UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ETSI Aeroespacial y Diseño Industrial

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial

Desarrollo de un simulador aeronáutico en Java con integración de NASA World Wind como motor gráfico

Trabajo Fin de Máster

Máster Universitario en Ingeniería Aeronáutica

AUTOR/A: Alfonsín Espín, Gabriel

Tutor/a: Rodas Jordá, Ángel

CURSO ACADÉMICO: 2023/2024

Universidad Politécnica de Valencia

DESARROLLO DE UN SIMULADOR AERONÁUTICO EN JAVA CON INTEGRACIÓN DE NASA WORLD WIND COMO MOTOR GRÁFICO

Autor: Gabriel Alfonsín Espín

Tutor: Ángel Rodas Jorda

Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial
Titulación: Máster Universitario en Ingeniería Aeronáutica
Curso Académico: 2023-2024

Índice general

Índice de figuras

Índice de tablas

Resumen

El presente Trabajo Fin de Máster se enfoca en la concepción y desarrollo de un simulador aeronáutico, utilizando prácticas de programación orientadas a la flexibilidad y la colaboración abierta. El núcleo gráfico del simulador se ha implementado con el apoyo de WorldWind, un SDK (Software Development Kit) de código abierto creado por la NASA. WorldWind permite la manipulación y análisis de información espacial en dimensiones bidimensionales y tridimensionales, proporcionando una plataforma potente para la representación realista de aviones, aeropuertos, ayudas a la navegación y servidumbres entre muchos otros, todo ello en un entorno tridimensional dinámico. El simulador abarca diversos ámbitos como la generación de rutas, el perfil vertical de la aeronave o la resolución de conflictos. Debido a la imposibilidad de incluir muchos otros ámbitos una de las piedras angulares de este proyecto es su capacidad de extensión y adaptabilidad. En este sentido, se hace un uso eficaz de interfaces en Java, permitiendo así que el simulador sea altamente modular. Esta aproximación modular asegura que distintos componentes del simulador, como los modelos atmosféricos, puedan ser fácilmente adaptados o reemplazados. Para garantizar la accesibilidad y fomentar la colaboración, el código fuente del proyecto se ha alojado en un repositorio público, invitando a desarrolladores a contribuir con su conocimiento, proponer mejoras y desarrollar nuevas funcionalidades. El desarrollo del simulador se ha realizado en Java, aprovechando su portabilidad y extensa biblioteca de recursos, y se ha llevado a cabo en el entorno de desarrollo Apache NetBeans.

Palabras clave: simulador, NASA WorldWind, extensión, interfaces, Java, modular.

Abstract

This Master Thesis focuses on the conception and development of an aeronautical simulator, using programming practices oriented to flexibility and open collaboration. The graphical core of the simulator has been implemented with the support of WorldWind, an open source SDK (Software Development Kit) created by NASA. WorldWind allows the manipulation and analysis of spatial information in two- and three-dimensional dimensions, providing a powerful platform for the realistic representation of aircraft, airports, navigation aids and easements among many others, all in a dynamic three-dimensional environment. The simulator covers various areas such as route generation, aircraft vertical profile or conflict resolution. Due to the impossibility of including many other areas, one of the cornerstones of this project is its extensibility and adaptability. In this sense, effective use is made of Java interfaces, thus allowing the simulator to be highly modular. This modular approach ensures that different components of the simulator, such as the atmospheric models, can be easily adapted or replaced. To ensure accessibility and encourage collaboration, the project's source code has been hosted in a public repository, inviting developers to contribute their knowledge, propose improvements and develop new features. The development of the simulator has been done in Java, taking advantage of its portability and extensive resource library, and has been carried out in the Apache NetBeans development environment.

Keywords: simulator, NASA WorldWind, extensibility, interface, Java, modular.

Resum

Aquesta Tesi de Màster se centra en la concepció i desenvolupament d'un simulador aeronàutic, utilitzant pràctiques de programació orientades a la flexibilitat i la col·laboració oberta. El nucli gràfic del simulador ha estat implementat amb el suport de WorldWind, un SDK (Software Development Kit) de codi obert creat per la NASA. WorldWind permet la manipulació i l'anàlisi de la informació espacial en dues i tres dimensions, proporcionant una plataforma potent per a la representació realista d'aeronaus, aeroports, ajudes a la navegació i servituds entre moltes altres, tot en un entorn tridimensional dinàmic. El simulador abasta diverses àrees com la generació de rutes, el perfil vertical de l'aeronau o la resolució de conflictes. A causa de la impossibilitat d'incloure moltes altres àrees, un dels pilars d'aquest projecte és la seua extensibilitat i adaptabilitat. En aquest sentit, es fa un ús efectiu de les interfícies de Java, permetent així que el simulador siga altament modular. Aquest enfocament modular garanteix que diferents components del simulador, com els models atmosfèrics, puguen ser fàcilment adaptats o reemplaçats. Per a garantir l'accessibilitat i fomentar la col·laboració, el codi font del projecte s'ha allotjat en un repositori públic, convidant als desenvolupadors a contribuir amb els seus coneixements, proposar millores i desenvolupar noves característiques. El desenvolupament del simulador s'ha realitzat en Java, aprofitant la seua portabilitat i la seua extensa biblioteca de recursos, i s'ha dut a terme en l'entorn de desenvolupament Apache NetBeans.

Paraules clau: simulador, NASA WorldWind, extensibilitat, interfície, Java, modular.

Capítulo 1

Introducción

En el campo de la **aeronáutica**, los **simuladores** desempeñan un papel crucial en la formación, la investigación y el desarrollo. Estos simuladores permiten a los profesionales del sector experimentar con escenarios de vuelo sin riesgos reales, mejorando la seguridad y la eficiencia operativa. Sin embargo, muchos simuladores existentes están diseñados con un **enfoque cerrado** y rígido, lo que limita su adaptabilidad y la capacidad de incorporar nuevas tecnologías y metodologías. Este proyecto busca abordar estas limitaciones ofreciendo una plataforma abierta y modular, donde los usuarios puedan adaptar y extender el simulador según sus necesidades específicas, promoviendo un entorno de aprendizaje y experimentación continua.

Figura 1.1: Simulador Aeronáutico del proyecto.

Durante el **Máster Universitario en Ingeniería Aeronáutica** se pueden cursar al-

gunas materias de interés para el autor por su relación con el *software* como por ejemplo 'Introducción a la supercomputación y el cálculo paralelo' o 'Mecánica de fluidos computacional y experimental'. Despertó especialmente mi interés las prácticas de la asignatura **Sistemas de Gestión de Vuelo por Computador**, las cuales consisten en diferentes desarollos en **Java**, incluyendo entre muchas otras, una práctica de un pequeño simulador. Estas prácticas sin duda han influido y motivado este proyecto.

Una de las principales motivaciones de este trabajo es la creación de un proyecto de *software* **extensible y adaptable**. A través de la programación orientada a objetos y el uso de interfaces en Java, se ha diseñado una **arquitectura modular** que facilite la adaptación o sustitución de componentes individuales. Aunque todavía queda trabajo por hacer para alcanzar un nivel ideal de modularidad y facilidad de extensión, la estructura actual del simulador permite que, con el tiempo y la colaboración de terceros, se puedan integrar nuevas funcionalidades y mejoras.

Para fomentar dicha colaboración, el código fuente del proyecto se aloja en un **repositorio público**, además de contar con una documentación. Esto no solo facilita el acceso al simulador para desarrolladores e investigadores, sino que también invita a la comunidad a contribuir con sus conocimientos, proponer mejoras y desarrollar nuevas características. El proyecto se puede copiar, manipular y redistribuir libremente.

Este trabajo fin de máster se organiza en varios capítulos que detallan los distintos aspectos del proyecto. Este apartado incluye la introducción y la motivación del proyecto. A continuación, el **Capítulo 2** describe los objetivos perseguidos y requesitos a cumplir, seguido por el **Capítulo 3**, que detalla la metodología seguida para alcanzar dichos objetivos. El estado del arte se aborda en el **Capítulo 4**, profundizando en las soluciones o proyectos actuales y sus ventajas e inconvenientes. Las funcionalidades implementadas y los componentes modulares se explican en el **Capítulo 5**. En el **capítulo 6** se profundiza en la interfaz gráfica de usuario. Posteriormente, el **Capítulo 7** muestra un simple caso de uso. Finalmente, el **Capítulo 8** concluye con un resumen de los logros alcanzados y las posibles direcciones futuras para el proyecto.

Capítulo 2

Objetivos y Requisitos

2.1. Objetivos

En un contexto donde el sector del *software* está creciendo a un ritmo sin precedentes, con innovaciones que transforman todos los aspectos de la industria y la academia, es crucial que las ingenierías tradicionales adopten el *software* como un **componente central**. Este proyecto nace de la necesidad de integrar estas habilidades en la formación y práctica de la ingeniería aeronáutica, demostrando que el *software* no debe ser un accesorio, sino una pieza nuclear en el diseño y desarrollo de soluciones complejas. Fruto de esta necesidad surgen los siguientes objetivos:

- Crear un **simulador aeronáutico** capaz de simular el comportamiento de aviones en distintas situaciones. Diseñar toda la infraesctructura que defina los comportamientos básicos de un simulador.

- Diseñar un simulador que sea **extensible y flexible** en contraposición a una solución cerrada y estática. Es decir, que sea posible para el usuario añadir funcionalidades en el futuro de manera autónoma.

- En el ámbito educativo, se busca que este simulador pueda ser utilizado como **material práctico y docente** para proporcionar a los alumnos las herramientas y el conocimiento necesarios para diseñar aplicaciones informáticas, aplicando los conceptos fundamentales sobre los que se basa este simulador.

Es tan crucial definir los objetivos de un proyecto como especificar sus límites y **acotar sus funcionalidades**. En este proyecto, no se pretende emplear los modelos teóricos más avanzados o detallados, sino establecer una base sólida que permita a otros incorporar dichas complejidades en el futuro. Por esta razón, en algunas ocasiones, se recurrirá a simplificaciones de comportamientos complejos.

2.2. Requisitos

Para cumplir con los objetivos establecidos, es esencial definir una serie de **requisitos** que concreten y detallan las funcionalidades y características que el sistema debe poseer. Los requisitos actúan como una guía precisa y medible de lo que se espera del simulador, asegurando que todos los aspectos necesarios para alcanzar los objetivos del proyecto sean considerados y desarrollados de manera estructurada. A continuación, se presentan los requisitos identificados, divididos en requisitos funcionales y no funcionales.

Los requisitos funcionales describen las funciones específicas que el sistema debe proporcionar. Estos requisitos detallan las acciones y comportamientos que el sistema debe ser capaz de realizar para cumplir con los objetivos del proyecto. Por otro lado, los requisitos no funcionales describen cómo debe comportarse el sistema y definen las cualidades y restricciones bajo las cuales debe operar, como el rendimiento, la usabilidad y la confiabilidad.

1. **Requisitos Funcionales**
    - El sistema debe permitir la **visualización** en tiempo real de los escenarios de vuelo mediante una interfaz gráfica interactiva.
    - El sistema debe permitir exportar las variables de la simulación para **postprocesar** sus resultados.
    - El simulador debe permitir la incorporación de nuevos algoritmos y modelos a través de una arquitectura modular basada en **interfaces**.
    - El sistema debe ser capaz de leer y procesar **escenarios** definidos en archivos CSV.
    - El usuario debe poder interactuar con el simulador mediante **comandos** específicos, facilitando la modificación de parámetros y condiciones de vuelo en tiempo real.
2. **Requisitos No Funcionales**
    - El simulador no debe de usar ningún **software comercial** que incurra en costes para el usuario.
    - El simulador debe ser **compatible** con múltiples plataformas, incluyendo Windows, macOS o Linux.
    - El código fuente del simulador debe estar disponible en un **repositorio público** para fomentar la colaboración y la extensibilidad.
    - El sistema debe estar **documentado** adecuadamente para facilitar su uso y extensión por parte de otros desarrolladores e investigadores.
    - El sistema debe ser capaz de manejar varios vuelos simultáneamente sin pérdida significativa de rendimiento.
    - El simulador debe ser **robusto** y capaz de manejar errores y excepciones sin interrumpir el funcionamiento general del sistema.

Capítulo 3

Metodología

3.1. Herramientas y tecnología

En esta sección se describen las herramientas y tecnologías utilizadas durante el desarrollo del simulador aeronáutico. Todas las elecciones tienen sus ventajas e inconvenientes pero son, todas ellas, sumamente conocidas en la industria del software.

3.1.1. Lenguaje de programación

Para el desarrollo del simulador se ha elegido **Java** como lenguaje de programación principal. Java es un lenguaje compilado, lo que proporciona una mayor eficiencia en tiempo de ejecución y una mejor gestión de recursos en comparación con los lenguajes interpretados [1]. Además, Java soporta completamente la programación orientada a objetos (POO), lo que permite un diseño modular y reutilizable del software, facilitando el mantenimiento y la escalabilidad del proyecto. En comparación con C++, Java es considerado más amigable para los desarrolladores debido a su gestión automática de memoria (recolección de basura) y su sintaxis más sencilla y menos propensa a errores.

Como se describe en el capítulo 4, otros lenguajes de programación como C/C++ o Python son perfectamente válidos para fines similares ya que cada uno tiene sus ventajas e inconvenientes. Además, lo habitual en herramientas de escala mucho más grande que este proyecto, es combinar varios lenguajes de programación.

Figura 3.1: Logo del lenguaje de programación java.

Otro motivo de gran peso para la elección de java es que es el lenguaje de programación utilizado en las prácticas de la asignatura *Sistemas de Gestión de Vuelo por Computador*, las cuales imparte el tutor de este trabajo.

3.1.2. Nasa WorlWind

Este simulador utiliza la librería **NASA WorldWind** para la implementación de la interfaz gráfica. NASA WorldWind es una plataforma de código abierto que proporciona herramientas poderosas para la visualización y análisis de datos geoespaciales. Gracias a sus capacidades avanzadas y su flexibilidad, WorldWind ha permitido la creación de una interfaz gráfica robusta y dinámica que facilita la interacción del usuario con el simulador.

La elección de NASA WorldWind ha sido fundamental para garantizar una representación visual precisa y detallada del entorno de simulación. Se ha elegido esta librería principalmente por haber sido desarollada en Java y por su **extensa documentación**. Además de tener el respaldo de una institución como la NASA. El capítulo 6 se profundiza en las capacidades de esta librería y su integración con el simulador aeronáutico.

Figura 3.2: Logo de NASA WorldWind.

3.1.3. Entorno de desarrollo

Para el desarrollo del simulador se ha utilizado **Apache NetBeans**, un entorno de desarrollo integrado (IDE por sus siglas en inglés) ampliamente reconocido por su compatibilidad con Java y su facilidad de uso. NetBeans proporciona un conjunto completo de herramientas que permiten la edición de código, la gestión de proyectos, la depuración y el despliegue de aplicaciones Java de manera eficiente. Entre las características más destacadas de NetBeans se incluyen:

- **Editor de Código**: Ofrece autocompletado, refactorización y navegación de código.
- **Depurador**: Permite la ejecución paso a paso del código, la inspección de variables y el seguimiento del flujo de ejecución.
- **Gestión de Proyectos**: Facilita la organización y gestión de los distintos componentes y dependencias del proyecto.
- **Integración con Git**: Soporte integrado para el control de versiones mediante Git.

Figura 3.3: Logo de Apache Netbeans.

3.1.4. Control de Versiones

El control de versiones es una parte esencial del desarrollo de software colaborativo y modular. En este proyecto, se ha utilizado Git como sistema de control de versiones. Git es una herramienta distribuida que permite a los desarrolladores trabajar de manera independiente y fusionar sus cambios de manera eficiente. Las principales ventajas de utilizar Git en este proyecto incluyen:

- **Rastreo de Cambios**: Permite un seguimiento detallado de los cambios realizados en el código a lo largo del tiempo.
- **Colaboración**: Facilita el trabajo en equipo mediante ramas y fusiones, permitiendo a múltiples desarrolladores trabajar en paralelo. Funcionalidad poco explotada ya que este proyecto ha sido desarollado únicamente por el autor.
- **Reversión de Cambios**: Habilidad para revertir a estados anteriores del proyecto en caso de errores o problemas.

El repositorio del proyecto se ha alojado en GitHub, una plataforma en la nube para alojar repositorios Git. GitHub proporciona una interfaz web amigable y una serie de herramientas que facilitan la gestión de proyectos, el seguimiento de problemas (issues), la revisión de código y la integración continua. Alojar el proyecto en GitHub no solo asegura que el código esté respaldado y accesible desde cualquier lugar, sino que también fomenta la colaboración abierta. Otros desarrolladores pueden contribuir con mejoras y nuevas funcionalidades o simplemente descargar y usar el simulador.

GitHub es ampliamente utilizado en la industria del software por sus capacidades de colaboración y su integración con otras herramientas de desarrollo. Alojar el repositorio en GitHub permite aprovechar estas características y facilita la apertura del proyecto a terceras personas. Los interesados pueden acceder al repositorio del proyecto a través del siguiente enlace: **https://github.com/oplaco/Simulator**

Figura 3.4: Logo sistema de control de versiones git.

3.2. Prácticas de programación

3.2.1. Programación orientada a objetos (POO)

La programación orientada a objetos (POO) es un paradigma de programación que utiliza objetos para modelar datos y comportamientos. Este enfoque se basa en cuatro principios fundamentales: encapsulación, herencia, polimorfismo y abstracción. La POO permite organizar el software de manera modular, facilitando su mantenimiento y ampliación.

- **Encapsulación**: Consiste en agrupar datos y métodos que operan sobre esos datos dentro de una misma unidad, llamada objeto. Esto no solo ayuda a proteger el estado interno del objeto, evitando accesos indebidos, sino que también promueve una interfaz clara y consistente para interactuar con el objeto.

- **Herencia**: Permite crear nuevas clases basadas en clases existentes, heredando atributos y métodos. Esto fomenta la reutilización de código y establece una jerarquía clara entre clases, donde las clases derivadas pueden extender o modificar el comportamiento de sus clases base.

- **Polimorfismo**: Facilita el uso de una interfaz común para diferentes tipos de objetos. Mediante polimorfismo, un solo método puede trabajar con diferentes tipos de objetos, permitiendo una mayor flexibilidad y la capacidad de intercambiar componentes sin alterar el código que los utiliza.

- **Abstracción**: Se refiere a la capacidad de definir clases abstractas que proporcionan una interfaz común pero no implementación concreta. Las clases derivadas implementan los detalles específicos, lo que permite diseñar sistemas más generales y adaptables.

En el contexto de la POO, una clase es una plantilla o modelo que define las características y comportamientos de los objetos que pertenecen a esa clase. Una clase puede incluir atributos (variables de instancia) y métodos (funciones o procedimientos). Por ejemplo, en este proyecto, la clase `TrafficSimulated`, que representa un avión, define atributos como velocidad o rumbo, y métodos como `setAltitude()` o `setSpeed()`.

Un objeto es una instancia de una clase. Es una entidad concreta que contiene valores específicos para los atributos definidos en la clase y puede ejecutar los métodos de la clase. Siguiendo el ejemplo anterior, un objeto `traffic` podría ser una instancia de la clase `TrafficSimulated` con velocidad 300 nudos y rumbo 80º.

3.2.2. Interfaces y extensibilidad

Una interfaz en Java es una referencia abstracta que define un conjunto de métodos **sin implementar ninguno** de ellos. Las interfaces establecen un **contrato** que las clases concretas deben cumplir, lo que promueve un diseño más flexible y extensible. Por ejemplo, una interfaz `Volable` podría definir métodos como `volar()` y `aterrizar()`, y cualquier clase que implemente esta interfaz estaría obligada a proporcionar una implementación para estos

métodos.

Las interfaces son fundamentales para lograr la extensibilidad y modularidad en el desarrollo de software. En este proyecto se ha hecho un uso extenso de las mismas (ver figura 5.2). Al diseñar sistemas utilizando interfaces, los desarrolladores pueden definir comportamientos genéricos que diferentes clases pueden implementar de diversas maneras. Esto permite desacoplamiento, flexibilidad y reutilización del código.

Capítulo 4

Estado del arte

El estado del arte es una revisión de las soluciones y tecnologías actuales con algún de grado de relación con este proyecto. En este caso, el objetivo es situar nuestro proyecto en el contexto de los simuladores aeronáuticos existentes, evaluando sus características y funcionalidades. Esta revisión es esencial para identificar las fortalezas y debilidades de las soluciones actuales y para justificar la necesidad y las contribuciones de nuestro proyecto.

En este capítulo, intentaremos encontrar y comparar proyectos similares al nuestro, destacando sus características principales. Es importante notar que no siempre será posible encontrar equivalentes exactos, ya que en el ámbito de los simuladores aeronáuticos, a menudo se piensa en programas muy avanzados y conocidos como Microsoft Flight Simulator [2], que no necesariamente comparten los mismos objetivos de investigación y o educativos que nuestro proyecto.

A continuación, presentamos una serie de simuladores aeronáuticos con diferentes características que serán comparados en términos de tecnologías utilizadas, interfaz gráfica, extensibilidad, ventajas e inconvenientes. Se empieza describiendo los principales simuladores existentes para más tarde hablar de proyectos de escala similar a este trabajo fin de máster.

4.1. Principales soluciones

4.1.1. Microsoft Flight Simulator

Microsoft Flight Simulator, desarrollado por Asobo Studio y distribuido por Microsoft, es uno de los simuladores de vuelo más reconocidos y ampliamente utilizados en el ámbito del entretenimiento. Desde su lanzamiento inicial en 1982, la serie ha evolucionado considerablemente, culminando en la última versión, Microsoft Flight Simulator 2020, que destaca por su **realismo gráfico** y extensas capacidades de simulación.

Microsoft Flight Simulator es un producto **comercial** cerrado, diseñado primordialmente para el mercado de consumo general. Su modelo de desarrollo controlado por Microsoft permite la incorporación de tecnologías de vanguardia, como la integración de datos de Bing

Maps y técnicas avanzadas de renderizado, ofreciendo una de las experiencias más inmersivas y visualmente impresionantes en el ámbito de simuladores de vuelo.

Aunque ofrece una gama de add-ons y la posibilidad de personalización a través de contenido de terceros, la capacidad de extensión y modificación del núcleo del simulador es limitada en comparación con plataformas de código abierto. Esto se debe a que el acceso al código fuente y a las funcionalidades internas está restringido, lo que puede limitar la investigación académica o desarrollos personalizados profundos.

Microsoft Flight Simulator ha sido utilizado para diversos propósitos además del entretenimiento, incluyendo entrenamiento preliminar para pilotos y familiarización con la cabina, gracias a sus detallados modelos de aeronaves y condiciones atmosféricas realistas. Sin embargo, su enfoque principal sigue siendo el mercado de videojuegos, donde ha establecido un estándar alto para simuladores de vuelo.

4.1.2. FlightGear

FlightGear es un simulador de vuelo de **código abierto** que destaca por su sofisticación y capacidad para modelar el vuelo y las condiciones ambientales con un alto grado de detalle. Desarrollado por una comunidad global de entusiastas, académicos y profesionales desde 1997, FlightGear ofrece una plataforma extensiva para simulaciones aéreas, entrenamiento de pilotos, investigación de ingeniería y desarrollo de sistemas de control de vuelo.

Una de las características más notables de FlightGear es su modelo de vuelo detallado, que incluye simulaciones precisas de la dinámica de vuelo, sistemas de propulsión y respuesta de la aeronave a diversas condiciones atmosféricas y de emergencia. Esto lo convierte en una herramienta valiosa no solo para aficionados, sino también para investigadores y desarrolladores que buscan un entorno robusto y realista para probar nuevas tecnologías y teorías de vuelo. Para más detalles sobre el uso profesional y educativo de FlightGear, visita la sección de "Professional and educational FlightGear users" en su Wiki.

Otro aspecto destacado de FlightGear es su arquitectura abierta, que permite a los usuarios modificar y extender casi todos los aspectos del simulador. Desde la adición de nuevas aeronaves hasta la modificación de modelos atmosféricos y de motores, FlightGear ofrece una plataforma altamente personalizable que se adapta a una amplia gama de necesidades y proyectos de investigación. Sin embargo, la documentación no explica como añadir modelos o funcionalidades adicionales de manera fácil. De hecho, existen trabajos de terceros que intentan remediar esta situación [3].

A pesar de su utilidad en el ámbito académico, es importante reconocer que FlightGear es mucho más que una herramienta de investigación; también es un proyecto muy ambicioso, casi a la escala de un videojuego, con una comunidad activa que participa en su desarrollo y mejora continua. Esta dualidad entre ser una herramienta de investigación y un pasatiempo hace de FlightGear una entidad única en el mundo de los simuladores de vuelo, proporcio-

nando tanto un entorno serio para estudios científicos como una experiencia inmersiva para los entusiastas del vuelo.

Figura 4.1: Aspecto de FlightGear. Extraído de FlightGear.org

En conclusión, aunque FlightGear puede ofrecer características más acordes para aplicaciones científicas y técnicas (al ser de código abierto), su envergadura y profundidad como proyecto también lo posicionan cerca del espectro de los videojuegos, lo que lo aleja un poco de los objetivos de este proyecto.

4.1.3. X-Plane

X-Plane es un simulador de vuelo avanzado desarrollado por Laminar Research, que se destaca tanto en el ámbito recreativo como profesional. Publicado inicialmente en 1995 por Austin Meyer, X-Plane ha evolucionado para ofrecer una plataforma altamente flexible que permite la creación de aeronaves, escenarios y funcionalidades diversas. Su naturaleza comercial no ha impedido que se forme una **fuerte comunidad** de usuarios y desarrolladores, quienes contribuyen activamente con complementos y mejoras.

Una de las características que diferencia a X-Plane de otros simuladores es su capacidad para predecir el comportamiento de una aeronave basándose únicamente en su geometría, sin necesidad de datos preexistentes sobre su vuelo. Esta funcionalidad es especialmente útil en el diseño aeronáutico, ya que permite simular y analizar el rendimiento de nuevos modelos de aeronaves. Para lograr esto, X-Plane emplea la teoría del elemento de pala, una técnica que discretiza la aeronave en múltiples elementos y calcula las fuerzas aerodinámicas en cada

uno de ellos.

Figura 4.2: Aspecto de X-Plane 11. Extraído de steam

Además, X-Plane es conocido por su compatibilidad con el desarrollo de **plugins** y la interacción en tiempo real con variables de vuelo. Utilizando el SDK de X-Plane, los desarrolladores pueden acceder a datos como la posición, velocidad y actitud de las aeronaves, lo que permite una personalización significativa. Esta capacidad de crear plugins facilita a los usuarios y desarrolladores extender las funcionalidades del simulador, adaptándolo a necesidades específicas de investigación o educativas.

Según su propia web, X-Plane también ofrece versiones profesionales **certificadas por la FAA** (*Federal Aviation Administration*), lo que permite a los pilotos realizar horas de vuelo en el simulador bajo ciertas condiciones y con el hardware adecuado. Esta certificación refuerza la validez de X-Plane como herramienta de entrenamiento, no solo como un simulador recreativo sino también como una plataforma profesional confiable.

En comparación con otras soluciones como FlightGear, X-Plane ofrece una alta fidelidad en la simulación de vuelo y una extensa biblioteca de modelos de aeronaves y escenarios. Sin embargo, su naturaleza comercial puede presentar ciertas limitaciones para quienes buscan una solución completamente *open-source*. A pesar de ello, las herramientas proporcionadas por X-Plane para el desarrollo de complementos y extensiones lo hacen una opción valiosa para la educación y la investigación.

4.1.4. OpenAP

OpenAP [4] es un proyecto activamente mantenido por el Dr. Junzi Sun, un investigador de la Universidad Técnica de Delft (TU Delft). Este proyecto es una continuación de su trabajo de doctorado sobre modelado de rendimiento de aeronaves utilizando datos abiertos, realizado entre 2015 y 2019.

OpenAP se caracteriza por su uso de **datos abiertos**, lo que permite la transparencia y reproducibilidad de los resultados. Estos datos incluyen información sobre las características de las aeronaves, su rendimiento en diferentes fases del vuelo y sus emisiones. Esto es crucial para estudios de eficiencia y sostenibilidad en la aviación. La herramienta proporciona modelos detallados que permiten simular el rendimiento de diversas aeronaves bajo distintas condiciones operativas y ambientales, facilitando así el análisis y la optimización de las rutas de vuelo y la reducción del impacto ambiental de la aviación.

Una de las principales ventajas de OpenAP es su diseño modular. Este enfoque modular permite a los usuarios agregar y modificar componentes fácilmente, lo que es particularmente útil para investigadores que desean integrar nuevos algoritmos o modelos específicos a sus necesidades. La API de OpenAP facilita la interacción con el simulador, permitiendo su integración con otras herramientas y plataformas de simulación. Esto amplía las posibilidades de uso y combinación con otros sistemas de análisis y modelado, ofreciendo un flujo de trabajo más eficiente para proyectos de investigación.

El proyecto es de código abierto y su código está **disponible en Github**, lo que permite a los usuarios acceder y modificar el código base. Sin embargo, no incluye tutoriales detallados sobre cómo extender o editar el código. Investigando el repositorio es posible encontrar los diferentes modelos y, de ser necesario, editarlos. Sin embargo, esto implica que aunque el proyecto es accesible, los nuevos usuarios pueden enfrentar desafíos al intentar personalizarlo sin una guía explícita.

Mientras que FlightGear y X-Plane ofrecen simulaciones de vuelo generales y son utilizados para una variedad de propósitos, OpenAP está diseñado específicamente para la investigación en eficiencia y sostenibilidad aeronáutica. OpenAP se especializa en el modelado de rendimiento y emisiones, proporcionando herramientas precisas y datos abiertos para análisis detallados, lo que lo hace particularmente útil para estudios de eficiencia y sostenibilidad en la aviación.

4.1.5. BlueSky

BlueSky, un simulador de tráfico aéreo de código abierto. Este simulador se distribuye bajo la GNU General Public License v3, una licencia que fomenta la adaptabilidad y la modificación abierta del código, haciendo de BlueSky una herramienta ideal para la investigación en gestión del tráfico aéreo (ATM) y el análisis de flujos de tráfico aéreo. Su utilización se extiende a una variedad de aplicaciones, proporcionando una base sólida para explorar y

optimizar las operaciones de tráfico aéreo y las políticas de gestión.

La interfaz gráfica de usuario (GUI) (Figura 4.3) de BlueSky está diseñada para ser intuitiva y eficiente, facilitando a los usuarios la visualización y manipulación de escenarios complejos de tráfico aéreo. Esta interfaz permite ajustar variables y condiciones ambientales en tiempo real, ofreciendo una simulación dinámica que refleja desafíos reales.

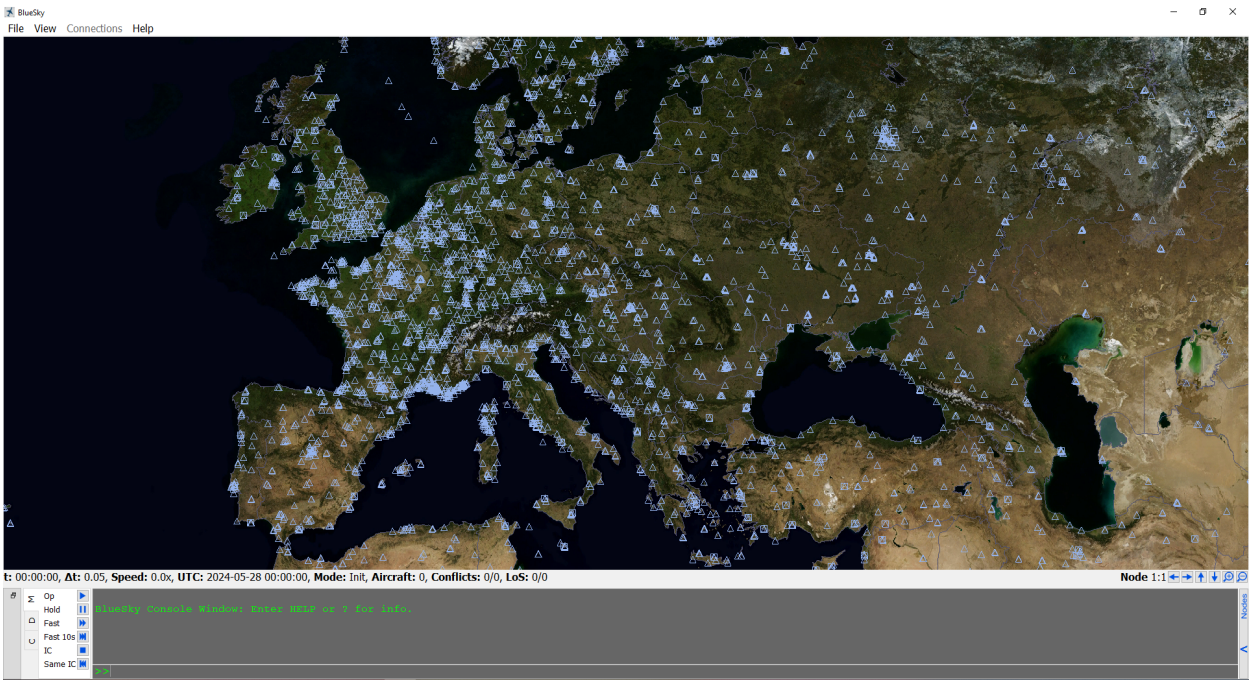


Figura 4.3: Interfaz Gráfica de Usuario (GUI) de BlueSky

En el núcleo de BlueSky se encuentran diversos modelos de simulación que abarcan desde las condiciones atmosféricas y los efectos del viento hasta las características operativas de los motores de las aeronaves. Estos modelos están diseñados para replicar de manera precisa y detallada el comportamiento de las aeronaves bajo una variedad de condiciones operativas y configuraciones. La habilidad de simular tales detalles permite a los investigadores y profesionales analizar y predecir el comportamiento de las aeronaves en diferentes escenarios, lo que es esencial para la seguridad y la eficiencia en la gestión del tráfico aéreo.

La implementación de BlueSky en **Python** es una decisión estratégica que aprovecha la flexibilidad y la potencia de este lenguaje de programación. Python, conocido por su extensa biblioteca de análisis de datos y herramientas científicas, facilita la integración de simulaciones complejas con algoritmos avanzados de procesamiento de datos y aprendizaje automático. Además, la estructura modular del código de BlueSky permite una fácil mantenibilidad y expansión, lo que ofrece a los desarrolladores la libertad de adaptar y extender el simulador según necesiten para sus investigaciones o aplicaciones específicas.

4.2. Otras soluciones

En el apartado anterior se han mostrado los simuladores aeronáuticos **más conocidos** o de escala considerable. Sin embargo, tienen una escala y objetivos que no son comparables con el alcance de este proyecto. El objetivo de este apartado es discutir algunos proyectos de una escala similar disponibles tanto en Github como en **RiuNet** (Repositorio Institucional de la Universitat Politècnica de València).

En primer lugar, haciendo una búsqueda por los repositorios públicos en Github, se observa que la inmensa mayoría de proyectos relacionados con simuladores aeronáuticos son **extensiones** de alguno de los grandes simuladores anteriormente mencionados, especialmente X-Plane. Esto denota de la gran popularidad de los mismos. Sin embargo, tienen una finalidad distinta a la de este proyecto, que no busca ser la extensión de ningún otro. Por lo general, se trata de expansiones para incluir información de nuevos aviones o aeropuertos pero no para introducir nuevos modelos.

Por otro lado, en RiuNet existen algunos proyectos que si comparten ciertas características con este. A continuación se mencionan dos de ellos:

- **Simulación y Guiado de Aeronaves utilizando Programación Orientada a Objetos** (Peris 2016[5]). Este proyecto implementa un simulador aeronáutico con una interfaz gráfica de la cabina de un avión hecha en java. Su objetivo es la representación de los datos aeronáuticos de la simulación mediante los instrumentos de cabina. Ambos proyectos comparten una interfaz gráfica hecha en Java y el uso de la programación orientada a objetos. Sin embargo, Peris 2016, utiliza una interfaz para comunicarse con X-Plane, el cual se utiliza como núcleo del simulador para obtener los datos.

Figura 4.4: Interfaz Gráfica que replica una cabina

- **Desarrollo de un simulador de vuelo SIL en el entorno *Simulink* para el autopiloto Veronte** (Robles 2023[6]). Este simulador reproduce el comportamiento de diversos tipos de aeronaves empleando una serie de modelos matemáticos configurables. Utiliza Matlab Simulink, y utiliza ecuaciones de mecánica de vuelo con seis

grados de libertad (6DOF) en contraposición a nuestro proyecto que **no profundiza** en la mecánica de vuelo. Este simulador en *Simulink* carece de interfaz gráfica y esta centrado únicamente en la mecánica de vuelo.

4.3. Discusión

En la tabla 4.1 se muestra una comparativa de los simuladores anteriormente mencionados.

Simulador	**Tecnología principal**	**Interfaz Gráfica**	**Extensibilidad**	**Ventajas**	**Desventajas**
Microsoft Flight Simulator	C++	Ultra Realista	Baja	Líder del sector	Software comercial y baja extensibilidad
FlightGear	C++, OpenGL	Realista	Media	Realismo, Detalle	Complejidad y es principalmente un videojuego
X-Plane	C / C++	Realista	Alta	Amplia Comunidad de Usuarios y extensibilidad	Software comercial
OpenAP	Python	Ninguna	Alta	Simplicidad	Proyecto casi unipersonal
BlueSky	Python	Pygame	Alta	Extensibilidad	Capacidades gráficas limitadas
Peris 2016	Java	Realista (cabina)	Alta	Extensibilidad	Depende de X-Plane
Robles 2023	Matlab Simulink	Ninguna	Media	Mecánica de vuelo precisa	Utiliza softwares comerciales y propietarios

Cuadro 4.1: Comparación de Simuladores Aeronáuticos

En primer lugar, se menciona Microsoft Flight Simulator, ampliamente reconocido como uno de los simuladores más avanzados y realistas en el mercado. Este producto de Microsoft es frecuentemente el primer referente que viene a la mente al pensar en simuladores aeronáuticos. Por otro lado, X-Plane y FlightGear presentan características más alineadas con las metas de nuestro proyecto. Aunque X-Plane sigue siendo un producto comercial, ofrece amplias posibilidades de personalización y cuenta con varios módulos de código abierto, facilitando así una mayor intervención por parte de los usuarios. FlightGear, siendo completamente de código abierto, permite una personalización aún más profunda, lo cual es ventajoso para propósitos de desarrollo específico. No obstante, ambos simuladores cuentan con componentes gráficos avanzados que exceden las necesidades de nuestro proyecto.

Es crucial diferenciar claramente entre estos simuladores de primera categoría desarrollados por una multinacional o una gran comunidad de usuarios y los objetivos que busca este proyecto de **escala más reducida**. Es por ello que se llevó a cabo también una consulta de proyectos más pequeños. En cuanto a los proyectos de escala similar consultados en RuiNet ambos tratan un tema muy concreto, representación de la información en instrumentos de cabina y desarollo de un autopiloto, de una manera excesivamente detallada para este proyecto. El cual pretence abarcar más temas pero con un grado menor de profundidad **dejando al usuario profundizar** en lo que considere oportuno.

Por último, se menciona a OpenAP y BlueSky. Por un lado, OpenAP es esencialmente un conjunto de modelos aeronáuticos aislados. Sin embargo, cabe una mención especial a BlueSky, que representa el enfoque más cercano a nuestro proyecto. BlueSky ha sido **especialmente influyente**, inspirando varios aspectos clave, como el uso de una clase de simulador central que administra todos los aspectos relacionados con la simulación, incluyendo el manejo del tiempo y la implementación de una serie de comandos que el simulador puede procesar, temas que se explorarán en profundidad a lo largo del proyecto.

Capítulo 5

Diseño e implementación

Este apartado trata sobre el diseño del simulador y como se ha implementando. En primer lugar, en la figura 5.1, se muestra la arquitectura de alto nivel del sistema. Esta imagen muestra los componentes principales que lo conforman y como interactúan entre ellos.

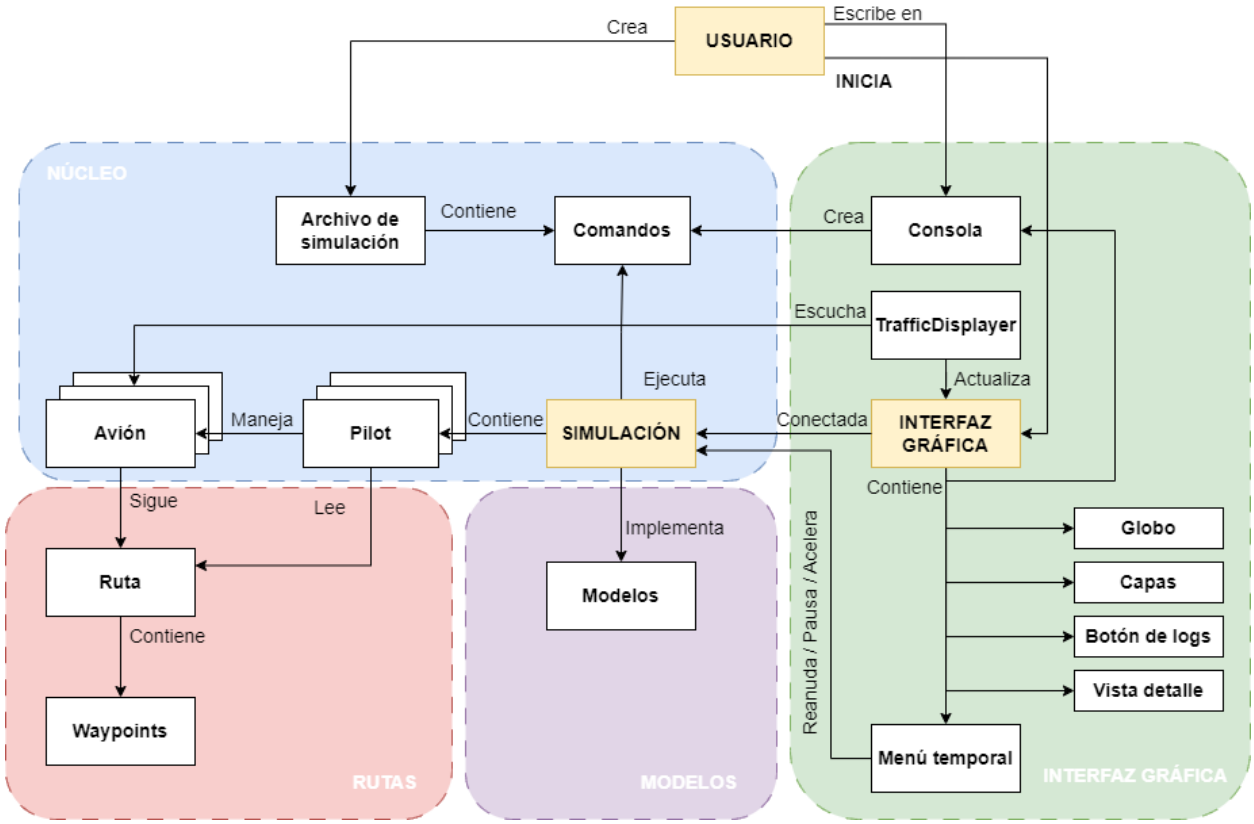


Figura 5.1: Arquitectura de alto nivel del sistema

Existen cuatro partes claramente diferenciadas y el usuario. En primer lugar, tenemos al usuario que se encarga de iniciar el programa e interactuar con la simulación o bien en vivo mediante la interfaz gráfica o de antemano creando un archivo de simulación. Por otro lado,

tenemos el núcleo de la simulación, que contiene las clases esenciales para su funcionamiento. Este núcleo implementa una serie de modelos que replican diferentes realidades (atmósfera, giros, etc). Por último, los aviones del núcleo siguen una serie de rutas que vienen definidas por una serie de waypoints. A lo largo de los siguientes apartados se irá explicando cada uno de estos componentes

En la figura 5.2, se muestran la mayoría de clases (aunque no todas) del proyecto. Estas clases representan muchos de los componentes mostrados en la arquitectura de alto nivel del sistema y están agrupadas en paquetes para mejorar su organización y facilitar su legibilidad. Estas clases son de la elaboración propia y interactúan entre ellas además de con clases y/o librerías de terceros.

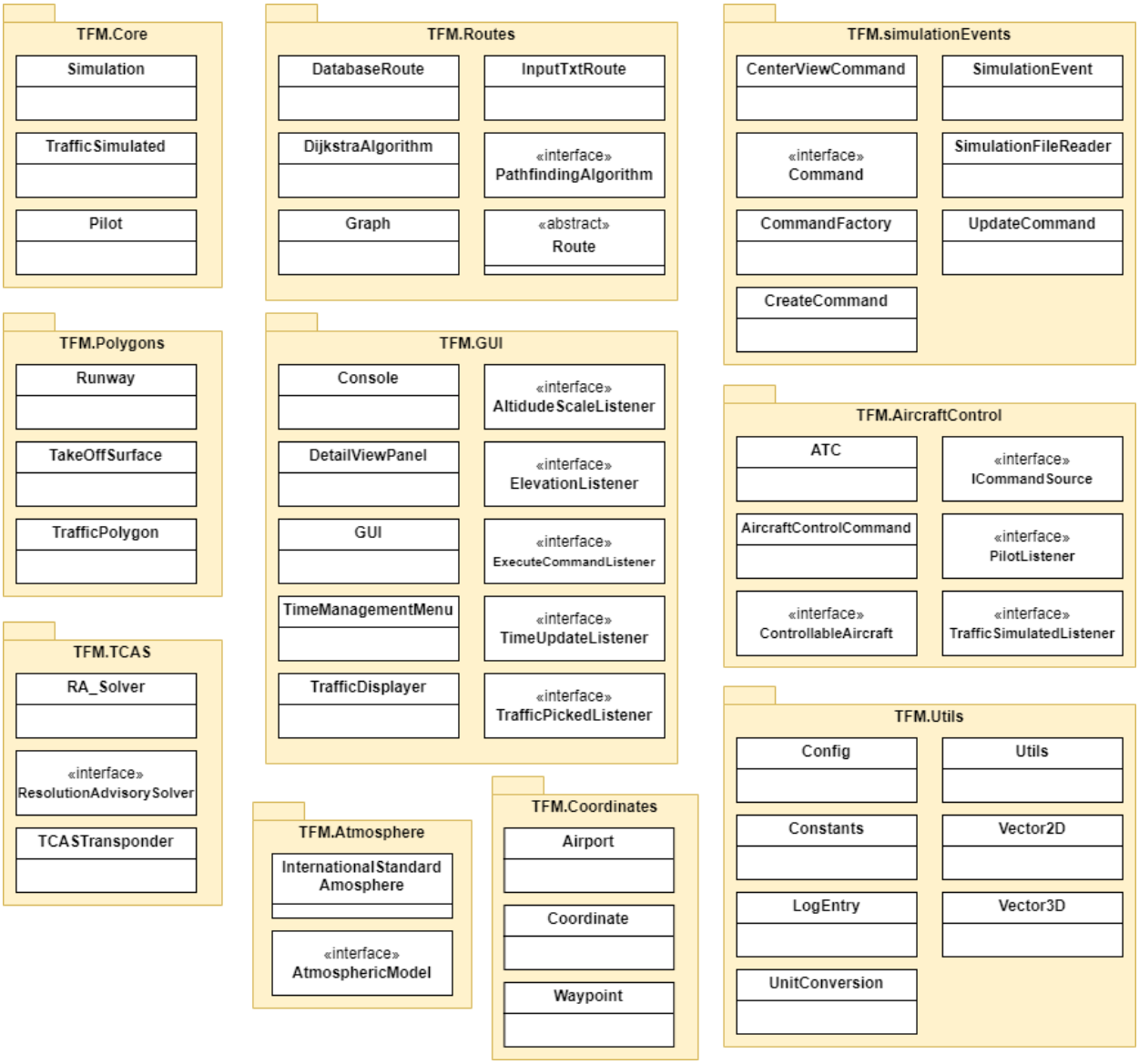


Figura 5.2: Clases del proyecto organizadas en paquetes

5.1. Núcleo del Simulador

En este apartado se profundiza en las clases que componen el núcleo del simulador, es decir las que forman parte del paquete `TFM.Core` (véase figura 5.2). El objetivo de este apartado no es explicar todos los atributos y métodos de cada clase, los cuales están disponibles en el repositorio público (sección 3.1.4) para quien lo considere oportuno si no describir su funcionamiento a grandes rasgos. Para ello se mencionan los atributos y métodos principales junto a un diagrama de flujo (si procede).

En primer lugar se explica la clase `Thread`, nativa del lenguaje de programación java, ya que es la clase de la cual heredan (siguiendo el concepto de herencia mencionado en el capítulo 3) tanto `Simulaton`, como `TrafficSimulated` y como `Pilot`. Adicionalmente, se profundiza también en la clase `SimulationEvent` para dar contexto necesario para la clase `Simulation`.

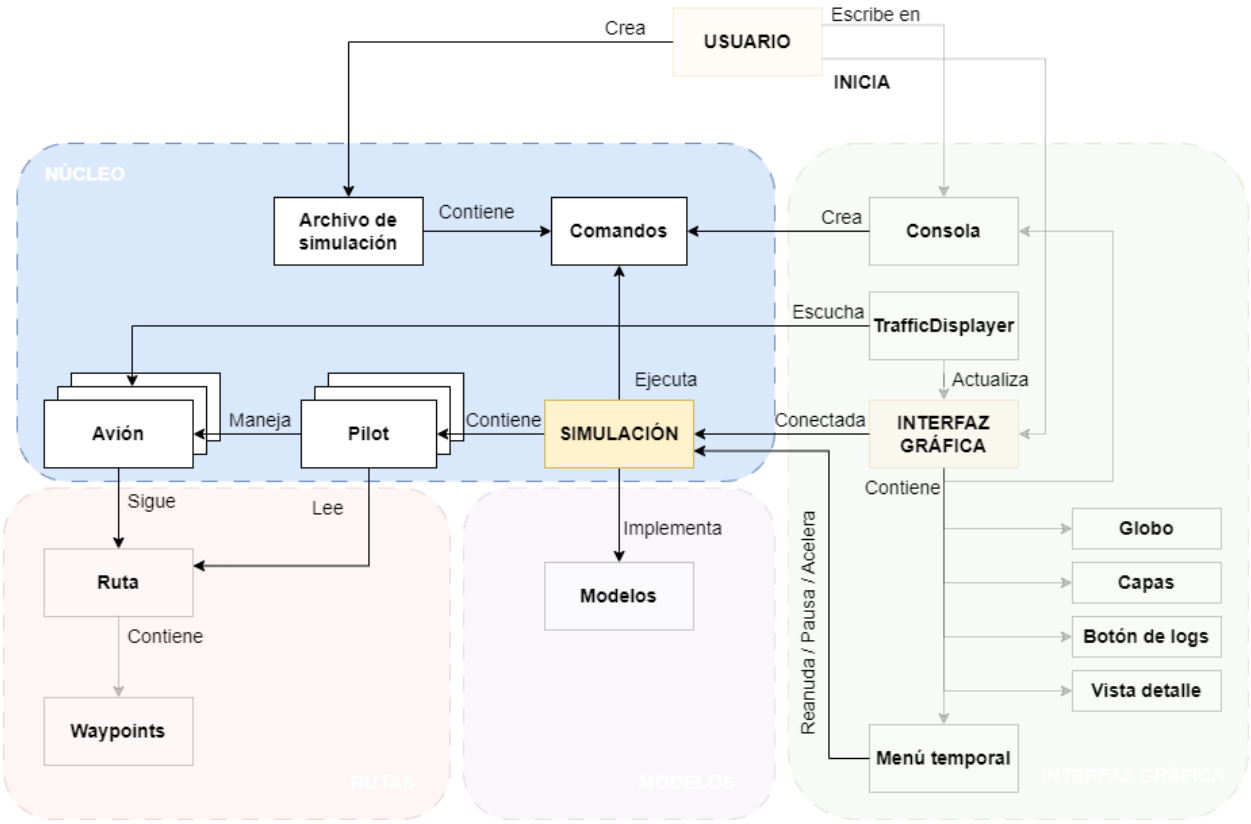


Figura 5.3: Arquitectura de alto nivel del núcleo del simulador

5.1.1. Clase Thread

En Java, la clase Thread representa un **hilo de ejecución independiente** en un programa. Un hilo (o *thread* en inglés) es la unidad más pequeña de procesamiento que puede ser gestionada de manera independiente por el sistema operativo. Cada hilo tiene su propia

secuencia de ejecución de código, lo que permite que múltiples operaciones se lleven a cabo de manera **concurrente** dentro de un mismo programa.

En la programación secuencial, el código se ejecuta secuencialmente, es decir, una línea de código se ejecuta después de la otra. Sin embargo, con los hilos, es posible ejecutar múltiples secuencias de código al mismo tiempo, lo que se conoce como paralelismo o concurrencia. El paralelismo no es más que ejecutar código secuancial en diferentes hilos simultánamente. O dicho de otra forma, la programación secuencial o *single threaded* no es más que hacer uso de un único hilo. En java, la *Java Virtual Machine (JVM)* ,que es el entorno donde se ejecuta java, crea un hilo principal o *main thread* el cual podemos utilizar para hacer programación secuencial o podemos crear más hilos dentro de el para hacer uso del paralelismo (véase figura 5.4).

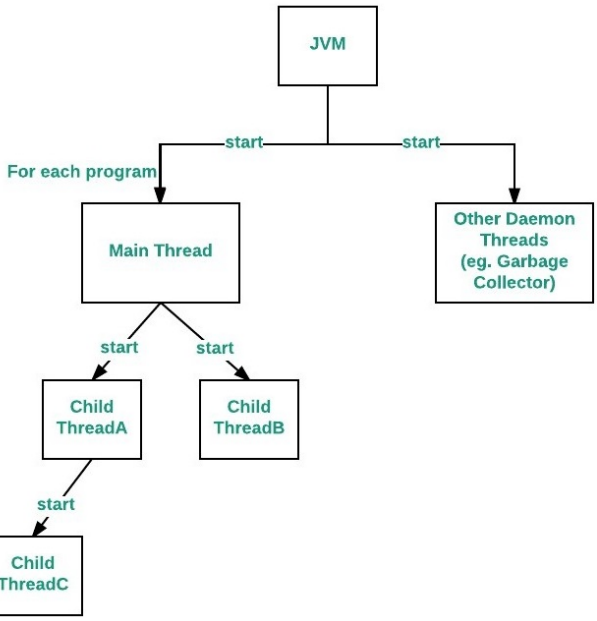


Figura 5.4: Diagrama de flujo de la JVM. Extraído de https://www.geeksforgeeks.org/main-thread-java/.

Los hilos son esenciales para crear aplicaciones que requieran realizar múltiples tareas de manera simultánea. Por ejemplo, en un simulador aeronáutico, es crucial que diferentes partes de la simulación, como los aviones, puedan operar simultánea e independientemente sin interferir entre sí. Las principales ventajas de los hilos son:

- **Mejor Utilización de los Recursos del CPU**: Permiten a una aplicación hacer un uso más eficiente de los recursos del CPU al ejecutar múltiples operaciones en paralelo.
- **Responsividad**: Mejoran la capacidad de respuesta de una aplicación, permitiendo que tareas intensivas en tiempo se ejecuten en segundo plano sin bloquear la interfaz de usuario. Esto aspecto es clave para una interfaz de usuario fluída.

Sim embargo, los hilos, implican una serie de desventajas o complicaciones añadidas:

- **Complejidad**: La programación concurrente puede ser compleja y difícil de depurar (si es que acaso es posible), especialmente cuando se trata de sincronización y gestión de recursos compartidos.
- **Problemas de Sincronización**: La necesidad de coordinar el acceso a recursos compartidos entre hilos puede llevar a errores difíciles de manejar, como *race condition* (que causa alrededor del **80 %** de los problemas de sincronización [7]) y *deadlocks*.

Los hilos en Java pueden ser creados extendiendo la clase `Thread` o implementando la interfaz `Runnable`. Cada hilo se ejecuta de manera independiente, y Java proporciona varios mecanismos para gestionar la sincronización y la comunicación entre hilos. Los métodos más importantes de la clase `Thread` son:

- `start()`: Es el punto de entrada para comenzar la ejecución concurrente de un hilo. Llama internamente al método `run()` en un nuevo hilo de ejecución.
- `run()`: Este método contiene el código que se ejecutará en el nuevo hilo. Por defecto, el método `run()` de la clase `Thread` no hace nada, por lo que debe ser sobreescrito cuando se crea una subclase de `Thread` o se proporciona una implementación de la interfaz `Runnable`. Aquí es donde se aplica la lógica que hemos definido para los aviones, pilotos y simulación respectivamente, en bucle hasta que se indique lo contrario.
- `sleep()`: Pone en suspensión al hilo actual durante el número especificado de milisegundos. Esto hace que el hilo deje de ejecutarse durante el tiempo indicado y pase al estado *Timed Waiting*. En el simulador se utiliza este método para regular la tasa de refresco de los datos. Si el tiempo de espera fuera 0 la tasa de refresco sería mayor pero el consumo de recursos sería máximo.
- `join()`: Espera a que el hilo en el que se llama termine su ejecución. Si se llama a `join()` en un hilo, el hilo que lo llama se suspenderá hasta que el hilo objetivo complete su ejecución.
- `interrupt()`: Señala que el hilo debe ser interrumpido. Si el hilo está en estado de espera, sueño o espera prolongada (`wait()`, `sleep()`, `join()`), se lanza una `InterruptedException`.
- `isAlive()`: Devuelve `true` si el hilo está en ejecución y `false` si ha terminado su ejecución.

5.1.2. Clase Simulation

La clase `Simulation` es el **núcleo del simulador**, es responsable de controlar el **tiempo de simulación** y gestionar la ejecución de **eventos y comandos** asociados. Esta clase hereda de `Thread` para permitir la ejecución concurrente, lo que facilita la actualización continua del estado de la simulación sin bloquear otras operaciones además de poder ejecutar varias simulaciones simultáneamente. Cuando se ejecuta el simulador, por defecto se crea

una simulación.

En la tabla 5.1 se muestran los atributos principales y en la tabla 5.2 se muestran los métodos principales de la clase `Simulation`.

Nombre	**Tipo**	**Descripción**
`simulationTime`	`long`	Almacena el tiempo transcurrido de simulación en milisegundos desde el inicio. Por ejemplo si tiene un valor 3500, la simulación se encuentra en el segundo 3,5.
`sleepTime`	`int`	Tiempo que espera (`sleep()`) el hilo entre ejecuciones. Podemos controlar entre otras cosas la tasa de refresco de la interfaz gráfica.
`speed`	`double`	Factor de velocidad de la simulación. Por defecto es en uno, es decir un segundo de la simulación equivale a un segundo real, pero puede interesar por ejemplo acelerar la velocidad de la simulación. Es decir, cambiar el paso temporal.
`trafficDisplayer`	`TrafficDisplayer`	En cada iteración notifica a la interfaz gráfica de cambios en la simulación.
`trafficSimulationMap`	`TrafficSimulationMap`	Almacena todos los `TrafficSimualted` (aviones) de la simulación.
`pilotMap`	`ConcurrentHashMap`	Almacena todos los pilotos de la simulación.
`events`	`List`	Listado de los eventos que irá ejecutando la simulación.

Cuadro 5.1: Atributos principales de la clase `Simulation`.

Adicionalmente, los diferentes modelos utilizados, también son atributos de la clase `Simulation` (e.g., modelo atmosférico). De esta manera, quedan definidos en la misma y el resto de clases del proyecto obtienen los modelos fácilmente. Esta centralización de la declaración de los tipos de modelos permite una rápida sustitución.

Nombre	Descripción
`setModels()`	Define los modelos que serán utilizados en la simulación (véase apartado 5.3).
`play()`	Arranca la simulación.
`pause()`	Pausa la simulación.
`stopit()`	Finaliza la simulación.
`setSpeed()`	Cambia la velocidad de la simulación.
`run()`	Método inicial del `Thread`. Consiste en un bucle while que mantiene la simulación funcionando. En cada iteración actualiza las variables temporales de la simulación, llama al método `updateSimulation()` y por último, recorre toda la lista de eventos y ejecuta los comandos necesarios con `executeCommand()`.
`updateSimulation()`	Actualiza todos los aviones en la interfaz gráfica y su información en la vista de detalle
`executeCommand()`	Ejecuta un comando.

Cuadro 5.2: Métodos principales de la clase `Simulation`.

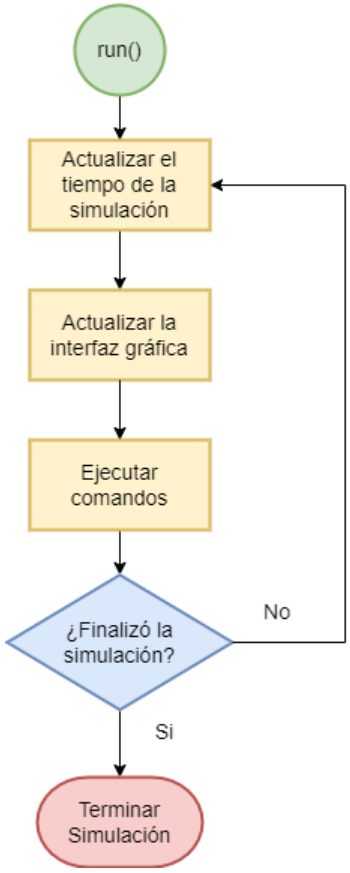


Figura 5.5: Diagrama de flujo dél método `run()` de la clase `Simulation`.

En la tabla 5.2 se puede observar, como ya hemos venido comentando, que la función

principal de esta clase es la gestión temporal de la simulación (inicio, pausa y fin) y la actualización de ciertas partes de la misma en cada iteración. Esto última se explica en el diagram de flujo la figura 5.5. En cada iteración, la clase `Simulation` actualiza el tiempo de la misma, lo cual es imprescindible para el cálculo de variables (e.g., velocidad, altura, etc.) en otros lugares de la simulación. En segundo lugar, se recorren todos los aviones de la simulación y se actualiza su posición en la interfaz gráfica. Por último, se ejecutan todos los comandos (véase sección 5.1.3) cuya marca temporal sea igual o inferior al actual tiempo de simulación.

5.1.3. Clase SimulationEvent

El paquete `SimulationEvents` constituye una parte fundamental del proyecto, proporcionando una interfaz amigable para que los usuarios **interactúen con el simulador** sin necesidad de modificar directamente el código fuente. Aunque la personalización del simulador a menudo requiere ajustes en el código, este paquete ofrece una alternativa más accesible al permitir el envío de órdenes al simulador de manera simple y directa para las tareas más cotidianas. La clase central de este paquete es `SimulationEvent`, cuya instancia llamaremos evento. Un evento se compone de los siguientes elementos:

- **Marca temporal**. En milisegundos, define en qué momento de la simulación ha de ejecutarse el comando. Por ejemplo una marca temporal de 10000 indica que el comando ha de ejecutarse en el segundo 10.
- **Comando**. Es la acción a ejecutar por el evento. Por ejemplo, crear un nuevo avión. Los comandos disponibles se muestran en la tabla 5.3.
- **Variables**. Algunos comandos necesitan variables adicionales para ejecutar la acción. Siguiendo el ejemplo anterior, para crear el avión es necesario introducir algunos parámetros como el identificador o la ruta.

En la figura 5.6 se muestra un ejemplo del comando utilzado para crear un avión en el segundo dos que siga la ruta LEST (Santiago de Compostela) - LEPA (Palma de Mallorca).

Figura 5.6: Ejemplo de un `CREATE SimulationEvent`

Se han dejado las bases definidas para crear y ejecutar comandos de manera sencilla. Sin embargo, no se han definido un gran número de ellos. Los comandos disponibles en la actualidad, se muestran en la siguiente tabla. No recogen ni mucho menos todas las

posibilidades del simulador y se invita a los usuarios a ir añadiendo comandos según sus necesidades.

Comando	Descripción
`CREATE`	Crea un avión con una serie diferente de parámetros.
`UPDATE`	Actualiza parametros de un avión existente.
`CENTERVIEW`	Centra la vista en un avión en concreto simulando una vista de seguimiento.
`START`	Pausa la simulación.
`RESUME`	Reanuda la simulación.
`SIMSPEED`	Cambia el factor de multiplicación de la velocidad de la simulación.

Cuadro 5.3: Comandos disponibles en el simulador.

Existen dos maneras de enviar comandos al simulador:

1. **Archivos de simulación**. No es más que un archivo de texto con comandos separados por líneas. El simulador lee todos los eventos al iniciarse y ejecutará los comandos a medida que se alcance la marca temporal.

2. **Línea de comandos**. Se pueden escribir comandos en tiempo real a mediante la consola al uso (figura 5.7) incluída en la interfaz gráfica. La marca temporal en este caso es la del mismo instante en el que se introduce el comando.

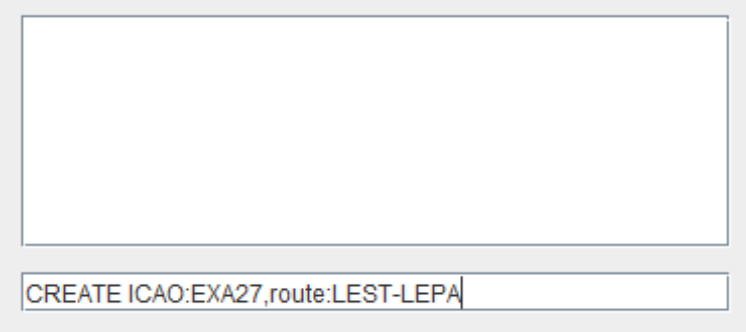


Figura 5.7: Ejemplo de introducción de un comando vía consola.

5.1.4. Clase Pilot

La función principal de la clase `Pilot` es **dirigir al avión**. Esta clase, al igual que `Simulation` también hereda de la clase `Thread`. Un avión tiene un único piloto y un piloto solo puede manejar un avión. Todos los pilotos se almacenan en el `pilotMap` mencionado en la tabla 5.1.

En la tabla 5.4 se indican los atributos principales de la clase pilot, en alguno de los cuales se profundiza en apartados propios.

Nombre	Tipo	Descripción
`route`	`Route`	Ruta seguida por el avión.
`plane`	`TrafficSimulated`	Avión que dirige el piloto.
`routeMode`	`int`	Modo en el que el piloto realiza la ruta. Hay dos opciones de ruta: loxodrómica y ortodrómica.
`bearingStrategy`	`BearingStrategy`	Lógica que aplica el piloto ante un cambio de rumbo. Por defecto se implementa la *Rate One Turn* (ROT) estándar que impone un giro de 360^{o} en 2 minutos o 3^{o} por segundo [8].
`myTCASTransponder`	`TCASTransponder`	Clase que simula el comportamiento de un transpondedor *Traffic Colision Avoidance System*.
`vp`	`VerticalProfile`	Clase que contiene el perfil vertical a seguir por el piloto durante la ruta.

Cuadro 5.4: Atributos principales de la clase `Pilot`.

Nombre	Descripción
`pauseThread()`	Pausa el piloto y el avión cuando se pausa la simulación.
`resumeThread()`	Reanuda el piloto y el avión cuando se reanuda la simulación.
`setSpeed()`	Cambia la velocidad de la simulación.
`run()`	Método que se ejecuta cuando se inicia el piloto. Inicia un bucle `for` que llama al método `fly()` para cada uno de los waypoints de la ruta.
`fly()`	Consiste en un bucle `while` que actualiza las variables necesarias para alcanzar siguiente waypoint de acuerdo con la ruta y el perfil vertical.

Cuadro 5.5: Métodos principales de la clase `Pilot`.

Como ya se ha comentado, el objetivo principal de la clase `Pilot` es controlar el avión. En la figura 5.8 se muestra un diagrama de flujo del método `run()` que permite entender el comportamiento básico de esta clase. El proceso a grandes rasgos es el siguiente:

1. La clase recorre todos los *waypoints* que conforman la ruta y dirige al avión al siguiente. Si es la primera iteración, se dirige evidentemente al primer *waypoint*.

2. Se ajusta el rumbo del avión para llevar al *waypoint* fijado.

3. El transpondedor TCAS hace una interrogación. Cabe destacar que el transpondedor TCAS, para ajustarse más a la realidad debería ser en si mismo un `Thread` y marcar sus propios tiempos de interrogación (1s para aviones a menos de 5 millas náuticas). Pero por simplicidad se introdujo dentro de este bucle.

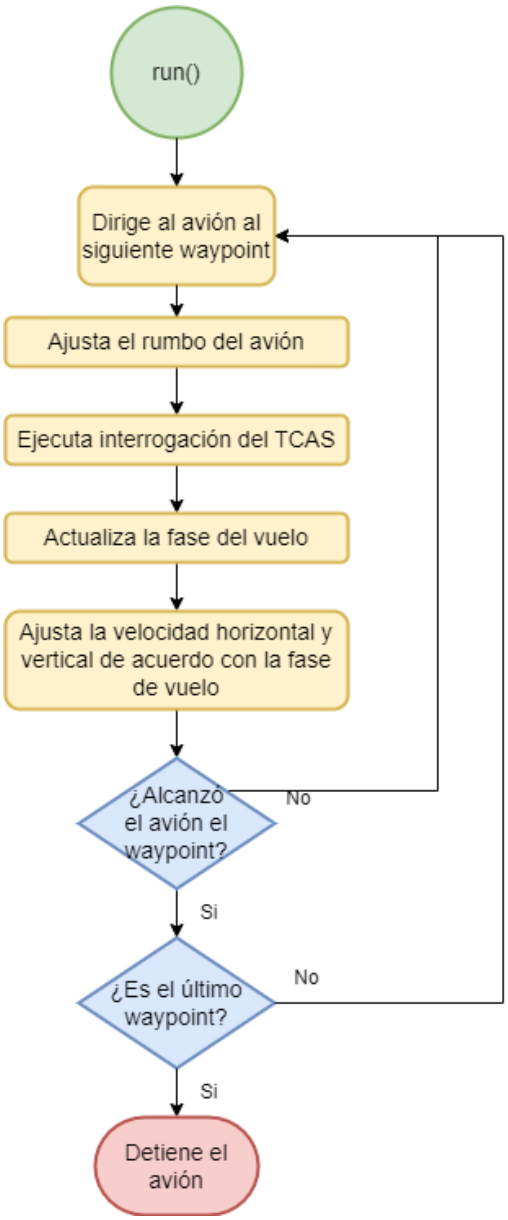


Figura 5.8: Diagrama de flujo dél método `run()` de la clase `Pilot`.

4. Se obtiene la fase de vuelo en la que se encuentra el avión (ascenso, crucero, etc.).

5. Se adapta la velocidad horizontal y vertical de acuerdo con la estipulada en la fase de vuelo.

5.1.5. Clase TrafficSimulated

La clase `TrafficSimulated` nace de la necesidad de **agrupar las características asociadas a un avión** en una única clase. Al igual que `Simulation` y `Pilot` también hereda de la clase `Thread` lo que le concede cierta independencia a la hora de actualizar sus variables. Sus atributos (tabla 5.6) definen el estado de un avión. Estos quedan definidos principalmente por un identificador (`HexCode`), posición, velocidad y rumbo. Mediante la lectura de estas variables se puede representar gráficamente el avión.

Nombre	**Tipo**	**Descripción**
`hexCode`	`String`	Identificador del avión. Sirve para ser identificado tanto para el usuario como para las diferentes estructuras de datos.
`position`	`Coordinate`	Define la posición del avión en todo momento en latitud, longitud y altura.
`speed`	`double`	TAS (*True Air Speed*) del avión [kts].
`verticalRate`	`double`	Velocidad vertical [ft/min]
`course`	`double`	Rumbo de la aeronave [º]
`logEntries`	`List` $< LogEntry >$	Lista con *logs* que almacena las variables del avión a lo largo del tiempo para poder hacer un postprocesado de los resultados.

Cuadro 5.6: Atributos principales de la clase `TrafficSimulated`.

En el diagrama de flujo de la figura 5.9 se resume el comportamiento de la clase que se comenta a continuación:

1. El piloto inicia el hilo. Como ya se ha comentado el avión es dirigido por la clase `Pilot`.

2. El avión entra en un bucle hasta que se alcance el destino marcado por el piloto. Este destino se define mediante un *waypoint* con latitud y longitud.

3. En cada iteración del bucle:

   - Se obtiene el tiempo actual de simulación y se calcula la diferencia con el último tiempo de simulación. Obteniéndose así el paso temporal.
   - Con el paso temporal y la velocidad del avión se calcula la distancia.
   - Con dicha distancia y el rumbo marcado por el piloto se actualiza la posición.
   - Con el paso temporal y velocidad vertical se actualiza la altura.
   - Almacenar las variables en ese instante en el atributo `logEntries` para más adelante poder obtener los *logs* del avión.

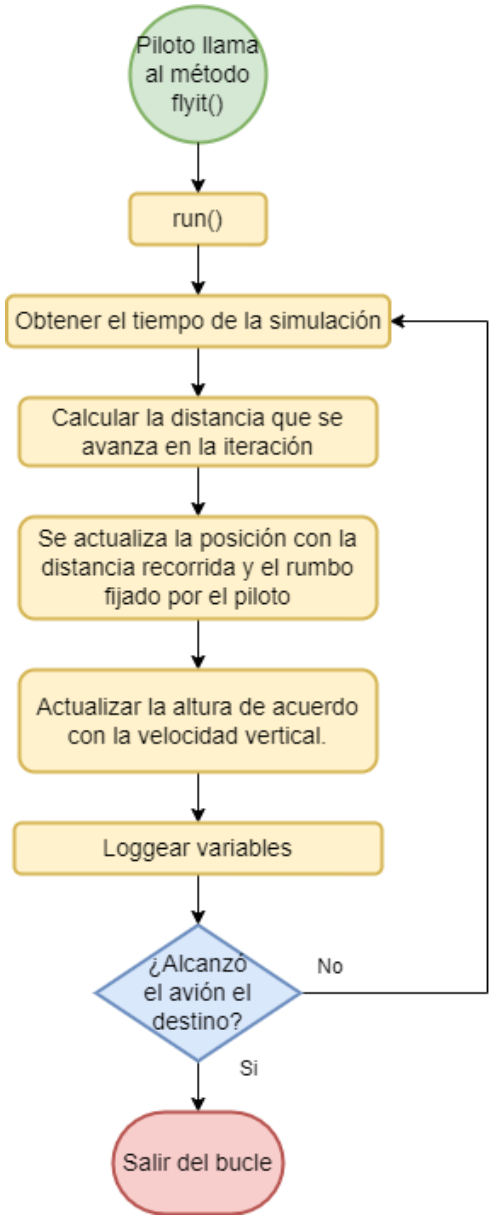


Figura 5.9: Diagrama de flujo de la clase `TrafficSimulated`.

5.2. Rutas

El paquete `TFM.Routes` engloba las clases relacionadas con rutas (véase figura 5.2). En el contexto de este simulador, una ruta se define como una serie de **puntos de navegación** entre el aeropuerto de origen y de destino que el avión sigue bajo el mando del piloto. Este paquete cubre la creación de las mismas, que tipos existen y como interactúan entre si.

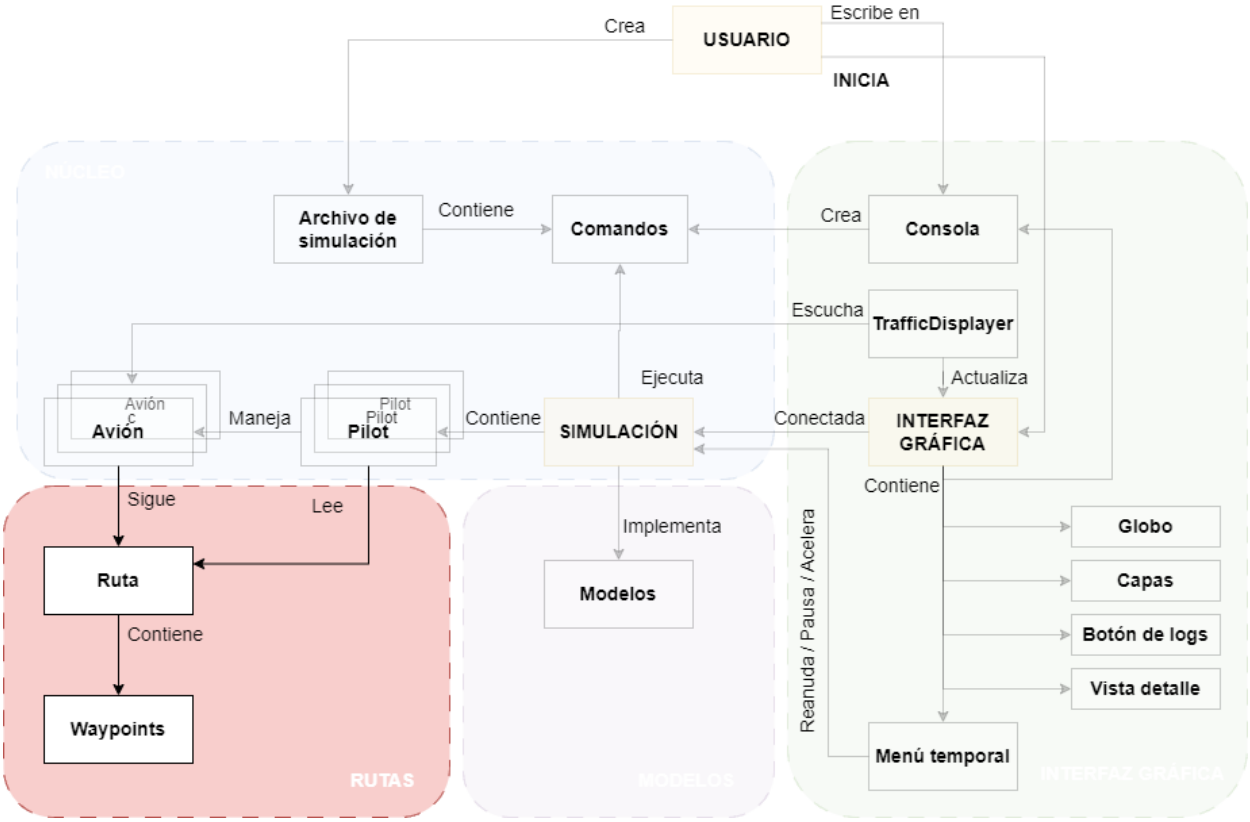


Figura 5.10: Arquitectura de alto nivel de las rutas.

Para este paquete, se ha realizado el diagrama de clases de la figura 5.11 para facilitar su explicación. Se puede observar que hay una clase abstracta princiapl llamada `Route` la cual extienden los diferentes tipos de rutas.

5.2.1. Clase Route

La clase `Route` es una **clase abstracta**. En primer lugar, una clase abstracta es una clase que no puede ser instanciada directamente y puede contener métodos abstractos, que son métodos declarados sin implementación. Estos métodos deben ser implementados por las subclases concretas. Además, una clase abstracta puede tener métodos con implementación completa. Las clases abstractas, se utilizan para proporcionar una **base común** con ciertas funcionalidades y exigir que las subclases implementen comportamientos específicos

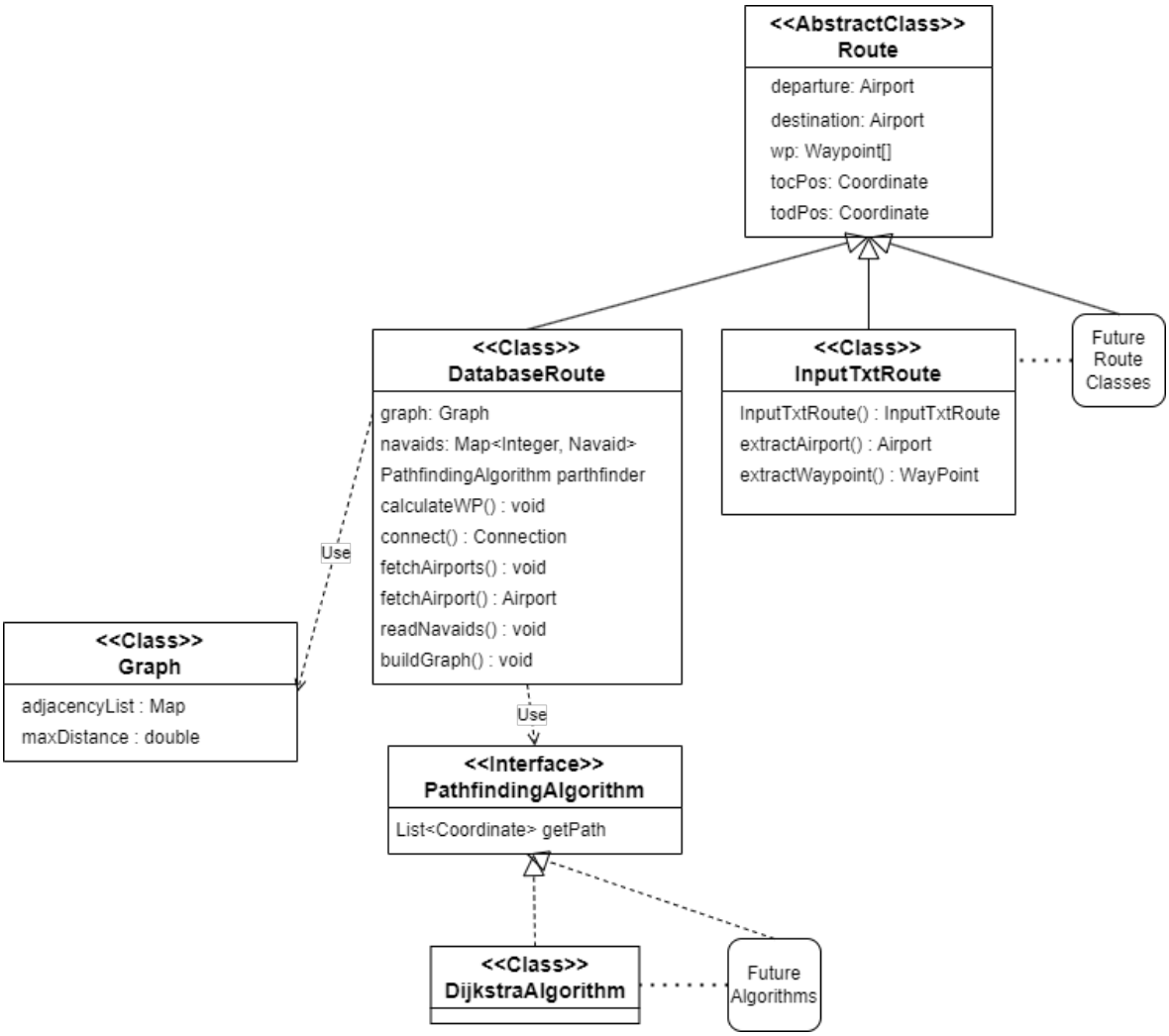


Figura 5.11: Diagrama de clases del paquete `TFM.Route`.

(de manera similar a las interfaces).

Es decir, se busca establecer unos criterios (atributos y métodos) mínimos que debe cumplir una clase para ser considerado una ruta. En este caso se refiere a:

- Un aeropuerto de salida.
- Un aeropierto de destino.
- Un listando que contiene los puntos de navegación que conforman la ruta.
- El TOC (*Top Of Climb*) de la ruta. Punto en la ruta de un avión donde se alcanza la altitud de crucero después del ascenso.
- El TOD (*Top Of Descent*) dela ruta. Punto en la ruta de un avión donde se inicia el descenso para aproximación y aterrizaje.

El como se determinan estos atributos se delega en las clases que la extiendan, como es el caso de `InputTxtRoute` y `DatabaseRoute`.

5.2.2. InputTxtRoute

Esta clase hereda de la clase base `Route`. Su función principal es generar una ruta a partir de un archivo de texto que enumera los *waypoints* que componen dicha ruta. El formato del archivo de texto sigue el estándar proporcionado por la herramienta en línea **RouteFinder**. Esto permite al usuario crear rutas entre cualquier par de aeropuertos en el mundo de manera sencilla. A continuación, se muestra un ejemplo de una ruta que conecta Valencia con Sevilla:

```
ID      FREQ   TRK   DIST   Coords                          Name/Remarks
LEVC            0      0   N39°29'22.00" W000°28'54.00" VALENCIA
ASTRO         232     46   N39°01'27.99" W001°15'47.00" ASTRO
POBOS         232     30   N38°43'08.99" W001°46'08.00" POBOS
XEBAR         232     19   N38°31'15.99" W002°05'33.99" XEBAR
YES     115.2 232     16   N38°21'38.99" W002°21'10.00" YESTE
BLN     116.2 259     61   N38°09'09.00" W003°37'29.00" BAILEN
LEZL          249    117   N37°25'05.00" W005°53'56.00" SEVILLA
```

Figura 5.12: Archivo de texto con la ruta de Valencia a Sevilla

En esta ruta, el primer *waypoint* se designa como el aeropuerto de origen (`departure`), y el último *waypoint* se establece como el aeropuerto de destino (`destination`). Los *waypoints* intermedios forman la lista `wp`, que representa la trayectoria que debe seguir el avión.

5.2.3. DatabaseRoute

Esta clase también hereda de la clase base `Route`. A diferencia de la clase `InputTxtRoute` que simplemente realiza una lectura de una ruta ya generada, este clase **genera su propia ruta**.

En primer lugar, obtiene las coordenadas (latitud y longitud) de los diferentes puntos que potencialmente pueden conformar la ruta de una base de datos en local para formar un grafo. Dicha base de datos ha sido previamente populada con la información de la base de datos de acceso libre proporcionado por *OurAirports*[9].

En segundo lugar, se aplica un algoritmo para obtener la ruta entre el aeropuerto de origen y de destino pasando por las coordenadas anteriormente extraídas. Como se muestra en la figura 5.11, este algoritmo puede ser cualquier clase que extienda la interfaz `PathfindingAlgorithm`. Dicha interfaz, solo contiene un método que se llama `getPath()` que calcula la ruta a partir de las coordenads de origen, de destino y un grafo de los puntos intermedios disponibles.

El simulador, por defecto, utiliza el `DijkstraAlgorithm` para buscar la ruta más corta posible sobre el grafo. Este algoritmo se explica en profundidad en el apartado 5.3.1. Extendiendo la interfaz `PathfindingAlgorithm` se podrían implementar otros algoritmos para buscar la ruta más corta posible o incluso otros objetivos (e.g. la ruta que cruce menos paises).

5.3. Modelos Implementados

En el contexto de este simulador, un "modelo"se refiere a una representación simplificada y abstracta de un sistema complejo que se utiliza para simular y analizar su comportamiento. Estos modelos son esenciales para replicar aspectos de la realidad aeronáutica dentro del entorno de simulación. Los modelos pueden abarcar desde la simulación de condiciones atmosféricas hasta la generación y optimización de rutas de vuelo. Cada modelo se construye con un conjunto específico de parámetros y reglas que definen cómo se comporta en la simulación. A continuación, se presentan algunos de los modelos.

5.3.1. Algoritmo de Dijkstra

En el simulador, se utiliza el **algoritmo de *Dijkstra*** para encontrar la ruta más corta entre un punto de origen y destino. Para ello, es necesario primero crear un **grafo**.

Un **grafo** es una estructura matemática utilizada para modelar relaciones entre objetos. Un grafo G se define como un par ordenado (V, E) donde:

- V es un conjunto de **vértices** o **nodos**.
- E es un conjunto de **aristas** o **bordes** que conectan pares de vértices.

Cada arista $e \in E$ puede representarse como un par (u, v), donde u y v son vértices en V. Las aristas pueden tener **pesos** asociados que representan el costo o la distancia entre los vértices. A continuación, se muestra una imagen de un grafo simple:

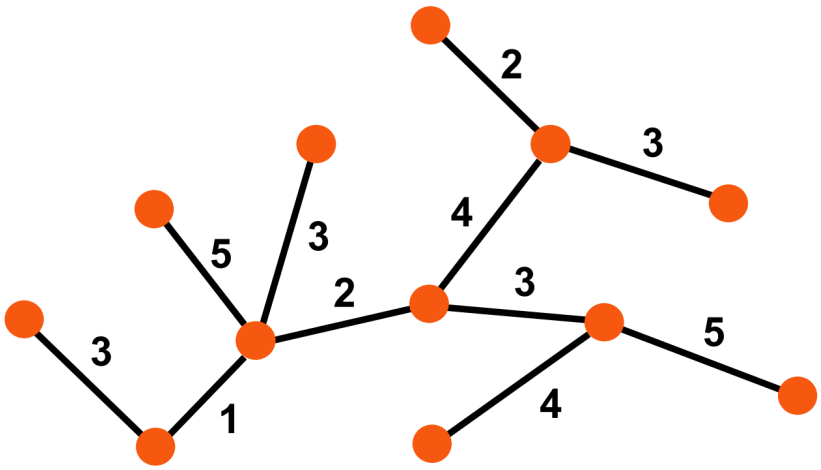


Figura 5.13: Ejemplo de un grafo simple.

En el simulador, la clase `Graph`, refleja el comportamiento de un grafo donde los vértices son coordenadas de puntos de navegación y las aristas definen las conexiones entre los mismos. Las aristas están limitadas a una distancia máxima que es un parámetro de la clase.

El **algoritmo de Dijkstra** es un método eficiente para encontrar la ruta más corta desde un nodo fuente a todos los demás nodos en un grafo ponderado dirigido. Este algoritmo es particularmente útil en aplicaciones como la navegación y la planificación de rutas. En el simulador viene recogido en la clase `DijkstraAlgorithm` que implementa la interfaz `PathfindingAlgorithm`.

El algoritmo de Dijkstra funciona manteniendo un conjunto de distancias mínimas desde el nodo fuente a cada uno de los otros nodos. El algoritmo sigue los siguientes pasos:

1. Inicializa la distancia a la fuente S como 0 y a todos los demás nodos como ∞.

2. Marca todos los nodos como no visitados. Establece la fuente como el nodo actual.

3. Para el nodo actual, considera todos sus vecinos no visitados y calcula sus distancias tentativas. Compara la distancia recién calculada con la distancia actual almacenada y guarda la menor.

4. Marca el nodo actual como visitado. Un nodo visitado no será revisado nuevamente.

5. Si el nodo destino ha sido marcado como visitado o si la menor distancia tentativa entre los nodos no visitados es ∞, el algoritmo termina.

6. De lo contrario, selecciona el nodo no visitado con la menor distancia tentativa y lo establece como el nuevo nodo actual, luego regresa al paso 3.

El pseudocódigo del algoritmo de Dijkstra es el siguiente:

Algorithm 1 Algoritmo de Dijkstra

Require: Grafo $G = (V, E)$, nodo fuente S
Ensure: Distancia mínima de S a cada nodo $v \in V$

```
Inicializar dist[v] ← ∞ para todos los v ∈ V
dist[S] ← 0
Inicializar conjunto de nodos no visitados Q ← V
while Q ≠ ∅ do
    u ← nodo en Q con dist[u] mínima
    Remover u de Q
    for cada vecino v de u do
        if v ∈ Q then
            alt ← dist[u] + peso(u, v)
            if alt < dist[v] then
                dist[v] ← alt
            end if
        end if
    end for
end while
```

La complejidad temporal del algoritmo de Dijkstra depende de la implementación de la estructura de datos utilizada para el conjunto de nodos no visitados Q. Utilizando una cola de prioridad (*heap*), la complejidad es $O((V + E) \log V)$, donde V es el número de vértices y E es el número de aristas.

5.3.2. Modelo atmosférico

El simulador incluye un modelo atmosférico de la **Atmósfera Estándar Internacional** (ISA por sus siglas en inglés). Es un modelo de la atmósfera terrestre que permite obtener valores de presión, temperatura, densidad y viscosidad del aire en función de la altitud. Su función es proporcionar un **marco de referencia invariante** para la navegación aérea y para la realización de cálculos aerodinámicos consistentes. En este proyecto se ha seguido el estándar de la ISO 2533:1955[10].

El modelo de la ISA divide la atmósfera en capas con distribuciones lineales de temperatura y se basa en condiciones promedio en latitudes medias. Lo que quiere decir que estos modelos atmosféricos sirven como referencia para cálculos previos al vuelo. Para datos reales, se necesita información sobre las condiciones meteorológicas reales que pueden diferir de este promedio.

En el simulador, este modelo se aplica en la clase `InternationalStandardAtmosphere` que implementa la interfaz `AtmosphericModel`. A partir de la altura geométrica del avión (z) se calcula la altura geopotencial (h) del mismo, mediante la ecuación 5.1. Conocida la altura geopotencial, la atmósfera estándar nos permite calcular las variables ya mencionadas utilizando las ecuaciones de la tabla 5.7.

$$z = \frac{R_e \cdot h}{R_e - h} \tag{5.1}$$

gradiente $a = 0$	**gradiente** $a \neq 0$
$T(h) = T_0$	$T(h) = T_0 + a \cdot (h - h_0)$
$p(h) = p_0 \cdot e^{\left(-\frac{g}{RT} \cdot (h - h_0)\right)}$	$p(h) = p_0 \left(\frac{T(h)}{T_0}\right)^{-\frac{g}{aR}}$
$\rho(h) = \rho_0 \cdot e^{\left(-\frac{g}{RT} \cdot (h - h_0)\right)}$	$\rho(h) = \rho_0 \left(\frac{T(h)}{T_0}\right)^{-1-\frac{g}{aR}}$

Cuadro 5.7: Ecuaciones para calcular T, p y ρ en función de h. Donde h_0, p_0, ρ_0 y T_0 son los valores en la base de la franje considerada y a el gradiente térmico, ambos valores conocidos.

Finalmente estas variables son utilizadas para hacer conversiones entre velocidades:

- IAS (Indicated Air Speed). La velocidad que muestra el indicador de velocidad del aire de una aeronave, sin corrección por errores de calibración ni densidad atmosférica.
- TAS (True Air Speed). La velocidad real de una aeronave en relación con el aire que la rodea, corregida por las condiciones atmosféricas y la altitud.

5.3.3. TCAS

El *Traffic Collision Avoidance System* (TCAS) es un sistema de a bordo utilizado en aeronaves para **reducir el riesgo de colisiones** en el aire entre aeronaves equipadas con transpondedores. Funciona mediante la emisión de interrogaciones de radar y la recepción de respuestas de transpondedores de otras aeronaves en el área. El TCAS analiza estas respuestas para determinar la distancia, la altitud y la velocidad de cierre de las aeronaves cercanas. Con esta información, el sistema puede generar alertas visuales y auditivas para los pilotos sobre el tráfico potencialmente conflictivo y, en situaciones críticas, proporcionar recomendaciones de maniobras evasivas, como ascender o descender, para evitar colisiones.

Existen varias versiones de TCAS, con el TCAS II siendo el más avanzado y comúnmente utilizado en la aviación comercial. El TCAS II no solo detecta y advierte sobre conflictos de tráfico, sino que también coordina las maniobras evasivas entre las aeronaves implicadas, asegurando que ambas realicen movimientos complementarios (por ejemplo, una sube mientras la otra baja). Este sistema es crucial para la seguridad aérea, especialmente en espacios aéreos congestionados, ya que actúa como una **última línea de defensa** contra colisiones en vuelo, complementando los servicios de control de tráfico aéreo.

Se define el *Closest Point of Approach* (CPA) como el punto en el cual la distancia entre la trayectoria de ambos aviones es mínima. El tiempo que transcurre entre la posición actual del avión y el CPA se conoce como τ. El TCAS distingue cuatro tipos de tráfico a su alrededor en función del CPA y del τ como se muestran en la figura 5.14:

- ***Other Traffic***: Tráfico detectado por el TCAS que no representa una amenaza inmediata, generalmente se muestra fuera de la zona de proximidad y no requiere ninguna acción.
- ***Proximate Traffic***: Tráfico cercano que está dentro de una distancia de 6 millas náuticas y 1200 pies verticalmente, pero no presenta una amenaza de colisión inminente.
- ***Traffic Advisory*** (TA): Alerta que indica tráfico potencialmente conflictivo, advirtiendo a los pilotos para que estén atentos y preparen una posible maniobra evasiva.
- ***Resolution Advisory*** (RA): Instrucción de maniobra emitida por el TCAS para evitar una colisión, sugiriendo acciones específicas como ascender o descender para mantener una separación segura. Este simulador, por el momento, solo tiene en cuenta las RA, para el resto de tipos de tráfico, no se toma ningún tipo de acción.

La clase `TCASTransponder` es la encargada de simular el comportamiento de un transpondedor TCAS en el presente simulador. Como se menciona en la figura 5.8, en cada iteración del piloto se ejecuta una iteración (o interrogación) del transpondedor. En cada iteración se siguen los pasos mostrados en la figura 5.15. Para cada avión dentro del rango del TCAS, se calculan el CPA y el τ. A continuación, si el tipo de tráfico se considera *Resolution Advisory* (RA) se llama a la clase `RASolver` que implementa la interfaz `ResolutionAdvisorySolver`.

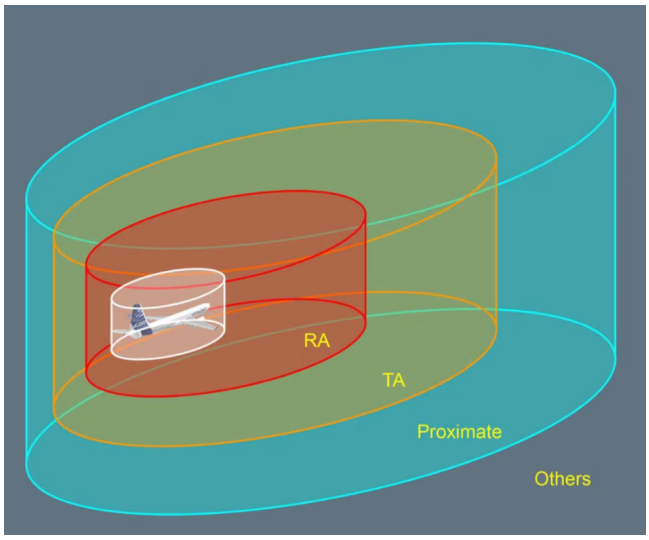


Figura 5.14: Envolvente TCAS. Extraído de https://simpleflying.com/tcas-working-principles-guide/

.

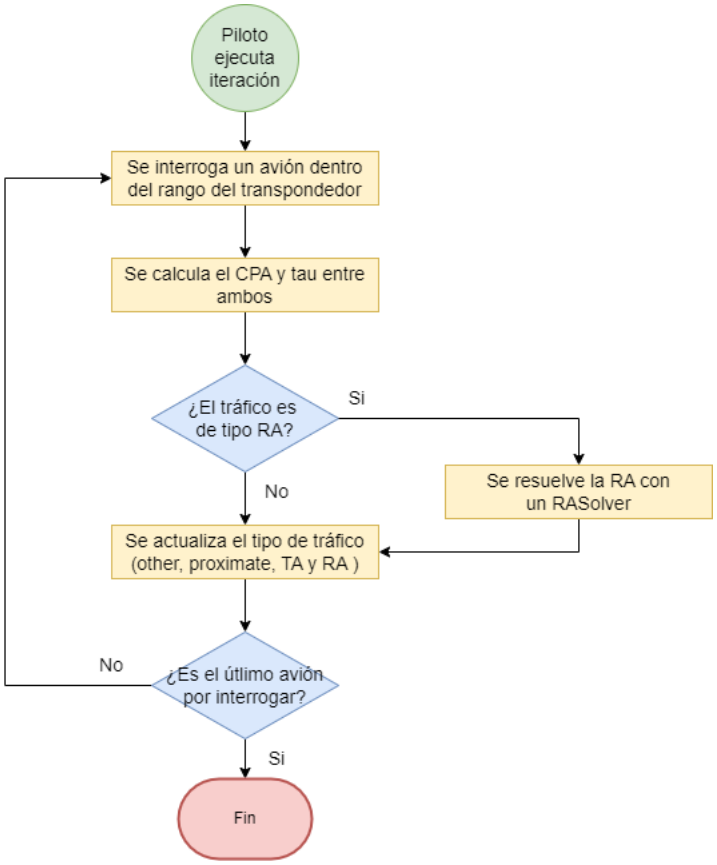


Figura 5.15: Diagrama de flujo de la clase `TCASTransponder`

La clase `RASolver` es una clase que implementa una lógica para **maniobras coordinas**

de evasión. Sin entrar en mucho detalle, esta clase implementa un algoritmo de determinación de la dirección que una aeronave debería tomar en el caso de un potencial conflicto con otra aeronave de acuerdo al *paper* presentado por Anthony Narkawicz [11]. Concretamente actuando sobre la velocidad vertical (obviando las componentes horizontales) de ambas aeronaves, es decir, sugiriendo que una ha de subir y otra ha de bajar, de manera coordinada para así reducir su riesgo de colisión (véase figura 5.16) .

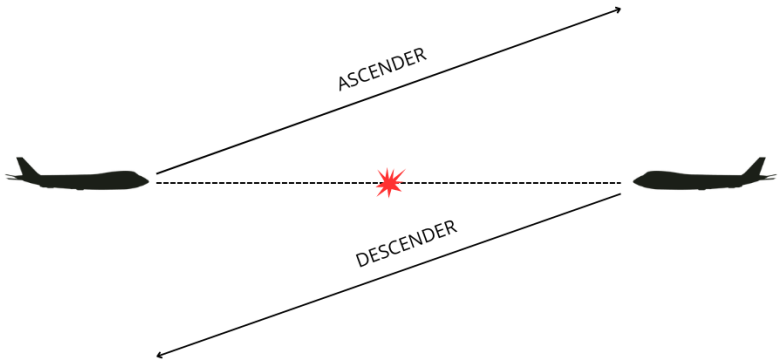


Figura 5.16: Maniobra coordinada de ascenso y descenso sugerida por el TCAS

5.3.4. Sustitución de un modelo

En el desarrollo de este proyecto, uno de los objetivos fundamentales ha sido crear un simulador que sea, en la medida de lo posible, extensible y que fomente la colaboración abierta. Esto significa que el simulador debe permitir la fácil incorporación de nuevas funcionalidades o modificaciones a las existentes, sin necesidad de realizar cambios significativos en su estructura básica. Para lograr esto, el uso de **interfaces** juega un papel crucial como ya hemos explicado.

El objetivo de este apartado es ejemplificar como haría el usuario para realizar un cambio de modelo ya existente. La capacidad de cambiar de un modelo a otro en el simulador es sencilla gracias al diseño basado en interfaces. Dentro de la clase `Simulator`, se encuentra el método `setModels()` que se encarga de inicializar los diferentes modelos utilizados en la simulación (figura 5.17).

```
private void setModels() {
    this.atm = new InternationalStandardAtmosphere(); // Atmospheric model
    this.pathfindingAlgorithm = new DijkstraAlgorithm(); // Pathfinding algorithm
    this.bearingStrategy = new SmoothBearingStrategy(); // Bearing strategy
    this.RASolver = new RA_Solver(); // Resolution advisory solver
}
```

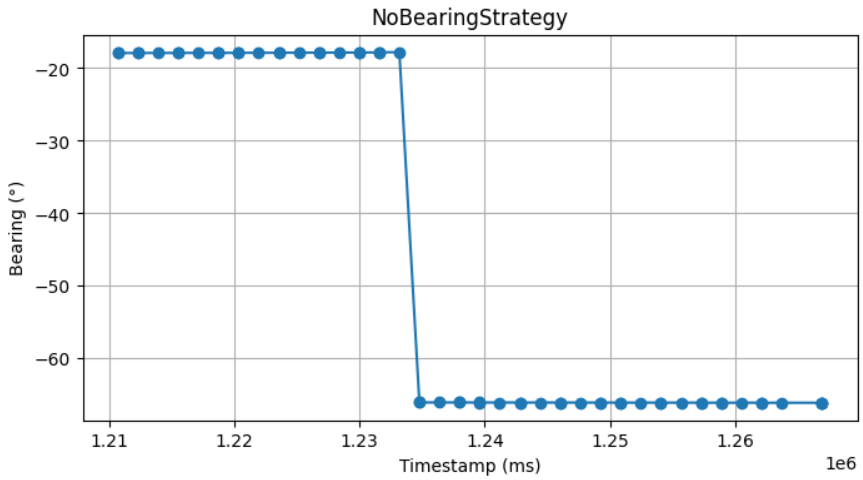
Figura 5.17: Método `setModels()` de la clase `Simulation`

Un ejemplo simple de esta extensibilidad se encuentra en el modelo de cambio de rumbo (giro) del avión. Cuando el avión alcanza un *waypoint* de la ruta, el piloto dirige al avión hacía el siguiente. Inicialmente, esto suponia un cambio instantáneo de rumbo (`NoBearingStrategy` véase figura 5.18(a)), lo que daba lugar a comportamientos muy poco realistas de aviones girando, por ejemplo, $100^{\underline{o}}$ al instante. Para mejorar este comportamiento, se definió la interfaz `BearingStrategy`, que especifica un método para calcular el nuevo rumbo del avión en función del rumbo actual, el rumbo objetivo y el tiempo transcurrido. De esta manera podemos utilizar varios modelos que implementen la misma interfaz que calcularán el giro de distinta manera.

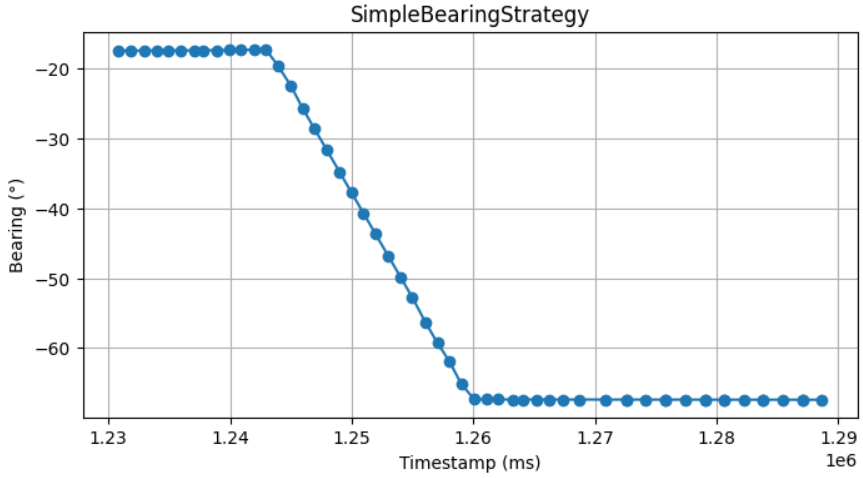
Posteriormente, se creó una estrategia simple llamada `SimpleBearingStrategy` que extendía la interfaz `BearingStrategy`. Esta estrategia asume una tasa de giro constante de 3 grados por segundo, lo que permite completar un giro de 360 grados en 2 minutos (véase figura 5.18(b)). Más adelante, se ha desarrollado una estrategia un poco **más sofisticada** denominada `SmoothBearingStrategy`. Esta estrategia simula un giro más gradual y realista, donde el avión inicialmente acelera su giro hasta alcanzar una tasa máxima y luego desacelera al acercarse al rumbo deseado (véase figura 5.18(c)).El resultado es una **transición de rumbo más fluida** y precisa. Por último, en el futuro se podrían añadir otros implementaciones más complejas que tengan en cuenta, por ejemplo, el alabeo del avión.

Para cambiar la estrategia de cambio de rumbo del avión, simplemente se asigna una nueva implementación de `BearingStrategy` en este método. Por ejemplo, se puede reemplazar `SimpleBearingStrategy` por `SmoothBearingStrategy` sin modificar otras partes del código del simulador.

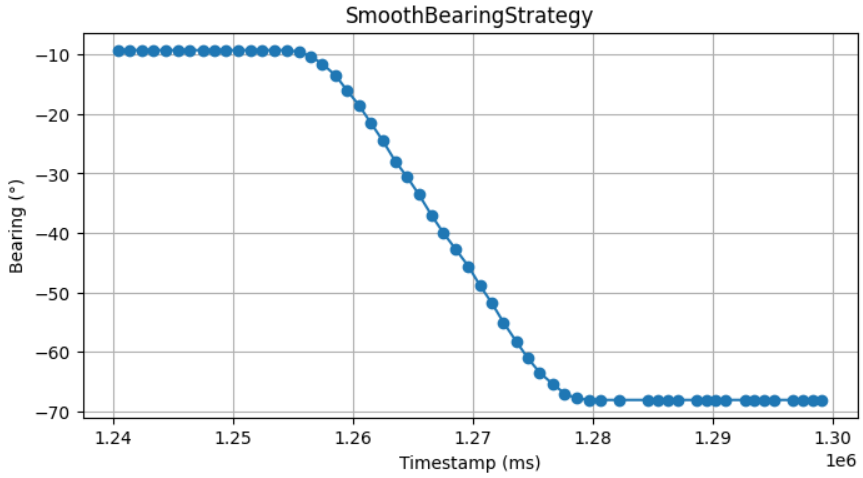
Esta arquitectura no solo facilita el cambio de modelos, sino que también fomenta la colaboración abierta. Otros desarrolladores pueden **crear sus propias implementaciones** de `BearingStrategy` y añadirlas al simulador, mejorando y ampliando sus capacidades sin necesidad de alterar el núcleo del sistema. Para el caso en el que un desarrollador necesite crear un modelo nuevo en lugar de una implementación de uno ya existente. Se invita a hacerlo de manera extensible mediante interfaces para posteriormente poder añadirlo al ya mencionado método `setModels()`.



(a) Giro utilizando `NoBearingStrategy`



(b) Giro utilizando `SimpleBearingStrategy`



(c) Giro utilizando `SmoothBearingStrategy`

Figura 5.18: Giro implementando de diferentes maneras la interfaz `BearingStrategy`

Capítulo 6

Interfaz Gráfica

En este capítulo, se describe la implementación de la **interfaz gráfica de usuario** (GUI, por sus siglas en inglés) del simulador. La GUI proporciona una representación visual interactiva que permite a los usuarios observar y controlar la simulación en tiempo real. Para ello, se ha utilizado NASA WorldWind, una plataforma de visualización de datos geoespaciales en 3D. Este capítulo cubrirá una visión general de NASA WorldWind, cómo se ha integrado con el simulador y una revisión detallada de la interfaz gráfica del usuario con capturas de pantalla explicativas.

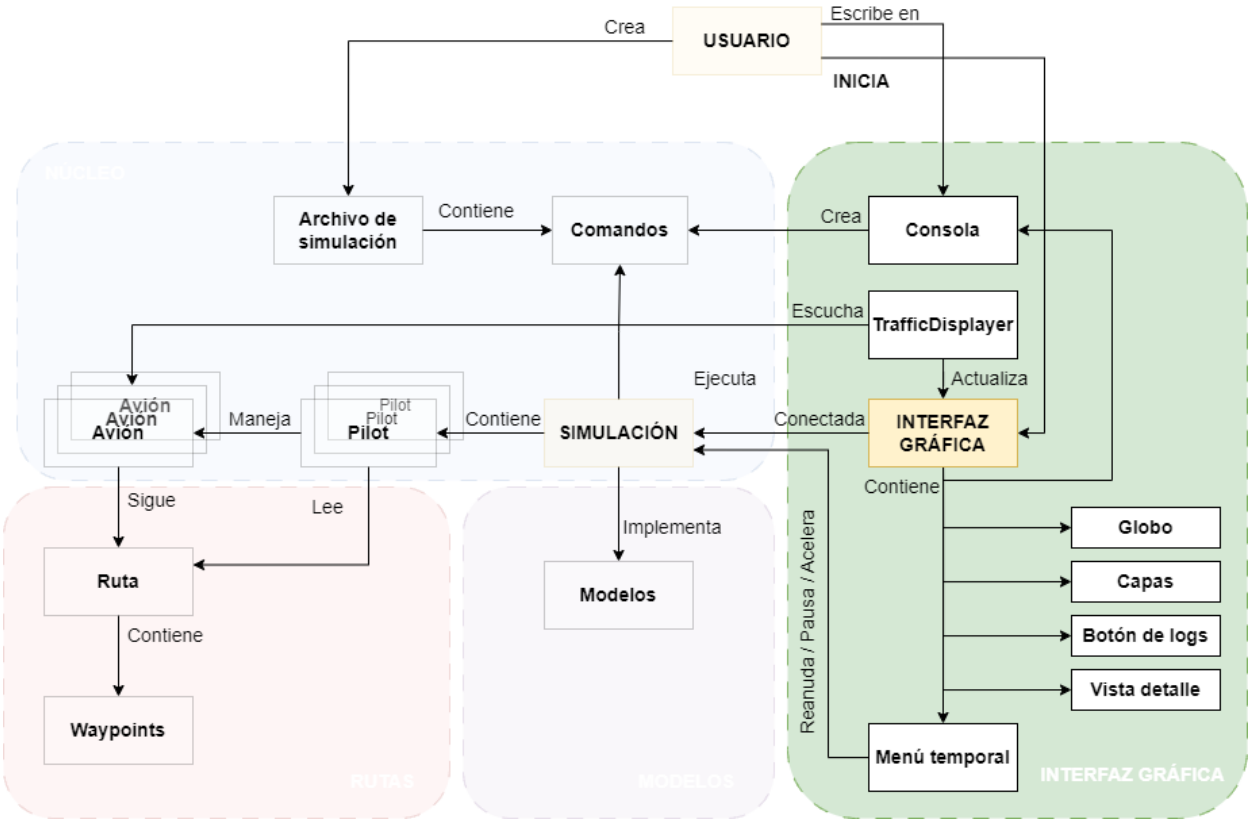


Figura 6.1: Arquitectura de alto nivel de la interfaz gráfica

6.1. NASA WorldWind

WorldWind es un **motor de gráfico** en 3D. WorldWind permite a los desarrolladores de aplicaciones crear rápidamente visualizaciones interactivas de información geográfica en un contexto planetario en 3D. Organizaciones de todo el mundo utilizan WorldWind para monitorear patrones climáticos, visualizar ciudades y terrenos, rastrear movimientos de vehículos, analizar datos geoespaciales y educar a la humanidad sobre la Tierra [12].

Es un proyecto de **código abierto** bajo la licencia NOSA. Lanzado inicialmente en 2003, su desarrollo ha sido impulsado por la comunidad desde 2004. La versión original fue desarrollada en .NET Framework, lo que limitaba su uso a sistemas Windows. Posteriormente, se ha expandido a otras plataformas, incluyendo una versión en Java (utilizada en este simulador), una versión para Android y, más recientemente, una versión web.

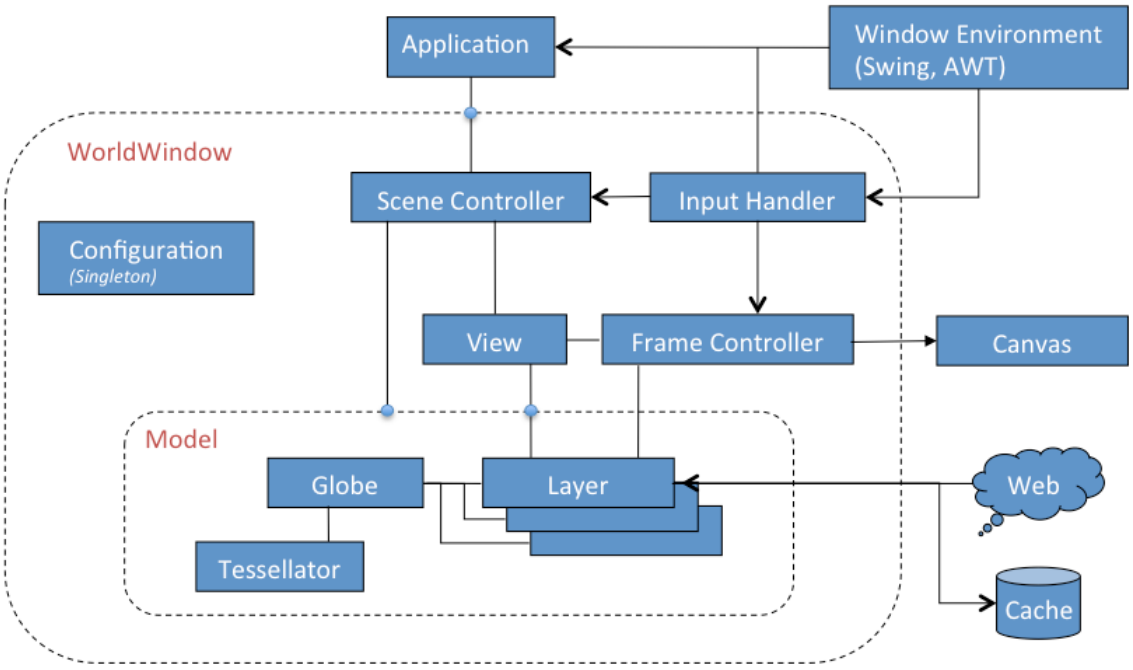


Figura 6.2: Diagrama de interfaces de java WorldWind. Extraído de https://worldwind.arc.nasa.gov/java/tutorials/concepts/.

En la figura 6.2 se muestra el diagrama de interfaces que componen la versión java de worldwind. No se pretende explicar su complejo funcionamiento pero si que es necesario mencionar dos componentes del mismo: el globo (`Globe`) y las capas (`Layer`).

6.1.1. Globe

El componente `Globe` en NASA WorldWind Java es esencial para la representación y manipulación de la forma y el terreno de un planeta (no está limitado a la Tierra). Esta

clase es fundamental para cualquier operación que involucre coordenadas geográficas y su transformación a coordenadas cartesianas.

La clase `Globe` representa la forma y el terreno de un planeta dentro del contexto de WorldWind. Actúa como el **modelo central** que define cómo se interpreta y maneja la geografía del planeta. Proporciona métodos para convertir posiciones geográficas (latitud, longitud y elevación) en coordenadas cartesianas y viceversa, lo cual es crucial para representar correctamente la posición de los objetos en la visualización 3D.

Además, `Globe` puede estar asociado con un `ElevationModel`, que proporciona datos de elevación para posiciones geográficas específicas en la superficie del globo. Este modelo de elevación permite la representación precisa del terreno en 3D, reflejando colinas, montañas y valles. Podría ser utilizado en el contexto de este simulador, por ejemplo, para desarollar modelos de evasión de colisiones con el terreno.

6.1.2. Capas (Layers)

En WorldWind, las *Capas* (`Layers`) son componentes que permiten **superponer** y gestionar diferentes tipos de datos geoespaciales en la visualización 3D. Estas capas son fundamentales para **personalizar y enriquecer** la información visual presentada sobre el globo.

Una `Layer` en WorldWind es una entidad que puede contener y gestionar datos geoespaciales específicos. Estas capas pueden incluir imágenes de satélite, mapas de terreno, rutas de vuelo, puntos de interés y más. Cada capa puede ser habilitada o deshabilitada individualmente, permitiendo a los usuarios personalizar la visualización de los datos según sus necesidades.

Las capas permiten añadir múltiples niveles de información sobre el globo, como imágenes de satélite y datos vectoriales. Además, pueden actualizarse en tiempo real, reflejando cambios en los datos geoespaciales a medida que ocurre la simulación. El orden en el que se renderizan las capas también es gestionable, lo que facilita el control de la prioridad de visualización de cada tipo de dato.

En el contexto de este simulador podemos diferenciar dos tipos de capas:

- Capas de representación del terreno **proporcionadas por WorldWind**. Entre ellas ordenadas de menos a más resolución (figura 6.3):
    1. NASA Blue Marble Image.
    2. Blue Marble May 2004.
    3. i-cubed Landsat.
    4. Bing Imaginery.
- Capas de **elaboración propia** para enriquecer la interfaz (figura 6.4):

(a) NASA Blue Marble Image. No tiene suficiente resolución para distinguir, por ejemplo, Palma de Mallorca

(b) Blue Marble May 2004

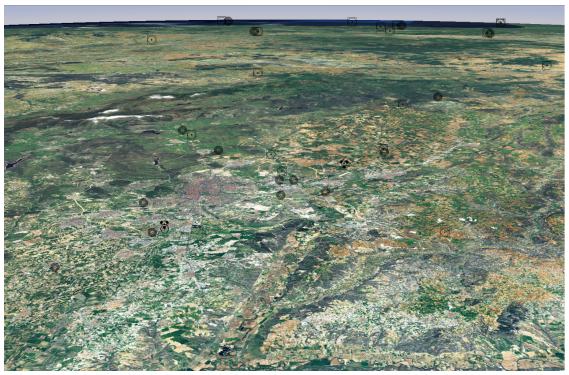
(c) i-cubed Landsat

(d) Bing Imaginery

Figura 6.3: Vista de Palma de Mallorca proporcionada por diferentes capas.

- **Capa de tráfico aéreo**. Incluye la representación gráfica de los aviones. Para ello se hace uso de la clase `TrafficPolygon` que no es más que un polígono definido por 4 vértices al que se le asigna la textura de un avión (figura 6.4(a)).
- Las capas de **pistas y de radioayudas** son otra pieza en la visualización. Estas capas extraen información de la base de datos de acceso libre proporcionada por *OurAirports* [9]. OurAirports es una comunidad que mantiene una base de datos global de aeropuertos y puntos de navegación aérea. Esta base de datos es de código abierto y ofrece información detallada sobre aeropuertos, pistas de aterrizaje, radioayudas (como VORs y NDBs), entre otros. Gracias a la contribución colaborativa, OurAirports proporciona datos actualizados que son cruciales para aplicaciones de aviación y simulaciones, facilitando la integración y visualización de información geoespacial en proyectos como este simulador.

(a) Capa de aviones



(b) Capa de radioayudas (VOR, DME, etc.)

(c) Capa de pistas

Figura 6.4: Capas desarrolladas para el proyecto superpuestas a la visualización del globo.

En cuanto a las capas de representación del terreno, cabe destacar que obtienen su información a partir de una combinación de **descarga dinámica** desde servidores remotos y el uso eficiente del almacenamiento en **caché local** (véase figura 6.2). Cuando el usuario navega por diferentes áreas del globo, WorldWind se conecta a servidores externos para descargar los datos más recientes de imágenes de satélite, mapas topográficos y otros recursos

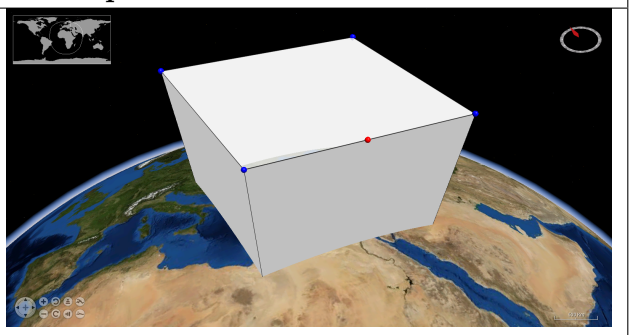
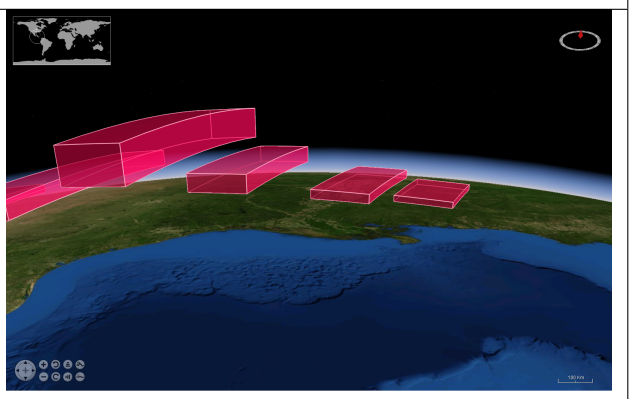
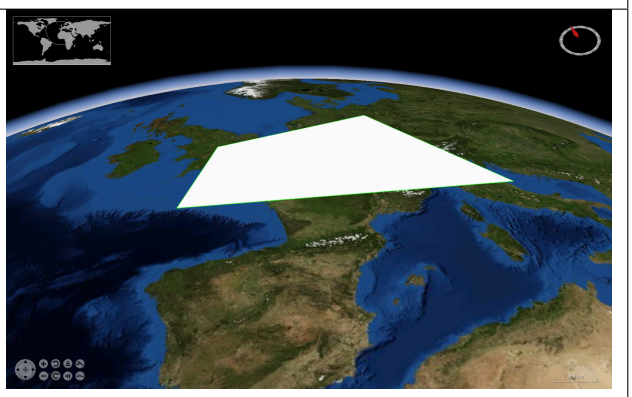
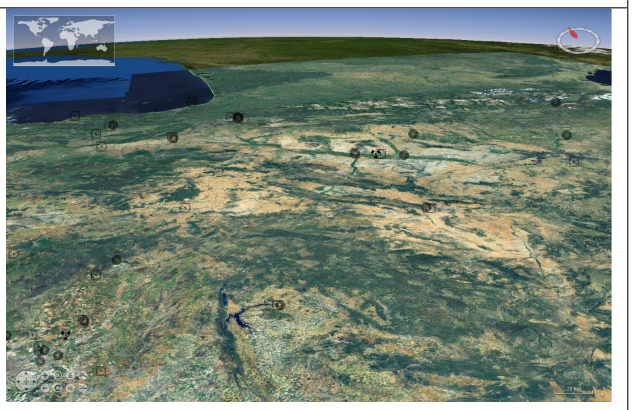
geoespaciales (6.5).

Esta información descargada es luego almacenada en la caché local de la aplicación. La caché actúa como un buffer que retiene los datos ya descargados, permitiendo que WorldWind acceda rápidamente a esta información cuando se vuelve a necesitar, sin requerir una nueva descarga desde los servidores. Esta estrategia no solo optimiza el rendimiento, reduciendo la latencia y el uso de ancho de banda, sino que también asegura que la visualización pueda continuar operando sin interrupciones incluso cuando la conexión a Internet no está disponible. Así, WorldWind combina de manera efectiva la obtención de datos en tiempo real y la eficiencia del almacenamiento en caché para ofrecer una experiencia de navegación fluida y detallada en su entorno tridimensional.

Figura 6.5: Ejemplo de descarga dinámica del terreno en WorldWind.

6.2. Elementos gráficos

La librería NASA WorldWind ofrece numerosos **elementos gráficos predeterminados** que facilitan la inclusión de diferentes componentes tanto en dos como tres dimensiones. En la tabla 6.1 se muestran algunos de los más utilizados. Posteriormente profundizaremos en `Polygon` y `UserFacingIcon` ya que son utilizados en este simulador para incluir varios elementos gráficos.

Nombre	Descripción	Descripción
`Volume`	Contiene diferentes clases para representación de figuras 3D como esferas, cilindros, conos, polígonos extruídos, etc.	
`Airspace`	Los `Airspace` son volúmenes extruidos en 3D definidos por coordenadas geográficas y límites de altitud superior e inferior.	
`Polygon`	Representa un polígono en tres dimensiones que queda definido por sus vértices.	
`UserFacingIcon`	Tipo de icono que siempre está orientado hacía el punto de vista de la cámara.	

Cuadro 6.1: Principales elementos gráficos que ofrece NASA WorldWind

6.2.1. Polígonos

Este proyecto aprovecha las capacidades de la clase `Polygon` principalmente de dos maneras:

- **Representación de aviones**. Es el uso principal y el elemento gráfico más importante. El simulador utiliza la clase llamada `TrafficPolygon` que tiene como atributo un polígono de la clase `Polygon` de WorldWind anteriormente mencionada para representar aviones. La posición (latitud, longitud y altura) del avión determina el centro del polígono. A partir de este centro se crea un cuadrilátero definiendo los cuatro vértices del polígono. Los polígonos en WorldWind permiten el uso de imágenes como texturas. Por lo tanto, se asigna la imagen de un avión (elegible por el usuario) al polígono y se orienta la dirección principal de dicha imagen utilizando el rumbo del avión. Por último se añade este polígono a la capa de aviones mostrada en la figura 6.4(a). Este proceso se resume en el esquema de la figura 6.6.

Figura 6.6: Esquema de construcción del componente gráfico del avión.

- **Pistas de aterrizaje**. El simulador hace tambiién uso de los polígonos para representar las pistas de aterrizaje de los aeropuertos. Se utiliza la información disponible en la base de datos *OurAirports* para construir el cuadrilátero a partir del ARP (Airport Reference Point), la longitud de la pista y la orientación de la misma. En la figura 6.7 se muestra un ejemplo de la diferencia entre un aeropuerto con la textura por defecto de WorldWind y otra con el polígono de la pista de aterrizaje superpuesta. Este polígono se puede utilizar para hacer cálculos en la aproximación y/o despegue.

(a) Sin polígono (b) Con polígono

Figura 6.7: Vista sin y con polígono de pista de aterrizaje

6.2.2. UserFacingIcon

La clase `UserFacingIcon` es un tipo de **icono** especial de WorldWind que siempre está orientado hacia el punto de vista de la cámara. El simulador lo utiliza en la clase `NavAid` para representar sobre el terreno las diferentes radioayudas en la capa de la figura 6.4(b). Es necesario indicar la latitud y longitud tanto como la imagen que desea utilizar para el icono. Los iconos disponibles se muestran en la tabla 6.2.

Radioayuda	**Icono**
NDB	
NDB-DME	
TACAN	
VOR	
VOR-DME	

Cuadro 6.2: Iconos utilizados en el simulador

6.3. Integración con el Simulador

En este apartado, se describe cómo se ha **integrado** NASA WorldWind con el simulador a través de la clase `GUI`. Esta clase actúa como el punto de entrada de la aplicación, siendo la cual ejecuta el usuario para inciar el simulador. Además, actúa como punto de unión entre los componentes de visualización geoespacial de WorldWind con los elementos de la simulación.

En primer lugar, la clase `GUI` utiliza como punto de partida la clase `ApplicationTemplate`. `ApplicationTemplate` es una plantilla ofrecida por el paquete WorldWind de como crear una aplicación en el mismo.

Esta plantilla tiene dos subclases llamadas `AppPanel` y `Aappframe` que heredan respectivamente de las clases `JPanel` y `JFrame` que son dos componentes fundamentales de la biblioteca `Swing`. No se va a entrar en detalle sobre este paquete, pero `Swing` es la **biblioteca estándar** para hacer interfaces de usuario en java. Estos dos componentes permiten renderizar ventanas y widgets en el sistema operativo.

La clase `GUI`, al igual que su homólogo `ApplicationTemplate`, está compuesta por dos subclases principales, `AppPanel` y `AppFrame`, cada una encargada de diferentes aspectos de la integración y visualización:

- `AppPanel` Es un panel de `Swing` que contiene el componente WorldWindow, el cual se encarga de la visualización 3D. Genera el globo (véase sección 6.1.1) necesario para el funcionamiento de worldwind y lo inserta en dicho panel. Adicionalmente añade algunos componentes, proporcionados por WorldWind a la visualización del globo como:
  - `StatusBar`. Indica variables como la altura del punto de vista actual o coordenadas del cursor sobre el globo.
  - `ToolTipController`. Inserta una serie de utilidades para manipular el globo como hacer zoom, rotar, exagerar la elevación, etc.
- `AppFrame`

  Es la ventana principal de la aplicación (hereda de `JFrame`). Esta subclase de GUI es el verdadero punto conexión entre la simulación y la interfaz gráfica. En la tabla 6.3 se muestran los atributos más relevantes de `AppFrame`.

El diagrama de flujo de la figura 6.8, ilustra el proceso de inicialización del simulador a grandes rasgos. El proceso da comienzo cuando el usuario ejecuta el simulador. En primer lugar, se crea una instancia de la clase `AppFrame`. Esta a su vez en su método de inicialización instancia la clase `AppPanel`, con los componentes de WorldWind que esta implica. A continuación, se crea una instancia de la clase `Simulation` **uniendola de esta manera con la interfaz gráfica**. Después se añaden los diferentes menús al panel de control dando lugar a la estructura que se muestra en la figura 6.9. Por último, se añaden todas las capas de representación al globo. Finalmente, el simulador se pone en funcionamiento y queda a la espera de ser cerrado.

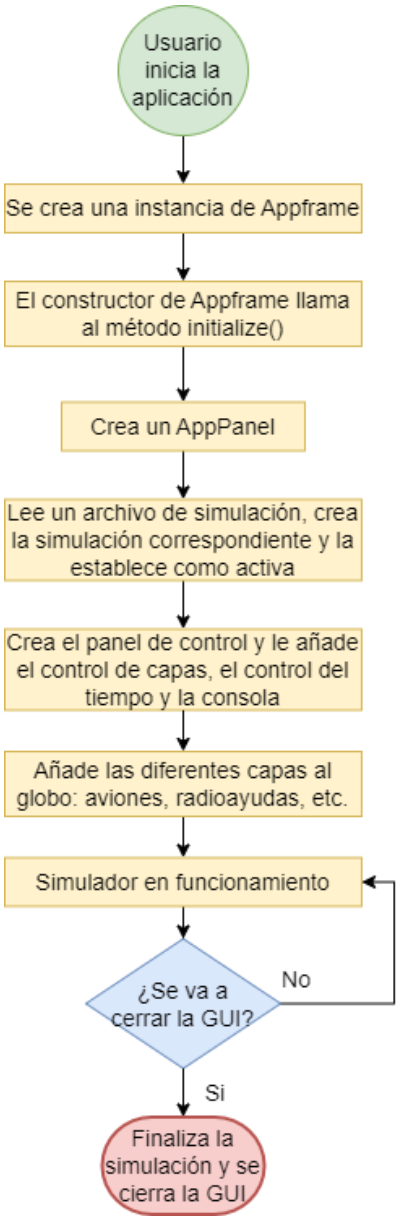


Figura 6.8: Diagrama de flujo de la interfaz gráfica.

Nombre	Tipo	Descripción
`wwjPanel`	`AppPanel`	Este es el punto de unión con el `AppPanel` que incluye el globo de WorldWind como hemos mencionado anteriormente.
`controlPanel`	`JPanel`	Panel auxiliar donde se incustran el resto de paneles.
`layerPanel`	`LayerPanel`	Menú que permite al usuario activar o desativar capas del globo.
`detailViewPanel`	`DetailViewPanel`	Vista de detalle que aparece cuando se clicka sobre un avión para mostrar su información.
`simulationMap`	`Map`	Estructura de datos para almacenar todas las simulaciones. Este es el **punto de unión** entre la interfaz gráfica y la simulación.
`activeSimulation`	`Simulation`	De todas las simulaciones posibles, este atributo indica cual está activa y por lo tanto ha de mostrarse al usuario.
`trafficDisplayer`	`TrafficDisplayer`	Esta clase es un *listener* de *TrafficSimulated* y refresca la capa de aviones de la interfaz gráfica cada vez que uno de estos aviones actualiza su estado.

Cuadro 6.3: Atributos principales de la clase `AppFrame`.

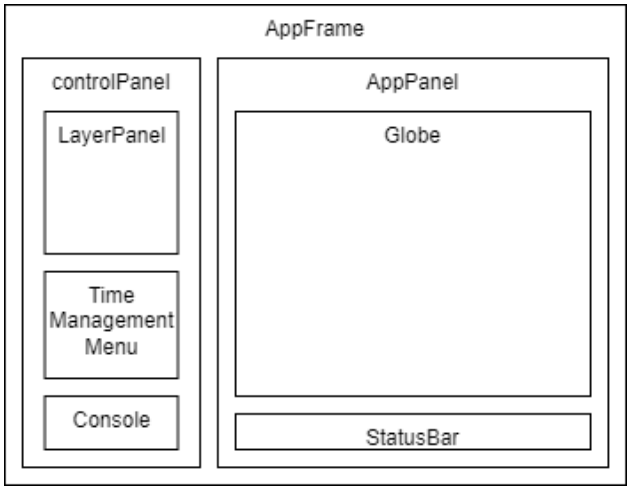


Figura 6.9: Estructura de la interfaz gráfica

6.4. Interfaz Gráfica

El objetivo de este último apartado referente a la interfaz gráfica es mostrar su aspecto final ante el usuario. Se muestra la interfaz en su conjunto en la figura 6.10 y se explican sus componentes individualmente en el siguiente listado:

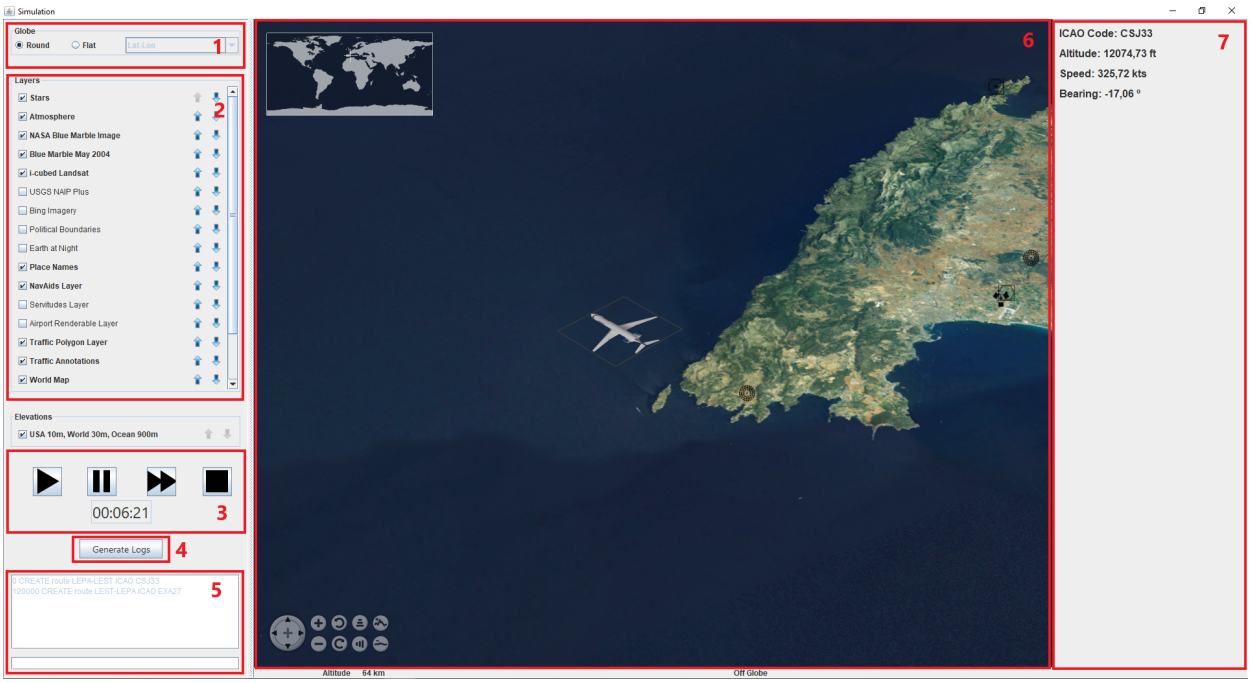


Figura 6.10: Interfaz gráfica de usuario (Componentes separados por bloques rojos para visualización).

1. **Panel de control del globo**. La representación en 3 dimensiones puede ser tanto esférica como plana. Este menú permite al usuario cambiar entra ambas y elegir el tipo de proyección plana (mercator, Lat-Lon, sinusoidal, etc) que desea.

2. **Panel de control de capas**. Se muestran todas las capas de representación disponibles en el globo y el usuario puede activarlas o desactivarlas. Por ejemplo, puede interesar desactivar la capa de radioayudas para no saturar el terreno con iconos en algunas zonas.

3. **Panel de control temporal**. En primer lugar muestra el tiempo de simulación a partir del momento inicial. Permite: iniciar, pausar, doblar la velocidad y resetear la velocidad de la simulación.

4. **Logs**. Botón para extraer los logs de todos los aviones de la simulación hasta ese momento. Se puede pulsar tantas veces como se desee.

5. **Consola de comandos**. Permite al usuario comunicarse directamente con la simulación vía una serie determinada de comandos (véase 5.1.3).

6. `AppPanel`. Ya mencionado anteriormente, contiene el componente de WorldWind, el cual se encarga de la visualización.

7. **Vista de detalle**. Consiste en un simple panel que se desliza desde la derecha cuando el usuario clicka encima de un avión y desaparece cuando clicka de nuevo en el mismo. Contiene las variables del avión en cada instante.

Capítulo 7

Caso de uso

El objetivo de este apartado es mostrar de forma breve un **ejemplo** de como ejecutar un caso de uso de principio a fin. Se plantea el escenario de, por ejemplo, un usuario que quiere comprobar como funciona el modelo TCAS implementado para la resolución de conflictos.

7.1. Archivos de simulación

En primer lugar, se crea un archivo de simulación, de acuerdo a lo indicado en el apartado 5.1.3. Para simular dos aviones que van a colisionar, lo más sencillo es hacerlos volar la misma ruta (con mismo perfil vertical) pero en sentidos opuestos. Para ello creamos el siguiente archivo se simulación:

```
2000,CREATE,ICAO:EXA27,route:LEST-LEPA
2000,CREATE,ICAO:CSJ33,route:LEPA-LEST
```

Este archivo contiene dos comandos `CREATE` que se ejecutan en el segundo 2 de la simulación. El primero crea un avión con identificador EXA27 que vuela desde el aeropuerto de Santiago de Compostela hasta el aeropuerto de Palma de Mallorca. El siguiente, crea el avión con CSJ33 para la misma ruta pero en dirección contraria.

7.2. Configuración

Existe una clase en el paquete `Utils` llamada `Config` donde se pueden configurar unos pocos parámetros (extensible en el futuro) de la simulación, dichos parámetros se muestran en la tabla 7.1.

En `simulationFilePath` indicamos la ruta a nuestro recién creado archivo de simulación y podemos dejar el resto de parámetros con los valores por defecto. Adicionalmente, en el método `setModels` de clase `Simulation` podríamos cambiar los modelos por defecto si así lo desearamos, como se explica en el apartado 5.3.4.

Atributo	Valor por defecto	Descripción
`nameOfDefaultSimulation`	DefaultSimulation	Nombre de la simulación, para cuando en el futuro se implementen varias simulaciones simultáneas.
`simulationFilePath`	src/simulation files/testTCAS.csv	Ruta relativa al archivo de simulación que se quiera ejecutar.
`logsPath`	logs/	Ruta relativa a la carpeta donde se almacenarán los logs.
`dbUrl`	jdbc:sqlite:src/OurAirports/OurAirports.db	Ruta relativa a la base de datos local de *OurAirports*.
`specificCountry`	ES	Código del pais del cual se quieran cargar las radioayudas, aeropuertos, etc. Esto se hace para evitar cargar todos los del planeta en la interfaz gráfica si la simulación está acotado por ejemplo a España.

Cuadro 7.1: Atributos clase `Config`.

7.3. Ejecutar el simulador

El siguiente paso sería iniciar el simulador, tal y como se indica en la guía de usuario (en el archivo `readme.MD`) del repositorio, ejecutando la clase `GUI`. Uno vez ejecutado, iniciamos la simulación dándole al botón de iniciar en el panel de control temporal, adicionalmente también podemos acelerarla en el mismo menú. En la figura 7.1 se puede observar a ambos aviones en rumbo de colisión en las cercanías de Pamploma.

Una vez la simulación haya terminado o bien el suceso el cual interesa al usuario haya sucedido, se pulsa el botón *Generate Logs* para obtener en la carpeta *logs* los archivos .csv, uno por avión, con las variables: velocidad, latitud, longitud, altura y rumbo a lo largo del tiempo. Una vez extraídos los logs se puede dar por finalizada la simulación cerrando el simulador.

7.4. Postprocesar los resultados

Una vez tenemos los .csv generados por el simulador, el usuario puede postprocesarlos en su herramienta de analítica de datos habitual (excel, matlab, python, etc). En este caso se ha creado un simple *jupyter notebook*[13] para dibujar una gráfica con la librería *matplotlib*.

En este caso de uso nos interesa ver como ha gestionado el TCAS la separación vertical de los aviones cuando entran en una situación de *Resolution Advisory (RA)*, la cual se explica en el apartado 5.3.3. Como podemos ver en la figura 7.2, llegados a un punto cercano

a la mitad de la ruta (evidentemente ya que están haciendo la misma en sentidos opuestos) el TCAS **detecta una posible colisión** y ordena al avión EXA27 ascender y al CSJ33 descender.

Figura 7.1: Avión EXA27 y CSJ33 en rumbo de colisión

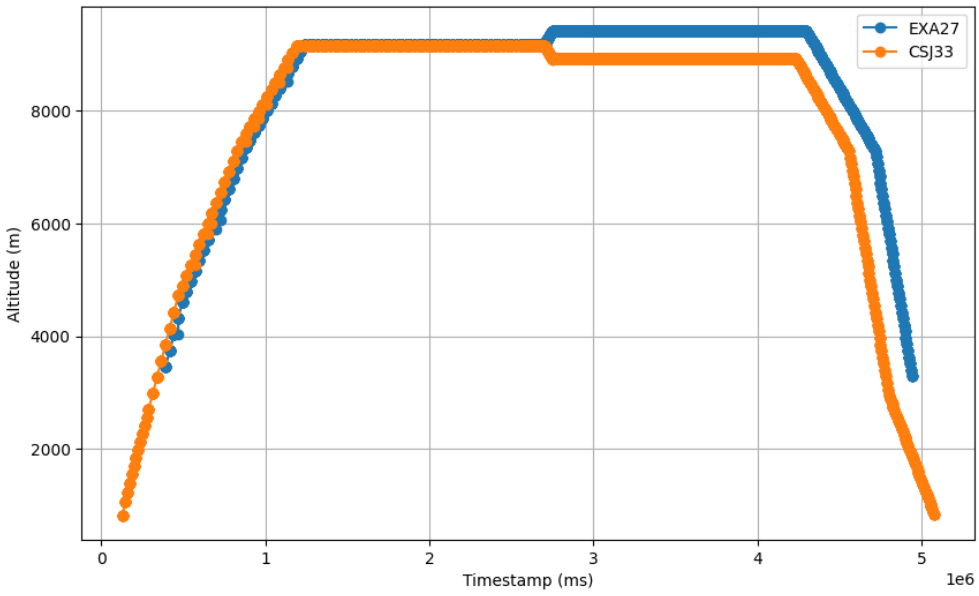


Figura 7.2: Evolución de la altura a lo largo del tiempo.

Capítulo 8

Conclusiones y líneas futuras

En este apartado final se presentan las conclusiones del proyecto, destacando tanto los logros alcanzados como las áreas que requieren mayor desarrollo, las cuales se detallan en las futuras líneas de trabajo.

8.1. Conclusiones

En este trabajo, se ha desarrollado un simulador aeronáutico que combina de manera efectiva la ingeniería aeronáutica y la ingeniería de software, creando una **plataforma abierta y extensible**. Este proyecto ha demostrado que es posible integrar estas disciplinas para producir una herramienta útil y adaptable, facilitando tanto el aprendizaje como la investigación.

Este enfoque ha permitido que el simulador no solo realice cálculos en un segundo plano, sino que también ofrezca una **representación visual interactiva** en tiempo real de los escenarios de simulación. Presentando al usuario la información de una manera más amigable. La utilización de Java y WorldWind ha sido fundamental para desarrollar una interfaz gráfica funcional y una gestión avanzada de datos geoespaciales.

El proyecto ha logrado crear una plataforma abierta y extensible. La arquitectura modular del simulador, basada en interfaces en Java, permite que otros desarrolladores e investigadores añadan, modifiquen o sustituyan componentes con facilidad. Al alojar el código en un **repositorio público** en una plataforma ampliamente conocida como GitHub, se facilita el acceso y se fomenta la colaboración. Esto permite que la comunidad contribuya con mejoras y nuevas funcionalidades, asegurando la evolución continua del simulador.

No obstante, a lo largo de este proyecto se han encontrado **varios obstáculos**. La gestión de varios hilos de ejecución simultáneos ha sido un desafío, debido a la necesidad de coordinar muchas clases que interactuan entre si. Además, leer y entender la documentación de librerías extensas como WorldWind ha sido complicado, ya que no siempre es fácil (o posible) encontrar la información necesaria. La búsqueda de bases de datos fiables y abiertas al público con información aeronáutica de aeropuertos y otros datos importantes, así como

la localización de artículos académicos y sitios que expliquen en detalle el funcionamiento de diversos sistemas del avión, también han presentado **dificultades significativas**. Sin embargo, estos desafíos han sido superados exitosamente.

En resumen, el simulador desarrollado cumple con los objetivos propuestos en el inicio de este documento, proporcionando una herramienta versátil que integra la ingeniería aeronáutica y de software. Al tratarse de un proyecto de fin de máster, su escala es reducida. Sin embargo, su diseño abierto y extensible lo convierte en una base útil para futuras investigaciones y desarrollos. Se espera que la comunidad académica y de investigación pueda aprovechar esta plataforma, contribuyendo a su crecimiento y mejora continua.

8.2. Líneas futuras

Aunque se ha establecido una base sólida, el proyecto no profundiza en muchos aspectos de la realidad aeronáutica. Se han implementado numerosas simplificaciones para hacer el simulador más accesible y viable con los medios y conocimientos al alcance del autor. Esta decisión garantiza que el proyecto sea una plataforma base que otros puedan **ampliar y mejorar**, incorporando mejoras en el futuro. La mayoría de estas mejoras estarían en el campo de la aeronáutica ya que el proyecto se centra más en el software, algunas sugerencias son:

- **Mecánica de vuelo**. Como se ha visto, el movimiento del avión es muy simplicado, avanzando a una velocidad constante (determinada por la fase del vuelo) en una dirección. Sería un paso interesante incorporar ecuaciones de mecánica de vuelo incluyendo las fuerzas en el simulador (empuje, sustentación, resistencia y peso) que permitirían realizar cálculos más interesantes desde el punto de vista aeronáutico.
- **Modelos meteorológicos**. En el simulador actual, se asume una atmósfera estándar es decir en calma, añadir soporte para tener en cuenta variables como el viento o gradientes de presión añadiría realismo a la simulación.
- **Perfil vertical**. El simulador actual, ya cuenta con la clase `VerticalProfile` para definir el mismo. El usuario puede definir uno propio si así lo desea pero por defecto se utiliza el perfical vertical del Airbus A320 extraído de Eurocontrol [14]. Sería de gran utilidad añadir los perfiles verticales de los aviones más utilizados para disponer de más opciones por defecto. Adicionalmente, hay que mejorar el cálculo del TOC (*Top Of Climb*) y TOD (*Top Of Descent*) del perfil vertical.
- **Comandos**. Se ha dejado lista la infraestructura para poder interacturar en directo con el simulador vía comandos. Sin embargo, no existen demasiados comandos disponibles de partida. La adición de más comandos haría la experiencia de usuario más satisfactoria.
- **Rendimiento**. Como se comenta en el apartado 5.1.1, se utiliza la clase `Thread` para pararelizar en la medida de lo posible las simulaciones, aviones y pilotos. Esto permite

un uso de la interfaz gráfica fluído y un mejor gestión de los recursos. Sin embargo, sería neceseario estudiar si esto es suficiente para hacer simulaciones a gran escala, con cientos o miles de aviones, o si serian necesarias más acciones.

- **Documentación**. El código fuente incorpora documentación en la mayoría de clases y funciones relevantes. Sin embargo, quedan secciones del simulador por documentar en profundidad. Una documentación exhaustiva es otra clave de los proyectos de colaboración abierta.

Por último, estas líneas de trabajo representan solo una sugerencia para futuras mejoras. Se invita a terceros, a ampliar este simulador según sus propias necesidades y objetivos que puede nada tengan que ver con estas sugerencias. La estructura abierta y extensible del proyecto está diseñada precisamente para facilitar este tipo de contribuciones y adaptaciones, permitiendo que cada usuario lo desarrolle y lo personalice como considere más oportuno.

Bibliografía

[1] E. Ampomah, E. Mensah, and A. Gilbert, "Qualitative assessment of compiled, interpreted and hybrid programming languages," *Communications on Applied Electronics*, vol. 7, pp. 8–13, 10 2017.

[2] "Microsoft flight simulator." Accessed: 2024-05-26.

[3] P. Cao, X. Hu, and G. Zhang, "Interface research and flight control based on flightgear," in *2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pp. 397–402, 2017.

[4] J. Sun, J. M. Hoekstra, and J. Ellerbroek, "Openap: An open-source aircraft performance model for air transportation studies and simulations," *Aerospace*, vol. 7, no. 8, p. 104, 2020.

[5] L. P. Lozano, "Simulación y guiado de aeronaves utilizando programación orientada a objetos," 2016.

[6] J. R. Robles Gómez, "Desarrollo de un simulador de vuelo sil en el entorno simulink para el autopiloto veronte," 2023. Curso Académico: 2022/2023.

[7] A. Upadhyay, V. Laxmi, and S. Naval, "Navigating the concurrency landscape: A survey of race condition vulnerability detectors," *arXiv preprint arXiv:2312.14479*, dec 2023. License: CC BY 4.0.

[8] Federal Aviation Administration, *Pilot's Handbook of Aeronautical Knowledge*. Flight Standards Service, 2016. FAA-H-8083-25B, Chapter 8: Flight Instruments, Turn Coordinator.

[9] O. Community, "Ourairports," 2024. Último acceso: 15 de junio de 2024.

[10] International Organization for Standardization, *Standard Atmosphere*. ISO 2533:1975, first edition ed., May 1975. Identical with the ICAO and WMO Standard Atmospheres from -2 to 32 km.

[11] A. Narkawicz, C. A. Muñoz, and A. Dutle, "Coordination logic for repulsive resolution maneuvers," *NASA Langley Research Center, Hampton, VA, 23881, US*, 2018.

[12] NASA WorldWind Development Team, "Nasa worldwind," 2024. Último acceso: 15 de junio de 2024.

[13] The Jupyter Project Contributors, *Project Jupyter*. Project Jupyter, 2023. Accessed: 2024-06-27.

[14] EUROCONTROL, "Aircraft performance database," 2024. Último acceso: 27 de junio de 2024.

[15] "Transformar nuestro mundo: la agenda 2030 para el desarrollo sostenible," 2024. Último acceso: 27 de junio de 2024.

[16] Oracle Corporation, *Javadoc Tool*, 2014. Accessed: 2024-06-29.

Apéndice A

Objetivos de Desarrollo Sostenible

La Agenda 2030 para el Desarrollo Sostenible, adoptada por todos los Estados Miembros de las Naciones Unidas, proporciona un plan compartido para la paz y la prosperidad para las personas y el planeta, ahora y en el futuro [15]. En la siguiente tabla, se detalla cómo el presente trabajo fin de máster se relaciona con varios de los **Objetivos de Desarrollo Sostenible (ODS)**.

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**Nulo**
1. Fin de la pobreza.				X
2. Hambre cero.				X
3. Salud y bienestar.				X
4. Educación de calidad.	X			
5. Igualdad de género.				X
6. Agua limpia y saneamiento.				X
7. Energía asequible y no contaminante.			X	
8. Trabajo decente y crecimiento económico.	X			
9. Industria, innovación e infraestructuras.	X			
10. Reducción de las desigualdades.				X
11. Ciudades y comunidades sostenibles.				X
12. Producción y consumo responsables.		X		
13. Acción por el clima.		X		
14. Vida submarina.				X
15. Vida de ecosistemas terrestres.				X
16. Paz, justicia e instituciones sólidas.				X
17. Alianzas para lograr objetivos.	X			

Cuadro A.1: Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Por último, se mencionan más en detalle aquellos ODS que tienen un grado de relación con el proyecto distinto de nulo.

- **Educación de calidad**. Quizás el objetivo con el que más se identifica este proyecto.

El simulador promueve una educación de calidad al proporcionar una herramienta de simulación que puede ser utilizada en instituciones educativas para la formación de estudiantes. La plataforma abierta y extensible permite a los educadores integrar contenido práctico en sus cursos, facilitando un aprendizaje interactivo.

- **Energía asequible y no contaminante** El simulador puede utilizarse para investigar y optimizar rutas de vuelo, lo que podría llevar a una reducción en el consumo de combustible y, por ende, a una disminución de las emisiones de gases contaminantes. Al promover la eficiencia energética en la aviación, el proyecto contribuye a la meta de lograr una energía más asequible y limpia.

- **Trabajo decente y crecimiento económico**. El desarrollo y la utilización del simulador fomentan la creación de empleos de alta cualificación en el sector de la tecnología y la ingeniería aeronáutica. Además, la mejora en la formación de los profesionales del sector puede impulsar la innovación y el crecimiento económico dentro de la industria aeronáutica.

- **Industria, innovación e infraestructuras**. Este proyecto apoya el desarrollo de infraestructuras tecnológicas avanzadas y fomenta la innovación en la industria aeronáutica. La arquitectura modular y extensible del simulador facilita la incorporación de nuevas tecnologías y metodologías, promoviendo un entorno de investigación y desarrollo continuo.

- **Ciudades y comunidades sostenibles**. La optimización de las rutas de vuelo y la mejora en la gestión del tráfico aéreo pueden contribuir a reducir el ruido y la contaminación en las zonas urbanas cercanas a los aeropuertos. Esto ayuda a crear ciudades y comunidades más sostenibles, mejorando la calidad de vida de sus habitantes.

- **Acción por el clima**. El simulador puede ser una herramienta crucial para estudiar y mitigar los efectos del cambio climático en la aviación. Al permitir la simulación de diferentes escenarios climáticos y su impacto en las operaciones de vuelo, el proyecto apoya los esfuerzos globales para combatir el cambio climático y sus efectos.

- **Alianzas para lograr los objetivos**. El hecho de que el proyecto esté alojado en un repositorio público y sea de código abierto facilita la colaboración entre investigadores, desarrolladores y educadores de todo el mundo. Esta colaboración global es esencial para avanzar en los Objetivos de Desarrollo Sostenible, promoviendo alianzas que potencien la innovación y el intercambio de conocimientos.

Apéndice B

Presupuesto

El presupuesto de todo proyecto se puede desglosar en coste de material, de *software* y humano.

- **Coste material**. No se han incurrido en costos materiales para la realización de este proyecto, ya que todo el trabajo se ha llevado a cabo utilizando recursos informáticos existentes y disponibles sin costo adicional.
- **Coste de *software***. El desarrollo del proyecto ha utilizado software y librerías de código abierto y gratuitos, como `Java`, `NetBeans`, y `NASA WorldWind`. Por lo tanto, no se han incurrido en costos de software.
- **Coste humano**. incluye las horas dedicadas por el ingeniero encargado del desarrollo del proyecto y el tiempo invertido por el tutor en la supervisión y guía del mismo. Los costes unitarios se han calculado en función de salarios medios disponibles para ingenieros según grado de experiencia. A continuación, se detallan estos costos:

Actividad	**Coste unitario [€/h]**	**Tiempo [h]**	**Coste [€]**
Investigación bibliográfica	10	30	300
Análisis de requisitos	10	10	100
Desarollo del simulador	10	300	3000
Documentación y memoria	10	100	1000
Supervisión y tutorización	30	30	900
Total			**5300 €**

Cuadro B.1: Actividades que implican coste humano

Sumando todos los costos de la tabla B.1, obtenemos un coste aproximado de 5300€ para la realización de este proyecto.

Apéndice C

Pliego de condiciones

Un **pliego de condiciones** en el contexto de un proyecto de *software* es un documento que define la **relación** entre el desarollador y el usuario. Definiendo que es lo que se puede esperar del *software* y que es lo que se espera del usuario para utilizar el *software*. A lo largo de este anexo se mencionan algunas de estas condiciones.

C.1. Condiciones de uso

- **Conocimientos previos**. Es necesario unos conocimientos previos de java y de programación orientada a objetos para que el usuario sea capaz de extender la aplicación por si mismo. Para un uso exclusivo de las funcionalidades ya existentes, la lectura de esta memoria sería suficiente pero desaconsejable.
- ***Software* necesario**. El simulador está escrito en Java 11. Para ejecutar el simulador, es suficiente tener instalado el *Java Runtime Environment* (JRE) correspondiente a Java 11. Sin embargo, si deseas desarrollar, modificar o ampliar el software, necesitarás el *Java Development Kit* (JDK) 11, que incluye las herramientas necesarias para compilar y depurar el código.
- ***Hardware* necesario**. Es necesario de disponer de los recursos de hardware mínimos para correr Java 11 que se pueden consultar en la web oficial de oracle. Así como la cantidad de memoria en disco suficiente para almacenar la caché de Nasa WorlWind como la base de datos SQLite con la información de OurAirports.
- **Internet**. Como ya se ha explicado, las capas geospaciales del terreno de Nasa WorldWind funcionan con una combinación de cacheado y descarga de las mismas a un servidor de la NASA. Por lo tanto, es necesario, al menos inicialmente, **disponer de conexión a internet** para poder descargar el terreno.

C.2. Condiciones de distribución

Este proyecto está licenciado bajo los términos de la **Licencia MIT**. Esto permite a cualquier persona copiar, modificar, distribuir y utilizar el software para cualquier propósito,

incluyendo trabajos derivados, siempre y cuando se reconozca la autoría original. No existen restricciones sobre el uso o la redistribución del software.

C.3. Mantenimiento y soporte

No se garantiza el mantenimiento y soporte continuado del simulador si no se espera que sea parte de la comunidad. Las funcionalidades se irán añadiendo y el soporte a dudas y problemas se proporcionará según la disponibilidad de tiempo. No hay un compromiso formal de asistencia o actualizaciones periódicas, por lo que el usuario debe tener en cuenta esta limitación al utilizar el simulador.

C.4. Responsabilidades

El autor y/o potenciales colaboradores no se hacen responsable de los resultados obtenidos mediante el uso de este simulador ni de los fines para los cuales se utilice. **Cada usuario es responsable** de verificar y validar sus propios resultados. Asimismo, no asumimos responsabilidad alguna por las modificaciones que los usuarios puedan hacer libremente al simulador.

C.5. Gestión del repositorio

El repositorio del proyecto (https://github.com/oplaco/Simulator) está abierto a contribuciones de la comunidad. Cualquier persona puede contribuir mediante la creación de **pull requests**, las cuales serán revisadas y validadas por el autor del proyecto y por futuros administradores que se puedan añadir. Se agradecen las contribuciones que mejoren la funcionalidad, corregir errores o añadan nuevas características, siempre y cuando cumplan con las directrices y estándares establecidos en el repositorio.

C.6. Documentación

La documentación disponible se encuentra directamente en el código, en las clases y métodos, utilizando el estilo de comentarios **Javadoc**[16]. Este estilo proporciona una descripción detallada de la funcionalidad, parámetros y comportamiento de cada componente del software documentado. No se garantiza que la totalidad del *software* esté documentado o que dicha documentación esté actualizada.