
# Documentación técnica

[75.41] Taller de Programación I
Primer cuatrimestre de 2019

Grupo 3

- Camila Bojman 101055 camiboj@gmail.com
- Cecilia Hortas 100687 ceci.hortas@gmail.com
- Nicolas Vazquez 100338 vazquez.nicolas.daniel@gmail.com

# 1 Requerimientos de software

En primer lugar el programa fue desarrollado enteramente en C++ en un sistema operativo Linux por lo que los comandos que se detallarán a continuación son para ese sistema operativo y sus distribuciones afines. Las bibliotecas utilizadas se presentan a continuación:

- `Box 2D`: se encuentra en el repositorio por lo que basta con clonar el mismo
- `Native JSON Benchmark`: igualmente se encuentra en el repositorio
- `SDL`: se debe instalar a partir de los siguientes comandos:

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

- `SDL-mixer`: se debe instalar a partir del siguiente comando:

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

- `YAML`: se debe instalar a partir del siguiente comando:

```
sudo apt-get install libyaml-cpp-dev
```

- `TTF`: se debe instalar a partir del siguiente comando:

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

- `ffmpeg`: se debe instalar a partir de los siguientes comandos:

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

- `CMake`: se debe instalar a partir de los siguientes comandos: [1]

  1. Desinstalar cualquier versión previa de CMake:

     ```
     sudo apt remove --purge --auto-remove cmake
     ```

  2. Ir a la página oficial de CMake y bajar la última versión:

     ```
     http://www.cmake.org/download
     ```

  3. Correr los siguientes comandos:

     ```
     $ version=3.14
     $ build=5
     $ mkdir ~/temp
     $ cd ~/temp
     $ wget https://cmake.org/files/v$version/cmake-$version.$build.tar.gz
     $ tar -xzvf cmake-$version.$build.tar.gz
     $ cd cmake-$version.$build/
     $ ./bootstrap
     $ make -j4
     $ sudo make install
     ```

  4. Verificar la versión de CMake:

     ```
     $ cmake --version
     cmake version 3.14.X
     ```

[1] Fuente: https://askubuntu.com/questions/355565/how-do-i-install-the-latest-version-of-cmake-from-the-command-line

### 1.1 Instalación

En el `root` del repositorio clonado se encuentra un archivo `install.sh`. Para realizar la instalación se deben seguir los siguientes pasos:

```
chmod +x install.sh
./install.sh <path>
```

donde `<path>` es el directorio donde se desea instalar el juego.

Para proceder a jugar al juego se deben seguir los siguientes pasos:

1. Levantar un servidor con el comando `./Server <port>`
2. Conectarse con el cliente con `./Client <host> <port>` siendo `<host>` una dirección IP y `<port>` el mismo puerto del servidor.

Para crear y editar mapas, ejecutar `./Editor`.

## 2 Descripción general

El proyecto se constituye de los siguientes módulos:

- **Servidor**

  Se encuentran tanto las clases designadas al soporte físico del juego como al soporte multijugador y multipartida.

- **Cliente gráfico**

  Se encuentran las clases designadas para modelar el cliente desde un lado gráfico con el uso de la librería `SDL` así como el soporte para manejar la recepción del estado actual del escenario para dibujarlo y el envío de los diversos eventos de usuarios para que sean modelados por el motor físico.

- **Editor**

  Se encuentran las clases designadas para el modelado del editor que se tratan escencialmente de *wrappers* de diversas funcionalidades de `SDL` y `ffmpeg`.

- **Archivos compartidos**

  Se tratan de diversas clases que se consideró que podían ser útiles para más de un módulo por lo que se colocaron en un espacio común para su correcta compilación.

## 3 Archivos compartidos

Se tratan de clases que no necesariamente están relacionadas entre sí por lo que se considera que no es pertinente a la documentación exponer un diagrama de clases. Sin embargo, sí se mostrará su relación con las futuras clases. De esta forma, las principales clases que componen este módulo son:

- **Socket**

Esta clase encapsula todas las operaciones realizadas para la comunicación a través de un socket.

- **Protocol**

Esta clase encapsula el protócolo utilizado para el desarrollo del juego. Contiene una referencia a un `Socket` y modela el envío y recepción de `strings`.

- **Thread**

Esta clase encapsula el funcionamiento de un thread. Su principal atributo es una variable de tipo `std::thread` a la que se le pasa como parámetro una referencia a la función `run()` que encapsula la ejecución.

- **UserEventQueue**

Se trata de una cola bloqueante a la que se encola los diversos eventos de usuarios que deben ser modelados por el motor físico.

- **StageStatusQueue**

Se trata de una cola bloqueante a la que se encola el estado del escenario en un step dado por el escenario para que sea dibujado por los diversos clientes.

- **Window**

Se trata de una clase wrapper de SDL_Window y SDL_Renderer. Se utiliza en absolutamente todas las vistas y operaciones que requieran dibujar a pantalla. La clase fue protegida debido a que ha de ser compartida entre el hilo que procesa input de usuario y el hilo que dibuja, para poder manejar el cambio de tamaño de la pantalla de Fullscreen a Windowed y viceversa.

- **SDLSession**

Se trata de una clase wrapper de la inicialización de los distintos módulos de SDL. Se utiliza en el Cliente y en el Editor.

- **Sprite**

Se trata de una clase wrapper para la carga de texturas en SDL.

- **AnimatedSprite**

Se trata de una clase wrapper para mostrar animaciones por pantalla. Utiliza tiras de sprites cargadas en un Sprite para poder dibujar cada cuadro de la tira de sprites cada cierto tiempo. Cuenta con un timer para determinar cada cuánto tiempo hay que actualizar el cuadro actual que será dibujado en el loop principal del juego.

# 4 Servidor

## 4.1 Descripción general

Las funcionalidades del servidor se dividen en dos partes:

1. Las relativas al motor físico, es decir, el modelado físico del juego.
2. Las relativas al manejo de threading y sockets para brindar un soporte multijugador y multipartida.

A continuación se enumeran las principales clases que se encargan de implementar dichas funcionalidades y sus principales atributos y métodos.

## 4.2 Clases

Para describir las clases utilizadas se dividirán las clases en 5 categorías:

- Soporte de comunicación (threading y sockets)
- Entidades del juego
- Manejo de portales
- Manejo del escenario
- Movimientos del juego

### 4.2.1 Soporte de comunicación

Estas clases se realizaron para modelar la comunicación con el cliente. Son las que encapsulan la comunicación por medio de sockets y el manejo de threads para brindar el soporte multipartida y multijugador del juego.

- **Stage Manager**

Se encarga de controlar todo lo relativo al escenario y la partida a crear. Tiene como principales atributos una cantidad máxima de jugadores, una referencia al escenario de partida, un parser que se encarga de manejar el archivo `YAML` que contiene la información de los niveles a utilizar, una cola de eventos y una cola de clientes que contiene el estado actual de los mapas. Cabe destacar que esta clase es la que se encarga de manejar los eventos de usuario y traducirlos a eventos en el mundo físico.

- **Room Worker**

Esta clase se encarga de manejar la creación y la unión de partidas. Por lo tanto, es necesario que mantenga una referencia al socket del cliente, a su respectivo protocolo que encapsula la comunicación y a la clase `Room Manager` que se encarga de brindar los recursos necesarios para llevar esto a cabo.

- **Room Manager**

Esta clase se encarga de organizar las diversas partidas creadas, agregar los jugadores a las mismas y remover las partidas que finalizaron.

- **Room Acceptor**

Esta clase se utiliza para encapsular la aceptación de los clientes al servidor por lo que tiene como principal atributo un vector de punteros a la clase `Room Worker` para lanzarlos con el método `start` para inicializar los threads.

- **Client Sender**

Esta clase se encarga de mandar a los clientes el estado actual del mundo físico por lo que sus atributos son referencias al socket del cliente y su protocolo así como una cola que guarda el estado actual del escenario.

- **Client Receiver**

Esta clase se encarga de recibir los eventos de usuario para luego manejarlos por lo que mantiene una referencia al socket del cliente y una cola de eventos de usuario para encolar la serie de eventos recibidos.

- **Client Handler**

Esta clase se encarga de manejar e inicializar las clases `Client Sender` y `Client Receiver` por lo que debe tener como atributos una referencia al socket del cliente y tanto una cola de eventos de usuario como una cola de estados actuales del escenario.

### 4.2.2 Entidades del juego

Se realizó una clase para cada elemento del juego de manera de modelizar sus responsabilidades y además se agregaron otras clases para hacer uso del polimorfismo y herencia de C++.

Es pertinente aclarar como elemento base del modelo que se utiliza una clase `Entity` de la cual derivan todas las clases de esta sección para poder modelizar las colisiones entre objetos. De esta manera, en caso de ocurrir una colisión, siempre sucede entre dos objetos de la clase `Entity` y por medio de la herencia se pudo manejar cada caso en particular con la sobreescritura de un método llamado `handleCollisions`.

- **Entity**

Tiene como atributos un string que denota el tipo de objeto y un puntero a un objeto de tipo `b2Body` de la libería Box2D. Tiene el método virtual puro `void handleCollision(Entity* entity)` para que lo implementen las clases derivadas y hagan un correcto manejo de colisiones.

- **Acid**

Su principal método consiste en contempla las colisiones con instancias de la clase `Chell` ya que debe matarla.

- **Shot**

Esta clase se encarga de modelizar el disparo. Tiene como atributos dos punteros a la clase `Coordinate` (que representa una coordenada) tanto para el origen como el destino del disparo, un booleano que indica si finalizó el recorrido, un ángulo de disparo y un puntero a la clase `Chell` para poder asignarle los portales creados a la misma.

- **Blue Shot**

Esta clase hereda de la clase `Shot` y maneja la colisión con un bloque de metal ya que debe crear un portal sobre el mismo de color azul.

- **Orange Shot**

Esta clase hereda de la clase `Shot` y maneja la colisión con un bloque de metal ya que debe crear un portal sobre el mismo de color naranja.

- **Cake**

Esta clase principalmente maneja las colisiones con objetos de la clase `Chell` ya que debe ganar en caso de colisionar.

- **Chell**

Esta clase tiene como atributos un puntero al objeto que alberga el portal naranja, otro al objeto que alberga el portal azul y dos punteros a la clase `Coordinate` que representan los dos portales a teletransportarse. Además guarda un puntero a la clase `PinTool` y a la clase `Rock` en caso de estar agarrando una roca. Sus principales métodos implican moverse a la derecha, moverse a la izquierda, saltar, agarrar una roca y dejarla, agregar los portales una vez realizados y contemplar los diversos casos de colisión con el resto de los objetos,

- **Diagonal Metal Block**

Esta clase tiene como atributo el ángulo de rotación para representar los cuatro bloques en diagonal y su principal método consiste en manejar las colisiones con los objetos de la clase `Energy Ball` para que cambie su dirección en caso de colisionar.

- **Metal Block**

Esta clase tiene como responsabilidad principal manejar las colisiones con las bolas de energía para invertir su sentido y además con los disparos `Blue Shot` y `Orange Shot` para generar los portales.

- **Rock Block**

Esta clase se encarga principalmente de manejar las colisiones con las bolas de energía para que desaparezcan y con los disparos para no generar portales en caso de colisionar.

- **Energy Ball**

Esta clase tiene como método principal el que consiste en volar de acuerdo a la dirección establecida por el emisor de energía y además cambiar su dirección en caso de colisionar con un objeto de la clase `Diagonal Block` o rebotar en caso de colisionar con un objeto de la clase `Metal Block`.

- **Energy Transmitter**

Esta clase se encarga principalmente de controlar la frecuencia de lanzamiento de las bolas de energía. De esta clase derivan las otras cuatro clases que designan las cuatro direcciones posibles que pueden tomar las bolas de energía (arriba, abajo, izquierda o derecha).

- **Energy Bar**

Esta clase principalmente maneja las colisiones con los disparos y con las rocas para evitar el traspaso de las mismas.

- **Gate**

Esta clase tiene como atributos un string que designa la lógica que debe seguirse para abrir la puerta y además un mapa que guarda los distintos botones involucrados en dicha lógica con su id como clave y un puntero al objeto `Button` como valor. Su principal método es el encargado de determinar si la puerta está abierta o cerrada de acuerdo al estado actual de los botones (encendido y apagado).

- **Item Activable**

Esta clase fue implementada para que los dos objetos que pueden ser encendidos del escenario puedan heredar de ella. Por lo tanto, sus principales métodos consisten en activarlos, desactivarlos y determinar su estado.

- **Button**

Esta clase hereda de la clase `Item Activable`. Su principal método consiste en manejar las colisiones con los objetos de la clase `Chell` y `Rock` para poder determinar su estado (encendido u apagado), guardado como atributo.

- **Energy Receptor**

Esta clase hereda de la clase `Item Activable`. Su principal método consiste en manejar las colisiones con los objetos de la clase `Energy Ball` para poder determinar su estado (encendido u apagado), guardado como atributo.

- **Portal**

Esta clase tiene como atributo un enum de tipo `Orientation` que designa la orientación vertical u horizontal del portal y un enum de tipo `Direction` que designa la dirección en que deben salir los objetos al teletransportarse por el otro portal, es decir, izquierda, derecha, abajo o arriba. Se encarga de manejar las colisiones con los objetos de la clase `Chell`, `Energy Ball` y `Rock` para que se teletransporten al topar con un portal.

- **Rock**

Esta clase tiene como atributos un booleano que designa si está vivo o muerto y otro para designar si está siendo agarrado o no por Chell. Su principal método consiste en manejar las colisiones con los diversos objetos del mundo, por ejemplo, un objeto de la clase `Button` para activarlo, de la clase `Chell` para matarla si está por encima de ella y morir en caso de topar con un objeto de la clase `Energy Bar`.

- **PinTool**

Esta clase tiene como método principal el que consiste en eliminar el objeto del escenario luego de pasado cierto tiempo predefenido.

### 4.2.3 Manejo de portales

Esta sección tiene las clases que se utilizan para designar los portales en el escenario. Es necesario seguir una cierta organización para unir los portales a un objeto de la clase `Chell` en particular, borrar un portal de un color en caso de crearse otro del mismo color y resetear los portales (borarlos del escenario). Las clases que se encargan de dichas responsabilidades son las que siguen:

- **Portal Holder**

Esta clase guarda como atributos un puntero a la clase `Coordinate` que representa las coordenadas de un portal, un enum `Orientation` que designa su orientación (vertical u horizontal) y un enum `Direction` que indica para donde deben teletransportarse los objetos en caso de toparse con su portal contrario.

- **Portal Manager**

Esta clase tiene como atributos una referencia a un `unordered_map` que tiene como claves los ids como string de los portales y como valor un puntero al objeto de la clase `Portal`, además de una referencia al objeto de clase `Stage`. De esta manera se encarga de eliminar los portales del mundo y de crearlos de acuerdo a los portales actuales de la clase `Chell`.

### 4.2.4 Manejo del escenario

Estas clases son las encargadas de la organización global de los objetos en el escenario, su disposición en el escenario y su representación en `Box2D`.

- **Coordinates**

Esta clase guarda dos atributos de tipo `float` que representan las coordenadas `x` e `y`.

- **Physics World**

Esta clase se encarga de encapsular la creación de objetos en el mundo de `Box2D`. Por eso sus atributos son un puntero a un objeto de tipo `b2World` y el ancho y el largo del escenario como `floats`. Los principales métodos consisten en agregar rectángulos estáticos y dinámicos al escenario, triángulos estáticos y el

caso particular para el objeto de la clase `Chell`: un rectangulo con dos ruedas en su parte inferior. Esto tuvo que realizarse para evitar que el objeto se trabe en los diversos rectángulos simulados como piso. Además, se encarga de la correcta destrucción de los objetos en el mundo y de simular el *step* de `Box2D`.

- **Stage**

Esta clase tiene como atributos una serie de `unordered_map` donde se guardan todos los objetos del mundo como valor y como clave un id de clase `string` o un objeto de la clase `Coordinate` para el caso de los bloques estáticos. Su principal método es el que simula el `step` y llama a los diversos métodos que deben convocarse en cada llamada a la modelación del mundo físico. Además tiene un método que devuelve el `json` necesario para actualizar las vistas de los diversos clientes.

- **Yaml Parser**

Esta clase es la encargada de unir el editor y el servidor. Tiene como atributos un nodo `YAML` y una referencia al `Stage`. De esta manera, lee el archivo `YAML` generado por el editor, agrega los objetos que debe agregar al escenario y genera los `json` pertinentes para inicializar las vistas.

### 4.2.5 Movimientos del juego

Estas clases se realizaron para designar los movimientos básicos de Chell y la roca. Se siguió una implementación en la que cada clase en su método de actualización llamada al método `move()`, sin importar qué clase se está llamando. De esta manera, las clases que simulan el movimiento `Move Right`, `Move Left` y `Stop` heredan de una clase `Dynamic` y sobreescriben el método `move()`.

La clase `Dynamic` alberga la responsabilidad de simular desde la física el salto de Chell, los movimientos de las piedras y todo lo relativo al vuelo de las bolas de energía así como el manejo de colisiones de los objetos y la teletransportación de los mismos al toparse con un portal.

## 4.3 Diagramas UML

Se procede a exponer los diagramas de clase realizados para las diversas secciones:

### 4.3.1 Entidades del juego

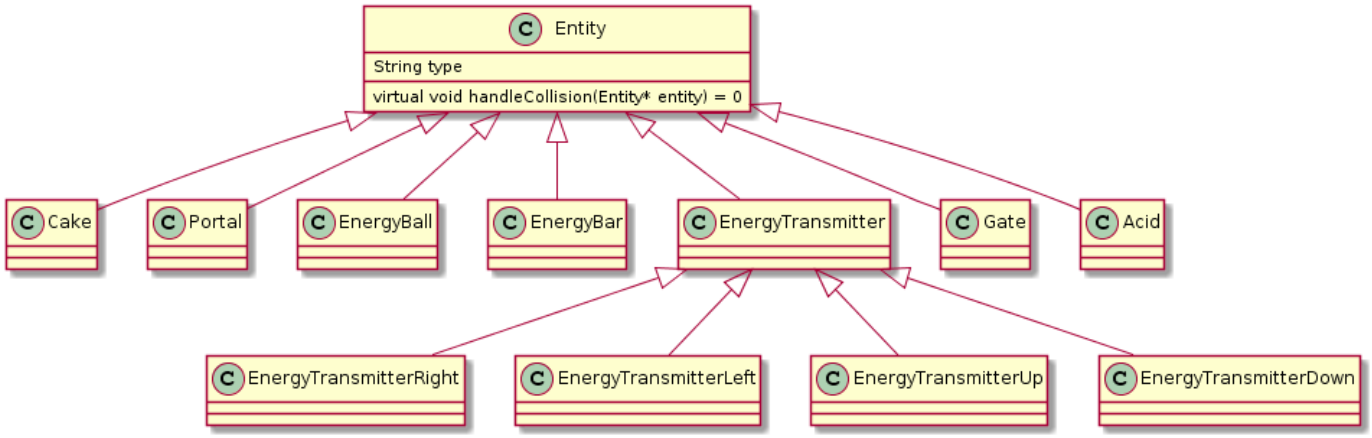


Figure 1: Diagrama de clases de los elementos del juego

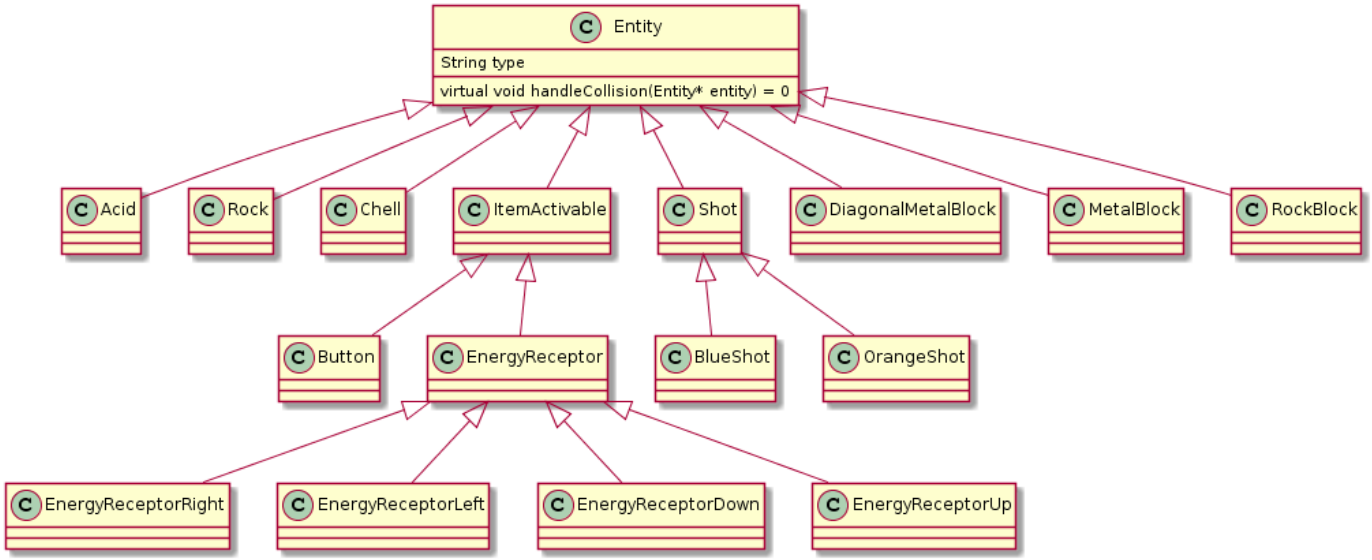


Figure 2: Diagrama de clases de los elementos del juego

#### 4.3.2 Manejo de portales

#### 4.3.3 Manejo del escenario

#### 4.3.4 Soporte de comunicación

Es pertinente aclarar que no se mostró en el diagrama la relación con la clase `Protocol` y la clase `Socket` por claridad.

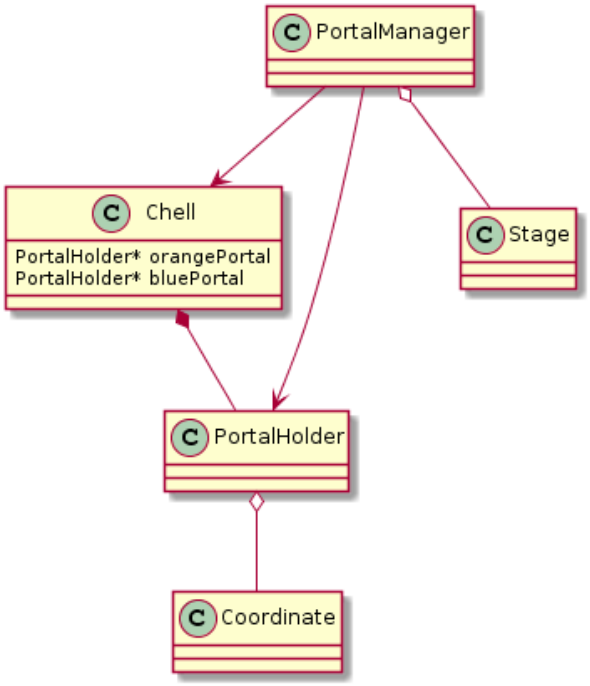


Figure 3: Diagrama de clases de los elementos utilizados para manejar los portales

## 4.4 Descripción de archivos y protocolos

El protocolo implementado consistió en enviar y recibir `strings` por medio de sockets utilizando el protocolo `TCP`. Estos strings provenían de objetos de la clase `nlohmann::json`, implementada por la librería `nlohman` [2] que tienen un método de conversión a strings. Para el envío de strings se envía primero el largo del string y luego el string. Para recibir simplemente se recibe un string.

La idea utilizada consistió en inicialmente mandar tres jsons de estado:

1. Metadata: largo y ancho del escenario:

   ```
   json[id] = {
    {"height", height}, {"width", width}
   };
   ```

2. Data de objetos estáticos (bloques metálicos, bloques en diagonal y bloques de roca):

   ```
   json[id] = {
   {"type", NAME}, {"x", x}, {"y", y}
   };
   ```

   La constante `NAME` varia para cada objeto y fue definida en un archivo de uso compartido `constants.h`.

[2] https://github.com/nlohmann/json

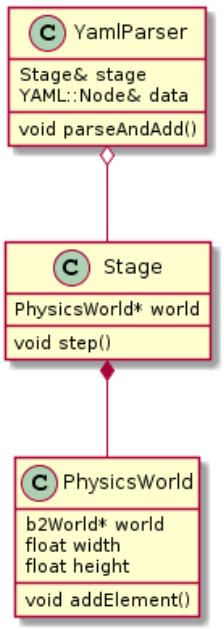


Figure 4: Diagrama de clases de los elementos utilizados para manejar el escenario

3. Data de objetos dinámicos (todo el resto de los objetos). Se trata de los objetos que fueron designados con un id en particular:

   ```
   json[id] = {
   {"state", state}, {type", NAME}, {"x", x}, {"y", y}
   };
   ```

   Nuevamente la constante `NAME` se trata de un `int` definido en el archivo `constants.h` y el estado se designa de acuerdo al tipo de objeto. Hay objetos que no tienen estado pero la gran mayoría sí:

   - Chell: IDLE, JUMPING, MOVING RIGHT, MOVING LEFT
   - Portal: HORIZONTAL o VERTICAL
   - Button: ON o OFF
   - Receptor: ON o OFF
   - Energy Bar: HORIZONTAL o VERTICAL
   - Gate: OPEN o CLOSED

Luego, en cada `step` simulado por `Box2D` se envía al cliente un `json` en forma de `string` como el último descrito, es decir, con la actualización de los objetos dinámicos en el mundo físico.

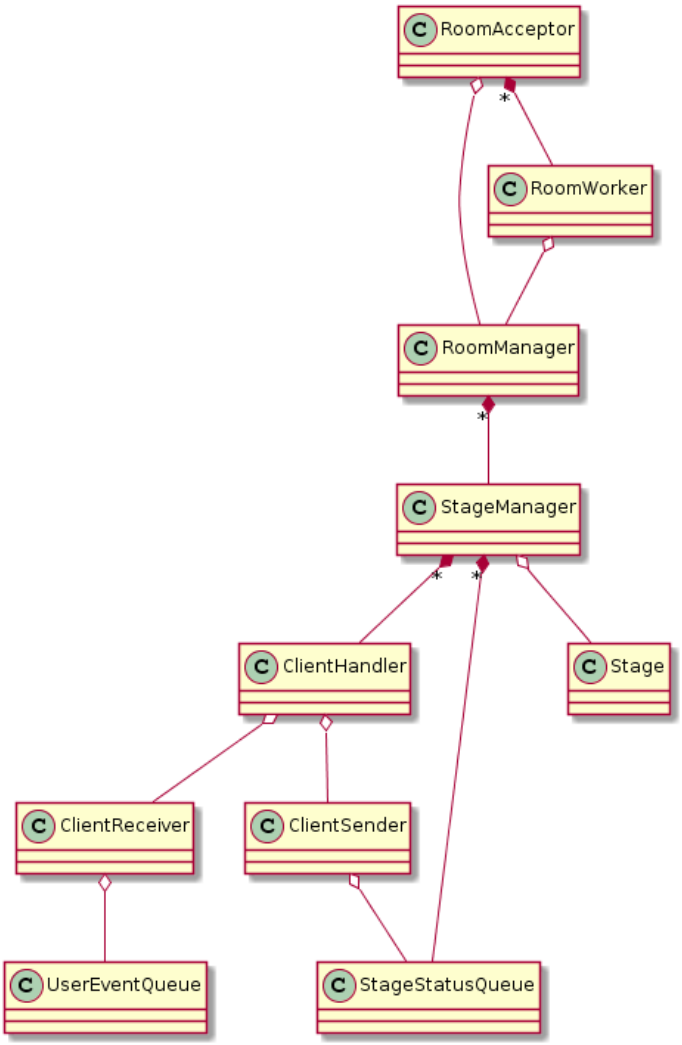


Figure 5: Diagrama de clases de los elementos utilizados para el soporte de comunicación

# 5 Cliente

## 5.1 Descripción general

Las funcionalidades del cliente se dividen en tres partes:

1. Las relativas al motor gráfico, es decir, aquellas que sólo se encargan de representar el modelo del juego visual y auditivamente y distintas pantallas a modo de menu.

2. Las relativas al manejo de sockets para poder comunicarse con un servidor.

A continuación se enumeran las principales clases que se encargan de implementar dichas funcionalidades y sus principales atributos y métodos.

## 5.2 Clases

Para describir las clases utilizadas se dividirán las clases en 3 categorías:

### 5.2.1 Soporte para input de usuario

- **UserEventHandler**

Sus atributos principales son una camara `Camara`, una cola de sonidos `SoundCodeQueue` y una referencia a una cola de eventos de usuario `UserEventQueue`. Se encarga de procesar eventos de SDL y crear eventos propios `UserEvent` para encolar en la cola, así como también encolar sonidos a reproducir cuando corresponda (por ejemplo, al disparar). Estos eventos luego serán consumidos por `EventSender`. Esta clase se corre en un thread propio, para poder procesar eventos de SDL sólo si existe uno, en lugar de hacer polling en el thread principal, lo cual además podría impactar la performance de dicho thread ya que allí se dibuja a pantalla, y es crítico mantener una performance aceptable para una mejor experiencia de usuario.

### 5.2.2 Soporte para vistas y sonidos

- **View**

Sus atributos principales son una ventana `Window`, un factor para pasar de unidades de Box2D a pixeles, y una caja de colisión `SDL_Rect` para revisar colisiones contra la cámara. Esta clase es virtual, es decir, ha de ser implementada por cada una de las vistas que representan entidades del modelo. Cuenta con métodos implementados, los cuales son `move` para actualizar la posición actual de la vista en pantalla, y `checkCollisionWithCamera` el cual es utilizado por las vistas para determinar si deben ser dibujadas o no. El fin de revisar colisión con la cámara es evitar dibujar innecesariamente texturas que el jugador no podrá ver, mejorando la performance del programa.

Los métodos virtuales que han de ser implementados por las clases derivadas son `playAnimation` y `setState`. El primero contendrá la lógica para ejecutar las animaciones de cada vista derivada, y el segundo se utiliza para la lógica de cambio de estado de una entidad del modelo, con el fin de representar con mayor fidelidad a las entidades modeladas en el motor físico.

No se describiran la totalidad de las vistas derivadas, sólo la más compleja que es `ChellView`.

- **ChellView**

Esta clase deriva de View. Se encarga de representar en pantalla a nuestro personaje principal. Entre sus atributos, tenemos un estado actual y un estado previo, esto se utiliza para encolar el sonido del salto en la cola de sonidos. Además cuenta con un timer para saber cuándo encolar un sonido al correr, y otro para saber cuándo dejar de dibujar la animación de muerte. En el método `setState`, se actualiza el estado previo y el estado actual con el estado recibido por parámetro. En el método `playAnimation` se decide qué animación dibujar, dependiendo del estado actual de Chell. Si Chell está corriendo, se chequea el timer para saber si hay que reproducir un sonido en este estado, el chequeo se encarga de actualizar dicho timer. En el caso de la muerte, se reproduce la misma durante una cantidad predeterminada de iteraciones.

- **Camara**

Esta clase consta de un SDL_Rect, el cual determinará qué se muestra en pantalla. La misma se centra sobre el jugador asignado al usuario. Es una clase protegida ya que debió ser utilizada entre dos distintos threads, por lo tanto fue pertinente protegerla para evitar race conditions.

- **ViewManager**

Esta clase se encarga de manejar todas las vistas que requieren una actualización de estado, tienen animación o requieren actualizar su posición en el mundo. Para esto se tiene un mapa donde cada clave es un ID y el valor asociado un puntero a una View. Adicionalmente, cuenta con otro mapa para asociar ID de ChellView con nombre de jugador, para así poder dibujar en pantalla el nombre del jugador sobre su personaje, para poder distinguir entre distintos jugadores. En cuanto a la actualización del modelo, se consumen JSONs desde una cola diseñada para contener el estado del modelo en dicho formato. La idea principal es que ViewManager sólo actualizará y dibujará aquellas Views cuyos IDs se hayan recibido en el JSON que contiene la actualización del modelo del juego. En cuanto a la creación, se recibe inicialmente un JSON con todas las vistas que serán necesarias durante la ejecución del nivel.

- **StageView**

Esta clase se encarga de dibujar todos los sprites estáticos que no cambiarán nunca de posición, como por ejemplo los bloques de metal, roca, diagonal, etc. En cuanto a la creación, se recibe inicialmente un JSON con todos los bloques que serán necesarios durante la ejecución del nivel. El dibujado de los bloques naturalmente posa un problema importante: Si se dibuja todo el nivel en todo momento, podríamos ocasionar un buffer overflow en el buffer que contiene las texturas dibujadas. Para evitar este problema, se pensó el diseño de la clase de forma tal que sólo se dibujen aquellos bloques que colisionarían con la cámara. Para esto se guardan los bloques según sus coordenadas, luego se toma el origen de la cámara y se lo transforma a metros. El paso siguiente es determinar dos rangos para iterar las claves del mapa de bloques, uno en X (se hace con el origen en X de la cámara y el ancho de la misma) y el otro en Y (misma idea pero en Y y con el alto), adicionalmente agregamos algunos bloques extra a modo de borde, para que la transicion entre texturas en la pantalla sea más suave y se obtenga una mejor experiencia de usuario. Finalmente, sólo dibujamos los bloques que existan en nuestro mapa y cuyas claves (coordenadas)estén dentro de este rango.

- **AudioSystem**

Esta clase se encarga de consumir códigos de sonido desde una cola de códigos de sonido, y reproducirlos. Los sonidos se guardan en un mapa donde tenemos como clave un código de sonido, y como valor asociado un `Mix_Chunk*` el cual contiene el sonido a reproducir. Adicionalmente, contamos con un mapa similar pero para música. En la implementación actual del TP no fue utilizado en todo su potencial este feature, pero en próximos releases agregaremos más canciones y la posibilidad de cambiar de música en medio de la partida.

- **Game**

Esta clase se encarga del flujo previo al inicio de una partida, es decir, creación/unión de partida. Cuenta con clases para representar las pantallas de unión/creación de partida, selección de nivel, selección de nombre de jugador.

- **Drawer**

Esta clase contiene a las clases necesarias para poder dibujar el modelo del juego en pantalla, así como también realizar la comunicación con un servidor dado. Aquí encontramos el game loop, en donde consumimos el estado actual del modelo enviado por el servidor, a través de una cola bloqueante. Si no hay un estado nuevo, simplemente utilizamos el estado anterior para dibujar, de esta forma evitamos cortar el dibujado en caso de alta latencia.

#### 5.2.3 Soporte para comunicación

- **Event Sender**

Sus atributos son un protocolo `Protocol` y una referencia a una cola de eventos de usuario `UserEventQueue`. Se encarga de enviar los diversos eventos de usuario al servidor, por medio del protocolo que encapsula el envío a través de sockets.

- **State Status Receiver**

Sus atributos son un protocolo `Protocol` y una referencia a una cola de estados del escenario `StageStatusQueue`. Se encarga de recibir los diversos estados del escenario, enviados por el servidor, para dibujar los elementos dinámicos en su estado actual.

### 5.3 Diagramas UML

#### 5.3.1 Soporte para sonidos

#### 5.3.2 Soporte para vistas

#### 5.3.3 Soporte para comunicación

### 5.4 Descripción de archivos y protocolos

Contamos con un archivo de configuración para setear la resolución de la ventana de juego, y del factor utilizado para convertir unidades de Box2D a pixeles. Es importante que la resolución sea múltiplo de este factor, para así obtener una transición suave entre texturas al mover la cámara.

Como ya se mencionó en la sección del servidor, se utiliza el formato JSON para el envío y recepción de información.

Adicionalmente a los requests ya descriptos en la sección del servidor, contamos con los siguientes requests:

1. Lista de juegos:

```
{ "games":
 ["Aprueben", "este", "tp", "por favor"]
};
```

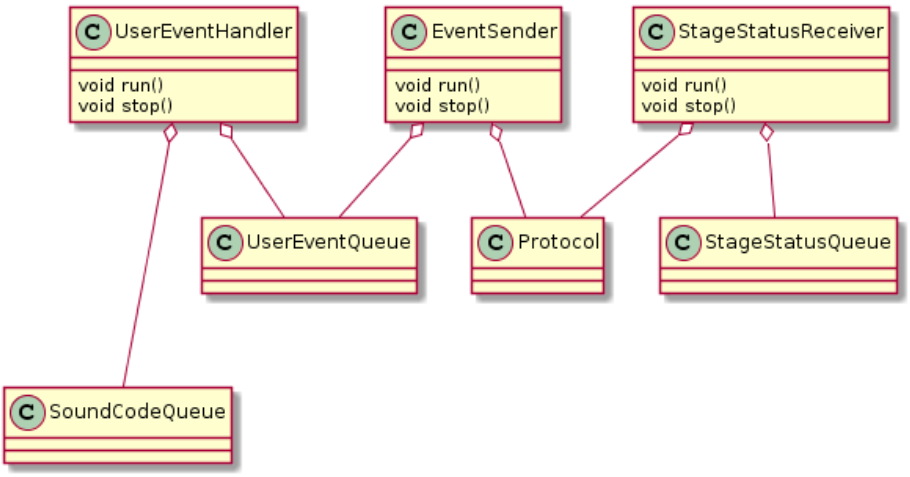


Figure 6: Diagrama de clases del soporte de comunicación del cliente

2. Creación o unión:

```
{ "action": CREATE_GAME_CODE
};
```

   Donde CREATE_GAME_CODE y JOIN_GAME_CODE son enteros, 0 y 1 respectivamente.

3. Campos de partida y ID jugador:

```
{ "game": gameName,
  "id": somePlayerName
};
```

4. Resultado de la acción procesada en el servidor:

```
{ "result": SUCCESS_CODE,
  "idChell": Chell1
};
```

   El campo "result" puede ser SUCCESS_CODE o FAIL_CODE, dependiendo de si salió todo bien o no. "idChell" es el ID asignado por el servidor a la Chell del jugador.

5. UserEvent enviado al servidor:

```
{ "id": userId,
  "eventType": eventTypeCode,
  "x": mousePositionX,
  "y": mousePositionY
};
```

Este request contiene toda la información que necesita el servidor para poder procesar un evento de usuario. X e Y respectivamente se dejan en 0 si el evento no requiere saber estos valores. EventTypeCode es un entero con distintos valores, el listado completo puede observarse en el archivo `common/constants.h`

# 6 Editor

## 6.1 Descripción general

## 6.2 Clases

## 6.3 Diagramas UML

## 6.4 Descripción de archivos y protocolos