Taller de Programación I (75.42 - 95.08)

Ejercicio Final - Portal

Documentación Técnica

1er Cuatrimestre 2019

Integrantes	
Alumno	**Padrón**
Sebastián Ignacio Penna	98752
Jonathan Claudio Medina	100052

Requerimientos de Software

El proyecto fue desarrollado por ambos integrantes en un sistema Ubuntu 18.04, por lo tanto el mismo será funcional dentro de los distintos sistemas Unix. Las herramientas necesarias para la correcta ejecución del juego, tanto cliente como servidor, y los distintos tests realizados son: CMake, SDL, QT, Box2D, CppUnit, yaml-cpp y ffmpeg. Algunas de estas deben ser instaladas por el usuario, proceso que se explicará posteriormente. La instalación de las mismas se puede realizar de la siguiente manera:

1. Instalar CMake:

```
sudo apt-get install cmake
```

2. Instalar SDL, y herramientas que utiliza:

```
sudo apt-get install libsdl2-dev
sudo apt-get install libsdl2-image-dev
sudo apt-get install libsdl2-ttf-dev
sudo apt-get install libsdl2-mixer-dev
```

3. Instalar QT:

```
sudo apt-get install qt5-default
```

4. Instalar ffmpeg, requiriendo una serie de bibliotecas:

```
sudo apt-get install libavutil-dev
sudo apt-get install libswresample-dev
sudo apt-get install libavformat-dev
sudo apt-get install libavcodec-dev
```

5. Instalar CPPUnit:

```
sudo apt-get install libcppunit-dev
```

Descripción General

El proyecto está dividido en dos secciones bien diferenciadas: el servidor, quién se carga de soportar conexiones y partidas así como mantener el modelo del juego, y el cliente, el cuál ejecuta el juego y permite que el usuario interactúe con el mismo. Ambas secciones están conectadas por medio de un protocolo definido por los alumnos, que permite el correcto envío y recepción de datos tanto previo, durante y posterior al juego.

Ninguna de ambas partes tiene conocimiento de cómo ésta implementada la otra, la única consigna que se debe respetar es utilizar el mismo protocolo, para garantizar que no haya fallas en el funcionamiento.

El proyecto se generó en lenguaje C++, utilizando estructuras como colas protegidas y bloqueantes, sockets TCP y threads, entre otras. Para la simulación física se utiliza la librería Box2D, totalmente encapsulada dentro del modelo, mientras que las tareas visuales

se generaron con la biblioteca SDL y QT, ésta en particular para la ventana de conexión e inicio del juego para el cliente.

Para la creación de mapas y cambios en la configuración se utilizó el lenguaje YAML, siendo útil la librería yaml-cpp para parsear dichos archivos y obtener los datos contenidos en ellos.

A continuación se explicarán detalladamente las distintas secciones, estructuras y bloques de código que tengan relevancia en el transcurso del trabajo.

Servidor

La sección del servidor tendrá como se mencionó previamente la responsabilidad de mantener abierto el canal para que los distintos jugadores puedan crear o unirse a partidas y mantener el modelo correspondiente a cada partida (para cada partida se tendrá un modelo de juego diferenciado).

Modelo

Dentro del modelo todos aquellos elementos que participan de la partida y aparezcan en la representación visual del juego heredan de una clase madre Collidable. De ésta manera se establecen métodos a respetar y utilizar por medio de todos los distintos objetos, reconocibles a través de un ID respectivo a cada clase que se establecerá como una macro.

World

Ésta clase representa el mundo de Box2D, es decir, donde se crearán y existirán los distintos objetos con los que el jugador interactúa. Será la encargada de almacenar en distintas estructuras (como vectores o mapas) todos aquellos objetos que se generen desde el archivo de configuración de YAML (no es responsabilidad de ésta clase parsear el mismo) como los distintos objetos que se puedan crear en el transcurso del juego, como los portales, pin tools o bolas de energía.

World funcionará como un intermediario entre el gameloop y los distintos elementos de la partida, generalmente delega la responsabilidad de las distintas acciones a otras clases, como pueden ser la creación de objetos, o el step dentro de la simulación física, entre otras. La única responsabilidad de creación de objetos que se le asignó a ésta clase es la de las bolas de energía, el resto de las acciones que realiza corresponden a quitar o agregar elementos dentro de las estructuras de almacenamiento.

Dos estructuras de almacenamiento relevantes son los vectores de *_objects_to_update* y *_objects_to_delete*. En cada uno de ellos se almacenarán aquellas entidades que durante un step realizaron alguna acción, diferente cada una para las distintas clases, o que deben ser eliminados de la partida ya que murieron por alguna de las distintas razones que existen en el juego. Ambos serán de gran utilidad para notificar a los clientes jugando en la partida como deben actualizar sus views.

También se almacenarán aquellos id’s de las chells que hayan decidido matar a otra para poder decidir si es posible efectivamente eliminarla del juego y así lograr la victoria.

RayCastCallback

Esta clase hereda de b2RayCastCallback (propia de Box2D) y permite la detección del primer cuerpo que colisionó con un disparo, pensada para la acción de disparar pin tools y portales.

ContactListener

Esta clase hereda de b2ContactListener (propia de Box2D) y permitirá la personalización de las distintas acciones a la hora de colisionar distintos cuerpos.

Dentro de la misma se realizará override de tres funciones:

- BeginContact: una vez que están colisionando dos cuerpos se notificará a los mismos con qué cuerpo colisionaron, delegando la responsabilidad a cada Collidable de realizar la acción que le corresponda en base a qué tipo de objeto sea el otro;
- EndContact: misma idea que en BeginContact, pero en éste caso se evaluará qué acción realizar al finalizar un contacto;
- PreSolve: mediante éste método el modelo tiene la capacidad de prevenir ciertos contactos previo a que sucedan. De ésta manera se podrá tener la opción de que algunos cuerpos ignoren la existencia de otros y no se tengan choques que modificarán la forma en que se desarrolla el juego. A diferencia de los primeros dos, éste método si se encargará de interactuar con algunos de los objetos para evaluar su estado y en base a eso definir qué acción realizar, como podría ser el caso de una compuerta que podría estar tanto cerrada como abierta, y en base a eso un cuerpo debería pasar por ella o no.

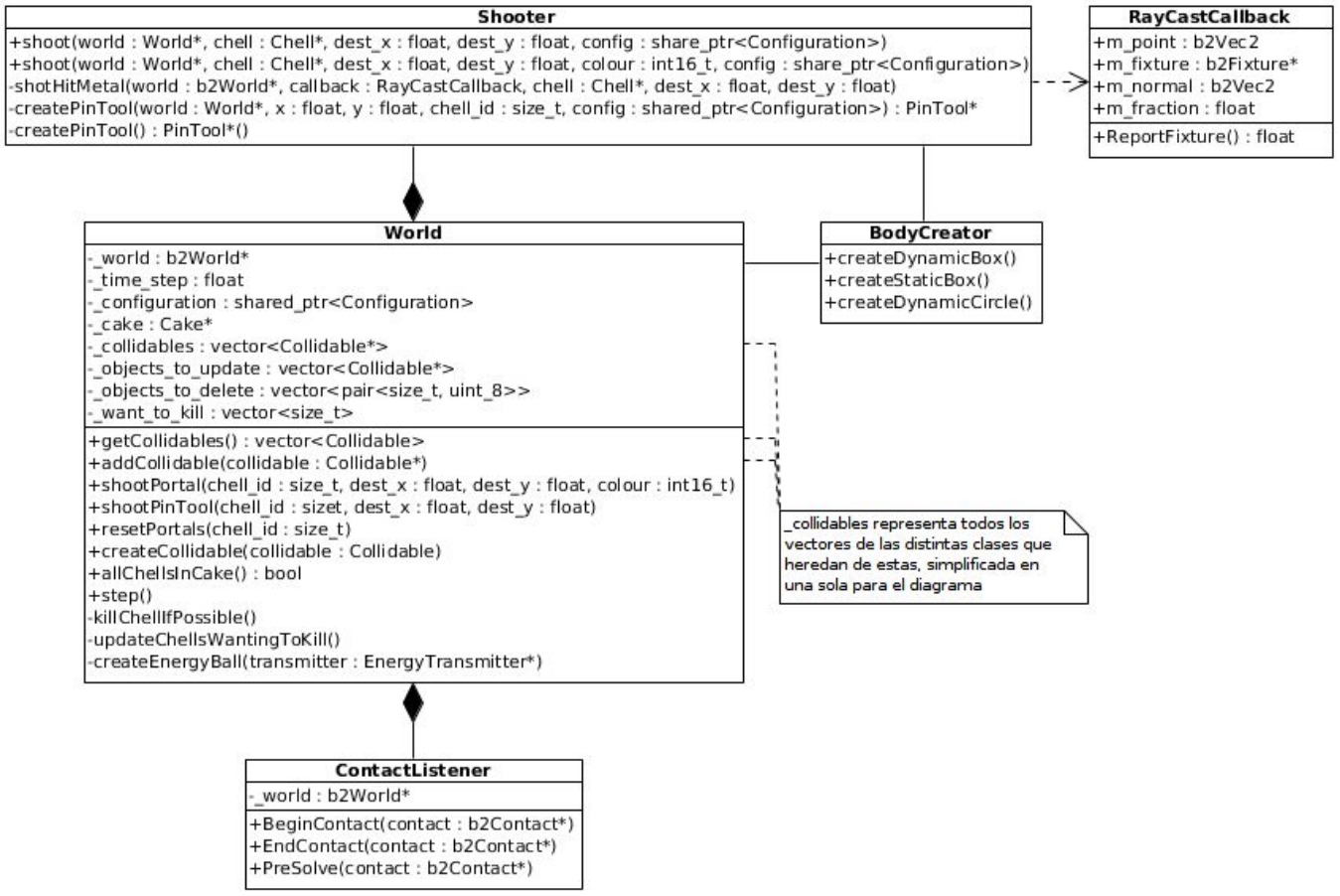
BoxCreator

Ésta clase es responsable de crear los fixtures y shapes de Box2D para generar los distintos bodies que serán agregados al world que le fue indicado. De ésta forma toda la creación de cuerpos ésta unificada en una sola clase.

Shooter

Esta clase encapsula el disparo a realizar por parte de un personaje (ya sea de portal o pin tool) y la detección del cuerpo con el colisionó, para saber si se trata de un bloque de metal donde se pueda generar la nueva entidad.

En caso de ser posible la creación de un objeto fuente del disparo, Shooter se encarga de crear los cuerpos y definir correctamente su posición y orientación, como es el caso de los portales que podrían ser creados sobre superficies diagonales. Todos estos cálculos se realizarán en base a la normal obtenida del punto de colisión y la posición del bloque donde acabó el disparo, para así poder representar el portal o pin tool en el centro de la cara del mismo.



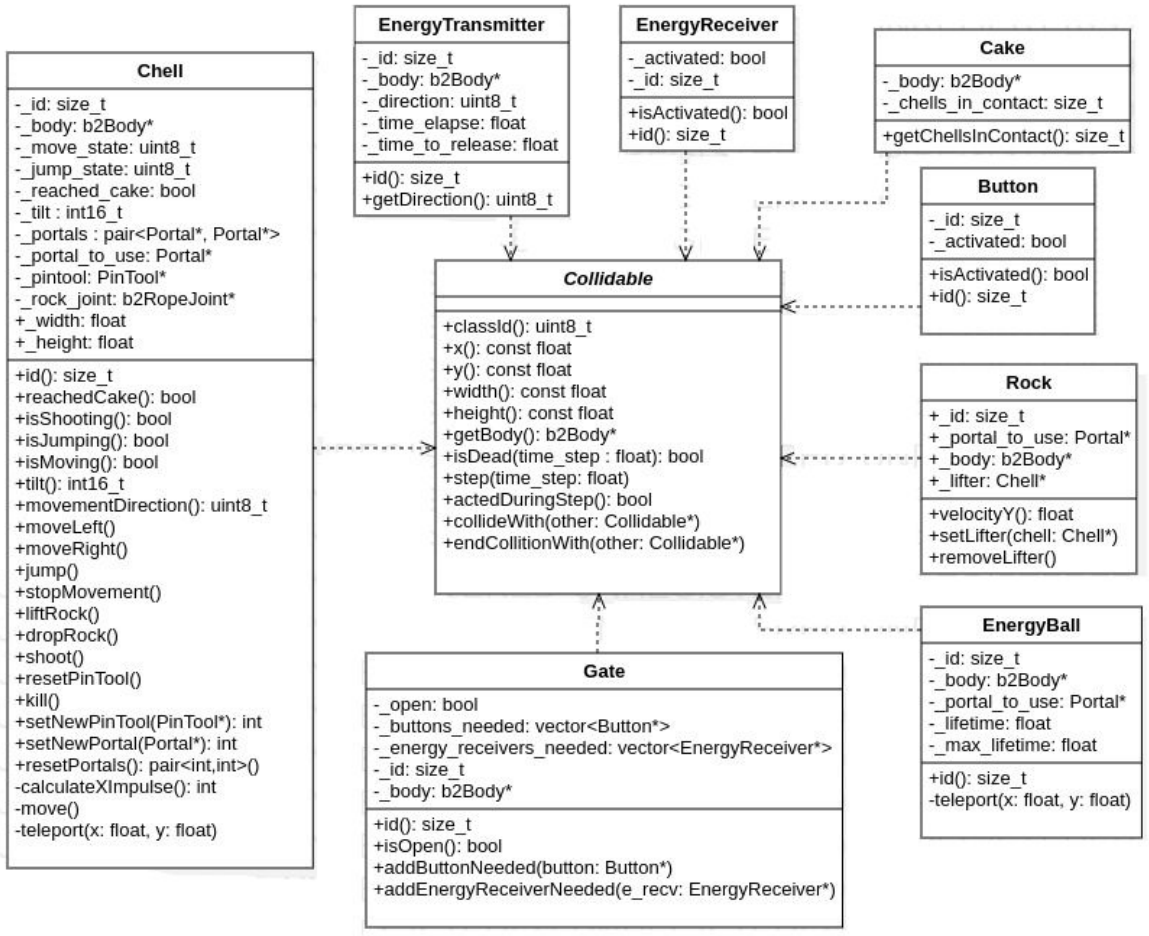
Collidable

A continuación se dará una lista de los distintos elementos que forman parte del juego y heredan de ésta clase. El funcionamiento y secuencia de tareas en los mismos se mostrará a través de diagramas posteriormente. Aquellos cuerpos que deban ser seleccionados particularmente para llevar a cabo ciertas acciones, como es el caso de las chell, botones, receptores, etc. tendrán entre sus atributos un id que permita diferenciarlos.

- **Chell:** personaje que utilizará el jugador en el transcurso del juego. Es responsable de aplicar todos los movimientos que el jugador requiera, así como la teletransportación. Internamente llevará un registro de su estado (teniendo éste distintas variables) a través de booleanos, los cuales son utilizados para definir el comportamiento de la misma y llevar a cabo los movimientos deseados, entre otras acciones. Se diferenciarán entre todas por medio de un id.
- **EneryBall:** lleva un registro de su tiempo de vida para saber cuándo debe ser eliminada. Al igual que la chell es responsable de su teletransportación;
- **Rock:** también es capaz de atravesar un portal, siendo responsable de su teletransportación. Por esta razón tendrá un puntero a una Chell, en caso que esté siendo levantada, y de esta forma se podría teletransportar tanto la roca como la chell una vez que se encuentren con un portal;

- **EnergyReceiver:** a través de un booleano guarda su estado, para notificar al modelo cuando fue activada por una bola de energía;
- **EnergyTransmitter:** al igual que la bola de energía internamente contabiliza el tiempo transcurrido para saber cuando debe indicar a world que debe crear una bola de energía;
- **Cake:** llevará un registro de la cantidad de chells que están en contacto con la misma a través del comienzo y fin de colisiones que se detecten. De ésta manera se podrá saber en qué momento de la partida los jugadores ganaron.
- **Button:** por medio de las colisiones que se detecten se modificará su estado, es decir, si está siendo o no presionado;
- **Gate:** una gate tendrá un vector de punteros tanto de botones como receptores que sean responsables modificar su estado. De esta manera al realizar un step la misma podrá saber si debe abrir o cerrarse, según cómo esté diseñada su lógica;
- **PinTool:** al igual que la bola de energía contiene un contador para saber en qué momento debe ser eliminado del juego. Almacena también el id de la chell que la creó, para posteriormente notificar a la misma que se eliminó su pin tool cuando acabó su tiempo de vida;
- **Portal:** un atributo particular de éstas entidades será el portal de salida, para saber adonde se debe teletransportar el cuerpo que pase por el mismo;
- **MetalBlock;**
- **RockBlock;**
- **MetlaDiagonalBlock;**
- **Acid;**
- **EnergyBarrier;**

A continuación se realizará un breve explicación de la función de cada método dentro de Collidable.



classId

Como se mencionó previamente, cada clase tiene un ID que la identifica y este método lo retorna.

step

Cada clase procesa de distinta manera el step de simulación del world.

actedDuringStep

Mediante éste método se sabrá si el objeto realizó alguna acción durante el step, ya sea un movimiento, activar en el caso de los botones o receptores o abrir y cerrar en el caso de las compuertas, por ejemplo. Luego se accederá a cada método particular de los distintos objetos en base a este resultado.

collideWith - endCollition

A través de éstos dos métodos funcionará la lógica de colisiones dentro del modelo para cada collidable.

YamlParser

Parser para los archivos conteniendo los mapas del juego, utiliza internamente la librería yaml-cpp. Cuenta con un método *parse* que se encarga de cargar todos los datos, excepto aquellos de la Chell que deben ser cargados por separados y almacenados en otra estructura, dado que inicialmente no se conocen cuantas chell se tendrán en el juego y quizas no todas las que aparezcan en el archivo YAML se utilicen.

CollidableData

Ésta clase abstracta representa a todos aquellos elementos que obtienen sus datos a través del archivo YAML del mapa. Cada uno de éstos deberá implementar el método ***build,*** el cual se espera cree el body de Box2D que corresponda, almacene la user data en el mismo y le indique a world que almacene el nuevo dato.

Configuration

Ésta clase consistirá tan solo de getters, dado que la misma se cargará con un archivo YAML con todos aquellos datos configurables que afecten al comportamiento del modelo Box2D al construirse. Luego se la utilizará por medio de un shared_ptr que compartirán todas las partidas que se encuentren en ejecución simultáneamente.

Eventos y generación DTOs

Event

Ésta clase corresponde a los elementos que se recibirán por parte del cliente y serán almacenados en una SafeQueue (se la mencionará en secciones siguientes), dado que se necesita tener tanto el DTO recibido como el id del jugador quien envió dicha información, conocido dado que cad jugador tiene un hilo separado.

PositionTranslator

Este traductor se encarga de convertir las posiciones que se utilizan en el modelo de Box2D en aquellas que se definieron en el protocolo (ver sección Protocolo para mayores especificaciones) y será utilizado previo a generar los DTOs a enviar al cliente.

WorldObjectDTOTranslator

Ésta clase es la responsable de generar los DTO que serán luego enviados por el socket a partir de los Collidable dentro de World. Estos DTO a enviar serán tanto aquellos de la configuración inicial del mapa como las distintas actualizaciones que tengan los objetos. En primer lugar se encarga de convertir la posición que se tiene en el modelo a

aquella definida dentro del protocolo y luego, en base a la clase del Collidable, obtendrá cada valor de su estado que se requiera enviar y lo agrega al nuevo DTO generado.

Tendrá un segundo método que generará los DTO de aquellas entidades que ya fueron eliminadas del world de Box2D y deben ser removidas de la vista. En éste caso recibirá el id del elemento eliminado (todos las clases de aquellos objetos que podrían ser eliminados contienen un id para diferenciar unos de otros) y el id de la clase de dicho elemento. De esta manera podrá generar el DTO correspondiente a la clase, pero en este caso activará el flag de *delete* (ver sección Protocolo) y asignará el resto en 0 por default, excepto el id de la entidad entregado.

DTOProcessor

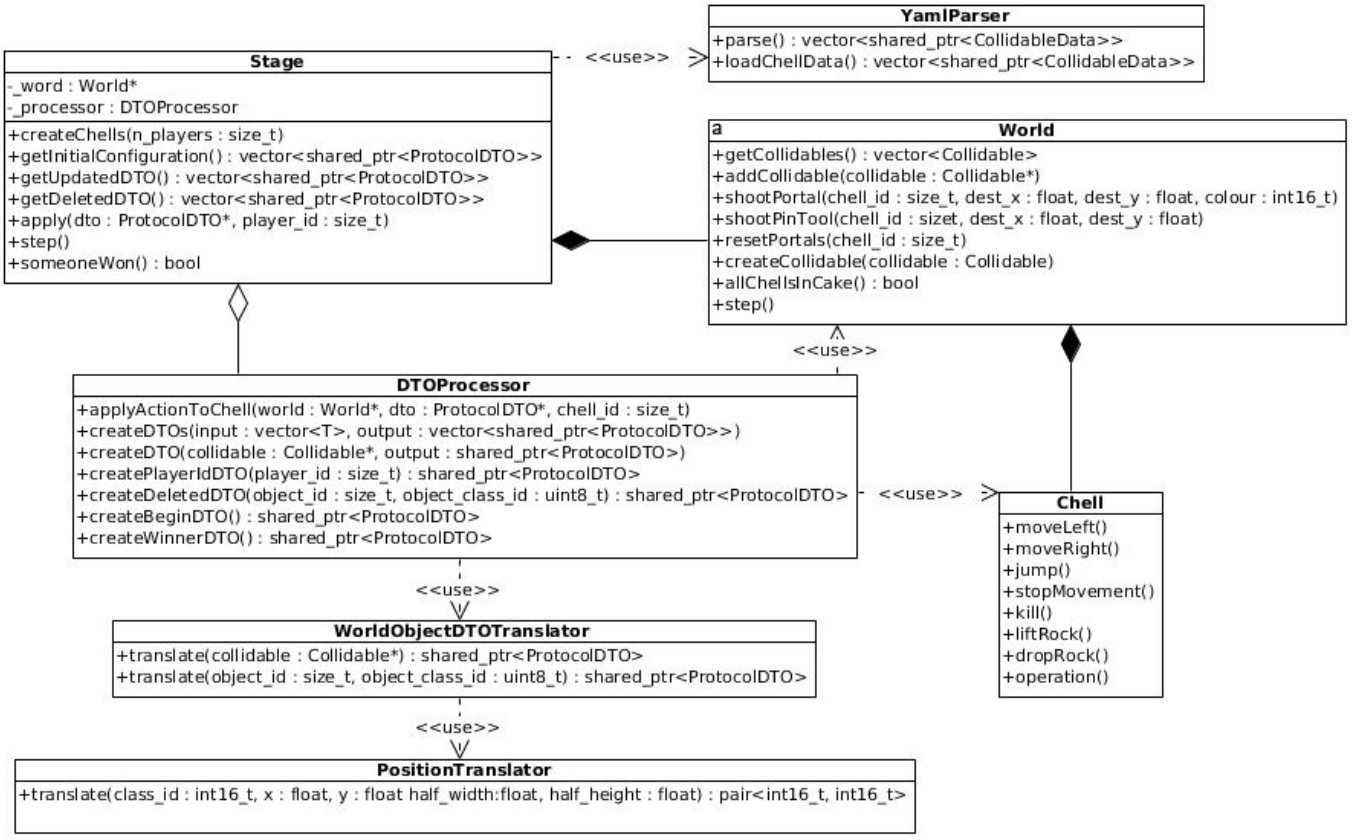
Ésta clase funciona como intermediaria entre el Stage (se explicará más adelante en este informe) y el WorldObjectDTOTranslator, para encapsular el mismo. Sus responsabilidades serán en primer lugar utilizar el traductor ya mencionado para entregar los DTO a quien lo utilice, generar DTOs más simple de configuración (ver Protocolo) que no involucran a Collidable's y por último se encarga de aplicar las acciones que lleguen a través de DTOs por parte del cliente. En base al id del DTO decodifica cual es la acción a realizar y le indicará al modelo lo que el cliente desea hacer.

Stage

Ésta clase será el punto medio final entre el gameloop y el modelo del juego. Cada partida tendrá una instancia de ésta clase la cual contendrá un puntero a World, el cual contiene el modelo.

En su constructor se le brindará el mapa seleccionado para la partida y a traves del parser de YAML cargará la configuración en un vector, el cuál luego iterará para que el world se encargue de crear los distintos objetos.

Previo al comienzo del juego y en el transcurso del mismo obtendrá los distintos DTOs que requieran ser enviados y se los brindará al gameloop.



Conexión con Clientes

Ya se desarrolló sobre aquellas clases que componen e interactúan con el modelo. Ahora se explicará cómo se diseñó y dividió el servidor que soporta múltiples clientes y múltiples partidas simultáneamente, así como los distintos hilos que estos generan, dado que se tendrán gran cantidad de threads al mismo tiempo para poder llevar a cabo la tarea deseada,

Lobby

El lobby será el encargado de recibir y aceptar conexiones. Cada vez que reciba una nueva conexión creará un nuevo Player, que como se explicará a continuación define si se debe crear una nueva partida o unirse a una existente, tareas de las que es responsable el lobby.

Ésta clase tendrá un vector de GameThreads, siendo éstas todas las partidas que están en funcionamiento. También cuenta con un vector de jugadores que se encuentran aún en el lobby, del cuál se los retirará una vez que se haya unido a una partida y pasará a ser responsabilidad de otra clase.

Para evitar el crecimiento infinito de éste vector de partidas se ejecuta un thread donde se controla si las distintas partidas continúan con jugadores dentro de ella o

finalizaron, ya sea porque hubo uno o varios ganadores o porque todos lo jugadores decidieron abandonarla.

Cuando se construye el lobby se inicializa también una instancia de Configuration, la cual fue mencionada previamente. De ésta manera se crea una única vez esta estructura y todas las partidas obtendrán un puntero a la misma teniendo siempre la misma configuración.

Player

Esta clase representa, como su nombre lo indica, a cada jugador que se conecta al lobby. Cada uno tendrá como atributo un Protocol (se explicará en la sección common) el cual permitirá el envío y recepción de datos.

Al construir un player se inicia un nuevo thread receptor de datos por el cual se recibirán los DTOs y se los encolará en una SafeQueue (propia del GameThread). Sin embargo, en primer lugar se realiza el *handshake* entre el cliente y el servidor, para que el jugador pueda definir cómo desea jugar. Se explicará en profundidad el Handshake en la sección common, pero el cliente recibirá del mismo la cola del GameThread antes mencionada para que pueda comenzar la recepción de eventos por parte del cliente.

GameThread

Como ya se dijo GameThread representa a cada partida que se cree dentro del servidor, teniendo un id como identificador de la misma. Cada game thread tendrá un listado de jugadores que lo conforman. Éste listado puede tanto incrementar como decrementar previo a la partida, ante jugadores que entran y salen en la espera de que el owner le dé inicio a la misma. Una vez que comienza la partida ya queda establecido el número de jugadores y ahora tan solo podrá verse reducido su tamaño.

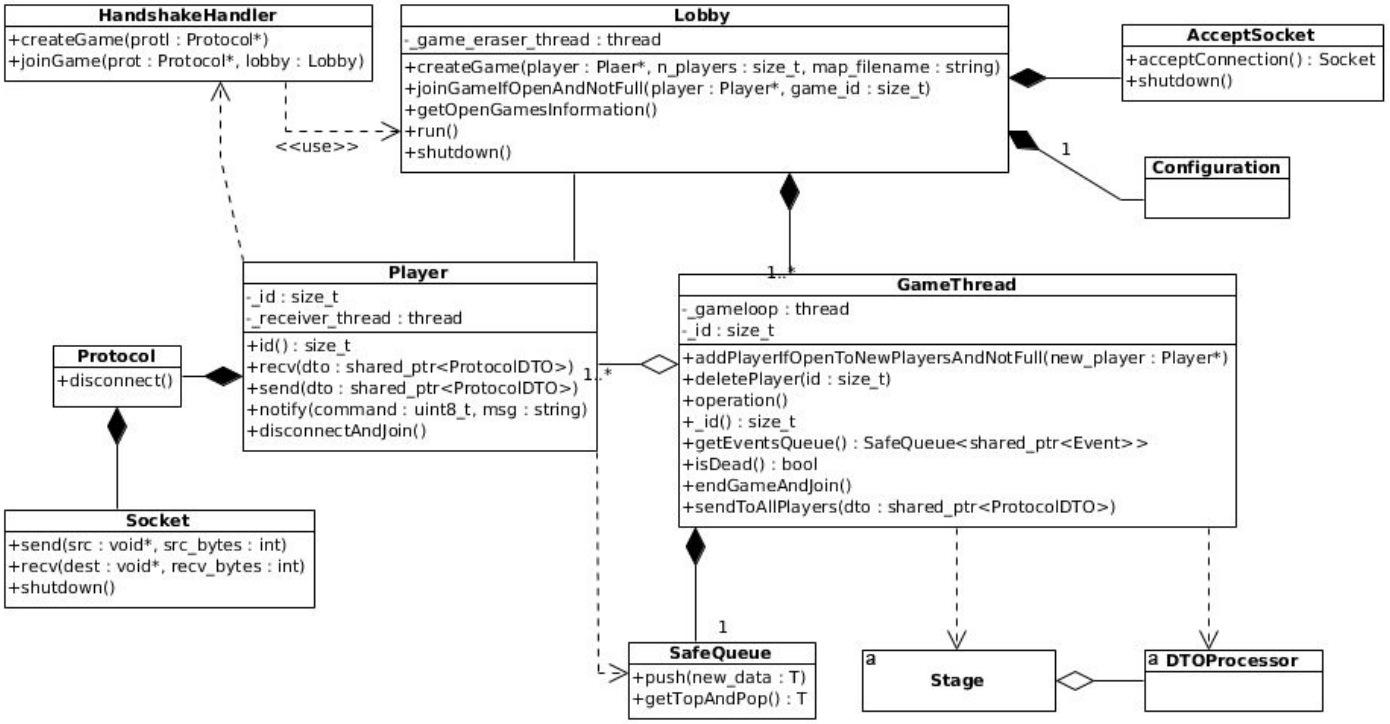
Cada GameThread tendrá un SafeQueue donde cada jugador encolará los distintos DTOs que reciba a través de la conexión y éste irá desencolando para procesarlo y delegar en otras clases para ejecutar la acción que corresponda en el modelo.

Al construir un game thread se inicia un nuevo thread donde se ejecutará el *gameloop*. Al iniciar el gameloop se crea el Stage en base al mapa seleccionado por el owner de la partida y luego comienza la siguiente serie de etapas:

1. Loop esperando a que owner de inicio a la partida. Dentro de éste loop se podría encontrar la situación que tanto el owner como otros jugadores abandonen la partida, cambios de los cuales se notificará a todos. En caso de que el owner se quien abandona la partida el siguiente jugador agregado será el nuevo owner. Para evitar estar constantemente desencolando DTOs esperando por el que de inicio se utiliza un sleep for que frena la ejecución del thread una cantidad dada de milisegundos.
2. Envío la configuración inicial del mapa;
3. Envío el id que corresponde a cada cliente;
4. Notifico que comienza la partida;
5. Inicia gameloop donde en primer lugar se desencolan los eventos registrados por cada Player, se aplican dichos eventos y se produce el step. Luego se envían los

datos de las entidades actualizadas y eliminadas durante el step. Por último se realiza un sleep del thread en base al tiempo transcurrido en éste ciclo en comparación al time_step definido por los FPS (frames por segundo);

Una vez que se tiene uno o varios ganadores o todos los jugadores abandonaron la partida el game thread se setea como finalizado, permitiendo que lobby lo cierre correctamente, libere sus recursos y lo elimine de su vector.



Common

Dentro de esta sección se encuentran todas aquellas clases que tienen vínculo con ambas partes de la conexión y dependen de ellas.

AcceptSocket

Socket aceptador para el servidor

Socket

Socket TCP general para ambos lados de la conexión.

Protocol

Esta clase encapsula el uso de sockets trabajando ya en base al protocolo que se estableció para el desarrollo del trabajo y será de la cual se cree una instancia para enviar y recibir datos, tanto en el cliente como en el servidor.

En la recepción de DTOs se encargará de generar el vector conteniendo los datos que se reciban a través del socket, para poder ser procesado por el ProtocolTranslator.

Thread

Encapsula el uso de threads.

SafeQueue

Cola protegida no bloqueante diseñada con templates, específicamente para almacenar punteros dentro de ella.

Protected BlockingQueue

Al igual que en SafeQueue, pero en éste caso se trata también de una cola bloqueante por medio de condition variables.

HandshakeHandler

El handshake handler fue diseñado para encapsular la conexión previa al inicio del juego entre servidor y cliente, para que el nuevo jugador pueda seleccionar qué experiencia realizar en su partida. A diferencia del protocolo que se explicará a continuación dentro del handshake no se trabaja con DTOs sino que envían strings y números en distintos formatos en un orden preestablecido, codeado totalmente dentro de ésta clase.

Del lado del servidor es Player quién utiliza ésta clase y cuando finalice su ejecución se le retornará un valor correspondiente para unirse o crear una partida y poder así obtener la cola con la cual trabajar a lo largo del juego.

ProtocolDTO

De ésta clase heredarán todos los posibles DTOs que existan en la partida. El único método que deberá implementar cada uno de ellos será aquel que entrege el ID de cada clase, para diferenciar uno de otro y aplicar las acciones o recibir los datos.

ProtocolTranslator

Ésta clase se encarga de convertir la información, tanto para ser enviada como ser procesada, luego de que se recibió los datos a través de Protocol.

En la recepción de datos recibirá un vector de enteros de dos bytes con signo, ya convertidos al endianess local. Se encargará de evaluar cada valor (según como se ha definido el protocolo) y generar cada DTO que corresponda, cada uno con una serie de argumentos particular.

Protocolo

A continuación se explicará como se definió el protocolo para enviar y recibir datos, ya se previó al juego o durante el mismo.

Handshake

El handshake consiste del envío y recepción tanto de strings como valores numéricos. El proceso general será siempre el mismo: el servidor enviará un mensaje con información al cliente y esperará una respuesta del mismo. Ésta respuesta podrá ser un byte sin signo o dos bytes sin signo, en base a si el jugador haya decidido crear una partida o unirse a una existente. En caso de enviar un string se enviará primero la longitud del mismo en un entero de 4 bytes sin signo y todos los valores numéricos se enviarán en formato big endian.

Cada vez que el cliente envía un valor representando una decisión el servidor verificará que éste valor sea adecuado y de lo contrario enviará una notificación de error y requerirá que el cliente reingrese dicho valor, hasta que el mismo sea o se encuentre en el rango esperado.

Una vez que concluyó la toma de decisiones el cliente continuará leyendo strings y valores numéricos, mientras que el servidor comenzará a esperar DTOs, ya sean para empezar la partida o retirarse de la misma.

DTO

Una vez que se finaliza el handshake y durante el gameloop se enviarán y recibirán DTOs creados por los alumnos. Se decidió enviar todos los datos como enteros de dos bytes con signo, considerando el rango de valores que se podría manejar en la posiciones y que las mismas podrían ser negativas inclusive.

Los mismos respetarán siempre la misma secuencia:

1. Enviar el ID del DTO ;
2. Enviar la cantidad de argumentos que posee el DTO;
3. Enviar todos los argumentos del DTO, definidos éstos para cada DTO particular.

En *protocol_macros.h* se definen en macros todos los valores de ID, cantidad de argumentos y posición de los argumentos para cada DTO, explicando a continuación cuales y como serán a continuación (no se incluye el valor del ID ni la cantidad de argumentos).

Envío de Posiciones

Para el envío de las posiciones se decidió enviar la posición de la esquina inferior izquierda, viendo cada cuerpo como una caja completa, incluso en el caso de bloques diagonales. Aquellos cuerpos que sean circulares se enviará la posición de su centro.

Cliente a Servidor

- PROTOCOL_MOVE_LEFT
- PROTOCOL_MOVE_RIGHT
- PROTOCOL_JUMP
- PROTOCOL_STOP
- PROTOCOL_SHOOT_PORTAL: { color, x, y}
- PROTOCOL_SHOOT_PIN_TOOL: { x, y}
- PROTOCOL_LIFT_ROCK: {id_rock}
- PROTOCOL_DROP_ROCK
- PROTOCOL_COMMIT_SUICIDE
- PROTOCOL_KILL_MISSING_CHELL
- PROTOCOL_RESET_PORTALS

Servidor a Cliente

- PROTOCOL_ROCK_BLOCK_DATA: {x, y, width, height}
- PROTOCOL_METAL_BLOCK_DATA: {x, y, width, height}
- PROTOCOL_METAL_DIAGONAL_BLOCK_DATA: {x, y, side_length, orientation}
- PROTOCOL_ENERGY_TRANSMITTER_DATA: {id, x, y, side_length, direction}
- PROTOCOL_ENERGY_RECEIVER_DATA: {id, x, y, side_length}
- PROTOCOL_ACID_DATA: {x, y, width, height}
- PROTOCOL_BUTTON_DATA: {id, x, y, width, height}
- PROTOCOL_GATE_DATA: {id, x, y, width, height}
- PROTOCOL_ENERGY_BARRIER_DATA: {N, x, y, width, height}
    - Width y height son suficientes para dibujar barrera vertical/horizontal
- PROTOCOL_PLAYER_CHELL_ID: {id}
    - Protocolo para informar que id le corresponde a cada jugador.
- PROTOCOL_CAKE_DATA: {x, y, side_length}

Hasta este punto eran datos que solo se enviaban a la hora de creación del mapa, no en el transcurso del juego, los que continúan podrán ser enviados constantemente.

- PROTOCOL_ROCK_DATA: {id, x, y, side_length, delete_state}
- PROTOCOL_ENERGY_BALL_DATA: {id, x, y, radius, delete_state}
    - En este caso x e y son del centro, radius determina el tamaño;
- PROTOCOL_PORTAL_DATA: {id, x, y, width, height, tilt, color, delete_state}
- PROTOCOL_PIN_TOOL_DATA: {id, x, y, width, height, delete_state}

- PROTOCOL_CHELL_DATA: {id, x, y, direction, tilted, moving, jumping, shooting, delete_state}
    - Direction solo será EAST o WEST
- PROTOCOL_BUTTON_CHANGE_STATE: {id, button_state}
- PROTOCOL_ENERGY_TRANSMITTER_ACTIVATE: {id}
- PROTOCOL_ENERGY_RECEIVER_ACTIVATE: {id}
- PROTOCOL_GATE_CHANGE_STATE: {id, gate_state}

Opciones para los argumentos

- *COLOR:*
    1. BLUE_PORTAL: 0
    2. ORANGE_PORTAL: 1
- *ORIENTATION:*
    1. NORTH_EAST: 0
    2. NORTH_WEST: 1
    3. SOUTH_EAST: 2
    4. SOUTH_WEST: 3
- *DIRECTION:*
    1. NORTH: 4
    2. SOUTH: 5
    3. EAST: 6
    4. WEST: 7
- *DELETE_STATE:* indica si el cuerpo debe ser eliminado
    1. DONT_DELETE: 0
    2. DELETE: 1
- *BUTTON_STATE*:
    1. NOT_PRESSED: 0
    2. PRESSED: 1
- *GATE_STATE:*
    1. CLOSED: 0
    2. OPEN: 1
- *MOVING:* indica si chell está en movimiento
    1. NOT_MOVING: 0
    2. MOVING: 1
- *JUMPING:* indica si chell debe saltar
    1. NOT_JUMPING: 0
    2. JUMPING: 1
- *TILTED:* indica si chell está sobre superficie diagonal
    1. NOT_TILTED: 8
    2. NORTH_EAST: 0
    3. NORTH_WEST: 1
- *SHOOTING:* indica si chell realizó un disparo
    1. NOT_SHOOTING: 0
    2. SHOOTING: 1

Conexión - Estado Partida

- PROTOCOL_QUIT
- PROTOCOL_BEGIN
- PROTOCOL_WINNER

Cliente

La sección del cliente tendrá como responsabilidad, la de proveer una interfaz gráfica con el cual el usuario podrá interactuar con el servidor.
Como se mencionó anteriormente para realizar el cliente se utilizó dos librerías diferentes: SDL y QT, se utiliza QT para poder establecer una conexión con el servidor a través de una interfaz gráfica y se utiliza SDL para la interacción del usuario con el videojuego.

Pantalla de Login

La idea principal era de proveer al usuario una interfaz gráfica amistosa para conectarse al servidor, anteriormente a esto se tenía un HandShaker mediante consola, la idea es mediante QT es simular al usuario como es que interactuara con la consola.
Lo que se hace primero es verificar que los address provistos son correctos, en caso de ser incorrecto mostrar un error.
Para el caso de mostrar los diferentes escenarios, y los diferentes matches se deben parsear los datos y ir añadiendo a una lista.
Y para cuando se llega al caso de esperar a que el owner inicialice la partida, se lanza un Thread (GUIReceiver), que está siempre esperando a que le llegue un mensaje, lo interpreta y le avisa a la GUI (a través de slots & signals) si debe mostrar un mensaje, o mostrar el botón para empezar la partida.

Factories:

Dado que muchos de las texturas, y sonidos que utilizaremos en la vista se repiten, y para bajar el consumo de memoria lo mejor posible, la creacion y destruccion de los mismos se encuentran aquí.

SoundFactory

Clase encargada de cargar los sonidos del videojuego guardados en el disco.
En caso de que hubo un error al cargar un archivo, se lanza una excepción SDLException.

TextureFactory

Clase encargada de cargar las texturas del videojuego.
En caso de que hubo un error al cargar un archivo, se lanza una excepción SDLException.

SDL Runner

Esta clase hereda de thread, tiene una cola no bloqueante, que desencola y interpreta mediante el id del protocolo de que protocolo se trata, y en base a eso actualizar a informar al WorldView sobre los cambios que debe realizar.
Sumado a que también lanza los sonidos de la aplicación. Contiene un SDL_Delay para no tener que consumir todo el tiempo los recursos del sistema.
Cada cierta frecuencia vuelve a renderizar, incluso si hay elementos en la cola no bloqueante, así poder tener un sistema con frames constantes, sin depender de la cantidad de eventos que lleguen.
Al momento inicializar la clase recibe todos objetos del mapa hasta que recibe un PROTOCOL_BEGIN, a medida que recibe los objetos del mapa los va añadiendo a un WorldView.
Una vez salida de esta fase empieza la parte de renderizado constante, y recibe solo estados nuevos, a medida que llega los interpreta y llama al método de WorldView correspondiente, en caso de que sea por ejemplo: abrir una puerta, o apretar un botón llama a MusicPlayer para reproducir un sonido,

GameController

Encola los DTOs a enviar a una cola bloqueante que luego la clase ClientSender enviará al servidor utilizando el protocolo.
Tiene todos los controles posible que el jugador podrá utilizar.
Debido a que está siempre procesando se le puso un SDL_Delay para no tener que consumir todo el tiempo recursos, aunque este podría no ser necesario dado que tiene un SDL_PollEvent que es bloqueante. Los DTOs los receive a través de un ClientReceiver.
La idea era intentar imitar un patrón MVC.

ClientReceiver

No es necesario poner un sleep aqui, dado que el protocolo tiene una función recvt, que es bloqueante. Dado de que esta clase siempre esta en ejecución, se hereda de Thread

ClientSender

Como se dijo previamente, esta clase desencola y envía los DTOs al servidor cada vez que le llega un elemento a la cola bloqueante. Dado de que esta clase siempre esta en ejecución, se hereda de Thread

Elementos de ventana

Sprite

Esta clase lo que hace es guardar un puntero a una textura, y permite “cortar” o tomar parte de esa textura. Tiene un método draw que recibe a una camara, y este llama al metodo draw de la camara.

AnimatedSprite

Hereda de Sprite, y el objetivo era proveer la suficiente flexibilidad para poder manejar todos los sprites de la aplicación. Se define el width, height, x inicial, y inicial, cantidad de imágenes por fila (Máxima), la cantidad de diferentes sprites a renderizar, un offset en X y offset en Y. Y ademas cuanto tiempo tiene que estar mostrando el mismo sprite.
Posee tres diferentes estados posibles:
onRepeat: una vez llegado al final del sprite se vuelve al principio
oneTime: Una vez llegado al final del sprite no se vuelve a repetir.
revert: Vuelve a pasar por los Sprite ya mostrados.

Camera

Determina si un objeto está en la vista del usuario, en caso de estarlo lo muestra.
Conoce cuánto vale un pixel a metros, y realiza la conversión.
Puede obtener el X,Y en metros para poder enviarlo al servidor.
Dibuja lo que el usuario verá en la ventana.

Window

Es un wrapper al objeto SDL_Window de SDL, representa la ventana del usuario

Renderer

Wrapper al objeto SDL_Renderer, tiene a la Window, y se encarga de copiar una textura a la pantalla del usuario

Position

Es utilizada para saber la posición de la chell.

View

Esta clase contiene toda la información de donde está posicionada en el mundo,
Y provee una interfaz para que las diferentes clases puedan dibujar.
Hay diferentes views:

- AcidView
- BackgroundView
- BallView
- BlockMetalView
- BlockRockView
- ButtonView
- CakeView
- ChellView
- DiagonalBlockMetalView
- EnergyBallView
- EnergyBarrierView
- EnergyReceiverView
- EnergyTransmitterView
- GateView
- PinToolView
- PortalBlueView
- PortalOrangeView
- RockView

A continuación se explicarán con mayor detalle las más “importantes”. Las demás view son una combinación de todas estas.

ChellView

Dado que la textura de chell era muy grande, se decidió cortarla en animaciones pequeñas. Esta clase hereda de View (valga la redundancia), recibe una **id** que la identificará durante todo el transcurso del juego, y las diferentes **texturas** que utiliza:

- dyingTexture
- firingTexture
- flyingTexture
- restingTexture
- runningTexture
- winningTexture

Posee las siguientes Animaciones:

- runningRightChell. Con el estado onRepeat
- standStillChell. Con el estado onRepeat
- firingChell: Con el estado oneTime.
- flyingChell Con el estado onRepeat
- dyingChell: con el estado oneTime
- winningChell: onRepeat

Contiene una lista de estados posibles (**ChellState**): standing, runningLeft, runningRight, firing, flying, dying, winning.
Y además permite saber si la chell está inclinada (**ChellIsTilted**): TILTEDLEFT, TILTEDRIGHT, NO.

Al crearse se pone por default el state: standing y tiltedState como NO.
Además se reajusta el tiempo por default que tienen las AnimationView que muestran cada frame.
Para dibujar pregunta que estado es, y en base a esto, llama al método draw correspondiente, de la animación correspondiente.
El set estados, se decidió preguntar primero si chell estaba disparando dado que se decidió a darle prioridad a esa animación sobre todas las demás.

GateView

Recibe una id, y una textura, que luego es recortada
Tiene un Sprite (**closedGate**) y un AnimatedSprite(**gate**) y contiene tres estados posibles:

- Open: Cuando esta en este estado se muestra el AnimatedSprite gate.
- closed: Cuando esta en este estado se muestra el el Sprite closedGate.
- isClosingBack: Y si esta en este estado, se muestra el AnimatedSprite gate, con la consideración de que una vez que el AnimatedSprite vuelva al principio se “resetee” (Vuelva a al estado oneTime)

Cuando llega un mensaje de que la puerta se tiene que cerrar, se cambia el estado del AnimatedSprite gate a revert. Y cuando se quiere abrir se setea el estado de GateState a Open

ButtonView

Recibe una id, y contiene dos Sprites, y dos estados posibles que identifican al boton apretando y no.
De acuerdo a que estado esta, se muestra uno u otro.

DiagonalBlockMetalView

Contiene un único sprite que se irá rotando en el método draw dependiendo de qué orientación llego a la clase.

EnergyTransmitterView

Recibe un id, y contiene un Sprite (notTransmiting) y un AnimatedSprite(transmiteAnimation), con un estado oneTime.
Inicialmente empieza con el estado notTransmiting.
Al tener que poder rotar este objeto en cuatro direcciones diferentes (Norte, sur, este, oeste) dependiendo qué orientación es, se rota los sprites.
Una vez finalizada la animación de transmitir se resetea y se vuelve al “notTransmiting”

RockView

El caso de las rocas es muy particular, se puede elegir cualquiera de 3 rocas. Para realizar esto se utilizo la funcion rand() provista por c++, con eso determinó el tipo.

BackgroundView

Además de las rocas otro muy particular es el de background, siempre tiene que estar en la cámara y siempre ser el mismo, por lo que se hizo fue directamente acceder al renderer, y copiar la textura que ocupe todo el espacio disponible.

WorldView

Esta clase conoce todos los Views que hay en el mapa, para dibujar, lo que hace es preguntarle a la cámara si ese objeto está en la vista, en caso de estarlo le pasa la cámara, y llama al método draw. Como última instancia el que realiza el draw es la cámara.
Dado que para generar la ilusión de una imagen sobre otra, se tuvo que determinar un orden.
Primero se dibuja el background, luego los bloques sin id particular, rocas, compuertas, bolas de energia, transmisores de energía, receptores de energia, botones, portales, pintool, cake, y por ultimo a chell.
En caso de estar chell muerta(La animación de dead terminó) se la saca del WorldView.
Contiene una serie de mensajes que sirve para interactuar con cualquier objeto del mapa, ejemplo:

openGate(id); -> abre la compuerta con esta id
closeGate(id); -> cierra la compuerta con esa id

FakeServer

Dado que tanto el cliente y el server estuvieron mucho tiempo desconectados, se decidio crear esta clase que lo unico que hace es agarrar lo que envian las teclas en la Blocking queue, y encolarlo a la SafeQueue que tiene el SDL_Runner, con valores muy parecidos a los que podria enviar el cliente.

Diagramas de clase de la GUI

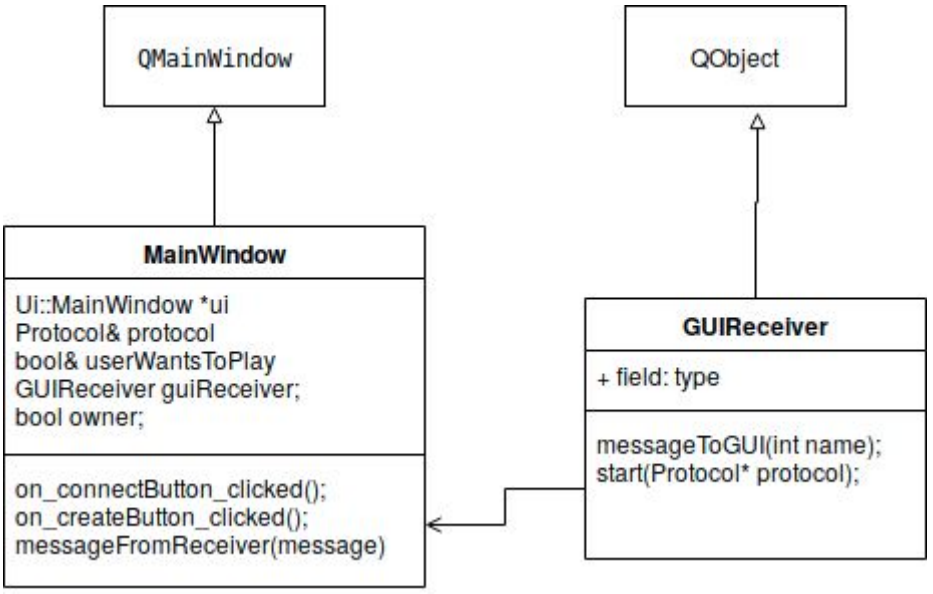
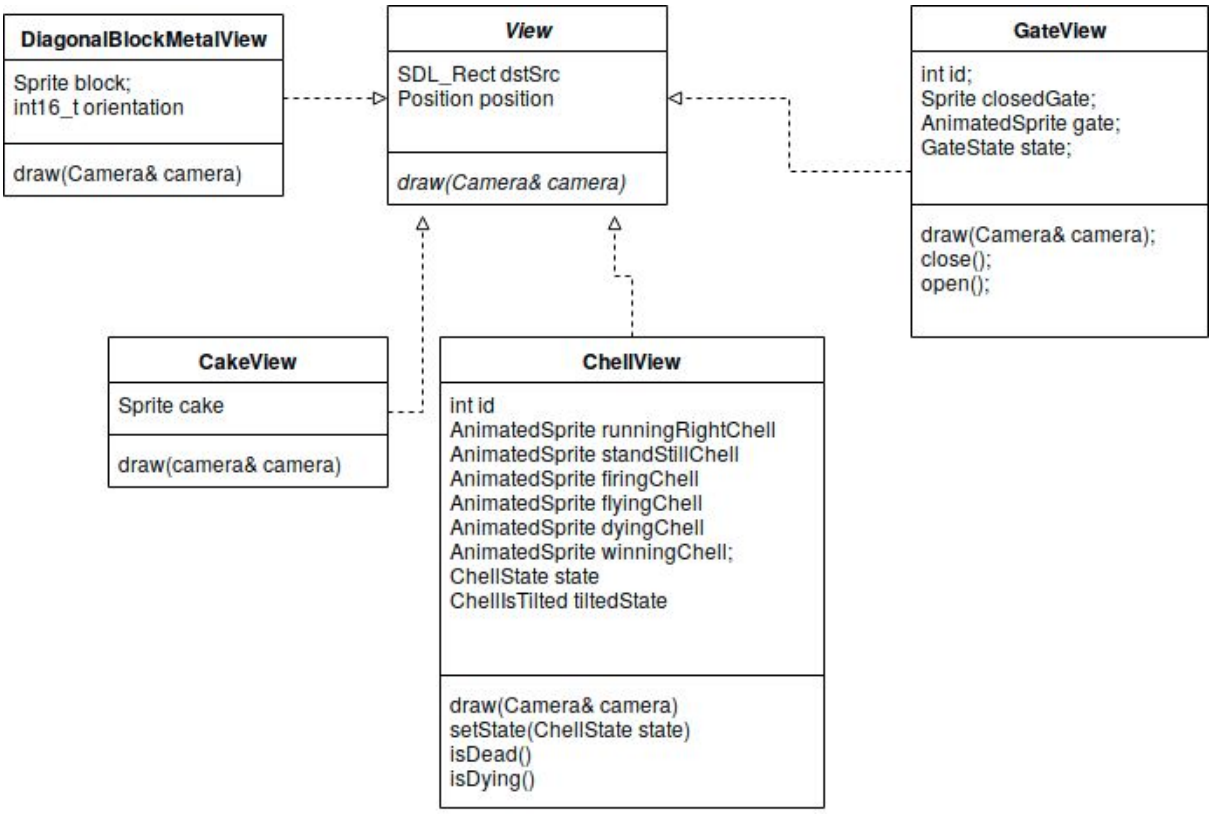


Diagrama de clase de las View



Dado la gran cantidad de views, se decidió solo mostrar un objeto de cada tipo, para tener una vista general de como estan compuestas las view.

Diagrama de Secuencia de cliente conexion con el server

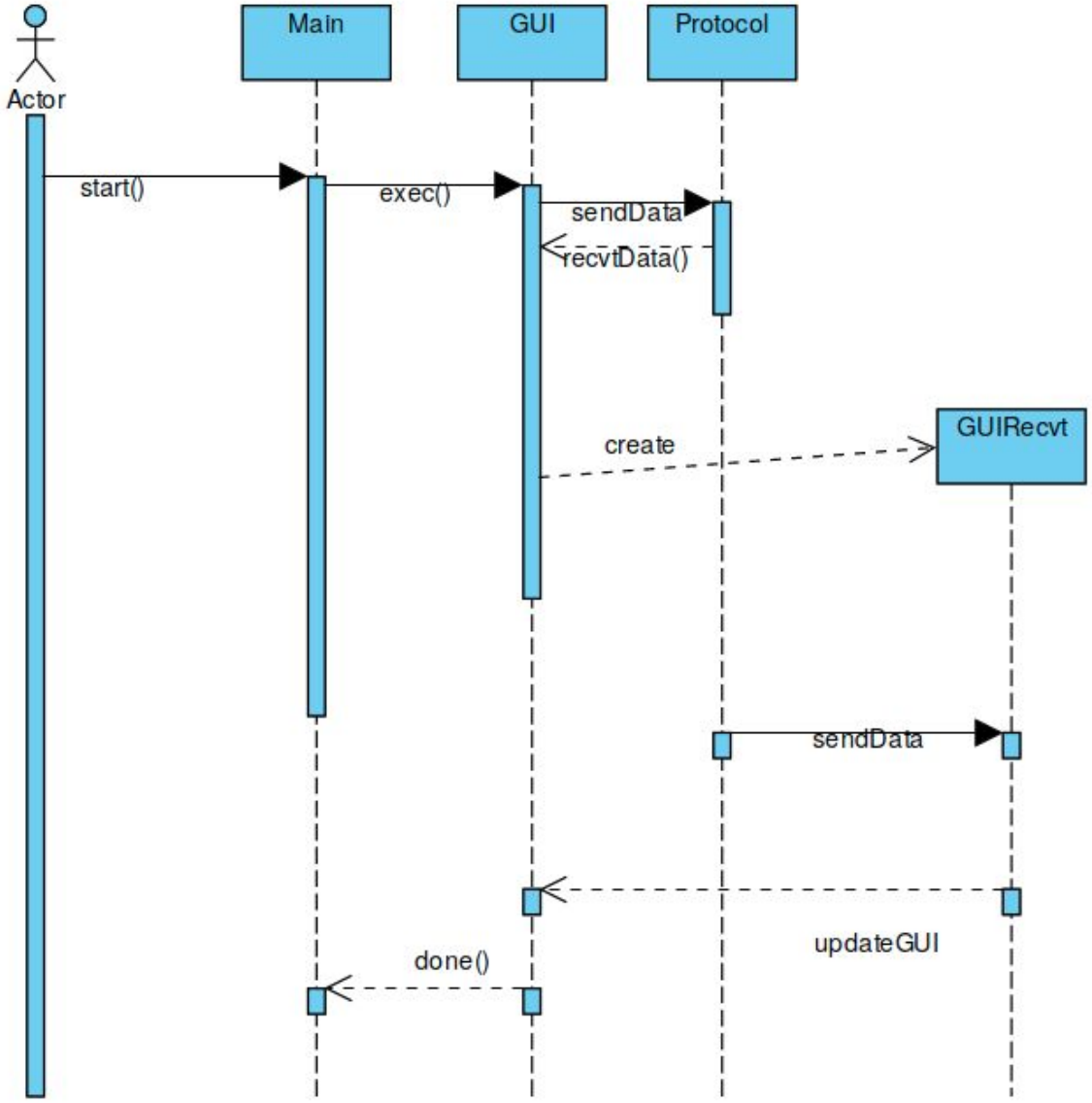

Actor
Main
GUI
Protocol
start()
exec()
sendData
recvtData()
create
GUIRecvt
sendData
updateGUI
done()

Diagrama de secuencias de envio de teclas

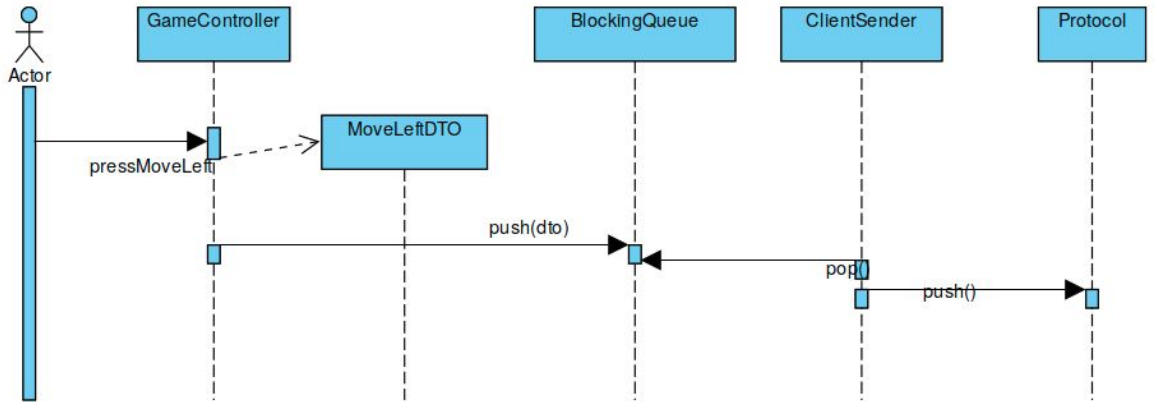
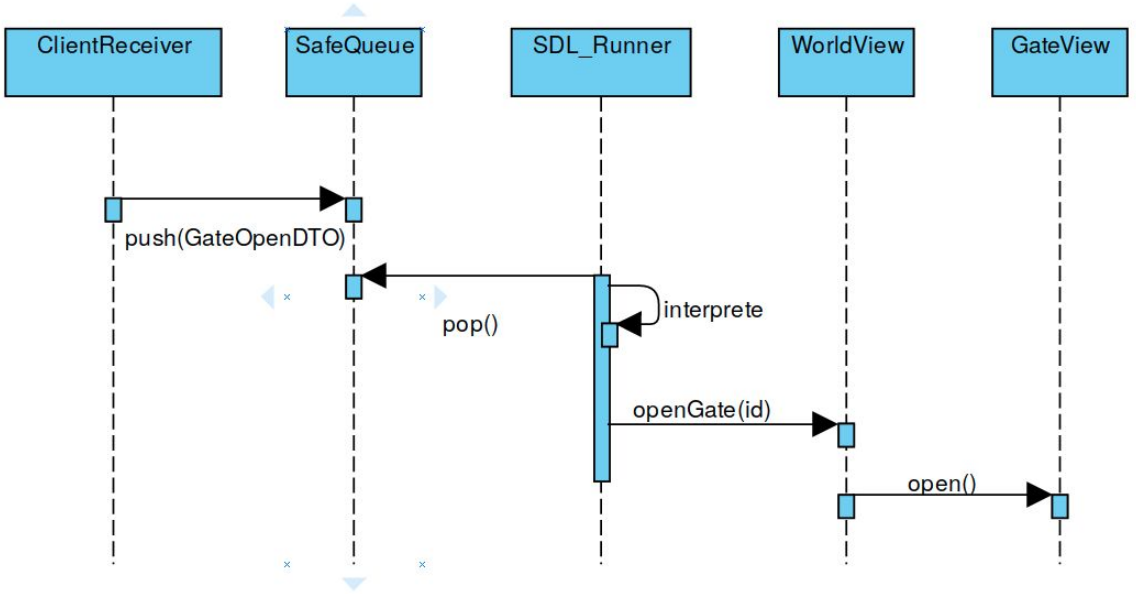
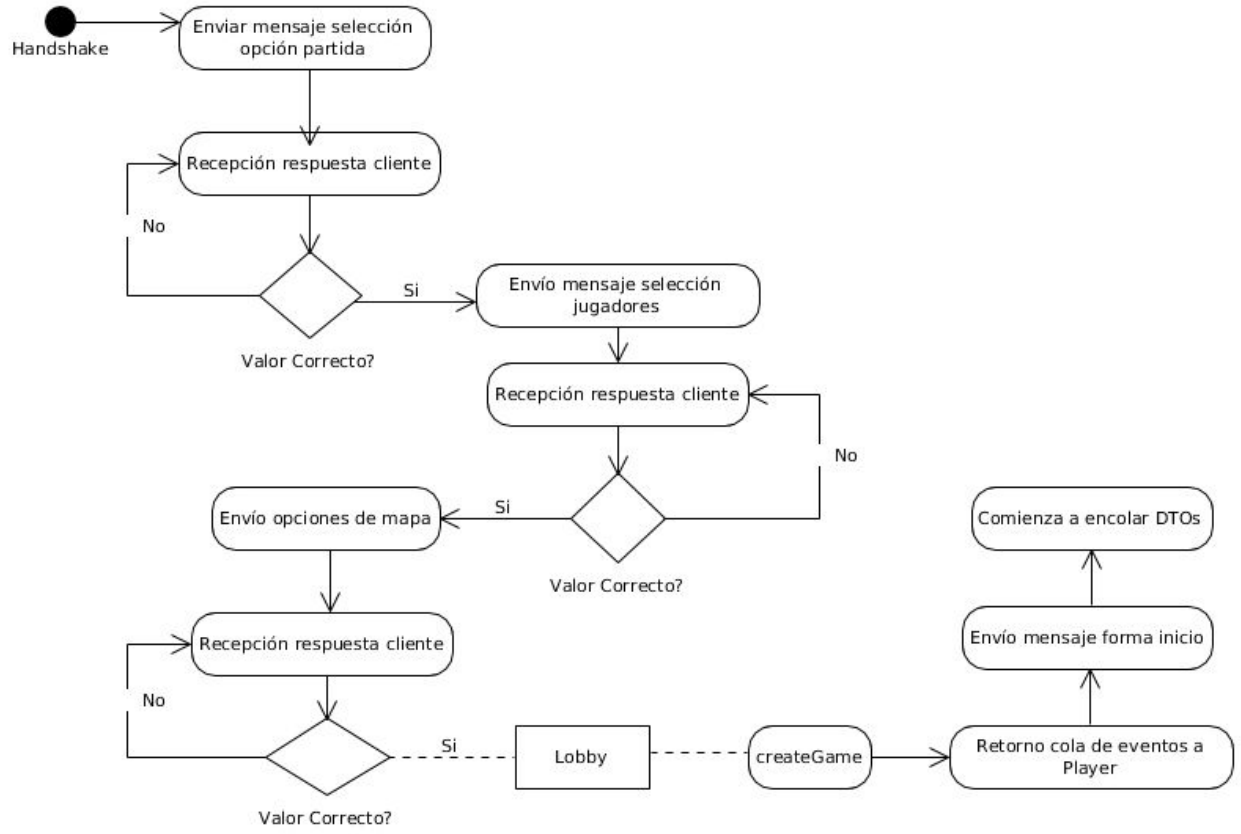


Diagrama de secuencia de abrir gate:

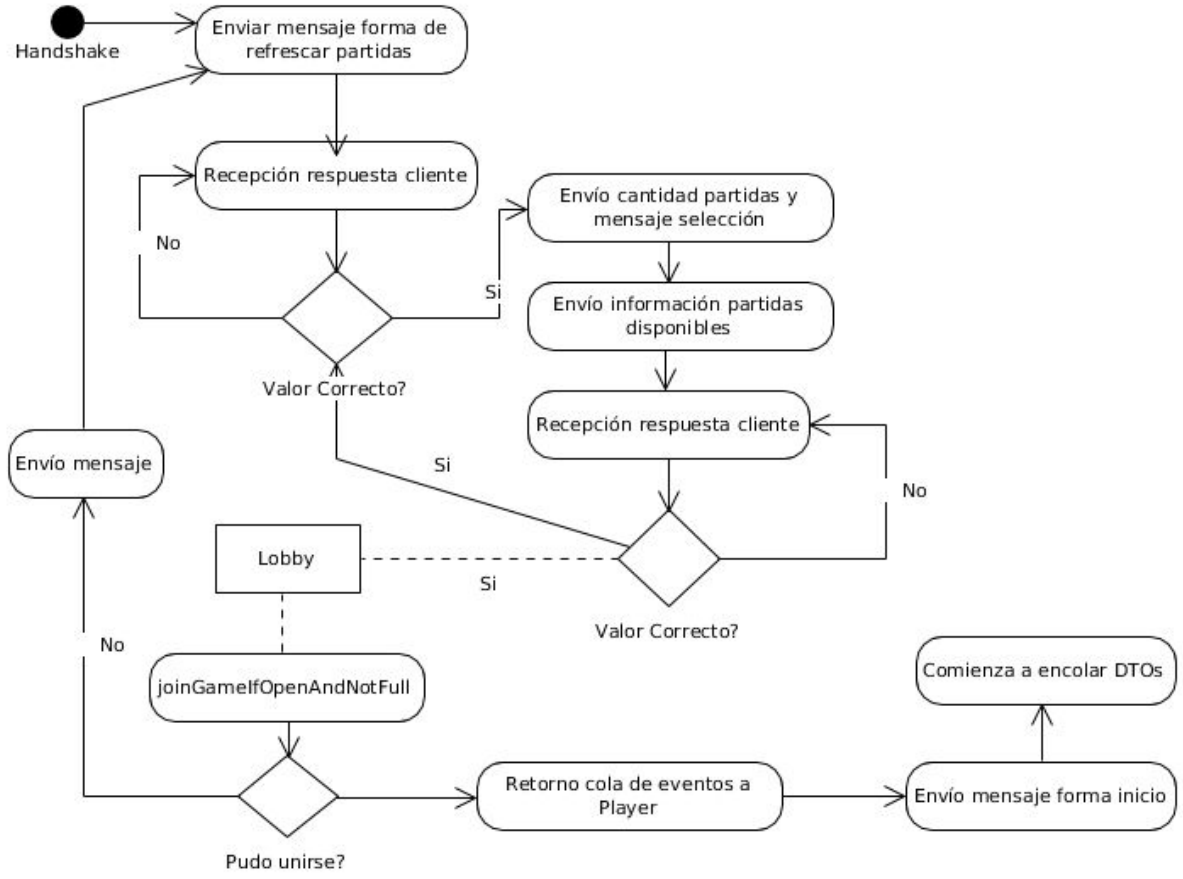


Diagramas Secuencia y Flujo Relevantes

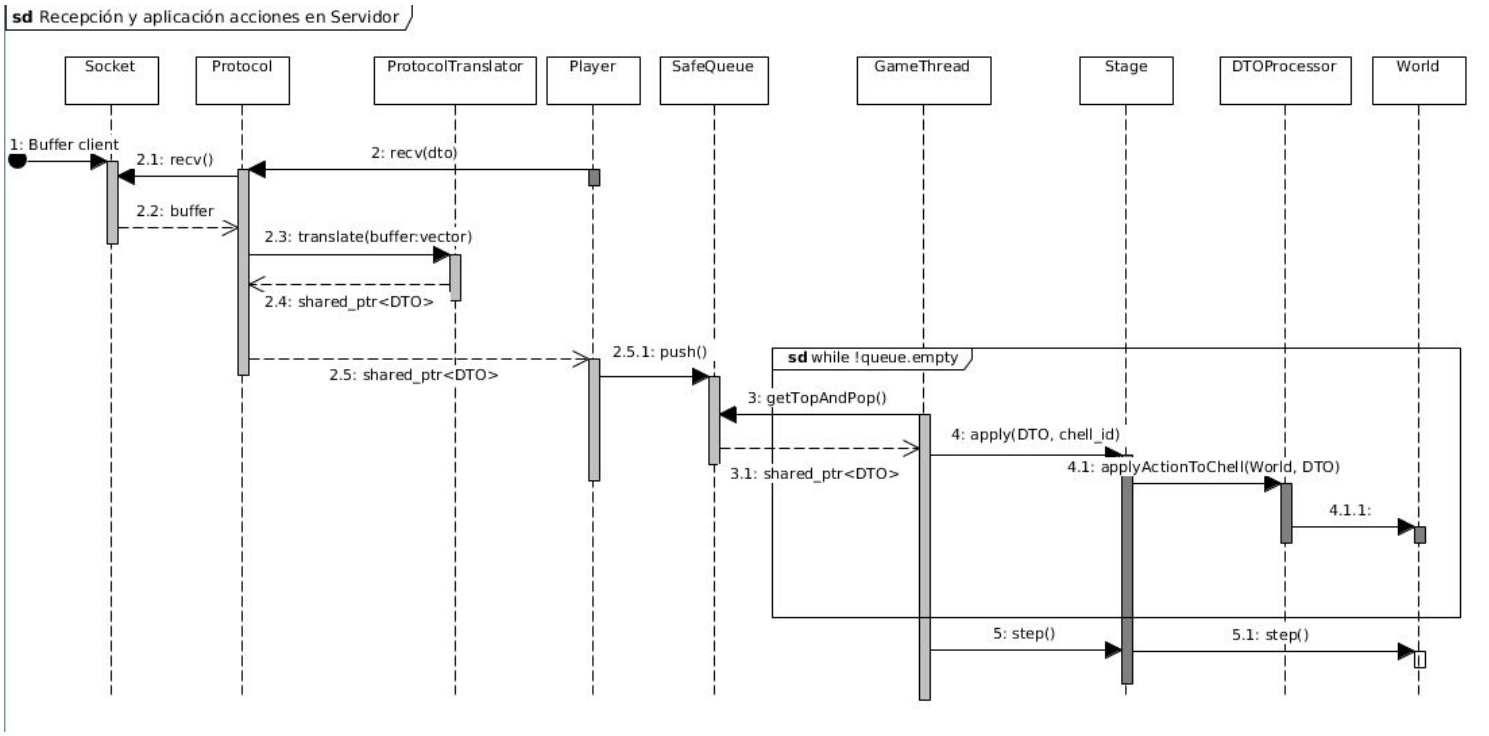
Handshake crear partida (visto desde el Servidor)



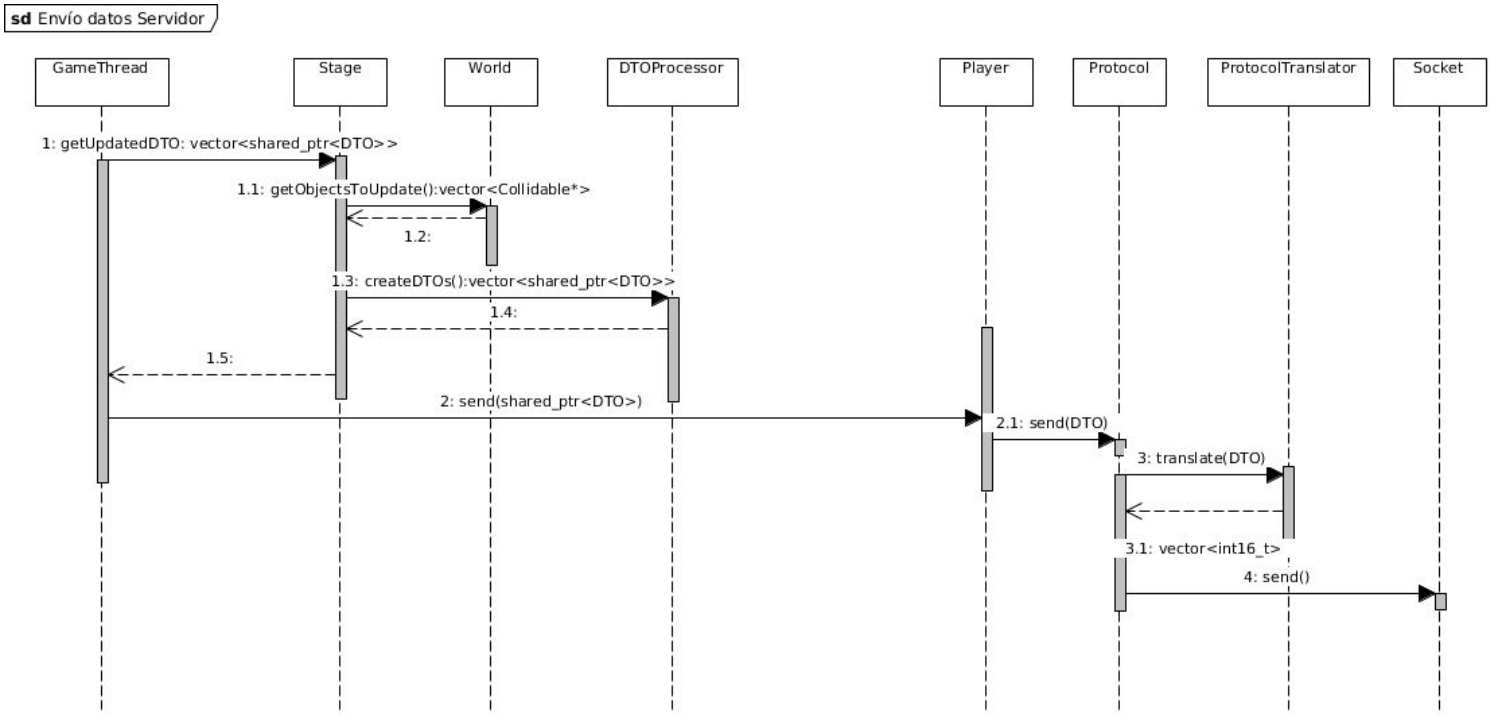
Handshake unir a partida (visto desde el Servidor)



Recepción datos desde el servidor



Envío datos desde servidor



En este diagrama se mostró el caso del vector de elementos a actualizar pero para aquellos elementos que deban ser eliminados el proceso a seguir será el mismo. También es importante aclarar que luego del envío de datos se produce un sleep en el hilo del gameloop acorde al tiempo que transcurrió y se vuelve al loop mostrado en la secuencia de recepción de datos, repitiendo esto hasta que finalice una partida, por cualquiera de las razones mencionadas en el desarrollo de éste informe y demás documentos del trabajo.

Programas intermedios y de Prueba

Para testear el modelo se llevaron a cabo pruebas de unit testing mediante CppUnit. Ya que la conexión al cliente se logró en un punto avanzado el trabajo fue de vital importancia desarrollar y mantener dichas pruebas para asegurar y facilitar el funcionamiento entre ambas partes, dando por hecho que las distintas partes del modelo trabajaban correctamente, haciendo más sencilla la tarea de debugging y encontrar posibles problemas dentro del código.

Tanto en el cliente como en el servidor a medida que se fue avanzando en el trabajo se decidieron crear tanto un cliente como servidor falso, De esta manera se pudo probar incrementalmente el funcionamiento de ambas partes, previo a realizar la conexión final. Este paso permitió la resolución de problemas que habiendo intentado directamente conectar al otro extreme hubiese sido mucho más complejo, dado que no se sabría, o sería bastante más complejo de detectar, cuál de las partes podría ser fuente bug en cuestión.