Universidad de Buenos Aires
Facultad de Ingeniería

96.08- Taller de programación 1

1^{er} Cuatrimestre de 2018

Trabajo Práctico Final

Manual de usuario

Grupo 2
Bourbon Navarro, Rodrigo- #96961
Gómez Peter, Federico- #96091
Lidjens, Axel- #95772

Índice

1. Requerimientos de software

El juego fue diseñado para correr bajo un sistema operativo *Debian*, particularmente fue desarrollado en *Ubuntu 16.04*. Como requisitos para poder compilar el cliente, servidor y el editor se encuentran:

- SDL2 versión 2.0.4.
- SDL2 image versión 2.0.1.
- SDL2 ttf versión 2.0.14.
- SDL2 mixer versión 2.0.1.
- Qt-5 versión 5.6.0.

Cabe mencionar que se utilizan las bibliotecas *Box2D* como motor físico del juego y *jbeder-yaml-cpp* para las comunicaciones y la configuración del mismo, que se encuentran ya incluídas.

La compilación e instalación de los programas se hace mediante *CMake* en su versión *3.1.0*, utilizando el compilador *g++* versión *5.4.0*. Como herramienta de *debug* se utilizó *GDB* versión *8.1*.

2. Descripción general

El proyecto se encuentra dividido en tres módulos, correspondientes a las aplicaciones de cliente, servidor y editor.

El cliente es una aplicación gráfica que permite conectarse al servidor indicando la ip del mismo y el puerto. Una vez conectado existen dos opciones, crear una partida, o unirse a una existente. En caso de crear una partida, el servidor envía la información de todos los niveles existentes al cliente (nombre y cantidad de jugadores), y una vez que se selecciona uno, envía el archivo correspondiente al mismo, que está en formato *YAML*, y sus fondos (arhcivos *png*), crea la partida y queda a la espera de que se conecten los jugadores faltantes para inciarla. Si se selecciona la opción de unirse a una partida, el servidor envía al cliente todas las partidas que están disponibles (cantidad de jugadores actual en la partida y cantidad de jugadores necesarios para inciarla), es decir aquellas que no comenzaron aún. Cuando el cliente se une a una partida, el servidor envía el archivo correspondiente al nivel y los fondos del mismo. Cuando se unió el último jugador necesario para iniciar la partida, esta comienza.

El juego transcurre y cuando solo queda un equipo o ninguno (todos perdieron), se muestra en los clientes una pantalla haciendo referencia a si ganaron o perdieron, y cerrando dicha ventana la aplicación termina. Si un cliente se desconecta de la partida, se muestra automáticamente la pantalla indicando que perdió.

Cada vez que una partida finaliza el servidor la remueve.

3. Cliente

Como lo explicado en la sección precedente, la aplicación del cliente inicia mostrando una sucesión de pantallas para conectarse al servidor y crear o unirse a una partida. Cada *input* de teclado o mouse que se detecta por parte de un jugador es procesado en caso de que sea el turno del mismo, no se le haya acabado el tiempo, la partida no haya terminado, y el jugador no haya perdido. La acción que debe realizarse en base al *input* del jugador se decide de acuerdo al estado en el que se encuentre el gusano, el cual responde y dicha respuesta es enviada al servidor. Se utilizó el patrón de diseño *State* para modelar todos los posibles estados del gusano. El servidor es el que tiene la lógica del juego, entonces siempre se envía al cliente el estado en el que se encuentran los gusanos. Cada estado tiene asociada una animación y en ciertos casos también un sonido.

Las texturas utilizadas en las animaciones y los sonidos (tanto los efectos de sonido como la música de fondo), se cargan en memoria una única vez al comienzo de la partida a fin de no comprometer la *performance* de la aplicación, ya que el proceso de animar se realiza permanentemente, y cargar las diferentes texturas una y otra vez no sería eficiente, al igual que en lo que respecta a los sonidos.

Para las diferentes armas que ofrece el juego se realizó otro patrón *State*, de modo que cada vez que hay un disparo, cada arma sabe cómo responder.

3.1. Clases

- ***Animation***: se construye a partir de una textura y se encarga de renderizar la misma de acuerdo a un *framrate* de 25 *frames* or segundo en la posición que se le indica. Puede renderizarse en *loop* desde el *frame* incial hasta el último y a continuación el primero nuevamente, en *loop* llegando al final y volviendo al comienzo (indicado con el *flag playReversed*), animarse una única vez quedando en el último *frame* (indicado con el *flag playOnce*), animarse en sentido inverso (indicado con el *flag playInverse* y utilizado para la teletransportación) o puede setearse un *frame* manualmente (para el caso en el que se esté apuntando un arma por ejemplo). La actualización del *frame* que debe renderizarse se hace mediante el método *update*, el cual recibe el tiempo que ha pasado desde el último cambio. Cuando este tiempo acumulado supera el *framrate* se realiza el cambio de *frame*.
  Los métodos principales son el ya mencionado *update*, el método *render*, que recibe la posición en donde debe renderizarse, el modo de *flip*, y la cámara del juego que es la que calcula las coordenadas y la muestra en pantalla. Finalmente el método *advanceFrame* es el que se encarga, en base a los *flags* que se encuentren seteados, de establecer el siguiente *frame* que debe ser animado.

- ***Camera***: se construye en base a la ventana donde se renderizará, la relación pixels/metro deseada y el ancho y alto del área a donde la cámara puede ir. Se destacan los métodos *isMoving* para saber si la cámara está en movimiento a la hora de decidir qué renderizar, *update* que se encarga de actualizar la posición de la cámara, y luego los métodos *draw* y *drawLocal* que dibujan en pantalla una textura o un rectángulo que reciben por parámetro.

- ***Texture***: encapsula la creación y liberación de recursos correspondiente a una textura. Únicamente devuelve el puntero correspondiente a la textura, su alto o su ancho.

- ***Font***: encapsula la creación y liberación de recursos correspondiente a una fuente de texto. Solo devuelve un puntero correspondiente a la fuente.

- ***Text***: se construye a partir de una fuente. Se le setea el texto deseado e internamente crea una textura con el mismo, la cual es renderizada con o sin fondo. El método *render* es el encargado de dibujar el texto en pantalla de acuerdo a una posición y la cámara recibidos por parámetro.

- ***WrapTexture***: se crea a partir de una textura, un ancho y un alto. Solapa la misma textura o la recorta en base a las dimensiones de la misma y al ancho y al alto especificados. Se destaca el método *render*, que dibuja la textura en pantalla en la posición deseada y el método *render* que hace lo propio pero especificando también un ángulo.

- ***Button***: representa un botón y se crea a partid de una posición, un ancho y un alto. Puede seteársele un mensaje y el color de texto y de fondo. Se destaca el método *inside* que establece a partir de la posición en la que el usuario hizo click recibida por parámetro si esta se encuentra dentro del botón, y *render*, que a partir de una *cámara* recibida por parámetro lo dibuja en pantalla.

- ***GameWindow***: es una interfaz para todas las ventanas que posee la aplicación. Se construye a partir de la ventana del juego, una fuente de texto y una cámara. Define una estructura *TextField* que procesa los *inputs* de texto del usuario, y tiene un vector de botones *Button*. Posee los métodos *handleKeyDown* (responde a *inputs* del usuario), *appendCharacter* (responde a *inputs* de texto del usuario), *buttonPressed* (responde en el caso que un usuario presione un botón), y *render* (dibuja todo lo que deba dibujarse en la ventana). Las clases que implementan esta interfaz son:

  - ***ConnectionWindow***: tiene dos *TextField* donde permite al usuario ingresar la *ip* y el puerto del servidor al que desea conectarse y un botón que al presionarlo crea la conexión.
  - ***SelecActionWindow***: tiene dos botones con los cuales permite al usuario elegir entre crear una partida o unirse a una existente.
  - ***CreateGameWindow***: permite crear una partida. Muestra en pantalla un nivel con su nombre y cantidad de jugadores y tiene tres botones, dos para alternar entre los niveles disponibles (anterior y siguiente), y otro para seleccionar el nivel.
  - ***JoinGameWindow***: permite unirse a una partida. Muestra en pantalla una partida con la cantidad de jugadores que hay actualmente en dicha partida y la cantidad de jugadores que debe haber para comenzar. Tiene tres botones, dos para alternar entre las partidas disponibles (anterior y siguiente), y otro para seleccionar la partida.

- ***WaitingPlayersWindow***: una vez que se seleccionó el nivel a crear o se eligió la partida a unirse, se muestra una pantalla con la cantidad actual de jugadores conectados y la cantidad necesaria para que comience el juego. Cuando esta cantidad es alcanzada, el juego comienza automáticamente.
- ***GameEndWindow***: al finalizar el juego o el jugador desconectarse, se muestra en pantalla un mensaje haciendo alusión a si ganó o perdió.

- ***WormState***: representa el estado del gusano. Todos los *inputs* del jugador están representados en métodos (*moveLeft*, *moveRight*, *jump*, *bazooka*, *startShot*, etc.), y cada estado sabrá responder en consecuencia. Se destaca el método que devuelve el *id* del estado Las clases que implementan esta interfaz son:
  - ***WormWalk***.
  - ***WormStill***.
  - ***WormStartJump***.
  - ***WormJump***.
  - ***WormEndJump***.
  - ***BackFlip***.
  - ***WormBackFlipping***.
  - ***WormEndBackFlip***.
  - ***Hit***.
  - ***Die***.
  - ***Dead***.
  - ***Drowning***.
  - ***Falling***.
  - ***Land***.
  - ***Sliding***.
  - ***Teleporting***.
  - ***Teleported***.
  - ***Batting***.
- ***TextureManager***: es un template que permite guardar texturas en un *unordered map* con un *hash* redefinido.
- ***SoundEffectManager***: idéntico funcionamiento que el *TextureManager* salvo que ahora se almacenan efectos de sonido en vez de texturas, y ya no es necesario el renderizador.
- ***BackgroundMusicManager***: idéntico funcionamiento que el *SoundEffectManager* salvo que ahora se almacenan archivos de música de fondo en vez de efectos de sonido.
- ***Armory***: se construye a partir de un *gameTextureManager*, que es un *TextureManager* cuyo *id* es de tipo *GameTextures* (clase *enum* con los *ids* de cada textura utilizada).

4. Servidor

Continuando con la descripción de los módulos, se verá ahora en detalle el servidor. Este comienza creando *GameLobby*, que se e encargará de aceptar conexiones que se realicen con el servidor, creando un *GameLobbyAssistant* para cada uno. Esta clase recibirá los comandos que realice el cliente luego de haber tenido una conexión exitosa. Las opciones que puede realizar son crear una partida, obtener los niveles disponibles, ingresar a una partida creada y obtener una lista de partidas creadas. Todo esto sucede en *threads* separados. Tanto el GameLobby como cada GameLobbyAssistant realizan sus tareas en hilos separados, el primero para poder aceptar clientes y dejar el hilo principal para recibir el comando por *stdin* necesario para comenzar el proceso de cerrado ordenado del servidor, y los segundos para que el primero pueda aceptar sin rechazar conexiones durante el lapso que el cliente tarda en comenzar una partida.

Cuando un cliente decide crear una partida, se creará una nueva instancia de la clase *Lobby*, por medio del uso de

la clase *Lobbies*. La clase Lobbies es una clase de importancia ya que es el *recurso compartido* que relaciona todas las conexiones que se realicen al servidor. En esta se guardan todas las partidas creadas. Dado que varios clientes distintos podrían querer conectarse a la misma sala de juego, esta también posee una *race condition* que debe ser tenida en cuenta. Lobbies opera como un *monitor*, que realiza las operaciones de crear partida, unirse a una partida y obtener los juegos creados de forma atómica. Para esto, dispone de un mutex de protección. Lobbies posee internamente un arreglo de *Lobby*, que tiene un registro de los clientes. Cuando la sala se completa, notifica al GameLobby que la partida puede comenzar. Este inmediatamente dota al Lobby de los sockets de cada cliente, para que este pueda iniciar en un hilo propio la partida. Es en este momento también que sucede la finalización de los GameLobbyAssistant involucrados. La liberación de los recursos de estas instancias (su destrucción), la realiza el GameLobby, quien revisa luego de aceptar una conexión todos los hilos que terminaron, aplicando su correspondiente *join* y su destrucción.

La partida transcurre en la clase *Game*. Esta fue pensada en un principio como una clase que iba a heredar de Thread, sin embargo, se delegó esa herencia en el Lobby que lo contiene. En esta clase se creará el mundo físico y se recibirán las interacciones que tenga el usuario con el cliente, para modificar este mundo en la medida de lo posible.

Hablar de los input/output workers acá.

hola

5. Editor