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
```

Agregar algo relativo al sh para la compilación

2 Descripción general

El proyecto se constituye de los siguientes módulos:

- Servidor
- Cliente gráfico
- Editor

VER ESTO PORQUE NO SÉ QUÉ PONER

3 Servidor

3.1 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.

3.2 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

3.2.1 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.

3.2.2 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 la gran mayoria de las clases para poder modelizar las colisiones entre objetos. De esta manera, en caso de ocurrir una colisión, siempre sucedía entre dos entidades 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**

Es una clase derivada de `Entity`. Contempla las colisiones con instancias de la clase `Chell` ya que debe matarla.

- **Shot**

Esta clase hereda de la clase `Entity` y 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 angulo 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 hereda de la clase `Entity` y maneja las colisiones con la clase `Chell` ya que debe ganar en caso de colisionar.

- **Chell**
- **Diagonal Metal Block**
- **Metal Block**
- **Rock Block**
- **Energy Ball**
- **Energy Transmitter**
- **Energy Bar**
- **Gate**
- **Item Activable**
- **Button**
- **Energy Receptor**
- **Portal**
- **Rock**