



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



**TFG del Grado en Ingeniería
Informática**

**Motor de físicas en Java con
cliente web.
Documentación Técnica**



Presentado por Daniel Meruelo Monzón
en Universidad de Burgos — 15 de febrero
de 2024

Tutor: Alejandro Merino Gómez

Índice general

Índice general	i
Índice de figuras	iii
Índice de tablas	iv
Apéndice A Plan de Proyecto Software	1
A.1. Introducción	1
A.2. Planificación temporal	1
A.3. Estudio de viabilidad	2
Apéndice B Especificación de Requisitos	7
B.1. Introducción	7
B.2. Objetivos generales	7
B.3. Catalogo de requisitos	7
B.4. Especificación de requisitos	8
Apéndice C Especificación de diseño	15
C.1. Introducción	15
C.2. Diseño de datos	15
C.3. Diseño procedimental	21
C.4. Diseño arquitectónico	23
Apéndice D Documentación técnica de programación	27
D.1. Introducción	27
D.2. Estructura de directorios	27
D.3. Compilación, instalación y ejecución del proyecto	29

D.4. Manual del programador	31
D.5. Pruebas del sistema	38
Apéndice E Documentación de usuario	41
E.1. Introducción	41
E.2. Requisitos de usuarios	41
E.3. Instalación	41
E.4. Manual del usuario	42
Bibliografía	47

Índice de figuras

A.1. Régimen general de la Seguridad Social	3
B.1. Diagrama de casos de uso del cliente web	9
C.1. Ciclo de actualización constante	19
C.2. Diagrama de clases simplificado	20
C.3. Diagrama de clases completo	21
C.4. Arquitectura	25
D.1. Función a reemplazar/modificar para determinar la conexión al backend	31
D.2. Estructura de ficheros del proyecto en IntelliJ IDEA	33
D.3. Paquete <i>Engine</i> . Aquí se pueden incluir o modificar nuevos algoritmos.	34
D.4. Paquete <i>Service</i> . Aquí se puede modificar cómo se usan los distintos algoritmos o la configuración inicial.	35
D.5. Este es el servicio que se encarga de correr la simulación.	35
D.6. Aquí se pueden realizar cambios en la configuración de la aplicación.	36
D.7. Análisis final de Codacy	37
D.8. Análisis final de Code Climate	38
D.9. Reporte de la ejecución de los tests del proyecto	39
E.1. Ejemplo de cuerpo seleccionado junto a uno no seleccionado	43
E.2. Ejemplo de tarjeta seleccionada junto a tarjetas no seleccionadas	44
E.3. Ejemplo de aeronaves dirigiéndose al punto de búsqueda junto a una aeronave siguiendo una rutina normal	45

Índice de tablas

A.1. Costes de personal	3
A.2. Costes de equipamiento	3
A.3. Costes de página web	4
A.4. Coste total	4
A.5. Dependencias del motor de físicas y sus licencias	5
B.1. CU-1 Añadir una aeronave.	10
B.2. CU-2 Añadir una planeta.	10
B.3. CU-3 Añadir una asteroide.	11
B.4. CU-4 Añadir un cuerpo aleatorio.	11
B.5. CU-5 Eliminar un cuerpo.	12
B.6. CU-6 Seleccionar un cuerpo.	13
B.7. CU-7 Establecer un objetivo de búsqueda.	14

Apéndice A

Plan de Proyecto Software

A.1. Introducción

Este apartado se centra en la organización del proyecto. Se aborda la programación temporal y las distintas tareas previstas, junto con los análisis de viabilidad económica y legal realizados para garantizar la factibilidad del proyecto.

A.2. Planificación temporal

La organización temporal se diseñó siguiendo una adaptación de la metodología ágil Scrum. Se ajustó a las necesidades del proyecto:

- Se eliminaron las reuniones diarias.
- Se eliminaron los roles de scrum master y product owner.
- Se utilizó como herramienta orientativa para dividir el trabajo.

El desarrollo se estructuró en iteraciones o Sprints, cada uno culminando en una versión incrementalmente mejorada y más completa del motor de físicas. La duración de los Sprints se estableció inicialmente en dos semanas. Sin embargo, para optimizar la gestión del trabajo y mejorar la respuesta a los desafíos emergentes durante la fase final del proyecto, los Sprints se ajustaron a una duración de una semana.

Inicialmente, se realizó un ejercicio de identificación y creación de tareas en la plataforma ZenHub. Sin embargo, debido a su cambio de política, no

ofreciendo un plan para estudiantes y la limitada disponibilidad temporal, se decidió abandonar el uso de la plataforma.

En cada Sprint, el objetivo era poder entregar una versión tanto del backend como del frontend. Con esto en mente, se decidió dividir el proyecto en fases:

1. Primero, se decidió obtener una conexión funcional entre el backend y el frontend. Se desarrollaron las configuraciones de WebSocket y unos endpoints temporales para comprobar que se podían hacer llamadas desde el cliente web.
2. Después, se desarrolló el modelo y una versión primitiva del integrador. Junto a ello, se integró p5js en el frontend para renderizar los contenidos que llegaban desde el backend. Aprovechamos el bucle de dibujo de p5js para hacer llamadas continuamente al backend.
3. A partir de aquí, se empezaron a implementar los distintos tipos de cuerpos e implementaciones de los algoritmos. A la vez, se desarrolló la sección de tarjetas de información del frontend.
4. Finalmente, se trabajó en el aspecto estético del cliente (CSS) y se produjeron refinamientos en el comportamiento de las aeronaves.

A.3. Estudio de viabilidad

Viabilidad Económica

Este apartado examina el impacto económico de desarrollar el proyecto del motor de físicas con fines comerciales, incluyendo un desglose de los costes asociados.

Costes de Personal

El desarrollo del motor de físicas fue realizado por un programador a lo largo de 2 meses, con una dedicación a tiempo completo. Tomando en cuenta un salario bruto anual de 35000 euros y añadiendo los costes relacionados con el IRPF y la cotización a la seguridad social, el coste total se calcula de la siguiente forma:

Para los cálculos de la cotización a la seguridad social, se han tomado en cuenta los porcentajes correspondientes a contingencias comunes, desempleo, y formación profesional, sumando un total aproximado de 29,9%^[13].

Concepto	Coste
Salario mensual neto	2332,45€
Retención IRPF (20,03 %)	7010,5€
Seguridad Social (32,4 %)	11340,00€
Salario mensual bruto	2916,70€
Salario anual bruto (12 pagas)	35000€
Total 2 meses	7.723,40€

Tabla A.1: Costes de personal

TIPOS DE COTIZACIÓN (%)				
CONTINGENCIAS	EMPRESA	TRABAJADORES	TOTAL	Accidentes de Trabajo y Enfermedades Profesionales
Comunes	23,60	4,70	28,30	Tarifa Primas establecida en la disposición adicional cuarta Ley 42/2006, de 28 de diciembre, PGE 2007, en la redacción dada por la Disposición Final Quinta del RDL 28/2018 de 28 de diciembre (BOE del 29) siendo las primas resultantes a cargo exclusivo de la empresa
Horas Extraordinarias Fuerza Mayor	12,00	2,00	14,00	
Resto Horas Extraordinarias	23,60	4,70	28,30	
Mecanismo Equidad Intergeneracional (MEI)	0,58	0,12	0,7	

Figura A.1: Régimen general de la Seguridad Social

Costes de Equipamiento

El proyecto requirió el uso exclusivo de un portátil valorado en 500 euros, operando con Arch Linux como sistema operativo. Al ser Arch Linux gratuito y de código abierto, no se han contemplado costes adicionales por software. La amortización del portátil, estimada al 26 % anual durante 2 meses, resulta en:

Concepto	Coste	Amortización (1 mes)	Amortización (2 meses)
Portátil	500€	10,83€	21,66€
Total	500€	10,83€	21,66€

Tabla A.2: Costes de equipamiento

Costes de Página Web

La presentación del proyecto se planteó con la adquisición de un dominio .com por un coste de 10 euros anuales, sin considerar gastos adicionales de renovación en esta evaluación.

Concepto	Coste
Registro dominio	10,00€
Total	10,00€

Tabla A.3: Costes de página web

Coste Total

Al sumar los costes identificados previamente, el coste total del proyecto sería:

Concepto	Coste
Costes de personal	7.723,40€€
Costes de equipamiento	21,66€
Costes de página web	10,00€
Total	7.755,06€

Tabla A.4: Coste total

Posibles Fuentes de Ingreso

Las potenciales fuentes de ingreso para el motor de físicas incluirían la venta de licencias del software a desarrolladores y estudios de videojuegos, servicios de personalización y soporte técnico, así como la posibilidad de ofrecer una API en modelo SaaS (Software as a Service) para simulaciones físicas en la nube. Adicionalmente, se podría considerar la implementación de cursos y tutoriales pagos sobre cómo integrar y optimizar el uso del motor de físicas en proyectos de desarrollo de videojuegos.

Viabilidad Legal

La selección de una licencia adecuada para el motor de físicas es crucial para proteger la propiedad intelectual del proyecto y definir los términos bajo los cuales otros pueden usar, modificar y distribuir el software. La adopción de una licencia de código abierto, como MIT o GPL, podría fomentar una comunidad activa de colaboradores, mientras que una licencia comercial proporcionaría control exclusivo sobre el uso comercial del software. Será importante también considerar las licencias de terceros para asegurarse de que el uso de bibliotecas y herramientas externas esté en conformidad con los objetivos legales y comerciales del proyecto.

Licencia

Las licencias de las dependencias y librerías utilizadas en el proyecto son:

Dependencia	Descripción	Licencia
JDK 17[5]	Eclipse Adoptium	GPLv2
Spring Boot[1]	Framework para simplificar la configuración y ejecución de aplicaciones Spring	Apache License 2.0
JTS Core[4]	Biblioteca para el manejo de geometrías espaciales	EDL/BSD
SockJS Client[7]	Biblioteca para facilitar la comunicación entre cliente y servidor sobre WebSockets	MIT
Stomp WebSocket[2]	Protocolo de mensajería para facilitar la comunicación basada en suscripciones	Apache License 2.0
Lombok[8]	Utilidad para minimizar el código repetitivo en Java	MIT
React[9]	Biblioteca de JavaScript para construir interfaces de usuario	MIT
react-dom[10]	Paquete de React para el manejo del DOM	MIT
react-scripts [11]	Conjunto de scripts y configuración utilizados por Create React App	MIT
p5js[6]	Biblioteca de JavaScript que facilita la programación creativa en el canvas del navegador	LGPL
stompjs[3]	Implementación del protocolo STOMP en JavaScript	Apache License 2.0

Tabla A.5: Dependencias del motor de físicas y sus licencias

Dada la diversidad de licencias, desde BSD hasta MIT y Apache License 2.0, se optó por seleccionar una licencia que fuera compatible con todas ellas y adecuada para el proyecto. Se decidió utilizar la **licencia BSD de 3 cláusulas** para el proyecto. Esta licencia es conocida por su simplicidad y por permitir un amplio uso, redistribución y modificación del código fuente, incluyendo aplicaciones en software propietario, siempre que se cumplan las tres condiciones principales: no utilizar los nombres de los autores y contribuyentes para promocionar productos derivados sin permiso previo, incluir el aviso de copyright en redistribuciones del código fuente, y también en redistribuciones en forma binaria.

La elección de la licencia BSD-3 se alinea con las posibles vías de expansión futuras donde buscaríamos fomentar la colaboración abierta y ofrecer flexibilidad para el desarrollo futuro del proyecto, incluyendo aplicaciones comerciales. Esta licencia asegura que el motor de físicas pueda ser utilizado y adaptado de manera amplia, respetando las contribuciones de la comunidad y los requisitos de las dependencias utilizadas, a la vez que protege el proyecto de posibles usos indebidos del nombre de los contribuyentes.

Apéndice B

Especificación de Requisitos

B.1. Introducción

Este anexo aborda los objetivos planteados para este proyecto, identifica los requisitos funcionales establecidos para alcanzar tales metas, y describe los casos de uso asociados a estos requisitos.

B.2. Objetivos generales

- Diseñar y desarrollar un motor de físicas que simule un entorno natural simplificado, en dos dimensiones y que ofrezca una interfaz para que se puedan conectar clientes y poder interactuar con la simulación.
- Crear una interfaz gráfica que permita visualizar la simulación y que se pueda interactuar con ella.

B.3. Catalogo de requisitos

A continuación, se listan los requisitos específicos que se han definido para el proyecto:

RF 1: El usuario debe poder acceder a un cliente gráfico (web) en el que pueda interactuar con la simulación.

RF 1.1: El cliente debe ofrecer una sección con una representación gráfica de la simulación.

- RF 1.2:** El cliente debe ofrecer una sección con información específica de los cuerpos.
 - RF 1.3:** El cliente debe ofrecer una interfaz para poder interactuar con la simulación.
 - RF 1.4:** El programa debe ofrecer un entorno inicial.
 - RF 1.5:** El usuario debe poder añadir y eliminar cuerpos a la simulación.
 - RF 1.6:** El usuario debe poder seleccionar cuerpos para poder examinarlos mejor.
 - RF 1.7:** Una vez existan aeronaves en la simulación y estén seleccionadas, el usuario debe poder establecer un punto del entorno al cual quiere que estas se dirijan.
- RF 2:** El motor debe ser capaz de simular un entorno natural en dos dimensiones.
- RF 2.1:** El entorno debe tener unas dimensiones específicas.
 - RF 2.2:** El motor tiene que poder admitir varios tipos distintos de cuerpos.
 - RF 2.3:** El motor deberá calcular la posición y comportamiento de los cuerpos en función a unos principios físicos determinados.
 - RF 2.4:** El motor debe ser capaz de generar cuerpos en función de unos parámetros.
 - RF 2.5:** El motor debe ser capaz de generar un asteroide con un número de vértices aleatorio.
 - RF 2.6:** El motor debe ofrecer un método de conectarse con él para obtener el estado de la simulación.
 - RF 2.7:** El motor debe ofrecer una interfaz para poder interactuar con la simulación.

B.4. Especificación de requisitos

Diagrama de casos de uso

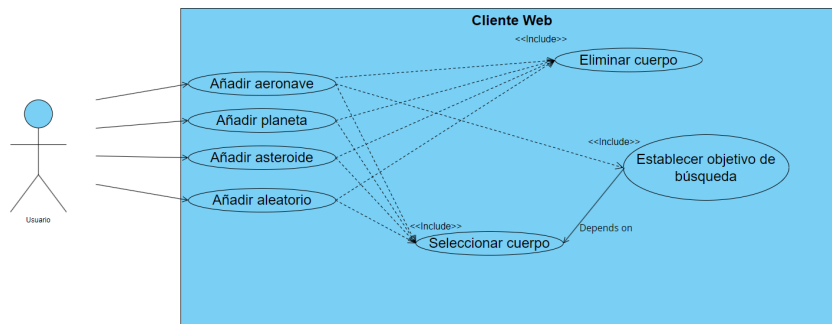


Figura B.1: Diagrama de casos de uso del cliente web

Actores

En nuestro sistema, solo existe un actor, el usuario. En caso de que, en un futuro, se implementen nuevas características, será necesario añadir más usuarios (administrador, multiusuario...)

Especificación de casos de uso

CU-1	Añadir una aeronave
Versión	1.0
Autor	Daniel Meruelo Monzón
Requisitos asociados	RF-1.3, RF-1.5, RF-2.2, RF-2.3, RF-2.4, RF-2.6, RF-2.7
Descripción	Permite al usuario añadir una aeronave a la simulación.
Precondición	El motor debe estar arrancado y el cliente web conectado a él.
Acciones	<ol style="list-style-type: none"> 1. El usuario accede al cliente. 2. El usuario utiliza el botón ‘Añadir aeronave’
Postcondición	Se crea un nuevo cuerpo del tipo Aeronave
Excepciones	N/A
Importancia	Alta

Tabla B.1: CU-1 Añadir una aeronave.

CU-2	Añadir un planeta
Versión	1.0
Autor	Daniel Meruelo Monzón
Requisitos asociados	RF-1.3, RF-1.5, RF-2.2, RF-2.3, RF-2.4, RF-2.6, RF-2.7
Descripción	Permite al usuario añadir un planeta a la simulación.
Precondición	El motor debe estar arrancado y el cliente web conectado a él.
Acciones	<ol style="list-style-type: none"> 1. El usuario accede al cliente. 2. El usuario utiliza el botón ‘Añadir planeta’
Postcondición	Se crea un nuevo cuerpo del tipo Planeta
Excepciones	N/A
Importancia	Alta

Tabla B.2: CU-2 Añadir una planeta.

CU-3	Añadir un asteroide
Versión	1.0
Autor	Daniel Meruelo Monzón
Requisitos asociados	RF-1.3, RF-1.5, RF-2.2, RF-2.3, RF-2.4, RF 2.5, RF-2.6, RF-2.7
Descripción	Permite al usuario añadir un asteroide a la simulación.
Precondición	El motor debe estar arrancado y el cliente web conectado a él.
Acciones	<ol style="list-style-type: none"> 1. El usuario accede al cliente. 2. El usuario utiliza el botón ‘Añadir asteroide’
Postcondición	Se crea un nuevo cuerpo del tipo Asteroide
Excepciones	N/A
Importancia	Alta

Tabla B.3: CU-3 Añadir una asteroide.

CU-4	Añadir un cuerpo aleatorio
Versión	1.0
Autor	Daniel Meruelo Monzón
Requisitos asociados	RF-1.3, RF-1.5, RF-2.2, RF-2.3, RF-2.4, RF 2.5, RF-2.6, RF-2.7
Descripción	Permite al usuario añadir un cuerpo de tipo aleatorio a la simulación.
Precondición	El motor debe estar arrancado y el cliente web conectado a él.
Acciones	<ol style="list-style-type: none"> 1. El usuario accede al cliente. 2. El usuario utiliza el botón ‘Añadir cuerpo aleatorio’
Postcondición	Se crea un nuevo cuerpo del tipo Asteroide Planeta Aeronave
Excepciones	N/A
Importancia	Alta

Tabla B.4: CU-4 Añadir un cuerpo aleatorio.

CU-5	Eliminar un cuerpo
Versión	1.0
Autor	Daniel Meruelo Monzón
Requisitos asociados	RF-1.1, RF-1.2, RF-1.3, RF-1.5, RF-1.6, RF-2.2, RF-2.3, RF-2.4, RF-2.6, RF-2.7
Descripción	Permite al usuario eliminar un cuerpo de la simulación.
Precondición	El motor debe estar arrancado y el cliente web conectado a él. Debe existir por lo menos un cuerpo en la simulación.
Acciones	<ol style="list-style-type: none"> 1. El usuario accede al cliente. 2. El usuario utiliza el botón ‘Eliminar’ en la tarjeta correspondiente al cuerpo que quiere eliminar.
Postcondición	Se elimina el cuerpo seleccionado de la simulación.
Excepciones	N/A
Importancia	Alta

Tabla B.5: CU-5 Eliminar un cuerpo.

CU-6	Seleccionar un cuerpo
Versión	1.0
Autor	Daniel Meruelo Monzón
Requisitos asociados	RF-1.1, RF-1.2, RF-1.3, RF-1.5, RF-1.6, RF-2.3, RF-2.7
Descripción	Permite al usuario seleccionar un cuerpo de la simulación para mejorar su visibilidad.
Precondición	El motor debe estar arrancado y el cliente web conectado a él. Debe existir por lo menos un cuerpo en la simulación.
Acciones	<ol style="list-style-type: none"> 1. El usuario accede al cliente. 2. El usuario hace click en un cuerpo en la representación gráfica o en la lista de cuerpos.
Postcondición	Si el cuerpo no estaba seleccionado, se selecciona. En la representación visual, el cuerpo se dibuja de otro color. En la representación de la información, la tarjeta cambia de color. Si estaba seleccionado, se efectúan los cambios a la inversa.
Excepciones	N/A
Importancia	Alta

Tabla B.6: CU-6 Seleccionar un cuerpo.

CU-7	Establecer objetivo de búsqueda
Versión	1.0
Autor	Daniel Meruelo Monzón
Requisitos asociados	RF-1.1, RF-1.2, RF-1.3, RF-1.5, RF-1.6, RF-1.7 RF-2.3, RF-2.6, RF-2.7
Descripción	Permite al usuario establecer un objetivo de búsqueda al cual se dirigirán las aeronaves seleccionadas.
Precondición	El motor debe estar arrancado y el cliente web conectado a él. Debe existir por lo menos un cuerpo de tipo aeronave en la simulación.
Acciones	<ol style="list-style-type: none"> 1. El usuario accede al cliente. 2. El usuario hace click en, al menos, un cuerpo del tipo aeronave en la representación gráfica o en la lista de cuerpos. 3. El cuerpo pasa al estado ‘seleccionado’. 4. El usuario hace click en un punto de la representación gráfica de la simulación.
Postcondición	Todos los cuerpos del tipo aeronave se dirigirán al punto indicado por el usuario, interrumpiendo sus rutinas habituales.
Excepciones	N/A
Importancia	Alta

Tabla B.7: CU-7 Establecer un objetivo de búsqueda.

Apéndice C

Especificación de diseño

C.1. Introducción

En este anexo se detallan las decisiones de diseño adoptadas para desarrollar un programa que satisfaga los requisitos especificados previamente. Se abordan las estructuras de datos empleadas en el programa, cómo se interrelacionan, las operaciones que llevan a cabo, y su organización a lo largo de los módulos del proyecto.

C.2. Diseño de datos

En esta sección se abordan las consideraciones respecto a cómo representar y almacenar los datos del programa que se han tomado en este proyecto. Dada la naturaleza del proyecto, no se ha considerado necesario la existencia de una base de datos (en un futuro, si se añaden usuarios o almacenaje de entornos, se podría considerar), así que se explicará el diseño de los datos del modelo y de algunos detalles de los algoritmos.

La clase **Body**

La clase **Body** representa los objetos físicos dentro de la simulación, como planetas, asteroides y naves espaciales. Cada *Body* tiene atributos clave que definen sus propiedades físicas y su estado en el entorno simulado:

- **ID:** Identificador único para cada cuerpo.

- **Position:** Ubicación del objeto en el espacio, representada por un *Vector2D*.
- **Velocity:** Velocidad actual del objeto, también un *Vector2D*.
- **Acceleration:** Aceleración actual, representada igualmente por un *Vector2D*.
- **Mass:** Masa del objeto, influencia cómo interactúa gravitacionalmente con otros objetos.
- **Bbox (Bounding Box):** Caja delimitadora que facilita la detección de colisiones.
- **BodyType:** Tipo de cuerpo (Planeta, Asteroide, Nave Espacial), que determina comportamientos específicos.
- **Selected:** Indica si el objeto está seleccionado en la interfaz de usuario.

Body es una clase abstracta, con varias subclases como *Planet*, *Asteroid*, y *Spaceship* que especifican comportamientos y propiedades adicionales. Por ejemplo, los asteroides tienen un atributo *spin* que representa su rotación, y las naves espaciales tienen métodos específicos como *wander*, *avoidCollisions*, y *seek*.

Especificaciones de Body

En el motor de físicas de nuestro proyecto, las clases *Planet*, *Asteroid*, y *Spaceship* son especializaciones de la clase abstracta *Body*, cada una representando diferentes entidades con comportamientos y propiedades específicas dentro de la simulación. A continuación, se realiza un análisis más detallado de estas clases.

- **La clase Planet:** La clase *Planet* representa los cuerpos celestes con masa significativa que ejercen una fuerza gravitacional sobre otros objetos. Los planetas son objetos estáticos dentro de la simulación, lo que significa que su velocidad inicial y aceleración son cero. Cada planeta tiene un *radius* que determina su tamaño visual en la simulación y su *BoundingBox* (Bbox), que se utiliza para las detecciones de colisión. La masa del planeta afecta la magnitud de la fuerza gravitacional que ejerce sobre otros cuerpos.

- **La clase Asteroid:** Los asteroides son cuerpos más pequeños y menos masivos que los planetas, con la capacidad de moverse a través del espacio. La clase *Asteroid* introduce el concepto de *spin*, que representa la rotación del asteroide sobre su eje. Esta rotación es puramente visual y no afecta la dinámica física del modelo. Los asteroides tienen una lista de *vertices* que define su forma irregular, la cual se calcula al instanciar el asteroide y se actualiza con el tiempo para simular la rotación. Al igual que los planetas, los asteroides tienen una masa y un radio que influyen en las interacciones gravitacionales y las colisiones.
- **La clase Spaceship:** La clase *Spaceship* modela naves espaciales controlables dentro de la simulación, capaces de realizar movimientos complejos gracias a métodos como *wander* (para moverse de forma aleatoria), *avoidCollisions* (para esquivar otros cuerpos), y *seek* (para dirigirse hacia un objetivo específico). Estos comportamientos se implementan aplicando fuerzas al vector de aceleración de la nave. La capacidad de maniobra de la nave está limitada por su *maxVelocity* y *maxForce*, que restringen la velocidad máxima y la fuerza máxima de aceleración que la nave puede alcanzar, respectivamente. Además, el método *applyDamping* simula una resistencia al movimiento, reduciendo gradualmente la velocidad de la nave para evitar que acelere indefinidamente.

Cada una de estas clases encapsula los datos y comportamientos específicos de los diferentes tipos de cuerpos dentro de la simulación, permitiendo una rica interacción entre ellos. La implementación detallada de estas clases facilita la simulación de un entorno dinámico donde la gravedad, las colisiones, y los movimientos dirigidos juegan un papel crucial en la evolución del sistema simulado. La flexibilidad de este diseño permite la fácil expansión del modelo para incluir nuevos tipos de cuerpos o comportamientos en futuras versiones del proyecto.

La Clase Vector2D

Vector2D es fundamental para el modelado de la física en el motor, proporcionando una representación de vectores en dos dimensiones que permite calcular posición, velocidad, y aceleración. Los métodos incluidos permiten operaciones vectoriales básicas como la adición, sustracción, multiplicación por un escalar, división, normalización, y cálculo de magnitud y distancia.

Los algoritmos de físicas

Los algoritmos de físicas se encapsulan en clases dedicadas a “resolver” aspectos específicos del comportamiento físico, tales como las atracciones gravitacionales, las colisiones, y el movimiento de las aeronaves.

■ **Attraction Resolver:**

- Propósito: Gestiona las fuerzas de atracción gravitatoria entre los cuerpos. Es fundamental para simular la interacción gravitacional que afecta el movimiento de los objetos en el espacio.
- Funcionamiento: Calcula la fuerza de atracción entre cada par de cuerpos y aplica la fuerza resultante a sus aceleraciones. Utiliza la ley de gravitación universal para determinar la magnitud de la fuerza basada en la masa de los cuerpos y la distancia entre ellos.
- Interacción con ***PhysicsService***: El “PhysicsService” invoca a “AttractionResolver” durante cada tick de la simulación para actualizar las fuerzas gravitatorias que actúan sobre los cuerpos.

■ **Collision Resolver:**

- Propósito: Detecta y maneja las colisiones entre cuerpos, asegurando una respuesta física realista como el rebote o la absorción.
- Funcionamiento: Verifica la intersección entre las áreas ocupadas por los cuerpos, determinando si ha ocurrido una colisión. Calcula el vector normal de la colisión y utiliza la conservación del momento para resolver la colisión, modificando las velocidades de los cuerpos involucrados.
- Interacción con ***PhysicsService***: Se encarga de revisar y resolver colisiones después de actualizar las posiciones de los cuerpos en cada ciclo de simulación.

■ **Movement Resolver:**

- Propósito: Controla el movimiento específico de las aeronaves, permitiendo comportamientos como el deambular, evitar obstáculos y buscar objetivos.
- Funcionamiento: Implementa algoritmos para mover las aeronaves de acuerdo a sus objetivos y el entorno circundante, ajustando su velocidad y dirección en base a factores como la evitación de colisiones y la navegación hacia puntos de interés.

- Interacción con ***PhysicsService***: Es utilizado para actualizar el estado de movimiento de las aeronaves en base a su lógica específica dentro del ciclo global de la simulación.

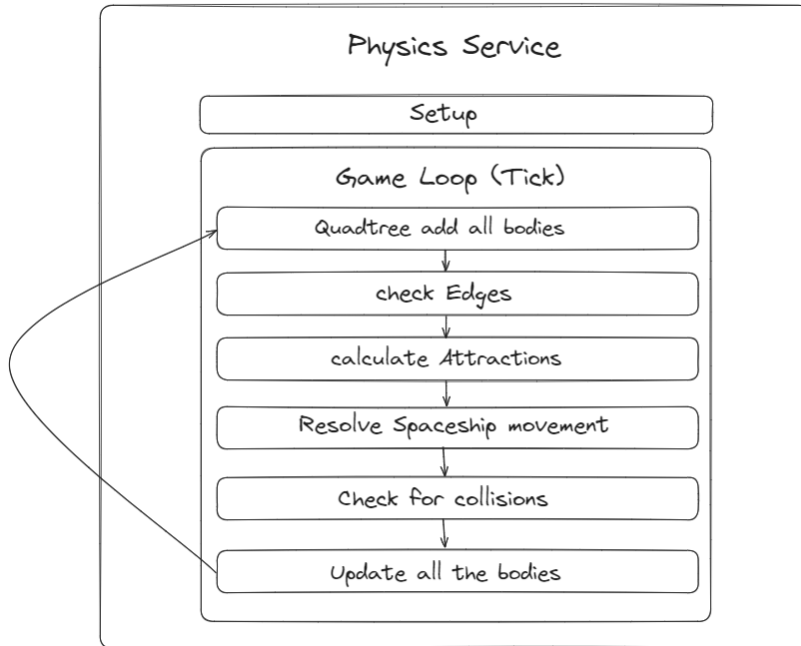


Figura C.1: Ciclo de actualización constante

El Quadtree

El Quadtree es una estructura de datos que divide el espacio de simulación en cuadrantes para optimizar la detección de colisiones y las interacciones entre objetos. Su implementación permite reducir la complejidad computacional al limitar las comprobaciones de colisiones y atracciones solo entre objetos cercanos.

La estructura del Quadtree se actualiza dinámicamente con la inserción y consulta de los objetos (Body) mediante sus *envoltorios de colisión* (BoundingBox). La inserción de un objeto se realiza mediante la creación de un *envelope* que encapsula su BoundingBox, permitiendo su clasificación en el cuadrante correspondiente. Para la consulta de objetos cercanos, se genera un *envelope* de consulta alrededor del BoundingBox del objeto de interés,

Diagrama de clases

The UML class diagram illustrates the architecture of a 2D physics engine project. The classes are organized into several layers and packages:

- WebSocketConfig** (package)
- PhysicsEngine** (package)
- TargetPositionDTO** (package)
- BodyIdDTO** (package)
- SPAController** (package)

Core Classes and Relationships:

- Vector2D**: A base class for **position** and **vertices**.
- BoundingBox**: A class that inherits from **Body** and has a **center** attribute. It is associated with **BodyType** via **box** and **bodyType** attributes.
- Body**: An abstract class with **bodies** as an attribute. It is associated with **BodyType** via **bodyType** and **bodyType** attributes.
- Asteroid**, **Planet**, and **Spaceship**: Concrete classes that inherit from **Body**.
- BodyDAOImpl**: A class that inherits from **BodyDAO** and is associated with **BodyDAO** via **bodyDAO** and **bodyDAO** attributes.
- CollisionResolver**: A class that inherits from **CollisionResolver** and is associated with **CollisionResolver** via **collisionResolver** and **collisionResolver** attributes.
- DataService**: A class that inherits from **DataService** and is associated with **DataService** via **dataService** and **dataService** attributes.
- AttractionResolver**: A class that inherits from **AttractionResolver** and is associated with **AttractionResolver** via **attractionResolver** and **attractionResolver** attributes.
- QuadtreeService**: A class that inherits from **QuadtreeService** and is associated with **QuadtreeService** via **quadtreeService** and **quadtreeService** attributes.
- MovementResolver**: A class that inherits from **MovementResolver** and is associated with **MovementResolver** via **movementResolver** and **movementResolver** attributes.
- BodyService**: A class that inherits from **BodyService** and is associated with **BodyService** via **bodyService** and **bodyService** attributes.
- PhysicsService**: A class that inherits from **PhysicsService** and is associated with **PhysicsService** via **physicsService** and **physicsService** attributes.
- WebSocketController**: A class that inherits from **WebSocketController** and is associated with **WebSocketController** via **websocketController** and **websocketController** attributes.

Associations and Multiplicities:

- Vector2D** to **position** and **vertices**: Multiplicity 1 at Vector2D, 1 at position, 1 at vertices.
- BoundingBox** to **BodyType**: Multiplicity 1 at BoundingBox, 1 at box, 1 at bodyType, 1 at bodyType.
- Body** to **BodyType**: Multiplicity 1 at Body, 1 at bodies, 1 at bodyType, 1 at bodyType.
- BodyDAOImpl** to **BodyDAO**: Multiplicity 1 at BodyDAOImpl, 1 at bodyDAO, 1 at bodyDAO.
- CollisionResolver** to **CollisionResolver**: Multiplicity 1 at CollisionResolver, 1 at collisionResolver, 1 at collisionResolver.
- DataService** to **DataService**: Multiplicity 1 at DataService, 1 at dataService, 1 at dataService.
- AttractionResolver** to **AttractionResolver**: Multiplicity 1 at AttractionResolver, 1 at attractionResolver, 1 at attractionResolver.
- QuadtreeService** to **QuadtreeService**: Multiplicity 1 at QuadtreeService, 1 at quadtreeService, 1 at quadtreeService.
- MovementResolver** to **MovementResolver**: Multiplicity 1 at MovementResolver, 1 at movementResolver, 1 at movementResolver.
- BodyService** to **BodyService**: Multiplicity 1 at BodyService, 1 at bodyService, 1 at bodyService.
- PhysicsService** to **PhysicsService**: Multiplicity 1 at PhysicsService, 1 at physicsService, 1 at physicsService.
- WebSocketController** to **WebSocketController**: Multiplicity 1 at WebSocketController, 1 at websocketController, 1 at websocketController.

Generalization:

- Asteroid**, **Planet**, and **Spaceship** generalize **Body**.
- BodyDAOImpl** generalizes **BodyDAO**.
- CollisionResolver** generalizes **CollisionResolver**.
- DataService** generalizes **DataService**.
- AttractionResolver** generalizes **AttractionResolver**.
- QuadtreeService** generalizes **QuadtreeService**.
- MovementResolver** generalizes **MovementResolver**.
- BodyService** generalizes **BodyService**.
- PhysicsService** generalizes **PhysicsService**.
- WebSocketController** generalizes **WebSocketController**.

Figura C.2: Diagrama de clases simplificado

También incluimos una imagen de alta resolución del diagrama completo:

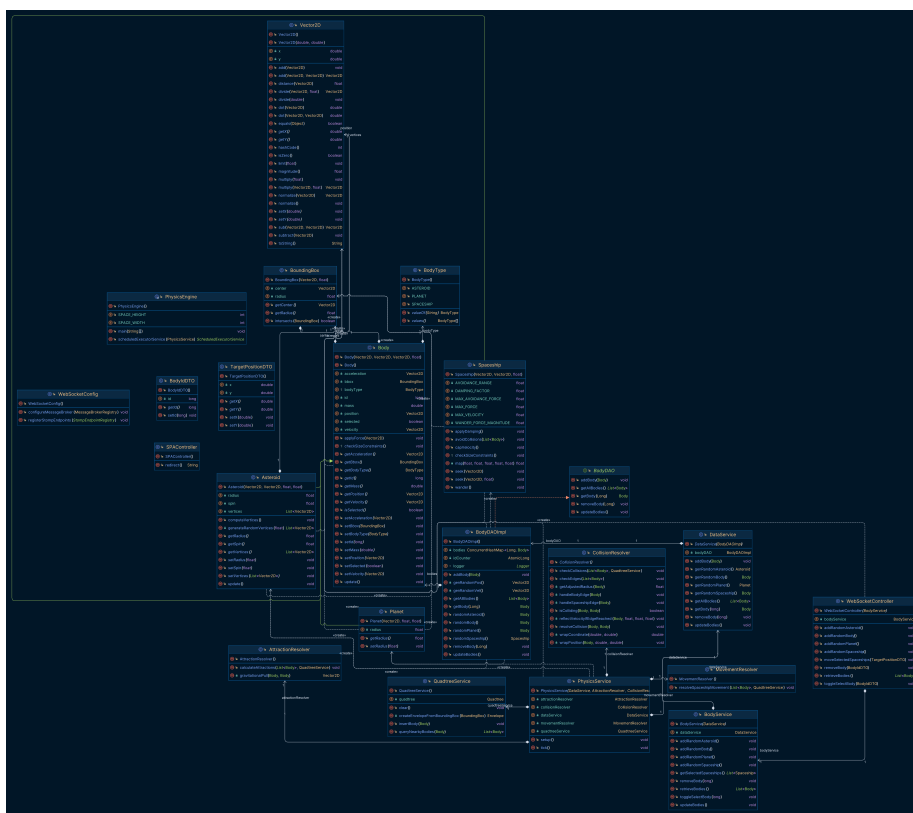


Figura C.3: Diagrama de clases completo

C.3. Diseño procedimental

El diseño procedimental del motor de físicas es un aspecto crucial que dicta cómo se estructuran y se ejecutan las diversas operaciones y procesos dentro del sistema. Esta sección detalla los procedimientos fundamentales que componen el motor de físicas, explicando cómo interactúan las entidades y cómo se simulan las leyes físicas.

Procedimientos Clave

Para el funcionamiento efectivo del motor de físicas, se han definido varios procedimientos clave que gestionan desde el movimiento de las entidades hasta la resolución de colisiones y la aplicación de fuerzas gravitacionales.

Inicialización del Sistema

El primer paso en la simulación física es la inicialización del sistema, donde se crean y configuran una serie de entidades (planetas, asteroides y naves espaciales) con sus propiedades iniciales. Este proceso también incluye la configuración del espacio de simulación y la inicialización del Quadtree para optimizar las consultas espaciales.

Simulación de Movimientos

El movimiento de cada entidad se simula mediante la actualización de sus vectores de posición y velocidad, basándose en sus aceleraciones actuales y las fuerzas aplicadas por otras entidades. Para las naves espaciales, este proceso también involucra la ejecución de comportamientos específicos como la evasión y la búsqueda.

Detección y Resolución de Colisiones

Una parte esencial del procedimiento es la detección de colisiones entre entidades. Cuando dos entidades colisionan, el sistema calcula el resultado de la colisión, ajustando sus vectores de velocidad según las leyes de conservación de la energía y el momento.

Aplicación de Fuerzas Gravitacionales

Las fuerzas gravitacionales entre las entidades se calculan y se aplican en cada tick de la simulación, afectando sus aceleraciones y, por ende, sus movimientos futuros. Este procedimiento simula la atracción mutua entre masas, permitiendo la creación de sistemas orbitales y la interacción dinámica entre objetos.

Optimización del Rendimiento con Quadtree

Para mejorar el rendimiento de la detección de colisiones y la aplicación de fuerzas, se utiliza un Quadtree para segmentar el espacio de simulación. Esto permite al sistema realizar consultas espaciales eficientes, reduciendo el número de comparaciones necesarias al determinar las interacciones entre entidades cercanas.

Ciclo de Simulación

El motor de físicas ejecuta estos procedimientos en un ciclo de simulación continuo, donde cada iteración representa un “tick.”^{en} el tiempo de simula-

ción. En cada tick, el sistema actualiza el estado de todas las entidades, aplica las fuerzas gravitacionales, detecta y resuelve colisiones, y procesa los movimientos y comportamientos específicos de las entidades. Este ciclo permite la simulación de un universo físico dinámico y interactivo.

Procedimientos de Interfaz

Además de los procesos internos, el motor de físicas proporciona interfaces para la manipulación y observación del sistema. Esto incluye funciones para agregar o eliminar entidades, modificar propiedades, y visualizar el estado del sistema. Estas interfaces facilitan la integración del motor de físicas con aplicaciones de usuario y permiten la experimentación y el análisis de diferentes configuraciones y escenarios físicos.

Este diseño procedimental asegura que el motor de físicas sea capaz de simular entornos complejos con precisión y eficiencia, manteniendo al mismo tiempo la flexibilidad necesaria para adaptarse a requisitos específicos de simulación y exploración de conceptos físicos.

C.4. Diseño arquitectónico

En este apartado, se detalla la estructura arquitectónica adoptada para el desarrollo del motor de físicas. Esta arquitectura ha sido diseñada con el objetivo de maximizar la reusabilidad del código, la claridad conceptual y la flexibilidad para futuras ampliaciones o modificaciones.

Arquitectura General

El proyecto se ha estructurado basándonos en la arquitectura de microservicios^[12], aprovechando la división en componentes con responsabilidades bien definidas para facilitar el desarrollo, la prueba y el mantenimiento del sistema. A continuación, se describen los principales componentes del sistema:

Modelo de Entidades Físicas

El núcleo del motor de físicas se basa en la definición de entidades físicas, representadas por la clase abstracta `Body` y sus derivados `Planet`, `Asteroid` y `Spaceship`. Cada una de estas entidades posee atributos como posición, velocidad, masa y un identificador único, permitiendo simular su comportamiento en el espacio de simulación.

Gestión de Colisiones

Para gestionar las interacciones entre entidades, se implementa el componente `CollisionResolver`, responsable de detectar colisiones entre entidades y de resolver sus efectos aplicando cambios en sus velocidades y posiciones según las leyes físicas.

Simulación de Fuerzas

El componente `AttractionResolver` se encarga de calcular y aplicar fuerzas gravitacionales entre las entidades, permitiendo simular atracciones o repulsiones que afectan el movimiento de las mismas.

Control de Movimientos de Aeronaves

Mediante el componente `MovementResolver`, se especializa el tratamiento de las entidades tipo `Spaceship`, aplicando lógicas de movimiento como deambulación, evasión y búsqueda, para simular comportamientos más complejos.

Optimización Espacial con Quadtree

La gestión eficiente del espacio de simulación se realiza a través del uso de un Quadtree, implementado en el servicio `QuadtreeService`. Este componente permite realizar consultas espaciales rápidas para la detección de colisiones y la búsqueda de entidades cercanas, optimizando el rendimiento del motor de físicas.

Servicio de Físicas

El `PhysicsService` actúa como orquestador de los diferentes resolvers (`AttractionResolver`, `CollisionResolver`, `MovementResolver`) y del `QuadtreeService`, coordinando la simulación de un tick y actualizando el estado del sistema.

Arquitectura de Software

Para el desarrollo del proyecto, se ha utilizado Spring Boot, facilitando la configuración y el despliegue del servicio. La estructura del proyecto se divide en paquetes bien diferenciados para modelos, servicios, controladores y configuraciones, promoviendo la separación de responsabilidades y facilitando la comprensión y mantenimiento del código.

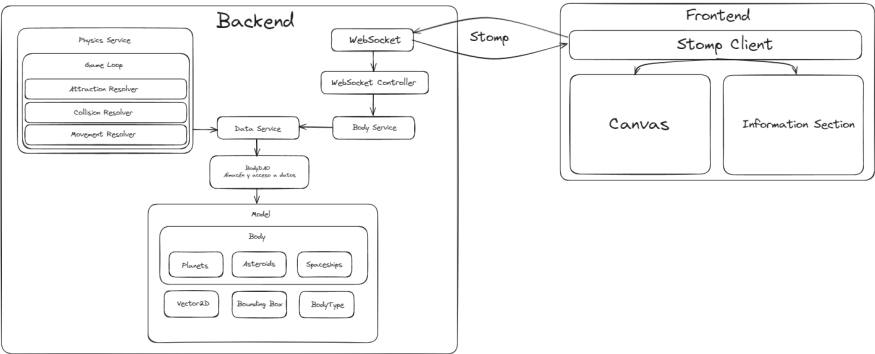


Figura C.4: Arquitectura

Apéndice D

Documentación técnica de programación

D.1. Introducción

En este apéndice se detallarán los elementos importantes para los desarrolladores, abarcando desde la organización de la estructura de directorios hasta las dependencias necesarias que deben ser instaladas en situaciones específicas, así como la funcionalidad de los diversos archivos presentes en el repositorio.

D.2. Estructura de directorios

- `src`: Este directorio contiene todo el código fuente, tanto el backend, como el frontend que se despliega junto al motor.
- `src/main/java/com/dmm/tfg`: Este directorio contiene el código fuente pertinente al motor de la aplicación.
- `src/main/java/com/dmm/tfg/config`: Este directorio contiene el código fuente pertinente a la configuración de los endpoints de websocket.
- `src/main/java/com/dmm/tfg/controller`: Este directorio contiene el código fuente pertinente a los controladores de requests procedentes del frontend.
- `src/main/java/com/dmm/tfg/engine`: Este directorio contiene el código fuente pertinente al modelo y a los algoritmos.

- `src/main/java/com/dmm/tfg/service`: Este directorio contiene el código fuente pertinente a los módulos que manejan tanto el sistema como la simulación.
- `src/main/resources`: Este directorio contiene el código pertinente a las propiedades del proyecto tanto como a la aplicación de frontend que se despliega junto al backend.
- `src/test/java/com/dmm/tfg`: Este directorio contiene todos los ficheros con los tests de la aplicación.
- `Docs`: Este directorio contiene la memoria, anexos y todos los ficheros que la componen.
- `WebApp`: Este directorio contiene el código fuente de la aplicación de frontend, a partir de la cual obtenemos la *build* que copiamos a `src/main/resources/static` y desplegamos junto al motor.
- `gradle/wrapper`: Este directorio contiene el wrapper de Gradle necesario para poder compilar y construir la aplicación sin tener que instalar Gradle.
- `.gitignore`: Fichero gitignore, que contiene la configuración de ficheros que no se subirán al repositorio al hacer los commits.
- `Dockerfile`: Fichero que contiene las instrucciones para construir la imagen de Docker que servirá para desplegar todo el aplicativo en un contenedor virtualizado.
- `LICENSE`: Fichero que contiene la licencia del proyecto.
- `README.md`: Fichero que contiene la descripción del repositorio en GitHub.
- `build.gradle`: Fichero de configuración de Gradle, que especifica las dependencias e instrucciones para compilar el proyecto.
- `build.ps1`: Script facilitado a los usuarios que quieran compilar y ejecutar la aplicación en Windows.
- `build.sh`: Script facilitado a los usuarios que quieran compilar y ejecutar la aplicación en un entorno Unix con bash.
- `gradlew`: Script del wrapper de Gradle.
- `gradlew.bat`: Script del wrapper de Gradle para Windows.
- `settings.gradle`: Configuración del proyecto de Gradle.

D.3. Compilación, instalación y ejecución del proyecto

Este documento proporciona una guía detallada para configurar y ejecutar las diferentes partes del proyecto, desde la aplicación web hasta el backend, destacando las dependencias necesarias y los pasos específicos para cada componente.

Preparación del Entorno para Docker

Para aquellos que prefieran un despliegue simplificado y contenerizado, se recomienda utilizar Docker. Este método asegura un entorno uniforme y facilita tanto la compilación como la ejecución.

Construcción de la Imagen Docker

Primero, asegúrate de tener Docker instalado en tu sistema. Posteriormente, puedes construir la imagen Docker necesaria para ejecutar la aplicación. Para ello, navega hasta el directorio que contiene el Dockerfile y ejecuta:

```
docker build -t [nombre-de-la-imagen] .
```

Ejecución del Contenedor Docker

Una vez construida la imagen, puedes iniciar el contenedor con el siguiente comando:

```
docker run -it -p [puerto]:3100 [nombre-de-la-imagen]
```

Este comando publicará la aplicación en el puerto especificado de tu máquina host, permitiendo el acceso a través del navegador.

Despliegue Tradicional

Para un despliegue más tradicional, que implica ejecutar la aplicación directamente en tu máquina host, seguir los siguientes pasos:

Requisitos

Backend: JDK 17 o superior. Asegúrate de que la variable JAVA_HOME apunta a la versión adecuada del JDK, ya que versiones superiores a la 21

pueden no ser compatibles. Esto es debido a que existe un bug con Lombok en la versión 21. Puede que cuando lo despliegues ya se haya solucionado. No es necesario instalar Gradle, la compilación se hace a través del Wrapper. Frontend: Node.js 20 LTS o superior para la aplicación web.

Instalación y Ejecución

Aplicación Web: Dentro del directorio del proyecto web, ejecuta

```
npm install
```

para instalar las dependencias y

```
npm build
```

para construir la aplicación. Luego, copia el resultado de la compilación al directorio `src/main/resources/static/` del backend. Servidor: Utiliza el comando

```
gradle build
```

para compilar el backend. Inicia la aplicación con

```
java -jar build/libs/TFG-1.0.0-RELEASE.jar
```

Compilación y despliegue independiente de la WebApp

Para compilar y lanzar la WebApp de forma independiente al backend, es crucial modificar el punto de conexión del WebSocket para asegurar la comunicación correcta entre ambos. Edita el archivo `WebApp/src/App.js` reemplazando la función `getWebSocketUrl` por:

```
function getWebSocketUrl() {  
    // Asegúrate de reemplazar 'direccion-del-backend' con la dirección actual  
    // del servidor backend y 'puerto-backend' con el puerto en el que esté corriendo.  
    return "http://direccion-del-backend:puerto-backend/physics-engine-websocket";  
}
```

Figura D.1: Función a reemplazar/modificar para determinar la conexión al backend

Después de realizar esta modificación, puedes proceder a compilar la WebApp con

```
npm build
```

Para servir la WebApp, puedes utilizar cualquier servidor HTTP de tu elección, como `serve` en Node.js, configurándolo para servir los archivos estáticos generados en el directorio de `build`.

Lanzamiento de la WebApp

Una vez compilada y configurada para apuntar al backend adecuadamente, la WebApp puede ser lanzada utilizando:

```
npm run deploy
```

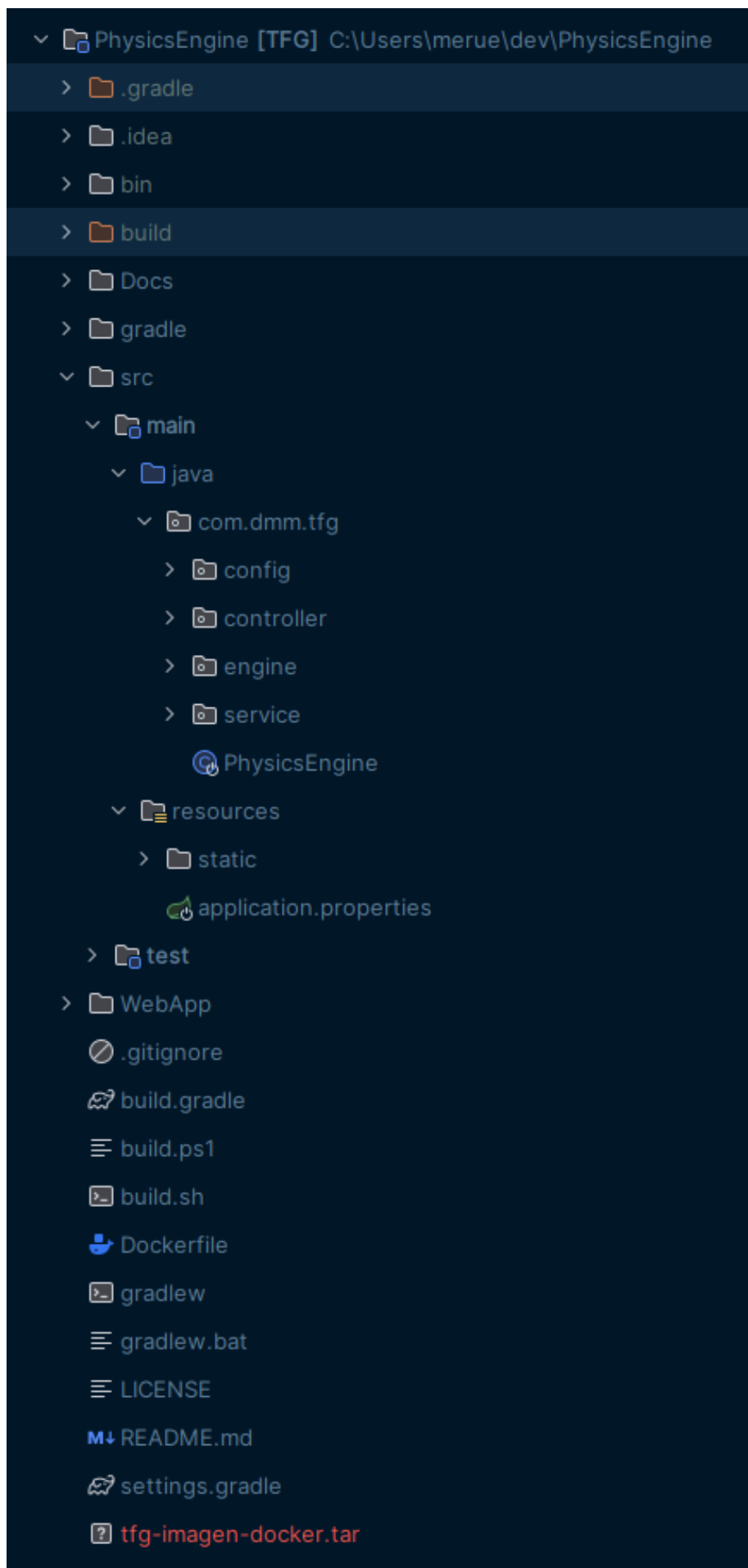
Asegúrate de abrir el puerto especificado en tu firewall si es necesario para permitir el acceso externo a la aplicación web.

D.4. Manual del programador

Desarrollo del backend

El proyecto de backend cuenta con un fichero `build.gradle` que ya está configurado para ser importado en cualquier entorno de desarrollo y poder compilar y ejecutar la aplicación. Cuenta también con tareas para tests y para generar reportes Jacoco, que serán utilizados para el control de calidad. En mi caso, he utilizado IntelliJ IDEA Ultimate para el desarrollo del proyecto. Encuentro que tiene muchas características útiles para desarrollar un proyecto de este estilo. Se puede acceder a las tareas de Gradle desde una

interfaz gráfica, genera diagramas y analiza estáticamente el código. Además, se pueden instalar plugins para cualquier otra funcionalidad necesaria.



Los algoritmos para los distintos tipos de conceptos físicos que se han aplicado han sido implementados en diferentes clases y después manejados desde un servicio principal (***PhysicsService***). De este modo, resulta más sencillo realizar modificaciones a las propiedades de los algoritmos o añadir nuevos conceptos al motor.

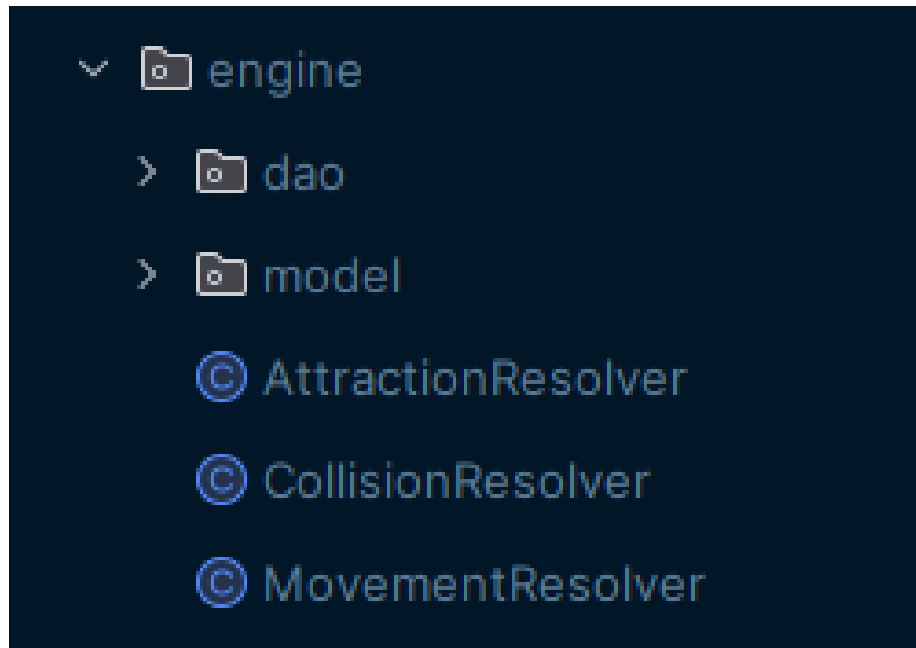


Figura D.3: Paquete *Engine*. Aquí se pueden incluir o modificar nuevos algoritmos.

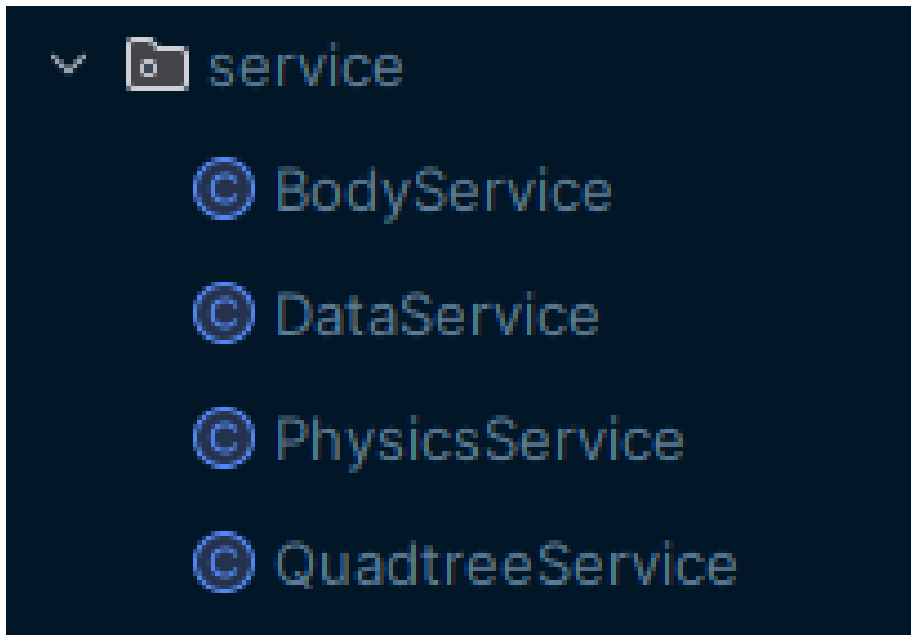


Figura D.4: Paquete *Service*. Aquí se puede modificar cómo se usan los distintos algoritmos o la configuración inicial.

```
@Service
@RequiredArgsConstructor
public class PhysicsService {

    private final DataService dataService;
    private final AttractionResolver attractionResolver;
    private final CollisionResolver collisionResolver;
    private final MovementResolver movementResolver;
    private final QuadtreeService quadtreeService;

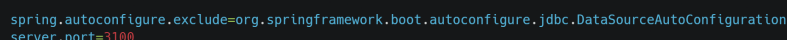
    public void setup(){
        dataService.addBody(new Planet(new Vector2D(400, 300), 100000000, 100));
        dataService.addBody(new Asteroid(new Vector2D(1,1), new Vector2D(0.5,0.5), 100000, 25));
        dataService.addBody(dataService.genRandomSpaceship());
    }

    public void tick() {
        quadtreeService.clear();
        for (Body body : dataService.getAllBodies()) {
            quadtreeService.insertBody(body);
        }

        collisionResolver.checkEdges(dataService.getAllBodies());
        attractionResolver.calculateAttractions(dataService.getAllBodies(), quadtreeService);
        movementResolver.resolveSpaceshipMovement(dataService.getAllBodies(), quadtreeService);
        collisionResolver.checkCollisions(dataService.getAllBodies(), quadtreeService);
        dataService.updateBodies();
    }
}
```

Figura D.5: Este es el servicio que se encarga de correr la simulación.

Otro factor a considerar es la configuración de Spring. En nuestro proyecto, es idéntica a la que viene por defecto con Spring Boot, pero puede que en un futuro, se decida modificar. El fichero que contiene esta configuración esta en `src/main/resources/application.properties`. Aquí, por ejemplo, se podría cambiar el puerto en el que se despliega el servidor de Tomcat que aloja a la aplicación.

A screenshot of a code editor showing two lines of configuration in a `application.properties` file. The first line is `spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration` and the second line is `server.port=3100`. The text is white on a dark background.

```
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
server.port=3100
```

Figura D.6: Aquí se pueden realizar cambios en la configuración de la aplicación.

Desarrollo del frontend

De cara al desarrollo del front, personalmente he utilizado VSCode, pero se podría utilizar cualquier editor a preferencia del usuario. Sin embargo, recomiendo VSCode ya que tiene un rico ecosistema de plugins que ha facilitado mucho el desarrollo del cliente web.

Entre los aspectos importantes a recalcar, encontramos:

- El punto de entrada principal se encuentra en el fichero `WebApp/src/App.js`. Este es el fichero que contiene la lógica de conexión al backend y que genera la estructura de componentes que, después, se renderizan en el HTML.
- Los componentes se encuentran en `WebApp/src/components`. Se pueden modificar individualmente, gracias a su diseño modular. Se puede añadir más funcionalidad añadiendo más componentes e incorporándolos al fichero `App.js`.
- No se ha utilizado ninguna herramienta externa para el CSS (Como Tailwind o Bootstrap). Se ha intentado desarrollar la aplicación de la manera más sencilla posible, para poder centrar toda la carga en el motor.

Control de calidad

Para el proyecto, se han utilizado dos herramientas para garantizar la calidad del código: Codacy y CodeClimate Quality.

Codacy ofrece una amplia gama de plantillas para el análisis de calidad, proporcionando revisiones automáticas para identificar problemas de seguridad, patrones de diseño ineficientes y errores de compilación en una variedad de lenguajes y formatos de archivo, incluidos Java, JavaScript, y archivos de configuración específicos del proyecto.

Code Climate, por su parte, se centra en aspectos cruciales para la mantenibilidad del código. Evalúa la complejidad del código, la longitud de los métodos y la claridad del código, facilitando la identificación de áreas que requieren refactorización o simplificación. Esta herramienta es especialmente útil para garantizar que el código se mantenga accesible y manejable a lo largo del tiempo.

Adicionalmente, hemos incorporado Codacy para el seguimiento de la cobertura de código, lo que nos permite asegurar que nuestros tests alcanzan una amplia gama del código fuente.

Durante el desarrollo, se hizo necesario deshabilitar ciertos patrones de análisis, especialmente aquellos relacionados con las importaciones en Java. Las políticas de importación de IntelliJ IDEA Ultimate y Codacy entraban en conflicto continuamente. Codacy consideraba que jamás se debería utilizar el asterisco en las sentencias de importación, mientras que IntelliJ consideraba propio utilizarlo si se importaban todas las clases de un paquete. He decidido desactivarlo porque considero que solo reduce la visibilidad de los verdaderos errores.

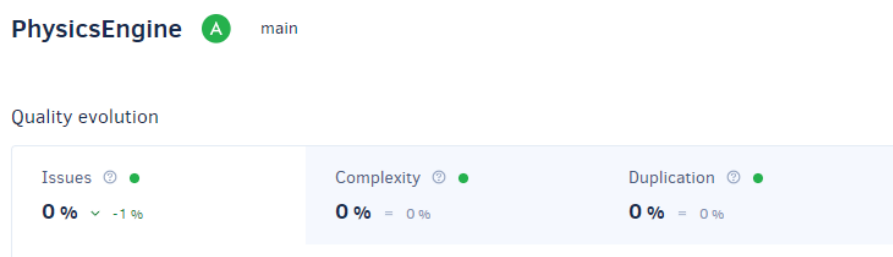


Figura D.7: Análisis final de Codacy

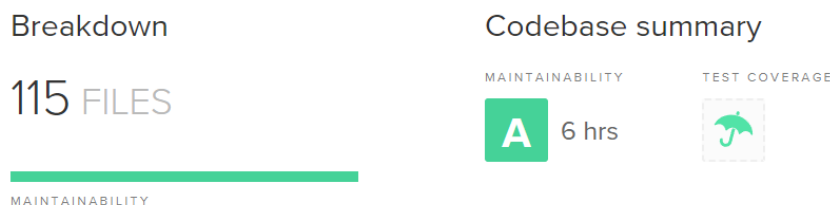


Figura D.8: Análisis final de Code Climate

D.5. Pruebas del sistema

Pruebas unitarias

Para asegurar la integridad y funcionalidad del código en nuestro proyecto, se implementaron pruebas unitarias que validan el comportamiento de los distintos componentes y métodos. Estas pruebas se organizan dentro del directorio `/src/test/java`, siguiendo la convención de nombrado `[NombrDelComponente]Test.java`, donde `[NombreDelComponente]` corresponde a la clase o funcionalidad específica bajo prueba. Actualmente, se cubre el 92 % del código, excluyendo las clases de configuración.

Durante el proceso de desarrollo, se empleó JUnit como el framework principal para la creación de estas pruebas unitarias, asignando un archivo de pruebas específico para cada clase de la aplicación. También se utilizó Mockito para la generación de mocks. En total, se generaron múltiples casos de prueba, abarcando desde la lógica de negocio hasta la interacción con el frontend, cuando corresponda. Cada archivo de test contiene casos detallados que cubren los escenarios esperados y excepcionales, acompañados de comentarios descriptivos que facilitan su comprensión y mantenimiento.

Este enfoque sistemático hacia las pruebas unitarias no solo permite validar el correcto funcionamiento de los componentes aislados del sistema, sino que también contribuye a la detección temprana de errores y a la mejora continua del código, asegurando un desarrollo robusto y confiable del proyecto.

Sin embargo, estas pruebas no son suficientes para asegurar el correcto funcionamiento de la aplicación. Dado que hay tantas partes móviles y el entorno está separado en dos partes bien diferenciadas: backend y frontend; debemos realizar una serie de pruebas de integración para asegurarnos de que no se haya producido ninguna regresión.

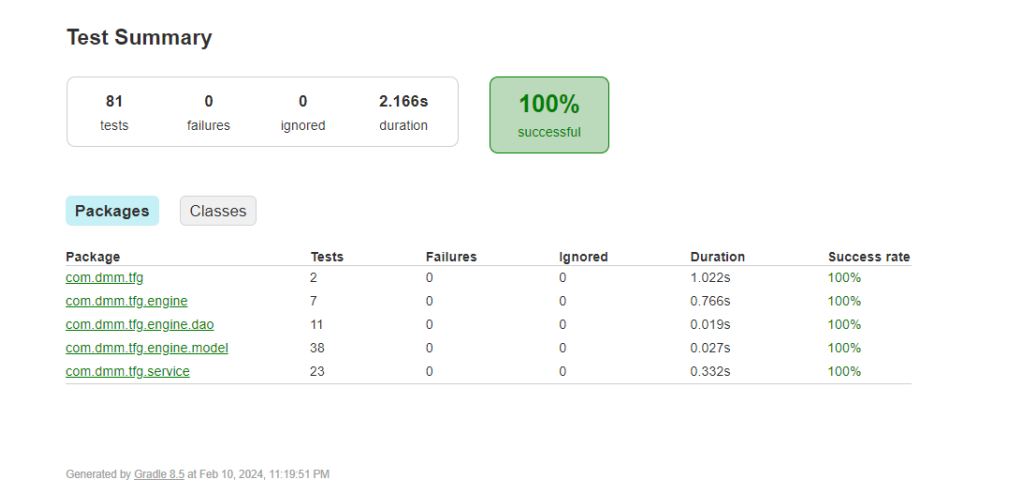


Figura D.9: Reporte de la ejecución de los tests del proyecto

Pruebas de integración

Cada vez que se modificaba un componente del sistema, si se pasaban los test unitarios, se realizaban unas pruebas de validación para comprobar tanto su correcto funcionamiento como que no se había producido ninguna regresión. Este proceso se realizaba de forma manual, testeando tanto el funcionamiento como la capacidad de compilar y lanzar tanto el backend como el cliente de frontend. Para este proceso, las pruebas se hacían con un cliente de frontend desplegado en un entorno separado al del motor, para garantizar que la integridad de la conexión no se veía afectada por los cambios.

Apéndice E

Documentación de usuario

E.1. Introducción

Este anexo proporciona una descripción detallada de los requisitos previos para la ejecución de la aplicación, así como las instrucciones para su instalación y activación. Adicionalmente, se ofrece una guía destinada a facilitar la navegación del usuario a través de las diversas interfaces de la aplicación.

E.2. Requisitos de usuarios

Los requisitos para ejecutar la aplicación son:

- Un sistema operativo compatible con Docker.
- Un navegador de internet para acceder al cliente web.
- OPCIONAL: Conexión a internet para construir la imagen, en vez de descargarla desde el repositorio.

E.3. Instalación

Para instalar y ejecutar tanto el motor como el cliente web, se deberán seguir estos pasos:

En caso de querer construir la imagen

1. Asegurarse que se cumplen todos los requisitos. Estos se encuentran disponibles en la página del repositorio.
2. Clonar el repositorio del proyecto. Esto puede hacerse con el siguiente comando:

```
git clone https://github.com/dmm1005/PhysicsEngine.git
```

3. Acceder al directorio del repositorio.
4. Una vez dentro, se deberá usar ejecutar el script correspondiente al entorno que esté utilizando el usuario:
 - `build.sh`: para entornos Linux (bash). Es posible que haya que dar permisos de ejecución (`chmod +x`) al script.
 - `build.ps1`: para entornos Windows (PowerShell).

En caso de querer descargarse la imagen de Docker

1. Acceder a este [Drive](#) y descargar la imagen del proyecto.
2. Asegure que Docker Engine está arrancado y ejecutar estos comandos:

```
docker load -i [ruta_a_la_imagen.tar]
```

```
docker run -it -P [puerto de la máquina host]:3100 tfg-latest
```

De esta manera, se guardará la imagen del proyecto en el repositorio local de la máquina host y se creará un contenedor con el motor y el cliente web publicados en el puerto especificado por el usuario.

E.4. Manual del usuario

La interfaz del cliente web ha sido diseñada para mantenerla lo más sencillo posible. Aún así, se incluye una pequeña descripción de las características menos evidentes para el usuario.

Seleccionar cuerpos

Si queremos comprobar a qué representación gráfica se está refiriendo un cuerpo de la lista, podemos hacer click en la tarjeta que contiene la información. Al hacer esto, tanto la tarjeta como la representación visual del cuerpo cambiarán de color, permitiéndonos una más sencilla localización. Esto también se puede realizar de la manera inversa; si se hace click en un cuerpo en la visualización, se marcarán tanto él, como la tarjeta.

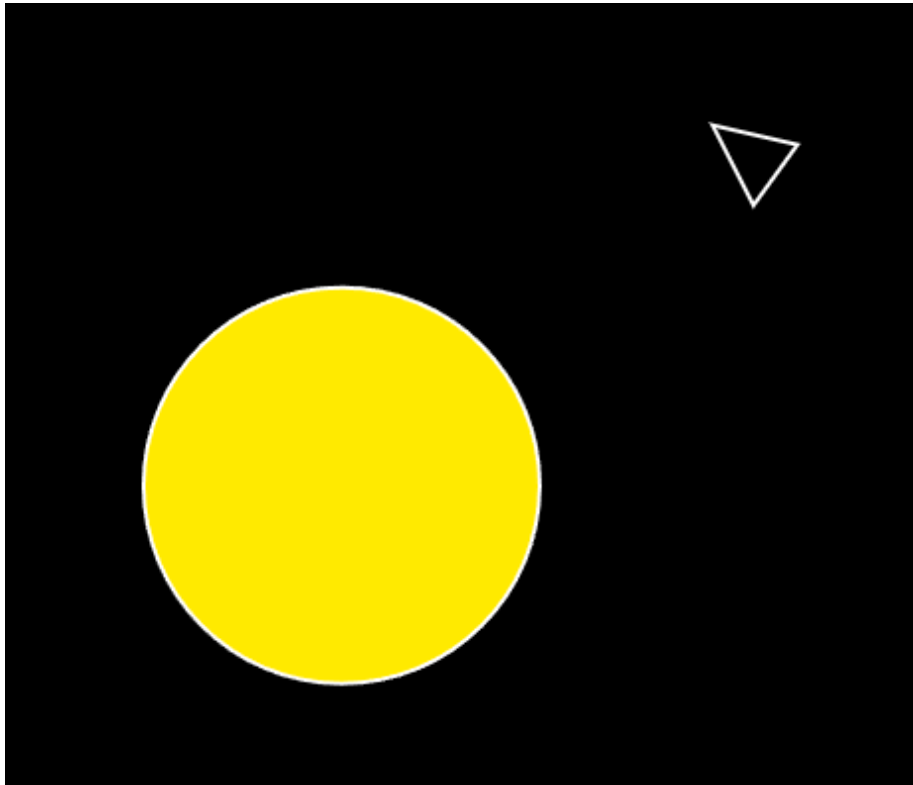


Figura E.1: Ejemplo de cuerpo seleccionado junto a uno no seleccionado



Body ID: 0 - Type: PLANET Body Velocity: <input type="button" value="Delete"/>	Body Mass: 100000000 x: 0.0003013,y: 0.0009609
Body ID: 1 - Type: ASTEROID Body Velocity: <input type="button" value="Delete"/>	Body Mass: 100000 x: 0.1986616,y: 0.4609429
Body ID: 2 - Type: SPACESHIP Body Velocity: <input type="button" value="Delete"/>	Body Mass: 1.0960525274276733 x: 0.3455189,y: -0.0481580

Figura E.2: Ejemplo de tarjeta seleccionada junto a tarjetas no seleccionadas

Establecer un punto de búsqueda para las aeronaves

Una de las características de las aeronaves es que pueden establecer un protocolo de búsqueda y dirigirse hacia un punto objetivo a gusto del usuario. Cuando el usuario hace click en cualquier parte de la representación gráfica de la simulación, se dibujará un marcador rojo en ese mismo punto. Esa será la localización objetivo para las aeronaves. Para que las aeronaves utilicen ese comportamiento, primero deberán ser seleccionadas. De esta manera, sólo las aeronaves seleccionadas pasarán a modo búsqueda, mientras que el resto permanecerá ejecutando sus rutinas habituales.

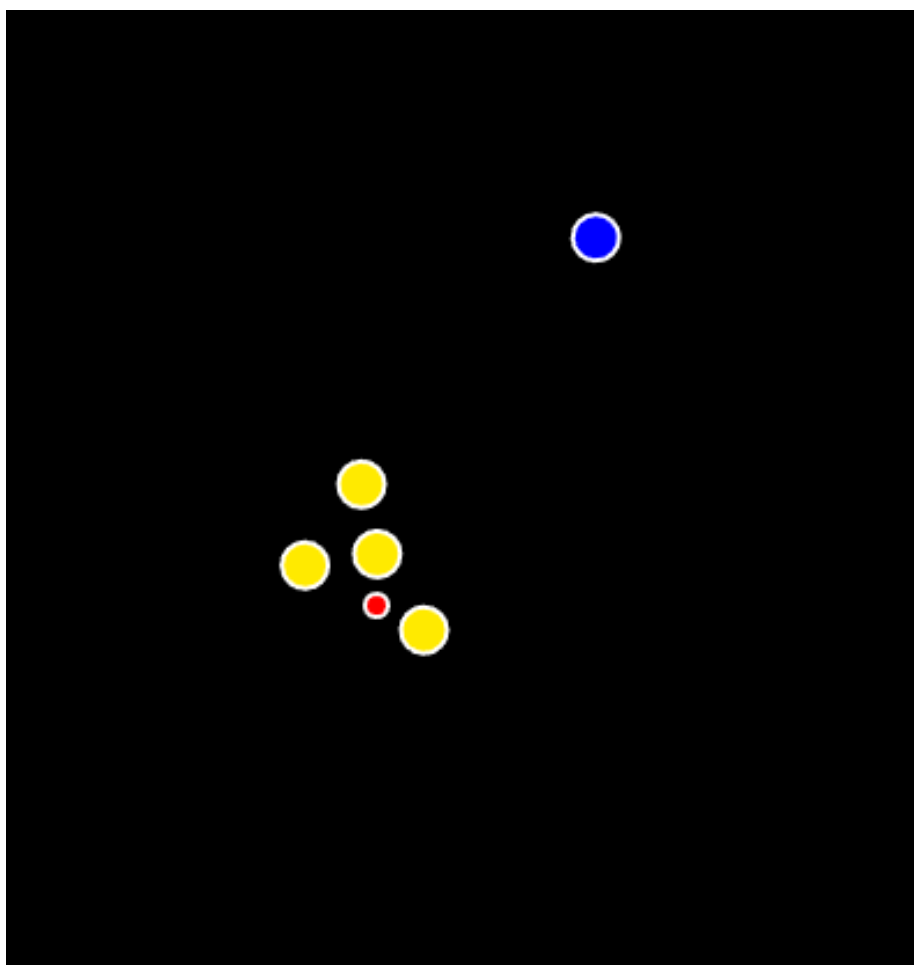


Figura E.3: Ejemplo de aeronaves dirigiéndose al punto de búsqueda junto a una aeronave siguiendo una rutina normal

Bibliografía

- [1] Apache License 2.0. <https://www.apache.org/licenses/LICENSE-2.0>.
- [2] Apache License 2.0. <https://www.apache.org/licenses/LICENSE-2.0>.
- [3] Apache License 2.0. <https://www.apache.org/licenses/LICENSE-2.0>.
- [4] EDL/BSD License. <https://opensource.org/licenses/BSD-3-Clause>.
- [5] GPL v2 with the Classpath Exception. <https://openjdk.java.net/legal/gplv2+ce.html>.
- [6] LGPL License. <https://www.gnu.org/licenses/lgpl-3.0.html>.
- [7] MIT License. <https://opensource.org/licenses/MIT>.
- [8] MIT License. <https://opensource.org/licenses/MIT>.
- [9] MIT License. <https://opensource.org/licenses/MIT>.
- [10] MIT License. <https://opensource.org/licenses/MIT>.
- [11] MIT License. <https://opensource.org/licenses/MIT>.
- [12] What are microservices? <https://microservices.io/>.
- [13] Seguridad Social. Bases y tipos de cotización 2022. <https://www.seg-social.es/wps/portal/wss/internet/Trabajadores/CotizacionRecaudacionTrabajadores/36537>, 2022. [Internet].