# Documentación Técnica


## Autores:

- Di Maria, Franco Martin, 100498
- Sicardi, Francisco, 99390


## 1. Requerimientos de Software

Se requiere instalar las librerías :

SDL
SDL_mixer
QT5
FFMPEG 3.X

El programa también utiliza las librerías :

YAML
Box2d

Estas ultimos estan integradas como librerías externas en el repositorio del proyecto.

## 2. Descripción general

El proyecto posee 4 módulos principales:

client/
server/
common/
libs/

En el módulo lib/ se encuentran las librerías de Box2d y YAML

## 3. Módulos

### Módulo *common*

#### 1. Descripción general

Este módulo corresponde a las clases que utiliza tanto el cliente como el servidor: sockets, colas, protocolo de comunicación.

#### 2. Clases

***/common***

- BlockingQueue : Es una cola bloqueante
- ThreadSafeQueue : Es una cola no bloqueante pero que segura para comunicar hilos.
- PortalException : Excepcion del juego
- Thread : clase padre para hilos de ejecución

***/common/connector***

- Socket : es un socket generico con métodos para ser usado para un cliente y para un servidor
- Connector : wrapper del socket, con métodos para enviar y recibir distintos tipos de datos
- SocketException : Excepción del socket.

***/common/protocol/event***

- Event : Clase padre de un evento del juego. Los eventos son enviados por el servidor al cliente, para comunicarle cambios en el juego.
- Los eventos tienen un nombre/tipo y pueden contener información adicional necesario para cada evento en específico
- GameStartsEvent : Evento para comunicar el inicio del juego.
- GrabRockEvent : Evento para comunicar que Chell debe agarrar una roca.
- ThrowRockEvent : Evento para comunicar que cierta roca debe ser soltada.
- ObjectSwitchEvent : Evento para comunicar que un objeto del juego de cambio su estado.
- ObjectMovesEvent : Evento para comunicar que un objeto del juego de cambiar su posición en el mapa de juego.
- PlayerDiesEvent : Evento para comunicar que un jugador a muerto
- PlayerLosesEvent : Evento para comunicar que un jugador ha perdido la partida.
- PlayerWinsEvent : Evento para comunicar que un jugador ha ganado la partida

***/common/protocol/game_action***

- GameAction : Clase padre de una acción del juego. Las acciones del juego son enviados por el cliente al servidor, para comunicar un acción a realizar.
- Las acciones del juego llevan un nombre, y pueden contener información adicional necesario para realizar dicha acción.
- CoordsAction : Es una acción del juego que contiene un informacion extra sobre una posicion x,y en el mapa de juego.

## 4. Descripción de archivos y protocolos

***/common/protocol***

- protocol_code.h : contiene varios enum, que se utilizan en la comunicación de entre cliente y servidor, y también entre hilos :
- enum CommandName : uint8_t {…}
    - Se utiliza para comunicar la forma en la que se quiere iniciar un juego: creando un nueva partida o uniéndose a una existente.
- enum JoinGameResponde : uint8_t {…}
    - Se utiliza para comunicar el resultado de la creación y unión a un juego.
- enum EventType : uint8_t {…}
    - Se utiliza para nombrar y comunicar los distintos tipos de eventos provenientes del juego. De esta forma el cliente sabe que evento está recibiendo y que clase usar.
- enum GameActionName : uint8_t {…}
    - Se utiliza para nombrar y comunicar las distintas las acciones hacia el juego. De esta manera, el servidor sabe que acción del juego está recibiendo y clase usar.
- enum ThreadStatus {…}
    - Se utiliza para comunicar el estado de un hilo de ejecución a otro, de forma de saber si ocurrio algun error.

# Módulo *client*

## *1.* Descripción general

Este módulo contiene la implementación de:

la comunicación del lado de cliente;
ejecución del ciclo del juego del lado de cliente;
las animaciones;
sonidos;
interfaz gráfica;

grabación de video

## 2. Clases

### *client/src*

- SdlSystem : Inicializa sistemas de audio y video de sdl asi como de su destruccion.

### *client/src/game*

- Game : Ejecuta el ciclo juego, poniendo a correr los distintos hilos de ejecución necesarios para el mismo, y encargándose de su destrucción.
- Client : Ejecuta el juego habiendo previamente inicializado comunicación con el cliente, y obteniendo datos para inicializar al juego.
- GameFactory : Crea un juego (Game) sin interfaz gráfica, utilizando argumentos del programa. Esta clase eventualmente se dejara de usar.
- GameConfig : Se configura para que luego cree un juego (Game)

### *client/src/user_interface*

- LineInterface : Ejecuta un interfaz de login por comando de línea.
- Login : Funcionará de widget principal para la interfaz de login qt. En él se alternan las distinta widgets de login, con cada una de sus opciones : LoginServer , LoginMode, Login New, LoginJoin.
- GameOver : Widget de fin de partida: muestra los resultados del juego.

***client/src/user_interface/uis :*** contiene las ui de los widgets.

### *client/src/window*

- Window : Ventana principal de juego. En ella estan todas las texturas del juego, que se encarga de administrar y renderizar.
- MapCreator: Se encargar de procesar una configutacion del juego .yaml (mapa del juego), e ir agregando texturas a la ventana (window)
- TextureFactory : Se encarga de devolver punteros unicos de tipo Texture, que almacenen texturas de distinto tipo.
- OSException : Excepcion que utiliza la ventana, y en las clases para grabar video.

### *client/src/window/record*

- OutputFormat : Se encarga de escribir frames de video en memoria. Para ello utiliza clases auxiliares como FormatContext, CodecContextWrapper, SwsContextWrapper, que wrappean clases y conjuntos de funciones de la librería FFMPEG

### *client/src/window/changes*

- Change : representa un cambio del juego que puede afectar a la ventana de juego, al resultado del mismo, o a los controles del usuario.
- De esta clase se derivan clases hijas para cada cambio en particular, por ejemplo: GameStartChange, TextureMoveChange, TextureSwitchChange, PlayerDiesChange, PlayerWinsChange, etc.

### *client/src/mixer*

- MixChunck : Reproduce un efecto de sonido, en un corto periodo de tiempo.
- MixMusic : Reproduce música durante un largo periodo de tiempo. Se utliza para la controlar música de fondo del juego.
- Mixer : Administra varios MixChunck y un MixMusic.
- PortalMixer : Instancia un Mixer con la música y todos los efectos de sonido del juego.

### *client/src/threads*

- PlayingLoopThread : Realiza ciclos en los que procesa input del usuario, aplica cambio

provenientes del servidor en el juego, renderiza la ventana y reproduce sonidos.

- KeySenderThread : Se encarga de enviar acciones del juego (GameAction) al servidor, los cuales recibe del PlayginLoopThread, mediante una cola bloqueante (BlockingQueue)
- EventGameReceiverThread : Se encarga de recibir eventos del juego (del servidor), enviarselos al PlayingLoopThread, mediante una cola no bloqueante (ThreadSafeQueue)
- Estos dos últimos hilos tienen una comunicacion extra con el PlaingLoopThread, mediante una cola no bloqueante del enum ThreadStatus, para indicarle si ocurrio algun error en dichos hilos.
- KeyReader : Procesa input del usuario y lo convierte en acciones del juego que envia al KeySenderThread, por medio de la cola bloqueante. Esta clase se utiliza en PlayingLoopThread.
- EventGameProcessor: Procesa eventos del juego y los aplica en la ventana de juego (Window). Esta clase se utiliza en PlayingLoopThread.
- PlayResult : Contiene información de los jugadores, y sobre el estado del juego. En esta clase se van volcando los resultados del juego.
- VideoRecordThread : Se encarga de recibir buffers llenos de un frame de ventana (cada uno), y escribirlos en memoria. Se realiza a través de una cola bloqueante.

***client/src/textures/common_texture***

- Area : Representa un área (x, y, width, height).
- BigTexture: Almacena una imagen grande con varios sprites incluidos en él. Se encarga de renderizarse en la ventana principal segun le indiquen. Imagínese una imagen grande, separada en recuadros, donde cada uno es una imagen mas chica que representa un objeto del juego, y en algunas casos, el mismo objeto realizando distintas acciones.
- Texture : Representa las características gráficas de un objeto del juego. Administra la ubicacion espacial del objeto del juego, y contiene la información necesaria para indicarle a su correspondiente BigTexture como renderizarse en la ventana. Varios Texture's pueden tener asociado el mismo BigTexture, y renderizarlo de distintas maneras.
- De Texture derivan clases que sobrecargan metodos como ChellTexture, PortalTexture, RecordTexture, etc.
- SpriteStrategy : Se encarga de manejar que serie de sprites (DynamicSprite) utiliza la textura para renderizarse, dado cambios en su estado, posición, etc.
- Dependiendo de como varie las administración de estos sprites se pueden usar clases derivadas como TwoSpritesStrategy, NullEndStrategy, ChellSpriteStrategy, PortalStrategy, etc.
- DynamicSprite : Posee información de una serie de sprites que hacen a una animacion. Los sprites son los recuadros en los que se divide una BigTexture. Esta clase guarda información de las coordenadas de la esquina superior izquierda de ciertos recuadros a usar para cierta animacion, asi como, las dimensiones de cada uno de estos recuadros. Por ejemplo, podríamos tener un DynamicSprite que contenga las coordenadas de cada recuadro de una animación de nuestro personaje principal (Chell) corriendo hacia la derecha.
- <nombre de textura>Sprite : Esta serie de clases, son funciones estáticas, que se encargan de construir DynamicSprite para cada animación en especifica. Por ejemplo, podriamos tener un ChellRunRightSprite que tiene un metodo get_sprite que devuelve un DynamicSprite con las coordenadas de cada Chell corriendo hacia la derecha, para cierta imagen.

En las demás carpetas de ***client/src/textures/*** se encuentran clases especializadas de Texture, SpriteStrategy, y constructores de DynamicSprite

# 3. UML

Diagrama de clases Event (modulo common) en relación con Change(modulo client)

Los eventos (Event) son clases que guardan información de cambios del juego realizados en el servidor y que el cliente debe aplicar en su modelo. Como en el módulo common no puede haber una dependencia de las clases del módulo client, surge un la clase Change. Esta se encarga de aplicar cambios en las distintas partes del modelo de cliente (Window, PlayResult, KeyReader). De la clase

Change heredan varias subclases que aplican cada cambio en especifico. Por ejemplo, tenemos a la clase TextureMoveChange que se aplica un cambio en la posición espacial de un textura que se encuentra en la ventana (Window).
Esta clase recibe a través un Connector, un evento ObjectMovesEvent, del cual extrae la información del cambio a realizar.

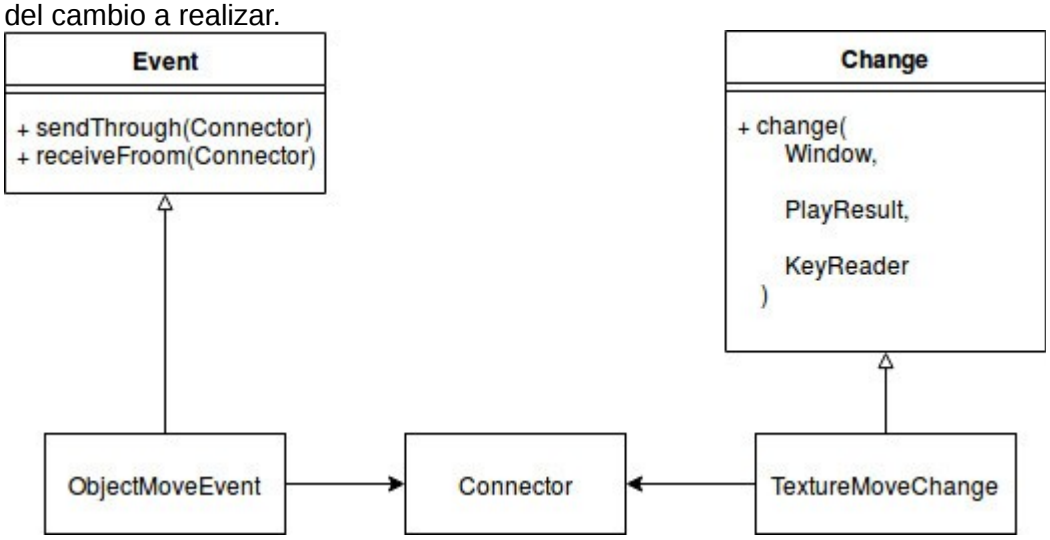



Diagrama de clases de Texture compuesta de SpriteStrategy, compuesta de DynamicSprite.
Las clases Texture son administradores de información gráfica de cada objeto del juego, por ejemplo, saben la rotación, color, e imágenes (sprites) a usar en determinado momento. La mayoría de las texturas, o bien tienen una imagen fija o una animación única. La información de la ubicación espacial de estas imágenes (sprites) se guardan en la clase DynamicSprite que lleva cuenta de que sprite usar a continuación. Algunas clases utilizan más de una animación a lo largo del juego, por ello surge la clase SpriteStrategy la cual administra qué DynamicSprite usar dado el estado actual de un objeto del juego. Las Texture además, guardan información de los distintos sonidos a reproducir según cambios en su estado.
Un ejemplo de texture “especializa” (no todas las texturas heredan de Texture, varias usan directamente la clase padre) es la clase ChellTexture. Esta tiene tener dos estados viva o muerta representados por ChellSpriteStrategy y ChellDeadStrategy respectivamente. Además las animaciones que puede realizar utilizando la primera, pueden ser (en otras) : Chell parada mirando hacia la derecha(ChellStandRightSprite), o Chell corriendo hacia la derecha (ChellRunRightSprite). Estas dos últimas clases no son DynamicSprite, sino que contienen métodos estáticos para construir el DynamicSprite necesario para cierta animación.
La orientación de hacia dónde voltear la animación (izquierda o derecha) se guarda en ChellTexture.

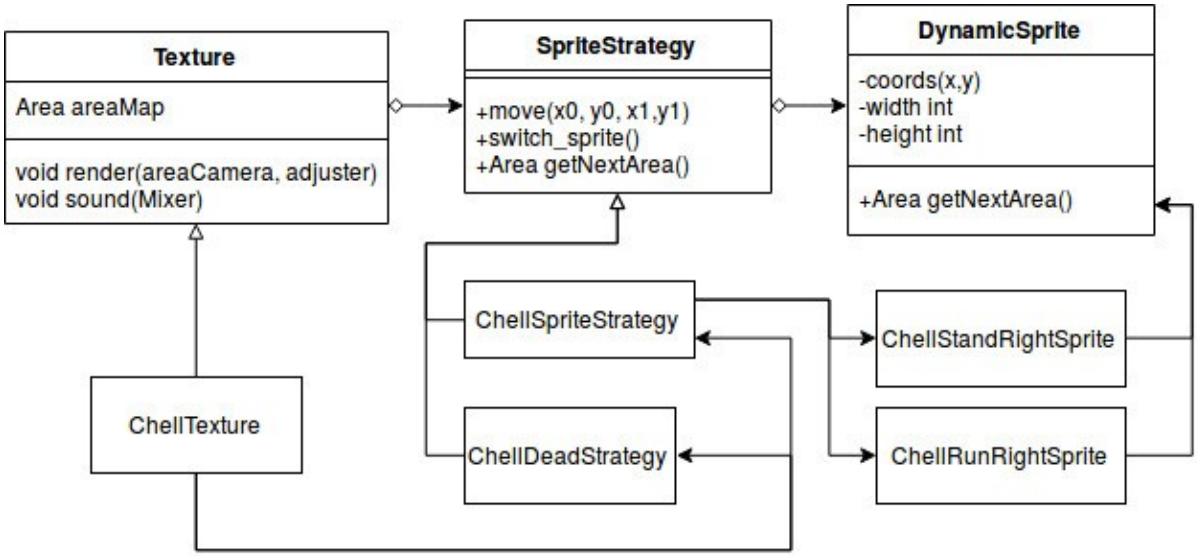



Diagrama de clases y de relación de hilos de ejecución en el ciclo del juego.

Game instancia los hilos EventGameReceiverThread, KeySenderThread y VideoRecordThread. Los dos primeros reciben y envían eventos y acciones, respectivamente, a través del Connector que está compuesto por un Socket. El último recibe buffers con el frames de video a guardar en memoria.

Game instancia en el mismo hilo principal PlayingLoopThread. Este se comunica, respectivamente, con los ya mencionados hilos anteriores, por medio de una colas no bloqueantes y bloqueantes, según se explica a continuación.

EventGameReceiverThread recibe eventos (Event) del juego (del servidor), los transforma a cambios del juego (Change), y los encola en una cola segura. De esta cola, el PlayingLoopThread procesar cambios. Tras (o antes de) procesar estos cambios procesa eventos (input) del usuario, y encola acciones (GameAction) en un cola bloqueante. De esta cola bloqueante el KeySenderThread envía a través del Connector, acciones a realizar por el jugador actual, al servidor.

Durante el ciclo de procesamiento, renderización, etc.., el PlayerLoopThread, si la aplicación está en modo de grabación, envía un buffer con frames de video a grabar en el disco, a través de una cola bloqueante al VideoRecordThread para que se encargue de ello.

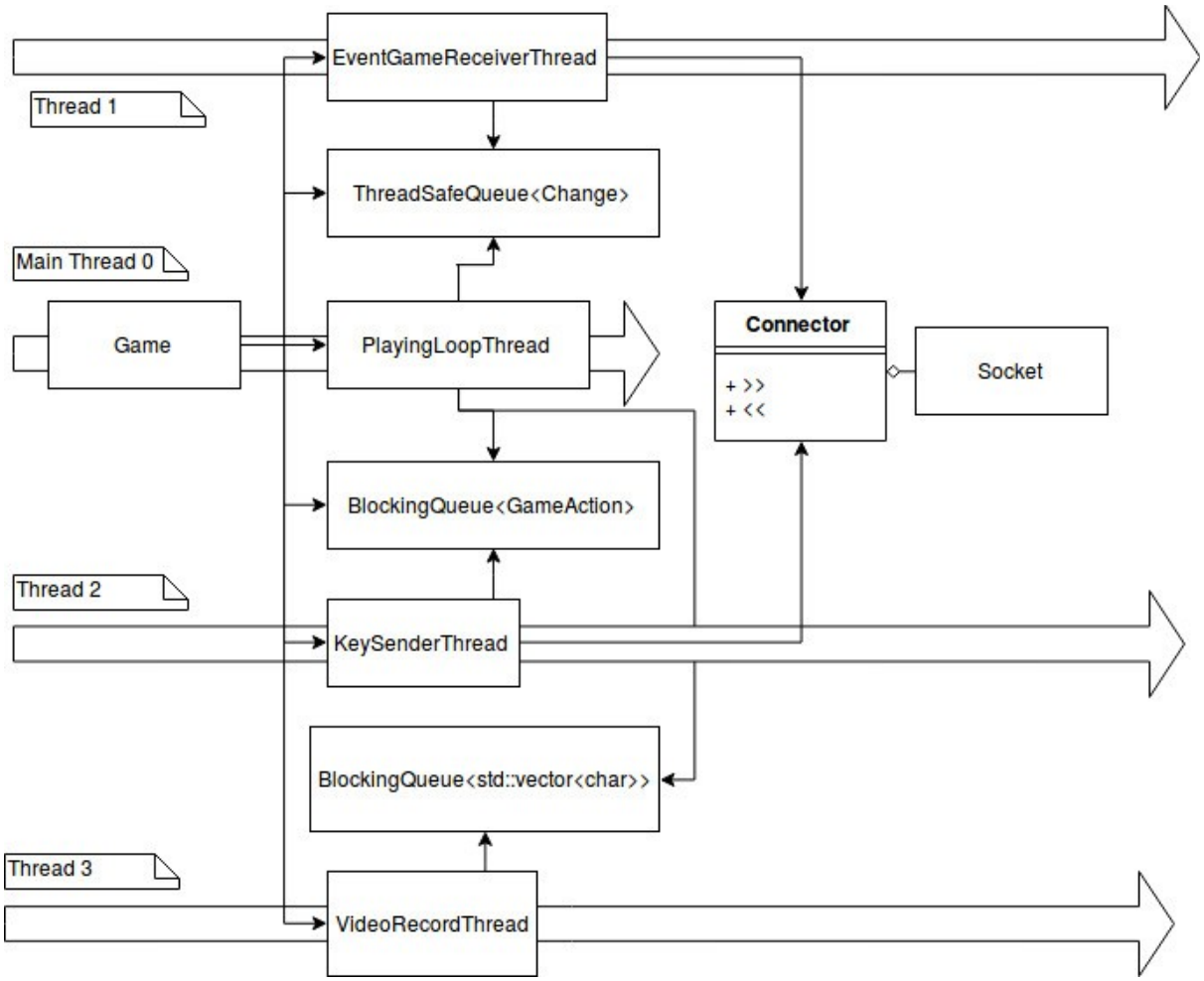



Diagrama de Window

Window encapsula la SDL_Window , el SDL_Renderer.

Tiene un metodo para agregar un puntero único a Texture. La clases Texture se inicializa a partir de una BigTexture.

Las BigTextures encapsulan una SDL_Texture, y que tienen un puntero al SDL_Renderer de la ventana, para poder renderizar por sí mismos.

Al agregar un BigTexture a la ventana, esta devuelve una referencia al BigTexture agregado. Esto permite que las texturas se inicialicen fuera de la ventana, se guarden en wrapper std::unique_ptr<Texture> para luego agregarla a la misma. Lo que se logra, es quitarle carga a la ventana y poder ponerla en las clases MapCreator y TextureFactory que tienen muchos métodos para inicializar y agregar cada textura específica a la ventana.

La función principal de la ventana es administrar la forma en que se produce cambio en ella así como los cambios en las texturas. Además, no está de más decir que es la encargada de renderizar todas las texturas en la sección del mapa que la cámara de juego indica en determinado momento.

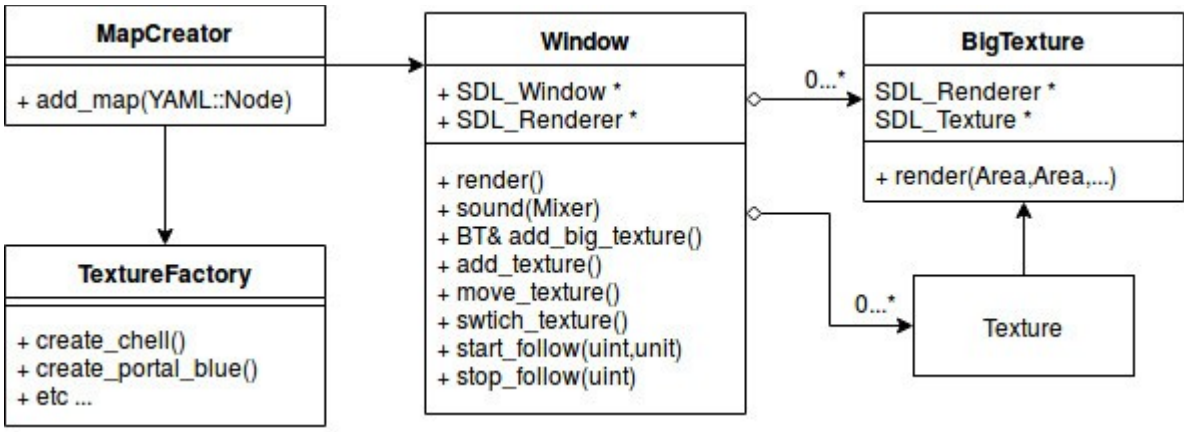



Diagrama de Mixer

El MixChunck encapsula un Mix_Chunk * (accidentalemente se utilizó un letra c de mas en el nombre de la clase). Puede reproducir un sonido durante un corto periodo.
El MixMusic encapsula un Mix_Music * . Reproduce música hasta que indicarle detenerse.
El Mixer esta compuesto de 1 solo MixMusic, pero puede tener varios MixChunck. Se encarga, de inicializar el Mix y de manejar todos los sonidos del juego.

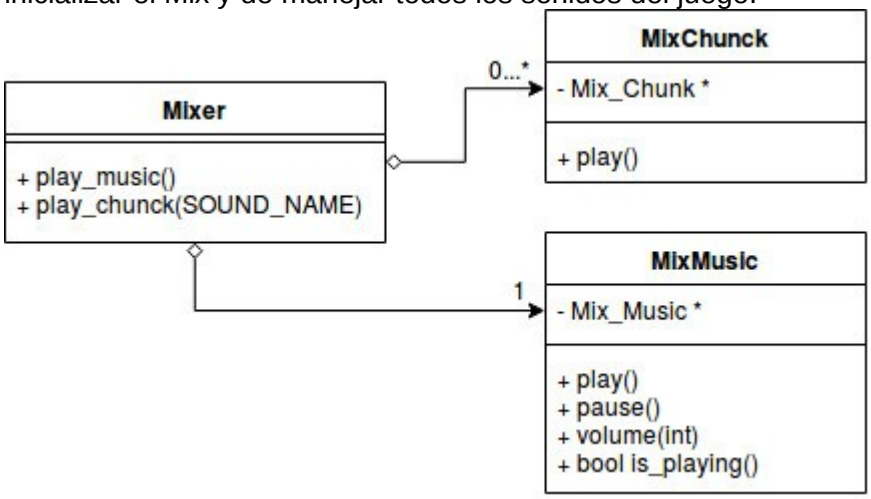



Diagrama de clases interno de PlayingLoopThread

PlayingLoopThread esta compuesto de EventGameReceiver y KeyReader como antes mencionamos. Además, tiene referencias a la ventana del juego (Window), al reproductor de sonidos del mismo (Mixer), y al resultado del juego PlayResult. KeyReader tiene referencia los dos primeros, pues extrae coordenadas de la ventana, y le permite dejar que el usuario pause y continúe la música a voluntad. EventGameProcessor refiere a los tres pues los cambios a procesar pueden incluir cambios en la ventana, en el resultado del juego, o en los controles del usuario. Con respecto a esto último, si el jugador muere, se desactivan los controles de movimiento, portales, etc del usuario, solo dejándole, salir del juego, grabar video, y reproducir música.

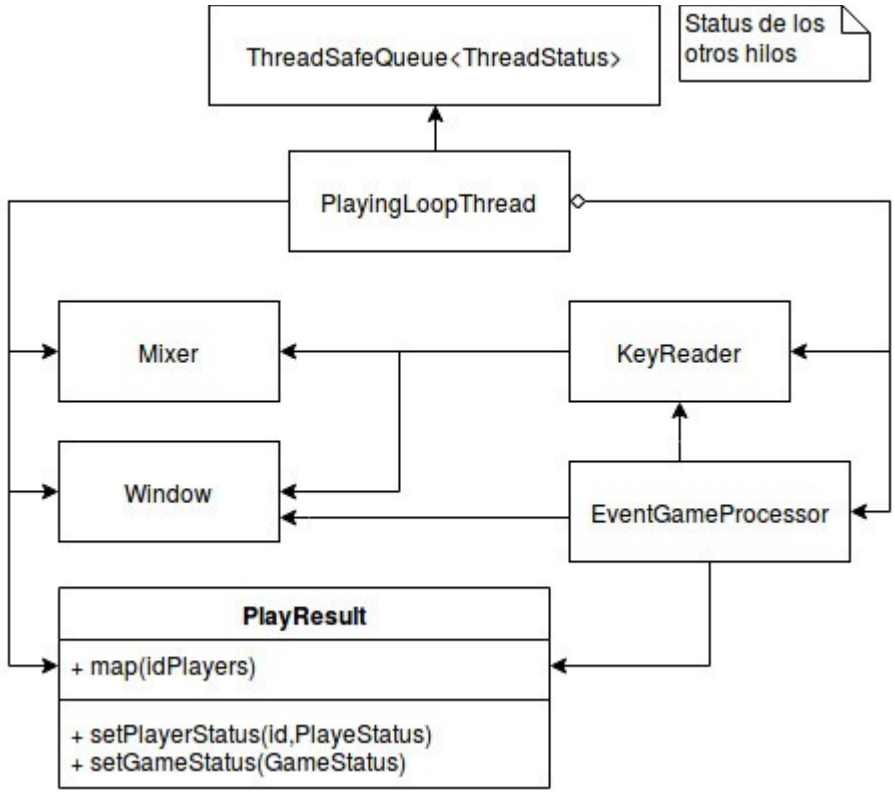
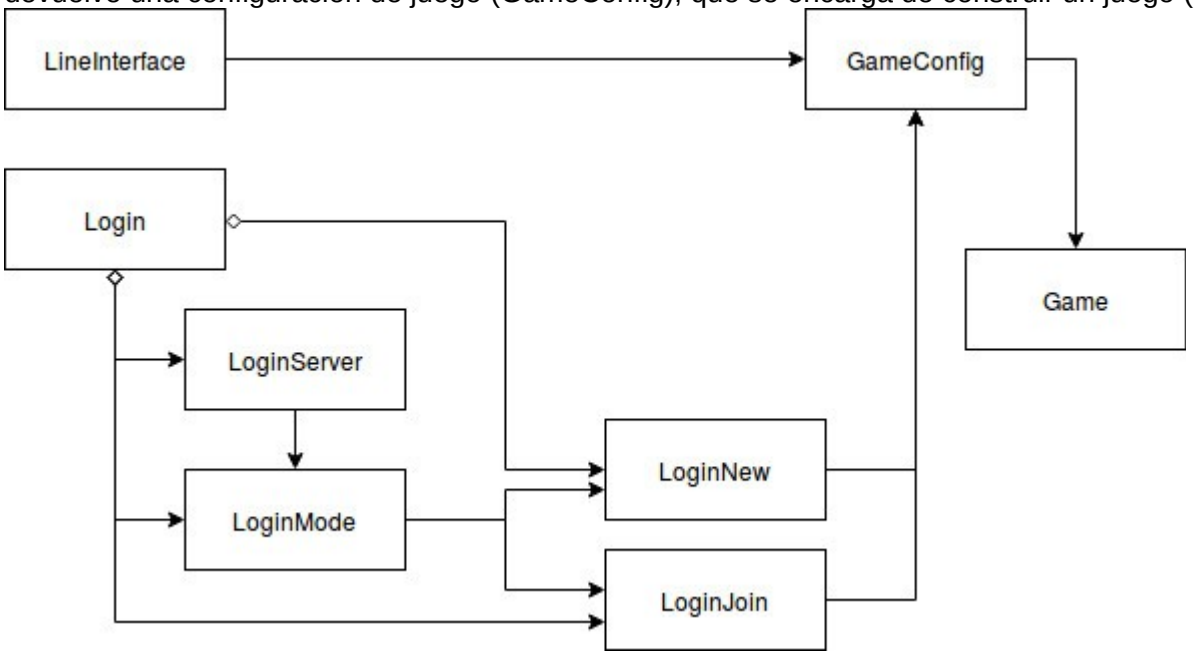



Diagrama de interfaces de usuario, y creación de un juego

Existen dos interfaces de usuario: la interfaz por línea de comando (LineInterface), y la interfaz por ventanas QT. Esta última es conformada por la clase Login, que es un QWidget principal, que administra mostrar, ocultar y cerrar el resto de los QWidget que hacen a cada parte de la interfaz : LoginServer, LoginMode, LoginNew, LoginJoin. Todos estos últimos a su vez se conocen, pues van volcando los datos recibidos por el usuario, de widget a widget. Finalmente, ambas interfaces devuelve una configuración de juego (GameConfig), que se encarga de construir un juego (Game).




## 4. Descripción de archivos y protocolos

***client/includes/textures/common_texture/images_path.h***

Posee un mapa (std::map) constante con los paths de las imágenes del juego, con el que se cargan

las distintas BigTexture para cada set de sprites.

***client/includes/textures/mixer/sounds_path.h***
Posee un mapa (std::map) constante con los path a cada sonido a reproducir

***client/includes/render_time.h***
Posee definiciones del tiempo mínimo que espera PlayingLoopThread en cada ciclo de juego (frames + sonidos + input del usuario)

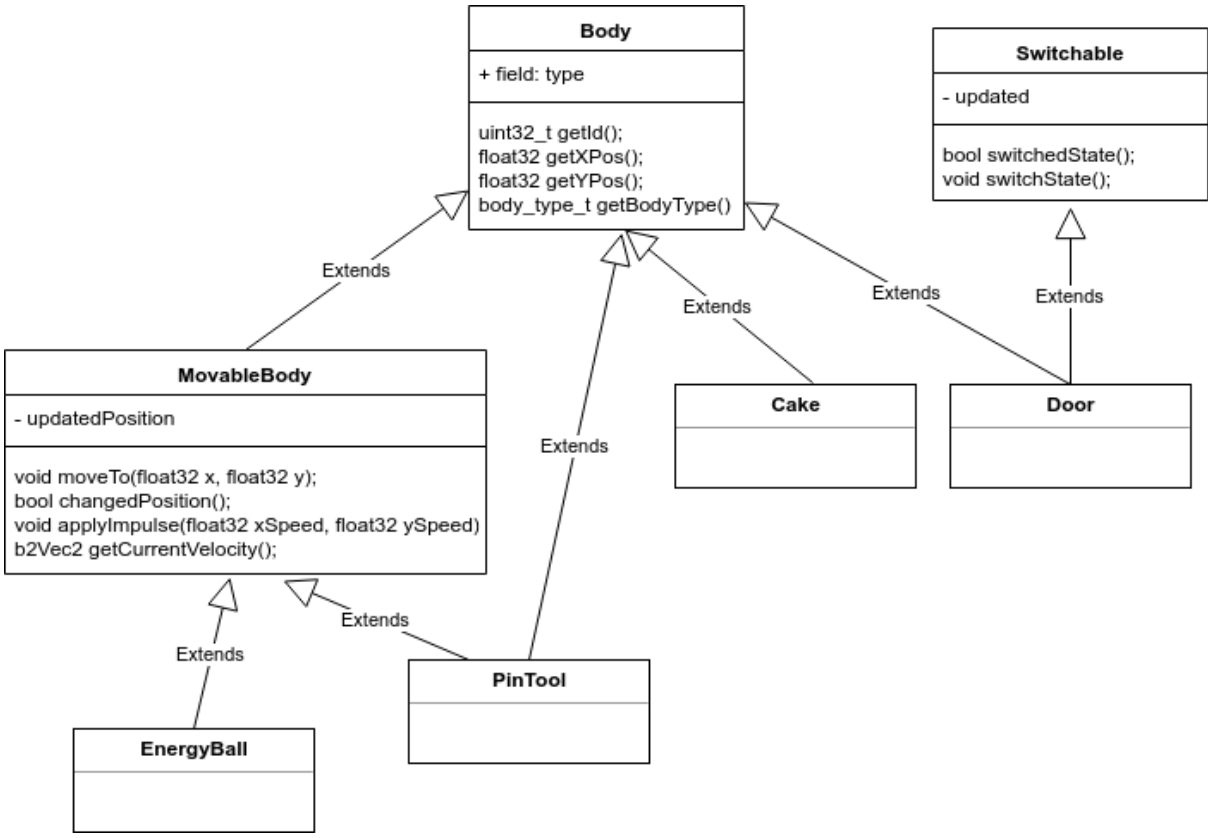
***client/assets***
Aquí se guardan todas las imágenes utilizadas en el juego.

# Módulo *server*

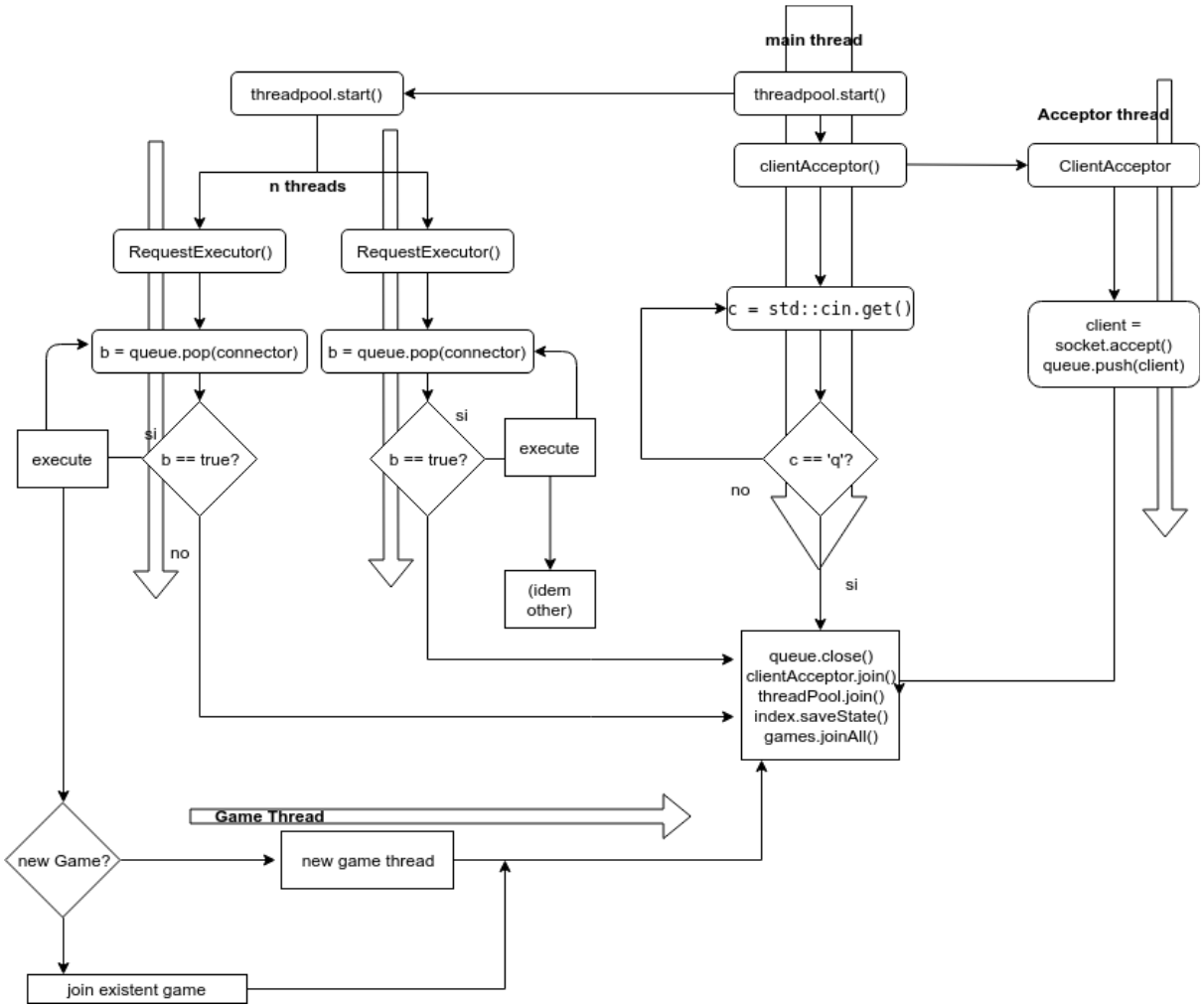
## *1.* Descripción general
Este módulo contiene la implementación de un servidor multipartida y multijugador del juego, que maneja la comunicación con todos los clientes y la implementación de la lógica del juego. Para el manejo de la física del juego se utilizó la librería Box2D recomendada por la cátedra.

## 2. Clases
- Server: clase principal del programa.
- ClientAcceptor: thread encargado de aceptar a los clientes y encolarlos.
- RequestExecutor: thread que se encarga de desencolar clientes y pasarlos al GameManager, indicándole si quieren crear un nuevo juego o unirse a uno existente.
- ThreadPool: contiene varios RequestExecutors que van recibiendo y procesando los requests. En clase nos marcaron que era una solución sobrediseñada y que no hacía falta, pero por falta de tiempo y ya que andaba bien decidimos por ahora dejarlo así y enfocarnos en los features que nos faltaban.
- GameManager: se encarga de guardar los distintos GameLobby de cada partida, crear nuevos y agregar clientes a los existentes, interactuando con el cliente para pedirle la información necesaria. En caso de error (por ejemplo partida inexistente o partida llena), envía códigos de error que son interpretados por el cliente.
- GameLoby: contiene el juego y agrega los jugadores a medida que llegan, puede ser accedido concurrentemente y por lo tanto está protegido. Inicia nuevo juego cuando se llega a la cantidad de jugadores necesaria.
- Game: se lanza en un thread nuevo, y se apropia de los conectores para comunicarse con los jugadores. Corre el ciclo de juego, en el cual recibe eventos, los procesa y luego envía actualizaciones a todos los jugadores.
- Map: se encarga de obtener la información del mapa de un archivo YAML.
- World: representa al mundo de Box2D y encapsula la física y la lógica del juego.
- Player: clase que encapsula la comunicación con el jugador desde que comienza la partida. Utiliza dos threads: el primero que recibe eventos del cliente y los encola en la cola del Game para que sean procesados, y el segundo desencola eventos de una cola propia y los envía al cliente.
- En la carpeta modelo, están todas las clases que encapsulan los objetos de Box2D. La clase abstracta Body contiene el id y los atributos de box2d. La clase abstracta MovableBody hereda de Body y agrega métodos para detectar y/o producir movimiento en esos objetos.
  La clase abstracta Swtichable provee una abstracción para los objetos que necesitan cambiar entre 2 estados, agregando métodos para producir y detectar ese cambio de estado.
  Todos las clases del modelo heredan de Body o MovableBody, y algunas además de Switchable.
  Esto permite que la clase world las abstraiga y las trate de la misma forma.
- ContactListener: implementa distintos callbacks que Box2D llama al iniciar y terminar un contacto, para especificar el comportamiento del choque.

- ContactFilter: implementa distintos callbacks que Box2D llama para determinar si se debe producir o no un choque.
- PortalRaycastCallback: implementa un callback que es utilizado por el RayCast de box2D, se utiliza para abrir los portales.
- GlobalConfiguration: provee un acceso global a las configuraciones del juego.
- La carpeta client_action contiene las clases que representan las acciones enviadas por el cliente y saben como ejecutarlas. Todas heredan o de ClientAction (si sólo depende del nombre) o de ClientCoordAction (si además depende de las coordenadas). A modo de ejemplo:




## 2. Esquema del manejo de concurrencia




# 4. Programas intermedios y de prueba

No se utilizaron programas de prueba.
El código fue avanzando poco a poco con pruebas individuales sin usar ningún framework, que nos sirvieron para orientarnos en el uso de las distintas librerías.
Estas pruebas se volvieron obsoletas con los cambios en el programa, y ya no están en el proyecto.

# 5. Código Fuente

Repositorio en Github. Solicitar invitación a creadores del juego: fsicardir / dima1997