FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Taller de Programación I TP Final Megaman

Facultad de Ingeniería de la U.B.A.

3 de julio de 2016

Índice

1. Manual de Proyecto

1.1. Integrantes

Aguilera Santiago, 95795.

Lazzari Santiago, 96735.

Stancanelli Martín, 95188.

1.2. Enunciado

La dinámica del juego propuesta para el Megaman es la siguiente:

No existen puntos ni hay tiempo para completar los niveles. El juego consta de partidas en las que colaboran hasta 4 jugadores con el objetivo de derrotar a los 5 robots para restaurar el orden. Los distintos jugadores se registran en el servidor, reciben número y con ello un Megaman para iniciar juntos la partida. Al ingresar el primer jugador se define una nueva partida donde los siguientes jugadores se unirán automáticamente previo ingreso de su nombre. Todos los jugadores que se unan a la partida recibirán notificaciones sobre los compañeros que ingresen pero sólo el primer jugador podrá iniciar el juego y elegir el próximo nivel. El límite de 4 jugadores será validado al iniciar la partida ya que, una vez iniciado el juego, no se admitirán nuevos jugadores. Cada Megaman recibirá un total de 3 vidas al inicio que podrá aumentar al encontrar el powerup correspondiente. Cuando un jugador muere, los otros participantes continúan jugando en el nivel hasta finalizarlo o morir. Si todos los jugadores mueren, el nivel vuelve a iniciar salvo que hayan llegado a la cámara del jefe. En ese caso, todos los jugadores vuelven a dicha cámara para continuar desde allí. Cuando las vidas de un jugador se acaban, no es regenerado en el próximo inicio de nivel y, en caso de que no quede ningún jugador con vidas, se declara el fin del juego reiniciando el ciclo de espera de participantes. Todo Megaman posee una barra de energía que disminuye al ser dañado por un enemigo. Cuando se agota la energía, Megaman muere perdiendo una vida.

1.3. División de tareas

Santiago Lazzari se encargó de la edición de los mapas, esto incluye lo relacionado a la construcción de los mismos, tanto para los niveles como para los jefes de cada nivel.

Santiago Aguilera se encargó del cliente. Esto incluye tanto la parte del front end, como la vista inicial para conectarse, el lobby donde los jugadores se irán conectando y pueden elegir un nivel a jugar y el juego en particular, como también se encargo de la parte de backend, es decir toda la lógica detrás de las vistas y pantallas, el envió y recepción de datos con el servidor, entre otros.

Martín Stancanelli se encargó del servidor. Esto implica el armado del modelo de la lógica del juego (monstruos, armas, bosses, powerups), el sistema de físicas, el logueo de eventos y el sistema multijugador, que permite la comunicación y el armado de partidas entre varios clientes.

Una vez que el grueso del juego se encontraba listo (o a medida que cada uno iba terminando sus tareas designadas), los tres nos fuimos encargando de la corrección de errores en las tres plataformas de forma indistinta.

1.4. Evolución del proyecto

El cronograma propuesto por la cátedra si intento seguir lo mejor posible, teniendo en cuenta imponderables que surgieron durante la realización del trabajo practico y el hecho de que los 3 participantes del grupo trabajan fuera del horario de la cursada.

El trabajo fue constante y los tiempos estimados que no se pudieron dedicar por inconvenientes que les surgieron a los integrantes fueron repartidos entre el resto.

1.5. Inconvenientes encontrados

1.5.1. General

- Instalación de GTKmm: Nos tomo muchísimo trabajo instalar la ultima versión de GTKmm. No tuvo solución, ya que siempre eran mas y mas dependencias y terminábamos rompiendo el OS. Utilizamos Ubuntu GNOME el cual tiene GTKmm 3.18, y mucho mas adelante nos dimos cuenta que no necesitábamos una versión tan alta, así que downgradeamos las clases que usábamos a una versión que tuviesen tanto Ubuntu 14.04 como Linux Mint 17.3 (GTKmm 3.10)

- Envió/Recepción de datos: Inicialmente el cliente/servidor fueron hechos con un único thread para el envió/recepción de datos, lo cual creaba un problema ya que al ser el receive() bloqueante, a veces no se podía enviar datos o generaba una starvation sobre el thread. Lo resolvimos teniendo un thread para cada operación necesaria

- Movimientos del usuario: Un problema encontrado fue el envió de los movimientos de un usuario al servidor. Después de un debate y analizar los pros y contras de cada modelo, terminamos decidiendo por tener un mapeo de los botones posibles y enviar siempre el modelo como un char array (de flags dependiendo de si el botón esta presionado o no)

- Transformación de coordenadas: Como el mundo de Box2d utiliza coordenadas cartesianas y tanto el cliente como el editor trabajan con coordenadas de píxel tuvimos algunos problemas para estandarizar la conversión de coordenadas. Finalmente se decidió el uso de coordenadas píxel para el manejo de mensajes y su transformación interna a metros dentro del servidor.

1.5.2. Editor

De los problemas encontrados en el editor, el primer inconveniente fue la instalación de las librerías correspondientes a **Gtk** para poder crear una aplicación visible en pantalla. Otros inconvenientes surgieron de **Gtk** y las clases correspondientes a las mismas, es decir el uso de la librería de manera correcta requirió mucha prueba y error hasta que logró funcionar.

1.5.3. Cliente

- Manejo de datos entre muchos threads: En el cliente inicialmente, como GTKMM ya es bloqueante, no encontrábamos forma de poder pasarnos datos entre threads (ya que para realizar una "programación orientada a eventos"necesitaríamos estar loopeando sobre una queue de eventos). Logramos solucionarlo mediante una clase de Glib, la cual te permite attachear un método y emitir señales para que ese método se ejecute sobre el hilo principal. De esta forma un thread puede (mediante una interfaz) ejecutar un método, el cual se comunica con el dispatcher y el mismo hace que el método attacheado se ejecute en el hilo principal
- SDL / Cairo: Inicialmente no se sabia con cual de las dos librerías renderizar el juego. Nos terminamos decidiendo por SDL debido a su extensa documentación.
- SDL2pp objetos en el stack: Como SDL2pp (Wrapper de SDL2 para c++) no tiene constructores vacíos para sus clases, no se podían crear objetos de SDL2 en el stack. La solución fue tener que instanciar todo en el heap.
- Pantalla persiguiendo a los usuarios: Un problema que tuvimos fue lograr centrar el mapa en los usuarios, ya que el mismo dependía del tamaño de cada pantalla y de las posiciones de los usuarios. Esto se logro hacer mediante un centro de masa entre todo.

1.5.4. Servidor

- Testbed de Box2d: Para poder probar la librería Box2d que se encarga del manejo de las físicas del juego, esta incorpora un Testbed, que permite instanciar distintos objetos fisicos y probar sus características. Sin embargo la ultima version de Box2d que se encuentra en el repositorio del autor no esta preparada para compilar en Linux y se generaban errores de dependencias faltantes al querer usarla. Finalmente se decidio usar el Testbed de una versión mas antigua de Box2d, aunque esto nos limito la posibilidad de probar funcionalidades que utilicen características mas recietes.
- Alteración del mundo de Box2d: Durante la realización del trabajo nos encontramos con la dificultad de que Box2d no permite que el mundo sea modificado mientras esta procesando los datos. Esto nos obligo a generar listas que contengan objetos marcados como a crear o a destruir y ejecutar acciones correspondientes sobre ellas una vez que el step de Box2d había terminado.

1.6. Análisis de puntos pendientes

- El juego no contiene sonidos. Se encuentra desarrollado un controlador para los mismos (y a su vez todas las vistas reproducen en el momento correcto el sonido adecuado, por ejemplo cuando megaman pierde una vida se escucha el sonido de que fue dañado) pero debido a problemas con SDL Mixer fue comentado del proyecto. Quedaría como un feature a realizar.
- Las balas provocan que los personajes se desplacen en el sentido contrario al cual fueron golpeados. Si bien esto agrega cierto realismo a la físicas, puede provocar que la posición de los objetos a veces no sea realmente la que se ve en pantalla. Una solución posible seria emitir un mensaje cada vez que un objeto es golpeado por un proyectil con su nueva posición, pero esto masificaría el envió de mensajes, degradando la experiencia del usuario. Queda pendiente una resolución adecuada a este problema.

1.7. Herramientas

1. SDL2
2. SDL2pp
3. Glog
4. Rapidjson
5. GTK
6. GTKmm
7. Glib
8. Sigc++
9. X11
10. Box2d
11. Glade
12. Pngquant
13. Cmake
14. Makefile
15. Doxygen
16. GNU G++
17. Git

18. Gimp
19. Photoshop
20. Callgrind

1.8. Conclusiones

El trabajo se presento como un desafió mas que importante. Nos ocupo la mayor parte de nuestro tiempo libre y nos obligo a juntarnos durante varios días seguidos para poder llegar a tiempo con la entrega.

Sin embargo, también implico mucha investigación y utilización de librerías nuevas, lo cual nos aporto mucho conocimiento sobre utilidades que antes desconocíamos.

Como punto a mejorar quizá sea conveniente migrar a C++11, ya que varias de las librerías que se encuentran hoy vigentes utilizan ese estándar y fue necesario compilar con el flag de ese estándar activado para poder sacarles mayor provecho.

2. Documentación Técnica

2.1. Requerimientos de software

La computadora a correr necesita contar con los siguientes requisitos:

1. OS: Linux
2. Gtkmm: Versión 3.10.1 o mayor
3. Pantalla mayor a 700px en ambos laterales (Aunque desde el cliente se puede jugar con el tamaño del juego)
4. Conexión a internet
5. Tarball del juego

2.2. Descripción general

El juego consta de 3 aplicaciones.

- Cliente: Interfaz visual con la cual el usuario interactuara para jugar.

- Servidor: Aplicación corriendo en background la cual permitirá el juego en modo online y realizara el trabajo pesado del juego para no sobrecargar los clientes y para que sea mas seguro (Ninguno de los usuarios pueda realizar exploits).
- Editor: Interfaz visual desde la cual se podrán crear mapas customizables por el usuario.

Cada una de estas aplicaciones corre por separado y es independiente de las otras, sin embargo esto no quiere decir que no sean dependientes (ya que el cliente depende de un servidor, mientras que el servidor depende tanto del cliente como del editor).

A continuación entraremos mas en detalle para cada modulo

2.3. Módulo Cliente

2.3.1. Descripción general

El cliente es la aplicación con la cual el usuario va a interactuar a la hora de jugar. El mismo consta de 3 pantallas.

Inicialmente hay una pantalla de menú, en la cual un usuario se conecta al servidor con un nombre/tag para ser identificado por el resto de los jugadores.

Si el usuario logra conectarse al servidor de forma satisfactoria, ira a la pantalla de Lobby, donde se encuentran los otros jugadores y pueden elegir que mapa comenzar a jugar. Solo el administrador de la partida, el cual se identifica como el primer jugador en conectarse al servidor (o bien crear la sala) es el que puede tomar decisiones sobre que mapa jugar.

Cuando el admin elige un mapa, comenzara una partida. En la misma el usuario podrá ver a los otros usuarios junto a el y juntos deberán completar el nivel, el cual se completa si al menos uno de los jugadores logra derrotar al jefe final del nivel. Cabe destacar, que cada Megaman (o usuario) puede además de moverse, disparar para derrotar a los enemigos con varias armas o bien saltar y esquivar los obstáculos que se presenten. Megaman cuenta por default con un arma básica, la cual tiene munición infinita, pero además de ello, por cada nivel que vayan ganando, se les desbloqueara a todos los usuarios una nueva arma, especifica del nivel ganado, la cual si bien tiene munición limitada, puede presentar grandes ventajas para terminar los próximos niveles.

Sin embargo si todos los usuarios vacían sus tanques de vida 3 veces, perderán el nivel y no podrán continuar jugando, es decir, perderán el juego. Si completan el nivel, por mas que hayan perdido vidas o no tengan el tanque completo, se les regenerara de nuevo.

2.3.2. Clases

La estructura general se divide en:

1. Client: Clase principal desde donde se manejan las vistas/controladores/conexiones
2. Concurrent: Clases que tienen relación directa con multithreading.
    - Event: Clase abstracta para los eventos. Cuando se desea enviar datos de un thread a otro, los mismos se deben wrappear en un evento y sera lo que se envía/recibe.
    - Looper: Clase para que los threads manejen los eventos. La misma permite agregar/quitar/acceder a los eventos de forma segura desde cualquier thread. De esta forma un thread puede acolar eventos mientras que otra los recibe. Por default, a demás de poder crear instancias de loopers, ya hay una singleton que es la correspondiente al main thread (Esto significa que cualquiera puede postear eventos al main thread).
    - Garbage Collector: Para no tener que estar limpiando eventos por todos lados (Ya que no puede ser RAII, porque podría darse el caso de que alguien descole el evento pero lo tenga que usar luego), el looper internamente tiene un Garbage Collector el cual va limpiando eventos a medida que se llega a un limite.
3. Serializer: Clases que serializan modelos/entidades en algún formato. En nuestro caso todos los serializers serán para enviar datos al servidor respetando el protocolo cliente/servidor.
    - Change Weapon Serializer: Serializa el cambio de arma para notificar al servidor que arma se acaba de seleccionar
    - Key Map Serializer: Serializa el mapeo de las teclas seleccionadas para que el servidor pueda reaccionar a los inputs del usuario.
    - Player Connected Serializer: Serializa cuando un usuario se acaba de conectar y le envía los datos del mismo al servidor
    - Start Map Serializer: Serializa el mapa que el usuario selecciono para comenzar a jugar.
4. Controller: Tiene todos los controladores que se presentan en el cliente.
    - Key Map: Esta clase debería ser un modelo, se encuentra ubicada incorrectamente. La misma simplemente tiene un modelo de los inputs del usuario.
    - Main Screen Controller: Controlador de la vista para la pantalla principal. La misma responde a los cambios para cuando el usuario quiere conectarse y para cuando la conexión ya fue completada.
    - Lobby Controller: Controlador de la vista para la pantalla del lobby. La misma presenta una lista donde se muestran las conexiones/desconexiones de nuevos usuarios y además maneja la selección de un mapa nuevo a jugar.
    - Game Controller: Controlador de la vista para la pantalla del juego. La misma se encarga de recibir los inputs del usuario y mandar/recibir eventos y reaccionar frente a ellos.

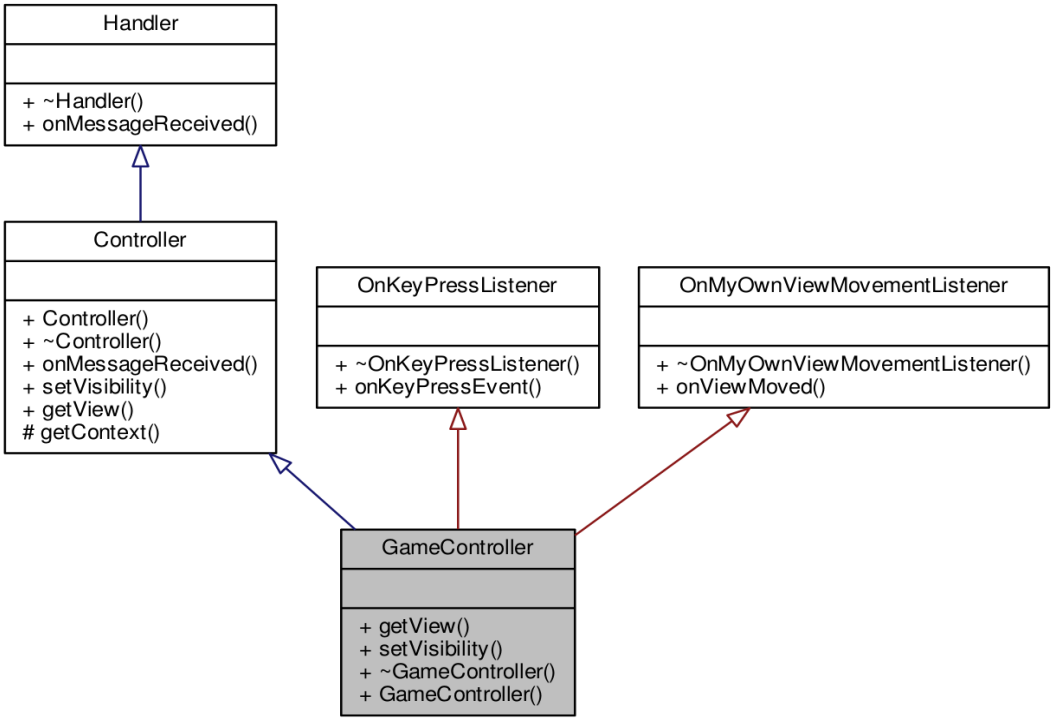

Figura 1: Diagrama de lases del controlador del juego

- Sound Controller: Controlador para el manejo de sonidos. El mismo no se esta usando actualmente.
- Concurrent: Contiene los threads para conectarse/enviar/recibir datos del servidor.
  - Receiver Contract: Interfaz que deben implementar los que quieran escuchar la recepción de datos del servidor
  - Connection Thread: Clase que realiza la conexión al servidor.
  - Receiver Thread: Clase que se encarga de la recepción de datos del servidor, la creación del evento correspondiente a los datos recibidos y el envió de los mismos.
  - Sender Thread: Clase que se encarga del envió de datos al servidor. El mismo tiene asociado un looper del cual va enviando los eventos que tenga acolados.

5. Event: Contiene todos los eventos posibles dentro del cliente

- Ammo Change: Evento para notificar el cambio de munición.
- Connection: Evento para notificar el estado de una conexión al servidor.
- Create Connection: Evento para crear una conexión con el servidor
- Disconnected Player: Evento para notificar que un usuario se desconecto de la instancia de juego.

- Gauge Change: Evento abstracto para notificar que algún contenedor de vida/munición cambio
- Hp Change: Evento para notificar que la cantidad de vida cambio (Esto es dentro de una misma instancia de vida)
- Life Change: Evento para notificar que la cantidad de vidas cambio

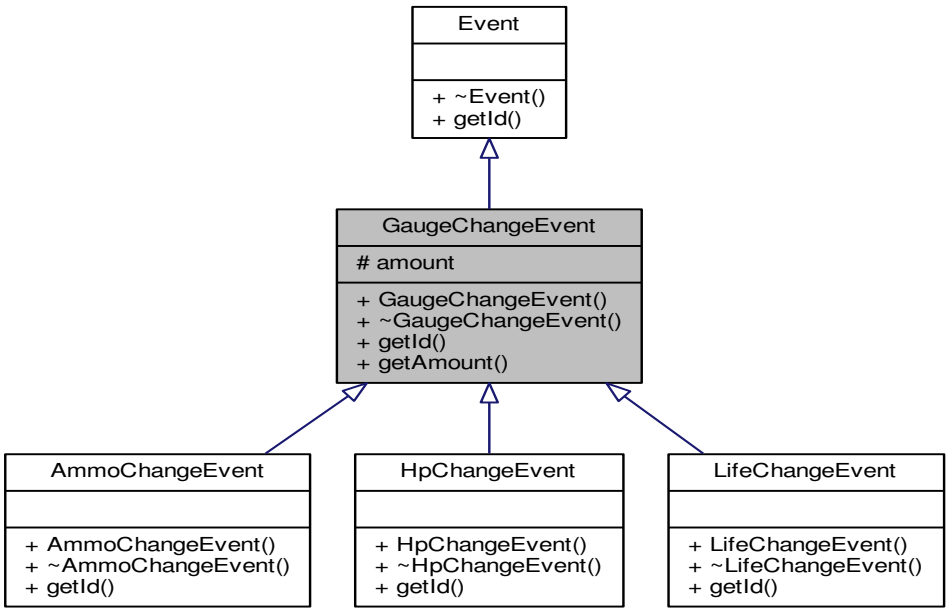

Figura 2: Diagrama de clase de los eventos para los cambios de vida/munición

- New Player: Evento para notificar que un usuario nuevo se conecto a la instancia de juego
- Quit: Evento para notificar que el cliente termino
- Received Map: Evento para notificar que se acaba de recibir un mapa de juego
- Start Map: Evento para notificar el comienzo de un mapa de juego
- Send Change Weapon: Evento para enviar el cambio de arma a usar por un usuario al servidor
- Send Key Map: Evento para enviar el nuevo mapeo de inputs del usuario al servidor
- User Has Defined Id: Evento para que nuestra vista del juego sepa que objeto bindear a nuestro usuario (que Megaman le corresponde)

6. View: Contiene las vistas del cliente

- Main Screen: Vista de la pantalla inicial. Donde el usuario ingresa sus datos y se conecta a un servidor

- Lobby: Vista de la pantalla del lobby
- Game: Vista de la pantalla del juego
- Game Engine: Contiene todas las vistas del juego en particular. Ya sea de personajes como del mundo.
    - Animated Factory View: Creador de vistas animadas (Vistas con comportamiento/no estáticas)
    - Animated View: Clase abstracta de una vista animada
    - Big Ammo View: Clase estática para la vista de un paquete de munición grande
    - Big Enery View: Clase estática para la vista de un boost de energía grande
    - Bumpy View: Clase animada para la vista del personaje Bumpy
    - Default Bar View: Clase estática para la vista de una barra de hitpoints/munición
    - Fireman View: Clase animada para la vista del personaje Fireman
    - Jumping Sniper View: Clase animada para la vista del personaje Jumping Sniper
    - Life Bar View: Clase estática para las vidas de Megaman actuales
    - Life View: Clase estática para una vida extra
    - Magnetman View: Clase animada para la vista del personaje Magnetman
    - Megaman View: Clase animada para la vista del personaje Megaman
    - Met View: Clase animada para la vista del personaje Met
    - Projectile View: Clase animada para la vista de un proyectil
    - Rendered View: Clase abstracta global para cualquier clase que vaya a ser renderizada (tenga vista)
    - Ringman View: Clase animada para la vista del personaje Ringman
    - Small Ammo View: Clase estática para la vista de un paquete de munición chico
    - Small Energy View: Clase estática para la vista de un paquete de energía chico
    - Sniper View: Clase animada para la vista del personaje Sniper
    - Sparkman View: Clase animada para la vista del personaje Sparkman
    - World View: Clase estática para la vista del terreno

7. Commons: Clases comunes para las demás aplicaciones

- Context: Clase padre para los que tienen un contexto. Permite el uso de diversos métodos que clases derivadas pueden llegar a requerir
- Controller: Clase padre para los controladores
- Handler: Clase padre relacionada con el multithreading (para la recepción de mensajes)

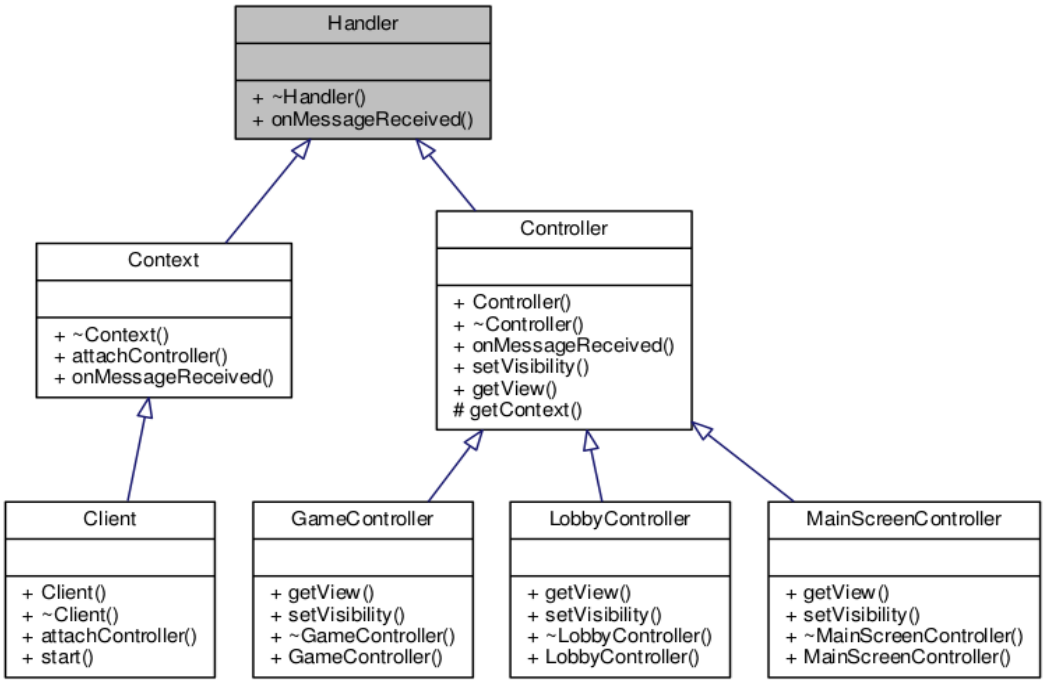

Figura 3: Estructura general del cliente

- Concurrent List: Wrapper de una lista protegida contra multithreading
- Lock: Wrapper de un mutex con RAII
- Mutex: Wrapper del Mutex de C
- Map Constants: Contenedor de cosas útiles del mapa
- Map View: Modelo de la vista del mapa
- Map View Parser: Parseador del modelo Map View
- Message Protocol: Contenedor del protocolo de mensajes cliente/servidor
- Obstacle View: Vista de un obstáculo
- Point: Punto (x,y)
- Serializer: Clase padre para un serializador de datos
- Socket: Clase wrapper con el comportamiento de un Socket
- Thread: Clase wrapper para un thread de C

2.3.3. Descripción de archivos y protocolos

- Como imágenes para las visuales del juego se uso como formato .PNG de 8 bits (256 colores) - Como formato de envió de datos con el servidor se uso Javascript Object Notation (JSON) - Como formato de sonido se utilizo .WAV, sin embargo por problemas con SDLMIXER, comentamos el feature y lo dejamos armado y listo pero sin utilizarse.

Los protocolos fueron definidos previamente con el servidor. Al final de la sección serán listados. Para operar con los mismos se utilizo una programación orientada a eventos, en la cual cada evento se mapeaba con el id de un protocolo y de esta forma se sabia responder de forma correcta a cada entrada.

El protocolo utilizado fue el siguiente:

1. PLAYER CONNECTED

   CODE - LENGTH - NAME

2. NEW PLAYER

   CODE - LENGTH - NAME

3. START GAME

   Cliente a servidor: CODE - LENGTH - MAPID

   Servidor a cliente: CODE - LENGTH - JSONMAP

4. SEND MAP

5. KEY PRESSED

   CODE - LENGTH - CHARARRAY WITH KEY PRESSES (0 FALSE 1 TRUE)

6. UPDATE MOVEMENTS

   CODE - LENGTH - { id: uint, type: TYPE, position: { x: uint, y: uint } }

7. OBJECT CREATED

   CODE - LENGTH - { id: uint, type: TYPE, position: { x: uint, y: uint } }

8. OBJECT DESTROYED

   CODE - LENGTH - { id: uint }

9. HP CHANGE

   CODE - LENGTH - { hp: int }

10. AMMO CHANGE

    CODE - LENGTH - { ammo: int, special: bool }

11. WEAPON CHANGE

    CODE - LENGTH - WEAPONID

12. END GAME

    CODE - LENGTH (ZERO)

13. LIFE CHANGE

    CODE - LENGTH - { life: uint }

14. DISCONNECTED PLAYER

    CODE - LENGTH - NAME

15. ENTERED BOSS CHAMBER

    CODE - LENGTH - MAPJSON

16. CONNECTED PLAYER ID

    CODE - LENGTH - { your id: uint }

2.4. Editor

2.4.1. Descripción general

El **Editor** es una aplicación que permite a un administrador editar los mapas de los distintos niveles del Megaman incluyendo los distintos mapas de los jefes de cada nivel.

En términos generales el editor esta dividido en dos pantallas, la pantalla principal y la pantalla del mapa.

La pantalla principal es donde se muestran los distintos niveles accesibles para editar. Ésta pantalla comienza con la navegación de la aplicación, luego seleccionando algún nivel se presenta la pantalla del mapa.

La pantalla del mapa muestra el mapa correspondiente al nivel que seleccionó previamente y le deja al usuario editarlo borrando y agregando elementos al mismo. Ésta te da la posibilidad de borrar los cambios realizados por el usuario o almacenarlos. De almacenarlos no se podrá volver al estado original del mapa.

2.4.2. Clases

1. Editor : Corre el controlador para que comience la ejecución del mismo.
2. Modelos : Contiene entidades especificas del editor
    - ObstacleView : Encapsula un obstáculo de cualquier tipo (bloque, spawners, powerup, etc), con una posición en el mapa.
    - ObstacleViewContainer : Contiene un obstacle view asociado a una imagen correspondiente al tipo de ese obstacle, luego ésta imagen sera la presentada en el **MapFixedView**.
    - DialogManager : Maneja los popups que aparecen cuando se requiere hacer un doble chequeo al usuario por determinadas acciones.
    - MapView : Encapsula la información de los mapas, es decir tiene una lista de **ObstacleView**, ancho, alto etc.

- MapViewConstants : Tiene la meta información relacionada al mapa, en el .h contiene macros con distintos números comunes a toda la aplicación, como por ejemplo el ancho de los bloques, también un enumberable con los distintos tipos de objetos de mapa. La clase en si tiene un map que, dado un tipo de objeto de mapa, devuelve un string que referencia a la imagen de dicho objeto.
- MapViewJsonWriter : Encargado de escribir el JSON del mapa para poder ser almacenado.
- MapViewParser : Se encarga de leer un archivo de json y lo transforma en un **MapView**

3. Vistas : Contiene entidades que representan vistas en el editor.

   - MapWindow : Es la **Gtk::Window** que se encarga de mostrar el mapa y la vista para la edición del mismo, éste se encarga de manejar los eventos y entonces poder realizar el drag and drop.

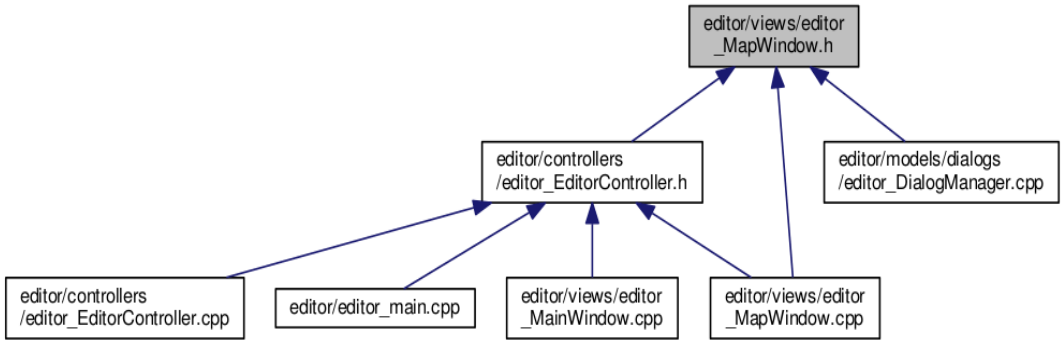

Figura 4: Diagrama de clases de MainWindow

- MainWindow : Contiene los botones de los niveles y captura las señales que envían los mismos para pasarle al controlador cual nivel fue seleccionado.

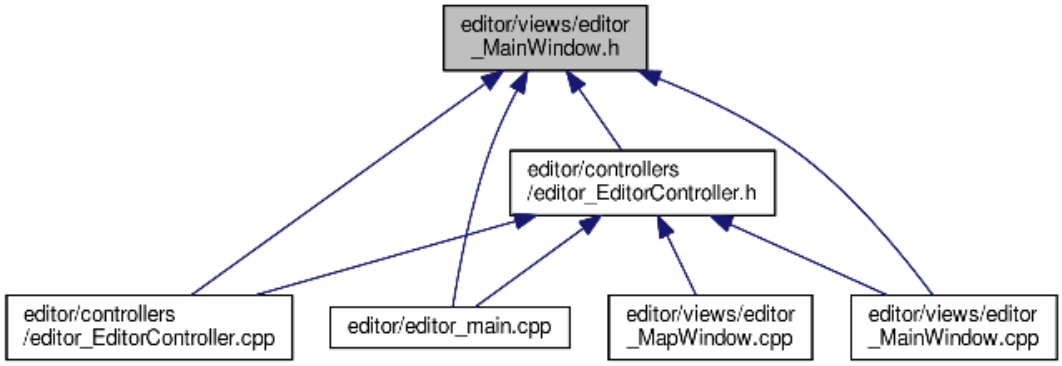

Figura 5: Diagrama de clases de MainWindow

- MapFixedWindow : Es una **Gtk::Fixed** que contiene todos los **ObstacleViewContainer** y éstos le agrega a **MapFixedWindow** las imágenes que tendrá como hijo.

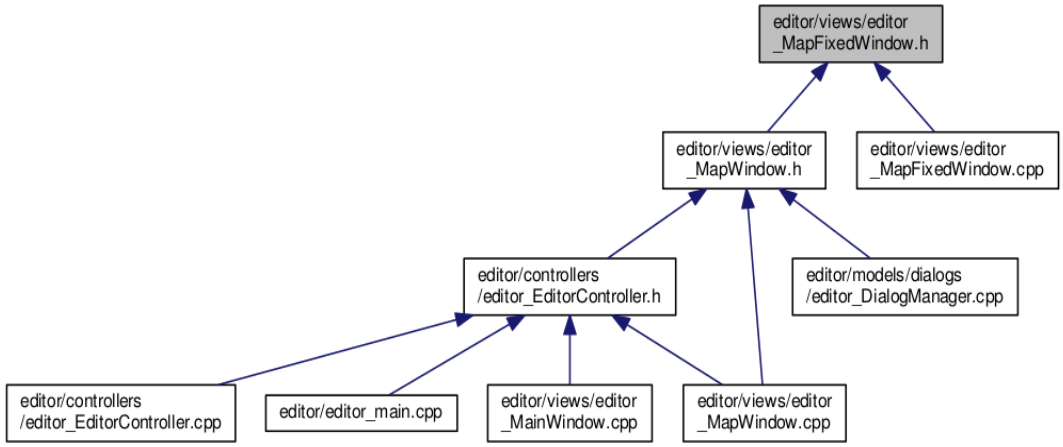

Figura 6: Diagrama de clases de MapFixedWindow

4. Controlladores : Contiene entidades que vinculan las vistas con el modelo y crean la navegación entre vistas

   - EditorController : Es la encargada de manejar el flujo entre las dos vistas, **MainWindow** y **MapWindow**. Es el delegado de ambas para la navegación lo cual hace que tanto MainWindow como MapWindow llamen a métodos de **EditorController** por ejemplo para guardar mapas o para editar un nivel especifico para que ésta haga la navegación correspondiente.

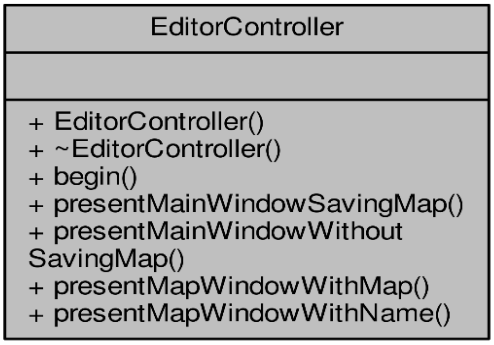

Figura 7: Diagrama de clases del EditorController

2.4.3. Diagramas

Diagramas Gráficos :

A continuación se verá cual era la idea original para la pantalla del editor y cuál fue el resultado final de la misma

Original :

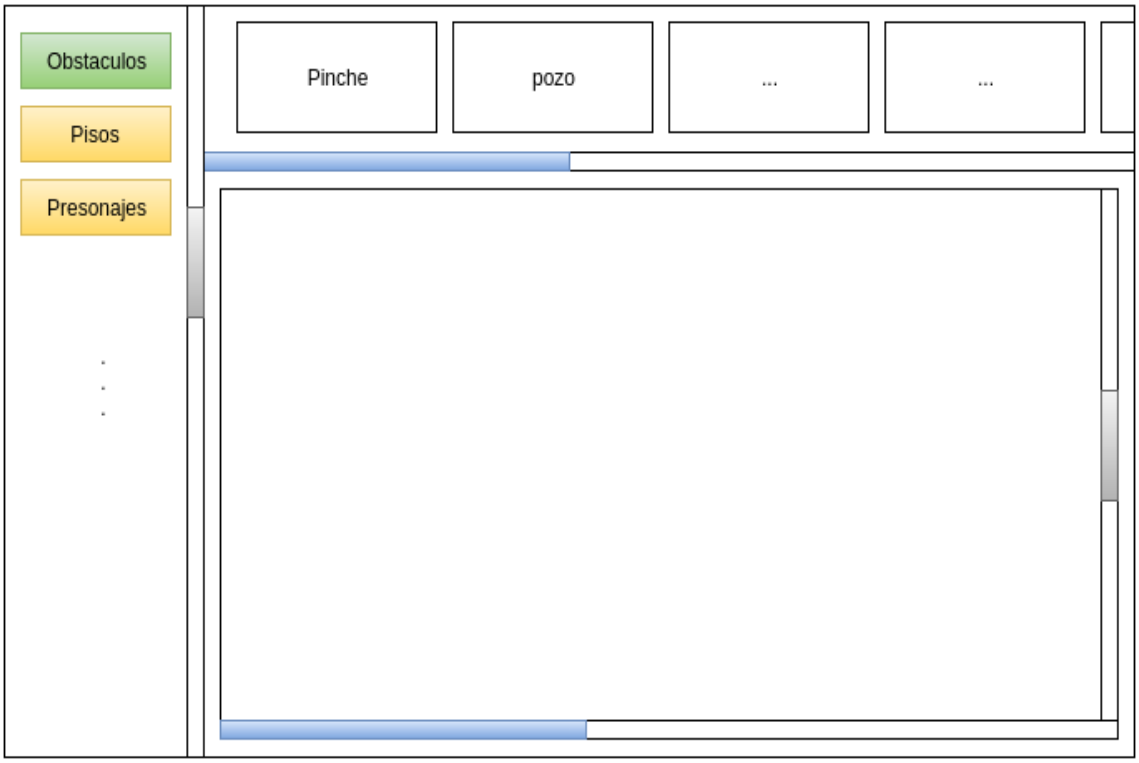

Figura 8: Diagrama original del Editor

Final :

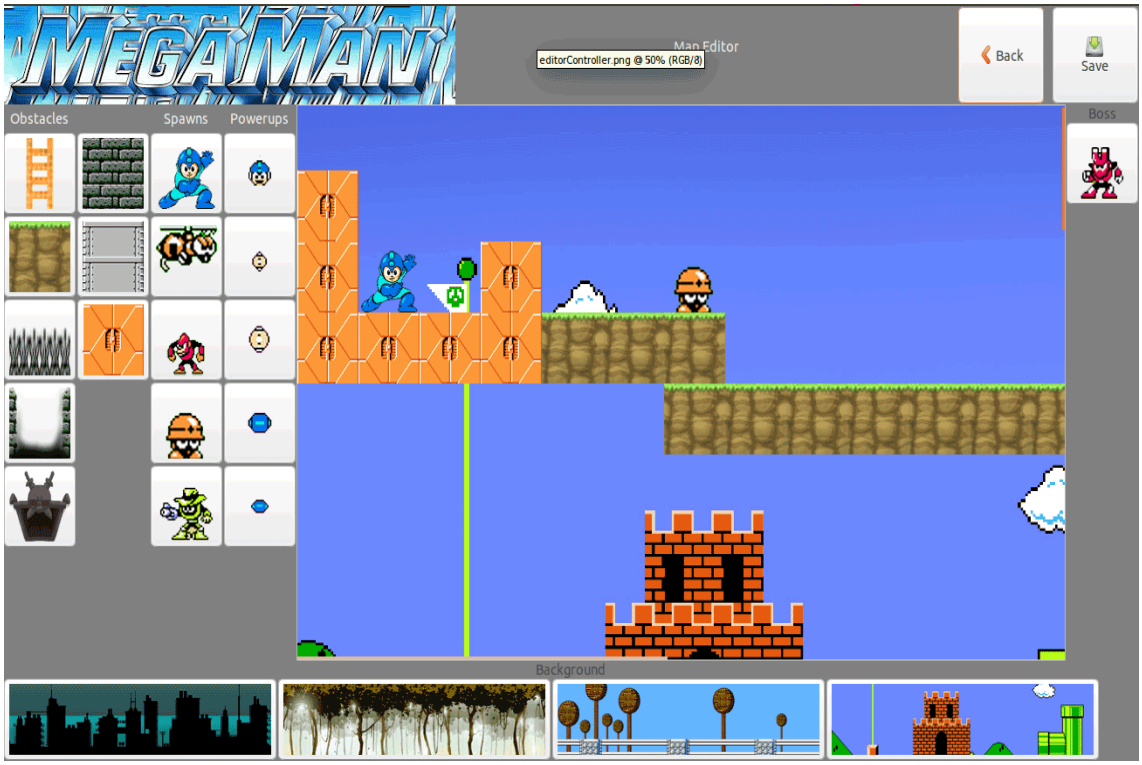

Figura 9: Apariencia final del editor

2.4.4. Descripción de archivos y protocolos

Para el almacenado de los distintos mapas se acordó usar el formato de **JSON** (JavaScript Object Notation) de la siguiente manera :

```
{
"map": {
    "name": "Bombman",
    "filename": "bosschamber1.json",
    "height": 420,
    "width": 910,
    "id": 6,
    "background_image": "res/drawable/background/background1.png",
    "obstacles": [
        {
            "x": 175,
            "y": 245,
            "type": 2
        },
        {
            "x": 175,
            "y": 385,
            "type": 2
```

```
            },
            {
                "x": 175,
                "y": 315,
                "type": 2
            }
        ]
    }
```

Como se puede ver en la representación del JSON del mapa, dentro de **obstacles** los JSON objects contenidos en ese array tienen un x, y, que representan el centro de dicho objeto y finalmente un type, este type está definido dentro de este enumerable.

```
typedef enum {
        //Obstacles

        ObstacleViewTypeLadder = 0,
        ObstacleViewTypeBlock = 1,
        ObstacleViewTypeBlock1 = 2,
        ObstacleViewTypeBlock2 = 3,
        ObstacleViewTypeBlock3 = 4,

        ObstacleViewTypeNeedle = 5,
        ObstacleViewTypePrecipice = 6,
        ObstacleViewTypeBossChamberGate = 7,

        //Spawns

        ObstacleViewTypeMegaman = 8,
        ObstacleViewTypeBumpy = 9,
        ObstacleViewTypeJumpingSnyper = 10,
        ObstacleViewTypeMet = 11,
        ObstacleViewTypeNormalSnyper = 12,

        //Powerups
        ObstacleViewTypeLife = 13,
        ObstacleViewTypeBigEnergyCapsule = 14,
        ObstacleViewTypeSmallEnergyCapsule = 15,
        ObstacleViewTypeBigAmmoPack = 16,
        ObstacleViewTypeSmallAmmoPack = 17,

        // Bosses
        ObstacleViewTypeBombman = 18,
        ObstacleViewTypeFireman = 19,
        ObstacleViewTypeMagnetman = 20,
```

```
        ObstacleViewTypeRingman = 21,
        ObstacleViewTypeSparkman = 22,

        // Projectiles
        ObstacleViewTypeBomb = 23,
        ObstacleViewTypeFire = 24,
        ObstacleViewTypeMagnet = 25,
        ObstacleViewTypeRing = 26,
        ObstacleViewTypeSpark = 27,
        ObstacleViewTypePlasma = 28
} ObstacleViewType;

typedef enum {
        //Obstacles

        ObstacleViewTypeLadder = 0,
        ObstacleViewTypeBlock = 1,
        ObstacleViewTypeBlock1 = 2,
        ObstacleViewTypeBlock2 = 3,
        ObstacleViewTypeBlock3 = 4,

        ObstacleViewTypeNeedle = 5,
        ObstacleViewTypePrecipice = 6,
        ObstacleViewTypeBossChamberGate = 7,

        //Spawns

        ObstacleViewTypeMegaman = 8,
        ObstacleViewTypeBumpy = 9,
        ObstacleViewTypeJumpingSnyper = 10,
        ObstacleViewTypeMet = 11,
        ObstacleViewTypeNormalSnyper = 12,

        //Powerups
        ObstacleViewTypeLife = 13,
        ObstacleViewTypeBigEnergyCapsule = 14,
        ObstacleViewTypeSmallEnergyCapsule = 15,
        ObstacleViewTypeBigAmmoPack = 16,
        ObstacleViewTypeSmallAmmoPack = 17,

        // Bosses
        ObstacleViewTypeBombman = 18,
        ObstacleViewTypeFireman = 19,
        ObstacleViewTypeMagnetman = 20,
```

```
        ObstacleViewTypeRingman = 21,
        ObstacleViewTypeSparkman = 22,

        // Projectiles
        ObstacleViewTypeBomb = 23,
        ObstacleViewTypeFire = 24,
        ObstacleViewTypeMagnet = 25,
        ObstacleViewTypeRing = 26,
        ObstacleViewTypeSpark = 27,
        ObstacleViewTypePlasma = 28
} ObstacleViewType;
```

El nombramiento de los archivos para los niveles fue **levelN.json** con N entre 1 y 5, para los niveles, mientras que para los bosses, **boschamberN.json** con el mismo rango de N.

2.5. Servidor

2.5.1. Descripción general

El servidor es la aplicación que corre en background y permite el modo multiplayer. También realiza el trabajo pesado de la lógica del juego, como el manejo de las físicas del juego.

Consta a grandes rasgos de 5 partes. Un engine cuyo propósito es manejar toda la lógica del juego, corriendo en base a Box2d para la gestión de las físicas. Los modelos encapsulan el comportamiento y las características de los distintos objetos, desde su forma y tamaño a por ejemplo la inteligencia artificial de los monstruos.

La sección de networking es la que encierra toda la comunicación con los clientes. Posee la responsabilidad de recibir mensajes, procesarlos y enviar los mensajes que el engine va acolando en la cola de eventos a enviar.

Los parsers se encargan de levantar los mapas y las configuraciones de los JSONs generados por el editor y de instanciar los objetos correspondientes en el modelo.

Los serializers tienen como responsabilidad serializar un mensaje que se va a acolar en la cola de eventos para que luego sea enviado a los clientes. Reciben un objeto o varios y generan una representación en JSON para ser enviada.

Finalmente los services son clases que prestan servicios al juego, como por ejemplo el convertidor de coordenadas pixel a metros o el mapper de teclas enviadas por los clientes.

2.5.2. Clases

1. Motor del juego

- Engine: Encierra la ejecución principal del juego, corriendo en su interior al mundo de Box2d y modificándolo de a acuerdo a los eventos que llegan y/o se van generando.
- EngineWorker: Thread para la ejecución del engine.
- EventContext: Interfaz para la acolación de mensajes dentro de la lista de eventos del engine.
- LootGenerator: Contiene un modelo probabilístico que se encarga de generar o no un loot al destruir un enemigo.
- Player: Contiene la información de los jugadores, como su nombre, id, Megaman asociado, etc.
- Fisicas:
  - ContactListener: Wraper del listener de Box2d, es un callback que se llama cada ve que ocurre una colisión entre objetos en el mundo y actúa en consecuencia, basado en el patrón Double-Dispatch.
  - PhysicObject: Wrapper del body de Box2d, contiene toda la información asociada a la física del juego. Todos los objetos que se instancian dentro del juego heredan de esta clase.

2. Modelo

- Characters
  - Character: Contiene las características comunes a todos los enemigos y los Megaman, como la vida, el arma y la física compartida.
    - Humanoids: Cada uno contiene su propia lógica, como vulnerabilidades, características particulares e inteligencia artificial.
      - Bombman
      - Fireman
      - Magnetman
      - Ringman
      - Sparkman
      - Megaman: Es el mas complejo de los Humanoids, ya que reacciona distinto a colisiones y puede por ejemplo intercambiar su arma seleccionada. No posee IA.
      - Humanoid: Clase padre de todas las anteriores, generaliza comportamientos.

        A continuación podemos ver quizás una de las estructuras de clases mas complejas del modelo como lo es la de la clase humanoid

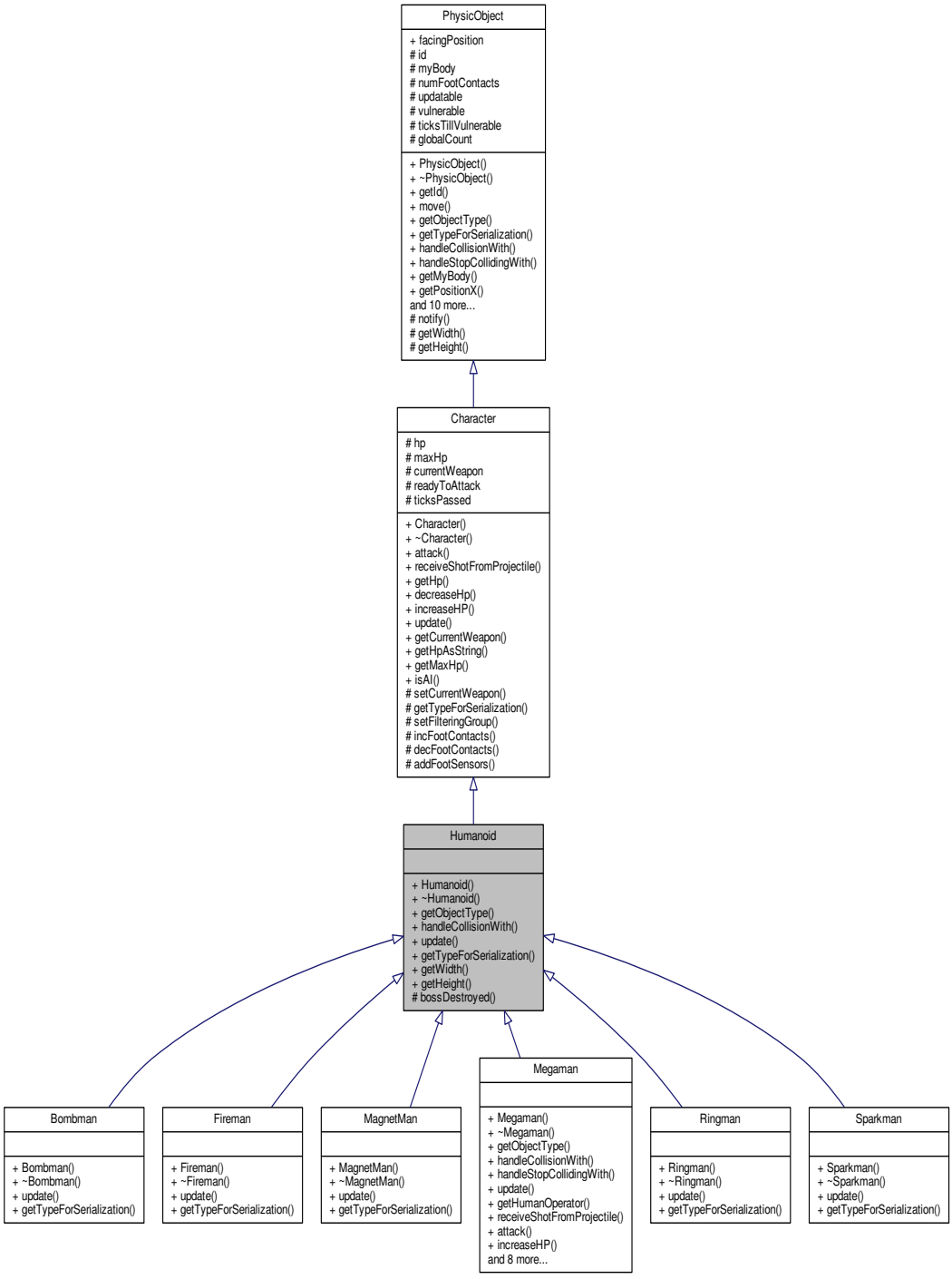

Figura 10: Diagrama de clases de Humanoid

- Mobs: Cada uno contiene la lógica en particular de los enemigos que no son bosses.
  - Bumpy
  - JumpingSniper

        - Met
        - NormalSniper
        - Sniper: Generalización del comportamiento de los snipers.
        - Mob: Generaliza todas las clases anteriores y sus comportamientos comunes ante un mensaje.
- Obstacles
    - Block: Bloque del mapa, no permite atravesarlo.
    - BossChamberGate: Puerta hacia el mapa del boss, llama al callback que vuelve a regenerar el juego.
    - Ladder: Escalera que permite subir a megaman a lugares lejanos.
    - Needle: Pinches que reducen la vida de megaman.
    - Precipicio: Reduce la vida del jugador.
    - Obstacle: Generalización de las clases obstáculo, especifica el contrato que deben definir las hijas.

      En el siguiente diagrama podemos ver la estructura de clases de la clase Obstacle

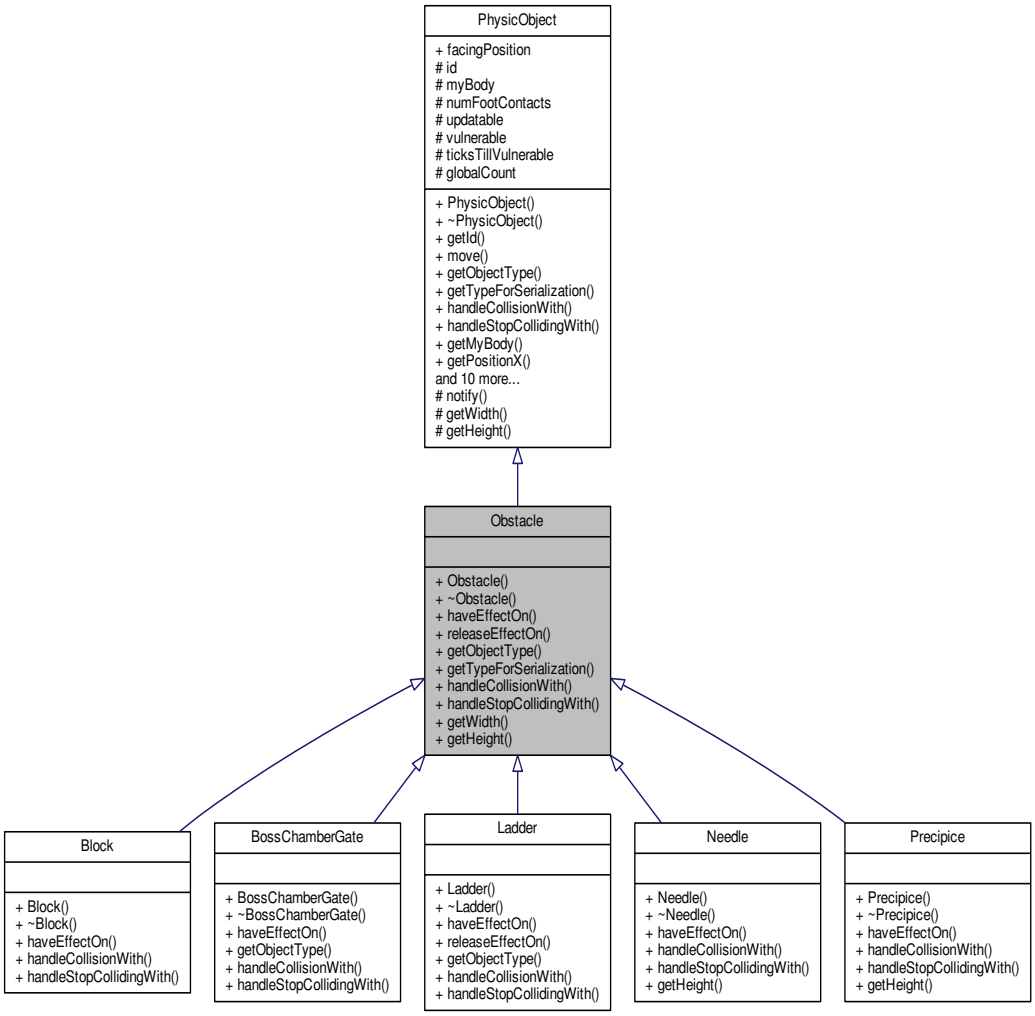

Figura 11: Diagrama de clases de Obstacle

- Powerups
  - Powerup: Aplica su efecto sobre un character, padre de los demás powerups.
  - AmmoPack: Incrementa la munición del arma actual del character en una cantidad determinada.
  - SmallAmmoPack
  - BigAmmoPack
  - EnergyCapsule: Incrementa la vida de megaman en una cantidad determinada.
  - SmallEnergyCapsule
  - BigEnergyCapsule
  - Life: Incrementa la vida del jugador.

    A continuacion se muestra un diagrama de clases de la clase powerup y sus hijas

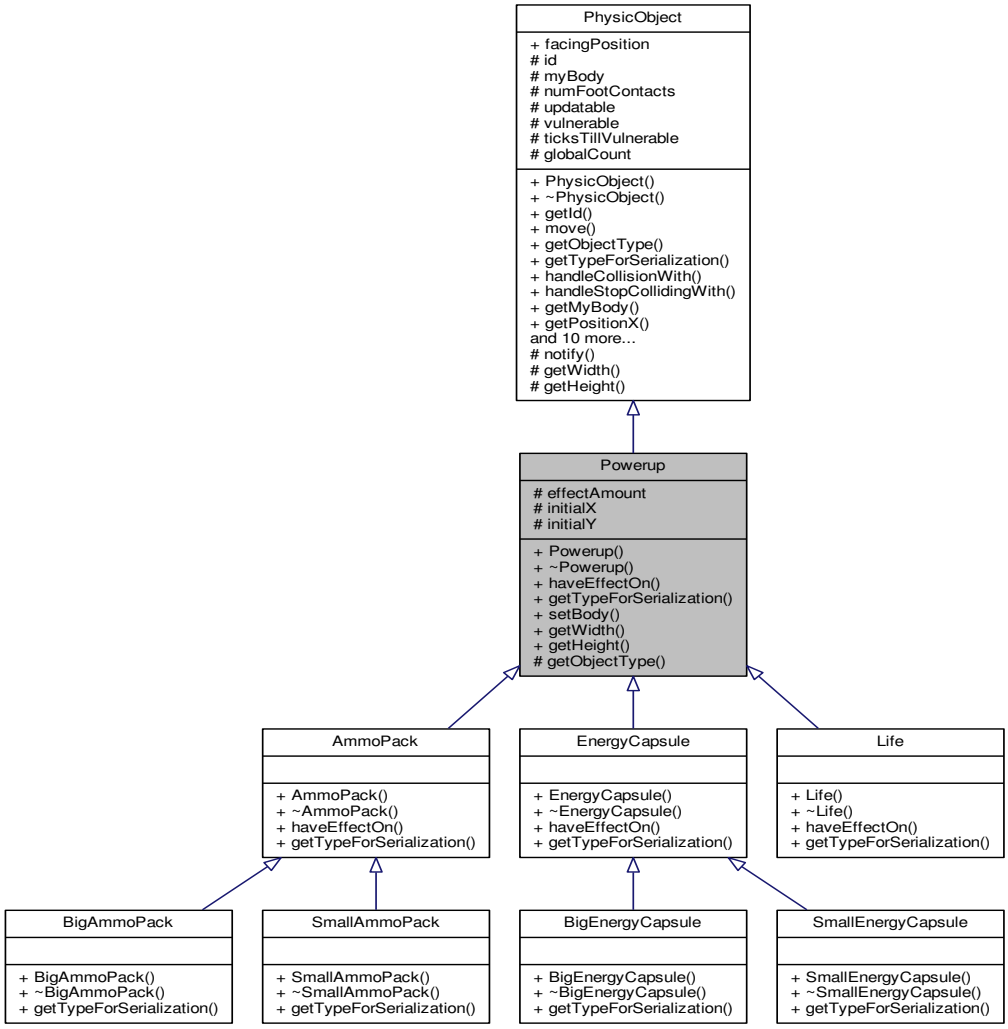

Figura 12: Diagrama de clases de Powerups

- Projectiles
  - Projectile: Padre de los demás proyectiles, generaliza comportamiento. Se encarga también de ejecutar los callbacks luego de la creación de cada proyectil en particular.
  - Bomb
  - Fire
  - Magnet
  - Plasma
  - Ring
  - Spark
- Weapons: Encapsulan la lógica de las armas y cada enemigo delega lo que es disparos en ellas.

- Weapon: Padre de las demás armas, generaliza comportamiento
- BombCannon
- Flamethrower
- MagnetCannon
- MobCannon: Posee munición ilimitada
- PlasmaCannon: Posee munición ilimitada
- RingTosser
- SparksCannon

3. Comunicaciones

   - AcceptorWorker: Thread que se encarga de la aceptación de nuevos jugadores
   - ClientProxy: Wrapper del socket asociado a un cliente
   - InboundMessagesController: Controller que se encarga del procesamiento de los mensajes que llegan a través de los clientes. Luego de procesar le indica al modelo correspondiente la acción que debe realizar.
   - ReceiverWorker: Thread encargado de recibir datos.
   - SenderWorker: Thread encargado del envió de datos.
     En el siguiente diagrama se muestra la estructura de threads del servidor. Notar que el par de threads sender y receiver es uno por cada cliente conectado

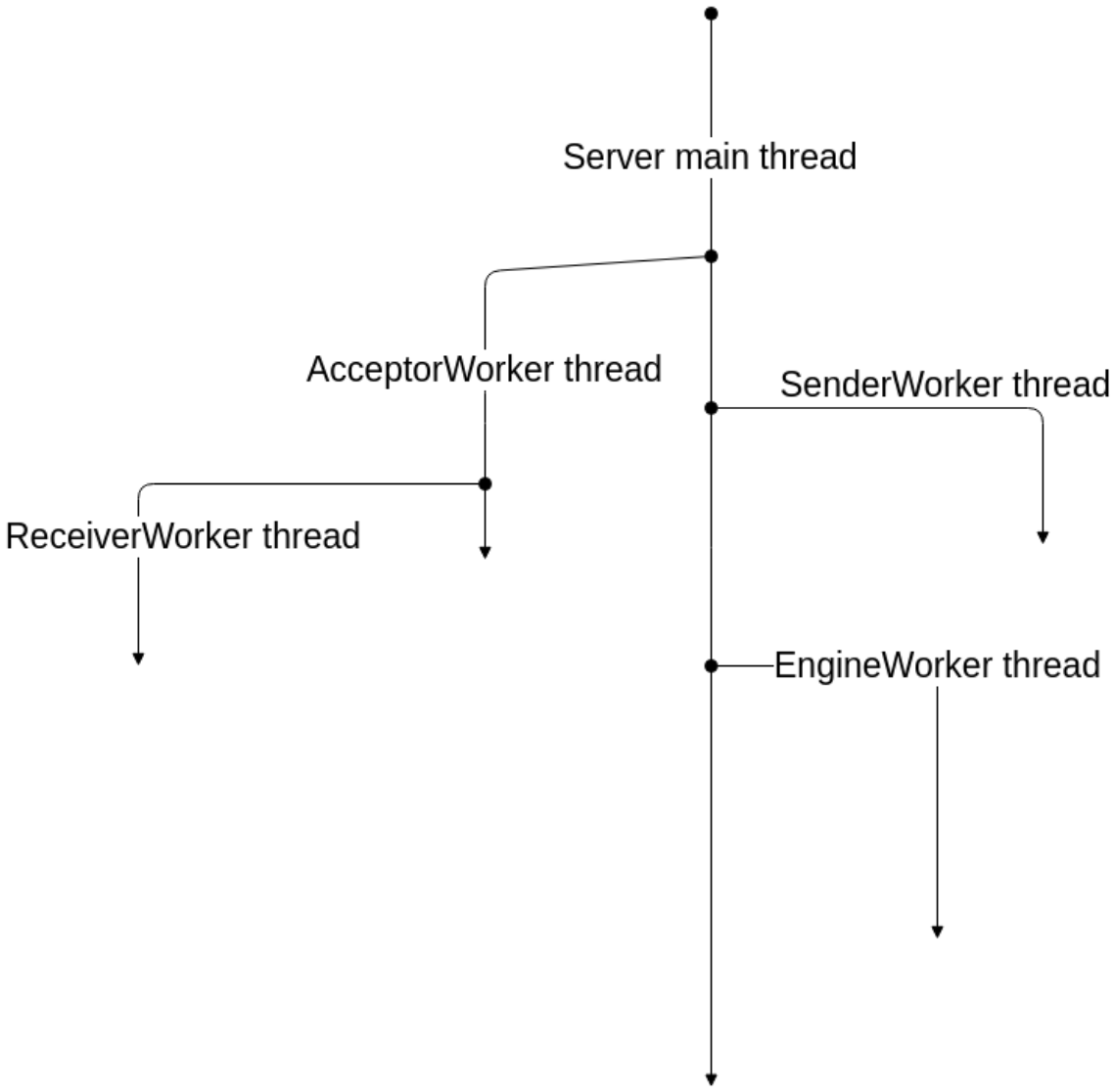

Figura 13: Diagrama de threads activos en el server. Los threads ReceiverWorker y SenderWorker van de a pares, dos por cada cliente conectado.

4. Parsers
    - ConfigParser: Levanta la configuración desde el archivo de configuraciones especificado al correr el servidor
    - JsonMapParser: Infla el mundo a partir del JSON generado por el editor
    - KeyMapParser: Parsea el keymap de movimientos enviado por los clientes
5. Serializers
    - AmmoChangeSerializer: Serializa un cambio en la munición
    - ConnectedPlayerSerializer: Indica a cada jugador su representación en el juego
    - EndGameSerializer: Envía que el juego ha concluido

- EnteredBossChamberSerializer: Prepara el mensaje indicando que se entro a la cámara del boss
- HpChangeSerializer: Indica que hubo un cambio en la vida de megaman
- LifeChangeSerializer: Indica un cambio en la vida de un jugador
- MovementSerializer: Notifica sobre un movimiento de un elemento
- NewPlayerSerializer: Envía la conexión de un nuevo jugador
- ObjectCreationSerializer: Notifica sobre la creación de un nuevo objeto en el mundo
- PositionSerializer: Serializa la posición de un objeto
- StartGameSerializer: Prepara el mensaje para iniciar el juego

A modo de ejemplo se muestra un diagrama de la clase ObjectDestructionSerializer

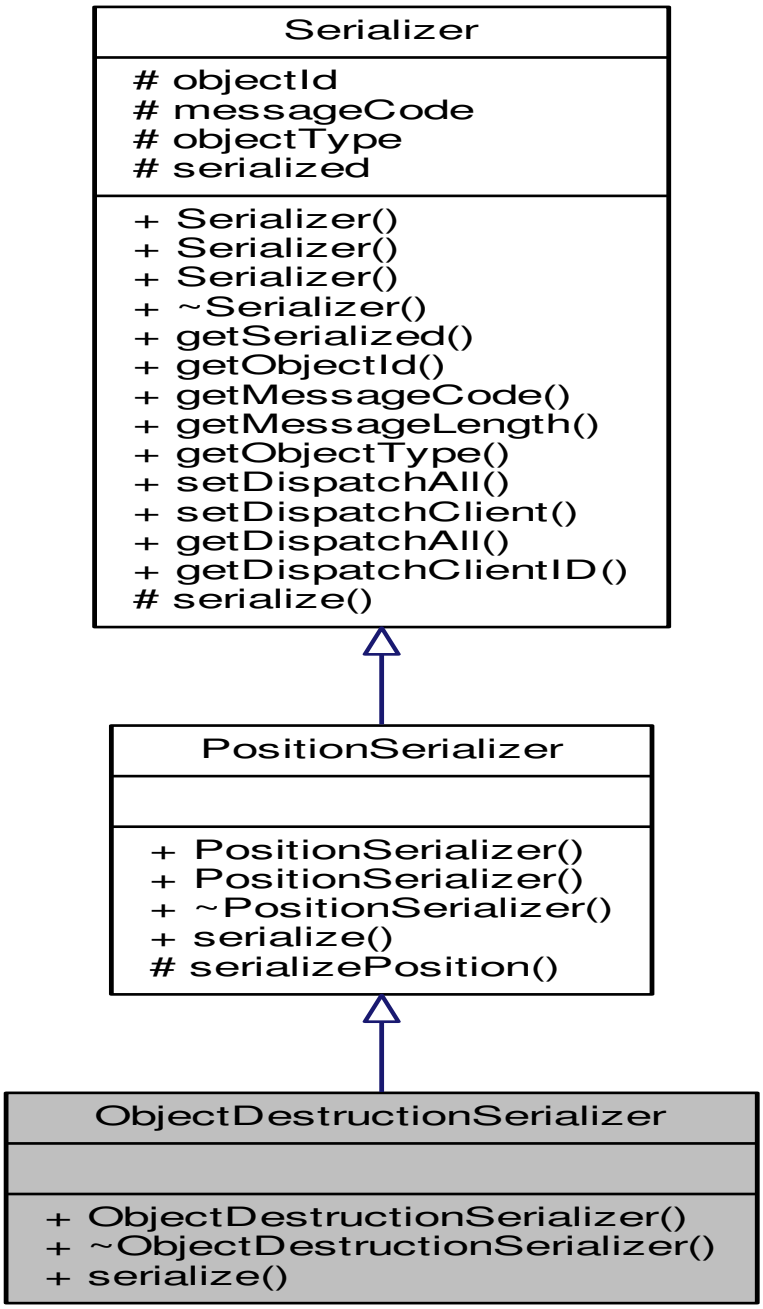

Figura 14: Diagrama de clases de ObjectDesctructionSerializer

6. Servicios

- CoordinatesConverter: Conversor de coordenadas pixel a metros y viceversa
- KeyMap: Mapper de teclas recibidas de los clientes.

7. Server: Guarda la lista de clientes conectados y el socket por el cual se van a aceptar nuevas conexiones
8. Logger: Wrapper de glog, se encarga de loguear eventos

2.5.3. Descripción de archivos y protocolos

El servidor basa su configuración en un archivo config.json que se encuentra dentro de la carpeta json. Si hay algún tipo de configuración faltante en dicho archivo se toman valores por default.

2.6. Programas intermedios y de prueba

Para la prueba del sistema de físicas se utilizo el Testbed que provee Box2d, el cual permite generar mundos ficticios de prueba y analizar los resultados de ir ajustando parámetros físicos de los objetos del mundo.

2.7. Código Fuente

https://github.com/saantiaguilera/fiuba-taller-I-megaman

3. Manual de Usuario

3.1. Instalación

1. Correr en una consola, dentro del directorio del juego

   chmod u+x install.sh

   De esta forma se le estará dando privilegios de usuario al script bash.

2. Correr el script

   ./install.sh

3. El script se encargara de lo demás. El mismo permite

   - Instalar dependencias
   - Crear todos los módulos
   - Crear modulo cliente
   - Crear modulo servidor
   - Crear modulo editor

- Borrar objetos asociados a los módulos

En caso de intentar crear algún modulo y no tener la aplicación instalada, se le notificara al usuario de que es necesario primero instalarlas.

3.2. Configuración

Toda la configuración queda seteada con la instalación, lo único que se necesita es saber la ip del servidor y su puerto para poder conectarse. El servidor especifica su puerto al correrlo y la ip de su maquina (si están en la misma red) puede averiguar se con el comando ifconfig.

3.3. Forma de uso

Para correr los módulos, desde el root del directorio hacer:

- Cliente: ./mclient
- Servidor: ./mserver ip config.json
- Editor: ./meditor

El servidor queda corriendo y no muestra mas nada por pantalla mas que un mensaje de bienvenida que dice "Welcome to Megaman 3 Server Edition".

Al correr el editor se desplegara la ventana principal que nos permite elegir que mapa queremos editar

Una vez seleccionado el mapa pasamos a la pantalla principal, donde tenemos el menú de objetos que se pueden agregar al mapa a la izquierda, el mapa a editar en el centro y el boss correspondiente a la derecha. Los objetos se pueden colocar en el mapa haciendo click izquierdo sobre ellos y dropeandolos. Pueden quitarse haciendo click derecho. Entre los elementos que podemos seleccionar se encuentran bloques, pinches, escaleras, precipicios, enemigos y powerups de vida, munición y vidas del jugador.

Una vez finalizada la edición podemos salir con los botones de save o de exit que se encuentran en la parte superior derecha de la pantalla.

Se adjuntan capturas de pantalla de las mencionadas

En la pantalla de selección de niveles el usuario puede seleccionar cualquiera de los 5 niveles y 5 bosschambers para editar el mapa. Luego de seleccionar un nivel se abrirá la pantalla de selección del mapa que fue seleccionado.

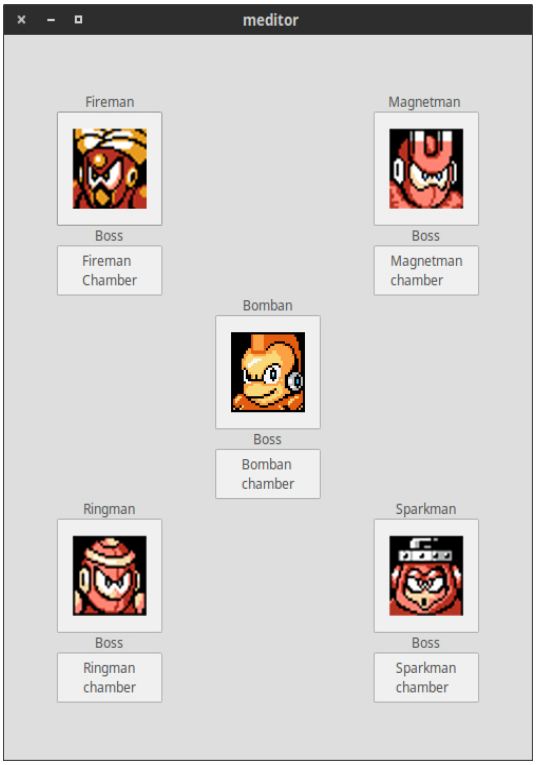

Figura 15: Pantalla de selección de niveles del editor

El la pantalla de edición de mapas, se pueden seleccionar todos los items a la izquierda, están los spawners, los obstaclulos y los powerups. Con los botones de back y save, se termina la edición del mapa. En el centro está el mapa editado, dentro de ésta pantalla se podrá scrollear libremente para ver todos los lugares del mapa. Finalmente abajo están todos los fondos de pantalla posibles para el mapa y al seleccionar uno de esos se cambiará automáticamente el fondo de la vista del mapa.

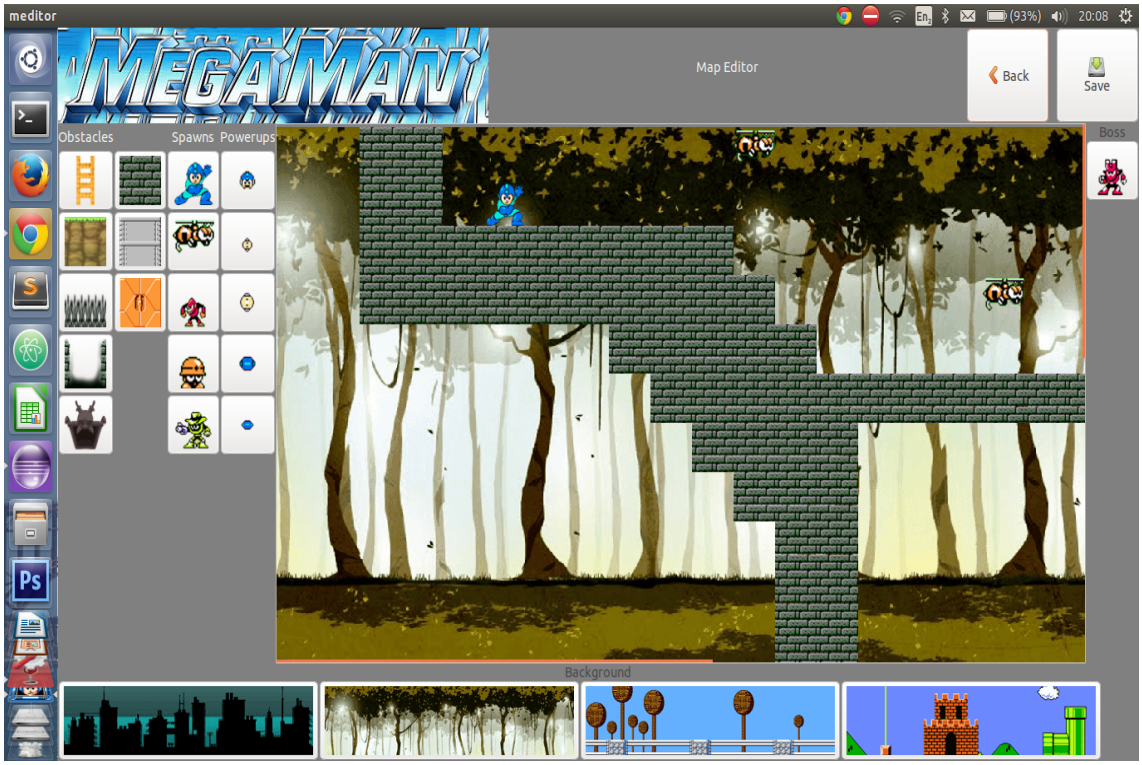

Figura 16: Pantalla de edición de mapa

En cuanto al cliente, al ejecutarlo llegamos al splash donde nos pide ingresar nuestro nombre y la ip:port del server al cual nos queremos conectar.

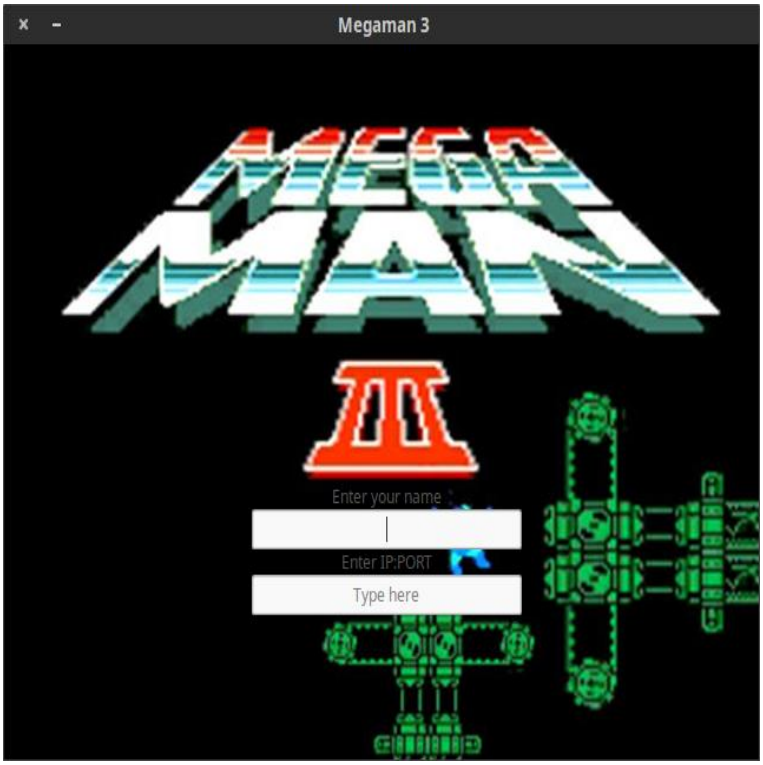

Figura 17: Pantalla de ingreso de información del usuario

Una vez conectados llegamos al lobby del juego, donde vemos los jugadores que se van conectando a la partida y podemos elegir el nivel que queremos jugar. El juego admite hasta 4 jugadores y solo el primero en conectarse puede seleccionar el nivel.

Figura 18: Pantalla de lobby

Una vez comenzado el juego entramos al mapa y debemos enfrentarnos a los enemigos en nuestro camino para llegar a la cámara del boss. Megaman se mueve con las flechas y dispara con la teclas. Para cambiar entre las armas disponibles podemos utilizar los números del 1 al 6 que están en la parte superior del teclado.

A lo largo del nivel podemos encontrar distintos powerups tirados que nos recobraran vida o munición del arma seleccionada. Los enemigos destruidos pueden tirar powerups.

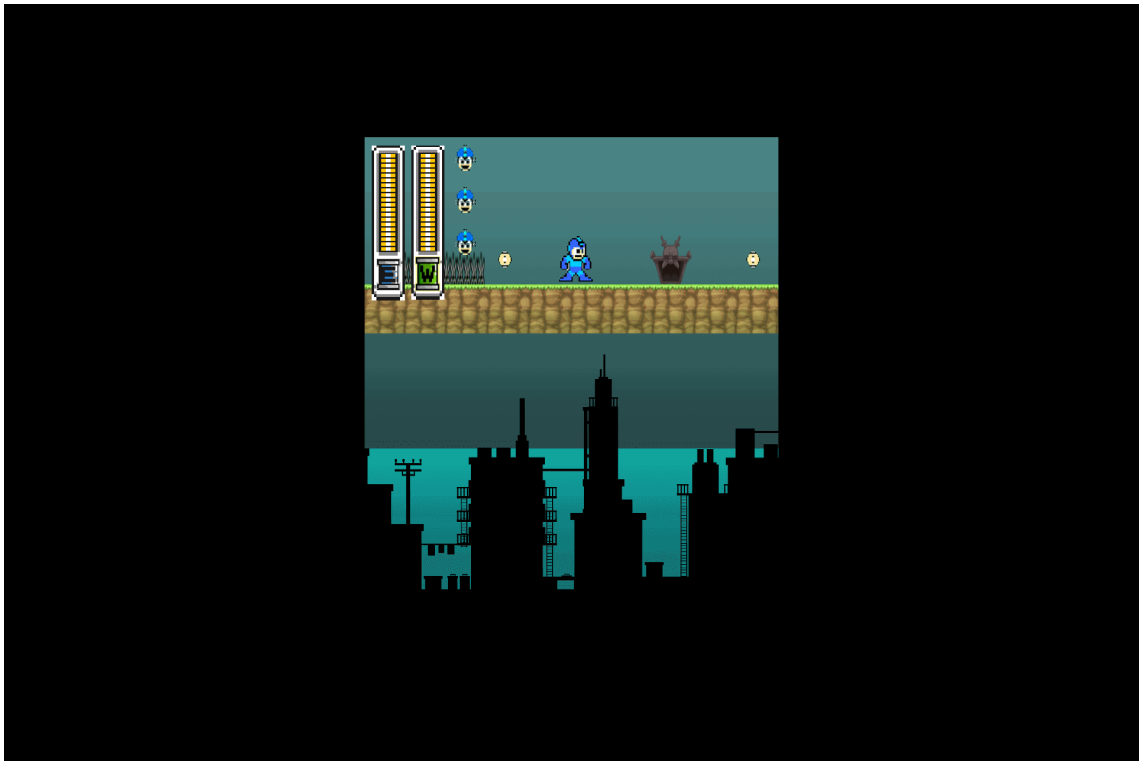

Figura 19: Pantalla del juego

Una vez que llegamos a la cámara del boss debemos matarlo para ganar y conseguir su arma.

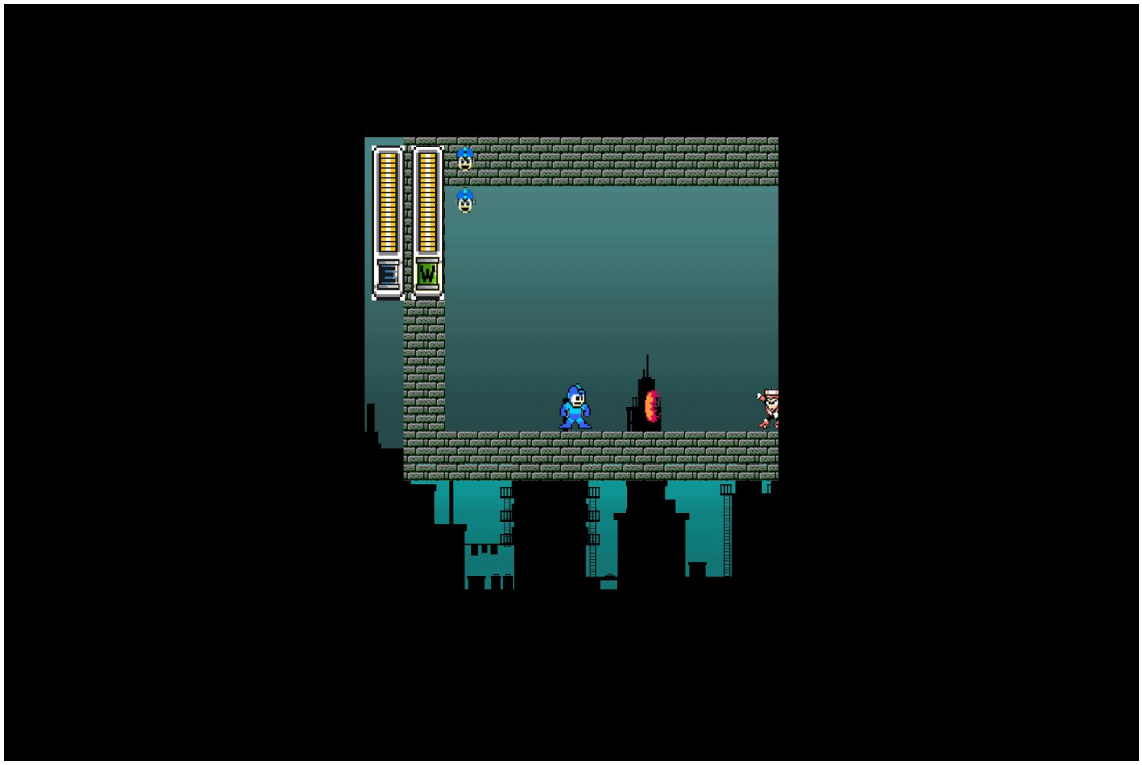

Figura 20: Pantalla del boss

3.4. Apéndice de errores

3.4.1. Editor

En el editor no se pudo solucionar el tema del background image de forma dinámica, es decir que a medida que el mapa se va agrandando no está el algoritmo para que la imagen se pegue una al lado de la otra. Sin embargo si solo se usa una imagen ésta se cropeara correctamente. El problema surge cuando el mapa es más grande que la imagen de fondo. La solución fue crear imágenes más grandes que tengan un mosaico de las imágenes de fondo así se puede notar la implementación del cropeo de la imagen cuando ésta es una sola.

Hay un problema que en algunas computadoras los eventos eventos los captura el **MapWindow**, y en otras no, en las que no los captura existe la posibilidad de que algunas vistas queden flotantes en la pantalla, se trató de reducir esto lo máximo que se pudo pero en ciertos casos siguen apareciendo éstos problemas.

3.4.2. Server

Queda pendiente encontrar alguna solución elegante a el problema de que las balas empujan a los jugadores, sin usar kinematic bodies de Box2d para las balas ya que esto reduce su funcionalidad.

3.4.3. Client

- Los sonidos están implementados pero fueron comentados ya que SDL Mixer nos generaba problemas para reproducirlos, entonces para no arriesgar la integridad del trabajo práctico se decidió dejar afuera esto

- Debido a falta de tiempo, estaría bueno poder reemplazar las variables y métodos globales que tiene GameView y hacer que los otros threads deleguen las acciones sobre la vista.

- A su vez, debido a falta de tiempo, hay muchas cosas que no se están validando del todo bien, como por ejemplo a la hora de seleccionar la IP:PORT al cual conectarse, si uno escribe hola:hola esto hará crashear el cliente, ya que el mismo solo valida la existencia de un separador : pero no de que este bien formado el campo.

4. Apéndice de correcciones

4.1. Editor

Se resolvió el background image en mosaico que anteriormente era una imagen grande que se cropeaba, ahora se calcula dinámicamente el tamaño del fondo y esto le aumentó la performance al editor cuando tiene que abrir los mapas ya que no tiene que cargar una imagen grande en memoria. Por otro lado esto también aliviano el juego en peso, ya que las imágenes grandes pesaban alrededor de 100kb.

Se agregaron validaciones que obligan al usuario que edita un mapa a que haya un spawner de megaman y uno de bosschamber, en el caso de que se esté editando un nivel, si se edita un bosschamber entonces la validación es que tenga un boss y un megaman spawner, si no cumple ésta validación entonces se presentara un diálogo que te imposibilitará almacenar el mapa, sin embargo siempre se podrá hacer back y tirar todos los cambios realizados

También se resolvió un problema que hacia a la inestabilidad del editor, era solo inicializar una variable en NULL ya que en algunas computadoras la se tenia memoria basura en esa variable lo que hacia que lanzara un segmentation fault.

4.2. Cliente

1. Se analizó con callgrind la localidad del código y cuáles eran los lugares críticos de procesamiento para luego poder atacar el problema de la performance en esos lugares.

   Uno de los lugares donde vimos un bottleneck de la performance fue en el Garbage Collector del cliente, ya que el mismo invocaba un thread y estaba loopeando para siempre viendo si podia recolectar, lo cual consumia muchisima memoria. Esto fue arreglado removiendo el garbage collector y borrando los datos en los lugares donde se quitaban los elementos (ya que el garbage collector solo limpiaba ciertos datos y la diferencia

era minúscula)

Otro lugar fue en el thread que envía datos al servidor. El mismo también corría para siempre preguntando si habia datos para enviar, y esto generaba que casi un core entero estuviese ocupado con el thread (Cuando tal vez no había datos para enviar). Lo arreglamos con un condition variable, el cual pone al thread en espera (y de esta forma el sistema puede hacer un context switch y no tener al thread ese en uso) y cuando se necesita enviar un dato se lo re activa.

En la siguiente imagen se puede ver la mejora de performance en el cliente.

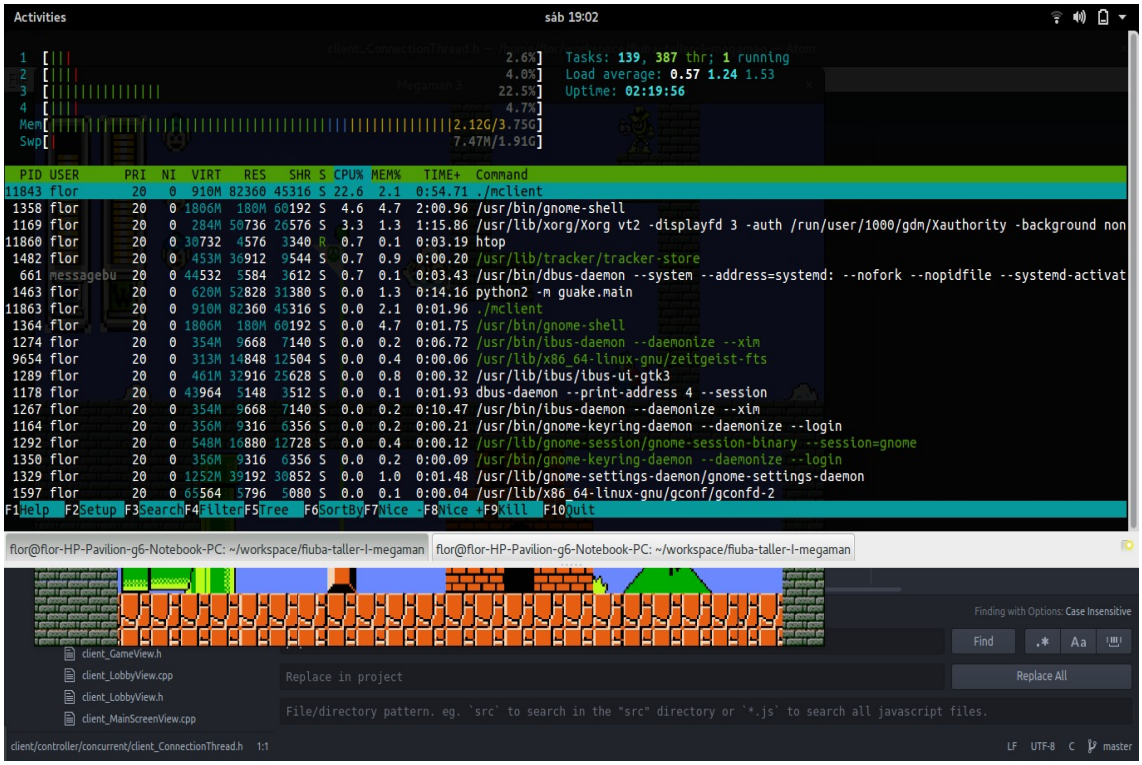

Figura 21: La performance del cliente luego de la optimización

Como se puede ver en la imagen, el cliente consume un 20 % de un thread cuando antes por hacer lo mismo se consumía el 390 %

2. Se achicaron las imágenes del terreno y personajes para que sea un poco mas amigable al usuario

3. El juego ya no es mas fullscreen por defecto y de forma obligatoria, sino que ahora tiene una resolución widescreen qHD mínima por defecto (960x540) y es capaz de resizearse a la resolución que uno desee (lo cual permite que se pueda jugar fullscreen si se desea)

4. Se puede pasar por parámetro la cantidad de FPS con los que se desea jugar el juego. Un ejemplo seria

   ./mclient 24 (Se va a jugar con 24 fps).

Vale aclarar que esto forzara la cantidad de fps, entonces si se usan pocos fps, se va a notar inconsistencias, mientas que si se usa una cantidad muy alta, puede que el juego no se vaya a ver nunca (ya que el tiempo para renderizar un frame es muy bajo).

Este parámetro es opcional, asi que si no se utiliza, se usaran 20 fps por defecto

5. Las validaciones se profundizaron, de forma que ahora si se ingresa un ip invalido se detecta y el juego no crashea.

6. Se agregaron sprites a ciertos personajes que antes no tenían, para aumentar la UX

7. Las imágenes de fondo no se mueven de forma tan drastica cuando llegan al final, de forma que ahora al ir moviéndose a lo largo del mapa el usuario no notara la diferencia de cuando una imagen de fondo llego a su fin y se reseteo.

4.3. Servidor

1. Se analizó con callgrind la localidad del código y cuales eran los lugares críticos de procesamiento para luego poder atacar el problema de la performance en esos lugares.

   El servidor a la hora de enviarle datos al cliente estaba loopeando siempre y preguntando si había nuevos datos por enviar, lo cual tenia al procesador corriendo el thread cuando tal vez no había actividad alguna por hacer.

   La solución fue usar un condition variable, el cual pone al thread en espera cuando no hay actividad por hacer, y cuando se necesitaba enviar datos se lo re activa.

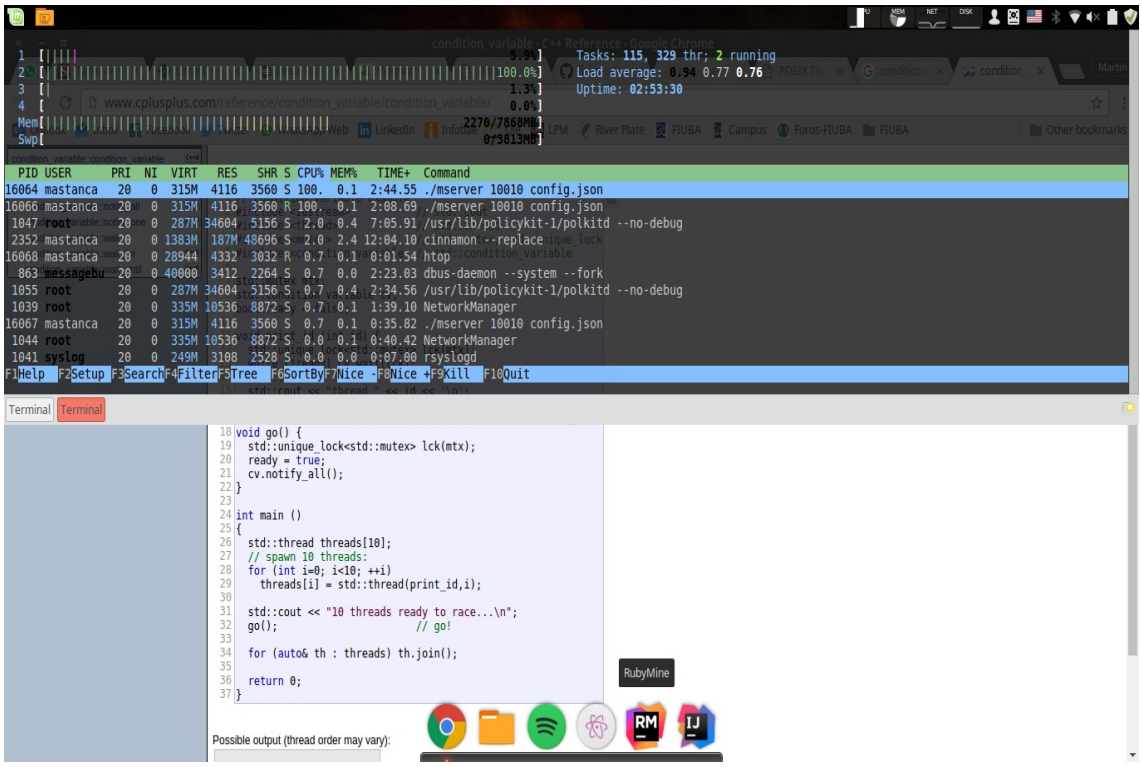

Figura 22: La performance del servidor luego de la optimización

como se puede ver en la imagen, el servidor consume 100 % en un thread de cuatro (ya que necesita estar haciendo cálculos constantemente de las posiciones y las físicas del juego) mientras que antes se usaban seis threads al 100 %

2. Al saltar, había que presionar la tecla muy rápido ya que si no el mismo no se apreciaba de forma correcta. Se arreglo para que sea como sea, el salto se lleve a cabo de forma correcta. También se aumento el impulso vertical para que salte un poco mas.

3. Ahora se pueden concatenar movimientos de diferentes ejes, de forma tal que si se desea saltar mientras se dirige a la derecha, el personaje se moverá con forma de "parábola"https://www.sharelatex.com/project/57641c5aea1cac8071b3b726

4. Al disparar y pegarle a un objeto, el objeto se moverá (y se le enviaran los datos del movimiento al cliente, cosa que antes no ocurría y los personajes quedaban en estados inconsistentes)