



Manual de Proyecto

Nombre:	Padrón:
Hugo Larrea Buendia	102140
Juan Diego Balestieri	101601
Manuel Sturla	100912

Enunciado:

División de Tareas:

Según el enunciado dado, se dividieron las tareas de la siguiente forma: Larrea - Servidor, Balestieri - Cliente, Sturla - Bibliotecas.

Hugo Larrea: Servidor la parte de Box2D principalmente, física de los autos y los modificadores. También la pista, las posiciones y la condición de victoria. Configuración del servidor, de los autos y las pistas en yaml. Junto con sturla hizo la parte de comunicación del lado del servidor.

Manuel Sturla: Script de lua, ffmpeg y los mods. También hizo el protocolo de comunicación de parte del cliente y el servidor.

Juan Diego Balestieri: Cliente, SDL y el menú de qt. También se ocupó de agregarle la musica y hacer las animaciones.

Evolución del proyecto:

Inconvenientes Encontrados:

Cliente

El primer problema que posee el cliente es el menú de Qt. Este afecta la jugabilidad y por eso tuvimos que crear otro menú que no utilice Qt para poder jugar correctamente.

Buscamos solucionar el problema pero no pudimos lograrlo, sospechamos que sea algo del sistema operativo que utilizamos (Ubuntu 18.04 lts) relacionado con el gestor de ventanas ya que en otra computadora con otro sistema no tuvimos el mismo problema.

Otro problemas que tuvimos fue un error los drivers de video de las gráficas integradas de intel: **i965 batch buffer: invalid argument**. Sospechamos que este error se deba a un problema con la computadora de algunos integrantes del grupo ya que no ocurría en todas, ni tampoco era algo que solo ocurría en aquellas con menos hardware.

Generales

- Problema para instalar librerías y las versiones correctas: nos encontramos con varios inconvenientes a la hora de instalar librerías como Box2D o por ejemplo luego de correr el trabajo varias veces en algún momento se perdía alguna dependencia de alguna librería y dejaba de compilar.
- Nos enfrentamos a muchos leaks (principalmente en el cliente) muy difíciles de arreglar por las presiones de tiempo y por leaks propios de las librerías.

Puntos pendientes:

Gran parte de estos puntos no fueron logrados por falta de tiempo. Aunque alguno de ellos están contemplados en el modelo.

- No regresa al menú cuando se termina una partida.
- No hay variedad de autos. Este es un cambio contemplado en el modelo, pero no se llegó a hacer la comunicación y los Eventos correspondientes para que el cliente los pueda elegir.
- Falta pulir las texturas y agregar otras que dejen el juego mas lindo, como agregarle árboles y casas para que lo alrededores no tuvieran solo pasto. Esto no se realizó por falta de tiempo, ya que agregar las texturas no debería ser complicado por cómo está implementado el código.
- También falta implementar un contador de vueltas en el cliente. Que muestre al jugador cuantas vueltas le faltan del total.
- Mejorar la pantalla de espera y la de finalizar el juego que muestra al ganador pero con un texto muy simple.
- Implementar una manera de volver al menú cuando se finaliza una partida, simplemente se muestra el ganador hasta que el usuario cierra el juego y en el caso de querer jugar otra partida debería volver a iniciarlo.
- Falta que los Mods se llamen cada una cierta cantidad de tiempo y no en cada iteración del loop de la partida.
- Falta realizar un cmake que instale correctamente todos los archivos y dependencias necesarios.
- Problema con la física de Box2D el auto sigue con el rozamiento del pasto aún cuando sale de este. Este efecto termina cuando se ingresa en otro trecho de pista.
- Cuando se comienza a grabar el uso de CPU aumenta considerablemente. No pudimos encontrar cual es la causa.
- A veces el video grabado se corta
- El sistema para aparecer el auto luego de que explota/muere puede fallar y hacerte aparecer en una posición inválida.
- Cada tanto el cliente se freezea, no pudimos descubrir la causa.

Herramientas

Como entorno integrado de desarrollo, todos los integrantes usamos CLion de JetBrains. Para debuggear el código se utilizó tanto el debugger de CLion como gcc, también se tuvo en cuenta la información del uso de la CPU con el comando top. Como herramienta de control de versiones se usó Github y GitKraken.



Documentación Técnica

Nombre:	Padrón:
Hugo Larrea Buendia	102140
Juan Diego Balestieri	101601
Manuel Sturla	100912

Requerimientos De Software

1. OS, bibliotecas y herramientas necesarias para compilar, desarrollar, probar y depurar el programa

Se requiere de un sistema operativo linux e instalar las siguientes bibliotecas:

- Qt5-default
- SDL2-dev
- SDL2_ttf-dev
- SDL2_Image-dev
- SDL2_Mixer-dev (libsdl2-mixer-dev)
- Lua5.3
- libgl-dev
- avformat
- avcodec
- avutil
- swscale
- libgl-dev

Debido a que el programa fue desarrollado en C++, se necesita un compilador de dicho lenguaje (Se recomienda usar g++). La herramienta usada para desarrollar, probar y depurar el código fue CLion.

Descripción General

Descripción General:

- El Servidor se divide en los siguientes módulos:
 1. **Objetos:** Este módulo se subdivide en Carros, Modificables y Suelos. Representan a todos los objetos con los que los usuarios pueden interactuar a lo largo de una carrera de manera directa (ej. su carro) o indirecta. Todas las clases de este módulo tienen asociado un cuerpo en Box2D.
 2. **Planos:** En este módulo se encuentran clases que permiten la creación de Objetos según los parámetros con los que fueron contruidos. Esto permite leer una única vez los archivos de Yaml de configuración del servidor y que puedan ser re utilizados eficientemente.
 3. **Partida:** Este módulo engloba las abstracciones necesarias de una partida. Los clientes pueden unirse a la partida, con lo que se les creara un carro y serán notificados de los cambios que ocurren en el MundoBox2D durante el transcurso de la misma.
 4. **Acciones:** Engloba las posibles acciones que puede hacer un cliente con su carro: girarlo, acelerar o frenar.

5. **Comunicación:** engloba los eventos y clases necesarias para la comunicación del servidor con los clientes.
- El cliente se divide principalmente en los siguientes módulos:
 1. **Hilo Receptor:** se encarga de recibir los mensajes o eventos del servidor y los ejecuta.
 2. **Hilo de Visualización:** se encarga de mostrar por pantalla la información necesaria al cliente.

El servidor y el cliente solamente comparten algunas clases relacionadas con la comunicación, como por ejemplo el SocketAmigo y los Hilos.

Ambos módulos hacen uso de bibliotecas externas para lograr ciertas funcionalidades.

- El Cliente utiliza Qt para los menús del usuario; SDL2 para renderizar lo que se ve por pantalla durante la carrera; Lua para la inteligencia artificial de CPU y FFMPEG para grabar video durante la carrera.
- El Servidor utiliza Box2D para la física.

En ambos módulos se utiliza Yaml como formato para guardar las configuraciones. Todas estas bibliotecas se intentaron de modularizar de forma RAII.

Cliente

Este módulo es el encargado de interactuar con el usuario y permitirle a este conectarse a una instancia del servidor en algún puerto de la red para poder jugar. Se ocupa de recibir mensajes del servidor, mostrárselo al usuario y enviar sus respuestas al servidor. Para esto posee 3 hilos principales, el hilo lector, el hilo receptor y el de visualización. Cada uno cumple uno de los 3 pasos mencionados, el receptor de recibir los mensajes del servidor, el de visualización de mostrárselo al usuario y el lector de enviar la respuesta. Para entender mejor el funcionamiento del cliente explicaremos como funciona cada una y cómo interactúan con el servidor y los demás hilos.

Hilo Lector

Este hilo es el más independiente de los 3, ya que no se comunica con los demás hilos directamente sino que lo hace a través del servidor. Se ocupa de leer los comandos de teclado y enviar el mensaje correspondiente al servidor. Para esto posee distintos comandos de teclado que se ejecutan constantemente.

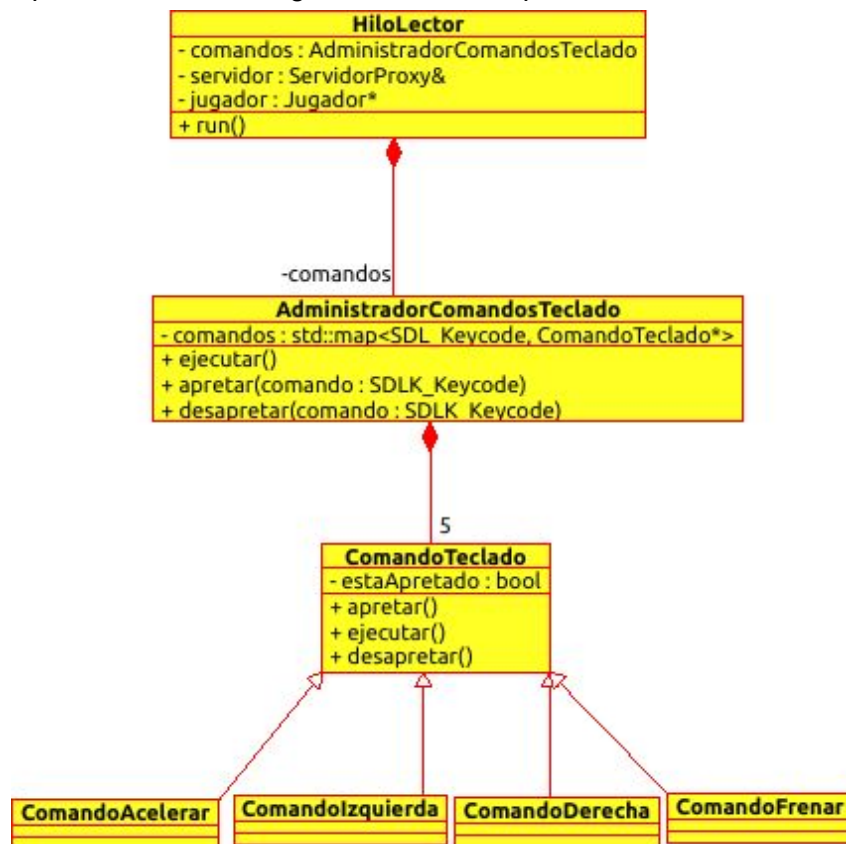
Versiones creadas:

La primer manera en la que se hicieron los comandos, fue ejecutando cada tecla cuando era apretada, o sea cuando se apretaba una tecla y se recibía un evento de `SDL_KEYDOWN` se ejecutaba el comando correspondiente a la tecla que había sido apretada y en caso contrario no se ejecutaba nada. El problema de esta versión fue que no permitía que se ejecutaran distintos comandos simultáneamente, ya que siempre se enviaba solo un comando por vez, lo que afectaba mucho la jugabilidad ya que no podía doblar y acelerar a la vez.

Para solucionar esta problemática pensamos en modificar el servidor y que este recibiera dos comandos distintos, uno de ejecutar y otro de dejar de ejecutar. Por ejemplo, si apreto para acelerar enviar un mensaje de acelerar una sola vez y cuando suelto la tecla envío otro mensaje de dejar de acelerar. El problema con esta idea estuvo en la complejidad de modificar eso en el servidor y la falta de tiempo. Por esa razón se buscó solucionar el problema del lado del cliente.

Para esto se le agregaron estados a los comandos de teclado y estos pasaron a ejecutarse constantemente. De esta manera, dependiendo de su estado el comando enviaba un mensaje al servidor o no. O sea, siempre que se aprieta una tecla se modifica el estado de esa tecla a apretado, lo opuesto ocurre cuando se deja de apretar la tecla. De esta manera, en todas las iteraciones se ejecutan todos los comandos y si estos están en un estado apretado envían mensajes al servidor y en caso contrario no hacen nada. Esto se pudo realizar ya que no contamos con muchos comandos para ejecutar y entonces iterar todos los comandos no es un problema.

A continuación podemos ver un diagrama de clases que muestra el hilo lector.



Esta última versión mejoró la jugabilidad pero sigue dependiendo fuertemente de la conectividad ya que necesita poder enviar varios mensajes al servidor constantemente. Este problema solo puede ser solucionado modificando tanto el cliente como el servidor.

Hilo Receptor

Este hilo se ocupa de recibir los mensajes del servidor, interpretarlos y modificar lo necesario. Es el que contiene a las clases “reales”, posee al auto, los extras y la pista, cada uno con sus respectivas posiciones. También conoce al renderizador y se ocupa de crear

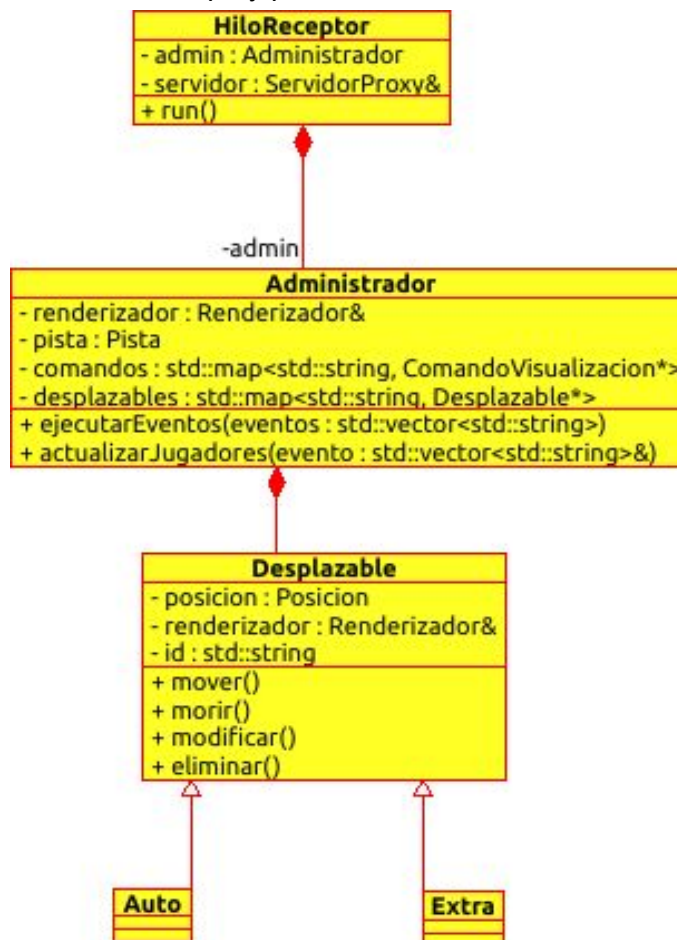
las texturas que sean necesarias. Luego se comunica con el hilo de visualización a través de las posiciones, modificandolas dependiendo de los mensajes que recibe del servidor.

Versiones creadas:

Primeramente, este hilo recibía eventos independientes uno por uno y se ocupaba de ejecutarlos y modificar lo que era necesario, esto funcionó bien hasta que se empezaron a recibir muchos eventos por parte del servidor. Esto era un problema ya que para poder modificar una posición o crear una textura se necesita de un mutex para proteger y evitar que se rendericen texturas que todavía no fueron creadas o que se rendericen en posiciones que no corresponden.

Para solucionar estos problemas fue necesario modificar tanto el cliente como el servidor. El servidor paso a enviar cuando terminaba una simulación, de esta manera el hilo receptor espera recibir una cierta cantidad de eventos antes de ejecutarlos. Luego, modificando el mutex pasamos a bloquear toda la ejecución de una simulación. De esta manera, evitamos estar constantemente bloqueando y desbloqueando procesos.

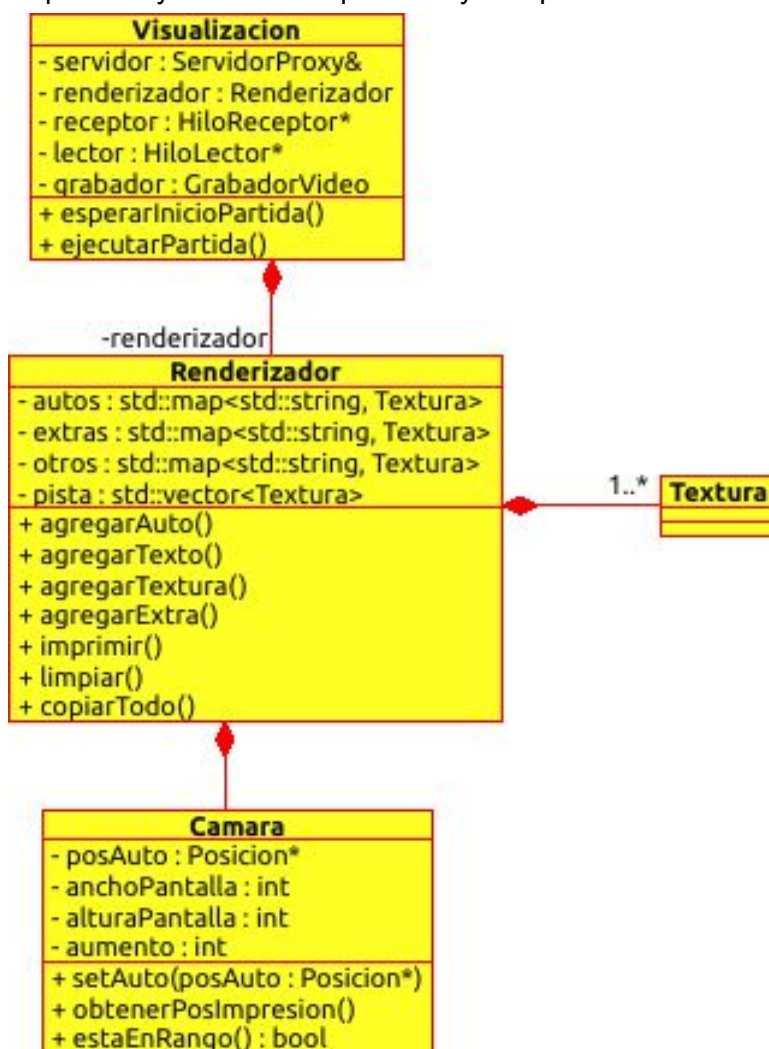
Para poner esto en práctica se creó una clase administradora que se ocupa de ejecutar los distintos eventos recibidos. El hilo lector espera a recibir una simulación completa y luego le envía esos eventos al administrador que bloquea el mutex y ejecuta todos esos comandos utilizando un mapa y polimorfismo.



Hilo de Visualización

Este hilo se ocupa de dibujar todas las texturas y de mostrarlas por pantalla. También es la que lanza los demás hilos (el lector y el receptor) y se ocupa de grabar los videos con la ayuda de otro hilo que escribe a disco. Para esto posee un loop de renderización con cantidad de frames por segundo configurables por un archivo yaml.

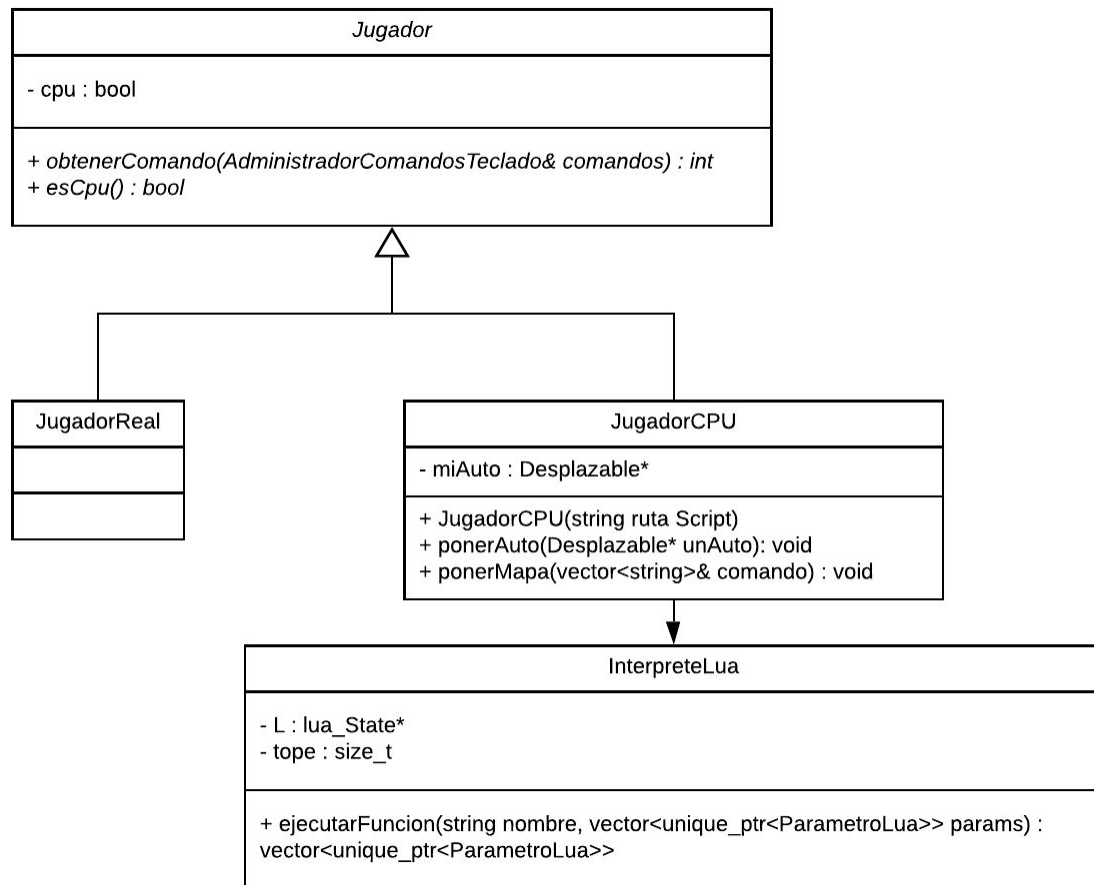
Esta clase posee al renderizador, el cual contiene a la cámara. La cámara se ocupa de “traducir” las posiciones del servidor a pixeles. Esta clase también sigue al auto modificando las posiciones en la que dibuja las demás texturas en función del auto que está siguiendo, todo esto sin modificar la posición de la textura que siempre es igual a la del servidor. También posee un aumento configurable por el mismo archivo yaml que permite modificar el tamaño de cada textura. La cámara también verifica si la posición de una textura aparece en pantalla y en caso de que no vaya a aparecer no la renderiza.



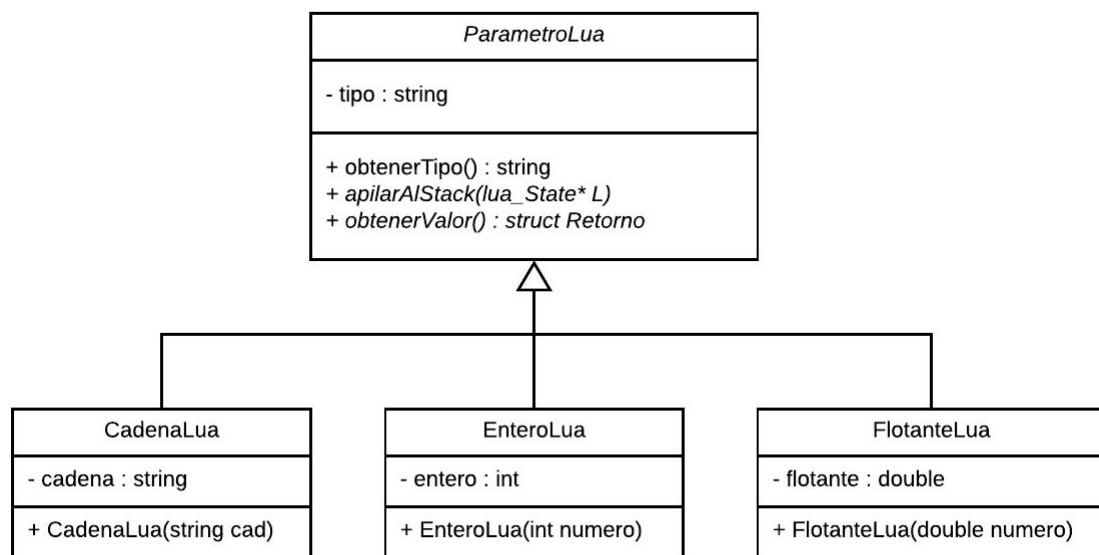
Jugador

En el cliente existen 2 tipos de jugadores, el “jugador real” que es el usuario indicando las acciones que debe realizar su auto o el “jugador CPU” en donde hay un script de Lua que

se encarga de la inteligencia artificial que indica las acciones a realizar. El jugador puede elegir jugar con cualquiera de los dos mediante el menú de QT.



El JugadorCPU es el encargado de transformar el nuevo estado (posición y ángulo) del auto a Parámetros de Lua de manera tal de llamar a *ejecutarFuncion* del intérprete con ellos. Y luego en función de los parámetros recibidos ejecuta un comando que se le aplicará al auto. Los parámetros de Lua se encuentran modelados de la siguiente manera:



Cada tipo de parámetro sobrescribe a la función obtener valor, devolviendo un struct Retorno en el cual solo guardan el atributo del tipo que son.

Servidor

Este módulo es el encargado de recibir conexiones de los clientes, permitirles crear partidas y unirse a ellas y poder jugar carreras entre ellos.

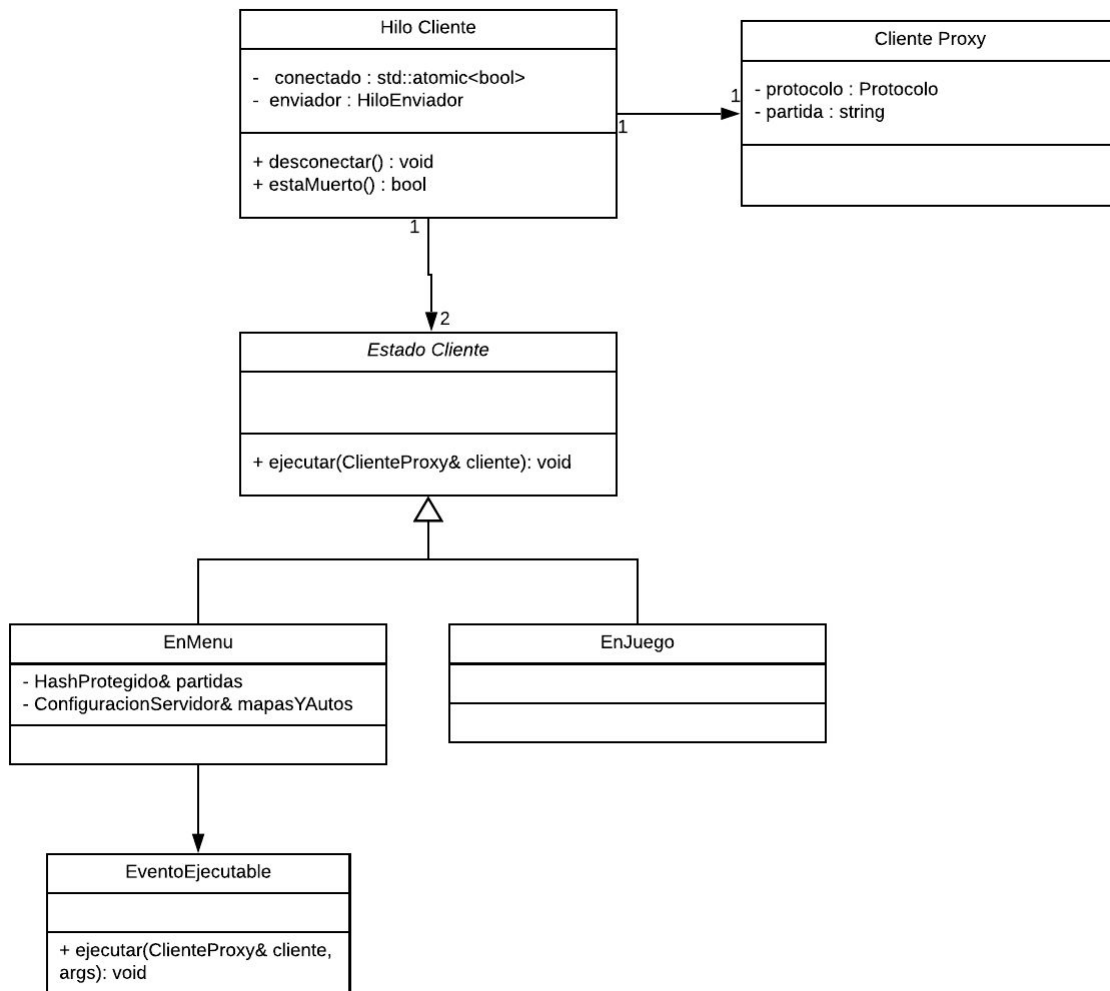
Servidor

La clase Servidor es la encargada de aceptar nuevas conexiones y iniciar los hilos de cada uno de estos clientes nuevos. También limpia los hilos de los clientes que se desconectaron así como las partidas terminadas.

Hilo Cliente

Estados

El Hilo Cliente es el Hilo que vive mientras el Cliente al cual representa se mantenga conectado. Todos los clientes tienen dos estados posibles: En Menú y En Juego. Cada uno de estos representando las dos etapas en las que puede estar un cliente conectado al servidor.



Cuando el Cliente se conecta comienza con el estado En Menú. Y por lo tanto solamente tiene acceso a los comandos:

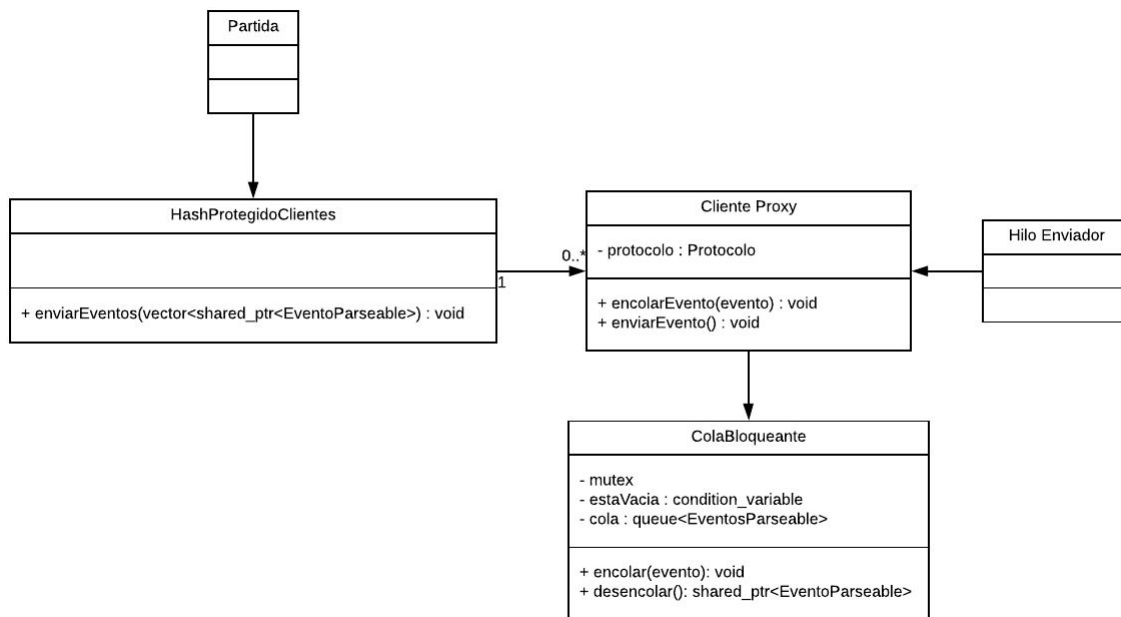
- **Crear Partida:** permite crear una partida nueva a partir de un nombre, cantidad de jugadores, cantidad de vueltas y un mapa.
- **Obtener Partidas:** envía al cliente todas las partidas que se encuentran en espera al momento de recibir el comando.
- **Unir a Partida:** une al cliente a una partida en espera a partir del nombre de la misma.
- **Obtener Pistas:** envía al cliente todas las pistas (mapas) posibles que tiene el servidor.

Luego de unirse a una partida, entra en el estado En Juego. Que lo que hace es ejecutar un Evento Ejecutable llamado **Recibir Acciones**.

Comunicación

Como vimos anteriormente es el Hilo Cliente el que se encarga de **recibir** eventos del Cliente. Estos eventos, son Eventos Ejecutables, dado que se ejecutan desencadenando

algún resultado en el Servidor. Ahora para **enviarle** la información tiene a su vez tiene un Hilo Enviador que utiliza Eventos Parseables, es decir que su función es poder, a partir de algunos datos, devolver un string parseado de forma que se pueda enviar el Cliente y este lo pueda entender. El Hilo Enviador tiene una referencia al Cliente Proxy, al cual le ejecuta constantemente el método *enviarEvento()*.

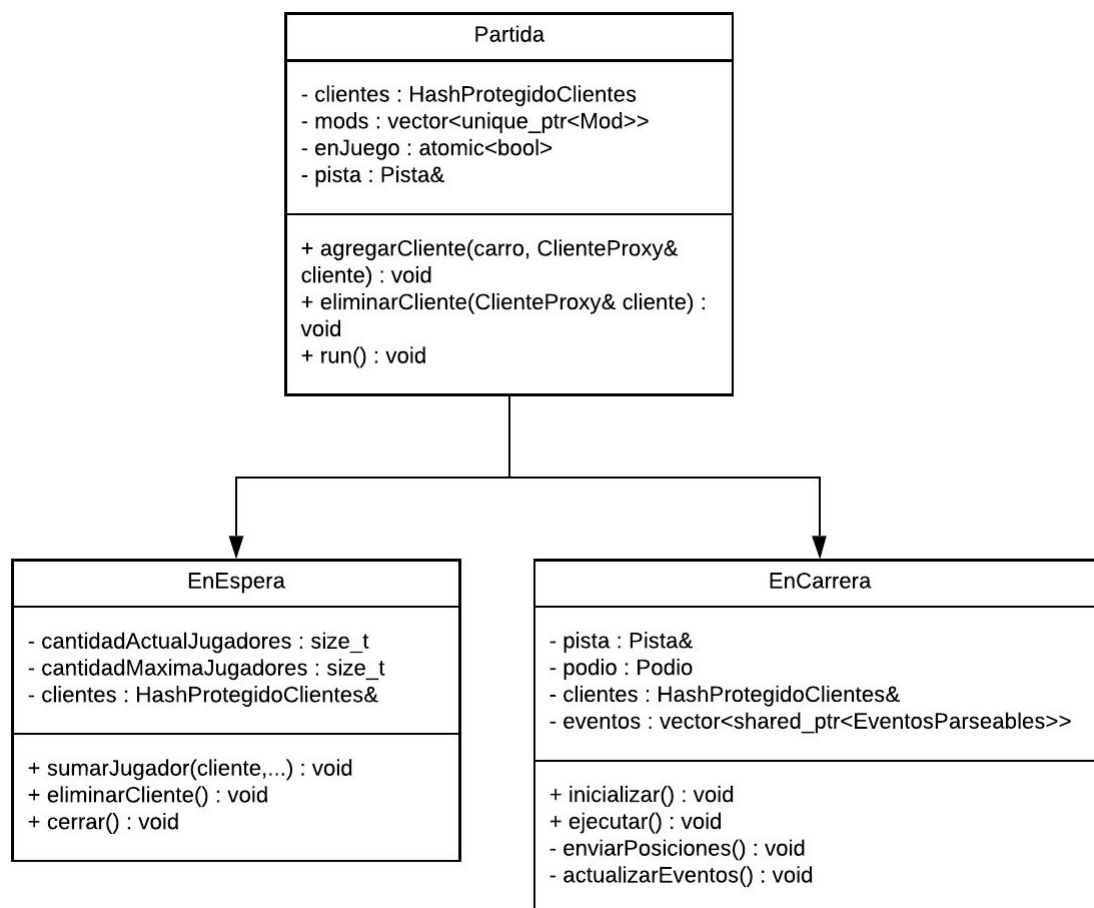


De esta manera el distintas partes del programa le indican al ClienteProxy *encolarEvento()*, lo cual hace que encole en evento recibido a una ColaBloqueante de eventos a enviar. Y el Hilo Enviador llama a *enviarEvento* que lo que hace es desencolar un evento (si no hay ninguno queda bloqueado), parsearlo y enviarlo al Cliente.

En el diagrama anterior se ejemplifica mostrando como la Partida, que tiene un Hash Protegido de Clientes del cual llama el método *enviarEventos* lo cual encola todos los eventos en un vector a la Cola Bloqueante de eventos a enviar de cada uno de los Clientes Proxy.

Partida

Esta clase representa a una Partida creada. Todas las partidas se almacenan en un HashProtegido de partidas que tiene el Servidor. Y las partidas son un Hilo para que pueda ejecutar toda la lógica que implica simular la partida de forma paralela e “independiente” al resto del servidor.



Cada vez que un Cliente ejecuta el comando Unir a Partida, se llama al agregarCliente de la partida. Si esta ya está en juego (la carrera empezó porque llegó al máximo de jugadores) se encolará un evento de NoSePudoUnir. Caso contrario se agrega el cliente al hash de clientes y se avisa a todos los jugadores la nueva cantidad de jugadores que hay en la partida.

Cuando la partida llega a la cantidad máxima se notifica y comienza a estar En Carrera. Se envían los mensajes iniciales correspondientes: el mapa por ejemplo y se inicializa el Podio.

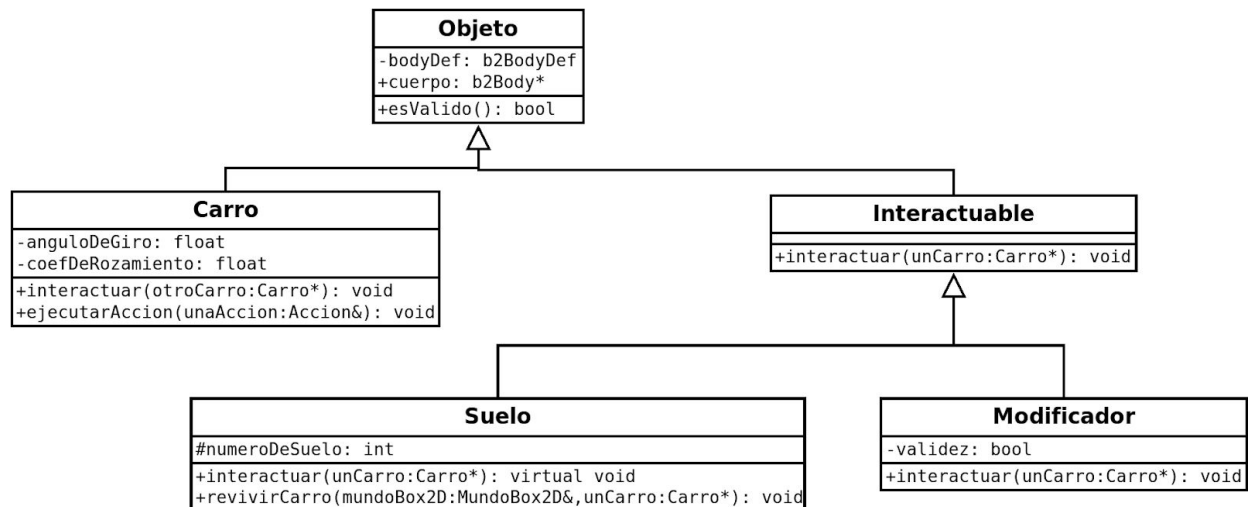
Configuración del Servidor

Esta clase es la encargada de cargar la información en los archivos de configuración. Es la que conoce y maneja YAML. Es un atributo del Servidor, por lo tanto se construye cuando este sea crea. Y utiliza los siguientes archivos:

- **pistas.yaml**: tiene todas las posibles pistas a elegir para una partida. Incluye la distribución de los suelos (asfalto, pasto) y información extra utilizada para calcular las posiciones de los autos en la carrera
- **mods.yaml**: tiene las rutas a los mods a cargar en las partidas y el tipo especificando si es un mod que modifica autos u otra cosa. (*ver puntos pendientes*)
- **carros.yaml**: tiene la información de los carros que pueden elegirse para la carrera. (*ver puntos pendientes*)

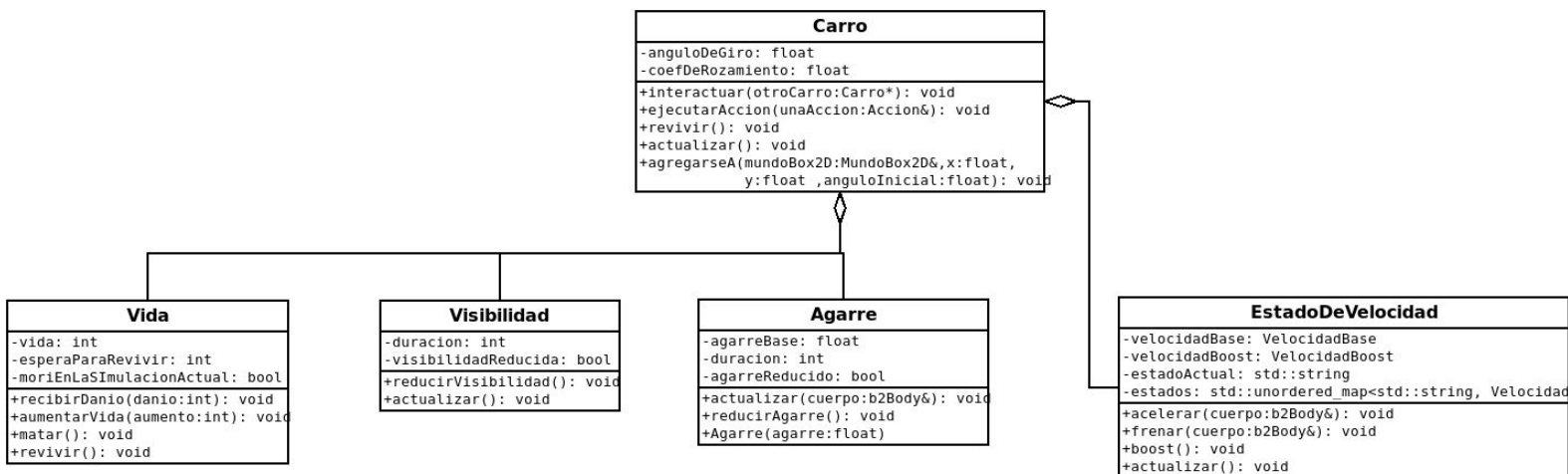
Objetos

Objetos: Este módulo se subdivide en Carros, Modificables y Suelos. Representan a todos los objetos con los que los usuarios pueden interactuar a lo largo de una carrera de manera directa (ej. su carro) o indirecta. Todas las clases de este módulo tienen asociado un cuerpo en Box2D.



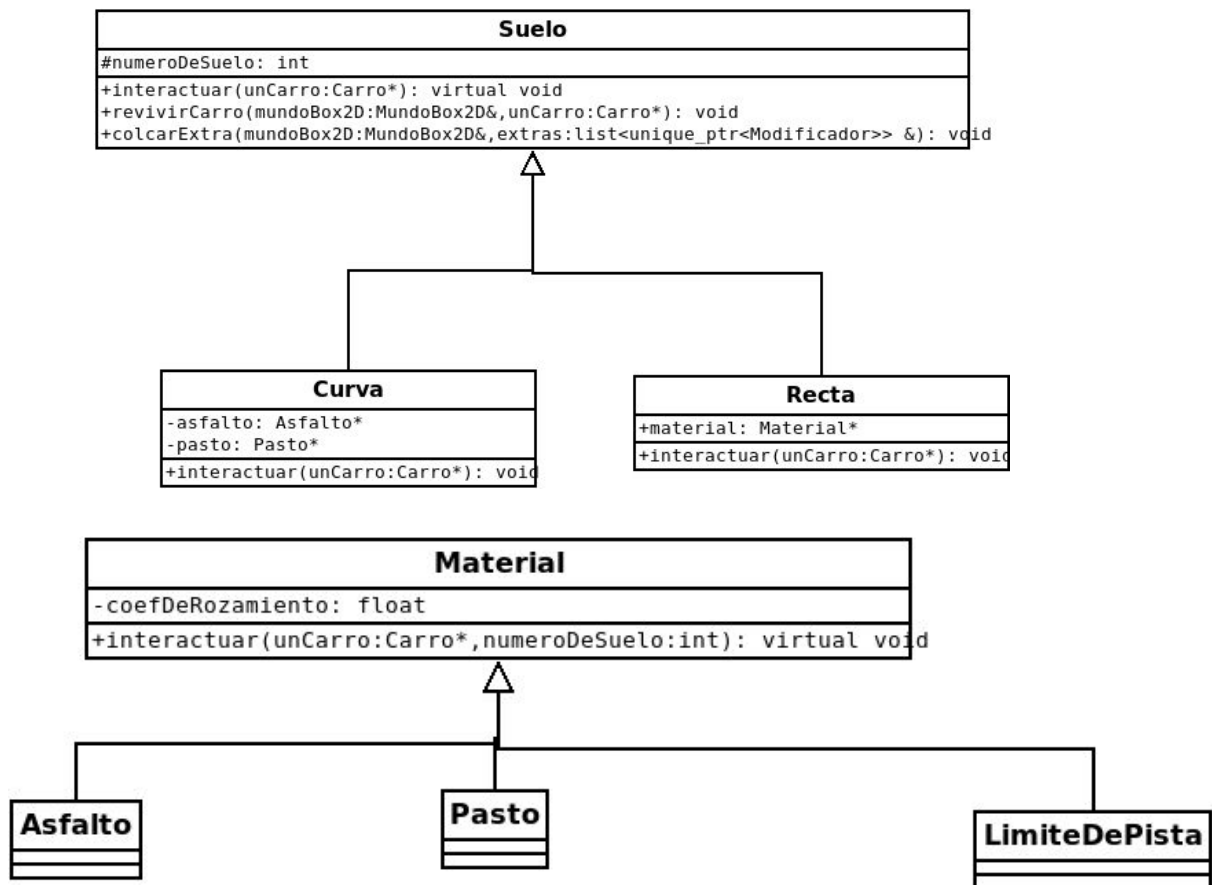
Como se muestra en el diagrama, todos los objetos tienen una representación en Box2D (programa usado para hacer la simulación física).

El carro, objeto controlado de forma directa ya sea por el cliente o por el script de Lua. Cuenta con clases que encapsulan el comportamiento esperado de este según los requerimientos del enunciado.

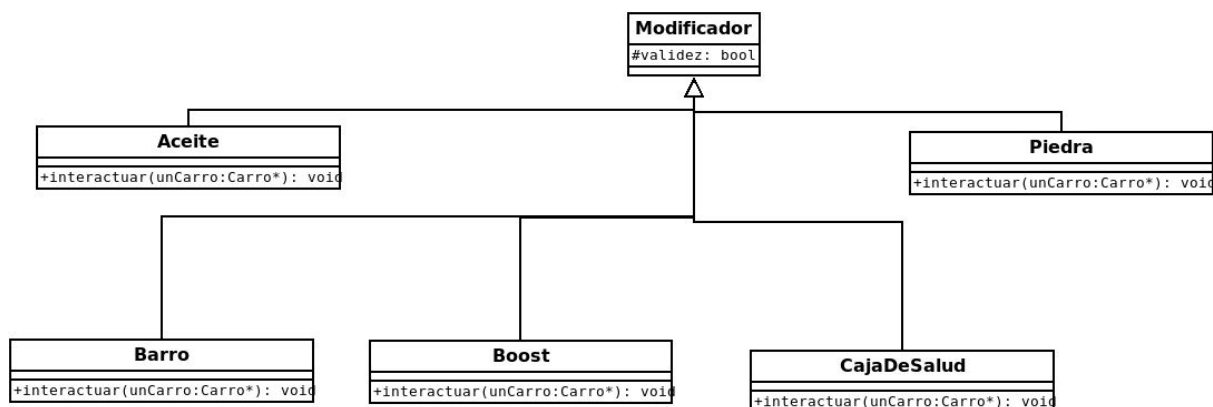


La visibilidad permite controlar si el usuario pisó un barro y cuando este dejó de tener efecto. La vida permite mantener registro de la vida de cada usuario, lo cual permite que el carro se destruya, además permite tener un lapso de espera hasta que el auto reviva para que el cliente tenga tiempo de realizar la animación de explosión. El estado de velocidad permite mantener registro de cual es la velocidad máxima del auto (la cual cambia dependiendo de si se agarra un boost). Finalmente, el agarre permite que el auto gire de mas dependiendo del agarre base del auto y si está o no reducido.

Los suelos permiten que el auto tenga un rozamiento distinto dependiendo de si este se encuentra en pasto o asfalto. También permiten controlar que el jugador no se aleje cierta distancia del mapa. Esto lo hacen delegando a su material, es cual se encarga de interactuar adecuadamente con el carro.

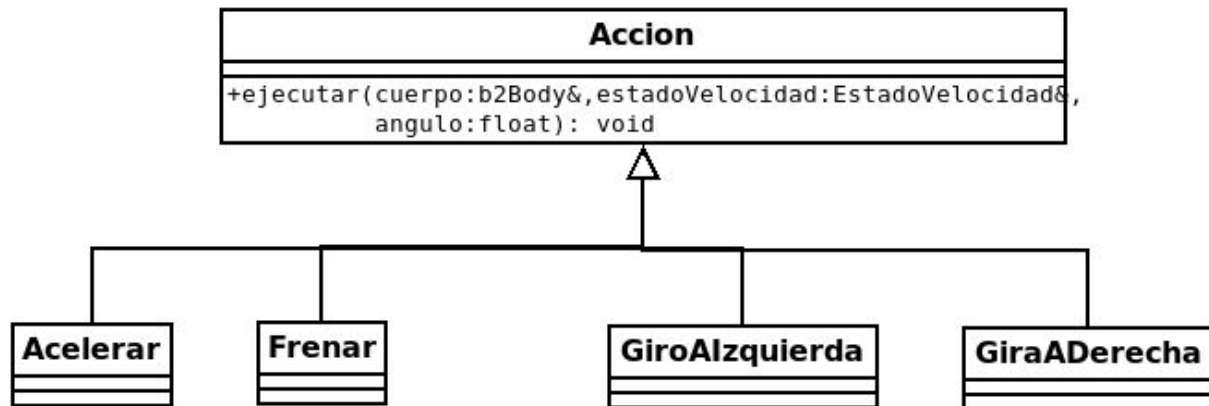


Los modificables interactúan con el carro, alterando uno o varios de sus atributos de forma indirecta con lo cual se ve afectada de una u otra manera la jugabilidad del jugador (ej. el barro no el permite ver lo que pasa en toda la pantalla)



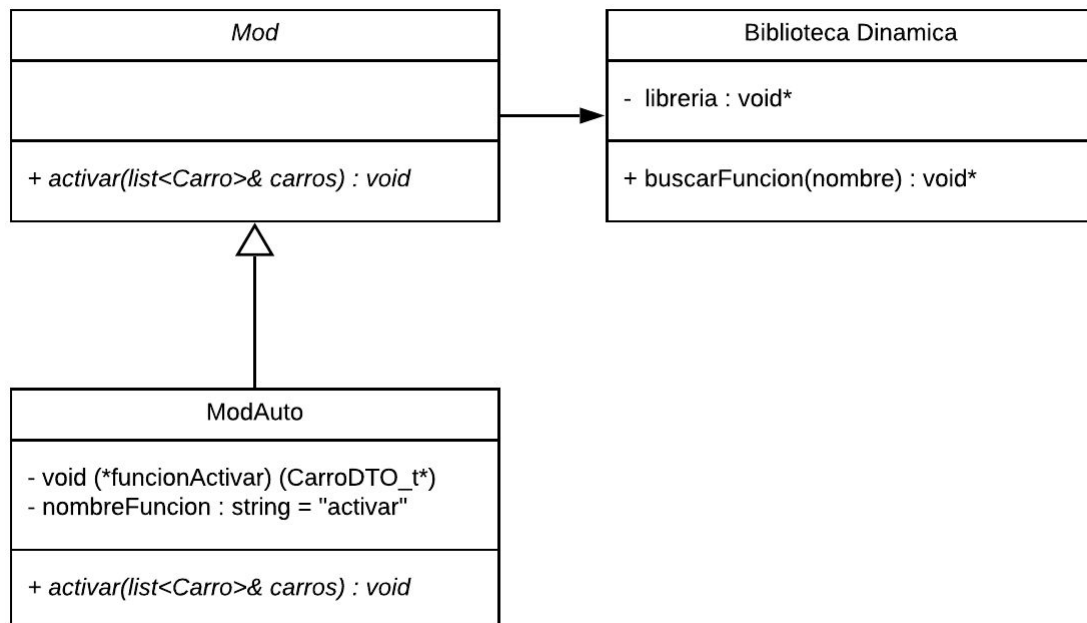
Acciones

Sirven para manejar el auto. Permiten hacer que este acelere, frene o gire según corresponda.



Mods

Los mods / plugins, son aquellos que se encargan de permitir el llamado a una biblioteca dinámica que modifica valores de algún objeto en particular. Todos los Mods tienen una Biblioteca Dinámica. Esta clase es la encargada de encapsular el cargado de la biblioteca externa (con `dlopen`), permite buscar una función y obtener el puntero a ella y cerrarla de forma RAII con (`dlclose`).



En particular, existen Mods de Autos, cuya función de activación recibe un puntero a un struct **CarroDTO** (Data Transfer Object). La función de este struct es que el **ModAuto** en su función `activar` le pide al **Carro** que guarde su estado en este struct, de manera que la biblioteca externa (dado que es un struct de C) puede recibirlo y modificar sus valores. Luego el **ModAuto** le pasa el **CarroDTO** al **Carro** para que actualice su estado.



Manual De Usuario

Nombre:	Padrón:
Hugo Larrea Buendia	102140
Juan Diego Balestieri	101601
Manuel Sturla	100912

Instalación

Requerimientos de software

Para utilizar este programa se requiere de un Sistema operativo basado en Linux. Deben instalarse

OS, bibliotecas y herramientas necesarias para instalar el programa:

1. Qt5-default
2. SDL2
3. Lua5.3
4. avformat, avcodec, avutil, swscale

Para requisitos más técnicos ver la documentación técnica.

Debido a que el programa fue desarrollado en C++, se necesita un compilador de dicho lenguaje (Se recomienda usar g++). La herramienta usada para desarrollar, probar y depurar el código fue CLion.

Requerimientos de hardware

Para el correcto funcionamiento de este programa se recomienda tener:

- 2GB o mas de memoria RAM
- Un procesador intel Intel Celeron N3050 2x 1.6 GHz
- 300MB de espacio en el disco para la instalación (incluye la librerías necesarias).

Proceso de Instalación

El proceso de instalación involucra instalar las librerías de los requisitos.

Como paso siguiente se recomienda descargar desde el repositorio de GitHub:

<https://github.com/hugomlb/MicroMachines>

Y finalmente ejecutar el cmake del servidor y el cliente por separado.

En caso de ser necesario descargar yaml-cpp <https://github.com/jbeder/yaml-cpp>. Y ponerlo en la carpeta del MicroMachines.

Configuración

Servidor

Para la configuración existen 3 archivos que modifican partes distintas

- **pistas.yaml**: tiene todas las posibles pistas a elegir para una partida. Incluye la distribución de los suelos (asfalto, pasto) y información extra utilizada para calcular las posiciones de los autos en la carrera
- **mods.yaml**: tiene las rutas a los mods a cargar en las partidas y el tipo especificando si es un mod que modifica autos u otra cosa.
- **carros.yaml**: tiene la información de los carros que pueden elegirse para la carrera.

Cliente

Mostramos cómo se debe implementar el archivo de configuración del cliente para que se pueda ejecutar el juego:

El archivo se debe llamar **config.yaml** y debe tener el siguiente formato:

```
configuraciones: {  
  aumentoCamara: 100,  
  fpsRenderizacion: 30,  
  alturaPantalla: 1000,  
  anchoPantalla: 1000  
}
```

Forma de uso

Servidor

Para utilizar el servidor basta con correr el archivo ejecutable indicando como parámetro el puerto en el cual se van a escuchar las conexiones entrantes de los clientes.

Para cuando se requiera cerrarlo por completo se puede ingresar una 'q' que cierra el servidor de manera ordenada.

Cliente

Para utilizar el cliente se debe ejecutar el archivo ejecutable.

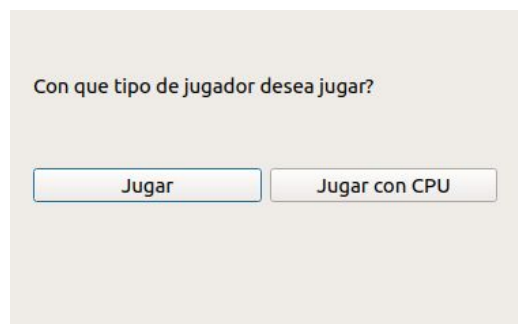
Menú

Con el menú tuvimos problemas al ejecutar Qt ya que este hace que se trabe el juego, fue un problema al que le buscamos solución pero que no pudimos encontrar. Por esta razón se hicieron dos menús distintos uno en Qt y otro más simple que se maneja por terminal. Para pasar de un menú a otro existen dos funciones en el main y se debe comentar una y ejecutar la otra. En esta parte hablaremos solamente del menú hecho con Qt ya que sería la versión oficial y la única si no fuera por cómo afecta la jugabilidad.

El menú contiene 4 ventanas distintas, la primera se ejecuta al iniciar el juego y permite ingresar el host y el servicio para conectarse al servidor, en el caso de no poder conectarse muestra un mensaje de error y otros datos deben ser ingresados.



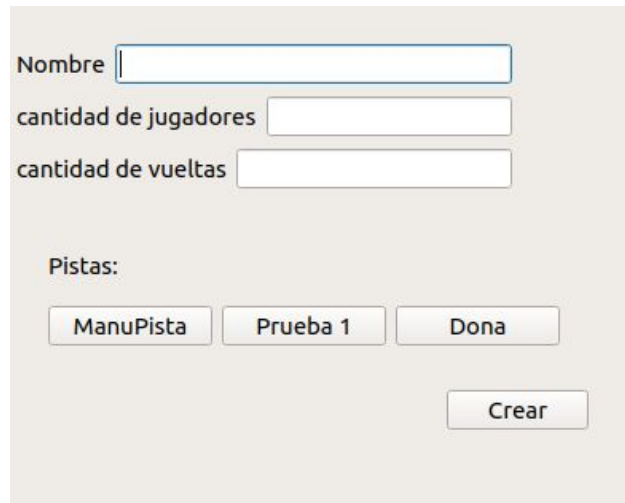
En el caso de poder conectarse al servidor se pasa al menu de elegir jugador. Aquí podemos seleccionar si deseamos jugar nosotros mismos o si deseamos que juegue una cpu, que es un script de lua.



Una vez seleccionado el tipo de jugador pasamos al Lobby que permite conectarnos a una partida ya creada o crear una partida propia.



En el caso de elegir alguna partida ya creada, el menú se cierra y empieza la partida. En el caso de seleccionar crear partida pasamos al menú de creación de partida. Este nos permite elegir 3 tipos de pistas distintos, elegir la cantidad jugadores y la cantidad de vueltas. También debemos ponerle un nombre a nuestra partida que debe ser distinto a las demás partidas ya creadas.



Formulario de creación de partida con los siguientes campos y botones:

- Nombre:
- cantidad de jugadores:
- cantidad de vueltas:
- Pistas:
ManuPista Prueba 1 Dona
- Crear

Al crear nuestra partida entramos a una pantalla de espera que nos muestra la cantidad de jugadores en esa partida. Debemos esperar a que se alcance la cantidad de jugadores ingresados para que empiece la carrera.

El juego

Una vez empezada la partida, podemos ver las posiciones de los jugadores y la nuestra que está en un color azul, mientras que la de los demás jugadores aparece en color blanco.



El auto rojo corresponde a nuestro auto mientras que los rivales son blancos, también se puede configurar por un archivo yaml el tamaño de la pantalla del juego.

Apéndice de errores

Si la aplicación tiene códigos de error, poner aquí su descripción, posible causa y posible solución