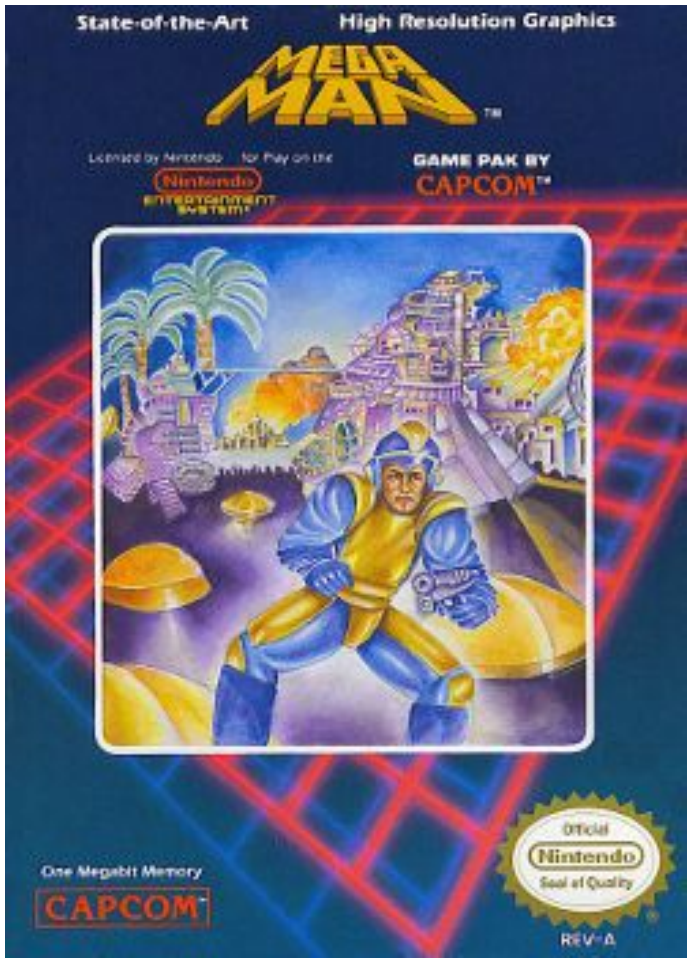
75.42 - Taller de Programación I
Informe de Trabajo Práctico Final:
Megaman Inicia



Alumnos:	Lucas Nicolas Dominguez 97150 Federico Amura 95202 Marcos Vrljicak 96693
Profesor a cargo:	Pablo Daniel Roca

1er Cuatrimestre 2016
Facultad de Ingeniería
Universidad de Buenos Aires

Manual de Proyecto

Integrantes y división de tareas

- Amura, Federico - Cliente
- Domínguez, Lucas Nicolás - Servidor
- Vrljicak, Marcos - Editor de niveles

Evolución del proyecto

A continuación se presenta un cuadro comparativo entre el calendario propuesto por la cátedra y el que se dió durante el desarrollo del proyecto. El cronograma propuesto está reorganizado en función del módulo correspondiente ya que las tareas entre los alumnos en algunos casos se intercambiaron o compartieron.

Semana	Tareas propuestas	Tareas realizadas
1	**Servidor:** ● Definición del modelo de concurrencia y protocolo de comunicaciones ● Versión básica de sistema servidor ● Diagramas de clases **Cliente:** ● Definición de interfaz gráfica y flujos de uso. ● Aplicación básica en gtkmm usando Cairomm **Editor:** ● Definición de interfaz gráfica y flujos de uso ● Aplicación básica en gtkmm usando Glade ● Diagramas de navegación (wireframes) **Otros:** ● Repositorio compartido y compilación automática	**Servidor:** ● Definición del modelo de concurrencia ● Diagrama de clases del modelo y del servidor básico. ● Servidor básico para un cliente **Cliente:** ● Definición de interfaz gráfica y flujos de uso iniciales ● Aplicación básica en gtkmm **Editor:** ● Definición de interfaz gráfica y flujos de uso (la cual sufrió múltiples modificaciones en las semanas siguientes) ● Aplicación básica en gtkmm usando Glade ● Modelo lógico del nivel y de entidades. **Otros:** ● Repositorio compartido y compilación automática

2	**Servidor:** ● Implementación del protocolo con servicios y resultados ficticios ● Implementación básica del modelo de juego ● Logging **Cliente:** ● Dibujos de sprites en Cairo/SDL ● Captura de teclado ● Movimiento veloz de sprites en base al teclado **Editor:** ● Aplicación estática editora con Glade ● Lectura y escritura de archivo XML/JSON de niveles con datos ficticios	**Servidor:** ● Servidor concurrente para varios clientes ● Logging **Cliente:** ● Dibujo de sprites en pantalla de nivel, con movimiento aleatorio **Editor:** ● Lectura y escritura de archivos JSON con datos reales. ● Replanteo de la lógica de nivel para adecuarse al enunciado **Otros:** ● Definición de entidades para todos los módulos y edición de los sprites correspondientes
3	**Servidor:** ● Modelo de partidas y niveles. ● Lectura de XML/JSON de niveles ● Solución concurrente para aceptación de clientes y ejecución de juegos. **Cliente:** ● Comunicación con servicios del Servidor. ● Empleo de imágenes secuenciales para el movimiento de jugadores **Editor:** ● Toolbox ● Drag and Drop de elementos al nivel ● Grabación básica de niveles **Otros:** ● Creación de imágenes para reutilizar en cliente.	**Servidor:** ● Implementacion basica de modelo de partidas ● Lectura básica de json, para obtener parametros de configuración. ● Modelo cero del juego (movimiento básico). **Cliente:** ● Comunicación con MockServer para movimiento de sprites a pedido ● Envío de teclas presionadas **Editor:** ● Toolbox ● Workspace básico basado en un grid **Otros:**
4	**Servidor:** ● Mejoras en lógica de modelo. ● Integración del protocolo con el modelo dinámico. ● Configuración de parámetros basados en archivos de texto. **Cliente:** ● Flujo de pantallas para Crear y	**Servidor:** ● Mejoras en el modelo, armado de niveles a partir del json del editor. ● Mejoras en el protocolo de comunicación. **Cliente:** ● Separación de vista y modelo

	Unirse a partidas. ● Reproducción de Sonidos. ● Configuración de parámetros basados en archivos de texto. **Editor:** ● Abrir, Guardar y Guardar como finalizados. ● Drag and Drop de elementos nuevos y existentes. ● Altas, Bajas y Modificaciones de niveles finalizados.	● Multithreading ● Manejo de pantallas automático según eventos del servidor **Editor:** ● Mudanza de ventanas armadas en Glade a código gtkmm puro ● Workspace basado en fixed con DrawingArea ● Selección y eliminación de elementos ● Drag and Drop de elementos al nivel ● Resize automático de nivel al agregar y quitar elementos **Otros:**
5	**Servidor:** ● Pruebas y corrección sobre estabilidad. **Cliente:** ● Pruebas y correcciones en la jugabilidad. **Editor:** ● Rotación y atributos extra para cada elemento. **Otros:** ● Detalles finales y documentación preliminar	**Servidor:** ● Añadida la lógica de vidas, daño, inmunidad. ● Movimiento de pantalla. ● Drops de items, items con efectos. ● Mejora de estabilidad y performance en server. ● Modelo de juego acepta varios jugadores. **Cliente:** ● Protocolo de comunicación formalizado ● Refactor recursos comunes ● Escalado definitivo **Editor:** ● Separación de nivel en main y chamber con sus respectivas tabs en el editor ● Diferenciación de niveles estáticos para chambers con niveles con resize automático ● Barra de manú con New, Open, Save y Save as **Otros:** ● Documentación preliminar
6	**Otros:** ● Testing y corrección de bugs ● Documentación	**Servidor:** ● Diagramas, documentacion. ● Pruebas, refactors y reasignación de responsabilidades. ● Más sprites y estados de

		megaman hacia el cliente **Cliente:** ● Fix bugs ● Mejorado el manejo de sprites ● Guardado el estado de la partida ● Agregadas las barras de salud ● Cursor invisible en nivel ● Rediseño pantalla de selección de nivel con capacidades dinámicas **Editor:** ● Selección de fondo de pantalla ● Drag and drop dentro de nivel ● Forma alternativa de borrar elementos. ● Validaciones de nivel **Otros:** ● Testing y corrección de bugs ● Diseño de los mapas por defecto
7	**Otros:** ● Correcciones sobre pre-entrega ● Testing ● Documentación ● Armado del entregable	**Servidor:** ● Alta de niveles validados por el editor y filtrado de niveles inválidos ● Se agregaron comportamientos nuevos a los jefes y otros enemigos ● Desconexión de jugadores apropiada en el modelo de juego. **Cliente:** ● Diseño final de pantallas ● Agregados mensajes de inicio y fin de nivel ● Sprites dinámicos y jugadores de colores ● Mayor estabilidad ● Limpieza de código y refactor **Editor:** ● Detalles menores. Se minimizó su desarrollo para focalizar en el servidor y el cliente **Otros:** ● Correcciones sobre pre-entrega

		● Testing ● Documentación ● Armado del entregable

Inconvenientes encontrados

Los mayores problemas se dieron por el cambio de paradigma que implicó la programación orientada a eventos y el uso de grandes frameworks y librerías, que a veces, no se comportan como se espera, o requiere una investigación y capacitación para la utilización correcta.

En el servidor/modelo tardo mucho mas de lo aparentemente asignado en diseñar todas las funcionalidades del modelo, incluso no habiendo implementado algunas. El diseño e implementación del modelo de juego resulto ser mas complicado de lo esperado, al haberlo construido contra una interfaz gráfica separada de este.

Análisis de puntos pendientes

Cliente

- Indicador del arma escogida.
- Indicador de vidas restantes.
- Reproducción de sonidos.

Servidor

- Jefes con sus armas correspondientes.
- Más armas para megaman / elegir armas / cargar y descargar armas.
- Enemigos con escudos.
- Daño diferencial según tipo de personaje y arma.
- No envía mensajes para reproducción de sonidos.
- Logs alternativos en modo debug.

Editor de niveles

- No se pudo encontrar una solución prolija y libre de bugs para el dibujado del fondo de pantalla. Si bien estos se pueden seleccionar, no se pueden visualizar hasta ya dentro del juego. Se intentaron varias alternativas, pero se encontraron varios inconvenientes en cada una. Se dejó de buscar una solución ante la directiva del profesor de concentrarse en los otros módulos del juego.

Bugs conocidos

Cliente

- A veces gtkmm no responde al cambio pedido sobre un dibujable, no es notorio cuando después se actualiza por otro motivo pero por esto a veces quedan balas sin desaparecer
- En raras ocasiones gtkmm lanza una excepción que puede no ser atrapada y terminar rompiendo el cliente

Servidor

- No se envía el mensaje de rechazo al cliente.
- Enemigos toman a un personaje de favorito (para intentar matarlo), no el más cercano.
- Items deberían desaparecer al salir de pantalla.

Editor de niveles

- En muchos casos cerrar la aplicación provoca un segmentation fault.

Herramientas

Para el desarrollo del proyecto se utilizaron las siguientes librerías, programas y servicios externos:

- Jsoncpp para la lectura y escritura de archivos json, utilizado tanto para las constantes del juego como para el guardado y cargado de niveles.
- Para el control de versiones se utilizó Git con un repositorio alojado en Github.
- Box2D para la lógica de física y colisiones.
- GLog para el logueo de informacion del server.
- Gtkmm y cairomm para las interfaces visuales.
- CMake para el compilado de las aplicaciones.
- Gdb para depuración.
- Gimp para la edición de imágenes y sprites.
- NetCat para pruebas de cliente y server.
- Glade fue usado para armar prototipos de las ventanas pero el programa en su estado actual no lo utiliza.

Conclusiones

Este proyecto fue nuestra primer experiencia desarrollando un programa de complejidad media con módulos orientados a eventos y con utilización de hilos y sockets. Como tal, se nos presentaron por primera vez las problemáticas y desafíos asociados a tales programas. Esto nos permitió, mediante prueba y error, aprender en cierta medida qué metodologías facilitan su desarrollo y dónde se encuentran los problemas que requerirán más atención.

Adicionalmente, al tratarse de aplicaciones gráficas, fue la primera vez que desarrollamos un programa dándole prioridad a la experiencia del usuario. Los paradigmas y métodos de diseño fueron sólo un vehículo y se supieron sacrificar en los casos adecuados.

Teniendo esto en cuenta, consideramos que el desarrollo de éste proyecto fue una experiencia formativa que tendrá un impacto positivo en otras futuras.

Documentación técnica

Descripcion general

El programa consiste de tres módulos independientes: Un servidor con la lógica del juego, un cliente que interactúa con los jugadores y se conecta remotamente con el servidor, y una aplicación para construir los escenarios del juego.

Server

Descripcion General

El servidor maneja la interacción entre los clientes y el modelo, que posee la simulación física e interacciones del juego.

Clases y diagramas

Las clases están divididas en dos secciones generales, la parte del modelo del juego, y la parte del servidor.

La parte del servidor se encarga de la comunicación con los clientes, y de la lógica de partidas (inicio, seleccion, finalización, dirigir mensajes recibidos al modelo). El modelo se encarga de la generación del nivel a partir del archivo generado por el cliente, y de simular el juego en sí, mandando mensajes a través del servidor al cliente, para el dibujado de las imágenes y etc. .

Todos los diagramas se encuentran también en formato .png en la carpeta de diagramas del entregable.

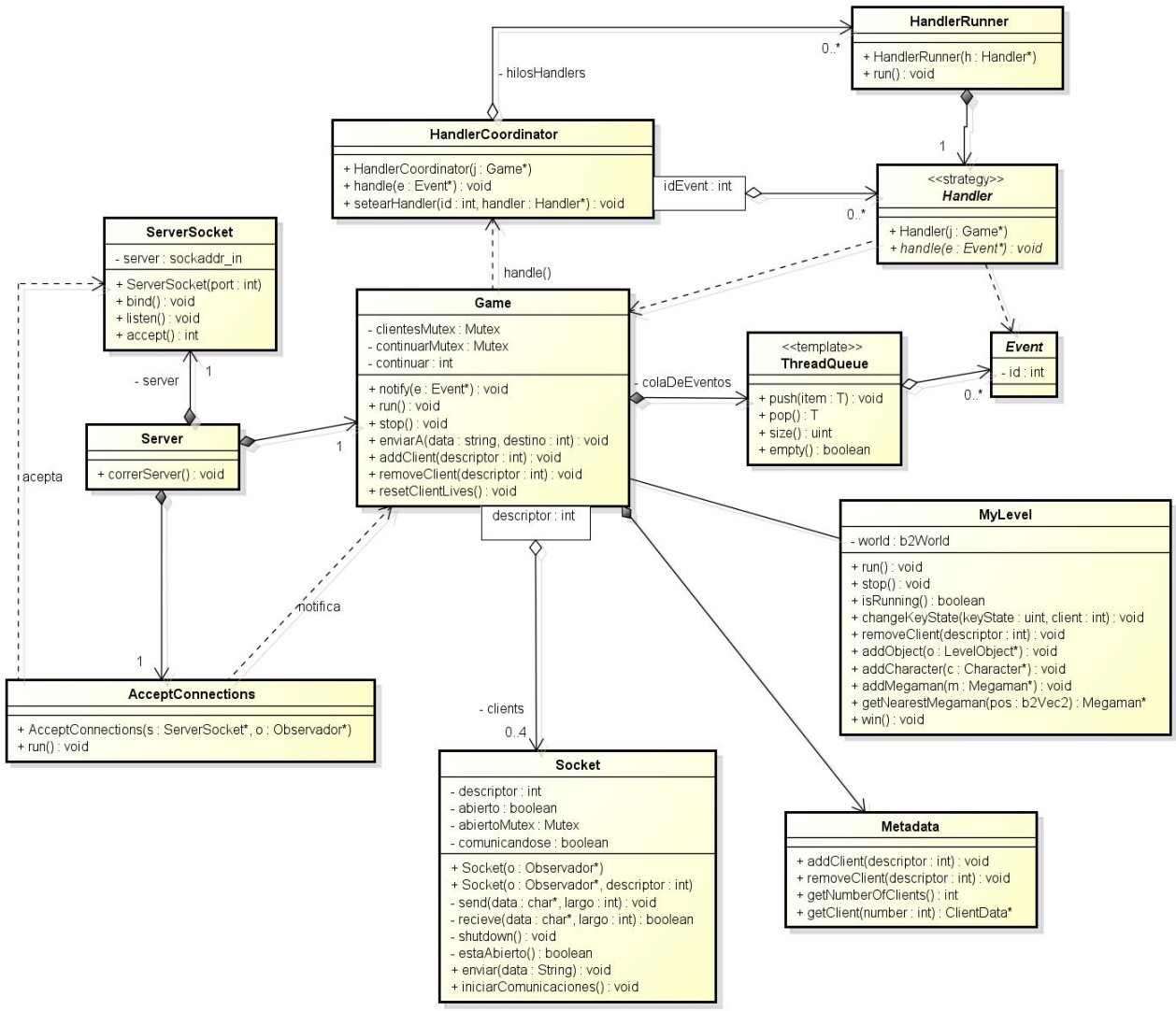


Diagrama general del server

El server posee un hilo principal, que a su vez crea el hilo aceptor de conexiones(AcceptConnections) y del juego(Game). Este hilo principal corre hasta que se ingrese el caracter q por consola. Una vez ingresado se completa el programa principal y se junta y destruyen los hilos del juego y de aceptar conexiones, a su vez cerrando las conexiones establecidas.
La clase Game posee el nivel actual, los clientes conectados, una cola de eventos y un grupo de manejadores de eventos. A su vez posee la metadata, para mantener información entre niveles.
Varios eventos, como la recepción de mensajes, el finalizado de un nivel, la conexión de un cliente; pueden ser generados a medida que se ejecuta el juego, los cuales entran a la cola de eventos, y son mandados al handler coordinator, el cual elige el handler correspondiente para el evento, manejándolo de esta forma. Una vez manejado el evento este es destruido.

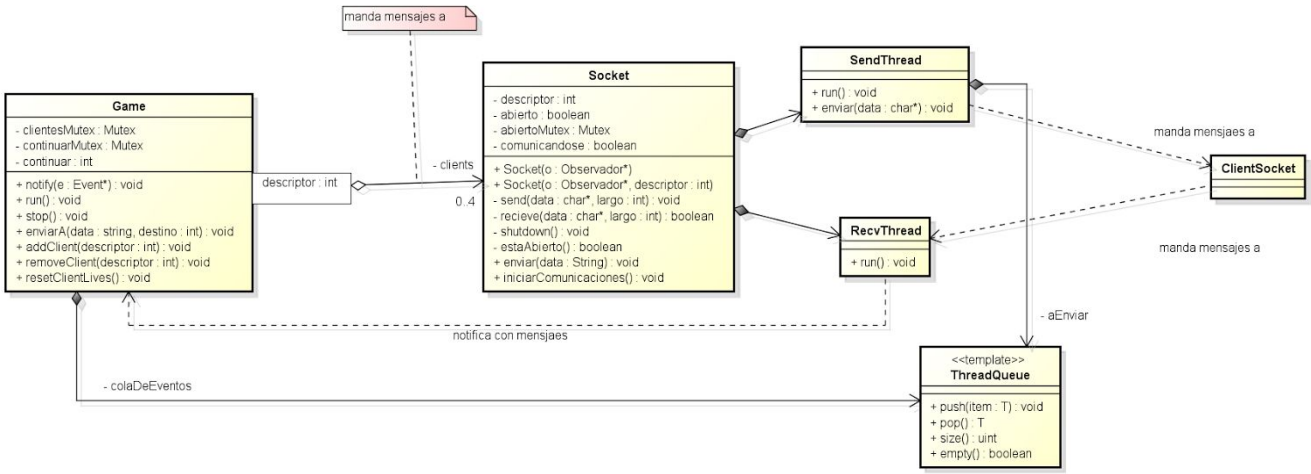
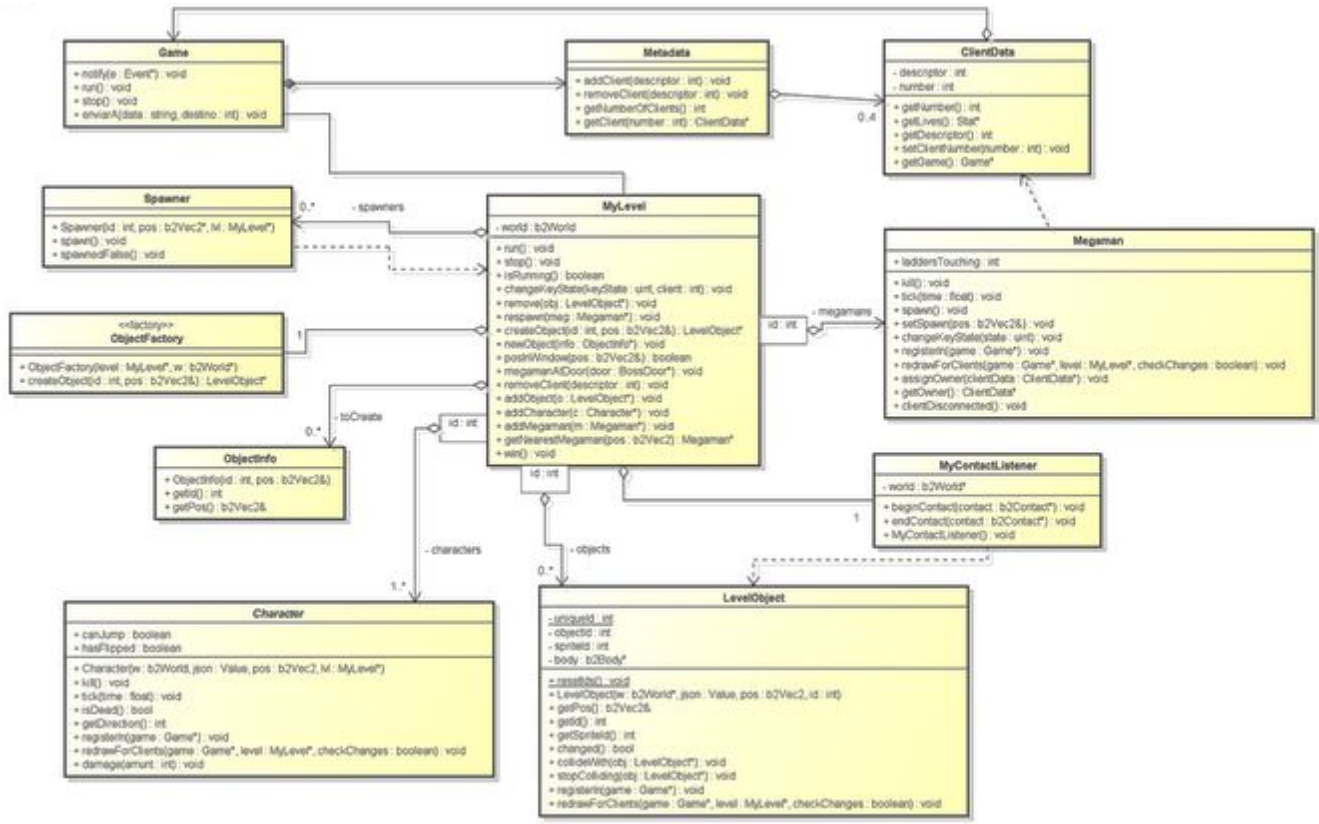


Diagrama de la parte de comunicaciones del server

Cada vez que se acepta una conexión nueva, se solicita una nueva conexión al juego (en forma de evento), la cual es rechazada o garantizada según la cantidad de clientes conectados y si ya comenzó un nivel. Al conectarse el cliente es asignado un número, el actual primer cliente teniendo control sobre la elección y finalización de niveles. En caso de desconectarse el primero, y habiendo otro, uno nuevo es asignado y notificado, según orden de conexión.
Cada cliente consiste de un socket, que a su vez tiene un hilo de recepción y un hilo de envío. Al enviarse un nuevo mensaje se agrega a la cola de envíos, la cual es enviada por medio del socket en el orden correspondiente. Cada recepción de mensaje genera un nuevo evento, del cual es notificado al juego, que a su vez realiza la acción correspondiente a ese evento.



Estructura general del modelo de un nivel.

Por cada cliente conectado, hay ClientData en la metadata que guarda la información sobre el número de jugador, vidas, e identificador. Esta data es utilizada para la comunicación y funcionalidad del nivel.
Al elegir un nivel, se crea una nueva instancia de este, el cual es construido a partir de los jsons, y además es un thread, lo que le permite correr separado del servidor.
Cada nivel consiste de un mundo de BOX2D, un manejador de contactos, un factory de objetos, el conjunto de objetos que lo pueblan (seguidos con un map), un conjunto de caracteres(enemigos y megamanes), y los megamanes (uno por cliente).
Para crearlo se levantan dos archivos json: el nivel elegido desde el server(a pedido del cliente), y el archivo de configuraciones. Se crean los objetos del escenario, los spawners de enemigos, y se posiciona tantos megamanes como clientes y posiciones iniciales en el nivel halla.
Una vez arrancado el nivel, se hacen iteraciones(simulado del paso del tiempo en el modelo) por segundo con una frecuencia determinada en el archivo de configuraciones.
Como BOX2D no soporta la creación o destrucción de objetos en medio de una iteración del mundo, estas cosas ocurren dentro de la iteración del nivel pero fuera de la iteración física.
En cada iteración del nivel se realiza en orden:

- Mueve la pantalla si corresponde, matando personajes que quedaron fuera y generando los que no estaban antes en ella.
- Remueven los objetos que deben morir.
- Se crea nuevos objetos.
- Se realiza la simulación física, generando colisiones, daños y movimientos.
- Se informa a los personajes del paso del tiempo.

- Se hace spawn a los personajes que hace falta.
- Se realiza un chequeo de si todos los megamanes están muertos, para revivirlos, o terminar el nivel.

Esto continúa hasta que se frene el nivel, ya sea por que perdieron, se pidió que se detenga, o se gano.

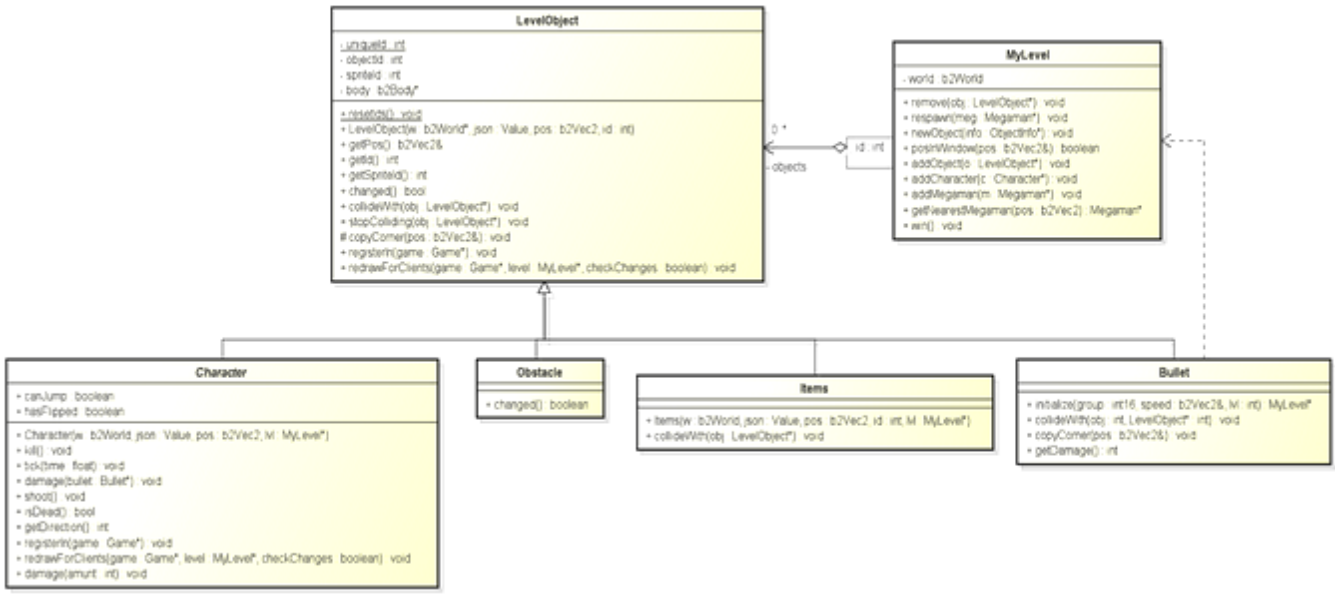
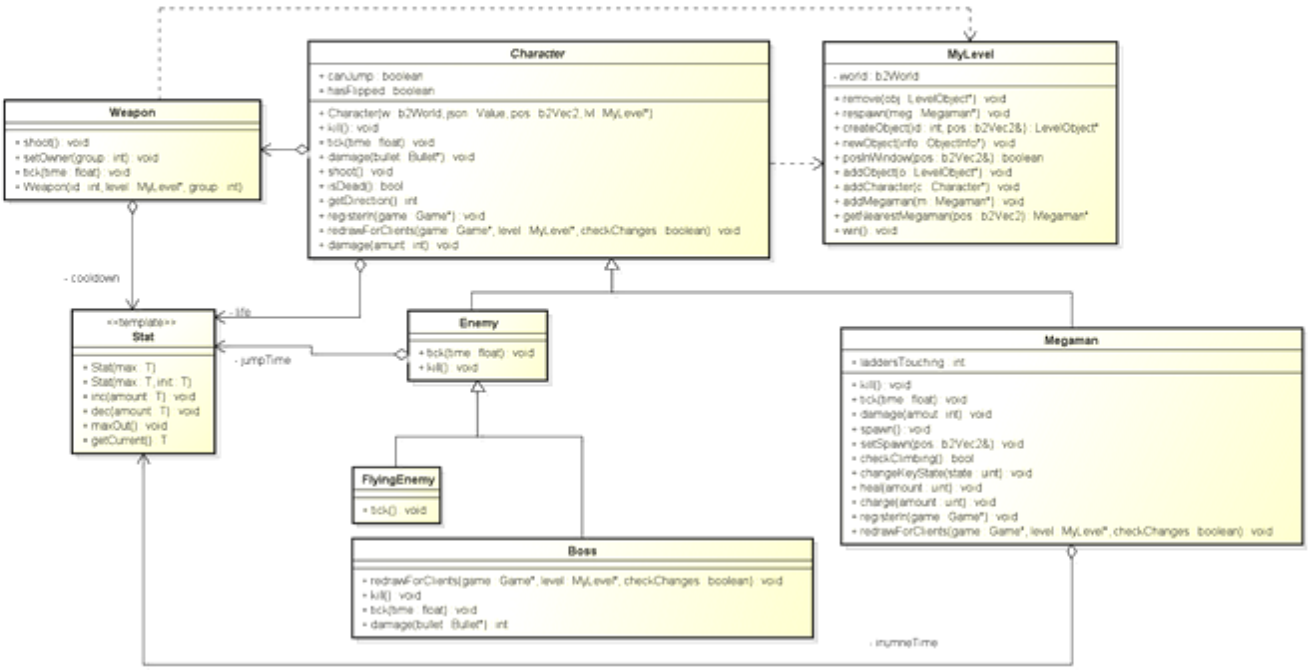


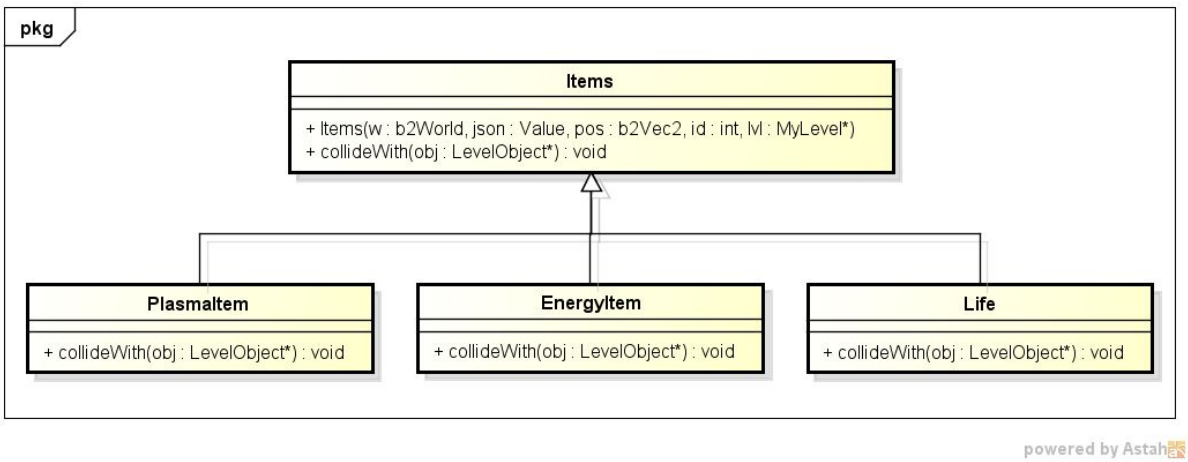
Diagrama General de los objetos.

Cada objeto es creado dentro del mundo, con una posición, y con características físicas y comportamiento según el archivo de configuraciones de json. Todos los objetos deben ser registrados en el nivel luego de creados, pudiendo estar en tres categorías no disjuntas: megaman, personaje(character), objeto. Cada objeto tiene un ID único, un spriteId con el cual informan de su estado a los clientes, y saben cómo dibujarse para un cliente. Además tienen comportamiento a realizar según las colisiones y pueden modificar su estado de movimiento y posición en la simulación física .



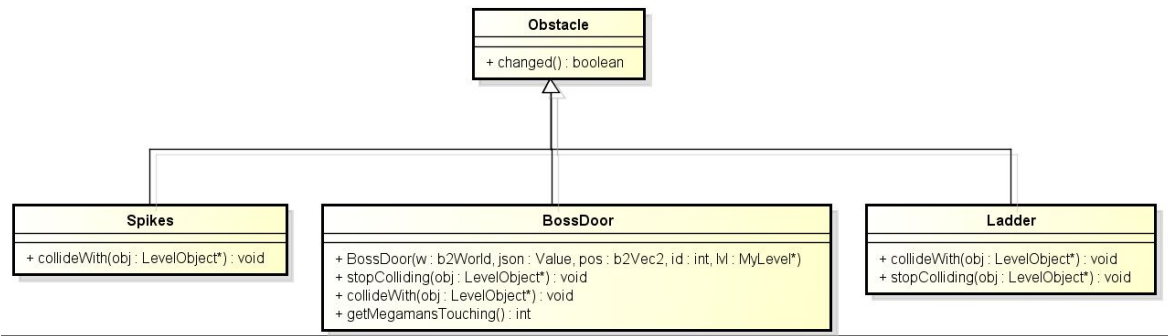
Detalle de los personajes.

Todo personaje(Character) puede ser dañado, siente el paso del tiempo(tick) y puede disparar. Para la vida y manejos de algunos tiempos se utiliza un Stat, el cual tiene un máximo valor y no puede ir menor que 0. Todo personaje posee un arma, la cual utiliza para disparar. Al disparar una nueva bala es creada, la cual es un objeto más , con su propio comportamiento al colisionar.
Los enemigos y los megamanes no colisionan entre sí , ni uno con el otro. Las balas solo colisionan con los personajes, pero no contra los del mismo bando.



Detalle de los items

Todo ítem realiza una acción al chocar con un Megaman(no colisionan con los enemigos). Luego de realizar dicha acción el ítem es destruido.



Detalle de los obstáculos.

Los obstáculos son objetos básicos del mundo. Son estáticos , es decir, no se mueven. Hay obstáculos especiales que tienen comportamiento especial, los spikes matan lo que choca con ellas, los ladder le permiten a Megaman trepar, y el boss door cuenta cuantos megamanes llegaron al final, si todos llegaron empieza el encuentro contra el jefe.

Descripcion de archivos y protocolos

El protocolo de comunicación entre cliente y servidor está documentado en el header CommunicationCodes.h. Dentro de este archivo está la estructura de los mensajes y además hay ejemplos de estos.

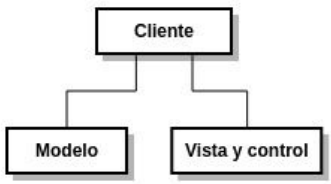
Cliente

Descripcion General

El cliente puede verse como una interfaz de conexión hacia el servidor. Básicamente cumple dos funciones, la de enviar los datos hacia el host y recibir y procesar los que este envía, para finalmente mostrarle al jugador el estado de la partida y permitirle interactuar.

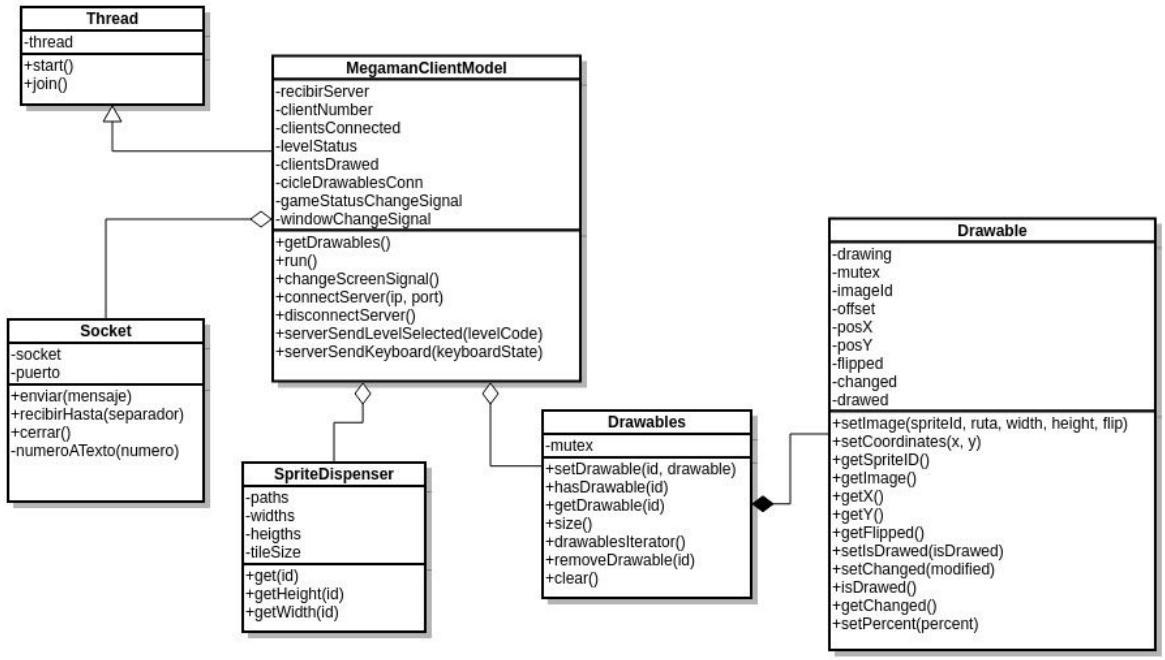
Clases y diagramas

La división de clases a grandes rasgos implica la separación entre el modelo por un lado y la vista con los controles por el otro.



- Modelo

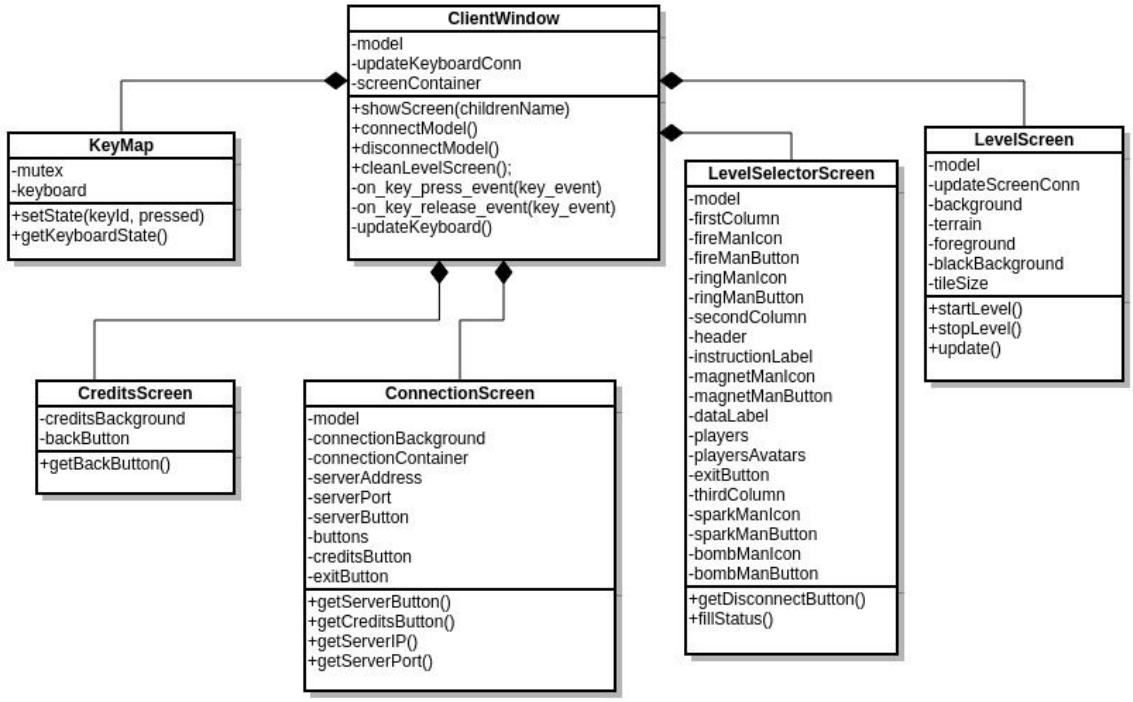
Primero vamos a ver el modelo, el cual maneja el estado del juego y proporciona las interfaces para poder acceder a estos datos como también para comunicarse con el servidor.



Como puede verse el modelo hereda de Thread, dado que corre en un hilo aparte de la interfaz gráfica para no interrumpirla, actualizando sus datos con los mensajes que recibe continuamente del server en su método run(). Como se ve, también incluye el socket que utiliza para la comunicación.
Además, posee un contenedor de Drawables, el cual simplemente contiene y administra a los elementos dibujables. Estos son el conjunto de imagen, posición y estado del elemento que se puede dibujar en pantalla, puede ser desde el personaje del jugador, con su correspondiente sprite actual y si debe ser vuelto a dibujar dado que cambio o un bloque del suelo asi como también los métodos necesarios para modificarlos o acceder a sus datos.

- Vista y controles

Dentro de la vista, se encuentra como contenedora principal la ventana del tipo Gtk::Window que nos permite utilizar su cola de eventos



Se omitieron las herencias de las pantallas para favorecer la lectura del diagrama.
Empezando por ClientWindow, que es un Gtk::Window, vemos que contiene una referencia al modelo, esto es útil para poder pasarle las acciones que sea necesario enviar al servidor, esta referencia se pasa a las demás pantallas por si estas necesitaran también comunicar algo, evitando ir pasando el mensaje innecesariamente.
Las pantallas son en sí widgets contenedores, que mediante el screenContainer del tipo Gtk::Stack se muestra uno solo en la ventana. Esto a pantalla completa siempre, modo típico de juegos en las computadoras. Este container se conecta a una señal que emite el modelo cada vez que quiere cambiar de pantalla por cambios en el server, lo que permite que el server controle en los clientes los inicios o finales de los niveles por ejemplo.
En el caso de CreditsScreen, LevelSelectorScreen y ConnectionScreen, son simplemente displays de los widgets que tienen, ya sea labels para mostrar texto, botones o campos donde el usuario puede ingresar datos como la dirección IP o el puerto. Estos campos se conectan a los métodos que deben ejecutar para cambiar de pantalla, conectar o desconectar el modelo al servidor o notificar a este que el jugador quiere elegir un nivel.
Pasando a la pantalla más compleja, LevelScreen, ésta hereda de Gtk::Fixed ya que posee tres capas superpuestas que actúan como el eje Z de la pantalla, siendo estas también Gtk::Fixed para poder dibujar libremente, superponiendo si es necesario. Además tiene un tamaño de tile estándar pero dinámico según la resolución y el fondo, y también la típica referencia al modelo y una conexion de sigc para llamar cada 15ms, buscando los 60fps, al método que actualiza lo que esta tiene dibujada. Este método itera el contenedor Drawables viendo los que fueron editados y dibujarlos o actualizarlos según corresponda. Este método y sus llamadas arrancan cuando esta pantalla se vuelve visible, para finalmente terminar de llamarse cuando lo opuesto ocurre, que además limpia el contenedor de dibujables.
Finalmente, la pantalla del cliente también tiene una clase KeyMap, esta contiene un mapa con las teclas que nos interesan observar y su estado. En cada evento de presionar o soltar

una tecla, esta clase es notificada cual es y actualiza su estado. Luego, cada 100ms, la ventana del juego informa por el modelo al servidor cual es el estado del teclado con el fin de que este responda correspondientemente.

Descripcion de archivos y protocolos

El protocolo de comunicación hacia el servidor está documentado en el header CommunicationCodes.h. Dentro de este archivo está la estructura de los mensajes y además hay ejemplos de estos.

Después, dado que el jugador no debería poder modificar los aspectos técnicos del juego, algunas configuraciones se encuentran centralizadas en MegamanBeginsConstants.h como el tamaño de la pantalla en tiles o los identificadores de las teclas apretadas.

Editor

Descripcion General

Aplicación gráfica para la construcción de escenarios que serán luego dados de alta en el servidor para ser usados como niveles.

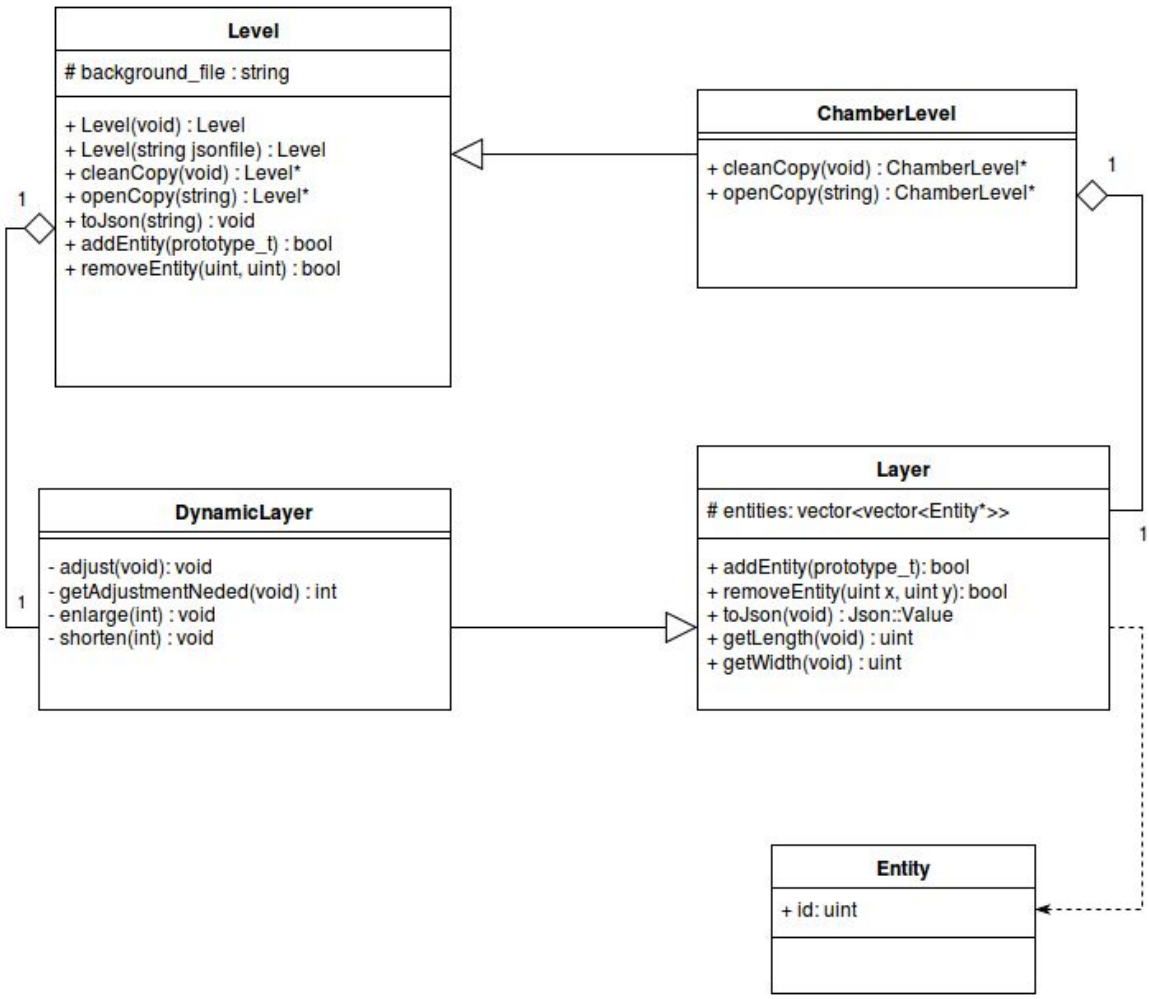
A su vez este módulo está funcionalmente separado en un modelo que encapsula la lógica de crear niveles, generarlos, y guardarlos, y una vista/control orientada a eventos que implementa la interfaz gráfica de la aplicación.

Clases

En el modelo, las clases más importantes son Level y Layer. Layer maneja el guardado y borrado de elementos en el nivel. Internamente consiste de una matriz de objetos de clase Entity, y los métodos más importantes de su API son los de agregar elementos (addEntity()), borrar elementos (removeEntity()) y presentarse en formato JSON (toJson()). Level encapsula a un Layer, pero agregándole métodos y atributos que no tienen directamente que ver con el guardado de elementos. Por ejemplo, toma cuenta del fondo de pantalla que utiliza el nivel, es capaz de generarse de un archivo JSON y guardarse en un archivo JSON, y también incluye chequeos de validez del nivel.

Existen dos clases de tipo Layer, que son la misma Layer y una clase hija DynamicLayer. Esta segunda permite que el escenario se agrande y achique dinámicamente cuando se le agregan o quitan elementos. Existen de la misma manera dos clases de tipo Level, que justamente se diferencian en el tipo de Layer que usan. Level usa DynamicLayer, mientras que ChamberLevel (para construcción de escenarios de jefe final) usa la clase madre Layer.

La clase Entity en este momento existe solamente por razones de extensibilidad. Si en un futuro se quiere agregar comportamiento o información a lo que es una entidad, pueden agregarse fácilmente desde ésta clase. Sin embargo, en su estado actual lo único que hace es guardar un identificador de qué es lo que se está guardando.



Por parte de la vista/control, la mayoría de las clases son simplemente derivados de las clases de gtkmm, pero con comportamiento extra para integrarlas al programa.

Dos clases que considero importante señalar son Workspace y WorkspaceEventManager. La primera hereda de Gtk::Fixed, y es básicamente la clase que interactúa con el modelo conteniendo un Level, y mostrándolo de forma adecuada. Si bien se eligió un Gtk::Fixed para implementar ésto, este Widget es bastante limitado en el manejo de señales ya que no recibe eventos. Para eso está WorkspaceEventManager, un Gtk::EventBox donde está contenido el Workspace. Toda la lógica correspondiente a la interacción del usuario con el Workspace está implementada en esa clase, a saber: las acciones de drag and drop, agregado de elementos con click, selección y borrado de elementos.

Descripcion de archivos y protocolos

Como se aludió anteriormente, los niveles se guardan en archivos de formato JSON, que pueden ser cargados por el servidor, o volver a ser cargados al editor para seguir siendo construidos.

Los IDs que identifican elementos son los mismos utilizados por el cliente y el servidor, y están especificados en el header entities.h. Los IDs a su vez están linkeados con los sprites que los representan en la clase SpriteDispenser, que es utilizada también por el cliente.

Manual de usuario

Instalación

Para la elaboración del manual de instalación se uso una maquina virtual corriendo un sistema operativo Linux (Ubuntu 14.10) recién instalado de 32 bits. Se decidió tomarlo como referencia para los requerimientos y las instrucciones. Por esta razón es posible encontrar diferencias en el proceso de instalación, ya que podría no ser necesario instalar alguna librería en caso que ya se encuentre en la plataforma.

Requerimientos de software

- Para correr tanto el servidor como el cliente o el editor, se debe contar con una distribución de linux compatible.
- También se requiere CMake y git si se desea compilar el fuente más actualizado desde el repositorio.
- Sistema operativo Linux (Ubuntu 14.10 como referencia)

Requerimientos de hardware

- Para ejecutar el cliente se requiere una pantalla con relación widescreen, de otra manera el juego no se verá correctamente.
- El usuario además debe contar con teclado o algún dispositivo de entrada similar.
- Procesador trabajando con1 nucleo, 1.9Ghz
- Conexión a una red local para el multijugador.

Proceso de instalación

Primero se debe compilar el programa. Para esto es necesario correr el comando cmake desde el directorio de instalación. Si no se tiene cmake se puede instalar siguiendo las instrucciones de la pagina https://cmake.org/install/ . Puede que sea necesario instalar compilador de c++ como g++ tambien.(escribir ”g++ --version” y por consola se obtiene instrucciones).

Instalar gtkmm 3.0 con el comando:
“sudo apt-get install gtkmm-3.0”

Una vez instalado cmake se instala glog. Para esto ir a la carpeta glog y ejecutar los siguientes comandos:
“./configure”
“cmake CMakeLists.txt”
“make”
“sudo make install”

Si no existe carpeta build en el directorio de instalación, crear una, e ingresar en ella. Desde allí ejecutar los comandos:
“cmake ..”
“make”

Configuración

Modelo del servidor

Para configurar los parametros y formas de las características del modelo, se debe modificar el archivo config.json localizado en [directorio de instalación] /server/Model/config.

“world”:

```
"world":
{
        "windowWidth":      27.0,
        "windowHeight":     14.0,
        "gravity":          -40.0,
"width" :           95.0,
        "height" :          95.0,
        "steps/second":     24.0
}
```

- window width y height son el alto y ancho de la pantalla del mundo, cosas fuera de esto no se dibujan, y mueren.
- Gravity es la intensidad de la gravedad en eje Y en m/s. Afecta velocidad de caída y altura de los saltos.
- Width y height son tamaños utilizados para el cálculo de coordenas a enviar al cliente.
- Steps/second es la cantidad de iteraciones del mundo por segundo. Este parametro es proporcional a la fluidez e inversamente proporcional a la performance del juego.

“WindowBoundaries”: contiene las figuras que serán los límites de la pantalla que evitan que megaman salga de pantalla cuando no se cumplen las condiciones. Deben coincidir con el alto y ancho de la pantalla marcados en world para una buena experiencia. No poner borde inferior si se quiere que megaman caiga por precipicios.

“10”**:

```
"1040": //Jumping Sniper
  {
        "id":               1040,
        "weaponId":         2006,
        "JSpeed" :          15.0,
        "life":             2.0,
        "jumps":            true,
        "jumpSpriteId":   1032,
        "jumpFreq":         2.0,
        "jumpSensor":
        {
                "type":     1,
                "x":        0.0,
                "y":        -0.5,
                "width":    0.25,
                "height":   0.1
        },
        "shapes":
        [
                ...
        ]
```

```
},
```

Son las configuraciones de los distintos enemigos. Estas son:

- Id es el sprite que identifica al enemigo
- WeaponId es el id del arma que usa, la cual se inicializa con el json de dicho id.
- Jspeed es la vel vertical del salto
- Life es la vida, la mayoría de los disparos bajan de a uno
- Jumps es true si el personaje salta. En dicho caso debe tener jump sensor, jump sprite, y frecuencia, sino no
- Jumpfreq es la cantidad de segundos entre saltos.
- Jump sprite es el id del estado de salto, para dibujar dicho sprite.
- Jump sensor tiene las formas física del sensor que detecta superficies para saber si puede saltar.
- Shapes son las formas físicas que representan al personaje. Más detalles sobre esto al final.

“1*”**: configuraciones de los jefes.

```
"1100": //Bombman
{
	"id":				1102,
	"weaponId":			2101,
	"JSpeed" :			22.0,
	"HSpeed":			2,
	"jumping time": 0.05,
	"attacking time": 0.05,
	"walking time": 0.05,
	"punching damage": 3,
	"life":				20.0,
	"jumps":		true,
	"jumpSpriteId":		1104,
	"jumpFreq":			2.0,
	"jumpSensor":
	{
		"type":			1,
		"x":			0.0,
		"y":			-0.5,
		"width":	0.25,
		"height":	0.1
	},
	"shapes":
	[
		{
			"type":			1,
			"width" :	0.4,
			"height" :	0.5,
			"X":			0.0,
			"Y":			0.0
		}
	]
},
```

Configuraciones de los jefes. Tiene todas las de los enemigos más:

- jumping time-attacking time-walking time son los tiempos que durará el jefe en cada uno de estos estados. Este tiempo no está en segundos, pero el valor 0.05 es aproximadamente 5 segundos. (segun el clock)

- punching damage es el danio que causa al estar megaman muy cerca.

“Megaman”:

```
"megaman":
  {
        "weaponId":         2004,
        "HSpeed" :          5.0,
        "ClimbSpeed":   3.0,
        "climbSpriteId":9010,
        "JSpeed" :          20.0,
        "inmuneTime":   0.5,
        "life":             30.0,
        "jumpSpriteId":   9005,
        "jumpSensor":
        {
              ...
        },
        "shapes":
        [
              ..
        ]
  },
```

Configuraciones de megaman. Tiene todas las propiedades de un personaje, pero además:

- Hspeed es la velocidad horizontal en m/s.
- climb speed velocidad vertical al trepar en m/s.
- Climb sprite id el id del estado de trepando.
- inmuneTime es el tiempo que es inmune a ataques y muerte luego de haber sido atacado, o haber muerto.
-

“Wall”: configuración física de los tiles comunes (escaleras,paredes,plataformas).
“spikes”: configuración física de los pinches (tienen forma triangular).
“2*”**:

```
"2004": //Megaman Beam
  {
        "damage":           1,
        "cooldown":         0.25,
        "bulletId":         2004,
        "speed":
        {
              "x":   8.0,
              "y":   0.0
        },
        "shapes":
        [
                    ….
        ]
  },
```

configuraciones de las armas.

- Shapes es la forma física que tomara la bala.
- Speed será la velocidad básica en m/s con la que saldrá la bala.
- Damage el danio que causara
- Cooldown el tiempo mínimo que debe pasar entre disparos para que le arma funcione de nuevo.
- bullet Id es el id de sprite que identifica la bala en el cliente.

“Drops”:

```
"drops":
  [
        {
              "id":       3005,
              "chance":   0.01
        },
        {
              "id":       3004,
              "chance":   0.10
        },
        {
              "id":       3003,
              "chance":   0.05
        },
        ...
  ],
```

Ids y chances de que al morir un enemigo haga drop a un ítem. Las chances no deben sumar mas de 1, y no deben haber items repetidos.

“300*”:

```
"3001": //Big Energy
  {
        "healAmount":   6,
        "shapes":
        [
              ...
        ]
  },
```

Configuraciones de los ítems. Tienen las formas y la cantidad o potencia de dichos items.

“Shapes”:En varias de estas configuraciones está un sub-valor llamado “shapes” este posee el tipo de figura que representa físicamente, y los parametros correspondientes para armarla. Todos los objetos con representacion fisica tienen este valor. Los tipos son:

1. Caja: width y height son el medio ancho y medio alto de la caja, X e Y la posición de la figura respecto al centro del cuerpo
2. Circulo: radio es el radio del círculo, X e Y posición respecto al centro del cuerpo
3. Triángulo: tiene tres pares X e Y que representan los tres vértices del triángulo respecto al centro del cuerpo.
4. Borde: son líneas sin grosor entre los vértices X1-Y1 y X2-Y2.

Ej: caja de ancho 0.7, alto 0.8, centrada en el centro del cuerpo.

```
"shapes":
        [
              {
                    "type":         1,
                    "width" :   0.3,
                    "height" :  0.4,
                    "X":            0.0,
                    "Y":            0.0
```

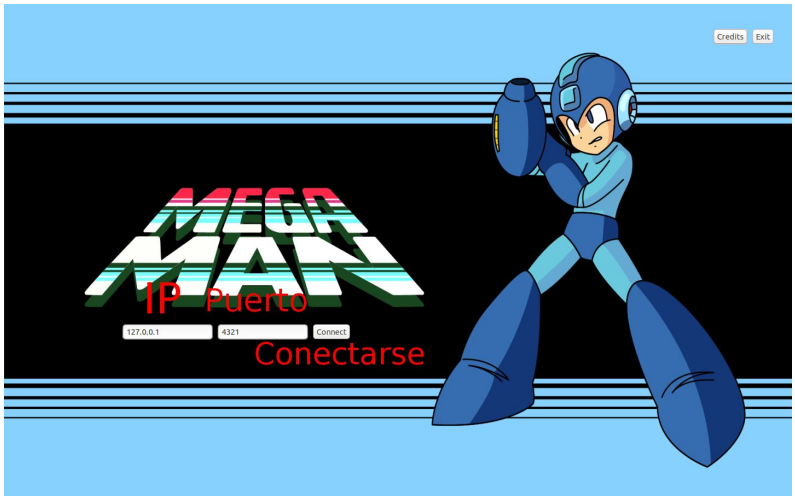
```
                }
        ]
```

Forma de uso

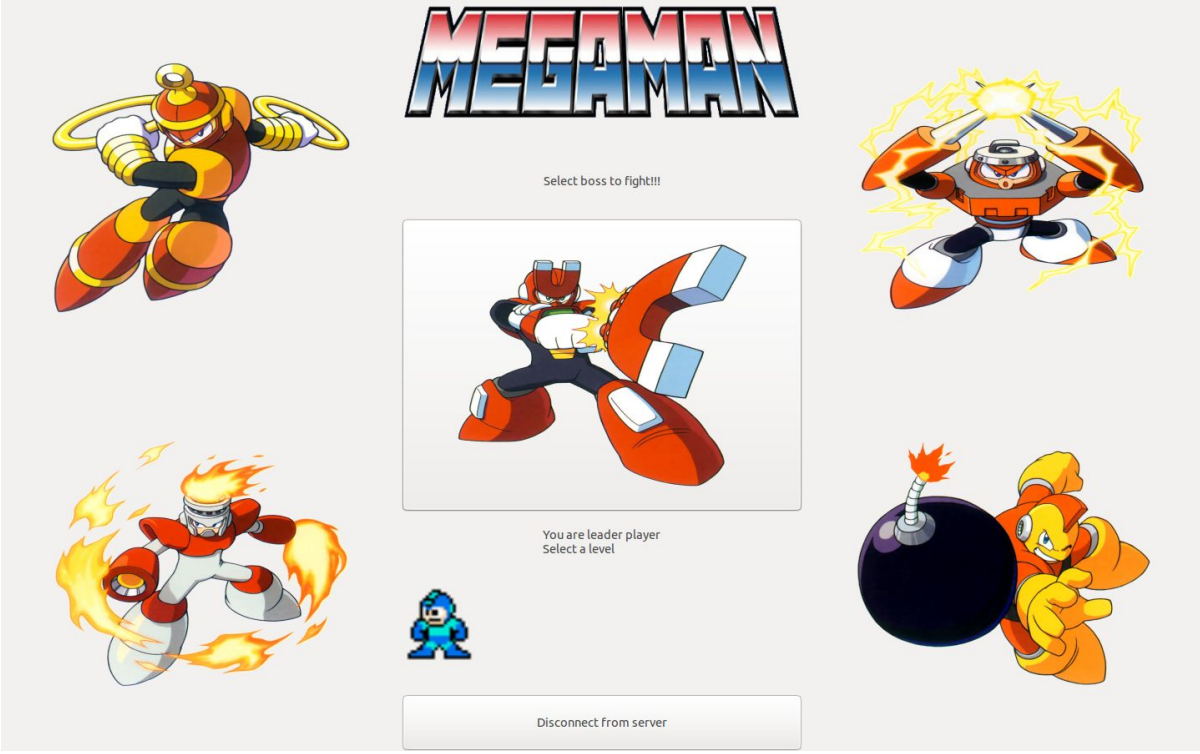
Cliente

El cliente simplemente se ejecuta mediante el comando que lo invoca. Si se está en la carpeta del ejecutable, alcanza con “./client“.

Una vez iniciado el programa, se puede ingresar la dirección IP, el puerto del cliente y darle al botón conectar.



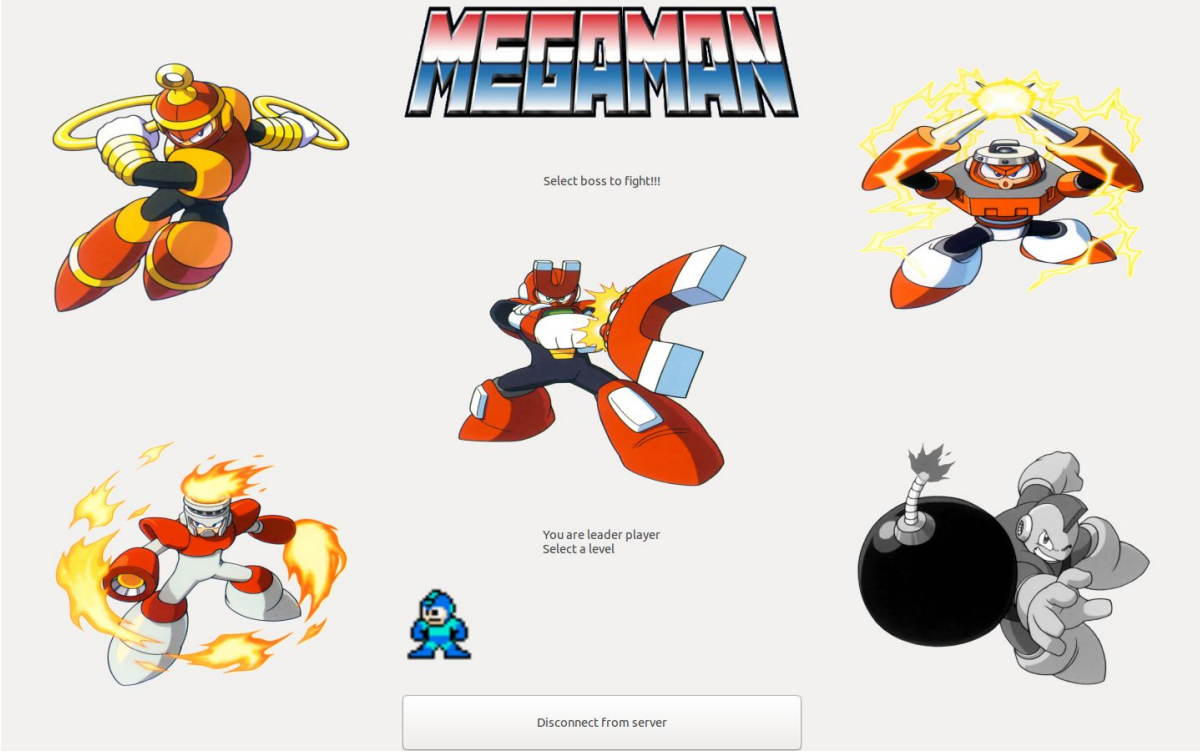
De lograrse la conexión se cambia automáticamente a la pantalla de selección del nivel, donde pueden seguir conectandose un maximo de 4 clientes hasta que el primero elija un jefe para derrotar.



Aquí se ve una captura de la pantalla de selección de nivel. Podemos ver en la parte central inferior una figura de megaman. En este caso, esta indica que hay un sólo jugador conectado. De haber más jugadores, se verán más de estas figuras. Por encima de esta vemos un mensaje que nos dice que somos el jugador líder. Esto quiere decir que estamos habilitados para elegir un nivel. Para hacerlo, se debe clickear el jefe del nivel que deseamos jugar. De no ser el jugador líder, hay que esperar a que el que lo sea elija un nivel. Por último, por debajo de la figura de megaman vemos el botón para abandonar el server. Esto adicionalmente cierra el juego.

Una vez elegido, se arranca el nivel correspondiente, finalizando cuando se derrota al jefe, todos los jugadores pierden o el primer jugador sale mediante la tecla ESC.
Para controlar al Megaman asignado durante el juego se utilizan las siguientes teclas:

- Flechas, para mover el megaman, saltando cuando es posible.
- Barra espaciadora, para disparar hacia el frente.
- Botones 1 a 5 para seleccionar el arma ganada de los jefes, de estar disponibles.
- ESC, para salir del nivel. Solo utilizable por el jugador 1.
  Al terminar un nivel derrotando al jefe, volvemos a la pantalla de selección de nivel.



Podemos apreciar que el jefe vencido ahora está de color gris, indicando que hemos ganado ese nivel.

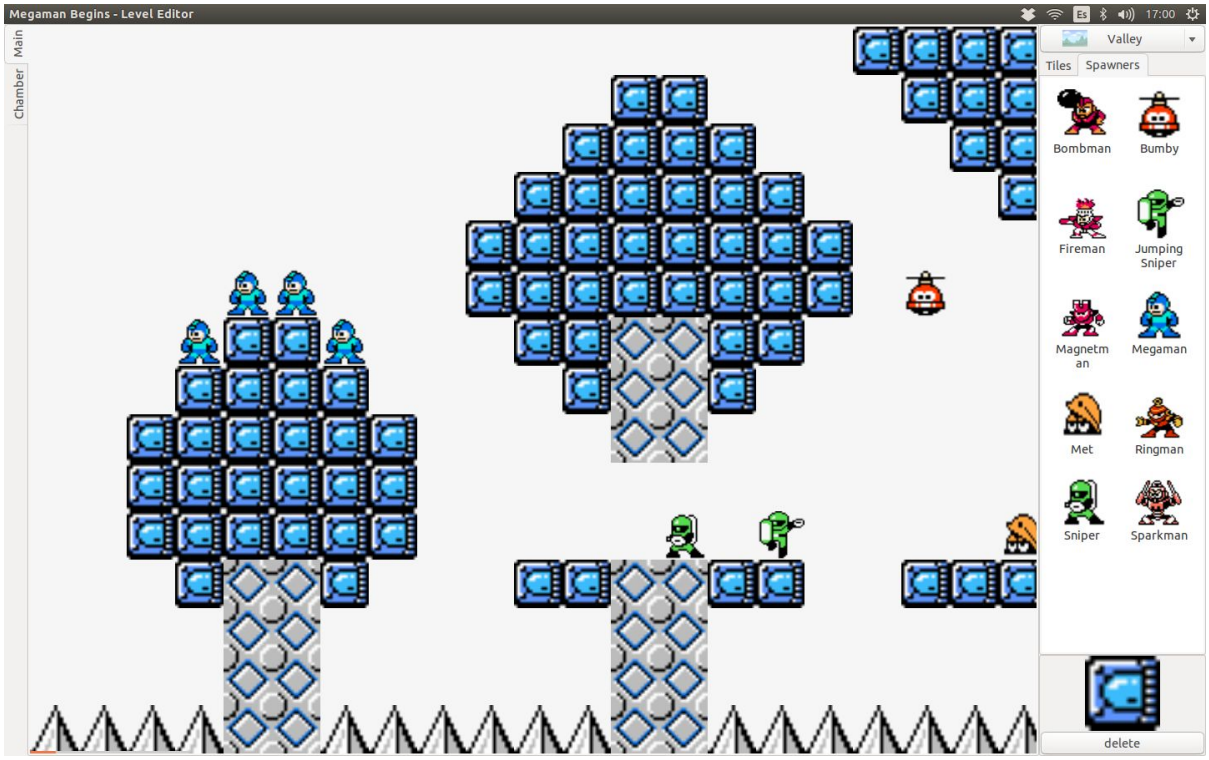
Servidor

El servidor se ejecuta con el comando “./server PUERTO” donde puerto es el puerto de conexión. Una vez iniciado, acepta conexiones hasta llegar a un máximo de 4 conexiones simultáneas. Ante la recepción del mensaje correspondiente del primer cliente conectado inicia el nivel seleccionado, y deja de aceptar conexiones hasta terminar el nivel, momento en el cual se retoma el proceso anterior (se pueden conectar y elegir nivel).
Durante la ejecución del nivel, se reciben los comandos correspondientes al código de KEY_STATE desde el cliente, los cuales ocasionan una acción del personaje Megaman correspondiente, o salen del nivel en caso de KEY_ESC. (Los códigos están en el archivo CommunicationCodes.h de la carpeta common)
En cualquier momento el ingreso por consola del caracter ‘q’ finaliza la ejecución del servidor.
No hay salidas por consola, pero en la carpeta de ejecución, en la carpeta /log/server/ se encuentran los logs de ejecución del servidor, que proveen información relevante, según el modo de ejecución.

Editor de niveles

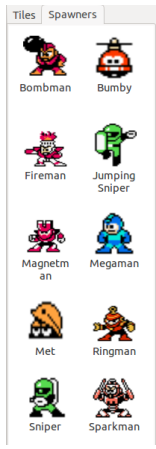
Para abrir la aplicación, puede ejecutarse su ejecutable por consola, o hacer doble click sobre el ícono de éste.

A continuación vemos una captura de pantalla de la ventana entera.



La aplicación consta de dos regiones básicas. El espacio de trabajo (workspace) a la izquierda, y la sección de herramientas a la derecha.

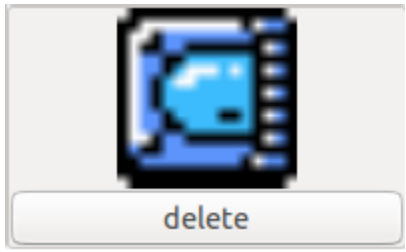
El workspace a su vez contiene dos solapas para el dibujado de tanto el nivel principal (main) como el escenario del jefe final (chamber). El escenario main se agranda y achica dinámicamente cuando se agregan o quitan elementos cerca del borde derecho del nivel. El escenario chamber en cambio siempre tiene un tamaño de 27x15 (la unidad siendo el ancho de un “tile”) ya que ese es el tamaño establecido para la pantalla en el juego.



A la derecha podemos ver el selector de elementos. Se puede apreciar que consta de dos solapas: Una para objetos inertes del nivel (“tiles”), y otra para definir el lugar donde aparecen los enemigos (“spawners”). Existen dos maneras de colocar elementos en el nivel. Pueden arrastrarse de entre los iconos del menú (de la solapa de “tiles” o “spawners” según corresponda), o pueden seleccionarse con un click y luego

seleccionar el lugar donde se quieran colocar con un click sobre el workspace.

Para quitar elementos, se seleccionan con un click desde el workspace. Una imágen del elemento seleccionado aparecerá en la esquina derecha inferior. Para eliminar, presionar “delete”, o la tecla “suprimir”.



También puede seleccionarse un fondo de pantalla para el nivel. Se puede seleccionar uno de entre los disponibles del selector que se encuentra en la esquina superior derecha de la ventana.

Para guardar, cargar, o crear un nivel nuevo, pueden seleccionarse las opciones desde la barra de menú, o utilizar los shortcuts ctrl+S, ctrl+O y ctrl+N respectivamente.