UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Teleoperación de un vehículo simulado con Unity mediante librerías de Machine Learning

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE SISTEMAS DE TELECOMUNICACIONES

Autor: Diego José Trias Ramos
Director: Juan Ángel Pastor Franco
Cartagena, 10 de diciembre 2020

Tabla de contenidos

Resumen

En la primera parte de este proyecto, estudiaremos el kit de herramientas ml-agents[1], el cual se aprovecha del motor de videojuegos Unity 3D[2] y de la librería TensorFlow[3], para crear entornos altamente configurables en los que entrenar agentes de *Machine Learning*. El hecho de que los entornos sean creados dentro del motor de Unity facilitará mucho el entrenamiento, veremos como muchas de las complejidades que suelen venir acompañadas de trabajar con Machine Learning, son abstraídas gracias a la combinación de Unity y *ml-agents*; además, también nos resultará bastante sencillo visualizar en todo momento lo que ocurre durante el entrenamiento.

Por otro lado, en la segunda parte de este proyecto, nos enfocaremos en la transmisión de datos desde Unity a un servicio REST, el cual será creado mediante el uso de Java[4] y el *framework* Spring[5]. Más adelante, visualizaremos estos datos en tiempo real con Elixir[6] y el *framework* Phoenix[7]. Una vez más, observaremos como muchas de las complejidades implícitas en estas tareas, serán abstraídas por las distintas herramientas utilizadas.

[1] https://github.com/Unity-Technologies/ml-agents
[2] https://unity.com/
[3] https://www.tensorflow.org/
[4] https://www.java.com/es/
[5] https://spring.io/
[6] https://elixir-lang.org/
[7] https://www.phoenixframework.org/

Lista de figuras

1. Introducción

Motivación: Este proyecto surge como un prototipo para explorar la integración del entorno unity con ML agents y el uso de servicios REST con Spring Boot, con objeto de más adelante utilizarlo para prácticas de programación, hackathons y como base para nuevas líneas de estudio.

Objetivos: con todo ello el objetivo general del proyecto es servir como ejemplo de algunas de las herramientas que Unity y ml-agents nos ofrecen para. Estos objetivos se pueden desglosar en:

1.- Introducción al motor Unity caso de estudio para probar el aprendizaje de un solo agente.

2.- Introducción a ml-agents y mención de algunos proyectos de ejemplo.

3.- Caso de estudio para probar el aprendizaje de un solo agente.

4.- Caso de estudio con varios agentes en distintos equipos

5.- Caso de estudio con varios agentes en un equipo, donde las decisiones de un agente están condicionadas a las decisiones del otro.

6.- Adición de servicios REST para almacenar y recuperar estadísticas del juego/simulación.

7.- Visualización de los datos mediante Elixir + Phoenix

8.- Referencias a tutoriales relacionadas con la materia, guías de instalación y uso de las distintas tecnologías utilizadas y recomendaciones generales.

9.- Conclusiones

Organización del documento: con todo ello, este documento servirá como un punto de partida para todo aquel que quiera empezar en el mundo de Machine Learning a través del motor Unity y ml-agents.

En el capítulo 2, veremos brevemente cuáles han sido las herramientas utilizadas

En el capítulo 3, ofreceremos una introducción al motor Unity para la creación de videojuegos, pues este es el motor que utilizaremos para las simulaciones.

En el capítulo 4, explicaremos algunas de las causas que dieron lugar a la inteligencia artificial y al machine learning.

En el capítulo 5, veremos la lógica detrás de ml-agents y la estructura que hay que seguir para su correcto uso y configuración

En el capítulo 6, haremos una breve mención de algunos casos de uso que vienen por defecto en el proyecto inicial de ml-agents.

En el capítulo 7, estudiaremos varios proyectos con aviones como agentes en distintos entornos y situaciones.

En el capítulo 8, usaremos Java y Spring para crear servicios REST para guardar datos de la simulación.

En el capítulo 9, utilizaremos Elixir y Phoenix para visualizar en tiempo real nuestra base de datos.

Y, finalmente, en el capítulo 10, se aportarán las conclusiones de este proyecto.

2. Herramientas y Tecnologías utilizadas

Ahora pasaremos a mencionar las herramientas y tecnologías que fueron necesarias para desarrollar este trabajo. La motivación detrás de la elección de estas tecnologías se debe a la popularidad actual que tienen, y el fácil acceso a guías y documentación sobre las mismas. Para cumplir los objetivos del proyecto, las siguientes tecnologías serán utilizadas:

a. Unity 3D (Lenguaje de programación: C#)

b. Kit de herramientas de Unity: ml-agents

c. Python 3.7 (con Anaconda)

d. Spring Boot 2.3.3 con Java

e. Elixir y Phoenix

En la Figura 1 se muestran las dependencias entre los paquetes software utilizados.

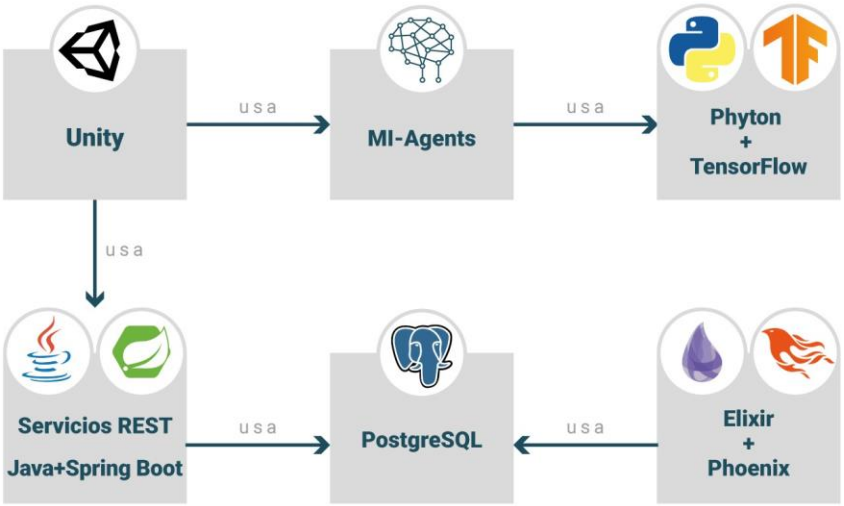


Figura 1: Diagrama de dependencias.

3. El Motor Unity

3.1. ¿Qué es Unity?

Unity es un motor para la creación de videojuegos que ayuda a los programadores en la ardua tarea que es el desarrollo de un videojuego.

Para esta tarea, Unity nos ofrece una gran ayuda, mediante una interfaz gráfica muy detallada y una serie de librerías que realizan gran parte del trabajo por nosotros, reduciendo así la complejidad del desarrollo. Entre las tareas en las que Unity nos ayuda, tenemos: simulación de la física del juego, adición de inteligencia artificial, renderizado de los gráficos, animaciones, efectos de sonido…

3.2. Elementos básicos de Unity

Para poder entender como Unity nos ayuda en el desarrollo de videojuegos, necesitamos tener una idea general de sus elementos más básicos:

1. **Escena**: Una escena (Figura 2) contiene el entorno y los menús del juego. Cada escena se podría entender como un nivel del juego.

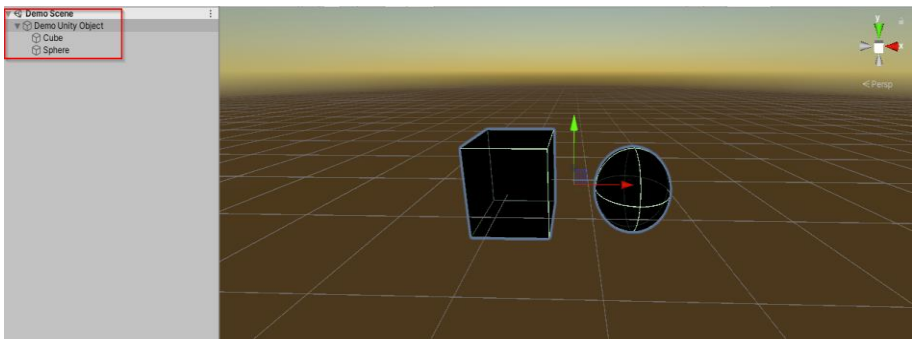


Figura 2: Representación de una escena en Unity

2. **Objeto de Unity**: Un objeto Unity (Figura 2) no es más que un contenedor al cual se le pueden añadir propiedades, lógica de juego y componentes. Todos los objetos de un juego son objetos Unity por defecto.

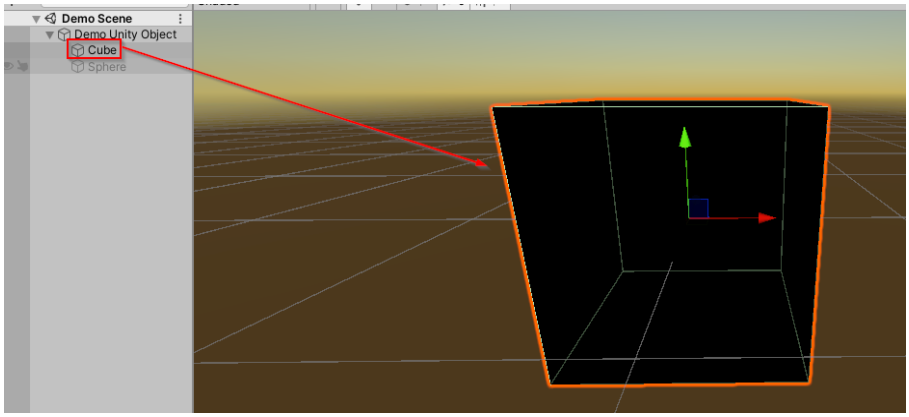


Figura 3: Representación de una escena en Unity

3. **Componentes de Unity**: Los componentes son elementos que se pueden añadir a un objeto Unity y que le añaden funcionalidad. Todos los objetos Unity tienen al menos un componente, "**Transform**", el cual viene por defecto en todos los Objetos de Unity y no se puede eliminar.

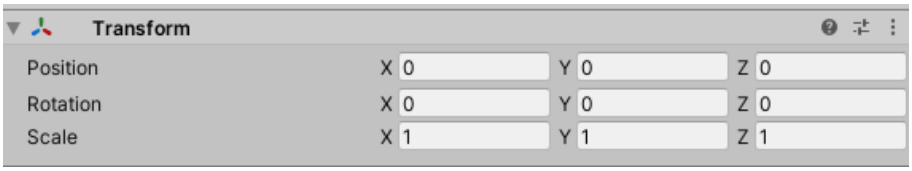


Figura 4: Componente Transform

Como podemos observar, este componente define la posición, rotación y escalado del objeto dentro de la escena de Unity.

Otros componentes importantes que pueden añadirse a un objeto son: un Colisionador (**Collider**), un renderizador (**Mesh Renderer**) y **scripts** de C#.

3.3. La Interfaz gráfica de Unity

Conociendo ya los elementos básicos de Unity, podemos pasar a entender su interfaz gráfica.

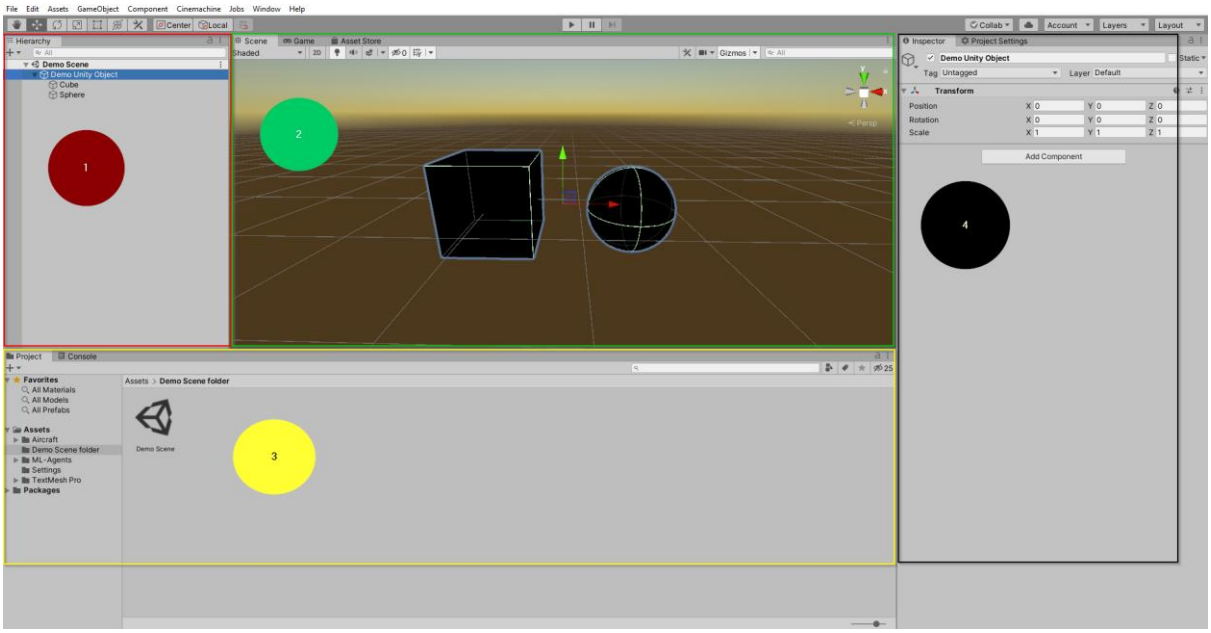


Figura 5: interfaz gráfica de Unity

La interfaz gráfica de unity contiene 4 elementos principales sobre los que trabajar:

1. **Hierarchy** (Jerarquía): En esta pestaña podemos ver un desglose de todos los objetos de Unity que hay en la escena.

2. **Scene** (escena): Aquí tenemos una vista anticipada del aspecto que tendrá la escena de nuestro juego. Es bastante similar a un editor de diseño gráfico (como Blender), aunque con menor funcionalidad, entre las posibles acciones que podemos realizar en esta pestaña: editar el tamaño, posición y rotación de un objeto; crear duplicados de objetos y cambiar la perspectiva

3. **Project** (proyecto): En esta zona tenemos un acceso rápido a los distintos directorios y archivos de nuestro proyecto.

4. **Inspector**: Al seleccionar un objeto en la pestaña *Scene*, podremos ver todas las propiedades de este objeto en la pestaña *Inspector*.

Con estos conceptos básicos sobre unity, ya tenemos las herramientas necesarias para adentrarnos en el mundo de machine learning y videojuegos.

4. Inteligencia artificial en videojuegos y el kit de herramientas ml-agents

4.1. ¿Usar juegos para entrenar inteligencia artificial?

El uso de juegos como punto de partida desde el cual desarrollar e investigar sobre inteligencia artificial tiene más de 70 años de antigüedad. **Claude Elwood Shannon** (1916-2001), conocido como *El padre de la teoría de la información*, fue uno de los pioneros en este sentido; publicando en 1950 una publicación titulada Programming a Computer for playing Chess[8], en la cual utilizaba una función de evaluación para intentar discernir qué jugador tenía ventaja en la posición actual, y de esa forma decidir la mejor jugada posible.

Pasarían otros 47 años, hasta que, en 1997, *Deep Blue*, la supercomputadora de IBM, derrotaría al antiguo campeón del mundo de ajedrez, Garry Kasparov: artículo de The New York Times del 12 de mayo 1997[9].

Otro detalle que tomar en cuenta, la mayoría de los juegos tienen puntuaciones y reglas que los convierten en entornos ideales para medir el progreso de una inteligencia artificial.

Para un ejemplo más moderno: en marzo 2016, el equipo de científicos e investigadores de Google, Deep Mind, crearon Alpha Go. Un proyecto que consiguió superar un gran obstáculo de la inteligencia artificial: ganar a un jugador profesional del popular juego de mesa asiático, Go, también llamado Baduk. Con reglas más simples que las del ajedrez, pero con una cantidad de posibles movimientos muy superior, los métodos utilizados para ganar a jugadores profesionales de ajedrez hasta la fecha, no servían en Go; sin embargo, mediante el uso de redes neuronales y árboles de búsqueda avanzados, *Alpha Go* ganó 4-1 a uno de los mejores jugadores de Go del momento, Lee Sedol. Y con el pasar el tiempo, *Alpha Go* se volvería más dominante todavía, con su versión entrenada sin partidas de humanos, Alpha Zero. Aquí se puede acceder a un documental sobre *Alpha Go*, publicado por el propio equipo de *Deep Mind*: enlace

4.2. ¿Qué es ml-agents?

Unity vió en su motor de videojuegos un entorno ideal para machine learning, por eso impulsaron la creación del kit de herramientas de aprendizaje automático de Unity “ml-agents” (Machine Learning Agents), un proyecto de código abierto, caracterizado por la posibilidad de utilizar el entorno de Unity para entrenar a “agentes inteligentes” mediante juegos y simulaciones.

Para más información se puede leer el documento hecho por los creadores de ml-agents: Unity: A General Platform for Intelligent Agents[10]

[8] Shannon, C. E. (1950). XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *41*(314), 256-275.
[9] Weber, B. (1997). Swift and slashing, computer topples Kasparov. *New York Times*, *12*, 262.
[10] Juliani, A., Berges, V. P., Vckay, E., Gao, Y., Henry, H., Mattar, M., & Lange, D. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

El kit nace con un principal objetivo: aportar a los investigadores de machine learning un entorno estable, fiable y altamente configurable, desde el cual realizar simulaciones para el entrenamiento de modelos de machine learning.

- Repositorio de github: ml-agents github[11]

[11] https://github.com/Unity-Technologies/ml-agents

5. Entendiendo ml-agents

Las piezas elementales de ml-agents son los **agentes**, los **comportamientos** y los **entornos.**

5.1. Agente y Entorno

Un **agente** es un objeto de unity, un actor independiente que interactúa con su **entorno**. Recolectando **observaciones** del entorno, realizando **acciones** y añadiendo **recompensas** positivas o negativas según el resultado de esas acciones. A cada agente se le asigna un **comportamiento**, varios agentes pueden tener el mismo comportamiento.

Un **entorno** es una escena de unity donde se posicionan a los distintos **agentes** y cualquier otra entidad con la que el agente deba interactuar.

En la Figura 5 y 6, nos encontramos dentro del proyecto de ejemplo 3Dball, y en el podemos ver al agente y al entorno, respectivamente.

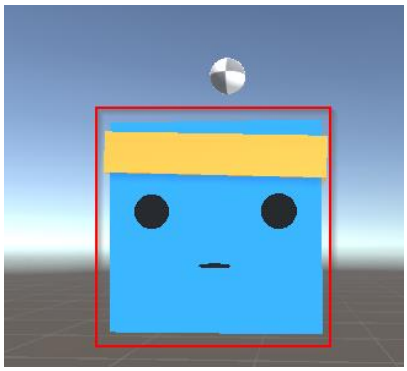


Figura 6: el cubo es el agente del entorno

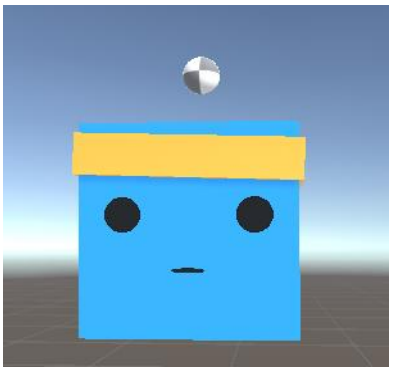


Figura 7: La esfera y el cubo juntos forman parte del entorno

5.2. Comportamiento

Un **comportamiento** determina cómo el **agente** toma decisiones. Entre los elementos del comportamiento encontramos:

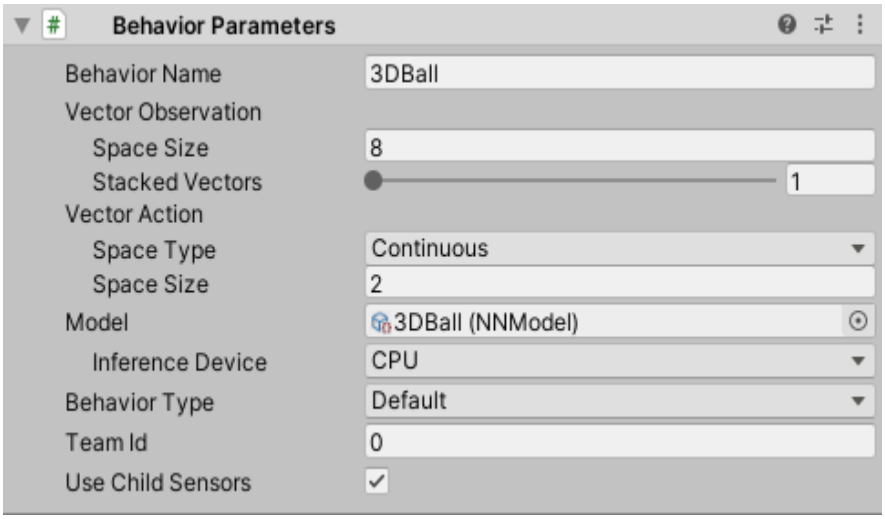


Figura 8: Behavior Parameters de un agente en 3Dball

- **Vector de Observación**: antes de cada decisión el agente recolecta información del estado de su entorno. El vector es un array de *floats* que contienen la información del entorno.

  La información que se observará es decidida de antemano por el investigador.

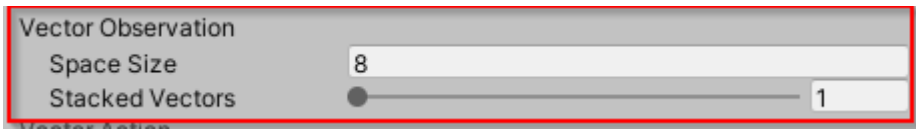


Figura 9: Parámetros del vector de observación

- **Vector de Acción:** es el vector de instrucciones (array de *floats*) que el agente utilizará para tomar decisiones en el entorno. Cada rama hace referencia a un tipo distinto de acción a tomar.

  Los valores pueden ser discretos (por ejemplo: decidir entre izquierda y derecha), o continuos (decidir la fuerza que ejercer en un salto).

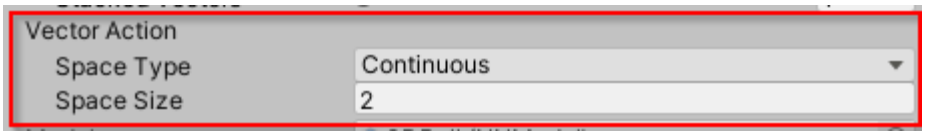


Figura 10: Parámetros del vector de acción

- **Modelo:** aquí podemos asignarle un modelo, previamente entrenado, a un agente. El agente utilizará este modelo para tomar decisiones (inferencia) si no se encuentra entrenando.

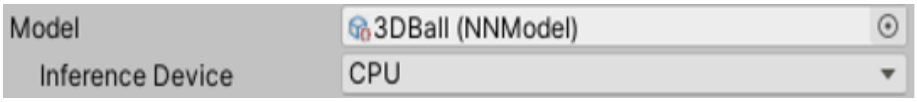


Figura 11: Modelo del comportamiento

- **Resto de parámetros:** el resto de los parámetros se dejarán con su valor por defecto en todo momento. *Behaviour Type,* nos permite elegir si el agente tomará decisiones provenientes del modelo o decisiones que hayamos programado nosotros; *Team Id*, se utiliza para distinguir a los distintos agentes cuando estos se separan en equipos; y, *Use Child Sensors*, en el caso de que un agente tenga otro agente dentro de él, seleccionar esta casilla hará que el agente padre añada a su vector de observación las observaciones del agente hijo.

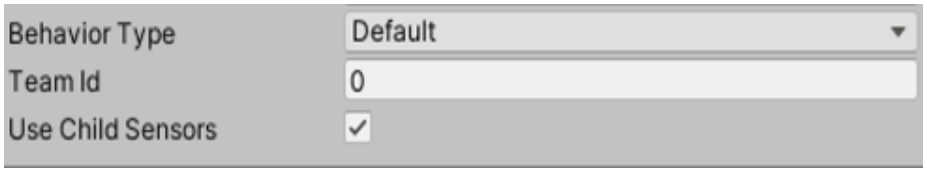


Figura 12: resto de parámetros del comportamiento del agente

5.3. La academia

La **academia** se encarga, durante el entrenamiento, de la comunicación entre la API de **python** y los **agentes** en Unity, así como de que los **agentes** entrenen de forma sincronizada. Solo existe una academia común a la que todos los agentes tienen acceso.

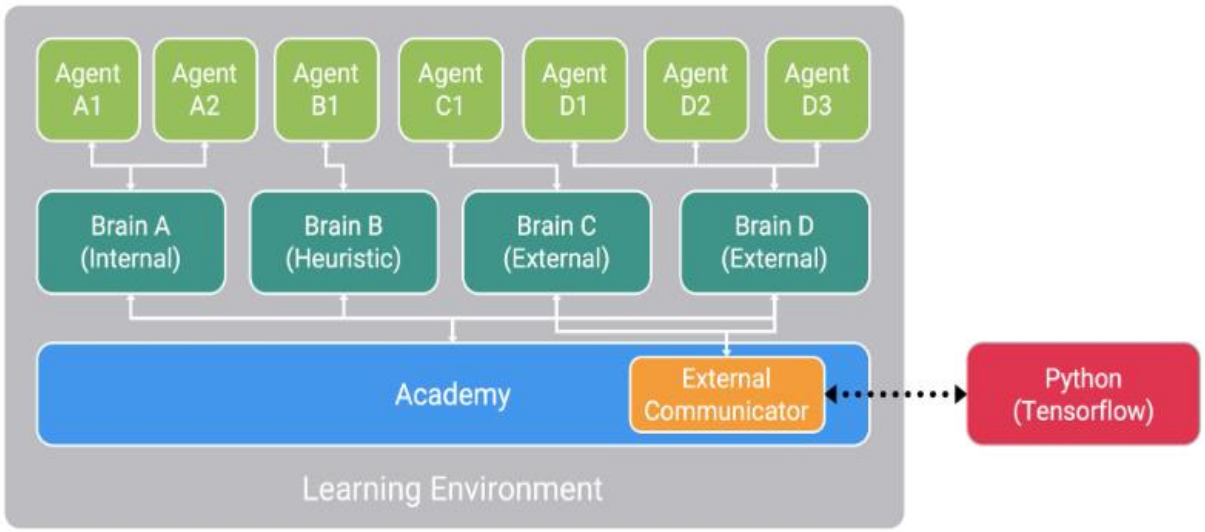


Figura 13: estructura academia de ml-agents. Imagen obtenida del [blog de unity 3d](https://unity3d.com/how-to/unity-machine-learning-agents)[12]

5.4. Entrenando agentes y parámetros de configuración

En ml-agents para poder entrenar a los agentes necesitamos tener Python instalado. Aunque no es obligatorio, recomiendo instalar [Anaconda](https://www.anaconda.com/products/individual)[13], la versión que tiene python 3.7 integrado nos servirá.

[12] https://unity3d.com/how-to/unity-machine-learning-agents
[13] https://www.anaconda.com/products/individual

Una vez tenemos python instalado. Abriremos el ejemplo 3D BALL de ml-agents:

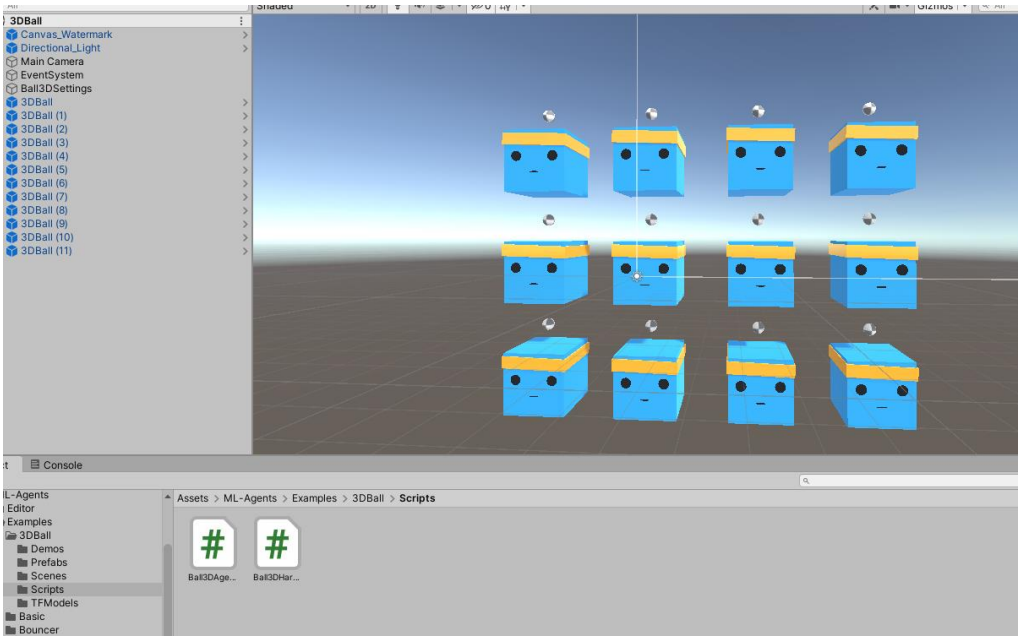


Figura 14: 3D Ball en Unity

Y, por otro lado, desde una ventana de comandos deberemos ejecutar el comando:

`mlagents-learn` **`<dirección trainer_config.yaml del proyecto Unity>`** `--run-id 3dball-training-01`

El archivo trainger_config.yaml se encuentra dentro del proyecto de ml-agents, bajo el subdirectorio /training/config

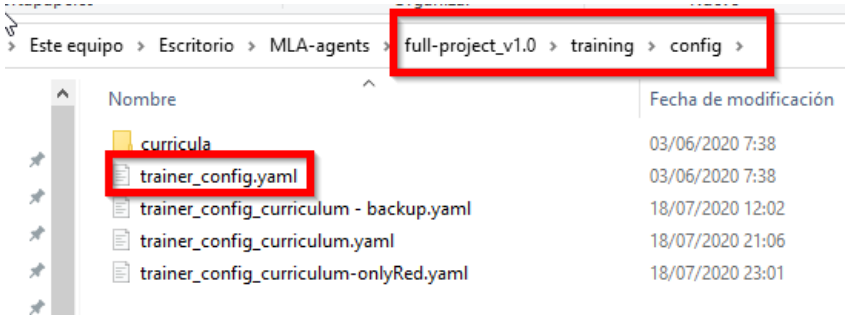


Figura 15: localización del archivo trainer_config.yaml

Una vez ejecutado el comando, si todo va bien, veremos algo similar a lo siguiente:

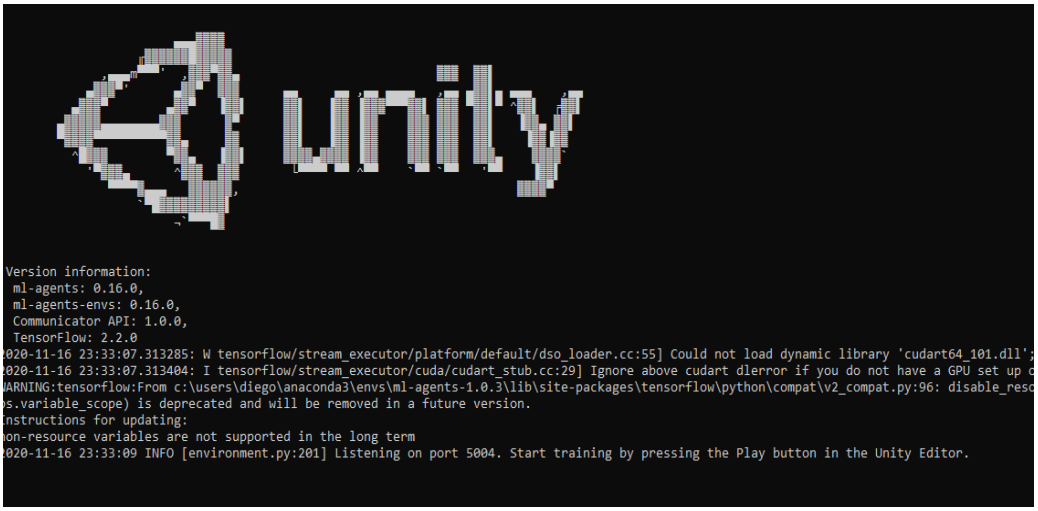


Figura 16: salida de consola al ejecutar comando mlagents-learn

Llegados a este punto, si le damos al botón de “Play” en Unity, nuestros agentes empezarán a entrenar

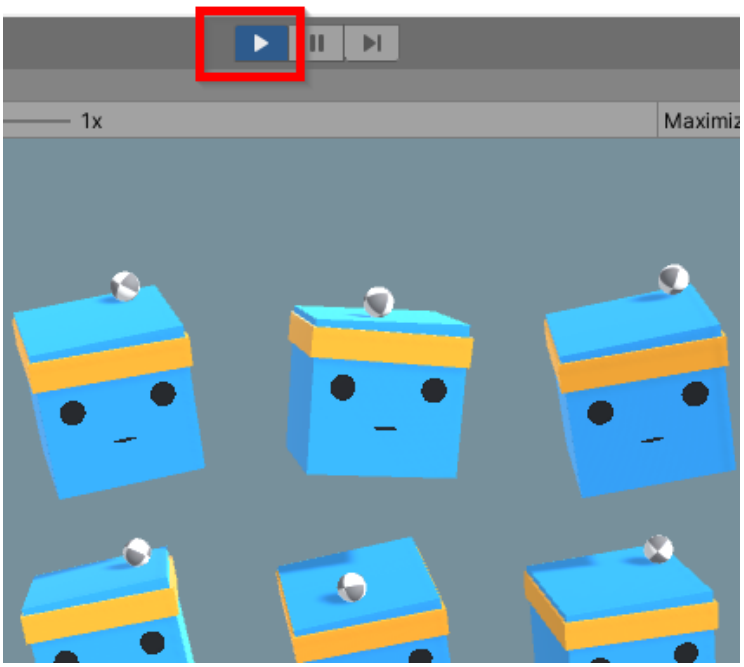


Figura 17: 3D Ball, agentes entrenando

Podemos ver el progreso de este entrenamiento en la ventana de comandos

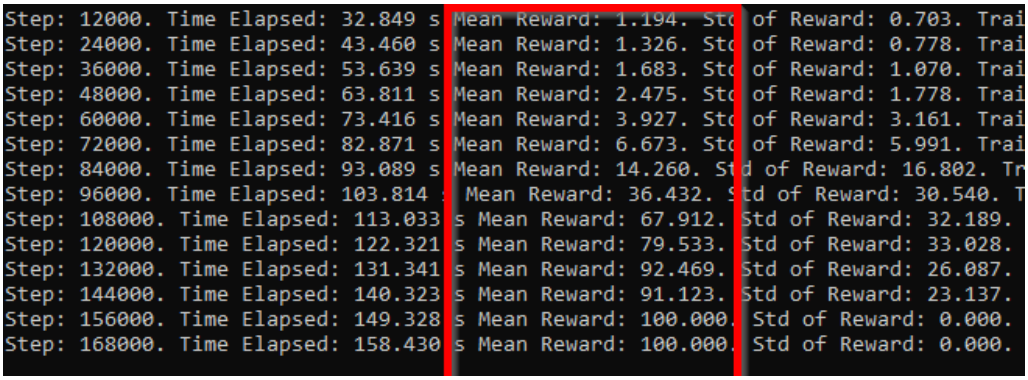


Figura 18: salida de consola durante el entrenamiento

El valor más importante para conocer el desempeño de nuestro agente es la **recompensa media**, si este valor va incrementando de forma consistente, significa que nuestro agente cada vez desempeña mejor aquellas tareas que le otorgan recompensas positivas, en el caso de este ejemplo, mientras la bola no caiga el agente recibirá una recompensa positiva.

Debido a la sencillez de este ejemplo, en tan solo 2 minutos, los agentes ya han aprendido a balancear la bola por completo.

Para cualquier duda en el proceso de entrenamiento, recomiendo leer la documentación[14] de ml-agents, en particular la sección "*Training with mlagents-learn*".

5.5. Evaluando el entrenamiento con TensorBoard

TensorBoard nos permite ver estadísticas de los entrenamientos de nuestros distintos modelos, también podemos añadir durante el entrenamiento nuestras propias estadísticas. Por ejemplo, podríamos añadir el número de saltos que un agente da durante un entrenamiento.

Dentro de las muchas estadísticas que podemos observar en TensorBoard, la de mayor interés es la de "Cumulative Reward" que nos indica cómo ha evolucionado la recompensa media de los agentes en entrenamiento a lo largo del tiempo.

[14] https://github.com/Unity-Technologies/ml-agents/blob/release_10_docs/docs/Getting-Started.md

Figura 19: gráfico de recompensa acumulada en TensorBoard

6. Ejemplos de proyectos por defecto de ml-agents

Dentro del repositorio de GitHub de ml-agents, podemos encontrar varios ejemplos interesantes que nos pueden servir como base para las distintas líneas de investigación que queramos llevar a cabo, aquí se mencionan 4 ejemplos:

3DBall: El agente intenta balancear una bola, evitando así que caiga.

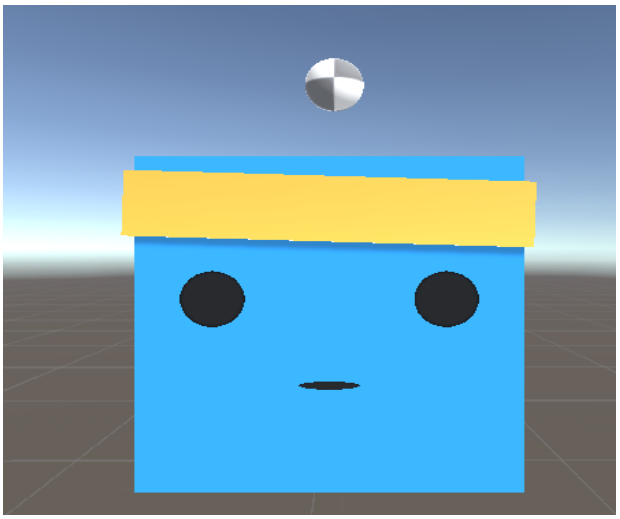


Figura 20: 3D Ball de ml-agents

Wall Jump: El agente debe saltar un muro de altura variable y tocar la zona blanca. Si la altura del muro es muy grande, el agente deberá mover una plataforma verde primero, la cual le servirá de apoyo.

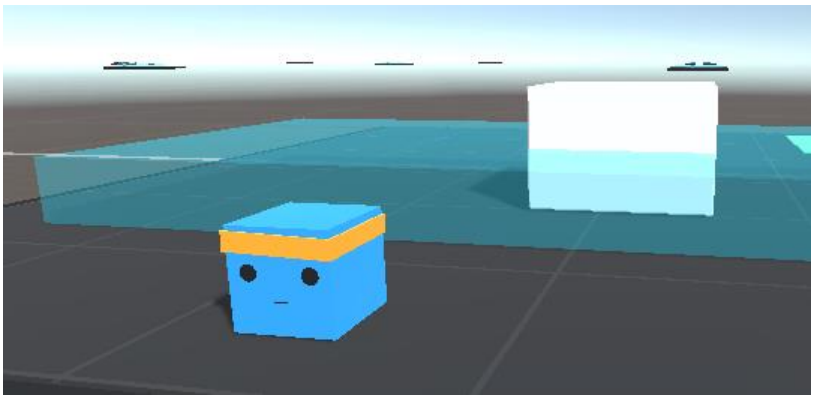


Figura 21: Wall Jump de ml-agents

Hallway: El agente verá, de forma aleatoria, una “X” o una “O” en una pared inicial, y dependiendo del símbolo que observe deberá escoger el camino correcto para llegar al mismo símbolo en la siguiente pared.

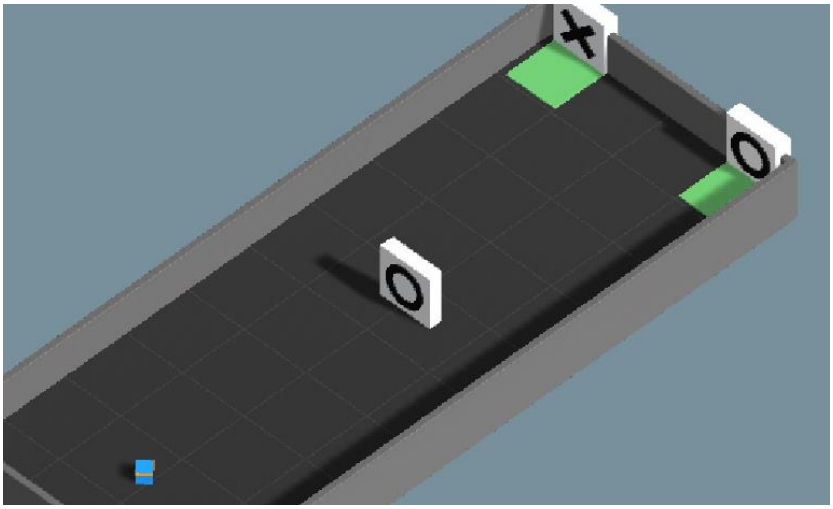


Figura 22: Hallway de ml-agents

Soccer: Los agentes forman equipos de dos e intentan anotar goles en la meta del equipo contrario

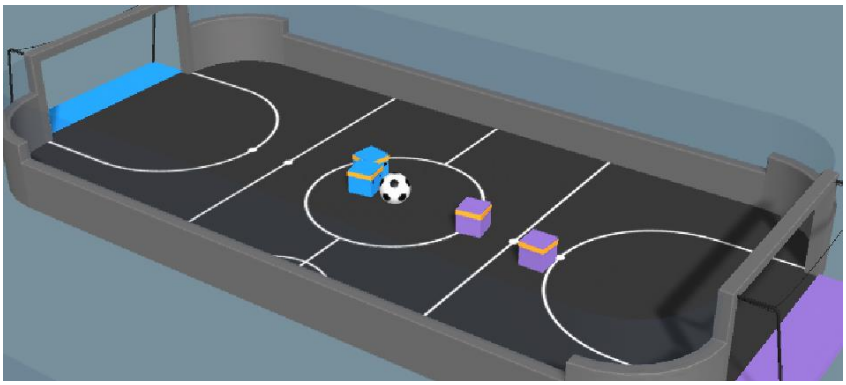


Figura 23: Soccer de ml-agents

Hay muchos más proyectos de ejemplo dentro del repositorio de ml-agents, los ejemplos aquí nombrados solo tienen como objetivo mostrarle al lector la amplia variedad de posibilidades que ofrece el kit de herramientas.

7. Ejemplos de entorno con aviones

7.1. Entorno base sobre el que se basarán las simulaciones

Para estos ejemplos utilizaremos como base, el entorno y los *assets* que vienen en el curso de Immersive Limit[15], Ai-Flight[16], modificando la lógica del juego, así como los parámetros de configuración, allá donde sea necesario. En este curso se nos enseña paso a paso, cómo crear un videojuego de aviones, donde los adversarios sean IA que hayan aprendido el juego a través del uso de ml-agents.

El entorno donde se colocará a los agentes se denomina **AircraftArea:**

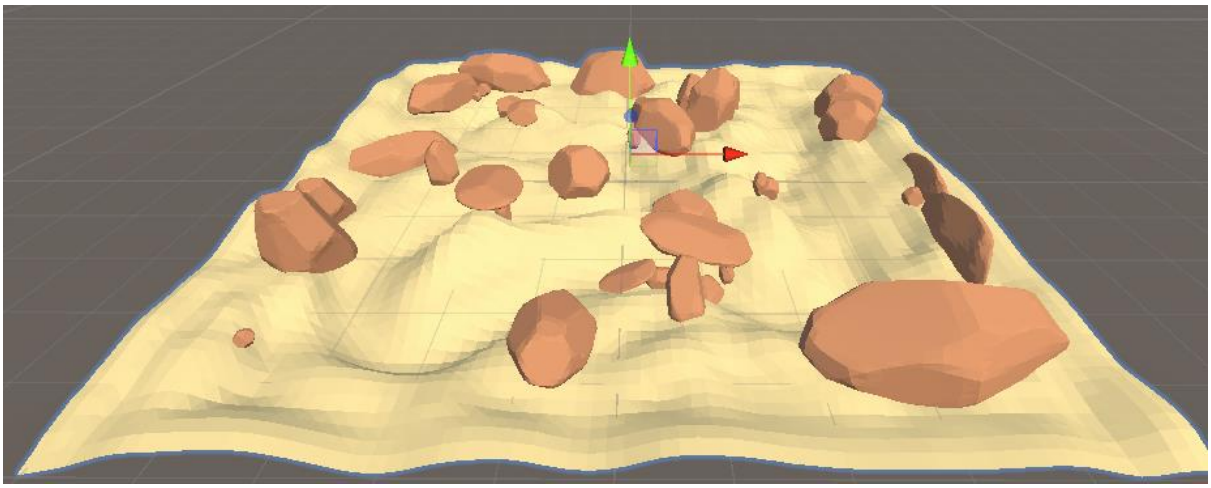


Figura 24: AircraftArea, entorno donde entrenarán los agentes

Se trata de un objeto de Unity con la siguiente estructura:

Figura 25: estructura del área dentro de Unity

En el recuadro rojo podemos ver los elementos base de **DesertArea (AircraftArea)**:

[15] https://www.immersivelimit.com/
[16] https://www.udemy.com/course/ai-flight/

desertTerrain: Es el terreno arenoso.

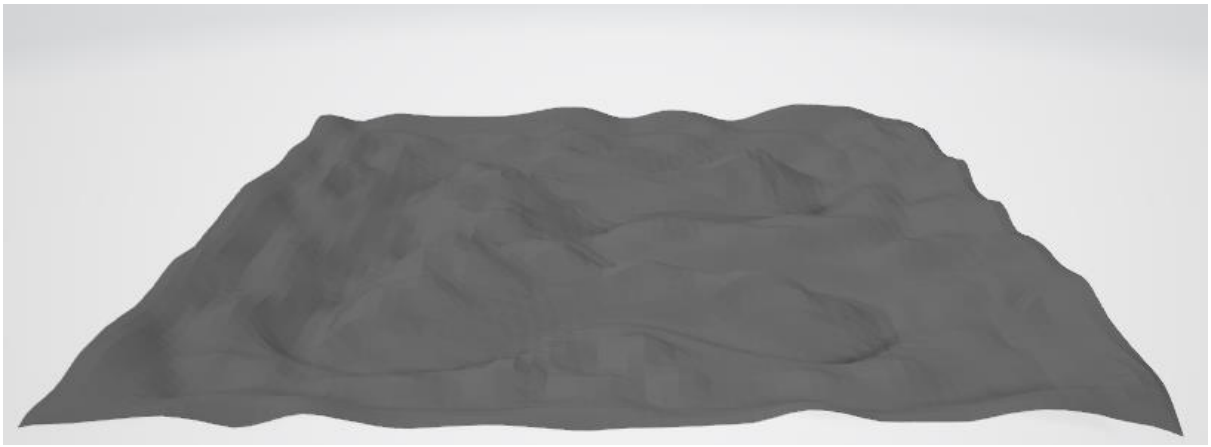


Figura 26: desertTerrain

Environment: Contiene una lista de todas las piedras/obstáculos situadas en el terreno

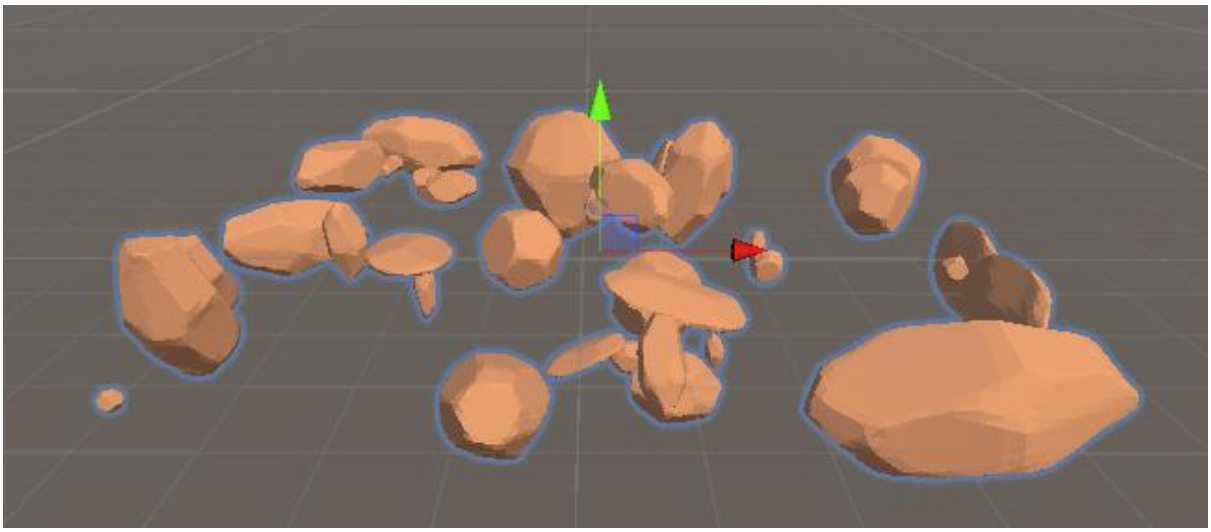


Figura 27: Environment

Boundaries: Límites invisibles dentro del terreno para evitar que los aviones vuelen en cualquier dirección de forma indeterminada.

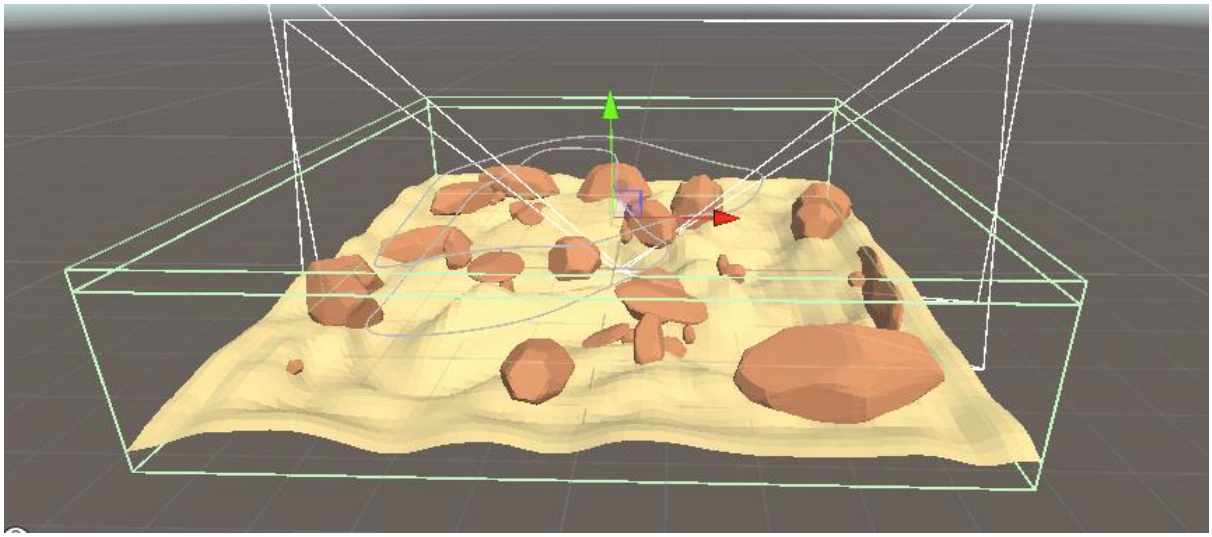


Figura 28: Boundaries

Luego tenemos el recorrido (**RacePath**) que seguirán nuestros agentes, hay caminos diferentes según el color del agente:

Figura 29: objetos unity para los recorridos

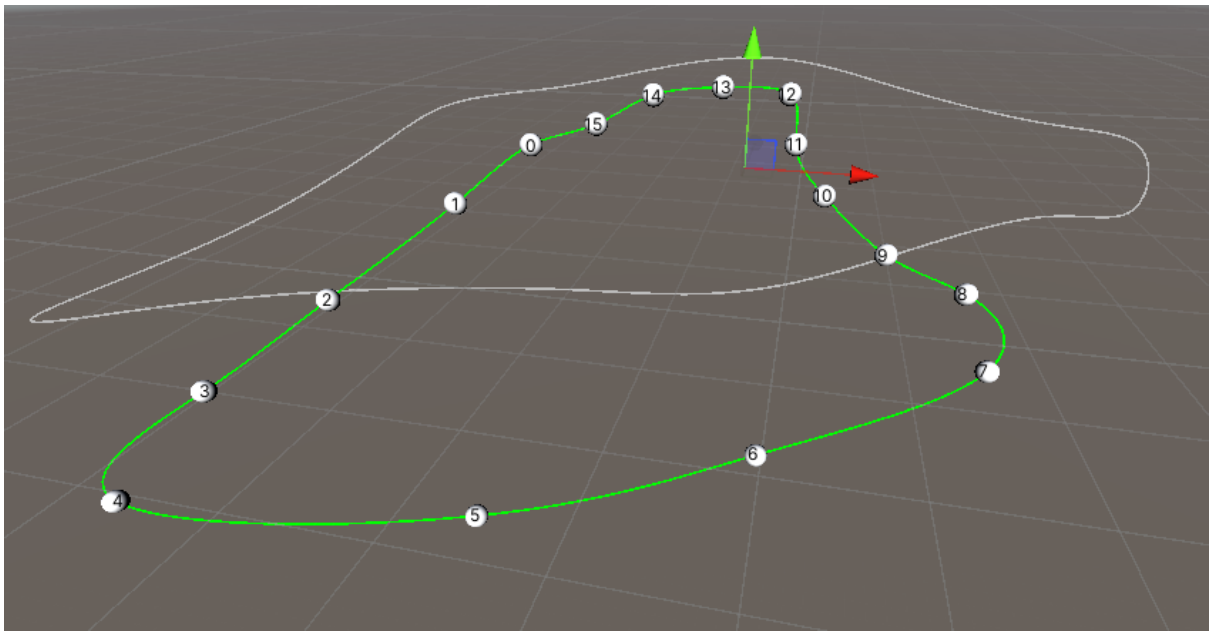


Figura 30: forma de los recorridos

Los puntos que debe atravesar el agente se generan en el centro de las esferas numeradas a lo largo del recorrido.

Y, por último, tenemos a los agentes en sí, aviones rojos o azules.

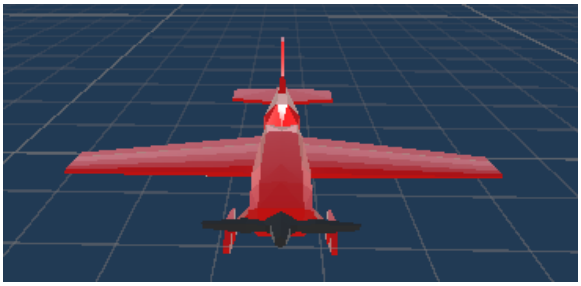


Figura 31: agente rojo

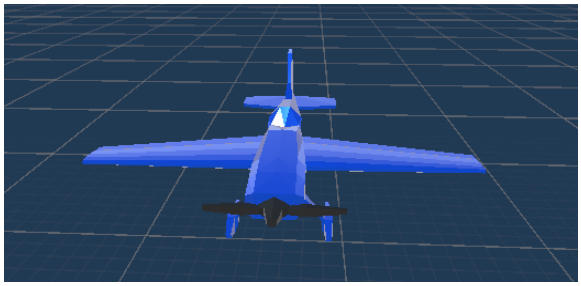


Figura 32: agente azul

7.2. Lógica del entorno

Script **AircraftArea.cs:** Este script se encarga, primordialmente, de la creación de los recorridos que seguirán los agentes al principio de la simulación, y también se encarga de regenerar a los agentes que hayan sido destruidos por chocar con algún obstáculo.

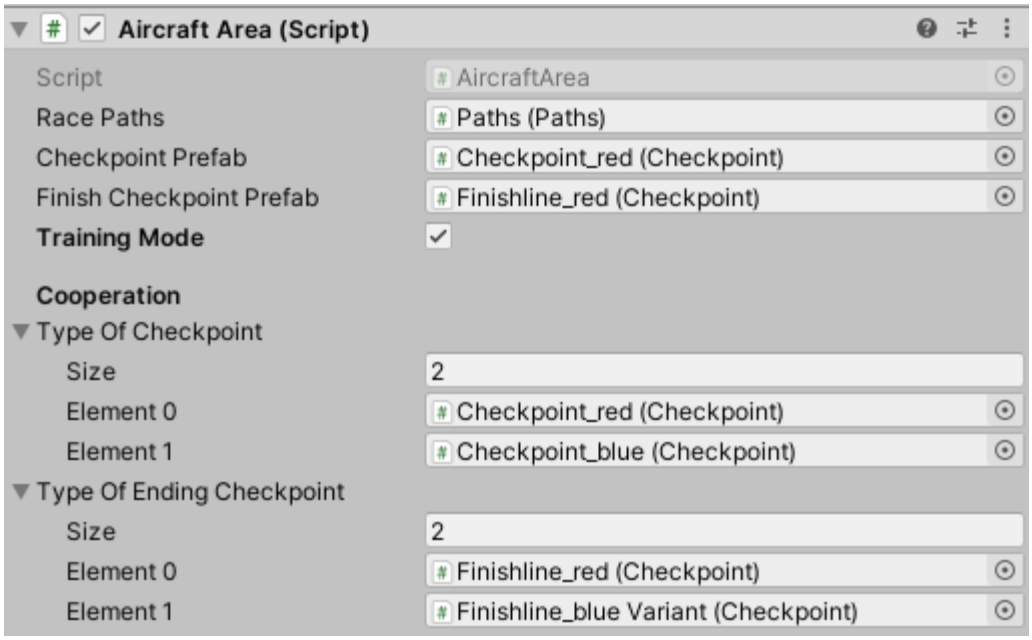


Figura 33: parámetros configurables del script Aircraft Area

Entre los objetos que debemos referenciar a este script desde el inspector de unity tenemos:

Race Paths: el objeto contiene los distintos recorridos que se generarán en el entorno.

Checkpoint Prefab y Finish Checkpoint Prefab: guardan referencias a los modelos de los objetos **Checkpoint** que se utilizarán por defecto si no se especifica ningún equipo.

Training Mode: casilla seleccionable para saber si estamos entrenando a los agentes.

Cooperation: en esta sección tenemos referencias a los objetos **Checkpoint,** son los puntos que el entorno debe generar a lo largo del recorrido (Figura 29) y que luego los agentes deben cruzar.

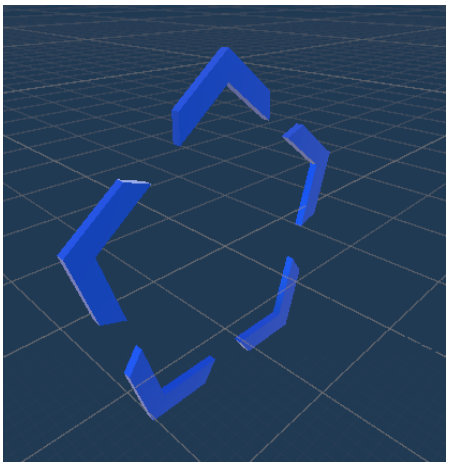


Figura 34: checkpoint_blue

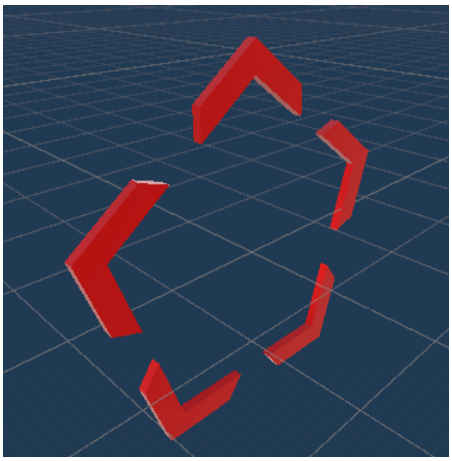


Figura 35: checkpoint _red

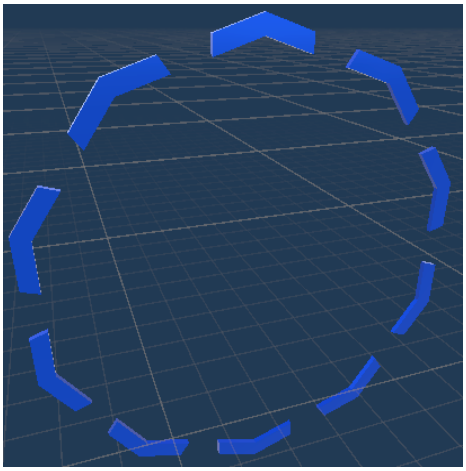


Figura 36: finishline_blue

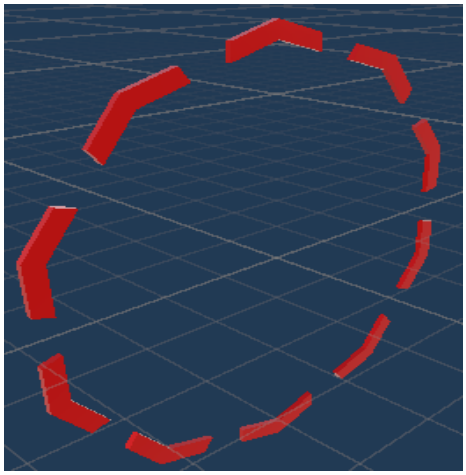


Figura 37: finishline_red

Nada más iniciar la simulación se crea una lista con todos los agentes dentro del entorno

```
private void Awake()
{
    // Find all aircraft agents in the area
    AircraftAgents = transform.GetComponentsInChildren<AircraftAgent>().ToList();
    Debug.Assert(AircraftAgents.Count > 0, "No AircraftAgents found");
}
```

Figura 38: código inicial del script Aircraftarea.cs

Después inicializamos los **Checkpoints** a lo largo del recorrido mediante un bucle *for*, guardando en un **Dictionary** (equivalente a HashMap en Java) el índice del color de cada **Checkpoint** y una referencia al objeto **Checkpoint** en cuestión.

```
private void Start()
{
    // Create checkpoints along the race path
    Debug.Assert(racePaths != null, "Race Paths were not set");
    checkpointsDictionary = new Dictionary<int, List<Checkpoint>>();
    listOfRacePaths = new List<CinemachineSmoothPath>();
    listOfRacePaths = racePaths.listOfRacePaths;
    for (int j = 0; j < listOfRacePaths.Count; j++) {
        CinemachineSmoothPath racePath = listOfRacePaths[j];
        checkpointsDictionary[j] = new List<Checkpoint>();
        int numCheckpoints = (int)racePath.MaxUnit(CinemachinePathBase.PositionUnits.PathUnits);
        for (int i = 0; i < numCheckpoints; i++)
        {
            // Instantiate either a checkpoint or finish line checkpoint
            Checkpoint checkpoint;
            if (i == numCheckpoints - 1) checkpoint = Instantiate<Checkpoint>(typeOfEndingCheckpoint[j]);
            else
                checkpoint = Instantiate<Checkpoint>(typeOfCheckpoint[j]);

            // Set the parent, position, and rotation
            checkpoint.transform.SetParent(racePath.transform);
            checkpoint.transform.localPosition = racePath.m_Waypoints[i].position;
            checkpoint.transform.rotation = racePath.EvaluateOrientationAtUnit(i,
                CinemachinePathBase.PositionUnits.PathUnits);

            // Add the checkpoint to the list
             checkpointsDictionary[j].Add(checkpoint);

        }
    }
}
```

Figura 39: método Start() de AircraftArea

También hay un método que se encarga de regenerar a los agentes cuando la simulación lo requiera **ResetAgentPosition.** Usualmente, los agentes se regeneran en el último punto que alcanzaron, pero en el entrenamiento es posible hacer que la regeneración ocurra en un punto aleatorio del recorrido.

```
public void ResetAgentPosition(AircraftAgent agent, bool randomize = false)
{
    List<Checkpoint> auxiliarCheckPointsList = new List<Checkpoint>();
    CinemachineSmoothPath racePath = null;

    if (randomize)
    {
        // Pick a new next checkpoint at random
        agent.NextCheckpointIndex = Random.Range(0, auxiliarCheckPointsList.Count);
    }

    // Set start position to the previous checkpoint
    int previousCheckpointIndex = agent.NextCheckpointIndex - 1;
    if (previousCheckpointIndex == -1)
    {
        previousCheckpointIndex = auxiliarCheckPointsList.Count - 1;
    }

    float startPosition = racePath.FromPathNativeUnits(previousCheckpointIndex,
        CinemachinePathBase.PositionUnits.PathUnits);

    // Convert the position on the race path to a position in 3d space
    Vector3 basePosition = racePath.EvaluatePosition(startPosition);

    // Get the orientation at that position on the race path
    Quaternion orientation = racePath.EvaluateOrientation(startPosition);

    // Calculate a horizontal offset so that agents are spread out
    Vector3 positionOffset = Vector3.right * (AircraftAgents.IndexOf(agent) - AircraftAgents.Count / 2f)
        * UnityEngine.Random.Range(9f, 10f);
    // Set the aircraft position and rotation
    agent.transform.position = basePosition + orientation * positionOffset;
    agent.transform.rotation = orientation;
}
```

Figura 40: método ResetAgentPosition() de AircraftArea

La posición y la orientación del agente se asignan en las dos últimas líneas del código.

Los objetos **Checkpoint** contienen un script **Checkpoint.cs,** la lógica de este script variará según en el ejemplo. Inicialmente, el script solo guarda una referencia al equipo al que pertenece el **Checkpoint**: azul o rojo.

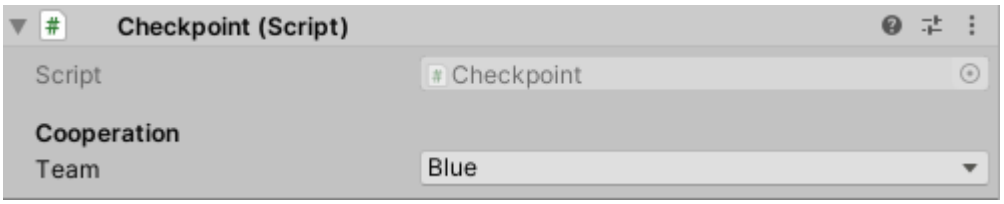


Figura 41: parámetros configurables del script Checkpoint.cs

7.3. Lógica de los agentes

Los agentes tienen como objetivo avanzar cruzando por dentro de los distintos puntos a lo largo del recorrido.

Cada agente cuenta con un script **AircraftAgent.cs**

Behavior parameters del agente:

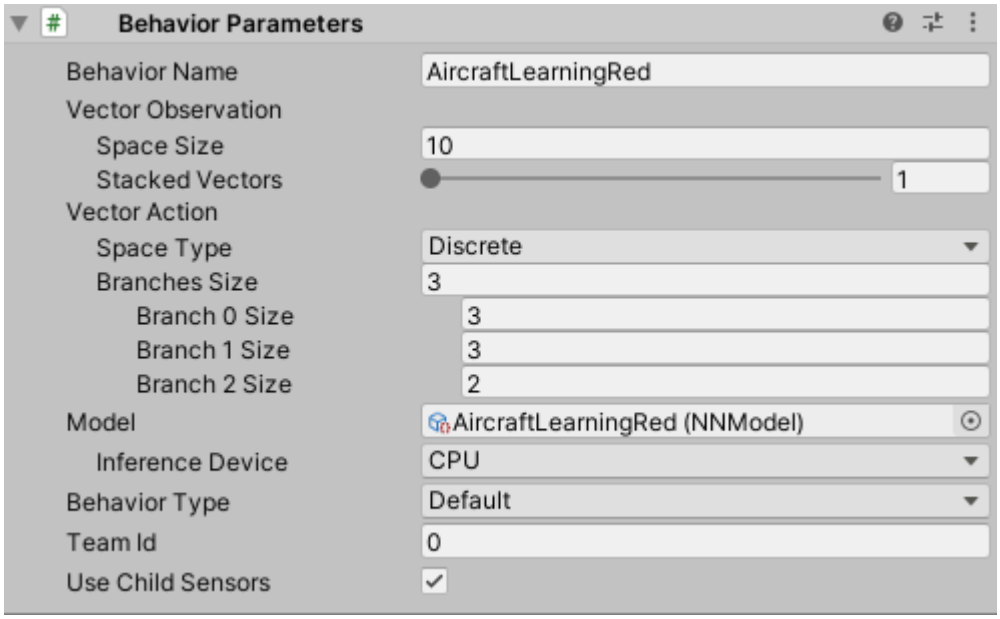


Figura 42: configuración del behavior parameters de un agente avión

En el vector de observación tenemos dos valores a tomar en cuenta:

1. El **número de *Rayos de Percepción* añadidos** al agente, así como los distintos de categorías de objetos que pueden detectar estos rayos. Más adelante, veremos cómo se añadan estos *Rayos de Percepción* al agente.
2. Las **observaciones** que añadimos **dentro del método CollectObservations()** del agente, las cuales serán:
    a. Vector de velocidad del agente.
    b. Vector apuntando al próximo punto al que debe dirigirse el agente.
    c. Vector de orientación del siguiente punto.
    d. El equipo al que pertenece el agente.

7.3.1. Rayos de Percepción

El agente consta de un total de 31 Rayos de Percepción (**RayPerception)** que perciben todo aquello que esté a una distancia menor o igual a 300 unidades.

Estos Rayos están distribuidos entre 5 objetos diferentes: uno por cada ángulo de dispersión.

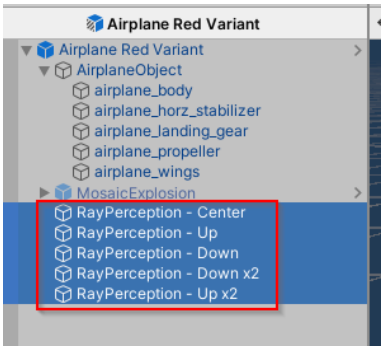


Figura 43: objetos conteniendo los rayos de percepción del agente

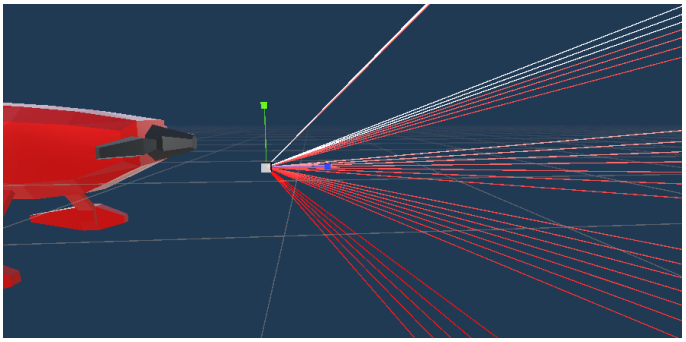


Figura 44: rayos de percepción del agente

El conjunto de los rayos se abre en un arco de 90º

- Los rayos de los extremos se encuentran a ±45º de los rayos centrales, con solo **5 rayos en total**.
- Los rayos intermedios se encuentran a ±18º de los rayos centrales, con **7 rayos en total**.
- Los rayos centrales son **7 en total**.

AircrafAgent.cs en el inspector de unity:

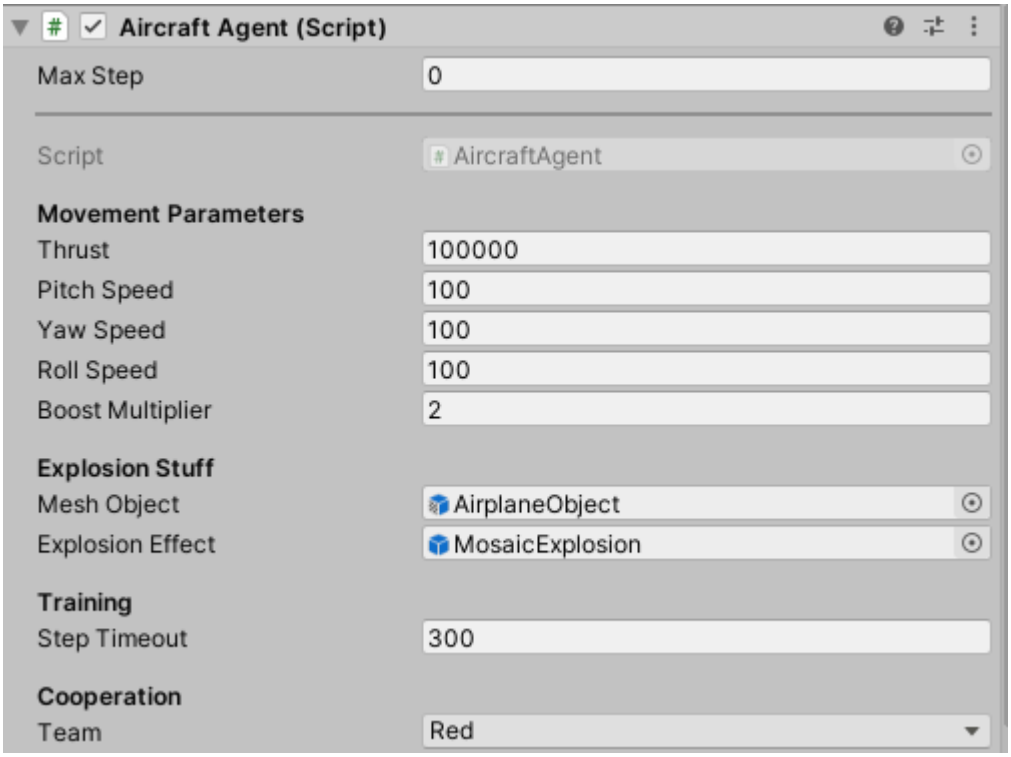


Figura 45: configuración del script AircraftAgent.cs

- **Max step:** El número de "pasos" de entrenamiento máximos hasta que se detenga el agente, este parámetro se configura desde el archivo **trainer_config.yaml.**
- **Movement Parameters:** Configuración por defecto de las distintas velocidades del agente.
    - **Thrust:** Fuerza de la propulsión del avión
    - **Pitch:** Cabeceo del avión (giro eje x)
    - **Yaw:** Guiñada del avión (giro eje z)
    - **Roll:** Alabeo del avión (giro eje y)
    - **Boost:** Multiplicador de la propulsión
- **Explosion Stuff:** aquí se guardan referencias para los efectos visuales de las explosiones.
- **Training:**
    - **Step Timeout**: Número de pasos que dará el agente en el entrenamiento antes de que vuelva a repetirlo desde el principio.
- **Cooperation:**
    - **Team:** Equipo al que pertenece el agente. También hubiera sido posible utilizar la variable **TeamId** del **Behavior Parameters** del agente.

7.3.2. Métodos heredados del objeto Agent.cs de ml-agents:

Método Initialize()

```
public override void Initialize()
{
    area = GetComponentInParent<AircraftArea>();
    rigidbody = GetComponent<Rigidbody>();
    trail = GetComponent<TrailRenderer>();

    // Override the max step set in the inspector
    // Max 5000 steps if training, infinite steps if racing
    MaxStep = area.trainingMode ? 5000 : 0;
}
```

Figura 46: Método para la inicialización del agente

Al inicializar el agente, únicamente le añadimos los objetos del entorno de Unity que necesitará más adelante: El entorno (**desertArea**), su componente **RigidBody** y el **TrailRenderer**.

Método OnEpisodeBegin()

```
public override void OnEpisodeBegin()
{
    // Reset the velocity, position, and orientation
    rigidbody.velocity = Vector3.zero;
    rigidbody.angularVelocity = Vector3.zero;
    trail.emitting = false;
    area.ResetAgentPosition(agent: this, randomize: area.trainingMode);
    //update checkpoint list
    getCheckPointList();
    // Update the step timeout if training
    if (area.trainingMode) nextStepTimeout = StepCount + stepTimeout;
}
```

Figura 47: Método ejecutado al inicio de cada episodio de entrenamiento

Es el método que se llama cada vez que el agente inicia un episodio de entrenamiento nuevo. Se reinicia su posición y sus parámetros iniciales. También debemos actualizar la variable **nextStepTimeout**, esta variable contiene el valor del paso máximo de entrenamiento, si se llega a este valor, se terminará el episodio de entrenamiento.

Método OnActionReceived()

```
public override void OnActionReceived(float[] vectorAction)
{
    // Read values for pitch and yaw
    pitchChange = vectorAction[0]; // up or none
    if (pitchChange == 2) pitchChange = -1f; // down
    yawChange = vectorAction[1]; // turn right or none
    if (yawChange == 2) yawChange = -1f; // turn left

    // Read value for boost and enable/disable trail renderer
    boost = vectorAction[2] == 1;
    if (boost && !trail.emitting) trail.Clear();
    trail.emitting = boost;

    if (frozen) return;

    ProcessMovement();

    //Rewards section if we are in training mode
    if (area.trainingMode)...
}
```

Figura 48: Método ejecutado al recibir un nuevo vector de acciones

Aquí es donde el agente recibe las acciones a realizar, son los valores finales obtenidos por la red neuronal para este paso de entrenamiento. En los ejemplos iniciales, las acciones recibidas son un array de 3 *floats*, que hacen referencia a las 3 ramas del Vector de Acción:

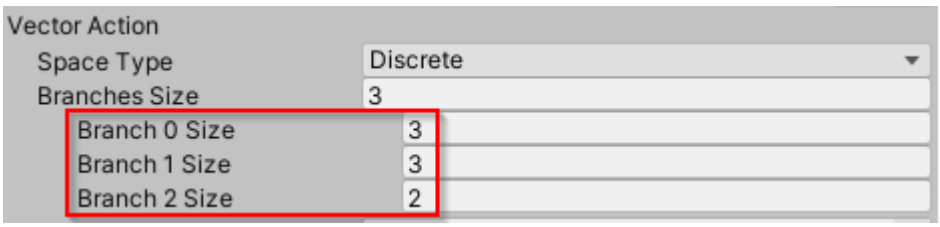


Figura 49: Configuración del Vector Action en el Behavior Parameters del agente

La **primera rama** es para el **Pitch** del avión (**Cabeceo**)

La **segunda rama** es para el **Yaw** del avión (**Guiñada**)

La **tercera rama** es para activar el **Boost** (Multiplicador de propulsión)

Las dos primeras ramas del vector pueden tener tres posibles valores diferentes: {0, 1, 2}, donde:

- 0: no hacer nada
- 1: movimiento en el eje de valor positivo
- 2: se traducirá en un movimiento en el eje de valor negativo

Y la tercera rama puede tomar dos posibles valores diferentes: {0, 1}, donde:

- 0: desactivar **Boost**
- 1: activar **Boost**

Método CollectObservations():

```
public override void CollectObservations(VectorSensor sensor)
{
    List<Checkpoint> checkPoints = area.checkpointsDictionary[(int)this.team];
    // Observe aircraft velocity (1 vector3 = 3 values)
    sensor.AddObservation(transform.InverseTransformDirection(rigidbody.velocity));

    // Where is the next checkpoint? (1 vector3 = 3 values)
    sensor.AddObservation(VectorToNextCheckpoint());

    // Orientation of the next checkpoint (1 vector3 = 3 values)
    Vector3 nextCheckpointForward = checkPoints[NextCheckpointIndex].transform.forward;
    sensor.AddObservation(transform.InverseTransformDirection(nextCheckpointForward));

    //Team or tag
    sensor.AddObservation((int) this.team);

    // Total Observations = 3 + 3 + 3 + 1 = 10
}
```

Figura 50: método donde se definen las observaciones del entorno que recolectará el agente

En este método, recolectamos las observaciones (Las entradas de la red neuronal). Este método es llamado en cada paso de entrenamiento.

Aparte de las observaciones obtenidas de los *Rayos de Percepción* que tenga el agente. Aquí recolectaremos observaciones adicionales, en este caso, se trata de 10 valores (*floats*) extras:

- Vector de velocidad del agente (Vector 3D = 3 valores)
- Vector apuntando al próximo punto (Vector 3D = 3 valores)
- Vector de orientación del siguiente punto (Vector 3D = 3 valores)
- El equipo al que pertenece el agente (número entero = 1 valor)

No hay ninguna regla específica que nos diga cuántas ni cuáles deberían ser las observaciones de un agente, pero mientras más sean, más tardará el entrenamiento. Así que hay que llegar a un compromiso.

7.3.3. Recompensas durante el entrenamiento

Para que el entrenamiento se realice correctamente, es necesario definir adecuadamente las recompensas que el agente recibirá según las acciones que tome. Una vez más, no hay reglas específicas para esta asignación de recompensas, ni a su valor, ni a cuándo deben ser otorgadas. Hay que llegar a un compromiso.

Las recompensas programadas variarán según el ejemplo, pero inicialmente estas son las recompensas:

1. Una pequeña recompensa negativa cada paso de entrenamiento: esto es para forzar al agente a tomar decisiones en vez de quedarse quieto.

```
// Small negative reward every step
AddReward(-1f / MaxStep);
```

Figura 51: recompensa negativa para forzar que el agente tome decisiones

2. Una recompensa negativa grande de -0.5, si el agente agota su tiempo de entrenamiento para este episodio.

```
// Make sure we haven't run out of time if training
if (StepCount > nextStepTimeout)
{
    AddReward(-.5f);
    EndEpisode();
}
```

Figura 52: recompensa negativa si al agente se le termina el tiempo del entrenamiento

3. Una recompensa positiva de 0.5 cuando el agente atraviesa un punto de su recorrido, también se le recompensa de forma indirecta aumentando la duración del episodio de entrenamiento actual.

```
//Reward for getting a Checkpoint
AddReward(.5f);
nextStepTimeout = StepCount + stepTimeout;
```

Figura 53: recompensa positiva por atravesar un punto

4. Una gran recompensa negativa de -1 cuando el agente choca con algún obstáculo. También se termina el episodio de entrenamiento cuando esto ocurre

```
// We hit something that wasn't another agent
if (area.trainingMode)
{
    AddReward(-1f);
    EndEpisode();
    return;
}
```

Figura 54: recompensa negativa por chocar con un obstáculo

7.4. Dirección única y dos equipos

Para este ejemplo, toda la lógica explicada anteriormente será utilizada, sin mayores cambios. La diferencia reside en la presencia de dos recorridos de colores distintos (rojo y

azul), y que ambos equipos deberán entrenar al mismo tiempo sobre el mismo entorno. Los agentes solo deben seguir el recorrido de su color en una única dirección.

Este será el nuevo aspecto del entorno con los dos recorridos:

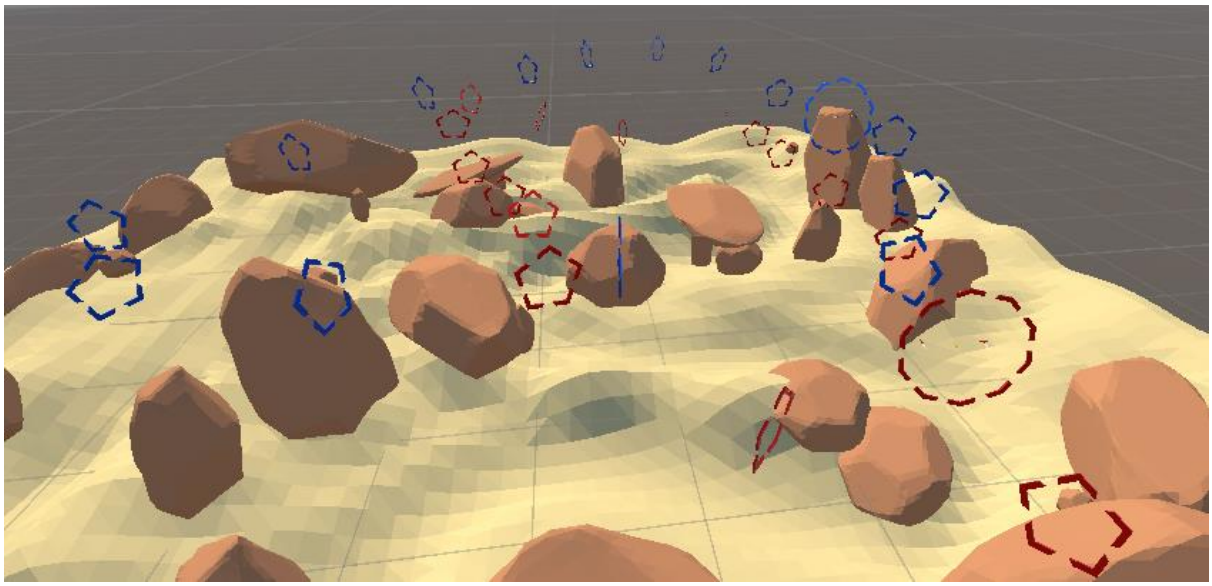


Figura 55: entorno desertArea con los dos recorridos

Para distinguir entre un equipo u otro, hemos creado una clase llamada **TeamCategory** la cual tiene una propiedad enum “Team”, donde el equipo rojo representa el valor “0 - red” y el equipo azul representa el valor “1 - blue”

```
public class TeamCategory : MonoBehaviour
{
    // New Elements for new learning situations in
    [System.Serializable]
    1 referencia
    public enum Team
    {
        none=2,
        red=0,
        blue=1
    };
```

Figura 56: enumarable Team con los distintos equipos

Los agentes tienen este enum como una de sus propiedades configurables:

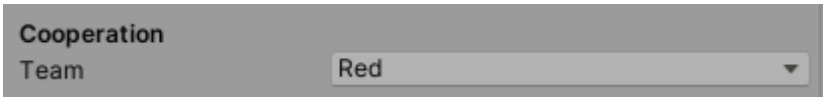


Figura 57: desplegable para seleccionar el equipo del agente dentro de la configuración del script AircraftAgent

Los agentes luego utilizarán este valor para decidir qué lista de puntos deben usar:

```
public void getCheckPointList() {
    listOfTeamCheckpoints = area.checkpointsDictionary[(int)this.team];
}
```

Figura 58: código de AircraftAgent para obtener la lista de puntos de su color/equipo

Para los puntos, no utilizaremos esta propiedad Team, sino que definiremos su tag a “red” o “blue” en el *prefab* de cada punto:

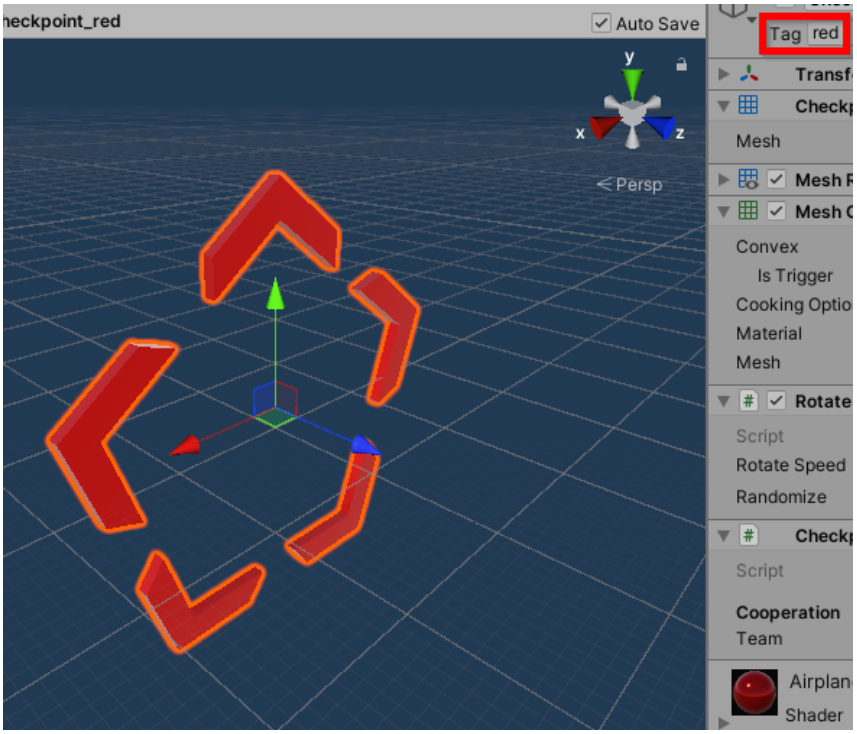


Figura 59: prefab checkpoint_red con el Tag "red"

Y en cuanto a las recompensas, sólo se otorgará recompensa si el punto cruzado tiene el mismo “tag” que el punto objetivo del agente, el cual podemos obtener de la lista de puntos del agente, mediante la propiedad **nextCheckpointIndex:**

```
Checkpoint nextCheckpoint = listOfTeamCheckpoints[this.NextCheckpointIndex];
```

Figura 60: código para la obtención del próximo punto objetivo del agente

```
if (collisionedCheckpoint.CompareTag(nextCheckpoint.tag) &&
    collisionedCheckpoint == nextCheckpoint)
{
    GotCheckpoint();
}
```

Figura 61: código donde se comprueba que el agente ha cruzado el checkpoint correcto

Estos son todos los cambios realizados para este ejemplo.

Aquí se puede ver un vídeo demostrativo: **Equipo rojo y azul en una única dirección**[17]

[17] https://www.youtube.com/watch?v=8RTX8AAHUtY

7.5. Dos direcciones y dos equipos

En este nuevo ejemplo, por cada color habrá dos agentes, uno de ellos atravesará el recorrido en el sentido contrario al usual. Con el objetivo de conseguir de forma más rápida todos los puntos.

Recorrido Rojo

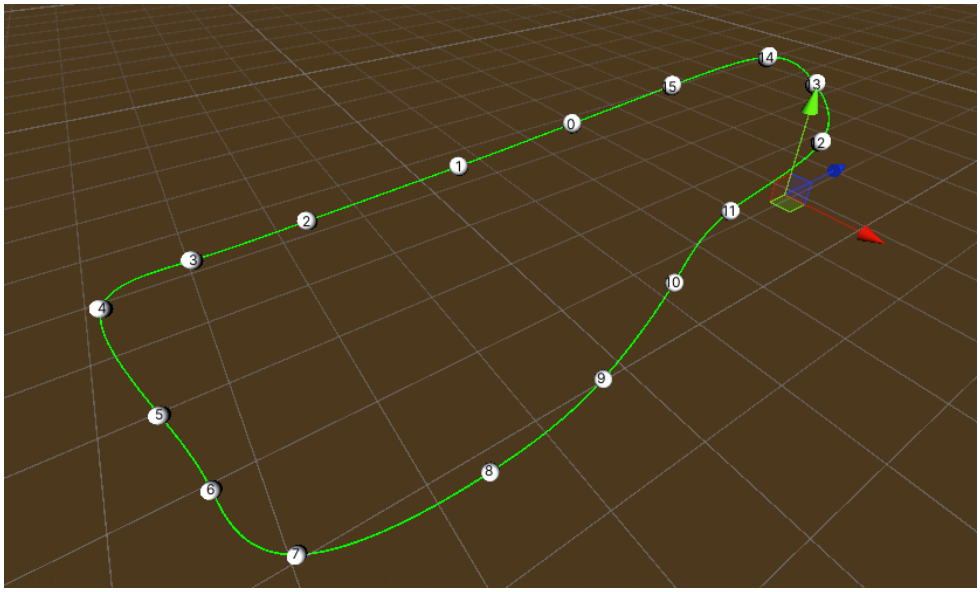


Figura 62: recorrido rojo sin los puntos generados

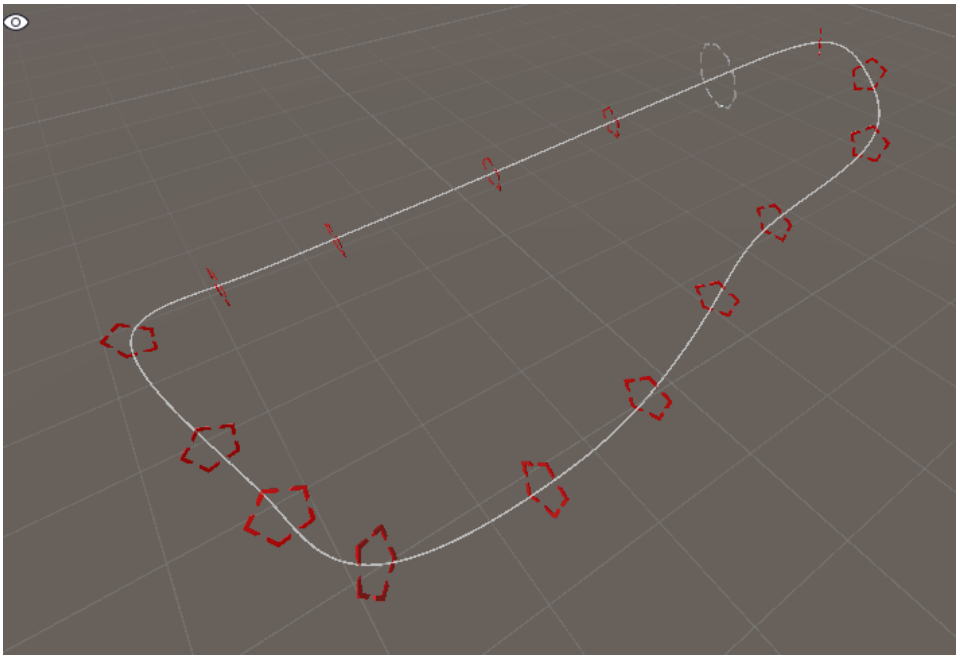


Figura 63: recorrido rojo con los puntos generados

Recorrido Azul

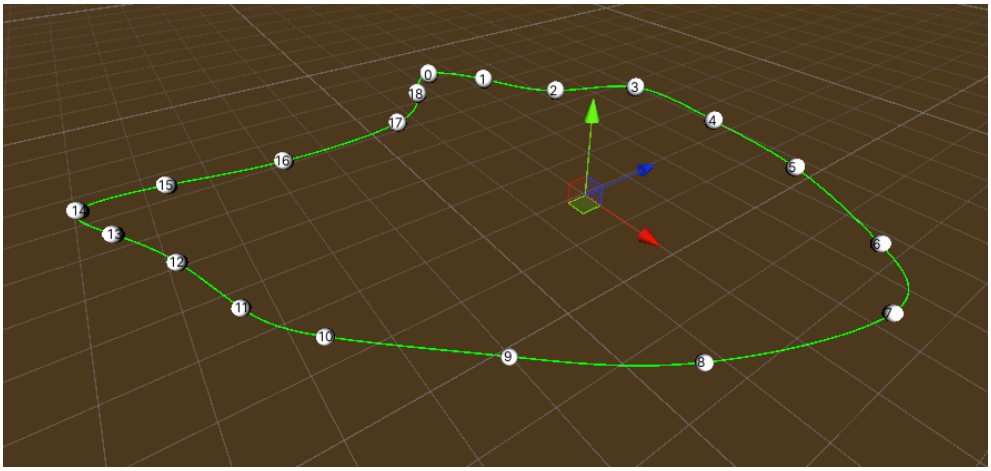


Figura 64: recorrido azul sin los puntos generados

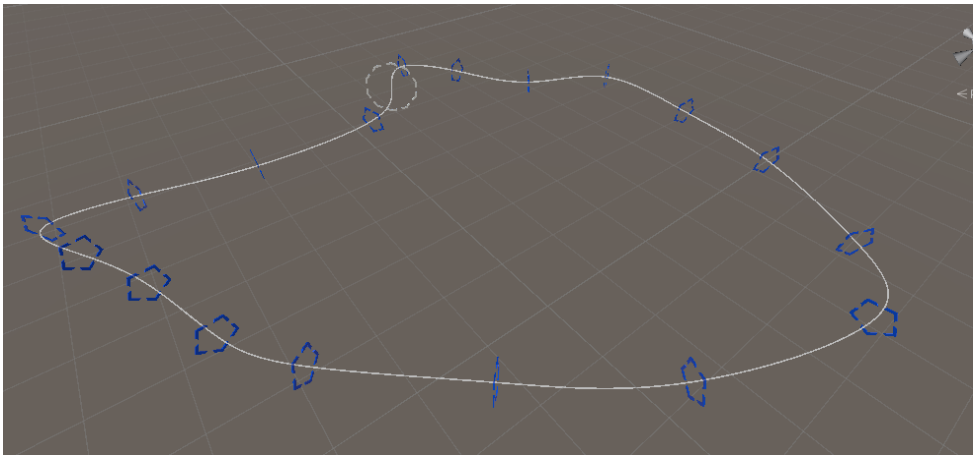


Figura 65: recorrido azul con los puntos generados

Ahora los equipos de los agentes se asignan mediante un objeto de unity que contiene el script **TeamCategory.cs**:

Figura 66: Objeto Unity conteniendo los distintos equipos

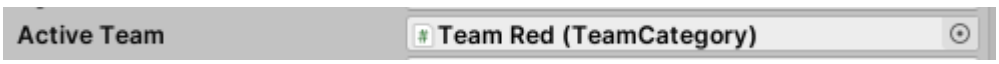


Figura 67: referencia al equipo rojo

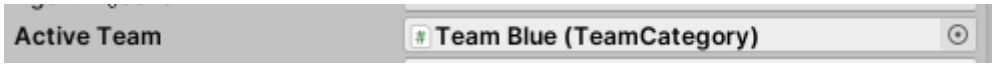


Figura 68: referencia al equipo azul

Para este ejemplo, deberemos modificar la lógica que calcula el próximo punto que debe alcanzar el agente en caso de que esté yendo en sentido contrario. Veamos los cambios realizados a la lógica de la simulación:

Empezamos añadiendo una propiedad más que definirá si el agente va en sentido contrario o el normal:

```
[Header("Cooperation")]
public AgentDirection agentDirection;
```

```
public enum AgentDirection
{
    backward = 0,
    forward = 1

};
```

Figura 69: propiedad con la que definir la dirección del agente.

Se ha añadido al objeto *Checkpoint*, una propiedad para saber si el punto ha sido cruzado por un agente, así como la lógica necesaria para actualizar el estado del objeto en cuanto este sea atravesado por un agente, pero solo si ese punto era el objetivo del agente en cuestión:

```
[Header("Cooperation")]
public Boolean checkPointPassed = false;
```

Figura 70: nueva propiedad para saber si el Checkpoint ha sido cruzado

También se ha añadido a los *Checkpoints* la propiedad *activeTeam* para que no sea necesario el uso de tags:

```
public TeamCategory activeTeam;
```

Figura 71: nueva propiedad que contiene el equipo al que pertenece el Checkpoint

Si el agente cruza un punto nuevo de su mismo color y, además, este punto era su objetivo, entonces la propiedad **checkpointPassed** pasará a “true”; adicionalmente, el color del punto cambiará a gris para tener una ayuda visual.

```
if (!checkPointPassed &&
    agent.activeTeam.team == this.activeTeam.team &&
    (agent.listOfTeamCheckpoints[agent.NextCheckpointIndex] == this)
    )
{
    this.GetComponent<MeshRenderer>().material = fadedMaterial;
    checkPointPassed = true;
}
```

Figura 72: actualización de un Checkpoint cuando este es cruzado por un agente

No queremos que durante el entrenamiento los agentes reciban recompensa por cruzar por un punto que ya fue cruzado. Esto no era un problema en el ejemplo anterior, pues el agente solo recibía recompensa si cruzaba el punto que tenía como objetivo, pero ahora existe un agente que se mueve en sentido contrario y podría ya haber cruzado ese punto objetivo.

También hemos añadido a las observaciones del agente la dirección que se le ha asignado:

```
// Direction to follow of the agent( int = 1 value )
sensor.AddObservation((int)this.agentDirection);
```

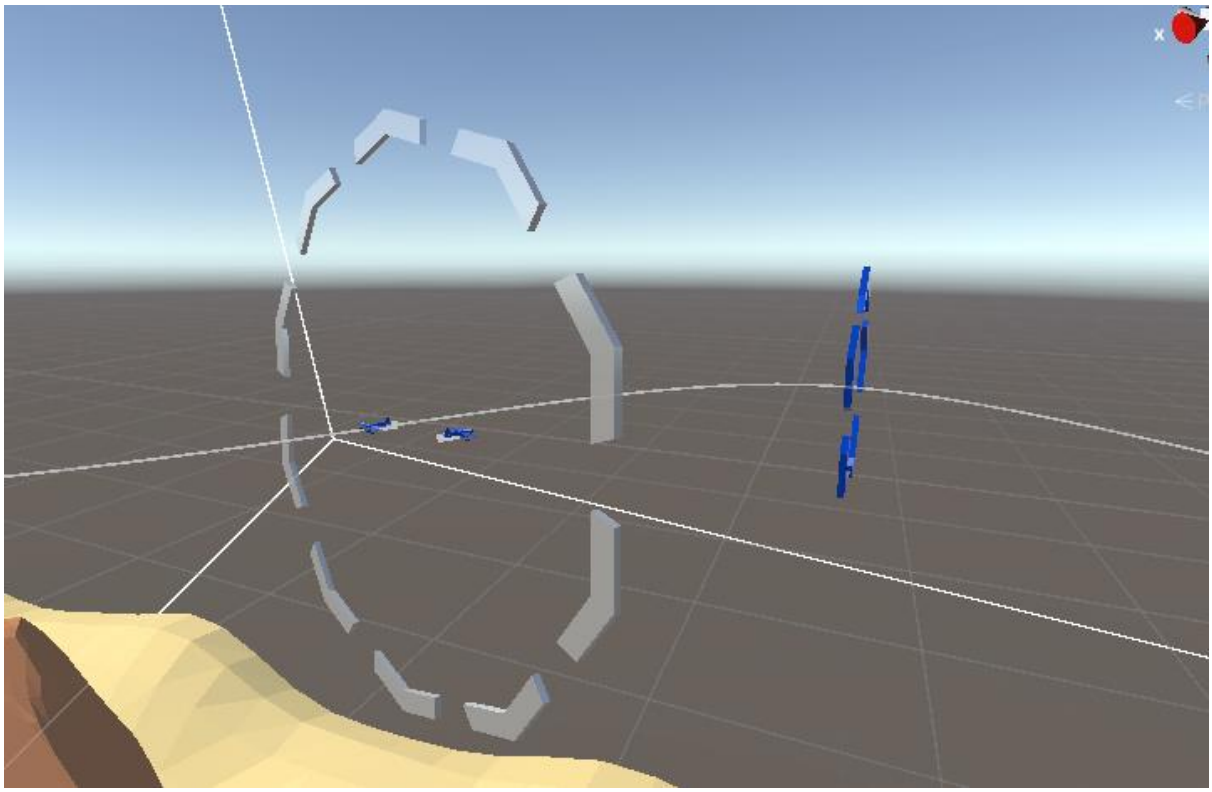
Figura 73: nueva observación dentro del método CollectObservations() del agente

En cuanto a la asignación del siguiente punto objetivo: para el agente que va en la dirección usual (forward), nada cambia; pero el agente que va en dirección contraria (backward), ahora cogerá los puntos de la lista en sentido contrario (restando en vez de sumando al índice):

```
if(this.agentDirection == AgentDirection.backward && this.NextCheckpointIndex == 0)
{
    this.NextCheckpointIndex = area.listNumCheckpoints[(int) this.activeTeam.team] - 2;
}
```

Figura 74: asignación del nuevo punto objetivo para el agente que va en sentido contrario

Algo a tomar en cuenta es que el último punto de la lista es en realidad el punto “inicial” de los agentes, este punto inicial ya está en estado “cruzado” (checkPointPassed = true), por lo que nunca será asignado a un agente.



Por otro lado, también hay métodos que se encargan de contar cuántas vueltas han dado los agentes y de reiniciar el estado de todos los puntos cuando acaben una vuelta. Hay un máximo de vueltas configurable que los agentes de un equipo deben dar, antes de que se les considere ganadores (en un principio se recompensaba un poco más al equipo ganador durante un entrenamiento, pero finalmente fue descartado por inconsistencias durante el entrenamiento).

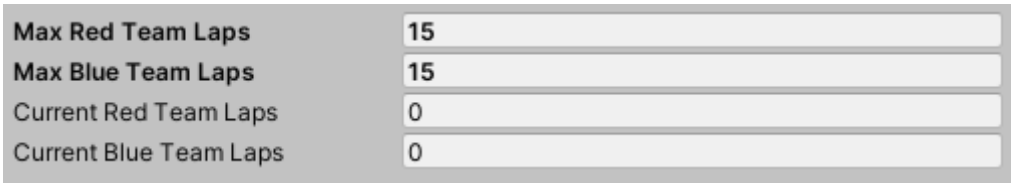


Figura 75: configuración del número de vueltas que debe dar cada equipo

Recompensas

En cuanto a las recompensas durante el entrenamiento, se mantienen las recompensas iniciales. Sin embargo, ahora cuando un agente choque contra un objeto, el episodio de entrenamiento no se detendrá, solo se reiniciará la posición del agente.

```
371         private void OnCollisionEnter(Collision collision)
372         {
373             if (!collision.transform.CompareTag("agent"))
374             {
375                 // We hit something that wasn't another agent
376                 if (area.trainingMode)
377                 {
378                     AddReward(-1f);
379+                    area.ResetAgentPosition(this, false);
380                     return;
```

Figura 76: la posición del agente se reinicia al chocar con un objeto

La única forma de que el episodio de entrenamiento termine, es que los agentes consigan todos los puntos objetivos o el tiempo máximo de entrenamiento se supere.

```
112                 // Make sure we haven't run out of time if training
113                 if (StepCount > nextStepTimeout)
114                 {
115                     AddReward(-.5f);
116                     EndEpisode();
117                 }
```

Figura 77: finalización del episodio al acabarse el tiempo de entrenamiento

```
239+                    if (!CheckpointsLeft()) {
240+                        // Check if the agent is team blue/red
241+                        SetWinningTeam();
242+                    }
```

Figura 78: finalización del episodio al alcanzar el número máximo de vueltas

Cuando todos los puntos de un recorrido son cruzados por un equipo, el contador de vueltas de ese equipo aumenta y el recorrido se reinicia, es decir, los agentes vuelven a su posición inicial y los puntos vuelven todos al estado “no cruzado”. Si un equipo llega al máximo de

vueltas configurado, el episodio de entrenamiento acaba. El objeto del entorno (AircraftArea) comprueba si un equipo ha conseguido todos los puntos y reinicia sus recorridos para que siga el entrenamiento. Esto lo hace mediante comprobaciones constantes en su método Update().

```
private void Update()
{

    if(this.redTeamFinished)
    {
        currentRedTeamLaps++;
        ResetTeamAgentsPositionsAndTeamCheckpoints(TeamCategory.Team.red);
    }

    if (this.blueTeamFinished)
    {
        currentBlueTeamLaps++;
        ResetTeamAgentsPositionsAndTeamCheckpoints(TeamCategory.Team.blue);
    }

  if (currentBlueTeamLaps == MaxBlueTeamLaps || currentRedTeamLaps == MaxRedTeamLaps)
  {
      Debug.Log("Race over, resetting all agents");
      ResetAllAgentsPositionsAndCheckpoints();
      this.currentBlueTeamLaps = 0;
      this.currentRedTeamLaps = 0;
  }
  this.redTeamFinished = false;
  this.blueTeamFinished = false;
}
```

Figura 79: comprobación del estado de los recorridos de cada equipo y reinicialización del recorrido si es necesario

Aquí se puede ver un vídeo demostrativo: Equipo azul y rojo - dos direcciones[18]

7.6. Maestro/Esclavo, un solo equipo

Hasta ahora los agentes se han estado aprovechando de la ventaja de que los puntos del recorrido están dentro de una lista ordenada, simplificando así la tarea de decidir cuál será el próximo punto al que se deben dirigir; sin embargo, esta vez el recorrido tendrá esta forma, donde el punto más cercano no es necesariamente el siguiente de la lista:

[18] https://www.youtube.com/watch?v=8szUEu1jMDY

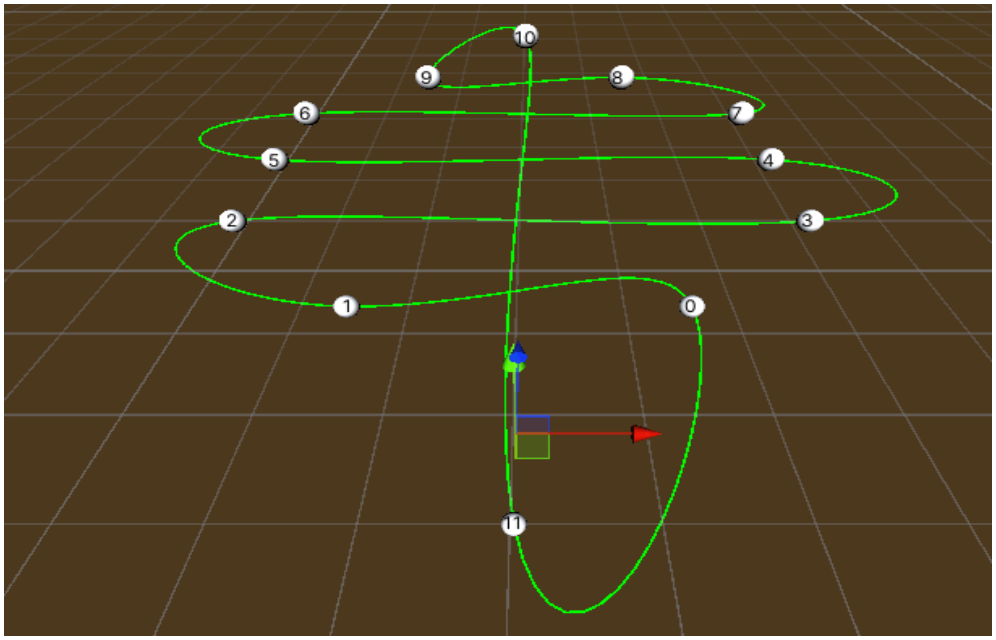


Figura 80: recorrido sin los puntos generados

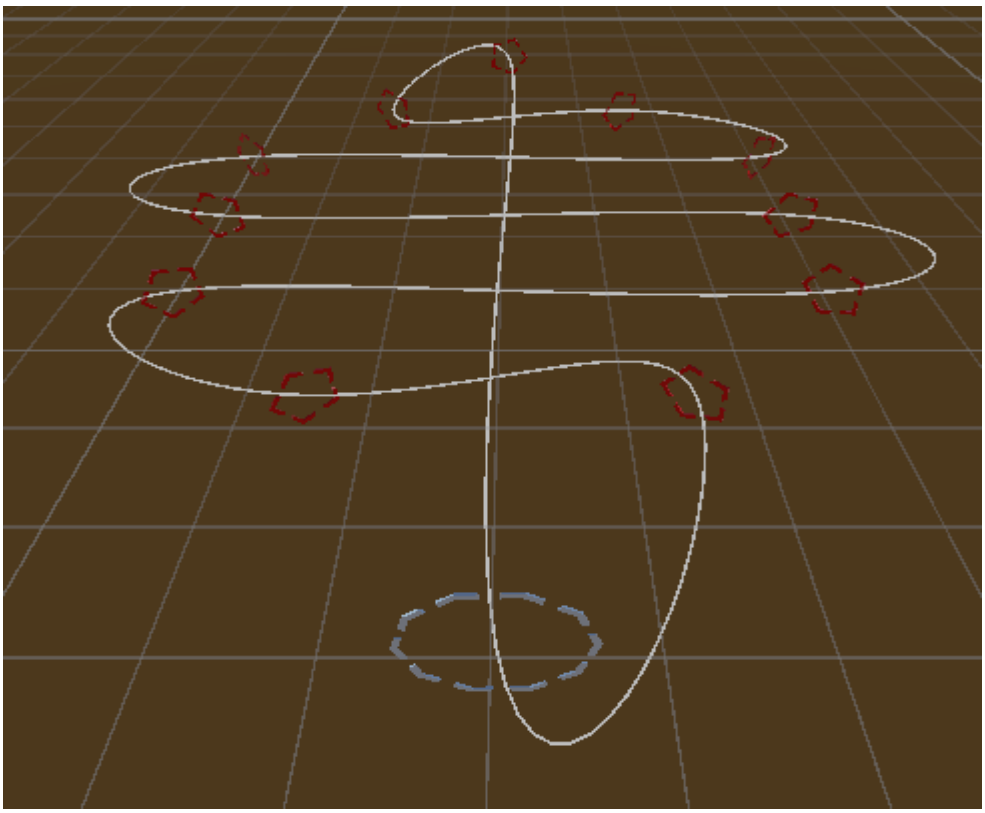


Figura 81: recorrido con los puntos generados

Además del cambio en el recorrido, aprovecharemos las categorías creadas en el ejemplo anterior (forward/backward) para otorgar dos categorías distintas a los agentes: maestro y esclavo.

Figura 82: nueva categoría isCaptain para los agentes

La lógica detrás de estas dos categorías será la siguiente:

- **Maestro**: El agente con esta categoría nunca cambiará el próximo punto del recorrido al que ya haya decidido dirigirse.
- **Esclavo**: El agente con esta categoría puede cambiar, en cualquier momento, el próximo punto del recorrido al que ha decidido dirigirse, siempre tomando en cuenta a donde se está dirigiendo el "Maestro"; en otras palabras, el esclavo intenta siempre dirigirse a puntos diferentes a los del "Maestro".

Como ya hemos dicho, para la toma de decisiones, los agentes ya no se pueden valer de una lista ordenada de puntos, ahora los agentes tendrán tres puntos a tomar en cuenta:

- Último punto conseguido (**PreviousCheckpointCrossed**)
- Primer punto más cercano al último punto conseguido (**ClosestCheckpointIndex**)
- Segundo punto más cercano al último punto conseguido (**SecondClosestCheckpointIndex**).

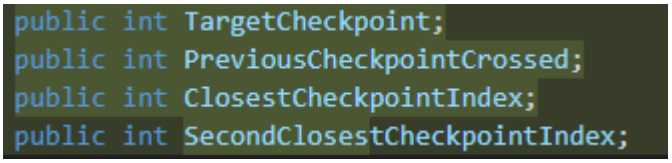


Figura 83: nuevas propiedades de los agentes con los distintos puntos a tomar en cuenta

Por defecto, los agentes elegirán como su próximo punto objetivo el primer punto más cercano (**targetCheckpoint**), pero como ya hemos visto, el agente esclavo puede cambiar su objetivo si lo considera necesario. Esta decisión de cambio de objetivo, la realizará mediante una nueva acción en su vector de acciones.

Para que el agente esclavo sea capaz de cambiar el punto al que va a dirigirse, será necesario añadir una nueva rama al árbol de decisiones (vector de acción) de su **comportamiento**.

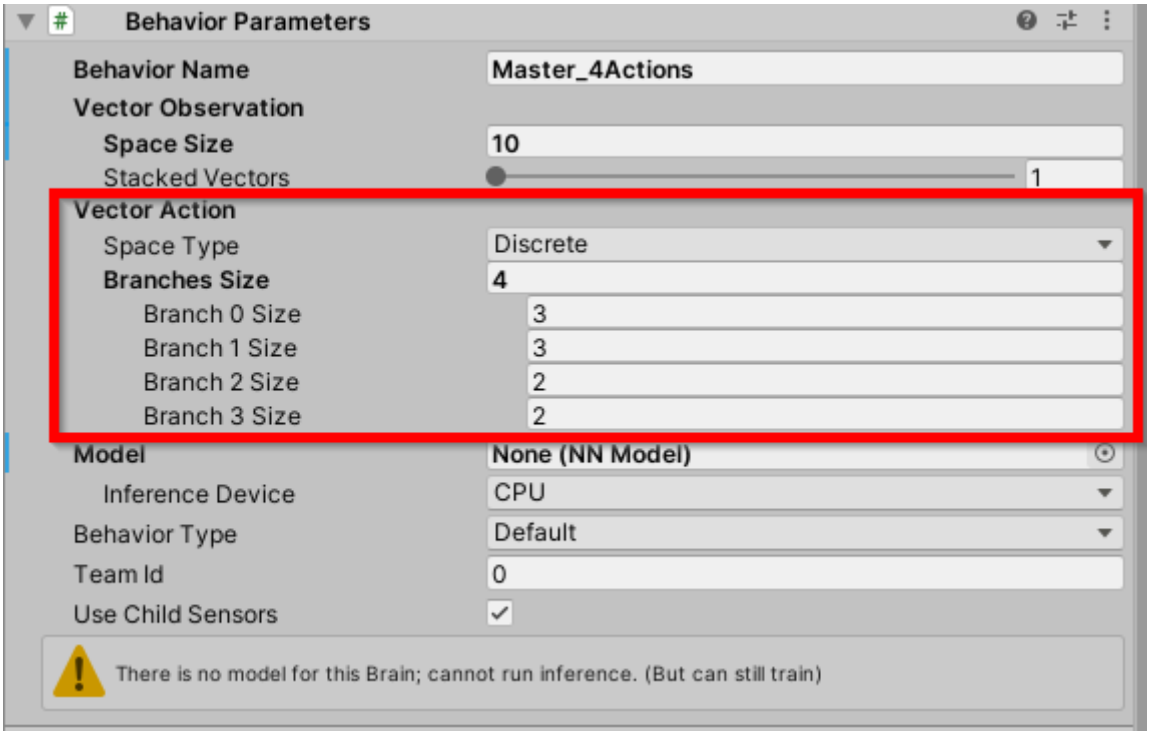


Figura 84: nueva configuración del Behavior Parameters de los agentes

Esta nueva decisión (**Branch 3** en la Figura 83) hará que el agente cambie su punto objetivo entre el primer o el segundo punto más cercano, se podría decir que esta decisión no es más que un interruptor que le hace elegir entre un punto u otro. Siendo un "0" igual a no hacer nada y "1" igual a cambiar el punto objetivo.

```
114         ////AI change ----------------------------------------
115         changeCheckpoint = Mathf.FloorToInt(vectorAction[3]) == 1;
116         ////AI change ----------------------------------------
```

Figura 85: cuando el agente recibe un ""1 de la rama 3 del vector de acciones, cambiará su punto objetivo

Para evitar un cambio constante e innecesario del punto objetivo, se hará uso del **enmascaramiento de acciones**, un método que nos ofrece **ml-agents** para que el agente ignore ciertas opciones de una acción si se cumplen ciertas condiciones. Por ejemplo: si solo queda un punto por conseguir, no tiene sentido que el agente intente cambiar el punto objetivo al que dirigirse.

```
253     public override void CollectDiscreteActionMasks(DiscreteActionMasker actionMasker)
254     {
255         // Mask action in branch 3 (changeCheckpoint) if you are the captain
256         // Or if you are the slave and your initial checkpoint is 0
257         if (this.isCaptain) {
258             actionMasker.SetMask(3, new int[1] { 1 });
259         }
260         if (!this.isCaptain)
261         {
262             if (this.TargetCheckpoint != 0)
263             {
264                 actionMasker.SetMask(3, new int[1] { 1 });
265             }
266         }
267     }
```

Figura 86: método para enmascarar la acción de cambio de punto

También es importante resaltar que las recompensas por pasar por el punto correcto son las mismas para el esclavo y para el maestro, estás recompensas positivas son iguales a las del ejemplo anterior. En un principio, se trató de darle una mayor recompensa al agente esclavo, ya que este debe tomar una decisión adicional (cambiar de punto si hace falta), pero finalmente, esto fue descartado por inconsistencias en el entrenamiento.

La única diferencia es que ahora el esclavo tiene una potencial recompensa negativa adicional. El agente esclavo es recompensando negativamente si al principio de la carrera tiene el mismo punto objetivo que el capitán

```
290             if (!this.isCaptain)
291             {
292                 if (checkpointCrossed.checkpointIndex == this.TargetCheckpoint
293                     && this.area.nextCheckpointLogic.NumberOfCheckpointsLeft(this) == 1)
294                 {
295                     AddReward(GOT_CHECKPOINT_REWARD);
296                 }
297                 else
298                     if (checkpointCrossed.checkpointIndex == this.TargetCheckpoint
299                         && checkpointCrossed.checkpointIndex != this.teamMate.TargetCheckpoint
300                         && this.area.nextCheckpointLogic.NumberOfCheckpointsLeft(this) > 1)
301                     {
302                         AddReward(GOT_CHECKPOINT_REWARD);
303                     }
304                     else
305                     {
306                         AddReward(-GOT_CHECKPOINT_REWARD * 2f);
307                     }
308             }
309         }
```

Figura 87: recompensas para el agente esclavo

Con esto explicado, podemos ver en acción a los agentes una vez entrenados: Un solo equipo - Maestro/Esclavo[19]

[19] https://www.youtube.com/watch?v=fHMeH15YF_A

8. Transmisión de información: Servicios REST con Spring Boot y Java

Ahora que ya hemos podido explorar las posibilidades que nos ofrece ml-agents, procederemos a desarrollar un método para transferir esta información en tiempo real. Para ello utilizaremos una de las soluciones más comunes que se utiliza hoy en día: los servicios REST.

8.1. Servicio REST

Un servicio REST utiliza el protocolo HTTP como puente para comunicar distintos programas o sistemas.

Un cliente envía a un servidor peticiones HTTP con rutas y parámetros específicos con los que el servidor puede generar una respuesta adecuada para el cliente.

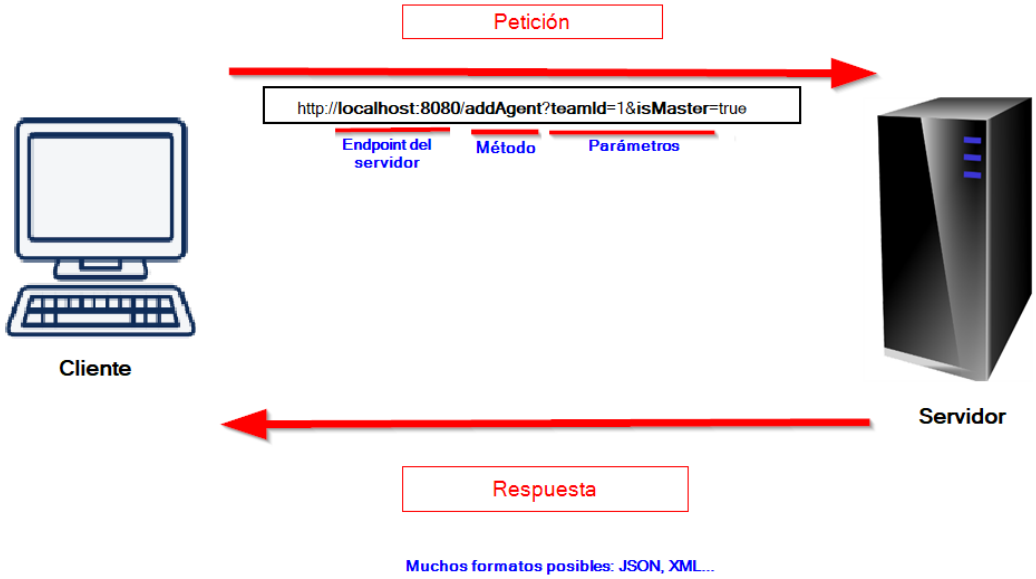


Figura 88: funcionamiento de un servicio REST

Su gran ventaja reside en su universalidad, mientras el programa o sistema pueda recibir o enviar peticiones HTTP, entonces será capaz de proveer o de consumir el servicio REST.

8.2. Creación del servidor REST mediante **Java + Spring**

Para crear el servidor que proveerá nuestro servicio REST, utilizaremos **Spring**, un framework de **Java** muy popular en la actualidad.

Empezamos yendo a Spring Initializr[20], que nos ayudará a crear nuestro proyecto Java inicial con Spring.

Dejaremos las configuraciones por defecto:

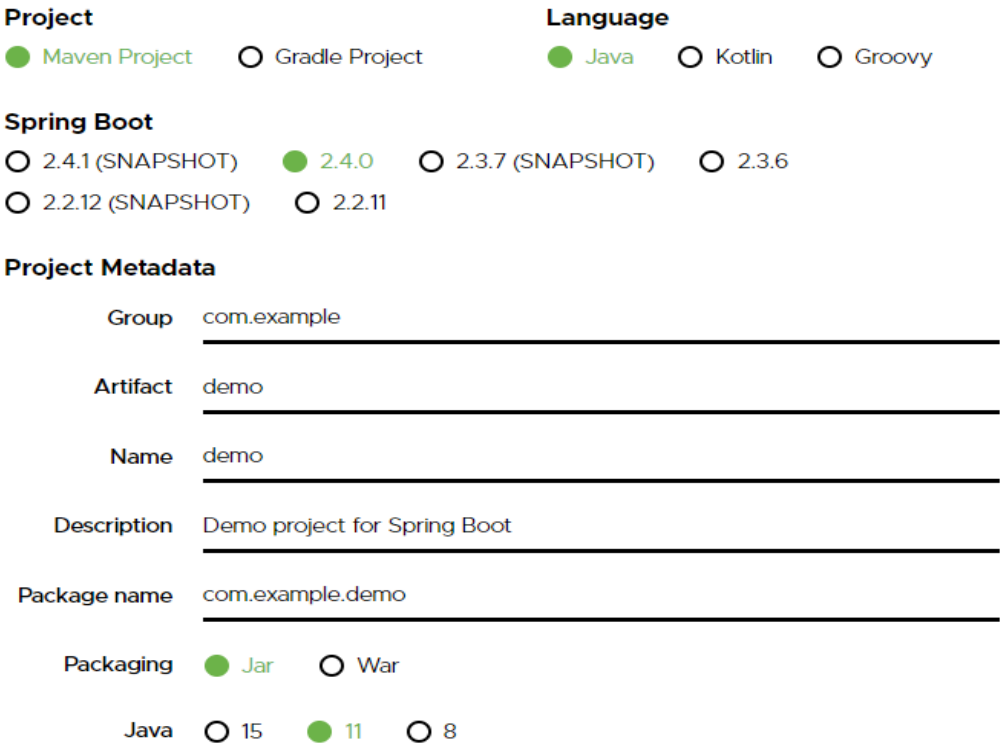


Figura 89: configuración por defecto del Spring Initializr

Y añadiremos las dependencias necesarias para el proyecto:

[20] https://start.spring.io/

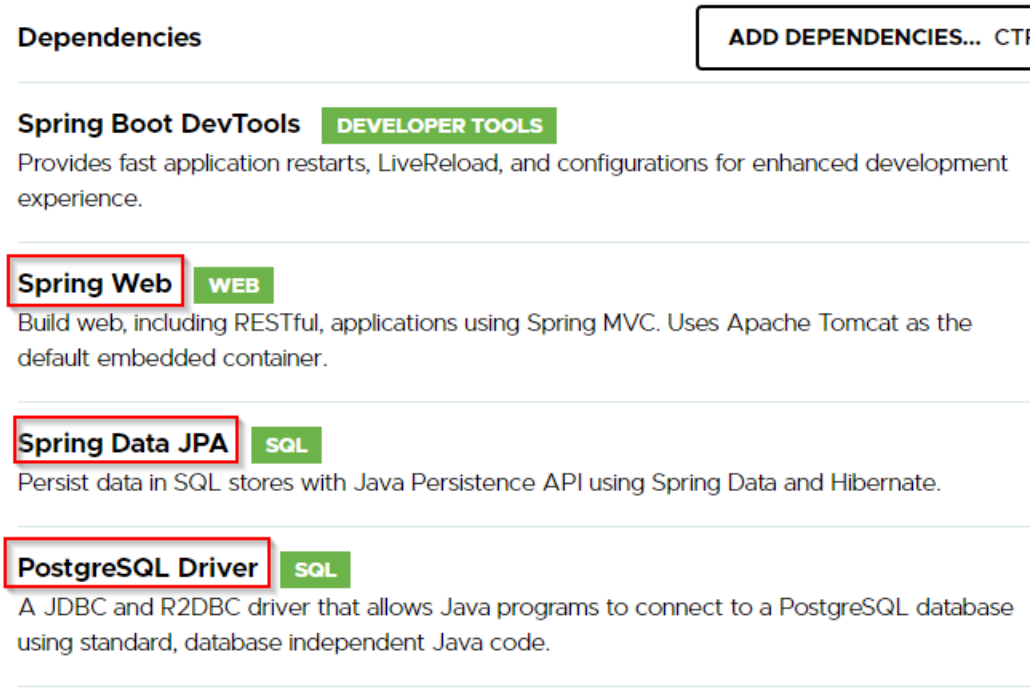


Figura 90: dependencias necesarias para nuestro servicio REST

- **Spring Web:** Nos aportará con los métodos y anotaciones necesarias para recibir peticiones HTTP.

- **Spring Data JPA:** Nos permitirá persistir datos a nuestra base de datos.

- **PostgreSQL Driver:** Es el driver necesario para persistir datos a una base de datos PostgreSQL.

- **Spring Boot DevTools**: Dependencia opcional. Esta dependencia recargará automáticamente nuestro código al hacer cambios en el mismo.

Una vez descargado y abierto el proyecto, deberíamos tener una estructura tal que así:

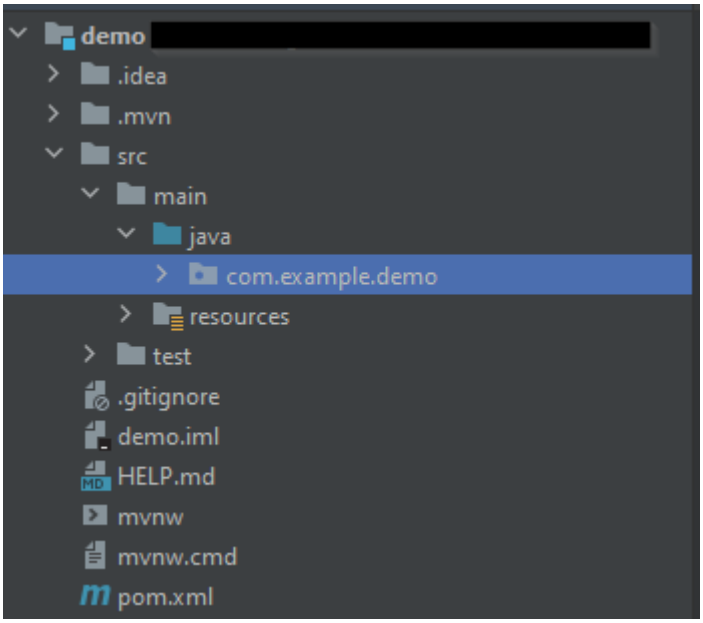


Figura 91: estructura inicial del proyecto de Spring

Podemos observar que en el archivo pom.xml se encuentran ya predefinidas todas las dependencias que habíamos seleccionado previamente:

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Figura 92: archivo pom.xml

Mucho se podría explicar sobre maven y el archivo pom.xml, pero nos bastará con saber que se encarga de gestionar por nosotros las dependencias que utiliza nuestro proyecto, reduciendo la complejidad de gestión de este.

Para almacenar nuestros datos, utilizaremos PostgreSQL, creando tres tablas: *teams*, *agents* y *checkpoints.* Para más información sobre la creación de esta base de datos revise la sección “creación de la base de datos” en el anexo.

Una vez comprobado que tenemos nuestras dependencias y ya habiendo creado la base de datos, procederemos a la creación de los modelos que utilizaremos para la persistencia de datos.

Para crear un modelo el cual se pueda persistir, será necesario crear una clase cuyas propiedades sean las columnas de la tabla en la base de datos a la que el modelo hace referencia. También será necesario el uso de varias anotaciones. Y muy importante. un

Objeto-Modelo, necesita tener métodos Get y Set públicos para todas sus propiedades, así como un constructor vacío.

Veamos un ejemplo para entenderlo mejor, crearemos el modelo de los equipos **Team**.

- **Team**:

```java
@Entity
@Table(name="teams")
public class Team {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    @OneToMany(
            cascade = CascadeType.ALL
    )
    @JoinColumn(name = "team_id")
    private List<Agent> agents;

    @OneToMany(
            cascade = CascadeType.ALL
    )
    @JoinColumn(name = "team_id")
    private List<Checkpoint> checkpoints;
```

Figura 93: Objeto Team

Podemos ver que entre las propiedades de un objeto Team tenemos: **id**, **name**, **agents** y **checkpoints**.

Las dos primeras hacen referencia a las 2 columnas que la tabla “teams” tiene en la base de datos: id, name.

```java
private int id;
private String name;
```

Figura 94: propiedades que hacen referencia a las columnas de la tabla

Y las dos siguientes, son equivalentes a las 2 llaves foráneas que referencian a la tabla “teams”: **team_id** en la tabla “agents” y en la tabla “checkpoints”.

```java
@JoinColumn(name = "team_id")
private List<Agent> agents;
```

Figura 95: propiedad que contiene la lista de agentes con un team_id específico

```java
@JoinColumn(name = "team_id")
private List<Checkpoint> checkpoints;
```

Figura 96: propiedad que contiene la lista de Checkpoints con un team_id específico

Expliquemos ahora las distintas anotaciones utilizadas en la creación del modelo/objeto (Figura 92):

@**Entity**: le especifica a Java que esta clase debe ser tratada como una entidad que se persistirá a una base de datos.

@**Table(name= "xxx")**: En esta anotación se escribe el nombre de la tabla en la base de datos a la que hace referencia este modelo en particular.

@**Id**: sirve para declarar que la próxima propiedad de la clase es la **id** del modelo (la llave primaria).

@**GeneratedValue**(strategy = GenerationType.IDENTITY): hace que la id sea un valor que se autogenere y que no tendremos que especificar a la hora de persistir nuevos elementos a la base de datos.

@**OneToMany**(cascade = CascadeType.ALL): nos sirve para especificar la relación Uno a Muchos entre este Objeto-Modelo y otro Objeto-Modelo. En nuestro caso, un equipo puede tener muchos agentes y muchos puntos. Con esta anotación al leer los datos de un equipo, automáticamente conseguiremos los objetos Agent y Checkpoint que pertenezcan a ese equipo, ahorrandonos así bastante trabajo. "CascadeType.ALL" sirve para que cualquier acción que realizamos en la referencia padre "**Team**" se propague a las referencias hijas "**Checkpoint**" y "**Agent**".

@**JoinColumn**(name= "xxx"): nos indica el nombre de la columna que contiene la llave foránea que hace referencia a este modelo, "**team_id**" en nuestro caso.

Ahora crearemos los modelos **Checkpoint** y **Agent**:

- **Checkpoint**:

```java
@Entity
@Table(name="checkpoints")
public class Checkpoint {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(name="checkpoint_index")
    private int checkpointIndex;
    @Column(name="team_id")
    private Integer teamId;
    @Column(name="agent_id")
    @Nullable
    private Integer agentId;
    @Column(name="is_passed")
    private boolean isPassed;
    @Column(name="passed_at")
    @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss")
    private LocalDateTime passedAt;
```

Figura 97: objeto Checkpoint

- **Agents**:

```java
@Entity
@Table(name="agents")
public class Agent {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(name="team_id")
    private Integer teamId;
    @Column(name="is_master")
    private boolean isMaster;

    @OneToMany(
            cascade = CascadeType.ALL,
            orphanRemoval = true
    )
    @JoinColumn(name = "agent_id")
    private List<Checkpoint> checkpoints;
```

Figura 98: objeto Agent

Podemos ver como la creación de los modelos no es más que una forma de intentar describir con un objeto Java una tabla en una base de datos y su relación con otras tablas.

Repasemos una vez más la organización de nuestro proyecto ahora que hemos creado nuestros modelos:

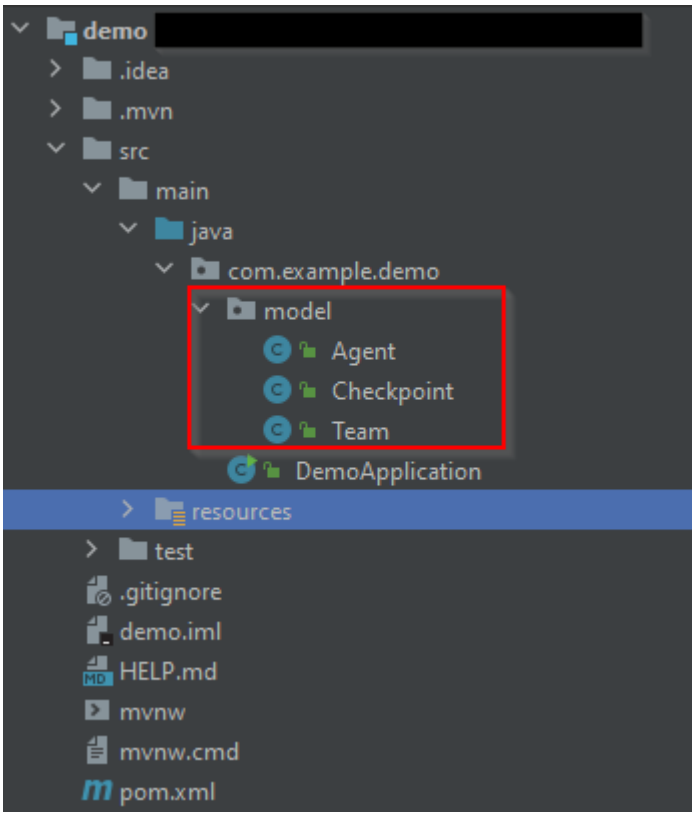


Figura 99: estructura del proyecto tras crear los modelos

Ahora que ya tenemos los modelos, deberemos crear un servicio para cada modelo que nos permita persistir los datos a nuestra base de datos.

Como siempre, utilizaremos el Modelo **Team** como ejemplo de cómo se debe hacer este proceso:

Primero creamos un repositorio del modelo **ITeamRepository**:

```
@Repository
public interface ITeamRepository extends JpaRepository<Team, Integer> {
    Team findByName(String name);
}
```

Figura 100: Repostorio del modelo Team

La anotación @**Repository** le indica a Spring que esta interfaz es un repositorio.

Vemos como debemos hacer que el repositorio herede de la clase **JpaRepository** pasándole el tipo de nuestro modelo y el tipo de la id de nuestro modelo: <**Team, Integer**>, en este caso.

JpaRepository viene con una serie de métodos ya implementados que nos serán muy útiles para persistir datos sin tener que escribir SQL: findAll, findById, save...

Por otro lado, **JpaRepository** goza de una serie de **palabras clave**[21] que nos servirán para la creación de queries SQL customizadas sin la necesidad de escribir código SQL, por ejemplo, el método `Team findByName(String name);` nos servirá para buscar un equipo de la base de datos con el "name" que le pasemos al método. Algo que en código SQL se podría escribir así:

```
SELECT * FROM teams WHERE name='red';
```

Figura 101: código SQL del método findByName

Además, recordemos que nuestro modelo **Team** tiene dos relaciones Uno a Muchos que hemos definido previamente. Esto significa que al obtener un objeto Team obtendremos todos los objetos **Agent** y **Checkpoint** que referencian a ese objeto en la base de datos. Por lo que, en realidad, un posible código equivalente más preciso sería:

```
SELECT * FROM teams t
INNER JOIN agents a
    ON a.team_id = t.id
INNER JOIN checkpoints c
    ON c.team_id = t.id
WHERE name='red';
```

Figura 102: código SQL del método findByName con las relaciones Uno a Muchos

Con el repositorio terminado, continuaremos creando el servicio que utilizará este repositorio, **TeamService:**

```
@Service
public class TeamService{
    @Autowired
    ITeamRepository teamRepo;

    public List<Team> getAll() { return teamRepo.findAll(); }

    public Team getByName(String name) { return teamRepo.findByName(name); }
}
```

Figura 103: servicio para el repositorio del modelo Team

@**Service** le indica a Spring que esta clase es un servicio.
@**Autowired** nos permitirá usar un objeto **ITeamRepository** sin necesidad de crearlo nosotros mismo en ningún momento.
El método **getAll**(), nos devolverá equipos de la tabla "teams" en la base de datos,

[21] https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repository-query-keywords

El método **getByName**(), nos devolverá el equipo con el nombre especificado.

Procederemos ahora a la creación de los repositorios y servicios para **Agent y Checkpoint**:

- **IAgentRepository**

```java
@Repository
public interface IAgentRepository extends JpaRepository<Agent, Integer> {
    public Agent findByTeamIdAndIsMaster(int teamId, boolean isMaster);
    public List<Agent> findByTeamId(int teamId);
}
```

Figura 104: repositorio del modelo Agent

- **ICheckpointRepository**

```java
@Repository
public interface ICheckpointRepository extends JpaRepository<Checkpoint, Integer> {

    public List<Checkpoint> findByTeamId(int teamId);
    public List<Checkpoint> findByTeamIdAndIsPassedIsTrue(int teamId);
    public List<Checkpoint> findByTeamIdAndAgentIdAndIsPassedIsTrue(int teamId, int agentId);
    public Checkpoint findByCheckpointIndexAndTeamId(int checkpointIndex, int teamId);

}
```

Figura 105: repositorio del modelo Checkpoint

- **AgentService**

```java
@Service
public class AgentService{
    @Autowired
    ICheckpointRepository checkpointRepository;
    @Autowired
    IAgentRepository agentRepository;

    public void createAgent(int teamId, boolean isMaster) {
        Agent agent = new Agent(teamId, isMaster);
        agentRepository.save(agent);
    }

    public Agent getAgent(int teamId, boolean isMaster) {
        return agentRepository.findByTeamIdAndIsMaster(teamId, isMaster);
    }

    public void deleteAllAgents() { agentRepository.deleteAll(); }

    public int countAllAgents() { return agentRepository.findAll().size(); }

    public int countAllAgents(int teamId) { return agentRepository.findByTeamId(teamId).size(); }

    public int countAllPassedCheckpointsByAgent(int teamId, boolean isMaster) {
        Agent agent =  getAgent(teamId, isMaster);
        return checkpointRepository.findByTeamIdAndAgentIdAndIsPassedIsTrue(teamId, agent.getId()).size();
    }
}
```

Figura 106: servicio del modelo Agent

- **CheckpointService**

```java
@Service
public class CheckpointService {
    @Autowired
    ICheckpointRepository checkpointRepository;
    @Autowired
    ITeamRepository teamRepository;
    @Autowired
    IAgentRepository agentRepository;

    public void createCheckpoint(int checkpointIndex, int teamId) {...}

    public void createCheckpoint(int checkpointIndex, int teamId, boolean isPassed) {...}

    public Checkpoint getCheckpoint(int checkpointIndex, int teamId) {...}

    public void deleteAllCheckpoints(){checkpointRepository.deleteAll();}

    public int countAllCheckpoints() { return checkpointRepository.findAll().size(); }

    public int countAllCheckpoints(int teamId) { return checkpointRepository.findByTeamId(teamId).size(); }

    public int countAllPassedCheckpoints(int teamId){...}

    public List<Checkpoint> getAllCheckpoints() { return checkpointRepository.findAll(); }

    public Checkpoint updateCheckpoint(int checkpointIndex, int teamId, boolean isMaster) {...}
}
```

Figura 107: servicio del modelo checkpoint

Después de crear todos los repositorios y servicios, nuestro proyecto tendrá la siguiente estructura:

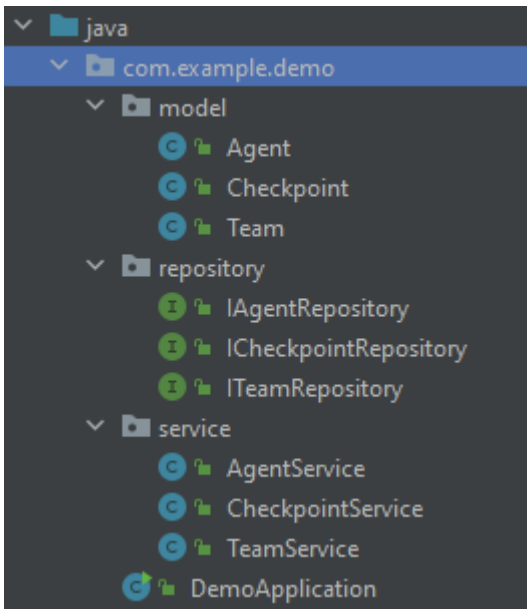


Figura 108: estructura del proyecto tras la creación de los repositorios y servicios

El último paso es crear el controlador que se encargará de llamar a los distintos servicios dependiendo de los parámetros que nos lleguen en la petición HTTP:

```
@Controller
public class RestController {
    @Autowired
    TeamService teamService;
    @Autowired
    CheckpointService checkpointService;
    @Autowired
    AgentService agentService;
```

Figura 109: servicios inyectados en el controlador

Empezamos el controlador llamando a la anotación @**Controller**, que le indica a Spring que esta clase es un controlador.

Luego queremos tener disponibles los distintos servicios de cada Modelo, por lo que utilizamos la anotación @**Autowired**, esta anotación se encargará de la inyección de dependencias.

En cuanto a los métodos del controlador, los podemos separar en tres grupos principales, dependiendo de cómo sean las peticiones HTTP. Veremos 3 ejemplos:

- **Sin parámetros ni ruta variable:** La petición HTTP no tiene parámetros ni rutas variables.

  Ejemplo: **/teams**

```
@GetMapping(value="/teams")
@ResponseBody
public ResponseEntity<List<Team>> readTeams() {
    List<Team> teams = teamService.getAll();
    return ResponseEntity.ok(teams);
}
```

Figura 110: método para recuperar los equipos de la base de datos si la ruta de la petición es /teams

  @**GetMapping**(value= “/xxx”) le indica al controlador que debe ejecutar este método cuando la ruta relativa de la petición HTTP(**GET**) sea **/teams.** También existen anotaciones para los otros tipos de peticiones HTTP (Post, Delete, Put y Patch).
  @**ResponseBody** hará que el *return* del método **readTeams()** se escriba en el cuerpo de la respuesta HTTP.

- **Con ruta variable y sin parámetros:** La petición HTTP contiene una ruta variable, pero sigue sin haber parámetros.

  Ejemplo: **/teams/{red}**

```java
@GetMapping(value="/teams/{teamName}")
@ResponseBody
public ResponseEntity<Team> readTeamByName(@PathVariable String teamName) {
    Team team = teamService.getByName(teamName);
    return ResponseEntity.ok(team);
}
```

Figura 111: método del servicio REST para una petición con ruta variable

Es muy similar al ejemplo anterior, excepto porque ahora el GetMapping tiene un elemento variable {teamName}, el método puede recibir este elemento variable de la ruta como un string o int mediante la anotación @**PathVariable**

- **Con parámetros y sin ruta variable:** La petición HTTP tiene parámetros, pero no hay ruta variable.

  Ejemplo: **/getCheckpoint?checkpointIndex=2&teamId=2**

```java
@GetMapping(value="/getCheckpoint")
public ResponseEntity<Checkpoint> readCheckpoint(
        @RequestParam("checkpointIndex") int checkpointIndex,
        @RequestParam("teamId") int teamId )
{
    return ResponseEntity.ok(checkpointService.getCheckpoint(checkpointIndex,teamId));
}
```

Figura 112: método para una petición con parámetros y sin ruta variable

En este caso, la ruta no es variable, sino que recibimos parámetros en la petición HTTP (**checkpointIndex** y **teamId**), para acceder a estos parámetros de la petición HTTP es necesario utilizar la anotación @**RequestParam** y que los parámetros de entrada del método tengan el mismo nombre que los de la petición.

Echémosle un vistazo por última vez a la estructura final de nuestro proyecto:

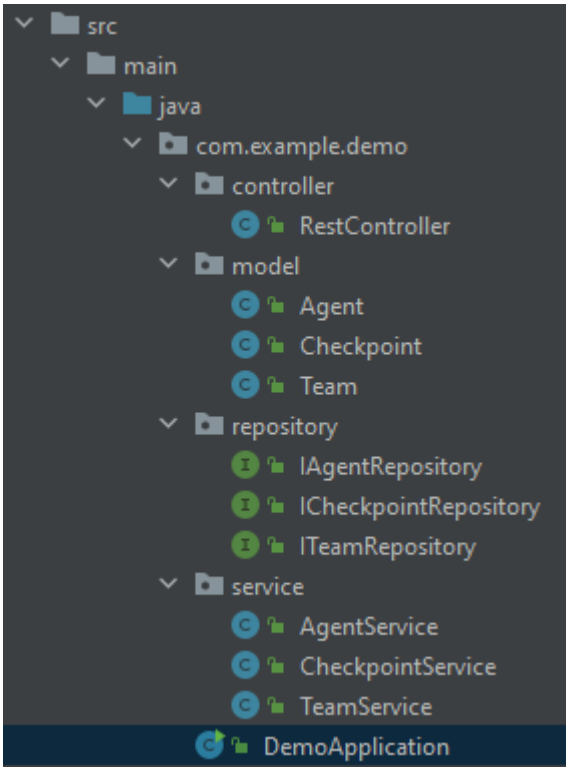


Figura 113: estructura final del proyecto Spring

Para iniciar el servidor es tan sencillo como ejecutar el método *main* de la clase **DemoApplication.**

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) { SpringApplication.run(DemoApplication.class, args); }

}
```

Figura 114: método main de nuestro proyecto Spring

Si el servidor se ha iniciado correctamente, veremos el siguiente mensaje en consola:

```
: Started DemoApplication in 3.522 seconds (JVM running for 4.065)
```

Figura 115: salido de consola si el servidor REST se inicia correctamente

A continuación, y para entender mejor la relación entre las peticiones HTTP y los distintos métodos del controlador, utilizaremos POSTMAN para probar el servidor REST.

8.3. Testeando el servidor REST con POSTMAN

Vamos a crear peticiones HTTP utilizando POSTMAN. Por defecto el servidor de Spring Boot se inicia en el puerto 8080.

Recordemos que en nuestra base de datos tenemos dos equipos creados:

	id [PK] bigint	name character varying (30)
1	1	red
2	2	blue

Figura 116: equipos creados en la tabla teams de nuestra base de datos

Primera petición: http://localhost:8080/teams

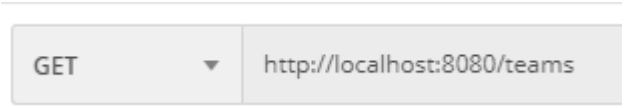


Figura 117: primera petición para obtener los equipos

Esta petición la recibirá el siguiente método de nuestro controlador:

```
@GetMapping(value="/teams")
@ResponseBody
public ResponseEntity<List<Team>> readTeams() {
    List<Team> teams = teamService.getAll();
    return ResponseEntity.ok(teams);
}
```

Figura 118: método readTeams que se ejecutará tras recibir la petición de la figura 116

El cual nos devolverá en una lista, todos los equipos que tenemos en la base de datos.

Al revisar la respuesta obtenida en POSTMAN, obtenemos los equipos:

```
[
    {
        "id": 1,
        "name": "red",
        "agents": [ ···
        ],
        "checkpoints": [ ···
        ]
    },
    {
        "id": 2,
        "name": "blue",
        "agents": [],
        "checkpoints": [
            {
                "id": 9207,
                "checkpointIndex": 1,
                "teamId": 2,
                "agentId": null,
                "passedAt": null,
                "passed": false
            }
        ]
    }
]
```


Figura 119: respuesta en POSTMAN a la petición anterior

Podemos ver como Spring transforma nuestra lista de objetos **Team** a formato JSON. Y debido a la relación Una a Muchos que configuramos en Spring, los objetos **Team** contienen también todos los objetos **Checkpoint** y **Agent** del mismo equipo.

Segunda Petición: http://localhost:8080/teams/blue

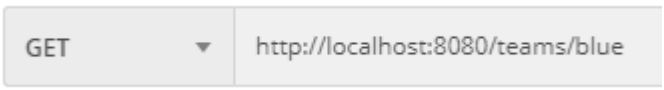


Figura 120: petición para recibir información del equipo "blue"

Esta petición llamará al siguiente método del controlador:

```java
@GetMapping(value="/teams/{teamName}")
@ResponseBody
public ResponseEntity<Team> readTeamByName(@PathVariable String teamName) {
    Team team = teamService.getByName(teamName);
    return ResponseEntity.ok(team);
}
```

Figura 121: método que se ejecutará tras recibir la petición anterior

Y este método nos devolverá el equipo cuyo nombre esté en la ruta variable, en este caso, “blue”.

Si observamos la salida de POSTMAN, encontramos el equipo azul en formato JSON.


```
{
    "id": 2,
    "name": "blue",
    "agents": [],
    "checkpoints": [
        {
            "id": 9340,
            "checkpointIndex": 1,
            "teamId": 2,
            "agentId": null,
            "passedAt": null,
            "passed": false
        }
    ]
```


Figura 122: resultado en POSTMAN de la petición

Tercera petición:

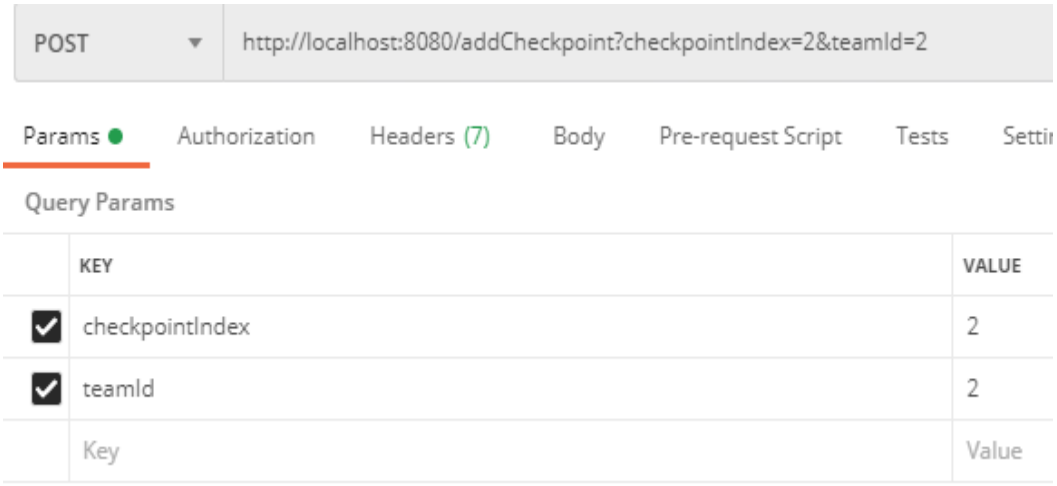


Figura 123: petición para añadir un checkpoint con índice 2 y team_id 2

Hasta ahora nos habíamos limitado a realizar peticiones de lectura de la base de datos, esta nueva petición realizará una escritura, en concreto, creará una entrada en la tabla **checkpoints** con team_id = 2 y checkpointIndex = 2.

Nótese que esta vez se trata de una petición HTTP POST.

El método al que llamará el controlador es el siguiente:

```
@PostMapping(value="/addCheckpoint")
public synchronized ResponseEntity<Checkpoint> addCheckpoint(
        @RequestParam("checkpointIndex") int checkpointIndex,
        @RequestParam("teamId") int teamId ){
    checkpointService.createCheckpoint(checkpointIndex, teamId);
    return ResponseEntity.ok(checkpointService.getCheckpoint(checkpointIndex, teamId));
}
```

Figura 124: método que se ejecutará tras recibir la petición anterior

el método creará una entrada en la base de datos con los parámetros que le hemos pasado y luego leerá la base de datos para devolvernos esa misma entrada. Revisemos la salida de POSTMAN para confirmarlo:


```
{
    "id": 9341,
    "checkpointIndex": 2,
    "teamId": 2,
    "agentId": null,
    "passedAt": null,
    "passed": false
}
```


Figura 125: Resultado en POSTMAN tras la petición

La entrada se ha creado. Es más, si utilizamos la **segunda petición**, este nuevo checkpoint debería aparecer entre la lista de Checkpoints del equipo:


```
{
    "id": 2,
    "name": "blue",
    "agents": [],
    "checkpoints": [
        {
            "id": 9340,
            "checkpointIndex": 1,
            "teamId": 2,
            "agentId": null,
            "passedAt": null,
            "passed": false
        },
        {
            "id": 9341,
            "checkpointIndex": 2,
            "teamId": 2,
            "agentId": null,
            "passedAt": null,
            "passed": false
        }
    ]
}
```


Figura 126: nuevo resultado en POSTMAN tras realizar de nuevo la segunda petición

Una vez realizadas estas pruebas, programaremos estas peticiones HTTP en Unity, para guardar los datos de nuestro entorno en la base de datos.

8.4. Consumiendo el servicio REST desde Unity

Ahora que tenemos un servicio REST funcional, sería ideal que, al iniciar una simulación, nuestros agentes y los puntos del entorno aparecieran en la base de datos y que el estado de los puntos se actualizase cuando un agente los cruza o cuando se reinicia la carrera.

Unity no tiene una manera sencilla para consumir servicios REST, así que, para simplificar el proceso, nos apoyaremos en la dependencia de dilmerv - UnityRestClient[22]. Esta dependencia nos otorgará acceso a los métodos HttpGet, HttpPost y HttpPut, que serán los que usaremos para consumir el servicio REST.

Crearemos una clase **RestExample** que tendrá todos los métodos necesarios que utilizarán nuestros agentes y nuestro entorno. Empezaremos por definir las rutas de nuestro servicio REST.

```
public class RestExample : MonoBehaviour
{
    [SerializeField]
    private string addCheckpointUrl = "http://localhost:8080/addCheckpoint?checkpointIndex={0}&teamId={1}";
    [SerializeField]
    private string addLastCheckpointUrl = "http://localhost:8080/addLastCheckpoint?checkpointIndex={0}&teamId={1}&isPassed={2}";
    [SerializeField]
    private string updateCheckpointUrl = "http://localhost:8080/updateCheckpoint?checkpointIndex={0}&teamId={1}&isMaster={2}";
    [SerializeField]
    private string countAllCheckpointsUrl = "http://localhost:8080/numberOfCheckpoints/{0}";
    [SerializeField]
    private string countPassedCheckpointsUrl = "http://localhost:8080/numberOfPassedCheckpoints/{0}";
    [SerializeField]
    private string deleteAllCheckpointsUrl = "http://localhost:8080/deleteCheckpoints";
    [SerializeField]
    private string addAgentUrl = "http://localhost:8080/addAgent?teamId={0}&isMaster={1}";
    [SerializeField]
    private string deleteAllAgentsUrl = "http://localhost:8080/deleteAgents";
```

Figura 127: propiedades de la clase RestExample con las rutas para las distintas peticiones del servicio REST

Y veamos 3 ejemplos de métodos que generan peticiones HTTP en Unity:

Método para crear un checkpoint: Al igual que en POSTMAN, le pasamos los parámetros **checkpointIndex** y **teamId** (sumándole uno, pues los índices de los equipos empiezan por 0 en unity). Estos parámetros los obtenemos del objeto **Checkpoint** que llama a este método.

```
public void createCheckpoint(Checkpoint checkpoint) {
    int checkpointIndex = checkpoint.checkpointIndex;
    int teamId = (int)checkpoint.activeTeam.team;
    StartCoroutine(RestWebClient.Instance.HttpPost(
        string.Format(addCheckpointUrl, checkpointIndex, teamId + 1),
        "body",
        (r) => OnRequestComplete(r)
        )
    );
}
```

Figura 128: método para crear un Checkpoint mediante una petición al servicio REST

Método para borrar checkpoints: este método borrará todas las entradas de checkpoints que haya en la base de datos, no se necesita ningún parámetro en la petición HTTP.

[22] https://github.com/dilmerv/UnityRestClient

```
public void deleteAllCheckpoints()
{
    StartCoroutine(RestWebClient.Instance.HttpDelete(
        string.Format(deleteAllCheckpointsUrl),
        (r) => OnRequestComplete(r)));
}
```

Figura 129: método para borrar todos los checkpoints mediante una petición al servicio REST

Este método será llamado cada vez que se inicie una nueva carrera.

Método para actualizar un Checkpoint cuando sea cruzado por un agente: este método recibe como entradas, el checkpoint cruzado, y el agente que lo cruzó; y con esto actualizará el checkpoint en la base de datos para que refleje que ha sido cruzado por el agente y cuando: nótese que este método solo llamará al servicio REST si el punto cruzado es del mismo equipo que el agente.

```
public void updateCheckpoint(Checkpoint checkpoint, AircraftAgent agent)
{
    if (checkpoint.checkPointPassed && checkpoint.activeTeam.team == agent.activeTeam.team)
    {
        int checkpointIndex = checkpoint.checkpointIndex;
        int teamId = (int)checkpoint.activeTeam.team;
        string isMaster ="false";
        if (agent.isCaptain)
        { isMaster = "true"; }
        else
        { isMaster = "false"; }
        StartCoroutine(RestWebClient.Instance.HttpPut(
            string.Format(updateCheckpointUrl, checkpointIndex, teamId + 1, isMaster),
            "body",
            (r) => OnRequestComplete(r)
            )
        );
    }
}
```

Figura 130: método para actualizar el estado de un checkpoint cuando es cruzado, mediante una petición al servicio REST

En cuanto a la implementación de estas llamadas al servicio REST dentro del código de Unity ya existente, el siguiente proceso se llevó a cabo:

Se creó un objeto Unity que contiene el script **RestExample**

Figura 131: Nuevo Objeto Unity añadido a la escena para las peticiones al servicio REST

Después, se creó un parámetro en el **AircraftArea** para poder activar y desactivar el uso de servicios REST.

```
[Tooltip("If true, enable using REST services")]
public bool RESTMode;
```

Figura 132: booleano que controlar si se deben utilizar los servicios REST

Al iniciar el área, se obtiene el objeto **RestExample**

```
private void Awake()
{
    if (RESTMode)
    {
        this.restExample = GetComponentInChildren<RestExample>();
    }
```

Figura 133: obtención del objeto RestExample cuando se inicializa AircraftArea

Se llama al servicio REST para borrar todos los agentes y puntos antiguos, y con los agentes que existen en el entorno, se crean nuevas entradas en la base de datos

```
if (RESTMode) {
    this.restExample.deleteAllAgents();
    this.restExample.deleteAllCheckpoints();
    foreach (AircraftAgent agent in AircraftAgents) {
        restExample.createAgent(agent);
    }
}
```

Figura 134: código que borra agentes y puntos antiguos, y añade a los nuevos agentes que existen dentro de AircraftArea

A continuación, mientras se instancian los distintos puntos del entorno, y aprovechando el bucle *for* en el que esto ocurre, se llama también al servicio REST para crear una entrada de cada punto en la base de datos:

```
restExample.createCheckpoint(checkpoint);
```

Figura 135: creación de una entrada en la base de datos por cada Checkpoint inicializado en AircraftArea

Luego se aprovecha el método **CheckpointEntered**(**Agent** agent) del objeto **Checkpoint** para llamar al servicio REST cuando el checkpoint pase al estado **checkPointPassed = true**

```
this.GetComponent<MeshRenderer>().material = fadedMaterial;
checkPointPassed = true;
if (agent.getArea().RESTMode) {
    agent.getArea().restExample.updateCheckpoint(this, agent);
}
```

Figura 136: actualización del Checkpoint cuando este es cruzado

Y, por último, al reiniciar la carrera, se borran los puntos de la base de datos y se vuelven a crear nuevas entradas para los mismos.

```
if (this.RESTMode) {
    this.restExample.deleteAllCheckpoints();
}
```

Figura 137: borrado de las entradas de los puntos al terminar la carrera

```
foreach (Checkpoint checkpoint in listofcheckpoints)
{
    if (this.RESTMode) {
        if (i == listofcheckpoints.Count - 1)
        {
            this.restExample.createLastCheckpoint(checkpoint);
        }
        else
        {
            this.restExample.createCheckpoint(checkpoint);
        }
    }
```

Figura 138: creación de nuevas entradas para los puntos

Y con esto ya podemos probar a consumir el servicio REST desde Unity al iniciar una carrera:

Empezamos borrando todos los puntos y agentes de la base de datos, mediante POSTMAN

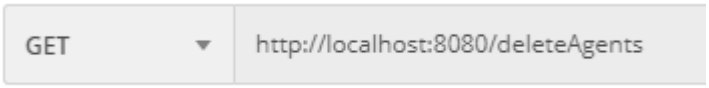


Figura 139: petición para borrar todos los agentes de la base de datos

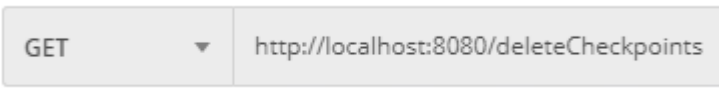


Figura 140: petición para borrar todos los puntos de la base de datos

Y ahora, el equipo rojo no tiene ni agentes ni puntos:


```
{
    "id": 1,
    "name": "red",
    "agents": [],
    "checkpoints": []
}
```


Figura 141: resultado en POSTMAN de la petición /teams/red

Iniciamos el entorno de Unity

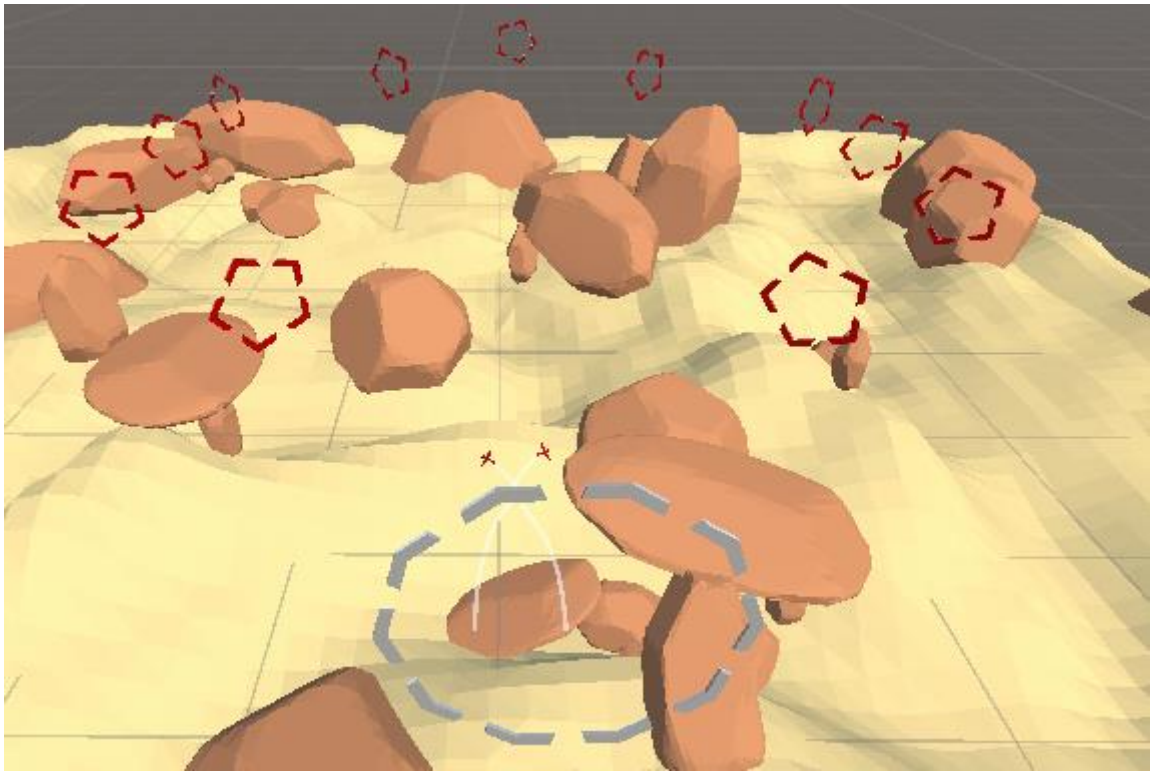


Figura 142: entorno de unity inicializado

Y revisamos de nuevo el equipo rojo en POSTMAN

```
{
    "id": 1,
    "name": "red",
    "agents": [
        {
            "id": 77,
            "teamId": 1,
            "checkpoints": [],
            "master": false
        },
        {
            "id": 78,
            "teamId": 1,
            "checkpoints": [],
            "master": true
        }
    ],
    "checkpoints": [
        {
            "id": 9402,
            "checkpointIndex": 0,
            "teamId": 1,
            "agentId": null,
            "passedAt": null,
            "passed": false
        },
        {
```


Figura 143: resultado de la petición /teams/red en POSTMAN

Ahora tiene dos agentes y 12 puntos. Lo que concuerda con los agentes y puntos del entorno

Si quitamos la pausa al entorno y dejamos que cada agente cruce un punto:

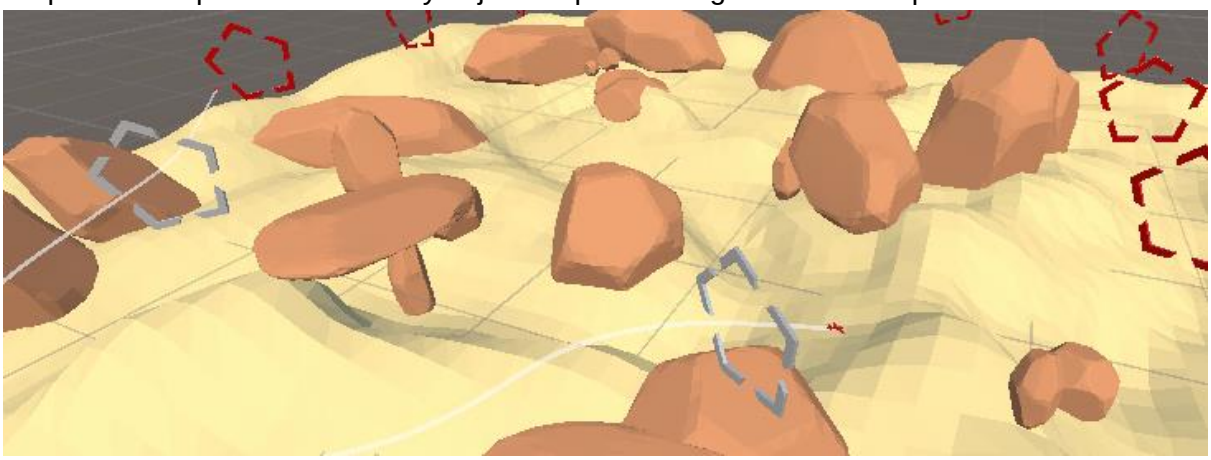


Figura 144: entorno después de que cada agente cruzara un punto

Podemos confirmar que ahora los agentes tienen un punto cada uno:

```
"agents": [
    {
        "id": 77,
        "teamId": 1,
        "checkpoints": [
            {
                "id": 9413,
                "checkpointIndex": 1,
                "teamId": 1,
                "agentId": 77,
                "passedAt": "2020-11-21 07:31:46",
                "passed": true
            }
        ],
        "master": false
    },
    {
        "id": 78,
        "teamId": 1,
        "checkpoints": [
            {
                "id": 9402,
                "checkpointIndex": 0,
                "teamId": 1,
                "agentId": 78,
                "passedAt": "2020-11-21 07:31:47",
                "passed": true
```


Figura 145: resultado en POSTMAN de la petición /teams/red enfocándonos en los agentes del equipo

Ahora que hemos comprobado que las entradas en la base de datos se crean correctamente, procederemos a visualizar los datos en tiempo real en el próximo apartado.

9. Visualización de los datos en tiempo real: Elixir + Phoenix

Para poder visualizar en tiempo real nuestros datos, utilizaremos un servidor web que vaya haciendo llamadas a la base de datos cada cierto tiempo y actualizando una vista en una página web con los nuevos datos.

Spring nos ofrece la posibilidad de realizar esto, sin embargo, con el objetivo de explorar distintas opciones, utilizaremos otro lenguaje de programación para esta tarea. Hasta ahora nos hemos apoyado en programación orientada a objetos, así que, para esta última tarea, cambiaremos el paradigma y utilizaremos Elixir, concretamente el web framework, Phoenix.

9.1. ¿Por qué Elixir + Phoenix?

En la actualidad y con la ayuda de internet, todo servicio que ofrezcamos debe estar preparado para proveer a una gran cantidad de usuarios. Para ello, necesitamos sistemas escalables, bajos en latencia y resilientes a errores.

Elixir[23] es un lenguaje de programación funcional, que utiliza la máquina virtual de Erlang[24], la cual fue creada en un principio para servicios de telecomunicaciones, donde las características previamente mencionadas son esenciales, debido a la alta concurrencia dentro de estos servicios.

Por otra parte, el framework web de Elixir, Phoenix[25], nos aporta una manera simple de crear servicios web de todo tipo.

En los próximos apartados, veremos lo sencillo que resulta la creación de una página web donde visualizar en tiempo real nuestra base de datos.

9.2. Creando un proyecto con Elixir y Phoenix Liveview

9.2.1. Creación del proyecto inicial

Crear un proyecto de Elixir que incluya el framework Phoenix y las dependencias Liveview resulta extremadamente sencillo. Asumiendo que tenemos instalado Elixir en nuestro sistema, es tan sencillo como ejecutar el siguiente comando:

```
mix phx.new live_app --live
```

Esto creará nuestro proyecto inicial que tendrá la siguiente estructura:

[23] https://elixir-lang.org/
[24] https://www.erlang.org/
[25] https://www.phoenixframework.org/

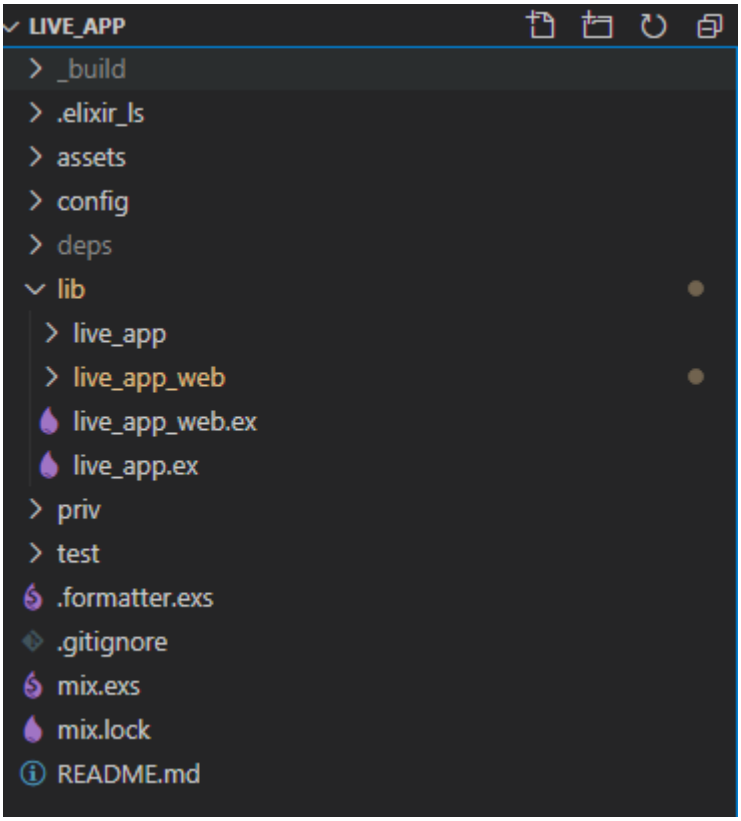


Figura 146: estructura inicial del proyecto Phoenix

9.2.2. Creando modelos con Ecto

Al igual que para el servidor REST, empezaremos por crear nuestros modelos para las 3 tablas de la base de datos.

Para crear los modelos en Elixir, nos ayudaremos de la librería **Ecto**, la cual se encargará de hacer las llamadas a la base de datos y de crear nuestros modelos con los campos que le especifiquemos.

Se explicará cómo se crea el modelo **Checkpoint** y se mostrará brevemente los modelos **Team** y **Agent.** El proceso guarda cierta similitud al que seguíamos en Spring.

Primero definimos el módulo que servirá como nuestro modelo:

```
defmodule LiveApp.AircraftArea.Checkpoint do
  use Ecto.Schema
  import Ecto.Changeset
  alias LiveApp.AircraftArea.Team
  alias LiveApp.AircraftArea.Agent
  alias LiveApp.AircraftArea.Checkpoint
```

Figura 147: módulo Checkpoint que contendrá nuestro modelo

Use: nos permite inyectar código del módulo descrito.
Import: importa el módulo, lo que nos permite llamar a sus métodos públicos.
Alias: Es opcional. Nos ayuda a simplificar la llamada a otros módulos. Por ejemplo, en vez de tener que escribir **LiveApp.AircraftArea.Team**, bastará con escribir **Team.**

Todo modelo necesita un **schema** de **Ecto**, es el sitio donde definiremos las columnas de la tabla. Definimos cada campo con la palabra clave **field**, si se trata de una columna independiente de la tabla, y con la palabra clave **belongs_to** si se trata de una llave foránea.

```
schema "checkpoints" do
  field :checkpoint_index, :integer
  field :is_passed, :boolean
  field :passed_at, :utc_datetime
  belongs_to :team, Team, primary_key: true
  belongs_to :agent, Agent
end
```

Figura 148: schema del módulo Checkpoint

Vemos también que en la misma línea en la que definimos cada campo, también definimos el tipo de dato, y también se podrían definir algunos parámetros más (como que el campo pueda ser nulo, o la longitud máxima, en el caso de los strings).

Una vez hecho el schema, ya tenemos un modelo funcional. Pero en Elixir es común agregar el método **changeset** que se encargará de hacer validaciones con los datos antes de introducirlos en la base de datos. Es como definir en SQL restricciones, pero estas se validan en elixir en vez de en la base de datos:

```
def changeset(%Checkpoint{} = checkpoint, attrs) do
  checkpoint      You, a week ago • Live App first commit
  |> cast(attrs, [:checkpoint_index, :is_passed, :passed_at, :team_id, :agent_id])
  |> validate_required(:checkpoint_index)
  |> foreign_key_constraint(:team_id)
end
```

Figura 149: método changeset para realizar validaciones de nuestro modelo

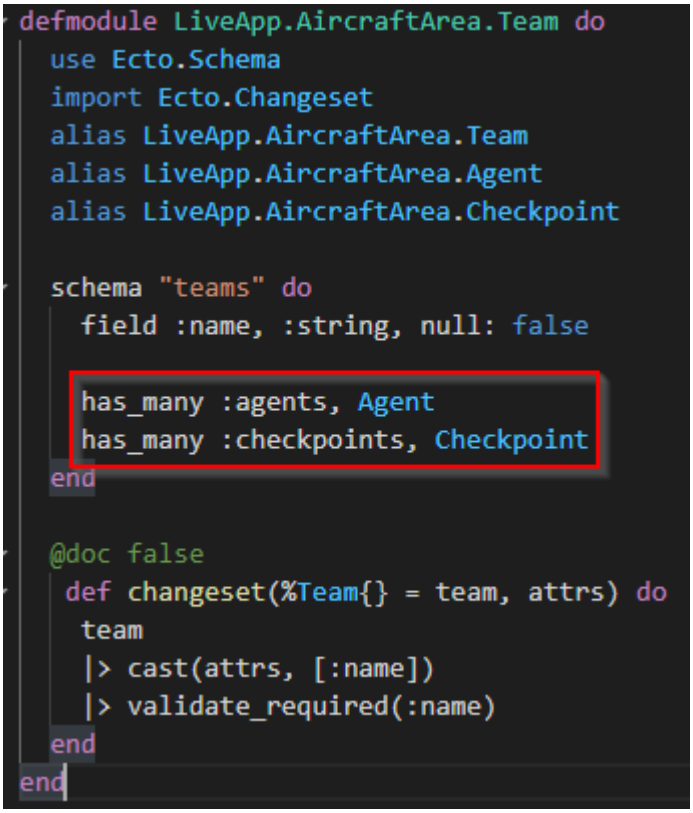
La función **cast** sirve para indicar los posibles campos para validar nuestro modelo.

La función **validate_required** obligará al modelo a que tenga un checkpoint_index definido.

La función **foreign_key_constraint** sirve para comprobar que la llave foránea **team_id** existe en la base de datos antes de introducir los datos.

Una vez creado el modelo **Checkpoint** procederemos a mostrar los otros dos modelos creados:

Team



```elixir
defmodule LiveApp.AircraftArea.Team do
  use Ecto.Schema
  import Ecto.Changeset
  alias LiveApp.AircraftArea.Team
  alias LiveApp.AircraftArea.Agent
  alias LiveApp.AircraftArea.Checkpoint

  schema "teams" do
    field :name, :string, null: false

    has_many :agents, Agent
    has_many :checkpoints, Checkpoint
  end

  @doc false
  def changeset(%Team{} = team, attrs) do
    team
    |> cast(attrs, [:name])
    |> validate_required(:name)
  end
end
```

Figura 150: modelo Team

Nótese que al igual en Spring, ponemos en el schema la relación **has_many** para indicar que, al leer un equipo de la base de datos, también queremos obtener los agentes y los checkpoints del mismo.

Agent

```elixir
defmodule LiveApp.AircraftArea.Agent do
  use Ecto.Schema
  import Ecto.Changeset
  alias LiveApp.AircraftArea.Team
  alias LiveApp.AircraftArea.Agent
  alias LiveApp.AircraftArea.Checkpoint

  schema "agents" do
    field :is_master, :boolean
    belongs_to :team, Team

    has_many :checkpoints, Checkpoint
  end
  @doc false
  def changeset(%Agent{} = agent, attrs) do
    agent
    |> cast(attrs, [:team_id])
    |> validate_required(:team_id)
    |> foreign_key_constraint(:team_id)
  end
end
```

Figura 151: modelo Agent

Una vez creado los modelos, procederemos a crear algo similar al servicio de Spring, pero en este caso, será un único servicio llamado **AircraftArea.**

Definimos el módulo:

```elixir
defmodule LiveApp.AircraftArea do
  require Ecto.Query
  alias Ecto.Query
  alias LiveApp.Repo
  alias LiveApp.AircraftArea.{
    Team      You, 6 minutes ago
  }
```

Figura 152: definición del módulo AircraftArea

Y creamos dos funciones para leer un equipo de la base de datos(**team_data**) y otro para el nombre y la id de todos los equipos existentes (**list_of_teams**).

```elixir
def team_data(team_id) do
  Team
  |> Query.where(id: ^team_id)
  |> Repo.one       You, seconds ago • Uncommitted changes
end

@spec list_of_teams :: any
def list_of_teams() do
  query = Query.from t in Team, select: %{id: t.id, name: t.name}

  Repo.all(query)
end
```

Figura 153: métodos para obtener los equipos de la base de datos

Vemos como no tenemos que escribir código SQL directamente, pero sí que debemos usar cláusulas típicas de SQL: where, from, select…

El módulo **Repo**, autogenerado por el módulo **Ecto** de Elixir, es el que se encarga de hacer las llamadas a la base de datos. **Repo.all** nos devolverá todos los resultados de la llamada, **Repo.one** solo funcionará si la llamada a la base de datos nos devuelve un único resultado.

Veamos cómo queda la estructura de nuestro proyecto ahora:

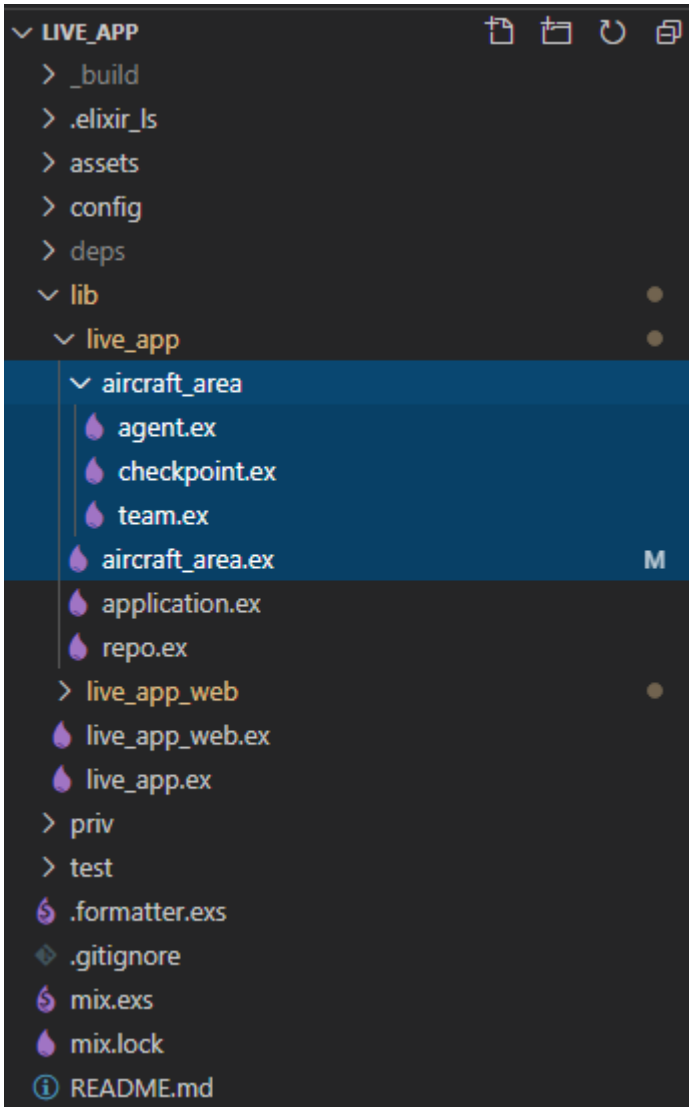


Figura 154: estructura del proyecto Phoenix tras la creación de los modelos y el servicio

9.2.3. Creando la vista con LiveView

Ahora que tenemos nuestro servicio y nuestros modelos creados, es hora de pasar a programar la vista. Al tratarse de una **LiveView** de **Phoenix**, no necesitaremos usar la capa del controlador, solo será necesario cargar nuestra vista con un módulo que debe tener al menos dos métodos: **mount** y **handle_event.**

*Para más información sobre el ciclo de vida de una LiveView, recomiendo mirar este vídeo[26] de uno de los creadores de Liveview (particularmente, del minuto 18:30 a 23).

Definimos el módulo **TeamsLive**:

[26] https://www.youtube.com/watch?v=8xJzHq8ru0M&feature=youtu.be&t=1098

```
defmodule LiveAppWeb.TeamsLive do
  use LiveAppWeb, :live_view
  alias LiveApp.AircraftArea
```

Figura 155: módulo TeamsLive

Use nos sirve para indicar que utilizaremos las dependencias de LiveView, las cuales han sido cargadas automáticamente en nuestro proyecto a la hora de crearlo.

Creamos la función **mount**, esta función es llamada dos veces, una primera vez para mantener la vista estática sin estados y una segunda vez, cuando se haya establecido una conexión websocket con la LiveView. Para el usuario final, será como si solo se llamase una única vez.

```
def mount(_params, _session, socket) do
  team_id = 1
  team = get_preloaded_team(team_id)
  available_teams = get_list_of_teams()

  if connected?(socket) do
    Process.send_after(self(), :update, 2000)
  end
  {:ok, assign(socket, team_id: team_id, team: team, available_teams: available_teams)}
end
```

Figura 156: método mount de la LiveView

Todos los datos que nuestra LiveView maneja se encuentran en tres sitios posibles: params, session y socket. Para este ejemplo, solo emplearemos el socket.

Lo primero que haremos será predefinir el team_id que queremos visualizar, por defecto es 1 (el del equipo rojo).

```
team_id = 1
```

Figura 157: team_id del equipo que visualizaremos por defecto

Lo siguiente será coger de la base de datos el equipo en cuestión

```
team = get_preloaded_team(team_id)
```

Figura 158: cargamos el equipo de nuestra base de datos

get_preloaded_team(team_id) es un método que precargará nuestro equipo con los agentes y checkpoints que hagan referencia a él. Para realizar esta precarga, se utiliza **Repo.preload**

```
defp get_preloaded_team(team_id) do
  AircraftArea.team_data(team_id)
  |> LiveApp.Repo.preload([:checkpoints, agents: :checkpoints])
end
```

Figura 159: precarga de los agentes y checkpoints del equipo

Después cargaremos el id y el nombre de todos los equipos disponibles, esto nos servirá luego para crear dos botones para cambiar entre la visualización de datos del equipo rojo o del equipo azul:

```
available_teams = get_list_of_teams()
```

Figura 160: carga del id y nombre de los equipos disponibles

get_list_of_teams() nos proporcionará estos datos, en este caso no será necesario precargar agentes ni checkpoints, pues solo queremos el id y el nombre de cada equipo:

```
defp get_list_of_teams() do
  AircraftArea.list_of_teams()
end
```

Figura 161: llamada al servicio que nos proporcionará la lista de equipos

Para que la vista se actualice en vivo, debemos crear dentro del proceso un evento que se ejecute cada cierto tiempo, sin embargo, solo queremos hacerlo una vez se haya establecido la conexión websocket con el usuario final:

```
if connected?(socket) do
    Process.send_after(self(), :update, 2000)
end
```

Figura 162: generación de un evento :update si la conexión con el websokcet ha sido realizada

Después de dos segundos, el evento **:update** será enviado a este mismo proceso, pero solo ocurrirá si el websocket ya se ha establecido. Más adelante, veremos cómo el código actuará cuando reciba este evento **:update**.

Y, como última línea de código del mount, actualizaremos nuestro socket para que tenga los nuevos datos:

```
{:noreply, assign(socket, team_id: team_id, team: team, available_teams: available_teams)}
```

Figura 163: actualización del socket con los datos obtenidos previamente

Con el mount listo, ya podremos cargar nuestra vista, pero para que reaccione a eventos que realice el usuario, será necesario otro método: **handle_event**.

Ahora crearemos el método **handle_event**. Idealmente esto se realizaría después de haber creado la vista y habiendo definidos el nombre y los valores de los eventos que el usuario pueda realizar. En nuestro caso, queremos que el usuario pueda cambiar entre visualizar al equipo rojo y el azul, dos botones serán suficientes:

Figura 164: botones para cambiar entre equipos en la vista

Llamaremos al evento “change” y el valor que recibiremos será la id del equipo en cuestión.

Sabiendo esto, el método handle_event quedaría de esta forma:

```
def handle_event("change", %{"teamid" => team_id}, socket) do
  team = get_preloaded_team(team_id)
  {:noreply, assign(socket, team_id: team_id, team: team)}
end
```

Figura 165: método handle_event que se ejecutará al presionar un botón

Vemos encuadrado en rojo en la figura 164, el nombre del evento y el nombre que le daremos a la variable que contiene el valor (id del equipo). Aquí nos estamos aprovechando de la búsqueda de patrones (pattern matching) de elixir. Este método solo se llamará si el evento tiene nombre "**change**" y el valor tiene nombre "**teamid**". En elixir es muy común aprovecharse de este "pattern matching", evitando el uso de **If/else**.

Una vez hemos recibido el evento con la id del equipo que queremos visualizar, solo tenemos que cargar los datos del equipo:

```
team = get_preloaded_team(team_id)
```

Figura 166: cargamos los datos del equipo a visualizar

Y lo reasignamos al socket:

```
{:noreply, assign(socket, team_id: team_id, team: team)}
```

Figura 167: asignamos los nuevos datos al socket

Con esto ya podríamos pasar a escribir el código HTML de nuestra vista. Sin embargo, como queremos que se actualice en tiempo real, necesitamos escribir un método más **handle_info**:

```
def handle_info(:update, socket) do
  Process.send_after(self(), :update, 1000)
  team_id = Map.get(socket.assigns, :team_id, 1)
  team = get_preloaded_team(team_id)
  available_teams = get_list_of_teams()

  {:noreply, assign(socket, team_id: team_id, team: team, available_teams: available_teams)}
end
```

Figura 168: método que se ejecutará al recibir un evento :update

Lo primero que vemos es que este método solo funcionará si el proceso recibe un evento del tipo **:update**. Recordemos que este es el evento que enviamos en el **mount** una vez habíamos establecido la conexión con el websocket:

```
if connected?(socket) do
    Process.send_after(self(), :update, 2000)
end
```

Figura 169: código que envía el evento :update desde la función mount

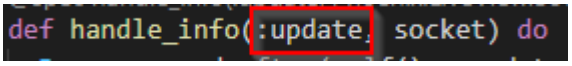


Figura 170: handle_info solo se ejecutará ante un evento :update

Una vez estamos dentro del método, lo primero es enviar otro evento :**update** para que la vista se actualice constantemente.

```
Process.send_after(self(), :update, 1000)
```

Figura 171: nuevo evento :update que se enviará desde el método handle_info

El código que viene a continuación es muy similar al del mount:

```
team_id = Map.get(socket.assigns, :team_id, 1)
team = get_preloaded_team(team_id)
available_teams = get_list_of_teams()

{:noreply, assign(socket, team_id: team_id, team: team, available_teams: available_teams)}
```

Figura 172: código del método handle_info para actualizar los datos del equipo

La única diferencia reside en que el team_id puede estar ya presente en el socket, por lo cual, debemos intentar conseguirlo del mismo, y en caso de no poder conseguirlo, le asignaremos 1 por defecto (equipo rojo).

Con esto solo nos quedaría por crear los archivos HTML para nuestra vista. Revisemos la estructura que tendrá el proyecto con los archivos HTML

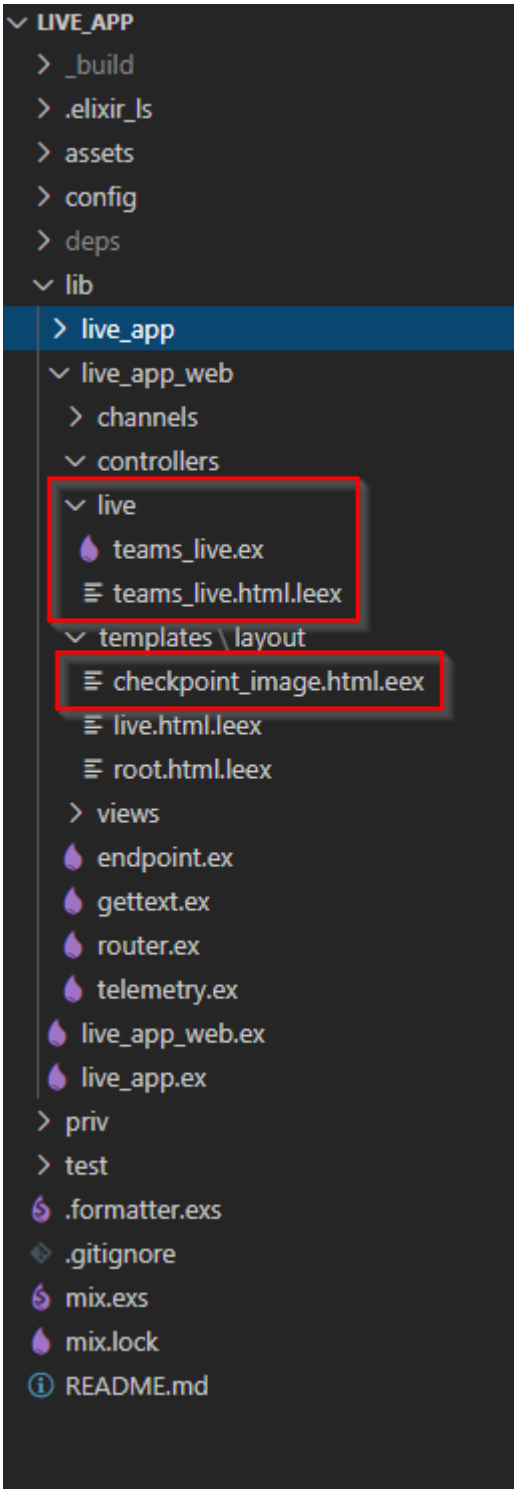


Figura 173: estructura del proyecto con la LiveView y las plantillas HTML creadas

Como el framework Phoenix realiza el renderizado de las vistas en el servidor (a diferencia de, por ejemplo, Angular), los archivos HTML dinámicos suelen terminar en .eex o, en el caso de las LiveView, en .leex. Esto nos permite ejecutar código de Elixir directamente en el archivo HTML.

Nota: los estilos css utilizados en la vista se sirven de TailwindCSS[27], la explicación de tailwindCSS queda fuera del alcance de este proyecto.

Para acceder a la vista desde el navegador, por defecto debemos acceder a **http://localhost:4000/**

En el archivo **router.ex** definimos las distintas rutas y las vistas a las que debemos llamar:

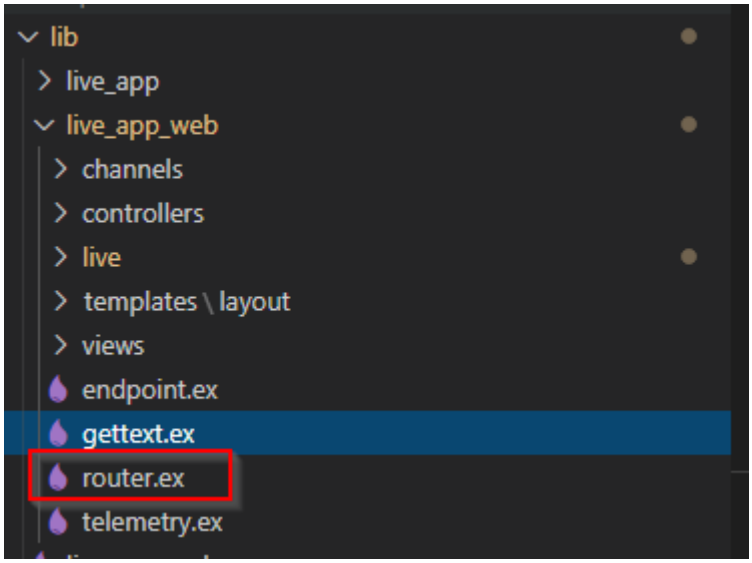


Figura 174: localización del archivo router.ex en el proyecto

```
scope "/", LiveAppWeb do
  pipe_through :browser
  live "/", TeamsLive, :index
end
```

Figura 175: ruta en la que se mostrará nuestra LiveView

la ruta "/" nos llevará al módulo de nuestra LiveView **TeamsLive.**

Ahora, empecemos por ver el diseño final que queremos que tenga la vista:

[27] https://tailwindcss.com/

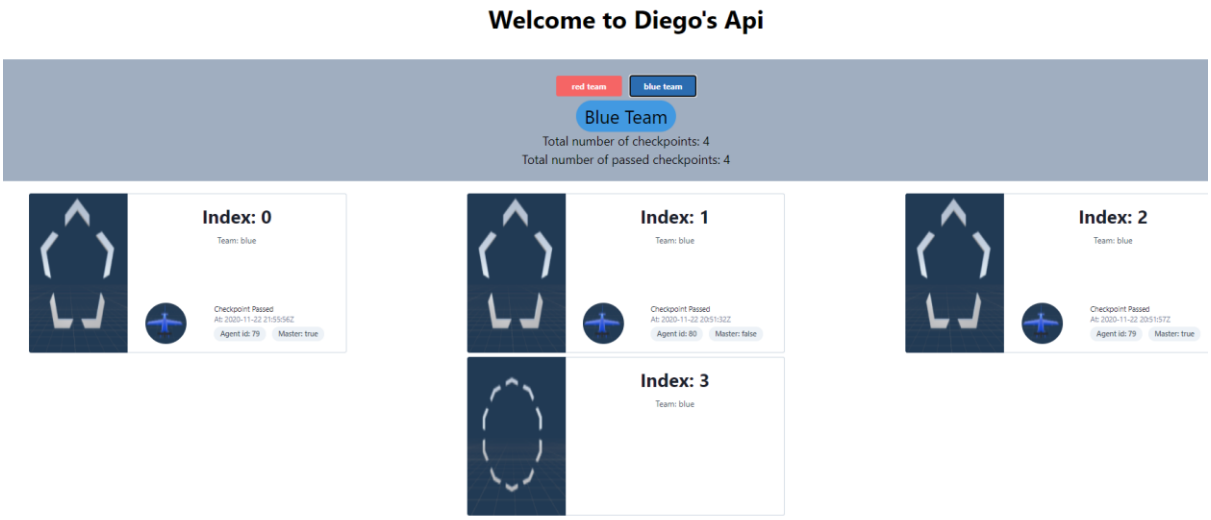


Figura 176: vista previa del diseño final que tendrá la página

Lo primero es crear el título:

```
<h1 class="text-2xl lg:text-5xl font-bold"> Welcome to Diego's Api </h1>
```

Figura 177: creación del título mediante código elixir embedido

Luego viene lo más interesante, la selección de equipo: todas las variables dentro de socket.assigns, son accesibles desde el archivo teams_live.html.leex, mediante "@xxx" donde "xxx" es el nombre de la variable. Para ejecutar código elixir en el archivo .html.leex nos servimos del tag "<%= %>".

```
<div class="flex flex-row justify-evenly">
  <%= for team <- @available_teams do %>
        <button phx-click="change"
          phx-value-teamid="<%= team.id %>"
           class="btn btn-<%=team.name %>">
            <%= team.name <> " team" %>
        </button>
  <% end %>
</div>
```

Figura 178: creación de los botones para seleccionar el equipo a visualizar

Si nos vamos al **mount** de nuestra LiveView en **teams_live.ex**, recordaremos que available_teams es una variable que contiene un mapa con las id y los nombres de los equipos:

```
available_teams = get_list_of_teams()
```

Figura 179: available_teams es un mapa que contiene las id y los nombres de los equipos

Para crear el botón para cambiar la vista entre cada equipo, realizamos un bucle for con todos los equipos posibles (**available_teams**)

```
for team <- @available_teams do
```

Figura 180: bucle for para la creación de un botón para cada equipo

En cada bucle nos limitamos a crear un botón, con los siguientes atributos:
phx-click: es el evento que se ejecutará en la LiveView al hacer click en el elemento

```
phx-click="change"
```

Figura 181: definición del nombre que tendrá el evento que ocurra cuando se haga click en este elemento HTML

phx-value-xxx: Es el valor que se pasará a la LiveView cuando ocurra el evento, "xxx" es el nombre que tendrá la variable que guarde ese valor.

```
phx-value-teamid="<%= team.id %>"
```

Figura 182: nombre y valor de una variable que se transmitirá junto al evento anterior

con `<%= team.id %>` podemos escribir de forma dinámica el id de cada equipo dentro del bucle for.

Si recordamos el método **handle_event** de la LiveView:

```
def handle_event("change", %{"teamid" => team_id}, socket) do
  team = get_preloaded_team(team_id)
  {:noreply, assign(socket, team_id: team_id, team: team)}
end
```

Figura 183: método handle_event de la LiveView teams_live.ex

Observamos que ahí también tenemos definido el nombre del evento "**change**" y el nombre de la variable "**teamid**".

Para el texto del botón recurrimos a <%= team.name %>

```
<%= team.name <> " team" %>
```

Figura 184: texto del botón para visualizar el equipo

Cómo tenemos dos equipos en la base de datos:

	id [PK] bigint	name character varying (30)
1	1	red
2	2	blue

Figura 185: equipos presentes en la base de datos

Se crearán dos botones

Figura 186: botones creados en la vista

Si revisamos el código html de los botones generados:

```
<button phx-click="change" phx-value-teamid="1" class="btn btn-red">
red team            </button> == $0
<button phx-click="change" phx-value-teamid="2" class="btn btn-blue">
blue team           </button>
```

Figura 187: código HTML de los botones

Observamos como la LiveView recibirá el teamid=1 para el botón rojo y teamid=2 para el botón azul.

Para el nombre del equipo y el número total de puntos, nos limitamos a coger el valor de la variable team y contar el número de checkpoints en la variable team.

```
<h3 class="py-1 px-4 text-xl lg:text-4xl rounded-full bg-<%=@team.name %>-500">
  <%= String.capitalize(@team.name) <> " Team" %>
</h3>
<h3 class="text-lg lg:text-2xl">
  Total number of checkpoints: <%= Enum.count(@team.checkpoints) %>
</h3>
```

Figura 188: código HTML con elixir embedido para mostrar el equipo que estamos visualizando y el número de puntos que tiene ese equipo

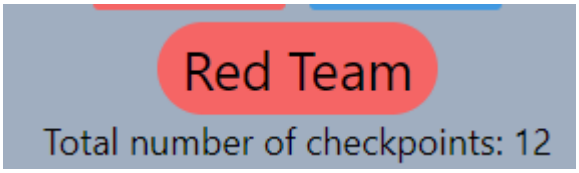


Figura 189: resultado en la vista del código anterior

Para el número de puntos cruzados, también nos limitamos a contar los puntos del equipo, pero solo aquellos cuyo valor *is_passed* esté a *true*.

```
<h3 class="text-lg lg:text-2xl">
  Total number of passed checkpoints: <%=  Enum.count(@team.checkpoints,fn c -> c.is_passed end) %>
</h3>
```

Figura 190: código para mostrar el número de puntos cruzados en la vista

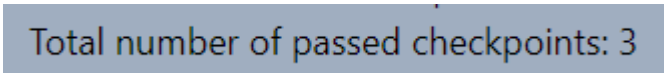


Figura 191: resultado en la vista del código anterior

Y ya para acabar debemos crear las “cartas” de cada checkpoint:

Cuando el punto no ha sido cruzado:

Figura 192: carta de información de un punto que no ha sido cruzado

Y cuando ha sido cruzado:

Figura 193: carta de información de un punto que ha sido cruzado

Empezaremos creando un bucle for que recorra todos los puntos del equipo, también los ordenamos según su índice de menor a mayor:

```
<%= for checkpoint <- Enum.sort(@team.checkpoints, fn a, b -> a.checkpoint_index < b .checkpoint_index end) do %>
```

Figura 194: bucle for para la generación de todas las cartas de los checkpoints del equipo

Lo primero que hacemos es renderizar la imagen del punto: El método render nos permite llamar otra plantilla .html donde podemos poner código repetitivo, no es necesario para este proyecto, pero es una práctica muy común cuando se trabaja en un proyecto con más personas.

```
<%= render LiveAppWeb.LayoutView, "checkpoint_image.html", conn: @socket, checkpoint: checkpoint, is_start: Enum.count(@team.checkpoints)- 1 == checkpoint.checkpoint_index %>
```

Figura 195: carga de la plantilla que renderiza la imagen del checkpoint según su color y estado

Esta nueva vista no tendrá acceso a las variables de la vista padre salvo que se las pasemos explícitamente. En nuestro caso, solo deberemos pasarle el punto actual que queremos dibujar "**checkpoint**" y una variable para saber si este punto es el punto inicial **is_start**; en nuestro proyecto de Unity, el punto "inicial" es el último punto de la lista, es decir, el que tiene el checkpoint_index mayor.

Dentro de la plantilla checkpoint_image.html tenemos el siguiente código:

Tenemos un algo similar a un *switch*/*case* para cargar la imagen dependiendo de si es de color azul, rojo o si ha sido cruzada.

```
<%= if @is_start do %>
```

Figura 196: condición para renderizar distintas imágenes dependiendo de si es el punto inicial o no

```
<%= cond do %>
    <% @checkpoint.is_passed -> %>
        <img class="checkpoint-image" src="<%= Routes.static_path(@conn,"/images/faded_start.png")%>" alt="faded_start">
    <% @checkpoint.team_id == 1-> %>
        <img class="checkpoint-image" src="<%= Routes.static_path(@conn,"/images/red_start.png")%>" alt="faded_red">
    <% @checkpoint.team_id == 2 -> %>
        <img class="checkpoint-image" src="<%= Routes.static_path(@conn,"/images/blue_start.png")%>" alt="faded_blue">
    <% true -> %>
        <img class="checkpoint-image" src="<%= Routes.static_path(@conn,"/images/faded_start.png")%>" alt="faded_start">
<% end %>
```

Figura 197: switch/case para el renderizado de las distintas imágenes del punto inicial

Este *switch/case* se repite una vez más, pero cargando imágenes diferentes si el punto no es inicial

```
<% else %>
    <%= cond do %>
        <% @checkpoint.is_passed -> %>
            <img class="checkpoint-image" src="<%= Routes.static_path(@conn,"/images/faded_checkpoint.png")%>" alt="faded_checkpoint">
        <% @checkpoint.team_id == 1-> %>
            <img class="checkpoint-image" src="<%= Routes.static_path(@conn,"/images/red_checkpoint.png")%>" alt="red_checkpoint">
        <% @checkpoint.team_id == 2 -> %>
            <img class="checkpoint-image"src="<%= Routes.static_path(@conn,"/images/blue_checkpoint.png")%>" alt="blue_checkpoint">
        <% true -> %>
            <img class="checkpoint-image" src="<%= Routes.static_path(@conn,"/images/faded_checkpoint.png")%>" alt="faded_checkpoint">
    <% end %>
```

Figura 198: switch/case para el renderizado de las distintas imágenes del resto de puntos

Routes.static_path es un método que nos da la dirección de nuestros archivos estáticos, debemos usar este método para evitar *hardcodear* rutas de nuestros archivos, lo cual nos podría causar problemas si desplegamos esta aplicación en algún servidor que no sea local. Los archivos estáticos los cargamos en la carpeta assets del proyecto:

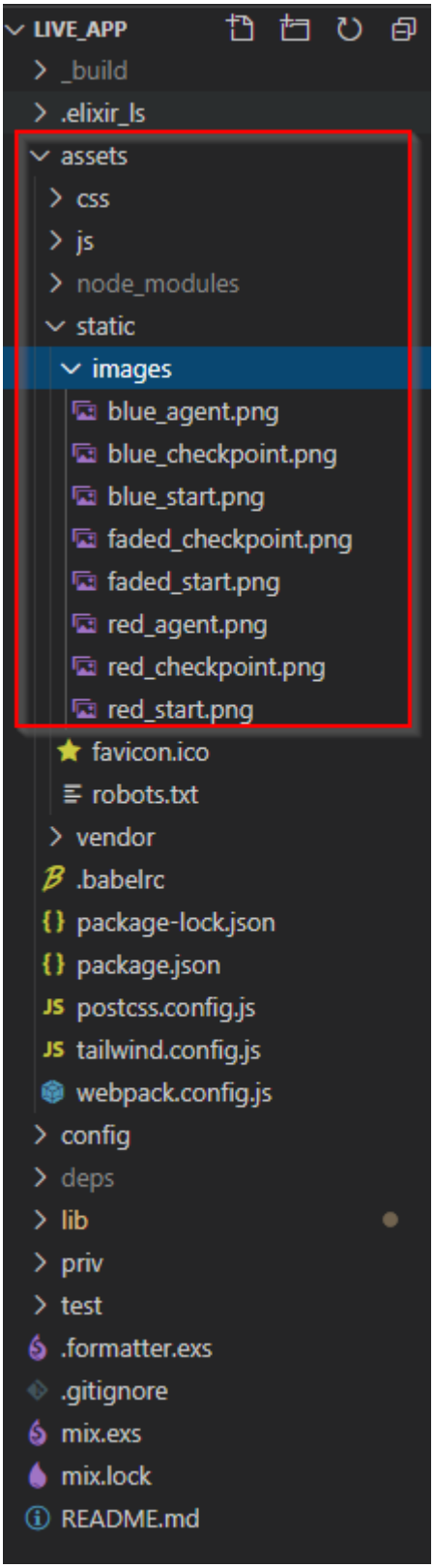


Figura 199: localización de los archivos estáticos del proyecto Phoenix

Veamos el resultado en html de todo esto para uno de los casos:

Figura 200: resultado en la vista de un punto cruzado

```
<img class="checkpoint-image" src="/images/faded_checkpoint.png" alt="faded_checkpoint">
```

Figura 201: código HTML generado

Para escribir el índice del punto y el nombre del equipo al que pertenece, recurrimos a **`checkpoint.checkpoint_index`** y **`team.name`**

```
<div class="text-gray-900 font-bold text-4xl mb-2">Index: <%= checkpoint.checkpoint_index %></div>
<p class="text-gray-700 text-base">Team: <%= @team.name %></p>
```

Figura 202: código para escribir el índice del punto y el equipo al que pertence

team lleva un "@" pues es una variable que proviene del socket, sin embargo, **checkpoint** es una variable creada por nuestro bucle *for* dentro de la vista y no se necesita del "@" para poder acceder a ella

En html el código anterior se ve así:

Index: 0

Team: red

Figura 203: resultado en la vista del código anterior

La siguiente parte del HTML solo se escribe si el punto ha sido cruzado y no se trata del punto inicial

```
<%= if  checkpoint.is_passed and checkpoint.checkpoint_index != Enum.count(@team.checkpoints) - 1 do %>
```

Figura 204: condición para ejecutar el código que generará la información adicional de un punto cruzado

La imagen del agente se carga igual que la del checkpoint, pero esta vez no se realiza en una plantilla externa:

```
<%= if checkpoint.agent_id != nil do %>
```

Figura 205: condición para pintar la imagen del agente solo si el checkpoint tiene definido un agent_id

```
<img class="w-20 h-20 rounded-full mr-4" src="<%= Routes.static_path(@socket,"/images/#{@team.name}_agent.png")%>" alt="agent">
```

Figura 206: código para el renderizado de la imagen del agente

dando lugar a este código HTML en caso de pintarse (para el caso del avión rojo):

```
<img class="w-20 h-20 rounded-full mr-4" src="/images/red_agent.png" alt="agent">
```

Figura 207: código HTML que ha sido generado por el código de la figura anterior

El resto de los datos se cargan también si existen dentro de la variable checkpoint:

Si el punto fue cruzado:

```
<p class="text-gray-900 leading-none">Checkpoint Passed</p>
```

Figura 208: código para mostrar que el punto fue cruzado

Checkpoint Passed

Figura 209: resultado en la vista

```
<p class="text-gray-900 leading-none">Checkpoint Passed</p>
```

Figura 210: código HTML generado

A qué hora y fecha fue cruzado:

```
<p class="text-gray-600">At: <%= checkpoint.passed_at %></p>
```

Figura 211: código para mostrar la fecha en la que el punto fue cruzado

At: 2020-11-21 07:31:47Z

Figura 212: resultado en la vista

```
<p class="text-gray-600">At: 2020-11-21 07:31:47Z</p>
```

Figura 213: código HTML generado

El id del agente que lo cruzó:

```
<span class="inline-block bg-gray-200 rounded-full px-3 py-1 text-sm font-semibold text-gray-700 mr-2 mb-2">
  Agent id: <%= checkpoint.agent_id %>
</span>
```

Figura 214: código para mostrar el id del agente que cruzó el punto

Agent id: 78

Figura 215: resultado en la vista

```
<span class="inline-block bg-gray-200 rounded-full px-3 py-1 text-sm font-semibold text-gray-700 mr-2 mb-2">Agent id: 78
</span>
```

Figura 216: código HTML generado

Por último, mostraremos si el agente que cruzó el punto era de la categoría maestro: Esta última parte tiene una lógica un poco más compleja, primero conseguimos del **checkpoint,** el id del agente y luego buscamos dentro del equipo, un agente que tenga esa id, finalmente, cogemos del agente la variable booleana **is_master** y la pintamos en el html:

```
<span class="inline-block bg-gray-200 rounded-full px-3 py-1 text-sm font-semibold text-gray-700 mr-2 mb-2">
    Master: <%= Map.get(Enum.find(
                @team.agents, %{}, fn agent -> checkpoint.agent_id == agent.id and checkpoint.team_id == agent.team_id end), :is_master, false) %>
</span>      You, seconds ago • Uncommitted changes
```

Figura 217: código para mostrar si el agente es maestro

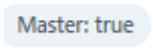


Figura 218: resultado en la vista

```
<span class="inline-block bg-gray-200 rounded-full px-3 py-1 text-sm font-semibold text-gray-700 mr-2 mb-2">
                Master: true</span>
```

Figura 219: código HTML generado

Finalmente, este es el resultado final de todo el código que hemos visto a partir de la figura 203:

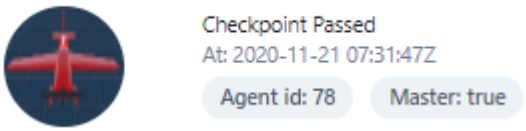


Figura 220: resultado final en la vista de todo el código anterior

Con esto, ya hemos terminado el código de nuestra vista, pero para que sea accesible desde el navegador en **http://localhost:4000/** debemos primero ejecutar el siguiente comando:

```
mix phx.server
```

Ahora ya tenemos nuestra vista en tiempo real terminada y accesible desde el navegador

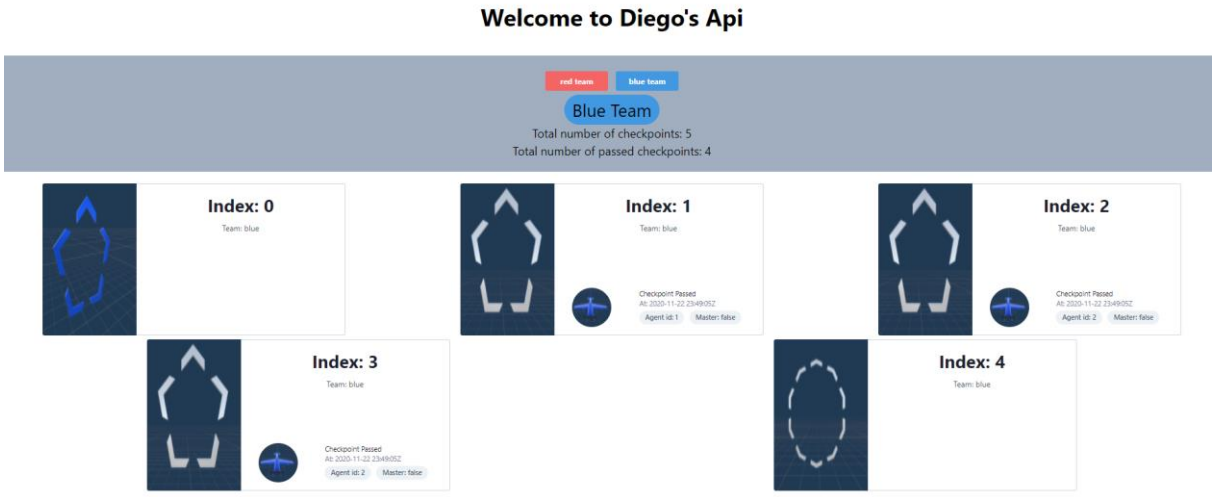


Figura 221: resultado final de nuestra vista

10. Conclusiones

Después del largo trayecto realizado en este proyecto, podemos sacar varias conclusiones y reflexiones.

La primera de ellas es la inmensa cantidad de recursos disponibles en línea para la resolución de cualquier problema, al fin y al cabo, aunque este trabajo lo realicé yo solo, no hubiera sido posible sin el trabajo previo de muchas personas como, por ejemplo, la de los creadores de ml-agents.

Segundo, ml-agents es un kit de herramientas prometedor y altamente configurable, que como hemos podido ver nos ofrece un amplio abanico de opciones con la que entrenar a nuestros agentes; y esto es algo que el motor de Unity permite de realizar de manera sencilla, no es necesario molestarse en simular la física ni el renderizado de imágenes, Unity lo hace todo por nosotros.

Tercero, en cuanto a los servicios REST, hemos podido observar una de las muchas posibles soluciones que existen hoy en día para crear un servidor REST. Spring Boot nos permitió de manera bastante sencilla leer e insertar datos en nuestra base de datos.

Cuarto, las LiveView de Phoenix nos ofrecieron una manera simple de visualizar los datos de nuestra base de datos mediante websockets, ocultando gran parte de la complejidad del proceso y evitando que tuviéramos que escribir javascript.

Leyendo el proyecto puede dar la impresión de que cada paso se realizó de manera rápida y sin problemas, pero nada más lejos de la realidad, hicieron faltas muchas horas leyendo documentación, viendo tutoriales, cursos, todo lo que fuera necesario para poder seguir avanzando con el proyecto.

En definitiva, me siento muy satisfecho con todo lo conseguido en este proyecto y me emociona saber que no he hecho más que rozar la superficie de todo lo que estas tecnologías son capaces de ofrecer. Creo que este proyecto puede servir como punto de partida para cualquiera que quiera trabajar con ml-agents, servicios REST y/o Phoenix.

11. Créditos

El proyecto de los aviones está basado en el curso de Immersive Limit: Reinforcement Learning: AI Flight with Unity ML-Agents

Algunas imágenes fueron obtenidas de Pixabay:

- Computadora y otros iconos
- Servidor

Agradecimientos a dilermv por su UnityRestClient en github y su vídeo explicativo del mismo

Tecnologías utilizadas:

- Unity + C# + kit de herramientas ml-agents
- Python + TensorFlow
- PostgreSQL
- Java + Spring Boot
- Elixir + Phoenix framework

12. Bibliografía

Shannon, C. E. (1950). XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *41*(314), 256-275.

Weber, B. (1997). Swift and slashing, computer topples Kasparov. *New York Times*, *12*, 262.

Juliani, A., Berges, V. P., Vckay, E., Gao, Y., Henry, H., Mattar, M., & Lange, D. (2018). Unity: A general platform for intelligent agents. arXiv preprint arXiv:1809.02627.

13. Anexo

13.1. Creación de la base de datos

Queremos crear una base de datos cuyo esquema será el siguiente:

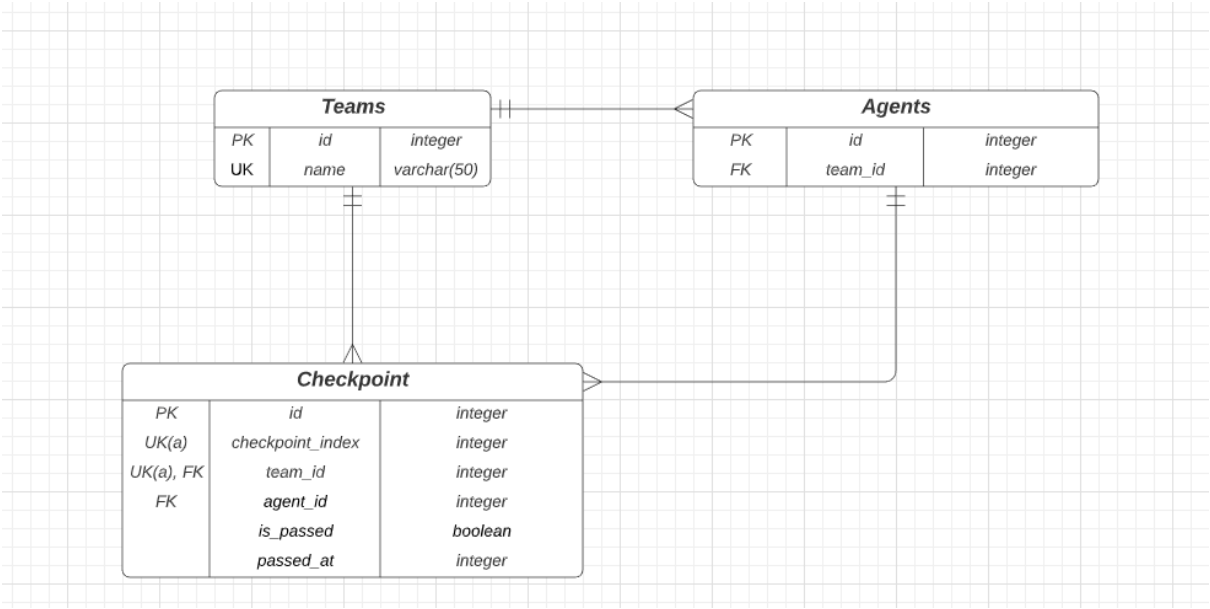


Figura 222: diagrama de la base de datos

Para la creación de la base de datos utilizaremos Elixir, debemos tener previamente postgreSQL[28] instalado en nuestro sistema.

El módulo Ecto.Migration[29] nos permitirá crear las tablas de nuestra base de datos.

Lo primero es crear un archivo de migración. Dentro de nuestro proyecto Elixir ejecutamos el comando:

```
mix ecto.gen.migration teams
```

Este archivo se puede encontrar en la carpeta **migrations**:

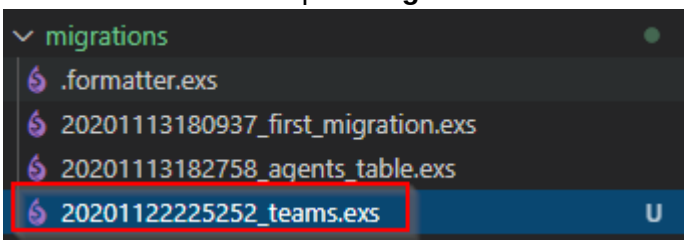


Figura 223:archivo de migración autogenerado por el comando ecto.gen.migration

En este archivo tenemos un módulo que utiliza **Ecto.Migration**

```
defmodule LiveApp.Repo.Migrations.Teams do
  use Ecto.Migration
```

Figura 224: módulo del archivo para la migración

[28] https://www.postgresql.org/
[29] https://hexdocs.pm/ecto_sql/Ecto.Migration.html

En nuestro caso, bastará con definir el método **change**

```
def change do

  create table(:teams) do
    add :name, :string, null: false, size: 30
  end

    create unique_index(:teams, :id)
end
```

Figura 225: método change para crear la tabla teams en la base de datos

`create table(:teams)` → creará una tabla con el nombre “teams”

*por defecto se creará una columna llamada “id”, de tipo “integer” y será una llave primaria.

`add :name, :string, null: false, size: 30` → añadirá una columna a la tabla con el nombre “name”, del tipo string que no puede tener valor nulo y su tamaño máximo es de 30 caracteres.

`create unique_index(:teams, :id)` → crearemos una restricción que hará que la id de cada entrada en la tabla deba ser única, como la id ya es por defecto una llave primaria, esto no es necesario, pero aquí se ha escrito para mostrar cómo se crearía esta restricción

Crearemos ahora otra migración donde se crearán las tablas **checkpoints** y **agents**

Tabla agents

```
create table(:agents) do
  add :team_id, references(:teams)
  add :is_master, :boolean, null: false, default: :false
end

create unique_index(:agents, [:team_id, :is_master])
```

Figura 226: migración para crear la tabla agents

Tabla checkpoints

```
create table(:checkpoints) do
  add :checkpoint_index, :integer, null: false
  add :is_passed, :boolean, null: false, default: :false
  add :passed_at, :utc_datetime
  add :team_id, references(:teams)
  add :agent_id, references(:agents)
end

create unique_index(:checkpoints, [:team_id, :checkpoint_index])
```

Figura 227: migración para crear la tabla checkpoints

Una vez creadas nuestras migraciones, debemos ejecutar 2 comandos:

`mix ecto.create`

`mix ecto.migrate`

Si todo va bien, veremos esto en nuestra consola:

```
The database for LiveApp.Repo has been created
```

Figura 228: resultado en consola del comando ecto.create

```
00:12:52.300 [info]  == Running 20201113180937 LiveApp.Repo.Migrations.FirstMigration.change/0 forward

00:12:52.303 [info]  create table teams

00:12:52.315 [info]  create index teams_id_index

00:12:52.321 [info]  == Migrated 20201113180937 in 0.0s

00:12:52.351 [info]  == Running 20201113182758 LiveApp.Repo.Migrations.AgentsTable.change/0 forward

00:12:52.351 [info]  create table agents

00:12:52.360 [info]  create index agents_team_id_is_master_index

00:12:52.363 [info]  create table checkpoints

00:12:52.368 [info]  create index checkpoints_team_id_checkpoint_index_index

00:12:52.370 [info]  == Migrated 20201113182758 in 0.0s
```

Figura 229: resultado en consola del comando mix ecto.migrate

Si por alguna razón fuese necesario borrar la base de datos y volverla a crear, el comando para ello es:

`mix ecto.drop`

Este comando nos permitirá borrar la base de datos.

Como paso opcional, si ya tenemos los modelos creados en el proyecto de elixir, podemos precargar algunos datos a nuestra base de datos, crearemos un archivo seed:

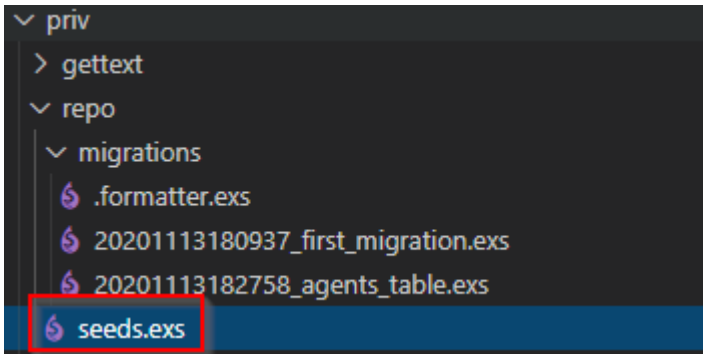


Figura 230: localización del archivo seeds.exs

Repo.insert! → nos sirve para insertar datos a la base de datos, debemos pasarle el módulo del modelo que queremos insertar y los datos de cada columna que queramos especificar.

Añadiremos 2 equipos:

```
# Teams
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Team{name: "red"})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Team{name: "blue"})
```

Figura 231: código para insertar dos equipos en la base de datos

Añadiremos 4 agentes, dos para cada equipo:

```
# Agents
# Red
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Agent{team_id: 1, is_master: true})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Agent{team_id: 1, is_master: false})
 #Blue
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Agent{team_id: 2, is_master: true})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Agent{team_id: 2, is_master: false})
```

Figura 232: código para insertar dos agentes en cada equipo en la base de datos

Y añadiremos 10 puntos, 5 para cada equipo:

```
# Checkpoints
# Red
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Checkpoint{checkpoint_index: 1, team_id: 1, is_passed: true,
 passed_at: DateTime.utc_now() |> DateTime.truncate(:second), agent_id: 1})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Checkpoint{checkpoint_index: 2, team_id: 1})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Checkpoint{checkpoint_index: 3, team_id: 1})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Checkpoint{checkpoint_index: 4, team_id: 1, is_passed: true,
 passed_at: DateTime.utc_now() |> DateTime.truncate(:second), agent_id: 2})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Checkpoint{checkpoint_index: 5, team_id: 1, is_passed: true})

#Blue
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Checkpoint{checkpoint_index: 1, team_id: 2})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Checkpoint{checkpoint_index: 2, team_id: 2, is_passed: true,
 passed_at: DateTime.utc_now() |> DateTime.truncate(:second), agent_id: 1})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Checkpoint{checkpoint_index: 3, team_id: 2, is_passed: true,
 passed_at: DateTime.utc_now() |> DateTime.truncate(:second), agent_id: 2})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Checkpoint{checkpoint_index: 4, team_id: 2, is_passed: true,
 passed_at: DateTime.utc_now() |> DateTime.truncate(:second), agent_id: 2})
LiveApp.Repo.insert!(%LiveApp.AircraftArea.Checkpoint{checkpoint_index: 5, team_id: 2, is_passed: true})
```

Figura 233: código para insertar 5 puntos a cada equipo en la base de datos

Y para ejecutar este archivo utilizamos el comando:
`mix run priv/repo/seeds.exs`