Nombre:
Daniel Rodríguez Ariza
George Picu
Grado: Grado en Diseño y Desarrollo de Videojuegos
Curso: 3º

DESARROLLO DE JUEGOS CON INTELIGENCIA ARTIFICIAL

PRÁCTICA 2
Aprendizaje Reforzado con Q-Learning

·ÍNDICE:

1. Introducción:

Memoria de la práctica

En este documento se detallan los pasos realizados para el desarrollo e implementación de la práctica 2, así como el contenido de la entrega.

El objetivo de la práctica es desarrollar un agente inteligente en Unity capaz de navegar un espacio, escapando de otro agente que navega el entorno haciendo uso de A*, persiguiéndolo para atraparlo.

1) Desarrollo de un agente inteligente capaz de navegar el entorno
2) Priorizar acciones que alejen al agente del enemigo
3) Hacer uso de aprendizaje por Q-Learning para entrenar al agente en estas tareas

Para el desarrollo de la práctica, se han creado clases especializadas para los diferentes algoritmos y tipos de datos necesarios para poder resolver de la manera más eficiente el problema.

El agente inteligente hará uso de técnicas de aprendizaje reforzado, específicamente Q-Learning, para ser entrenado y aprender a navegar un entorno mientras escapa de otro agente que navegará el entorno para atraparlo.

2. Contenidos de la entrega:

En esta entrega se han enviado los siguientes ficheros:

- MyQMindTrainer.cs:
  - Contiene la implementación de la clase **MyQMindTrainer**.
  - Permite entrenar al agente haciendo uso de Q-Learning.
  - Controla las iteraciones durante el entrenamiento.
  - Controla la ejecución de episodios y el control del aprendizaje.
  - Almacena la información del entrenamiento en una Tabla Q, dentro de un documento CSV.
  - Implementa la interfaz **IQMindTrainer**.

- MyQMindTester.cs
  - Contiene la implementación de la clase **MyQMindTester**.
  - Permite testear el agente entrenado haciendo uso de una Tabla Q obtenida a partir del entrenamiento.
  - Implementa la interfaz **INavigationAlgorithm**

- QTable.cs
  - Contiene la implementación de la clase **QTable**.
  - Implementación de Tabla Q para algoritmo de aprendizaje por refuerzo Q-Learning.
  - Hace uso de un diccionario para almacenar los diferentes estados posibles y los valores de Q obtenidos durante el aprendizaje.
  - Controla la lógica de inserción de estados y actualización de valores Q.
  - Controla la lógica de lectura y escritura de archivos en el disco para cargar y modificar la Tabla Q.

- ICSVConvertible.cs
  - Contiene la interfaz **ICSVConvertible**.
  - Sirve para implementar la funcionalidad común de todos los tipos de datos que serán convertidos en documentos CSV.
  - Contiene métodos para parsear y asignar valores a partir de un string CSV.
  - Contiene métodos para obtener un string CSV a partir del tipo de datos que implementa la interfaz.

- QTableData.cs
    - Contiene enums y structs con datos para los estados y acciones que puede realizar el agente. Esta información será utilizada para la construcción de la Q-Table.
    - Contiene el enum **QTableDistances**.
    - Contiene el enum **QTableAction**.
    - Contiene la implementación del struct **QTableState**.
    - Contiene la implementación del struct **QTableReward**.
    - El propósito de **QTableState** es almacenar la información correspondiente con cada uno de los estados de la Tabla Q.
    - El propósito de **QTableReward** es almacenar en un struct el vector de recompensas para cada uno de los estados y acciones de la Tabla Q.

- QTable.csv
    - Contiene la **Tabla Q** generada durante el entrenamiento.
    - Estos valores son utilizados para calcular la mejor acción para cada estado.

3. Escapar del oponente

El objetivo de la práctica es realizar la implementación de un agente inteligente capaz de huir de otro personaje, dando el mayor número de pasos posible sin ser atrapado.

Para ello, se ha realizado una implementación del algoritmo de aprendizaje por refuerzo Q-Learning, utilizando una implementación de Tabla Q. En esta tabla se simplifica el entorno del agente para limitar los estados a explorar.

Se ha dejado al agente agregar estados e información a la tabla mediante el algoritmo de aprendizaje Q-Learning durante las sesiones de entrenamiento, permitiendo que el agente aprenda la mejor acción a realizar en cada estado.

Dado su estado actual, el agente toma una decisión influenciada por un parámetro de exploración (épsilon, Є), permitiéndole explorar el entorno. En función del parámetro de exploración, el agente priorizará explorar más estados nuevos o reforzar los conocimientos adquiridos ejecutando las acciones que mejor resultados han generado durante el aprendizaje.

En función del resultado obtenido al ejecutar una acción determinada en un estado en concreto, se le otorga una recompensa positiva o negativa (se le recompensa o se le penaliza). Esta recompensa depende de si se cumple el objetivo o no.

Una vez el agente sea entrenado durante suficientes episodios, se podrá testear al agente inteligente en el entorno con el algoritmo de testing donde se utiliza lo aprendido anteriormente, aplicando los valores Q de la Tabla Q para obtener la acción que mejor resultado espere para cada estado.

4. Algoritmo de aprendizaje por refuerzo

Para la realización de la práctica, se ha empleado un algoritmo de aprendizaje por refuerzo para entrenar al agente inteligente, permitiéndole navegar por el escenario y escapar sin ser atrapado.

El funcionamiento de este algoritmo se divide en dos partes. La primera es la creación de un fichero vacío, que contendrá los estados. Posteriormente se ejecuta el algoritmo de aprendizaje. Durante la ejecución del algoritmo se le deja al agente en un punto concreto del mapa, y mira el estado actual en el que se encuentra. Este estado está determinado por unos parámetros que discretizan el mapa, y que se explicarán posteriormente con más detalle.

Para cada estado que se encuentre tiene cuatro posibles decisiones, el siguiente paso (norte, este, sur u oeste). Y toma una decisión en base al parámetro de exploración. En función de esa decisión recibe una penalización o una recompensa, y guarda este valor dentro de la tabla. Este proceso se repite para cada estado que encuentra, de esta manera se va completando la tabla a la vez que se ajustan los valores de salida.

Toda la información aprendida por el agente inteligente se mantiene de manera persistente en el documento creado en la primera ejecución. Este documento será utilizado para aplicar los conocimientos adquiridos a la hora de realizar su función dentro del entorno en el que se encuentre, además, gracias a la discretización el agente debe tener la capacidad de reaccionar en diferentes entornos.

5. Conversión de datos a CSV

Como posteriormente será necesario poder guardar los valores de la Tabla Q en un documento CSV y también cargarlos en memoria, se ha decidido crear la interfaz **ICSVConvertible**, que permite definir con facilidad una serie de métodos que contendrán todos los tipos de datos que deban ser convertidos a un string en formato CSV o que puedan cargar sus datos a partir de un string CSV.

La interfaz **ICSVConvertible** contiene los siguientes métodos:

ICSVConvertible.cs:

```
public interface ICSVConvertible
{
 public void CSVSetData(string csvLine, char[] csvSeparators, int offset =
0);
 public void CSVSetData(string[] csvElements, int offset = 0);
 public string CSVGetData(char separator = ';');
 public int CSVGetNumElements();
}
```

El método **CSVSetData()** permite cargar los datos del struct a partir un string en formato CSV. Se tomará el string y se parseará para extraer el valor de los diferentes valores almacenados en el documento CSV. El parámetro de entrada offset controla a partir de qué elemento del string CSV de entrada se comienza a parsear. Esta propiedad es utilizada al cargar las filas de datos del documento CSV.

El método **CSVGetData()** permite extraer los datos del struct y transformarlo en un string en formato CSV. El parámetro de entrada de este método permite especificar el tipo de símbolo utilizado para separar los elementos del CSV, por defecto “;” (basándose en los delimitadores utilizados más comúnmente en CSV, siendo estos “ “, “,” y “;”).

El método **CSVGetNumElements()** devuelve el número de miembros del struct o clase que implemente la interfaz. El propósito de este método es facilitar el proceso de parseo al hacer uso de la función **CSVSetData()**, permitiendo controlar el offset con mayor facilidad.

6. Determinación de los parámetros de estados

Para la creación de los posibles estados, se ha tenido en consideración diferentes posibles implementaciones.

Implementación 1:

Los estados son representados por una serie de parámetros que determinan en qué dirección puede desplazarse el agente y un valor numérico que determina el cuadrante en el que se encuentra el enemigo respecto de la posición del agente.

En esta implementación, se calcula el ángulo entre la posición del agente y la posición del enemigo.

```
Vector2 SelfPosition;
Vector2 OtherPosition;

Vector2 direction = OtherPosition - SelfPosition;

float angle = Mathf.Atan2(direction.y, direction.x) * (180.0f / Mathf.PI);
```

El ángulo obtenido con la función matemática **Mathf.Atan2()** da valores de ángulos en los intervalos [0,180] y [-180,0] al traducirlos de radianes a grados, ya que esta función matemática devuelve el ángulo más pequeño entre el ángulo de origen y el punto especificado.

Para poder calcular el cuadrante correctamente, es necesario obtener el ángulo positivo en los casos en los que se devuelva un valor negativo, por lo que se suma 360 grados al valor calculado:

```
angle = angle < 0 ? 360 + angle : angle;
```

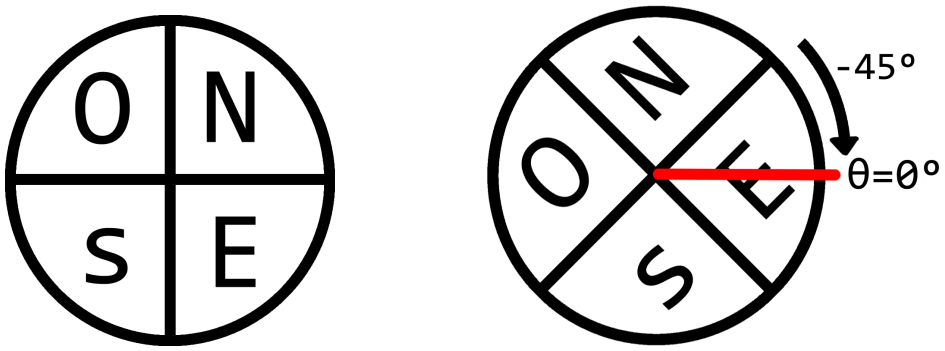
El ángulo obtenido puede ser utilizado para calcular el cuadrante donde se posiciona el enemigo, realizando la división entera del ángulo obtenido entre el número de cuadrantes que se quiera considerar.

Por ejemplo, para considerar 8 cuadrantes (Norte, Sur, Este, Oeste, Noreste, Noroeste, Sureste, Suroeste):

```
int quadrant = (int)(angle / numSegments);
```

El problema es que, actualmente, el ángulo está calculado respecto del eje X, por lo que todos los cuadrantes tienen una división que coincide con los ejes. Esto significa que es difícil detectar los desplazamientos del enemigo a lo largo de direcciones alineadas en la cuadrícula con el agente, lo cuál es especialmente notable en el proyecto actual, ya que los agentes únicamente pueden desplazarse en direcciones alineadas con los ejes y no en diagonales.

Para solucionar este problema, se puede transformar el ángulo obtenido rotando el sistema de tal manera que se alinee el centro de los cuadrantes Norte y Sur con el eje Y, y el centro de los cuadrantes Este y Oeste con el eje X.



Para conseguir esto, el giro necesario es de la mitad del ángulo de uno de los cuadrantes en sentido negativo, por lo que se puede calcular el ángulo corregido de la siguiente manera:

```
int numSegments = 8;
float segmentAngle = (360 / numSegments);
float halfSegmentAngle = (segmentAngle / 2.0f);
float angle = Mathf.Atan2(direction.y, direction.x) * (180.0f / Mathf.PI);
angle = angle * halfSegmentAngle;
angle = angle < 0 ? 360 + angle : angle;
```

Finalmente, se puede aplicar el mismo cálculo para obtener el cuadrante de forma correcta:

```
int quadrant = (int)(angle / numSegments);
```

Implementación 2:

Los estados son representados por una serie de parámetros que determinan en qué dirección puede desplazarse el agente y una serie de parámetros booleanos determinan si el enemigo se encuentra en cada una de las direcciones respecto de la posición del agente.

En esta implementación, sencillamente se calcula la dirección en la que se encuentra el enemigo respecto de la posición del agente.

```
Vector2 selfPosition;
Vector2 otherPosition;

if (otherPosition.y > selfPosition.y)
      isUp = true;

if (otherPosition.y < selfPosition.y)
      isDown = true;

if (otherPosition.x > selfPosition.x)
      isRight = true;

if (otherPosition.x < selfPosition.x)
      isLeft = true;
```

Al calcular si el otro agente se encuentra posicionado en una de las direcciones, se obtiene de forma automática combinaciones de valores booleanos donde 2 valores son verdaderos, permitiendo simular 8 cuadrantes con una lógica casi idéntica a la utilizada en caso de utilizar 4 cuadrantes, permitiendo obtener pesos para las diagonales con poco esfuerzo y sin necesidad de operar con ángulos.

Otro beneficio de esta implementación, además de su sencillez, es la velocidad a la hora de computar los resultados, siendo extremadamente rápida al evitar utilizar funciones trigonométricas. Esto no es un problema a día de hoy en computadoras modernas, pero es un aspecto a considerar al entrenar modelos de inteligencia artificial en entornos con recursos limitados.

Además, esta implementación, al hacer uso de múltiples variables para detectar si hay un enemigo en cada dirección, soporta de forma automática situaciones en las que haya más de un enemigo en el escenario, por lo que facilita la extensión del proyecto en un futuro.

Por las razones aquí expuestas, se ha decidido hacer uso de la segunda implementación.

7. Implementación de QTableState

Se ha creado un struct **QTableState** que permite almacenar la información relacionada con cada uno de los estados del entorno en el que se va a desplazar el agente.

Los estados que se han escogido para ser considerados son:

NorthIsWalkable : Determina si la casilla al norte de la actual es caminable.
EastIsWalkable : Determina si la casilla al este de la actual es caminable.
SouthIsWalkable : Determina si la casilla al sur de la actual es caminable.
WestIsWalkable : Determina si la casilla al oeste de la actual es caminable.

EnemyIsNorth : Determina si el enemigo se encuentra por el norte de la casilla actual.
EnemyIsEast : Determina si el enemigo se encuentra por el este de la casilla actual.
EnemyIsSouth : Determina si el enemigo se encuentra por el sur de la casilla actual.
EnemyIsWest : Determina si el enemigo se encuentra por el oeste de la casilla actual.

EnemyDistances : Contiene un valor discretizado para representar la distancia a la que se encuentra el enemigo.

Para almacenar los valores de distancias sin tener un número excesivo de estados, es necesario crear una discretización. En este caso, se ha decidido discretizar las distancias asignando 3 valores de un enum a diferentes rangos de distancias.

La definición del enum **QTableDistances** es la siguiente:

```
public enum QTableDistances
{
    Close = 0,
    Middle,
    Far
}
```

Close : Distancias inferiores a 5 unidades.
Mid : Distancias en el intervalo [5, 10).
Far : Distancias superiores o iguales a 10 unidades.

La definición del struct **QTableState** es la siguiente:

```
public struct QTableState : ICSVConvertible
{
    public bool NorthIsWalkable;
    public bool EastIsWalkable;
    public bool SouthIsWalkable;
    public bool WestIsWalkable;

    public bool EnemyIsNorth;
    public bool EnemyIsEast;
    public bool EnemyIsSouth;
    public bool EnemyIsWest;

    public QTableDistances EnemyDistance;

    // etc ...
}
```

Para permitir que el struct se pueda convertir con facilidad entre formato string CSV y formato struct en memoria, se implementa la interfaz **ICSVConvertible**.

La implementación del método **CSVGetData()** sencillamente hace uso de format strings de **C#** para generar un string en el que se encuentran las variables miembro enumeradas y separadas con el carácter separador.

```
public string CSVGetData(char separator = ';')
{
 return $"{NorthIsWalkable}{separator} {EastIsWalkable}{separator}
{SouthIsWalkable}{separator} {WestIsWalkable}{separator}
{EnemyIsNorth}{separator} {EnemyIsEast}{separator} {EnemyIsSouth}{separator}
{EnemyIsWest}{separator} {(int)EnemyDistance}{separator}";
}
```

La implementación del método **CSVSetData()** hace uso de un string segmentado en un array de strings, donde cada sting es el texto correspondiente a cada elemento. Se hace uso de las funciones estándar Parse() para obtener los valores de los tipos de datos correspondientes.

```
public void CSVSetData(string[] dataStrings, int offset = 0)
{
 this.NorthIsWalkable = bool.Parse(dataStrings[offset + 0]);
 this.EastIsWalkable  = bool.Parse(dataStrings[offset + 1]);
 this.SouthIsWalkable = bool.Parse(dataStrings[offset + 2]);
 this.WestIsWalkable  = bool.Parse(dataStrings[offset + 3]);
 this.EnemyIsNorth    = bool.Parse(dataStrings[offset + 4]);
 this.EnemyIsEast     = bool.Parse(dataStrings[offset + 5]);
 this.EnemyIsSouth    = bool.Parse(dataStrings[offset + 6]);
 this.EnemyIsWest     = bool.Parse(dataStrings[offset + 7]);
 this.EnemyDistance   = (QTableDistances)int.Parse(dataStrings[offset + 8]);
}
```

8. Implementación de QTableReward

Se ha creado un struct **QTableReward** que permite almacenar los valores de Q relacionados con cada uno de los estados y acciones posibles que puede realizar el agente.

Las acciones que el agente puede realizar son:

GoNorth : Desplaza al agente en dirección norte.
GoEast : Desplaza al agente en dirección este.
GoSouth : Desplaza al agente en dirección sur.
GoWest : Desplaza al agente en dirección oeste.

Para facilitar la representación de estas acciones, se ha creado un enum QTableAction que asigna un valor numérico a cada acción:

```
public enum QTableAction
{
    GoNorth = 0,
    GoEast,
    GoSouth,
    GoWest
}
```

Entonces, el struct **QTableReward** queda definido como un vector de 4 floats para almacenar los valores de Q correspondientes a cada acción dentro de un estado específico.

```
public struct QTableReward : ICSVConvertible
{
    public float rewardNorth;
    public float rewardEast;
    public float rewardSouth;
    public float rewardWest;
    // etc ...
}
```

Al igual que el struct **QTableState**, se ha implementado la interfaz **ICSVConverticle** para facilitar el proceso de conversión entre formato textual CSV y datos en memoria.

9. Implementación de QTable

Para realizar la implementación de la Tabla Q, se ha creado una clase que contiene toda la lógica relacionada al control de la tabla, la manipulación y actualización de valores Q y la carga y el almacenamiento de documentos CSV en memoria.

Se ha creado la clase **QTable**, que implementa el comportamiento necesario para controlar la Tabla Q.

La Tabla Q está implementada como un diccionario de estados y valores Q. Cada entrada está compuesta por una llave, que es el estado, y un valor correspondiente al estado, que es el vector de valores Q correspondientes a cada acción dentro de dicho estado.

```
private Dictionary<QTableState, QTableReward> qTable;
```

El diccionario se inicializa como un diccionario vacío, y los estados se van insertando en la tabla según se explora el entorno.

La clase **QTable** contiene métodos para acceder y manipular los valores de Q almacenados en la tabla.

Estos métodos son:

· **GetQ()** : Devuelve el valor de Q almacenado en la tabla para el estado **state** y la acción **action**.
· **GetMaxQ()** : Devuelve el valor de Q más alto que hay almacenado en el estado **state**.
· **SetQ()** : Asigna el valor **value** al valor Q almacenado para el estado **state** y la acción **action**.
· **UpdateQ()** : Actualiza el valor de Q haciendo uso de la **regla de aprendizaje**.

La regla de aprendizaje es la siguiente:

$$Q(s, a)' = (1 - alpha) \cdot Q(s, a) + alpha \cdot (r + gamma \cdot maxQ(s', a'))$$

Como la regla de aprendizaje es una combinación de una proporción de alpha multiplicado por el valor de Q del estado actual sumado a alpha por el valor obtenido por el aprendizaje, se puede comprender la regla de aprendizaje como una **interpolación lineal** entre el valor de Q actual y el valor de recompensa obtenido:

$$Q(s, a)' = lerp(Q(s, a), r + gamma \cdot maxQ(s', a'), alpha)$$

Entonces, la función de actualización queda definida como:

```
public void UpdateQ(QTableState state, QTableState nextState, QTableAction action,
float reward, float alpha, float gamma)
{
 float newQ = Mathf.Lerp(GetQ(state, action), reward + gamma * GetMaxQ(nextState), alpha);
 SetQ(state, action, newQ);
}
```

Además de manipular y almacenar valores Q, la clase **QTable** se encarga de controlar el proceso de guardado y cargado de archivos.

Para almacenar los datos en el disco, se hace uso del método público **SaveTable(),** que internamente hace uso del método **WriteFile()**.

Este método hace uso de la funcionalidad implementada en los structs que implementan la interfaz **ICSVConvertible** y la funcionalidad de StreamWriter para almacenar en un documento de texto la representación CSV de la tabla.

Se iteran todos los estados almacenados en el diccionario **qTable** y se extraen los strings en formato CSV de los estados y los valores de Q correspondientes.

```
private void WriteFile()
{
  string filename = GetFileName();
  StreamWriter writer = new StreamWriter(filename, false);
  foreach (var entry in qTable)
  {
    string stateString = entry.Key.CSVGetData();
    string valueString = entry.Value.CSVGetData();
    writer.WriteLine($"{stateString} {valueString}");
  }
  writer.Close();
}
```

De esta manera, cada vez que sea necesario, se puede invocar el método, almacenando los datos de la Tabla Q que hay en memoria dentro de un fichero llamado **QTable.csv**.

En el caso de este proyecto, se ejecuta cada vez que se completa una serie de episodios (valor especificado en el controlador **QMindTrainer** del mapa de entrenamiento **TrainPlayGround.unity)**.

Para cargar los datos que hay almacenados en el fichero **QTable.csv**, se hace uso del método público **LoadTable(),** que internamente hace uso del método **ReadFile()**.

Se hace uso de **StreamReader** para cargar el archivo y leer todas las líneas del documento, obteniendo los valores almacenados en el CSV para cada uno de los estados y valores de Q.

Se iteran todas las líneas de texto almacenadas en el documento de texto y se subdividen los strings en substrings cada vez que se encuentra uno de los caracteres separadores CSV. Después, se hace uso de los métodos **CSVSetData()** y se almacena la información dentro del diccionario.

```
private void ReadFile()
{
  string filename = GetFileName();
  StreamReader reader = new StreamReader(filename);
  string csvLine;
  while ((csvLine = reader.ReadLine()) != null)
  {
    string[] dataStrings = csvLine.Split(csvSeparators);
    QTableState state = new QTableState();
    state.CSVSetData(dataStrings, 0);
    QTableReward reward = new QTableReward();
    reward.CSVSetData(dataStrings, state.CSVGetNumElements());
    qTable.Add(state, reward);
  }
  reader.Close();
}
```

De esta manera, se puede cargar la Tabla Q en cualquier momento que se desee.

En el caso de este proyecto, se carga cada vez que se inicia una simulación.

10. Implementación de MyQMindTrainer

Para realizar la implementación del algoritmo de entrenamiento con aprendizaje por refuerzo con Q-Learning se ha creado una clase **MyQMindTrainer**, cuyo propósito es entrenar al agente controlando los episodios y los parámetros del mundo durante el entrenamiento.

Para ello, tiene acceso a una serie de parámetros que le permiten controlar la información del entorno simulado y los episodios e iteraciones por episodio:

```
private WorldInfo worldInfo;
private int currentEpisode;
private int currentStep;
```

Esta clase también está a cargo de controlar las recompensas y penalizaciones que se le den al agente durante el entrenamiento, controlando la actualización de los valores de Q de la Tabla Q.

Para ello, tiene acceso a una serie de parámetros de tipo float que le permiten almacenar los valores de recompensa que se administrarán en cada estado:

```
private const float smallPenaltyScore = -10.0f;
private const float largePenaltyScore = -100.0f;
private const float rewardScore = 100.0f;
private const float neutralScore = 0.0f;
```

También contiene un miembro de tipo **QMindTrainerParams** que le permite almacenar los parámetros de la simulación, permitiendo de esta manera configurar el proceso de entrenamiento:

```
QMindTrainerParams qMindTrainerParams;
```

La clase **MyQMindTrainer** contiene un miembro privado **qTable**, que es del tipo **QTable**.

```
private QTable qTable;
```

Esta variable sirve para almacenar la Tabla Q, y es donde se almacenarán los valores de Q obtenidos durante el entrenamiento. Esta variable también permite realizar la carga y el almacenamiento de datos en un fichero.

En el método de inicialización **Initialize()**, se inicializa el valor de las variables, además de cargar la Tabla Q en caso de que exista un documento.

```
public void Initialize(QMindTrainerParams qMindTrainerParams, WorldInfo
worldInfo, INavigationAlgorithm navigationAlgorithm)
{
	this.qMindTrainerParams = qMindTrainerParams;
	this.worldInfo = worldInfo;
	this.navigationAlgorithm = navigationAlgorithm;
	this.navigationAlgorithm.Initialize(this.worldInfo);
	this.qTable = new QTable();
	LoadQTable();
	Debug.Log("MyQMindTrainer: Initialized");
	StartEpisode(0);
}
```

Se llama al método **StartEpisode()**, que inicia un episodio con el índice especificado. Este método se encarga de ejecutar el proceso de inicialización de cada uno de los episodios de la simulación.

Se cambia la posición del agente y del enemigo y se posicionan en casillas aleatorias del entorno.

```
private void StartEpisode(int episodeIdx)
{
	currentEpisode = episodeIdx;
	AgentPosition = worldInfo.RandomCell();
	OtherPosition = worldInfo.RandomCell();
	OnEpisodeStarted?.Invoke(this, EventArgs.Empty);
	this.currentStep = 0;
	SaveQTable();
}
```

El método **SaveQTable()** es llamado siempre que se inicia un episodio, pero éste solo permite guardar la Tabla Q siguiendo el parámetro de configuración **EpisodesBetweenSaves**. De esta manera, se evita ralentizar la simulación, evitando acceder al disco de forma innecesaria.

```
private void SaveQTable()
{
   if (currentEpisode > 0 && currentEpisode % episodesBetweenSaves == 0)
   {
      qTable.SaveTable();
   }
}
```

El método DoStep() permite al entrenador ejecutar un paso de la simulación. En este método, se actualiza la información sobre el estado actual, y se da un paso en base a esta información, desplazando la posición del agente y del enemigo.

Una vez se termina de dar el paso, se actualiza la información almacenada en la Tabla Q en base a las recompensas obtenidas en el estado actual al ejecutar cierta acción.

```
public void DoStep(bool train)
{
  QTableState state = GetState();
  QTableAction action = GetAction(state);
  float reward = GetReward(state, action);

  UpdateDisplayRewardValues(reward);

  if(train)
    UpdateQTable(state, action, reward);

  if (((qMindTrainerParams.maxSteps >= 0) && ((currentStep + 1) >
qMindTrainerParams.maxSteps)) || reward < (smallPenaltyScore - 0.01))
  {
    OnEpisodeFinished?.Invoke(this, EventArgs.Empty);
    NextEpisode();
  }

  MovePlayer();
  MoveAgent(action);

  ++this.currentStep;
}
```

En esta función se ejecutan varios métodos que son clave para la correcta actualización de los valores de la Tabla Q y el proceso de entrenamiento del agente inteligente:

· **GetState()** : Obtiene el estado actual. Utiliza a **worldInfo** y transforma los datos en un valor de tipo **QTableState**.

· **GetNextState()** : Obtiene el siguiente estado en el que se encontrará el agente en base al estado actual y una acción a realizar.

· **GetAction()** : Obtiene la acción a realizar en el estado actual.

· **GetReward()** : Obtiene la recompensa correspondiente al estado actual y la acción seleccionada.

· **UpdateQTable()** : Actualiza el valor de la Tabla Q a partir de esta información.

En el método **GetState()**, se obtiene el estado comprobando si las casillas adyacentes son caminables o no, obteniendo si el otro agente se encuentra en cada una de las direcciones, y calculando la discretización de la distancia Manhattan entre ambos agentes.

```
private QTableState GetState(CellInfo cell)
{
      QTableState state = new QTableState(
            GetIsWalkable(cell, Directions.Up),
            GetIsWalkable(cell, Directions.Right),
            GetIsWalkable(cell, Directions.Down),
            GetIsWalkable(cell, Directions.Left),
            GetOtherIsInDirection(cell, Directions.Up),
            GetOtherIsInDirection(cell, Directions.Right),
            GetOtherIsInDirection(cell, Directions.Down),
            GetOtherIsInDirection(cell, Directions.Left),
            GetOtherDistance(cell)
      );
      return state;
}
```

En el método **GetNextState()**, se hace uso del método GetState() para obtener el estado en el que se encontrará el agente al desplazarse en cada una de las direcciones posibles.

```
private QTableState GetNextState(QTableAction action)
{
 switch (action)
 {
  case QTableAction.GoNorth:
   return GetState(worldInfo.NextCell(AgentPosition, Directions.Up));
  case QTableAction.GoEast:
   return GetState(worldInfo.NextCell(AgentPosition, Directions.Right));
  case QTableAction.GoSouth:
   return GetState(worldInfo.NextCell(AgentPosition, Directions.Down));
  case QTableAction.GoWest:
   return GetState(worldInfo.NextCell(AgentPosition, Directions.Left));
  default:
   return GetState(AgentPosition);
 }
}
```

En el método **GetAction(),** se obtiene la acción a realizar en el estado actual.

Durante el entrenamiento, se quiere promover que el agente explore nuevas opciones, además de reforzar su conocimiento sobre las opciones ya exploradas.

Es por esta razón que se hace uso del parámetro épsilon (grado de exploración) para determinar si el agente ha de realizar una acción aleatoria (explorar) o si ha de reforzar su conocimiento tomando la mejor decisión posible.

```
private QTableAction GetAction(QTableState state)
{
      float n = UnityEngine.Random.Range(0.0f, 1.0f);

      if (n <= qMindTrainerParams.epsilon)
            return GetRandomAction();

      return GetBestAction(state);
}
```

La recompensa obtenida es calculada por el método **GetReward()**, el cuál hace uso del estado actual y la acción que se ha tomado en dicho estado para determinar la calidad del estado futuro en el que se encontrará el agente inteligente.

De esta manera, se puede calcular la recompensa obtenida o la penalización obtenida, dependiendo de cómo de mejor o peor sea el estado que alcance el agente al tomar una decisión determinada.

Con este sistema de recompensas, el agente priorizará mantenerse lo más lejos posible del enemigo y, en caso de no poder lograrlo, intentará evitar ser capturado caminando a su alrededor, pasando por la acción más eficiente que evite ser capturado.

```
private float GetReward(QTableState state, QTableAction action)
{
	bool cannotWalk =
		(!state.NorthIsWalkable && action == QTableAction.GoNorth) ||
		(!state.EastIsWalkable && action == QTableAction.GoEast) ||
		(!state.SouthIsWalkable && action == QTableAction.GoSouth) ||
		(!state.WestIsWalkable && action == QTableAction.GoWest)
		;
	bool caught = OtherPosition.Distance(AgentPosition,
CellInfo.DistanceType.Manhattan) <= 1;
	bool walkingTowardEnemy =
		state.EnemyDistance == QTableDistances.Close &&
		(
			(state.EnemyIsNorth && action == QTableAction.GoNorth) ||
			(state.EnemyIsEast && action == QTableAction.GoEast) ||
			(state.EnemyIsSouth && action == QTableAction.GoSouth) ||
			(state.EnemyIsWest && action == QTableAction.GoWest)
		)
	;
	if (cannotWalk || caught)
		return largePenaltyScore;
	if (walkingTowardEnemy)
		return smallPenaltyScore;
	QTableState nextState = GetNextState(action);
	if (nextState.EnemyDistance > state.EnemyDistance &&
nextState.EnemyDistance == QTableDistances.Far)
		return rewardScore;
	return neutralScore;
}
```

El método **UpdateQTable()** se encarga de actualizar los valores de Q a partir del estado actual, el siguiente estado, la recompensa obtenida, y los valores de los parámetros **alpha** (learning rate), **épsilon** (exploration rate) y gamma (discount factor).

Este proceso lo realiza llamando al método **QTable.UpdateQ()** y haciendo uso de los métodos y parámetros de configuración contenidos en la clase **MyQMindTrainer**, permitiendo que el proceso de entrenamiento sea configurable.

```
private void UpdateQTable(QTableState state, QTableAction action, float
reward)
{
      qTable.UpdateQ(
      state,
      GetNextState(action),
      action,
      reward,
      qMindTrainerParams.alpha,
      qMindTrainerParams.gamma
      );
}
```

11. Implementación de MyQMindTester

La clase **MyQMindTester** se encarga de controlar el entorno de simulación para el testeo del modelo entrenado por medio de Q-Learning.

Esta clase es muy simple y únicamente contiene métodos para facilitar la inicialización y el proceso de toma de decisión.

Al inicializar, la clase **MyQMindTester** almacena una referencia a la información del entorno de simulación y se carga el documento **QTable.csv**, que contiene la Tabla Q obtenida durante el proceso de entrenamiento.

```
public void Initialize(WorldInfo worldInfo)
{
	this.worldInfo = worldInfo;
	this.qTable = new QTable();

	qTable.LoadTable();

	Debug.Log("MyQMindTester: Initialized");
}
```

En el método **GetNextStep()**, se obtiene el estado actual y se decide con la Tabla Q cuál es la mejor acción a realizar en el estado actual.

Posteriormente, se calcula la nueva celda en la que se encontrará el agente y se devuelve el valor.

```
public CellInfo GetNextStep(CellInfo currentPosition, CellInfo otherPosition)
{
	Debug.Log("MyQMindTester: GetNextStep");
	QTableState currentState = GetState(currentPosition, otherPosition);
	QTableAction bestAction = qTable.GetBestAction(currentState);
	CellInfo ans = MoveAgent(currentPosition, bestAction);
	return ans;
}
```

De esta manera, el agente hace uso de la información del estado del entorno de simulación y los valores de la Tabla de Q obtenidos durante el entrenamiento para tomar las mejores decisiones posibles y escapar del agente A* de forma que logra evitar ser capturado.

12. Selección de Recompensas

Para el entrenamiento del agente inteligente por medio de Q-Learning, es necesario escoger una serie de parámetros para determinar las recompensas que serán dadas por cada acción realizada por el agente durante el proceso de entrenamiento.

Se ha decidido que el agente podrá recibir tanto recompensas positivas como negativas, con el objetivo de reforzar el aprendizaje de las acciones que ofrezcan la mayor recompensa posible.

Se han diseñado los siguientes tipos de recompensas:

Positive Reward : 100
Neutral Reward : 0
Small Penalty : -10
Large Penalty : -100

Para que el proceso de entrenamiento de buenos resultados, es de gran importancia que las recompensas sean coherentes y no se den pequeñas recompensas por acciones que no son necesariamente erróneas.

El objetivo del agente inteligente es aprender a escapar de un enemigo que le persigue haciendo uso de A*. Es por eso que se ha decidido que el objetivo a cumplir para obtener la recompensa positiva será alcanzar la distancia máxima posible entre el agente y el enemigo. Esta recompensa solo se le otorgará cada vez que el agente logre pasar de una distancia inferior a la distancia máxima de los 3 niveles de la discretización de la distancia explicada previamente (Close, Mid, Far). Es decir, el agente es recompensado cada vez que pasa desde Close o Mid hasta Far.

De esta manera, se refuerza el aprendizaje de caminos por los que se maximice la distancia entre el agente y el enemigo, priorizando así la supervivencia.

El agente será penalizado cada vez que realice una acción que le ponga en una peor situación que la anterior, y únicamente será recompensado si logra llegar a una posición en la que cumpla el objetivo, que es mantenerse lo más lejos posible del enemigo.

Es por eso que se ha decidido crear 2 tipos de penalizaciones: una penalización leve otorgada cada vez que se pase a un estado peor (la distancia entre el agente y el enemigo disminuye) y una recompensa grave que es otorgada cada vez que el agente fracasa (colisiona con una pared o entra en contacto con el enemigo).

Cada vez que se otorga la recompensa grave, la simulación finaliza el episodio actual.

13. Proceso de Entrenamiento

Para el entrenamiento de la inteligencia artificial del agente inteligente por medio de Q-Learning, se han realizado múltiples simulaciones con miles de episodios para garantizar que el agente aprenda la mayor cantidad de información posible.

Para maximizar la información aprendida, se han dividido las sesiones de entrenamiento en 2 categorías:

Sesiones de Exploración:
Se ha entrenado al agente haciendo uso de valores épsilon grandes, favoreciendo la exploración aleatoria del entorno, lo que ha permitido que el agente aprenda inicialmente la mayor cantidad de información posible.

Sesiones de Refuerzo:
Se ha entrenado al agente haciendo uso de valores épsilon pequeños, favoreciendo el refuerzo de la información aprendida. El agente prioriza navegar por los caminos que tengan mayor valor de Q, reforzando los conocimientos adquiridos en las sesiones de entrenamiento anteriores donde se favoreció la exploración del entorno.

Para garantizar que el agente pueda aprender durante las sesiones de entrenamiento la mayor cantidad de información y casos posibles, se han diseñado diferentes niveles de entrenamiento y de prueba. Los niveles de entrenamiento han sido diferentes a los diseñados para testear el agente. De esta manera, se ha podido evitar caer en el error del overfitting.

Se han ejecutado sesiones de entrenamiento de 10.000 episodios haciendo uso de **épsilon = 0.85**, seguidas de 5.000 episodios con **épsilon = 0.3** en cada uno de las escenas de entrenamiento diseñadas.

Los resultados obtenidos son satisfactorios, ya que el agente ha aprendido a escapar de forma exitosa del enemigo, siendo capaz de sobrevivir de forma indefinida en los 12 mapas de testeo diseñados, superando más de 5.000 pasos en cada simulación de prueba.