FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Argentum - Manual Técnico

[7542/9508] Taller de Programación
Primer cuatrimestre de 2020

FRITZ, Lautaro Gastón	102320
FABBIANO, Fernando	102464
NOCETTI, Tomás	100853

Link al repositorio de GitHub:
https://github.com/tomasnocetti/taller-trabajo-final

Índice

1. Introducción

El siguiente trabajo practico tiene como objetivo la implementación de una versión simplificada del juego Argentum, con su parte de cliente y servidor, que integre los conocimientos adquiridos durante la cursada de Taller y Programación I. No es el objetivo de este informe entrar en detalles sobre como fue el desarrollo en su totalidad, sino realizar un análisis sobre como se realizo la implementación de algunas partes del sistema. Algunos conceptos generales que se buscaran explicar refieren a temas que incluyen:

- Aplicaciones Cliente-Servidor multi-threading.
- Manejo de Socket de comunicación TCP bloqueantes.
- Streams y manejo de archivos.
- Arquitectura y patrones de programación.

2. Requerimientos de software

En esta sección se procede a analizar los distintos componentes de software que facilitaron la realización del trabajo.

2.1. Entorno

El trabajo fue desarrollado y probado en su totalidad bajo el sistema operativo Ubuntu 18.04.04 lts. En el caso de uno de los integrantes se utilizo OSX 10.14.6 para parte del desarrollo, pero no se asegura que el programa funcione en su totalidad bajo dicho OS.

2.2. Librerías

Lo que respecta a la interfaz gráfica fue realizado utilizando íntegramente algunas las bibliotecas provistas por SDL2, específicamente SDL2_image (para la carga y gestión de imágenes), SDL2_ttf (para los textos y fuentes) y SDL2_mixer (para la música y efectos de sonido).

3. Módulo 1: interfaz gráfica e interacción con el usuario

3.1. Descripción General

Este módulo funciona de una forma parecida al patrón MVC, aunque no exactamente igual. Hay una serie de controladores (uno para el jugador principal, uno para los enemigos, otro para el mapa, etcétera) que, en cada ciclo del juego, reciben información del modelo, el cual se encuentra en el Servidor (en realidad, el modelo deja los datos actualizados en un proxy, que es de donde los controladores toman la información), y en base a ella operan de alguna forma para luego incidir sobre las entidades que luego serán renderizadas.

Algunos controladores pueden recibir acciones del usuario, en cuyo caso llamarán a la función correspondiente del proxy para que se produzca el cambio deseado en el modelo.

En cualquier caso, una vez actualizadas las entidades, se procede a renderizarlas. Para esto, la pantalla fue dividida en "viewports" (una subdivisión de la pantalla utilizada para renderizar una parte en específico). De esta forma, se podía trabajar en más de una parte de la pantalla al mismo tiempo, sin que se pisen entre sí. Cada una de estas solicita las entidades que debe renderizar al controlador indicado, para luego llevarlas a la pantalla.

3.2. Clases

3.2.1. Controladores

Se cuenta con un total de siete controladores diferentes, siendo estos: MainPlayerController (para controlar todo lo referido al jugador: barras de vida y experiencia, cantidad de oro, el jugador en sí, etcétera), EnemyController (para los monstruos y los otros jugadores), InventoryController (para generar las imágenes correspondientes al contenido del inventario), ChatController (para el chat), MapController (para el mapa y los drops), LoginController (para la pantalla de inicio) y GlobalController (para la música de fondo y la imagen que oficia como fondo de pantalla).

Lo que todos tienen en común es que poseen al menos un método `update`, en el que primero toman la información necesaria del proxy y luego, con esa información, actualizan a las entidades que luego vayan a ser renderizadas, y uno o más métodos que devuelven dichas entidades (getters).

Como se mencionó anteriormente, algunos de estos (en particular MainPlayerController y ChatController) pueden recibir acciones por parte del jugador, por lo que poseen los correspondientes métodos `handleEvent`, en los cuales llaman a determinado método del proxy.

3.2.2. Entidades

En el contexto de la aplicación, se llama entidad a todo objeto que contenga una textura (o una animación, que a su vez contiene a la textura) para poder ser llevado a la pantalla. Hay una clase de entidad para cada uno de estos objetos (ejemplo: hay una clase Bar para las barras, una TextEntity para los textos, DropEntity para los drops, etcétera).

Todas estas heredan de la clase abstracta Entity, cuyo único método virtual puro es `paint`. Sin embargo, este está sobrecargado para aceptar dos versiones: una con cámara y una sin cámara, ya que hay entidades que deben renderizarse relativas a la posición del personaje. Este método, a su vez, llama al método homónimo de la textura/animación que posea (con el chequeo opcional de ver si está o no en el rango de la cámara, en caso de ser necesario).

3.2.3. Animaciones

Algunas entidades, como el jugador o los monstruos, están animadas. Esto implica que la entidad está en relación con una clase que gestiona esa labor.

Hay una clase de animación por cada tipo de monstruo, una para el jugador y una para el fantasma que aparece cuando el jugador muere. A su vez, todas son herederas de la clase Animation. Todas sobreescriben el método `cropAnimationFrames`, en el cual se recortan los cuadros a medida, porque las diferentes texturas son de tamaños distintos.

El resto de los métodos están definidos en la clase madre, ya que son iguales para todas las animaciones. Otro método importante es `setCurrentAnim`, que elige la foto adecuada dependiendo de la información que reciba. De esta forma, las animaciones no son más que una colección de cuadros que recortan la imagen adecuada para dar la ilusión de movimiento.

3.2.4. Viewports

Como se mencionó anteriormente, la pantalla de juego fue dividida en Viewports. Hay un Viewport por cada subdivisión lógica que podía realizarse de la pantalla; en total son seis. Por ejemplo, hay uno para el mapa, uno para el inventario, uno para el chat, y así.

Lo que todos los Viewports tienen en común es que poseen un método `paint`, en el cual reciben un vector de punteros a entidades y llaman al método `paint` de la entidad correspondiente. Esto se manejó de esta forma ya que los Viewport conocen a la ventana del juego, y por lo tanto pueden pedirle un factor de escalamiento. De esta forma, se puede lograr que las imágenes se ajusten al tamaño de la pantalla.

3.2.5. ServerProxy

Esta clase funciona como un mediador entre los controladores y el modelo: si el controlador requiere algún cambio en el modelo, llama al método correspondiente del ServerProxy, y este genera la instrucción necesaria para producirlo. Por otro lado, el modelo deja los datos en esta clase, para que luego cada controlador tome los que necesita.

En cuanto a métodos principales, quizás el más importante es `setGameModelData`, en el cual el modelo deja los datos actualizados para ser tomados por los controladores.

3.2.6. CApp

CApp es la clase que articula todo el flujo descripto anteriormente. Tiene como atributos a todos los controladores y todos los viewports, y en cada ciclo del juego ejecuta `OnUpdate`, que actualiza a todos los controladores, y luego ejecuta `OnRender`, donde, con las entidades que devuelven los controladores, llama al `paint` de los viewports para llevarlas a la pantalla. Se espera que dicho game loop se ejecute en una determinada cantidad de tiempo. Si este tiempo de ejecución es menor, se pone a dormir el hilo hasta llegar al tiempo deseado. En el caso en el que el ciclo llevo más tiempo del esperado, este será recuperado en iteraciones futuras. De esta forma se logra mantener el frame constante, y así una mayor fluidez en el juego.

También, antes de que comience el juego, carga todas las texturas y sonidos que pueden ser necesarios a lo largo de la partida (en el método `LoadAssets`).

3.3. Diagramas

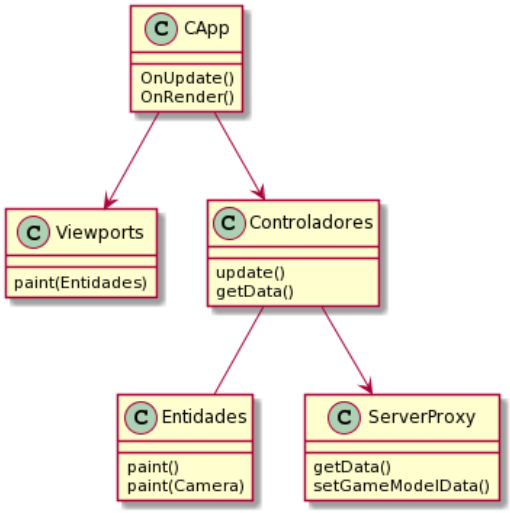

Figura 1: Pantallazo general de las clases que componen el módulo. El método getData() no existe como tal, sino que es una generalización de los varios getters que existen y que serán detallados más adelante.

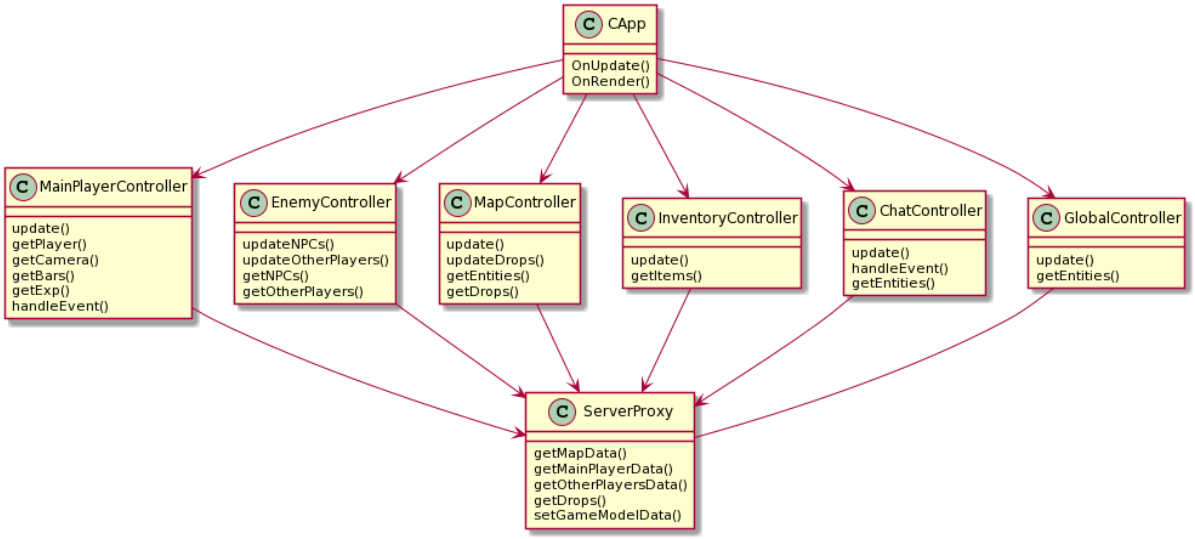

Figura 2: Controladores. Como se puede ver, algunos tienen sus update específicos, pero en líneas generales todo lo explicado previamente aplica a todos los controladores.

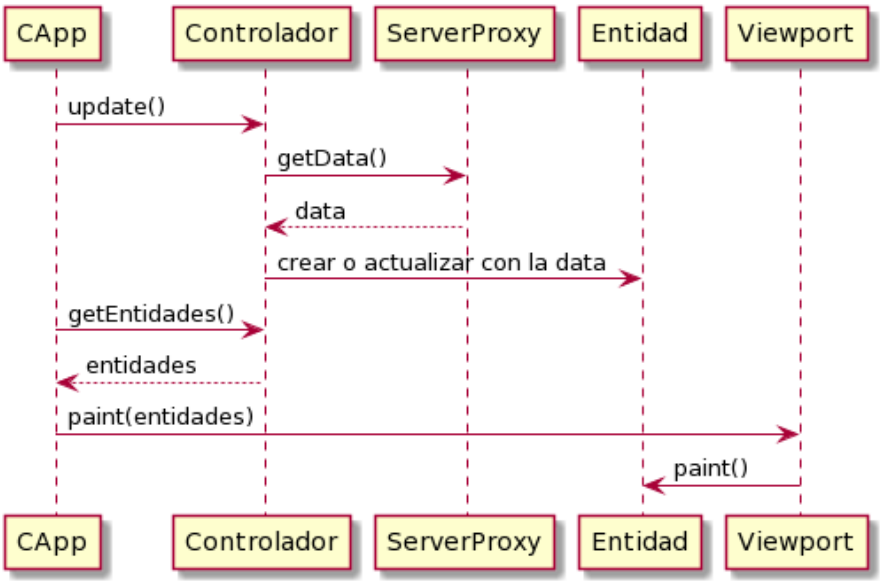

Figura 3: Diagrama de secuencia de un ciclo de juego básico del lado del cliente.

4. Módulo 2: Comunicación Cliente-Servidor

4.1. Descripción General

La comunicación Cliente - Servidor se realiza mediante sockets, utilizando la librería msgPack para serializar los datos a enviar. A continuación, detallamos el protocolo elegido para la serialización de los mensajes:

- Un byte indicando el tipo de respuesta que se va a interpretar (este byte es incluido únicamente en el flujo de datos del servidor al cliente). Este puede estar representando que lo recibido

proveniente del servidor es la información del mapa, o la información del modelo del juego (incluye datos de los distintos jugadores y criaturas).

- 4 bytes, indicando la longitud del mensaje que se va a recibir/enviar.
- El cuerpo del mensaje.

4.2. Cliente

Cuando se conecta un nuevo cliente al servidor, se lanzan dos hilos: `ServerProxyRead` y `ServerProxyWrite`.

El primero se encarga de generar instrucciones de acuerdo a las interacciones del jugador con el juego, y enviarlas en un formato que pueda ser interpretado por el servidor. Estas instrucciones cuentan con dos atributos: un dato correspondiente al tipo de instrucción (createPlayer, buy, move, etc.), y un vector de strings. Cada una de estas cadenas de caracteres, representa un parámetro que acompaña a la instrucción. Ejemplo: para la instrucción createPlayer, es necesario enviarle al servidor los parámetros nick, password, raza y clase. Estos parámetros se envian como strings en el vector mencionado de la instrucción.

El segundo hilo es el encargado de leer y setear la información actualizada proveniente del servidor. Esta información contiene las actualizaciones del propio cliente, y del resto de los jugadores y criaturas del juego.

4.3. Servidor

Al levantar un servidor, se pone a correr un hilo llamado `GameServer` que es el encargado de orquestar todo lo que pasa en el juego (interpretar instrucciones provenientes del cliente, actualizar los modelos). De ahora en más, este hilo será referenciado como el `Hilo Principal` del juego. Este, a su vez, luego de ser iniciado, lanza además dos hilos que son de gran importancia para manejar la lógica del juego: `GameCron` y `ClientAcceptor`.

El `GameCron` va a ser el encargado de manejar toda la lógica del movimiento de los jugadores y NPC's, y se va a hacer cargo de distintas acciones del juego que necesitan periodos de tiempo para ejecutarse, como ser el ataque de los NPC's, la lógica de resurrección, tanto de NPC's como de jugadores, y la recuperación de los atributos del jugador con el paso del tiempo.

Con el objetivo de brindar una buena experiencia al usuario, tanto `GameCron` como `GameServer` manejan un frame constante para realizar el loop del juego. En el caso del `GameServer`, si bien esta en constante comunicación con el `GameCron` para asegurarse de que este tenga la información actualizada, no propaga el modelo del juego a todos los clientes si no paso una determinada cantidad de tiempo. Es decir, tanto `GameServer` como `GameCron` siguen actualizando el modelo, solo que los clientes no se enteran de las actualizaciones hasta que dicho tiempo se cumpla, y entonces si, el hilo orquestador se encarga de difundir esa información. En el caso del `GameCron`, sucede algo análogo, sólo que luego de ejecutarse el código correspondiente a la lógica de dicho hilo, este se pone a dormir por una determianda cantidad de tiempo, que varía dependiendo cuanto tarde el thread en correr su lógica, pero, al fin y al cabo, el tiempo que tarda el `GameCron` en hacer un loop es siempre el mismo; lo que varia es la cantidad de tiempo que este pasa inactivo.

Por otro lado, el `ClientAcceptor` es el hilo que se encarga de esperar nuevas conexiones de jugadores, así como luego es el encargado de finalizar esas conexiones y liberar recursos en el momento en el que ya no se estén utilizando. Por cada nuevo jugador que se conecta al servidor, este lanza un nuevo hilo `ClientProxy`, que se encargará de que el jugador se inserte en el juego, y se pueda comunicar tanto con el `hilo principal`, como con el cliente al que esta referenciando.

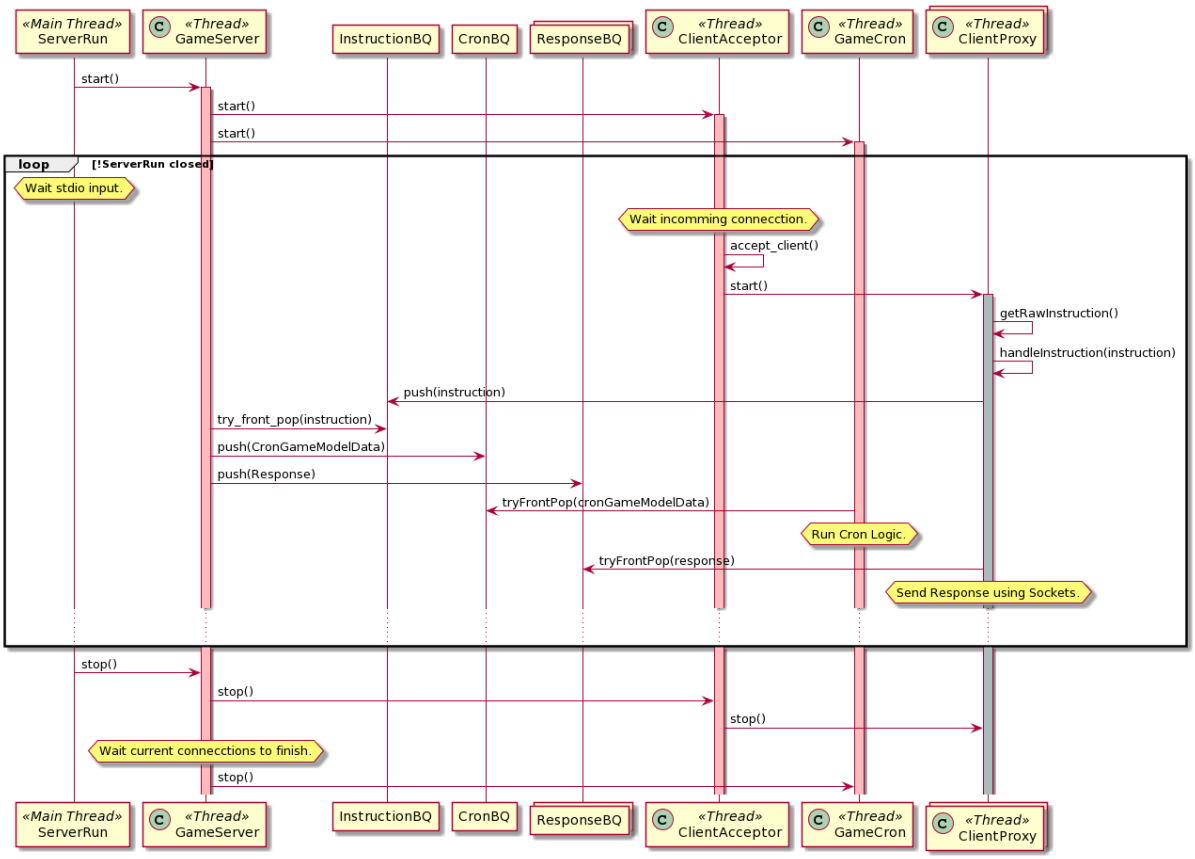

Figura 4: Diagrama de secuencia del ciclo que realiza el servidor para interpretar las interacciones del cliente con el juego, y para generar las actualizaciones, respuestas y propagaciones correspondientes a las mismas.

Luego de interpretar y procesar las instrucciones provenientes del cliente, el servidor se encarga de preparar un modelo actualizado a cada jugador. Dicho modelo contiene ni mas ni menos que la información que necesita el cliente para actualizar la vista. Esto quiere decir que para cada jugador, se genera un conjunto de datos personalizado, logrando asi que cada cliente solo conozca los datos necesarios del resto de los jugadores. Al `Jugador1` le es irrelevante conocer, por ejemplo, el chat o los datos bancarios del `Jugador2`, y por eso es que el modelo del primero no contiene esos datos del segundo.

5. Módulo 3: Sistema de instrucciones

Como se explico brevemente en el `Módulo 2: Comunicación Cliente-Servidor`, el cliente le notifica al servidor de las interacciones del jugador con la vista mediante la utilización de instrucciones. En esta parte del programa, estas no son más que estructuras de datos que especifican el tipo de Instrucción, y los parámetros necesarios para interpretarla y ejecutarla. Estas estructuras serán enviadas utilizando el protocolo ya mencionado.

A la hora de interpretar la instrucción del lado del servidor, lo primero que se hace es identificar el tipo. Una vez identificado, se procede a crear una instancia de alguna de las hijas de la clase `Instruction`. Ésta, por polimorfismo, sabrá como ejecutarse.

Cabe destacar que no todas las instrucciones afectan directamente al modelo, sino que en algunos casos modifican algún dato de un jugador. Esta modificación la captará y procesará el `GameCron` para luego si, generar una instrucción que tendrá incidencia directa en el modelo del

juego, y por ende en la vista del jugador. Es decir, tenemos dos tipos de instrucciones: las que se generan en primera instancia, como respuesta a lo recibido del cliente, y las que se generan debido a la acción del `GameCron`.

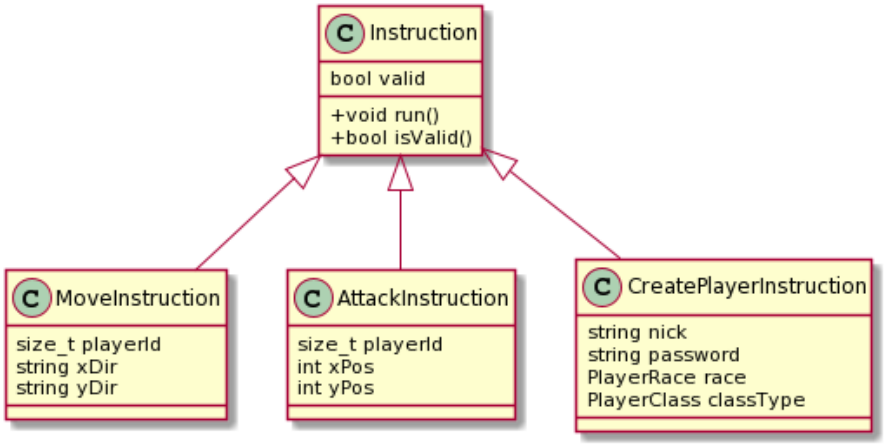

Figura 5: Diagrama de clase que muestra algunos ejemplos de las instrucciones que puede manejar el servidor para actualizar el modelo.

6. Módulo 4: Archivo de configuración

En el inicio del proyecto se trabajó con macros para declarar todas las variables configurables del juego, tanto las exigidas por el enunciado, como las propias de la implementación.

Esto sin duda era problemático, sobre todo en términos de tiempo desperdiciado para los desarrolladores. Por qué? Simple. Con solo querer modificar una constante configurada en alguna de esas macros (todas en archivos .h), era necesario recompilar gran parte del proyecto. Esto impedía poder hacer testeos y pruebas con agilidad, y por ende el tiempo de codeo se demoraba mucho. Al notar esto, y ver que era un cambio necesario, se procedió a migrar todas esas constantes a un archivo de configuración del tipo json. Esto nos permitiría modificar constantes y ver cambios en tiempos mínimos, sin necesidad de recompilar.

Para evitar cualquier problema de Race Conditions, se utilizó el patrón de diseño `Singleton`, que crea una única instancia al inicio del juego, que es accesible desde cualquier módulo del programa.

Por último, para para la manipulación del ya mencionado archivo de configuración, se utilizó la librería JsonCpp.