Introducción

Luego de la aprobación de los trabajos prácticos individuales se planteo el desarrollo grupal de un sistema de complejidad media-alta que incorporara todos los conceptos vistos durante la materia incluyendo el desarrollo grafico que no había sido evaluado anteriormente. Para esto se propuso la creación de un videojuego de plataformas, multijugador de hasta 4 jugadores del cual hablaremos en este informe.

Estructuración y manejo del proyecto

Para encarar el desarrollo del proyecto se propuso la división de tareas más natural que surgió de inmediato: un modulo sería el encargado de la gestión del juego multijugador, esto es, generaría el flujo de la aplicación desde el “login flow”, pasando por la selección de niveles y las condiciones de fin de partida hasta el control de los estados instante a instante de tiempo. El otro modulo sería el encargado del juego en sí, es decir, de la lógica del modelo que es la aplicación de un solo jugador si se quiere. De esta forma existirían muy pocas colisiones entre los integrantes y permitiría una mayor efectividad y libertad al momento de reemplazar porciones de código. Si bien no podía hablarse de completa independencia de los módulos dado a que existía una cantidad de código compartido si podía desarrollarse con cierta flexibilidad.

Así, pudo empezarse a trabajar desde el primer momento, en principio con ambos módulos separados, tratando de que esta fase durara lo menos posible, y luego ya en conjunto integrando con la mayor celeridad posible.

Para la primera semana del desarrollo se había propuesto la creación de un camino de datos dummy, que pretendía emular el flujo de datos entre el servidor y el cliente sin utilizar la red pero manteniendo un esquema que permitiera con facilidad la incorporación de esta característica. Además se tenía la intención de tener una estructura de clases para el modelo muy cercana a la que finalmente se usaría.

Finalmente por cuestiones de disponibilidad de los integrantes solo pudo cumplirse el segundo objetivo.

Luego además de terminar la estructura de red anteriormente dicha se debía comenzar a probar el modelo con lo que debía incorporarse una pequeña aplicación grafica que dibujara cajas en donde estaban los elementos, esto permitiría ir acercando el modelo a su versión final, arreglando los problemas que tuviera y agregando las características pedidas. Además, se debía incorporar a lo atrasado en la semana anterior la división de las aplicaciones en cliente-servidor y una comunicación rudimentaria entre ellas. Si bien se lograron cumplir estos objetivos diferentes bugs presentes hicieron que no pudieran ser presentados.

La tercera semana comenzaría con el arreglo de estos bugs y el establecimiento de los protocolos finales de comunicación entre el servidor y el cliente y el establecimiento de un modo de serialización e hidratación de datos a comunicar. Del lado del modelo se tenía que incorporar a los enemigos de nivel, las escaleras, el movimiento del personaje mediante las teclas, las zonas de aparición de los personajes y enemigos, la creación de los power ups, los distintos tipos de armas y la creación de la camara. Si bien pudo completarse el servidor, quedaron algunas cosas del modelo por completar como las escaleras.

En la cuarta semana, anterior a la pre entrega se debería seguir con el desarrollo del servidor, mas especificadamente con el flujo de unión a las partidas, la definición de las interfaces, y la carga de niveles dinámicamente. Del lado del cliente se tenía que incorporar los atrasado mas la carga de niveles desde un archivo .xml, la inteligencia artificial de los enemigos de nivel, los gráficos de todos los elementos incluyendo animaciones, la inteligencia artificial especial de los malos finales, las puertas de la cámara del malo y el ajuste del comportamiento general de todos los elementos para ser adecuado. Se pudieron cumplir con todas expectativas, se dio la primera prueba real del juego multijugador que se observo que funcionaba y se tuvieron algunos inconvenientes con el rendimiento general que fueron arreglados de inmediato.

Durante la quinta semana se limarían los aspectos ya implementados agregando algunas cosas que habían quedado tanto del lado del servidor, algunas refactorizaciones y la creación de un sistema más robusto para la transmisión de estados, como del lado del modelo, como la carga de los parámetros del juego desde un archivo de texto, arreglo

de algunos bugs o glitches y afinar la inteligencia artificial de los malos finales para ajustarlos a los criterios especificados.

Durante la sexta y séptima semana se siguió con la depuración del código, el ajuste de características y la confección parcial del informe, ya a paso más lento debido a la necesidad de cumplir con otras responsabilidades.

En la semana final se confecciono la versión final del informe, se agregaron las animaciones a los malos de nivel, se confeccionaron los 4 niveles restantes, se realizo una pequeña cantidad de testing para asegurar que no hubiera inestabilidades. Se ajustaron algunos problemas con el flujo del juego (transición entre niveles, etc.), se incorporo el log de eventos. Además se uso Valgrind para eliminar las lagunas de memoria del servidor y se mejoro la legibilidad del código mediante refactorizaciones y comentarios.

Si bien se han cumplido los objetivos propuestos en el trabajo práctico **quedan mejorables algunas cosas**, principalmente:

- Realizar un ajuste de la jugabilidad para presentar mayor desafío al usuario para esto deberían ajustarse los parámetros del archivo externo y además dotar a los malos finales de las habilidades presentes en el juego original, cosas que están muy por fuera del alcance de este trabajo.

- Hacer una distinción tajante entre la zona física de la cámara y la zona en pantalla de la cámara, es decir, que la resolución de la ventana no esté relacionada con el área física que muestra. Para esto debería implementarse un protocolo mucho más complejo que permita que muchos usuarios tengan distintas resoluciones de ventanas pero a la vez vean la misma cantidad de mundo para que sea justo. En nuestra versión solo esta soportada la resolución base de 800x600.

Inconvenientes durante el desarrollo

No se han tenido problemas mayores con la utilización de C++ que es quizás uno de los mayores desafíos del trabajo y de la materia ni con las demas herramientas tales como GTK, Cairo ni sockets o hilos. Los problemas encontrados más bien estuvieron relacionados a la falta de tiempo y la constante aparición de bugs y glitches dado el alto acoplamiento de un sistema de estas características.

El primer inconveniente encontrado fue la ausencia de un integrante, esto complico excesivamente las cosas, si bien fue recortado el alcance del trabajo practico no creemos que realmente el integrante faltante hubiera necesitado 8 semanas para realizar el editor de niveles, sino que hubiera utilizado el tiempo sobrante para apoyar en el testing, la corrección de bugs, la confeccion del informe y probablemente la ayuda con los recursos gráficos, esto hubiera alivianado mucho la carga dado que si se estaban programando las animaciones no debía interrumpirse para generar los cuadros buscando en internet.

El segundo problema fundamental fue el tiempo, si bien 8 semanas es más que suficiente, cuando se tienen otras materias el tiempo se hace escaso. Con lo que es fundamental la buena planificación y el cumplimiento de esta, poniéndose objetivos reales dentro de las posibilidades, cosa que no siempre sucedió.

Luego estuvo el problema del trabajo grupal en los momentos donde los trabajos debían cruzarse y alguno de los integrantes no había llegado a lo que se necesitaba con lo cual se debía dejar la tarea pendiente y continuar con otra para luego retomar esa.

La integración de los módulos también tuvo momentos conflictivos cuando al integrar se generaban bugs, por suerte solo fueron pocos momentos y la mayoría de las veces todo estaba bien, en parte debido a que la disponibilidad horaria de los integrantes hacia que nunca ambos trabajaran a la vez y entonces no había colisiones en los aportes al repositorio.

Además se tuvieron problemas técnicos ya que un integrante no instaló Ubuntu en una partición sino en un pendrive, luego en una máquina virtual, y en ninguno de los medios funcionaba de forma tal que el juego corriera con los FPS necesarios. Este integrante sólo instaló ubuntu faltando 10 días para la entrega del TP. Así, era imposible hacer que la aplicación funcionara eficientemente en un ambiente en el que siempre funcionaría mal. Queda como recomendación para los profesores que les

insistan mucho a los alumnos de siguientes cuatrimestres que instalen linux de forma tal que las aplicaciones gráficas medianamente exigentes del TP final funcionen bien.

Para las correcciones se llevó una netbook del plan conectar igualdad, que no tenía aceleración gráfica, con lo que se acentuaba el problema comentado anteriormente.

Herramientas

Se utilizaron varias herramientas para apoyar el desarrollo. Probablemente la más importante sea GitHub que se utilizo como sistema de control de versiones la cual facilito la comunicación entre integrantes dado que simplemente haciendo pull se tenía la última versión y esta además incluía el feed dado por el otro integrante lo cual es my importante, mantener la comunicación.

Para la compilación se utilizo un makefile especial, ajustado a los requerimientos del proyecto, que utiliza g++ como compilador para generar los diferentes ejecutables.

Como editor de interfaz grafica se utilizaron dos, cada integrante utilizo el que más le conviniera o le resultara familiar. En particular fueron Gedit y Geany. Y para el debug se utilizo gdb ya que proporcionaba de la manera más rápida y simple las pocas funciones necesitadas.

Para la obtención de los gráficos se utilizaron tres herramientas graficas. Gimp, Inkscape y el editor Online-Image-Editor disponible en http://www.online-image-editor.com/.
Para la confección de este informe se utilizo el editor Word y openoffice.

Conclusiones

Fue un trabajo practico difícil de llevar, duro y complejo pero al llegar al final y ver el resultado claramente se ve que se aprendieron algunos conceptos tanto generales de programación como de trabajo. Lo único que se puede acotar es la extrema dependencia del trabajo y de la materia en general del sistema operativo Linux dado que no es una

plataforma comercialmente viable ni masiva, es decir, que este curso con este TP no prepara para el desarrollo de videojuegos comerciales (si este fuera el objetivo) y además para peor este juego no podrá ser mostrado al 90% de personas interesadas en verlo.

Documentación técnica

Requerimientos de software

Los requisitos para el **desarrollo del programa** son:

- Ubuntu 12
- Gedit u otro IDE.
- Box2D (ya viene incluido en el proyecto base)
- Gtkmm (se instala con **sudo apt-get install libgtkmm-3.0-dev**)

Para la **compilación** se utilizan:

- ./compilar.sh: hace una compilación desde cero del cliente como si fuera la primera vez.

- ./compilar_s.sh: compila desde cero el servidor como si fuera la primera vez.
- Make posta: recompila el cliente solo utilizando los archivos que fueron modificados y los binarios existentes (puede ocasionar errores si se modificaron .h)
- Make –f s posta: recompila el servidor utilizando solo los archivos modificados y los binarios existentes, puede ocasionar los mismos problemas que make posta.

//Pone que archivos genera.

// Pone el Log

Para **depurar el programa** se utiliza gdb escribiendo por consola ‘gdb ./holi’ o ‘gdb ./serv’ dependiendo si se quiere debuggear el cliente o el servidor. Luego se escribe ‘r’ dentro de gdb para correr el programa. Si hay algún problema se escribe ‘backtrace’ para ver el stack de las llamadas. Para salir se escribe ‘q’ y luego ‘y’ cuando nos pregunte.

Para **probar el programa** simplemente se debe correr un solo servidor escribiendo por consola ‘./serv’ y tantos clientes como se desee con ‘./holi’.

Descripción general

Como se comento anteriormente el programa está dividido en dos grandes bloques o módulos, si bien no puede hablarse de modularización en el sentido de independencia y circunscripción de responsabilidades si en cuanto a funcionalidad definida, el modulo net implementa el flujo de juego online con las pantallas de selección y la interface que ven los jugadores y el otro implementa el juego offline con los gráficos de pantalla.

A nivel general y técnico pueden comentarse varias cosas y algunas decisiones de diseño. Una es que el juego desde el inicio estaba pensado

para poder compilarse bajo Linux y Windows utilizado algunas macros se podía generar códigos particulares para uno u otro sistema pero finalmente fue descartada la idea por no conseguir una biblioteca grafica que fuera CrossPlatform o que almenos fuera de mediana simple instalación en ambos SO. Hubiera tenido mayor valor si la idea prosperara.

La idea de juego online se encaro de la forma más simple posible y menos costosa siempre hablando de los recursos de red, en este punto es donde nos alejamos ligeramente del enunciado dotando a cada cliente de un sistema de juego completo de procesamiento en lugar de que los clientes fueran simples ventanas de visualización y captura de teclas, pasando de un juego de streaming a uno completo. Claramente esto no vino lejos de los problemas pero si produjo además un mecanismo de predicción que solo necesita actualización parcial por parte del servidor y eventualmente actualización total.

Para la física se utilizo el motor Box2D montado sobre nuestro sistema de clases para así encapsularlo. De esta forma se mantiene independiente del motor de física y aun así pueden usarse todas sus características y además extenderlas y realizar optimizaciones cuando sea necesario.

Para lo gráficos se utilizo un patrón denominado “Visitor” en el que es menester de cada objeto saber cuál es su forma, su dibujo, pero para dibujarse utiliza los métodos de otra clase. De esta forma se mantiene el encapsulamiento de la API grafica y se obtiene la menor redundancia en este sentido, a diferencia de otros patrones que segmentan mas el código pero generan demasiado overhead, por ejemplo, MVC.

Se creó un pequeño sistema rudimentario de animación que cumple con los requerimientos necesitados por el juego, ni más ni menos.

En la siguiente sección se describen con detalle ambos módulos.

Modulo offline

Se encarga de proporcionar la jugabilidad. Es, de hecho, un juego que puede ser jugado en single player utilizando el modo debug. No es dependiente del multiplayer sino que este último lo manipula para poder tener el mismo estado en cada cliente pero mediante unas pocas líneas

se puede transformar en un juego de punta a punta single player. Para activar el modo debug se debe definir la constante DEBUG en ‘Debug.h’ en la carpeta ‘src/net’.

Esta encabezado por la clase Mundo que es la puerta de entrada y tiene toda la lógica del juego.

A continuación se presenta un diagrama reducido de clases explicando cada una de las clases principales. Solo se muestran las relaciones entre clases y, si se considera necesario, se detalla en la descripción de esa clase los métodos y atributos.

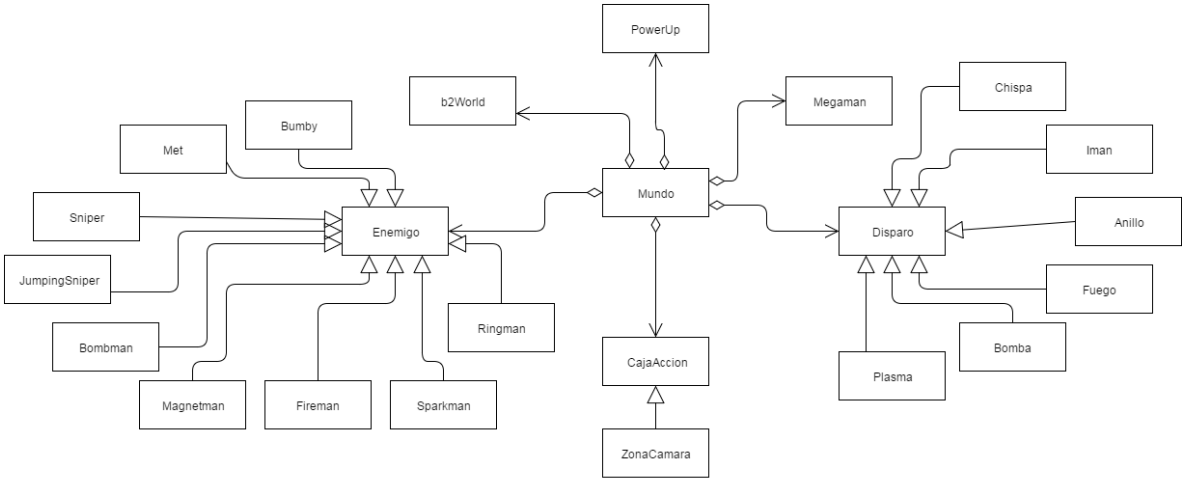


Ilustración - Diagrama UML de clases parcial

Descripción de funcionalidades desde las clases más esenciales hasta las más compuestas.

La **clase Cuerpo,** representa cualquier cosa que tenga presencia física en el mundo. Hace uso internamente del motor físico Box2D pero esto es totalmente opaco al usuario. Tiene su interface orientada a sus propiedades física, es decir, es posible obtener y modificar sus parámetros físicos clásicos como la velocidad, su posición o aplicarle un impulso. Posee dos métodos que deben ser aclarados para no haber confusión. El primero es “agregarCuerpoInmaterial”, permite adosar al cuerpo otro cuerpo de características físicas definidas, por ejemplo, un detector de extremos para saber cuándo se está frente a una pared. La otra esa “tipoCuerpo” en varios escenarios (Callbacks de Box2D mayormente) es necesario saber el tipo de cuerpo para obtener el resultado deseado, por lo que, cada clase que herede de Cuerpo deberá

redefinir este parámetro. El 90% de los objetos heredan de Cuerpo o de sus hijos.

La **clase Actualizable,** es una de las interfaces más importantes. Las clases derivadas de esta son aquellas que pertenecen al ciclo de simulación y, como tales, se actualizan en cada llamada a 'actualizar' pasándole el diferencial de tiempo.

La **clase Saltador,** simplemente define si una entidad puede o no saltar.

La **clase Entidad,** representa todas aquellas entidades vivas, en este caso, son enemigos y megamanes. Define los parámetros de energía y determina como es atacado un personaje y como actúa al ser dañado, y al morir. Las primitivas principales son "virtualo void atacado(uint daño, Disparo *disparo)" que provee un comportamiento por defecto a todas las entidades hijas y "virtual void alMorir()" que define como actua al morir una unidad ya que por ejemplo megaman no debe ser eliminado al morir pero si los enemigos.

La **clase Enemigo,** de esta heredan todas las entidades enemigas ya que provee mecanismos para el manejo de la inteligencia artificial, define las acciones que son posibles para los enemigos, tales como saltar, cubrirse, correr, virar, mirar a los lados, disparar; y provee la ruleta de powerUps si se setea al crearse. Las principales primitivas pueden verse en el código, pero nos gustaría resaltar una que ofrece funcionalidad para seguir expandiendo la clase, la primitiva "real numeroAleatorio(real desde, real hasta)", esta es la base para la IA de todos los personajes no jugables, desde el Met hasta Fireman. De esta forma se puede simular la voluntad de elección de la forma necesaria para este juego.

Las **clases derivadas de Enemigo,** estas clases permiten materializar al enemigo dándole forma visual y dándole comportamiento y, así, personalidad. Quizá haga falta mencionar como es que funciona este mecanismo, es la clásica maquina de estados finitos que se usa en cualquier videojuego solo que menos compleja. Se implementa dentro de la primitiva actualizar. A lo largo del juego puede verse la cantidad de personalidades distintas que pueden obtenerse solo con una maquina de estados finitos y un generados pseudoaleatorio de números.

La **clase PowerUp y sus derivadas,** sencillamente son los objetos que megaman puede agarrar, de manera que al colisionar con ellos, se les llama el método “void aumentar(Megaman* megaman)” y este modifica en algún aspecto a megaman, ya sea recargándole energía, plasma, o dotándolo de nuevas armas. Al crear un powerUp debe asignársele una probabilidad de aparición de manera que pueda haber objetos raros y objetos comunes.

La **clase CajaAccion y sus derivadas,** son objetos muy especiales que no tienen una presencia visible en el mundo pero que ofrecen funcionalidades necesarias como las implementadas y que se describen a continuación. Las cajasAccion tienen su efecto mayormente al colisionar con y solamente con megaman y al hacerlo se agrega una tarea diferida en el mundo para que luego interactúe con este. El derivado ZonaMortal es una caja de acción que al tocarla mata instantáneamente a megaman. La ZonaTransporte al estar todos los megamanes adentro los transporta a una determinada posición. La ZonaGuardado una vez que todos los megamanes están adentro genera un Checkpoint de manera que si mueren se regeneran (si tuvieran vidas) en esa zona. La ZonaCerradura al estar todos los megamanes adentro cierra una puerta determinada por su ID en el .xml del nivel. Las CajaSpawn, simplemente reciben un ID y generan un enemigo en ese lugar, el ID siempre será el mismo con lo que se considera que el monstruo es el mismo.

La **clase Callback,** es una clase especial dedicada a manejar las colisiones del Box2D ya que dice que hacer en cada caso y dependiendo del fin para el que fue creado y el tipo de objeto. De esta forma se pueden definir varios “ColisionHandlers” en lugar de uno solo que soporta Box2D.

La **Clase ZonaCamara,** derivada de CajaAccion, merece un apartado. Es la que controla el recinto dentro del cual es posible moverse. Si algún jugador se va fuera de este recinto es reinsertado en la posición interna más cercana con lo que todos los jugadores siempre están en pantalla. Esta está bloqueada en el eje Y por decisiones de diseño pero podría extenderse a ambas dimensiones sin complicaciones. Si se quiere dibujar en pantalla es necesario primero preguntar por las dimensiones de esta zona y además por su posición.

La **Clase Megaman,** hereda de Entidad. Es uno de los pilares fundamentales y fue desarrollada a lo largo de 2 meses para ir

incorporando los requerimientos del videojuego. Tiene primitivas similares a las de Enemigo pero incorpora muchas más, como por ejemplo las que permiten que megaman se agarre de las escalera, las suba, las baje, seleccionar que arma usar, etc. También hereda de la clase Animado con lo que internamente tiene puntos de cambio de animación para reflejar el estado de megaman que es por lejos el cuerpo que mayor cantidad de estados tiene.

Y finalmente la clase principal, la clase que nuclea a todas las clases del modelo, la clase que modera el juego, la **clase Mundo.** Tiene casi 700 líneas siendo la más extensa de todas debido a la gran cantidad de responsabilidades que tiene. Utiliza como base la clase b2World de Box2D para la simulación del mundo físico. Es la encargada de cargar desde el archivo .xml el mundo en el que se va a jugar. Permite preguntar desde afuera cual es el estado del mundo actual para poder así tomar decisiones como terminar el juego o avanzar al siguiente mundo.

Además soporta una gran cantidad de consultas sobre elementos del mundo, tales como el megaman mas cercano a un punto, el enemigo más cercano, consultar entidades por ID, dañar todas las entidades de una zona, eliminar elementos, etc. Además permite agregar tareas diferidas, lo que permite ejecutarlas en un momento donde sea seguro ya que una gran cantidad de estas pueden estar relacionadas al motor Box2D y deben ser realizadas fuera del ciclo “step”.

Tiene otras primitivas que son fundamentales para el rendimiento como pueden ser, “limpiar” que elimina todas las entidades fuera de un determinado radio; “obtenerElementosCamara” que devuelve una lista con todos los elementos que están dentro de la zona de cámara.

El **archivo de definiciones,** se encuentra en el directorio raíz junto a los ejecutables, es el configuracion.conf y dentro de el están todas aquellas cosas que tiene sentido que puedan ser ajustadas sin recompilar, como ser velocidades y tamaño de cosas. Se implemento con un objeto global especial dentro del juego que permite cargar dinámicamente estos valores. Luego es los remapea como constantes del preprocesador para enmascararlas y mejorar la legibilidad del código.

Descripción de archivos y protocolo

Coso. El dialecto XML que generaste o como se le digan a esas cosas

Modulo online(net)

La tarea del módulo online cumple un papel análogo al de una infraestructura. El módulo determina la interacción entre la ventana y el servidor, y el servidor y los clientes.

La idea central al diseñar el módulo fue que el Mundo, es decir, el estado del módulo offline, se mantuviera sincronizado tanto en el servidor como en el cliente. Aunque de ambos lados se corre la simulacón física, el Mundo del servidor siempre sobreescribe el Mundo del cliente.

El módulo online tiene varios submódulos:

- sockets
- servidor
- cliente
- snapshots

A continuación se trata cada uno por separado:

Modulo sockets

El módulo sockets provee tanto bindings de las structs de C como clases que se ocupan de la emisión y recepción de datos, aislando de esta forma al código en Servidor y en Cliente de los detalles del protocolo.

Clases:

Socket: El binding de C, aunque provee algunas funcionalidades extras.

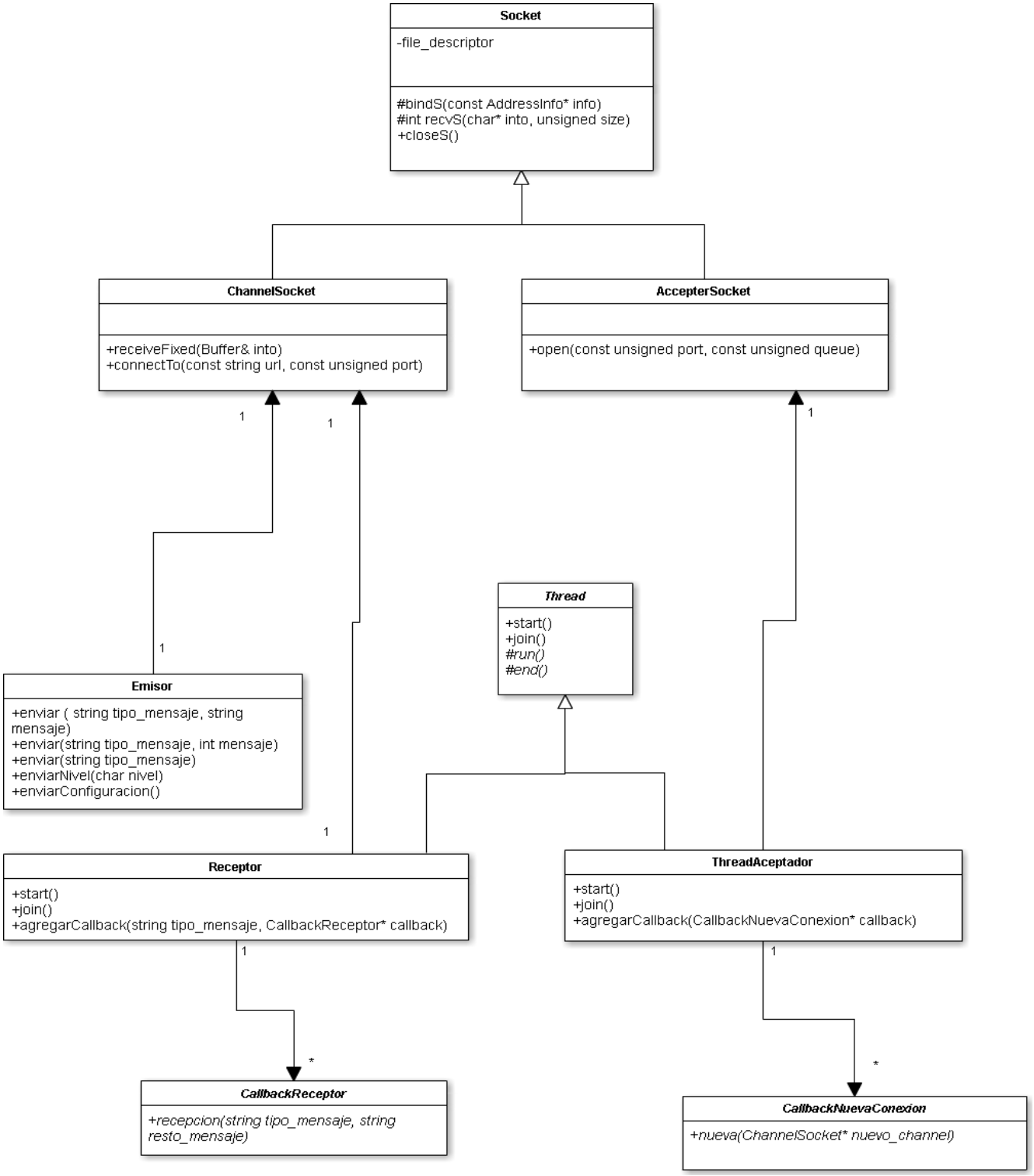
No se usan Sockets, sino sus derivadas: **AccepterSocket** y **ChannelSocket**. AccepterSocket representa el socket que acepta conexiones (creando ChannelSockets), y ChannelSocket el que posibilita la comunicación.

Thread aceptador: Es un Thread que espera conexiones de un AccepterSocket y cuando las recibe llama el **CallbackNuevaConexion** que se le pase.

Receptor: Es un hilo, al recibir mensajes de de un ChannelSocket llama al CallbackReceptor correspondiente. Se puede registrar un solo CallbackReceptor a cada tipo de mensaje, y un único CallbackReceptor además puede recibir varios tipos de mensaje.

Emisor: Emite duplas *(tipo de mensaje, mensaje)* o elementos *(tipo de mensaje)*. Agrega funcionalidad a los ChannelSocket. Es casi un decorator de ChannelSocket.

Diagramas UML



Descripción de archivos y protocolo

El protocolo online se centra en la emisión y recepción tanto de de pares *(tipo de mensaje, mensaje)* como de elementos *(tipo de mensaje)*, este último se aplica para mensajes más simples. La dupla se emite de la siguiente manera:

1. Emisión del tipo de mensaje (3 bytes)
2. Emisión del largo del mensaje(4 bytes, en caso de no emitirse un mensaje se emite 0)
3. Emisión del mensaje como un string. En caso de que el largo del mensaje sea 0, no se lo emite.

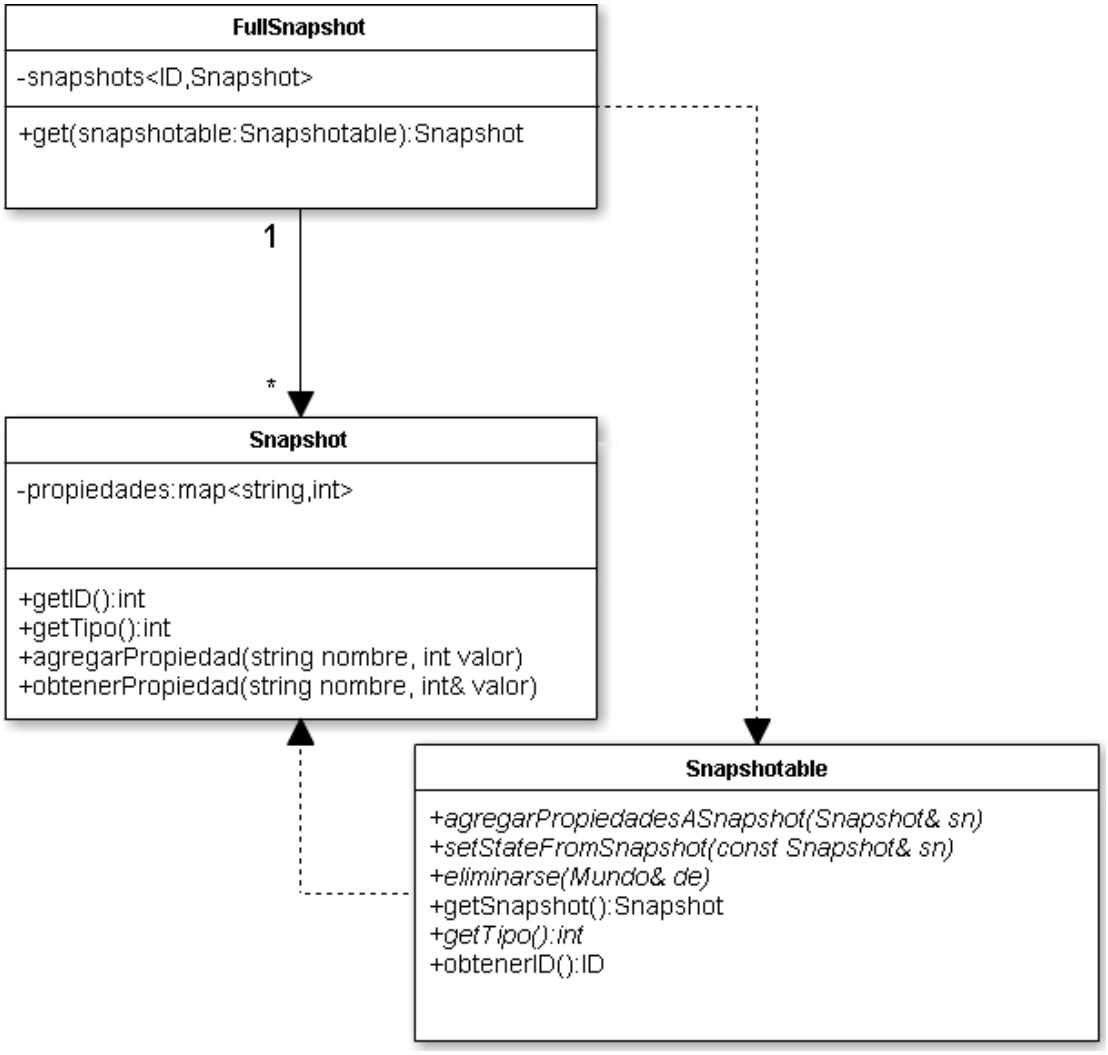
En caso de que se envíe un número, éste deberá enviarse como string. Es decir, el número 4287 se envía como el string “4287”.

No se aplica ningún tipo de compresión ni codificación.

Modulo snapshots

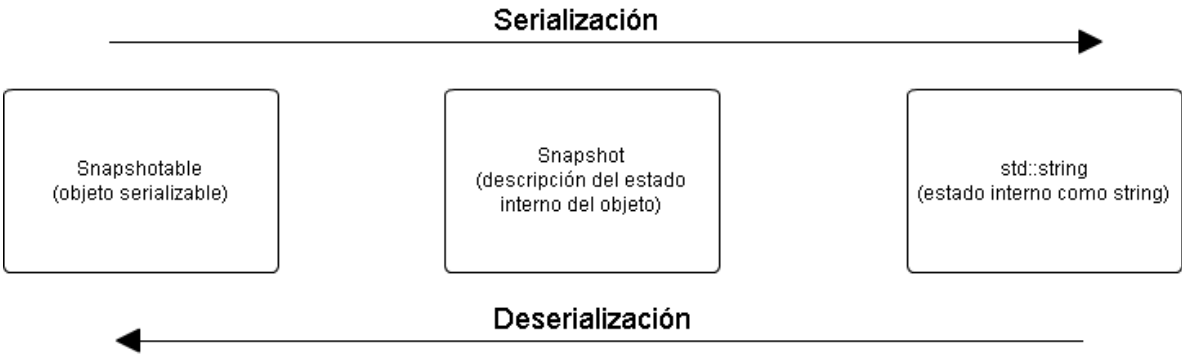
Un **Snapshot** es un estado de un objeto **Snapshotable**. Un **FullSnapshot** es un conjnto de snapshots, es decir, el estado del Mundo.

Diagramas UML



Descripción de archivos y protocolo

Los Snapshot pueden serializarse a strings. Así, un FullSnapshot se serializa a un conjunto de strings. De esta manera, para serializar un Snapshotable, primero debe generarse un Snapshot que lo represente, y luego, a partir de ese Snapshot, se genera un string. El proceso inverso se sigue para deserializarlos.



Todos los Snapshotables deben tener:

- un tipo (representa la clase del objeto)
- un ID (único de cada objeto)

Estos requerimientos se evidencian como métodos abstractos de Snapshotable. Los snapshotables almacenan duplas (nombre de propiedad, valor de la propiedad). El nombre de la propiedad es representado como un string, su valor siempre se representa como un int. Para lograr esto, con todos los tipos que es posible, se hace un casteo. Para los apuntadores a otros Snapshotables, se utiliza su ID. Los números de punto flotante se multiplican por un número grande al serializarse, y se dividen por ese mismo número al deserializarse.

Para transformar cada Snapshot a string y viceversa, se aprovechan los stringstreams. Cada elemento de las duplas (nombre, valor) se ingresa al stream en orden, es decir, primero se ingresa el nombre y después el valor. Para la lectura, se ejecuta la tarea inversa. El orden de las duplas no importa. No se puede repetir el nombre de las propiedades para cada objeto.

Para la captura del estado de los snapshotables se utilizan macros, stringification y pasajes por referencia, permitiendo que el código sea limpio. Así, el código de serialización y deserializacion se ve como:

serialización:

SN_AGREGAR_PROPIEDAD(vidas)

SN_AGREGAR_PROPIEDAD(velocidad)

...

deserialización:

SN_OBTENER_PROPIEDAD(vidas)

SN_OBTENER_PROPIEDAD(velocidad)

...

Nótese que no sólo se aprovecha la serialización para Cuerpos del Mundo, también se utiliza para el objeto EstadisticasMundo. Se decidió no utilizarlo para el envío de la configuración, aunque habría sido posible.

El proceso de inyección de FullSnapshots en Mundo lo que hace es :

1. Remueve del Mundo los objetos que no se encuentren en el Fullsnapshot.
2. A todos los objetos del Mundo que se encuentran en el FullSnapshot, se les inyecta el Snapshot que les corresponda
3. Todos los objetos del FullSnapshot que no se encuentran en el Mundo son creados a partir de sos Snapshot.

Modulos cliente y servidor

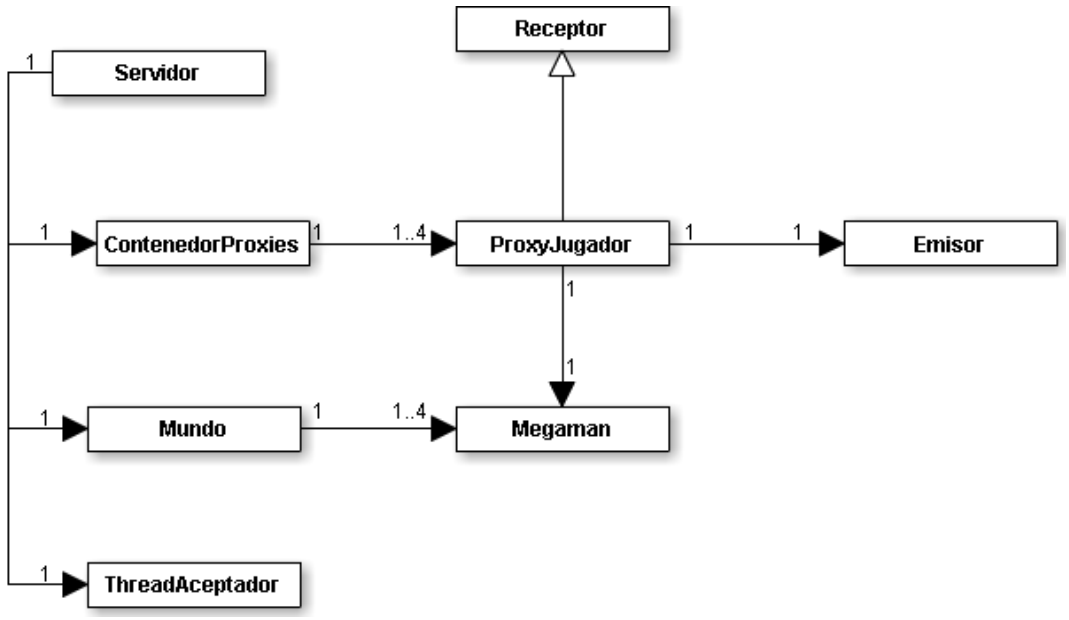
Los módulos cliente y servidor se ocupan de tareas radicalmente distintas pero están fuertemente relacionadas.

Tanto Cliente como Servidor tienen la responsabilidad de coordinarse para regular el flujo del juego. En el caso del Cliente, se reciben y emiten señales (y se delega su recepción y emisión) tanto de la red como de la máquina local.

Cliente aisla la VentanaJuego del protocolo cliente-servidor, y coordina el input del teclado. **ReceptorCliente**, se encarga se transmitir los mensajes recibidos desde el servidor al resto de la aplicación cliente. **Jugador** recibe mensajes del teclado y los transmite al Megaman correspondiente dentro del Mundo.

Servidor coordina los componentes del servidor, reaccionando de manera acorde tanto a eventos generados por el fin o inicio del Mundo, o provenientes de la red. **EscuchadorInput** espera input del teclado y lanza un evento al Servidor para que se cierre de recibirse la Q. **ContenedorProxy** se ocupa de coordinar y manejar los ProxyJugador. **ProxyJugador** representa un cliente, se ocupa de la emisión y recepción de mensajes de cada uno de ellos.

Diagramas UML



Descripción de archivos y protocolo

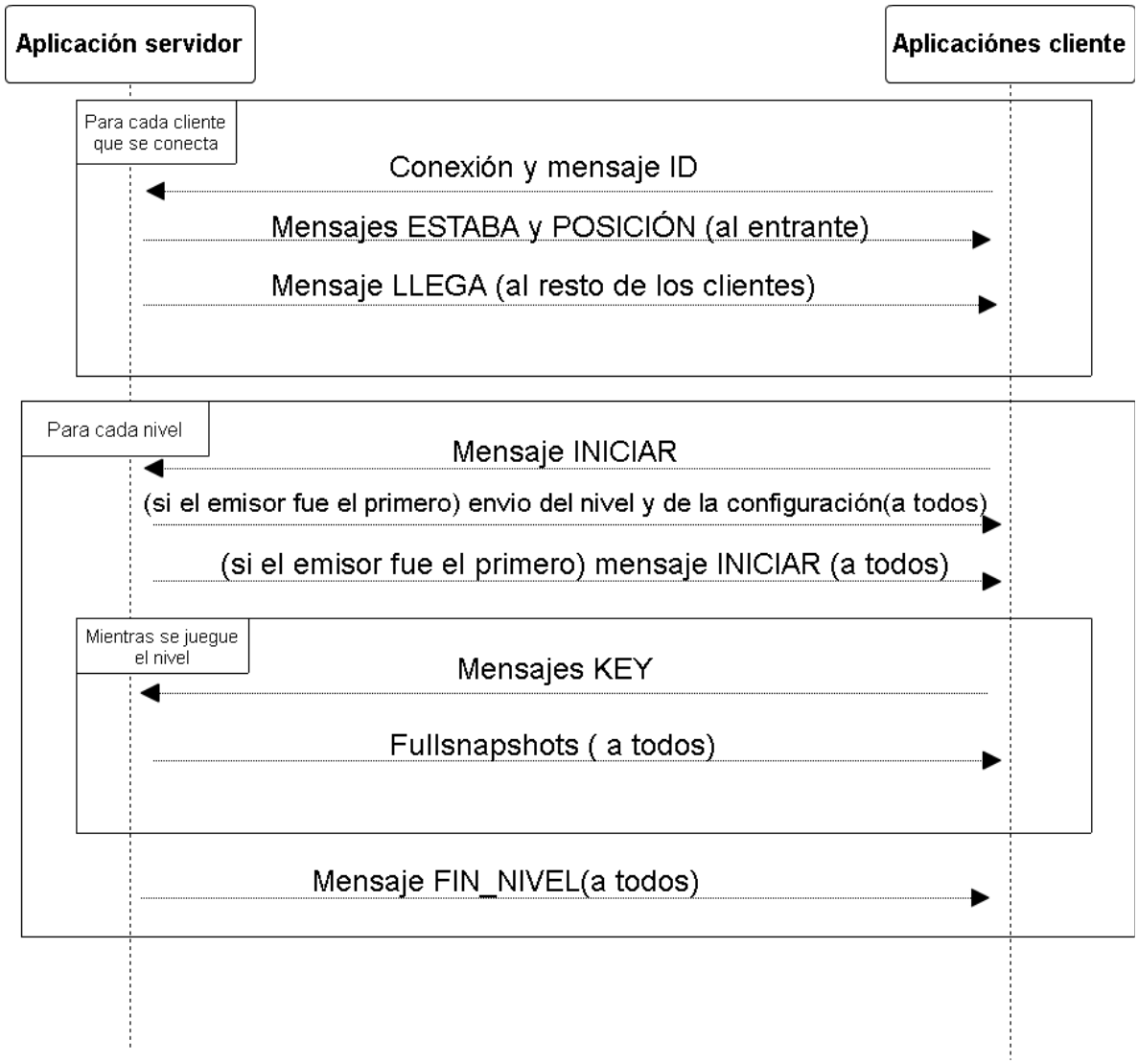
En el archivo defines_protocolo.h se encuentran sumarizados todos los tipos de mensaje que admite el protocolo. A continuación se los detalla.

- ID: es el primer mensaje que envía el cliente al receptor, viene acompañado del nombre de usuario asociado al jugador entrante.
- LLEGA, ESTABA, POSICION: son mensajes utilizados en la etapa previa a la ejecución del nivel. LLEGA y ESTABA son utilizados para notificar al resto de los jugadores la llegada de algún otro. Vienen acompañados id del jugador que entró, o que estaba conectado antes de que llegara el receptor. POSICION indica al cliente la posición en que llegó (si primero, segundo, tercero o cuarto). El número recibido en POSICION es el que se utiliza internamente para determinar a qué Megaman dirigirle los controles.
- INICIAR: Siempre acompañado del nivel, indica o bien que el cliente emisor quiere iniciar el nivel que se marca, o bien que el cliente receptor debe lanzar el nivel correspondiente.
- FIN_NIVEL: indica a los clientes el fin del nivel.
- KEY_...: Son utilizados para transmitir los controles al servidor desde el cliente. Asi, cuando el usuario del lado del cliente juega, en realidad “toca botones” del servidor. Continuando esta metáfora, cada uno de los mensajes KEY representa un botón.
- INICIAR_ENVIO_FULLSNAPSHOT, ENVIO_SNAPSHOT y FIN_ENVIO_FULLSNAPSHOT: Son utilizados para el envío de snapshots.

Utilizando el vocabulario de la sección anterior, los mensajes de tipo INICIAR y FIN no tienen mensajes asociados. El mensaje ENVIO_SNAPSHOT, por otro lado, viene acompañado del snapshot serializado, es decir, del string que devuelve el método correspondiente de Snapshot.

- INICIAR_ENVIO_NIVEL, ENVIO_NIVEL, TERMINAR_ENVIO_NIVEL: funcionan de forma similar al envío de snapshots: Los mensajes de inicio y envío están “vacíos”, mientras que el ENVIO_NIVEL viene acompañado de una línea del archivo xml correspondiente.

- INICIAR_ENVIO_CONFIG, ENVIO_CONFIG, TERMINAR_ENVIO_CONFIG: Funcionan de la misma manera que el envío de nivel, pero envían el archivo de configuración. Para mantener el sistema simple se prefirió agregar estos mensajes en vez de generalizar en un ENVIO_ARCHIVO.



Del lado del cliente el archivo de configuración, proveniente del servidor, se guarda en <nombre de usuario>configuracion.conf, y el nivel recibido se guarda en <nombre de usuario>nivel.xml. Una vez recibidos, se leen de la misma forma que del lado del servidor: ese fue el motivo por el que se decidió guardar los archivos en el disco. Se incluyó el nombre de usuario prefijo para que se puedan correr varias instancias de la misma aplicación cliente.

Pese a que no son necesarios durante la ejecución del programa, los archivos se borran sólo al terminar la aplicación cliente.

Módulo gráfico(graficos)

El módulo de gráficos contiene tanto clases que se abocan al dibujado de gráficos como al cargado de imágenes y la creación y manejo de eventos provenientes de la ventana.

Aunque lo ideal hubiera sido separar éste paquete en subpaquetes, se hará aquí un agrupamiento temático de las clases:

1. Dibujado del mundo/gráficos propiamente dichos: Dibujable, Imagen, ImagenEscalada, Fondo, Animado, Animacion.
2. Manejo de archivos: ArchivoImagen y CacheImagenes
3. Ventana: VentanaJuego, DareaSplash y Callbacks varios.

1-Dibujado del mundo:

Dibujable es el tipo base de todo aquello que se puede dibujar en la pantalla. También contiene métodos estáticos que asisten en el dibujado de formas e imágenes.

Imagen e **ImagenEscalada** son hijas de Dibujable. La primera, representa una imagen que se amplía manteniendo sus proporciones. La segunda, no mantiene las proporciones de la imagen sino que la ajusta a los bordes del objeto.

Animado deriva de **Imagen**, representa un objeto que puede reproducir una o varias animaciones.

Animacion no deriva de ninguna de las clases anteriores, representa la animación que reproduce Animado.

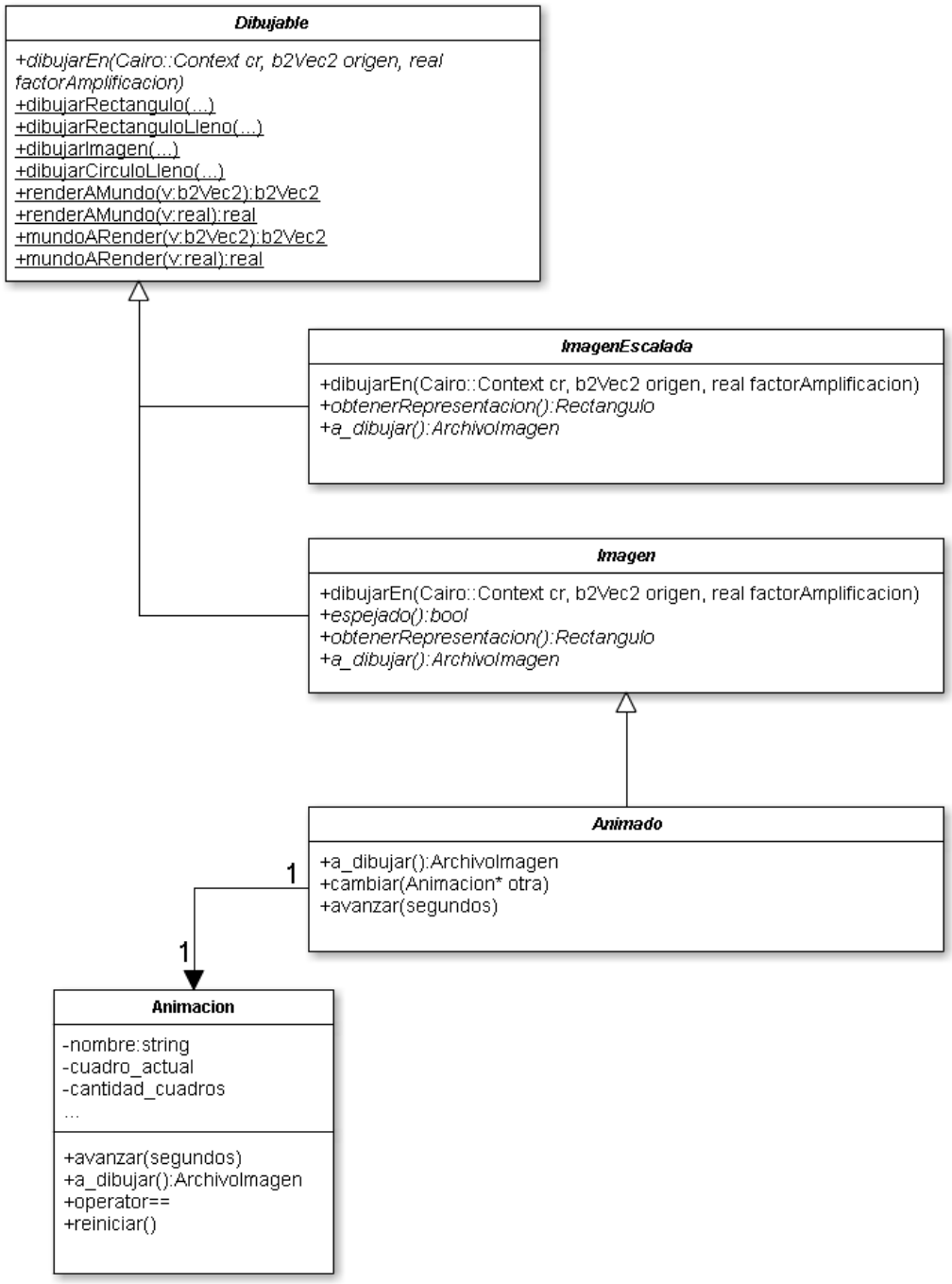
2-Manejo de archivos

ArchivoImagen es un tipo que aísla el sistema de la clase Pixbuf de Cairo. Representa una imagen en disco. **CacheImagenes** asocia cada ArchivoImagen a un Pixbuf, evitando de esta forma que exista más de un Pixbuf por cada imagen visible.

3-Ventana

DareaSplash deriva de Cairo::DrawingArea, es el “splashscreen” del princípio, que contiene el título del juego. **VentanaJuego** deriva de Gtk::Window, representa la ventana y maneja parte de los eventos que lanza Window. Los varios Callbacks del paquete se utilizan para que la ventana reciba eventos del Receptor a traves del Cliente y pueda hacer que se muestren. Así, por ejemplo, uno de los callbacks es llamado cuando el Receptor recibe un mensaje ESTABA, agregando el usuario correspondiente al label titulado “Lobby”.

UML



Descripción de archivos y protocolo

Las imágenes correspondientes a la misma animación deben almacenarse en una carpeta, numeradas. De esta forma se permite que Animación detecte automáticamente el largo de la animación. Todas las imágenes se almacenan dentro de la carpeta “imagenes”.

Manual de usuario

Requerimientos mínimos

Ubuntu 12.04 o superior
CPU 32bits 2,0Ghz
GPU 12MB VRAM
512MB de RAM
100MB HDD
Monitor de resolución 800x600

Configuración

El archivo configuracion.conf contiene los parámetros ajustables del lado del servidor. Nótese que modificar éste archivo del lado del cliente no tiene ningún efecto.
Del lado del servidor también es posible modificar los niveles, contenidos en la carpeta niveles. Para ésto tendrá que usar un editor de texto, no se disponibiliza un editor de niveles.
Los archivos ip.conf y port.conf, definen a qué IP y port se conectará el cliente. Del lado del servidor, sólo se usa port.conf.

Forma de uso

Uso del servidor

Debe ejecutarse el ajacutable del servidor por línea de comando. Una vez lanzado, puede cerrarse en cualquier momento ingresando Q. Modificar el archivo de configuración o los niveles mientras funciona el servidor puede generar comportamiento inesperado.

Uso del cliente

Una vez lanzada la aplicación del cliente, verá la siguiente pantalla:

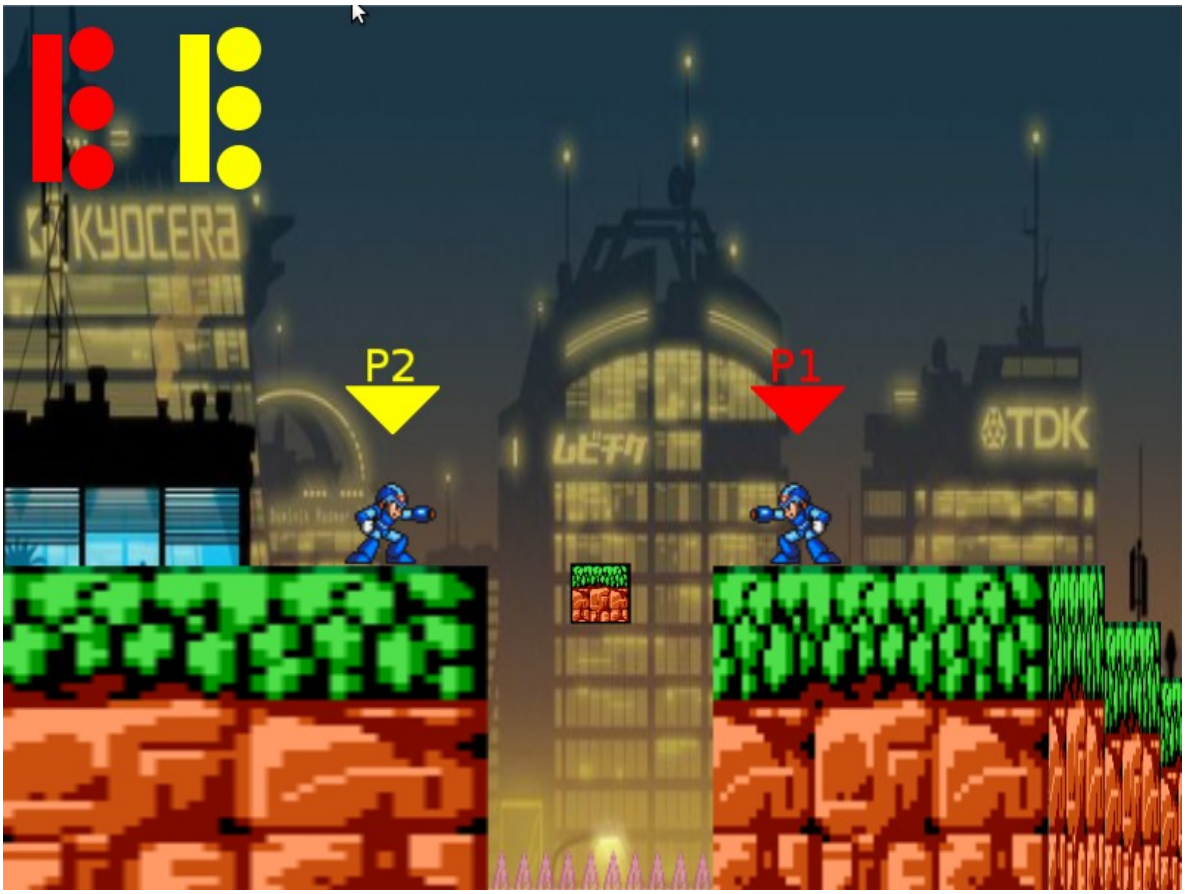
Luego de presionar una tecla cualquiera, verá la siguiente:

Esta es la ventana de selecicón de niveles. Sea durante la selección de niveles o durante el juego propiamente dicho, puede cerrar la aplicación haciendo click en [1]. El recuadro [2] muestra qué acción debe realizar a continuación, o qué es lo que está esperando. Inicialmente, debe ingresar su nombre en el recuadro [3] y luego presionar ENTER para conectarse con el servidor. Una vez hecho ésto, en [5] se le mostrarán los jugadores que están actualmente en el lobby. Si es el primer jugador en entrar, ésto estará escrito en [2], y se habilitarán los botones en [4] para elegir un nivel. Tenga en cuenta que cuando se elija el nivel, iniciará, con lo que debe esperar a que se unan el resto de los jugadores antes de presionar cualquiera de los botones en [4].

De no ser usted el primer jugador en conectarse, debe esperar a que el primero elija el nivel. Una vez hecho ésto, se iniciará el nivel también en su pantalla.

Cada vez que retorne a ésta pantalla deberá proceder de igual manera.

A continuación, verá una pantalla como la siguiente:



El color que se le asigne depende de en qué posición entró al juego. A continuación se explica el significado de los símbolos de la esquina superior izquierda:

Los círculos representan la cantidad de vidas que posee el jugador correspondiente, la barra a la ziquierda indica la salud. Se irá agotando a medida que su Megaman reciba daño. De seleccionar un arma con plasma limitado (es decir, una que no sea el megabooster), se verá otra barra, más angosta, a la derecha de la barra de vida, la cual se irá agotando a medida que se utilice el arma correspondiente.

Si su salud se agota, perderá una vida, y su Megaman morirá. Si todos los Megaman mueren, reaparecerán o bien al princípio del nivel o bien justo antes de iniciar la lucha contra el jefe final. Aquellos megaman que no tengan vidas no reaparecerán. Si a ningún Megaman le quedan vidas, en vez de reaparecer, se lanzará la pantalla de selección de nivel nuevamente. Puede recuperar vida agarrando bonus que dejan caer los enemigos al morir.

Cuando derroten a cualquier jefe final, su arma se agregará a su arsenal. Presionando las teclas 1 a 6 puede ciclar entre las armas seleccionadas. Excepto el megabooster de Megaman, todas las armas tienen un depósito de plasma limitado, deberá llenarlo recogiendo contenedores de plasma que dejan caer los enemigos.