FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Wolfenstein 3D
Manual Técnico

[75.42] Taller de programación
Segundo cuatrimestre de 2020
Repositorio en Github
Grupo 8

BERTOLOTTO, Francisco	fbertolotto@fi.uba.ar	102671
LÓPEZ NÚÑEZ, Agustín	alopezn@fi.uba.ar	101826
SANTONI, Mauro	msantoni@fi.uba.ar	102654
FERNÁNDEZ, Andrés	andyfer@fi.uba.ar	102220

Índice

1. Requerimientos de software

1.1. Bibliotecas utilizadas

Importante: notar que las dependencias necesarias se instalan automáticamente si el servidor es instalado mediante el paquete *.deb*.

1.1.1. Bibliotecas básicas

Para compilar todos los módulos, es necesario instalar GCC, Make (ambas incluidas en el paquete build-essential) y CMake. Cada una de estas puede instalarse mediante:

```
$   sudo apt-get install build-essential
$   sudo apt-get install cmake
```

1.1.2. SDL2 y sus derivados de imágen, sonido y texto

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

1.1.3. Bot con IA - Lua

```
$   sudo apt-get install lua5.3 liblua5.3-dev
```

1.1.4. YAML

Adjuntado dentro del repositorio de manera que permita su fácil ejecución y evite la necesidad de instalarla ya que debe ser instalada de manera muy específica generando problemas innecesarios.

1.1.5. Editor y menú del Cliente - QT5

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

2. Compilación y ejecución

Todos los módulos poseen dos formas de instalación, manual o por paquete *.deb*. Esto permite versatilidad a la hora de querer una instalación más simple o más fácil de modificar. A continuación se demarcan las ventajas y desventajas de cada forma de instalación.

Tanto los mapas como el archivo de configuración son tomados por cada módulo en el momento de su compilación, de manera que para cada tipo de instalación suceden cosas distintas. En los anexos de instalación se especificará cuáles son los pasos a seguir para instalar los módulos.

- **Instalación manual:** si se desea modificar los mapas o la configuración se debe editar en el caso de los mapas los de la carpeta `maps` en la raíz del repositorio y en el caso del archivo de configuraciones se debe editar el archivo `config.yaml` dentro de la carpeta `config` en la raíz del repositorio. Luego, al ejecutar nuevamente el comando `make run` se volverá a compilar el módulo respectivo contemplando el cambio en mapa o configuración. De esta manera permite recompilar contemplando nuevos cambios sin ninguna acción extra. Es importante

notar también que estos cambios requieren recompilación tanto del servidor como del cliente ya que ambos toman los mapas y la configuración en su ejecución.

- **Instalación mediante paquete deb:** en este caso la recompilación no sucederá ya que el código está instalado en la máquina. Como los mapas y la configuración se cargan sólamente al ejecutar el juego, para cambiar alguna configuración se debe hacerlo tanto al servidor como al cliente en sus respectivas carpetas de `maps` o `config` en `/usr/local/share/wolfenstein3d-<modulo>` siendo modulo `client` o `server`.

2.1. Server

Como fue descripto previamente, existen dos maneras de instalar el servidor, una es mediante el paquete *.deb* obtenido a través del repositorio o bien compilando el código fuente manualmente.

2.1.1. Instalación y ejecución mediante paquete deb

Para esto se debe descargar el paquete de instalación del siguiente link.
Una vez descargado los pasos a seguir son los siguientes:

- Ingresar a la carpeta donde se fue descargado el paquete.
- Ejecutar el siguiente comando:

```
$   sudo apt install ./wolfenstein3d-server_1.0.deb
```

Y esto concluye la instalación. Si todo salió bien, Wolfenstein3D-Server puede ejecutarse de la siguiente manera:

```
$   wolfenstein3d-server <puerto>
```

2.1.2. Instalación y ejecución mediante compilación manual

Se proveen *makefiles* oportunos para facilitar la instalación y posterior ejecución del juego.
Los pasos a seguir son:

- Clonar el repositorio mediante:

```
$   git clone https://github.com/mjsantoni/taller_wolfenstein3D.git
```

- Esto descargará el repositorio en la carpeta *taller_ wolfenstein3D*. Ingresar a la carpeta y luego a la carpeta del servidor.

```
$   cd taller_wolfenstein3D
$   cd server_src
```

- Una vez dentro ejecutar el makefile que permitirá compilar el programa para su posterior ejecución

```
$   make
```

- Para ejecutar el servidor se podrá hacer de dos maneras. La primera es:

```
$   make run <puerto>
```

- La segunda es ingresando a la carpeta *build* y corriendo el ejecutable manualmente:

```
$   cd build
$   ./wolfenstein3d-server <puerto>
```

Si se ejecuta de la primer forma, el programa se recompilará cada vez de ser necesario. Si se ejecuta de la segunda forma y se quiere volver a compilar se deberá ejecutar dentro de la carpeta del servidor:

```
$   make build
```

El makefile incluído en esta carpeta permite también eliminar los archivos necesarios para la ejecución y retornar la carpeta a su estado inicial mediante

```
$   make clean-all
```

2.2. Cliente

Como fue descripto previamente, existen dos maneras de instalar el servidor, una es mediante el paquete *.deb* obtenido a través del repositorio o bien compilando el código fuente manualmente.

2.2.1. Instalación y ejecución mediante paquete deb

Instalar el cliente mediante el paquete *.deb* es la manera más simple y rápida.
Para esto se debe descargar el paquete de instalación del siguiente link.
Una vez descargado los pasos a seguir son los siguientes:

- Ingresar a la carpeta donde se fue descargado el paquete.
- Ejecutar el siguiente comando:

```
$   sudo apt install ./wolfenstein3d-client_1.0.deb
```

Y esto concluye la instalación. Si todo salió bien, Wolfenstein3D-Client puede ejecutarse de la siguiente manera:

```
$   wolfenstein3d-client
```

2.2.2. Instalación y ejecución mediante compilación manual

Se proveen *makefiles* oportunos para facilitar la instalación y posterior ejecución del juego.
Los pasos a seguir son:

- Clonar el repositorio mediante:

```
$   git clone https://github.com/mjsantoni/taller_wolfenstein3D.git
```

- Esto descargará el repositorio en la carpeta *taller_ wolfenstein3D*. Ingresar a la carpeta y luego a la carpeta del cliente.

```
$   cd taller_wolfenstein3D
$   cd client_src
```

- Una vez dentro ejecutar el makefile que permitirá compilar el programa para su posterior ejecución

```
$   make
```

- Para ejecutar el cliente se podrá hacer de dos maneras. La primera es:

```
$   make run
```

- La segunda es ingresando a la carpeta *build* y corriendo el ejecutable manualmente:

```
$   cd build
$   ./wolfenstein3d-client
```

Si se ejecuta de la primer forma, el programa se recompilará cada vez de ser necesario. Si se ejecuta de la segunda forma y se quiere volver a compilar se deberá ejecutar dentro de la carpeta del cliente:

```
$   make build
```

El makefile incluído en esta carpeta permite también eliminar los archivos necesarios para la ejecución y retornar la carpeta a su estado inicial mediante

```
$   make clean-all
```

2.3. Editor

Como fue descripto previamente, existen dos maneras de instalar el servidor, una es mediante el paquete *.deb* obtenido a través del repositorio o bien compilando el código fuente manualmente.

2.3.1. Instalación y ejecución mediante paquete deb

Instalar el editor mediante el paquete *.deb* es la manera más simple y rápida.
Para esto se debe descargar el paquete de instalación del siguiente link.
Una vez descargado los pasos a seguir son los siguientes:

- Ingresar a la carpeta donde se fue descargado el paquete.

- Ejecutar el siguiente comando:

```
$   sudo apt install ./wolfenstein3d-editor_1.0.deb
```

Y esto concluye la instalación. Si todo salió bien, Wolfenstein3D-Editor puede ejecutarse de la siguiente manera:

```
$   wolfenstein3d-editor
```

2.3.2. Instalación y ejecución mediante compilación manual

Se proveen *makefiles* oportunos para facilitar la instalación y posterior ejecución del juego. Los pasos a seguir son:

- Clonar el repositorio mediante:

```
$   git clone https://github.com/mjsantoni/taller_wolfenstein3D.git
```

- Esto descargará el repositorio en la carpeta *taller_ wolfenstein3D*. Ingresar a la carpeta y luego a la carpeta del editor.

```
$   cd taller_wolfenstein3D
$   cd client_src
```

- Una vez dentro ejecutar el makefile que permitirá compilar el programa para su posterior ejecución

```
$   make
```

- Para ejecutar el editor se podrá hacer de dos maneras. La primera es:

```
$   make run
```

- La segunda es ingresando a la carpeta *build* y corriendo el ejecutable manualmente:

```
$   cd build
$   ./wolfenstein3d-editor
```

Si se ejecuta de la primer forma, el programa se recompilará cada vez de ser necesario. Si se ejecuta de la segunda forma y se quiere volver a compilar se deberá ejecutar dentro de la carpeta del editor:

```
$   make build
```

El makefile incluído en esta carpeta permite también eliminar los archivos necesarios para la ejecución y retornar la carpeta a su estado inicial mediante

```
$   make clean-all
```

2.4. Formato de los mapas YAML

Para el correcto funcionamiento de los mapas es necesario que sigan este lineamiento:

```yaml
dimensions:
  width: 25
  height: 25
scenarios:
  wood_wall: []
  rock_wall: [[0, 0]]
  stone_wall: [[4, 2], [4, 3]]
  blue_wall: []
  barrel: []
  locked_door: [[5, 20], [6, 6], [19, 10]]
  unlocked_door: []
  fake_wall: [[4, 1], [13, 10]]
  table: [[23, 17], [23, 23]]
items:
  machine_gun: [[1, 1], [11, 3], [13, 15], [16, 7], [21, 7]]
  rpg_gun: [[3, 21]]
  chain_gun: [[3, 9], [12, 19], [16, 17]]
  bullets: [[2, 5], [2, 14], [9, 16], [10, 10], [11, 22], [16, 4], [19, 21], [20, 3]]
  chest: []
  cross: [[1, 21]]
  crown: [[1, 20], [1, 22]]
  goblet: []
  food: []
  key: [[4, 12], [5, 5], [22, 23], [23, 1]]
  medkit: [[4, 23], [15, 14], [21, 23], [22, 17]]
  water_puddle: [[21, 17], [22, 16], [22, 18]]
players:
  0: [[1, 17]]
  1: [[8, 13]]
  2: [[14, 1]]
  3: [[16, 20]]
```

- **dimensions:** El tamaño del mapa.
- **scenarios:** Todos los obejtos bloqueantes del mapa.
- **items:** Items del mapa, armas, tesoros, paquetes de vida.
- **players:** Posiciones de los spawns de cada player.

3. Empaquetado de los módulos

Dentro de cada carpeta de cada módulo hay directorios llamados *package_ generation* dentro del cual se encuentra un script de Makefile que permite generar un paquete *.deb* de manera simple y rápida para ser instalado. Este paquete se genera con todo lo necesario para correr el juego. El

paquete generado instala el binario ejecutable en `/usr/local/bin` y sus respectivos recursos necesarios para visualizarlo correctamente en `/usr/local/share/wolfenstein3d-<modulo>` siendo el modulo `client`, `server` o `editor`.

Para generar los paquetes se debe primero clonar el repositorio:

```
$   git clone https://github.com/mjsantoni/taller_wolfenstein3D.git
```

3.1. Servidor

Para generar el paquete del servidor se deberá hacer lo siguiente, parado en la carpeta raíz del repositorio ya clonado:

```
$   cd server_src/package_generation
$   make
```

Para luego instalar el paquete:

```
$   make install
```

Si se desea generar e instalar el paquete directamente con un solo comando:

```
$   make install-all
```

Si se desea desinstalar el paquete ya instalado:

```
$   make remove
```

3.2. Cliente

Para generar el paquete del cliente se deberá hacer lo siguiente, parado en la carpeta raíz del repositorio ya clonado:

```
$   cd client_src/package_generation
$   make
```

Para luego instalar el paquete:

```
$   make install
```

Si se desea generar e instalar el paquete directamente con un solo comando:

```
$   make install-all
```

Si se desea desinstalar el paquete ya instalado:

```
$   make remove
```

3.3. Editor

Para generar el paquete del editor se deberá hacer lo siguiente, parado en la carpeta raíz del repositorio ya clonado:

```
$   cd editor_src/package_generation
$   make
```

Para luego instalar el paquete:

```
$   make install
```

Si se desea generar e instalar el paquete directamente con un solo comando:

```
$   make install-all
```

Si se desea desinstalar el paquete ya instalado:

```
$   make remove
```

4. Servidor

El servidor fue construido para funcionar de forma concurrente para varios jugadores que estuviesen conectados en múltiples partidas al mismo tiempo. Para esto se construyó un módulo principal que maneja el flujo general de la partida y procesamiento de eventos, delegando responsabilidades a otras clases. El sistema de comunicación con los clientes se delega a un ClientsManager. Se utiliza un hilo receptor y un hilo emisor para cada cliente, así como también un hilo por cada bot para permitir completa concurrencia entre jugadores. El servidor principal que acepta clientes corre en un hilo aparte de manera que pueda aceptar clientes y generar partidas sin influir en las demás. A continuación se muestra un gráfico que explica estas relaciones entre hilos demarcados con líneas punteadas.

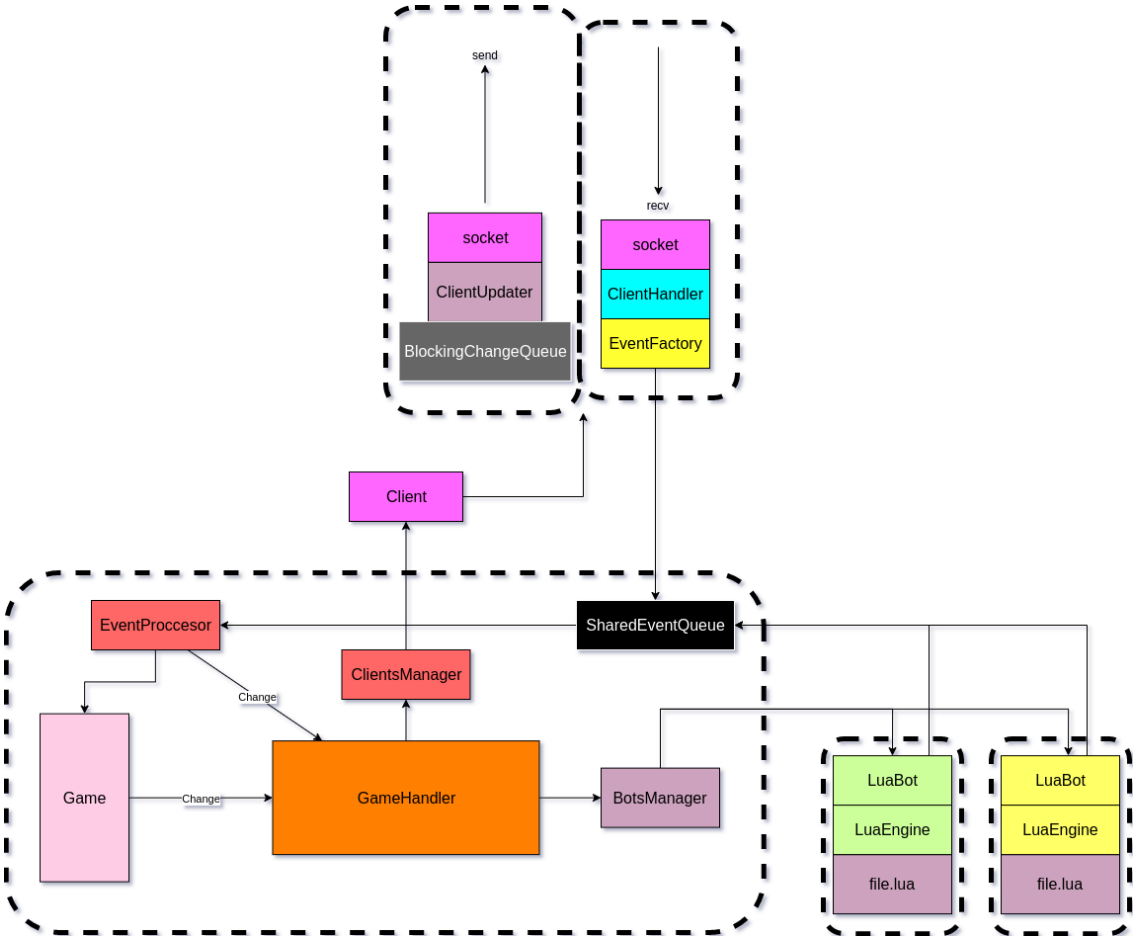


Diagrama general de threads y flujo del servidor

4.1. Manejo de multiples partidas

Para el correcto funcionamiento de las diferentes partidas se dividió la lógica en varias clases.

- **Server:** Esta clase se encarga de aceptar a los nuevos clientes y delegarlos al ServerMenuHandler. Para asi poder seguir atendiendo clientes mientras los demas van optando por crear o unirse.
- **ServerMenuHandler:** Maneja la comunicación con el cliente, previo al juego en si. En base a las decisiones que va tomando un cliente, el Server envía una respuesta acorde; por ejemplo en el caso de que el cliente desee unirse a las partidas existentes, es importante enviarle cuales están disponibles. En caso de que decida crear una partida; esta clase crea un nuevo GameHandler y lo guarda en la clase de Matches.
- **Matches:** Clase contenedora de GameHandlers, posee todas las partidas en curso y se encarga de que el acceso a las mismas sea de forma ordenada a traves de los multiples threads.
- **GameHandler:** Clase principal de una partida. Cuando un cliente decide crear una partida nueva, lo que hace es instanciar un nuevo GameHandler, en el cae la responsabilidad de ir agregando a los nuevos clientes que llegan por el join, crear las clases que facilitaran la comunicación entre ellos y posteriormente ejecutar el "game-cycle", que es procesar eventos de los clientes, generar los cambios y enviárselos a cada uno de ellos. Esto se repite hasta que termine la partida, ya sea por tiempo o por que un jugador logro la victoria.

- **EventProcessor:** Clase que procesa los diferentes eventos recibidos por el GameHandler; es el encargado de traducir el evento a acciones; luego pasara dichas acciones al juego principal y en función de las respuestas, generara mas cambios o no de los que obtuvo.
- **Game:** Esta clase realiza las acciones propias del juego que se le piden, ya sea mover un jugador, hacer que un jugador dispare, intentar abrir una puerta, etc. Luego, produce los cambios y los devuelve. En rasgos generales es un contenedor del estado completo del juego, el cual se ve modificado únicamente cuando se le pide realizar una acción.
- **Map:** Clase que encapsula la lógica del mapa, posee todos los items y posiciones de los jugadores. Para facilitar el acceso se utiliza una clase Coordinate, que simularía un par (x,y); esto nos permite definir métodos propios que nos facilitan otras operaciones.

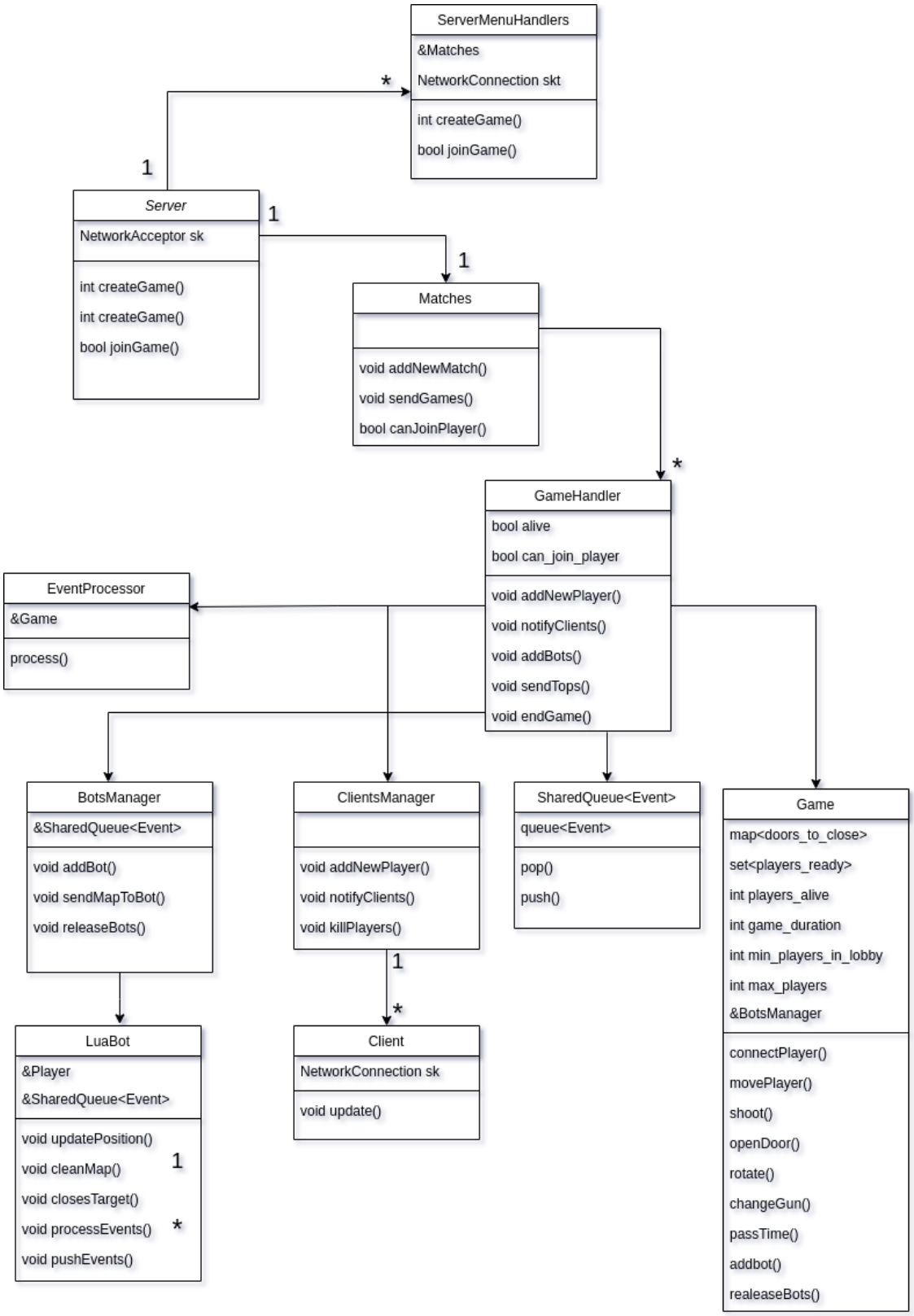
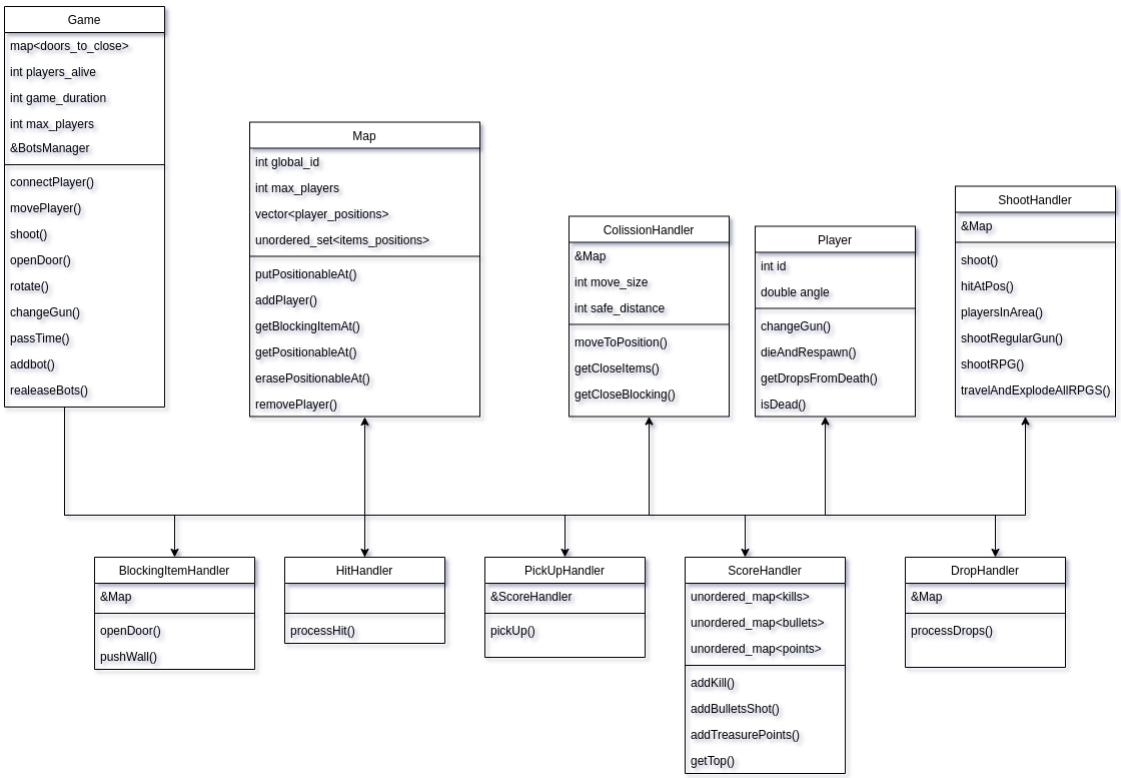
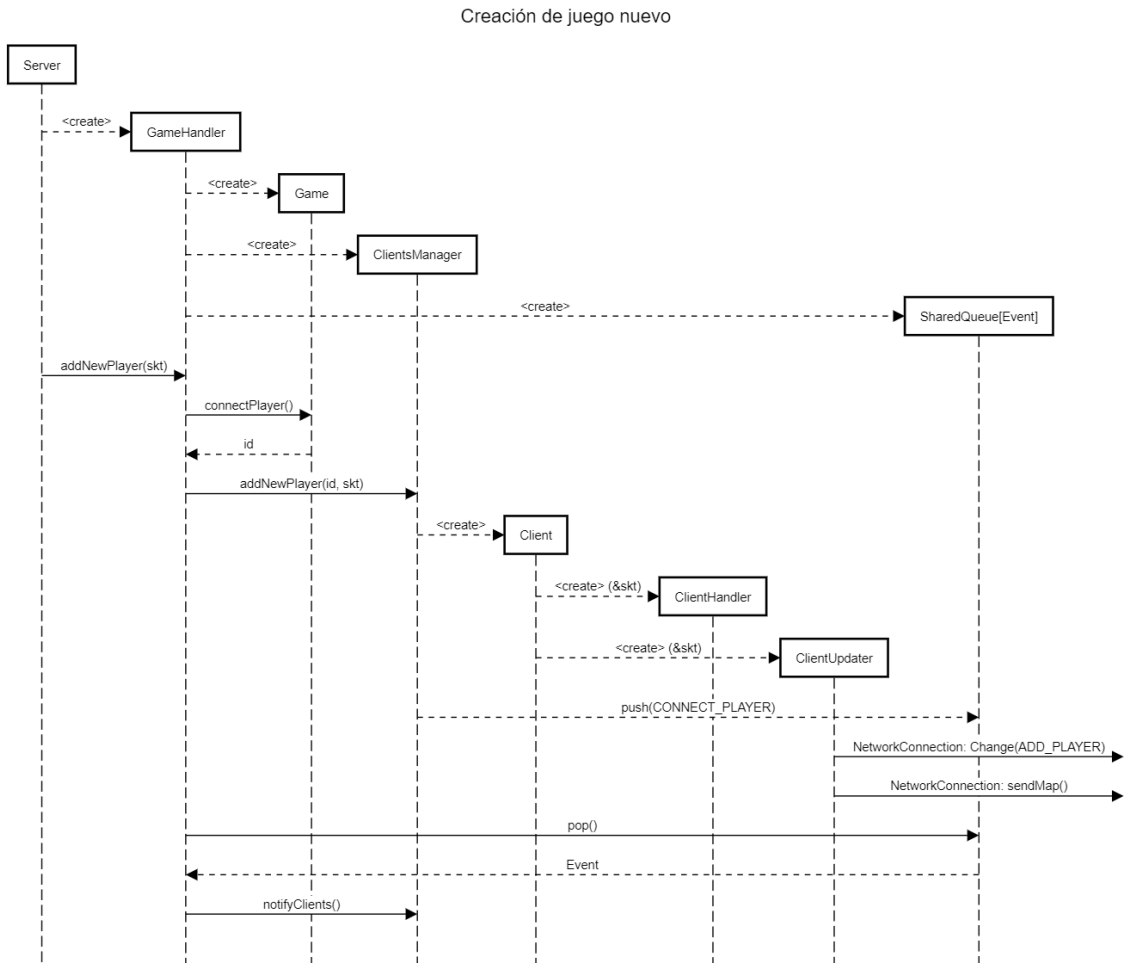


Diagrama de clases de la estructura general de una partida.



Detalle mas especifico de la clase Game.

A continuación se visualiza un diagrama que demuestra el flujo que sucede al crear una nueva partida.



Creación de juego nuevo

4.2. Comunicación con clientes

El GameHandler posee además un ClientsManager que es el encargado de poseer el manejo de clientes y sus comunicaciones. Este ClientsManager posee un vector de Client alocados en el heap que permite abstraer la lógica del envío y recepción de datos. La clase Client almacena:

- **ClientHandler:** clase que corre en un thread aparte y es la encargada de recibir los eventos generados por el cliente conectado.
- **ClientUpdater:** clase que corre en un thread aparte y es la encargada de enviar los cambios producidos en el juego al cliente conectado.
- **El socket de la conexión.**

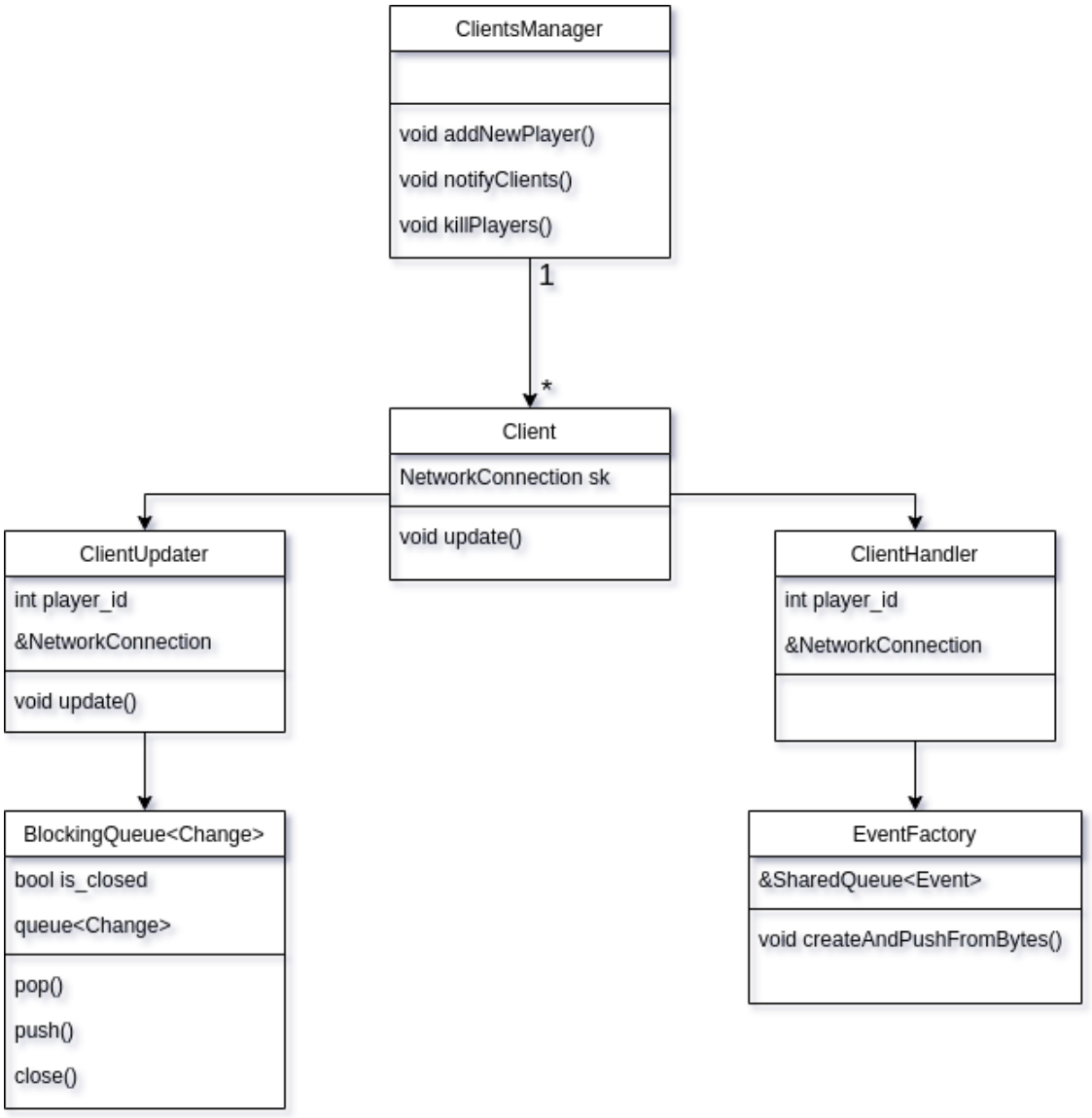
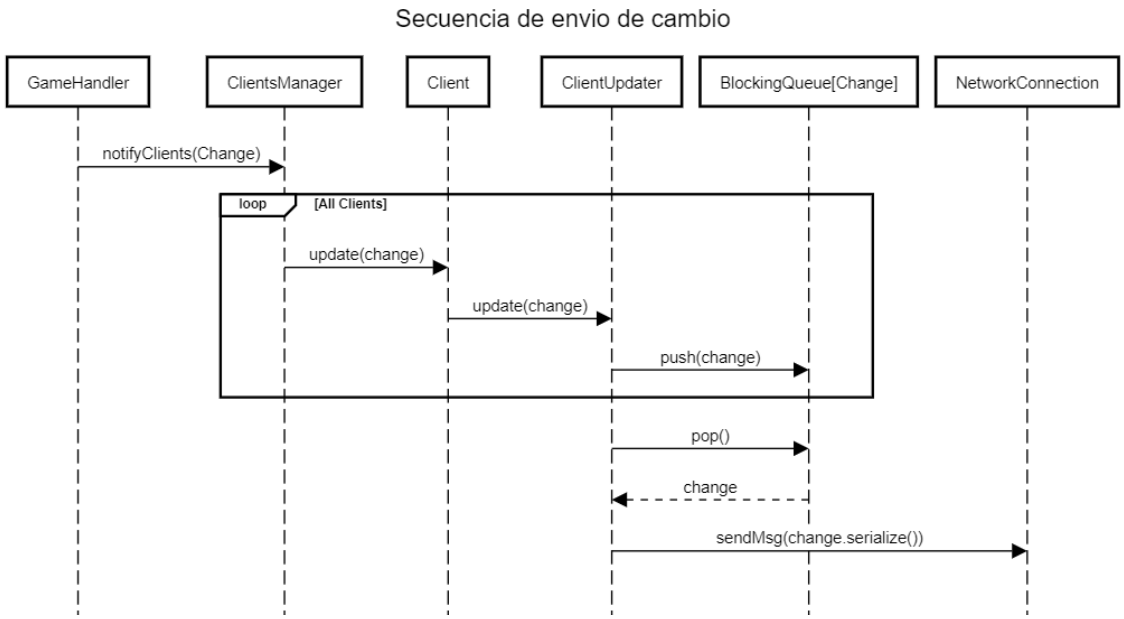


Diagrama de clases del apartado cliente

A continuación se evidencia el flujo de envío de cambios hacia los clientes, cuyo encargado es el ClientUpdater.



Envío de cambios

4.2.1. Protocolo de envio de información

El cliente al presionar una tecla genera un Event, un evento, este paquete generado posea 3 valores clave, el ID del evento, el ID del jugador y un valor. Este último es utilizado en los casos donde existen más de una alternativa, por ejemplo rotar la cámara, que puede ser para la izquierda o la derecha; entonces en ese campo se carga la dirección.

El servidor al procesar los eventos, obtiene un Change (un cambio); similar al evento, esta clase posee 4 campos claves; el id del cambio, el id del jugador y a diferencia del evento en este caso obtenemos 2 campos de valores. Ahora necesitamos 2 en vez de 1 por que al tener un mapa bidimensional, es necesario mandar una coordenada (x, y) cuando alguien se mueve, se elimina un ítem del suelo o se modifica alguna pared/puerta.

Para facilitar el envió, ambas clases poseen un método de serialización. Esto no es mas que tomar los valores ordenados y armar una cadena de char en donde estos valores están separados por un '/'.

Lista de eventos (se encuentra en include/common/events.h):

```
#define INVALID 0
#define CONNECT_PLAYER 1
#define MOVE_PLAYER 2
#define SHOOT 3
#define OPEN_DOOR 4
#define PUSH_WALL 5
#define TURN_CAMERA 6
#define CHANGE_GUN 7
#define PLAYER_READY 8

//VALORES PARA EVENTOS
#define MOVE_LEFT 0
#define MOVE_RIGHT 1
#define MOVE_UP 2
#define MOVE_DOWN 3
```

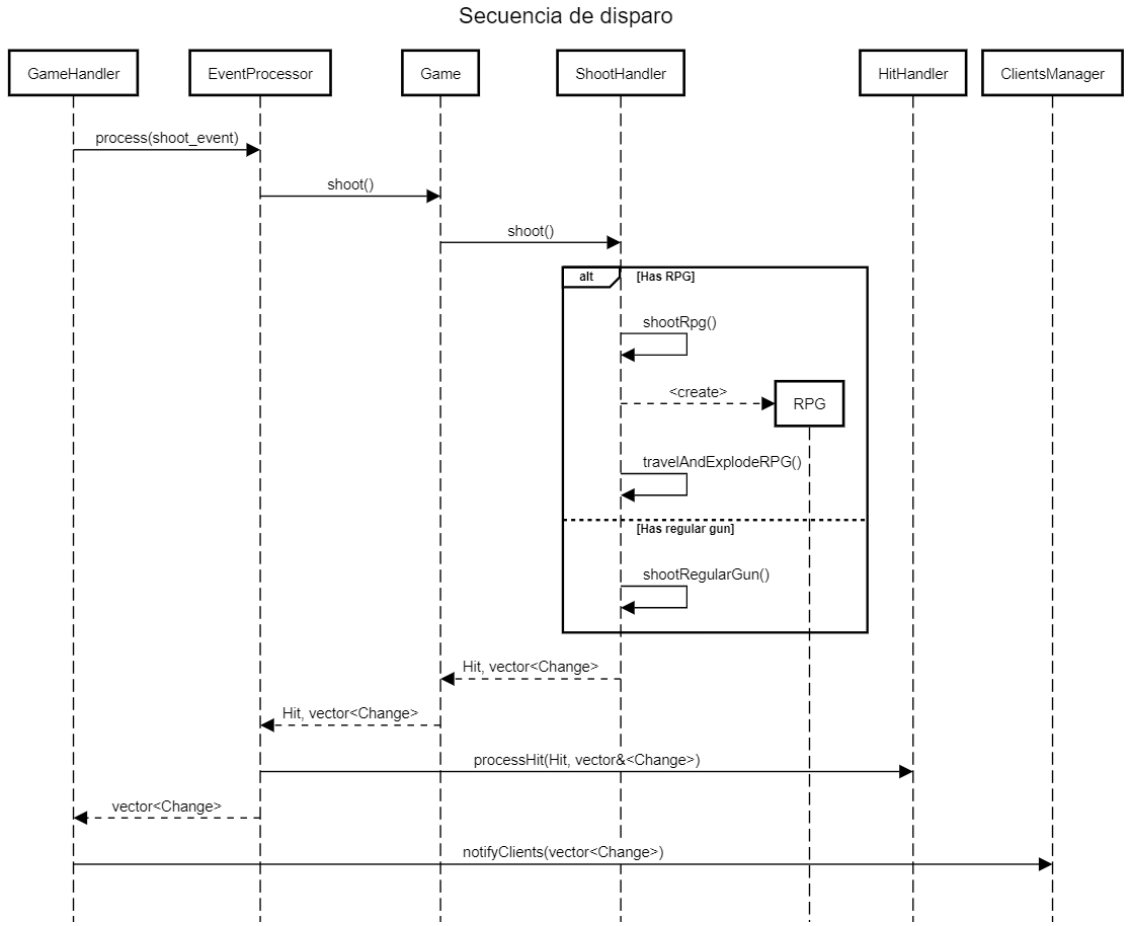
```
#define CAMERA_LEFT 1
#define CAMERA_RIGHT (-1)
```

Lista de cambios (se encuentra en include/common/changes.h):

```
#define INVALID 0
#define REMOVE_POSITIONABLE 1
#define MOVE_PLAYER 2
#define CHANGE_POINTS 3
#define CHANGE_HP 4
#define CHANGE_AMMO 5
#define CHANGE_WEAPON 6
#define CHANGE_KEY 7
#define KILL_PLAYER 8
#define RESPAWN_PLAYER 9
#define ADD_PLAYER 10
#define ADD_BLOOD_PUDDLE_AT 11
#define ADD_BULLETS_AT 12
#define ADD_KEY_AT 13
#define ADD_MACHINE_GUN_AT 14
#define ADD_CHAIN_GUN_AT 15
#define ADD_RPG_GUN_AT 16
#define ADD_UNLOCKED_DOOR 17
#define RPG_MOVE_TO 18
#define RPG_EXPLODE_AT 19
#define MAP_INITIALIZER 20
#define TOP_KILLER 21
#define TOP_SHOOTER 22
#define TOP_SCORER 23
#define TOTAL_PLAYERS_CONNECTED 24
#define GAME_START 25
#define GAME_OVER 26
#define GAME_OVER_NETWORK_ERROR 27
// CAMBIOS AUTO GENERADOS POR EL CLIENTE
#define CL_UPDATE_DIRECTION 28
```

Entonces, si el cliente apretara la tecla espacio o el click izquierdo y fuera el jugador 1, el evento generado seria: 3/1/0 El servidor al procesar este evento (suponiendo que no le pego a nadie y gasto una bala) devolvería: 5/1/-1/0

4.3. Lógica de Disparo



Disparo

4.4. Lógica de apertura de puertas/paredes falsas

A partir de un evento generado por el cliente, ya sea por presionar la 'E' (abrir puerta) o la 'F' (empujar pared falsa) se procesa el cambio a través de la clase Game con su respectiva clase encargada de realizar la lógica de obtención de la puerta o pared cercana que esté en ángulo visible por el jugador para luego removerlas del mapa. En el caso de la pared ésta desaparece completamente, mientras que la puerta vuelve a colocarse en el mapa luego de cierto paso del tiempo generando su cierre, ésta estará siempre desbloqueada. Las puertas se cierran y pasan a estar desbloqueadas por dos motivos: la puerta cerrada fue abierta con una llave lo que genera que pase a quedar desbloqueada o la puerta ya estaba desbloqueada y fue abierta sin necesidad de una llave.

4.5. Lógica del paso del tiempo

Cada cierta cantidad de milisegundos, se genera un "paso del tiempo", esto implica:

- Avanzar todos los RPGs que se encuentran en vuelo. Esto no es más que intentar mover al rpg por su ruta definida al momento de su lanzamiento, validando que no se encuentra con un player o con una pared en estas nuevas posiciones. En caso de colisionar se ejecuta la explosión.

- Intentar cerrar todas las puertas abiertas. Las puertas se cierran cada aproximadamente 6 segundos; pero si al momento de intentar cerrarse hay un jugador en la zona, se cancela el cerrado y se deja pendiente para el próximo paso del tiempo.

5. Cliente

El Cliente es el módulo encargado de la comunicación más cercana con el usuario, tanto recibiendo el input del mismo a lo largo de todo el juego y traduciéndolo para que el Servidor genere las acciones de juego correspondientes, como siendo el encargado de renderizar constantemente la pantalla del juego para que el usuario pueda ver lo que está pasando. De esta manera, se puede dividir al Cliente en tres secciones diferenciadas:

- Las clases encargadas de procesar los eventos generados por el usuario y los cambios recibidos desde el Servidor como consecuencia de estos eventos.
- Las clases encargadas de renderizar la pantalla del jugador, basándose en la técnica de Ray Casting (resumida más adelante).
- Las clases encargadas propiamente de comunicarse con el Servidor, enviando eventos y recibiendo cambios constantemente.

A continuación, se da una vista a alto nivel de los tres grupos de clases, a través de un Diagrama de Clases.

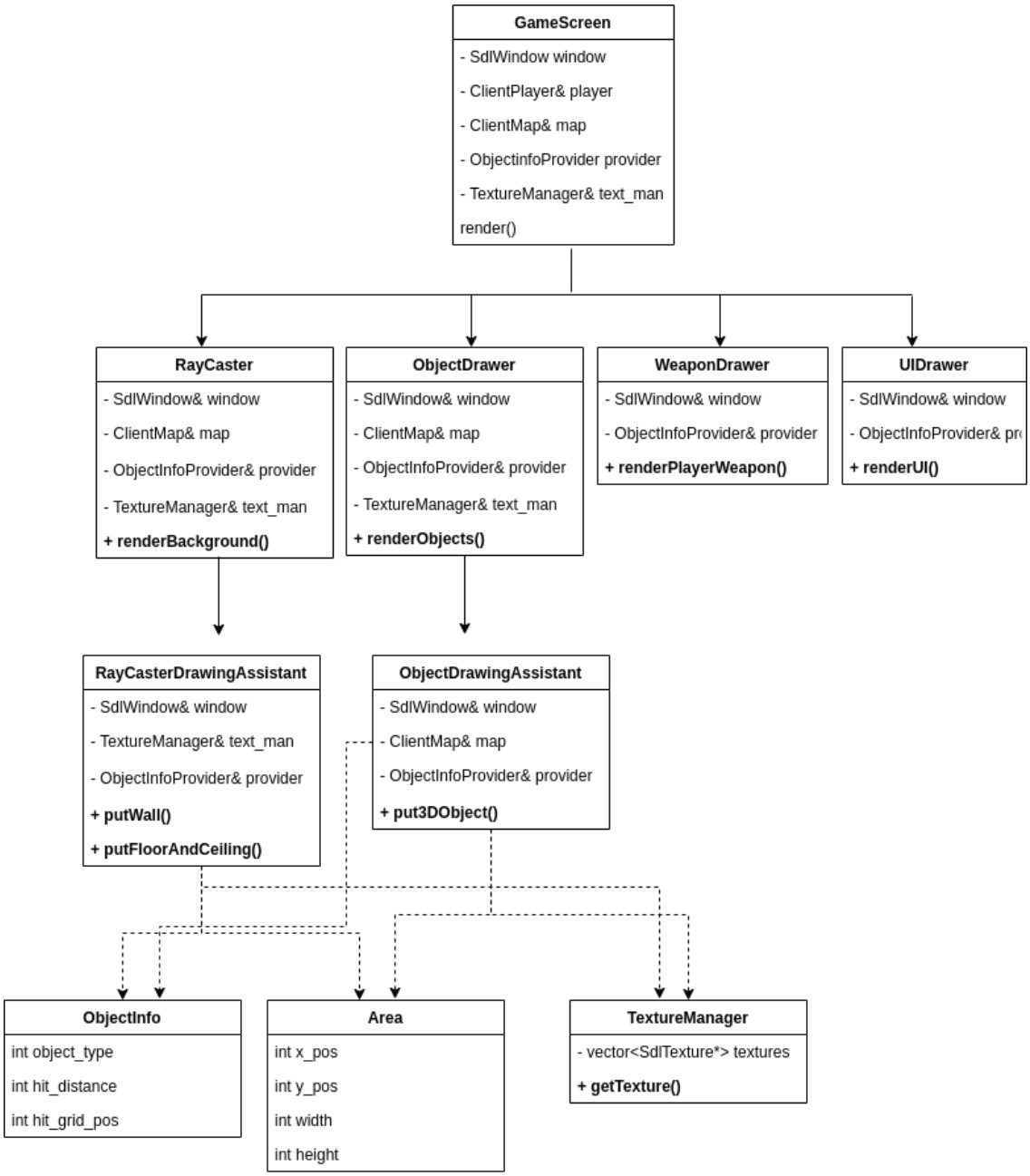


Diagrama de clases: renderización de pantalla

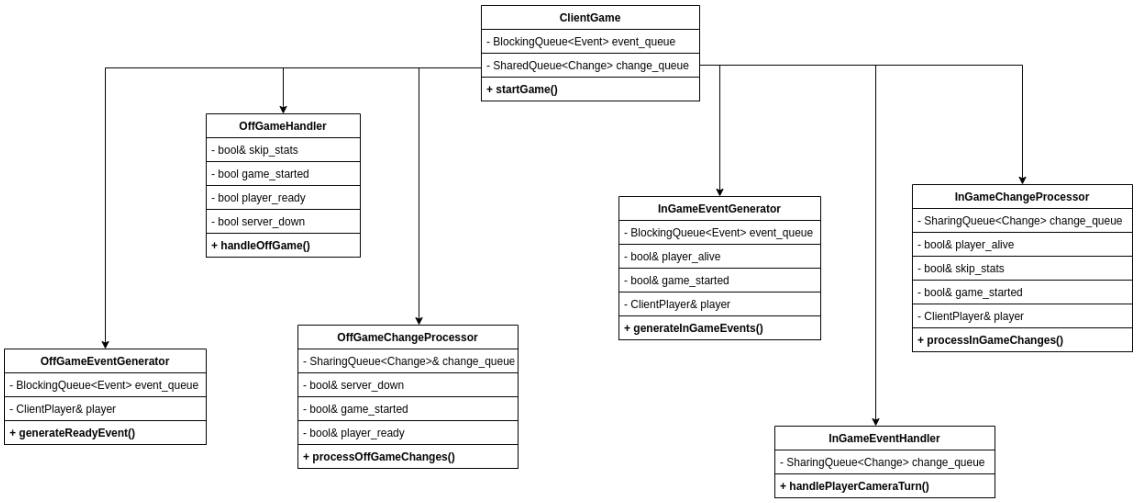


Diagrama de clases: eventos y cambios

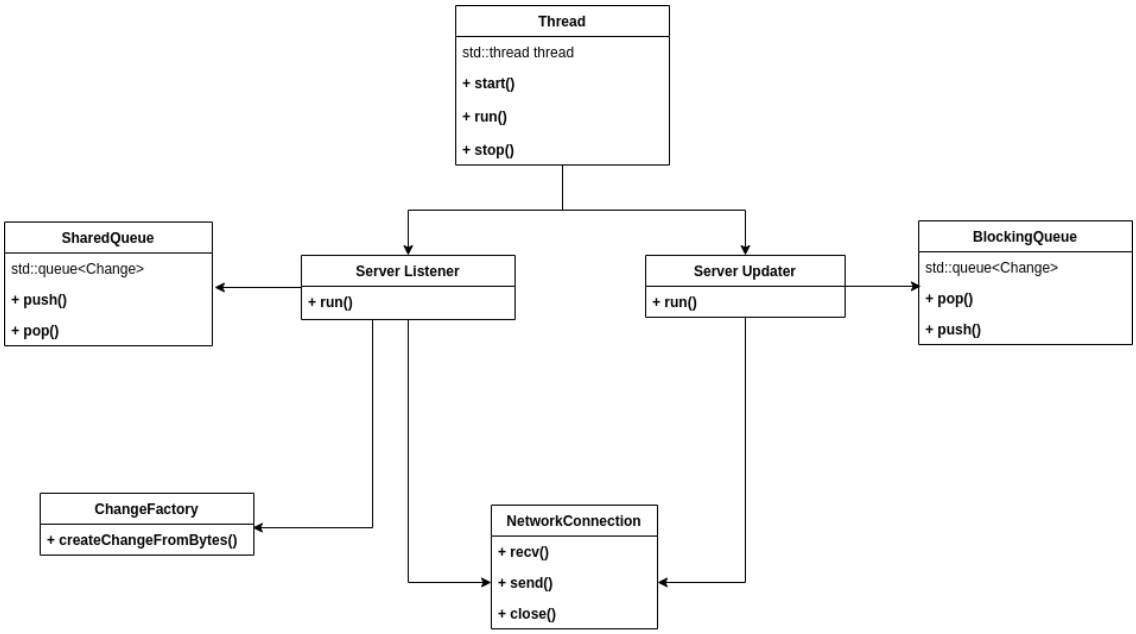


Diagrama de clases: comunicación con el Servidor

5.1. Threads y comunicación con el servidor

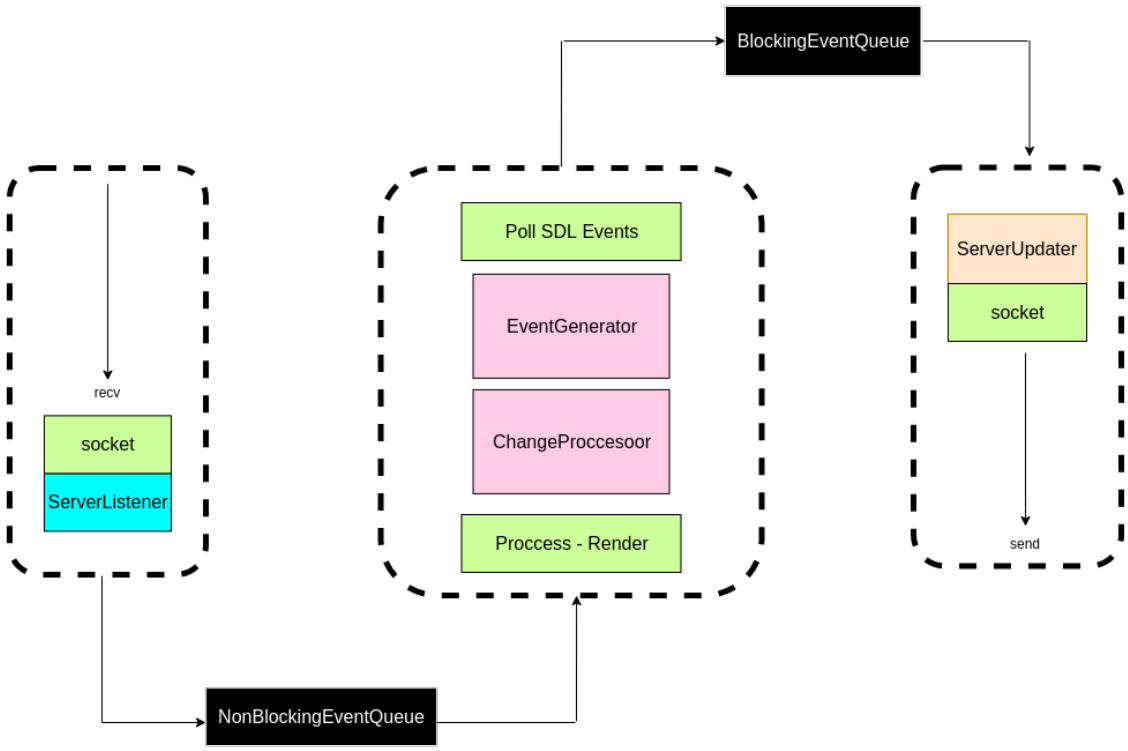


Diagrama general de threads del Cliente

Esta tarea es llevada a cabo por dos threads, uno que le envía Events en forma de mensajes constantemente (Server Updater) y otro que recibe cambios procesados por el Servidor (Server Listener). Ambos cuentan con una referencia a un socket y a una de dos colas, cuyo dueño es la clase Client La principal diferencia entre ambos threads, es que mientras que el Server Listener queda bloqueado recibiendo un mensaje del Socket, el Server Updater queda bloqueado en la función pop de la Blocking Queue.

5.2. Renderización en pantalla

5.2.1. Introducción

Wolfenstein 1992 es un juego que utiliza una técnica conocida como Ray Casting para traducir un mapa en dos dimensiones (ancho y largo) en una pantalla que simula ser 3D, ya que permite al usuario sentir una sensación de profundidad a lo largo de la partida. Esta técnica, utilizada por primera vez en 1985 en el juego Alternate Reality: The City, consiste en, tomando al jugador como un punto en un mapa de dos dimensiones, lanzar a partir de su posición, y dependiendo del ángulo en el que mire, una serie de rayos (el número de rayos varía según la implementación) los cuales le permiten al jugador conocer la distancia exacta de las paredes o puertas (elementos del entorno) que tenga más cerca. Conociendo el ángulo y la distancia a la que se encuentra una pared, se puede dibujar la misma en la pantalla de manera de simular una visión 3D del jugador.

5.2.2. Proceso de renderización del entorno del jugador

El siguiente diagrama de secuencia muestra a muy alto nivel el proceso de renderización de las paredes o puertas que conforman el entorno del jugador, para una posición y ángulo determinados.

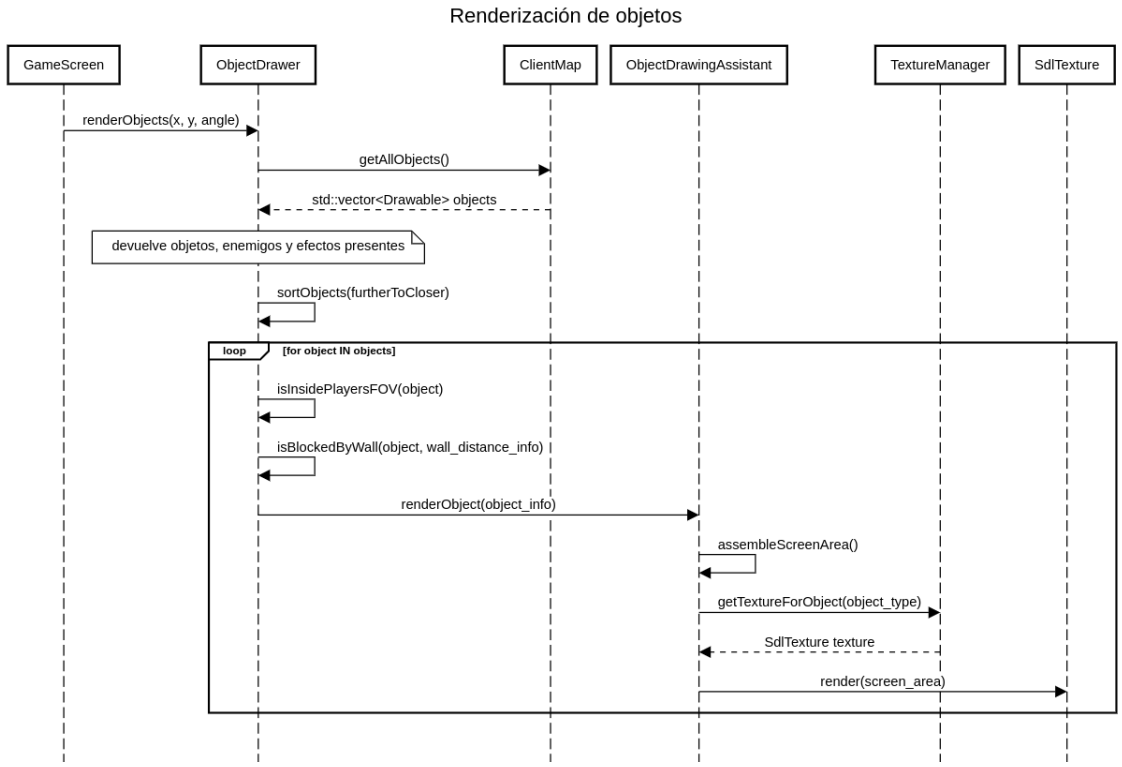


Diagrama de secuencia: generación y envío de eventos

Sobre las clases que participan en esta secuencia:

- **Ray Caster:**
  Es la clase principal en la renderización del entorno del jugador. A partir de una posición **x**, una posición **y**, y un determinado ángulo, se encarga de mostrar las paredes y/o puertas más cercanas al jugador. No se encarga de la renderización de objetos presentes en el mapa, aunque guarda en un diccionario <ángulo, distancia>la distancia más cercana obtenida para cada ángulo, la cual será importante para la clase encargada de renderizar objetos.

- **Ray Casting Drawing Assistant:**
  Se encarga de renderizar propiamente las texturas en la pantalla. El RayCaster encuentra, para cada ángulo, la información necesaria (distancia, tipo de objeto encontrado, posición relativa en la que el rayo chocó con ese objeto) y se la transfiere al Assistant a través de la clase ObjectInfo. Esta clase puede, a partir de esos datos, obtener la posición y dimensiones de pantalla en la que deberá dibujar tanto la pared como el suelo y el techo. Para conseguir la textura correspondiente, el Assistant recurre al TextureManager, quien, a partir del tipo de objeto, puede devolverle la SdlTexture (wrapper para SDL_Texture, propia de la librería SDL), quien puede dibujarse a sí misma en pantalla.

- **Object Info:**
  Contiene toda la información necesaria sobre un objeto, desde el tipo de objeto y la ruta de su imagen, hasta sus dimensiones en el mapa. También puede guardar información contextual, como la distancia y posición relativa en la que un rayo la encontró (que varían según la posición y ángulo del jugador).

- **Client Map:**
  Es una clase relativamente central del Cliente, y que es muy utilizada tanto por el RayCaster

como por el ObjectDrawer. El RayCaster le consulta continuamente, para cada rayo, si en una determinada posición existe o no una pared o puerta. También necesita consultarle si un rayo está o no dentro del rango del mapa, ya que utiliza esta respuesta como condición de corte al lanzamiento de cada rayo.

5.2.3. Proceso de renderización de objetos en el mapa

El siguiente diagrama de secuencia muestra a muy alto nivel el proceso de renderización de los objetos presentes en el mapa, para una posición y ángulo determinados.

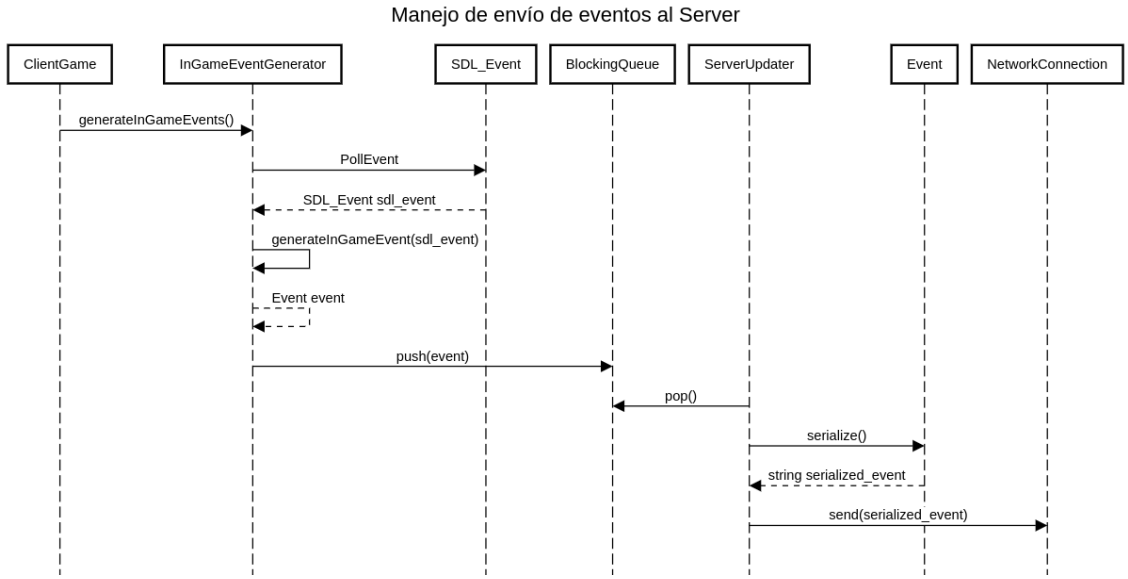


Diagrama de secuencia: generación y envío de eventos

Sobre las clases que participan en esta secuencia:

- **Object Drawer:**
  Es la clase principal en la renderización de objetos. Le solicita al ClientMap todos sus objetos (entre los cuales se incluyen objetos propiamente dichos, enemigos y efectos (un misil, una explosión o un efecto de sangre). A partir de una posición **x**, una posición **y**, y un determinado ángulo, verifica qué objetos deben mostrarse (sólo aquellos que entren en el ángulo de visión del jugador y que no estén bloqueados por una pared). Para estos objetos, calcula la distancia y el ángulo relativo al jugador en el que se encuentran, y delega al ObjectDrawingAssistant la responsabilidad de dibujarlos en la pantalla.

- **Object Drawing Assistant:**
  De forma análoga al RayCastingDrawingAssistant, se encarga de renderizar los objetos en la pantalla, también a partir de información contenida en un objeto ObjectInfo. Calcula en qué parte de la pantalla debe dibujar el objeto, pide la SdlTexture a TextureManager, y simplemente lo renderiza.

- **Client Map:**
  En este caso, su rol es mucho más limitado, ya que solamente es utilizado por el ObjectDrawer para obtener la lista de objetos presentes.

5.2.4. Renderización del arma del jugador

Es un proceso mucho más simple que los anteriores, ya que es simplemente dibujar la textura del arma del jugador en la pantalla. La GameScreen le indica al WeaponDrawer tanto cuál es el arma

que tiene el jugador como la animación actual que debe mostrar (la animación varía dependiendo de si el jugador está o no disparando el arma). El WeaponDrawer utiliza al TextureManager para obtener la imagen adecuada y simplemente la renderiza en la pantalla, con un tamaño prefijado.

5.2.5. UI del jugador

Al igual que la renderización del arma del jugador, es un proceso simple y principalmente harcodeado, ya que se divide a la pantalla en 7 secciones, y en cada una de ellas se muestra un valor correspondiente al jugador. Recibe de la GameScreen la proporción de la vida actual del jugador con respecto a la inicial, para saber qué animación debe mostrar para la cara del jugador (que varía según su vida).

5.3. Generación de eventos y procesamiento de cambios

5.3.1. Generación de eventos

Las acciones del usuario (input de teclado y de mouse) generan eventos en el cliente. En primera instancia, estos eventos son propios de SDL (SDL_Event), los cuales son traducidos en eventos propios del código (Event). Estos eventos son serializados y enviados al Servidor a través del Socket correspondiente.

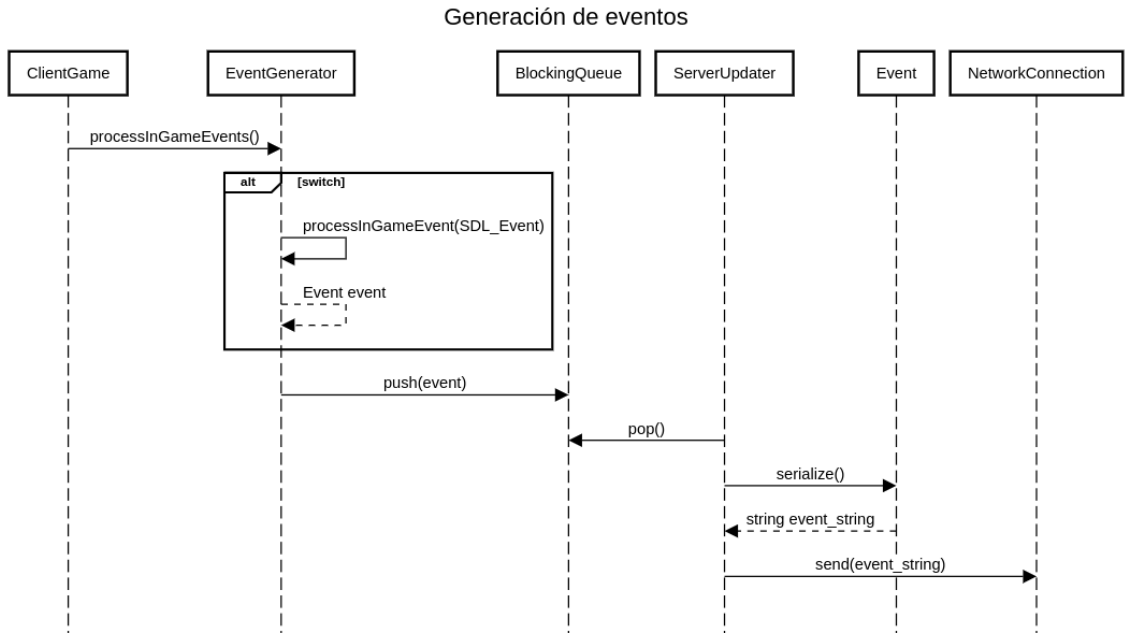


Diagrama de secuencia: generación y envío de eventos

Las clases que participan en esta secuencia son:

- **Event Generator:** Es quien lee los SDL_Event y los traduce en eventos de la clase Event. Simplemente los encola en una BlockingQueue, y vuelve a esperar un nuevo evento.
- **BlockingQueue:** Es una cola bloqueante, en la cual en este caso se encolan Events.
- **Server Updater:** [Thread] Queda bloqueado en el método pop de la Blocking Queue. En cuanto recibe un Event, lo serializa y lo envía al Servidor a través de la Network Connection.

5.3.2. Procesamiento de cambios

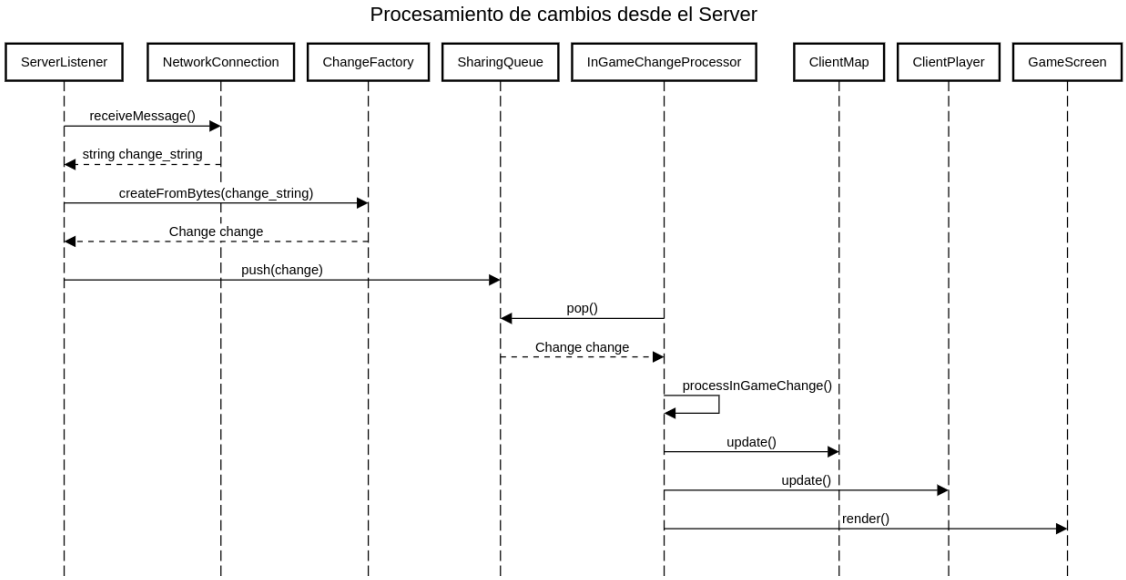


Diagrama de secuencia: procesamiento de cambios

Los eventos generados por los clientes (o por los bots) producen en el juego constantes cambios, que son procesados por el Servidor y enviados a los clientes. El Cliente recibe estos cambios y debe a su vez procesarlos internamente para poder informárselos al usuario (a través de un cambio en el Client Map, en el Client Player, en la Game Screen o en el Audio Manager).

Las clases que participan en esta secuencia son:

- **Server Listener:** [Thread] Está frenado recibiendo mensajes en la NetworkConnection. En cuanto termina de recibir un mensaje, lo transforma en un cambio a través de la Change Factory y lo encola en una Sharing Queue.
- **Sharing Queue:** Es una cola no bloqueante, en la cual en este caso se encolan Changes.
- **In Game Change Processor:**
  Es la clase principal en el procesamiento de cambios, ya que es quien traduce un Change en una acción del juego del cliente. Algunos cambios implican modificaciones en el mapa, en el jugador, en la pantalla o en el audio, por lo que el In Game Change Processor cuenta con todas estas clases como atributos (algunas como referencia, ya que también son usadas por el Off Game Change Processor, entre otras clases). En cuanto un cambio se procesa, se renderiza o no la pantalla de acuerdo a las modificaciones implicadas.
- **Client Map:**
  Es una clase muy importante en esta secuencia, ya que la mayoría de los cambios están relacionados a eventos que suceden dentro del mapa (actualizar la posición de un enemigo, mover al jugador, agregar un efecto).
- **Client Player:**
  Es otra clase importante, ya que se ve modificada por cualquier cambio que afecte directamente al jugador.
- **Game Screen:**
  Es la clase que renderiza los cambios que se produzcan. Puede verse afectada directamente por un cambio (por ejemplo, cuando el jugador cambia de arma), pero, incluso de no estarlo, es la clase que le permite al usuario ver los cambios procesados.

En el diagrama de secuencia de arriba, se mostró el procesamiento de un cambio genérico. A continuación, se mostrarán ejemplos del procesamiento de algunos cambios concretos.

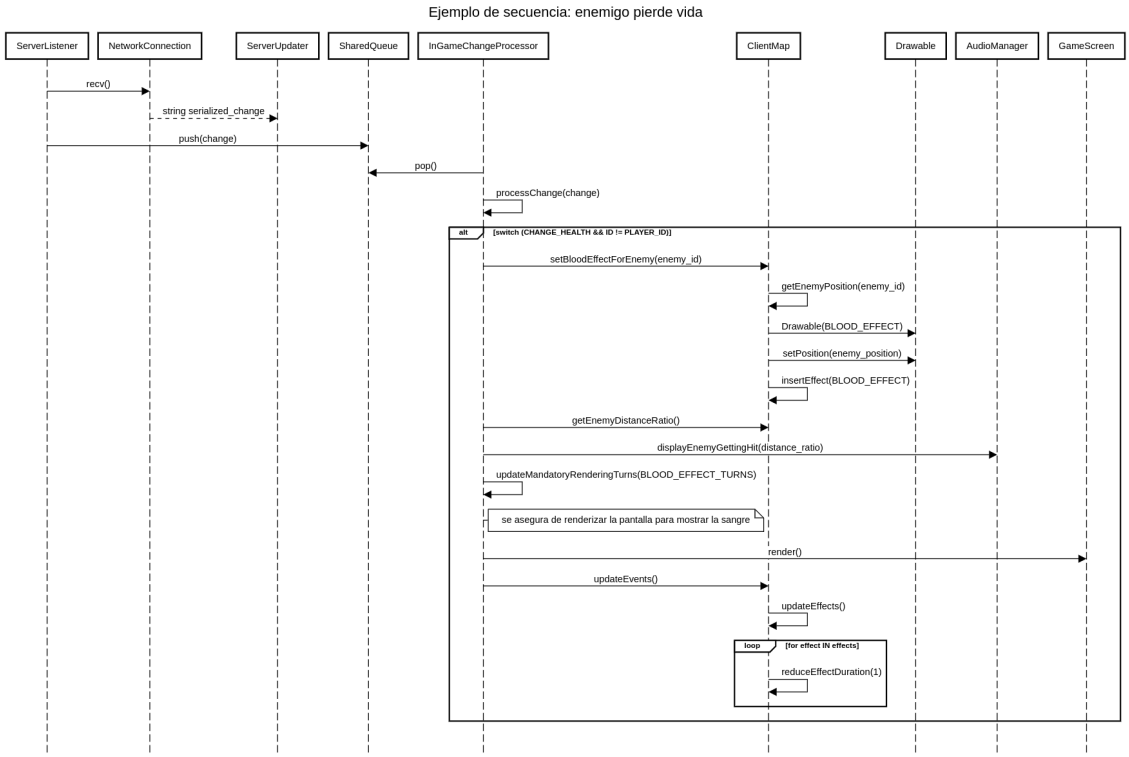
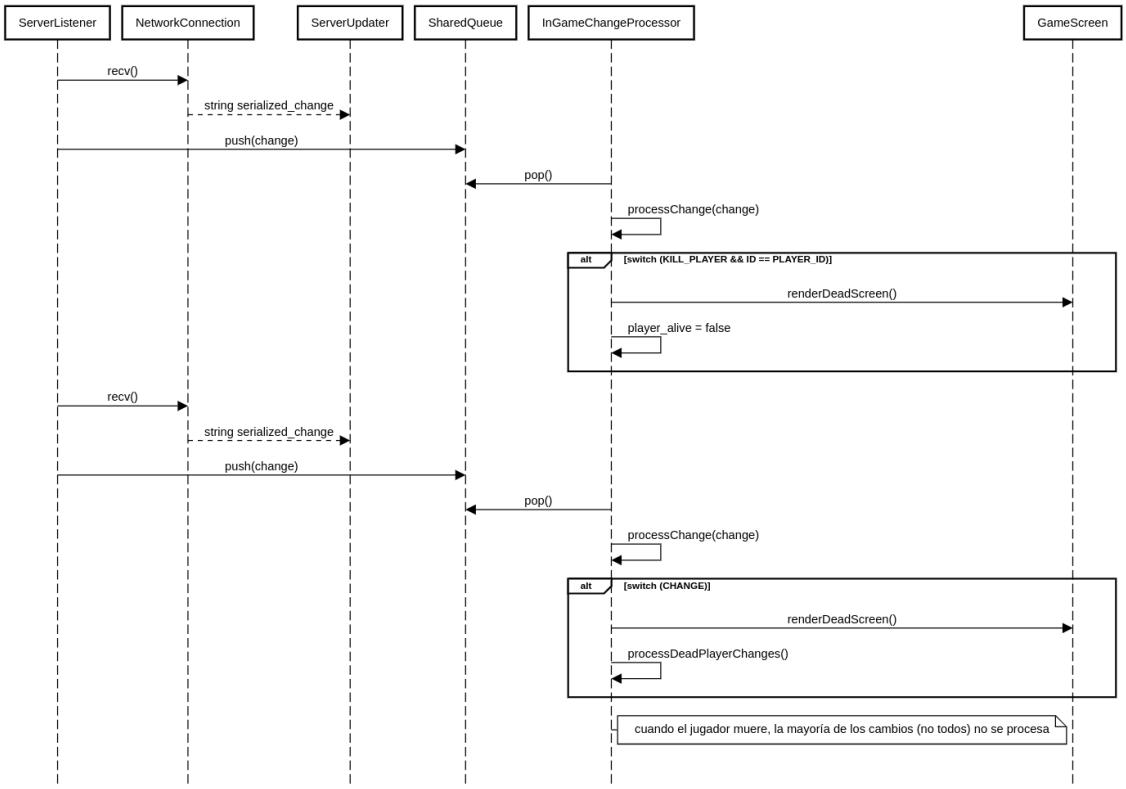
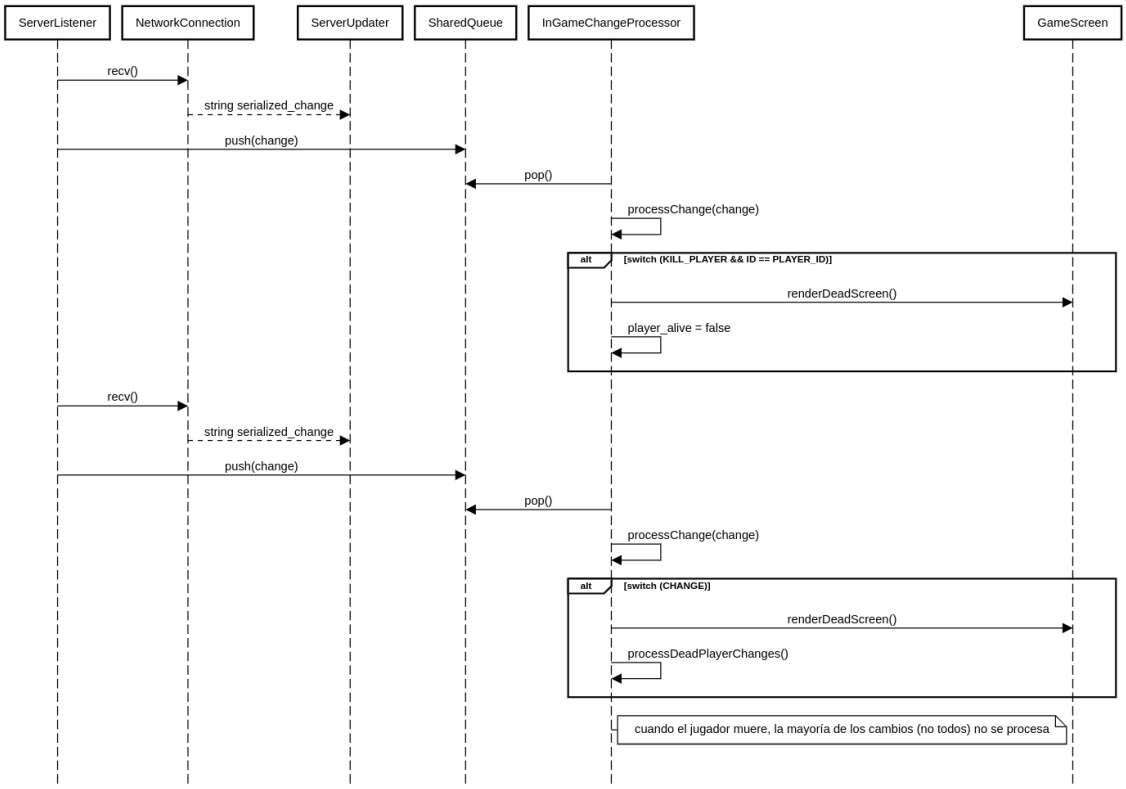


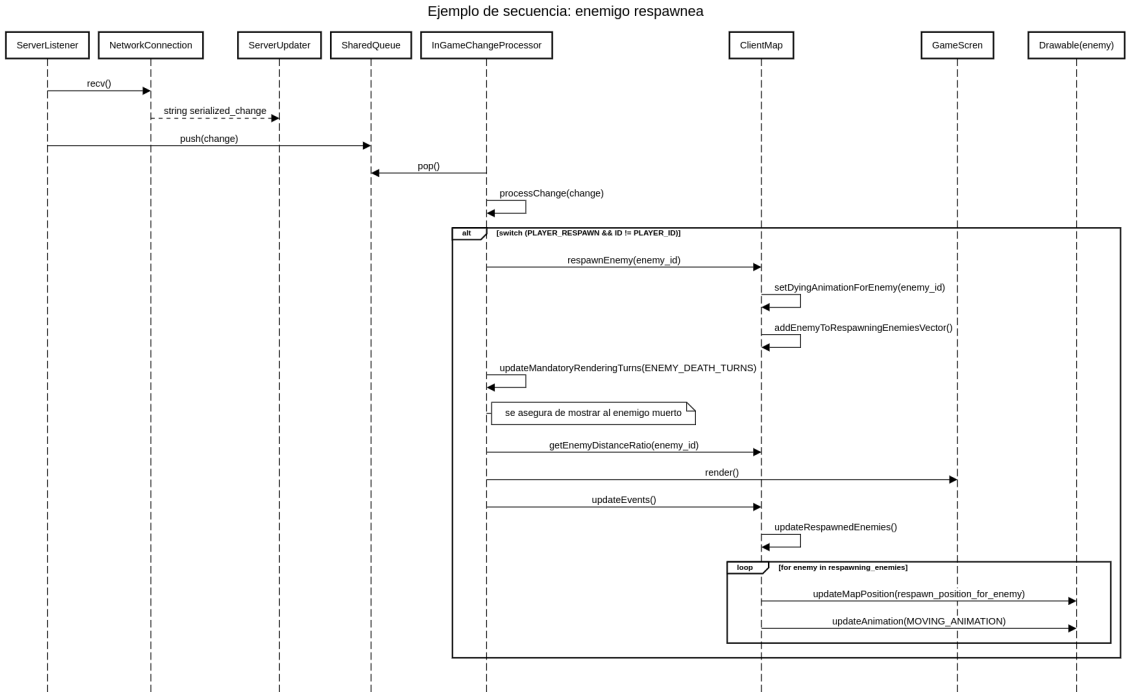
Diagrama general de threads del Cliente

Ejemplo de secuencia: muerte del jugador

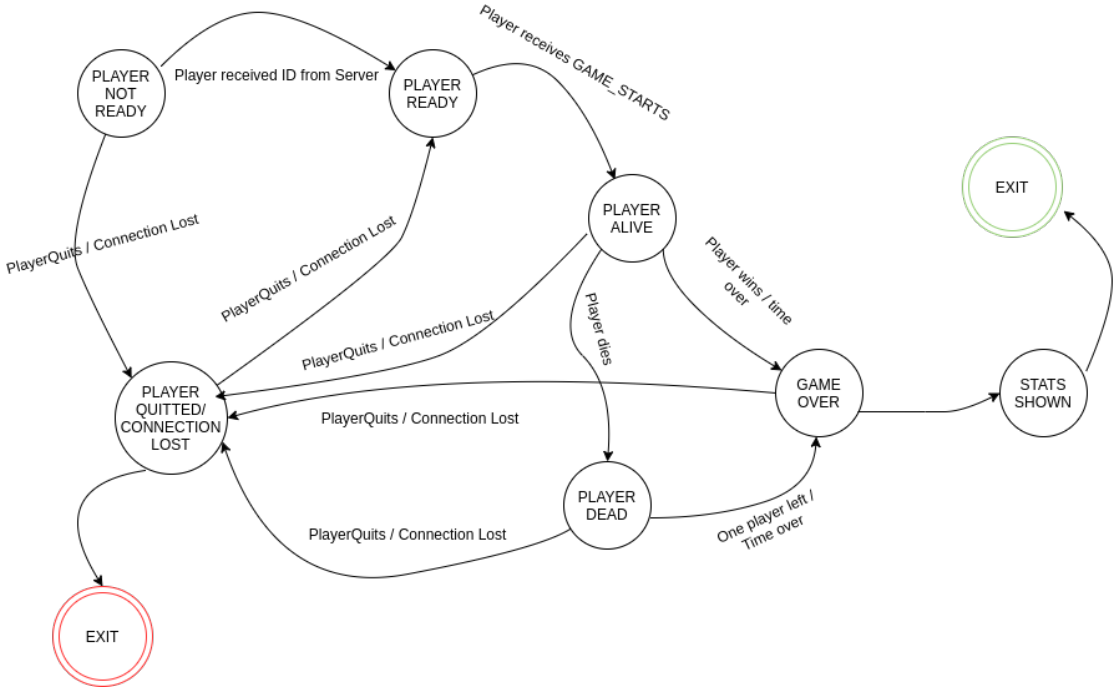


Ejemplo de secuencia: muerte del jugador





5.4. Estados del juego



Estados del juego

5.5. Renderizado del audio

Para renderizar el audio del juego se utilizan principalmente dos clases:

Diagrama de clases: reproducción de audio

- Sdl Audio Player: es una especie de wrapper directo que utiliza métodos de SDL_Mixer, clase propia de la librería SDL.
- Audio Manager: es la clase utilizada tanto por Client Game como por In Game Change Processor para renderizar el audio, y la cual tiene instanciada un Sdl Audio Player.

A continuación, se muestra con más detalle el proceso de reproducir un sonido durante el juego.