Documentación Técnica

MicroMachines v 1.00

Requerimientos de Software

Los requerimientos recomendados para correr este juego son:

- Ubuntu 18 o mayor
- Más de 4GB de RAM.

Descripción General

Micro Machines es un clásico juego de carreras, donde los autos pueden golpearse, irse de pista, explotar y explorar diferentes y divertidos escenarios. Este proyecto es una versión más simple de este juego, pero sin perder la diversión de este.

El proyecto consta de dos módulos, siendo uno el cliente y otro el server. El server es el encargado de comunicarse y actualizar a todos los clientes en las diferentes partidas que se estén llevando a cabo. El cliente, por otra parte, se encarga de mostrar las distintas escenas al jugador, además de comunicar las acciones de este al server.

El jugador tiene la posibilidad de jugar con un script realizado en Lua (que puede el crear u/o modificar). Finalmente, puede también crear sus propios Plugins, los cuales modifican aspectos del juego en tiempo de ejecución.

Módulos

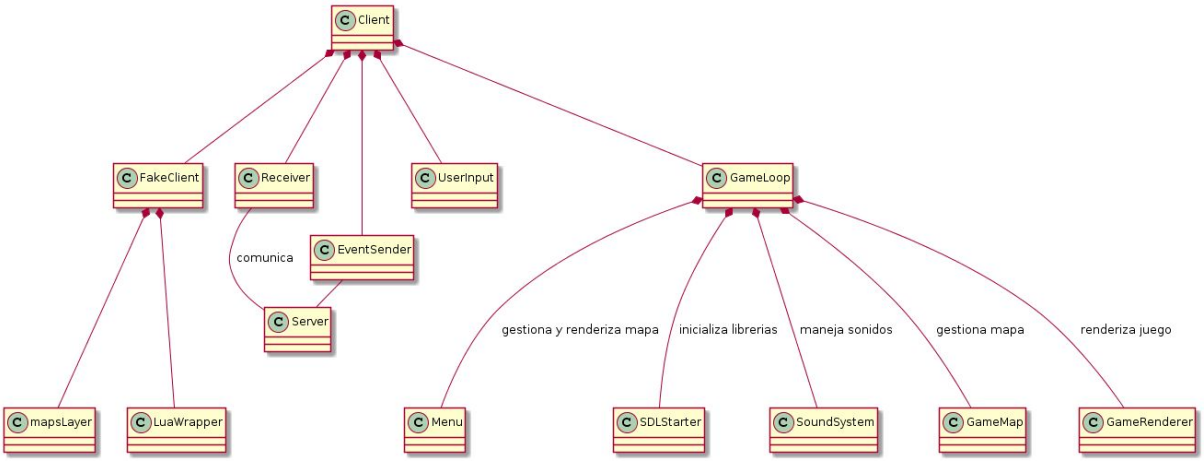
Client

Descripción general

El Cliente es el encargado de:

- Renderizar las diferentes escenas (menú, carrera, podio, etc) para que el cliente pueda elegir las distintas opciones y visualizar lo que está jugando.
- Reproducir los sonidos acordes a las situaciones que suceden
- Grabar la partida del jugador (de ser requerido por este)
- Obtener y enviar los eventos del jugador al Servidor
- Recibir y procesar las modificaciones mandadas por el servidor

En el siguiente gráfico se puede observar la conexión entre esta clase y el resto del trabajo.



Clases

Renderizado

Para renderizar la interfaz gráfica se utilizaron las siguientes clases

Tiles

Para la creación de los distintos tiles utilizados en el juego se utilizó el método Factory. De esta manera al leer el mapa simplemente se da el número a este y nos devuelve el Tile que queremos.
Su información, por otra parte, se encuentra en un archivo yaml, el cual posee la dirección de las imágenes.
Las clase utilizadas son, entonces

- *TileFactory*
  Carga toda la información de los tiles en un InfoBlock (más detalles sobre este en common). Luego, dado un int, devuelve el Tile correspondiente a este. Como se debe devolver un “new Tile(..)” se guarda este en un shared_ptr, para librar a quien use este Tile de la tarea de liberar la memoria de este
- *Tile*
  Guarda la información del tile indicado, y se sabe dibujar en la ventana.

GameMap

Todo lo explicado anteriormente, es utilizado por la clase encargada de gestionar el conjunto total de Tiles que representan a las tres capas (Ground,Road,Extras) que terminan siendo el mapa en sí. Sus funciones son pocas pero las suficientes como para que un GameMap no necesite a otro objeto para funcionar. Puede cargar un mapa. Puede renderizar dicho mapa y puede renderizar la decoración que está sobre ese mapa.

Animaciones

Para las animaciones se utilizó una interfaz Animación que es implementada por las tres animaciones creadas para esta versión: StainAnimation, DustAnimation y Explosion.

- StainAnimation: Representa la animación asociada al ítem barro. Muestra una mancha de barro en toda la pantalla. Se activa al momento de interactuar el auto con una instancia del objeto. La animación consta de una imagen que va ganando transparencia frame a frame.
- DustAnimation. La animación funciona de manera similar. La diferencia radica que se coloca sobre los ítems para simular el impacto de los mismos contra la pista.
- Explosión. Finalmente, esta clase representa la explosión del vehículo cuando su vida llega a cero. Esta lógica no se encuentra dentro de la animación. Lo que sí tiene es la carga y fragmentación de una imagen que tiene todos los frames necesarios para simular el movimiento.

Vehículo:

Cada uno de los vehículos está modelado por la clase Car. Esta clase, no solo se encarga de representar al auto lógicamente, sino que también consta de su propia textura, y su propio método de renderizado dependiente de la cámara. Esto es así, debido a que la cámara debe seguir al autor principal en todo momento.
Finalmente, el vehiculo tambien es capaz de cargar las animaciones asociadas a sí mismo. En esta versión, solamente tiene asociada la explosion al momento de tener la vida en 0, sin embargo se le podrían agregar fácilmente otras. Por ejemplo, una animación de tirar tierra a su paso.

Ítem:

A diferencia de como se trato a las animaciones, solo existe una clase Ítem que representa a todos los items en el juego. Esto es así, porque toda la lógica se encuentra del lado del servidor. Por tanto, el objeto solo contiene la información necesaria para que el servidor lo reconozca (un id) y aquella necesaria para su renderizado (Textura)

Menú:

La clase menú es la encargada de manejar las 3 pantallas que se pueden visualizar en el juego que no pertenecen al juego en sí. Pantalla de inicio, pantalla de selección de mapas y pantalla de espera. Hay muchísimas similitudes entre la primera y la segunda, mientras que la tercera simplemente cuenta con una imagen de fondo. Es por ello que se decidió que toda la lógica esté en un solo objeto. Por tanto, contiene métodos para gestionar estas pantallas y poder hacer la transición al juego en sí. Para poder realizar esto y recabar la información brindada por el usuario, utiliza los siguientes elementos interactuables

Elementos interactuables:

- Botones: Existen varios tipos de botones. La clase principal que engloba la mayoría del comportamiento se llama Button. Esta, incluye la información y métodos básicos necesarios para el renderizado de la mayoría de los botones del juego. Tanto los

botones que representan los mapas y los autos disponibles, funcionan solamente con esta clase. Sin embargo, hay dos tipos más de botones. Por un lado, el botón de conectar reescribe la función de renderizado para tener el tamaño deseado. Por su parte, el botón que activa el uso de la IA, si bien tiene el mismo tamaño, en esta ocasión se le agrega un mensaje. Lo que sí tienen todos en común, es la función que realiza el comportamiento de cualquiera de los botones al ser presionado. Esto es así, porque no esta codeado dicho comportamiento, sino que se le puede agregar diferentes funciones externamente. Con esto, el botón simplemente se encarga de saber si fue pulsado o no y ejecutar las funciones que tiene asociadas.

- Textos: Para todos los textos que se pueden observar, existe la clase TextLabel. Esta, es la encargada de todos los textos que se muestran en pantalla, tanto su gestión, como su renderizado. El único caso en el que hablamos de un caso diferente, es el texto que muestra el input del usuario. Para este, tenemos la clase TextBox. La motivación detrás de la implementación de esta clase extra es la modificación de la forma de renderizar. Mientras que TextLabel muestra por pantalla el mismo mensaje siempre, TextBox tiene la lógica suficiente detrás para poder mostrar el input del usuario.

SDLStarter

Todas las clases anteriormente utilizan de una forma u otra las librerías de SDL. Estas, necesitan ser inicializadas antes de su uso. SDLStarter se encarga de realizar esto.

TextureLoader

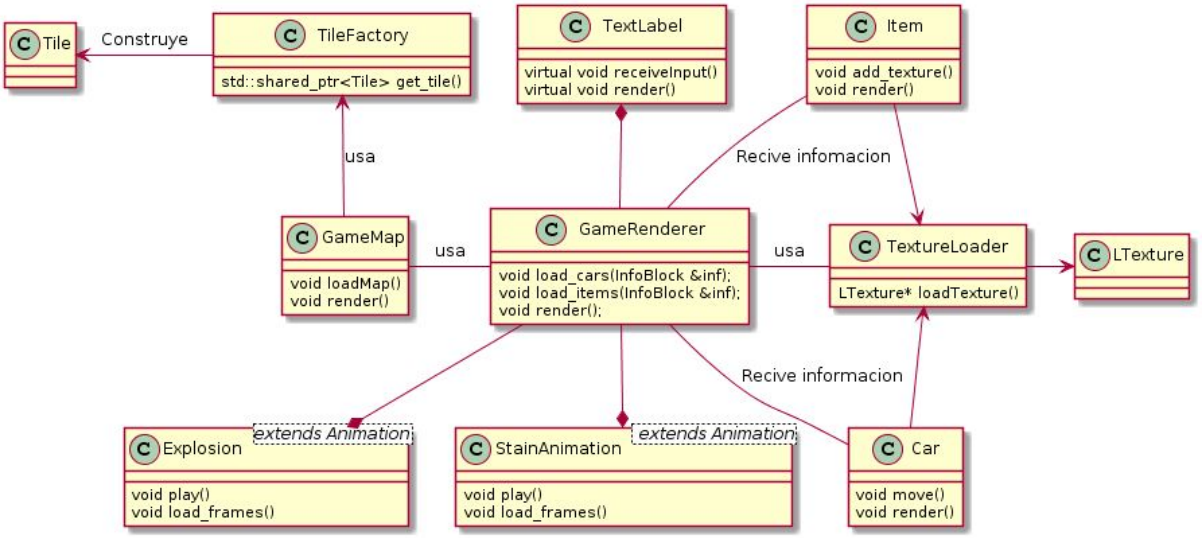
Exceptuando casos concretos, TextureLoader es la clase que se debe utilizar al momento de cargar alguna imagen por pantalla. Por dentro, utiliza las librerías de SDL para funcionar. El detalle de que no siempre se utiliza esto, es por el hecho de que solo carga una vez la textura, la engloba en una LTexture (ver más adelante) y devuelve un puntero a la misma. Sin embargo, si ya se cargó previamente esta textura, nos devuelve el puntero a la textura en cuestión. Esto puede ser muy beneficioso para evitar duplicar cargas, pero a veces es indeseado.

LTexture

La última clase destacable es LTexture. Es utilizada de alguna forma por todos las clases anteriormente mencionadas. Su función es la de englobar todo el comportamiento de la librería de SDL para manejo de texturas.

GameRenderer:

Todas estas clases, con la excepción obvia del menú, son usadas internamente por GameRenderer, el objeto encargado de renderizar el juego en sí. Para entender esto mejor, se muestra a continuación un esquema con las conexiones que tiene este objeto:



Como verá, GameRenderer se encarga de juntar toda la información necesaria para el correcto funcionamiento del resto de los objetos, transmitirla y ejecutar, en orden, las funciones de renderizado correspondientes.

Sonido

Para reproducir los distintos sonidos se utilizaron las herramientas provistas por SDL. La clase SoundSystem se encargar de desencolar, de haber, un sonido por iteración del juego. Por no ser alta esta cantidad de sonidos no encontramos la necesidad de utilizar un thread aparte para esta clase. Por otra parte, se tiene una música de fondo a bajo nivel.

Finalmente, los sonidos se cargan desde YAML a un mapa, del cual al desencolar un evento se utiliza para saber qué sonido se desea, y reproducirlo.

El sonido del juego puede ser pausado y reanudado.

Lua (inteligencia artificial)

Para proveer al jugador la posibilidad de no jugar pero que un ai lo haga por él se hace uso de las siguientes clases

- *FakeClient*
  Es el encargado de obtener las distintas informaciones necesarias para llamar al wrapper de lua, y además encolar su decisión luego para que sea enviada al server, como si fuese un cliente jugando normalmente. El utilizar LuaWrapper la abstrae totalmente de las diferentes acciones que hay que realizar para mandar parámetros y recibir entre lua y c++, y pueda concentrarse en la comunicación.
- *LuaWrapper*
  Wrappea la función del script de Lua que se llama para obtener la decisión de este. Esta es una que recibe una matriz, la posición x e y del auto y finalmente la última acción realizada.

Para esto se envuelve la función en lua en una en c++. Teniendo funciones auxiliares para pasar correctamente matrices, número y demás a Lua. Se hace uso del stack para pasar estos distintos parámetros (como también para luego recibirlos)

- *Script en Lua*
  Pese a no ser una clase de c++, se debe incluir además un script en Lua llamado fake_player.lua. Este es el encargado de tomar las decisiones para con el jugador. Las instrucciones para cómo realizar uno se pueden encontrar en el manual de usuario.

Video

Para proveer al jugador la posibilidad de grabar la partida se hace uso de las siguientes clases

- *FormatContext*
  Wrapper de avformat para que sea RAII

- *OutputFormat*
  Encargado de escribir los frames dados a disco.

- *VideoRecorder*
  Llena el buffer con la información de imagen dispuesta a ser renderizada. Por ser el proceso de escritura muy lento, en vez de escribir aquí, se encola en una SafeQueue (más explicaciones de esta) de tamaño.

- *VideoWriter*
  Este thread es el encargado de desencolar el buffer que encola videoRecorder y escribirlo en el archivo creado. Todo video finaliza agregando unos bytes finales, necesarios para el archivo se lea correctamente, por lo que sabemos que al desencolar esto el thread ya no es necesario.

GameLoop

Finalmente, la clase encargada de gestionar el conocido “loop del juego”. Este objeto se encarga de gestionar no solo el Menu y el GameRenderer, sino que también se ocupa de las transiciones entre unos y otros. En sí, su trabajo es el de llamar en el momento correcto a métodos de los objetos anteriormente mencionados para que el juego funcione.

Server

Descripción general

El Server es el encargado de:

- Crear múltiples partidas
- A su vez, posibilitar múltiples jugadores en cada una de estas

- Para cada carrera, reproducir el mundo físico de esta
- Recibir la información de cada cliente y procesarla en el mundo físico
- Mandar las actualizaciones de posiciones a los diferentes clientes en juego
- Correr cada n tiempo prefijado los plugins

Clases

Modelo Físico

El modelo físico es la abstracción realizada para operar sobre la librería física Box2D y cuenta con las siguientes clases:

- *GameWorld*
  Es el que engloba el modelo físico bWorld2d teniendo referencias y la capacidad de crear autos (y hacerlos respawnear una vez muertos), items y las secciones del mapa. Posee el método step() para avanzar la simulación

- *CollisionsManager*
  es la clase que maneja las colisiones overrideando el detector base de box2d

- *Entity*
  Entity es una de las clases mas importantes del modelo ya que representa todo objeto “simulable”. Es una clase abstracta cuyo comportamiento es definido por los distintos objetos que componen el modelo

    - RaceCar
      Es la representación del auto en el juego, posee las distintas hitboxes para manejar colisiones y es la encargada de procesar su movimiento basándose en sus CarStats una vez aplicados los efectos de los StatusEffects de los cuales actualmente padece

        - CarStats
          Engloba las características del auto como health, max health, current speed, max speed, acceleration, rot force entre otras. Esta es la clase que se verá alterada por los StatusEfects

    - OffRoad
      Esta clase representa “la parte de afuera de la pista” y aplica un efecto de perdida de vida (HealthEffect) y reduccion de velocidad (SpeedEffect) luego de un pequeño lapso de tiempo al colisionar con el sensor del auto

    - FinishLine
      Al colisionar con un auto, este objeto incrementa en 1 la cantidad de vueltas y aplica un cooldown al auto

    - Todos los Items (ItemBoost, ItemHealth, ItemRock, etc)

aplican un efecto en particular al auto y luego son eleminados

- *StatusEffect*
  Es una clase abstracta y sus clases hijas son parte de un sistema de componentes que aplica distintos efectos a las caracteristicas de un auto, ya sea por un intervalo o en un punto en particular de tiempo. Estos componentes son agregados por medio de Entity::addEffect y se resuelve internamente dependiendo de cada efecto.
  - SpeedEffect
    aplica un efecto a la velocidad
  - LapCooldown
    mantiene el cooldown para contabilizar vueltas
  - CallbackEffect
    permite llamar una funcion callback despues de un tiempo, actualmente se usa para respawnear vehiculos
  - HealthEffect
    aplica un modificador a la vida del auto
  - DragEffect
    aplica un modificador a la capacidad de rotar del auto

Plugins

Los plugins son librerías dinámicas ya compiladas que se corren en tiempo de ejecución del juego. Las que se encuentren en la carpeta plugins al momento de comenzar una carrera son las que se utilizarán en esta.
Por una parte tenemos la clases que hacen posible la lectura y ejecución de estas librerías

- *PluginLibrary:*
  Carga y guarda todos los plugins que se encuentran en la carpeta plugins. Cada uno es englobado por la clase pluginLoader, la cual explicaremos a continuación. Por otra parte, lleva un contador interno para poder decidir si efectivamente correr o no los plugins cuando se le pide.
  Finalmente, destruye todos lo plugins en su destructor.
- *PluginLoader:*
  Carga el nombre de plugin indicado en su constructor. Como todo plugin debe tener una función para crearse y destruirse (útil para hacer RAII los plugins, ya que estos podrían necesitar pedir memoria que debe ser luego liberada) llama a estos en su propio constructor y destructor.
- *Plugin:*
  Todo Plugin que se cree debe heredar de Plugin.h . Esta es la manera de conectar al
  servidor con esta librería que fué compilada aparte, ya que permite que puedan comunicarse entre sí.
- *PluginEjemplo:*
  Se deja un plugin de ejemplo con su respectiva explicación para que los usuarios puedan realizar su propio.

Partidas

Las clases principales que nos proveen el soporte multijugador y multipartida son:

- *GameManagerThread*
  Es el encargado de guardar referencias a todos los juegos activos y de aceptar conexiones entrantes de jugadores los cuales son asignados a un ChoosingLobbyThread

- *ChoosingLobbyThread*
  Es el encargado de gestionar un jugador que esté decidiendo el nombre de la partida a la cual unirse. Una vez recibido el nombre de la partida, el thread verifica que sea válido: Si es de un juego que esté en modo lobby, añadirá al jugador a ese juego, si el juego se encuentra en transcurso pedirá que se ingrese otro nombre y por último si no hay juegos con ese nombre, creara uno y asigna al jugador como dueño del lobby

- *GameThread*
  El GameThread es el encargado de actualizar y manejar el estado del juego. Este thread se inicia en modo lobby donde puede aceptar jugadores para que se unan a él. Una vez que el líder del lobby elija una pista e inicie la partida el thread se encargará de procesar con el modelo físico los input que hayan recibido cada uno de los threads asociados a los jugadores
  Una vez terminada la partida, el thread anunciará los ganadores y el juego terminará

Diagramas UML

Diagrama de la relación de clases en el servidor

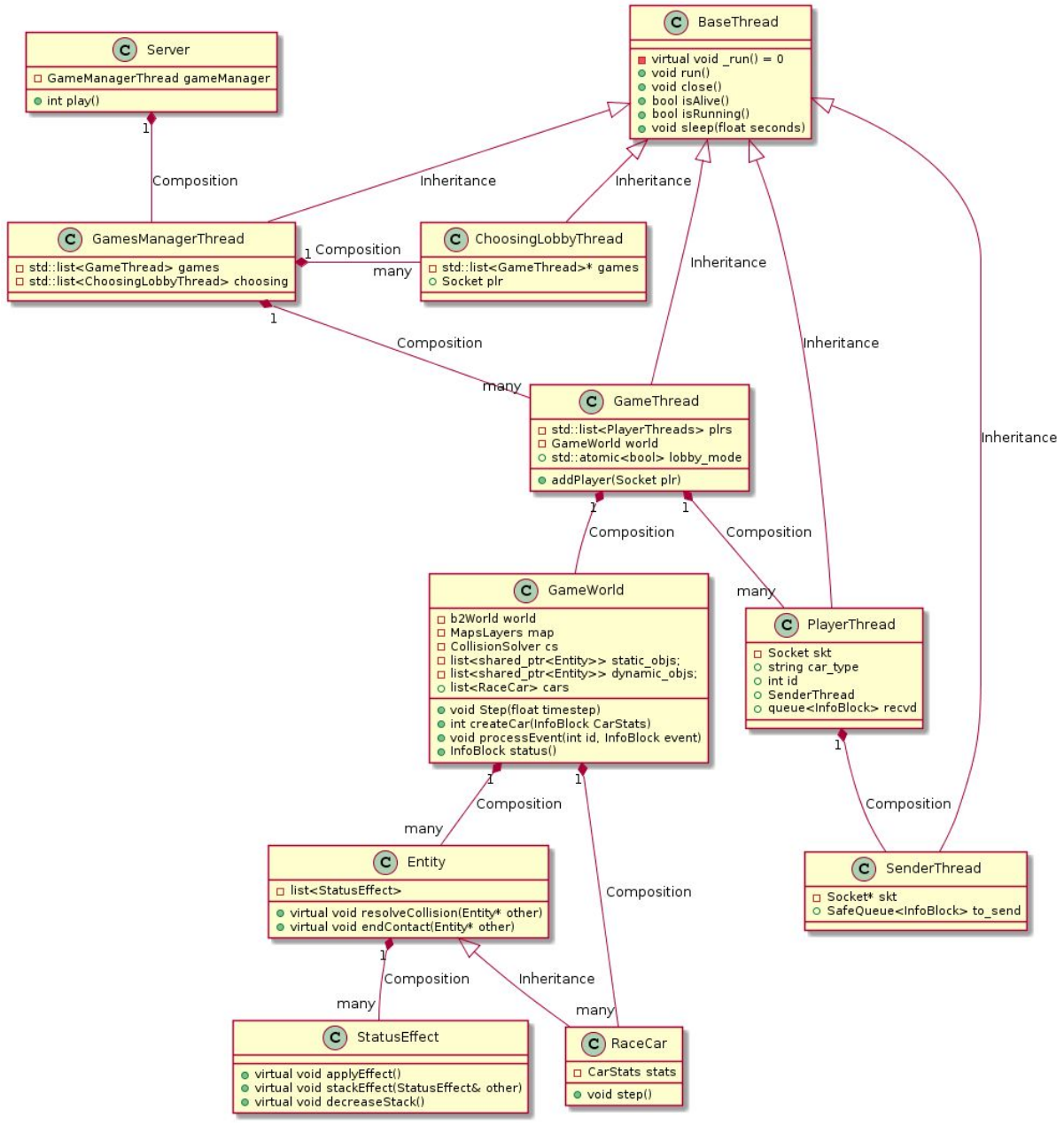


Diagrama de secuencia general del server, mostrando los eventos correspondientes al inicio de un servidor, aceptar jugadores, correr los juegos y cerrar el servidor

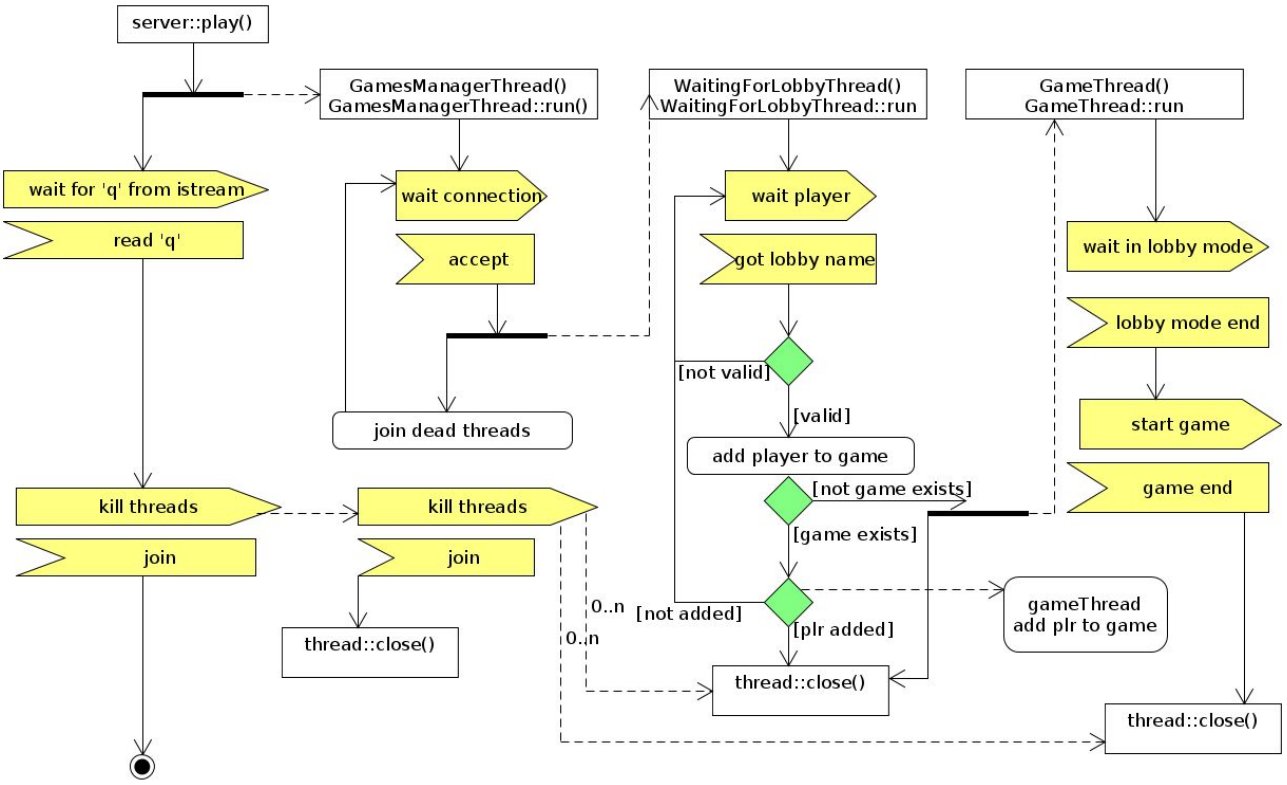


Diagrama de secuencia de la ejecución del GameThread

Common

En la carpeta common encontramos todas las clases en común, que comparten tanto el módulo del cliente como el del server. Procedemos entonces a explicar las más importantes de estas

- *InfoBlock*
  Es la clase que encapsula el uso de de la librería YAML y la principal forma de transmitir información dentro del programa. Se usa principalmente para guardar mensajes que serán enviados o fueron recibidos por medio de un socket

- *SafeQueue*
  Cola segura template que puede ser abierta o cerrada y tener una capacidad. El que se cierre o no nos ayudó para utilizarlas durante toda la vida del cliente, permitiendo el ingreso de información únicamente en los momentos que necesitábamos.
  Por otra parte, por su cv esta cola quedará dormido hasta tener un elemento que desencolar en caso de querer popear en una vacía.
  Finalmente, su capacidad de tener un máximo y poder pushear un elemento emplace nos permitió que al grabar un video, e ir pusheando los buffers a escribir, no guardaramos frames que sabíamos no iba a llegar a dibujar el escritor.

- *ThreadQueue*
  Esta es una cola que posee un único mutex, para evitar que distintos threads quiten y pongan cosas en esta a la vez. Es utilizada únicamente para el sistema de sonido de juego, ya que no necesitábamos que se quede esperando de no haber sonidos.

- *Protocolo*
  Para la comunicación entre el server y cliente se mandaron los mensajes siguiendo la formato:
  _ Primero se envía el largo del mismo, el cual es un uint_32
  _ Luego se envía el propio mensaje
  Por otra parte, toda mensaje recibido se traduce a un InfoBlock (clase antes explicada). Esto nos permitió mantener un formato en la comunicación y evitar problemas de parseo.

- *MapsLayer*
  Carga las distintas capas del mapa en sus respectivas matrices. Cada mapa (que se lee de un archivo YAML) posee un piso, una ruta y finalmente sus respectivas decoraciones. Esto le sirve tanto al cliente para renderizar la escena como también al server para poder crear el mundo físico.

- *Socket*
  Trabaja como los sockets anteriormente utilizados en la materia. Se puede decidir cargar a este como un client o como un server. La comunicación a partir de aquí será la misma para los dos.

- *BaseThread*
  Thread base utilizado para la creación de todos los threads del proyecto. Brindan la posibilidad de pausar la ejecución de un thread, para luego volver a reanudarla. Esto nos fue muy útil para, por ejemplo, el Sender del cliente, el cual había momentos en los cuales no queríamos mandar nada al Server.

Descripción de archivos y protocolos

ARCHIVOS DE CONFIGURACIÓN:

El formato de archivo utilizado para las configuraciones y diseño de mapas fue YAML. Se consideró la utilización del formato JSON, pero se terminó utilizando únicamente el anteriormente mencionado.

Pueden encontrar en este formato las distintas configuraciones:

- Configuración de los tiles (posee tamaño y path de cada una de las tiles del juego)
- Configuración de los autos (tiene la salud, agarre,máxima velocidad y derrape de cada uno)
- Configuración de los sonidos (nuevamente, nombre y path a este)

PROTOCOLO:

Como se explica mejor en la sección Common, se utilizó un simple protocolo de :
_ enviar primero el largo del mensaje
_ enviar luego efectivamente el mensaje

Conscientes de que este no es necesariamente el método más rápido decimos implementarlo y repensarlo de tener problemas de performancia. Al no encontrarnos con esto decidimos mantenerlo.
Algo parecido sucedió con la librería utilizada para YAML, pese a no ser la más rápida, no nos dio problemas de tiempo significantes como para repensar la utilización de esta.