UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO

Diseño de un motor de videojuegos 3D para el estudio de avatares virtuales inteligentes

Realizado por
Jesús López Pujazón

Para la obtención del título de
Doble Grado en Ingeniería Informática y Matemáticas

Dirigido por
Pablo García Sánchez
Carlos Ureña Almagro

Realizado en el departamento de
ATC y LSI

Convocatoria de Junio, curso 2021/22

Agradecimientos

Quiero agradecer a X por...

También quiero agradecer a Y por...

Resumen

A lo largo de la primera década del siglo XXI surgieron diversas videoconsolas que normalizarían la presencia de estas en cualquier hogar. Desde entonces, la industria del videojuego ha crecido de manera casi exponencial en importancia hasta alcanzar una increíble fama e influencia en todo el mundo. Uno de los aspectos más importantes en la mayoría de videojuegos se trata de la Inteligencia Artificial que poseen los NPC (Non Playable Characters). En ocasiones, juegos cuyos conceptos podrían tener un gran potencial, se ven eclipsados por la presencia de enemigos cuya inteligencia les obliga a perseguir de forma patosa al personaje que controlamos. En este trabajo, modelaré algunas estructuras y personajes mediante la técnica de *Voxel Art* y aplicaré animaciones a los personajes en una web definiendo sus articulaciones principales para dar lugar a un escenario de un videojuego. Asimismo, aplicaré técnicas de aprendizaje por refuerzo a una inteligencia artificial para comprobar si se podrían aplicar estas técnicas en la industria para dar lugar a una experiencia de juego más rica. El objetivo de este trabajo será, por tanto, estudiar la eficiencia de algoritmos de aprendizaje por refuerzo en un caso real para comprobar su eficacia.

Palabras clave: Inteligencia Artificial, NPC, Voxel Art, Aprendizaje por refuerzo, Modelar.

Abstract

The first decade of the 21st century saw the emergence of various video game consoles that would normalise their presence in every home. Since then, the video game industry has grown almost exponentially in importance to achieve incredible fame and influence around the world. One of the most important aspects of most video games is the Artificial Intelligence that NPCs (Non Playable Characters) possess. Sometimes, games whose concepts could have great potential, are overshadowed by the presence of enemies whose intelligence forces them to chase after the character we control in a clumsy way. In this paper, I will model some structures and characters using the technique of *Voxel Art* and I will apply animations to the characters in a web defining their main articulations to create a video game scenario. I will also apply reinforcement learning techniques to an artificial intelligence to see if these techniques could be applied in the industry to create a richer gaming experience. The aim of this work will therefore be to study the efficiency of reinforcement learning algorithms in a real case to test their effectiveness.

Keywords: Artificial Intelligence, NPC, Voxel Art, Reinforcement Learning, Model.

Índice general

Índice de figuras

Índice de extractos de código

1. Introducción

2. Objetivos

2.1. Introducción

Breve introducción al capítulo

2.2. Objetivos

2.3. Metodología

2.4. Planificación

2.5. Conclusiones

3. Análisis del problema

3.1. Introducción

En este capítulo explicaremos...

3.2. Herramientas

Para el desarrollo del proyecto ha sido necesario el uso de distintas herramientas. Para el modelado de las estructuras y los personajes he utilizado el software gratuito **MagicaVoxel** [2], que permite modelar mediante el empleo de voxels, así como renderizar los objetos creados y aplicarles distintas texturas. Para crear las animaciones de los personajes he empleado la página **Mixamo** [1], que permite generar distintas animaciones predeterminadas para los esqueletos que creamos y, posteriormente, exportarlas de manera eficiente para poder utilizarlas en otros software como Unity, Blender o Unreal Engine. Para diseñar la escena y poder implementar el videojuego junto con el algoritmo de Aprendizaje por Refuerzo he empleado **Unity** [3], que permite la creación de escenas y videojuegos tanto 2D como 3D y posee diversos tutoriales y ayudas para sus usuarios.

Figura 3.1: Herramientas empleadas

3.3. Objetivos

3.4. Conclusiones

En este capítulo concluimos que...

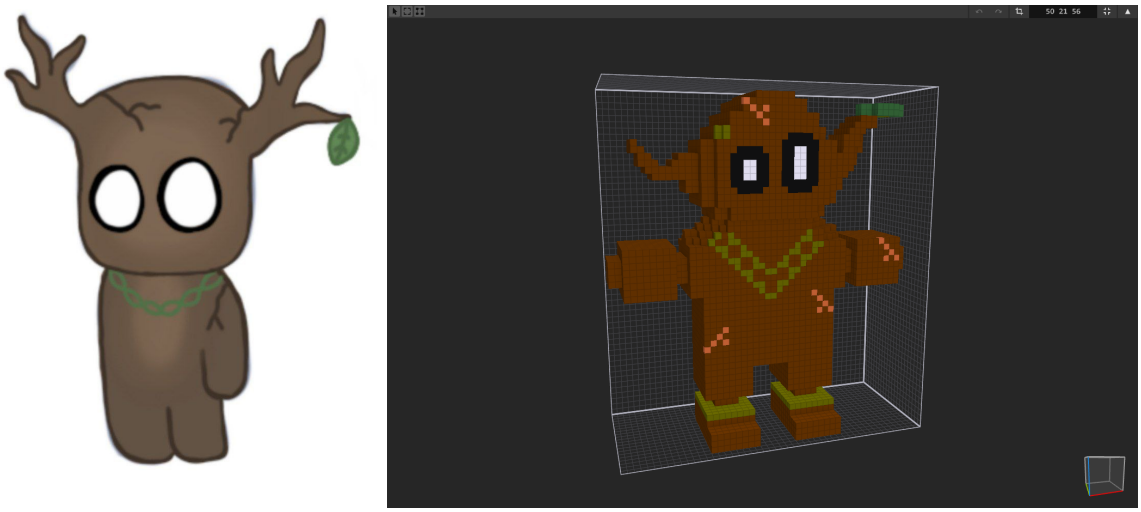

Figura 3.2: Boceto y modelo del enemigo básico

4. Implementación

4.1. Introducción

En este capítulo explicaré el procedimiento que he seguido para el modelado, tanto de las estructuras como los avatares que componen la escena principal, la creación de dicha escena a partir de todas esas componentes y la implementación del algoritmo de Aprendizaje por Refuerzo que utilizará la IA para aprender a atacar la fortaleza principal y maximizar el daño que esta reciba.

4.2. Modelado

Para el desarrollo de un primer escenario sencillo será necesario, en primer lugar, modelar tanto las tres estructuras básicas que compondrán la escena y un enemigo básico. Para el modelado emplearé, como se ha comentado en el apartado anterior, el programa *MagicaVoxel*.

En primer lugar comenzaré modelando el enemigo básico que utilizaré durante todo el desarrollo del problema. Para ello emplearé una cuadrícula de tamaño 50x21x56. En este caso utilizaré una cuadrícula más grande que la que usaré para modelar los tres edificios principales ya que será necesario definir bien las partes del enemigo donde se encontrarán sus articulaciones (las muñecas, los codos y las rodillas). Para el diseño, me he basado en un boceto de un personaje que realicé hace tiempo y le he añadido algunas modificaciones. La prinncipal modificación necesaria es eliminar voxels en las zonas donde se encuentran las articulaciones para poder detectarlas correctamente en *Mixamo*. El modelo del personaje obtenido es el siguiente:

Figura 4.1: Enemigo básico: Modelo renderizado

A continuación, crearé una animación de movimiento para este enemigo utilizando la web *Mixamo*. Para ello, incio sesión en la web y en la página principal encontramos

un modelo de ejemplo para generar animaciones. Para poder generar la animación, cargo el modelo a partir de un fichero zip que contendrá el modelo del enemigo, las texturas y los materiales. Una vez cargado el modelo, es necesario indicar las posiciones de las articulaciones para poder generar la animación correctamente. A continuación se muestra el modelo con la barbilla, codos, muñecas, rodillas y cintura marcadas. Están representadas mediante un círculo de color azul, amarillo, verde, naranja y rosa respectivamente.

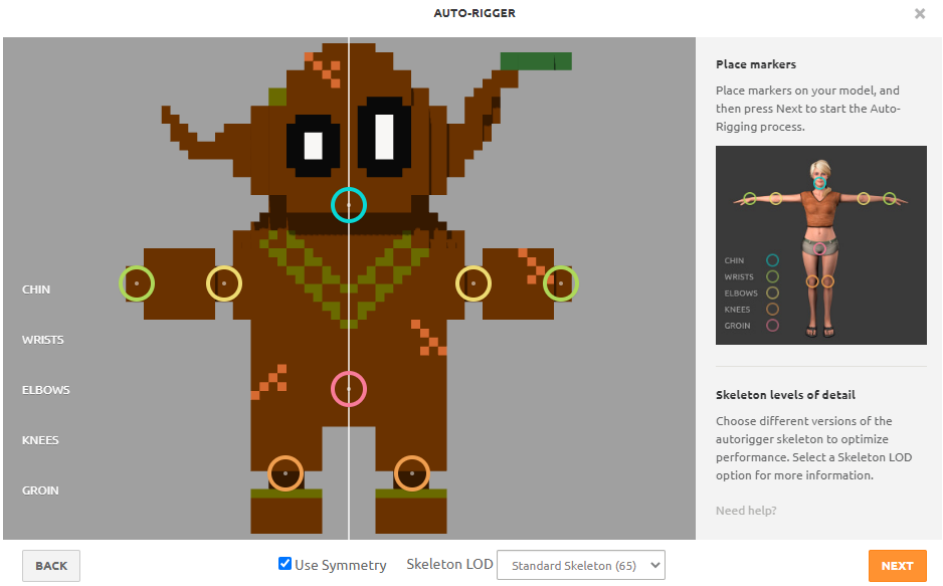

Figura 4.2: Definición del esqueleto del modelo

Para finalizar, es necesario escoger una animación entre las que ofrece la web, en este caso he escogido la animación *Mutant Walking*. Una vez finalizado, necesitaré modelar las tres estructuras básicas que compondrán el escenario del juego. Estas estructuras son: la torreta, el almacén de oro y la fortaleza. Para modelar estas tres usaré una cuadrícula del mismo tamaño, en este caso una cuadrícula de tamaño 40x40x40. Para el diseño, esta vez me he inspirado en las estructuras del videojuego *Clash of Clans*. Los modelos renderizados de las estructuras pueden observarse a continuación.

Figura 4.3: Estructuras: Modelos renderizados

Para el modelado de la torreta, decido separar la parte superior de la inferior para tener dos modelos diferenciados. De este modo la parte superior podrá girar cuando se desarrolle el videojuego sin necesidad de girar toda la estructura.

4.3. Diseño de la escena

Para el desarrollo restante del proyecto usaré la herramienta *Unity*. En primer lugar, Unity permite crear distintos tipos de proyectos los cuales se muestran a continuación:

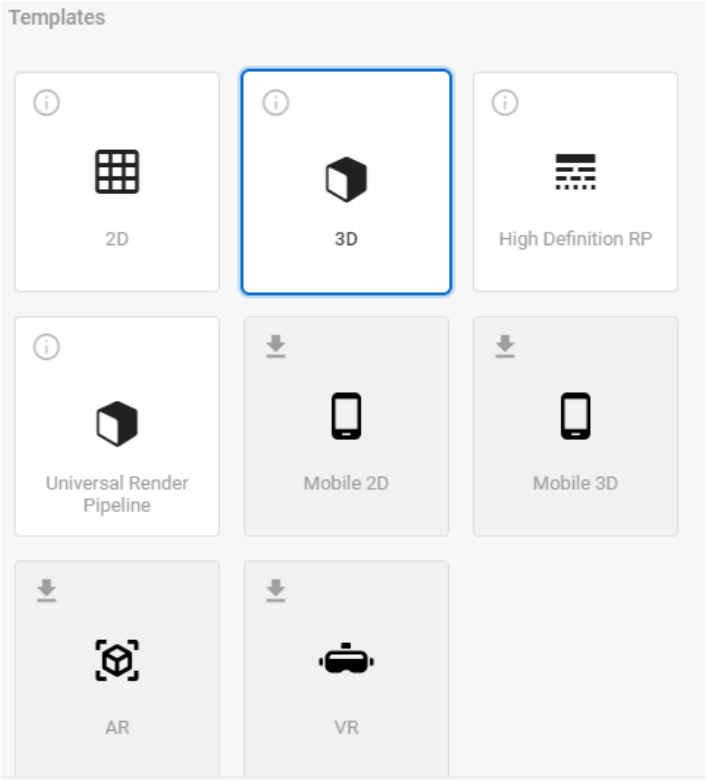

Figura 4.4: Tipos de proyecto disponibles en Unity

En este caso elijo crear un proyecto 3D, para ello solo es necesario introducir el nombre del proyecto y el directorio donde se creará. Una vez creado, se generan automáticamente una escena que contiene únicamente una cámara y una fuente de luz. Antes de colocar los modelos en la escena creo un plano que servirá como suelo en el que apoyarlos. Además, creo un material básico de color verde para asignárselo al suelo. Coloco ahora los modelos generados en el apartado anterior en la escena. Para poder colocar los modelos en la escena es necesario exportarlos junto con sus materiales y animaciones (en el caso del enemigo). Como el objetivo principal del proyecto es conseguir reducir la salud de la fortaleza lo máximo posible, coloco esta en el centro y, a izquierda y derecha de esta, coloco el almacén de oro y la torreta respectivamente. El enemigo lo coloco justo en frente de la fortaleza, de espaldas a la cámara (la cual elevo levemente para facilitar la visión de toda la escena), para poder observar con facilidad sus movimientos. Obtenemos así la siguiente escena, que a partir de ahora utilizaré para testear el algoritmo de Aprendizaje por Refuerzo.

Como se observa en la imagen, he incluido una barra que indica la salud de la fortaleza para visualizar más fácilmente el estado en el que se encuentra durante la ejecución del algoritmo. La implementación de esta se desarrollará en el siguiente apartado.

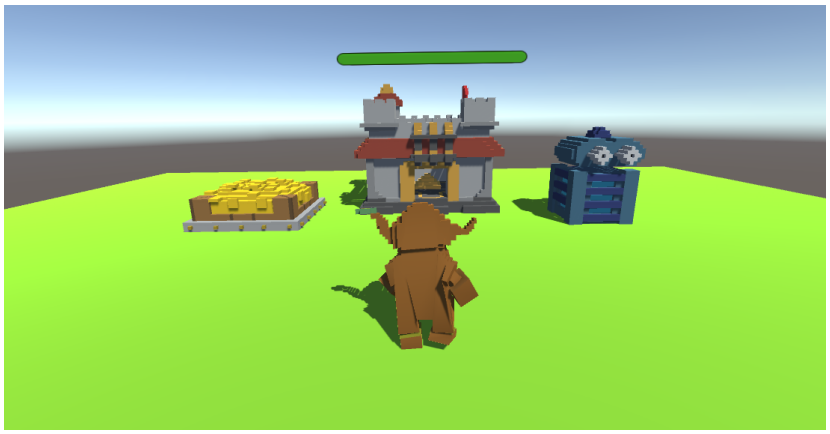

Figura 4.5: Escenario básico para el desarrollo del proyecto

4.4. Implementación del algoritmo

Antes de comenzar a implementar el algoritmo de Aprendizaje por Refuerzo, crearé un script para poder visualizar la salud de la fortaleza mediante una barra de vida. Para ello, añadimos un canvas a la fortaleza con un slider. El slider tendrá el fondo rojo y un relleno verde, de este modo, cuando la salud del enemigo se reduzca, la barra verde descenderá dejando visible el fondo rojo. A continuación muestro el código de las funciones perteneciente al script de la barra de salud:

```
void Start()
{
    health = maxHealth;
    slider.value = calculateHealth();
}

// Update is called once per frame
void Update()
{
    slider.value = calculateHealth();

    if (health <= 0) //Si la salud se reduce a cero
       destruimos el objeto asociado (Fortaleza)
    {
        Destroy(gameObject);
    }

}

//Funcion para calcular la salud actual
//Devuelve un valor entre 0 y 1 (que son los valores que
   toma el slider)
float calculateHealth()
{
```

```
        return health / maxHealth;
    }
```

Extracto de código 4.1: FortalezaHealth.cs

En la función *Start()* inicializo la salud de la fortaleza al máximo y el valor del slider. Para calcular el valor del slider en cada momento se emplea la función *calculateHealth()*, que calcula un valor entre 0 y 1 (necesario para establecer el valor del slider correctamente), dividiendo su salud actual entre la salud máxima de la fortaleza. En la función *Update()* se actualizará el valor del slider cuando la fortaleza reciba daño y, en el caso de que la salud de la fortaleza llegue a 0, se destruye el objeto.

A continuación, desarrollo el algoritmo de Aprendizaje por Refuerzo que utilizará la IA para el estudio de este trabajo. Para ello, utilizaré *ML-Agents*, un proyecto open-source que permite crear entornos para el entrenamiento de agentes inteligentes. Para poder utlizar correctamente *ML-Agents*, necesitaremos una versión de python igual o superior a la recomendada en la documentación. En este caso ya tengo instalada una versión de python superior a la recomendada (la versión 3.10.0). Ahora es posible crear un entorno virtual en el directorio donde se encuentra el proyecto de Unity mediante el comando:

```
python -m venv venv
```

Extracto de código 4.2: Crear un entorno virtual con python

El último parámetro que recibe el comando será el nombre del directorio que se cree, en este caso he decidido llamarlo venv para referirme al entorno virtual (Virtual Environment). Una vez creado, accedo al directorio *venv/Scripts* que acaba de crearse y ejecuto *activate* desde la consola de comandos. Para poder continuar, es necesario comprobar que está instalada la versión mas reciente de pip. Una vez instalado, ejecutamos el primer comando para instalar el paquete *pytorch*:

```
pip install torch=1.11.0 -f https://download.pytorch.org/
   whl/torch_stable.html
pip install mlagents
mlagents-learn --help
```

Extracto de código 4.3: Instalación del paquete pytorch y mlagents

Dependiendo de la versión de *ML-Agents* que se desee utilizar será necesario comprobar qué versión de pytorch se necesita en la documentación del github. Mediante el segundo comando del extracto de código superior instalo el paquete mlagents y compruebo que se ha instalado correctamente ejecutando el último comando de ese extracto. Para finalizar los preparativos será necesario instalar ML Agents desde el gestor de paquetes de Unity. Una vez comprobado, comienzo a desarrollar el algoritmo en el proyecto de Unity.

4.4.1. Prueba inicial

En primer lugar desarrollaré un algoritmo básico que permita a la IA avanzar hasta la fortaleza. Para ello, colocaré muros alrededor del escenario para que reciba

una penalización si colisiona con ellos y recibirá un refuerzo positivo si colisiona con la fortaleza. El escenario resultante es el siguiente:

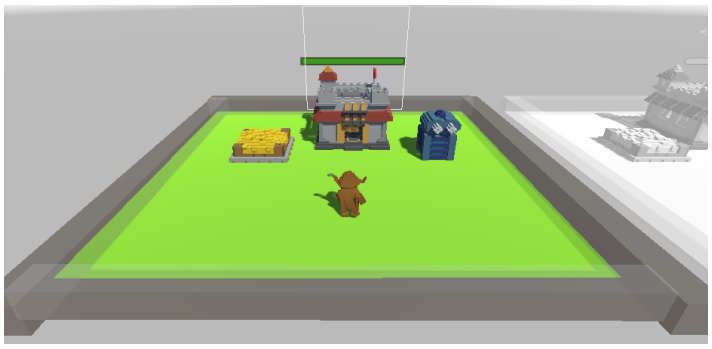

Figura 4.6: Escenario final para el desarrollo del proyecto

El primer paso será crear un script C# que llamaré *GoToGoal*, cuya clase heredará de la clase Agent. Para el correcto funcionamiento del algoritmo de aprendizaje por refuerzo será necesario que el agente realice una observación, tome una decisión, ejecute una acción y reciba un refuerzo (positivo o negativo) cíclicamente. De la observación se encargará la función *CollectObservations()*, que añade a las observaciones tanto la posición local del agente como la del objetivo (a continuación explicaré por qué trabajo con posiciones locales). La función *OnActionReceived()* se encargará de tomar una decisión y ejecutarla. Para ello, a partir de los valores continuos generados por el agente, la función se encargará de desplazarlo por los ejes X y Z a una velocidad determinada. Por último, la función *OnTriggerEnter()* se encargará de añadir una recompensa positiva (si el agente llega a la fortaleza) o negativa (si el agente colisiona con un muro) y termina el episodio. A continuación se muestra parte de este código inicial:

```
public override void CollectObservations(VectorSensor
   sensor)
{
    sensor.AddObservation(transform.localPosition);
    sensor.AddObservation(targetTransform.localPosition);
}


public override void OnActionReceived(ActionBuffers
   actions)
{
    float moveX = actions.ContinuousActions[0];
    float moveZ = actions.ContinuousActions[1];

    float moveSpeed = 10f;
    transform.localPosition += new Vector3(moveX, 0,
       moveZ) * Time.deltaTime * moveSpeed;
}
```

```
public override void Heuristic(in ActionBuffers
   actionsOut)
{
    ActionSegment<float> continuousActions = actionsOut.
       ContinuousActions;
    continuousActions[0] = Input.GetAxisRaw("Horizontal")
       ;
    continuousActions[1] = Input.GetAxisRaw("Vertical");
}

private void OnTriggerEnter(Collider other)
{
    if(other.TryGetComponent<Fortaleza>(out Fortaleza
       fortaleza))
    {
        Debug.Log(1);

        AddReward(+1f);
        EndEpisode();
    }
    if (other.TryGetComponent<Muro>(out Muro muro))
    {
        Debug.Log(-1);

        SetReward(-1f);
        EndEpisode();
    }
}
```

Extracto de código 4.4: Versión inicial del algoritmo

He empleado la función *Heuristic()* para poder comprobar funcionalidades manualmente sin necesidad de ejecutar el algoritmo de aprendizaje por refuezo. Para acelerar el proceso de aprendizaje de la IA, clonaré el escenario de modo que se ejecuten varios escenarios simultáneamente (por esto es necesario trabajar con coordenadas locales en lugar de globales, ya que con coordenadas globales todos los personajes aparecerían en el mismo punto). Además, establezco un número máximo de pasos para la ejecución de modo que el agente no aprenda únicamente a esquivar los muros manteniéndose en una posición constante.

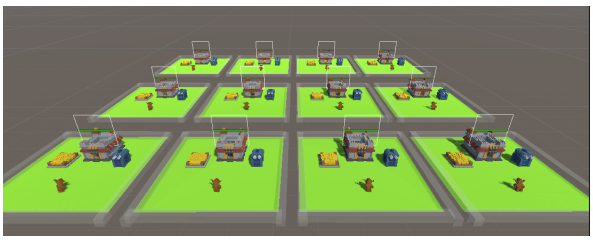

Figura 4.7: Entrenamiento de la IA en múltiples escenarios

Una vez que la IA consiga llegar hasta la fortaleza, el siguiente problema a resolver será que la IA haga daño a la fortaleza para intentar destruirla. Para ello será necesario añadir las siguientes variables y modificar algunas funciones del archivo *GoToGoal.cs*:

```
public FortalezaHealth saludFortaleza;
public float danoAFortaleza = 0f;

 public override void OnEpisodeBegin()
{
    transform.localPosition = new Vector3(-0.8f,5.9f,-15f
       );

    saludFortaleza.health = saludFortaleza.maxHealth;
}

public override void OnActionReceived(ActionBuffers
   actions)
{
    float moveX = actions.ContinuousActions[0];
    float moveZ = actions.ContinuousActions[1];
    danoAFortaleza = actions.ContinuousActions[2];

    float moveSpeed = 10f;

    transform.localPosition += new Vector3(moveX, 0,
       moveZ) * Time.deltaTime * moveSpeed;

}

private void OnTriggerEnter(Collider other)
{
    if(other.TryGetComponent<Fortaleza>(out Fortaleza
       fortaleza))
    {
        AddReward(+1f);

        //Si ha llegado hasta el objetivo ataca
        saludFortaleza.health -= Mathf.Abs(danoAFortaleza
           );
        AddReward(+Mathf.Abs(danoAFortaleza));

        if (saludFortaleza.health <= 0)
        {
            EndEpisode();
        }
    }
    if (other.TryGetComponent<Muro>(out Muro muro)) //Si
       se choca con un muro acaba el episodio
```

```
        {
            SetReward(-1f);
            EndEpisode();
        }
    }
```

Extracto de código 4.5: Segunda versión del algoritmo para atacar la fortaleza

En primer lugar añado un objeto de la clase *FortalezaHealth* para poder acceder desde el enemigo a la salud de la fortaleza y creo otra variable (*dañoAFortaleza*) con el daño que hará a la fortaleza en cada ataque, que será aleatorio al igual que el movimiento. Para asignar la fortaleza creada en el escenario a la variable FortalezaHealth creada simplemente basta con arrastrarla desde el editor de Unity al espacio generado para esta:

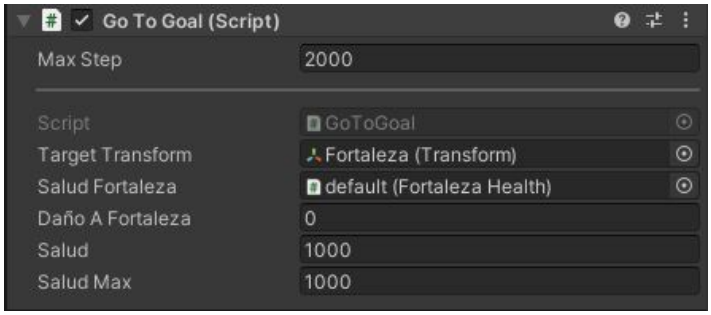

Figura 4.8: Variables en el editor de Unity

Para generar el daño que hace a la fortaleza en cada turno modifico el tamaño del vector ContinuousActions a 3 y añado en la función *OnActionReceived* una nueva línea para obtener el valor de ContiniuousActions. En la función *OnTriggerEnter* añado ahora una nueva recompensa que dependerá del daño que haga a la fortaleza y reduzco la salud de esta. Cuando la salud de la fortalezca se reduzca a 0 el episodio terminará. Al incio de cada episodio vuelvo a inicializar su salud en la función *OnEpisodeBegin*.

Una vez más compruebo su correcto funcionamiento mediante el empleo de la función *Heuristic* sin necesidad de ejecutar el entrenamiento. Para continuar con el desarrollo del algoritmo, haré que la torreta apunte hacia el enemigo constantemente y este reciba daño en función de la distancia a la que se encuentre de esta. Para ello, será necesario añadir una variable con la salud del enemigo y crear un script sencillo para la torreta. En *GoToGoal* será necesario añadir los siguientes fragmentos de código:

```
public float salud;
public float saludMax;

public override void OnEpisodeBegin()
{
    transform.localPosition = new Vector3(-0.8f,5.9f,-15f
       );

    //Volvemos a inicializar la salud del enemigo y la
       fortaleza
```

```
        salud = saludMax;
        saludFortaleza.health = saludFortaleza.maxHealth;
    }

    private void Update()
    {
        if(salud < saludMax){
            AddReward(salud - saludMax);
        }

        if (salud <= 0)
        {
            EndEpisode();
        }
    }
```

Extracto de código 4.6: Añadidos a GoToGoal para indicar la salud el enemigo

Al igual que en la fortaleza añado dos variables nuevas, una para la salud máxima del enemigo y otra para su salud actual. Del mismo modo, cada vez que comienza un nuevo episodio se reinicia la salud del enemigo al máximo en la función *OnEpisodeBegin*. Creo además la función *Update()*, que comprobará en cada llamada si la salud del enemigo se ha reducido, añadiendo en ese caso un refuerzo negativo que será la diferencia de su salud actual con el máximo. Además, si la salud de este se reduce a 0, el episodio termina.

A continuación muestro el script que creo para la torreta:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Torreta : MonoBehaviour
{
    public GoToGoal enemigo;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        //Se reduce la salud del enemigo segun la funcion
            gaussiana
        if (enemigo.salud > 0)
        {
```

```
                float x = Mathf.Pow((enemigo.transform.
                    localPosition.x - transform.localPosition.
                    x),2)/2;
                float y = Mathf.Pow((enemigo.transform.
                    localPosition.y - transform.localPosition.
                    y),2)/2;
                enemigo.salud-= Mathf.Exp(-(x+y));

                transform.LookAt(enemigo.transform);
            }

    }
}
```

Extracto de código 4.7: Algoritmo de la torreta

Como comenté al principio, este algoritmo se asocia a la parte superior de la torreta para que ésta pueda girar mientras la parte inferior se mantiene estática. Al igual que en el enemigo, creo una variable del tipo GoToGoal, que contiene la información sobre la salud de éste, y se la asocio desde el editor de Unity. La principal función es la función *Update()*, que en cada frame actualiza el ángulo al que mira la torreta mediante la función *LooAt(enemigo.transform)* que recibe como parámetro la localización del objetivo. Además, esta función comprueba si la salud del enemigo es positiva y, en ese caso, la reduce según una función gaussiana de dos dimensiones. De este modo el enemigo recibirá más daño cuanto más cerca se encuentre de la torreta y menos daño cuanto más se aleje.

$$f(x,y) = Ae^{-(\frac{(x-x_0)^2}{2}+\frac{(y-y_0)^2}{2})} \tag{4.1}$$

En este caso A vale 1, x_0 e y_0 representan las coordenadas de la torreta, y x e y son las coordenadas del enemigo en cada frame. Cuando el objetivo esté cerca de la torreta el daño que recibirá será cada vez más cercano a 1 y cuando se aleje este tenderá a 0.

4.5. Conclusiones

En este capítulo concluimos que...

5. Pruebas

5.1. Introducción

En este capítulo explicaremos...

5.2. Conclusiones

En este capítulo concluimos que...

6. Conclusiones y vías futuras

7. Bibliografía

[1] Adobe. Página principal de mixamo, 2022. URL https://www.mixamo.com/#/.

[2] MagicaVoxel. Página principal de magica voxel, 2021. URL https://ephtracy.github.io/.

[3] Unity Technologies. Página principal de unity, 2020. URL https://unity.com/es.