UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Motor de físicas basado en Java simulando un entorno natural

Presentado por Daniel Meruelo Monzón
en Universidad de Burgos — 15 de febrero de 2024
Tutor: Dr. Alejandro Merino Gómez

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

D. Alejandro Merino Gómez, profesor del departamento de nombre departamento, área de nombre área.

Expone:

Que el alumno D. Daniel Meruelo Monzón, con DNI 45573344V, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado Motor de físicas basado en Java.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 15 de febrero de 2024

Vº. Bº. del Tutor:

D. Alejandro Merino Gómez

Resumen

En este trabajo se propone el desarrollo de un sistema simulador de un entorno en dos dimensiones utilizando el lenguaje de programación Java. Con este esfuerzo, queremos explorar cuan complejo sería diseñar, desarrollar y trabajar con un sistema que trata de emular a uno natural. Aprovechándonos de las facilidades que nos permiten los frameworks actuales tanto para el despliegue como para la facilidad de desarrollo, hemos desarrollado un "motor" que calcula las interacciones entre unas entidades que pretenden emular a varios cuerpos celestes y que también hace las labores de servidor, ofreciendo una comunicación con posibles clientes a través de WebSockets, una tecnología que garantiza una comunicación más fluida que otros protocolos tales como REST o SOAP. También, se ha desarrollado un cliente web basado en React que nos permitirá tanto visualizar como interactuar con la simulación.

Descriptores

Motor de físicas, Java, Spring, WebSockets, React, simulación, espacio, interacciones... ...

Abstract

This paper proposes the development of a two-dimensional environment simulator system using the Java programming language. With this effort, we want to explore how complex it would be to design, develop and work with a system that tries to emulate a natural one. Taking advantage of the facilities that current frameworks allow us for both deployment and ease of development, we have developed an .engine"that calculates the interactions between entities that pretend to emulate various celestial bodies and that also acts as a server, offering communication with potential clients through WebSockets, a technology that guarantees a more fluid communication than other protocols such as REST or SOAP. Also, a React-based web client has been developed that will allow us to visualize and interact with the simulation.

Keywords

Physics engine, Java, Spring, WebSockets, React, simulation, space, interactions...

Índice general

Índice de figuras

Índice de tablas

Introducción

Este proyecto nace a raíz de un objetivo personal de desarrollar un motor de físicas. Hace unos años, empecé a consumir contenido, tanto vídeos en Youtube como entradas en blogs, del Profesor Daniel Shiffman, de la Universidad de Nueva York, que se centraban en simular entornos naturales y producir arte a partir de código. Este tipo de proyectos siempre me habían resultado muy interesantes y atractivos, por lo que quería aprovechar esta oportunidad que me ofrece el trabajo de fin de grado como excusa para poder desarrollar algo similar.

Habitualmente, un proyecto de este tipo se realizaría con un lenguaje más orientado hacia el rendimiento puro con el objetivo de conseguir que la simulación fuera lo más realista, generando el máximo de información posible por cada ciclo de CPU. Sin embargo, dadas las condiciones y el alcance del proyecto, he elegido Java para el núcleo del motor. Java es habitualmente considerado un lenguaje que ofrece un rendimiento inferior a otros tales como C, C++ o Rust, pero es suficientemente potente como para llevar a cabo una simulación de este calibre (y, invirtiendo más tiempo optimizando la arquitectura y operaciones, seguramente simulaciones mucho más complejas). Además, partiendo de que tengo más experiencia en Java que en el resto de lenguajes mencionados, esto supone un rendimiento mayor en ***tiempo de desarrollo***.

Lo que se propone en este trabajo es el desarrollo de un motor de físicas, un núcleo que gestione las entidades y realice todas las operaciones necesarias para la simulación que, además, sea agnóstico de la implementación de la visualización o de cómo se quiera utilizar la información que puede generar.

Dadas las restricciones temporales que tengo para realizar el desarrollo, he optado por una selección más limitada de entidades (que llamaremos

cuerpos) que maneja el sistema. En nuestro sistema existen tres tipos de cuerpos:

- Planetas: Cuerpos inicialmente estáticos (velocidad = 0) que están dotados de una gran masa. Pueden ser más o menos densos.
- Asteroides: Cuerpos dotados de una velocidad y aceleración inicial aleatoria que están dotados de una masa inferior a la de los planetas. Se generan aleatoriamente una serie de "vértices.ª la hora de crear el cuerpo que suponen una figura convexa irregular.
- Aeronaves: Agentes autónomos que deambulan por el espacio, trazando trayectorias aleatorias e intentando esquivar al resto de cuerpos con los que se crucen. Pueden ser seleccionadas y dirigidas hacia un punto a voluntad del usuario. Este comportamiento (**SEEK**) sobrescribe al resto de rutinas mientras este activo.

El sistema se encargará de almacenar los cuerpos y las características de estos en memoria y realizar los cálculos que modificaran la velocidad de los mismos. Condiciones que se tendrán en cuenta:

- Fuerzas gravitacionales atractivas: cada cuerpo genera un campo gravitatorio cuya fuerza dependerá de las masas de los cuerpos y su distancia.
$$F = G\frac{m_1 m_2}{r^2}$$
- Colisiones: Se implementa un sencillo algoritmo que detecta colisiones a través de *"bounding boxes"*[1]. Si se detecta una colisión, se resuelve utilizando una respuesta de colisión basada en impulsos.
- En el caso de las aeronaves, las posibles interacciones que puedan tener con el resto de cuerpos.

También se ha desarrollado un cliente web basado en React para poder visualizar el estado de la simulación. Con el fin de tener una experiencia cercana a lo que está realizando el motor del servidor, el frontend se compone de una sección donde se renderiza la visualización y otra donde se muestra la información de cada cuerpo y con la que se pueden gestionar estos mismos.

[1] En nuestro caso, circunferencias que rodean a cuerpos con unas figuras más complejas que incrementarían la dificultad de detectar las colisiones.

1.1. Estructura de la memoria

La estructura de esta memoria es la siguiente:

- **Introducción:** contiene una descripción del proyecto, además de una explicación de la estructura de la memoria y de los materiales complementarios.
- **Objetivos del proyecto:** listado de los objetivos, tanto generales como específicos.
- **Conceptos teóricos:** explicación de los principales conceptos teóricos necesarios para la comprensión del proyecto
- **Técnicas y herramientas:** descripción de las técnicas, metodologías y herramientas utilizadas, indicando por qué han sido elegidas frente a sus alternativas.
- **Aspectos relevantes del desarrollo del proyecto:** explicación de las diferentes decisiones tomadas durante el transcurso del proyecto, además de otras cuestiones que se consideren importantes.
- **Trabajos relacionados:**
- **Conclusiones y líneas de trabajo futuras:** observaciones obtenidas después de haber completado el trabajo, y áreas en las que se puede profundizar o mejorar en caso de continuar el trabajo en el proyecto.

Además de la memoria, se cuenta con los siguientes apéndices:

Apéndice A - Plan de proyecto software: contiene tanto la planificación temporal del proyecto como los estudios de viabilidad realizados.

Apéndice B - Especificación de requisitos: lista los objetivos, requisitos y casos de uso del trabajo.

Apéndice C - Especificación de diseño: cubre las decisiones tomadas a la hora de diseñar los datos y procedimientos del sistema, además de una explicación detallada de su situación final.

Apéndice D - Documentación técnica de programación: incluye todos los aspectos que se consideren relevantes para los programadores, desde la estructura de directorios o las instalaciones necesarias hasta las características especiales de los ficheros fuente.

Apéndice E - Documentación de usuario: contiene un conjunto de explicaciones orientadas a los usuarios finales para que sean capaces de utilizar la aplicación correctamente y sin problemas.

1.2. Materiales adicionales

Junto con estos documentos de memoria, se ha provisto un repositorio que alberga todo el código y los manuales necesarios para poder compilar y arrancar el proyecto.

Dicho repositorio también se encuentra en el siguiente enlace: https://github.com/dmm1005/PhysicsEngine

Objetivos del proyecto

Este proyecto cuenta con varios objetivos:

2.1. 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.

2.2. Objetivos Específicos:

1. Aprender a desarrollar y desplegar un proyecto con Spring Boot, pudiendo aplicar las técnicas y procesos factibles al desarrollo del proyecto.
2. Diseñar e implementar un motor que sea agnóstico de como se quiera representar la información visualmente. El motor cuenta de un modelo y de unos procesos que se podrían extender en el caso de querer añadir funcionalidad extra al sistema en futuras versiones, pero estas están completamente desacopladas de la parte externa que se encargará de tratar esta información.
3. Se debe ofrecer una interfaz con la que un cliente pueda interactuar con el motor.

4. Ofrecer una forma de visualizar el estado de la simulación, representando los cuerpos que existan en ella y su estado actual por cada iteración[2] de la simulación (tiempo real[3]).

5. Proporcionar una interfaz, a modo de demostración, que ofrezca una visualización de la simulación y elementos que permitan interactuar con el sistema.

[2]Con iteración, nos referimos a los "ticks" de la simulación; es decir, el número de veces por segundo que el motor cálcula y actualiza el estado de la simulación.

[3]Entendiéndose tiempo real como el hecho de que se pide al motor la información más actualizada que pueda ofrecer. Existe una latencia entre el cliente y el servidor que impide que sea tiempo real.

Conceptos teóricos

Este capítulo introduce los conceptos teóricos fundamentales para la comprensión del proyecto. Se abordan tanto los principios físicos como las tecnologías de software utilizadas, con énfasis en cómo estos conceptos se interrelacionan para construir un entorno de simulación dinámica y autónoma.

3.1. Motor de físicas

Conceptos básicos

Un motor de físicas es un componente de software esencial en el desarrollo de videojuegos, simulaciones de realidad virtual y sistemas de entrenamiento virtual. Su propósito es simular las leyes físicas del mundo real en un entorno digital, permitiendo que objetos virtuales interactúen entre sí y reaccionen a fuerzas externas de manera realista. Los elementos clave incluyen:

- **Gravedad:** Simula la atracción gravitatoria que afecta a todos los objetos. En el motor de físicas, se puede ajustar la intensidad y dirección de la gravedad para replicar diferentes entornos, desde la Tierra hasta entornos sin gravedad en el espacio exterior.

- **Colisiones:** Detecta y maneja las interacciones entre objetos. Cuando dos objetos entran en contacto, el motor calcula la respuesta física, como rebotes, basándose en propiedades como la masa, velocidad, y coeficiente de restitución. Este proceso es crucial para crear interacciones convincentes entre los objetos de la simulación.

- **Fuerzas y momentos:** Permite la aplicación de fuerzas externas e internas a los objetos, como empujes, arrastres, y torsiones[4]. Estas fuerzas alteran la velocidad y la orientación de los objetos, permitiendo simular acciones como el vuelo de una aeronave o el movimiento de un vehículo.

La simulación en un motor de físicas se realiza mediante la actualización continua del estado de cada objeto en función del tiempo. Este proceso, conocido como integración temporal, calcula la posición, velocidad, y orientación de los objetos en cada cuadro de la simulación, basándose en las fuerzas que actúan sobre ellos y sus interacciones mutuas.

Además, los motores de físicas modernos ofrecen optimizaciones como el **culling** de colisiones y la utilización de estructuras de datos espaciales, como los árboles cuaternarios o los octrees, para mejorar el rendimiento al simular un gran número de objetos. Esto es especialmente importante en entornos complejos y detallados, donde se requiere un alto grado de precisión y realismo.

Conceptos físicos

Fuerzas y vectores

Las fuerzas en el universo de la simulación se modelan como vectores que tienen tanto magnitud como dirección, influyendo directamente en el estado de movimiento de los cuerpos. Por ejemplo, la aplicación de una fuerza sobre un objeto puede ser representada matemáticamente tal como se muestra en la equación 3.1:

$$\vec{F} = m \cdot \vec{a} \tag{3.1}$$

donde F es el vector fuerza aplicado sobre el cuerpo, m es la masa del cuerpo, y a es la aceleración resultante. Este principio se utiliza para simular todo, desde el empuje de un motor de cohete hasta el efecto de un golpe.

En el proyecto, utilizamos vectores para representar no solo las fuerzas sino también las velocidades y las posiciones de los objetos, facilitando el cálculo de las interacciones físicas y el movimiento. Por ejemplo, la posición $\vec{p}$ de un objeto en cualquier instante se actualiza basándose en su velocidad $\vec{v}$ como:

[4]En nuestro motor, las torsiones no han sido implementadas. No creo que aporten nada a una simulación tan sencilla como esta.

$$\vec{p}_{nuevo} = \vec{p} + \vec{v} \cdot \Delta t \tag{3.2}$$

donde Δt es el paso de tiempo entre cada actualización de la simulación. En nuestro caso, se omite t porque manejamos la integración de otra manera. Tenemos un servicio ejecutor que actualiza la simulación 150 veces por segundo ($150Hz$).

Atracción gravitacional

La simulación de la atracción gravitacional se basa en la ley de gravitación universal de Newton, que se expresa como:

$$F = G \cdot \frac{m_1 \cdot m_2}{r^2} \tag{3.3}$$

donde F es la magnitud de la fuerza gravitatoria entre dos cuerpos, G es la constante gravitacional, m_1 y m_2 son las masas de los dos cuerpos, y r es la distancia entre los centros de masa de los cuerpos.

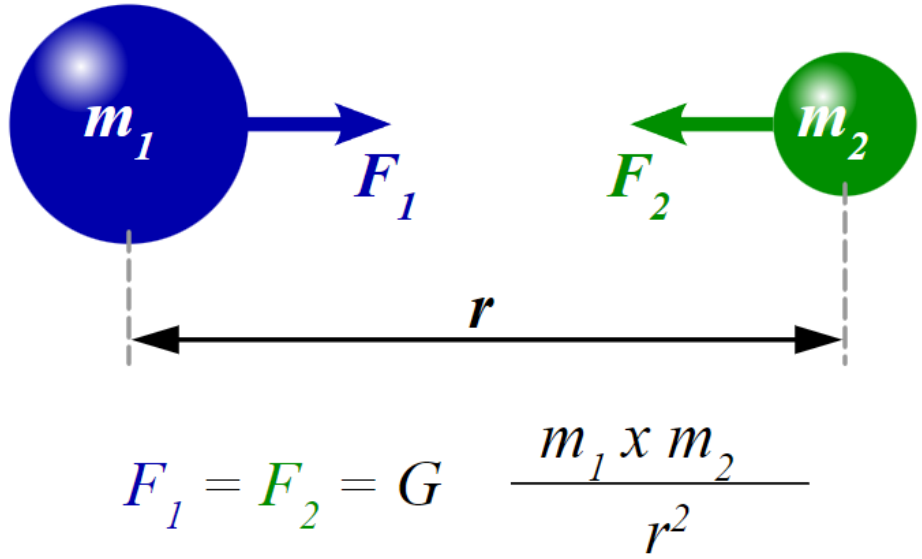

Figura 3.1: Diagrama de la Ley de Gravitación Universal. Wikipedia.

En nuestro motor de físicas, cada objeto en la simulación experimenta una fuerza gravitatoria hacia cada otro objeto, lo que resulta en trayectorias realistas de cuerpos en el espacio, como planetas orbitando estrellas o la caída de objetos hacia la tierra.

Colisiones

El manejo de colisiones es fundamental para simular interacciones realistas entre objetos. Primero determinamos si dos cuerpos ocupan el mismo espacio en un momento dado mediante la detección de colisiones, que implementamos a través de pruebas de intersección de bounding boxes o esferas de colisión.

Una vez detectada una colisión, aplicamos principios de conservación del momento y energía para calcular las velocidades resultantes. Si m_1 y m_2 son las masas de los cuerpos colisionantes y $\vec{v}_1$ y $\vec{v}_2$ sus velocidades antes de la colisión, las nuevas velocidades $\vec{v}_1'$ y $\vec{v}_2'$ después de una colisión elástica se determinan utilizando las ecuaciones de conservación del momento y energía cinética.

Por simplificación, en colisiones elásticas donde los objetos rebotan sin perder energía, las velocidades post-colisión se pueden calcular directamente a partir de las masas y las velocidades pre-colisión, ajustando por el coeficiente de restitución que determina la 'elasticidad' de la colisión.

El tratamiento de colisiones en nuestro proyecto no solo considera el impacto directo y la respuesta inmediata sino también cómo estas interacciones afectan a la trayectoria y orientación futuras de los objetos, proporcionando así una simulación dinámica y continua de interacciones físicas.

El manejo de colisiones en nuestro motor de físicas se realiza en varios pasos, comenzando con la detección del vector normal entre dos cuerpos y finalizando con la aplicación de un vector de impulso que ajusta sus velocidades post-colisión para simular un rebote realista.

Paso 1: Cálculo del vector normal y la velocidad relativa El primer paso involucra el cálculo del vector normal entre los dos cuerpos colisionantes, que apunta desde el cuerpo 1 al cuerpo 2, y se normaliza para tener una magnitud unitaria. Esto se logra mediante:

$$\vec{n} = \frac{\vec{p}_2 - \vec{p}_1}{||\vec{p}_2 - \vec{p}_1||} \tag{3.4}$$

donde $\vec{p}_1$ y $\vec{p}_2$ son las posiciones de los cuerpos 1 y 2, respectivamente. Luego, calculamos la velocidad relativa como:

$$\vec{v}_{rel} = \vec{v}_2 - \vec{v}_1 \tag{3.5}$$

donde $\vec{v}_1$ y $\vec{v}_2$ son las velocidades de los cuerpos 1 y 2.

Paso 2: Velocidad relativa a lo largo del normal Determinamos la componente de la velocidad relativa en la dirección del vector normal:

$$v_{rel\_n} = \vec{v}_{rel} \cdot \vec{n} \tag{3.6}$$

Si $v_{rel\_n} > 0$, significa que los cuerpos se están separando, y no se realiza ningún cálculo adicional.

Paso 3: Cálculo del impulso El impulso se calcula con base en el coeficiente de restitución, que determina la 'elasticidad' de la colisión, y la velocidad relativa a lo largo del normal:

$$j = \frac{-(1+e) \cdot v_{rel\_n}}{\frac{1}{m_1} + \frac{1}{m_2}} \tag{3.7}$$

donde e es el coeficiente de restitución (en este caso, 0.8), y m_1 y m_2 son las masas de los cuerpos 1 y 2.

Paso 4: Aplicación del vector de impulso Finalmente, aplicamos el impulso calculado a los cuerpos para modificar sus velocidades post-colisión:

$$\vec{v}_1' = \vec{v}_1 - \frac{j}{m_1} \cdot \vec{n} \tag{3.8}$$

$$\vec{v}_2' = \vec{v}_2 + \frac{j}{m_2} \cdot \vec{n} \tag{3.9}$$

donde $\vec{v}_1'$ y $\vec{v}_2'$ son las nuevas velocidades de los cuerpos 1 y 2 tras la colisión.

Esta metodología asegura que las colisiones en la simulación no solo sean realistas en términos de conservación de momento y energía, sino que también permitan ajustar el efecto de rebote mediante el coeficiente de restitución.

Agentes autónomos (Aeronaves)

Los agentes autónomos en nuestro proyecto, **aeronaves**, están diseñados para simular comportamientos inteligentes mediante algoritmos que les permiten interactuar con el entorno de manera dinámica. Estos comportamientos incluyen wandering (deambulación), evasión, y búsqueda, cada uno con su propia implementación y propósito específico dentro de la simulación.

Wandering (deambulación)

El wandering es un comportamiento que imita el movimiento aleatorio o exploratorio sin un destino fijo. Para implementarlo, se utiliza un vector aleatorio que se añade a la dirección actual del agente para alterar su trayectoria de forma gradual, evitando cambios bruscos que resultarían en movimientos no naturales. Matemáticamente, este vector se recalcula en cada tick del simulador, siguiendo la fórmula:

$$\vec{v}_{deambulacion} = \vec{v}_{actual} + \vec{v}_{aleatoria} \tag{3.10}$$

donde $\vec{v}_{deambulacion}$ es el nuevo vector de velocidad después de aplicar wandering, $\vec{v}_{actual}$ es el vector de velocidad actual, $\vec{aleatoria}$ es un vector unitario con dirección aleatoria.

Evasión

La evasión permite a las aeronaves detectar objetos cercanos potencialmente peligrosos y maniobrar para evitar colisiones. Este comportamiento se basa en la predicción de la trayectoria de los objetos cercanos y la generación de un vector de escape. Si la distancia prevista entre el agente y cualquier objeto es menor que un umbral definido, el agente calcula un vector de evasión como:

$$\vec{v}_{evasion} = \vec{v}_{actual} - \vec{v}_{peligro} \tag{3.11}$$

$\vec{v}_{peligro}$ es el vector de velocidad del objeto que se aproxima. Este vector de evasión se suma al vector de movimiento actual para ajustar la trayectoria del agente.

Búsqueda

Búsqueda es un comportamiento dirigido que permite a las aeronaves dirigirse hacia un objetivo específico. Se implementa calculando el vector directriz hacia el objetivo y ajustando la velocidad del agente para minimizar la distancia al objetivo con cada actualización. La fórmula para calcular el vector de búsqueda es:

$$\vec{v}_{búsqueda} = normalizar(\vec{objetivo} - \vec{posicion}) \cdot v_{max} \tag{3.12}$$

donde $\vec{objetivo}$ es la posición del objetivo, $\vec{posicion}$ es la posición actual del agente, y v_{max} es la velocidad máxima del agente. Este vector de búsqueda reemplaza la velocidad actual del agente, orientándolo directamente hacia el objetivo.

Estos comportamientos no solo dotan a las aeronaves de una apariencia de autonomía y propósito, sino que también permiten interacciones complejas dentro de la simulación, como la navegación en entornos con obstáculos, la persecución de objetivos o la evasión de amenazas, proporcionando una experiencia dinámica y realista.

Spring y SpringBoot

Spring Framework es una plataforma integral para el desarrollo de aplicaciones Java con énfasis en la flexibilidad y la inversión de control a través de inyección de dependencias. Facilita el desarrollo de aplicaciones robustas, testables y mantenibles al proporcionar una arquitectura de aplicación cohesiva.

Spring Boot, por su parte, es una herramienta de Spring que ofrece una forma rápida y fácil de configurar y ejecutar aplicaciones basadas en Spring, eliminando la necesidad de configuraciones XML extensas. Ofrece un conjunto de herramientas y convenciones predefinidas que permiten levantar un proyecto rápidamente, enfocándose en la simplicidad y el rápido desarrollo. Con Spring Boot, es posible crear microservicios, aplicaciones web, y más, con mínima configuración.

En nuestro proyecto, utilizamos Spring Boot por su capacidad para automatizar la configuración de proyectos basados en Spring, lo que nos permite enfocarnos en la lógica del negocio en lugar de la configuración del entorno de desarrollo. Spring Boot facilita la integración de WebSockets, seguridad, acceso a datos, y otras funcionalidades críticas para la aplicación, proporcionando un robusto soporte para la construcción de nuestro simulador físico.

El uso de Spring Boot ha permitido simplificar la configuración de los componentes de la aplicación, incluyendo el servidor de aplicaciones y la configuración de WebSockets, lo que resulta en un despliegue rápido y eficiente de la aplicación. La anotación y el escaneo de componentes de Spring Boot también han simplificado significativamente el proceso de inyección de dependencias, permitiéndonos desarrollar un código más limpio y modular.

WebSockets

WebSockets es un protocolo avanzado que facilita la comunicación bidireccional y en tiempo real entre clientes y servidores a través de una única conexión persistente. Esta tecnología es fundamental en aplicaciones web que requieren interacciones en tiempo real, como juegos en línea, chat en vivo, y, en nuestro caso, simulaciones físicas interactivas.

La implementación de WebSockets en nuestro proyecto permite que los estados de la simulación se sincronicen en tiempo real entre el servidor y los clientes. Cuando ocurre un evento en el servidor, como una actualización de estado debido a la simulación de fuerzas físicas o colisiones, este cambio se envía inmediatamente a todos los clientes conectados, asegurando que todos los usuarios vean una representación consistente y actualizada de la simulación.

Para integrar WebSockets en nuestra aplicación, utilizamos el soporte integrado en Spring Boot, específicamente el módulo **'spring-boot-starter-websocket'** Esto nos permitió definir puntos de conexión y manipuladores de mensajes con anotaciones simples, facilitando la implementación de la lógica de comunicación en tiempo real sin tener que lidiar con los detalles de bajo nivel del protocolo WebSocket.

Además, hemos implementado mecanismos de control de flujo y gestión de sesiones para manejar múltiples conexiones de clientes de manera eficiente, permitiendo que nuestra simulación escale y sea accesible por numerosos usuarios simultáneamente. Esta capacidad de comunicación en tiempo real ha sido crucial para proporcionar una experiencia de usuario dinámica y interactiva, permitiendo a los usuarios observar y manipular los elementos de la simulación en vivo.

3.2. Cliente

La interacción con la simulación de físicas se realiza a través de un cliente web. Este enfoque permite a los usuarios acceder y manipular la simulación desde cualquier dispositivo con un navegador web, sin necesidad de instalar software adicional. El cliente web ofrece una plataforma accesible y versátil para visualizar y experimentar con los modelos físicos en tiempo real.

Tipos de clientes. Clientes Web

El cliente web es diseñado para ser universalmente accesible, aprovechando las tecnologías web estándar para garantizar compatibilidad y rendimiento. Se basa en HTML5, CSS3 y JavaScript, permitiendo una rica interacción con la simulación a través de interfaces gráficas intuitivas. Este tipo de cliente hace posible que usuarios con distintos niveles de habilidad puedan explorar conceptos físicos complejos de manera sencilla y directa.

React

Para la implementación del cliente web, se ha elegido React, una biblioteca de JavaScript desarrollada por Facebook. React destaca por su eficiencia en la actualización y renderizado de componentes, lo cual es fundamental para representar los estados dinámicos de la simulación de físicas. Utilizando el modelo de componentes de React, la interfaz del usuario se organiza en elementos reutilizables y autónomos que responden de manera fluida a los cambios en los datos de la simulación, ofreciendo una experiencia de usuario interactiva y responsive.

React también facilita el manejo del estado de la aplicación y la comunicación entre componentes, lo cual es crucial para sincronizar los datos de la simulación con la representación visual en el cliente. A través de WebSockets, React recibe actualizaciones en tiempo real del servidor, lo que permite a los usuarios observar y manipular la simulación de físicas con una retroalimentación instantánea.

Adquirir los datos

La adquisición de datos desde el servidor se realiza mediante una conexión WebSocket, estableciendo un canal de comunicación bidireccional y en tiempo real entre el cliente y el servidor. Esta tecnología es esencial para transmitir eficientemente los estados actualizados de la simulación, como posiciones, velocidades, y otros parámetros relevantes de los objetos simulados.

Representar los datos

Una vez recibidos, los datos se utilizan en dos secciones distintas: el canvas, dónde se realiza la visualización gráfica de la simulación; y una sección donde se presenta la información de cada cuerpo en una tarjeta diferente. Las tarjetas son seleccionables y, una vez seleccionada, se resaltará

el cuerpo asociado con la tarjeta en la simulación. Así mismo, pinchar en un cuerpo de la simulación, resaltará qué tarjeta es la suya.

Técnicas y herramientas

En el siguiente capítulo, se exponen las distintas metodologías aplicadas durante el desarrollo del proyecto, así como todas las herramientas utilizadas. Además, se discuten las diversas opciones que se contemplaron, explicando los motivos por los cuales prevaleció está opción sobre las otras.

4.1. Metodologías de trabajo

Metodología ágil: Scrum

En el desarrollo de este proyecto, se ha elegido adoptar metodologías ágiles, con un enfoque particular en Scrum[18], frente a metodologías tradicionales como Waterfall.

Scrum destaca por su naturaleza iterativa e incremental, garantizando que al final de cada iteración, o Sprint, se entregue una versión funcional del producto, enriquecida con más funcionalidades que la versión anterior. Esto se logra mediante la organización de los requisitos en tareas dentro del Product Backlog, priorizándolas y seleccionándolas para el Sprint Backlog al comienzo de cada Sprint. Los roles clave en Scrum incluyen el Product Owner, quien define los requisitos y el backlog en comunicación con el cliente, y el Scrum Master, que facilita el seguimiento del modelo Scrum, asegurando la solución de impedimentos que el equipo de desarrollo pueda encontrar. Además, se llevan a cabo diferentes tipos de reuniones para promover la eficiencia y la transparencia, incluyendo la planificación de Sprints, seguimiento diario y la revisión al final de cada Sprint.

Para este proyecto, se implementaron Sprints de 1 semana con ciertas adaptaciones:

- Durante los primeros sprints, cuando se analizó el problema y se estableció una arquitectura a implementar, no se generó ningún producto funcional.
- Dado que estaba desarrollando el proyecto solo, no se estableció una división de roles estricta; asumí las responsabilidades de todos los roles, con el tutor interviniendo ocasionalmente como Product Owner.
- No se realizaron reuniones diarias, ya que sería muy difícil coordinar los horarios entre el alumno y el tutor para poder llevarlas a cabo. Además, no creo que fueran de mucha utilidad, sabiendo que el proyecto estaba siendo desarrollado por una sola persona.
- Se inició un seguimiento de las tareas en ZenHub, pero se abandonó al poco de comenzar el proyecto. Las tareas cambiaban a medida que se avanzaba y se descubrían nuevos errores de diseño en el producto.

La elección de Scrum sobre metodologías tradicionales se debe a su flexibilidad, la rápida entrega de un producto funcional, y la capacidad de adaptar y expandir el proyecto de manera ágil, aspectos cruciales para el éxito del proyecto.

Gestión del código: Git y GitHub

Para la gestión del código fuente y la colaboración en el proyecto, se evaluaron diversas plataformas de control de versiones y repositorios en línea, incluyendo opciones como GitLab, Mercurial, SVN, y Bitbucket. Sin embargo, se seleccionó **Git**[6] junto con **GitHub**[8] por varias razones fundamentales.

Git es ampliamente reconocido por su eficiencia en la gestión de versiones de código, permitiendo un trabajo en equipo fluido y un control detallado sobre las revisiones del código. Su sistema de ramificación facilita la experimentación y el desarrollo paralelo de características sin afectar el código base principal.

GitHub, por su parte, se distingue como la plataforma líder para alojamiento de repositorios Git, promoviendo la colaboración abierta. Su popularidad y la familiaridad previa que tenía con la plataforma durante mi formación académica y profesional simplifican la curva de aprendizaje y la integración en el flujo de trabajo del proyecto.

La decisión de utilizar GitHub también se basa en su robusto ecosistema de integraciones y herramientas de terceros, lo que amplía significativamente

las capacidades de gestión de proyectos y seguimiento de problemas. Además, la naturaleza pública preferida del repositorio del proyecto hizo que GitHub fuera especialmente atractivo, proporcionando visibilidad y facilitando una futura apertura a la colaboración abierta con la comunidad.

Además, GitHub nos facilitaba a mi tutor y a mí compartir opiniones sobre el código sin tener que estar empaquetándolo y mandándolo por correo constantemente.

Calidad del código:

Para garantizar la calidad del código desarrollado, se utilizan diversas herramientas integradas con GitHub que revisan los archivos en los repositorios designados, identificando una serie de errores o prácticas no recomendadas. Cuanto menor sea la incidencia de estos errores, más legible, mantenible y funcional será el código, reflejando así una calidad superior.

Entre las herramientas disponibles, se han seleccionado **Codacy**[5] y **CodeClimate Quality**[4] para esta tarea. Codacy sobresale por su capacidad para identificar un espectro más amplio de problemas y por su compatibilidad con una diversidad de tipos de archivos, incluyendo CSS. Sin embargo, ciertos aspectos como la complejidad cognitiva, que refiere a la dificultad para entender el código, son evaluados exclusivamente por CodeClimate. Esta complementariedad en el análisis de ambas herramientas fue determinante para su elección, asegurando una evaluación integral de la calidad del código.

4.2. Herramientas

A continuación se detallan las herramientas que hemos utilizado durante el desarrollo del proyecto:

Lenguaje de programación para el backend: Java

Uno de los objetivos a nivel personal del proyecto era permitirme seguir trabajando y formándome en Java, aplicando lo aprendido en la universidad y ampliando conocimientos al usar un framework de desarrollo como Spring. Además, muchas de las características que ofrece Spring se adecúan perfectamente a los objetivos del proyecto. Las librerías a nivel de backend utilizadas fueron:

- **Spring Boot Starter Web:**[26] Proporciona todas las dependencias necesarias para construir aplicaciones web, incluyendo RESTful aplicaciones usando Spring MVC. Facilita la creación de servicios web de alta performance.

- **Spring Boot Starter WebSocket:**[27] Ofrece soporte para funcionalidades de WebSocket, permitiendo el desarrollo de aplicaciones que requieren comunicaciones bidireccionales en tiempo real entre el cliente y el servidor.

- **Spring Boot Starter Data JPA:**[24] Facilita la implementación de capas de persistencia de bases de datos utilizando Java Persistence API (JPA). Simplifica la configuración y el acceso a bases de datos para operaciones CRUD.

- **JTS Core:**[11] Una biblioteca para la creación y manipulación de geometrías en Java. Esencial para proyectos que requieren cálculos espaciales y geográficos complejos. Hemos utilizado el Quadtree de esta biblioteca.

- **SockJS Client y Stomp WebSocket:**[20][28] Estas bibliotecas se utilizan conjuntamente para facilitar la comunicación sobre WebSockets en aplicaciones web, empleando SockJS para la compatibilidad del lado del cliente y STOMP para el protocolo de mensajería.

- **Lombok:**[12] Reduce el boilerplate en Java, automatizando la generación de getters, setters, constructores y otros métodos comunes a través de anotaciones.

- **Spring Boot Devtools:**[23] Proporciona herramientas de desarrollo que facilitan la experiencia de programación, como el reinicio automático de aplicaciones para reflejar cambios en el código.

- **Spring Boot Configuration Processor:**[22] Mejora la asistencia al desarrollador en la configuración de aplicaciones Spring Boot, proporcionando metadatos útiles para el autocompletado de propiedades en archivos de configuración.

- **Spring Boot Starter Test:**[25] Incluye una amplia gama de herramientas de prueba, como JUnit, Hamcrest y Mockito, para facilitar la implementación de pruebas unitarias y de integración en aplicaciones Spring Boot.

Lenguaje de programación para el frontend: HTML, CSS y Javascript

De cara al frontend, se utilizó una estructura básica de un proyecto de React, utilizando Javascript en vez de Typescript pero sin utilizar ningún tipo de librería externa para el frontend. Todo el CSS y el HTML se desarrolló 'a mano'.

- **stompjs:**[29] Utilizada para la comunicación sobre WebSockets, esta librería permite a las aplicaciones cliente suscribirse a los canales de mensajes y enviar mensajes a través de STOMP, un protocolo de mensajería simple que opera sobre WebSockets. Esencial para aplicaciones en tiempo real que requieren intercambio de datos fluido y eficiente.
- **p5js:**[15] Una librería de JavaScript que simplifica la programación de gráficos y visualizaciones interactivas en el navegador. Es ampliamente utilizada en proyectos creativos de codificación, educación en programación, y arte digital, proporcionando un lienzo flexible para la experimentación visual y la interacción.
- **React:**[16] Una librería de JavaScript para construir interfaces de usuario. React facilita la creación de vistas para cada estado en la aplicación, actualizando y renderizando de forma eficiente los componentes correctos cuando los datos cambian. Es fundamental para el desarrollo de aplicaciones de página única (SPA) que requieren interfaces dinámicas y reactivas.
- **react-dom:**[17] Complementa a React al permitir la manipulación del DOM y la gestión eficiente del árbol de componentes en aplicaciones web. Esta librería es necesaria para integrar React en la página web, permitiendo que las interfaces de usuario de React interactúen con el navegador.
- **react-scripts:**[1] Forma parte de Create React App y proporciona un conjunto de scripts para automatizar tareas de desarrollo comunes, como la construcción, pruebas y lanzamiento de aplicaciones React. Simplifica la configuración inicial y reduce la necesidad de gestión de configuraciones complejas.
- **sockjs-client:**[21] Proporciona una alternativa a la API de WebSocket que puede trabajar en una amplia gama de navegadores, incluyendo aquellos que no soportan WebSocket. Es útil para garantizar que las

aplicaciones web en tiempo real sean accesibles a todos los usuarios, independientemente de las limitaciones de su navegador.

4.3. Entorno de desarrollo

Para este proyecto, la selección de un entorno de desarrollo integrado (IDE) adecuado es crucial para la eficiencia y efectividad del proceso de desarrollo. Se consideraron diversos IDEs tanto para el desarrollo del backend en Java como para el frontend utilizando React. A continuación, se detallan las decisiones tomadas para cada uno de estos componentes.

Backend: IntelliJ IDEA Ultimate

Para el desarrollo del backend en Java, se evaluaron varios IDEs, incluyendo:

- Eclipse
- IntelliJ IDEA Ultimate
- Visual Studio Code

Después de una cuidadosa consideración, se seleccionó **IntelliJ IDEA Ultimate**[10] por las siguientes razones:

- **Soporte avanzado para Java y frameworks relacionados:** IntelliJ IDEA Ultimate ofrece una integración profunda con el ecosistema Java, incluyendo soporte excepcional para frameworks populares como Spring, Spring Boot, Hibernate, y muchos otros, lo cual es indispensable para el desarrollo backend moderno.
- **Herramientas de desarrollo y debugging integradas:** Esta versión proporciona herramientas avanzadas de debugging, perfiles, y testing, que son superiores a las ofrecidas por su versión Community o alternativas como Eclipse y NetBeans.
- **Facilidades para el desarrollo de aplicaciones web y empresariales:** IntelliJ IDEA Ultimate viene con características específicas para el desarrollo de aplicaciones web y empresariales, tales como soporte para tecnologías de frontend, bases de datos, y herramientas de control de versiones integradas que simplifican el flujo de trabajo.

- **Licencia gratuita:** JetBrains, la empresa que desarrolla IntelliJ IDEA, ofrece una licencia gratuita para estudiantes, por lo que aprovechamos la oportunidad y utilizamos la versión Ultimate.

La elección de IntelliJ IDEA Ultimate se justifica por su capacidad para mejorar significativamente la productividad del desarrollador, proporcionando un entorno de desarrollo cohesivo y potente para el backend Java.

Frontend: Visual Studio Code

Se optó por **Visual Studio Code**[13] debido a:

- **Extensa biblioteca de extensiones:** Visual Studio Code ofrece una vasta gama de extensiones disponibles para el desarrollo en React, facilitando desde la sintaxis y el resaltado de código hasta la integración con sistemas de control de versiones y herramientas de debugging.
- **Ligero pero potente:** A diferencia de IDEs más pesadas como WebStorm, Visual Studio Code es ligero, rápido al iniciar y ejecutar, pero al mismo tiempo ofrece funcionalidades poderosas para el desarrollo de aplicaciones web.
- **Amplio soporte comunitario:** La popularidad de Visual Studio Code asegura un soporte comunitario extenso, lo que se traduce en una gran cantidad de recursos de aprendizaje, tutoriales, y plugins actualizados para mejorar el flujo de trabajo de desarrollo en React.

La combinación de estas características hace de Visual Studio Code una opción ideal para el desarrollo del frontend, equilibrando flexibilidad, potencia y facilidad de uso.

4.4. Gestión del repositorio: GitKraken

Para la gestión del repositorio Git del proyecto, se decidió utilizar **GitKraken**[2], una herramienta de interfaz gráfica de usuario (GUI) para Git, por varias razones clave que la distinguen de otras opciones disponibles:

- **Interfaz de Usuario Intuitiva:** GitKraken presenta una interfaz visualmente atractiva y fácil de navegar que simplifica la visualización del historial de commits, ramas, y merges, lo cual es especialmente

útil para equipos que incluyen miembros menos familiarizados con la línea de comandos de Git.

- **Integración con Servicios de Repositorios:** Ofrece integración directa con servicios de repositorios populares como GitHub, GitLab, y Bitbucket, facilitando la clonación de repositorios, el push y el pull de cambios, y la gestión de pull requests sin salir de la aplicación.
- **Funcionalidades Avanzadas de Gestión de Ramas:** GitKraken proporciona herramientas avanzadas para la gestión de ramas, incluyendo un fácil manejo de fusiones y resolución de conflictos, lo que mejora la eficiencia en el manejo de proyectos complejos y el trabajo colaborativo.

La elección de GitKraken se basó en su capacidad para hacer el proceso de gestión de Git más accesible y eficiente, gracias a su combinación de una interfaz de usuario amigable, integración con múltiples plataformas de hosting de código, y potentes herramientas de gestión de ramas y conflictos. Esto ha permitido al equipo centrarse más en el desarrollo y menos en los desafíos técnicos de la gestión de versiones, lo que en última instancia contribuye al éxito del proyecto.

4.5. Documentación

Se consideraron los siguientes programas para escribir esta documentación, dado que se nos había provisto de plantillas en formato *.tex* y *.odt*:

- Microsoft Word
- LibreOffice Writer
- TexMaker
- Overleaf

Finalmente, seleccionamos **Overleaf** por varias razones:

- Es extremadamente flexible.
- La curva de aprendizaje y facilidad de manejo es menos inclinada que la de TexMaker. Esto me permitió ser más eficiente a la hora de escribir y moverme entre ficheros, hecho mucho que me resultaba mucho más difícil en un entorno de TexMaker.

- Nos permite trabajar en LaTeX, que ofrece una gran cantidad de posibilidades para textos académicos.

Aspectos relevantes del desarrollo del proyecto

Este capítulo aborda los elementos clave del proceso de desarrollo del proyecto, desde la elección del tema y la planificación inicial hasta los desafíos encontrados durante su realización. Se detalla el razonamiento detrás de las decisiones adoptadas a lo largo de este proceso.

5.1. Origen de la idea

Este proyecto se concibió con el objetivo de llevar a cabo una de las ideas que llevaban un tiempo rondando mi cabeza. Desde jóven, los videojuegos han sido un elemento fundamental de mi vida y, poco a poco, mi interés se fue orientando hacia el ***cómo***, las técnicas, estrategias y herramientas que se utilizan para desarrollarlos. Mi interés en este tema se fue avivando viendo vídeos sobre cómo se utilizaban ciertas técnicas para optimizar videojuegos para consolas de anterior generación, sobre cómo conseguir obtener el máximo rendimiento de un hardware limitado, sobre como comprimir un juego en un cartucho...[5]

Poco después, descubrí el trabajo de Daniel Shiffman, profesor asociado de artes en la Escuela artística TISCH de la Universidad de Nueva York, dónde se propone la creación de obras artísticas a través de la programación y de conceptos matemáticos. Gracias a sus vídeos, fui introducido a The Nature of Code [19], un libro en el que se explora la representación de entornos naturales y sus componentes a través de Processing [7], una librería

[5]Por ejemplo, este vídeo de Modern Vintage Gamer donde se consigue comprimir un juego de PSX (CD) en un cartucho de N64.

de Java para desarrollar "sketches" y diseños. En él, se abordan temas como las leyes del movimiento de Newton, agentes autónomos y demás conceptos que después he utilizado en el proyecto.

Combinando estos dos hechos, y dada la necesidad de escoger un tema para mi trabajo de fin de grado, consideré que era la oportunidad adecuada para invertir el tiempo y esfuerzo necesario para desarrollar un pequeño motor de físicas, inspirado en los sistemas que se proponen en The Nature of Code.

5.2. Primeros pasos. Análisis del problema.

Una vez decidido que iba a desarrollar el motor, surgen varios problemas a resolver:

- **Arquitectura y estrategia.** Para este primer punto, hubo que hacer un ejercicio de investigación para establecer un punto de arranque. Tras revisar libros(Game Engine Architecture, Jason Gregory[9]) y estrategias que implementaban otros motores (dyn4j[3]) de videojuegos, se optó por un modelo relativamente sencillo que se apoyase en una arquitectura simplificada basada en un motor clásico. En este, se trabajaría con un *game loop*, básicamente un bucle infinito, en el que en cada iteración se toma el estado actual del modelo y se actualiza basándose en cálculos sobre este mismo. También, se estableció que el cliente web, utilizaría una estrategia de *polling*[6] para pedir datos continuamente al backend, pero al ritmo que pueda el cliente, eliminando el problema del *back pressure*[7].

- **Herramientas. Lenguaje de programación. Plataforma.** Respecto al lenguaje de programación, partíamos de que queríamos utilizar

[6]El *polling* en programación se refiere a la técnica de sondeo activo donde un programa o dispositivo constantemente verifica el estado de uno o varios recursos externos o condiciones para determinar si algún trabajo específico necesita ser ejecutado. Esta técnica es comúnmente utilizada para monitorear la disponibilidad de nuevos datos en un buffer, verificar el estado de una conexión de red, o esperar por un evento específico.

[7]El *back pressure* se refiere a la situación en sistemas de programación asíncrona o basados en flujos donde un componente recibe datos a un ritmo más rápido de lo que puede procesar. Esto puede llevar a una acumulación de datos no procesados, afectando negativamente el rendimiento del sistema y potencialmente causando la pérdida de datos si el buffer se llena. Para manejar este problema, los sistemas pueden implementar mecanismos que permitan regular la velocidad de entrada de datos, pausando o ralentizando la fuente de datos hasta que el componente pueda procesar la acumulación.

Java para construir nuestro motor, luego no surgieron dudas[8]. Además, esto también esclarecía la cuestión de la plataforma, ya que en un lenguaje JVM[9], solo tienes que programar de cara a la JVM, sin importar arquitectura o sistema operativo. Sin embargo, sí que hubo que realizar un análisis para determinar que framework podría ayudarnos a atajar el problema de la forma más eficiente, equilibrando el rendimiento real con la dificultad de adaptarse al framework y su curva de dificultad. En este caso, se eligió Spring por encima de otros como Quarkus dado que tenía unas nociones previas en el framework y nos ofrecía un conjunto de herramientas en Spring Boot que iban a simplificar el despliegue y desarrollo de la aplicación notablemente. De cara al frontend, haremos un breve resumen en el siguiente apartado.

- **Interacción con el usuario.** De cara a la interacción con el usuario, aquí surgieron varios problemas. No tenía apenas experiencia previa en el mundo del desarrollo web y hay *muchas* opciones. La cantidad de frameworks, librerías, lenguajes, etc, disponibles para el desarrollo web puede resultar apabullante. Debido a esto, al principio propuse realizar el cliente frontend en puro HTML+Javascript, pero pronto me di cuenta de que había ciertas cuestiones de diseño que serían mucho más fáciles de resolver utilizando una librería o un framework. Tras experimentar un poco con unas cuantas opciones, al final me decanté por desarrollar una pequeña aplicación utilizando React, que también era nuevo para mí, pero que tiene una comunidad enorme y un montón de tutoriales información disponibles para ayudarme a desarrollar.

5.3. Arquitectura de la aplicación

Se incluye un diagrama describiendo la arquitectura de la aplicación:

[8]Lenguajes como Kotlin, que también se ejecutan en la JVM (Máquina Virtual de Java), representan opciones válidas para el desarrollo debido a su interoperabilidad con Java, permitiendo un uso mixto de ambos lenguajes en el mismo proyecto.

[9]Java Virtual Machine

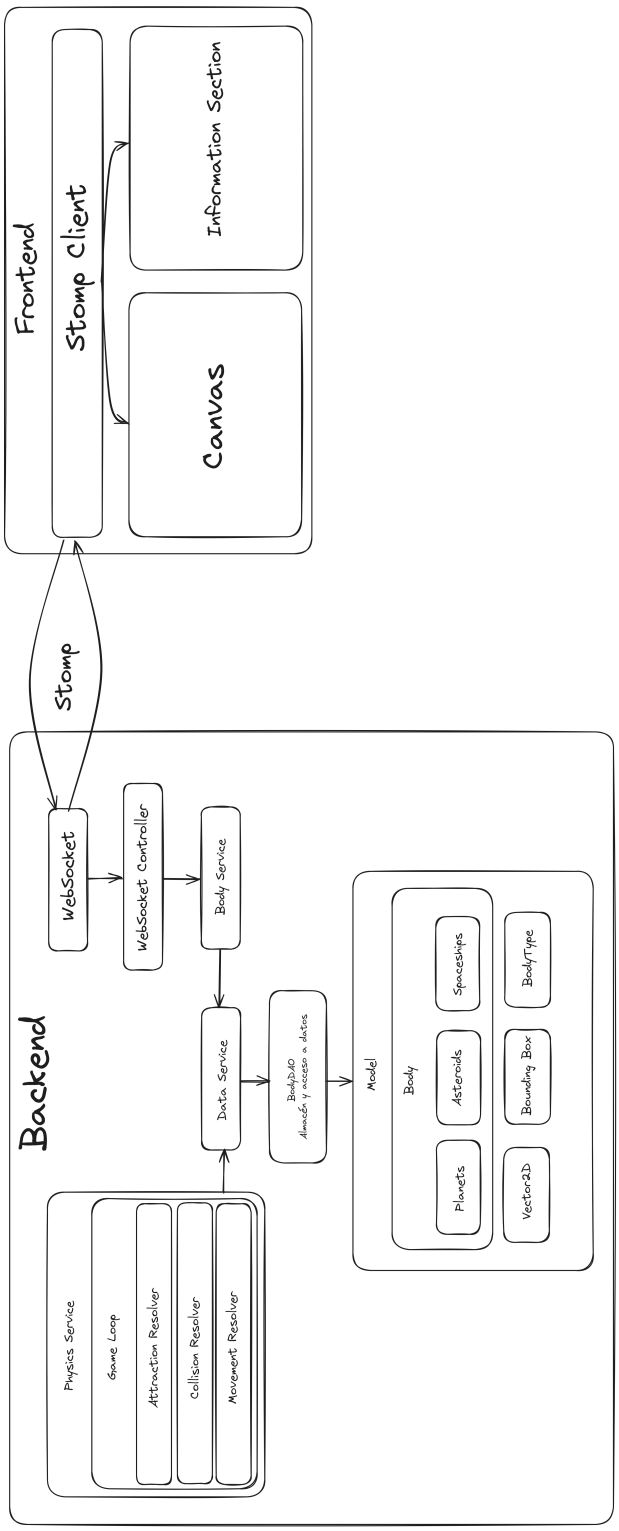

Figura 5.1: Arquitectura de la aplicación.

5.4. Backend

El backend del motor de físicas, desarrollado con Spring Boot y complementado con servicios especializados, constituye el núcleo de la simulación física. Este sistema se encarga de gestionar la lógica de simulación, el procesamiento de entidades físicas, y la comunicación en tiempo real con el frontend a través de WebSockets. El backend se estructura alrededor de varios componentes clave que colaboran para simular un entorno físico dinámico:

- **PhysicsEngine:** Punto de entrada de la aplicación, define las dimensiones del espacio de simulación y configura un executor programado para actualizar la simulación a una tasa fija (tickRate). Este enfoque asegura que la simulación avance de manera consistente y predecible.

```
public ScheduledExecutorService
scheduledExecutorService(PhysicsService physicsService) {
    physicsService.setup();
    ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);
    int tickRate = 150;
    executor.scheduleAtFixedRate(physicsService::tick, 0, 1000 / tickRate, TimeUnit.MILLISECONDS);
    return executor;
}
```

Figura 5.2: ExecutorService, ticks

- **PhysicsService:** Corazón de la lógica de simulación, coordina las operaciones esenciales como la configuración inicial del entorno, la actualización periódica de estados (tick), y la ejecución de algoritmos para resolver atracciones, colisiones y movimientos. Este servicio interactúa con varios resolutores especializados y un servicio de datos para manejar el ciclo de vida y la dinámica de las entidades.

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

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 5.3: Procedimiento ‘tick‘

- **DataService y BodyDAOImpl:** Gestionan el almacenamiento, recuperación y manipulación de las entidades físicas (Body). Permiten la adición, actualización y eliminación de cuerpos, además de generar entidades aleatorias para dinamizar la simulación.
- **Resolutores (AttractionResolver, CollisionResolver, MovementResolver):** Implementan los algoritmos específicos para calcular fuerzas gravitacionales, detectar y resolver colisiones, y actualizar el movimiento de las entidades, respectivamente. Estos componentes aplican las leyes de la física para modelar interacciones realistas entre cuerpos.
- **QuadtreeService:** Optimiza la detección de colisiones mediante la división del espacio de simulación en sectores. Este enfoque reduce significativamente el número de comparaciones necesarias al identificar posibles colisiones solo entre cuerpos cercanos.

De cara a la comunicación en tiempo real con el cliente, la integración de WebSockets, configurada en **WebSocketConfig**, facilita la comunicación bidireccional y en tiempo real entre el backend y el frontend.

```java
@Override
public void configureMessageBroker(MessageBrokerRegistry configurer) {
    configurer.setApplicationDestinationPrefixes("/app");
    configurer.enableSimpleBroker("/topic");
    configurer.setUserDestinationPrefix("/user");
}


@Override
public void registerStompEndpoints(StompEndpointRegistry registry) {
    registry.addEndpoint("/physics-engine-websocket")
            .setAllowedOriginPatterns("*")
            .withSockJS();
}
```

Figura 5.4: Declaración del endpoint y topics

Esto permite:

- **Transmisión de estados:** El estado actualizado de la simulación se envía al frontend, permitiendo a los usuarios visualizar los cambios en tiempo real.
- **Recepción de requests:** El backend recibe y procesa comandos del frontend, como la adición de cuerpos o la modificación de parámetros de simulación, ofreciendo una experiencia interactiva y dinámica.

Estos comandos o 'requests' se gestionan desde una clase controlador. En concreto, 'WebSocketController', maneja los mensajes entrantes de los clientes, ejecutando acciones como agregar nuevas entidades o modificar las existentes. Esto incluye comandos para generar cuerpos aleatorios, seleccionar o mover entidades específicas, y actualizar la vista del cliente con los cambios en el estado de la simulación.

```java
@MessageMapping("/random-body")
@SendTo("/topic/bodies")
public void addRandomBody() {
    bodyService.addRandomBody();
}
```

Figura 5.5: Ejemplo de una request al backend

Modelo

El modelo del proyecto en nuestro motor de físicas desarrollado en Java, específicamente diseñado para la simulación de entornos dinámicos, se basa en la representación y manejo de entidades denominadas çuerpos"(Body). Estos cuerpos interactúan entre sí y con su entorno bajo reglas físicas definidas, incluyendo gravedad, colisiones, y movimientos dirigidos.

Figura 5.6: Diagrama UML del modelo

Body:

La clase abstracta Body es el núcleo del modelo, representando cualquier objeto físico dentro de la simulación. Cada Body posee propiedades fundamentales como posición (position), velocidad (velocity), aceleración (acceleration), masa (mass), y un cuadro delimitador (BoundingBox) para detección de colisiones. La capacidad de ser seleccionado (selected) permite interactuar con cuerpos específicos dentro de la simulación, facilitando su manipulación o estudio detallado.

- **Aplicación de fuerzas:** Los cuerpos pueden tener fuerzas aplicadas a ellos, alterando su estado de movimiento conforme a la segunda ley de Newton. La implementación facilita la simulación de interacciones como empujes y atracciones.
- **Restricciones de tamaño:** La verificación de restricciones asegura que los cuerpos permanezcan dentro de los límites espaciales definidos, evitando que escapen del entorno visible de la simulación.
- **Actualización de su estado:** La dinámica del cuerpo se actualiza en cada ciclo del motor, recalculando su posición basándose en su velocidad y aceleración actuales.

BoundingBox

El BoundingBox proporciona una representación simplificada del espacio ocupado por un cuerpo, crucial para la detección eficiente de colisiones. La intersección de cuadros delimitadores indica una potencial colisión, un paso preliminar antes de realizar cálculos de colisión más detallados.

Especializaciones de Body

- **Planet:** Representa cuerpos celestes con masa significativa y generalmente estacionarios. Su radius define el tamaño visible y la influencia gravitacional.
- **Asteroid:** Cuerpos más pequeños y posiblemente irregulares que pueden tener trayectorias complejas. Se caracterizan por un conjunto de vértices que definen su forma y un factor de rotación (spin), añadiendo realismo a su representación visual.
- **Spaceship:** Entidades controlables dentro de la simulación, capaces de comportamientos autónomos como deambular, evitar colisiones y

buscar objetivos. Las aeronaves utilizan una combinación de fuerzas para simular movimiento dirigido y maniobrabilidad.

Vector2D

La clase Vector2D, una representación de vectores bidimensionales, es una de las piezas fundamentales en el desarrollo del motor de físicas. Esta entidad no solo modela la posición, velocidad, y aceleración de objetos en el espacio bidimensional, sino que también facilita la implementación de operaciones vectoriales esenciales para la simulación de dinámicas físicas. 'Vector2D encapsula las coordenadas x e y de un vector, proveyendo una base para representar direcciones y magnitudes en el plano. A través de sus constructores, se permite la inicialización a valores predeterminados (0,0) o a coordenadas específicas, otorgando flexibilidad en la creación y manipulación de vectores. La clase ofrece las siguientes operaciones vectoriales:

- **Producto escalar:** a capacidad de calcular el producto escalar entre vectores es crucial para determinar la orientación relativa y la proyección de un vector sobre otro, aplicaciones que van desde la detección de colisiones hasta el cálculo de fuerzas.
- **Magnitud y Normalización:** Estas operaciones permiten determinar la longitud de un vector y ajustarlo a una magnitud unitaria, respectivamente. La normalización es especialmente útil para mantener consistentes las magnitudes de las fuerzas aplicadas en la simulación, garantizando un comportamiento físico coherente.
- **Adición y Sustracción:** Facilitan la composición y descomposición de vectores, operaciones fundamentales en el cálculo de velocidades y aceleraciones resultantes de la aplicación de múltiples fuerzas.
- **Multiplicación y División por Escalar:** Estas operaciones ajustan la magnitud de los vectores, permitiendo escalar las fuerzas aplicadas y las distancias recorridas sin alterar su dirección.
- **Distancia:** La función de distancia entre dos vectores apoya el cálculo de separaciones espaciales, esencial para determinar la proximidad entre objetos y gestionar interacciones como atracciones gravitacionales o repulsiones.
- **Limitación:** Restringir la magnitud de un vector a un valor máximo es vital para controlar velocidades y prevenir comportamientos físicos no realistas dentro de la simulación.

Atracción

Para simular la atracción gravitacional en nuestro motor de físicas, desarrollamos el componente AttractionResolver. Este componente es responsable de calcular y aplicar la fuerza de atracción gravitacional entre todos los cuerpos presentes en la simulación. El proceso se realiza en dos pasos principales:

1. **Cálculo de la Fuerza de Atracción:** Para cada par de cuerpos, calculamos la dirección y magnitud de la fuerza de atracción entre ellos. Utilizamos la posición relativa de los cuerpos para determinar la dirección de la fuerza y aplicamos la fórmula de la ley de gravitación para calcular su magnitud. Este cálculo toma en consideración la distancia entre los cuerpos para asegurar que la fuerza disminuye con el cuadrado de la distancia, tal como dicta la ley de Newton.

2. **Aplicación de la Fuerza:** Una vez calculada la fuerza de atracción, la aplicamos a los cuerpos afectados. Esto se traduce en una modificación de sus vectores de velocidad según la segunda ley de movimiento de Newton, que establece que la aceleración de un objeto es directamente proporcional a la fuerza neta actuante sobre él e inversamente proporcional a su masa.

```
public void calculateAttractions(List<Body> bodies, @NonNull QuadtreeService quadtreeService) {
    for (Body body : bodies) {
        List<Body> nearbyBodies = quadtreeService.queryNearbyBodies(body);
        for (Body nearbyBody : nearbyBodies) {
            if (!body.equals(nearbyBody)) {
                Vector2D pull = gravitationalPull(body, nearbyBody);
                body.applyForce(pull);
            }
        }
    }
}
```

Figura 5.7: Proceso del cálculo de todas las fuerzas atractivas

```
    private Vector2D gravitationalPull(@NonNull Body attractor, @NonNull Body mover) {
        //1. Calculate the direction of the force
        Vector2D force = Vector2D.sub(attractor.getPosition(), mover.getPosition());
        float distance = force.magnitude(); // We will use it later.
        force.normalize();

        //2. Calculate the magnitude of the force
        double GRAVITATIONAL_CONST = 6.67428e-11;
        float mag = (float) ((GRAVITATIONAL_CONST * attractor.getMass() * mover.getMass()) /
(Math.pow(distance, 2)));

        //3. Apply said magnitude.
        force.multiply(mag);

        return force;
    }
```

Figura 5.8: Cálculo de la fuerza atractiva entre dos cuerpos

Para mejorar la eficiencia del proceso de cálculo de atracciones y evitar la computación innecesaria de interacciones entre cuerpos distantes con efectos despreciables, integramos un servicio de Quadtree. Este servicio nos permite consultar rápidamente los cuerpos cercanos a un objeto dado y limitar el cálculo de la fuerza gravitacional solo a aquellos pares de cuerpos con una probabilidad significativa de influirse mutuamente.

Colisiones

El tratamiento de colisiones es un componente fundamental en la simulación de físicas para lograr interacciones realistas entre objetos. En nuestro motor, este proceso se divide en dos etapas principales: la detección de colisiones y su resolución.

Detectar

La detección de colisiones comienza con una evaluación preliminar para identificar pares de cuerpos que potencialmente interactúan. Implementamos esta fase utilizando un servicio de Quadtree, una estructura de datos que divide el espacio de simulación en cuadrantes para optimizar las consultas espaciales. Para cada cuerpo, consultamos el Quadtree para obtener una lista de cuerpos cercanos y luego examinamos si hay intersección entre sus volúmenes de contención, típicamente representados por bounding boxes o esferas.

```java
public boolean isColliding(@NonNull Body body1, @NonNull Body body2) {
    // Get the center and radius from the bounding boxes
    Vector2D center1 = body1.getBbox().getCenter();
    Vector2D center2 = body2.getBbox().getCenter();
    float radius1 = body1.getBbox().getRadius();
    float radius2 = body2.getBbox().getRadius();

    // Calculate the distance between the two centers
    double distance = center1.distance(center2);

    // Check if the distance is less than or equal to the sum of the two radii
    return distance <= (radius1 + radius2);
}
```

Figura 5.9: Detectamos si dos cuerpos están colisionando

Para determinar si dos cuerpos están colisionando, calculamos la distancia entre los centros de sus bounding boxes o esferas. Si esta distancia es menor o igual a la suma de sus radios, concluimos que los cuerpos están colisionando.

Resolver

Una vez detectada una colisión, procedemos a resolverla calculando las fuerzas involucradas y ajustando las velocidades de los cuerpos para reflejar el impacto. Este proceso se basa en las leyes de conservación del momento y la energía cinética.

1. **Calculamos el vector normal** de la colisión, que es un vector unitario que apunta desde un cuerpo hacia el otro. Este vector es fundamental para determinar la dirección de las fuerzas aplicadas como resultado de la colisión.

```java
// Step 1: Calculate the normal and relative velocity
Vector2D normal = Vector2D.normalize(Vector2D.sub(body2.getPosition(), body1.getPosition()));
Vector2D relVel = Vector2D.sub(body2.getVelocity(), body1.getVelocity());
```

Figura 5.10: Calculamos el vector normal

2. **Determinamos la velocidad relativa** de los cuerpos en la dirección del vector normal. Si esta velocidad es positiva, indica que los cuerpos se están separando y no se requiere más acción.

```
// Step 2: Find the relative velocity along the normal
double relVelAlongNormal = Vector2D.dot(relVel, normal);
```

Figura 5.11: Determinamos la velocidad relativa

```
// Step 3: Early exit if bodies are separating
if (relVelAlongNormal > 0) {
    return;
}
```

Figura 5.12: Descartamos cuerpos

3. **Calculamos el impulso escalar** necesario para resolver la colisión. Este valor depende de la velocidad relativa de los cuerpos, sus masas, y el coeficiente de restitución, que representa la .elasticidad"de la colisión. Un coeficiente de restitución de 1 indica una colisión perfectamente elástica, mientras que un valor de 0 indica una colisión perfectamente inelástica.

```
// Step 4: Calculate the impulse
double restitution = 0.8; // Adjust this value to get the desired bounce effect
double impulseScalar = -(1 + restitution) * relVelAlongNormal;
impulseScalar /= (1 / body1.getMass()) + (1 / body2.getMass());
```

Figura 5.13: Calculamos el impulso escalar

4. **Aplicamos el impulso** calculado a cada cuerpo, ajustando sus velocidades en la dirección del vector normal. Esto simula el efecto del rebote y asegura que los cuerpos se separen tras la colisión.

```
// Step 5: Find the impulse vector and apply the impulse
Vector2D impulseVector = Vector2D.multiply(normal, (float) impulseScalar);
Vector2D velocityChange1 = Vector2D.multiply(impulseVector, (float) (1 / body1.getMass()));
Vector2D velocityChange2 = Vector2D.multiply(impulseVector, (float) (1 / body2.getMass()));

body1.setVelocity(Vector2D.sub(body1.getVelocity(), velocityChange1));
body2.setVelocity(Vector2D.add(body2.getVelocity(), velocityChange2));
```

Figura 5.14: Aplicamos el impulso

Aeronaves

Estas entidades están diseñadas para navegar el entorno de simulación, interactuando tanto con objetos estáticos como dinámicos. La implementación de estos comportamientos se basa en principios tales como deambulación aleatoria, evasión de obstáculos y seguimiento de objetivos, que se detallan a continuación.

Deambular

El comportamiento de deambulación proporciona a las aeronaves una capacidad para moverse de manera exploratoria sin un destino predeterminado. Este mecanismo se implementa mediante la generación de una fuerza aleatoria que modifica la trayectoria actual de la aeronave en cada ciclo de actualización del motor. Este vector de fuerza aleatoria, aplicado continuamente, resulta en un movimiento impredecible y variado, simulando la tendencia natural de un organismo a explorar su entorno. La magnitud de esta fuerza se ajusta para garantizar que el movimiento resultante sea suave y coherente, evitando cambios bruscos de dirección que podrían romper la inmersión del usuario en la simulación.

1. **Generación de la Fuerza Aleatoria:** En cada ciclo de actualización, se genera un vector de fuerza aleatorio que determina la nueva dirección de movimiento. Esta fuerza simula una decisión espontánea de cambio de dirección, imitando el comportamiento exploratorio en seres vivos o vehículos autónomos.

2. **Aplicación de la Fuerza al Movimiento:** La fuerza aleatoria se aplica a la aeronave, modificando su vector de velocidad actual. Esto resulta en cambios graduales en la dirección de movimiento,

permitiendo a la aeronave deambular de forma natural por el espacio de simulación.

```
public void wander() {
    // Create a random vector for wandering
    double angle = Math.random() * 2 * Math.PI; // Random angle
    Vector2D wanderForce = new Vector2D(Math.cos(angle), Math.sin(angle));
    wanderForce.multiply(WANDER_FORCE_MAGNITUDE); // Adjust the magnitude as needed

    this.applyForce(wanderForce);
}
```

Figura 5.15: Exploración o deambulación

Evitar al resto de cuerpos

Para simular la percepción espacial y la capacidad de las aeronaves para reaccionar ante la presencia de obstáculos, se implementa el comportamiento de evasión. Este proceso involucra la detección de objetos cercanos y la generación de una fuerza de evasión que redirige la aeronave, evitando colisiones. La dirección de esta fuerza es opuesta a la del objeto detectado, y su magnitud es inversamente proporcional a la distancia al objeto, lo que significa que cuanto más cerca esté el obstáculo, mayor será la fuerza aplicada para evitarlo. Este mecanismo es esencial para mantener la integridad física de las aeronaves en entornos densamente poblados o complejos.

1. **Detección de Obstáculos Cercanos:** mediante el uso de sensores virtuales o consultas al sistema de cuadros (como Quadtree), la aeronave identifica objetos cercanos que potencialmente representan una amenaza de colisión.

2. **Cálculo de la Fuerza de Evasión:** para cada objeto detectado, se calcula una fuerza de evasión proporcional a la distancia al objeto. Esta fuerza tiene dirección opuesta al objeto detectado, empujando a la aeronave en la dirección segura.

3. **Aplicación de la Fuerza de Evasión:** la fuerza de evasión calculada se suma al vector de fuerza actual de la aeronave, alterando su trayectoria para evitar el obstáculo.

```
public void avoidCollisions(List<Body> nearbyBodies) {
    Vector2D avoidanceForce = new Vector2D(0, 0);

    for (Body nearbyBody : nearbyBodies) {
        float distance = this.getPosition().distance(nearbyBody.getPosition());
        if (distance < AVOIDANCE_RANGE && distance > 0) {
            Vector2D awayFromBody = Vector2D.sub(this.getPosition(), nearbyBody.getPosition());
            if (!awayFromBody.isZero()) {
                awayFromBody.normalize();
                awayFromBody.divide(distance);
                avoidanceForce.add(awayFromBody);
            }
        }
    }

    // Cap the avoidance force to avoid excessively large values
    if (avoidanceForce.magnitude() > MAX_AVOIDANCE_FORCE) {
        avoidanceForce.normalize();
        avoidanceForce.multiply(MAX_AVOIDANCE_FORCE);
    }

    this.applyForce(avoidanceForce);
}
```

Figura 5.16: Evasión

Buscar

La capacidad de las aeronaves para identificar y dirigirse hacia un objetivo específico es fundamental para simular comportamientos dirigidos, como la persecución o el seguimiento de rutas. Este comportamiento se modela calculando un vector de deseo que apunta hacia el objetivo desde la posición actual de la aeronave. La velocidad de la aeronave se ajusta entonces para alinearla con este vector de deseo, permitiendo una aproximación suave y controlada hacia el objetivo. A medida que la aeronave se acerca a su destino, la magnitud de la fuerza aplicada se reduce para simular una desaceleración natural, facilitando un aterrizaje o una parada precisa. El vector objetivo se adquiere a través de la entrada del usuario, que puede pinchar en cualquier sección del canvas desde el cliente y se mandan unas coordenadas que después serán convertidas en un vector:

```java
@MessageMapping("/move-selected-spaceships")
public void moveSelectedSpaceships(TargetPositionDTO positionDTO) {
    List<Spaceship> selectedSpaceships = bodyService.getSelectedSpaceships();
    Vector2D target = new Vector2D(positionDTO.getX(), positionDTO.getY());
    for (Spaceship spaceship : selectedSpaceships) {
        spaceship.seek(target);
    }
}
```

Figura 5.17: Llamada al backend con el vector objetivo

```java
@Getter
public class TargetPositionDTO {
    private double x;
    private double y;

    public void setX(double x) {
        this.x = x;
    }
    public void setY(double y) {
        this.y = y;
    }
}
```

Figura 5.18: Tratamiento de la entrada del usuario recibida desde el frontend

1. **Cálculo del vector objetivo:** Se calcula un vector objetivo que apunta directamente desde la aeronave hacia el objetivo. Este vector guía el movimiento de la aeronave, indicando la dirección óptima hacia el objetivo.

2. **Ajuste de la velocidad hacia el objetivo:** La velocidad de la aeronave se modifica en función del vector objetivo. A medida que la aeronave se acerca al objetivo, la magnitud de la fuerza aplicada se ajusta para simular una desaceleración, permitiendo una llegada suave o un acercamiento preciso.

3. **Maniobra de acercamiento:** En la zona de acercamiento, se aplican fuerzas de dirección y magnitud calculadas para reducir la velocidad de manera proporcional a la distancia restante hasta el objetivo, facilitando una transición fluida desde el movimiento rápido hasta el estado de llegada o estacionamiento.

```java
public void seek(Vector2D target) {
    float slowingRadius = 100f;
    this.seek(target, slowingRadius);
}

public void seek(Vector2D target, float slowingRadius) {
    Vector2D desired = Vector2D.sub(target, this.getPosition());
    float distance = desired.magnitude();
    desired.normalize();

    if (distance < slowingRadius) {
        // Inside the slowing area (Arrive behavior)
        float m = map(distance, 0, slowingRadius, 0, MAX_VELOCITY);
        desired.multiply(m);
    } else {
        // Outside the slowing area (Seek behavior)
        desired.multiply(MAX_VELOCITY);
    }

    // Steering force calculation
    Vector2D steer = Vector2D.sub(desired, this.getVelocity());
    steer.limit(MAX_FORCE);

    // Apply the combined steering force
    this.applyForce(steer);
}
```

Figura 5.19: Búsqueda

5.5. Desarrollo del frontend

El desarrollo del frontend para nuestro motor de físicas se ha llevado a cabo utilizando React, un popular framework de JavaScript, lo que facilita la creación de interfaces de usuario interactivas y dinámicas. El frontend se compone de varios elementos clave diseñados para proporcionar una visualización rica y control sobre la simulación física ejecutándose en el backend. La interacción entre el frontend y el backend se establece mediante WebSockets, utilizando las librerías SockJS y Stomp. Esta conexión bidireccional permite una comunicación en tiempo real, esencial para transmitir el estado actual de la simulación física al usuario y recibir comandos del usuario para influir en la simulación.

Establecer una conexión con el backend.

Implementación:

1. **Inicialización de la conexión WebSocket:** Al iniciar la aplicación, se configura y abre una conexión WebSocket al backend. Esta conexión se mantiene activa para facilitar la comunicación continua durante la sesión del usuario.

```
useEffect(() => {
  const client = Stomp.over(function(){
    return new SockJS(getWebSocketUrl());
  });
  client.reconnect_delay = 5000;
  client.debug = () => {};
  client.connect({}, () => {
    setStompClient(client);
  });
  return () => {
    if (client) {
      client.disconnect();
    }
  };
}, []);
```

Figura 5.20: Conexión al websocket del backend

2. **Suscripción a topics:** El cliente se suscribe a topics específicos en el servidor (por ejemplo, /topic/bodies"), lo que le permite recibir actualizaciones en tiempo real sobre los cuerpos físicos en la simulación.

```
useEffect(() => {
    if (stompClient) {
      stompClient.subscribe("/topic/bodies", (message) => {
        const receivedBodies = JSON.parse(message.body);
        setBodies(receivedBodies);

        const selectedIds = receivedBodies
          .filter((body) => body.selected)
          .map((body) => body.id);
        setSelectedBodyIds(selectedIds);
      });
    }
  }, [stompClient]);
```

Figura 5.21: Suscripción a un topic

3. **Envío de requests:** El usuario puede interactuar con la simulación a través de la interfaz, como mover naves espaciales o seleccionar cuerpos, enviando comandos específicos al backend a través de la conexión WebSocket.

```
const SimulationControl = ({ stompClient }) => {
    const sendMessage = (endpoint) => {
        stompClient && stompClient.send(endpoint, {}, "");
    };

    return (
        <div className="simulationControl">
            <button onClick={() => sendMessage("/app/random-planet")}>Add Planet</button>
            <button onClick={() => sendMessage("/app/random-asteroid")}>Add Asteroid</button>
            <button onClick={() => sendMessage("/app/random-spaceship")}>Add Spaceship</button>
            <button onClick={() => sendMessage("/app/random-body")}>Add Random Body</button>
        </div>
    );
};

const sendTargetPosition = (x, y) => {
    // Send the target position to the backend
    stompClient.send("/app/move-selected-spaceships", {}, JSON.stringify({ x, y }));
};
```

Figura 5.22: Ejemplos de requests al backend

Componentes

La interfaz de usuario se compone de varios componentes React, incluidos Header, Footer, Canvas, InfoSection, y SimulationControl. Cada componente cumple con una función específica dentro de la aplicación:

- **Header y Footer:** contienen información del alumno, tutor y título del TFG.
- **Canvas:** Un componente clave donde se visualiza la simulación física. Utiliza la biblioteca p5 para dibujar los cuerpos físicos en un lienzo HTML5, representando su posición, movimiento y colisiones en tiempo real.
- **InfoSection:** Muestra información detallada sobre los cuerpos físicos presentes en la simulación, como su masa y velocidad. También permite al usuario interactuar con la simulación, por ejemplo, seleccionando y eliminando cuerpos.
- **SimulationControl:** conjunto de botones que nos permite añadir los distintos tipos de cuerpos a la simulación.

Canvas

El componente Canvas es donde se lleva a cabo la representación visual de la simulación. A través de la integración con p5, este componente dibuja en tiempo real los cuerpos físicos, basándose en los datos recibidos del backend.

Funcionalidades:

- **Visualización dinámica:** Se actualiza constantemente para reflejar el estado actual de la simulación, proporcionando una visualización interactiva de los cuerpos físicos y su comportamiento.
- **Interacción con el usuario:** Permite a los usuarios interactuar directamente con la simulación, como seleccionar cuerpos o definir objetivos, mediante acciones del ratón.

Tarjetas de información

Las tarjetas de información, presentadas en el componente InfoSection, ofrecen una vista detallada de los atributos y el estado de los cuerpos individuales dentro de la simulación. Cada tarjeta muestra datos específicos como el ID del cuerpo, tipo, masa y velocidad, y permite al usuario realizar acciones como seleccionar o eliminar cuerpos de la simulación.

Características:

- **Actualización en tiempo real:** La información se actualiza dinámicamente para reflejar cualquier cambio en el estado de los cuerpos físicos, asegurando que el usuario tenga acceso a la información más reciente.
- **Interactividad:** Las tarjetas permiten a los usuarios interactuar con elementos específicos de la simulación, facilitando la localización y borrado de cuerpos.

Trabajos relacionados

Este capítulo introduce diversos estudios relacionados con el ámbito del proyecto, proporcionando recursos adicionales para quienes estén interesados en explorar más a fondo los distintos temas tratados.

Existen una amplia cantidad de trabajos y proyectos que abordan el problema de desarrollar un motor de físicas. Dependiendo del lenguaje, las estrategias a utilizar y el rendimiento obtenido varía drásticamente. Si nos centramos en Java, podemos encontrar unos cuantos proyectos con una considerable reputación y que pueden resultar muy interesantes para investigar. Debo añadir, que la complejidad de estos proyectos es ampliamente superior a la del nuestro, ya que el objetivo de estos es proveer una librería para el uso profesional/personal, completa y continuamente actualizada (en la mayoría de los casos).

dyn4j

Dyn4j[3] es una biblioteca de físicas de código abierto para Java, diseñada para ser utilizada en proyectos de simulación y juegos. Su nombre proviene de "Dynamic"(Dinámico) y "4 Java", reflejando su propósito y el lenguaje de programación para el cual fue desarrollado. Esta biblioteca proporciona una sólida base para simular ambientes físicos, incluyendo soporte para la detección de colisiones, respuesta a colisiones, y dinámica de cuerpos rígidos.

Características principales

- Detección de Colisiones: Dyn4j incluye un sistema avanzado para la detección de colisiones entre objetos, utilizando algoritmos eficientes

para calcular cuándo y cómo los objetos en un entorno simulado se intersectan.

- Respuesta a Colisiones: Más allá de detectar colisiones, dyn4j gestiona las respuestas físicas a estas colisiones, asegurando que los objetos se comporten de manera realista al impactar unos con otros, aplicando fuerzas de reacción apropiadas según las leyes de la física.

- Dinámica de Cuerpos Rígidos: La biblioteca permite simular la dinámica de cuerpos rígidos, incluyendo la rotación, fricción, y gravedad, proporcionando un modelado detallado del movimiento y la interacción de objetos sólidos.

- Optimizado para Rendimiento: Diseñado con el rendimiento en mente, dyn4j es adecuado para juegos y aplicaciones que requieren simulaciones físicas en tiempo real sin sacrificar la fluidez o la calidad de la simulación.

- Fácil de Usar: Aunque ofrece una amplia gama de funcionalidades avanzadas, dyn4j está diseñado para ser accesible para desarrolladores con distintos niveles de experiencia en programación física.

Aplicaciones

Las aplicaciones de dyn4j son variadas, abarcando desde juegos hasta simulaciones de ingeniería y educativas. En el contexto de desarrollo de juegos, puede ser utilizado para añadir efectos realistas de física a personajes, objetos, y entornos. En el ámbito educativo o de investigación, dyn4j ofrece una plataforma para experimentar y aprender sobre la física de manera interactiva.

Comunidad y soporte

Dyn4j es apoyado por una comunidad activa de desarrolladores y usuarios que contribuyen a su desarrollo y mejora continua. La documentación, ejemplos de código, y foros de discusión están disponibles para ayudar a los usuarios a integrar dyn4j en sus proyectos.

JBox2D

JBox2D[14] es una biblioteca de física de código abierto para Java, que es un *port*[10] del muy conocido motor de físicas Box2D, originalmente escrito en C++ por Erin Catto. Box2D se ha utilizado en numerosos juegos y simulaciones debido a su robusta implementación de la física de cuerpos rígidos, y JBox2D trae estas capacidades al ecosistema de Java, permitiendo a los desarrolladores incorporar física realista en aplicaciones y juegos Java.

Características principales

- Simulación de Cuerpos Rígidos: JBox2D maneja la dinámica de cuerpos rígidos, incluyendo la simulación de movimiento, rotación, y la aplicación de fuerzas, lo que permite crear simulaciones realistas de objetos interactuando en un entorno.
- Detección y Respuesta de Colisiones: La biblioteca ofrece un sistema comprensivo para la detección de colisiones y la respuesta apropiada a estas, incluyendo el manejo de rebotes, fricción, y otras fuerzas relacionadas con colisiones.
- Joint y Constraints: Permite la creación de joints (articulaciones) y constraints (restricciones) entre cuerpos, facilitando la simulación de objetos complejos que se mueven en conjunto, como vehículos, cadenas, y sistemas de poleas.
- Sistema de Partículas: Algunas versiones y extensiones de JBox2D incluyen soporte para la simulación de partículas, permitiendo la creación de efectos como líquidos y gases.
- Optimizado para Rendimiento: Aunque la física de simulación puede ser intensiva en recursos computacionales, JBox2D está optimizada para mantener un rendimiento sólido en aplicaciones en tiempo real.

Aplicaciones

JBox2D es ampliamente utilizado en el desarrollo de juegos para Java, desde pequeños proyectos independientes hasta juegos más complejos que requieren física detallada. También se utiliza en contextos educativos para enseñar conceptos de física y programación, así como en la investigación para prototipar simulaciones de sistemas físicos.

[10] Un "port" consiste en adaptar o reescribir un proyecto desarrollado en una plataforma/lenguaje a otra.

Comunidad y soporte

Al ser un proyecto de código abierto, JBox2D cuenta con el apoyo de una comunidad activa de desarrolladores. Hay documentación disponible, tutoriales y ejemplos que ayudan a los nuevos usuarios a comenzar con la biblioteca. Los foros y plataformas de desarrollo colaborativo, como GitHub, proporcionan lugares para discusión, soporte y contribución al proyecto.

Game Physics: An Analysis of Physics Engines for First-Time Physics Developers

[30] Este artículo se centró en el análisis de motores de física para asistir a desarrolladores novatos en la implementación de física en videojuegos. Se realizó un estudio profundo sobre diferentes motores de física utilizados en la industria, como *Box2D, PhysX y Bullet*, evaluando sus características en términos de usabilidad, conveniencia, flexibilidad, eficiencia, soporte de plataformas y extensibilidad. El trabajo abordó la complejidad de simular física desde un enfoque teórico hasta la implementación práctica en motores de juego, destacando los desafíos como la detección de colisiones y la resolución de contactos.

Se introdujo el motor *Esoteric*, desarrollado por el autor, que pretende facilitar a los desarrolladores la integración de física en aplicaciones, con especial atención en simplificar la arquitectura y optimización para rendimiento. Se discutió en detalle la arquitectura de este motor, incluyendo componentes de modelo, controlador y física, con el objetivo de proporcionar una herramienta elegante y fácil de usar para desarrolladores que abordan la física en aplicaciones por primera vez.

Finalmente, el artículo resaltó las limitaciones actuales y futuras direcciones de trabajo en la detección y simulación de física en videojuegos, señalando la importancia de este estudio como punto de partida para futuras investigaciones en el campo. Este enfoque comprehensivo sobre la selección y evaluación de motores de física sirve como una guía valiosa para desarrolladores principiantes en la materia, promoviendo la innovación y creatividad en el diseño de videojuegos.

Conclusiones y líneas de trabajo futuras

En esta sección se presentan las observaciones y conclusiones derivadas de la finalización del proyecto, así como las posibles direcciones para futuros desarrollos si se decide continuar trabajando en él.

7.1. Conclusiones

Tras concluir el proyecto, hemos extraído las siguientes conclusiones:

Por una parte, hemos desarrollado prácticamente todas las características que se propusieron inicialmente. Se ha creado un motor que, aunque sencillo, ofrece un rendimiento más que suficiente para la máxima carga de trabajo que le podemos pedir desde el frontend. Sin embargo, se ha quedado fuera la que quizá pudiera haber sido una de las características más interesantes de implementar. Se había propuesto que cada cliente que se conectara tuviese asignada una aeronave única que podrían controlar. Esto implicaría un acceso concurrente a los datos al tener, potencialmente, múltiples usuarios modificando las características de su aeronave a la vez. También estaría la parte de cómo implementar un sistema de tal manera que fuera fácil de entender para el usuario. La idea era aplicar el protocolo de búsqueda (que después se reutilizó para los agentes autónomos) a cada aeronave pero, finalmente, debido a las restricciones temporales, se desechó la idea.

De cualquier manera, se ha podido trabajar en un entorno generador de datos al cual podemos acceder desde un cliente ligero, lo cual considero un éxito.

También, ha servido para reforzar lo estudiado en la carrera: Java, sistemas distribuidos, control de calidad, etc; a la vez que se iban incorporando nuevos conceptos: Spring, Spring Boot, React... He podido aplicar conceptos como las metodologías ágiles, el TDD[11], diseño web, contenedores de Docker y ampliar lo conocido gracias al alcance del proyecto.

Además, trabajar en este proyecto me ha brindado una valiosa experiencia inicial en el desarrollo de proyectos a largo plazo, cubriendo desde la fase de planificación hasta su ejecución y mejora continua. He adquirido habilidades para estimar el esfuerzo necesario para diversas tareas y comprender mejor mi capacidad de trabajo dentro de diferentes plazos.

Si bien es cierto que estoy satisfecho con la versión actual del proyecto, hay varias ideas que se han quedado en el tintero y que me gustaría haber implementado si hubiese dispuesto de algo más de tiempo. Dichas ideas son descritas en la siguiente sección.

7.2. Líneas de trabajo futuras

Para expandir el alcance del proyecto e incorporar características adicionales, se proponen las siguientes posibles mejoras:

- **Ampliar la cantidad de cuerpos disponibles y la manera de interactuar con ellos:** me hubiese gustado implementar un sistema en el cual se podría seleccionar cualquier cuerpo y poder cambiar sus características en tiempo real. Esto permitiría a la simulación ser más interactiva y que el usuario tuviese más control sobre ella.

- **Aeronaves únicas por cada usuario:** cada usuario tendría asignada una aeronave única que podría controlar con el teclado y de esta manera interactuar con el resto de usuarios y la simulación. Sin embargo, al estar el cliente y el servidor separados, esto suponía una capa que podría introducir mucha latencia y habría que explorar más detenidamente las estrategias que podríamos implementar para poder ofrecer una experiencia más fluida y en tiempo real.

- **Añadir una gestión de usuarios:** implementar un sistema de almacenamiento y autenticación de usuarios para que cada usuario pudiese acceder de forma privada a sus simulaciones. Esto supondría una

[11] Test Driven Development

importante cantidad de cambios en la arquitectura del sistema, suponiendo poder generar distintas instancias del motor por cada usuario y que cada una funcionase a pleno rendimiento.

- **Añadir un editor el cual te permita crear y guardar escenarios de simulación:** como muchos motores ofrecen, crear un editor el cual permitiese al usuario diseñar un entorno con unas características determinadas a su gusto y poder guardarlo para ser reutilizado.
- **Implementar algoritmos neuronales para que las aeronaves mejoren sus comportamientos:** añadir la capacidad de 'sobrevivir' (y por ende, morir) a las aeronaves y que estas vayan mejorando los parámetros para sus rutinas de evitar colisiones.
- **Paralelismo y concurrencia:** algo que he aprendido en el poco tiempo que llevo trabajando como desarrollador de software, es que el paralelismo, multithreading, la concurrencia, son cuchillas de doble filo; extremadamente potentes si se implementan correctamente, pero potencialmente detrimentales en el caso contrario. De todos modos, en la arquitectura propuesta, se han dividido ciertas tareas en procesos (o, cómo nosotros los hemos apodado, servicios) que llevan a cabo el cálculo de ciertas fuerzas que se aplican a los cuerpos. Estos servicios, por definición, ya están divididos en una estructura que es compatible con la concurrencia (*Concurrency is not parallelism, Rob Pike*); luego, tras estudiar detenidamente cual es la mejor manera (o mejor dicho, la más adecuada para el caso) de implementar un nivel de paralelismo que pueda mejorar el rendimiento del motor, podríamos invertir tiempo en introducir dicho paralelismo y comprobar que mejoras supone.

Bibliografía

[1] Create React App. Create react app: Set up a modern web app by running one command. https://create-react-app.dev/docs/getting-started#npm-start-or-yarn-start. [Internet].

[2] Axosoft. Gitkraken. https://www.gitkraken.com/. [Internet].

[3] William Bittle. dyn4j. v5.0.1; 31-diciembre-2022].

[4] Code Climate. Code climate quality: Engineering insights for software teams. https://codeclimate.com/quality/. [Internet].

[5] Codacy. Codacy: Automatically identify issues through static code analysis. https://www.codacy.com/. [Internet].

[6] Software Freedom Conservancy. Git. https://git-scm.com/. [Internet].

[7] Processing Foundation. Processing. Internet; descargado 30-enero-2024].

[8] Inc. GitHub. Github: Where the world builds software. https://github.com/. [Internet].

[9] Jason Gregory. *Game Engine Architecture*. CRC Press, 2018.

[10] JetBrains. Intellij idea ultimate. https://www.jetbrains.com/idea/. [Internet].

[11] LocationTech. Jts core. https://locationtech.github.io/jts/. [Internet].

[12] Project Lombok. Project lombok. https://projectlombok.org/. [Internet].

[13] Microsoft. Visual studio code. https://code.visualstudio.com/. [Internet].

[14] Daniel Murphy. Jbox2d, 2014. v5.0.1; 31-diciembre-2022].

[15] p5.js. p5.js. https://p5js.org/. [Internet].

[16] React. React – a javascript library for building user interfaces. https://reactjs.org/. [Internet].

[17] React. Reactdom – react package for working with the dom. https://reactjs.org/docs/react-dom.html. [Internet].

[18] Scrum.org. What is scrum? https://www.scrum.org/resources/what-is-scrum. [Internet].

[19] Daniel Shiffman. *The Nature of Code: Simulating Natural Systems with Processing*. Shiffman, Daniel, 2012.

[20] SockJS. Sockjs client. https://github.com/sockjs/sockjs-client. [Internet].

[21] SockJS. Sockjs-client. https://github.com/sockjs/sockjs-client. [Internet].

[22] Spring. Spring boot configuration processor. https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-annotation-processor. [Internet].

[23] Spring. Spring boot devtools. https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.devtools. [Internet].

[24] Spring. Spring boot starter data jpa. https://spring.io/projects/spring-data-jpa. [Internet].

[25] Spring. Spring boot starter test. https://spring.io/guides/gs/testing/. [Internet].

[26] Spring. Spring boot starter web. https://spring.io/projects/spring-boot. [Internet].

[27] Spring. Spring boot starter websocket. https://spring.io/guides/gs/messaging-stomp-websocket/. [Internet].

[28] STOMP. Stomp over websocket. https://stomp.github.io/. [Internet].

[29] STOMP.js. Stomp.js. https://stomp-js.github.io/. [Internet].

[30] Russell M. Templet. Game physics: An analysis of physics engines for first-time physics developers. *CALIFORNIA STATE UNIVERSITY, NORTHRIDGE*, 2020.