Universidad Politécnica
de Madrid

Escuela Técnica Superior de

Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

MSiM - Entorno de simulación para sistemas multi-robot

Autor: Miguel Morales Galisteo
Tutor: Javier de Lope Asiaín

Madrid, enero de 2025

Resumen

La robótica es una pieza fundamental en el desarrollo y evolución del mundo moderno. La transformación que han experimentado innumerables sectores aumentando la precisión, productividad y eficiencia gracias a la industria robótica es una realidad que hemos vivido en los últimos cincuenta años. La industria manufacturera actual es prácticamente inconcebible sin la productividad que aporta la robótica, sectores como el de la salud se ha visto revolucionado por robótica que es capaz de aportar en procedimientos quirúrgicos y de diagnóstico. En la industria de la exploración espacial o en la submarina, son los robots quienes consiguen extender los límites que los humanos no podemos alcanzar, pudiendo llevar a cabo misiones en entornos del todo inaccesibles para nosotros.

De esta industria robótica nacen en los últimos treinta años los sistemas multi-robot como una evolución más en la robótica. Tareas que para un sólo robot pueden resultar demasiado complejas de lograr. Esto se fundamenta en el desglose de las tareas en otras más pequeñas para que los robots puedan lograr un fin mayor realizando las tareas de forma coordinada. Ya podemos ver esto como por ejemplo con los drones que trabajan coordinadamente para realizar búsquedas o rescates en gestión de desastres o también con los drones que conforman los espectáculos de luces que vamos viendo en las celebraciones de los últimos años sustituyendo a los tradicionales fuegos artificiales.

Los algoritmos que manejan estas flotas de robots pueden llegar a ser extremadamente complejos y requieren de un entorno en el cual puedan desarrollarse y probarse. Justo ese es el objetivo que tenemos con este proyecto.

Concretamente, en este trabajo vamos a desarrollar un entorno de simulación para sistemas multi-robot en el que modelaremos el movimiento de robots estilo E-puck y manejaremos las interacciones que tengan con el resto de los elementos del entorno en la simulación. Es interesante la conceptualización de este entorno y del planteamiento del proyecto con la integración de un motor de físicas con el que simular el movimiento de todos los cuerpos de nuestro entorno y así conseguir una herramienta en la que poder probar y desarrollar algoritmos de manejo de sistemas multi-robot.

Abstract

Robotics is an integral factor in the development and evolution of the modern world we live at. The transformation experienced by endless industries increasing precision, productivity and efficiency because of the robotics industry is a reality that has changed the world over the last fifty years. Today's manufacturing industry is almost inconceivable without the productivity that robotics provides, fields as healthcare have experienced absolute revolution with robotics that are able to contribute to surgical procedures and diagnostics. In the space or underwater exploration fields, robots are the ones that have pushed the limits that we humans cannot achieve, conducting missions in environments inaccessible to humans.

In the last thirty years, multi-robots have emerged as a further evolution in robotics. These systems address tasks that may be too big or complex for a single robot to accomplish. The approach is based on breaking down tasks into smaller tasks, more manageable ones, able to accomplish by the robots composing the system, enabling them to achieve a greater goal by working co-ordinately. We can already see this in several examples as the drones that work together in search and rescue operations during disasters or at the recent times at celebrations replacing the traditional fireworks with drones performing light shows.

The algorithms that manage these fleet of robots can be extremely complex and require an environment where they can be developed and assessed. This is exactly the goal of our project.

Specifically in this project we will develop a simulation environment for multi-robot systems, modelling the movement of E-puck style robots and managing their interactions with other elements in the simulation environment. The conceptualization of this environment and the project approach is particularly interesting due to the integration of a physics engine to simulate the movement of all the objects in our environment. This will result in a tool capable of testing and developing algorithms for managing multi-robot systems.

Tabla de contenidos

1. Introducción

El todo es mayor que la suma de sus partes.

En la naturaleza existen numerosos ejemplos de sistemas complejos formados por individuos simples y autónomos que consiguen trabajar de una manera conjunta y organizada para conseguir un fin que, de forma individual no podrían lograr.

Las hormigas se coordinan a la hora de buscar alimento, al construir y defender sus colonias. Las colonias de hormigas pueden encabezarse por una o varias reinas, donde cada hormiga tiene una función específica dentro del todo. Las hormigas se comunican entre sí y cooperan. De una forma eficiente y estrechamente unidas, para poder lograr cometidos que de forma individual no serían capaces de alcanzar.

Es este tipo de comportamiento, en el que un grupo de individuos autónomos trabajan de manera organizada y armónica, el que conforma uno de los pilares de los sistemas multi-robot. En estos sistemas se llevan a cabo distintos procesos entre diferentes elementos que comparten un espacio de trabajo común.

Los robots que conforman los sistemas multi-robot trabajan de forma organizada para realizar tareas complejas que, de forma individual no serían posibles de realizar por un único robot aún más capacitado. El concepto que hay detrás es el desgranar las tareas en tareas más pequeñas que puedan ser asignadas a los robots para que individualmente las puedan cumplir con éxito, consiguiendo de esta manera resolver soluciones a problemas más complejos.

El construir un sistema multi-robot resulta ser más rentable que construir un único robot que sea capaz de cumplir con todas las capacidades y funcionalidades que podemos. Resulta mucho más costoso. Esta solución también resulta ser más tolerante a fallos al ser sistemas por lo general descentralizados, distribuidos y redundantes. Esto hace que la fiabilidad y la robustez del sistema sea mayor. [1][2]

1.1 Definición del trabajo

En este trabajo vamos a desarrollar un software que cumpla con la función de ser un entorno de simulación en el que se puedan modelar cuerpos dinámicos y movimiento con la finalidad de que pueda ser empleado para la simulación de sistemas multi-robot. Para modelar el movimiento y las interacciones entre los elementos de nuestro entorno de simulación vamos a utilizar motores de físicas.

Este entorno de simulación debe ser eficiente para trabajar a tiempo de CPU. Buscamos que sea lo más optimizado posible por lo que va a ser requerido codificar en programas como C, C++ o Python, además del uso de estas librerías y motores de físicas ya desarrollados y optimizados.

Queremos poder simular largos períodos de tiempo de ejecución en un tiempo mucho menor, por lo que no vamos a buscar que la simulación sea a tiempo real. Aparte de los datos que nuestro programa debe calcular con el motor de físicas que empleemos, debemos de tener también una salida gráfica para de forma visual comprobar el movimiento de los elementos simulados.

También debemos notar que, pese a que en las pruebas no vayamos a tener poblaciones excesivamente grandes, debemos tener capacidad para escalar el proyecto a mayores capacidades, por lo que debemos fijarnos en la optimización del sistema y plantear futuras modificaciones futuras si pretendemos escalar el alcance.

El enfoque propuesto consiste en cinco etapas: investigación, diseño, desarrollo, pruebas y análisis.

En la investigación nos centramos en obtener conocimiento sobre la materia de los entornos de simulación con motores de físicas y librerías existentes que podamos aplicar en nuestro trabajo. Ahondaremos más sobre estos temas en el capítulo 2 sobre el Estado del Arte.

En la siguiente fase de diseño entramos a plantear cómo debemos realizar nuestra solución, identificando los requisitos generales de nuestro sistema y estableciendo una arquitectura para dar paso a la siguiente fase de desarrollo.

En la fase de desarrollo del proyecto debemos crear primero el entorno de simulación y su integración con el motor de físicas, para posteriormente desarrollar e implementar los elementos del entorno, los cuales detallaremos en el capítulo 5.1.

Tras el desarrollo de los elementos del entorno y la lógica de su comportamiento, así como del manejo de eventos debemos configurar la salida gráfica del movimiento de los elementos. Por último, debemos configurar un manejo de los entornos de simulación para poder realizar distintas simulaciones en el sistema.

En el despliegue del entorno de simulación debemos realizar las pruebas pertinentes para comprobar que se cumplen todos los requisitos definidos para el proyecto, así como comprender el proceso de ejecución de nuestro sistema y poder identificar puntos de mejora para futuras versiones del simulador.

Tras todo este proceso de pruebas debemos analizar los resultados obtenidos y extraer unas conclusiones del proyecto, así como evaluar el impacto del mismo y los puntos de mejora para futuras versiones que hemos identificado.

1.2 Objetivos

Podemos definir los objetivos de este proyecto de la siguiente manera:

- Revisión bibliográfica de entornos de simulación y motores de físicas
- Definir los requisitos del proyecto
- Desarrollo del entorno de simulación
- Integración del motor de físicas
- Modelizar las interacciones dinámicas entre los elementos del entorno de simulación
- Configuración de una salida gráfica
- Maximizar la eficiencia de las soluciones
- Creación, guardado y carga de distintos entornos de simulación
- Realizar pruebas de todo el sistema
- Realizar análisis y conclusiones

2. Estado del arte

En este capítulo recopilamos el estado actual de las tecnologías existentes en el ámbito que abarca este proyecto, de entornos de simulación y motores de físicas. Nuestro objetivo es recopilar la información y el conocimiento obtenido tras la fase de investigación para poder determinar cuál es la manera adecuada de desarrollar el proyecto.

Antes de iniciar a recopilar información y de lanzarnos a investigar, debemos tener claro qué debemos buscar. En un punto inicial que suponga una primera toma de contacto con la materia es natural preguntarse, ¿qué es un entorno de simulación? ¿Qué hace un motor de físicas? ¿Qué es un sistema multi-robot? ¿Cómo utilizar todo esto en conjunto? Estas preguntas son básicas para la comprensión de este proyecto. Y debemos aclararlas antes de continuar.

Un entorno de simulación es un software que nos permite replicar la realidad. Un entorno físico. Nos permite probar algoritmos, realizar experimentos y cálculos sin las limitaciones, costes o riesgos de la realidad física. Pero el entorno de simulación no es la realidad. Trata de simularla, pero requiere de mucho trabajo de diseño y configuración para alcanzar, en distintos niveles de precisión todas las capas, objetos, entidades, estructuras y escenarios que pueden darse en la realidad.

Traído a este proyecto, nuestro entorno de simulación va a ser el mundo en el que van a existir nuestros robots. Nuestro proyecto va a ser el laboratorio en el que vamos a hacer nuestros experimentos, vamos a probar el funcionamiento de nuestros robots y vamos a configurar cómo se mueven y cómo interactúan los robots con los elementos del entorno. Esto va a ser gracias al motor de físicas.

Un motor de físicas puede reproducir las condiciones físicas que queramos que rijan dentro de nuestra simulación. Podemos controlar la gravedad, la fricción, densidad, flexibilidad, etc. No sólo reproduce las propiedades del entorno, sino que también se encarga de los cálculos relativos a la cinemática de los cuerpos dinámicos, así como del manejo y detección de eventos, que en este caso serán colisiones.

Los robots serán los elementos claves en este proyecto, puesto que son los que cuentan con un movimiento propio y debido a ellos se producirán los eventos en nuestro sistema. Vamos a basarnos en los robots estilo E-puck [3] para representar a los robots de nuestro entorno de simulación.

[Imagen 1] Fotografía de un robot E-puck

El E-puck es un robot ampliamente utilizado de forma educativa y en la investigación diseñado en la EPFL suiza que basa su movimiento en el giro de dos ruedas que contiene una a cada lado. De esta manera, conocemos las entradas que vamos a recibir para el movimiento de nuestro robot: *lWheel* y *rWheel*, siendo cada una la fuerza (positiva, negativa o nula) de cada rueda.

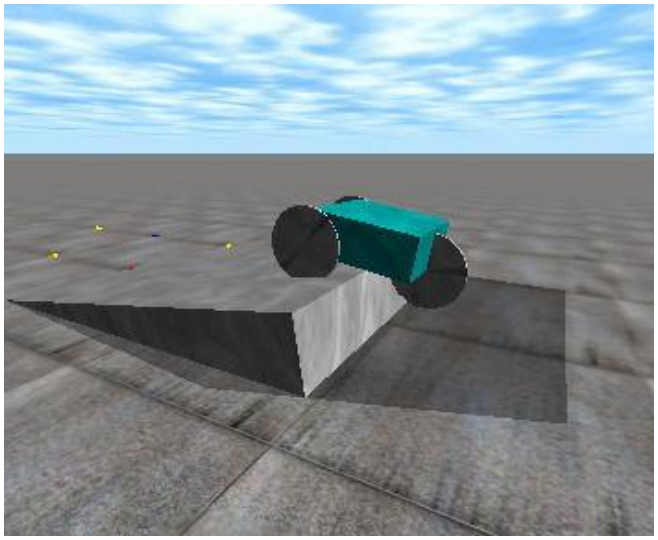
2.1 Motores de físicas y simulación

La gran parte de este proyecto radica en la simulación. El entorno de simulación es aquel que tenemos que construir y diseñar en base a motores de físicas y librerías optimizadas ya existentes. En este capítulo recopilamos la búsqueda de información y conocimiento en aras de encontrar la tecnología que mejor se pueda adaptar a nuestro proyecto.

ODE

Open Dynamics Engine [4] es una librería de código abierto de alto rendimiento para la simulación tridimensional de entidades rígidas dinámicas. Es una librería que cuenta con una API para realizar llamadas desde C/C++.

Este motor de física lanzado en 2001 sigue recibiendo actualizaciones en la actualidad y es especialmente particular en proyectos de desarrollo de videojuegos y aplicaciones gráficas, aunque su uso ha disminuido con la aparición de motores como PyBullet y NVIDIA PhysX. Es también popular en simulaciones de robótica en investigaciones de locomoción robótica para estudiar la estabilidad y equilibrio de robots como por ejemplo el robot humanoide de 2010 iCub del Instituto Italiano de Tecnología que emplea Gazebo [5], un simulador tridimensional basado en ODE.

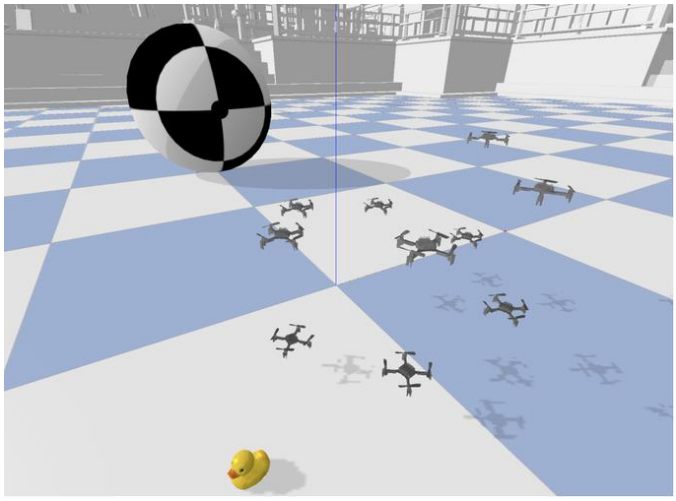


[Imagen 2] Snapshot de una simulación del movimiento de una entidad rígida dinámica utilizando ODE

ODE es una tecnología ideal para nuestro proyecto debido a su capacidad para manejar simulaciones físicas complejas donde los robots interactúan entre sí y con el entorno, con las dinámicas de movimiento y modelaje de colisiones que podemos necesitar para nuestro proyecto.

PyBullet

PyBullet [6] es un módulo fácilmente integrable en cualquier proyecto en Python que emplea el motor de físicas Bullet, un MF de código abierto bajo licencia zlib que simula la detección de colisiones y la simulación dinámica de entidades rígidas y blandas.



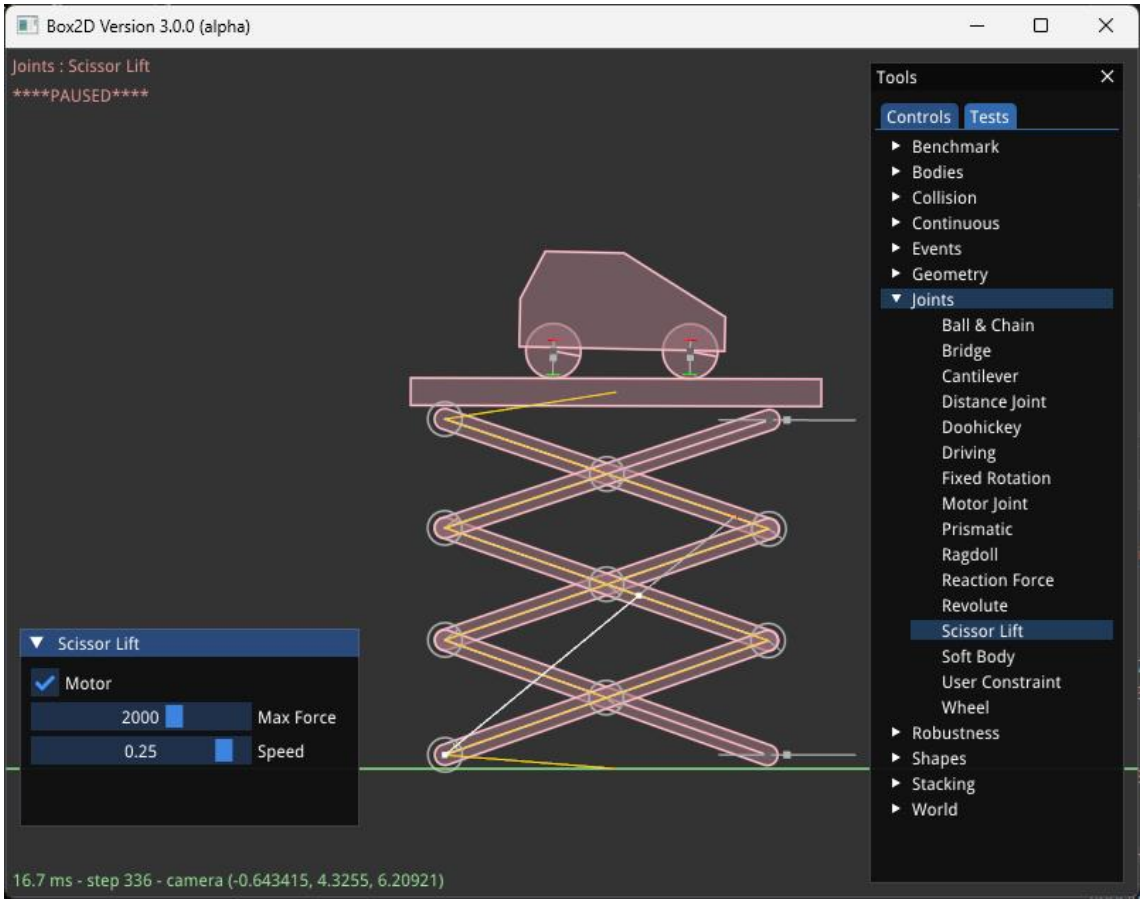
[Imagen 3] Snapshot de una simulación de un SMR empleando PyBullet

PyBullet es un software de uso ampliamente extendido en la industria de la animación tridimensional, siendo usado en numerosas películas de animación de éxito como referencian en su página web, así como también es utilizado en la industria de los videojuegos, siendo utilizado por RAGE (Rockstar Advanced Game Engine) en videojuegos de éxito como por ejemplo Grand Theft Auto V.

Box2D

Box2D [7] es una librería de simulación de entidades rígidas de código abierto que implementa un motor de físicas bidimensional utilizada popularmente en la industria del videojuego. Está desarrollado en C++ por Erin Catto y publicado en 2006. Desde entonces ha evolucionado con mejoras en la precisión de simulaciones y optimizando el rendimiento.

Box2D cuenta con una interfaz cinemática a partir de la v2.2.1. La simplicidad y eficiencia de Box2D la hace una opción idónea para nuestro proyecto. Cuenta con una documentación muy accesible para cualquier usuario que quiera desarrollar proyectos implementándolo.



[Imagen 5] Snapshot de una simulación empleando

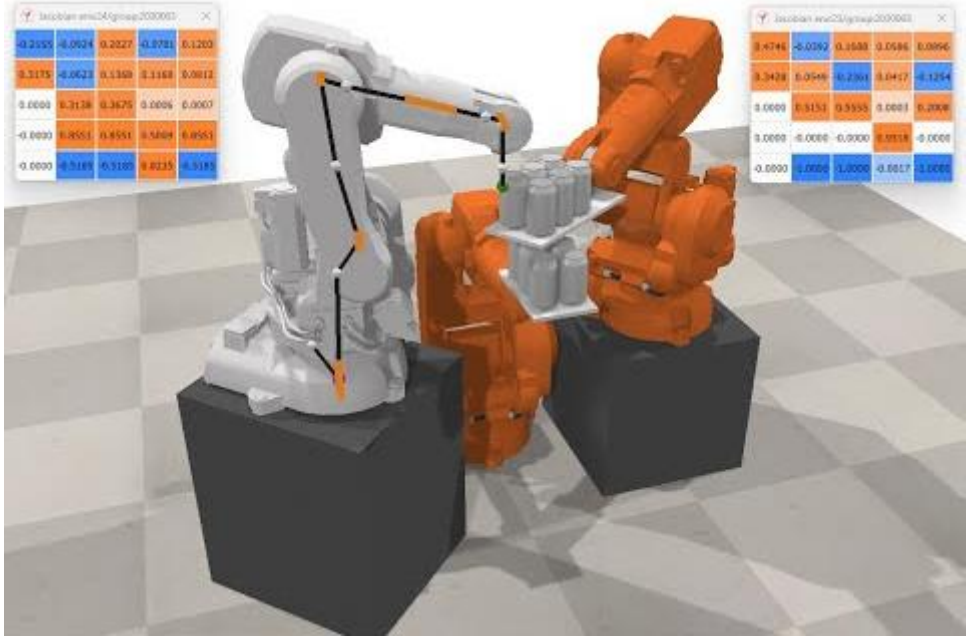
la interfaz cinemática de Box2D

También existe una versión de Box2D llamada JBox2D para su uso con Java en lugar de C++. JBox2D integra Box2D con el MF de Google LiquidFun, un MF bidimensional que simula entidades rígidas y fluidos basado en el propio Box2D.

En Box2D podemos crear muchas instancias que actúen como mundos (entornos) de forma independiente y avanzar el tiempo en cada uno de ellos en cada simulación. Tiene un alto rendimiento que podemos revisar para poder optimizar en tiempo de computación para nuestro ES, buscando el paralelizar las simulaciones si la cantidad de mundos creados en Box2D nos obliga a ello en aras del tiempo de computación.

CoppeliaSim

CoppeliaSim [8] no es un motor de físicas sino un simulador creado por Coppelia Robotics en 2013 lanzó este software de simulación robótica bajo el nombre V-REP (Virtual Robot Experimentation Platform). Es en 2019 cuando pasa a llamarse CoppeliaSim. Es una herramienta de simulación tridimensional avanzada de robótica. Se usa en la industria y en el ámbito académico para proyectos de prototipado y pruebas en simulación paso previo a la implementación en hardware.



[Imagen 6] Snapshot de una simulación de un SMR empleando CoppeliaSim

Emplea varios motores de físicas como ODE y es compatible con lenguajes como Python, Matlab, Java, Lua y C/C++, permitiendo controlar las simulaciones mediante scripts programados por el usuario.

Esta opción es también muy adecuada para nuestro proyecto dado que pese a no ser de código abierto como pueden ser otras opciones, cuenta con una licencia educativa que nos permite acceder a este software de simulación robótica profesional.

Webots

Webots [9], como CoppeliaSim, no es un motor de físicas, sino que es un simulador de robótica de código abierto en tres dimensiones creado en 1996 en el instituto federal suizo de tecnología EPFL de Laussane. El mismo instituto en el que se lanzó años después el robot en el que vamos a basarnos, el E-puck. Emplea también ODE como motor de físicas y OpenGL para el renderizado. Es ampliamente utilizado para la simulación de toda clase de robots y entornos, desde drones voladores, robots subacuáticos o coches circulando por una carretera.

3. Box2D

Por lo accesible que resulta Box2D como solución respecto al resto de tecnologías que hemos expuesto en el 2.1, va a ser la tecnología que implementemos en nuestro proyecto.

En este capítulo nos dedicaremos a explicar cómo funciona, con qué elementos cuenta y cómo se ejecutan. Para ello nos basaremos en la documentación oficial de Box2D, así como en distintos repositorios de la comunidad de Box2D, en particular iforce2d [10].

Box2D va a ejecutar un entorno de simulación al que va a llamar mundo. Dentro de este mundo podremos inicializar y programar el movimiento de distintos elementos, distinguiendo entre los siguientes:

- **Estáticos:** Su masa es infinita y por ende son inamovibles ante las colisiones que puedan suceder en el entorno. No se les puede programar una velocidad o fuerza, ya que por su propia naturaleza estos elementos no se van a mover nunca. Los elementos dinámicos son los únicos que pueden interactuar con ellos.

- **Cinemáticos:** Son similares a los elementos estáticos, pero con la diferencia de que los elementos cinemáticos sí que pueden ser programados con una velocidad o fuerza. Igualmente, sólo interactúan siendo inamovibles con los elementos dinámicos ante colisiones con éstos.

  En Box2D no se permite la superposición de elementos, así que, en el caso de una colisión con un elemento estático, el movimiento del elemento cinemático se mostrará limitado por él.

  Éstos son los únicos elementos de Box2D que no vamos a integrar en nuestro sistema. Puede ser un punto a incluir en futuras versiones.

- **Dinámicos:** Son los elementos que más juego nos van a dar en nuestro entorno de simulación. Son los elementos que se van a mover por el mundo y van a interactúan con el resto de elementos estáticos, cinemáticos y otros elementos dinámicos que se encuentren.

El planteamiento conceptual de los elementos en Box2D puede resultar algo contraintuitivo, ya que a estos elementos los define como *bodies*, siendo por sí mismos algo invisibles e intangibles. Conforman una entidad que cuenta con las propiedades intangibles de masa, velocidad, velocidad angular, ubicación y ángulo.

Para que estos elementos se acaben acercando más a la concepción de elemento que nos podemos imaginar en un entorno de simulación como éste, para cada *body* vamos a necesitar un *fixture*.

Con esto definiremos el tamaño, la forma y las propiedades del material del elemento (restitución, fricción y densidad).

Las formas que podemos programar para nuestros elementos son o bien círculos o bien polígonos. Estos polígonos pueden ser tanto un cuadrado, un rectángulo o cualquier polígono convexo dispuesto a nuestra confección en base a sus vértices.

Box2D cuenta con código propio en el que nos muestra un banco de pruebas ya cargadas que son útiles para entender de forma visual el funcionamiento del motor de físicas. Vamos a utilizarlo para programar nuestro propio banco de pruebas y demostrar cómo funcionan estos elementos que hemos explicado.

En el siguiente fragmento codificamos un suelo, un techo y dos paredes, para delimitar nuestro espacio de pruebas para la simulación que vamos a realizar:

```
b2BodyDef bodyDef;
bodyDef.type = b2_staticBody;
bodyDef.position.Set(0, 0);
b2Body* floor = m_world->CreateBody(&bodyDef);

b2PolygonShape boxShape;
boxShape.SetAsBox(100,5);

b2FixtureDef fixtureDef;
fixtureDef.shape = &boxShape;

floor->CreateFixture(&fixtureDef);
```

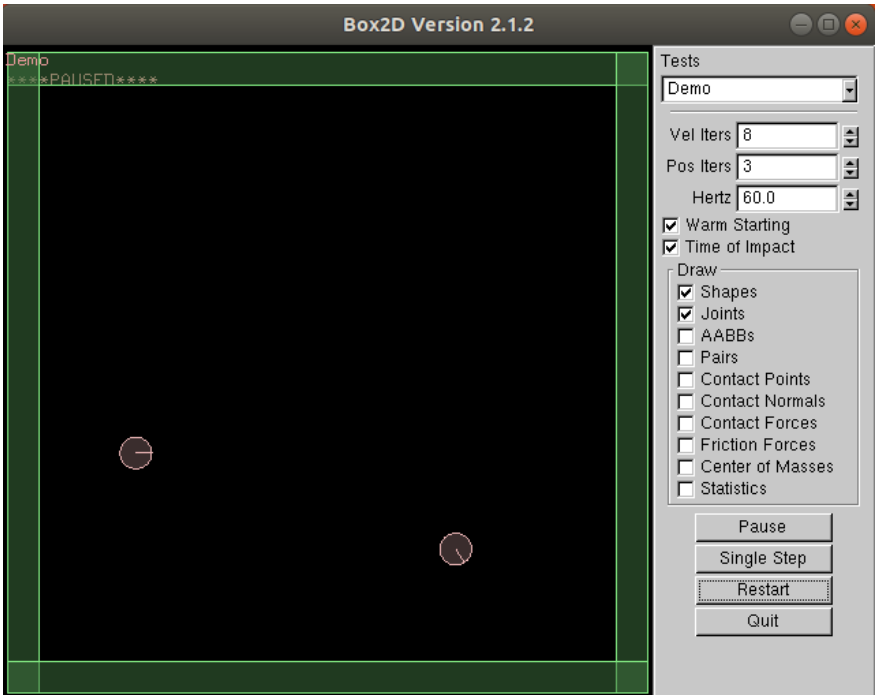
Codificación de un elemento suelo

Replicando este código configuramos el resto de elementos estableciendo los límites de nuestra demostración. Procedemos a introducir un par de elementos dinámicos con los que probar el funcionamiento de Box2D.

```
bodyDef.type = b2_dynamicBody;
bodyDef.position.Set(-60,70);
bodyDef.angle = 0;
b2Body* bot1 = m_world->CreateBody(&bodyDef);

b2CircleShape circleShape;
fixtureDef.shape = &circleShape;
circleShape.m_radius = 5;

fixtureDef.density = 1;
fixtureDef.restitution = 1;
fixtureDef.friction = 1;
bot1->CreateFixture(&fixtureDef);
```

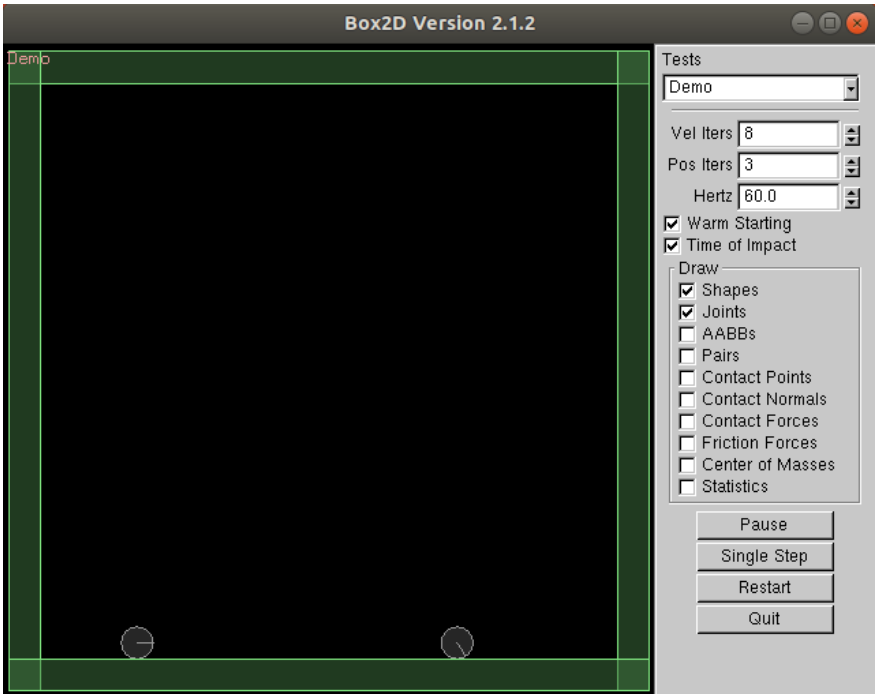
Codificación de un elemento dinámico

Replicamos el código configurando un segundo elemento dinámico y ejecutamos este código. Lo que nos aparecerá en la ventana gráfica de Box2D será lo que se puede apreciar en la imagen 7.



[Imagen 7] Snapshot de la demostración pausada en Box2D

Podemos apreciar en la esquina superior izquierda cómo la ejecución se encuentra pausada. Esto lo hemos hecho de forma intencionada, puesto que si no la detuviésemos lo que veríamos sería lo que muestra la imagen 8.



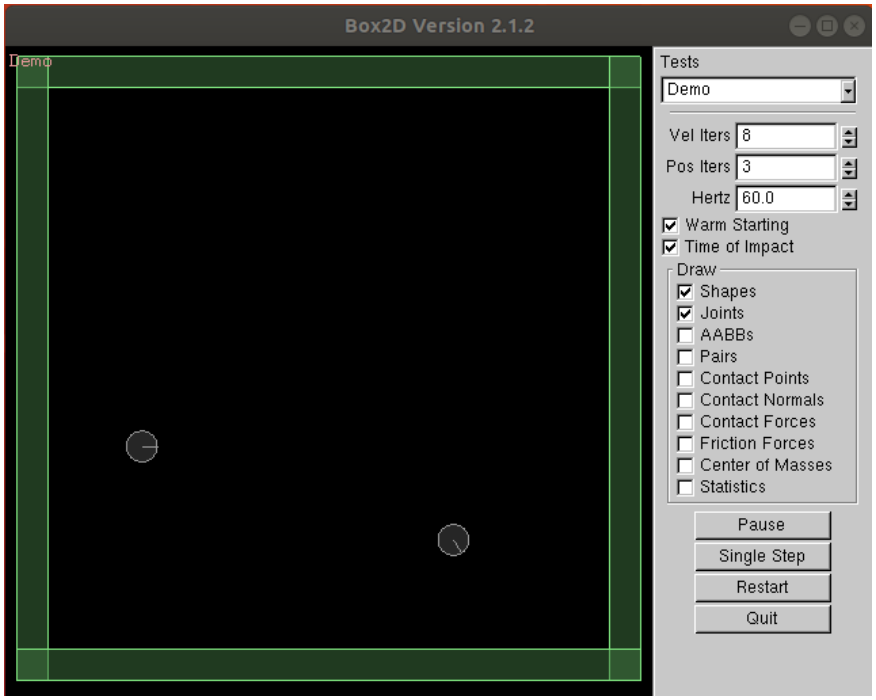
[Imagen 8] Snapshot de la demostración en Box2D

Los dos elementos dinámicos que hemos introducido en la simulación cambian de posición durante unos segundos de simulación y se quedan inmóviles junto al elemento estático suelo que hemos programado.

La simulación en Box2D es bidimensional. Al configurar una demostración en el banco de pruebas proporcionado por Box2D lo que se simula tiene una vista frontal. Por la gravedad que existe en el mundo de Box2D. Para poder pasar a una vista cenital, establecemos la gravedad del mundo a cero.

```
m_world->SetGravity(b2Vec2(0,0));
```

Esta gravedad afectará a todos los elementos dinámicos, ya que ni a los cinemáticos ni a los estáticos una gravedad u otra les afecta de ninguna manera. Ahora pues, nuestra demostración quedará exactamente igual que la imagen 7 sin necesidad de pausarla.



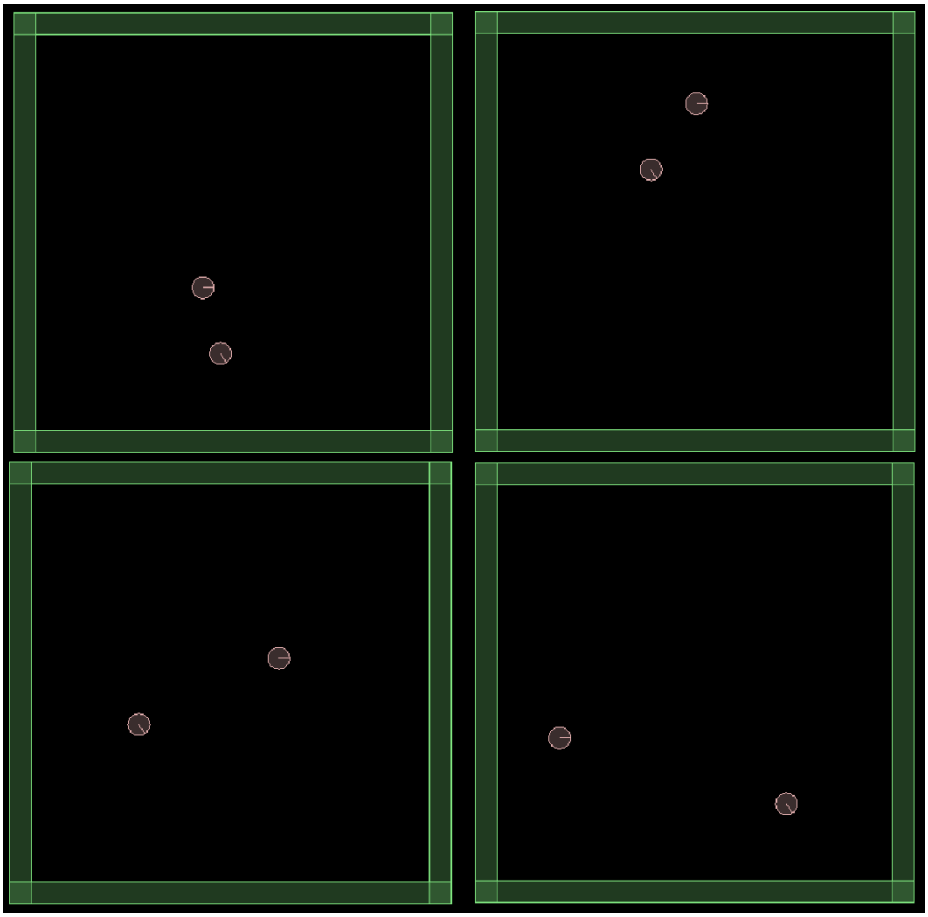
[Imagen 9] Snapshot de la demostración en Box2D

Podemos apreciar una ligera diferencia entre la imagen 7 y la imagen 9. Los colores de los dos elementos dinámicos son diferentes en una y otra simulación. Esto se explica por la optimización que emplea Box2D en su motor de físicas. Los elementos dinámicos que están en reposo, es decir, sin un movimiento y sin fuerzas externas que interactúen con ellos, pueden entrar en un estado computacional de reposo para optimizar la simulación. Eso de forma visual se traduce en un cambio de color.

Si queremos o no que esta optimización se lleve a cabo debemos configurarlo a la hora de inicializar el mundo. En la versión de Box2D que vamos a utilizar (v2.1.2) [11] no se puede modificar de forma posterior, aunque sí que podemos deshabilitar el estado computacional de reposo para elementos dinámicos individualmente. Al inicializar un mundo pues, lo configuraremos de la siguiente manera, donde pasamos el parámetro de la gravedad como un vector con sus fuerzas en el eje x e y así como la habilitación de estados computacionales de reposo.

```
b2World* m_world = new b2World(b2Vec2(0,0), true);
```

Continuamos ahora programando movimiento para ambos elementos dinámicos que hemos creado antes en nuestro código: *bot1* y *bot2*. Sacamos la declaración de ambos elementos del constructor para poder llamarlos desde la función Step de Box2D que hace que avance el tiempo. Programamos un bucle muy simple con la función *SetLinearVelocity*. Ejecutamos y vemos el movimiento que realiza.

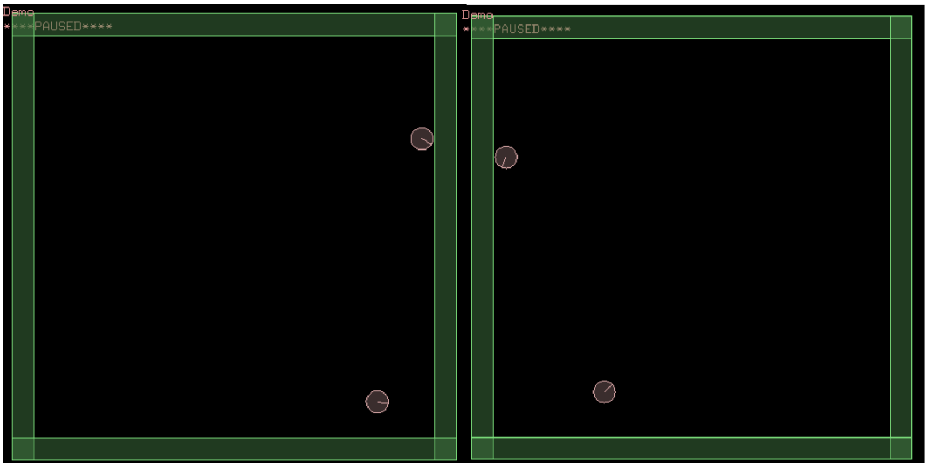


[Imagen 10] Snapshot del movimiento en la demostración en Box2D

En las *snapshot* que vemos en la imagen 10 los elementos estáticos están realizando con su codificación un movimiento que traza una forma de cuadrado. No colisionan en ningún momento entre ellos ni con otro elemento de la simulación.

Es en las colisiones en donde radica el quid de este motor de físicas. Recordando que el objetivo de este proyecto es realizar un entorno de simulación para sistemas multi-robot, las colisiones juegan un papel fundamental en las implementaciones que se puedan realizar en nuestro entorno de simulación. Debemos ser capaces de detectar las colisiones que suceden en nuestro entorno de simulación y poder conocer cuándo y dónde se producen, así como también identificar los elementos que colisionan entre sí.

Box2D también cuenta con funcionalidades para poder detectar estas colisiones que implementamos en nuestro código de demostración. Entraremos a detalle con esa funcionalidad a la hora de desarrollar nuestro proyecto. Ahora programaremos una manera de mostrar por consola las colisiones que sucedan en nuestra simulación.



[Imagen 11] Snapshot de los contactos en la demostración en Box2D

Tras haber alterado el movimiento de los elementos dinámicos y este par de *snapshots* la salida por consola que nos quedaría de la siguiente manera mostrando los contactos que suceden en nuestra simulación.

```
paco@paco-VirtualBox: ~/tests/Box2D_v2.1.2/Box2D/Build/Testbed
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
paco@paco-VirtualBox:~/tests/Box2D_v2.1.2/Box2D/Build/Testbed$ ./Testbed
Step: 12 - Contact betweeen BodyA(0x55bed1c5aef0) and BodyB(0x55bed1c5b070)
at position (-89.9928, 76.3346)
Step: 13 - Contact betweeen BodyA(0x55bed1c5aef0) and BodyB(0x55bed1c5b070)
at position (-89.2658, 76.6619)
Step: 22 - Contact betweeen BodyA(0x55bed1c5ae30) and BodyB(0x55bed1c5b130)
at position (89.9928, 37.9068)
Step: 23 - Contact betweeen BodyA(0x55bed1c5ae30) and BodyB(0x55bed1c5b130)
at position (89.4933, 37.8448)
Step: 129 - Contact betweeen BodyA(0x55bed1c5ae30) and BodyB(0x55bed1c5b070)
 at position (89.9928, 114.602)
Step: 130 - Contact betweeen BodyA(0x55bed1c5ae30) and BodyB(0x55bed1c5b070)
 at position (89.4475, 114.71)
Step: 193 - Contact betweeen BodyA(0x55bed1c5aef0) and BodyB(0x55bed1c5b130)
 at position (-89.9928, 27.3681)
Step: 194 - Contact betweeen BodyA(0x55bed1c5aef0) and BodyB(0x55bed1c5b130)
 at position (-89.7431, 27.3475)
```

[Imagen 12] Snapshot de la salida por consola detectando colisiones

Apreciamos en la imagen 12 que los contactos aparecen “duplicados”. Básicamente aparece cada contacto dos veces, con un ligero cambio en las posiciones, así como con el *nStep* consecutivo. Esto sucede porque los *Step* pueden durar más de un *Step*, ya que cada uno en esta ejecución es de 1 Hz.

Así pues, y tras haber explicado el funcionamiento de Box2D con esta demostración, avanzamos en nuestro proyecto hacia la planificación, definición de requisitos y de la arquitectura teniendo en cuenta las particularidades de la tecnología que vamos a desarrollar.

4. Diseño

Diseñamos nuestro proyecto como una librería que va a emplear el motor de físicas Box2D para ejecutar sus simulaciones. Esta librería está orientada a ser utilizada por un usuario que quiera realizar sus propias simulaciones e implantaciones de sistemas multi-robot. Nuestro proyecto se enfoca en crear el entorno de simulación necesario para que esto se lleve a cabo.

Box2D puede resultar algo contraintuitivo de utilizar en ciertas ocasiones. Necesitamos un entorno de simulación que sirva de forma personalizada a los requisitos de nuestro proyecto y que facilite la interacción del usuario la programación y configuración de sus simulaciones frente a lo que nos encontramos con el motor de búsqueda.

Además, queremos que esta librería también pueda gestionar los datos de las simulaciones, pudiendo almacenarlas y mostrarlas cuando el usuario se lo indique.

No pretendemos que las simulaciones se realicen a tiempo real. Si el usuario quisiera simular cientos de horas tendría que esperar todo ese tiempo para que la simulación terminara y pudiera comprobar los resultados. Requerimos que las simulaciones funcionen a tiempo de CPU o lo más cercano a ello. Por ello, prima la optimización de la solución.

En nuestra solución el entorno de simulación debe de contar con elementos estáticos, elementos dinámicos y elementos dinámicos con movimiento propio que hagan las funciones de robots. Debemos poder configurar la inicialización de tamaño y posición de los elementos, así como poder obtener durante la ejecución los datos relativos a ellos que puedan variar durante la simulación.

En el caso de los robots debemos poder programar su movimiento. Y como va a haber movimiento, es posible que haya colisiones, por lo que debemos también conocer si hay colisiones, dónde, cuándo y entre qué elementos se producen.

Para programar el movimiento debemos atender a las ecuaciones del modelo cinemático de un robot diferencial:

$$F_X = \frac{F_L + F_R}{2} * \cos\theta$$

$$F_Y = \frac{F_L + F_R}{2} * \sin\theta$$

$$\omega = \frac{F_R - F_L}{d}$$

Donde:

- F_X es la fuerza a imprimir en el eje x
- F_Y es la fuerza a imprimir en el eje y
- ω es la velocidad angular resultado
- F_L es la fuerza recibida de la rueda izquierda
- F_R es la fuerza recibida de la rueda derecha
- θ es el ángulo al que apunta el frente del e-puck
- d es la distancia entre ruedas

Estableceremos los requisitos en el siguiente apartado.

4.1 Requisitos

Tras plantear el diseño de nuestro sistema podemos concretar todas las necesidades que debemos cubrir con nuestra solución en este apartado de requisitos con la siguiente tabla:

REQ	**Descripción**
REQ 1	Crear elementos estáticos *Fee*
REQ 1.1	Configurar *Fee*: posición, orientación y tamaño
REQ 2	Crear elementos dinámicos *Dee*
REQ 2.1	Configurar *Dee*: posición, orientación, tamaño y propiedades
REQ 3	Crear elementos dinámicos *Bot*
REQ 3.1	Configurar *Bot*: posición, orientación, tamaño y propiedades
REQ 3.2	Asignar velocidades a las ruedas de *Bot*
REQ 3.3	Obtener velocidades de las ruedas de *Bot*
REQ 4	Obtener elementos *Fee, Dee* y *Bot*
REQ 5	Obtener posición y orientación de elementos *Fee, Dee* y *Bot*
REQ 6	Obtener colisiones de la simulación
REQ 7	Avanzar la simulación
REQ 8	Mostrar gráficamente el movimiento de un elemento
REQ 9	Obtener listado de elementos
REQ 10	Crear entorno de simulación
REQ 11	Guardar entorno de simulación
REQ 12	Crear más de un entorno de simulación

4.2 Arquitectura

Teniendo en cuenta tanto lo que hemos escrito en la página anterior como los requisitos de nuestro proyecto, vamos a contar con la siguiente arquitectura de carpetas que explicaremos a continuación:

```
/MSiM
├── /build
├── /include
│   └── /elements
│       └── /subelements
├── /src
│   └── /elements
│       └── /subelements
├── /storage
├── /testing
└── /third_party
```

Build

Esta carpeta va a contener todos los archivos binarios producto de la compilación y creación de la librería de nuestro proyecto. Vamos a emplear CMake [12] para esta tarea. Aquí también encontraremos los ejecutables de pruebas que codifiquemos en la propia librería del proyecto. Antes de la ejecución de CMake se encontrará vacía, pero esto será algo en lo que entraremos un poco más adelante en la memoria.

Include

En las librerías de código es una convención bastante extendida el "incluir" una carpeta con este nombre, también al utilizar CMake. Aquí será donde guardaremos todos los ficheros de cabecera de nuestro código. Es decir, todos los archivos con extensión *".h"*.

La estructura interna con la que va a contar nuestra carpeta *include* será idéntica a la siguiente carpeta src donde contendremos el código correspondiente a los ficheros de cabecera que tendremos aquí. Se divide en la carpeta *elements* y la subcarpeta *subelements*, dentro de *elements*.

Src

De forma paralela a la carpeta include, la carpeta src contiene el código en ficheros con extensión *".cpp"* de los ficheros de cabecera incluidos en la carpeta include que conforman los objetos con los que vamos a trabajar en nuestro proyecto.

Cuenta con la misma estructura interna dividida en la carpeta *elements* y la subcarpeta *subelements*, dentro de *elements*.

Storage

Nuestro proyecto debe de contar con una funcionalidad que almacene diferentes simulaciones y los estados de dichas simulaciones. Será en esta carpeta donde guardaremos los ficheros en los que almacenaremos esta información. A esta carpeta no debe de tener acceso el usuario, por lo que preservamos el formato y no corrupción de la información

Testing

Es en esta carpeta donde crearemos nuestros archivos que prueben las funcionalidades que vayamos desarrollando en nuestro proyecto. Estas pruebas codificadas empleando el programa en ficheros con extensión *".cpp"* posteriormente tendrán sus binarios en *MSiM/build/bin*.

Third party

En esta carpeta guardaremos todas los recursos externos con los que vamos a contar en nuestro proyecto. Aquí será donde tengamos la librería Box2D. Asimismo y aunque lo expliquemos más adelante también incluiremos la librería de SFML (*Simple and Fast Multimedia Library*) [13] que emplearemos para la salida gráfica de nuestro proyecto.
Ahora que hemos explicado las carpetas que van a constituir nuestra arquitectura, entramos a la fase de desarrollo en el siguiente capítulo.

5. Desarrollo

5.1 Elementos del simulador

Ahora entraremos a comentar cada uno de los elementos con los que vamos a contar en nuestro entorno de simulación. Hay algunos elementos que van a ser contenedores de información, en los que no hay ejecución alguna. Empezaremos por ellos y sólo comentaremos los ficheros de cabecera ya que los métodos que contienen son meros *getters* y *setters*. Posteriormente entraremos más en detalle con el código que rige el funcionamiento del sistema en los elementos *World* y *MSiM*.

Pos

```
Class Pos {

      Pos (float x, float y);

      float getX ();
      float getY ();

};
```

Pos es un elemento contenedor de dos números *float* (que van a ser los únicos que utilicemos en el sistema para interactuar con Box2D) que representan una posición. Es un elemento muy básico que se encuentra dentro de la carpeta *subelements*.

Presets

```
Class Presets {

            Presets (float density, float friction,
                     float restitution);

            float getDensity ();
            float getFriction ();
            float getRestitution ();

};
```

Este elemento *Presets* también se encuentra dentro de la carpeta *subelements* y es un elemento contenedor de la información contenida en *Fixture* que programamos en el apartado 3. Contiene su información en números *float* de la densidad, fricción y restitución del elemento dinámico al que sea asociado.

Entity

```
Class Presets {


            Entity (int id, Pos pos);
            virtual ~Entity();

            int getId ();
            Pos& getPos ();
            int setPos (Pos newPos);

};
```

El elemento *Entity* es un elemento que programamos para que sea heredado por los elementos *Fee*, *Dee* y *Bot* para posteriormente facilitar la codificación de la clase *World*. Es nuevamente un elemento contenedor que cuenta con un atributo id de tipo *int* y con una posición. Tiene *getters* y *setters*, así como de un destructor virtual.

Fee

```
Class Fee : public Entity {

            Fee (int id, Pos pos, float xSize, float ySize,
                 float angle);

            float getXSize ();
            float getYSize ();
            float getAngle ();

};
```

El elemento *Fee* es otro de los elementos contenedores que en este caso representa al elemento estático que existe en Box2D. En este caso lo hemos llamado *Fee* (*Fixed Environment Element*). Aquí almacenaremos la información relevante a la hora de construir un elemento estático en Box2D, ya que estos elementos no necesitan de *Fixture* más allá de su forma. La forma de estos objetos la estableceremos como rectángulos.

Dee

```
Class Dee : public Entity {

            Dee (int id, Pos pos, float xSize, float ySize,
                 float angle, Presets presets);

            float getXSize ();
            float getYSize ();
            float getAngle ();
            void setAngle (float newAngle);
            Presets getPresets ();

};
```

El elemento *Dee* (*Dynamic Environment Element*) es otro elemento contenedor que en este caso va a representar al elemento dinámico de Box2D. A diferencia del elemento *Fee*, el elemento *Dee* sí que va a necesitar de propiedades que vaya a inicializar *Fixture*, por lo que precisa de contar con el elemento que antes hemos descrito *Presets*. También va a tener forma de rectángulo.

Bot

```
Class Bot : public Entity {

            Bot (int id, Pos pos, float size, float angle,
                 Presets presets, float lWheel, float rWheel);

            float getSize ();
            float getAngle ();
            Presets getPresets ();
            float getLWheel ();
            float getRWheel ();

            void setAngle (float newAngle);
            void setLWheel (float newLWheel);
            void setRWheel (float newRWheel);

};
```

El elemento *Bot* es el elemento contenedor más importante del sistema. Es idéntico al elemento *Dee* representando al elemento dinámico de Box2D con la salvedad de que *Bot* contiene movimiento propio. De forma parecida a la demostración de Box2D del 2.2 cuando configurábamos una velocidad y un movimiento a los elementos dinámicos.

Esta velocidad va a venir dada por dos números *float*, la velocidad positiva o negativa de cada una de sus ruedas. Más adelante en la explicación del elemento *World* (el más esencial de todo el sistema ya que es el que rige la ejecución), entramos a detallar cómo funciona esta velocidad y qué papel juegan estas ruedas.

Podemos apreciar que en *Bot* no existe variable *xSize* o *ySize*, sino que únicamente contamos con una variable *size*. Esto es porque este *size* va a suponer el radio del robot, ya que lo vamos a modelizar con una forma de círculo.

Collision

```
Class Collision {

            Collision (int id, int tck, Pos pos, string id1,
                       string id2);

            int getId ();
            int getTck ();
            string getId1 ();
            Pos getPos ();
            string getId2 ();

};
```

El elemento *Collision* es un contenedor en el que almacenaremos la información que obtengamos cuando suceda un contacto en nuestro motor Box2D. Cuenta con un identificador de la colisión, un *int* que guarda el número de iteración, la posición del contacto y punteros a los *b2Body* de ambos elementos involucrados en el contacto.

CollisionListener

```
Class CollisionListener : public b2ContactListener {

          CollisionListener ();
          virtual ~CollisionListener ();

          vector<Collision> getCollisions();

          virtual void BeginContact (b2Contact* contact) override;
          virtual void EndContact (b2Contact* contact) override;

};
```

El elemento *CollisionListener* es un elemento que extiende el elemento *b2ContactListener* propio de Box2D. No es un elemento contenedor como los anteriores. Este elemento es inicializado posteriormente en el elemento *World*.

El elemento *b2ContactListener* se utiliza para detectar y gestionar colisiones entre elementos en el motor de físicas. Lo que aquí hacemos es programar lo que queremos que haga el sistema cuando se detecta una colisión.

Lo que haremos será sobrescribir el método *BeginContact* para que en el momento en el que detecte una colisión en el motor de físicas, la información de ésta se guarde en el array *collisions*. A *collisions* se podrá acceder con el método *getCollisions*.

World

Es el elemento más importante de todo el proyecto. Es el elemento en el cual vamos a juntar a todo el resto de elementos (menos *MSiM*), inicializaremos y modificaremos nuestros elementos *Fee, Dee* y *Bot*, manejaremos las colisiones, generaremos nuestra salida gráfica e interactuaremos con Box2D. Salvo la clase *CollisionsListener* que está diseñada para sobrescribir ciertos métodos propios de Box2D, éste es el único elemento del proyecto que interactúa con Box2D.

El tamaño del código *cpp* de esta clase es excesivo como para incluirlo íntegro en esta memoria. Vamos a incluir la cabecera de *World* y explicaremos la funcionalidad de cada método, así como el funcionamiento de la clase en su conjunto.

```
Class World {

            World ();
            ~World ();

            int addFee (float xSize, float ySize, Pos pos,
                        float angle);
            int addDee (float xSize, float ySize, Pos pos,
                        float angle, Presets presets);
            int addBot (float size, Pos pos, float angle,
                        Presets presets);

            Fee* getFee (int id);
            Dee* getDee (int id);
            Bot* getBot (int id);
            map<int, string> getIds();

            int isFee (int id);
            int isDee (int id);
            int isBot (int id);

            Pos* getPos (int id);
            float getAngle (int id);

            float getWheel (int id, int wheel);
            int setWheels (int id, float left, float right);

            vector<Collision> getCollisions ();

            void tick ();
            void print (int id, int x, int y);

};
```

Como hemos dicho, *World* es el elemento de nuestro proyecto que interactúa con Box2D. Esto lo hace a través del b2World que tiene como atributo y al que acceden diversos métodos de nuestra clase. En el constructor simplemente inicializamos todos los atributos, instanciamos un nuevo mundo con gravedad cero y habilitamos los estados computacionales de reposo de los que hablábamos en el 2.2. También asignamos nuestro *collisionListener* al *world* de Box2D.

Los *adders* obtienen toda la información necesaria para construir los elementos dentro de nuestro mundo de Box2D. Dentro de cada *adder* construimos el *b2Body* para incluirlo en *world* incluyéndole los *presets* que correspondan. El *b2Body* lo guardaremos en el mapa *bodies* junto con el identificador que le asignamos. Por defecto los elementos *Fee* tendrán un identificador que comienza con 30000, los *Dee* con 20000 y los *Bot* con 10000.

Tal y como está configurado el sistema hay capacidad para 9999 elementos de cada. Puede ser un posible trabajo a futuro mejorar el sistema de identificación de objetos y cambiar este límite de elementos.

Con los *getters* lo que recibimos es o bien un puntero al elemento que queremos buscar (*Fee, Dee* y *Bot*) sacándolo del mapa *bodies* o bien nos devuelve un mapa que contiene los identificadores de todos los elementos que también nos indica con un *string* qué tipo de *Entity* es. Este método es llamado luego en MSiM para guardar la información de las entidades de un mundo, pero también puede ser consultado por el usuario para controlar su simulación y obtener la información de los elementos cargados en el sistema.

Para estos *getters* consultamos las funciones *isFee*, *isDee* y *isBot* que chequean los identificadores para comprobar el tipo de *Entity* que corresponde al identificador proporcionado. Estas funciones también son públicas disponibles para el usuario.

También contamos con dos *getters* que devuelven atributos particulares de un elemento en particular proporcionado mediante su identificador, *getPos* y *getAngle*.

Ahora entramos a explicar cómo funciona el movimiento de los elementos en nuestro entorno de simulación. Es importante para poder comprender lo que hacen los métodos relacionados con las ruedas (*Wheels*) de nuestro sistema.

Los elementos *Fee* (*Fixed Environment Element*), que representan a los elementos estáticos del motor de físicas, no se mueven en la ejecución. Los elementos *Dee* (*Dynamic Environment Element*), que representan a los elementos dinámicos del motor de físicas sí que se mueven en la ejecución, pero únicamente debido a fuerzas externas (colisiones) que actúen sobre ellos.

Son los elementos *Bot* los que van a representar también a los elementos dinámicos del motor de físicas que cuentan con movimiento en la ejecución, tanto por fuerzas externas que actúen sobre ellos como por fuerzas propias que sean configuradas por el usuario.

Estos elementos *Bot* representan a los robots que se van a utilizar para simular los sistemas multi-robot que es el último fin de este proyecto. Los robots que vamos a simular se basan en el movimiento del E-puck.

De este robot conocemos las entradas que vamos a recibir para su movimiento: *lWheel* y *rWheel*, siendo cada una la fuerza (positiva, negativa o nula) de cada rueda.

Recordamos ahora cómo Box2D imprime fuerzas en un cuerpo con el ejemplo empleado en la demostración del apartado 3.

```
bot1->SetLinearVelocity(b2Vec2(0, 20));
bot2->SetLinearVelocity(b2Vec2(0, 20));
```

Box2D requiere de un vector al que debemos aportar su desglose en el eje de coordenadas x y el del eje y. Debemos calcular ese vector, así como su desglose en ambos ejes. Además, aunque no lo incluyéramos en la demostración, Box2D cuenta también con un método para configurar la velocidad angular. Para que nuestra simulación de robot se comporte como un robot real, debemos también calcularla y añadirla al sistema. Para ello, debemos implementar las ecuaciones del modelo cinemático de un robot diferencial que hemos descrito en el capítulo 4.

Esto se traducirá a Box2D con la función *SetLinearVelocity* y con la función *SetAngularVelocity* de la siguiente manera:

```
float d = bot1->getSize() / 2;
float fL = 0;
float fR = 20;

float angle = bot1->GetAngle();
float omega = (fR - fL) / d;

float fX = ((fL + fR) / 2) * cos (angle);
float fY = ((fL + fR) / 2) * sin (angle);

bot1->SetLinearVelocity(b2Vec2(fX, fY));
bot1->SetAngularVelocity(omega);
```

Esto es lo que hará la función *setWheels*, así como también guardará en nuestro mapa de *ids* la información de las nuevas velocidades para las ruedas. Este método es accesible por el usuario, pero también es empleado por el propio programa por una cuestión que plantea Box2D.

Si esto lo ejecutamos una única vez, el elemento dinámico se moverá de forma indefinida en línea recta hasta colisionar con algo que haga que su movimiento cambie. Puede resultar algo contraintuitivo teniendo en cuenta que hemos calculado una velocidad angular y se la hemos aportado a Box2D. Debería de girar según la velocidad angular que hemos calculado y dirigirse en esa línea recta direccionada al frente del ángulo que le corresponda en cada momento.

Pero Box2D no hace eso. Si ejecutamos esto una única vez lo que va a suceder es que el robot va a girar sobre sí mismo, pero va a continuar en su línea recta que describíamos antes. Es decir, no actualiza la velocidad lineal en base al ángulo al que apunte el frente de nuestro robot. Esto lo solucionamos dentro del método *tick*.

El método *tick* es aquel que hace avanzar el tiempo del *b2World*. Esto lo hace con la función *Step* de Box2D. Este *Step* requiere de parámetros el tiempo en segundos que se quiere que avance la simulación (introduciremos 1 Hz) y un par de variables que afectan a las iteraciones que realiza el motor de físicas para calcular la velocidad y la posición de los cuerpos. Cuanto mayor sean, más precisión tendrán estos cálculos, pero también aumentará el tiempo de cálculo. Lo común en el uso de esta función es emplear 8 para las iteraciones relativas a las velocidades y 3 para las iteraciones relativas a las posiciones.

Nuestro método *tick* no simplemente hace avanzar el tiempo, sino que en cada *tick* le pregunta al *collisionListener* que configuramos en el constructor de la clase si ha sucedido una nueva colisión desde el último *tick* que ha habido en el entorno. Si ha detectado alguno, lo devuelve para guardarlo en nuestro array de colisiones. Este array de colisiones puede ser consultado por el usuario con el método *getCollisions*.

Esto es importante ya que las colisiones y su manejo serán de gran utilidad para el usuario que configure sus sistemas multi-robot, pero lo realmente importante que hace esta función es actualizar los valores de cada uno de los elementos dinámicos que van variando tras el paso del tiempo en la ejecución.

Obtenemos de cada elemento del *b2World* su nuevo ángulo y su nueva posición. Esta información la guardamos en nuestro mapa de elementos MSiM donde tenemos todos los elementos. Y no sólo esto, sino que en el caso de los elementos *Bot* también empleamos esa información que recibimos más la información que ya tenemos sobre las ruedas para poder ejecutar *setWheels* en cada *tick* y que realmente el robot simule el movimiento real de un robot e-puck.

Finalmente, *World* también cuenta con el método *print* que una vez llamado, genera la salida gráfica que representa el movimiento en una ejecución de un elemento proporcionando su identificador. Es en el siguiente capítulo 5.2 donde explicaremos más en detalle la salida gráfica.

MSiM

La clase *MSiM* es la clase principal de nuestro sistema. El código *cpp* de la clase *MSiM* aunque es menor que el de la clase World, es de unos cuantos cientos de líneas. Demasiado como para incluirlo íntegro en esta memoria. Para consultar el código completo, el anexo incluye un enlace al repositorio del proyecto. Igual que con la clase World, vamos a incluir la cabecera de *MSiM* y explicaremos la funcionalidad de cada método, así como el uso de la clase en su totalidad.

```
Class MSiM {

            MSiM ();
            ~MSiM ();

            World* loadWorld (int id);
            void showWorlds ();
            int createWorld (string filename);
            int insertWorld (string path);
            void saveWorld (World* world, string filename);
            void eraseWorld (int id);

};
```

MSiM es la interfaz de nuestra librería. Es lo que el usuario va a tener que instanciar para poder utilizar nuestro sistema. En esencia, esta interfaz gestiona las tareas de guardar, crear, insertar, eliminar y cargar mundos. Estos mundos serán distintos entornos de simulación que pueden existir en nuestro sistema. Pueden instanciarse distintos mundos a la vez.

Lo primero que se hace es cargar al sistema por defecto todos aquellos ficheros que se encuentren dentro de la carpeta */storage* en el momento que se instancia un nuevo elemento MSiM. Recopila sus rutas y las carga en el mapa *worlds* asignándoles un identificador.

Con el método *showWorlds* por pantalla mostrará un listado de los ficheros que hay cargados en el sistema con sus identificadores y la ruta del fichero. Si queremos eliminar algún mundo que haya cargado en el sistema tendremos que proporcionar el identificador del mundo a *eraseWorld*, que no sólo eliminará el mundo del sistema, sino que también eliminará el fichero guardado en la carpeta */storage*.

Para crear un nuevo mundo tenemos dos opciones, o bien crear uno nuevo desde cero aportando a *createWorld* únicamente el nombre que queremos que tenga el fichero que se va a crear dentro de la carpeta */storage*, o bien podemos aportar a *insertWorld* la ruta a un fichero de datos para nuestro sistema. Este fichero tiene que seguir el formato indicado en el fichero *Template.txt* para que pueda ser introducir en el sistema. Podemos encontrar este formato a continuación.

```
[
(FEE,{posX, posY, xSize, ySize, angle})
...
(DEE,{posX, posY, xSize, ySize, angle, pDensity, pFriction,
pRestitution})
...
(BOT,{posX, posY, size, angle, pDensity, pFriction, pRestitution,
lWheel, rWheel})
...
];
```

Se pueden incluir tantos elementos como se quiera. Todos ellos deberán ser introducidos a razón de uno por línea siguiendo este formato. Los puntos suspensivos no forman parte del formato, solamente constituyen una expresión de que es posible añadir más elementos. De hecho, no deben estar ordenados los elementos entre *Fees*, *Dees* o *Bots*, sino que pueden intercalarse a placer. Los elementos no deben de estar separados por comas y los números deberán ser de tipo *float*.

Ahora nos encontramos con el método más relevante de toda la clase, *loadWorld*. Este método devuelve el puntero al mundo que se solicita mediante su identificador si es que el identificador está en el sistema, el fichero asociado en el mapa *worlds* es accesible y no produce ningún error a la hora de leer el fichero.

Es en este método en el que se analiza el contenido del fichero de datos, donde línea por línea analiza el elemento que ha de cargar (*Fee, Dee* o *Bot*) y comprueba los argumentos introducidos para su carga.

Si no genera ningún error en su ejecución, devuelve un puntero a una instancia de objeto *World* en el cual ya hemos cargado e instanciado todos los objetos indicados en el fichero de datos analizado.

Dentro de este método utilizamos el método auxiliar privado *splitByComma*.

Ya por último, con el método *saveWorld* lo que codificamos es el obtener el estado de todos los elementos de nuestro mundo pasado como argumento y guardamos esta información en un fichero *filename* en la carpeta */storage* siguiendo con el formato de los ficheros de datos del sistema.

5.2 Salida gráfica

Como mencionábamos en la explicación de la clase *World*, es su método *print* el que ejecuta la salida gráfica del elemento que el usuario quiera visualizar. Para esto vamos a emplear SFML (*Simple and Fast Multimedia Library)*. Ésta es una librería utilizada para representar elementos gráficos por pantalla.

El método *print* requiere del usuario el identificador del objeto que desea pintar, así como dos números enteros que van a representar la resolución de la ventana deseada para la salida gráfica.

SFML crea una ventana en la que podemos dibujar elementos dependiendo de lo que nuestro código le pida a SFML que dibuje. En los primeros contactos con la librería al recibir salidas gráficas completamente distintas a las esperadas, nos damos cuenta de que el origen de coordenadas para SFML es la esquina superior izquierda.

Es entonces cuando configuramos la herramienta para que el origen de coordenadas sea el centro de la ventana (independientemente de la resolución configurada). Para facilitar la visualización, le incluimos también ejes de coordenadas de color blanco con el fondo negro para identificar fácilmente las posiciones de nuestro elemento. Las posiciones de nuestro elemento las obtiene extrayendo el vector de posiciones *positions* empleando el identificador aportado a *print*.

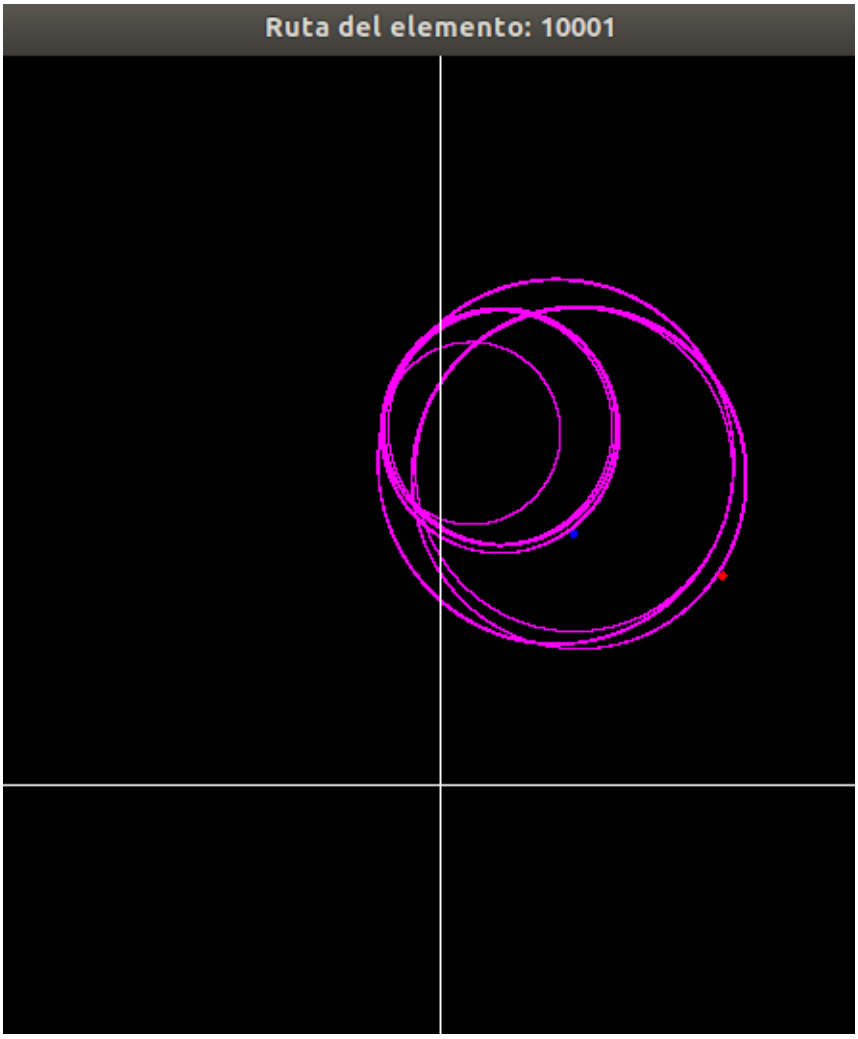
En una primera versión, creábamos un array de vértices a partir de ese array de posiciones y con esos vértices dibujábamos líneas en SFML. Con esta versión encontrábamos una barrera en las 7500 iteraciones para no producir un error de memoria. A iteración por Hz, esto equivale a apenas dos minutos de simulación para que ya no fuese posible pintar el recorrido de nuestro elemento. Además, el último vértice forzaba el dibujar una línea al eje de coordenadas, elemento que no deseamos tener en nuestra salida.

Cambiamos a una segunda versión en la que en lugar de crear líneas dibujamos círculos en cada posición. Al tener tantos círculos juntos, el efecto sería el mismo que el de tener líneas. Y solucionando la parte de la línea que conectaba el eje de coordenadas. Sin embargo, no conseguíamos aumentar el número de iteraciones y, por ende, de simulación posible para mostrar gráficamente el resultado.

Planteamos sacrificar precisión en el dibujo de los resultados a cambio de poder aumentar el tiempo de simulación, probando a dibujar únicamente las posiciones con *tick* impar. De esta manera podemos duplicar el tiempo de simulación a 15000 iteraciones (4 minutos) al sólo dibujar una de cada dos posiciones.

Esto abre la puerta a cómo considerar si la precisión que perdemos es realmente relevante o no. Quizás no lo sea en un movimiento circular que se repite constantemente y no es siquiera apreciable la diferencia. O quizás tampoco sea importante en un movimiento que tenga poca velocidad y la diferencia entre la posición en una iteración con respecto a diez iteraciones anteriores o posteriores sea ínfima. Pero, ¿si hay muy pocas iteraciones y pocos puntos que dibujar? ¿Y si la velocidad del movimiento es tan alta que la diferencia entre una posición y su siguiente es radicalmente diferente?

Puede ser una implementación en una nueva versión el aumentando el tiempo de simulación optimizando el trazado de la salida gráfica. Pero por suerte, encontramos una solución para poder aumentar de forma radical el tiempo de simulación de nuestro sistema para que sea posible dibujarlo. Empleamos la opción más óptima de SFML que son los *VertexArray*. Lo que hacemos es poblarlo de nuestras posiciones y configurar la ventana para que lo dibuje. Le pintamos también en la primera posición (inicio) un punto rojo y en la última posición (final) un punto azul para así facilitar la visualización del movimiento del elemento. Programamos un ejemplo para el que nos aparece como resultado la siguiente *snaphot*.



[Imagen 13] Snapshot de la salida gráfica de MSiM

Con esta última configuración hemos conseguido ejecutar 27.000.000 iteraciones en esta simulación. Esto equivale a 7500 minutos. O lo que es lo mismo, 5 días y 5 horas. Con esta solución mejoramos un 187400 % nuestra anterior solución sin perder un ápice de precisión.

6. Pruebas

Una vez ya desarrollado todo el código debemos comprobar que efectivamente, el sistema funciona y cumple con los requisitos establecidos en el apartado 4.1. Como puede ser evidente, no hemos desarrollado el código durante el transcurso del proyecto sin haberlo ido probando y ajustando debidamente.

Es ahora cuando vamos a diseñar y ejecutar distintos casos de prueba que nos mostrarán si nuestro código funciona y cumple con los requisitos o no. Vamos a desglosar los distintos casos de prueba en tres secciones.

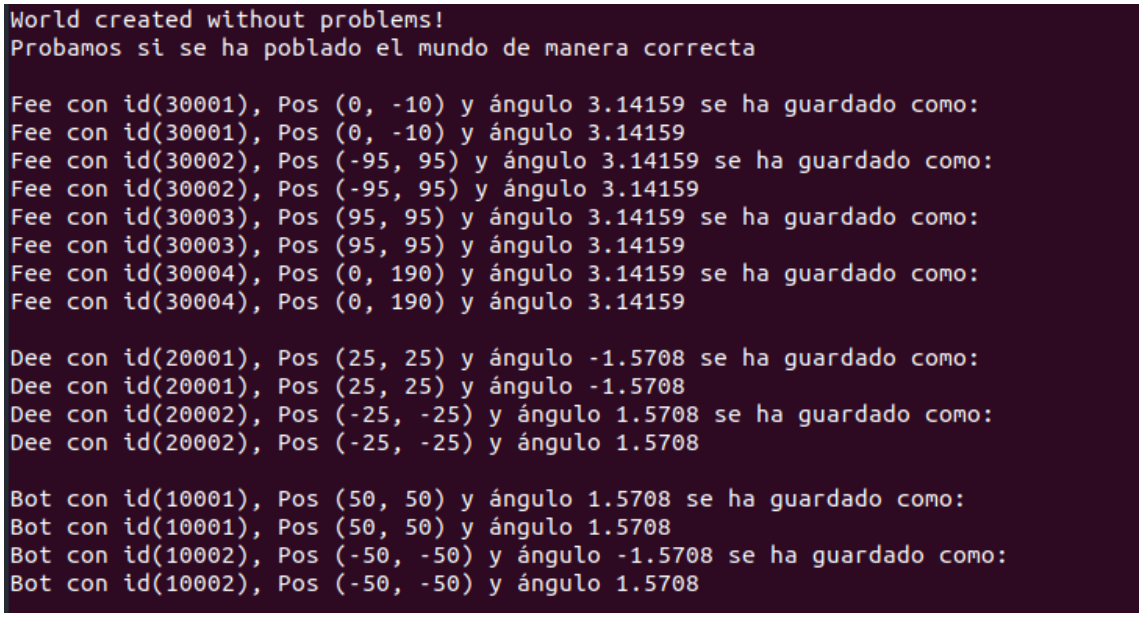
6.1 Lógica interna

En esta sección vamos a probar la lógica del sistema. Debemos probar pues los requisitos del REQ 1 al REQ 9 sin incluir al REQ 8, que debemos probar en la siguiente sección. Ésta es la sección en la que más requisitos debemos probar.

Lo que haremos será:

1. Inicializar un entorno de simulación (aunque eso se considere también como REQ 10 tendremos que evaluarlo y comprobarlo posteriormente en el 5.3).
2. Poblar este entorno con algunos elementos *Fee, Dee* y *Bot*.
3. Comprobar que el mundo se ha poblado de manera correcta con los datos que hemos introducido.
4. Imprimir movimiento a *Bot*.
5. Comprobar que se ha impreso correctamente ese movimiento.
6. Calcular movimiento y comprobar que *Bot* se está moviendo correctamente.
7. Calcular colisiones y comprobar que se han producido correctamente.

Para empezar con nuestras pruebas y chequear los tres primeros puntos, vamos a poblar el mundo de elementos. Empleando los *getters* del sistema conseguimos que el resultado que nos devuelva por pantalla nuestra prueba sea el que se muestra en la siguiente *snapshot*.



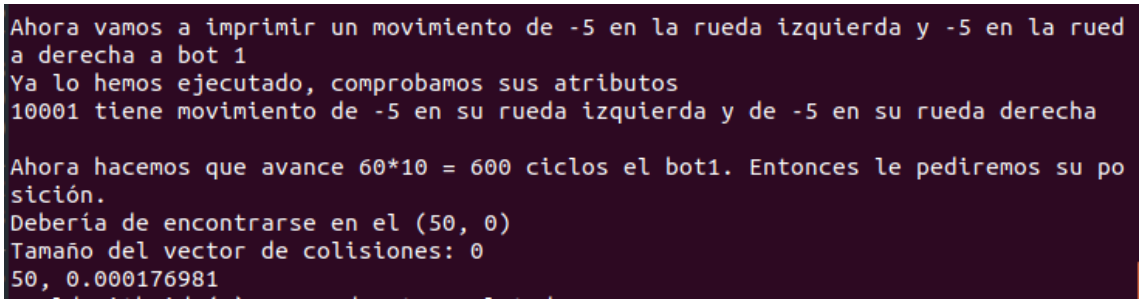
[Imagen 14] Snapshot de la salida por consola tras poblar el mundo

Como podemos apreciar, los tres primeros puntos de nuestras pruebas se cumplen. Con esta prueba hemos comprobado **REQ 1**, **REQ 1.1**, **REQ 2**, **REQ 2.2**, **REQ 3**, **REQ 3.1**, **REQ 4** y **REQ 5**. Para este primer apartado de lógica nos falta por comprobar REQ 3.2, REQ 3.3, REQ 6, REQ 7 y REQ 9.

Lo siguiente que haremos será comprobar los puntos 4 y 5. Con esto comprobaremos los REQ 3.2 y REQ 3.3.

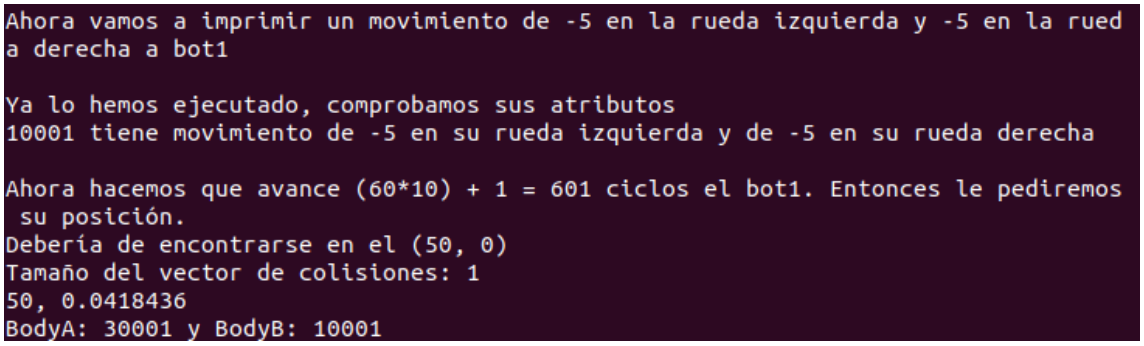
Tenemos un *Bot* con *id(10001)* que se encuentra en la posición (50, 50). Le vamos a imprimir una fuerza de -5 en la rueda izquierda y de -5 en la rueda derecha. Según nuestras fórmulas de movimiento, esto va a suponer una velocidad de 0 en el eje X, -5 en el eje Y y 0 de velocidad angular. Lo estamos programando pues, para que vaya hacia abajo, a entrar en contacto en el eje X con el elemento *floor*.

Teniendo en cuenta que una velocidad de valor 5 significa que cada segundo (=60 Hz) recorremos 5 posiciones. Calculando, deberíamos de colisionar con el elemento *floor* en la iteración 600. Lo codificamos y vemos la salida que produce por pantalla en la siguiente *snapshot*.



[Imagen 15] Snapshot de la salida por consola tratando de detectar colisiones

Podemos ver que el resultado no es del todo el esperado. Llega a la posición esperada, pero le faltan apenas ciento setenta millonésimas para alcanzarlo. Como aún no ha llegado al eje X, es de esperar que lo haga si iteramos un ciclo más, ya que, calculando, cada iteración deberá moverse 0.083 posiciones, suficiente como para alcanzarlo. Modificamos una iteración más, le incluimos un extractor de información de la colisión y aparece la salida de la siguiente snapshot.



[Imagen 16] Snapshot de la salida por consola tratando de detectar colisiones

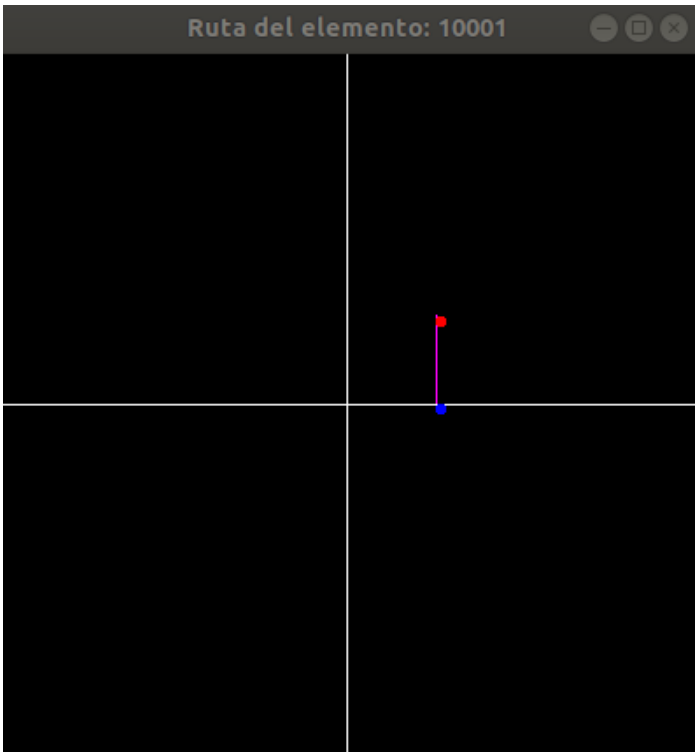
Como podemos comprobar ahora, con una iteración más sí que se produce la colisión esperada entre el *floor (30001)* y *bot1(10001)*. Con esto no sólo hemos cumplido con todos los puntos de esta sección, sino que aparte de comprobar **REQ 3.2** y **REQ 3.3**, también hemos cumplido **REQ 6** y **REQ 7**. Nos falta únicamente en esta sección comprobar **REQ 9**, que comprobamos fácilmente llamando a *getIds* y mostrando el contenido por consola mediante un iterador. Podremos comprobar que la información coincide con la mostrada por la imagen 14, por lo que comprobamos que, en efecto, funciona correctamente.

```
Elementos del entorno de simulación:
        - Id: 10001  (Bot)
        - Id: 10002  (Bot)
        - Id: 20001  (Dee)
        - Id: 20002  (Dee)
        - Id: 30001  (Fee)
        - Id: 30002  (Fee)
        - Id: 30003  (Fee)
        - Id: 30004  (Fee)
```

[Imagen 17] Snapshot de la salida por consola mostrando los elementos

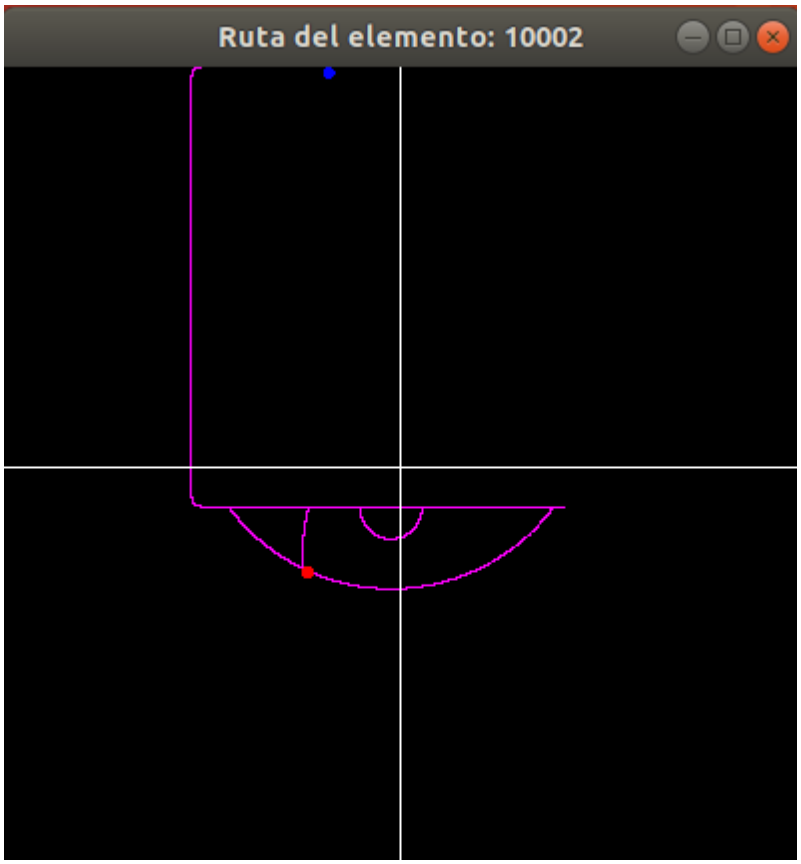
6.2 Salida gráfica

Tenemos un único requisito que comprobar aquí, REQ 8. Para esto simplemente llamaremos al método *print* probándolo con *bot1*.



[Imagen 18] Snapshot de la salida gráfica

La representación es ciertamente algo pobre si pretendemos apreciar cómo se dibuja un movimiento en el sistema, vamos a iterar *bot2* imprimiéndole un movimiento más vistoso para que se pueda apreciar mejor en la salida gráfica.



[Imagen 19] Snapshot de la salida gráfica

Con esta representación conseguimos comprobar **REQ 8**, quedando ya únicamente REQ 10, REQ 11 y REQ 12.

6.3 Ficheros

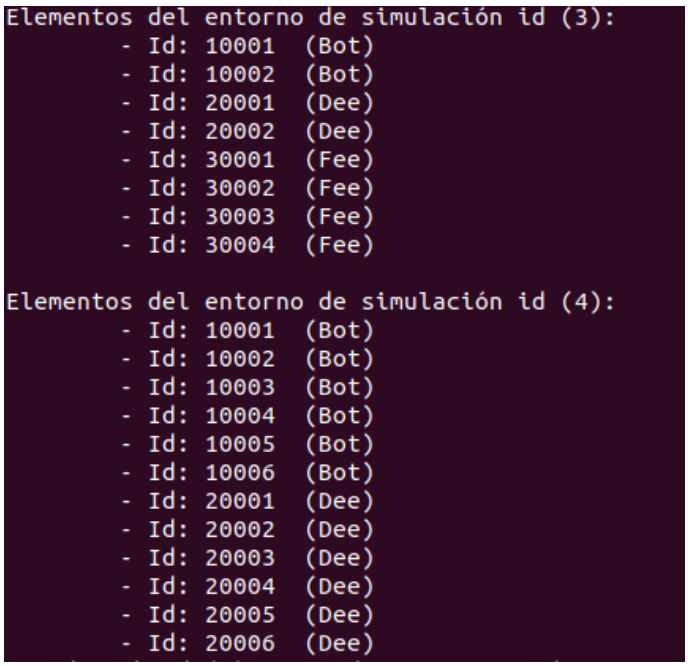
Debemos ahora comprobar los requisitos relativos al manejo de ficheros. Debe de ser posible que nuestro entorno de simulación se cree, podamos guardarlo, y que podamos crear más de uno.

El primer requisito **REQ 10** ya lo hemos cumplido. Hemos estado creando y eliminando el entorno de simulación en el que hemos trabajado cada vez que ejecutábamos nuestras pruebas. De todas maneras, vamos a volver a ilustrarlo.

Para los otros dos requisitos, lo que vamos a hacer es primero guardar el entorno en el que hemos estado trabajando y la siguiente vez que ejecutemos nuestras pruebas, cargaremos ese entorno y crearemos otro distinto, alternando código de uno y de otro en la misma prueba.

```
msim.saveWorld(world, "Pruebas2");

// Eliminamos esta línea y ejecutamos lo siguiente

MSiM msim;

msim.showWorlds();

int idWorld1 = 3; // Lo hemos conocido por showWorlds

World* world1 = msim.loadWorld(idWorld1);

int idWorld2 = msim.creatWorld("Pruebas");

World* world2 = msim.loadWorld(idWorld2);
```

Esto es lo interesante del código que ahora ejecutamos para hacer esta prueba. El resto del código es poblar de elementos el *world2*. Después con un par de iteradores mostramos por pantalla los elementos de *world1* y *world2*.



[Imagen 20] Snapshot de la salida por consola mostrando la información de los dos entornos

En esta *snapshot* podemos apreciar que se cargan de forma correcta los elementos que teníamos en el anterior entorno, por lo que podemos concluir que los entornos de simulación se pueden guardar exitosamente.

A continuación, podemos ver cómo está también cargado el segundo entorno de simulación a la vez que el anterior y, por ende, conseguimos cumplir todos los requisitos que establecimos para nuestro proyecto.

7. Conclusiones

Cuando empezábamos con este proyecto teníamos la misión principal de desarrollar un entorno de simulación para sistemas multi-robot. Tras los resultados de las pruebas al sistema que hemos realizado e incluido en esta memoria podemos concluir que hemos cumplido con esa misión de forma satisfactoria. También hemos comprobado haber satisfecho todos los requisitos que establecimos para este proyecto.

Este proyecto me ha supuesto un gran reto en lo relativo al diseño y a la concepción del sistema. Han sido más de media docena de diseños los que hemos planificado para llevar a cabo este proyecto, que se han ido sucediendo en el tiempo a medida que elegíamos una tecnología para integrar en el sistema, la integrábamos, rediseñábamos en base a los cambios que suponíamos que iba a conllevar, volvíamos a rediseñar en base a los cambios que realmente conllevaba la implementación del motor de físicas o de la librería gráfica.

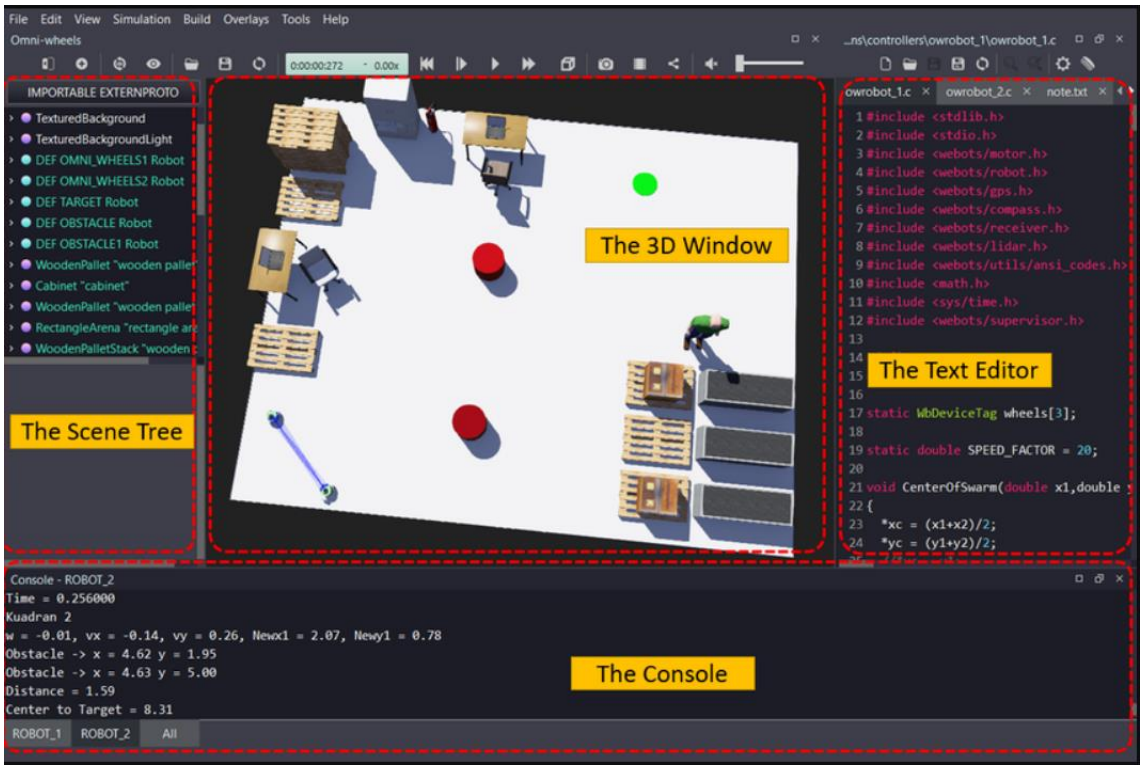
El llegar a la conclusión del proyecto con este conocimiento adquirido en base a la experiencia tanto con el propio código desarrollado para este entorno de simulación, así como el manejo de Box2D o de SFML me ha hecho notar puntos de mejora de la aplicación que por falta de tiempo no he podido implementar. Los recopilo dentro del siguiente capítulo.

Siento satisfacción con el resultado al que hemos llegado. Han sido varios meses de trabajo, búsqueda de información y recursos en la web, libros, foros y compañeros. Tras este trabajo siento un orgullo que hace plantearme cómo mejorar en el siguiente proyecto al que me enfrente.

8. Futuros proyectos

En este capítulo expondremos qué futuros proyectos creemos que se pueden desarrollar a partir de nuestro entorno de simulación. También y como hemos mencionado en capítulos anteriores en esta memoria, recopilamos e incluimos todos los puntos de mejora o posibles funcionalidades futuras que hemos identificado tras el desarrollo del proyecto.

Este entorno de simulación es una primera versión de un entorno que se puede desarrollar y expandir en muchas vertientes. A la hora de investigar el estado del arte y probar los simuladores CoppeliaSim y Webots encontramos que cuentan con una interfaz gráfica de desarrollo que facilita notablemente el uso de sus entornos de simulación en comparación con nuestra solución.



[Imagen 21] Snapshot de la interfaz gráfica de desarrollo de Webots

En esta *snapshot* de la interfaz de Webots podemos ver cómo se contiene un IDE en su totalidad, pudiendo codificar a la vez que vemos la salida por consola y una visualización gráfica del entorno simulado. Un proyecto orientado a realizar un IDE puede llevar nuestro MSiM a otro nivel de uso, alcance e impacto muy superior.

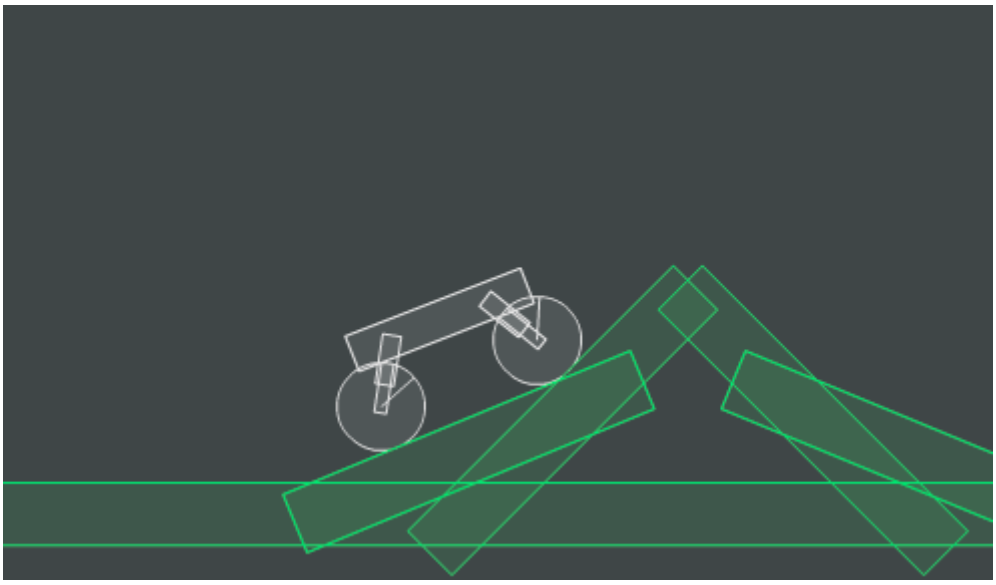
El paso a un simulador en un entorno tridimensional es un planteamiento que descartamos como proyecto futuro tomando éste de base, puesto que lo aquí desarrollado no aportaría mucho a un entorno tridimensional, ya que no sería útil nada de la integración con Box2D, ni la configuración de la salida gráfica, así como tampoco el diseño de los elementos que deberían de ser rediseñados para poder ser integrados en este nuevo entorno. Es por ello pues que éste es un buen planteamiento a desarrollar, pero no con la base de este proyecto.

Este entorno de simulación MSiM puede emplearse en su estado actual para proyectos educativos para configurar de forma programática interacciones entre robots, así como también para probar sistemas multi-robot. La realidad es que el nivel de este proyecto no es un nivel tan alto como para ser superior a otros simuladores ya existentes que se pueden utilizar en proyectos de investigación y desarrollo como para que MSiM pueda ser considerado en su estado actual una mejora respecto a ellos.

Si tuviéramos más tiempo para desarrollar una nueva versión mejorada de MSiM lo que deberíamos hacer sería no centrarnos en nuevas funcionalidades como por ejemplo puede ser la interfaz gráfica de la que hemos hablado, sino que debemos centrarnos en los puntos de mejora que tiene nuestro sistema actualmente. Este es el punto de mejora más interesante que hemos identificado de nuestro proyecto:

- **Replantear el cuerpo de los robots.** En nuestro sistema planteamos el robot E-puck como un círculo que cuenta con una orientación y al que le imaginamos una rueda derecha y una izquierda. Sólo puede ir hacia delante o hacia atrás. Y es el juego de las velocidades impresas en cada rueda lo que hace que se mueva mediante la programación de las ecuaciones de movimiento del robot diferencial.

Estamos simulando esas ruedas. Pero, ¿y si el robot realmente tuviera esas ruedas en la simulación?



[Imagen 22] Snapshot de una simulación con joints en Box2D

Box2D cuenta con un elemento llamado *Joint* cuya función es la de conectar dos elementos. De esta manera, se pueden simulares interacciones como pistones, cuerdas o en nuestro caso, ruedas.

Explorar esta mejor puede conllevarnos una mejora en la optimización del sistema, ya que no debemos realizar ningún cálculo para desglosar y reintroducir en el motor de físicas ya que lo que mandaremos moverse hacia delante o hacia detrás serán las propias ruedas, que, fijas al robot, reproducirán también el movimiento real del robot.

9. Análisis de impacto

Este proyecto de desarrollo de un entorno de simulación para sistemas multi-robot supone un gran impacto en mi formación como profesional. Durante estos años de carrera universitaria he tenido que realizar y documentar varios proyectos. En mi carrera profesional también he tenido que realizar otros tantos. Pero ninguno de ellos ha podido conllevar tanto trabajo y dedicación como puede ser el Trabajo de Fin de Grado de la carrera universitaria.

El impacto que este proyecto puede suponer en mi formación como profesional atiende a la búsqueda del esfuerzo y la dedicación requeridas por este proyecto en los proyectos y trabajos venideros, así como de la autocrítica y de la mejora constante de la que hablaba en el capítulo 7.

En lo relativo al impacto más allá de mi persona que puede tener este proyecto, se puede alinear también a los ODS (Objetivos de Desarrollo Sostenible) definidos por la Asamblea General de las Naciones Unidas contenidos en la Agenda 2030, podemos relacionar nuestro trabajo de una forma directa a dos de ellos:

- **ODS 4 - Educación de calidad.** Este proyecto puede ser utilizado en un entorno educativo para aprender con la experiencia a desarrollar código y programar movimiento de un robot E-puck. También puede servir de base para otros proyectos como los que comentábamos en el apartado anterior y que futuros estudiantes puedan servirse de ello para experimentar un impacto similar en su formación como profesionales.

- **ODS 9 - Industria, innovación e infraestructura.** Siendo un proyecto de desarrollo de software con la finalidad de construir un elemento de simulación de sistemas robóticos, la vinculación a este ODS es clara. Como comentábamos en el capítulo anterior, la realidad es que el impacto que tiene este proyecto en la industria robótica en el momento actual no es muy grande. Pero puede ser la base para construir algo que sea aún mayor, ya que el todo es mayor que la suma de sus partes.

10. Bibliografía

[1] Una revisión de los sistemas multi-robot: Desafíos actuales para los operadores y nuevos desarrollos de interfaces, J.J. Roldän Gómez, Jorge de León Rivas, Pablo García Auñón, Antonio Barrientos, 2020, UPV

Último acceso: 11 de noviembre 2024

Enlace: https://doi.org/10.4995/riai.2020.13100

[2] A Review of Research in Multi-Robot Systems, Avinash Gautam, Sudeept Mohan, 2012, IEEE

Último acceso: 11 de noviembre 2024

Enlace: https://www.researchgate.net/profile/Avinash-Gautam-5/publication/235939781_A_Review_of_Research_in_Multi-Robot_Systems/links/0fcfd514870abaa1b2000000/A-Review-of-Research-in-Multi-Robot-Systems.pdf

[3] The e-puck, a Robot Designed for Education in Engineering, Francesco Mondada, Michael Bonani, Adam Klaptocz, 2009

Último acceso: 12 de enero 2025

Enlace: https://www.researchgate.net/profile/Francesco-Mondada/publication/37468823_The_e-puck_a_Robot_Designed_for_Education_in_Engineering/links/00b49516c573430e21000000/The-e-puck-a-Robot-Designed-for-Education-in-Engineering.pdf?origin=publication_detail&_tp=eyJjb250ZXh0Ijp7ImZpcnN0UGFnZSI6InB1YmxpY2F0aW9uIiwicGFnZSI6InB1YmxpY2F0aW9uRG93bmxvYWQiLCJwcmV2aW91c1BhZ2UiOiJwdWJsaWNhdGlvbiJ9fQ&__cf_chl_tk=JRxL34hmMek5ZZcWC4y3qfSxrJCI2wYEyTET.MLqI3M-1736682676-1.0.1.1-_R5ZCum6S14UMOHOLcl7Movty8Ba2o7WRA6rqUtsdC8

[4] ODE Documentation

Último acceso: 11 de noviembre 2024

Enlace: https://ode.org/wiki/index.php/Main_Page

[5] Gazebo Simulator

Último acceso: 11 de noviembre 2024

Enlace: https://gazebosim.org/features

[6] PyBullet – Bullet Real-Time Physics Simulation

Último acceso: 11 de noviembre 2024

Enlace: https://pybullet.org/wordpress/

[7] Box2D Documentation

Último acceso: 13 de enero 2025

Enlace: https://box2d.org/documentation/

[8] CoppeliaSim User Manual

Último acceso: 11 de noviembre 2024

Enlace: https://manual.coppeliarobotics.com/

[9] Webots Reference Manual

Último acceso: 13 de enero 2025

Enlace: https://cyberbotics.com/doc/reference/index

[10] iforce2d Box2D C++ tutorials

Último acceso: 13 de enero 2025

Enlace: https://www.iforce2d.net/b2dtut/

[11] Google Code Archive – Box2D_v2.1.2

Último acceso: 11 de noviembre 2024

Enlace: https://code.google.com/archive/p/box2d/downloads

[12] CMake Documentation

Último acceso: 10 de enero 2025

Enlace: https://cmake.org/documentation/

[13] SFML Documentation

Último acceso: 13 de enero 2025

Enlace: https://www.sfml-dev.org/documentation/3.0.0/

[14] Github

Último acceso: 14 de enero 2025

Enlace: https://github.com/

[15] Ubuntu 18.04.6 LTS

Último acceso: 10 de enero de 2025

Enlace: https://releases.ubuntu.com/18.04/

[16] Oracle VM Virtualbox

Último acceso: 8 de enero de 2025

Enlace: https://www.virtualbox.org/

11. Anexo

11.1 Repositorio del proyecto

El código referido de nuestro proyecto estará accesible en el siguiente repositorio de Github [14]:

https://github.com/mmorales21/MSiM

11.2 Configuración del entorno

Para este proyecto vamos a emplear una distribución Linux. En particular la distribución de 2021 de Ubuntu 18.04.6 LTS [15]. Esta distribución la vamos a ejecutar dentro de una máquina virtual de Oracle VM VirtualBox [16]. En sí estos detalles no son particularmente muy relevantes. Simplemente los mencionamos para explicar el porqué de no utilizar una versión más actual de Ubuntu.

El proyecto nace no sólo con la finalidad de servir como Trabajo de Fin de Grado, sino que también (y principalmente) tiene la finalidad de ser empleado en el futuro por usuarios que desarrollen más funcionalidades o programas distintos usando este entorno de simulación como base para otros proyectos.

Elegimos un entorno de desarrollo que sea fácil de replicar para otros que puedan venir después, que no dependa de herramientas o bibliotecas más modernas que puedan requerir de más recursos y que pueda ser utilizado con unos recursos accesibles por la gran mayoría de ordenadores. Además, Ubuntu 18.04.6 LTS cuenta con soporte hasta 2028, por lo que a fecha de publicación de esta memoria sigue contando con al menos tres años de cobertura.

Igualmente, el código desarrollado aquí es portable a otras distribuciones de Linux. Únicamente cuenta con una particularidad en las distribuciones de Ubuntu que detallaremos a continuación en la configuración de este entorno.

Para configurar nuestro entorno de desarrollo, el primer paso es ejecutar la distribución de Linux de nuestra elección. Debemos asegurarnos de que contamos con el compilador *c++* instalado.

Asimismo, como mencionamos en el capítulo 4.2, también vamos a emplear la herramienta CMake. Esto lo hacemos para aprovechar el fichero CMake con el que ya cuenta la versión de Box2D para facilitarnos la construcción de la librería Box2D. Aprovecharemos más adelante y también usaremos esta herramienta para construir nuestra librería *libMSiM.a*.

Nos aseguramos de contar con ambas herramientas con el siguiente comando en la terminal:

```
sudo apt-get install g++ cmake libglu-dev libxi-dev
```

Tras esto, vamos a instalar Box2D. Para ello, debemos descargar el archivo comprimido de la v2.1.2 de Box2D [15]. Tras descomprimirlo debemos situarnos en su carpeta Build (Box2D_v2.1.2/Box2D/Build) y ejecutar los siguientes comandos.

```
cmake ..

make
```

Aquí es donde encontramos la particularidad de Ubuntu. Si ejecutáramos estos dos comandos aparecería un error al ejecutar el segundo. Es por eso por lo que al construir la librería Box2D en Ubuntu debemos modificar ligeramente el archivo link.txt ubicado en la carpeta generada por CMake: */Box2D_v2.1.2/Box2D/Build/Testbed/CMakeFiles/Testbed.dir* y añadir al final del archivo **-lX11**. Esto lo que hace es añadir la libería libx11 al listado de librerías requeridas en el proceso de compilación.

Si hubiésemos ejecutado ya el make y se hubiese producido el error, debemos limpiar la carpeta Build y volver a ejecutar el comando *cmake ..* para asegurarnos de hacer una construcción limpia de la librería.

Una vez que make ha terminado de forma correcta y sin generar ningún error, ya podemos comprobar que nuestra instalación ha sido exitosa ejecutando los siguientes comandos y probando el banco de pruebas del que hablábamos en el capítulo 3.

```
cd Testbed

./Testbed
```

Con esto ya tendríamos instalado el entorno de desarrollo de nuestro proyecto. Más adelante hablaremos de la salida gráfica, para la cual necesitaremos descargar también SMFL.

11.3 CMake

Como indicábamos en la configuración del entorno de desarrollo del proyecto, vamos a necesitar de la herramienta Cmake para construir Box2D. Por lo que simplifica la compilación y construcción de proyectos y ya que la hemos empleado para Box2D, la emplearemos también para nuestro proyecto. Para ello, necesitamos redactar un fichero *CMakeLists.txt* en nuestra carpeta raíz */MSiM*. Ahora sólo debemos entrar en la carpeta */build* y desde ahí ejecutar la herramienta con los siguientes comandos.

```
cmake ..
```

```
make
```

Esto generará en la carpeta */build* el fichero *libMSiM*.a, compilado y listo para poder ser utilizado por nuestros usuarios.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Tue Jan 14 20:29:08 CET 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)