7542 Taller de programacion I

Documentacion tecnica

Integrantes:

Alumno	padron
Pinto, Santiago Augusto	96850
Blanco, Sebastian Ezequiel	98539

Fecha de Entrega: 27/06/2017

Índice

1. Requerimientos de software

Para el correcto funcionamiento de la aplicación se deben tener los siguientes programas instalados:

- **Sistema operativo:** Ubuntu 16.04
- Compilador de ISO C++11
- Instalación de las librerías `SDL 1.2` y `SDL-image 1.2`, de no tenerlas se requiere escribir los siguientes comandos en la terminal:
  - `sudo apt-get install libsdl1.2-dev`
  - `sudo apt-get install libsdl-image1.2-dev`
- Para depurar el programa debe utilizarse `GNU gdb 7.11.1`

2. Descripcion General

El proyecto se puede dividir en dos módulos principales que son el cliente y el servidor. Estos a su vez, pueden dividirse en submódulos que pasamos a detallar a continuación.

2.1. Cliente

El cliente está representado con la clase `Player`, la misma ejecuta un loop durante la ejecución del juego en el cual, en cada iteración, se realizan tareas de dibujo, interacción con el usuario y comunicación con el server, cada una siendo responsabilidad de un modulo distinto, mencionado a continuación.

2.1.1. Módulo de dibujo

En este módulo del cliente se encuentra toda la lógica requerida para poder trasladar lo que ocurre en el juego del lado del servidor a una escena que pueda ser entendida por un usuario que hace uso de la aplicación. En este modulo las clases a destacar son `Map` y `GameObject`, que cumplen casi todas las tareas que hacen posible esto y cuyas responsabilidades se detallaran más en la sección de módulos a continuación.

2.1.2. Módulo de interacción con el usuario

En este módulo, como dice el titulo, se encuentran todas las clases que permiten que el usuario interactúe y tenga incidencia en el desarrollo de una partida. Las clases que son encargadas de hacer posible la interacción son `ClientMouseEventHandler` y `ClientKeyboardEventHandler` que, respectivamente, habilitan el ingreso de información tanto por mouse como por teclado. Otra clase importante en este módulo es la clase cámara, la cual muestra al usuario toda la información visual que necesita.

2.1.3. Módulo de comunicación Cliente-Servidor

En este módulo se encuentran todas las clases que involucran el traspaso de información de cliente a servidor. En el mismo se destacan las clases `ProxyGame` y `Communicator`. Su función es recibir, deserializar y almacenar la información enviada desde el servidor de tal forma que el cliente pueda entenderla y manipularla correctamente.
Además son los encargados de realizar peticiones al servidor, como el envío del modelo cuando sea necesario, entre otras.
Tanto el módulo de dibujo como el de interacción con el usuario se ejecutan en el mismo hilo, mientras que toda la parte de comunicación con el servidor se realiza en paralelo (módulo de comunicación), para lograr una mayor eficiencia.

2.2. Servidor

El servidor está representado por la clase `Game` la cual posee distintos métodos que modifican, actualizan y depuran el modelo del juego. El loop principal del servidor está representado por la clase `ProxyClient` que es un hilo que se encarga de establecer la comunicación con un cliente en particular.

2.2.1. Módulo de aceptación de clientes

Este módulo está representado por la clase `Server` la cual tiene un loop principal en donde en cada iteración espera hasta que un cliente establezca una conexión, siendo este loop bloqueante, ya que se utiliza la aceptación de sockets.

2.2.2. Módulo de comunicación con el cliente

Este módulo está representado por la clase `ProxyClient` la cual es un hilo que recibe pedidos de parte del cliente a través de la comunicación TCP usando el socket que aceptó el módulo anterior. Además se encarga de notificarle el pedido al modelo de manera tal que este último realice los cambios deseados por el cliente. Por último, se encarga de devolver la serialización del modelo en caso que el pedido sea ese, ya que el `proxyClient` sabe que cuando el cliente lo requiere, el `ProxyClient` necesita responder inmediatamente.

2.2.3. Módulo del modelo servidor

Este módulo está representado por la clase `Game` la cual interactúa con muchos `ProxyClient`, que son hilos. Por lo tanto esta clase es protegida. Está compuesta de métodos que definen las base del modelo del juego como la creación de unidades, el ataque a objetos y unidades y los movimientos de unidades. Además hay un método principal llamado `update()` el cual se encarga de actualizar el mismo recorriendo todas las unidades del juego, las municiones y los objetos y actualizando sus estados en caso de que estos necesiten serlo.

2.2.4. Modulo deserializacion del modelo

Este módulo está representado por la clase `ServerInterpreter` la cual se encarga de serializar el modelo servidor y almacenar dicha serialización en un vector. Además se encarga de desserializar los pedidos que llegan desde el cliente y llamar al método correspondiente del modelo.

3. Descripcion de modulos

3.1. Cliente

3.1.1. Modulo de dibujo

Como se dijo anteriormente, la responsabilidad de este módulo es la de convertir la información del modelo, la cual se encuentra en el server, en información visual que un usuario pueda entender. Las clases que más se destacan en este módulo son `Map` y `GameObject`.
A continuación se detallan las clases que componen este módulo, así como sus responsabilidades:

3.1.1.1 Map

Es la clase con mayor lógica de este módulo. Almacena todo el conjunto de unidades, edificios, objetos de terreno, balas , explosiones y tiles presentes en el mapa que se dibuja. Su mayor responsabilidad es dibujar en cada iteración del cliente todos los objetos anteriormente mencionados. Para realizar esto, posee un método `draw` que recibe en cada iteración un objeto de tipo `SDL_Surface*`, que es la pantalla donde se dibujara la escena. El mapa entonces recorre los elementos de la escena uno por uno y les comunica que deben dibujarse, luego cada objeto en particular es responsable de saber cómo dibujarse con el sprite indicado. Los métodos más importantes de esta clase se detallan a continuación:

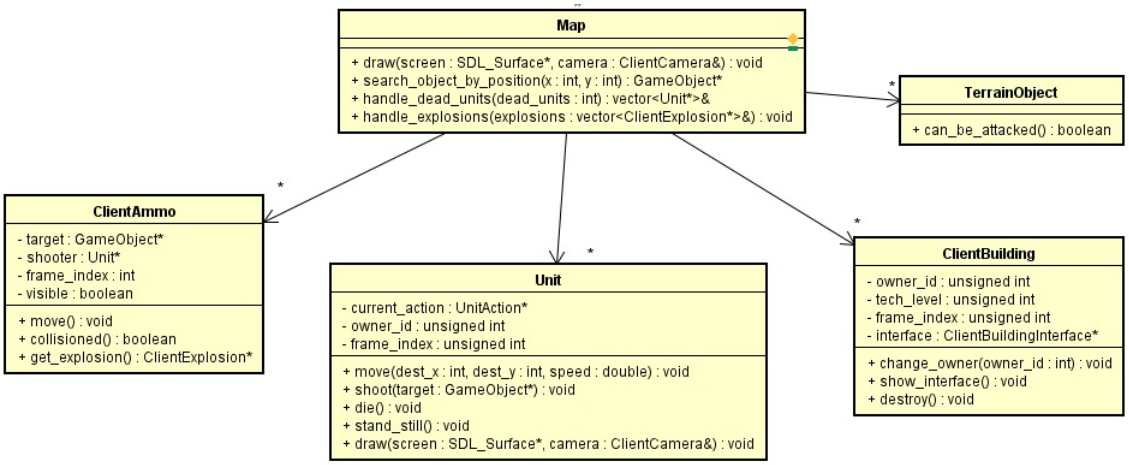

Figura 1: Map

3.1.1.1 draw

Este metdoo recibe como parametro la pantalla donde se dibujara la escena, y recorre todas las estructuras de datos donde se almacenan los objetos y le pide a cada objeto en particular que se dibuje. Adicionalmente en este método se realizan la liberación de recursos de aquellos objetos que, por alguna razón, ya no deben ser dibujados, ya sea una unidad que murió, una bala que ya colisionó o bien una explosión que ya terminó su animación. Como ya no es necesario dibujar estos objetos se realizan en métodos aparte la liberación de recursos de los mismos.

3.1.1.1 create_grunt

Simplemente agrega un nuevo robot de tipo grunt al mapa de unidades, para que en el siguiente llamado al método `draw`, este aparezca en la escena. Existen métodos similares a este como `create_laser`, `create_pyro`, `create_light_tank`, etc. Todos son análogos a este con la diferencia de que tipo de unidad agregan a la escena.

3.1.1.1 unit_move

Recibe el id de la unidad que se tiene que mover, las coordenadas del destino y la velocidad con la que se debe desplazar. Este método le asigna a dicha unidad una acción `ClientWalkAction`, dicha acción en cada llamada al método `draw` hará que la unidad se desplace una parte del trayecto que debe recorrer hasta llegar a la posición final. En resumen lo que este método hace es setear a una unidad una acción de movimiento que no solo hace que esta se desplace, sino que también se encarga de que la unidad cambie el sprite de movimiento a medida que cambia su posición.
Analogamente se enceuntra el metodo `bullet_move`, que funciona de fomra analoga a este, pero para las balas

3.1.1.1 unit_shoot

Este método recibe por parámetro el id de la unidad que ejecuta un disparo y además un objeto target (que puede ser otra unidad, un edificio o bien una roca o puente). Su función es asignar a la unidad especificada por id una acción `ClientShootAction`, la cual en cada ciclo de draw hará que la unidad cambie su sprite al de uno de disparo. Además lo que hace es crear una bala en la posición de la unidad que dispara y la agrega al mapa de proyectiles, para que en la siguiente llamada al método draw esta aparezca y comience a ejecutar su movimiento hacia el objetivo.

3.1.1.2 GameObject

Una gran parte de los objetos en el cliente son simplemente entidades que tienen una posición en un determinado instante, representada por las coordenadas x e y, y además un id para identificarlos unívocamente. A raíz de esto la clase `GameObject` intenta generalizar el manejo de las entidades ofreciendo una interfaz que es básica para todo objeto que se quiera dibujar en el mapa. La razón por la cual esta clase es tan importante en el modulo de dibujo es por que es la clase responsable de efectivamente dibujar el objeto que representa en la pantalla cuando el mapa así se lo pide. Entre sus métodos mas destacados se encuentran:

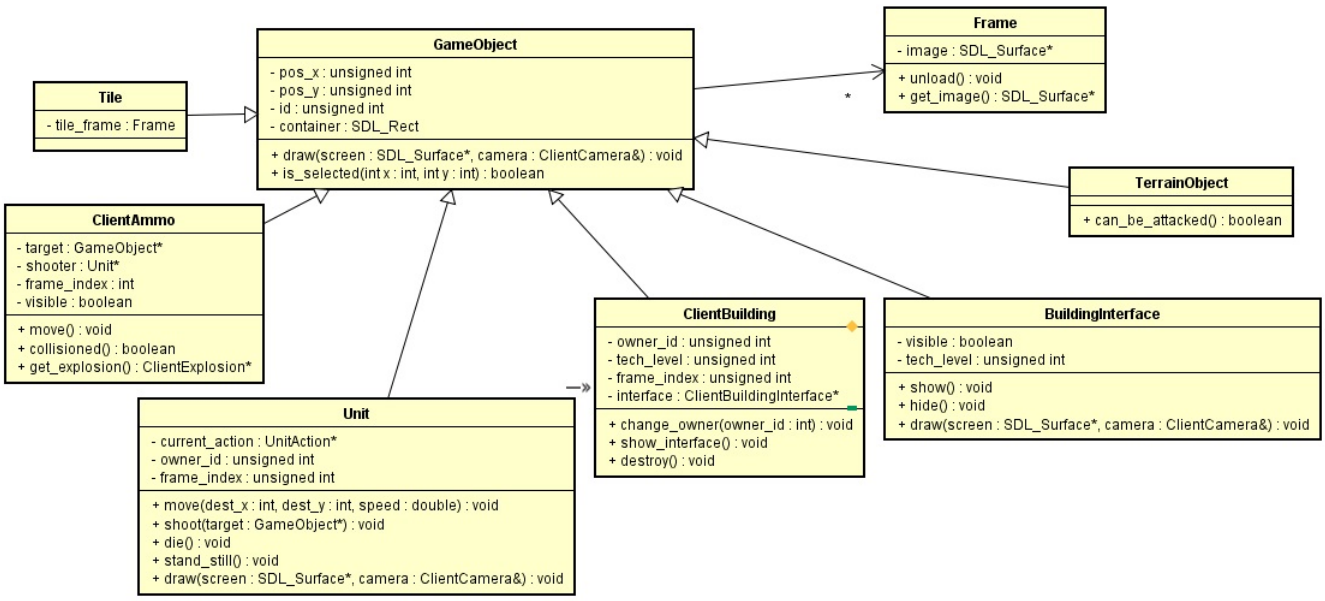

Figura 2: GameObject

3.1.1.2 get_x/get_y

Dado que todo objeto dibujable debe tener una posición, `gameObject` permite conocerla mediante estos 2 métodos.

3.1.1.2 draw

Se trata del método más importante de la clase, el mismo se encarga de que el fotograma del objeto a mostrar en esta renderizacion sea el que efectivamente aparezca en la escena para que el usuario lo vea.

3.1.1.2 is_selected

es un método más relacionado con la interacción usuario-aplicación, pero tiene cierta relación con el dibujo. Todo objeto debe ser capaz de saber si el mouse lo está seleccionando o no, según la posición del mapa donde se encuentre dibujado. Este método cumple con la tarea de informar a quien le invoque si el objeto en cuestión está seleccionado o no.
En particular las clases `Building`, `Unit`, `Ammo`, `Explosion` y `TerrainOject` heredan de `GameObject` y todas implementan por diferencia el método `draw`. Básicamente en cada método `draw` de las clases hijas se selecciona que fotograma dibujara el `draw` de `GameObject`.

3.1.1.3 Frame

Una de las clases más primitivas del modulo de dibujo ya que tiene muy poco comportamiento. En resumen cada objeto que tenga que animarse debe poseer una serie de fotogramas que sean cambiados en cada dibujo de la escena. Esta clase representa dichos fotogramas, resultando que cada `GameObject` esté relacionado con 1 o más.

3.1.1.3 get_image

Como un fotograma no es más que una imagen, este método simplemente devuelve la imagen asociada al fotograma, en forma de una `SDL_Surface*` (tipo de objeto que es dibujable), para que otro objeto lo dibuje en una escena.

3.1.2. Módulo de interacción con el usuario

Este módulo se encuentra compuesto por dos clases que se encargan de que el usuario pueda influir en el desarrollo del juego, dichas clases son `ClientMouseEventHandler` y `ClientKeyboardEventHandler`.

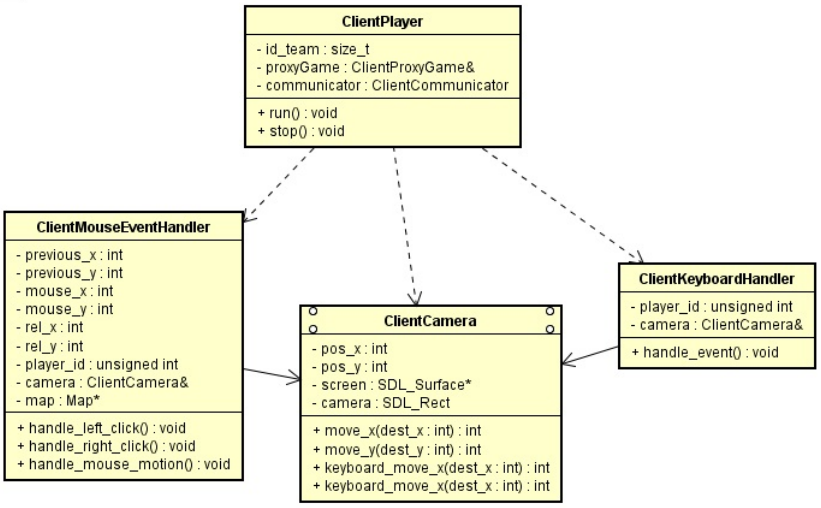

Figura 3: handlers

3.1.2.1 ClientMouseEventHandler

Permite la interacción entre el usuario y el juego mediante el mouse. El mismo permite a un jugador seleccionar unidades, indicarles un lugar a donde se tienen que desplazar, ordenar el ataque a otra unidad u objeto, uso de interfaces de los edificios para la creación de unidades y además permite llevar a cabo un scroll de pantalla que se detallara en otro módulo. Los métodos más importantes de esta clase son:

3.1.2.1 handle_left_click

Este método se encarga única y exclusivamente de gestionar la selección de objetos en el juego. Cuando un usuario hace click con el botón izquierdo del mouse sobre el mapa, se almacena una posición en el `MouseEventHandler` la cual se contrasta contra la de todos los objetos del juego para saber si alguno fue seleccionado. En esencia este método no modifica el desarrollo del juego, pero permite que un jugador acceda al control de una de sus unidades o a alguna de las interfaces de creación de alguno de sus edificios.

3.1.2.1 handle_right_click

A diferencia del `handle_left_click`, este método si tiene un impacto en el desarrollo de una partida. Siempre y cuando una unidad propia haya sido seleccionada previamente mediante el click izquierdo, este método decidirá si en la posición en la que se hizo click derecho corresponde que dicha unidad se desplace o ataque a otro objeto/unidad presente en ese lugar.

3.1.2.1 handle_mouse_motion

Si bien este metodo esta muy relacionado con lo que se dibuja en la pantalla, consideramos que sigue siendo parte del módulo de interacción. Este método simplemente compara la posición anterior del mouse (previa al último desplazamiento del mismo) con la final de desplazamiento, y le notifica a la cámara del movimiento en caso de ser necesario un scroll en la pantalla.

3.1.2.2 ClientKeyboardEventHandler

Esta clase está dedicada a permitir la interacción entre el usuario el juego mediante el teclado. A diferencia del `MouseEventHandler` este administrador se limita únicamente al movimiento de la cámara. Su método más importante es el siguiente:

3.1.2.2 handle_event

Independientemente de la posicion de la camara, si el usuario aprieta alguna de las flechas del teclado este método hará que la cámara ejecute un scroll en la dirección de la flecha que se apretó. El teclado permite únicamente interacción con la cámara.

3.1.2.3 Camera

Entidad que muestra al jugador cierta porción del mapa en la cual trascurre el juego. El jugador puede desplazarse mediante el teclado y/o mouse, para que esta le muestre distintas partes del terreno según este lo requiera. Sus métodos más importantes son:

3.1.2.3 move_x/move_y

Estos métodos están relacionados con el movimiento del mouse sobre el mapa del juego. Cuando el mouse se encuentre en alguna posición de la pantalla cercana a los bordes y se mueva un poco más allá de cierto límite, se ejecutará un scroll en la dirección del desplazamiento, permitiendo al usuario ver otra porción del mapa(siempre y cuando no salga de los limites del mismo).

3.1.2.3 keyboard_move_x/keyboard_move_y

Estos dos métodos, aunque parecidos con los dos anteriores, están relacionados con el uso de las teclas de movimiento. A diferencia de move_x y move_y, estos métodos solo se aseguran de que no ocurra un scroll fuera del mapa, pero en cualquier otro caso correrán la cámara una cantidad de píxeles fija.

3.1.3. Módulo de comunicación cliente-servidor

Este módulo realiza algunas de sus tareas en paralelo respecto de los otros dos módulos, dada la necesidad de que la aplicación sea lo más eficiente posible. La información mostrada por pantalla del lado del cliente debe reflejar lo que ocurre del lado del servidor con la mayor exactitud posible. La importancia de este módulo radica en la necesidad de pedirle al servidor la información necesaria para que el módulo de dibujo la pueda representar visualmente. Entre las clases más importantes de este módulo se encuentran `ProxyGame`, `Communicator` y `Interpreter`.

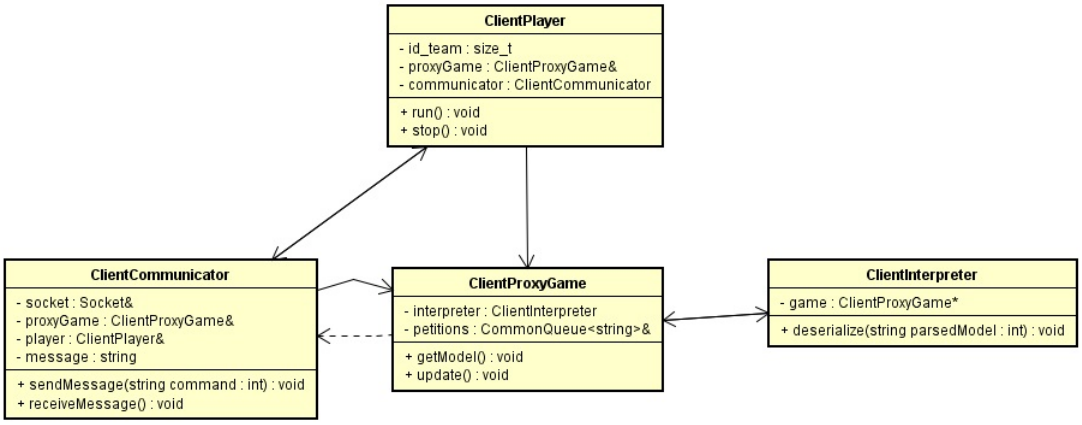

Figura 4: Communication

3.1.3.1 ProxyGame

Esta clase es un espejo de lo que ocurre en el juego del lado del servidor. Almacena un conjunto de objetos proxy que contienen la información necesaria para que el cliente pueda dibujar lo que ocurre en el server. Entre sus métodos más importantes se encuentran:

3.1.3.1 getModel

Arma un string en cierto formato, pidiéndole el modelo al servidor y lo encola en una cola de peticiones. Luego de un tiempo el comunicador desencolará esta petición y la enviará vía socket hacia el server.

3.1.3.1 update

De forma similar a get model, arma un string en cierto formato, pidiendo al servidor que actualice las posiciones de los objetos que viven en el modelo, ya sean balas o unidades moviéndose.

3.1.3.1 RefreshProxyGame

Elimina todos aquellos objetos, unidades o proyectiles que desaparecieron del lado del server ya sea porque fueron destruidos, murieron o colisionaron respectivamente.

3.1.3.2 Communicator

Es el encargado de desencolar las peticiones encoladas por el `ProxyGame` y enviarlas al server para que este actualice el modelo o lo envié vía socket. Por cuestiones de eficiencia, el comunicador corre en un hilo aparte, de esta forma independizamos la comunicacion via socket, del dibujo de la escena. Entre los métodos importantes del communicator se encuentran:

3.1.3.2 sendMessage

Recibe por parámetro un string que contiene un comando parseado en un formato que el server puede entender. Este método simplemente envía vía socket dicho string.

3.1.3.2 ReceiveMEssage

Recibe cualquier mensaje enviado via socket desde el servidor y lo almacena hasta que el `ClientGameProxy` se la pida para actualizar su información.
En resumen el `Communicator` encapsula el envío y recepción de mensajes via socket, para poder hacerlo en paralelo. El protocolo de comunicación que utilizamos consiste en convertir los bytes a ser enviados a formato big endian, luego se procede a enviar el largo del mensaje en un primer send, seguido de otro send con los datos.

3.1.3.3 Interpreter

Objeto del que se vale el `ProxyGame` para poder parsear los strings recibidos por el Communicator desde el server y de esta forma interpretarlos para reconstruir los objetos del modelo en el lado del cliente. Sus métodos son similares, con la diferencia de que cada uno se encarga de parsear tipos de objetos del modelo distintos. Por ejemplo:

3.1.3.3 deserializeObject

Parsea los distintos campos del objeto, su posicion, id, estado, nivel de tecnología si fuese un edificio y se los asigna a un nuevo objeto proxy del mismo tipo que almacena el server, con la diferencia de que el objeto proxy no tiene ningún comportamiento, simplemente esta para obtener su información.
El resto de los métodos siguen con la misma lógica, reciben un string en un formato determinado, el interpreter los parsea y crea un objeto del tipo análogo al que

tiene el server con la formación extraída de dicho string, para que el `proxyGame` lo agregue a su copia del modelo.

3.2. Servidor

3.2.1. Modulo de aceptación de clientes

3.2.1.1 mainLoop

Este método se encarga de aceptar las nuevas conexiones de los clientes a través del socket de la clase que representa este módulo. Usando el método del socket llamado `accept`, el cual es bloqueante, espera un nuevo cliente. Cuando esto sucede, crea un hilo de comunicación, lo inicia y vuelve a esperar una nueva conexión.

3.2.1.2 stop

Este método se encarga de finalizar la ejecución de todo el servidor. Recorre todos los hilos `ProxyClient` que tiene almacenados en un vector y llama a un método que estos poseen también llamado `Stop` el cual los finaliza correctamente.

3.2.2. Modulo de comunicación con el cliente

3.2.2.1 run

Este método es el que ejecuta el hilo en cuestión Se trata de un loop en el cual la condición de corte es una variable booleana la cual se ve modificada cuando se llama al método `stop()`. En cada iteración se recibe un pedido del cliente y se analiza que es lo que quiere realizar. En caso de que el pedido sea `"GET_MODEL"`, el cual significa que el cliente pide una serialización del modelo, entonces se sabe que inmediatamente se requiere responder enviando la serialización y como ultimo, el mensaje `"end"` para que el cliente sepa cuando dejar de recibir el mismo. Además hay dos tipo de pedido del modelo. El inicial consta del envío de todo el mapa, con sus territorios y objetos para que de esta manera, el cliente pueda dibujar el mapa. Luego, como todo lo referidos a `Tiles` y territorios, no se modifica, no es necesario enviarlos mas.
El otro tipo de envío de modelo es el de las unidades, municiones y objetos, que de hecho son las únicas tres cosas que el cliente necesita conocer para actualizar el dibujo del mapa.
Por último, en caso de recibir el pedido `"CREATE_PLAYER"`, el `ProxyClient` sabe que debe responder inmediatamente al cliente con el id del nuevo jugador creado.
En caso de recibir otro tipo de pedido, como por ejemplo, el de crear cierta unidad, se llama a un método del modelo, el cual se llama `receivePetition()` que se encarga de cumplir con el mismo.

3.2.2.2 SendCommand

Este método recibe un string como argumento y se encarga de enviar al cliente dicho mensaje. Primero envía el largo del mensaje para que el cliente sepa la cantidad de bytes a recibir. Luego envía el string.

3.2.2.3 ReceiveCommand

Se encarga de recibir y almacenar en un string el menaje enviado por el cliente. Primero sabe que va a recibir cuatro bytes que representan el largo del mensaje para que luego almacene en un buffer el mismo.

3.2.2.4 ReturnModel

Este método recibe un vector con el modelo serializado, y se encarga de enviara de a uno cada string de cada posición del vector usando el método `sendCommand()`.

3.2.3. Modulo del modelo servidor

3.2.3.1 CreateRobotGrunt

Este método recibe el id del edificio que pidió la creación de la unidad. De ese modo, esta función se encarga de crear una clase `ServerTaskCreateRobotGrunt` la cual se encarga de ejecutar en su método `execute()` la creación concreta de la unidad. Pero este método será ejecutado cuando el tiempo de espera, hasta que dicha creación se llevada acabo, haya pasado. Esto ocurre así ya que esta clase, que hereda `Task`, tiene un tiempo de espera que conoce y usando la librería de c++ de time chequea que dicho tiempo haya pasado. Este método createRobotGrunt es equivalente a todas las restantes unidades.

3.2.3.2 MoveUntil

Este método recibe la posición en 'x' y en 'y' en forma de píxeles y el id de la unidad que se desea mover. Lo que hace es buscar y obtener a la unidad mediante el diccionario de unidades. Luego llama a uno de los métodos de la misma llamado `goTo()` que devuelve una cola de clases NodePath la cual almacena una posición, y la posición a la cual se está moviendo. Es decir, calcula con el algoritmo `Astar` y genera una cola donde devuelve el camino mínimo a través de estas clases. Luego, el método `MoveUnit` crea las tareas de movimiento `TaskMove` donde se le pasa la clase `NodePath`. Estas tareas se irán cumpliendo cuando si tiempo de espera se haya consumido, el cual depende de la velocidad dela unidad y además es el tiempo acumulado ya que se suma a los tiempos de las tareas anteriores a sí misma. Este tiempo de espera se le pasa como argumento en el constructor.

3.2.3.3 AttackUnit

Este método recibe el id de la unidad que dispara y el id de la unidad que recibirá el disparo. Verifica que en caso de que el atacante esté o no en rango de disparo. En el caso de que no lo esté llama al método moveUntil el cual le genera todas las tareas para que la unidad se mueva hasta estar en rango. Luego crea la tarea `TaskShoot` la cual al ejecutarse la misma, cuando su tiempo de espera se consuma, crea la munición correspondiente a la unidad y genera sus tareas de movimiento pero en este caso para la munición donde el tiempo de espera de cada uno de estos depende de la velocidad de la munición y que cada tarea posee el tiempo acumulado respecto a la anterior. También crea la tarea `ServerAttackUnit` donde su tiempo de espera es mínimamente mayor a la ultima tarea de movimiento de la munición que es la que llega a destino y que cuando se ejecuta le saca vida a la unidad que recibirá el disparo.

3.2.3.4 AttackObject

Este método es equivalente al método `attackUnit` con la diferencia que en vez de buscar el id de una unidad como objetivo, busca el id de un objeto en el mapa de objeto del mapa.

3.2.3.5 Update

Esta función se encarga de actualizar el modelo. Se encarga de iterar todas las unidades y preguntar si esta murió. En ese caso la agrega a un diccionario de unidades muertas y continua con la siguiente unidad. Verifica si la unidad está parada sobre una bandera, donde en ese caso cambio de dueño a su favor. Luego se fija en la cola de prioridad de tareas a cumplir de la unidad y se fija se la tarea de mayor prioridad, que sería la que tiene menor tiempo de espera, es ejecutable. Es decir, si su tiempo de espera se consumió. En caso positivo, la desencola y la ejecuta modificando así el estado de la unidad. Luego monitorea el estado de la unidad donde en caso de estar quieta, verifica si hay enemigos en rango, si está atacando unidades u objetos, verifica si murieron. Repite el mismo proceso para las municiones donde refresca las tareas de las mismas y en caso de que la munición haya impactado la agrega a un diccionario de municiones obsoletas. Y para los objetos verifica si están rotos. En caso de ser piedras o bloques de hielo, lo agrega a un diccionario de objeto rotos. En caso de que un fuerte esté roto, muestra la bandera del territorio asociado al fuerte.

3.2.4. Modulo de serializacion del modelo

3.2.4.1 deserializePetition

Recibe un string que representa un pedido del cliente y lo parsea. Verifica que función del modelo desea llamar y la llama pasándole los argumentos correspon-

dientes que parseó de la petición.

3.2.4.2 InitialSerialize

Este método llama a métodos privados que serializan las dimensiones del mapa, a los nodos, los objetos, los territorios.

3.2.4.3 Serialize

Este método llama a métodos privados que serializan las unidades, los objetos y las municiones.

4. Programas intermedios y de prueba

Para probar el programa en la parte únicamente del cliente, contamos con un proyecto aparte. En el mismo únicamente se prueban las funciones de selección de unidades, movimiento y dibujo del mapa. Esta únicamente destinado a testear el correcto dibujo de unidades y ejecución de animaciones, dado que es la primera vez que utilizamos SDL.

En el caso del server se "hardcodeó" el mapa y ciertos parámetro para llevar a cabo el algoritmo de Astar.