Java Untersee

Proyecto de innovación

Grado en Desarrollo de Aplicaciones Multiplataforma

I.E.S. Valle del Jerte

2º DAM
30 de mayo de 2019

Sebastián Lavigne Kálmar
76121350K

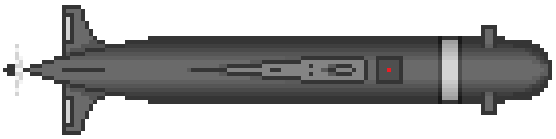
Índice

1. Descripción

Java Untersee es una simulación de submarinos desarrollada en *Java* con un motor de juego propio basado en la librería *Swing*.

Toma inspiración de varios juegos de simulación y estrategia, así como de varios conceptos vistos a lo largo del ciclo y en las prácticas de empresa.

Está publicado en la plataforma *GitHub*[1] bajo la Licencia Pública General de GNU, *GPLv3*[2].



[1] Alojado en el repositorio https://github.com/SebLavK/java-untersee
[2] Más información sobre la licencia en https://www.gnu.org/licenses/gpl-3.0.en.html

2. Objetivos

El proyecto se puede definir como un conjunto de distintos desafíos de programación, siendo cada uno un objetivo surgido de forma orgánica durante su desarrollo. El objetivo principal consiste, entonces, en integrar los distintos componentes creados a partir de estos desafíos para presentar un juego con una experiencia única y original.

Para materializar un programa sólido y expansible se tienen en cuenta las siguientes restricciones:

- Crear componentes avanzados sin recurrir a librerías externas para explorar y comprender diversos conceptos de programación.
- Ahondar en las posibles implementaciones de las librerías que sí están presentes para aprovechar su potencial.
- Construir el software en siguiendo el patrón de arquitectura *Modelo Vista Controlador*.
- Perseguir las buenas costumbres en la estructura y calidad del código.
- Mantener el proyecto alojado, licenciado y documentado en la nube de forma pública para servir de referencia e inspiración a otras personas.
- Trabajar con la metodología *Git Flow* para el control de versiones.

Java Untersee, como juego, no tiene una visión o un estado final definido sino que sirve como aliciente para mejorar y conocer nuevas vías en el mundo del desarrollo de software.

3. Justificación

Este proyecto parte de una tarea del módulo *Desarrollo de Interfaces* cuyo objetivo original era presentar un programa que trabajase con gráficos en tiempo real. Durante su desarrollo comencé a implementar varias características que iban más allá del ámbito de la tarea.

Como se ha visto en los Objetivos, estas características presentan desafíos que me impulsaron a descubrir el funcionamiento detrás de procesos informáticos involucrados en tareas como el análisis sintáctico, el renderizado de imágenes o la simulación.

El programa es un cajón de arena en el que probar varias ideas y compartir soluciones con otros programadores. Su estado actual está definido tanto por las características ya implementadas como por conceptos que se han pensado para su futuro. Está destinado con carácter abierto a entusiastas de la programación y la simulación.

4. Descripción técnica

4.1. Estructura del proyecto

El proyecto está construido siguiendo el patrón de arquitectura *Modelo Vista Controlador* [3].

El modelo

Contiene una representación numérica del mundo simulado. Aquí existen los distintos barcos y el submarino del jugador contenidos dentro de un *escenario* que está aislado del resto del programa. Es decir, el modelo no conoce sobre la existencia de la vista, del controlador, o incluso del jugador que influye en su evolución. Como se verá más adelante, existen dentro de él algunas clases que actúan de enlace entre el escenario y el resto del programa. La evolución del modelo a lo largo del tiempo se detalla en la Subsección 4.3 Motor de juego, en la página 13.

La vista

Se encarga de representar gráficamente los datos simulados en el modelo. Se podría decir que la vista sí conoce sobre la existencia del modelo, aunque sólo toma los datos necesarios relativos a la posición de los distintos elementos del juego en cada momento para mostrarlos por pantalla. Su implementación se explica en la Subsección 4.4 Motor gráfico, en la página 15.

El controlador

Es el núcleo del programa. Arranca el juego, cargando los datos necesarios e inicializando la vista y el modelo. Su pieza más importante es el *bucle de juego* (definido en

la Subsección 4.2, página 13), que ordena al modelo a avanzar en el tiempo y a la vista a actualizar la información representada. Actúa como puente para la totalidad del programa a través de clases *híbridas*, o intermedias.

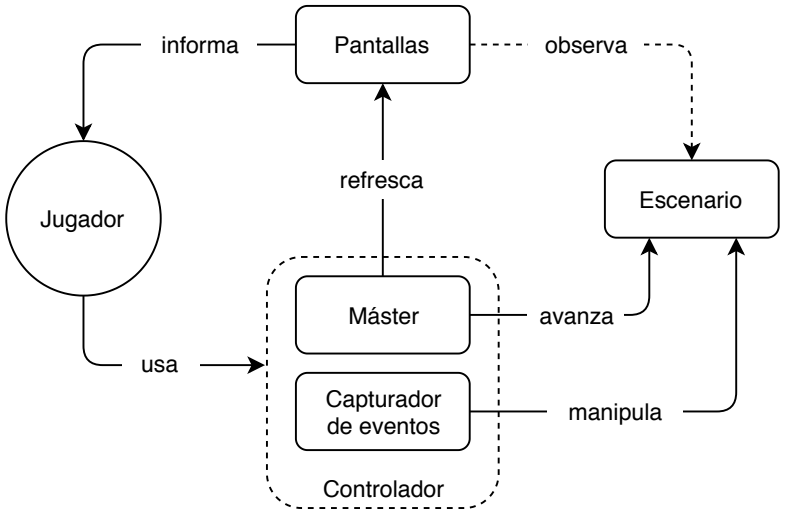


Figura 1: Implementación de la arquitectura *Modelo Vista Controlador*.

Clases híbridas

Para facilitar la interacción entre las distintas secciones del programa, existen clases que cumplen funciones que se extienden más allá del ámbito propio de la vista o el modelo. Por ejemplo, la *Cámara* y el *Segundo al mando* están contenidos en el modelo, pero "son conscientes" del usuario y contienen información que sólo es relevante a la vista y no a la simulación. Su uso se explica respectivamente en las subsecciones 4.4 Motor gráfico (pág. 15) y 4.5 Intérprete de comandos (pág. 18) respectivamente.

Clases comunes

En el programa se encuentran algunas clases que son de uso común para todos los componentes. Son la clase utilidad *Magnitudes*, para la conversión de las distintas unidades usadas en la simulación y la representación gráfica; la clase *Reloj*, para cálculos

temporales relacionados con el tiempo de ejecución del programa; y la clases *Orden* y *Verbose*, que definen una acción a realizar por el submarino, comandada por el jugador, y su reconocimiento verbal por parte de la tripulación.

4.2. Bucle de juego

El bucle de juego está contenido en la clase *Máster*, que toma el control del programa una vez se haya arrancado y se hayan cargado sus componentes. Es la fuerza motriz detrás de la simulación y la visualización. [2, 20]

Consiste en un bucle infinito que manda a la vista repintar la pantalla, avanzar temporalmente la simulación y esperar un tiempo determinado antes de la siguiente iteración. Este tiempo de espera varía entre cada iteración dependiendo del tiempo transcurrido en la ejecución de la simulación, que tiende a ser constante, y la representación gráfica, que es mayor o menor en función de los objetos que haya en pantalla.

El bucle busca iterar con una frecuencia de 60 veces por segundo, o cada 16 milisegundos. Cada iteración se guarda la hora del sistema antes y después de las fases de simulación y gráficos, y se calcula el tiempo de espera en base a su diferencia. Esta tasa de refresco está definida en la clase *Reloj* para poder ser ajustada fácilmente.

4.3. Motor de juego

El paso del tiempo

Por cada iteración del bucle de juego se ejecuta un paso en la simulación. El motor de juego tiene en cuenta el tiempo que dura cada “tic”, y por tanto siempre evoluciona igual en tiempo real independientemente de la frecuencia del bucle. Es decir, un barco moviéndose por la pantalla a una velocidad fija siempre recorrerá la misma distancia en el mismo tiempo desde el punto de vista del jugador. Modificar el valor de refresco

en la clase *Reloj* sólo alterará la fluidez y la precisión de la simulación.

El escenario

La simulación gira en torno a la clase *Escenario*. Ésta contiene el submarino del jugador, una lista de barcos y una lista de proyectiles. Sus funciones consisten en repercutir la orden de avanzar un "tic" en todos los objetos del mapa, y eliminarlos si han sido marcados como *destruido*.

La clase Navío

La clase abstracta *Navío* es la base de todos los objetos del juego. Contiene las propiedades de posición, dirección y velocidad actual y deseada de un navío, y sus capacidades de aceleración, velocidad máxima y relación de viraje.

También contiene métodos que modifican los valores actuales por cada "tic" en función de los valores deseados y capacidades del barco, y métodos de cálculo de posición y dirección relativas a otros objetos.

Cada barco y proyectil del juego heredan de esta clase y definen sus propios atributos, o modifican y añaden sus métodos para definir sus propias habilidades.

Física de los navíos

El movimiento de los objetos en el mapa procura ser fiel a la realidad, siguiendo patrones básicos de mecánica clásica y dinámica naval [10, 11, 12]. Esto da al jugador la sensación de estar controlando un navío de gran tamaño, debiendo tener en cuenta el tiempo de respuesta para tomar decisiones por adelantado. Al mismo tiempo, el código procura hacer un balance entre la complejidad del movimiento y la jugabilidad.

Todos los objetos del escenario están contenidos en un espacio bidimensional, con

la propiedad adicional de *profundidad*, que está fijada en cero para todos los barcos con la excepción del submarino del jugador. Esto significa que los cálculos de posición relativa siempre están contenidos en el plano, sin tener en cuenta la tercera dimensión que añade la profundidad.

Los cálculos en el motor de juego se realizan con unidades marítimas.

4.4. Motor gráfico

El proyecto cuenta con un motor gráfico propio para representar la simulación. Usa la librería *Swing* como base para soportar la ventana y los paneles en los que dibujar las imágenes. La vista cuenta con varias clases para representar información en los distintos paneles, siendo la *Pantalla de Mapa* la principal. [4, 21, 27]

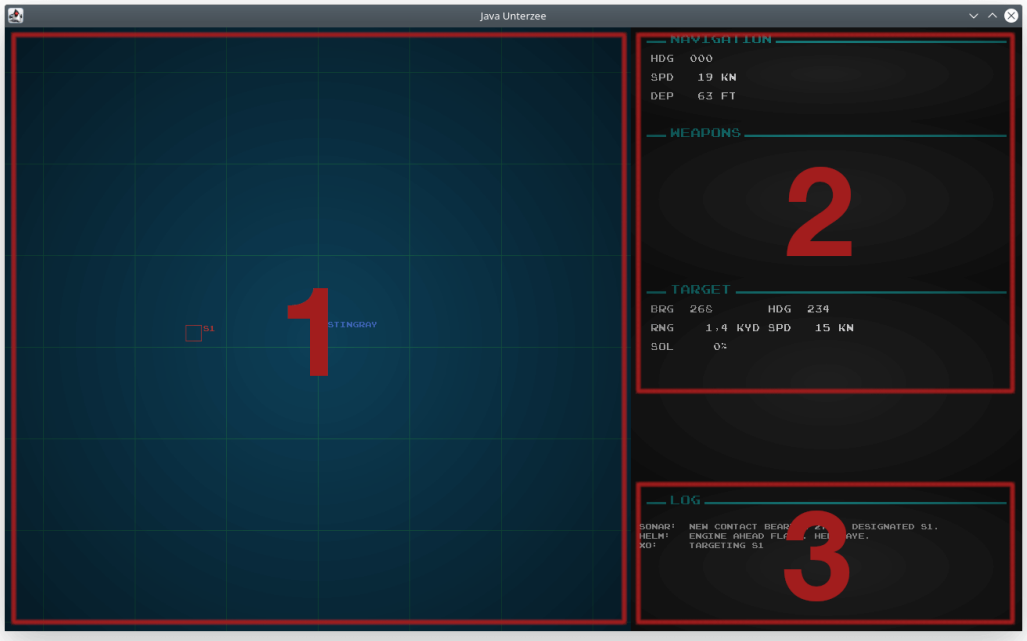


Figura 2: La ventana del juego.1. Pantalla de mapa. 2. Pantalla de información. 3. Consola del jugador.

La cámara

El objeto *Cámara* es una de las piezas principales en el momento de visualizar el juego por pantalla. Su clase está incluida en el modelo y hereda de *Navío*, por lo que respecto a la simulación es un objeto que se encuentra en el plano junto a los demás barcos. De esta manera puede acceder a los mismos métodos de posición relativa a otros navíos. Posee el atributo propio de zoom, que no es relativo a la simulación pero que es esencial para la vista.

La cámara también admite las acciones del jugador que recibe del controlador. El jugador puede cambiar el nivel de zoom, enfocarse en el submarino o mover la cámara a otros puntos del mapa usando el ratón.

La pantalla de mapa

La pantalla de mapa se encarga de representar visualmente la simulación desde un punto de vista cenital. Sitúa la cámara en el centro del panel y dibuja el mar los barcos calculando sus coordenadas en pantalla a partir de la posición relativa en el plano. Para traducir las unidades marítimas que utiliza el juego a píxeles en la pantalla se usa la clase utilidad *Magnitudes*.

En función del zoom que tenga la cámara en cada momento se pinta la pantalla de una forma u otra. Para niveles de zoom cercanos se muestra el juego en una vista “real”:

- Se rellena el fondo con las casillas del mar animadas. El número y tamaño de casillas a mostrar se calcula en base al zoom de la cámara de forma que ocupen el panel.
- Se superpone una imagen con transparencia que simula el brillo del sol en el mar.
- Se dibuja el submarino, el resto de barcos y proyectiles presentes en el escenario. El tamaño de su sprite y su posición relativa al centro de la pantalla dependerán del zoom y de su posición relativa a la cámara en la simulación. El submarino

cambia a colores más o menos oscuros en función de su profundidad.

Para niveles de zoom lejanos se muestra el juego en una vista “estratégica”:

- El fondo toma un color fijo y se pintan líneas de cuadrante que representan una distancia de una kiloyarda.
- Se marca la posición del submarino y de los barcos con un cuadrado y el nombre con el que hayan sido designados por el sonar en el juego.
- Se superpone una imagen con transparencia que simula la pantalla de un viejo monitor.

La cámara tiene definidos los niveles de zoom en los que se pinta el juego de una manera u otra, así como el rango de transición de un modo a otro, en el que se mezclan las dos vistas usando transparencias.

Figura 3: Comparación de la vista *real* (izquierda) y la vista *estratégica* (derecha).

4.5. Intérprete de comandos

El intérprete es la pieza central de la interfaz con el jugador. Cuando se introduce un comando por teclado, un parseador lo traduce a una orden que el intérprete puede ejecutar. [6, 7, 13, 25]

El Segundo al mando, parte I

El *Segundo al mando* es el intérprete en sí. Es una clase híbrida que forma parte del modelo y que recibe los comandos escritos por el jugador. Su ejecución se realiza fuera del bucle de juego en el hilo de despacho de eventos de *Java AWT* [8, 19] cuando se captura la tecla intro en la consola.

Esta clase contiene métodos de tipo *Consumidor*, es decir, métodos que aceptan un parámetro y no devuelven nada. Este parámetro es una *Orden* de un tipo genérico. El por qué del tipo genérico de hará evidente más adelante, al definir el objeto *Orden* y ver qué hace el *Segundo al mando* con él.

Los métodos consumidores tienen una referencia al submarino del jugador y cada uno contiene instrucciones para alterar sus atributos como, por ejemplo, su velocidad o rumbo deseados.

Al recibir un comando, se pasa al *Parseador* y se pide que genere una *Orden*.

La clase Orden

A partir de esta clase se instancian objetos que contienen los datos necesarios para la realización de un cambio por parte del *Segundo al mando*. Es una clase paramétrica, de forma que al crear un nuevo objeto debe establecerse un tipo genérico. Como atributos contiene:

verbo Una referencia a un método consumidor de *órdenes del mismo tipo genérico que la clase*.

objeto Un objeto *dato* del tipo genérico de la clase.

verbose Un objeto *Verbose* con el mensaje que se mostrará al jugador en pantalla. Su funcionamiento se detalla en la Subsección 4.6 Propiedades e internacionalización, en la página 21.

El Segundo al mando, parte II

Cuando el *Segundo al mando* recibe la *Orden*, obtiene de ella la referencia al método consumidor que debe ejecutarse. El consumidor *acepta* la orden como argumento y recorre su código. El objeto *dato* contenido en la orden es la variable que define el comportamiento a seguir por el submarino. Por ejemplo, un doble que establece el nuevo rumbo a tomar.

Al usar tipos genéricos, en el momento de crear una *Orden* se hace obligatorio pasar una referencia a un método que consuma el mismo tipo que la orden. De esta manera, *Java* da un error de compilación si los tipos difieren y se evitan excepciones en tiempo de ejecución. Por ejemplo:

> Al constructor de la *Orden* se pasa una referencia a un consumidor de órdenes tipo entero y un objeto *dato* tipo cadena. Si la clase *Orden* no estuviese parametrizada, el programa compilaría sin errores y el programador no tendría oportunidad de visualizar el fallo hasta que el proceso terminase con una excepción.

El hecho de que un método consumidor reciba la misma *Orden* que lo referencia en vez de simplemente recibir el objeto *dato* da más flexibilidad en las acciones que se pueden realizar. Por ejemplo, el método podría modificar el texto a mostrar en pantalla

en función del estado del submarino.

El Parseador

Es una clase utilidad que recibe una cadena con el comando del jugador, la convierte en una *Orden* y la devuelve al *Segundo al mando* para que la interprete. La Secuencia del intérprete se puede consultar en el Apéndice B, en la página 33.

El *Parseador* contiene un árbol sintáctico, definido usando *mapas de hashes* y referencias a métodos *Runnable*[1]. Estos mapas son constantes y se cargan con el arranque del programa, guardando cadenas como claves y referencias a métodos *Runnables* como sus valores.

Cada nodo del árbol está representado por un método. Los mapas de la clase y los controles de flujo dentro de estos métodos definen las ramificaciones del árbol. Los nodos finales generan una *Orden*, en la mayoría de los casos obteniendo un valor del array para pasarlo en la orden como objeto *dato*.

Antes de comenzar el parseo, como preparación el *Parseador* divide la cadena del comando en un array de palabras.

Como primer paso, se manda al al mapa principal la primera palabra en el array como clave y se obtiene y ejecuta la referencia un método. Los subsecuentes métodos contienen instrucciones propias para procesar el resto de la sentencia. Por ejemplo:

> Dado el comando “velocidad 5”, primero se pide al mapa principal una referencia con la clave “velocidad”. El método que devuelve lee la segunda palabra, en este caso “5”, y crea una nueva *Orden* de tipo doble con una referencia al método *cambiar velocidad*, con el dato 5 y con una respuesta

[1] Es importante destacar que el uso de *Runnable* no implica que se abran nuevos hilos para ejecutar estos métodos. *Runnable* es una interfaz funcional con un método que no recibe parámetros y no retorna nada, por tanto se puede referenciar cualquier otro método que tenga esta misma firma mediante una expresión *lambda* simplificada. [25]

para mostrar por pantalla.

Dado el comando “adelante a un tercio”, se pide al mapa principal una referencia con la clave “adelante”. El método que devuelve esta vez se encarga de obtener una referencia a otro método, ya que para “adelante” existen varias ramificaciones. Pide al mapa de *parseo de “adelante”* una nueva referencia usando el resto de la cadena como clave. El método final genera una *Orden* igual a la anterior, pero con el dato definido para la opción “a un tercio”. Otras opciones incluyen, por ejemplo, “estándar”, “total” o “en flanco”.

Se puede observar el camino de estos ejemplos en la Sección del árbol sintáctico del *Parseador.*, en la página 22.

4.6. Propiedades e internacionalización

Los textos que aparecen en la consola [26] como respuesta a los comandos del jugador o eventos del juego llegan a la clase *Panel Lateral* en forma de un objeto tipo *Verbose*, que contiene las claves de los textos a mostrar. El *Panel Lateral* obtiene los textos literales pidiéndolos a un archivo de propiedades usando dichas claves. Así, se puede configurar la internacionalización de los textos en varios idiomas apuntando a distintos archivos de propiedades, donde las claves se mantienen y los literales están traducidos.

Usando esta misma aproximación, en el futuro se podrán configurar los parámetros que definen el funcionamiento de la simulación mediante ficheros de propiedades, en vez de usar codificación fija.

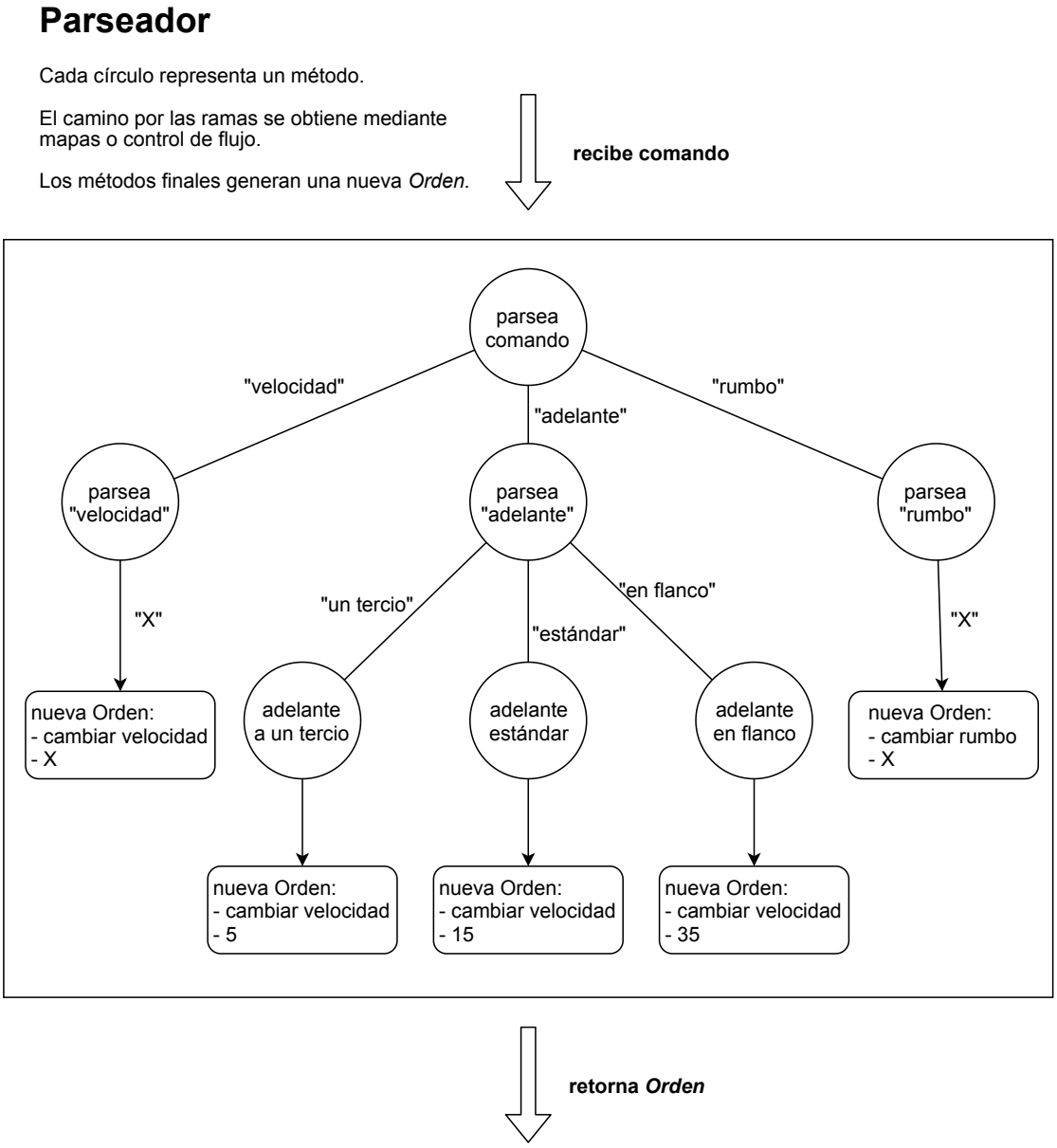


Figura 4: Sección del árbol sintáctico del *Parseador*.

5. Medios utilizados

5.1. Software

Eclipse Entorno de desarrollo de *Java*, utilizado en los principios del proyecto.

IntelliJ IDEA Otro entorno de desarrollo. El proyecto se migró a este IDE a partir de la versión v0.1.

Sonarlint Extensión para IDEs con funciones de análisis de calidad del código.

GIMP Manipulador de imágenes, para crear y retocar algunos de los recursos gráficos.

TeXstudio Editor de LaTeX, utilizado para crear estas mismas memorias y otra documentación.

Atom Editor de código fuente, usado en la generación de documentos *Markdown* para la documentación en *GitHub*.

Git Software de control de versiones distribuidas.

gitflow-avh Extensión de *Git* para el modelo de ramas de Vincent Driessen con funciones añadidas.

GitHub Servicio de almacenamiento de repositorios en la nube.

exe4j Generador de lanzadores para *Windows*.

draw.io Editor web de diagramas y gráficos.

5.2. Recursos

El proyecto cuenta con recursos gráficos usados bajo licencias libres o con permiso de los autores.

Sea Warfare set, ships and more

Sprites para los distintos barcos y el submarino del jugador.

Autor Lowder2

Licencia CC0 1.0 Universal (CC0 1.0) Dedicación de Dominio Público

Dirección https://opengameart.org/content/sea-warfare-set-ships-and-more

The Battle for Wesnoth water animation

Sprites para el mar animado y encasillable.

Autores Zabin y zookeper

Licencia CC0 1.0 Universal (CC0 1.0) Dedicación de Dominio Público

Dirección https://opengameart.org/content/the-battle-for-wesnoth-water-animation

Joystix Monospace

Fuente tipográfica usada en el juego.

Autores Typodermic Fonts

Licencia The Fontspring Desktop Font EULA Version 1.7.0 - February 26, 2017

Licencia The Fontspring Application Font EULA Version 1.7.2 - May 14, 2018. Usada con fines académicos bajo permiso del autor.

Dirección https://opengameart.org/content/the-battle-for-wesnoth-water-animation

USS Columbus (SSN 762) performing an emergency ballast blow

Foto de un submarino mostrada al arrancar el juego.

Autores U.S. Navy photo by Photographer's Mate 2nd Class David C. Duncan

Licencia Etiqueta de Dominio Público 1.0) Dedicación de Dominio Público

Dirección Wikimedia Commons

USS Devilfish (SS-292) being sunk as a target by Wahoo (SS-565) at San Francisco, CA.

Foto de un barco siendo atacado usada en el póster.

Autor US Navy

Licencia Etiqueta de Dominio Público 1.0) Dedicación de Dominio Público

Dirección Wikimedia Commons

Vladivostok Submarine S-56 Control room

Foto del interior de un submarino usada en el póster.

Autor Alexxx1979

Licencia Reconocimiento-CompartirIgual 4.0 Internacional (CC BY-SA 4.0)) Dedicación de Dominio Público

Dirección Wikimedia Commons

6. Dificultades

6.1. Rendimiento

La mayor carga en el rendimiento del programa es la representación gráfica. La librería *Swing* no está optimizada para el tipo de actualización de gráficos en tiempo real que requiere el proyecto. Como se explicó en la Subsección 4.2 Bucle de juego, en la página 13, el tiempo de repintado varía considerablemente en función de los elementos a pintar.

Sobre todo, la fluidez del programa sufre mucho en el periodo de transición de la vista *real* a la vista *estratégica* en la pantalla del mapa, ya que se deben realizar los cálculos y los pintados de ambos modos de visualización.

Además, el pintado de los elementos se hace secuencialmente en la misma pantalla. Es decir, los elementos se van añadiendo uno a uno en el lienzo que ve el jugador. Estas acciones no están sincronizadas con el refresco de la salida de vídeo y da lugar a animaciones menos fluidas.

Una solución a este problema, no implementada aún en el proyecto, consiste en generar una imagen en el trasfondo con la que actualizar la pantalla en una única acción [23].

6.2. Código legado y refactorización

Pese a haber tenido un periodo de desarrollo relativamente corto, volver a un código escrito en el pasado y que había seguido otros objetivos supuso un pequeño obstáculo en la continuación del proyecto.

El estudio, comprendimiento y re estructuración del programa se convirtió en un ejercicio en buenas maneras y preparación del código para cualquier agregación futura,

independientemente de quien haga la aportación. [9]

6.3. Estructura del parseador

Una de las limitaciones de la clase *Parseador* es la necesidad de comenzar el análisis de un comando por, y sólo por, la primera palabra. Con el conjunto de comandos que se pueden ejecutar ahora no supone una traba. Sin embargo, añadir frases más complejas incrementaría el tamaño del árbol de forma poco lógica y crearía ramificaciones con sin sentido si se sigue el funcionamiento del parseador actual.

Quizá un primer paso más analítico, que use listados de palabras y expresiones regulares para recrear la estructura sintáctica de la frase permitiría que el jugador usase comandos más “humanos” para comunicarse con el juego.

Referencias

[1] *Java™ Platform, Standard Edition 8 API Specification*

[2] Koen Witters, *The Game Loop*

[3] tutorials point, *Design Patterns - MVC Pattern*

[4] Peter Mikhalenko, *Learn what the Java 2D API graphics package offers*, 24 octubre 2007

[5] Venkat Subramaniam, *Get a Taste of Lambdas and Get Addicted to Streams by Venkat Subramaniam*, 11 noviembre 2015

[6] Esteban Herrera, *Java 8 Method Reference: How to Use it*, l18 enero 2017

[7] Luis Santiago, *How to start working with Lambda Expressions in Java*, 23 diciembre 2017

[8] The Java™ Tutorials, *The Event Dispatch Thread*

[9] Feathers, M., (2004), *Working Effectively with Legacy Code*

[10] Subhodeep Ghosh, *Understanding Different Types Of Manoeuvres of a Vessel*, 30 septiembre 2017

[11] Discover Boating, *Sailing Basics: 10 Nautical & Sailing Terms To Know*, 23 mayo 2016

[12] Ships business, *Turning circle diameter for a Container ship*, 2009

Baeldung

[13] baeldung, *Lambda Expressions and Functional Interfaces: Tips and Best Practices*, l12 mayo 2019

[14] baeldung , *Guide to the Diamond Operator in Java*, 7 mayo 2019

[15] baeldung, *Java 8 Stream findFirst() vs. findAny()*, 23 agosto 2018

[16] baeldung, *Operating on and Removing an Item from Stream*, 7 mayo 2019

[17] Alejandro Ugarte, *Guide to Stream.reduce()*, 8 marzo 2019

[18] baeldung, *Getting Started with Java Properties*, 7 mayo 2019

[19] baeldung, *Avoiding the ConcurrentModificationException in Java*, 7 mayo 2019

stack overflow

[20] William Morrison, *Java Main Game Loop*, 16 agosto 2013

[21] mthmulders, *getResourceAsStream filepath while running .jar*, 15 abril 2015

[22] Varios, *Why are static variables considered evil?*, 11 agosto 2011

[23] Consty, *Java2D Performance Issues*, 13 octubre 2008

[24] Edwin Dalorzo, *why polymorphism doesn't treat generic collections and plain arrays the same way?*, 30 mayo 2012

[25] Eran, *functional interface that takes nothing and returns nothing*, 30 mayo 2014

[26] MC10, *How to display a Custom Font Text in a JFrame in java?*, 18 diciembre 2015

[27] Yodabyte, *How to control rendering quality using JAVA*, 30 marzo 2016

[28] Ognyan, *Checking "instanceof" rather than value using a switch statement*, 2 febrero 2015

A. Lista de comandos

Comandos básicos de navegación	
speed *X*	Establece la velocidad deseada a ***X*** nudos.
heading *X*	Establece el rumbo deseado a ***X*** grados.
depth *X*	Establece la profundidad deseada a ***X*** pies.

Comandos estándar de velocidad	
ahead flank	Establece la velocidad a **35 nudos**. Esta es la velocidad máxima.
ahead full	Establece la velocidad a **20 nudos**.
ahead standard	Establece la velocidad a **15 nudos**.
ahed 2/3	Establece la velocidad a **10 nudos**.
ahead 1/3	Establece la velocidad a **5 nudos**.
all stop	Para motores. Establece la velocidad a **0 nudos.**
back 1/3	Establece la velocidad a **4 nudos marcha atrás**.
back 2/3	Establece la velocidad a **8 nudos marcha atrás**.
back full	Establece la velocidad a **12 nudos marcha atrás**.
back emergency	Establece la velocidad a **16 nudos marcha atrás**.

Comandos de profundidad	
depth *X*	Establece la profundidad deseada a ***X*** pies.
surface boat	Hace surgir el submarino a la superficie.
periscope depth	Lleva el submarino a profundidad de periscopio (**63 pies**).
emergency dive	Sumerge el submarino lo más rápido posible.
emergency blow	Soplar lastres y llegar a la superficie lo más rápido posible.

Comandos de armas y objetivos	
target *X*	Establece ***X*** como objetivo, siendo ***X*** la designación que *Sonar* da a un barco mostrado en el mapa.
launch tube *X*	Lanza el torpedo cargado en el tubo X (1-4).

B. Secuencia del intérprete

1. El jugador escribe un comando en la consola del *Panel Lateral* y pulsa intro.

2. El controlador captura el evento y manda el texto introducido al *Segundo al mando*.

3. El *Segundo al mando* pide al *Parseador* traducir el comando.

4. El *Parseador*:

    a) Divide el texto y genera un array de palabras.
    b) Obtiene un método que ejecutar llamando al mapa principal con la primera palabra del array como clave.
    c) Recorre el árbol sintáctico siguiendo las sucesivas instrucciones de los métodos pasados por referencia.
    d) Llega a un método final, que genera una *Orden* que contiene la referencia a un *Consumidor* del *Segundo al mando* (el *verbo*), el dato relevante al método consumidor y un *Verbose* con la respuesta que se mostrará por pantalla.
    e) Retorna la *Orden* generada al *Segundo al mando*.

5. El *Segundo al mando* obtiene la referencia a uno de sus métodos consumidores mediante el *verbo* y manda *aceptar* la orden.

6. El consumidor recibe la *Orden* como argumento y ejecuta sus instrucciones usando, en la mayoría de las ocasiones, el dato contenido en la orden.

7. Finalmente, el *Segundo al mando* manda el *Verbose* al *Panel Lateral* para mostrar la respuesta en la consola.

C. Registro de versiones

Changelog

v0.2

- Reached v0.2!

v0.1.5

- Created a torpedo sprite based on the submarine sprite
- Tweaked gitignore

v0.1.4

- More refactoring
- Project now resembles MVC architecture a lot more
- Implemented command history

v0.1.3

- Replaced some loops with stream pipelines
- Minor refactoring

v0.1.2

- Ingame text now loads from a property file
- Created Configuration class to load and read file
- Created Verbose class and modified Order to use new property file
- Migrated game console log to this system

v0.1.1

- Major refactoring across whole project
- Parser is now a static class
- Added GNU GPLv3 License

v0.1

- Reached v0.1!

v0.0.12

- Submarine can now launch torpedoes
- Added intro screen with fading in and out animation

v0.0.11

- Added more ships to starting scenario
- Added targeting command
- Information of targeted ship now appears on side panel
- Started weapon and projectile structure
- Added Sonar class for the submarine

v0.0.10

- Added camera panning, now you can choose between following the sub or free roaming
- Smoothed zooming
- When zooming out the map changes background into a “strategy” view

- Transition between close zoom and far zoom, dynamic change of vessel display mode
- Added a grid to the strategy map
- Added some transparent images for beautiness, both to map screen and side panel

v0.0.9

- Added antialiasing
- Started work on the data screen
- Vessels can now dive!
- Submarine sprite changes to a bluer color the deeper it is
- Added depth related commands

v0.0.8

- Various bugfixes and a few changes to crew messages

v0.0.7

- Added side panel with data and command section*s
- Commands are now given through the game's command license
- Crew messages now appear in game loading
- Map is now zoomable by scroll wheel

v0.0.6

- Perfected ship turning dynamics
- Added camera class, all vessels are displayed in the map relative to it. Can free roam or follow the player submarine
- Added scenario class for game object creation and handling
- Added classes for each kind of ship

v0.0.5

- Map zoom now uses floating point variable
- All sprites scale accordingly when zooming the map
- Vessel speeds are now more realistic
- Better control of sleep time in between frames
- Player gets notified after certain events (e.g. reached intended speed)

v0.0.4

- Added wavy sea background
- Added image assets
- Map is now zommable

v0.0.3

- Parametrized orders for safety in interactions between parser and interpreter

v0.0.2

- Added a Parser for commands interpreted by the Executive Officer

v0.0.1

- A working prototype!