:** Es la vista principal la cual se encarga de enviar las configuraciones al servidor, para crear una partida o unirse a una ya existente.

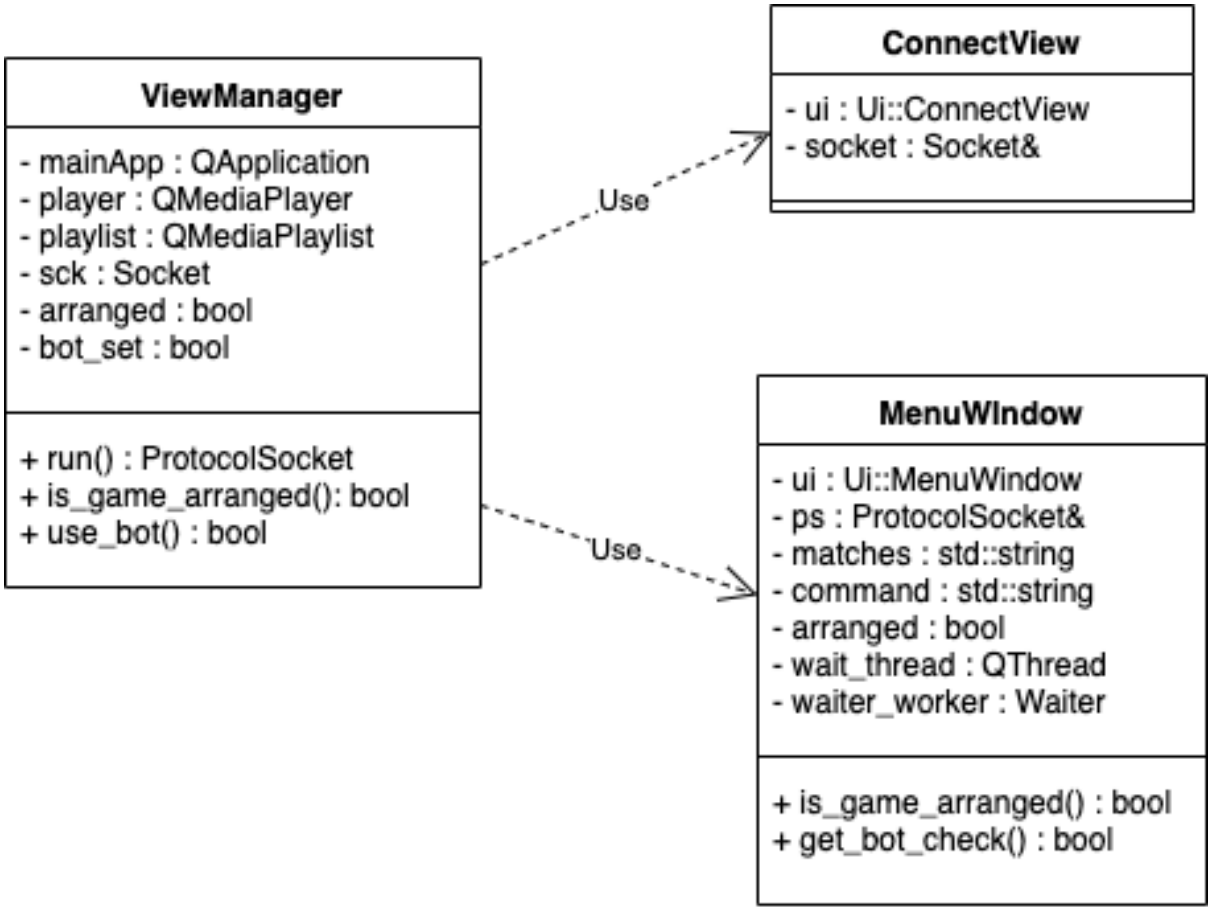


Figura 10: Vemos como la clase ViewManager usa las clases MenuWindow y ConnectionView

Una vez que se inicializa ViewManager, se llama al método run de esta clase. Este método se encarga de mostrar la vista de conexión, una vez que la vista de conexión es cerrada, si la conexión se realizo exitosamente, se abre la vista del menú, en caso de que la conexión no se haya realizado, el cliente se cierra ordenadamente. Una vez que la vista del menú esta abierta, el usuario puede ingresar a la vista donde se crea una partida o donde se une a una partida ya creada. En ambos casos, la clase ViewManager se queda esperando que la vista se cierre. Si el arreglo de la partida pudo ser realizado correctamente, se inicia el juego, en caso contrario el cliente se cierra. Una vez que el método termina en alguno de sus puntos, el mismo devuelve por movimiento una instancia de la clase ProtocolSocket la cual, en caso de haberse desarrollado correctamente los pasos anteriores. El cliente utilizara para comunicarse con el server y el mismo pasara a ser su responsabilidad.

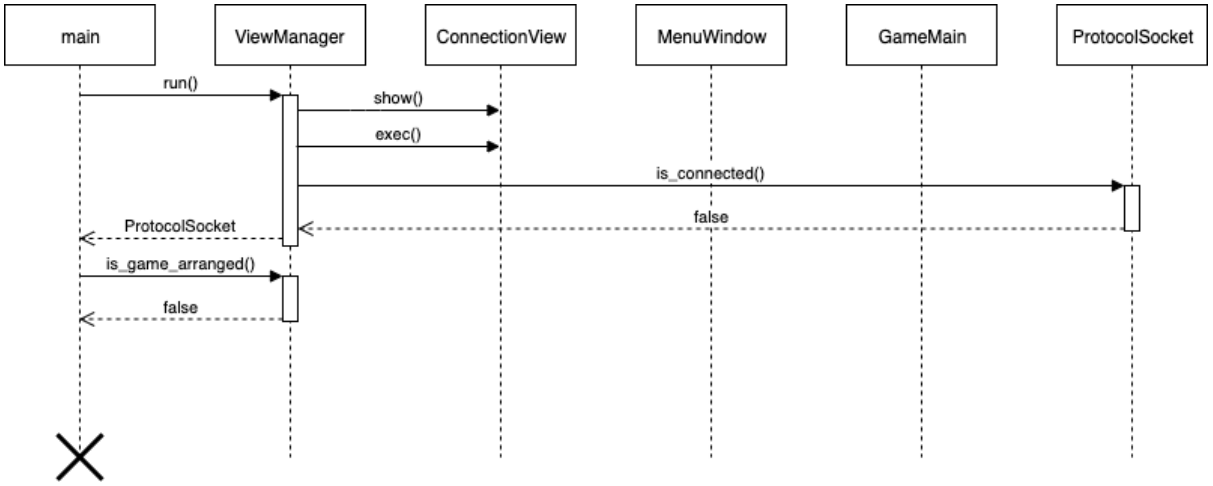


Figura 11: El método run de ViewManager corta su ejecución ya que el socket no esta conectado

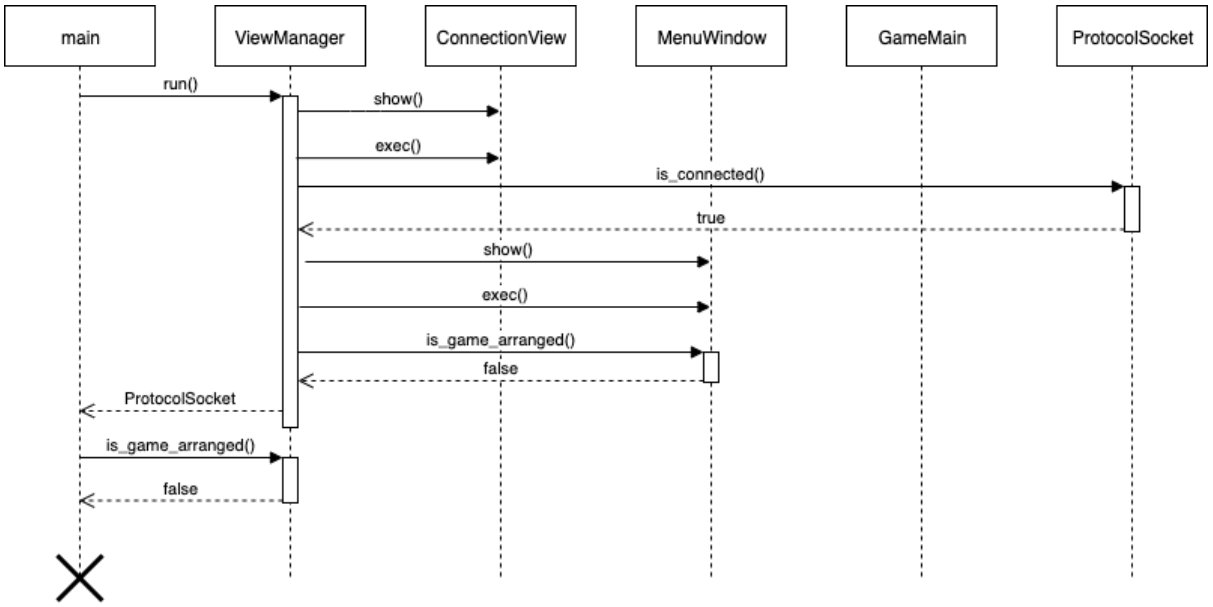


Figura 12: El método run de ViewManager corta su ejecución ya que el juego no logro ser configurado correctamente

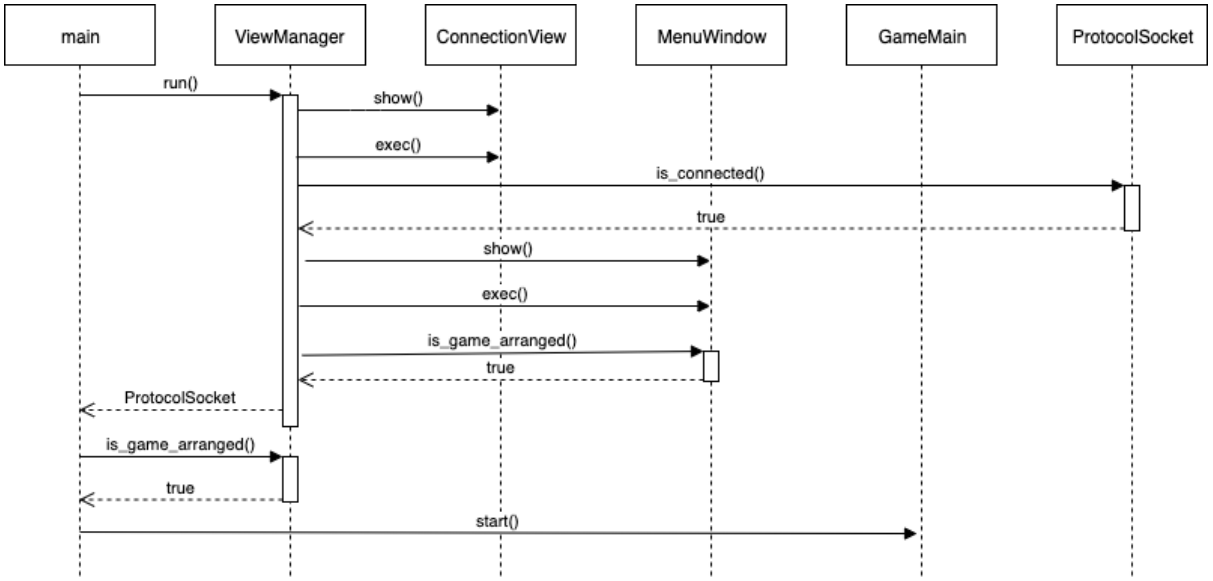


Figura 13: La configuracion de la partida es correcta y el juego se ejecuta

5.2. SDL2

Cuando finaliza la etapa de Qt, a continuación se instancia un GameMain, que se va a encargar de encapsular toda la etapa del juego que funciona en SDL2. Instancia un objeto **Scene** y un objeto **ProtectedQueue**. Se encarga de lanzar los Threads que se encargan de controlar el juego, los cuales son:

- **ThreadDrawer**: Se encarga de dibujar el juego a intervalos regulares de 1/60 segundos (60 FPS) (Si la potencia de la computadora no lo permite será más lento).
- **ThreadKeyMonitor**: Espera un evento de teclado por parte del usuario.
- **ThreadReceiver**: Espera un mensaje del servidor.
- **ThreadSender**: Espera en la **ProtectedQueue** antes mencionada a que llegue un mensaje y lo envía al servidor.
- **ThreadBot**: Cada cierto tiempo envia instrucciones al servidor para mover el vehiculo segun la informacion que se le provee.

El recurso protegido al que accederán los diferentes hilos es **Scene** (a excepción de ThreadSender y ThreadBot), el cual contiene todo lo necesario para manejar la lógica del juego:

- **Scenario**: Encapsula a los elementos que conforman el juego, ya sea el mapa, hud, elementos dinamicos, efectos de pantalla, vehiculos, etc.
- **Camera**: Contiene la lógica para dibujar sobre la pantalla proveyendo una interfaz genérica con ese propósito, puede dibujar tanto elementos estáticos como elementos que dependen de la posición, estos últimos los dibuja si se encuentran en el rango de visión.
- **ProtectedQueue**: Se utiliza cuando sea necesario comunicarse con ThreadSender para enviar mensajes al servidor.
- **Menu**: Una clase abstracta que puede ser diferentes menúes concretos, encapsula la lógica de qué hacer frente a las teclas que se reciben. Por ejemplo: estando en el menú de pausa no se enviarán las teclas de manejo del coche al servidor, sino que se manejará el caso de un ESC o un Q para salir del menú, o salir del juego, respectivamente.

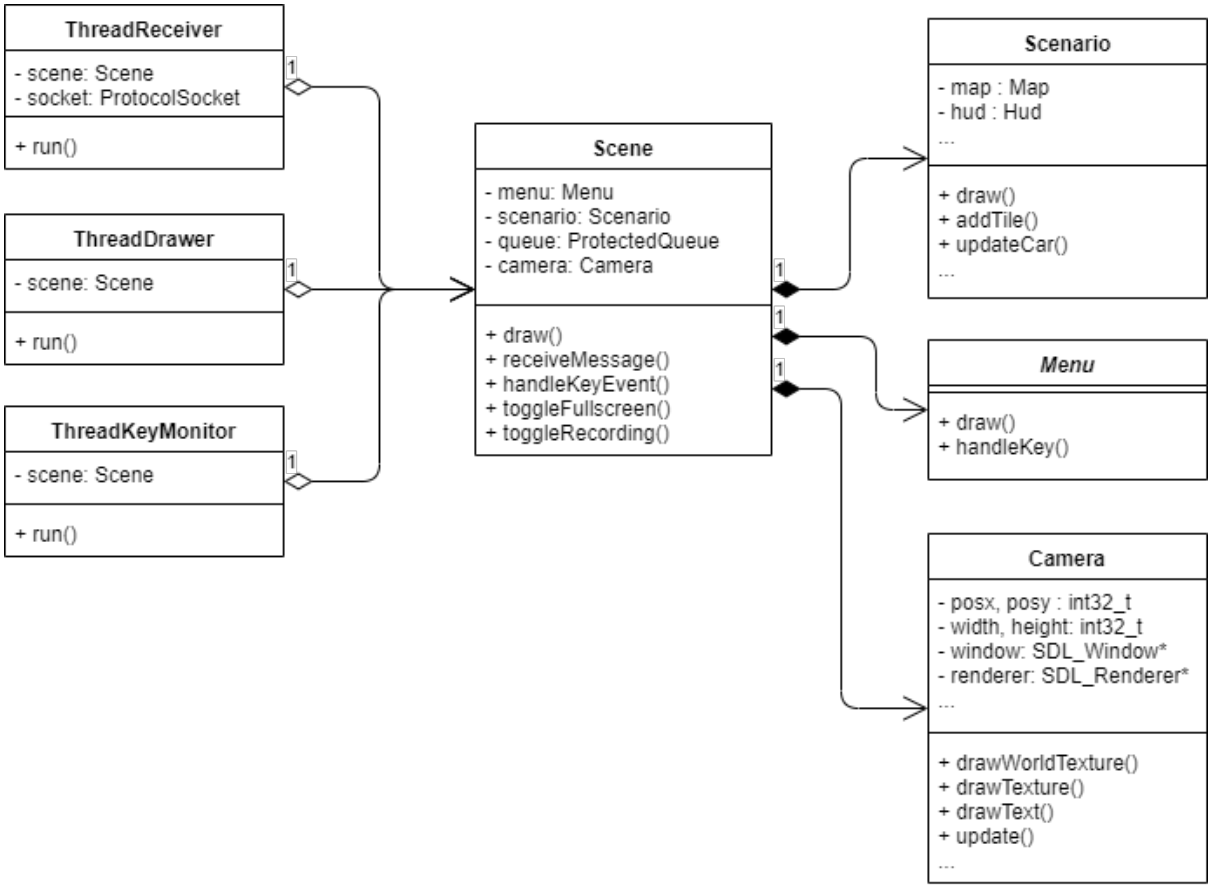


Figura 14: Vemos como *Scene* protege a los recursos compartidos

5.3. Hilo que recibe del servidor

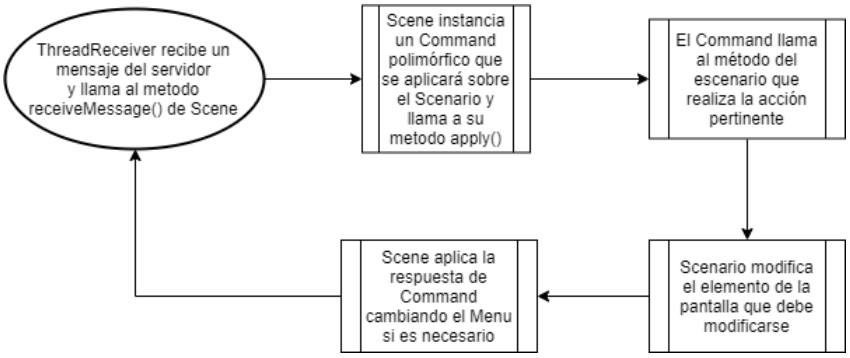


Figura 15: Ciclo que realiza el hilo que recibe mensajes del servidor

Command es una clase abstracta que implementa el patron command, que actúa sobre Screnario, sus clases hijas son comandos concretos que llaman un método especifico de Scenario.

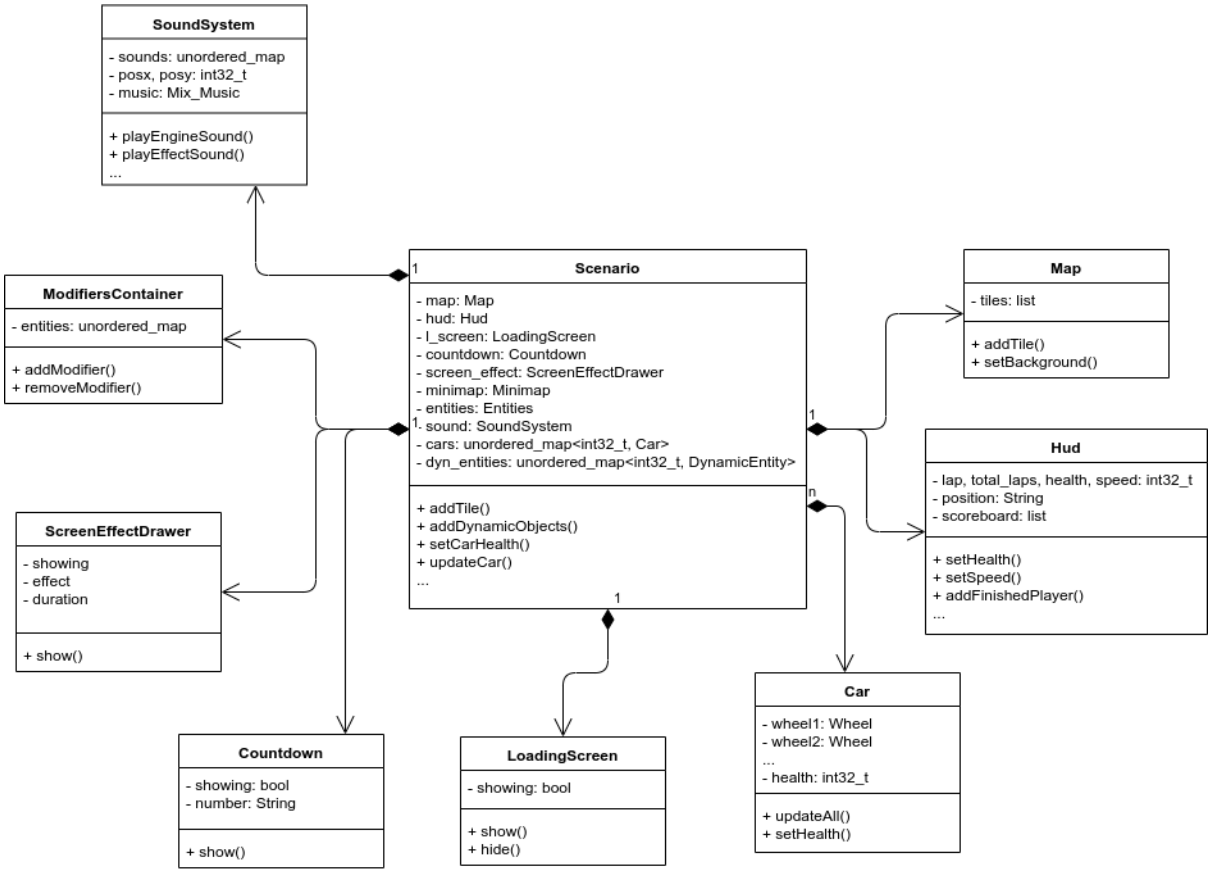


Figura 16: Cuando Command llama alguno de sus métodos el Scenario llama a los metodos de los elementos del juego para actualizarlos

Scenario también contiene el sistema de sonido, en una instancia de la clase **SoundSystem**, cuando actualiza los elementos o bien les envía este SoundSystem para que reproduzcan los sonidos, o bien lo hace el mismo si es un sonido que no depende de los elementos en sí.

5.4. Hilo que dibuja

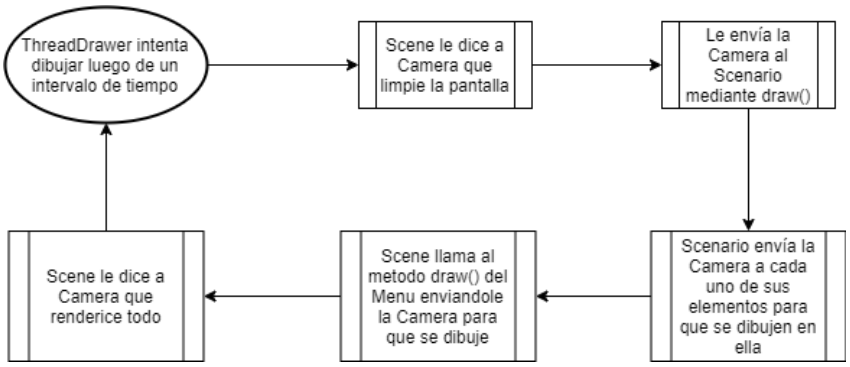


Figura 17: Ciclo que realiza el hilo que dibuja

Scene contiene una instancia de Camera, la cual es enviada al Scenario mediante el método `draw()`, luego Scenario la envía a cada uno de sus elementos, los cuales saben dibujarse utilizando los métodos genéricos de la Camara (ver los elementos de Scenario en la figura 16).

Como se mencionó antes, Camera contiene métodos genéricos para dibujar elementos en la pantalla, ya sea relativo a la posición de la misma, o estáticos (en un punto de la pantalla que recibe por parámetro).

Delega el manejo de las texturas a **TextureFactory** y el dibujo de texto a **TextDrawer**.

Para la grabación de partidas se delega la responsabilidad a **ScreenRecorder**.

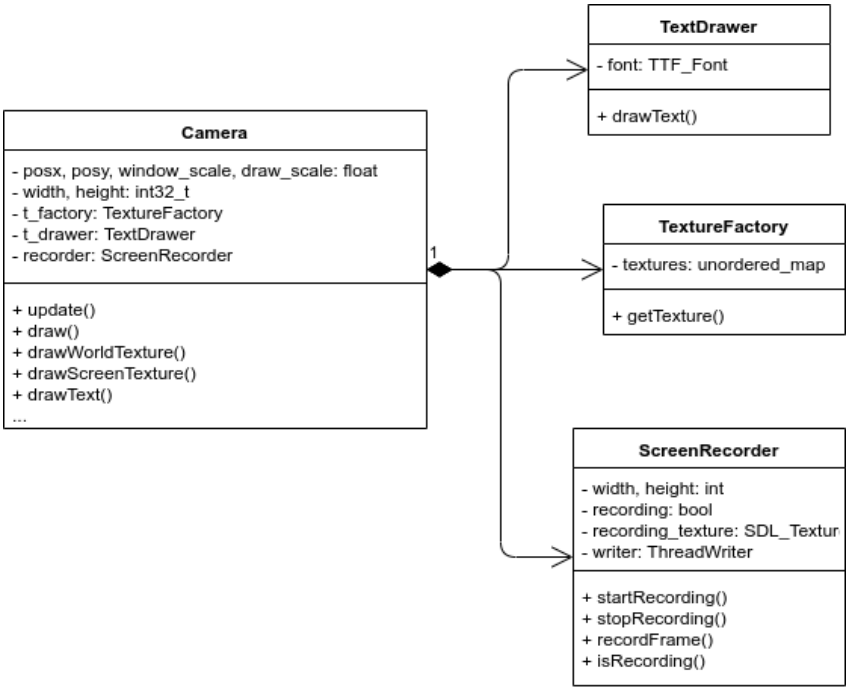


Figura 18: Diagrama de Camera

ScreenRecorder lanza un hilo que se queda esperando a que le lleguen frames para escribir a disco. Cuando se comienza a grabar, se llama al método `ThreadWriter::setup()` para que configure el formato de salida. Cuando se termina de grabar se llama a `ThreadWriter::saveVideo()` para escribir los bytes finales del video y cerrarlo. Durante la grabación, se llama en cada frame al método `ScreenRecorder::recordFrame()` que pone en una cola la informacion del frame, luego ThreadWriter desacola la informacion y la pasa a disco usando un **OutputFormat**.

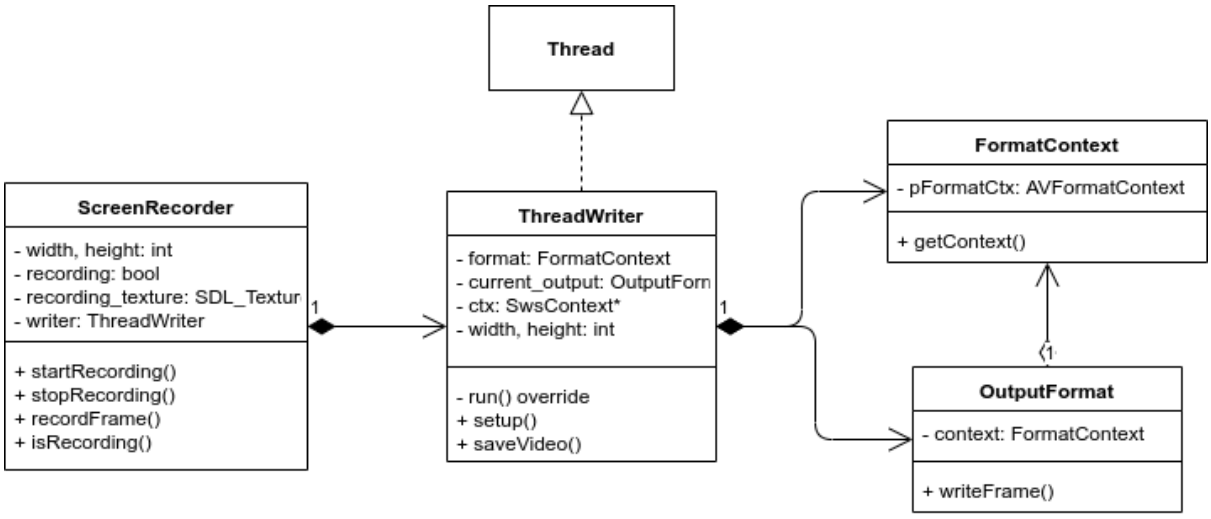


Figura 19: Diagrama del grabador de pantalla

5.5. Hilo que espera teclas

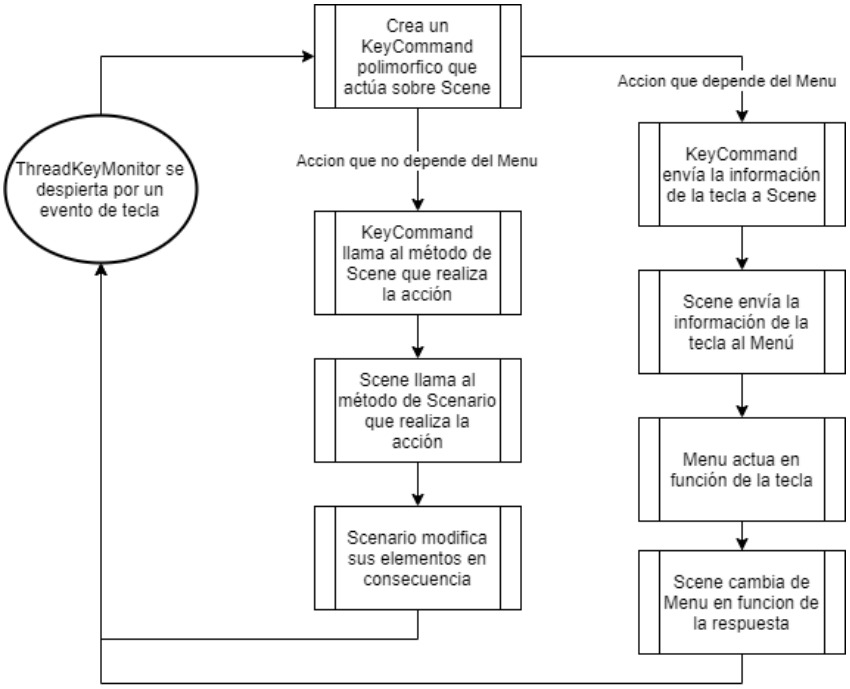


Figura 20: Ciclo que realiza el hilo que espera teclas

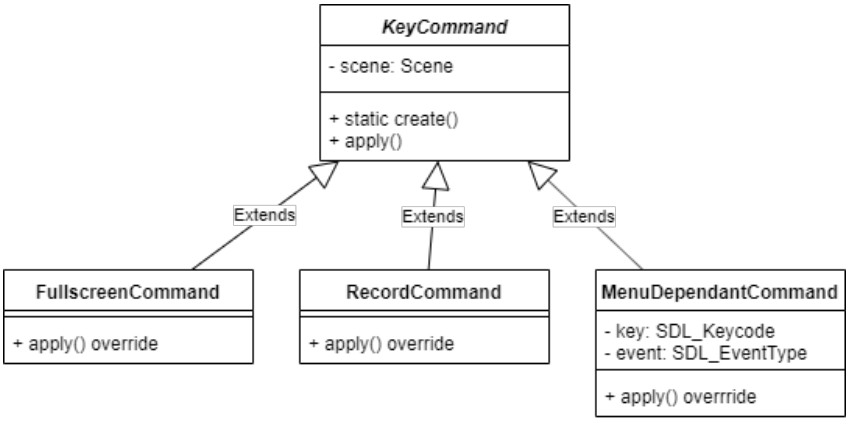


Figura 21: Las clases FullscreenCommand y RecordCommand van a llamar al método *Scene::toggleFullscreen()* y *Scene::toggleRecording()*, respectivamente. Mientras que MenuDependantCommand llama a *Scene::handleKeyEvent()* para que luego se envie la tecla a Menu. Así se distingue el caso de que la acción dependa del Menu o no.

Como se mencionó anteriormente, la clase **Menu** es una clase abstracta cuyas clases concretas encapsulan la manera de responder ante las teclas. Se utiliza en un patron *State*, estar en un menu, o no estarlo representa un estado, y ese estado cambia en función de ciertas teclas.

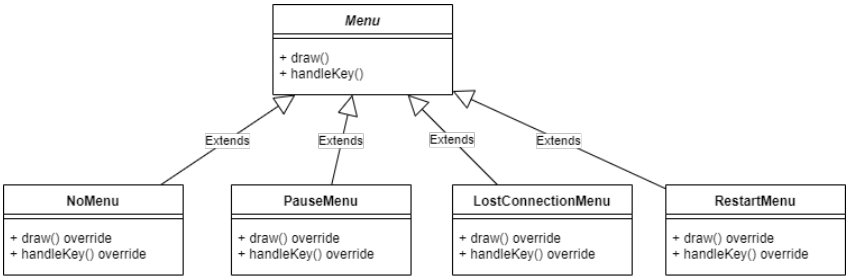


Figura 22: El método *handleKey* recibe la cola de envios para enviar al servidor en caso de ser necesario, ya sea teclas de control del auto, o un mensaje para reiniciar la carrera (en el caso de RestartMenu)

5.6. Bot

El bot encargado de manejar el auto, en caso de que el cliente lo deseara, es un script escrito en lua. Para la correcta comunicación con este script se implemento la clase Bot, la cual en su constructor recibe por parámetro una referencia a la cola protegida donde serian encolados los comandos recibidos por teclado en caso de que el jugador sea el que controla el auto. De esta forma el bot puede encolar los comandos necesarios para controlar el auto. Para lograr que el script decida que comandos encolar es necesario que se llame al metodo execute cada un intervalo fijo de tiempo, para cumplir con esta tarea implementamos un hilo cuya unica responsabilidad es llamar a este método.

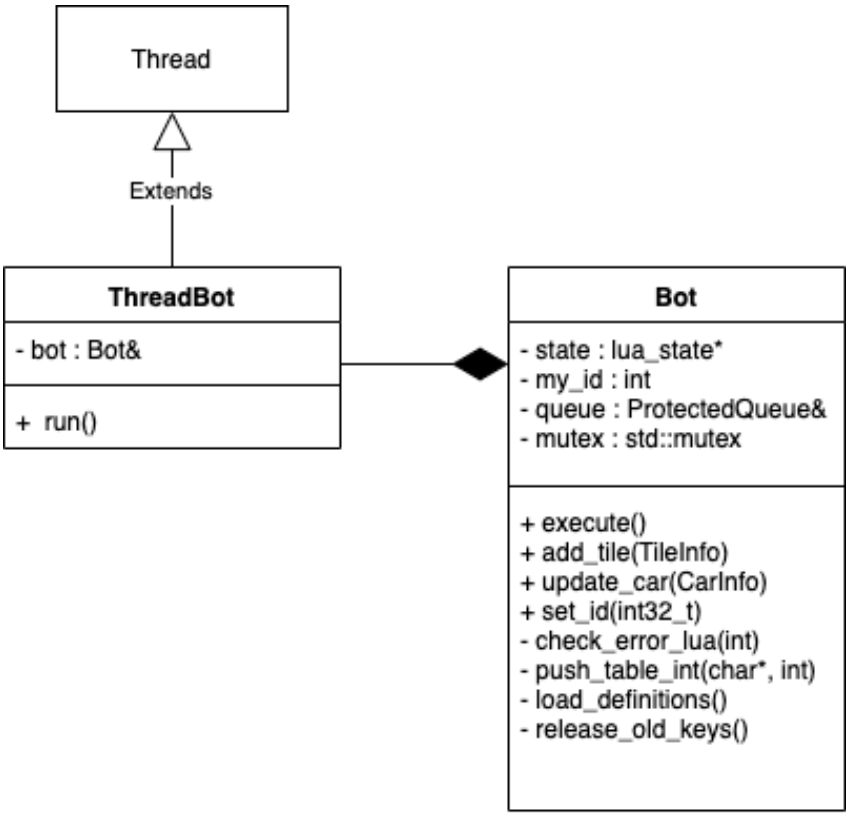


Figura 23: Estructura del ThreadBot

La instancia de la clase Scenario, también tiene una referencia a la instancia de la clase Bot, de esta forma cuando el Scenario recibe una actualización de parte del server se encarga de comunicárselo a la clase Bot y esta se lo comunica al script, de esta forma cuando el script es llamado por el hilo tiene la información actualizada y sabe que decisión debe tomar. Los métodos que se implementaron para lograr la comunicación entre el Scenario y el script son:

- **add_tile(TileInfo):** Esta función es llamada cada vez que el servidor le comunica al cliente que se tiene

que agregar un nuevo cuadrado al mapa. De esta forma el Bot sabe como esta compuesto el mapa y sabiendo su posición actual puede saber si tiene que seguir derecho o doblar.

- **update_car(CarInfo):** Cada vez que el server actualiza la información del auto, esta función se encarga de brindarle esta nueva información al script. Así, el Bot sabe la posición actual, la velocidad y la dirección del mismo.

Con el fin de lograr que la comunicación entre la clase y el script fueran mas amenas, se implementaron algunos métodos privados que nos ayudan con esta tarea:

- **check_error_lua(int error):** Cada vez que se utiliza alguna función para comunicarnos con el script de las librerías de lua, esta función nos devuelve un valor entero el cual hace referencia a un error. Para poder facilitar la detección de errores, se implemento esta función que en caso de haber tal error, lo imprime en salida estándar.
- **push_table_int(char *key, int value):** Esta función nos permite agregar un conjunto clave valor a una tabla previamente definida que luego podrá ser recibida por parámetro por una función del script.